IS-322 _Information Retrieval

Faculty of Computers and Artificial Intelligence

Cairo University

Report

Assignment #2

ID	Name	Group
20230609	Duaa Abd-Elati Abd-Elazeem	All
20220089	Tasneem gomaa anter abdl maged	All
20221066	Ziad Tawfik ZiadTawfik	All
20221128	Mohamed Awad Sayed Hasan	S1
20190769	Fatemah Atef Mahmoud	All
20210442	Heba Azouz Seif Ibrahim	All

1. Overview

This document provides a concise overview of the Java-based web crawler and information retrieval system you have implemented. It covers four main areas:

1. Approach to Designing the Crawler
2. Construction of the Inverted Index
3. Computation of TF–IDF and Cosine Similarity
4. Challenges Encountered and Solutions

---

2. Approach to Designing the Crawler

The WebCrawler class uses **Jsoup** for HTML fetching and parsing. The key design points are:

- **Depth-First Recursion with Limits**:
  A recursive crawl(String url, int depth) method tracks both the remaining depth and a global page count. Crawling stops when either the maximum depth is reached or the configured maxPages limit is met.
- **URL Normalization & Deduplication**:
  URLs are stored in a HashSet<String> visited to avoid revisiting the same page. Each URL is normalized by removing the protocol and replacing non-alphanumeric characters when naming the output file.
- **Storage of Page Content**:
  Crawled pages are stored as text files under a pages directory. Each file begins with the URL followed by the page's raw body text, obtained via doc.body().text().
- **Link Extraction**:
  After saving the current page, the crawler selects all <a href> links, resolves them to absolute URLs via l.absUrl("href"), and recurses.

This design balances simplicity and robustness, allowing easy extension (e.g., politeness delays, robots.txt compliance) if needed.

---

3. Building the Inverted Index

The InvertedIndex class encapsulates all logic for building and querying a term-based index:

1. **Directory Traversal**:
   buildInvertedIndex(String folderPath) scans the pages directory for .txt files.

2. **Content Reading & Tokenization**:
    - readContent(File file) reads entire files into a single string.
    - TextProcessing.processText(String text):
        - Removes Wikipedia-style citations (e.g., [1]).
        - Normalizes tokens by lowercasing and stripping special characters.
        - Filters out stop words, purely numeric tokens, and tokens shorter than three characters.
        - Porter stemming (Stemmer class) Applies to reduce each token to its root form.
3. **Term Frequency Counting**:
   For each document, a Map<String, Integer> TF collects the frequency of each stemmed token.
4. **Posting List Construction**:
   For each term, a Posting object (docId, TF) is created and added to the InvertedIndex under that term's entry.

This inverted index structure enables efficient retrieval of all documents containing a given term, along with term frequency information needed for ranking.

---

4. TF–IDF and Cosine Similarity Computation

The TFIDFCalculator class provides methods to weight terms and rank documents.

1. **Document Vector Computation (computeDocumentVector)**:
    - **Term Frequency Weighting**:
      For each term t in document d, compute:
      tft,d=1+log⁡10(TF)tf_{t,d} = 1 + \log_{10}(TF)tft,d=1+log10(TF)
    - **Inverse Document Frequency**:
      With NNN total documents and document frequency dftdf_tdft (# of postings for term ttt), compute:
      idft=log⁡10(N/dft)idf_t = \log_{10}(N/df_t)idft=log10(N/dft)
    - **TF–IDF Weight**:
      Multiply to get weight:
      wt,d=tft,d×idftw_{t,d} = tf_{t,d} \times idf_twt,d=tft,d×idft
2. **Query Vector Computation (computeQueryVector)**:
    - Tokenizes the input query, computes TF for each query term, and applies the same weighting formula.

3. **Cosine Similarity (cosineSimilarity)**:
    - Given two vectors (document v and query q)

      Terms not present default to zero weight.
      This normalized dot product ranks how closely a document matches the query.
4. **Ranking (rank Documents)**:
    - Computes the similarity score for each document, sorts in descending order, and returns the top 10 document IDs.

These computations enable a classic vector-space retrieval model using TF–IDF weighting.

---

5. Challenges and Solutions

1. **Handling HTML Noise**:
   Early tests showed that raw text extraction included navigational and boilerplate text.
   Solution: Focused on doc.body().text() and relied on stop words + stemming to minimize noise impact.
2. **Term Normalization**:
   Documents contained punctuation, citations, and mixed casing.
   Solution: A dedicated TextProcessing pipeline with regex filters, lowercasing, and a Porter stemmer ensured consistent token forms.
3. **Recursion Depth vs. Page Limit**:
   Without careful checks, crawling could exceed desired limits or recurse infinitely.
   Solution: Combined both the depth parameter and a global currentCount against maxPages, with immediate returns if exceeded.
4. **Efficient Similarity Calculation**:
   Computing all pairwise similarities naively can be slow for many documents.
   Solution: Our implementation processes each document on demand; a production system might precompute norms or use sparse vector structures.

---

6. User Manual

6.1 Prerequisites

- Java 8 or higher
- Jsoup library added to the classpath

6.2 Setup

1. Compile all .java files under com.example and invertedIndex packages.
2. Create an empty folder named pages in the working directory.

6.3 Running the Crawler

Command to run the crawler:

java com.example.WebCrawler <start-URL> <max-depth> <max-pages>

This command will crawl starting from https://example.com, to a maximum depth of 2 levels and 100 pages total.

6.4 Building the Index

After crawling, build the inverted index with:

java com.example.InvertedIndex pages

6.5 Querying the Index

To search using a query:

java com.example.MainQueryApp "your search terms here"

6.6 Example Main Program

To run everything (crawling, indexing, searching) automatically:

java com.example.Main

It will crawl, build the index, compute TF–IDF for documents and queries, and rank the top 10 documents for a given query.