Search Engine System Project Report

CPS847 F2025 - Abdur-Rehman Rizvi

This project implements a web search engine that combines text-based relevance ranking (BM25) with link-based authority ranking (PageRank). The system crawls approximately 500 web pages from Python-related documentation sites and provides a web interface for searching the indexed content.

The system consists of five independent Python files, an HTML file for the web interface, and a text file for the stopwords borrowed from assignment 1:

cps842f25_prj_Rizvi/

crawler.py - Downloads and stores web pages

indexer.py - Builds inverted index from crawled content

pagerank.py - Computes page authority scores

searchengine.py - Processes queries, computes BM25 and serves results

webapp.py - Web interface for search engine

stopword.txt - List of 31 stopwords

Templates/

Index.html - For the Web interface

Data/ (will contain outputs from the Python files)

doc_info.json, index.json, pagerank.json, pages.json, and stats.json

Web Crawler (crawler.py)

Downloads HTML web pages while respecting crawling ethics such as robots.txt and a 2 second delay between requests to the same domain. Has a starting seed set of three links with many informative outlinks about the Python programming language. As it's crawling, it outputs the current link and overall total of links crawled so far. This Python script outputs a JSON file containing doc_id, url, title, text, and outlinks for each page.

Functions:

- **get_robot_parser(url)** - Checks robots.txt for crawling permissions
- **can_fetch(url)** - Validates if URL can be crawled
- **respect_politeness(url)** - Enforces a 2-second delay between requests to same domain
- **fetch_page(url)** - Downloads HTML and ignores non-HTML content (PDFs, images)

- **extract_links(html, base_url)** - Parses HTML and extracts all links
- **extract_text(html)** - Extracts title and body text from HTML
- **crawl()** - Crawl pages, calls all other functions

Indexing (indexer.py)

Processes documents to build an inverted index for efficient search. It loads the crawled pages from data/pages.json, gives title 3x weight than body text as they're more important to user in search results, tokenizes and counts term frequencies, then builds then inverted index of structure: {term: {doc_id: frequency}}. It also calculates document lengths and statistics.

Functions:

- **tokenize(text)** - Converts text to lowercase, extracts alphanumeric tokens and removes stopwords
- **build_index()** - Creates inverted index mapping terms to documents

Outputs:

- **data/index.json** - Inverted index
- **data/doc_info.json** - Document data (url, title, length)
- **data/stats.json** - Collection statistics (total docs, avg length, vocab size)

PageRank (pagerank.py)

Calculates PageRank score for each of the 500 pages based on link structure. Only has one function **calculate()** which builds a directed graph using the NetworkX library where each crawled document is a node and hyperlinks between pages are edges. It then calls NetworkX's built-in `pagerank()` function which implements Google's PageRank algorithm using power iteration with a damping factor of 0.85, iterating until convergence to calculate importance scores based on the link structure. Finally, the raw PageRank scores are normalized to a [0,1] range using min-max normalization before being saved.

Outputs:

- **data/pagerank.json** with normalized authority scores for each page

Search Engine (searchengine.py)

The search engine begins by loading the inverted index, document's data, and PageRank values, then builds a BM25 corpus by reconstructing each document as a list of tokens repeated according to their term frequencies. When a query arrives, it is first tokenized into lowercase terms. The engine then gathers all documents that contain at least one of these terms, reducing the scoring work to a smaller candidate set. BM25Okapi is used to compute relevance scores, which are then normalized to a 0–1 range so they can be combined with PageRank values. Each candidate document receives a final score calculated as a weighted blend of normalized BM25 and PageRank, with 0.7 for BM25 and 0.3 for PageRank. The results are sorted by this combined score and the top K documents are returned along with the total time taken to process the query. K is set in **webapp.py**. This script only has a class that is used by **webapp.py**, so running this file by itself won't do anything.

Functions:

- **tokenize(text) -** Processes query into tokens
- **search(query, top_k) -** Main search function

Output: List of results with doc_id, url, title, combined score, BM25 score, and PageRank score. Used by webapp.py.

Web Application (webapp.py)

This is the web interface for the search engine using Flask. It uses **searchengine.py**, and returns top 20 results to the user. The HTML page templates/index.html provides the interface where the user types a query, and its JavaScript function sends that query back to webapp.py backend using a request like `/search?q=...`. Flask receives the query, passes it to the `SearchEngine` class to compute BM25 and PageRank–based scores, and then returns the results as JSON. The JavaScript code in index.html then takes that JSON, formats the results, and displays them on the page, creating the full loop between user input, backend processing, and frontend output.

How to Run

First run **crawler.py** which will crawl 500 pages. This will last for up to 10 minutes. Then run **indexer.py** which will build the inverted index in a few seconds. Finally, run **pagerank.py** to calculate the PageRank score for each page. At this point the search engine is ready to be used, and can run the search engine in the web interface by running **webapp.py**.

Libraries required:

- BeautifulSoup4 (for HTML parsing)
- NetworkX (for PageRank calculation)
- rank_bm25 (for BM25 scoring)
- Flask (for web interface)