Project 4: A Search Engine in the Terminal
COSC423/523, Fall 2021

Due: Nov 30, 11:59pm

1 Overview and Instructions

Modern search engines are powered by algorithms that automatically assess the relevance of documents to user-supplied queries. These search engines also often automatically recommend documents that are related to other documents that are user may finds relevant. This project requires that you implement a command-line search engine with an embedded web crawler alongside a ranking and retrieval system.

2 The Search Engine Architecture

Search engines are software systems that take text-based queries as input and provide a list of relevant documents as output. This project requires that your build a search engine that includes:

1. A `WebCrawler` class that scrapes and stores data from the UTK-affiliated websites.
2. A `SearchInterface` class that creates an interface for issuing search queries.
3. A `SearchEngine` class that instantiates the texttttWebCrawler and `SearchInterface` classes while also storing and calculating relevance using TF-IDF indexing.

3 Implementation Requirements

This project requires that you write a command-line utility that implements several path planning algorithms. The implementation requirements for this project are as follows:

1. Language and Library Requirements
   - Your implementation must be written in Python 3.6 or higher.
   - You must use BeautifulSoup4 (v4.8.0) for web crawling.
   - You must use Scikit-Learn (v0.15.0) for TF-IDF vectorization.
   - You may use standard libarries (e.g., re, string) to assist with string management.
   - You may use standard libraries (e.g., argparse) to assist with argument-reading.
2. Organizational Requirements
   - You must implement a `SearchEngine` that instantiates an instance of the Web Crawler class and instantiates an instance of the `SearchInterface` class. Both instances should be stored as class variables.

- { All three classes should be stored in their own individual files named like so: `engine.py`, `crawler.py`, and `interface.py`. All files should be stored at the same directory level. The `SearchEngine` file should import the crawler and interface files.

- You must implement a file named `main.py` that creates an instance of the `SearchEngine` class. After creating the SearchEngine object, the `main.py` file should call the class's *start*() method. The only file that should be imported in `main.py` is `engine.py`.

3. Argument Requirements

   - Your implementation should require several command-line arguments:
     - { **-root**: A string that represents a URL. Any string that begins with "http" or "https" is a valid value.
     - { **-mode**: A string that determines the search engine's mode of operation. Valid values are "C" and "I".
     - { **-query**: A string that should be used as a query to the search engine. This argument is only used *and* required in "Command-Line Mode".
     - { **-verbose**: A string that controls the verbosity of the program's output. Valid values are "T" or "F".
   - If "mode" and "root" are not provided, the program should not execute and report an error: "ERROR: Missing required arguments".
   - If "mode" and "root" are not provided, but they are invalid, the program should not execute and report an error: "ERROR: Invalid arguments provided".
   - If "mode" is set to Command-Line ("C"), the "query" argument is required. If it is not supplied as an argument, the program should not execute and report an error: "ERROR: Missing query argument".
   - If "mode" is set to Interactive ("I"), the "verbose" argument is required. If it is not supplied as an argument, the program should not execute and report an error: "ERROR: Missing verbose argument".

4. Interface Mode Requirements

   - Your implementation should be operational in two interface modes as dictated by the "-mode" argument:
     - { **Command-Line Mode:** A mode in which your program takes the supplied query from the command-line argument and returns the search results for the query. After returning the search results, the program should exit.
     - { **Interactive Mode:** A mode in which your program provides an interactive search interface for issuing queries. The search interface should be identical to the format shown in Figure 1. In addition to issuing queries, the interface should allow you to issue a series of "administrative" commands that begin with ":" in the interface:
       - ∗ **:delete** : Call's the `SearchEngine`'s 'delete' method.
       - ∗ **:train** : Call's the `SearchEngine`'s 'train' method.
       - ∗ **:exit** : Exits the program.

5. Debugging / Output Requirements

   - Your program's output should vary based on the "verbose" command line argument:
     - **Normal Mode:** A mode in which users only see the queries issued to the search system and the results from the issued query.
     - **Verbose Mode:** A mode a wealth of debugging information is provided as output in addition to the Normal Mode output.
   - When the "verbose" argument is set to True ("T"), your program should print a wealth of information in each of your components. For ease of readability, the "Verbosity Requirements" are detailed in each of the Search Engine component sections below.

6. Component #1: Web Crawler Requirements

   - The `WebCrawler` class must include a constructor that receives the "root" (i.e., a URL) and the verbose flag as parameters and stores them locally. The class must implement the following functions:

- `get_documents(self)`: Returns the list of cleaned documents.
- `set_documents(self, d)`: Sets the list of cleaned documents.
- `get_links(self)`: Returns the list of collected links.
- `set_links(self, l)`: Sets the list of collected links.
- `collect(self, s, d)`: Collects the list of links starting with site `s` and "hopping" a depth of `d`.
- `crawl(self)`: Extracts and stores all relevant text from the list of collected links.
- `clean(self)`: Modifies text extracted from webpages. Returns the cleaned documents in a list.

   - Functional requirements for collect, crawl, and clean methods:
     - **collect()**: The Collect method should find and store any link (i.e. the "href" attribute of <a> elements) that includes in "utk.edu". You should not collect the same link twice. The method should "hop" `d` times per the parameter supplied in the constructor. A depth of 0 indicates that your crawler should stop collecting *after* it has collected links from your root site. In contrast, a depth of 1 indicates that your crawler you stop collecting after it has collected links from all of the links it initially collected on your root site.
       - Note: "https://eecs.utk.edu" and "http://eecs.utk.edu/" are identical.
     - **crawl()**: The Crawl method should extract all text from two element types:
       - <p> elements inside elements that have the "entry-content" or "person_content" as their class attribute.
       - <table> elements that have the "table_default" as their class attribute.
     - **clean()**: Makes the following modifications to all extracted text:
       - Remove all Unicode characters.
       - Remove all punctuation (i.e., quotes, commas, !, ?, etc.)
       - Remove all Twitter handle mentions (i.e., "@UTK_EECS" should be deleted.)
       - Remove all instances of double-spaces.
       - Convert all characters to lowercase.

Figure 1: A screenshot of "Interactive Mode". After showing the SearchEngine header, the program prints a ">" and waits for user input. After entering a query, the system returns relevant results.

- The following statements should be printed at the appropriate time when "verbosity" is set to True ("T"):
  - 1. **collect()**: [VERBOSE] 1. COLLECTING LINKS - STARTED
  - 2. **collect()**: [VERBOSE] COLLECTED: LINK (#)
  - 3. **collect()**: [VERBOSE] 1. COLLECTING LINKS - DONE
  - 4. **crawl()**: [VERBOSE] 2. CRAWLING LINKS - STARTED
  - 5. **crawl()**: [VERBOSE] CRAWLING: LINK (#/N)
  - 6. **crawl()**: [VERBOSE] 2. CRAWLING LINKS - DONE
  - 7. **clean()**: [VERBOSE] 3. CLEANING TEXT - STARTED
  - 8. **clean()**: [VERBOSE] 3. CLEANING TEXT - DONE

  Note: "#" refers to enumeration of the encountered link. For the first collected link, the program would use "1" in place of the #. "N" refers to the total number of links traversed. It is expected that your program would produce a print-statement for each link collected (2) and each link crawled (5).

7. Component #2: SearchInterface Requirements

- The `SearchInterface` class must include a constructor that receives three parameters: (1) "mode", which refers to the type of interaction mode; (2) an "engine", which is a reference to the SearchEngine component that houses an instance of the `SearchInterface`; and (3) a "query" parameter that should only be used during "Command-Line Mode". All parameters should be saved as class variables.

  The class must implement the following functions:

  - `listen(self)`: This method's functionality is based on the specified mode:
    - If "Command-Line Mode" is active and a query was supplied, the program should send the query to the parental `SearchEngine` component's "handle_query" function and print the results to the terminal.
    - If "Interactive Mode" is active, the program should print the search engine header shown and then begin interactively accepting queries. The interface should facilitate an input loop in which a query can be supplied and related results are printed in the terminal. When a query is received, the query should be sent to the "handle_input" method below. The interface should stop sending input to this method when ":exit" is provided.

- { `handle_input(self)`: This method should facilitate the routing of queries to functions in the search engine. Specifically, this method routes administrative commands (e.g. ":train" and ":delete"). When a query is not recognized as an administrative command, it should be sent to the parental `SearchEngine` component's "handle_query" function.

8. Component #3: SearchEngine Requirements

   - The `SearchEngine` class must include a constructor that receives five parameters: (1) "mode", which controls the interaction mode; (2) an "verbosity" string, which controls verbosity; (3) a "query" parameter that should only be used during "Command-Line Mode"; (4) a "root" parameter that serves as the URL root for crawling; and (5) a "depth" parameter that controls crawling depth. All parameters should be saved as class variables.

     The constructor should also instantiate an instance of the `WebCrawler` and an instance of the `SearchInterface` classes. It should save both instances as class variables. After doing so, the constructor should call the `SearchEngine` class's `train()` method.

     The class must implement the following functions:

     - { `train(self)`: This method should call the collect, crawl, and clean methods using the `WebCrawler` instance created in the `SearchEngine`'s constructor. After these methods have completed, this method should save the cleaned documents to a file named "docs.pickle" and the crawled links to "links.pickle". If these files already exist, this method should ***not*** run the collect, crawl, and clean methods and should instead load both the cleaned documents and crawled links from their appropriate files. In both cases, this method should conclude by calling the class's `compute_tf_idf()` method.
     - { `delete(self)`: This method should delete any .pickle files created from the class's `train()` method.
     - { `compute_tf_idf(self)`: This method should read the cleaned documents from the instance of the `WebCrawler` class created in the constructor and vectorize the documents using Scikit-Learn's TfidfVectorizor. The function should return a Pandas dataframe that sets the TFIDF vocabulary as the index.
     - { `handle_query(self)`: This method should vectorize the query, calculate cosine similarity to the texts extracted from the web, and print a list of up to five documents relevant to the query. If no relevant documents are found, the method should print "Your search did not match any documents. Try again.". For each relevant document found, the method should print the document number, the URL, and the similarity score. For example: `[1] http://web.eecs.utk.edu/acw (0.22)`
     - { `listen(self)`: This method should call the listen() method of the `SearchInterface` created in the constructor.

   Note that the visual presentation of your query and the results returned from your search engine are important. Your program's search interface should be identical to the format shown in Figure 1.

9. Your code should be adequately documented. Each function should have a supporting docstring followed the PEP 257 convention. The top of `main.py` should include your name as an author and provide a brief description of your source code.

10. You should include a "README" file that provides an overview of your code. You should provide instructions for running your code locally.

4 Grading and Submission

The assignment is worth 100 points. Create a ZIP file that includes all of your Project 4 files. Upload the ZIP file to the Project 4 submission folder on Canvas by the due date. For questions regarding the late policy for assignment submission, please consult the syllabus.

4.1 Grading Scheme

The following grading scheme will be used for undergraduate students:

Requirement	**Point Value**
Documentation: README	10 points
Documentation: Code Documentation	10 points
Requirements: Organization	10 points
Requirements: Arguments	10 points
Requirements: Interface Modes	10 points
Requirements: Debugging Output	10 points
Implementation: SearchEngine	15 points
Implementation: WebCrawler	15 points
Implementation: SearchInterface	10 points
Total	100

5 Commentary on Rate-Limiting

Rate-Limiting is a defensive web technique in which websites automatically limit the number of HTTP requests that can be made to them. Often times, web administrators (e.g., EECS IT Staff) will configure automatic rate-limiting to prevent the website from succumbing to an overwhelming number of HTTP requests.

In our assignment, it is possible that you may encounter rate-limiting. BeautifulSoup4 will likely produce an error if you're rate-limited. Alternatively, your crawler may start crawling very slowly. Regardless, you should make an effort to **not** get yourself rate-limited. Try not to make an unnecessary number of requests (e.g., by allowing your crawler to run endlessly).

If you believe you've been rate-limited or have issues with accessing the EECS UTK website, please reach out to Dr. Williams via email.