Assignment-03

Session: 2021 – 2025

Submitted by:

Muhammad Yaqoob 2021-CS-118

Submitted to:

Dr Syed Khaldoon Khurshid

Department of Computer Science

University of Engineering and Technology UET Lahore

Table of Contents

Table Of Figures

1 Probabilistic Retrieval Model

1.1 Introduction

The Probabilistic Retrieval Model ranks documents based on their likelihood of relevance to the user's query. We implement the Binary Independence Model (BIM), which assumes that terms in a document are statistically independent.

1.2 Objective

To retrieve and rank documents using a probabilistic approach and present the top-K most relevant documents.

1.3 Steps to implement Probabilistic Retrieval Model

1.3.1 Preprocessing

I need to tokenize, stem, and remove stop words from both the documents and the query. Here is the function for the preprocessing.

```
def preprocess(text):
    # Tokenize
    tokens = text.lower().split()
    # Remove stop words
    stop_words = {'the', 'is', 'at', 'of', 'on', 'and', 'a', 'to'}
    tokens = [t for t in tokens if t not in stop_words]
    return tokens
```

1.3.2 Term Weighting

For each document and the query, create a binary vector where each term is marked as 1 if present and 0 otherwise.

```
def create_binary_vector(terms, vocab):
    vector = [1 if term in terms else 0 for term in vocab]
    return vector

def term_weighting(documents):
    # Preprocess documents and build vocabulary
    processed_docs = [preprocess(doc) for doc in documents]
    vocab = sorted(set([term for doc in processed_docs for term in doc]))  # Unique terms in all documents

    # Create binary vectors for each document
    doc_vectors = [create_binary_vector(doc, vocab) for doc in processed_docs]
    return doc_vectors, vocab
```

1.3.3 Query Representation

Convert the user's query into a binary vector using the vocabulary (terms across all documents).

```
def query_representation(query, vocab):
```

```
    query_terms = preprocess(query)
    return create_binary_vector(query_terms, vocab)
```

1.3.4 Document Scoring

The `scipy.spatial.distance` module includes a function called dice that computes the `Dice dissimilarity` between two boolean 1-D arrays. We can convert this dissimilarity to similarity by subtracting it from 1. For more details go to this link [dice — SciPy v1.14.1 Manual. (n.d.).]

```
from scipy.spatial.distance import dice

def calculate_dice_coefficient(query_vector, doc_vector):
    return 1 - dice(query_vector, doc_vector)
```

1.3.5 Ranking

Rank the documents based on their Dice coefficient scores.

```
def rank_documents(query_vector, doc_vectors):
    scores = [(i, calculate_dice_coefficient(query_vector, doc_vec)) for i, doc_vec in enumerate(doc_vectors)]
    ranked_docs = sorted(scores, key=lambda x: x[1], reverse=True)  # Higher Dice score means more similarity
    return ranked_docs
```

1.3.6 Retrieval

I have retrieved the top-K most similar documents. If user doesnt provide, i will return first five top documents.

```
def retrieve_top_k_documents(ranked_docs, K=5):
    return ranked_docs[:K]
```

1.4 Probabilistic Retrieval Model with Example

Given documents:

- Doc1: "The quick brown fox jumps over the lazy dog."
- Doc 2: "Never jump over the lazy dog quickly."
- Doc 3: "A fox is quick and a dog is lazy."

Query: "quick fox"

- Vocabulary: ['a', 'brown', 'dog', 'fox', 'is', 'jump', 'lazy', 'never', 'over', 'quick', 'the']
- Query vector: [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0]
- • Document vectors:
    - Doc 1: [0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1]
    - Doc 2: [0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1]
    - Doc 3: [1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0]
- Similarity Scores using the dice coefficient b/w documents and Query:
    - Doc 1: 0.67

- Doc 2: 0.5
- Doc 3: 0.75

Ranked results: Doc 3, Doc 1, Doc 2.

1.5 Real World Scenario

A search engine ranking webpage based on a user's query. For example, when searching "best smartphones 2024," the engine uses probabilities to rank pages most likely relevant to the query, ensuring users see the best results at the top.

1.6 DFD

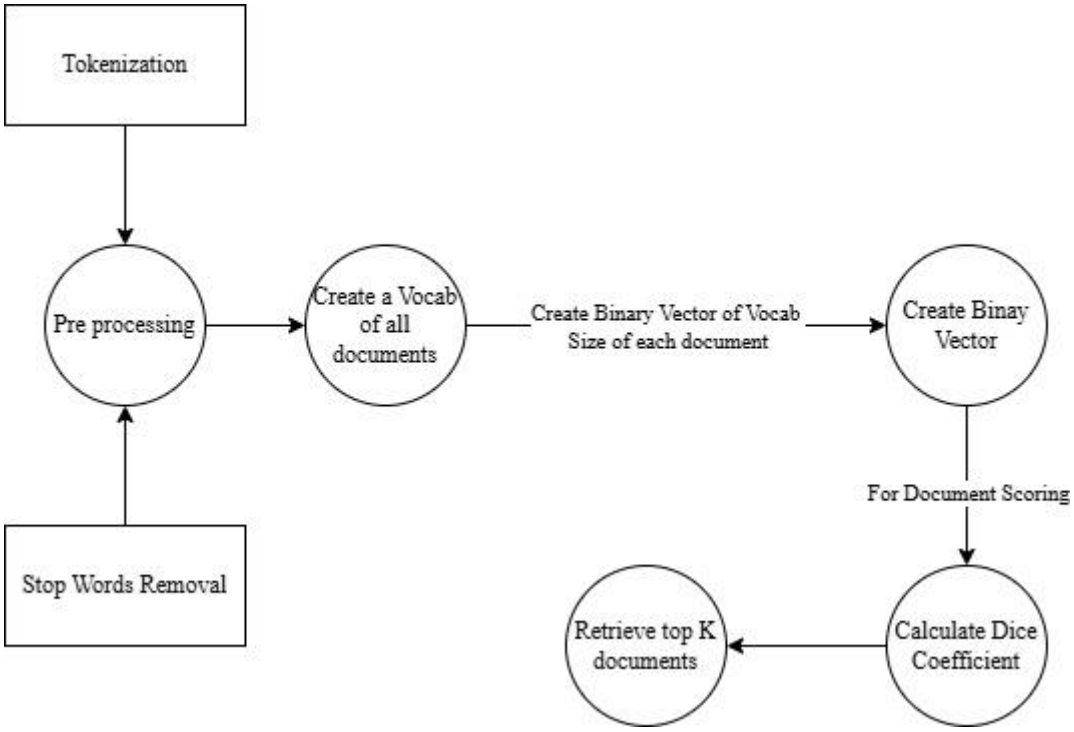

Figure 1: DFD of Probabilistic Retrieval Model

2 Non-Overlapped List Model

2.1 Introduction

This model retrieves documents containing any of the specified terms, ensuring no overlap between the results for different terms.

2.2 Objective

To combine document lists for different terms using a union operation, avoiding redundancy. Make use of the Link List data Structure.

2.3 Steps to implement Non-Overlapped List Model

2.3.1 Defining the Link List data structure

```
# Linked List class
class LinkList:

    # Private Node class
    class Node:
        def __init__(self, data):
            self.data = data
            self.next = None

    def __init__(self):
        self.head = None

    # Method to add a new node to the end of the list
    def append(self, data):
        new_node = self.Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    # Method to convert linked list to a set for set operations
    def to_set(self):
        elements = set()
        current = self.head
        while current:
            elements.add(current.data)
            current = current.next
        return elements

    # Method to print the linked list
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
```

```
            current = current.next
        print("None")
```

2.3.2 Create Link List for each term

```
# Create linked lists for each term
docs_machine_learning = LinkList()
docs_machine_learning.append("Introduction to machine learning and its applications.")
docs_machine_learning.append("Machine learning models and data science.")
docs_machine_learning.append("Combining machine learning and data visualization.")

docs_data_visualization = LinkList()
docs_data_visualization.append("Data visualization techniques and tools.")
docs_data_visualization.append("Effective data visualization methods.")
docs_data_visualization.append("Combining machine learning and data visualization.")
```

2.3.3 Combine Lists

I have defined a link list to set method within a Link List class to use set operations to find the union of the two sets of documents. This will give us the non-overlapping set of documents containing either of the terms.

```
set1 = docs_machine_learning.to_set()
set2 = docs_data_visualization.to_set()
non_overlap_set = set1.union(set2)
```

2.4 Real World Scenario

A library catalog where users search for books by multiple authors. The model retrieves unique books from each author without repeating results, helping users find a wider variety of books efficiently.

2.5 DFD

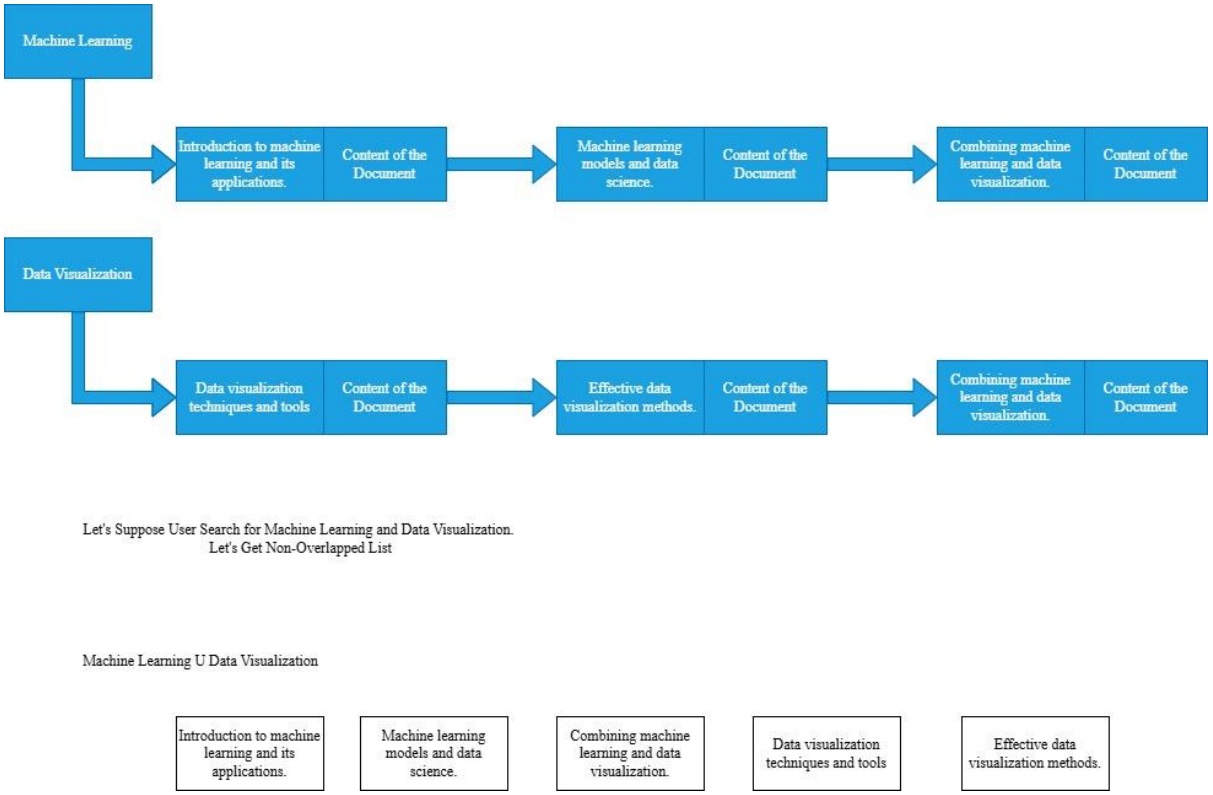

Figure 2: DFD of Non-Overlapped List Model

3 Proximal Nodes Model

3.1 Introduction

This model retrieves documents connected to nodes/entities in a graph. It is useful for exploring relationships in a network.

3.2 Objective

To identify and retrieve documents related to the query by leveraging graph connections.

3.3 Steps to implement Proximal Nodes Model

3.3.1 Defining the Graph data structure

I have used the dictionary with adjacency lists to represent the graph. Each node will be a key in a dictionary, and its value will be a list of connected nodes.

```
# Graph class to represent the network of documents and entities
class Graph:
    def __init__(self):
        # Initialize the graph with an empty adjacency list
        self.graph = {}

    # Add a node to the graph
    def add_node(self, node):
        if node not in self.graph:
            self.graph[node] = []
```

```
    # Add an edge between two nodes (undirected by default)
    def add_edge(self, node1, node2):
        if node1 in self.graph and node2 in self.graph:
            self.graph[node1].append(node2)
            self.graph[node2].append(node1)

    # Retrieve connected nodes (documents) to the given node
    def get_connected_nodes(self, node):
        return self.graph.get(node, [])

    # Display the graph (for debugging purposes)
    def display(self):
        for node in self.graph:
            print(f"{node}: {self.graph[node]}")
```

3.3.2 Add Nodes and Edges

```
# Create a graph instance
document_graph = Graph()

# Adding nodes (documents/entities)
document_graph.add_node("NASA")
document_graph.add_node("astronauts")
document_graph.add_node("space missions")
document_graph.add_node("moon landing")
document_graph.add_node("Mars exploration")
document_graph.add_node("space exploration")
document_graph.add_node("space telescopes")

# Adding edges (relationships)
document_graph.add_edge("NASA", "astronauts")
document_graph.add_edge("NASA", "space missions")
document_graph.add_edge("astronauts", "moon landing")
document_graph.add_edge("space missions", "Mars exploration")
document_graph.add_edge("NASA", "space telescopes")
document_graph.add_edge("space exploration", "NASA")
document_graph.add_edge("space exploration", "Mars exploration")
```

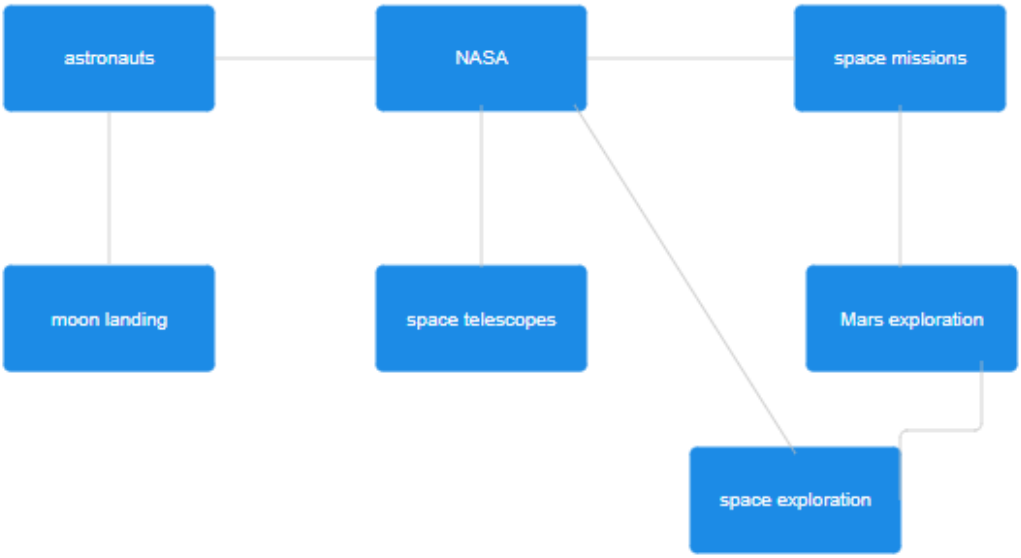

Figure 3: Exploring Graph

3.3.3 Retrieval

We'll traverse the graph to find all nodes connected to these proximal nodes.

```
# Function to explore network relationships and find connected documents
def retrieve_documents(graph, proximal_nodes):
    connected_documents = set()
    for node in proximal_nodes:
        # Retrieve all nodes directly connected to each proximal node
        connected_nodes = graph.get_connected_nodes(node)
        # Add the connected nodes to the result set
        for connected_node in connected_nodes:
            connected_documents.add(connected_node)
    return connected_documents

# Retrieve the connected documents
# Identify proximal nodes based on interest
proximal_nodes = ["moon landing"]

relevant_documents = retrieve_documents(document_graph, proximal_nodes)
```

3.4 Real World Scenario

Social media platforms analyzing content connections. For instance, when exploring a topic like "climate change," the model retrieves posts or articles connected to related entities, such as "renewable energy" or "global warming," to provide broader context.

3.5 DFD

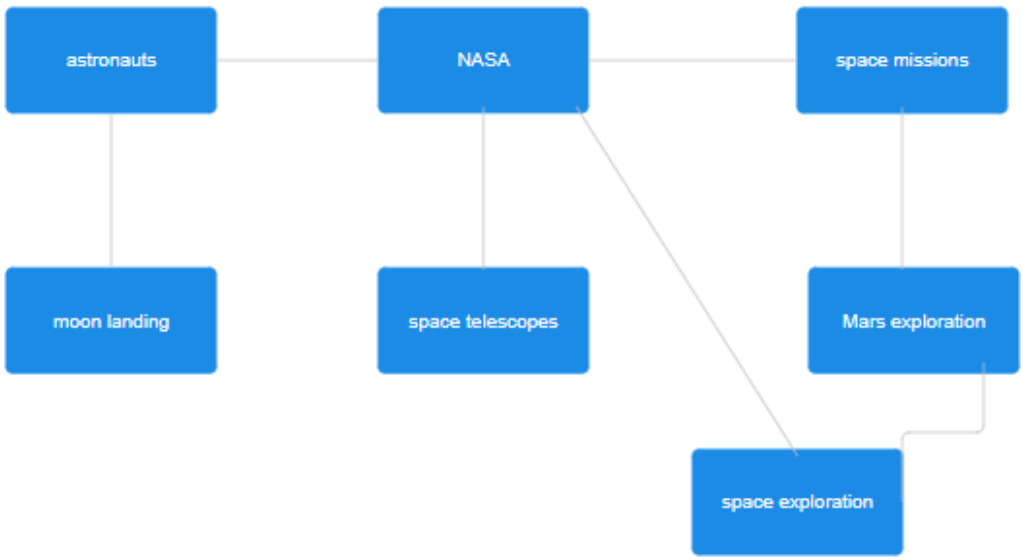

Figure 4: DFD of Proximal Nodes Model

4 Conclusion

Each retrieval model offers unique advantages:

- **Probabilistic Retrieval Model:** Effective for ranking relevance.
- **Non-Overlapped List Model:** Ensures non-redundant results.
- **Proximal Nodes Model:** Leverages relationships in a network for context-aware retrieval.