Implement BIM, Non-overlapped list and Proximal Nodes 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 BIM, NOLM and PNM, 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 Probabilistic Model

Probabilistic Model is a framework used for information retrieval that predicts the probability that a document is relevant to a given query. It is based on the concept of uncertainty and aims to estimate the likelihood that a document matches the user's intent, often using statistical methods to make these predictions.

3.1 Probabilistic Model Formula

Definition: In the probabilistic model, the relevance of a document is determined by the probability that the document is relevant to the query, given its features (terms). This can be expressed as:

$$P(\text{relevant}|\text{doc}, \text{query}) = \frac{P(\text{doc}|\text{query}) \cdot P(\text{query})}{P(\text{doc})} \tag{1}$$

where:

- $P(\text{relevant}|\text{doc}, \text{query})$ = Probability that the document is relevant given the query.
- $P(\text{doc}|\text{query})$ = Likelihood of the document given the query.
- $P(\text{query})$ = Prior probability of the query.
- $P(\text{doc})$ = Prior probability of the document being relevant.

3.2 Probabilistic Model Calculation

Explanation: The probabilistic model relies on the assumption that each document has a certain probability of being relevant to a given query. It evaluates the likelihood of relevance by considering:

- Document features, like term frequency or inverse document frequency, to calculate the likelihood of relevance.
- Statistical modeling techniques to estimate how well the document matches the query.

3.3 Ranking with Probabilistic Model

Procedure:

- For each document in the collection, the model estimates the probability of relevance to the query.
- Documents are ranked based on their relevance probability, with higher probabilities indicating greater relevance.

3.4 Example

Example: In a probabilistic model, if a query is "climate change," documents that contain frequent mentions of climate, environment, and related terms will have a higher probability of being relevant, even if those exact terms do not exactly match the query. The model ranks these documents according to their estimated relevance.

4 Non-Overlapping Lists Model

Non-Overlapping Lists Model is a retrieval model that operates on the principle of partitioning the term space into separate, non-overlapping lists for each query and document. The goal is to focus on documents that contain specific terms from the query, ignoring others that do not contribute to relevance.

4.1 Non-Overlapping Lists Model Formula

Definition: The model assumes that terms in the query and the document are mutually exclusive or non-overlapping. This is often formulated as:

$$\text{Relevance} = \sum_{t \in \text{query terms}} \text{TF}(t) \times \text{IDF}(t) \tag{2}$$

where:

- $\text{TF}(t)$ = Term frequency of term t in the document.
- $\text{IDF}(t)$ = Inverse document frequency of term t in the collection.

4.2 Non-Overlapping Lists Model Calculation

Explanation: The model works by assigning weight to terms that match between the query and the document. Since the lists are non-overlapping, it focuses on counting how many times query terms appear in the document and applies weight to the matching terms:

- The model ignores terms that do not appear in both the query and the document.
- It ranks documents based on the presence of terms from the query.

4.3 Ranking with Non-Overlapping Lists Model

Procedure:

- For each document, calculate the sum of term frequencies and inverse document frequencies for matching terms.
- Rank documents based on the computed sum, where a higher sum indicates higher relevance to the query.

4.4 Example

Example: For a query "climate change," a document containing frequent mentions of "climate" but not "change" will have a lower relevance score. A document that mentions both "climate" and "change" frequently will be ranked higher due to the non-overlapping consideration of query terms.

5 Proximal Nodes Model

Proximal Nodes Model is a model that captures the proximity or closeness of terms in documents to assess their relevance to a query. The idea is to consider not only the occurrence of terms but also how close the terms are to each other within the document.

5.1 Proximal Nodes Model Formula

Definition: The proximity of terms is often represented as a function of their distances in the document, which can be expressed as:

$$\text{Proximity Relevance} = \sum_{t \in \text{query terms}} \frac{1}{1 + \text{Distance}(t, q)} \tag{3}$$

where:

- $\text{Distance}(t, q)$ = Distance between term t and the query term q within the document.

5.2 Proximal Nodes Model Calculation

Explanation: In the proximal nodes model, terms that appear closer together in the document are considered more relevant than terms that are farther apart. This model captures the local structure of the document and the relationship between terms:

- Term proximity is considered to determine relevance.
- Closer terms within a document indicate higher relevance for a query.

5.3 Ranking with Proximal Nodes Model

Procedure:

- Calculate the proximity score for each term pair in the query and document.
- Rank the documents based on the total proximity score, with documents having closely related terms ranked higher.

5.4 Example

Example: For a query "climate change," a document that mentions "climate" and "change" within a few words of each other would be considered highly relevant. A document that mentions "climate" at the beginning and "change" at the end, separated by a large number of words, would be ranked lower due to the distance between the terms.

6 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 **BIM, Non-overlapped list and Proximal Nodes model**.
  - Ensures relevant documents appear higher in search results.

7 Implementation

7.1 Document Handling

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.

7.1.1 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.

7.1.2 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.

7.2 Text Preprocessing

app.py

```
# 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.

7.2.1 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.

7.2.2 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.

7.3 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.

app.py

```
# 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.

7.4 Dice Similarity Calculation

1. The function calculates the Dice similarity between a query and a document.
2. It takes two input vectors: the query vector and the document vector, which represent the term frequencies.
3. It computes the intersection by summing the product of matching terms from the query and document vectors.
4. It calculates the total number of terms by adding up the values in both the query and document vectors.
5. The Dice similarity score is then calculated as twice the intersection divided by the total number of terms.
6. If the total number of terms is zero (i.e., no terms in either vector), the similarity score is returned as zero.

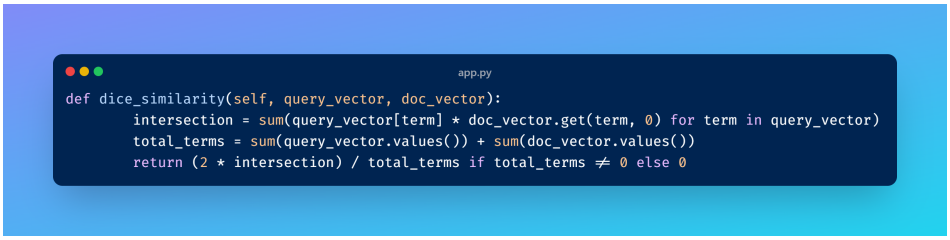

Figure 4: Dice Similarity Calculation In Search Engine.

7.5 Build Proximity Graph

1. Graph Initialization: Create an empty graph using NetworkX (nx.Graph()).
2. Iterate over Documents: Loop through each document in the provided documents list, assigning a doc_id to each document.
3. Preprocess Text: Preprocess the title, author, and content of each document by applying a text preprocessing function (preprocess_text()).
4. Combine Nodes: Combine all the processed terms (title, author, content) into a single set of nodes (all_nodes).
5. Add Document Node: Add the document itself as a node to the graph with the doc_id and its title as attributes (type=”document”).
6. Add Term Nodes: Add the individual terms from the document’s title, author, and content as nodes to the graph (type=”term”).
7. Create Edges: For each term node in all_nodes, create an edge between the document node and the term node, representing the proximity of terms to the document.
8. Return Graph: After all documents and their terms are processed, return the created proximity graph.

app.py

```python
# Build proximity graph
    def build_proximity_graph(self, documents):
        graph = nx.Graph()

        for doc_id, doc in enumerate(documents):
            title_nodes = self.preprocess_text(doc['title'])
            author_nodes = self.preprocess_text(doc['author'])
            content_nodes = self.preprocess_text(doc['content'])
            all_nodes = set(title_nodes + author_nodes + content_nodes)

            # Add document and related nodes to the graph
            graph.add_node(doc_id, type="document", title=doc['title'])
            for node in all_nodes:
                graph.add_node(node, type="term")
                graph.add_edge(doc_id, node)

        return graph
```

Figure 5: Graph Building For PNM.

7.6 Search Query with BIM

7.6.1 query_to_binary_vector function

1. The function takes a query and converts it into a binary vector.
2. It preprocesses the query, converts it into a set of terms, and then creates a binary vector.
3. Each term in the index is checked, and the corresponding value in the binary vector is set to 1 if the term is present in the query, otherwise 0.

7.6.2 rank_documents_bim function

1. The function ranks documents based on their similarity to the query.
2. It first converts the query into a binary vector using query_to_binary_vector.
3. It iterates over all document IDs that contain any of the query terms.
4. For each document, it creates a binary vector based on the presence of terms in the document.
5. It calculates the Dice similarity between the query vector and the document vector.
6. If the similarity is greater than 0, the document is added to a list with its similarity score.
7. The list of documents is sorted in descending order based on similarity.
8. Finally, the function returns a list of document IDs, sorted by their similarity to the query.

7.7 Search Query with NOLM

1. The function takes a search query, a TF-IDF dictionary, and a list of documents as input.
2. It preprocesses the query to extract relevant keywords.
3. For each keyword in the query, the function gathers a set of documents containing that keyword from the TF-IDF data.

Figure 6: Searching with BIM.

4. A combined set of documents is created, ensuring that documents containing any of the query keywords are considered.
5. Each document in the combined set is scored based on the sum of the TF-IDF values for the query keywords.
6. Documents are ranked by their score in descending order, and duplicates are avoided using a seen_docs set.
7. For each ranked document, a snippet is generated, and the document's title, author, snippet, and file path are included in the results.
8. The function returns a list of documents with their title, author, snippet, and file path.

Figure 7: Searching with NOLM.

7.8 Search Query with PNM

7.8.1 get_related_terms function

1. Input: Takes a query_term, a graph, and an optional threshold for fuzzy matching.
2. Identify Term Nodes: Extract all terms (nodes) from the graph that are of type 'term'.
3. Fuzzy Matching: Use difflib.get_close_matches() to find terms in the graph that are close to the query_term based on the threshold value.
4. Return Related Terms: Return the list of related terms that are similar to the query_term.

7.8.2 proximal_nodes_search function

1. Input: Takes a query, a graph, and documents.
2. Preprocess Query: Preprocess the query into individual terms (nodes).
3. Initialize Proximal Nodes: Create an empty set proximal_nodes to store nodes connected to the query terms.
4. Exact Matches: For each query term, check if it's in the graph. If it is, add its neighbors (nodes connected to it) to proximal_nodes.
5. Fuzzy Matches: For each query term, find related terms using get_related_terms(). Add the neighbors of these related terms to proximal_nodes.
6. Identify Connected Documents: Find documents that are connected to the proximal_nodes (term nodes) in the graph by checking if the node type is 'document'.
7. Create Result Snippets: For each connected document, generate a snippet of its content (first 180 characters) and remove "Abstract" from the snippet if it exists.
8. Return Results: Return a list of documents with their titles, authors, snippets, and file paths.

app.py

```
# Fuzzy matching for related terms
    def get_related_terms(self, query_term, graph, threshold=0.6):
        all_terms = [node for node, attrs in graph.nodes(data=True) if attrs.get('type') == 'term']
        related_terms = difflib.get_close_matches(query_term, all_terms, n=5, cutoff=threshold)
        return related_terms

    # Proximal Nodes Model Search
    def proximal_nodes_search(self, query, graph, documents):
        query_nodes = self.preprocess_text(query)
        proximal_nodes = set()

        for node in query_nodes:
            # Add exact matches
            if node in graph:
                proximal_nodes.update(nx.neighbors(graph, node))

            # Add fuzzy matches
            related_terms = self.get_related_terms(node, graph)
            for related_node in related_terms:
                if related_node in graph:
                    proximal_nodes.update(nx.neighbors(graph, related_node))

        # Fetch documents connected to the proximal nodes
        connected_docs = {node for node in proximal_nodes if graph.nodes[node].get('type') == 'document'}
        results = []

        for doc_id in connected_docs:
            doc = 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 8: Searching with PNM.

7.9 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.

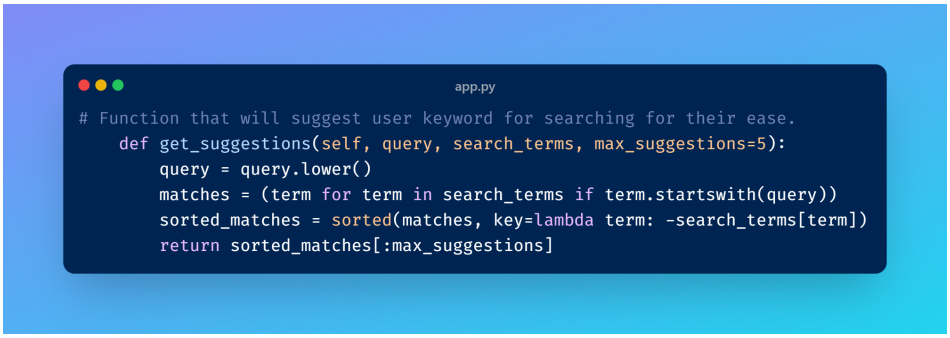

Figure 9: Keyword In Search Engine.

7.10 Evaluate Performance

1. The function evaluates the performance of the search engine based on precision, recall, and F1-score.
2. It starts by searching for results using the search_query_bim function with the given query.
3. It initializes two sets: one for relevant documents (relevant_docs) and another for retrieved documents (retrieved_docs).
4. It retrieves feedback for the query using get_feedback_for_keyword and processes each feedback item.
5. It then converts each retrieved document ID into a string and adds it to the retrieved_docs set.
6. The function returns the calculated precision, recall, and F1-score as a dictionary.

8 API Endpoints

8.1 `/suggestions`

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

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

app.py

```python
def evaluate_performance(self, query):
        results = self.search_query_bim(self.documents, self.content_index, query)

        relevant_docs = set()
        retrieved_docs = set()

        feedback = self.get_feedback_for_keyword(query)
        for feedback_item in feedback:
            if feedback_item["relevance"] == "relevant":
                relevant_docs.add(feedback_item["doc_id"])

        for result in results:
            # Convert retrieved doc_id to string for consistency
            retrieved_docs.add(str(result["doc_id"]))

        precision = len(relevant_docs & retrieved_docs) / len(retrieved_docs) if retrieved_docs else 0
        recall = len(relevant_docs & retrieved_docs) / len(relevant_docs) if relevant_docs else 0
        f1_score = (2 * precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

        return {'precision': precision, 'recall': recall, 'f1-score': f1_score}
```

Figure 10: Performance Evaluation of Search Engine.

8.2 /api/v3/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"
    }
]
```

8.3 /api/v3/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"
    }
]
```

8.4 /api/v3/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"
    }
]
```

9 Data Flow Diagram

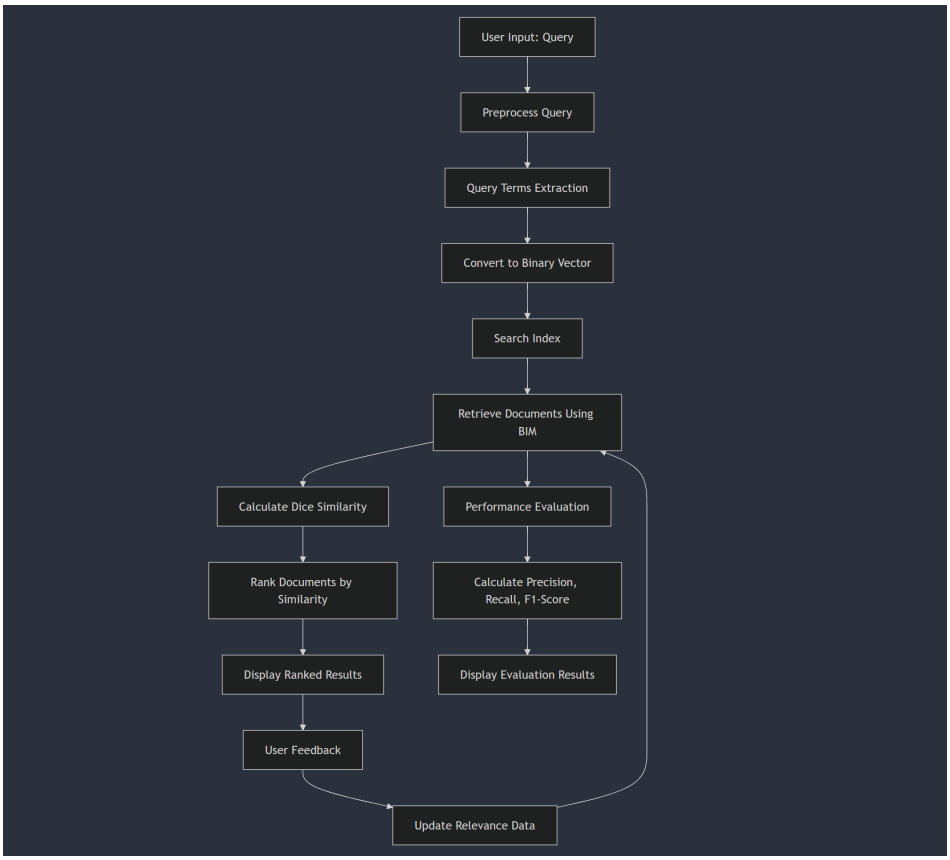

Figure 11: BIM Data Flow Diagram.

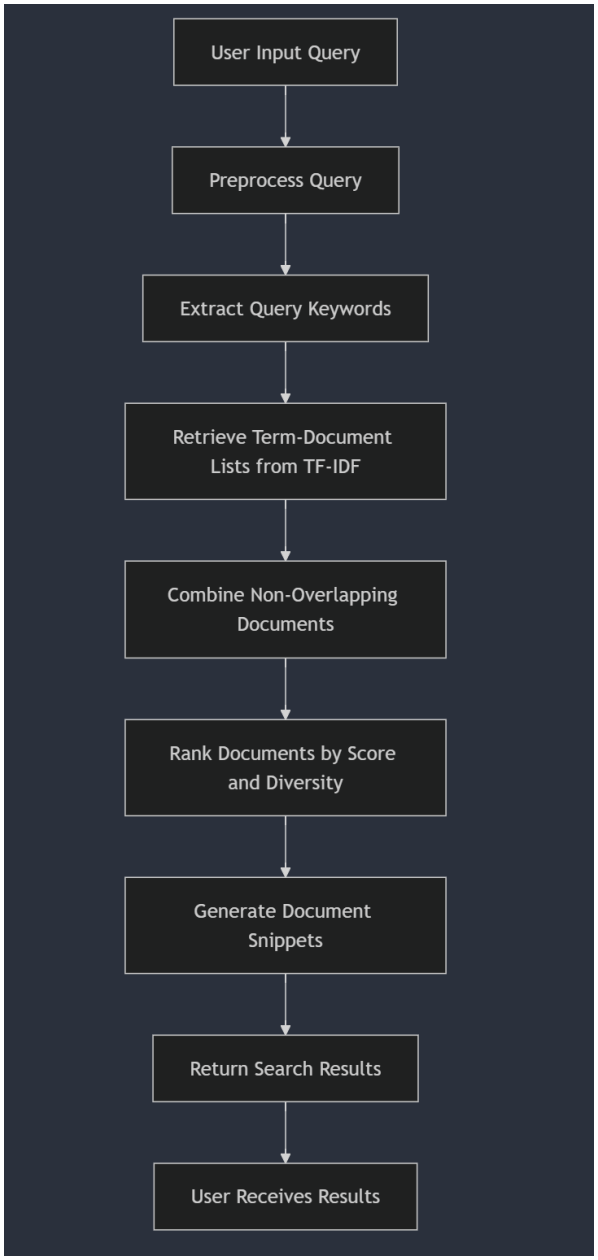

Figure 12: NOLM Data Flow Diagram.

10 Results

- The search engine successfully indexes and ranks research articles using the BIM model, ensuring more diverse and relevant search results.
- The integration of the NOLM model enhances the ranking by considering non-overlapping results and improving the query-document match score.
- The Proximal Nodes Model (PNM) enables advanced search by leveraging graph-based proximity and fuzzy matching, enhancing the retrieval of contextually relevant documents.
- Keyword suggestions guide users in query formulation, improving the precision of search results.

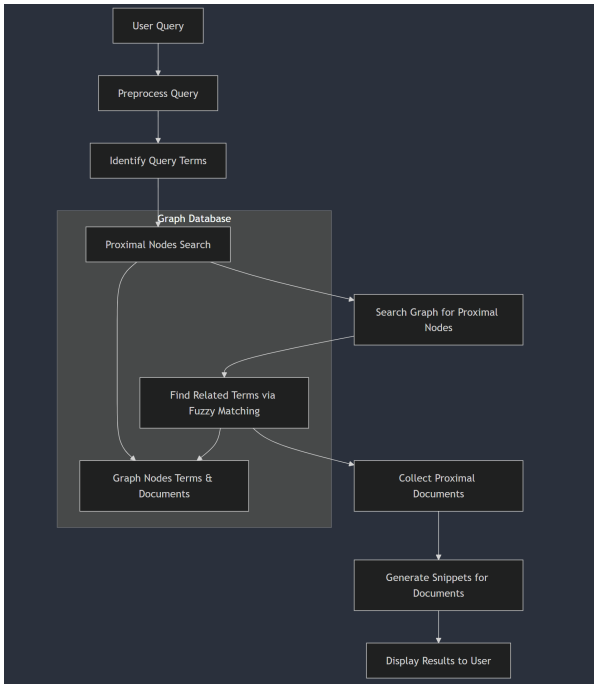

Figure 13: PNM Data Flow Diagram.

- API endpoints provide flexible search options for full text, titles, and authors, supporting multiple retrieval models.

11 Future Enhancements

1. Implement word embeddings for ranking to improve relevance, allowing for semantic-based search and enhancing the effectiveness of BIM and NOLM.
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.
4. Optimize performance for larger datasets by fine-tuning the algorithms of BIM, NOLM, and PNM models, ensuring faster retrieval and more accurate rankings.