Simple Search Engine

Instructor

Prof. Syed Khaldoon Khurshid

Submitted by

M. Shahzaib Ijaz 2021-CS-75

Department of Computer Science
University of Engineering and Technology
Lahore Pakistan

Contents

1 Introduction

The exponential growth of digital information has led to a significant increase in the need for effective search engines. Researchers, educators, and professionals often require access to relevant information from vast repositories of documents. This project aims to address this need by developing a search engine tailored for research articles. By leveraging the TF-IDF ranking algorithm, the search engine prioritizes the relevance of search results, making it easier for users to find the information they need.

2 Objective

The objective of this project is to create a search engine capable of indexing research articles and providing users with a robust search experience. Users should be able to:

- Search for specific words within documents.
- Find documents related to specific topics.
- Receive keyword suggestions based on indexed data.

Additionally, the search engine should rank results using the **TF-IDF (Term Frequency-Inverse Document Frequency)** model for better relevance.

3 TF-IDF

TF-IDF is a numerical statistic used to evaluate the importance of a word in a document relative to a collection of documents (corpus). It is widely used in information retrieval and text mining.

3.1 Term Frequency (TF)

Definition: The frequency of a term in a document. It is calculated as:

$$TF = \frac{f_t}{T} \tag{1}$$

where:

- f_t = Number of times the term appears in the document.
- T = Total number of terms in the document.

3.2 Inverse Document Frequency (IDF)

Definition: A measure of how unique or rare a term is across all documents in the corpus. It is calculated as:

$$IDF = \log \frac{N}{1 + n_t} \tag{2}$$

where:

- N = Total number of documents in the corpus.
- n_t = Number of documents containing the term.

3.3 TF-IDF Score

Formula: The product of TF and IDF gives the TF-IDF score:

$$\text{TF-IDF} = TF \times IDF \tag{3}$$

A higher TF-IDF score indicates that the term is more relevant to the specific document in the context of the corpus.

4 Key Features

- **Keyword Suggestions**
  - Detects typos in user queries and provides corrected keyword suggestions.
  - Enhances user experience and ensures accurate searches.
- **Result Details**
  - Displays the following for each query:
    - **Title** of the document.
    - **Author Name(s)**.
    - **Snippet** of the document's context.
- **Filters**
  - Allows users to refine their search by applying filters:
    - **Search by Title**: Matches query against document titles.
    - **Search by Author**: Filters results based on author names.
    - **Search by Context (Default)**: Searches within the document's full content.
- **TF-IDF Ranking**
  - Ranks documents based on **Term Frequency-Inverse Document Frequency (TF-IDF)**.
  - Ensures relevant documents appear higher in search results.

5 Implementation

5.1 Document Handling

5.1.1 extract_text_from_documents function

1. Reads a .docx file using the Document class from the python-docx library.
2. Extracts key sections:
   (a) **Title:** The first non-empty paragraph is assumed to be the document's title.
   (b) **Author:** A paragraph starting with "Author:" is processed to retrieve the author's name.
   (c) **Content:** All other text is stored as the main content.
3. Returns a structured dictionary with title, author, and content fields.

5.1.2 document_extractor function

1. Iterates through all files in a specified folder (self.folder_path).
2. Filters for valid .docx files.
3. Calls extract_text_from_documents to extract data from each file. Adds the file path for traceability and appends the structured data to a list.
4. Returns a collection of extracted data for all .docx files in the folder.

app.py

```python
# Function to extract doc_data from specific docx file
    def extract_text_from_documents(self, file_path):
        document = Document(file_path)
        doc_data = {'title': "", 'author': "", "content": ""}

        paragraphs = (p.text.strip() for p in document.paragraphs if p.text.strip())
        for text in paragraphs:
            if not doc_data['title']:
                doc_data['title'] = text
            elif text.startswith("Author:"):
                doc_data['author'] = text[7:].strip()
            else:
                doc_data['content'] += f"{text}"

        return doc_data

    # Function to extract all content from the Document folder
    def document_extractor(self):
        documents = []

        for file_name in os.listdir(self.folder_path):
            file_path = os.path.join(self.folder_path, file_name)

            if not (os.path.isfile(file_path) and file_name.endswith('.docx')):
                continue

            doc_data = self.extract_text_from_documents(file_path)
            doc_data['file_name'] = file_path

            documents.append(doc_data)

        return documents
```

Figure 1: Document Handling In Search Engine.

5.2 Text Preprocessing

5.2.1 stem function

1. A list of common suffixes (suffixes) is defined.
2. Iterates over the suffixes and checks if the word ends with one of them using word.endswith(suffix).
3. If a match is found, it removes the suffix by slicing the string up to -len(suffix) and returns the modified word.
4. If no suffix matches, the original word is returned unchanged.

5.2.2 preprocess_text function

1. Converts the text to lowercase using .lower().
2. Removes punctuation with .translate(translator).
3. Splits the text into individual words using .split().
4. Filters out stopwords and applies the stem function to each remaining word.
5. Returns the processed words as a list.

Figure 2: Text Preprocessing In Search Engine.

5.3 Indexing

1. index:
   (a) Uses a defaultdict to initialize each term with:
      i. An empty list for document IDs (doc_ids).
      ii. A nested dictionary (tf) for storing term frequencies by document.
2. doc_lengths:
   (a) Stores the total number of words in each document.
3. search_terms:
   (a) Tracks the occurrence of terms and phrases for suggestion purposes.
4. Iterates through each document and assigns it a unique doc_id (its position in the list).
5. For each word in the preprocessed document:
   (a) Updates the term frequency (tf) for the current doc_id.
   (b) Adds the term to doc_ids in the index.
6. For each word in the document, the function also:
   (a) Updates search_terms for single words.
   (b) Constructs phrases of up to 5 consecutive words.
   (c) These phrases are added to search_terms to track their frequency.

5.4 TF-IDF Calculation

5.4.1 compute_idf function

- `index`: The inverted index which contains term-document mappings, including the list of document IDs that contain each term.
- `num_documents`: The total number of documents in the collection.

IDF Formula: The IDF for each term is calculated using the following formula:

$$\text{IDF}(t) = \log\left(\frac{\text{Total Documents}}{1 + \text{Number of Documents Containing the Term}}\right)$$

The 1 in the denominator ensures that we don't divide by zero if a term appears in every document.

Figure 3: Indexing In Search Engine.

5.4.2 compute_tf_idf function

- `index`: The inverted index that contains the term frequencies (TF) for each term in every document.
- `idf`: A dictionary of precomputed IDF values for each term.
- `doc_lengths`: A dictionary where the key is the document ID, and the value is the length (word count) of that document.

TF-IDF Formula: The TF-IDF score for each term in a document is computed using the following formula:

$$\text{TF-IDF}(t,d) = \left(\frac{\text{Term Frequency (TF)}(t,d)}{\text{Document Length}(d)}\right) \times \text{IDF}(t)$$

Where:

- $\text{TF}(t,d)$ is the frequency of term t in document d.
- $\text{Document Length}(d)$ is the total number of terms in document d.
- $\text{IDF}(t)$ is the inverse document frequency for the term t, precomputed by the `compute_idf` function.

Figure 4: TF-IDF In Search Engine.

5.5 Search Query

- **Input Parameters**:
  - `query`: The search query entered by the user.
  - `tf_idf`: A dictionary containing TF-IDF scores for terms in each document.
  - `idf`: A dictionary containing IDF scores for each term.
  - `doc_lengths`: A dictionary with the number of words in each document.
  - `documents`: A list of documents, each containing a title, author, and content.
- **Preprocessing the Query**:
  The query is cleaned by removing stopwords (common words like "the", "and") and reducing words to their base form (like "questioning" to "question").

```
query_keywords = self.preprocess_text(query)
```

- **Calculating Query TF-IDF Scores**:
  The TF-IDF score for each keyword in the query is fetched from the IDF dictionary. If the word isn't in the IDF dictionary, its score is set to 0.

```
query_tf_idf = {keyword: idf.get(keyword, 0) for keyword in query_keywords}
```

- **Matching the Query with Documents**:
  A `matched_docs` dictionary is created to keep track of the relevance score of each document. For each word in the query, the function checks if it exists in any document and updates the document's score based on the query's TF-IDF score and the document's TF-IDF score.

```
for word, query_score in query_tf_idf.items():
    for term in tf_idf:
        if word in term.lower():
            for doc_id, doc_score in tf_idf[term].items():
                matched_docs[doc_id] += query_score * doc_score
```

- **Ranking the Documents**:
  The documents are sorted by their relevance score, so the most relevant documents appear first.

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

- **Returning the Final Result**:
  The function returns a list of documents, ranked by how relevant they are to the user's query.

```
return result
```

Figure 5: Searching In Search Engine.

5.6 Keyword Suggestions

1. Converts the user's input to lowercase: This makes the search case-insensitive, so the function works regardless of whether the user types in uppercase or lowercase.
2. Finds terms that start with the user's input: It checks all available search terms and selects the ones that begin with the text the user has entered so far.
3. Sorts the matching terms by popularity: The terms are ranked based on how often they have been searched or used, with the most popular ones appearing first.
4. Limits the suggestions: It only shows a fixed number of the top matches, making the suggestions manageable and easy to read.

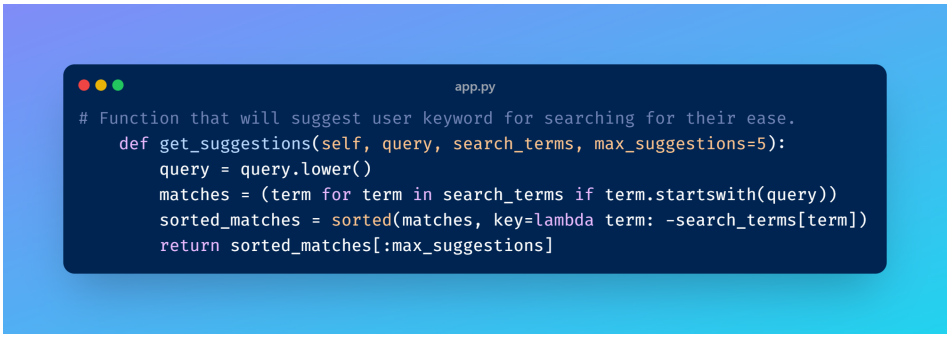

Figure 6: Keyword In Search Engine.

6 API Endpoints

6.1 `/suggestions`

- **Method**: `GET`
- **Description**: Provides keyword suggestions for the input query.
- **Parameters**: `query` (string)

- **Response**:

```
["keyword1", "keyword2", "keyword3"]
```

6.2 /api/v1/search/fulltext

- **Method**: `POST`
- **Description**: Searches the full text of documents.
- **Parameters**: `query` (string)
- **Response**:

```
[
    {
        "title": "Title of Document",
        "author": "Author Name",
        "snippet": "Snippet of content...",
        "file_path": "path/to/document.docx"
    }
]
```

6.3 /api/v1/search/title

- **Method**: `POST`
- **Description**: Searches for documents by title.
- **Parameters**: `query` (string)
- **Response**:

```
[
    {
        "title": "Title of Document",
        "author": "Author Name",
        "snippet": "Snippet of content...",
        "file_path": "path/to/document.docx"
    }
]
```

6.4 /api/v1/search/author

- **Method**: `POST`
- **Description**: Searches for documents by author name.
- **Parameters**: `query` (string)
- **Response**:

```
[
    {
        "title": "Title of Document",
        "author": "Author Name",
        "snippet": "Snippet of content...",
```

```
            "file_path": "path/to/document.docx"
        }
    ]
```

7 Data Flow Diagram

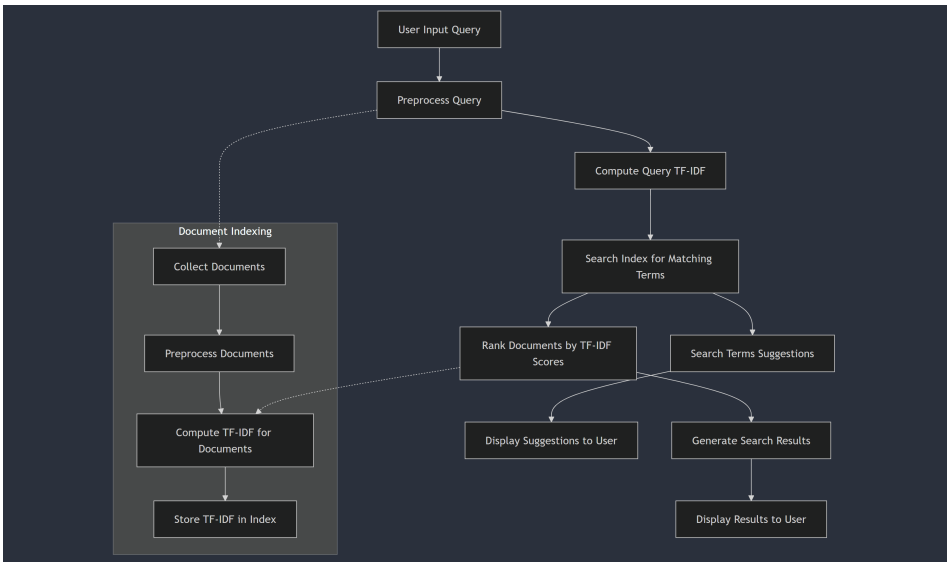

Figure 7: Data Flow Diagram.

8 Results

- The search engine successfully indexes and ranks research articles.
- Keyword suggestions enhance the user experience by guiding query formulation.
- API endpoints provide flexible search options for full text, titles, and authors.

9 Future Enhancements

1. Implement cosine similarity for ranking to improve relevance.
2. Add support for additional file formats, such as PDF.
3. Introduce advanced filtering options for search results (e.g., publication date).
4. Optimize performance for larger datasets.