Implement Set-theoretic Model

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 set-theoretic model, 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.

3 Set-Theoretic Model with Fuzzy Sets

Set-Theoretic Model with Fuzzy Sets is a model that represents documents and queries as fuzzy sets, where the membership degree of a term in a set is not binary but can range from 0 to 1. This allows for handling partial matches and imprecise queries. The model employs fuzzy logic to calculate the relevance of a document to a query by considering the degree of membership of each term in both the query and the document.

3.1 Set-Theoretic Model with Fuzzy Sets Formula

Definition: The relevance score between a document and a query in the Set-Theoretic Model with fuzzy sets can be expressed as:

$$\text{Relevance}(d, q) = \sum_{t \in q} \mu_{t,q} \times \mu_{t,d} \tag{1}$$

where:

- $\mu_{t,q}$ = Membership degree of term t in the query q.
- $\mu_{t,d}$ = Membership degree of term t in the document d.

3.2 Set-Theoretic Model with Fuzzy Sets Calculation

Explanation: The model calculates the relevance by aggregating the degree of match for each term between the query and the document. Each term's contribution is weighted by its membership degree in both the query and the document, allowing for partial and fuzzy matches:

- Terms that are highly relevant to both the query and the document contribute more to the overall relevance score.
- The fuzzy set allows the model to handle variations, misspellings, or partial term matches between the query and the document.

3.3 Ranking with Set-Theoretic Model with Fuzzy Sets

Procedure:

- For each document, calculate the fuzzy membership degree for each query term.
- Sum the products of the membership degrees to compute the relevance score for each document.
- Rank documents based on the computed relevance score, with higher scores indicating greater relevance to the query.

3.4 Example

Example: For a query ”climate change,” a document containing terms like ”environment,” ”global warming,” and ”climate” will have fuzzy membership degrees assigned to each term. Even if some terms are not an exact match, the fuzzy set allows for partial matches, ensuring relevant documents are ranked highly.

4 Key Features

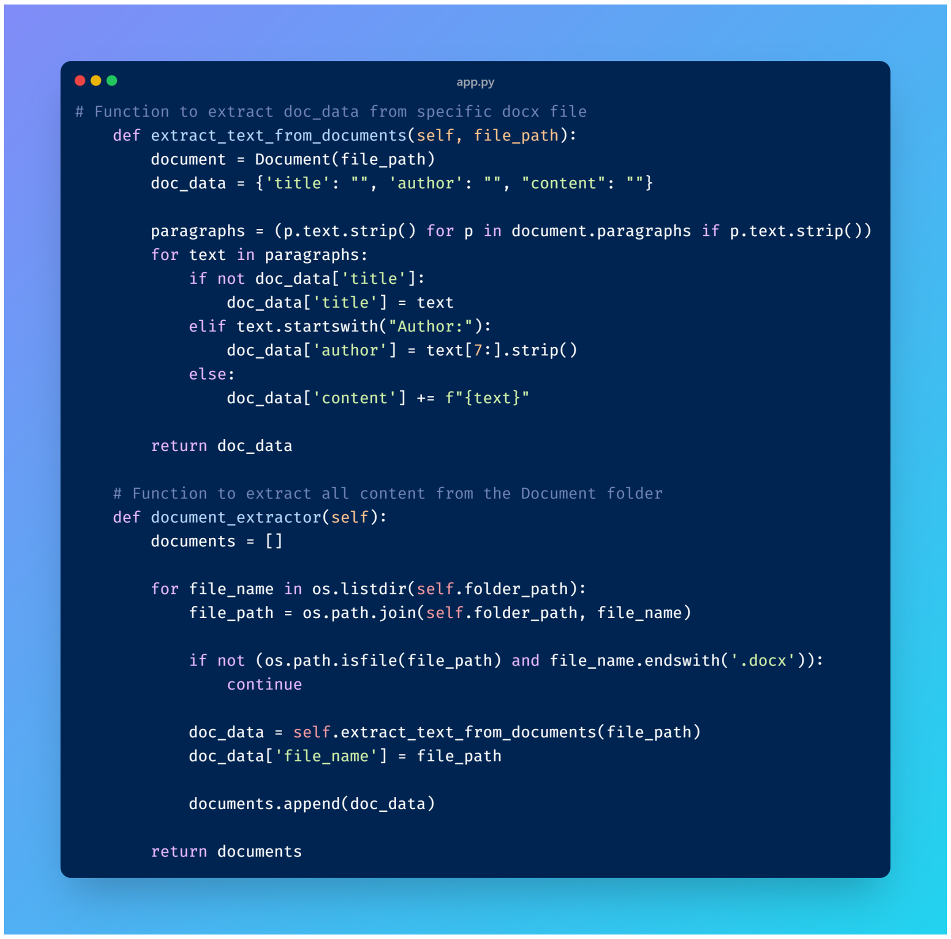
- **Keyword Suggestions**
  - Detects typos in user queries and provides corrected keyword suggestions.
  - Enhances user experience and ensures accurate searches.
- **Result Details**
  - Displays the following for each query:
    - **Title** of the document.
    - **Author Name(s)**.
    - **Snippet** 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 **set-theoretic model**.
  - 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.

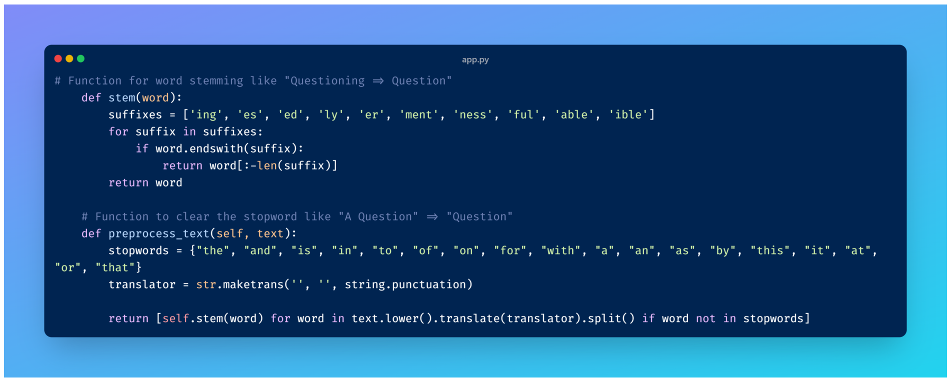
5.1.2 document_extractor function

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.

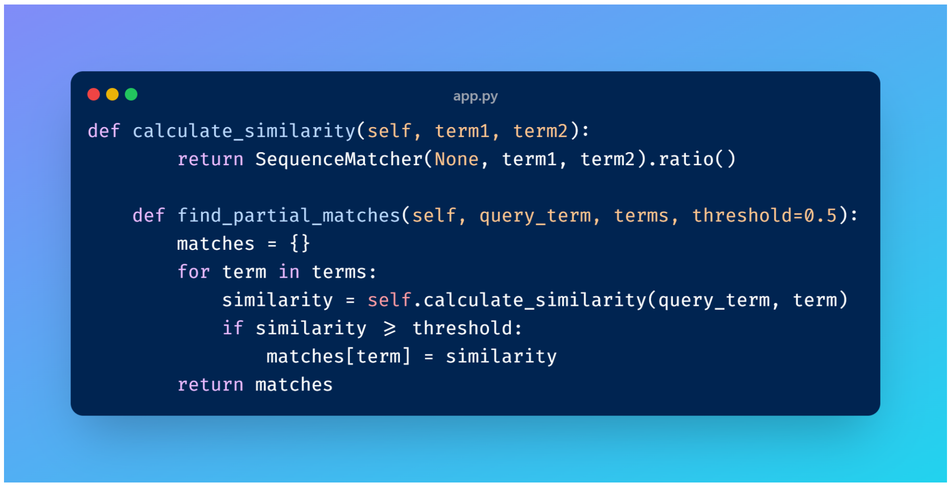
5.3 Partial Matching

5.3.1 calculate_similarity function

1. Uses the SequenceMatcher class from the difflib module to compute the similarity ratio between two terms.
2. The None parameter is for ignoring junk elements (which isn't needed here).
3. The function returns a similarity score (between 0 and 1) representing how similar term1 and term2 are.

5.3.2 find_partial_matches function

1. Iterates through a list of terms and compares each term with the query_term using the calculate_similarity function.
2. For each comparison, it calculates the similarity score.
3. If the similarity score is greater than or equal to the provided threshold (default is 0.5), the term is added to the matches dictionary with its similarity score.
4. Finally, it returns the matches dictionary, which contains terms that have a sufficient similarity to the query_term.

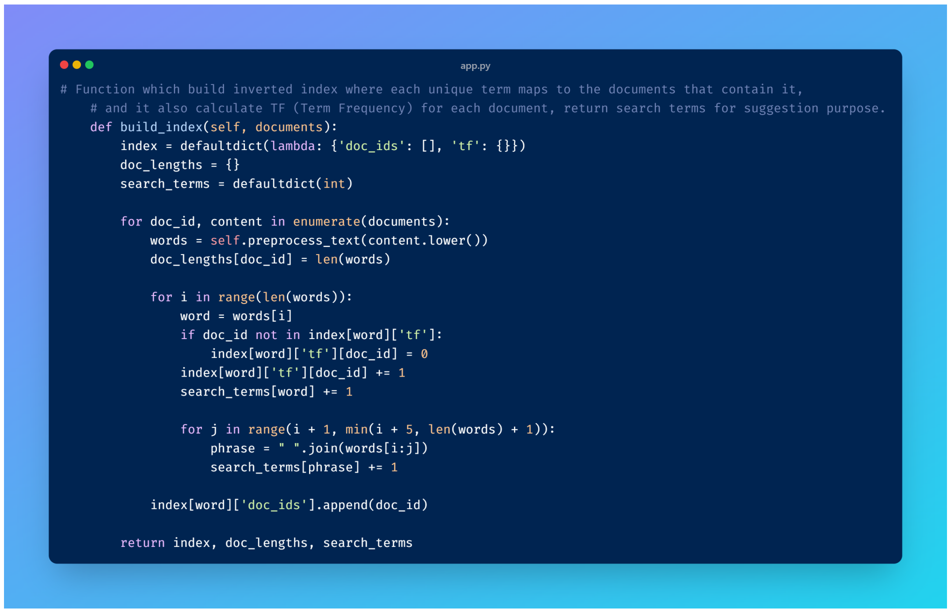


```
def calculate_similarity(self, term1, term2):
        return SequenceMatcher(None, term1, term2).ratio()

    def find_partial_matches(self, query_term, terms, threshold=0.5):
        matches = {}
        for term in terms:
            similarity = self.calculate_similarity(query_term, term)
            if similarity ≥ threshold:
                matches[term] = similarity
        return matches
```

Figure 3: Partial Matching In Search Engine.

5.4 Indexing

1. The function creates an index of words and n-grams (phrases) for a list of documents.
2. The input is a list of documents, and the output is an index and a count of search terms.
3. It initializes an empty index and a dictionary to count word frequencies.
4. For each document, it preprocesses the content by converting it to lowercase and removing unwanted characters.
5. It indexes each unique word in the document, adding the document ID to the index if the word appears.
6. It also counts how many times each word or n-gram (phrases of 1 to 4 words) appears in the documents.
7. The function returns the index of words and n-grams along with the term frequency counts for all terms.

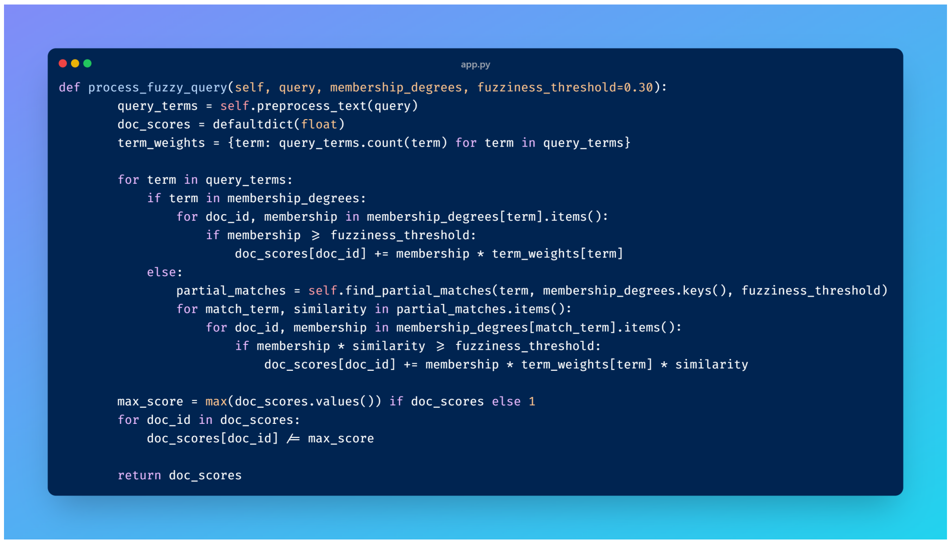


```python
# Function which build inverted index where each unique term maps to the documents that contain it,
    # and it also calculate TF (Term Frequency) for each document, return search terms for suggestion purpose.
    def build_index(self, documents):
        index = defaultdict(lambda: {'doc_ids': [], '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 4: Indexing In Search Engine.

5.5 Search Query

1. The function preprocesses the query to get individual terms.
2. It calculates the weight for each term based on its frequency in the query.
3. For each term, if it exists in membership_degrees, it checks if the document has a high enough membership value to add a score to that document.
4. If the term doesn't exist in membership_degrees, it finds similar terms and adds a weighted score for documents where those similar terms have a high enough membership.
5. After scoring, the function normalizes the document scores by dividing them by the highest score.
6. The function returns the normalized document scores based on the fuzzy query.

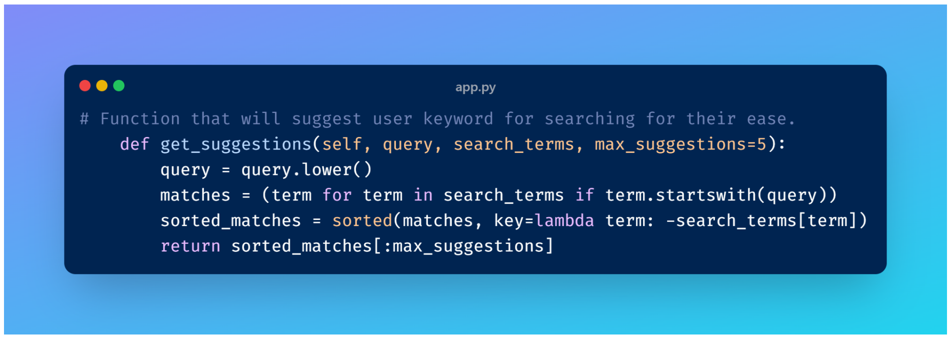


```python
def process_fuzzy_query(self, query, membership_degrees, fuzziness_threshold=0.30):
        query_terms = self.preprocess_text(query)
        doc_scores = defaultdict(float)
        term_weights = {term: query_terms.count(term) for term in query_terms}

        for term in query_terms:
            if term in membership_degrees:
                for doc_id, membership in membership_degrees[term].items():
                    if membership >= fuzziness_threshold:
                        doc_scores[doc_id] += membership * term_weights[term]
            else:
                partial_matches = self.find_partial_matches(term, membership_degrees.keys(), fuzziness_threshold)
                for match_term, similarity in partial_matches.items():
                    for doc_id, membership in membership_degrees[match_term].items():
                        if membership * similarity >= fuzziness_threshold:
                            doc_scores[doc_id] += membership * term_weights[term] * similarity

        max_score = max(doc_scores.values()) if doc_scores else 1
        for doc_id in doc_scores:
            doc_scores[doc_id] /= max_score

        return doc_scores
```

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.



```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/v8/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/v8/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/v8/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

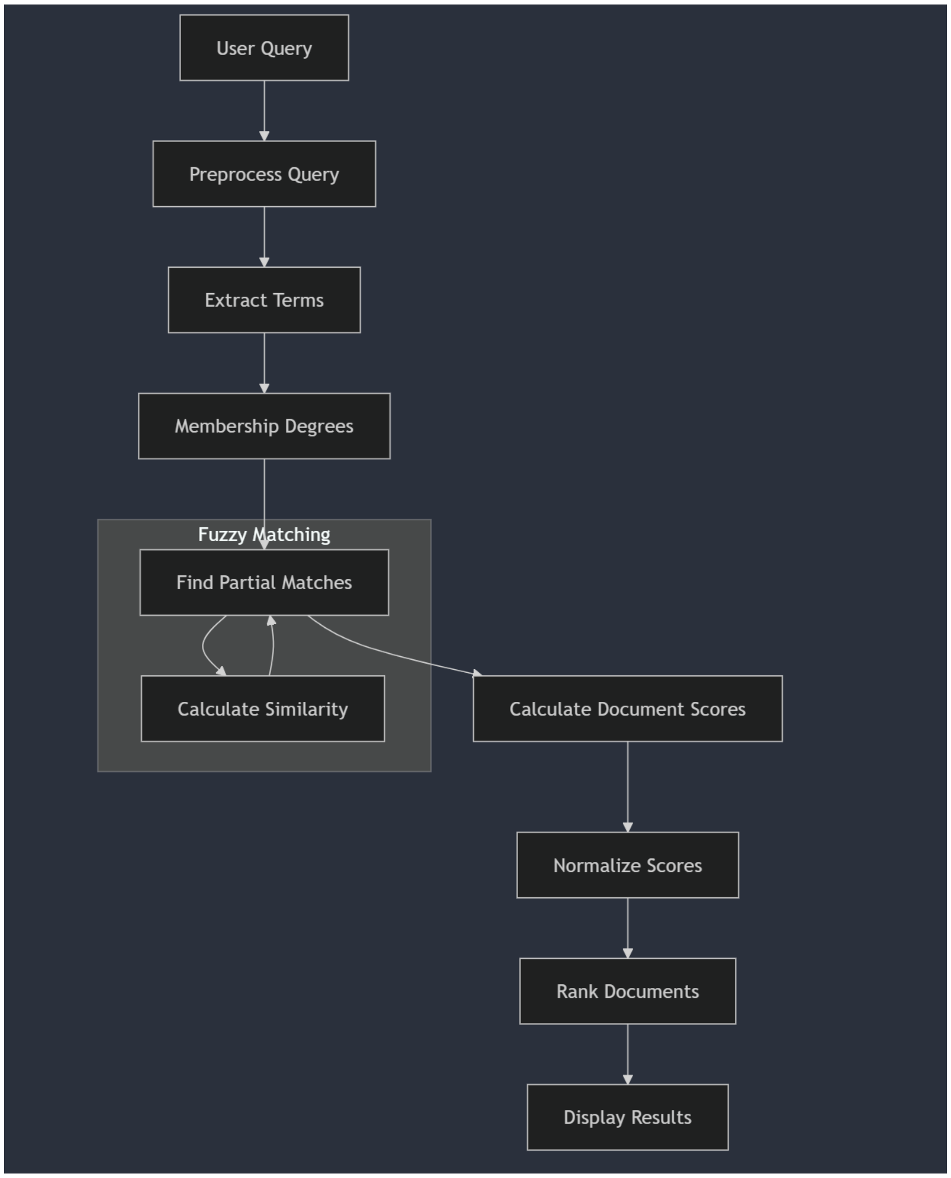


Figure 7: BIM Data Flow Diagram.

8 Results

- The search engine successfully indexes and ranks research articles using the set-theoretic model, ensuring more diverse and relevant search results.
- Keyword suggestions guide users in query formulation, improving the precision of search results.
- API endpoints provide flexible search options for full text, titles, and authors, supporting multiple retrieval models.

9 Future Enhancements

1. Implement word embeddings for ranking to improve relevance, allowing for semantic-based search.
2. Add support for additional file formats, such as PDF, ensuring the search engine is versatile across various document types.
3. Introduce advanced filtering options for search results (e.g., publication date, author affiliation) to refine search outcomes based on user preferences.