Cairo University

Faculty of Computers and Artificial Intelligence

Cairo University, Faculty of Computers and Artificial Intelligence

IS322-Information Retrieval Course

Java Web Crawler with TF-IDF and Cosine Similarity

Name	ID
Israa Mohamed El-Sayed	20221252
Joseph Sameh Fouad	20220099
Youssef Joseph Adib	20220389
Jonathan Mokhles William	20220100
Amany Mohamed Hussein	20221026
Salma Mohamed Sarhan	20220152

Project overview:

Our project is a simple search engine for wikipedia pages

Main steps:

1)Crawl: visit web pages starting from two wikipedia links.

2)Build an inverted index: collect and organize the words from the pages.

3)User query: the user enters some search words.

4)Tf-idf and cosine similarity:we calculate how similar each page is to the query.

5)Ranking: show the top 10 most relevant pages based on similarity scores.

Web Crawler Module Explanation: by Israa Mohamed & Amany Mohamed

- Manages crawling of Wikipedia pages.
- Tracks:
  - ❖ visited set to avoid duplicate URLs.
  - ❖ docCounter to count the number of documents crawled.
- crawl() method:
  - ❖ Takes seedUrl (starting URL) and maxPages (maximum pages to crawl, set to 10).
  - ❖ Initializes a queue starting with the seed URL.
  - ❖ Skips URLs that:
    - Are already visited.
    - Do not start with "https://en.wikipedia.org/wiki/".
  - ❖ Fetches each valid URL using Jsoup.connect(currentUrl).get().
  - ❖ Extracts:
    - Page title using doc.title().
    - Paragraphs using doc.select("p").
  - ❖ Cleans text by:
    - Removing citations like [1], [note 3] using regex.
    - Skipping empty paragraphs.
    - Appending cleaned text to a StringBuilder.

Saving and Expanding the Crawl:

- Saves crawled content into a **Documents** folder:

- Each page saved as a separate text file (e.g., doc1.txt).
- File includes URL, page title, and cleaned paragraph text.
- Updates counters:
  - Increments docCounter and localSaved after each page.
- Extracts all hyperlinks (a:href) from the page.
- Adds only valid links to the queue:
  - Must belong to English Wikipedia domain.
  - Should contain important keywords (like Ancient, Akhenaten, etc.).
  - Must not already exist in the visited set or the queue.
- Continues crawling until 10 pages are successfully saved per seed.

Inverted Index Class Explanation: by Youssef Joseph & Joseph Sameh

Purpose of Inverted Index:

- Builds an inverted index from a collection of documents (text files or list of strings).
- Inverted Index maps each term (word) to a list of postings (document ID + term frequency).
- Used in search engines and information retrieval systems for fast keyword lookups.

Key Components:

- Map<String, List<Posting>> index -------> Stores the inverted index. Key = term, Value = list of postings
- Normalizer normalizer -------> Preprocesses the text: tokenization, lemmatization

Main Methods:

1- buildIndex(String folderPath)

- Reads all `.txt` files from a folder.
- Skips metadata lines (like "URL:" and "Title:").
- Normalizes and tokenizes the content.
- Builds term frequencies and updates the index.

2- buildIndex(List<String> documents)

- Builds the index directly from a list of in-memory document strings.
- Useful when documents are not in files.

3- Helper Methods

- **`extractFileNumber(String name)`:** Extracts numbers from filenames for proper sorting.

- `printIndex()`: Prints the inverted index nicely formatted.
- `getDocumentCount()`: Returns how many unique documents are indexed.
- `getTerms()`: Returns the set of all terms.
- `getDocumentFrequency(String term)`: Returns how many documents contain a given term.

- `getTermFrequencies(int docId)`: Returns term frequencies for a specific document.
- `getAllDocumentIds()`: Returns the set of all document IDs.

How the Index is Built:

1-Read documents → ignore metadata.

2- Normalize text → lowercasing, cleaning, lemmatization.

3- Tokenize text into words.

4-Count term frequency (TF) for each document.

5-Update index: For each term, add or update its posting list.

TF-IDF Calculation & Ranking Explanation: by Jonathan Mokhles & Salma Mohamed

TF-IDF Calculator:

Fields

- **`invertedIndex`: A map storing terms and their associated postings (document occurrences)**
- **`totalDocuments`: The total number of documents in the corpus**
- **`documentVectors`: Stores the final TF-IDF vectors for each term-document pair**

Main Methods:

`compute()`

Calculates TF-IDF scores for all terms across all documents by:

1. **Iterating through each term in the inverted index**
2. **Calculating IDF for the term**
3. **For each document containing the term:**
    - **Calculating term frequency (TF) weight**
    - **Computing TF-IDF score (TF × IDF)**
    - **Storing the result in `documentVectors`**

`calculateIDF(String term):`

Computes Inverse Document Frequency using:

`IDF = log10(totalDocuments / documentFrequency).`

`getTFIDF(String term, int docId):`

Retrieves the precomputed TF-IDF score for a specific term-document pair.

`printIndex():`

Prints the TF-IDF index in a human-readable format.

Supporting Class: `DocumentTFIDF:`

A simple data structure that pairs:

- **`docId`: Document identifier**
- **`tfidf`: Computed TF-IDF score for a term in this document**

How Ranking Works:

1. Preprocessing:

- Query terms are processed (tokenized, maybe lemmatized).
- Document terms have already been processed and indexed with their TF-IDF scores.
- You fetch a list of documents and their tf-idf vectors for terms matching the query.

2. Build Document Vectors:

- You **group all tf-idf weights by document**.
- Each document is represented as a **vector**:
  `{term → tf-idf weight}`

3. Normalize Document Vectors:

- Each document vector is normalized to unit length:
  normalized weight=tf-idf weight / vector length.

- normalizeVector(docVector) is called for every document.

- Vector length = square root of the sum of squares of tf-idf weights.

4. Build Query Vector

- For the query:

    - For each **term**, get its **idf** (from corpus).
    - Compute **query weight** = tf × idf (usually tf = 1).
    - Where tf = frequency of the term in query.

- Then normalize the query vector **the same way** (divide by its vector length).

5. Compute Cosine Similarity:

- For each document:

    - Compute the **dot product** between the normalized **query vector** and the normalized **document vector**:
      cosine similarity = ∑ terms in query(query weight × document weight).
    - for all terms in the query => compute query weight × document weight.
- This gives a score for how similar the document is to the query.

6. Ranking:

- After computing scores for all documents:

    - Sort the documents by their **similarity scores in descending order**.

    - Return top-ranked documents as the **final ranking**.

Code Components Mapping:

Main Methods	Purpose
`Ranker.java/ buildNormalizedDocumentVectors()`	Creates and normalizes document vectors

Ranker.java/ normalizeVector(Map<String, Double>)	Divides each vector(query or document) component by the vector length
CosineSimilarity.java/ calculate(queryVec, docVec)	Computes cosine similarity via dot product
Ranker/ rankAndDisplayTopDocuments() method	Calls all the above, sorts the scores, returns ranked results

Simple Diagram:

Query Terms → build Query Vector → normalize

↓

Document TFIDFs → build Doc Vectors → normalize

↓

Compute dot products → Cosine Similarities

↓

Sort by score → Ranked documents