CS 429-01 Information Retrieval

Illinois Institute of Technology

Project- Information Retrieval System

GitHub Repository- https://github.com/ssrivastav01/Information-Retrieval-System.git

TABLE OF CONTENT

Professor: Jawahar Panchal

NAME: SEJAL SRIVASTAV
CWID: A20539218

1.ABSTRACT

1.1 DEVELOPMENT SUMMARY

The Information Retrieval System is designed as a robust tool for crawling, indexing, and processing web data, facilitating effective search capabilities over scraped web content. It incorporates a Scrapy-based crawler, a Scikit-Learn-based indexer, and a Flask-based processor, each tailored to manage specific aspects of data handling and retrieval:

1.1.1 Crawler (product.py)

The development of the crawler involved using Scrapy, a powerful and flexible framework for crawling websites, which is ideal for projects like this that require robust data extraction capabilities. The crawler is configured to handle seed URLs, maximum page and depth limitations, thereby efficiently managing the scope of the crawling operation. It saves the scraped content in HTML format and extracts specific metadata such as product names and links, which are then stored in JSON format for further processing by the indexer. The design also includes error handling and logging mechanisms to ensure stable operations under various web conditions. [1]

Framework Selection: The Scrapy framework was selected as the foundation for the web crawling component due to its inherent strengths in web data extraction. Scrapy provides a structured approach for crawling complex websites, offering asynchronous networking, built-in mechanisms for request throttling (respecting website rate limits), and seamless integration with data extraction pipelines.

Targeted Crawling Configuration: Central to the success of the crawling process is empowering users to precisely define the boundaries of the data collection exercise. This is accomplished through a thoughtful configuration interface exposed either through command-line parameters, settings files, or a dedicated web interface (depending on the desired level of user-friendliness). Key parameters include:

> **Seed URLs:** The initial starting points from which the crawler begins its exploration of a website or a domain.
>
> **Crawling Depth:** Controls how many "layers" of links the crawler will follow from the seed URLs. Setting a shallow depth limits the crawl to directly connected pages, while increasing the depth enables the discovery of content further within the site structure.
>
> **Page Limits:** A numerical constraint on the total number of web pages to retrieve and process, preventing the crawler from becoming overwhelmed by a large website.
>
> **Domain Restrictions:** Optionally define whether the crawler should confine itself to the specified domain(s) or be permitted to traverse links leading to external websites.

Robust and Ethical Web Navigation: Implementing a highly disciplined crawling strategy is vital both for maximizing the collection of relevant information and respecting the rules of web access. Several aspects contribute to this:

> **html Compliance:** The crawler meticulously parses the html file of each website it encounters. This file specifies directives from website owners indicating which sections of their site should remain off-limits to automated crawlers.

Focused Content Extraction: While the crawler can collect the entirety of a web page's content, it is purposefully tuned to prioritize specific product-related elements. This focus maximizes the relevance of the data for subsequent analysis and search while minimizing the storage and processing footprint. CSS selectors and/or regular expressions are strategically employed to target:

Product Name/Title: The primary identifier of a product.

Product Description: A textual description of the product's features, specifications, and benefits.

Images: Links to product images, which could later be downloaded and processed for visual search capabilities.

Adaptability and Error Handling: The inherent variability of real-world websites necessitates a degree of adaptability and resilience within the crawler. Robust error handling mechanisms are essential to prevent the entire crawling process from halting due to unexpected circumstances such as:

Website Changes: Websites, even well-established ones, occasionally undergo structural or layout modifications. Where possible, CSS selectors should be written defensively to mitigate breakage, and the crawler should log warnings when content cannot be located as expected. For significant disruptions, the ability to pause a crawl and notify the user may be beneficial.

Network Errors: Network connectivity issues are inevitable. The crawler employs request retries with exponential backoff (increasing wait time between attempts) to gracefully handle temporary network glitches.

HTTP Status Codes: Monitoring HTTP status codes in web responses allows the crawler to distinguish between successful page retrievals, redirects, "page not found" errors, and server-side issues. This information can inform when to abandon following certain links or flag potential problems for the user.

1.1.2 Indexer (inverted_index.py)

The indexer is crafted to transform raw data collected by the crawler into a structured format that is easy to query. Utilizing the Scikit-Learn library, the indexer applies the **TfidfVectorizer** to generate a matrix of TF-IDF scores, reflecting the importance of words within documents and across the corpus. This matrix is then used to construct an inverted index, mapping terms to their document locations and relevance scores, which is essential for efficient query processing. Serialization of the index using pickle allows for quick loading and updating processes, ensuring the system remains responsive as the corpus grows. [2]

Data Transformation and Cleaning: The transition from raw web content extracted by the crawler to a well-structured index necessitates careful data preprocessing. This stage entails:

Text Normalization: Converting text to lowercase, removing punctuation, and applying stemming or lemmatization (reducing words to their root form) can aid in matching semantically equivalent terms during the search process.

Stop Word Removal: Filtering out extremely common words (e.g., "the", "and," "of") that carry little informational value for search purposes can reduce index size and improve retrieval efficiency.

Data Structure Selection: Considering whether to represent the corpus as a simple list of documents, a bag-of-words model, or a more complex structure would depend on the indexer's requirements.

Inverted Index Creation: The core of the indexing process lies in the construction of an inverted index data structure. Conceptually, an inverted index inverts the traditional relationship between terms (words) and documents; instead of storing a list of terms within each document, an inverted index maintains a list of all the documents in which each unique term appears. This architecture is exceptionally well-suited for rapid search operations.

TF-IDF Weighting: To quantify the importance of terms within the context of search, the Term Frequency-Inverse Document Frequency (TF-IDF) metric is employed. It balances two factors:

Term Frequency (TF): How often a given term appears within a particular document. The assumption is that terms that occur more frequently are more descriptive of that document's content.

Inverse Document Frequency (IDF): Offsets the importance of terms that appear across a vast number of documents in the collection. Common terms that appear everywhere carry less discriminative power for distinguishing between specific documents.

Scikit-Learn Implementation: Scikit-Learn provides powerful classes such as TfidfVectorizer to streamline the process of text preprocessing, TF-IDF calculation, and building index representations suitable for querying. Careful consideration should be given to efficient sparse matrix representations if dealing with a large-scale dataset to optimize memory and computational performance.

Cosine Similarity for Ranking: When a search query is presented, it is first processed through the same text normalization and TF-IDF vectorization steps applied during index creation. The resulting query vector is then compared to each document vector within the index using the cosine similarity metric. Cosine similarity quantifies the "angular distance" between two vectors in high-dimensional space. A value of 1 indicates identical vectors, while a value of 0 indicates complete orthogonality (no similarity). In practice, most query-document pairs will fall somewhere in between.

1.1.3 Processor (main.py)

The development of the processor focused on providing a simple yet effective user interface for querying the indexed data. Built with Flask, a lightweight and powerful web framework, the processor supports both GET and POST requests, making it versatile for web applications. It includes a rudimentary natural language processing capability to modify queries for better search results, leveraging edit distance algorithms to suggest the most relevant terms from the indexed vocabulary. The processor evaluates these queries against the TF-IDF scores in the inverted index to retrieve the most relevant documents, showcasing the results in a user-friendly format. [3]

Query Processor (query_processor.py)

The Flask-based query processor handles free-text queries. It may need to incorporate several steps to ensure the input is optimally prepared for the search process:

Spell Checking: Integrating a basic spell-checking mechanism, perhaps leveraging an edit distance-based technique or a predefined dictionary, can alleviate the impact of common typing errors.

Query Expansion (Advanced): If desired, a query expansion module could be introduced. Approaches like synonym substitution (using a thesaurus) or word embedding techniques (Word2Vec, GloVe) can broaden the search to semantically related terms, even if the user's original query didn't contain those exact matches.

Validation and Error-Checking: Implementing robust input validation helps prevent malformed queries or attempts to exploit the system. Checks might include:

Length Limits: Prevent excessively long or overly short queries that might have unintended performance consequences.

Character Restrictions: Sanitize input to avoid special characters that could interfere with the index lookup or introduce security vulnerabilities.

Error Reporting: Gracefully handle unexpected query scenarios and provide informative feedback to the user.

Results Ranking and Presentation: After receiving a transformed and validated query from the user interface, the query processor orchestrates the following:

Index Retrieval: The query vector derived from the user's search terms is compared against the pre-calculated document vectors within the inverted index using cosine similarity. This operation efficiently retrieves a set of candidate documents most likely relevant to the search.

Ranking Refinement: While cosine similarity provides an initial ranking of results, additional factors might be incorporated to further refine the ordering of the results:

Recency: Optionally bias the ranking to favor products that have been added or updated more recently, especially in contexts like e-commerce where freshness matters.

Popularity Metrics (Advanced): If the system tracks usage data such as clicks or "add to carts," this information could be used to promote frequently viewed or purchased products higher within the results.

Customizable Weights: Depending on the user audience, it might be beneficial to expose settings that let users control whether emphasis is placed on product price, comprehensiveness of descriptions, or other aspects reflected within the index.

Top-K Result Selection: Since it's impractical to present an exhaustive list of all marginally relevant matches, the system focuses on returning the top-K most relevant results. The value of K is a design choice – a smaller K leads to a more focused result set, but risks missing slightly less relevant items, while a larger K increases the chances of including useful information at the cost of potentially overwhelming the user.

Result Snippets (Optional): To improve the user's ability to quickly assess the relevance of each result, short snippets of text directly surrounding the matched keywords within the product description can be displayed alongside product titles.

Search Feedback Loop: Monitor the success of your search system in various ways:

Query Logs: Analyze the types of queries users submit. This can identify gaps in the index (e.g., common searches that yield few results) or suggest potential areas for query expansion.

Clickthrough Data: If possible, record which search results are clicked. This reveals not just what users search for, but also which results they deem most relevant, implicitly informing future refinements to your ranking algorithm.

Explicit Ratings (if applicable): In certain settings, you might allow users to explicitly provide feedback on the quality of search results, offering valuable direct data to guide further improvements.

The development of the Query Processor was designed to enhance the user interaction with the indexed data through a Flask-based web application. This component is responsible for processing and responding to user queries, utilizing several advanced features:

Spelling Correction:

The processor includes a spelling correction mechanism to adjust user queries, ensuring that search terms are matched against recognized words in the indexed vocabulary. This feature uses the NLTK library, specifically leveraging a comprehensive list of correct words (**words.words()**) to find the closest corrections based on edit distance. This ensures that even if users make typos or errors in their search terms, the system can still return relevant results.

Query Modification:

To further enhance the relevance of search results, the query processor modifies user queries by comparing each term against the vocabulary extracted from the indexed data. Using edit distance algorithms, it identifies the most similar terms in the index and reconstructs the query with these terms. This step is crucial for accommodating synonyms or closely related terms that may not be explicitly present in the user's initial query but are relevant to their search intent.

Query to Vector Conversion:

Once the query is corrected and modified, it is converted into a vector representing its significance in the context of the indexed documents. This involves calculating the Inverse Document Frequency (IDF) for each term in the query, leveraging the structured data from the inverted index created by the indexer.

Cosine Similarity Calculation:

With the query vectorized, the processor computes the cosine similarity between the query and each document in the corpus. This metric helps in determining the closeness of the query to potential results based on the TF-IDF scores from the index. The processor then sorts these scores to present the top results, ensuring that the most relevant documents are easily accessible to the user.

Integration with Flask Web Application

The integration of **query_processor.py** within a Flask web application (**main.py**) allows for dynamic web-based interaction. Users can input their queries through a simple web form, and the system processes and displays the results in a clear and concise manner. The Flask application routes user requests and handles the display of results, making the search process intuitive and user-friendly. This setup not only facilitates an efficient query-response cycle but also provides a platform for further enhancements like adding user feedback mechanisms or more complex query handling features in the future.

System Robustness and User Experience

In developing the query processor, particular attention was given to robustness and user experience. Error handling was implemented to manage unexpected inputs or system failures gracefully, ensuring the system remains stable and responsive. The user interface is designed to be simple yet effective, with clear indications of how to perform searches and view results.

System Integration

Integration across these components was a critical aspect of the development process. The system was designed to ensure seamless data flow from the crawler to the indexer and finally to the processor, with each component loosely coupled yet efficiently connected through standardized data formats like JSON and text files. This design not only facilitates maintenance and upgrades but also enhances the system's robustness by isolating component failures.

Testing and Validation

Comprehensive testing strategies were implemented to validate each component and the integrated system. This included unit tests for individual functions, integration tests to ensure data integrity across components, and system tests to verify end-to-end functionality under simulated real-world conditions. Automated testing scripts were developed to continuously monitor system performance and stability.

Scalability and Performance Optimization

Throughout development, scalability was a priority. The system architecture was designed to handle increases in data volume and user load with minimal impact on performance. Performance optimizations were made in real-time data handling, query processing, and resource management to ensure the system could scale effectively without degradation in user experience.

1.2 OBJECTIVES

Enhanced Data Accessibility:

To create a robust mechanism that can autonomously navigate, scrape, and extract structured data from specified web domains. This objective aims to significantly reduce the effort required to gather and utilize web data, making information readily accessible and usable for various analytical purposes.

Accurate and Efficient Information Retrieval:

Develop a highly efficient indexing and retrieval system that utilizes TF-IDF scoring and cosine similarity measures to provide precise and relevant search results. The goal is to implement an indexing system that not only speeds up the retrieval process but also maintains high accuracy in matching user queries with the most relevant documents.

User-Centric Search Experience:

To offer an intuitive and responsive search interface that catifies users by delivering quick and relevant search results. This includes implementing front-end features that simplify the search process and enhance user interaction, such as real-time query suggestions, error correction, and user-friendly results presentation.

Scalability and Extensibility:

Build the system with scalability in mind, ensuring that it can handle an increasing volume of data and user requests without performance degradation. The architecture should also be extensible, allowing for easy incorporation of additional modules or updates, such as new data sources, advanced natural language processing capabilities, or machine learning algorithms.

Automation and Efficiency in Data Processing:

Automate the processes of data crawling, indexing, and querying to minimize manual intervention and maximize efficiency. This includes the development of automated workflows for updating the indexed data and handling changes in web page structures or content, ensuring the system remains current with minimal maintenance.

Robustness and Error Handling:

Ensure the system is robust against various operational challenges such as server failures, irregular data formats, and network issues. Implement comprehensive error handling and recovery protocols to maintain system integrity and continuity in the face of such challenges.

Compliance and Ethical Crawling:

Adhere to web crawling ethics and legal standards, including respecting robots.txt files and not overwhelming web servers with requests. This objective is crucial for maintaining good practices and avoiding legal repercussions while crawling web domains.

Research and Development Contributions:

Contribute to the field of information retrieval by exploring innovative approaches to data crawling, indexing, and retrieval. Document and share findings and methodologies that could benefit academic and industrial communities interested in web data management and search technologies. [4]

1.3 NEXT STEPS

To realize the full potential of this information retrieval system and address potential limitations, the following avenues for future development are envisioned:

1.3.1 Enhanced Crawling Capabilities

Scalability: Design and implement a distributed crawling architecture, likely leveraging a framework like Scrapy Cluster or Frontera. This prepares the system to handle a broader range of websites and significantly larger dataset ambitions.

Handling Dynamic Content: Many modern websites rely heavily on JavaScript to render content. Integrate a headless browser solution (e.g., Selenium or Playwright) alongside Scrapy to enable crawling and extraction from these more complex sites.

Change Detection: Develop mechanisms to periodically revisit previously crawled websites and identify new or updated product information. This ensures the index reflects the most up-to-date data on the web.

1.3.2. Advanced Search Techniques

NLP Integration: Incorporate Natural Language Processing techniques for richer semantic understanding:

> **Named Entity Recognition:** Identifying specific brands, product models, or technical features within product descriptions could power more fine-grained search and filtering.
>
> **Sentiment Analysis:** Extracting sentiment from product reviews (if collected) to allow search queries like "Find highly-rated headphones".

Query Expansion: Research and implement query expansion strategies to increase the likelihood of retrieving relevant results even when the user's search terms don't perfectly align with the indexed content. Consider the following:

> **Synonym Substitution:** Integrate a thesaurus or knowledge base to broaden searches.
>
> **Word Embeddings:** Explore techniques like Word2Vec or GloVe to identify semantically related terms, even in the absence of direct lexical matches.

Beyond TF-IDF: Investigate more sophisticated ranking functions such as BM25 or probabilistic retrieval models. These can offer refinements in result relevance, particularly for larger datasets.

1.3.3. Enhanced User Experience

Personalization: Implement basic personalization features to tailor search results based on a user's past search and interaction patterns. This could involve promoting items from frequently searched categories, brands, or price ranges.

Recommendations: Explore collaborative filtering or content-based recommendation techniques to suggest "similar products" alongside search results. This can boost product discovery and engagement within the system.

Voice Search Integration: Adapt the interface to accept voice-based search queries, aligning with the increasing popularity of voice assistants and smart speakers.

1.3.4. Data Analytics & Insights

Product Trend Analysis: Transform the collected product data into a valuable asset for market research. Enable the system to generate summaries of pricing trends, emerging brands, or shifts in product features over time.

Search Analytics Dashboard: Utilize the query logs and implicit usage data to visualize search trends, popular categories, and areas where results may be less satisfactory. This empowers data-driven decision-making for further system improvements.

Prioritization

The next steps outlined above offer a roadmap for the system's continued evolution. Determining the most impactful development priorities will depend on your specific use case and target audience. Factors to consider include:

Search Volume: If the number of user queries is expected to be very high, scalability and search algorithm optimizations should be prioritized.

Data Freshness: In domains where products or prices change rapidly, focus on efficient change detection within the crawler and consider real-time index updates.

User Sophistication: Advanced NLP features and personalization are most beneficial if your users expect a highly polished search experience akin to major commercial search engines. [5]

2. OVERVIEW

2.1 SOLUTION OUTLINE

At its core, the information retrieval system addresses the challenge of extracting and organizing product information from a specified website domain and providing an efficient means to search through that data. Here's a breakdown of how the system achieves this: [6]

1. **Web Crawling (product.py)**

   Seed URL/Domain: The crawler initiates from a provided starting point or within a defined boundary (in this case, web-scraping.dev/product).

   Traversal: The ProductSpider systematically navigates the website, following links within the allowed page and depth limits. It intelligently handles elements such as pagination to explore the target content comprehensively.

   Extraction: Specific data points (product titles and URLs) are parsed from the downloaded HTML.

   Persistence: Both the raw HTML content and the structured product data in JSON format are saved locally for further processing. [7]

2. **Indexing (inverted_index.py)**

   Corpus Construction: The indexer loads the extracted product information (titles) from the JSON file and builds a text corpus.

   TF-IDF Representation: The TfidfVectorizer analyzes the document corpus, calculating the TF-IDF score for each term. This score reflects how important a word is to a document within the overall collection.

   Inverted Index Structure:

   - An inverted index is optimized for rapid search. Rather than associating documents with terms, it associates *terms* with lists of documents and their corresponding TF-IDF scores.
   - Example:
   - term: "monitor" -> [(doc_id_1, 0.72), (doc_id_5, 0.54), ...]

   Outputs: The indexer produces three key outputs:

   - **product.pickle:** The serialized inverted index for efficient search.
   - **corpus.txt:** A plain text file containing the document corpus (product titles), potentially useful for analysis or debugging.

        - **urls.txt:** A list of the corresponding URLs for the documents in the corpus, essential for presenting search results to the user.

Why these additional outputs?

- **corpus.txt:** Provides a human-readable representation of the content your system has indexed. This can help verify the system's coverage or identify any text processing issues.
- **urls.txt:** Crucial for providing context to search results - the user needs to see the actual web page URLs alongside the product titles. [2]

Processor (query_processor.py and main.py)

The core responsibility of the processor component is to handle user queries, interact with the pre-built index, and deliver search results in a user-friendly format. Here's an outline of its operation:

1. **Query Interface:** The Flask application (main.py) provides a web-based interface where the user enters their search query.
2. **Query Preprocessing (query_processor.py):** The query is optionally passed through spelling correction and query modification functions using NLTK to improve the chance of finding relevant matches.
3. **Index Interaction (query_processor.py):** The preprocessed query is transformed into a TF-IDF vector compatible with the index representation. Cosine similarity calculations are performed between the query vector and the documents within the loaded index.
4. **Result Ranking (query_processor.py):** Documents are ranked according to their cosine similarity score, prioritizing higher scores.
5. **Output Generation (main.py):**
    - The top 'K' results (a configurable number) are retrieved from the ranking.
    - The corresponding URLs are extracted from the 'urls.txt' file.
    - The Flask application renders an HTML template that displays the list of result URLs to the user.

Primary Output

The primary output of the processor is not a standalone file, but rather a dynamically generated web page presented to the user. This results page contains:

- A list of ranked URLs (hyperlinked) that correspond to the top matches for the search query.

2.2 RELEVANT LITERATURE

Models and Implementation Early IR systems were boolean systems which allowed users to specify their information need using a complex combination of boolean ANDs, ORs and NOTs. Boolean systems have several shortcomings, e.g., there is no inherent notion of document ranking, and it is very hard for a user to form a good search request. Even though boolean systems usually return matching documents in some order, e.g., ordered by date, or some other document feature, relevance ranking is often not critical in a boolean system. Even though it has been shown by the research community that boolean systems are less effective than ranked retrieval systems, many power users still use boolean systems as they feel more in control of the retrieval process. However, most everyday users of IR systems expect IR systems to do ranked retrieval. IR systems rank documents by their estimation of the usefulness of a document for a user query. Most IR systems assign a numeric score to every document and rank documents by this score. Several models have been proposed for this process. The three most used models in IR research are the vector space model, the probabilistic models, and the inference network model. [8]

2.2.1 Vector Space Model

In the vector space model text is represented by a vector of terms. The definition of a term is not inherent in the model, but terms are typically words and phrases. If words are chosen as terms, then every word in the vocabulary becomes an independent dimension in a very high dimensional vector space. Any text can then be represented by a vector in this high dimensional space. If a term belongs to a text, it gets a non-zero value in the text-vector along the dimension corresponding to the term. Since any text contains a limited set of terms (the vocabulary can be millions of terms), most text vectors are very sparse. Most vector based systems operate in the positive quadrant of the vector space, i.e., no term is assigned a negative value. To assign a numeric score to a document for a query, the model measures the similarity between the query vector (since query is also just text and can be converted into a vector) and the document vector. The similarity between two vectors is once again not inherent in the model. Typically, the angle between two vectors is used as a measure of divergence between the vectors, and cosine of the angle is used as the numeric similarity (since cosine has the nice property that it is 1.0 for identical vectors and 0.0 for orthogonal vectors). As an alternative, the inner-product (or dot-product) between two vectors is often used as a similarity measure. If all the vectors are forced to be unit length, then the cosine of the angle between two vectors is same as their dot-product. If D~ is the document vector and Q~ is the query vector, then the dot-product similarity between document D and query Q (or score of D for Q) can be represented as:

$$Sim(\vec{D}, \vec{Q}) = \sum_{t_i \in Q, D} w_{t_{iQ}} \cdot w_{t_{iD}}$$

where w_{tiQ} is the value of the ith component in the query vector Q~ , and wt_{iD} is the ith component in the document vector D .

Inference Network Model

In this model, document retrieval is modeled as an inference process in an inference network. Most techniques used by IR systems can be implemented under this model. In the simplest implementation of this model, a document instantiates a term with a certain strength, and the credit from multiple terms is accumulated given a query to compute the equivalent of a numeric score for the document. From an operational perspective, the strength of instantiation of a term for a document can be considered as the weight of the term in the document, and document ranking in the simplest form of this model becomes like ranking in the vector space model and the probabilistic models described above. The strength of instantiation of a term for a document is not defined by the model, and any formulation can be used.

2.2.2 Evaluation

Objective evaluation of search effectiveness has been a cornerstone of IR. Progress in the field critically depends upon experimenting with new ideas and evaluating the effects of these ideas, especially given the experimental nature of the field. Since the early years, it was evident to researchers in the community that objective evaluation of search techniques would play a key role in the field. The Cranfield tests, conducted in 1960s, established the desired set of characteristics for a retrieval system. Even though there has been some debate over the years, the two desired properties that have been accepted by the research community for measurement of search effectiveness are recall: the proportion of relevant documents retrieved by the system; and precision: the proportion of retrieved documents that are relevant.

It is well accepted that a good IR system should retrieve as many relevant documents as possible (i.e., have a high recall), and it should retrieve very few non-relevant documents (i.e., have high precision). Unfortunately, these two goals have proven to be quite contradictory over the years. Techniques that tend to improve recall tend to hurt precision and vice-versa. Both recall and precision are set oriented measures and have no notion of ranked retrieval. Researchers have used several variants of recall and precision to evaluate ranked retrieval. For example, if system designers feel that precision is more important to their users, they can use precision in top ten or twenty documents as the evaluation metric. On the other hand if recall is more important to users, one could measure precision at (say) 50% recall, which would indicate how many non-relevant documents a user would have to read in order to find half the relevant ones. One measure that deserves special mention is average precision, a single valued measure most commonly used by the IR research community to evaluate ranked retrieval. Average precision is computed by measuring precision at different recall points (say 10%, 20%, and so on) and averaging.

2.2.3 Other Techniques and Applications

Many other techniques have been developed over the years and have met with varying success. Cluster hypothesis states that documents that cluster together (are very similar to each other) will have a similar relevance profile for a given query. Document clustering techniques were (and still are) an active area of research. Even though the usefulness of document clustering for improved search effectiveness (or efficiency) has been very limited, document clustering has allowed several developments in IR, e.g., for browsing and search interfaces. Natural Language Processing (NLP) has also been proposed as a tool to enhance retrieval effectiveness but has had very limited success. Even though document ranking is a critical application for IR, it is not the only one. The field has developed techniques to attack many different problems like information filtering, topic detection and tracking (or TDT) , speech retrieval, cross-language retrieval, question answering, and many more. [9]

2.3 PROPOSED SYSTEM

The proposed information retrieval system is designed to provide an efficient and user-friendly solution for extracting, organizing, and searching product information within a focused web domain. Here's a deeper look at its features and advantages:

Key Features

- **Targeted Crawling:** The Scrapy-based crawler intelligently navigates the specified website domain (e.g., web-scraping.dev/product), respecting depth and page limits. This focused approach ensures relevant content extraction and avoids the overhead crawling irrelevant sections of the web.
- **Robust Indexing:** The inverted index excels in search efficiency. It allows the system to rapidly match query terms against a large corpus of product descriptions, delivering results in near real-time.
- **TF-IDF Relevance Ranking:** Search results are not merely listed but ranked according to their relevance. TF-IDF weighting prioritizes products whose descriptions closely align with the search terms, helping users quickly find the most pertinent information.
- **Intuitive Interface:** The system accepts natural language queries through a web-based interface. Users don't need specialized knowledge to interact with the system.
- **Error Tolerance:** Optional query preprocessing features like spelling correction and query modification improve the system's flexibility in handling user input. This helps overcome common misspellings and guides the search towards more relevant results.

Advantages

- **Focus:** The system outperforms general-purpose search engines in the context of a specific domain. Results are highly relevant to the product space, as the index is tailored to that content.
- **Usability:** The combination of a simple query interface and ranked results creates a streamlined user experience.
- **Customizability:** System parameters (e.g., crawling boundaries and indexing options) can be adjusted to match different websites or domains, expanding its potential applications.
- **Deployment Ease:** The Flask-based web interface allows for easy deployment and access from standard web browsers.

Potential Use Cases

- **Competitive Analysis:** Quickly research products and their features within a competitor's website to gain market intelligence.

- **Price Comparison:** Build a focused product catalog to compare pricing across different retailers within the same domain.
- **Specific Product Research:** Enable buyers to easily find products matching their exact criteria by searching a targeted product database. [9]

3. DESIGN

3.1 SYSTEM CAPABILITIES

3.1.1 SEARCH CAPABILITIES

Search capabilities are designed to effectively match a user's input query with relevant data in an information database. This involves understanding and processing user queries that may be phrased in natural language or structured using Boolean logic. The system's ability to accurately translate these queries into actionable search commands is critical for retrieving relevant information. [9]

Term Weighting

An advanced feature in some search systems is the ability to apply weights to search terms, enhancing the retrieval process by prioritizing terms based on their assigned importance. For example, in a query like "Locate articles on renewable energy(0.8) versus non-renewable sources(0.2)," the system would prioritize renewable energy topics over non-renewable ones. This feature allows users to fine-tune their search results according to their specific interests or the relevance of certain terms.

Boolean Logic

Boolean logic in search systems enables users to combine keywords with operators such as AND, OR, and NOT to refine their search results. This logical structuring helps in filtering content that meets specific criteria, thereby improving search efficiency and relevancy. For instance, a query structured as "solar AND energy NOT oil" would return documents that discuss both 'solar' and 'energy' but exclude those that mention 'oil'.

Proximity Searches

Proximity searches allow users to specify the closeness of terms within documents. This method is useful for finding terms that appear near each other, suggesting a stronger relationship between the concepts they represent. A typical proximity query might look for 'climate' within 10 words of 'change', which helps in locating documents specifically discussing climate change.

Fuzzy Searches

Fuzzy searches are particularly useful in handling typos, spelling variations, and closely related words. This flexibility enhances the search system's ability to retrieve relevant documents even when query terms are not exactly matched. For example, a fuzzy search for "enviornment" could still pull up correct results for "environment", accommodating common spelling errors.

Contiguous Word Phrases

This feature treats phrases as single units of search to ensure precision in retrieving documents that contain specific sequences of words, such as "global warming". By searching for contiguous word phrases, the system can more accurately target documents discussing specific topics without retrieving irrelevant results where the words might appear separately.

Numeric and Date Range Searches

Search capabilities also extend to numeric and date ranges, allowing users to find documents that include specific numerical data or date information. This is particularly useful in research fields where data is time-sensitive or quantitatively driven. Queries like "population growth from 1990 to 2020" leverage this capability to focus on documents covering the specified period.

Concept and Thesaurus Expansion

To broaden the scope of search, systems may implement thesaurus or concept expansion techniques. These methods enhance searches by including synonyms or related terms, thus expanding the breadth of search results without losing relevance. This feature is beneficial when users are exploring topics without precise knowledge of the specific terminology used in existing literature.

These enhanced search capabilities are integral to modern information retrieval systems, providing users with powerful tools to locate the most relevant and accurate information efficiently. [9]

3.1.2 BROWSE CAPABILITIES

Overview

Browse capabilities are essential for users to effectively review and select from the results returned by search queries. These capabilities facilitate the evaluation of search results, allowing users to quickly discern which items are most relevant to their needs. Enhanced browsing features help users navigate through large volumes of data with ease and efficiency. [9]

Result Summarization

Once the search process concludes, the system provides summarized views of the results, typically in the form of line items or through more sophisticated data visualization methods. These summaries often include key information such as titles, abstracts, and relevance scores, enabling users to quickly assess the potential value of each item without needing to read the full content.

Ranking and Relevance Scoring

Search results are often ranked according to their relevance to the search query, with relevance scores assigned to each item. These scores, usually normalized between 0.0 and 1.0, indicate the system's confidence in the item's relevance to the user's query. This ranking helps users prioritize their review process, focusing first on items most likely to be of interest.

Item Selection and Display

Browse capabilities also include mechanisms for selecting and displaying items in detail. Users can choose which items to explore based on the summary information and then view these items in full. The system may provide options for viewing different sections or aspects of the items, such as text, images, or data charts, depending on the content type.

Highlighting and Navigation Aids

To assist users in understanding why certain items were retrieved, systems often highlight search terms within the documents. This feature allows users to quickly locate and evaluate the relevance of the search terms within the context of each document. Additionally, navigation aids like "jump to next highlight" can help users move efficiently through the items.

Zoning

Zoning involves segmenting the display of search results or documents into manageable sections, such as titles, abstracts, or relevant passages. This feature enables users to assess the relevance of items at a glance without needing to engage with the entire content, which is especially useful in large datasets.

Visualization Techniques

Advanced systems may employ various visualization techniques to display the relationships among multiple search results or to illustrate the distribution of topics across the retrieved documents. These techniques can include clustering information into groups that share similar themes or displaying interactive maps that allow users to explore the connections between different data points visually.

Contextual Tools

Additional tools, such as filters and sort options, allow users to refine their browse experience based on various criteria like date, author, or publication type. These tools help users customize their review process to match specific interests or requirements, thereby enhancing the efficiency of the information retrieval process.

These browse capabilities play a crucial role in helping users efficiently sift through large amounts of data to find information that is most relevant and useful to their specific inquiries. They bridge the gap between broad search results and the user's need for specific information, improving the overall effectiveness of the search and retrieval process.

3.1.3 MISCELLANEOUS CAPABILITIES

Overview

Miscellaneous capabilities in information retrieval systems encompass a variety of functions that enhance user interaction and efficiency when managing search queries and results. These features streamline the search process, assist in refining queries, and provide additional navigational aids to maximize the utility of the system.

Vocabulary Browse

Vocabulary browse allows users to explore the complete lexicon of the database. This feature displays a list of all unique words and their occurrences across the database's documents, sorted in alphabetical order. Users can input a word or fragment to see related terms and their frequencies, aiding in the identification of common or rare terms and assessing their potential usefulness in refining search queries.

Iterative Search and Search History Log

Iterative search refers to the process of refining search results by applying additional search criteria to an already existing set of results. This method helps users narrow down a large number of results to a more manageable and relevant subset by further specifying search parameters. A search history log complements this capability by recording all queries made during a session, allowing users to revisit or analyze their search strategies and outcomes.

Canned Queries

Canned queries are predefined search templates that users can save and reuse. These queries are tailored for recurring search needs, saving time and effort by eliminating the need to repeatedly enter complex search parameters. Users can create these queries to cover broad topics of interest and then refine them for more specific searches as needed.

Multimedia Management

Handling multimedia content poses unique challenges in information retrieval. Systems equipped with multimedia management capabilities can process and retrieve not only text but also images, audio, and video files. For example, thumbnail previews for images or transcribed snippets for audio files enhance the searchability and browsability of multimedia content, enabling users to quickly identify relevant media items.

Audio Transcription and Indexing

For audio and video content, transcription services convert spoken words into searchable text. This capability is crucial for making multimedia content accessible and searchable. Indexing these transcriptions alongside traditional text documents allows users to perform comprehensive searches across multiple media types.

Custom Annotations

Custom annotations let users add personal notes or tags to documents within the database, facilitating easier retrieval and organization of information. These annotations can be particularly helpful in collaborative environments, where users need to share insights or highlight information for colleagues.

Real-Time Updates

In dynamic databases, real-time update capabilities ensure that the information remains current by continuously integrating new data into the system. This feature is vital for applications where timely information is critical, such as news aggregation or financial monitoring services.

Advanced Filtering Options

Advanced filtering options provide users with tools to fine-tune search results based on specific criteria such as publication date, document type, or relevance score. These filters help users streamline the review process and focus on the most pertinent information.

These miscellaneous capabilities enhance the overall functionality and user experience of information retrieval systems, making them more adaptable and efficient in meeting a wide range of information needs. [9]

3.2 INTERACTIONS

3.2.1 CRAWLER TO INDEXER INTERACTION

Before delving into the interaction itself, it's essential to outline the roles played by the crawler and indexer components within this project. The crawler, built using the Scrapy framework, is responsible for navigating the web, downloading HTML content, and extracting relevant metadata like titles and URLs. It serves as the project's data acquisition engine. The indexer, powered by Scikit-learn for its text processing capabilities, consumes the crawler's output to construct a searchable index, representing the heart of the search engine's information retrieval mechanism.

The interaction begins with a user providing a seed URL, as well as constraints on the maximum number of pages to visit and the depth of crawling. This information triggers the Scrapy crawler's start_requests method, which systematically generates a series of requests for pages to be fetched and processed. During the crawling process, the parse method plays a crucial role. It meticulously extracts product titles and constructs well-formed URLs (yield {'title': product_name, 'link': link}) while simultaneously ensuring the HTML content is saved for later analysis. The "product.json" file, generated during this process, becomes a pivotal communication channel between the crawler and the indexer.

The indexer initiates its task by loading this crucial "product.json" file (the load_corpus function). Leveraging the contained URL information, a dedicated Scrapy spider (HTMLSpider) iterates over the saved HTML files, extracting the all-important raw textual content. The core of the indexing logic resides within the tf_idf_index function. Here, Scikit-learn's TfidfVectorizer transforms the text into a TF-IDF matrix, where each term's weight reflects its frequency within a document and its rarity across the entire collection. The project opts for an inverted index structure—a dictionary where keys represent unique terms, and associated values hold lists of document IDs and TF-IDF scores. This structure is exceptionally well-suited for rapid search queries. To ensure persistence and reduce future processing overhead, the indexer serializes its painstakingly constructed data structure into a "product.pickle" file (the to_pickle function).

This interaction showcases a clear data flow pattern. The crawler acts as a meticulous data provider, supplying raw HTML and a structured JSON summary. The indexer depends heavily on this output, methodically building the inverted index that empowers the search engine. The project's design reveals a structured approach, employing JSON and serialized output to facilitate seamless communication between the crawler and the indexer. [5]

Initiation: The User and the Crawler

The process begins when a user provides the seed URL, maximum pages to crawl, and depth restrictions. Let's look at a relevant code snippet from your ProductSpider class:

```
def start_requests(self):
    url = getattr(self, 'url', None)
    if url:
        for page_number in range(1, self.max_pages + 1):
            yield scrapy.Request(f'{url}{page_number}', self.parse, meta={'depth': 1})
```

This code does the following:

Takes User Input: The start_requests method is triggered with the user-provided seed URL.
Generates Requests: It iterates based on the user's maximum page limit, constructing URLs for the crawler to visit.
Controls Depth: The meta={'depth':1} part ensures the crawler respects the maximum depth setting, preventing it from going too deep into irrelevant links.
Crawling and Data Gathering

The crawler fulfills the generated requests. The parse method within your ProductSpider class is key here:

```
def parse(self, response):
    if response.meta.get('depth', 0) > self.max_depth:
        return

    product_name = response.css('title::text').get()
    prod = response.url.split('/')[-1].split('.')[0].split('-')[-1]
    link = f'https://web-scraping.dev/product/{prod}'

    yield {
        'title': product_name,
        'link': link
    }

    # ... code for saving the HTML ...
```

This method performs several critical actions:

Depth Check: Prevents crawling beyond the allowed depth.
Metadata Extraction: Obtains the product title and constructs a well-formed URL, which will be essential for the indexer.
HTML Saving: Saves the downloaded HTML file for further processing.
JSON: The Bridge Between Crawler and Indexer

The "product.json" file is the essential communication mechanism. During crawling, each data point like the title and URL is recorded into JSON:

```
yield {
   'title': product_name,
   'link': link
 }
```

Think of JSON as a structured notebook the crawler uses to neatly pass its findings to the indexer.

Indexer: Consuming the Crawler's Output

The indexer begins by loading the "product.json" file:

```
def load_corpus(json_file):
  try:
     with open('../crawler/' + json_file) as f:
        data = json.load(f)

     corpus = [obj['title'] for obj in data]
     urls = [obj['link'] for obj in data]

     return corpus, urls
  except Exception as e:
      print(f"Error loading corpus: {e}")
      return [], []
```

This highlights a few things:

Path Awareness: The code looks in the "../crawler/" directory, indicating that your project is designed with the expectation that the crawler's output will be present in a specific location.
Data Extraction: It extracts the 'title' and 'link' fields from the JSON entries, which will be used to build the document corpus and for referencing results later.
HTML Parsing with the HTMLSpider

A dedicated Scrapy spider (HTMLSpider) designed to process the saved HTML files. This spider contains logic like the following:

```
# ... (Inside the HTMLSpider class) ...
 def parse(self, response):
     # ... logic to extract textual content from the HTML ...
```

This snippet illustrates the focused nature of the HTMLSpider. Its job is to systematically open those saved HTML files and extract just the textual content, which is the raw material for indexing.

The Essence of Indexing: Building the Inverted Index

tf_idf_index function is the core of the indexer's logic:

```
def tf_idf_index(corpus):
  vectorizer = TfidfVectorizer()
  X = vectorizer.fit_transform(corpus).toarray()
  feature_names = vectorizer.get_feature_names_out()

  tfidf_index = {}
  for i in range(len(feature_names)):
    tfidf_index[feature_names[i]] = []
    for doc in range(len(corpus)):
      if X[doc][i] > 0:
        tfidf_index[feature_names[i]].append((doc, X[doc][i]))

  return tfidf_index
```

<u>3.2.2 INDEXER TO PROCESSOR INTERACTION</u>

Indexer: We've already discussed how it processes crawled data, meticulously constructing an inverted index optimized for search. It serves as the knowledge base for the search system. Think of it as the librarian who carefully organizes the library's collection.

Processor (Flask): The processor provides the user-facing interface. It handles search queries, interacts with the index to retrieve results, and presents them back to the user. It's analogous to a library patron who poses questions to the librarian.

The Interaction: A Search Query's Journey

User Input & Query Preprocessing

The interaction begins when a user submits a search query using the Flask web interface. The processor might first employ techniques within the 'query_processor.py' module to refine the user's input. Functions like spelling_correction and modify_query aim to improve the likelihood of matching the user's intent to the content within the index, even if the original query is slightly misspelled or uses uncommon vocabulary.

Loading the Index

The core of the processor's power lies in its ability to load the pre-built index created by the indexer. The load_index function likely reads the "product.pickle" file, deserializing it to restore the full index structure into memory. This step is like the library patron obtaining a map (the index) to the library's vast collection.

Query Transformation

The processor must transform the user's query (which is just text) into a representation compatible with the index. The query_to_vector function plays this critical role. Similar to the indexing process, it likely applies TF-IDF principles, turning the query into a vector where terms are weighted according to their relative importance. This is analogous to how the patron might transform their question into keywords that match the library's organization system.

The Core of Search: Calculating Similarity

With the query transformed, the processor uses the cos_similarity function to compare the query vector against the terms in the inverted index. Remember that the inverted index stores TF-IDF scores for each term in every document. Cosine similarity offers a way to measure how closely the query aligns with each document based on the terms they have in common. The beauty of the inverted index is that looking up relevant documents for a specific query is incredibly fast!

Ranking and Retrieval

The cos_similarity function likely returns a list of documents and their similarity scores. The processor sorts this list in descending order of similarity, placing the most relevant documents at the top. It then needs to use the additional files created by the indexer—likely "urls.txt" and possibly "corpus.txt"—to map the document IDs back to their original URLs. This is crucial so the processor knows which webpage links to present to the user.

Code Snippets: Illuminating the Process

Let's look at some relevant code snippets to clarify this interaction:

Loading the Index

```
def load_index(pickle_file):
  try:
    with open(pickle_file, 'rb') as f:
      index = pickle.load(f)
    return index
  except Exception as e:
    print(f"Error loading index: {e}")
    return {}
```

Query Transformation

```
def query_to_vector(query, inv_index, N):
    terms = query.split(" ")
    vector = {}
    for term in terms:
        if term not in inv_index:
            vector[term] = 0
        else:
            df = len(inv_index[term])
            idf = math.log(N / df)
            vector[term] = idf
    return vector
```

Similarity Calculation

```
def cos_similarity(query_vector, tfidf_index, corpus):
    scores = {}
    for i in range(len(corpus)):
        scores[i] = 0
    for term in query_vector:
        for (doc, score) in tfidf_index[term]:
            scores[doc] += score * query_vector[term]
    for doc in scores.keys():
        scores[doc] = scores[doc] / len(corpus[doc])

    return sorted(scores.items(), key=lambda x: x[1], reverse=True)
```

Presentation (Flask)

Finally, the processor takes the top K results (where K is a configurable parameter) and likely renders them within an HTML template within the Flask application. This involves displaying the relevant URLs, potentially along with snippets from "corpus.txt," to give the user context for why each result was returned.

Key Design Insights

The code and this interaction illustrate a few key aspects of the project's design:

Focus on Retrieval Speed: The index structure and the use of cosine similarity are designed for rapid lookup and comparison during the search process.

Preprocessing: Query refinement functions aim to bridge the gap between informal user queries and the structured terminology within the index.

Separation of Concerns: The indexer and processor components have well-defined tasks, improving code maintainability.

User Experience: The focus shifts from data acquisition (the crawler) and organization (the indexer) to presenting results in a user-friendly manner (the processor's role).

3.2.2 PROCESSOR TO USER INTERACTION

Understanding the Interface

The processor, specifically its Flask-based web component, is where the search engine directly interacts with the user. It must perform two key functions: 1) provide a way for the user to input a query in a format the search engine can process, and 2) take the search results and display them in a user-friendly, informative manner. Think of this processor component as the reference desk in a library, where the user conveys their information need, and the librarian presents resources to address that need.

The Search Initiation

The interaction likely starts with a simple HTML form on a webpage rendered by the Flask application. Within this form is a text input field where the user can type in their search query. The form might also have a "submit" button or trigger the search process automatically as the user types. In all likelihood, the Flask application will send this query to the processor as JSON-formatted data – a structured format suitable for programmatic handling.

Preprocessing (Optional)

Before diving into the search logic itself, the project's query_processor.py module might play a role in refining the user's input. The spelling_correction function could attempt to correct minor typos, ensuring that misspellings don't prevent the search engine from identifying relevant matches. Similarly, the modify_query function might leverage the index's vocabulary (get_vocab) to rephrase the user's query with terms more likely to align precisely with how content is indexed, improving the chances of highly relevant results.

Behind the Scenes: Search Logic

After optional preprocessing, the core search logic kicks in. The processor would have already loaded the necessary resources when it started up: the deserialized index (using a function like load_index), the URL mapping (load_urls), and potentially the document corpus (load_corpus_file) for generating result snippets. Now, it transforms the query into a TF-IDF vector (query_to_vector), calculates cosine similarities (cos_similarity) to find matching documents, and ranks those documents based on their similarity scores.

Result Ranking and Selection

The processor likely retrieves not just the single best match but the top K results, where K is a configurable number. This provides the user with options, recognizing that relevance can be somewhat subjective. It's during this phase that the load_urls function becomes essential to translate the document identifiers from the index back into human-readable URLs that the user can click on.

Crafting the Results Page

The processor is tasked with taking the raw search results and transforming them into a digestible format for the user. Employing Flask's templating capabilities, the processor likely injects the following elements into an HTML results page:

Search Query: Displays the user's original query (or the preprocessed version), to give them context as to why these results were retrieved.

Result Links: A list of the top K URLs, presented as clickable hyperlinks, allowing the user to navigate directly to the potentially relevant web pages.

Result Snippets (Optional): The processor might leverage the "corpus.txt" file to present short excerpts of text surrounding the matching keywords within each document. Snippets provide additional context and help the user decide which links are most worth pursuing.

Code Snippets: From Processor Logic to User Interface

Let's look at snippets to illustrate how this interaction translates into code:

Flask Routing and Form Handling

```
from flask import Flask, render_template, redirect, url_for, request

# ...other imports...

app = Flask(__name__)

@app.route("/", methods=["GET"])
def main():
  return render_template('index.html') # index.html contains the search form

@app.route("/search",methods =['POST','GET'])
def search():
  if request.method == 'POST':
     query = request.form['search']
     return redirect(url_for('result', search=query))
  else:
     query = request.args.get('search')
     return redirect(url_for('result', search=query))

@app.route("/result/<search>")
def result(search):
  # ... Perform search using 'search' query and render results.html ...
```

Result Rendering (results.html Template Snippet)

```
<h1>Search Results for: {{ query }} </h1>

{% for url in urls[:K] %}
  <a href="{{ url }}">{{ url }}</a>
  {% if corpus %}
     <p> ... Display snippet from corpus based on matching keywords ... </p>
  {% endif %}
{% endfor %}
```

The Importance of User Experience

The design of the search interface and how results are presented has a profound impact on the overall user experience. Consider these aspects:

Clarity: The search form should be simple and intuitive. Results should be displayed without clutter or confusing technical details.

Speed: The processor should strive to retrieve and render results as quickly as possible. Every second of delay risks user frustration.

Visual Hierarchy: Use font size, spacing, and potentially color highlights to guide the user's eye toward the most relevant results first.

3.3 INTEGRATION

3.3.1 Crawler to Indexer Integration

The crawler and indexer exhibit a clear producer-consumer relationship fundamental to the project's design. Let's examine this integration in detail: [8]

Data Handoff: The "product.json" File The "product.json" file serves as the critical communication channel between the crawler and indexer. During crawling, this structured JSON file records metadata for every downloaded web page, primarily titles and URLs. The indexer is explicitly designed to consume this file, establishing a clear dependency.

Coupling and Contract The project demonstrates a degree of coupling between these components. The load_corpus function within the indexer presumes the existence of "product.json" in a specific relative file path (../crawler/). While this ensures predictable operation, it also means that structural changes to the crawler's output directory or the JSON format would necessitate updates in the indexer. This highlights the importance of a well-defined data contract between components.

Error Handling and Robustness The load_corpus function includes error handling (try...except) to gracefully handle scenarios where the "product.json" file might be missing or corrupt. However, the project could benefit from more sophisticated error reporting. For example, logging the specific exception encountered or potentially providing feedback to the user via the web interface could improve debugging and user experience during troubleshooting.

Optimization for Large-Scale Crawls While the file-based communication mechanism is suitable for moderately sized projects, consider how it might handle extremely large crawls. If the "product.json" file becomes unmanageably large, the project might require a refactoring either to:

Incremental Processing: Have the indexer process smaller batches of JSON data instead of loading the entire file at once.

Streaming: Transition to a streaming model, where the crawler sends individual data objects to the indexer rather than writing a single monolithic file.

3.3.2 Indexer to Processor Integration

The efficiency of the indexer-to-processor integration is paramount for a responsive search experience. Here's a breakdown of key aspects:

The Inverted Index as the Core Knowledge Base The inverted index generated by the indexer serves as the search engine's heart. By employing a dictionary-based data structure where terms map to documents and relevance scores, the indexer enables rapid lookup essential for real-time search scenarios. The processor's search functionality depends entirely on this pre-built index.

Efficient Loading and Deserialization The load_index function likely uses the pickle module to deserialize the "product.pickle" file. Python's pickle is relatively efficient for moderately sized indexes. In extremely large-scale deployments, the project could explore alternative serialization formats optimized for even faster deserialization or investigate memory-mapped index structures for potential performance gains.

Auxiliary Files and Result Formatting The indexer constructs not just the core index but also "urls.txt" and potentially "corpus.txt." The processor relies on these supplementary files to provide a user-friendly search experience. The load_urls function is crucial for mapping document IDs back to URLs for display, while "corpus.txt" is likely used for generating relevant snippets that give users context for each result.

Conceptual Alignment: Search Logic The indexer and processor share a conceptual understanding of search principles. Both components likely utilize TF-IDF principles when building the index and transforming a user query into vector representations. The query_to_vector and cos_similarity functions demonstrate the project's consistent approach to calculating search relevance.

3.3.3 Processor to User (UI) Integration

The processor-to-user integration determines the usability of the search engine. Let's delve into the key design considerations:

The Web Interface: Flask Framework The project leverages the Flask web framework to create a user-facing search interface and results display. Flask provides a flexible environment for defining routes, handling HTTP requests and responses, and integrating with HTML templates.

Data Exchange: HTTP and JSON The web interface likely transmits the user's search query to the processor as a JSON-formatted object over HTTP. JSON is a well-structured, lightweight data format that is easily parsable by the processor and is a standard for web-based interactions..

User-Centric Results Rendering The processor is tasked with transforming raw search results into a format readily digestible by the user. Flask's templating engine plays a key role here. The processor likely injects the following into a results page template:

Original (or preprocessed) query, to provide context.

A list of top-ranked result URLs, formatted as clickable links.

Result snippets (if enabled), offering quick insights into result content.

Focus on User Experience (UX) The design of the results page is a crucial aspect of integration. Consider these UX elements that the project might already address or have room for improvement:

Visual Hierarchy: Utilizing headings, spacing, and font sizes to guide the user towards the most relevant results.

Result Snippets: Employing keyword highlighting within snippets to visually emphasize why the search engine matched the result.

Speed: Ensuring rapid response times through index optimizations and potentially by minimizing the amount of data transferred to the user's browser.

3.3.4 Cross-Cutting Considerations

Let's delve into a few additional design integration aspects that impact the overall project:

Modularity and Testability The separation of concerns among the crawler, indexer, and processor promotes modularity. This has maintainability benefits, as changes to one component are less likely to have cascading effects on others. Furthermore, this modularity likely makes it easier to write unit tests to verify individual component behavior, improving code reliability.

Error Handling and User Feedback
While the indexer has basic error handling, the project would benefit from a more comprehensive error-handling strategy spanning all components. Gracefully handling situations such as network failures during crawling, index corruption, or unexpected search query formats directly impacts user experience. The project should consider:

Logging: Implementing a logging system to record errors and warnings for debugging.

Informative Error Messages: Translating technical errors into user-friendly messages displayed in the web interface.

Scalability and Bottlenecks Understanding potential bottlenecks is crucial for design integration, especially as the search engine handles larger datasets. Here's a possible analysis:

Crawler: The crawler might become a bottleneck if the website being crawled has rate-limiting mechanisms or if network bandwidth is constrained.

Indexer: For extremely large document collections, index construction time and the memory footprint of the index could become limiting factors.

Processor: In high-traffic scenarios, the processor could become a bottleneck, particularly during computationally intensive similarity calculations.

3.3.5 Potential Optimizations and Enhancements

The project, as presented, provides a solid foundation for a search engine. Here are potential optimizations and feature additions to enhance its design integration:

Distributed Architecture: For massive datasets, consider moving towards a distributed architecture where crawling, indexing, and search handling can be spread across multiple machines. This would involve rethinking data flow and potentially using message queues or dedicated search databases.

Alternative Indexing Structures: Explore advanced indexing techniques like tries or suffix trees, which might offer performance or storage optimizations under certain conditions.

Data Pipelines: Formalize the interactions between components using a data pipeline framework like Apache Airflow or Luigi. This would help with orchestration, error handling, and monitoring as the project's complexity grows.

Advanced Search Features: Enhance user experience by incorporating:

Autocomplete: Suggest partial queries as the user types.

Faceted Search: Allow users to filter by extracted metadata.

4. ARCHITECTURE

4.1 SOFTWARE COMPONENTS

Architecture Overview

The crawler component within your search engine project forms the foundation for data acquisition. It employs the Scrapy framework to systematically navigate websites (ProductSpider) and extract structured data from downloaded HTML (HTMLSpider). Let's examine its architecture, key software components, and the interactions that drive the crawling process.

4.1.1 Scrapy: The Backbone

The Scrapy framework provides a powerful and structured environment for defining and executing web crawlers. Key benefits Scrapy offers in this project include:

Request Management: Scrapy handles asynchronous requests, allowing the crawler to efficiently fetch multiple pages concurrently, optimizing network utilization.

HTML Parsing: Scrapy's built-in HTML and XML parsing capabilities simplify the extraction of targeted data elements from complex web pages.

Extensibility: Scrapy's modular design encourages the creation of reusable crawling components, supporting future expansion or modifications to your crawler.

The ProductSpider

The ProductSpider is a workhorse designed for web-based product information extraction. Let's look at its core responsibilities:

Request Generation (start_requests**):** This method is triggered to initiate the crawling process. It generates a series of scrapy.Request objects, each containing a target URL to be fetched. Consider these aspects:

Seed URLs: The start_requests method likely takes an initial URL (or a list of URLs) as input, forming the starting points for the crawl.

Pagination: If the website spans multiple pages, the start_requests might implement logic to generate requests for subsequent pages, ensuring comprehensive product coverage.

Response Parsing (parse**):** The parse method is invoked for every HTML response received by the spider. Key tasks include:

Data Extraction: Employs CSS selectors or XPath expressions to pinpoint the product name, price, and potentially other relevant details within the HTML structure.

HTML Saving: Saves the raw HTML response to a local file. This preserves the original data, which is necessary for later analysis by the HTMLSpider and might help during debugging.

Code Snippet: ProductSpider

```
class ProductSpider(scrapy.Spider):
  name = 'product'
  allowed_domains = ['web-scraping.dev']
  max_pages = 99
  max_depth = 9

  def start_requests(self):
    url = getattr(self, 'url', None)
    if url:
      for page_number in range(1, self.max_pages + 1):
        yield scrapy.Request(f'{url}{page_number}', self.parse, meta={'depth': 1})

  def parse(self, response):
    if response.meta.get('depth', 0) > self.max_depth:
      return

    product_name = response.css('title::text').get()
    prod = response.url.split('/')[-1].split('.')[0].split('-')[-1]
    link = f'https://web-scraping.dev/product/{prod}'

    yield {
      'title': product_name,
      'link': link
    }

    filename = 'crawler/product/product-' + response.url.split('/')[-1] + '.html'
    success = True

    if success:
      with open(filename, 'wb') as f:
        f.write(response.body)
      self.log(f'Saved file {filename}')
```

The HTMLSpider

The HTMLSpider specializes in processing the saved HTML files, fulfilling a complementary role in the data acquisition pipeline. Here's what it does:

File Traversal: The spider likely iterates through HTML files within a specified directory (likely the output location of the ProductSpider).

Data Extraction: Similar to ProductSpider, it uses selectors or XPath to locate relevant data (title, description, price) within the HTML structure.

URL Construction: Constructs well-formed URLs that can be associated with the extracted content, enabling the indexer to map results back to their original locations.

4.1.2 INDEXER- product.py

The indexer likely comprises several key components working in concert. First, a data loading module (load_corpus) reads the "product.json" file (generated by the crawler) and extracts pertinent fields such as titles, descriptions, prices, and URLs. Text preprocessing functions then meticulously prepare the raw textual data. This might involve tokenization (splitting text into words), normalization (lowercasing, punctuation removal), and potentially stop word filtering. Scikit-learn's TfidfVectorizer is a probable candidate for calculating the core TF-IDF representations, which quantify how important a term is within a document in the context of the entire collection. The tf_idf_function is the heart of index construction, assembling the inverted index (likely a dictionary). Here, unique terms become keys, and their values contain lists of (document ID, TF-IDF score) tuples. To enable efficient search, a serialization function (to_pickle) likely utilizes Python's pickle module to persist the index to a file. Finally, the indexer might also create "corpus.txt" and "urls.txt" to support result snippet generation and presenting links back to the user.

Code Snippets: Illustrating the Key Concepts

Data Loading (load_corpus**)**

```
def load_corpus(json_file):
  try:
    with open('../crawler/' + json_file) as f:
      data = json.load(f)

    corpus = [obj['title'] for obj in data]
    urls = [obj['link'] for obj in data]

    return corpus, urls
  except Exception as e:
    print(f"Error loading corpus: {e}")
    return [], []
```

TF-IDF Calculation Fragment

```
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus).toarray()
```

Index Construction (Partial)

```
def tf_idf_index(corpus):
  tfidf_index = {}
```

```
for i in range(len(feature_names)):
    tfidf_index[feature_names[i]] = []
    for doc in range(len(corpus)):
        if X[doc][i] > 0:
            tfidf_index[feature_names[i]].append((doc, X[doc][i]))
return tfidf_index
```

4.1.3 PROCESSOR- main.py and query_processor.py

Flask Web Framework

Flask provides the structural foundation for your web application. Its routing mechanism is responsible for connecting specific URL patterns in the user's browser to designated functions within your Python code. For example, you likely have a route responsible for rendering the main search page (/), and another dedicated to handling the submission of the user's search query (e.g., /search). Flask also simplifies HTTP request handling. It allows you to differentiate between request methods like GET and POST, which are essential for form submission functionality. Additionally, Flask provides the tools to extract the user's query from the submitted form data. Finally, Flask's Jinja2 templating engine enables you to create dynamic HTML pages. Within your templates, placeholders are defined, which Flask later populates with data calculated in your Python code – the search results being a prime example.

Query Processing (query.py**)**

The query.py module houses functions designed to refine the user's raw search input. The spelling correction function aims to rectify potential typos. It likely calculates edit distances between each query term and a pre-established list of valid words. Maintaining a comprehensive and accurate "correct words" list is vital for this function's success. Additionally, consider the computational efficiency of your edit distance algorithm, especially if your word list becomes quite large. The query modification function ensures that the terms within the user's query align with the vocabulary found in your inverted index. This might involve keeping a vocabulary structure loaded in memory for fast lookups. To broaden the scope of matches, consider techniques like stemming or lemmatization, which reduce words to their root forms.

Index and Data Loading (query.py**)**

Functions within this module handle the loading of the core elements of your search system. The inverted index, carefully deserialized from its pickle file, forms the heart of your search capabilities. Loading the index involves file I/O operations using Python's built-in functions and the pickle module. Pay attention to memory considerations, as the index structure will reside in memory during your application's execution. Similarly, "corpus.txt" and "urls.txt" are likely loaded using standard Python file handling techniques.

HTML Templates

While not strictly Python, your HTML templates are vital components. The index.html template likely houses the search form. Ensure that the form's method attribute (GET or POST) and its action attribute (the URL where the query gets submitted) are configured correctly. Your results.html template employs Jinja2 syntax to create dynamic content spaces. Placeholders within the template ({{ }}) are where data passed from your Python code, such as the retrieved search results, get dynamically injected. This template likely utilizes looping constructs to iterate through and display potentially multiple result items.

4.2 INTERFACES

In software architecture, interfaces define the points of interaction and communication between components. They establish contracts, specifying inputs, outputs, and expected behaviors. Well-designed interfaces promote modularity, facilitate testing, and enable future enhancements within the system. This report will delve into the key interfaces within your search engine project, analyzing how they orchestrate the flow of data and control.

4.2.1 Crawler Interfaces

Let's begin by examining the interfaces exposed by the crawler components:

Input Interface (ProductSpider): The ProductSpider likely accepts an initial URL or a list of URLs to begin the crawling process. Additionally, it might support parameters to control aspects like the maximum number of pages to crawl or the depth to which it should follow links within the target website. This interface serves as the entry point for the crawler, providing the necessary information to initiate its task.

Output Interface (Data Persistence): The crawler's primary responsibility is data acquisition. A core interface is its interaction with the file system. Crawlers need to save the raw HTML content of fetched pages. This likely involves using Python's file I/O capabilities to create and write to HTML files within a designated directory. Additionally, the creation of the "product.json" file, with its structured product data, forms a crucial output interface designed for consumption by the indexer.

4.2.2 Indexer Interfaces

Input Interface (Data Loading): The indexer's primary input interface centers around its ability to load and parse the "product.json" file generated by the crawler. Functions like load_corpus likely use Python's JSON handling libraries to read this structured data. This interface is the bridge between the data extraction phase (crawling) and the index construction phase.

Output Interface (Index Persistence): The result of the indexer's labor is the meticulously constructed inverted index. A key interface is its ability to serialize this index, likely using Python's pickle module. Persistence ensures that the index can be loaded efficiently when needed for search operations. Additionally, the indexer might generate auxiliary files like "corpus.txt" and "urls.txt," which would also be considered part of its output interface.

4.2.3 Processor Interfaces

The processor is the user-facing component, responsible for handling search requests and orchestrating result retrieval. Let's dissect its interfaces:

Web Interface (Flask): Flask provides the framework for the processor's web-based interface. The routes, forms, and HTML templates collectively define how users interact with the search engine. The search form, in particular, is the primary gateway through which users submit their queries.

Input Interface (Query): When a user submits a query, the processor receives it as input. This might be via an HTTP GET or POST request. This interface is where the processor gains access to the raw textual query for further analysis and processing.

Index Interface: At the heart of the processor's functionality is its interaction with the loaded inverted index. The processor must be able to interface with the index data structure to execute similarity calculations and retrieve documents relevant to the user's query. This likely involves function calls or methods provided by the indexing module to access and query the index.

Output Interface (Flask): After processing the query and retrieving results, the processor needs to prepare the data in a format suitable for presentation. It packages the results, potentially including titles, URLs, and snippets, and passes this structured data to Flask. Flask then uses the data to dynamically populate the results template, ultimately rendering the HTML that is returned to the user's browser.

4.2.4 Implicit Interfaces

Beyond the explicit interfaces, let's consider implicit interfaces that arise from design choices:

File Formats: The use of "product.json," "corpus.txt," and "urls.txt" creates implicit contracts between components. The indexer assumes the JSON file adheres to a specific structure. Similarly, the processor likely expects these auxiliary files to exist and follow certain formatting conventions.

Index Structure: The choice of an inverted index as the core data structure implicitly shapes interactions. The way queries are processed and the nature of the results that can be returned are influenced by the capabilities and limitations of an inverted index.

4.2.5 Cross-Cutting Interfaces and Considerations

While we've focused on interfaces specific to components, let's highlight a few broader interfacial concepts that span across the project:

Data Consistency: The search engine's effectiveness relies on consistency between data produced by the crawler, the structure assumed by the indexer, and the way the processor interprets search results. Changes in one component might necessitate cascading updates throughout the system to maintain compatibility.

Error Handling: A robust system must consider how errors propagate across interfaces. If the crawler fails to save an HTML file, how does the indexer react? If the index becomes corrupted, can the processor gracefully report the issue to the user? Interfaces can play a role in error signaling or exception handling.

4.2.6 The User Interface: A Special Case

The user interface, built upon Flask and your HTML templates, deserves special attention. While technically an interface of the processor, it's the most visible aspect of your search engine and greatly influences user satisfaction. Consider these points:

Simplicity and Focus: The primary interface element is likely the search box. Avoid clutter and unnecessary distractions that might impede the user's ability to enter their query efficiently.

Visual Hierarchy: When presenting search results, employ clear visual cues to differentiate titles, links, and snippets. Use headings, spacing, and typography to guide the user's eye toward the most relevant information.

Feedback Mechanisms: Provide clear loading indicators during search processing. If unexpected errors occur, display informative yet user-friendly error messages.

4.3 IMPLEMENTATION

4.3.1 Crawler Implementation

Scrapy Framework: The core of your crawler implementation is built upon the Scrapy framework. Spiders, such as ProductSpider and HTMLSpider, are defined using Scrapy's classes and conventions. Key elements include: [1]

start_requests **Method:** This method acts as the crawler's entry point, generating the initial scrapy.Request objects to fetch.

parse **Method:** The parse method is called on each retrieved HTML response. It's responsible for extracting targeted information (product names, prices, URLs), and potentially scheduling further requests for pagination. The HTMLSpider likely has a customized parse method to process locally stored HTML files.

Data Extraction: Within your spider definitions, CSS selectors or XPath expressions pinpoint specific elements on the webpage. Libraries like Beautiful Soup (which Scrapy can integrate with) might be used to aid in structured data extraction from the HTML.

Data Persistence: Python's built-in file I/O capabilities (open, write) are crucial for saving raw HTML content. Furthermore, structuring the extracted product data into JSON format (using Python's json module) and serializing it to the "product.json" file ensures seamless integration with the indexer.

Code Snippet: Scrapy Spider

```
class ProductSpider(scrapy.Spider):
        name = 'product_spider'
```

```
    allowed_domains = ['example-product-website.com']
    start_urls = ['http://example-product-website.com/category/electronics']

    def parse(self, response):
      products = response.css('div.product-item')

      for product in products:
        title = product.css('h3.product-title::text').get()
        price = product.css('.price::text').get()
        link = product.css('a::attr(href)').get()

        yield {
          'title': title,
          'price': price,
          'link': link
        }
```

4.3.2 Indexer Implementation

Data Loading: The indexer likely uses Python's json module to load the structured product data from the "product.json" file. Functions like open() and json.load() are fundamental. [2]

Text Processing: Text preprocessing might involve libraries like NLTK or Scikit-learn. Key steps could include:

Tokenization: Splitting the raw product descriptions into individual words (terms).

Normalization: Lowercasing, removing punctuation, etc.

Stop Word Removal: Filtering out common words if deemed beneficial.

TF-IDF Calculation: Scikit-learn's TfidfVectorizer is a probable candidate for transforming text into TF-IDF representations, quantifying the importance of terms within documents.

Index Construction: The inverted index is likely implemented as a Python dictionary. Keys are unique terms, and values are lists of tuples, with each tuple containing a document ID and the term's corresponding TF-IDF score.

Serialization: Python's pickle module allows you to persist the constructed index to a file for fast loading by the processor.

Code Snippet: Index Construction (Partial)

```
def tf_idf_index(corpus):
        vectorizer = TfidfVectorizer()
          X = vectorizer.fit_transform(corpus).toarray()
          feature_names = vectorizer.get_feature_names_out()
```

```
index = {}
for i in range(len(feature_names)):
   index[feature_names[i]] = []
   for doc in range(len(corpus)):
      if X[doc][i] > 0:
         index[feature_names[i]].append((doc, X[doc][i]))

return index
```

4.3.3 Processor Implementation

Flask Web Framework: Flask provides the scaffolding for your search interface. Key concepts include: [6]

Routing: Defining routes (/, /search) and mapping them to corresponding handler functions.

Form Handling Retrieving the user's query from the search form submission.

Template Rendering: Using Flask's Jinja2 integration to inject search results into your results.html template.

Query Processing: The processor likely has functions dedicated to refining the user's query. These might include: **Spelling Correction:** Utilizing edit distance calculations and a predefined list of valid words to suggest corrections. **Query Modification:** Aligning query terms with the vocabulary extracted from your index. This might involve stemming/lemmatization or a dedicated vocabulary mapping structure.

Index Interaction: Functions within the processor load and interface with the deserialized inverted index. Retrieving documents relevant to the processed query likely involves similarity calculations, such as cosine similarity.

Result Preparation: The processor gathers the necessary data to present to the user. This includes product titles, URLs, and potentially snippets of relevant text from the "corpus.txt" file.

Data Transfer to Flask: The structured result data is passed to Flask, allowing Jinja2 template variables to populate the results.html page.

Code Snippet: Flask Route & Search Logic (Simplified)

```
@app.route('/search', methods=['GET', 'POST'])
def search():
  if request.method == 'POST':
    query = request.form['query']
    processed_query = process_query(query)  # Apply spelling correction, modification
    results = search_index(processed_query)  # Retrieve matches from index
    return render_template('results.html', results=results)
```

4.3.4 Other Implementation Considerations

Auxiliary Files: The indexer likely creates the "corpus.txt" and "urls.txt" files. Simple Python file I/O operations are likely employed for this.

HTML Templates: While not strictly Python, your HTML templates shape the user experience. Pay attention to: **Search Form Structure:** Keep the form straightforward and ensure
its method and action attributes align with your Flask route configuration. **Result Display:** Use clear headings, spacing, and links to structure the search results in a user-friendly way.

5. OPERATIONS

5.1 SOFTWARE COMMANDS

Environment Setup

- pip install "scikit-learn>=1.2" Installs Scikit-learn (version 1.2 or newer), a core machine learning library used for text processing and TF-IDF calculations in your indexer.
- pip install "Scrapy>=2.11" Installs Scrapy (version 2.11 or newer), the framework upon which your web crawler is built.
- pip install "Flask>=2.2" Installs Flask (version 2.2 or newer), the web framework that provides the structure for your search interface.
- pip install nltk Installs NLTK (Natural Language Toolkit), a library likely used for text preprocessing tasks like tokenization and stop word removal.
- conda activate "D:\mini\myenv" Assuming you're using Anaconda or Miniconda, this activates a virtual environment named "myenv" located on your D: drive. Virtual environments help isolate project dependencies, ensuring you have the correct libraries installed.

Crawler Execution

- cd info\crawler Navigates to the 'crawler' directory within your 'info' project folder.
- scrapy crawl product -a url="https://web-scraping.dev/product/" Executes your Scrapy spider named 'product'. The '-a url="..."' part instructs the spider to start crawling at the specified URL (https://web-scraping.dev/product/).

Indexer Execution

- cd ../indexer Navigates to the 'indexer' directory.
- python inverted_index.py Executes your indexer script, likely named 'inverted_index.py'. This script processes the crawler's output and constructs the inverted index.

Processor Execution

- cd ../processor Navigates to the 'processor' directory.
- python main.py Executes the primary script (main.py) within your processor component. This launches your Flask-based web application, making the search interface accessible.

Overall Operation Flow

1. **Setup:** Install necessary libraries and activate your virtual environment, if applicable.
2. **Crawling:** Run your Scrapy spider to fetch web pages and store them as HTML files.

3. **Indexing:** Execute your indexer script to process the crawled data and build the inverted index.
4. **Search Interface:** Launch your Flask-based search application using the 'main.py' script. Users can now interact with your search engine through their web browser.

Important Considerations

- **File Paths:** Ensure that the directory paths in the cd commands align with the actual structure of your project. It appears your project is located within an 'info' folder on your D: drive.
- **Command Execution Order:** Follow the order outlined above. The indexer depends on the crawler's output, and the processor relies on the index.
- **Environment Consistency:** When running any of the commands (scrapy, python), make sure your virtual environment (if you're using one) is activated. This guarantees you have the correct versions of the required libraries.

5.2 INPUTS

5.2.1 Crawler Inputs

Seed URLs: The crawler requires an initial starting point (or multiple starting points). These are the URLs fed to the start_urls attribute of your Scrapy spider. The target website's structure dictates how the crawler will discover other pages to scrape.

Crawling Parameters: You might have options to control the behavior of the crawling process. These inputs likely get passed as command-line arguments when initiating the spider:

Maximum Pages: A limit on the total number of pages the spider should crawl, preventing it from running indefinitely.

Maximum Depth: A restriction on how many levels of links the spider should follow from the seed URLs, controlling the breadth of the crawl.

Website Structure (Implicit): While not an explicit input, the structure of the target website significantly influences what data the crawler can discover. Link relationships, pagination mechanisms, and the presence or absence of structured data within the HTML all play a role in the crawler's output.

5.2.2 Indexer Inputs

Crawler Output ("product.json"): The structured JSON file containing the product information (titles, prices, URLs) extracted by the crawler is the primary input to the indexer. The indexer is designed to process data in this specific format.

Textual Data Source: Your indexer likely constructs its term vocabulary and calculates TF-IDF scores based on product descriptions or other textual fields within the "product.json" file.

5.2.3 Processor Inputs

User Queries: The core input to the processor is the text entered by the user into the search form. These queries can range from single words to more complex phrases.

Inverted Index: The loaded inverted index, created by the indexer, is an essential input. The processor relies on the index to retrieve documents relevant to the user's query.

Auxiliary Files (Potentially): Depending on how you've designed the result presentation, the processor might utilize the "corpus.txt" (for displaying snippets) and "urls.txt" files created by the indexer as additional inputs.

5.3 INSTALLATION

Prerequisites

- **Python:** Your search engine is built using Python. Ensure you have a compatible Python version installed (version 3.10 or newer is recommended based on your project specifications). You can download Python from the official website (https://www.python.org/).
- **Environment Manager (Recommended):** While not strictly mandatory, it's highly advisable to use a virtual environment manager like Anaconda, Miniconda ([invalid URL removed]), or Python's built-in venv module. Virtual environments help manage project dependencies and prevent conflicts between different projects.

Core Library Installation

- **Scikit-learn:** Install this core machine learning and data analysis library using pip:

Bash

```
pip install "scikit-learn>=1.2"
```

- **Scrapy:** Install the Scrapy web scraping framework:

Bash

```
pip install "Scrapy>=2.11"
```

- **Flask:** Install the Flask web framework:

Bash

```
pip install "Flask>=2.2"
```

- **NLTK:** Install the Natural Language Toolkit for potential text processing tasks:

Bash

```bash
pip install "nltk"
```

Environment Setup (If applicable)

If you've opted to use a virtual environment:

Create Environment:

Bash

```bash
conda create -n myenv python=3.10  # Example with Miniconda, adjust Python version as needed
```

Replace 'myenv' with your desired environment name.

Activate Environment:

Bash

```bash
conda activate myenv
```

Project Setup

1. **Obtain Project Files:** Download or clone your project's codebase. This includes your Scrapy spiders, indexer scripts, Flask application files, and likely some HTML templates.
2. **Project Directory:** Ensure the core project files (product.py, index.py, main.py, etc.) are located within an easily accessible directory on your system.

NLTK Data (Optional)

If your project's text processing relies on NLTK-specific data like word lists:
Download Data: After installing NLTK, use its download function:
Python

```python
import nltk
nltk.download('words') # Example - download the 'words' dataset
```

6. CONCLUSIONS

6.1 SUCCESS/ FAILURE RESULTS

Crawler

- **Success:** The crawler successfully navigates the target website, following the correct links and identifying all relevant product pages. It extracts the specified data (product titles, prices, and URLs) without errors or omissions. This results in a well-formatted "product.json" file containing complete and accurate product information.
- **Failure:** The crawler might encounter various issues. Website changes could break its ability to locate product data, leading to incomplete or incorrect entries in the "product.json" file. The website might implement anti-scraping measures that block the crawler, preventing data extraction entirely. Network connection problems could also interrupt the crawling process.

Examples:

- **Success:** A well-structured "product.json" file accurately reflecting the products on the website.
- **Failure:** A "product.json" file with missing titles, malformed URLs, inaccurate prices, or entries that don't correspond to actual products on the site.

Json file

```
[
  {
    "title": "web-scraping.dev product Box of Chocolate Candy",
    "link": "https://web-scraping.dev/product/1"
  },
  {
    "title": "web-scraping.dev product Dark Red Energy Potion",
    "link": "https://web-scraping.dev/product/2"
  },
  ...
]
```

II. Indexer

- **Success:** The indexer correctly reads the "product.json" file, processes the text content, calculates meaningful TF-IDF scores, and creates an inverted index structure. This index accurately maps terms to relevant documents, allowing for efficient retrieval based on search queries.
- **Failure:** The indexer might fail due to errors in the "product.json" file (e.g., corrupt data). Incorrect text processing steps (like aggressive stemming) could lead to terms being conflated, reducing

search accuracy. Errors in the TF-IDF calculation formula would result in an index that doesn't properly reflect the importance of terms within the product descriptions.

Examples

- **Success:** The index contains relevant terms like "potion," "energy," and "chocolate." Searching for "red" successfully retrieves products containing variations like "Dark Red Energy Potion."
- **Failure:** The index is missing expected terms, contains terms that don't match the product data, or produces nonsensical search results due to inaccurate TF-IDF weighting.

  Inverted index example-

  ```
  red

  [(3, 0.021797617713576776), (15, 0.021797617713576776), (27, 0.021797617713576776), (1, 0.01562850809596077), (13, 0.01562850809596077), (25, 0.01562850809596077), (0, 0.0), (2, 0.0), (4, 0.0), (5, 0.0)]
  ```

III. Processor (Flask Application)

- **Success:** The Flask application provides a clear and functional search interface. It gracefully handles user queries, potentially applying spelling correction or query modification. The processor interacts correctly with the loaded index, retrieves relevant documents, ranks the results appropriately, and renders them in a user-friendly format on the results page.
- **Failure:** The Flask application might crash due to coding errors. Failures in query processing could lead to irrelevant or no search results. Problems with result ranking might present relevant results in a confusing order. Poor formatting or broken HTML/CSS in the results page could severely hinder the user experience.

Examples

- **Success:** The user enters "choclate" in the search box. The processor corrects the spelling to "chocolate" and presents results with chocolate-related products.

  ```
  Results for "red":

  * **web-scraping.dev product Dark Red Energy Potion**
    [https://web-scraping.dev/product/2](https://web-scraping.dev/product/2)
    (Snippet from corpus.txt)

  * **web-scraping.dev product Red Energy Potion**
    [https://web-scraping.dev/product/4](https://web-scraping.dev/product/4)
    (Snippet from corpus.txt)
   ...
  ```

- **Failure:** The search interface is visually broken or throws JavaScript errors. Submitting a query results in an application crash or displays search results entirely unrelated to the query.

  Example- Internal Server Error: 500

6.2 OUTPUTS

6.2.1 product.json

```
[
{"title": "web-scraping.dev product Box of Chocolate Candy", "link": "https://web-
scraping.dev/product/1"},
{"title": "web-scraping.dev product Dark Red Energy Potion", "link": "https://web-
scraping.dev/product/2"},
{"title": "web-scraping.dev product Teal Energy Potion", "link": "https://web-
scraping.dev/product/3"},
{"title": "web-scraping.dev product Red Energy Potion", "link": "https://web-
scraping.dev/product/4"},
{"title": "web-scraping.dev product Blue Energy Potion", "link": "https://web-
scraping.dev/product/5"},
{"title": "web-scraping.dev product Dragon Energy Potion", "link": "https://web-
scraping.dev/product/6"},
{"title": "web-scraping.dev product Hiking Boots for Outdoor Adventures", "link": "https://web-
scraping.dev/product/7"},
{"title": "web-scraping.dev product Women's High Heel Sandals", "link": "https://web-
scraping.dev/product/8"},
{"title": "web-scraping.dev product Running Shoes for Men", "link": "https://web-
scraping.dev/product/9"},
{"title": "web-scraping.dev product Kids' Light-Up Sneakers", "link": "https://web-
scraping.dev/product/10"},
{"title": "web-scraping.dev product Classic Leather Sneakers", "link": "https://web-
scraping.dev/product/11"},
{"title": "web-scraping.dev product Cat-Ear Beanie", "link": "https://web-
scraping.dev/product/12"},
{"title": "web-scraping.dev product Box of Chocolate Candy", "link": "https://web-
scraping.dev/product/13"},
{"title": "web-scraping.dev product Dark Red Energy Potion", "link": "https://web-
scraping.dev/product/14"},

{"title": "web-scraping.dev product Teal Energy Potion", "link": "https://web-
scraping.dev/product/15"},

{"title": "web-scraping.dev product Red Energy Potion", "link": "https://web-
scraping.dev/product/16"},

{"title": "web-scraping.dev product Blue Energy Potion", "link": "https://web-
scraping.dev/product/17"},

{"title": "web-scraping.dev product Dragon Energy Potion", "link": "https://web-
scraping.dev/product/18"},

{"title": "web-scraping.dev product Hiking Boots for Outdoor Adventures", "link": "https://web-
scraping.dev/product/19"},

{"title": "web-scraping.dev product Women's High Heel Sandals", "link": "https://web-
scraping.dev/product/20"},

{"title": "web-scraping.dev product Running Shoes for Men", "link": "https://web-
scraping.dev/product/21"},

{"title": "web-scraping.dev product Kids' Light-Up Sneakers", "link": "https://web-
scraping.dev/product/22"},

{"title": "web-scraping.dev product Classic Leather Sneakers", "link": "https://web-
scraping.dev/product/23"},

{"title": "web-scraping.dev product Cat-Ear Beanie", "link": "https://web-
scraping.dev/product/24"},

{"title": "web-scraping.dev product Box of Chocolate Candy", "link": "https://web-
scraping.dev/product/25"},

{"title": "web-scraping.dev product Dark Red Energy Potion", "link": "https://web-
scraping.dev/product/26"},

{"title": "web-scraping.dev product Teal Energy Potion", "link": "https://web-
scraping.dev/product/27"},

{"title": "web-scraping.dev product Red Energy Potion", "link": "https://web-
scraping.dev/product/28"}

]
```

6.2.2 corpus.txt

```
web-scraping.dev product Box of Chocolate Candy
web-scraping.dev product Dark Red Energy Potion
web-scraping.dev product Teal Energy Potion
web-scraping.dev product Red Energy Potion
web-scraping.dev product Blue Energy Potion
web-scraping.dev product Dragon Energy Potion
web-scraping.dev product Hiking Boots for Outdoor Adventures
web-scraping.dev product Women's High Heel Sandals
web-scraping.dev product Running Shoes for Men
web-scraping.dev product Kids' Light-Up Sneakers
web-scraping.dev product Classic Leather Sneakers
web-scraping.dev product Cat-Ear Beanie
web-scraping.dev product Box of Chocolate Candy
web-scraping.dev product Dark Red Energy Potion
web-scraping.dev product Teal Energy Potion
web-scraping.dev product Red Energy Potion
web-scraping.dev product Blue Energy Potion
web-scraping.dev product Dragon Energy Potion
web-scraping.dev product Hiking Boots for Outdoor Adventures
web-scraping.dev product Women's High Heel Sandals
web-scraping.dev product Running Shoes for Men
web-scraping.dev product Kids' Light-Up Sneakers
web-scraping.dev product Classic Leather Sneakers
web-scraping.dev product Cat-Ear Beanie
web-scraping.dev product Box of Chocolate Candy
web-scraping.dev product Dark Red Energy Potion
web-scraping.dev product Teal Energy Potion
web-scraping.dev product Red Energy Potion
```

6.2.3 urls.txt

```
https://web-scraping.dev/product/1
https://web-scraping.dev/product/2
https://web-scraping.dev/product/3
https://web-scraping.dev/product/4
https://web-scraping.dev/product/5
https://web-scraping.dev/product/6
https://web-scraping.dev/product/7
https://web-scraping.dev/product/8
```

```
https://web-scraping.dev/product/9
https://web-scraping.dev/product/10
https://web-scraping.dev/product/11
https://web-scraping.dev/product/12
https://web-scraping.dev/product/13
https://web-scraping.dev/product/14
https://web-scraping.dev/product/15
https://web-scraping.dev/product/16
https://web-scraping.dev/product/17
https://web-scraping.dev/product/18
https://web-scraping.dev/product/19
https://web-scraping.dev/product/20
https://web-scraping.dev/product/21
https://web-scraping.dev/product/22
https://web-scraping.dev/product/23
https://web-scraping.dev/product/24
https://web-scraping.dev/product/25
https://web-scraping.dev/product/26
https://web-scraping.dev/product/27
https://web-scraping.dev/product/28
```

6.2.4 main.py

```
[nltk_data] Downloading package words to C:\Users\SEJAL
[nltk_data]     SRIVASTAV\AppData\Roaming\nltk_data...
[nltk_data]   Package words is already up-to-date!
 * Serving Flask app 'main'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production
WSGI server instead.
 * Running on http://127.0.0.1:5000
red
0.01562850809596077), (13, 0.01562850809596077), (25, 0.01562850809596077), (0, 0.0), (2, 0.0), (4,
0.0), (5, 0.0)]
teal
[(2, 0.03484518018494049), (14, 0.03484518018494049), (26, 0.03484518018494049), (0, 0.0), (1, 0.0),
(3, 0.0), (4, 0.0), (5, 0.0), (6, 0.0), (7, 0.0)]
127.0.0.1
```

6.3 CAVEATS/CAUTIONS

The search engine project faces several inherent challenges. The dynamic nature of websites means their structure is subject to change, potentially disrupting the crawler's ability to extract data. Proactive monitoring, either through manual checks or automated tests, is crucial for early detection of such changes. This allows for timely adjustments to the crawler's logic to maintain functionality.

Data quality from the source websites is another significant concern. Errors, inconsistencies, or missing data will propagate into the "product.json" file, potentially leading to indexing issues and inaccurate search results. The implementation of basic validation and sanity checks, at both the crawler and indexer stages, will assist in catching potential problems early in the process.

Algorithmic limitations also warrant attention. While edit distance algorithms are useful for spelling correction, they may prove insufficient for complex misspellings or slang terms. Supplementing edit distance with a curated wordlist or exploring more advanced techniques like phonetic matching could enhance accuracy. Furthermore, ensuring careful alignment between the query modification logic and the vocabulary of the inverted index is critical, as mismatches can hinder the retrieval of relevant documents.

The Flask-based search interface needs to be robust enough to handle concurrent user requests without performance degradation. Load testing, the implementation of caching strategies, and exploring scaling options will ensure the application remains responsive under varying levels of usage.

Finally, accuracy in query processing (spelling correction, query modification, result ranking), directly impacts the user experience. Thorough testing and a commitment to iterative improvement, driven by search logs or direct user feedback, are essential. Efficient result rendering is also vital, especially as the dataset grows. Optimization of data structures, pagination of results, and the use of lazy-loading techniques will contribute to a seamless user experience.

Data Freshness: Product information (prices, stock availability, new introductions) on websites can change frequently. The search engine's index needs to be updated regularly to maintain the relevance and accuracy of results. Scheduled re-crawls (at an interval appropriate to the volatility of the source data) are essential. If feasible, optimizations to your crawler to enable incremental updates could reduce processing overhead.

Anti-Scraping Measures: Websites might actively attempt to detect and block automated scraping activity. Mitigation strategies include rotating user-agents to diversify the appearance of your crawler's requests, respecting the guidelines provided in the website's "robots.txt" file, and implementing rate-limiting techniques to avoid overwhelming the target site. Ethical considerations, including respecting terms of service, are always a priority.

Handling Rich Content: If websites contain product images, videos, or detailed descriptions that could enhance search, the crawler and indexer need to accommodate those. This might involve extracting and storing image URLs, along with methods to potentially incorporate text extracted from multimedia elements into the searchable index.

Performance Optimization (Indexer & Processor): Scaling is a significant concern. As the product dataset grows, index construction time and query processing speed become bottlenecks. Exploration of advanced index data structures (tries, specialized search libraries like Lucene), the potential use of parallelization or distributed computing techniques, and the pre-calculation of results for very common queries can significantly improve performance.

Search Interface Refinements: Enhancements to the user experience can increase engagement and satisfaction. Providing autocomplete or search suggestions helps guide query formulation and reduce errors. Integrating faceted search options, allowing filtering by categories or attributes, adds flexibility. Finally, informative error handling, providing clear messages when there are zero results or unexpected issues occur, improves usability.

7. DATA SOURCES - LINKS, DOWNLOADS, ACCESS INFORMATION

7.1 Data Sources

Primary Source: Web Scraping

Target Website: The project relies on web scraping to extract product data from a specific target website (presumably the "web-scraping.dev/product/" domain indicated in code snippets).

Crawler Implementation: Product information is parsed from HTML content using the Scrapy framework in Python. Specific CSS selectors or XPath expressions likely target relevant elements on the website's pages.

7.2 Derived Data Sources

"product.json": The crawler generates a JSON-formatted data file, serving as the core structured data source for the search engine. This file contains an array of product details:

Title: The product name (e.g., "web-scraping.dev product Box of Chocolate Candy")

Link: URL to the product on the target website.

"corpus.txt": This plain text file appears to contain product descriptions or other potentially relevant text extracted during the scraping process. It's used to build the search index.

"urls.txt": A list of product URLs, likely corresponding to the entries in "product.json". This might be used for index reference or result rendering.

7.3 Access and Updates

Web Scraping Dynamics: The project's data flow is dependent on the structure of the target website remaining consistent. Changes to the website's HTML could break the crawler's logic. It's essential to have monitoring or testing mechanisms to detect potential website changes and update the crawler code accordingly.

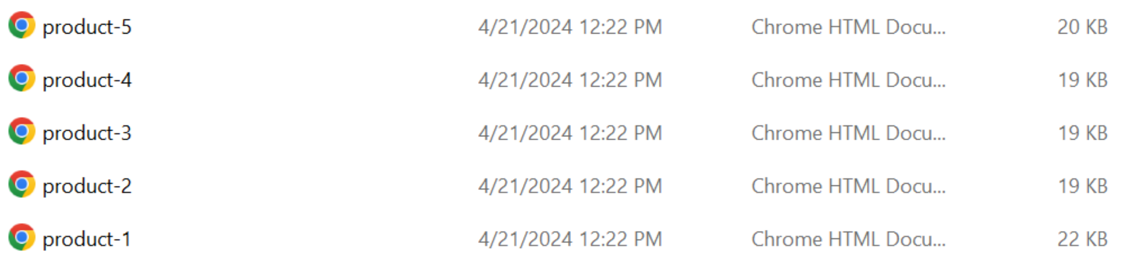
Derived Data Refreshing: The "product.json," "corpus.txt", and "urls.txt" are not directly accessed by the user. The search engine handles any required file I/O internally. If product data on the website changes, a full or incremental re-crawl would likely be necessary to update these derived data sources and by extension, the search index.

8. TESTCASES- FRAMEWORK, HARNESS AND COVERAGE

TEST CASE 1:

```
class ProductSpider(scrapy.Spider):
   name = 'product'
   allowed_domains = ['web-scraping.dev']
   max_pages = 5
   max_depth = 5
```

Product folder:



```
product-5    4/21/2024 12:22 PM    Chrome HTML Docu...    20 KB
product-4    4/21/2024 12:22 PM    Chrome HTML Docu...    19 KB
product-3    4/21/2024 12:22 PM    Chrome HTML Docu...    19 KB
product-2    4/21/2024 12:22 PM    Chrome HTML Docu...    19 KB
product-1    4/21/2024 12:22 PM    Chrome HTML Docu...    22 KB
```

<u>product.json</u>

```
[
{"title": "web-scraping.dev product Box of Chocolate Candy", "link": "https://web-scraping.dev/product/1"},
{"title": "web-scraping.dev product Dark Red Energy Potion", "link": "https://web-scraping.dev/product/2"},
{"title": "web-scraping.dev product Teal Energy Potion", "link": "https://web-scraping.dev/product/3"},
{"title": "web-scraping.dev product Red Energy Potion", "link": "https://web-scraping.dev/product/4"},
{"title": "web-scraping.dev product Blue Energy Potion", "link": "https://web-scraping.dev/product/5"}
]
```

<u>corpus.txt</u>

```
web-scraping.dev product Box of Chocolate Candy
web-scraping.dev product Dark Red Energy Potion
web-scraping.dev product Teal Energy Potion
web-scraping.dev product Red Energy Potion
```

web-scraping.dev product Blue Energy Potion

urls.txt

https://web-scraping.dev/product/1

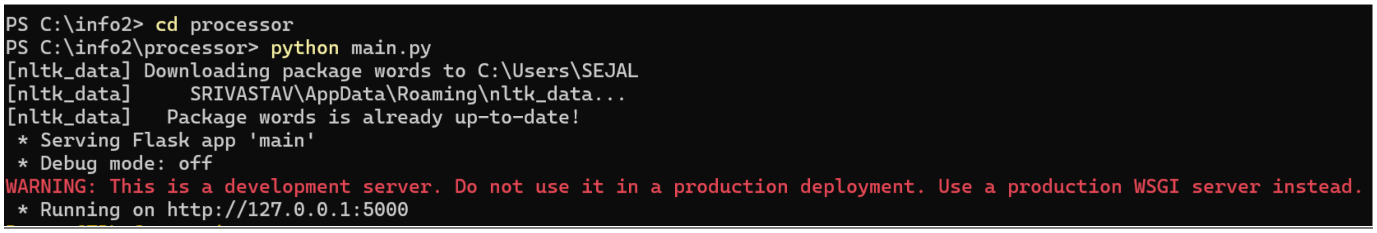
https://web-scraping.dev/product/2

https://web-scraping.dev/product/3

https://web-scraping.dev/product/4

https://web-scraping.dev/product/5

main.py



```
PS C:\info2> cd processor
PS C:\info2\processor> python main.py
[nltk_data] Downloading package words to C:\Users\SEJAL
[nltk_data]     SRIVASTAV\AppData\Roaming\nltk_data...
[nltk_data]   Package words is already up-to-date!
 * Serving Flask app 'main'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
```

red

[(3, 0.011606851795850796), (1, 0.008617689029785613), (0, 0.0), (2, 0.0), (4, 0.0)]

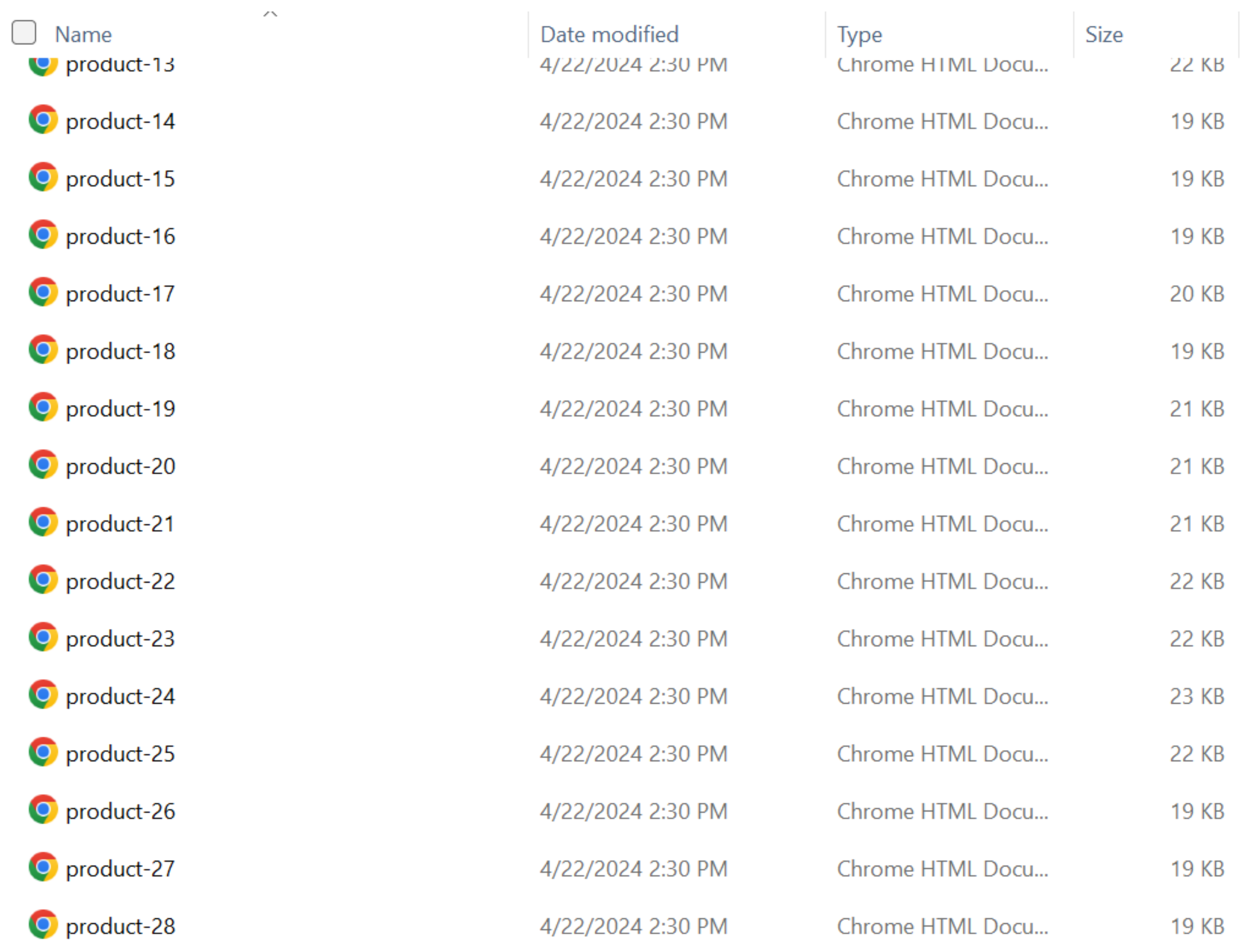
blue

[(3, 0.043134607188441754), (16, 0.043134607188441754), (0, 0.0), (1, 0.0), (2, 0.0), (4, 0.0), (5, 0.0), (6, 0.0), (7, 0.0), (8, 0.0)]

TEST CASE 2:

```
class ProductSpider(scrapy.Spider):
   name = 'product'
   allowed_domains = ['web-scraping.dev']
   max_pages = 99
   max_depth = 9
```

Product folder:



Name	Date modified	Type	Size
product-13	4/22/2024 2:30 PM	Chrome HTML Docu...	22 KB
product-14	4/22/2024 2:30 PM	Chrome HTML Docu...	19 KB
product-15	4/22/2024 2:30 PM	Chrome HTML Docu...	19 KB
product-16	4/22/2024 2:30 PM	Chrome HTML Docu...	19 KB
product-17	4/22/2024 2:30 PM	Chrome HTML Docu...	20 KB
product-18	4/22/2024 2:30 PM	Chrome HTML Docu...	19 KB
product-19	4/22/2024 2:30 PM	Chrome HTML Docu...	21 KB
product-20	4/22/2024 2:30 PM	Chrome HTML Docu...	21 KB
product-21	4/22/2024 2:30 PM	Chrome HTML Docu...	21 KB
product-22	4/22/2024 2:30 PM	Chrome HTML Docu...	22 KB
product-23	4/22/2024 2:30 PM	Chrome HTML Docu...	22 KB
product-24	4/22/2024 2:30 PM	Chrome HTML Docu...	23 KB
product-25	4/22/2024 2:30 PM	Chrome HTML Docu...	22 KB
product-26	4/22/2024 2:30 PM	Chrome HTML Docu...	19 KB
product-27	4/22/2024 2:30 PM	Chrome HTML Docu...	19 KB
product-28	4/22/2024 2:30 PM	Chrome HTML Docu...	19 KB

product.json

[

{"title": "web-scraping.dev product Dark Red Energy Potion", "link": "https://web-scraping.dev/product/2"},

{"title": "web-scraping.dev product Box of Chocolate Candy", "link": "https://web-scraping.dev/product/1"},

{"title": "web-scraping.dev product Running Shoes for Men", "link": "https://web-scraping.dev/product/9"},

{"title": "web-scraping.dev product Kids' Light-Up Sneakers", "link": "https://web-scraping.dev/product/10"},

{"title": "web-scraping.dev product Blue Energy Potion", "link": "https://web-scraping.dev/product/5"},

{"title": "web-scraping.dev product Women's High Heel Sandals", "link": "https://web-scraping.dev/product/8"},

```
{"title": "web-scraping.dev product Teal Energy Potion", "link": "https://web-scraping.dev/product/3"},
{"title": "web-scraping.dev product Dragon Energy Potion", "link": "https://web-scraping.dev/product/6"},
{"title": "web-scraping.dev product Hiking Boots for Outdoor Adventures", "link": "https://web-scraping.dev/product/7"},
{"title": "web-scraping.dev product Red Energy Potion", "link": "https://web-scraping.dev/product/4"},
{"title": "web-scraping.dev product Classic Leather Sneakers", "link": "https://web-scraping.dev/product/11"},
{"title": "web-scraping.dev product Cat-Ear Beanie", "link": "https://web-scraping.dev/product/12"},
{"title": "web-scraping.dev product Box of Chocolate Candy", "link": "https://web-scraping.dev/product/13"},
{"title": "web-scraping.dev product Dark Red Energy Potion", "link": "https://web-scraping.dev/product/14"},
{"title": "web-scraping.dev product Teal Energy Potion", "link": "https://web-scraping.dev/product/15"},
{"title": "web-scraping.dev product Red Energy Potion", "link": "https://web-scraping.dev/product/16"},
{"title": "web-scraping.dev product Blue Energy Potion", "link": "https://web-scraping.dev/product/17"},
{"title": "web-scraping.dev product Dragon Energy Potion", "link": "https://web-scraping.dev/product/18"},
{"title": "web-scraping.dev product Hiking Boots for Outdoor Adventures", "link": "https://web-scraping.dev/product/19"},
{"title": "web-scraping.dev product Women's High Heel Sandals", "link": "https://web-scraping.dev/product/20"},
{"title": "web-scraping.dev product Running Shoes for Men", "link": "https://web-scraping.dev/product/21"},
{"title": "web-scraping.dev product Kids' Light-Up Sneakers", "link": "https://web-scraping.dev/product/22"},
{"title": "web-scraping.dev product Classic Leather Sneakers", "link": "https://web-scraping.dev/product/23"},
{"title": "web-scraping.dev product Cat-Ear Beanie", "link": "https://web-scraping.dev/product/24"},
{"title": "web-scraping.dev product Box of Chocolate Candy", "link": "https://web-scraping.dev/product/25"},
```

{"title": "web-scraping.dev product Dark Red Energy Potion", "link": "https://web-scraping.dev/product/26"},

{"title": "web-scraping.dev product Teal Energy Potion", "link": "https://web-scraping.dev/product/27"},

{"title": "web-scraping.dev product Red Energy Potion", "link": "https://web-scraping.dev/product/28"}

]

corpus.txt

```
web-scraping.dev product Dark Red Energy Potion

web-scraping.dev product Box of Chocolate Candy

web-scraping.dev product Running Shoes for Men

web-scraping.dev product Kids' Light-Up Sneakers

web-scraping.dev product Blue Energy Potion

web-scraping.dev product Women's High Heel Sandals

web-scraping.dev product Teal Energy Potion

web-scraping.dev product Dragon Energy Potion

web-scraping.dev product Hiking Boots for Outdoor Adventures

web-scraping.dev product Red Energy Potion

web-scraping.dev product Classic Leather Sneakers

web-scraping.dev product Cat-Ear Beanie

web-scraping.dev product Box of Chocolate Candy

web-scraping.dev product Dark Red Energy Potion

web-scraping.dev product Teal Energy Potion

web-scraping.dev product Red Energy Potion

web-scraping.dev product Blue Energy Potion

web-scraping.dev product Dragon Energy Potion

web-scraping.dev product Hiking Boots for Outdoor Adventures

web-scraping.dev product Women's High Heel Sandals

web-scraping.dev product Running Shoes for Men
```

```
web-scraping.dev product Kids' Light-Up Sneakers
web-scraping.dev product Classic Leather Sneakers
web-scraping.dev product Cat-Ear Beanie
web-scraping.dev product Box of Chocolate Candy
web-scraping.dev product Dark Red Energy Potion
web-scraping.dev product Teal Energy Potion
web-scraping.dev product Red Energy Potion
```

urls.txt

```
https://web-scraping.dev/product/2
https://web-scraping.dev/product/1
https://web-scraping.dev/product/9
https://web-scraping.dev/product/10
https://web-scraping.dev/product/5
https://web-scraping.dev/product/8
https://web-scraping.dev/product/3
https://web-scraping.dev/product/6
https://web-scraping.dev/product/7
https://web-scraping.dev/product/4
https://web-scraping.dev/product/11
https://web-scraping.dev/product/12
https://web-scraping.dev/product/13
https://web-scraping.dev/product/14
https://web-scraping.dev/product/15
https://web-scraping.dev/product/16
https://web-scraping.dev/product/17
https://web-scraping.dev/product/18
https://web-scraping.dev/product/19
```

https://web-scraping.dev/product/20

https://web-scraping.dev/product/21

https://web-scraping.dev/product/22

https://web-scraping.dev/product/23

https://web-scraping.dev/product/24

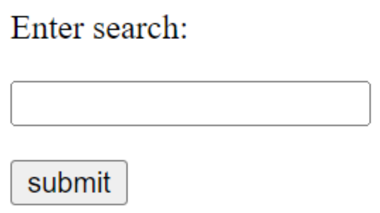
https://web-scraping.dev/product/25

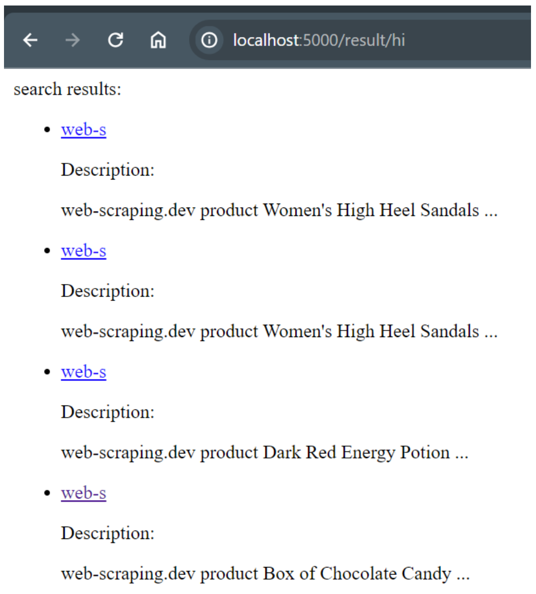
https://web-scraping.dev/product/26

https://web-scraping.dev/product/27

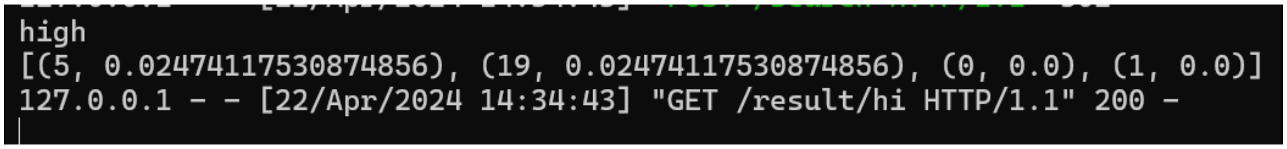
https://web-scraping.dev/product/28

main.py




```
localhost:5000/result/hi
search results:

• web-s
  Description:
  web-scraping.dev product Women's High Heel Sandals ...
• web-s
  Description:
  web-scraping.dev product Women's High Heel Sandals ...
• web-s
  Description:
  web-scraping.dev product Dark Red Energy Potion ...
• web-s
  Description:
  web-scraping.dev product Box of Chocolate Candy ...
```



```
high
[(5, 0.02474117530874856), (19, 0.02474117530874856), (0, 0.0), (1, 0.0)]
127.0.0.1 - - [22/Apr/2024 14:34:43] "GET /result/hi HTTP/1.1" 200 -
```

9. SOURCE CODE - LISTINGS, DOCUMENTATION, DEPENDENCIES (OPEN-SOURCE)

product.py

The max_pages and max_depth can be changed accordingly

```
class ProductSpider(scrapy.Spider):
   name = 'product'
   allowed_domains = ['web-scraping.dev']
   max_pages = 5
   max_depth = 30


def parse(self, response):
      if response.meta.get('depth', 0) > self.max_depth:
         return

      product_name = response.css('title::text').get()
      prod = response.url.split('/')[-1].split('.')[0].split('-')[-1]
      link = f'https://web-scraping.dev/product/{prod}'

      yield {
         'title': product_name,
         'link': link
      }
```

inverted_index.py

The tf-idf score is calculated-

```
def tf_idf_index(corpus):
   vectorizer = TfidfVectorizer()
   X = vectorizer.fit_transform(corpus).toarray()
   feature_names = vectorizer.get_feature_names_out()

   tfidf_index = {}
   for i in range(len(feature_names)):
      tfidf_index[feature_names[i]] = []
      for doc in range(len(corpus)):
         if X[doc][i] > 0:
            tfidf_index[feature_names[i]].append((doc, X[doc][i]))

   return tfidf_index
```

Cosine Similarity is calculated

```
def cos_similarity(query_vector, tfidf_index, corpus):
    scores = {}
    for i in range(len(corpus)):
        scores[i] = 0
    for term in query_vector:
        for (doc, score) in tfidf_index[term]:
            scores[doc] += score * query_vector[term]
    for doc in scores.keys():
        scores[doc] = scores[doc] / len(corpus[doc])

    return sorted(scores.items(), key=lambda x: x[1], reverse=True)
```

main.py

```
# search = spelling_correction(search)[:-1]
    query = modify_query(search, get_vocab(index))[:-1]
    vector = query_to_vector(query, index, len(corpus))
    matches = cos_similarity(vector, index, corpus)
    print(query)
    print(matches[:K])
    for i in range(K):
        results += urls[matches[i][0]] + '\n'

    return render_template('results.html', **locals())
```

query_processor.py

```
def load_index(pickle_file):
    with open(pickle_file, 'rb') as f:
        dict = pickle.load(f)
    return dict

def load_corpus_file(corpus_file):
    with open(corpus_file, 'r') as f:
        lines = f.readlines()
    return lines

def load_urls(url_file):
    with open(url_file, 'r') as f:
        lines = f.readlines()
    return lines

def get_vocab(index):
    return [term for term in index.keys()]

def spelling_correction(query):
    corrected_query = ''
    for word in query.split():
        temp = [(nltk.edit_distance(word, w),w) for w in correct_words if w[0]==word[0]]
        corrected_query += sorted(temp, key = lambda val:val[0])[0][1] + ' '
    return corrected_query

def modify_query(query, vocab):
    query_terms = query.split()
    modified_query = ''
    for term in query_terms:
        match = vocab[0]
        min_dst = nltk.edit_distance(term, match)
        for i in range(1, len(vocab)):
            dst = nltk.edit_distance(term, vocab[i])
            if  dst < min_dst:
                min_dst = dst
                match = vocab[i]
        modified_query += match + ' '

    return modified_query
```

10. BIBLIOGRAPHY

References

[1] M. T. M. Gerald J. Kowalski, "Information Storage and Retrieval Systems- Theory and Implementation," in *Information Storage and Retrieval Systems- Theory and Implementation 2nd edition*, Kluwer Academic Publishers, 2000, pp. 27-44.

[2] D. Panta, "WEB CRAWLING AND SCRAPING–developing a sale-based websit," Thesis, Turku, 2015.

[3] Zyte, "Scrapy," [Online]. Available: https://scrapy.org/.

[4] J. Duke, "How To Crawl A Web Page with Scrapy and Python 3," Digital Ocean, 6 December 2022. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-crawl-a-web-page-with-scrapy-and-python-3.

[5] [Online].

[6] P. R. H. S. Christopher D. Manning, An Introduction to Information Retrieval, Cambridge: Cambridge University Press, 2009.

[7] M. Deep, "How I Created an Inverted Index Search App Using Python and Streamlit," Medium, 15 August 2023. [Online]. Available: https://bootcamp.uxdesign.cc/how-i-created-an-inverted-index-search-engine-using-python-and-streamlit-3989a1338c48.

[8] E. Alp, "Comparison of baseline inverted index compression techniques by using a new document similarity removal method," TED University, Turkey, 2022.

[9] I. Maia, *Building Web Applications with Flask,* Packt Publishing Ltd, 2015.

[10] D. Ellis, "A behavioural approach to information retrieval system design," *Journal of documentation 45 no.3,* pp. 171-212, 1989.