**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

[1 Probabilistic Retrieval Model 4](#_Toc184021420)

[1.1 Introduction 4](#_Toc184021421)

[1.2 Objective 4](#_Toc184021422)

[1.3 Steps to implement Probabilistic Retrieval Model 4](#_Toc184021423)

[1.3.1 Preprocessing 4](#_Toc184021424)

[1.3.2 Term Weighting 4](#_Toc184021425)

[1.3.3 Query Representation 4](#_Toc184021426)

[1.3.4 Document Scoring 5](#_Toc184021427)

[1.3.5 Ranking 5](#_Toc184021428)

[1.3.6 Retrieval 5](#_Toc184021429)

[1.4 Probabilistic Retrieval Model with Example 5](#_Toc184021430)

[1.5 Real World Scenario 6](#_Toc184021431)

[1.6 DFD 6](#_Toc184021432)

[2 Non-Overlapped List Model 7](#_Toc184021433)

[2.1 Introduction 7](#_Toc184021434)

[2.2 Objective 7](#_Toc184021435)

[2.3 Steps to implement Non-Overlapped List Model 7](#_Toc184021436)

[2.3.1 Defining the Link List data structure 7](#_Toc184021437)

[2.3.2 Create Link List for each term 8](#_Toc184021438)

[2.3.3 Combine Lists 8](#_Toc184021439)

[2.4 Real World Scenario 8](#_Toc184021440)

[2.5 DFD 9](#_Toc184021441)

[3 Proximal Nodes Model 9](#_Toc184021442)

[3.1 Introduction 9](#_Toc184021443)

[3.2 Objective 9](#_Toc184021444)

[3.3 Steps to implement Proximal Nodes Model 9](#_Toc184021445)

[3.3.1 Defining the Graph data structure 9](#_Toc184021446)

[3.3.2 Add Nodes and Edges 10](#_Toc184021447)

[3.3.3 Retrieval 11](#_Toc184021448)

[3.4 Real World Scenario 11](#_Toc184021449)

[3.5 DFD 12](#_Toc184021450)

[4 Conclusion 12](#_Toc184021451)

Table Of Figures

[Figure 1: DFD of Probabilistic Retrieval Model 6](#_Toc184021452)

[Figure 2: DFD of Non-Overlapped List Model 9](#_Toc184021453)

[Figure 3: Exploring Graph 11](#_Toc184021454)

[Figure 4: DFD of Proximal Nodes Model 12](#_Toc184021455)

# Probabilistic Retrieval Model

## 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.

## Objective

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

## Steps to implement Probabilistic Retrieval Model

### 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 |

### 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 |

### 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) |

### 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.).]](https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.dice.html)

|  |
| --- |
| from scipy.spatial.distance import dice  def calculate\_dice\_coefficient(query\_vector, doc\_vector):      return 1 - dice(query\_vector, doc\_vector) |

### 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 |

### 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. 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.

## 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.

## DFD

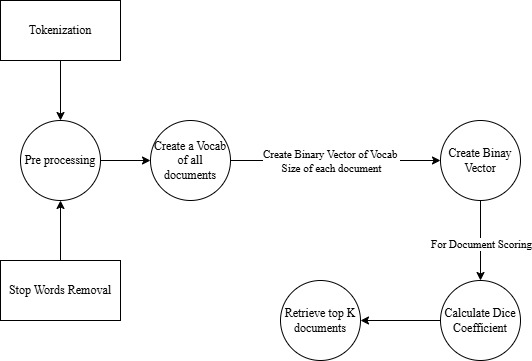


Figure 1: DFD of Probabilistic Retrieval Model

# Non-Overlapped List Model

## Introduction

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

## Objective

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

## Steps to implement Non-Overlapped List Model

### 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") |

### 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.") |

### 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) |

## 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.

## DFD

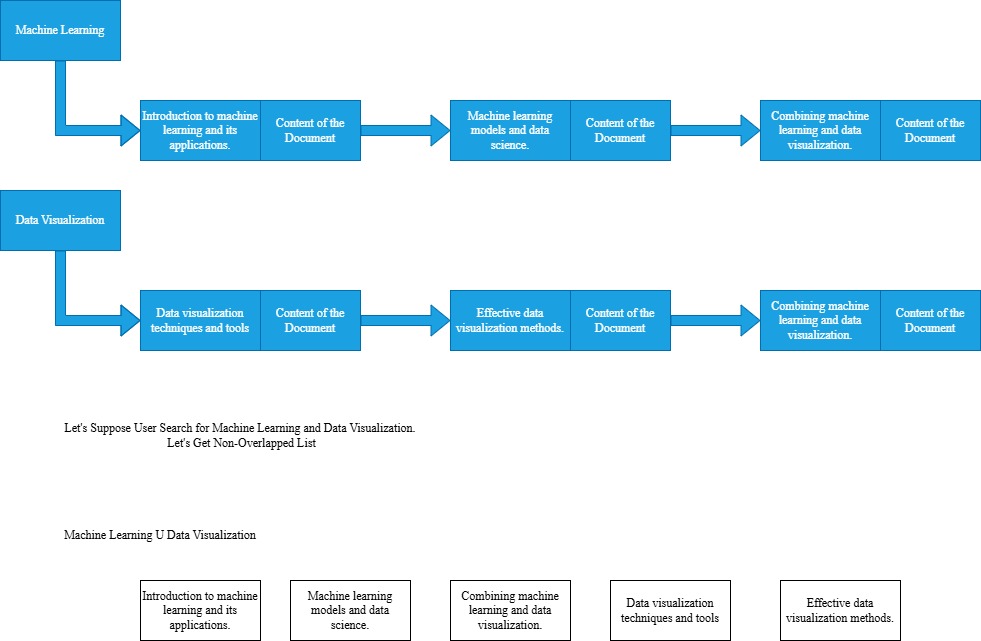


Figure 2: DFD of Non-Overlapped List Model

# Proximal Nodes Model

## Introduction

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

## Objective

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

## Steps to implement Proximal Nodes Model

### 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]}") |

### 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") |

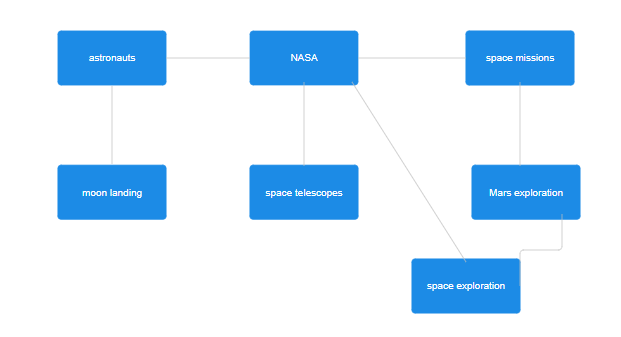


Figure 3: Exploring Graph

### 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) |

## 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.

## DFD

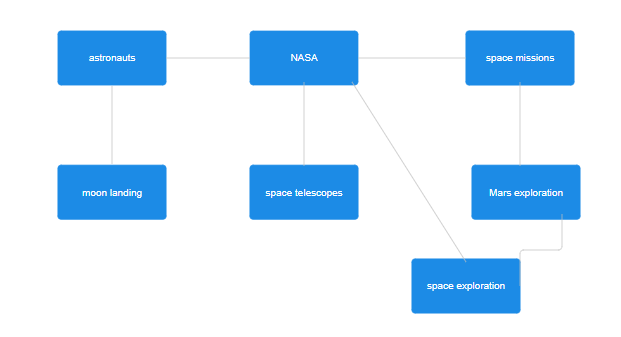


Figure 4: DFD of Proximal Nodes Model

# 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.