Implement Search Engine with Ranking

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 **Cosine Similarity** model for better relevance.

3 Cosine Similarity

Cosine Similarity is a metric used to measure the similarity between two documents based on their vector representations. It calculates the cosine of the angle between two vectors, where each vector represents a document in a multi-dimensional space, with each dimension corresponding to a term in the document collection.

3.1 Cosine Similarity Formula

Definition: Cosine similarity is computed using the following formula:

$$\text{cosine similarity} = \frac{\mathbf{Q} \cdot \mathbf{D}}{||\mathbf{Q}||, ||\mathbf{D}||} \tag{1}$$

where:

- $\mathbf{Q}$ = Query vector, representing the user's search query.
- $\mathbf{D}$ = Document vector, representing a document in the collection.
- $\mathbf{Q} \cdot \mathbf{D}$ = Dot product of the query and document vectors.
- $||\mathbf{Q}||$ = Magnitude (norm) of the query vector.
- $||\mathbf{D}||$ = Magnitude (norm) of the document vector.

3.2 Cosine Similarity Calculation

Explanation: Cosine similarity measures the cosine of the angle between two vectors, where:
If the vectors are pointing in the same direction (i.e., the angle between them is 0 degrees), the cosine similarity is 1, meaning the vectors (documents) are highly similar. If the vectors are orthogonal (i.e., the angle between them is 90 degrees), the cosine similarity is 0, indicating no similarity between the documents. If the vectors are pointing in opposite directions (i.e., the angle between them is 180 degrees), the cosine similarity is -1, indicating complete dissimilarity. The formula normalizes the vector lengths, so the comparison focuses purely on the directionality of the vectors, not their magnitude.

3.3 Ranking with Cosine Similarity

Procedure:

- Each document and query is represented as a vector in a multi-dimensional space where each dimension corresponds to a term in the document collection.
- Cosine similarity is calculated between the query vector and each document vector.
- The documents are ranked based on their cosine similarity score, with the most similar documents appearing at the top of the results.

3.4 Example

Example: Suppose a user submits a query ”machine learning” to a search engine. The documents in the corpus are represented as vectors, and their cosine similarity with the query vector is calculated. Documents that contain terms such as ”artificial intelligence,” ”deep learning,” or ”neural networks” will still be ranked highly due to their semantic similarity to the query, even if they don’t contain the exact phrase ”machine learning.”

4 Key Features

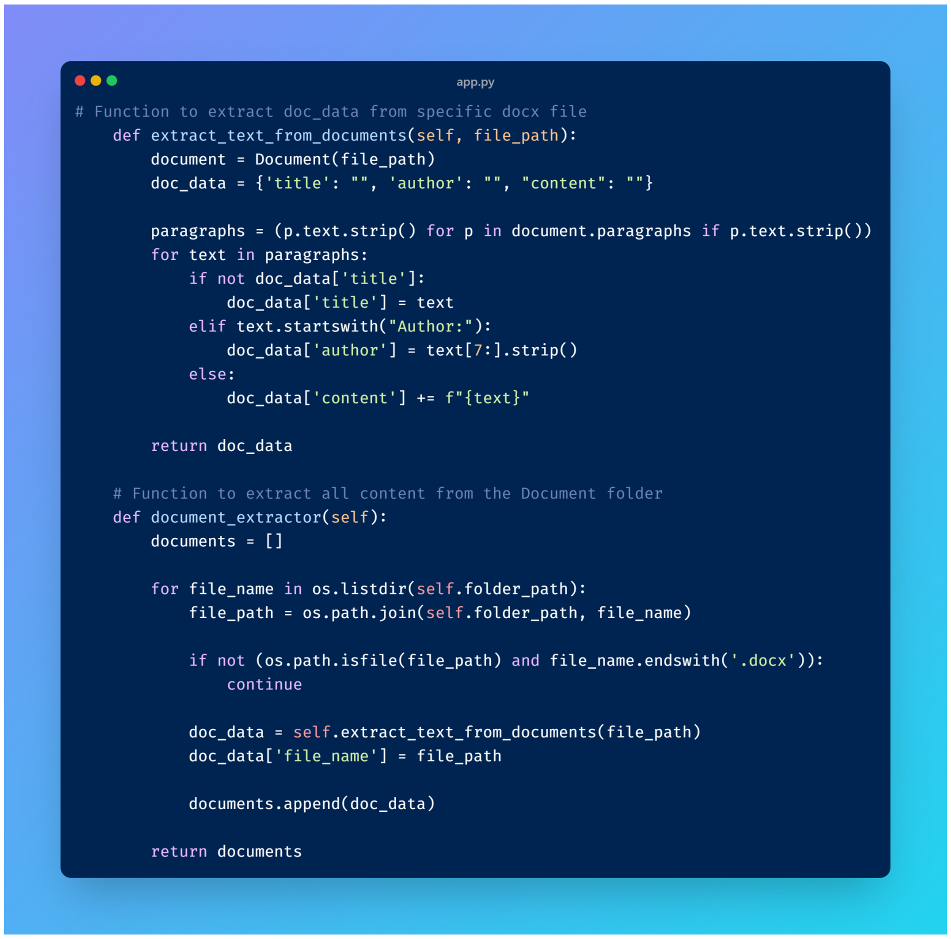
- **Keyword Suggestions**
  - Detects typos in user queries and provides corrected keyword suggestions.
  - Enhances user experience and ensures accurate searches.
- **Result Details**
  - Displays the following for each query:
    - **Title** of the document.
    - **Author Name(s)**.
    - **Snippet** 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.
- **Ranking**
  - Ranks documents based on **Cosine Similarity**.
  - 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.



```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.

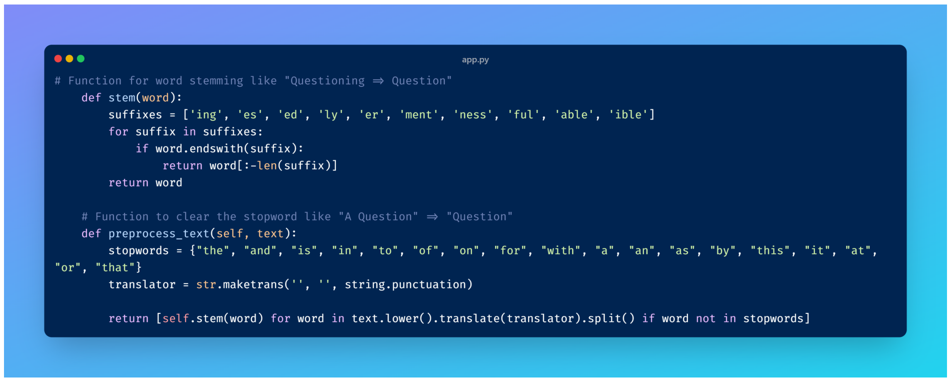
5.1.2 document_extractor function

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.

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.



```python
# Function for word stemming like "Questioning ⇒ Question"
    def stem(word):
        suffixes = ['ing', 'es', 'ed', 'ly', 'er', 'ment', 'ness', 'ful', 'able', 'ible']
        for suffix in suffixes:
            if word.endswith(suffix):
                return word[:-len(suffix)]
        return word

    # Function to clear the stopword like "A Question" ⇒ "Question"
    def preprocess_text(self, text):
        stopwords = {"the", "and", "is", "in", "to", "of", "on", "for", "with", "a", "an", "as", "by", "this", "it", "at",
"or", "that"}
        translator = str.maketrans('', '', string.punctuation)

        return [self.stem(word) for word in text.lower().translate(translator).split() if word not in stopwords]
```

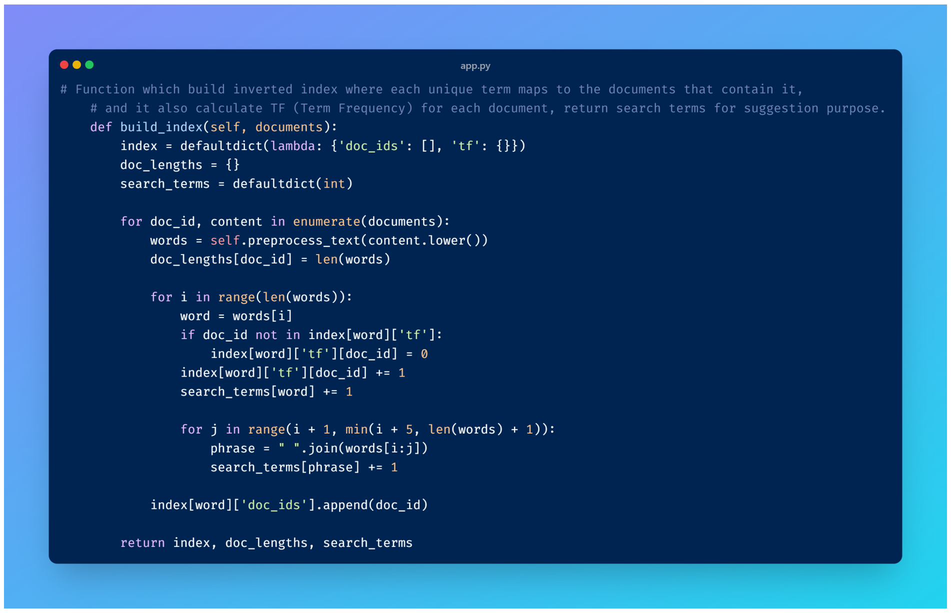
Figure 2: Text Preprocessing In Search Engine.

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.

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.

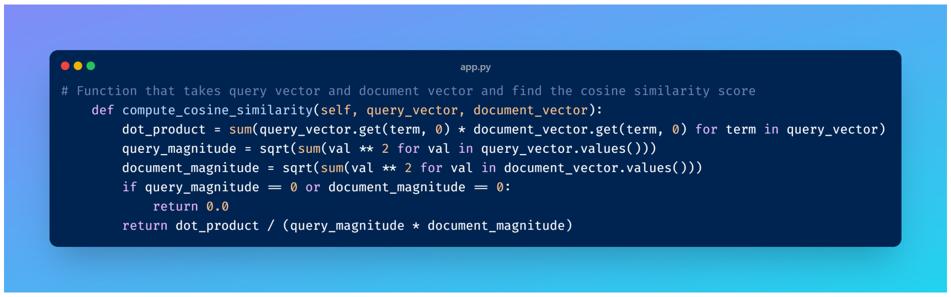


```python
# Function which build inverted index where each unique term maps to the documents that contain it,
    # and it also calculate TF (Term Frequency) for each document, return search terms for suggestion purpose.
    def build_index(self, documents):
        index = defaultdict(lambda: {'doc_ids': [], 'tf': {}})
        doc_lengths = {}
        search_terms = defaultdict(int)

        for doc_id, content in enumerate(documents):
            words = self.preprocess_text(content.lower())
            doc_lengths[doc_id] = len(words)

            for i in range(len(words)):
                word = words[i]
                if doc_id not in index[word]['tf']:
                    index[word]['tf'][doc_id] = 0
                index[word]['tf'][doc_id] += 1
                search_terms[word] += 1

                for j in range(i + 1, min(i + 5, len(words) + 1)):
                    phrase = " ".join(words[i:j])
                    search_terms[phrase] += 1

            index[word]['doc_ids'].append(doc_id)

        return index, doc_lengths, search_terms
```

Figure 3: Indexing In Search Engine.

5.4 Cosine Similarity Calculation

1. Dot Product Calculation
   (a) The function calculates the dot product between the query vector and the document vector.
   (b) For each term in the query vector, it multiplies the corresponding term values from the query and document vectors and sums them up.
   (c) If a term is absent in either the query or document vector, its value is considered as zero.
2. Magnitude of Query Vector
   (a) The magnitude (length) of the query vector is computed by summing the squares of the term values in the query vector.
   (b) The square root of this sum gives the magnitude of the query vector.
3. Magnitude of Document Vector
   (a) Similarly, the magnitude of the document vector is calculated by summing the squares of the term values in the document vector.
   (b) The square root of this sum gives the magnitude of the document vector.
4. Zero Magnitude Check
   (a) If either the query vector or the document vector has a magnitude of zero (indicating an empty vector or no valid terms), the cosine similarity is returned as 0.
   (b) A zero magnitude means that there is no meaningful similarity to compute.



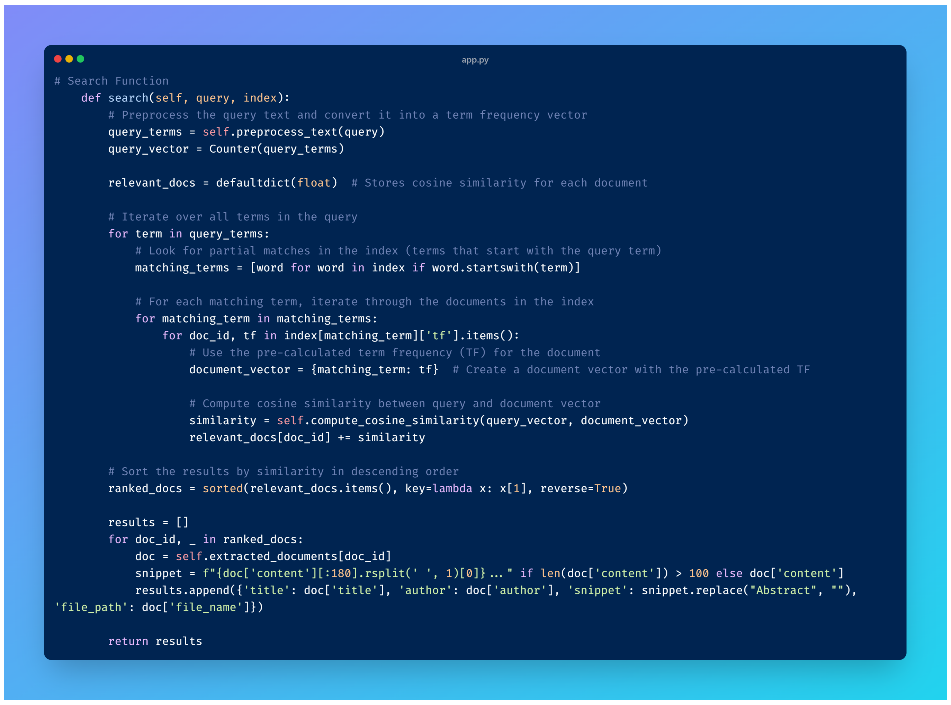
```python
# Function that takes query vector and document vector and find the cosine similarity score
    def compute_cosine_similarity(self, query_vector, document_vector):
        dot_product = sum(query_vector.get(term, 0) * document_vector.get(term, 0) for term in query_vector)
        query_magnitude = sqrt(sum(val ** 2 for val in query_vector.values()))
        document_magnitude = sqrt(sum(val ** 2 for val in document_vector.values()))
        if query_magnitude == 0 or document_magnitude == 0:
            return 0.0
        return dot_product / (query_magnitude * document_magnitude)
```

Figure 4: Cosine Similarity Calculation In Search Engine.

5.5 Search Query

The `search` function is responsible for finding and ranking documents based on a user's query. The process is explained below:

1. **Preprocessing the Query:**
   - The query text is preprocessed by removing stopwords and applying stemming.
   - A term frequency vector is created for the query using the preprocessed terms.
2. **Initialize Relevant Documents:**
   - A dictionary `relevant_docs` is created to store the cosine similarity score for each document.
3. **Iterate Over Query Terms:**
   - For each term in the query, the function searches for matching terms in the index (i.e., terms that start with the query term).
4. **Matching Term Handling:**
   - For each matching term, the function iterates through the documents in the index and retrieves the pre-calculated term frequency (TF) for that document.
   - A document vector is created using the term frequency of the matching term.
5. **Cosine Similarity Calculation:**
   - The cosine similarity between the query vector and the document vector is computed using the `compute_cosine_similarity` function.
   - The similarity score is added to the relevant document's score.
6. **Ranking Documents:**
   - The documents are ranked based on their similarity scores in descending order.
7. **Generate Results:**
   - The results include the document's title, author, a snippet of the content, and the file path.
   - If the content length exceeds 100 characters, a snippet is generated (up to 180 characters).
8. **Return Results:**
   - The final list of documents is returned, ranked by relevance to the query.



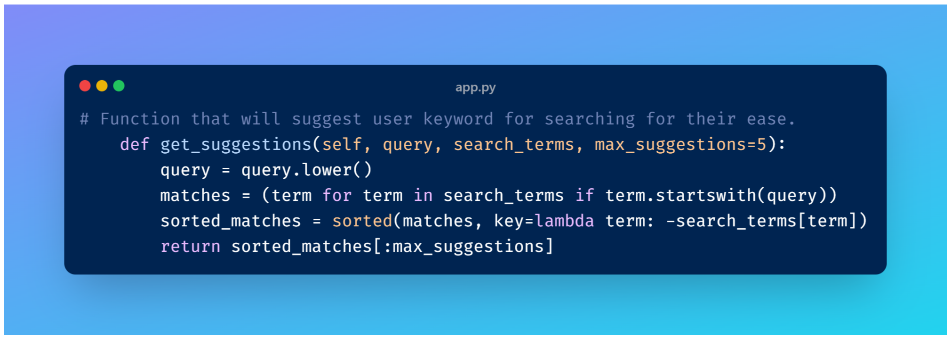
app.py

```python
# Search Function
    def search(self, query, index):
        # Preprocess the query text and convert it into a term frequency vector
        query_terms = self.preprocess_text(query)
        query_vector = Counter(query_terms)

        relevant_docs = defaultdict(float)  # Stores cosine similarity for each document

        # Iterate over all terms in the query
        for term in query_terms:
            # Look for partial matches in the index (terms that start with the query term)
            matching_terms = [word for word in index if word.startswith(term)]

            # For each matching term, iterate through the documents in the index
            for matching_term in matching_terms:
                for doc_id, tf in index[matching_term]['tf'].items():
                    # Use the pre-calculated term frequency (TF) for the document
                    document_vector = {matching_term: tf}  # Create a document vector with the pre-calculated TF

                    # Compute cosine similarity between query and document vector
                    similarity = self.compute_cosine_similarity(query_vector, document_vector)
                    relevant_docs[doc_id] += similarity

        # Sort the results by similarity in descending order
        ranked_docs = sorted(relevant_docs.items(), key=lambda x: x[1], reverse=True)

        results = []
        for doc_id, _ in ranked_docs:
            doc = self.extracted_documents[doc_id]
            snippet = f"{doc['content'][:180].rsplit(' ', 1)[0]}..." if len(doc['content']) > 100 else doc['content']
            results.append({'title': doc['title'], 'author': doc['author'], 'snippet': snippet.replace("Abstract", ""),
'file_path': doc['file_name']})

        return results
```

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.



app.py

```python
# Function that will suggest user keyword for searching for their ease.
    def get_suggestions(self, query, search_terms, max_suggestions=5):
        query = query.lower()
        matches = (term for term in search_terms if term.startswith(query))
        sorted_matches = sorted(matches, key=lambda term: -search_terms[term])
        return sorted_matches[:max_suggestions]
```

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

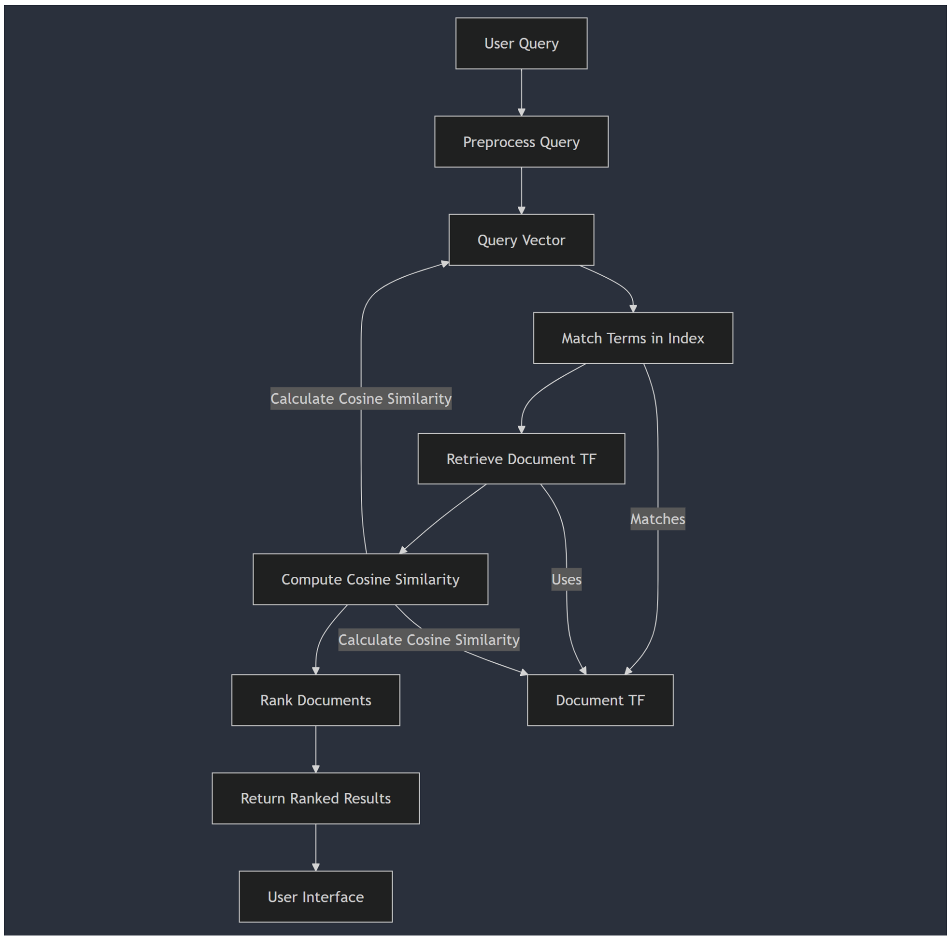


Figure 7: Data Flow Diagram.

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 word embedding 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.