CS-488 Information Retrieval C

Final Report

Submitted To:

Dr. Syed Khaldoon Khurshid

Submitted by:

Name	Reg. No.
Muhammad Yaqoob	2021-CS-118
Safiullah Sohail	2021-CS-120
Umair Shahid	2021-CS-144
Ahmed Raza	2021-CS-161

Department of Computer Science

University of Engineering and Technology (UET) Lahore

ACKNOWLEDGEMENT

The successful completion of these projects would not have been possible without the support and guidance of several individuals, to whom we owe our heartfelt gratitude.

Firstly, we would like to express our profound appreciation to our supervisor, Dr Syed Khaldoon Khurshid for his continuous guidance, constructive criticism, and encouragement throughout the course of this project. His invaluable insights and expertise provided a solid foundation for our work, helping us navigate challenges effectively.

We are equally indebted to our family for their unwavering support and understanding during the completion of these projects. Their patience and encouragement have been a constant source of motivation.

We would also like to thank our friends and colleagues for their valuable suggestions and moral support, which significantly contributed to this project. Their collaboration and feedback have been incredibly helpful in refining our ideas and approach.

Lastly, We extend our gratitude to the institution and department for providing the necessary resources and a conducive environment for learning and development. These projects have been an enriching experience, and we are grateful for the opportunity to apply theoretical knowledge to practical implementation.

Table of Content

1. Project 1: Document Search Engine

1.1. Goals of the search engine

Create a fast and optimized search engine intelligent enough to retrieve documents based upon either name or content of the document. Here are the main objectives of our document search engine.

- Quick Retrieval of data
- Allow retrieval based upon file name or content
- Allow user to query data even if the query does not exactly match
- Keeps the index up to date with changes in the documents
- Optimized usage of computer resources
- Support multiple document formats

1.2. Proposed Solution

- **Indexing:** For fast retrieval of data we can create indexes of files that can keep updating and provide us an easier way of knowing file contents. Just as how Table of contents works for a book.
- **User Interaction:** Explicitly ask the user on whether he wants to retrieve the document location only based upon name or to identify the document based upon contents.
- **Query Handling:** Match incomplete queries with existing terms and show relevant documents to the user.
- **Index Updating:** Keep updating the index so it stays filled with the latest contents from documents.
- **Optimizing Usage:** Because the data may become very large the indexes may also become very space consuming, we optimize the memory usage by loading query relevant parts of the index.
- **Document Support:** Add support for multiple types of documents like txt, csv, pdf, office documents, other media, etc.

1.3. Implementation

1.3.1. Data Gathering/Selection

A folder "data" is a dedicated directory where all the documents are to be collected, our application will look for documents and subdirectories in this folder and create indexes for all documents.

1.3.2. Development

The application is developed using Python. It is a simple Command-Line interface based Menu-driven Application allowing users to easily input queries and view results.

1.3.3. Indexing Documents

Indexing is like creating a quick reference guide for a large amount of information.

Imagine a giant book without a table of contents or an index at the back. Finding a specific topic would take a lot of time because you'd have to flip through all the pages. But with an index, you can quickly find the page numbers for any topic you're interested in.
In the same way:
In search engines: An index is created for each word or topic so that when you search for something, the search engine can find it quickly without looking through every single document.
In our system 2 types of indexes are being created:

1. **Filename Index:** It stores the locations of documents in the data directory along with the last updated timestamp of that document.
   Representation: demo.txt: "data/business/demo.txt", "data/medicine/demo.txt"
2. File Content Index: It stores words from documents in along with the document name and a snippet of near words at the occurrence of that word in that specific document.

1.3.4. Searching Functions

Following two searching functions are used in the application:

1. **Search Content:** This function searches for the query in contents index and displays the result. It is also responsible for updating the contents index depending upon the user query because not all of the index is loaded at once.
2. **Search Filename:** This function is responsible for retrieving the location of a file from the filename index.

Both searching functions allow exact match retrievals and pattern matching results. Pattern matching means if the query is incomplete the functions will look for the words that contain all letters of the queried word/filename and then show them as results.

*Current dataset consists of data from 10 newsgroups each containing 100 files. Resource(Kaggle): (10)Dataset Text Document Classification

1.4. Optimization

1.4.1. File monitoring

We have used a file monitor which looks for changes in the data folder, if a file is added, updated or deleted the monitor tells the app to update the index.
This allows the indexes to be always up to date.

1.4.2. Reading Subindex

The search engine whenever looking for content based searches read only those specific words at first that have the same first letter as the queried word. This helps in optimizing both space and time. This part of the index is kept in memory until the first letter of the query changes.

1.4.3. Multithreading

Multithreading lets a program do multiple tasks at once, like having two hands work together to build something faster.
Example: Imagine you're writing a story with one hand while drawing with the other—both tasks happen at the same time, separately but together.

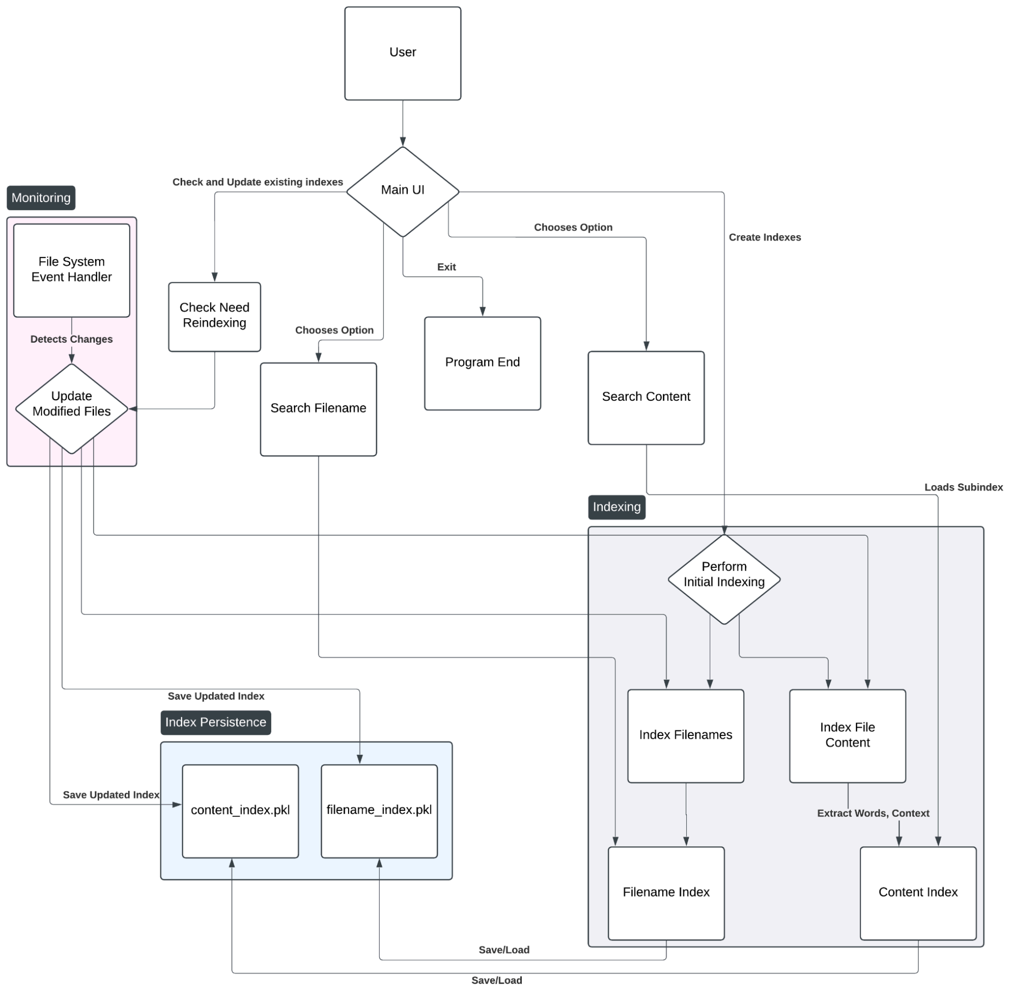
The indexer is using multithreading to read multiple subdirectories at once. The file monitor is also being run on a separate thread so it runs independent of the app.

1.4.4. Check Reindexing Eligible

In the filenames index we are storing the file's last update timestamp which is compared with the current timestamp of the file and we know if the file was updated or not. This helps if the file was updated while the application was closed. On restart the index can be updated by using this check.

1.5. Design and workflow

- Loads indexes from files if available; otherwise, performs a full indexing of the `data` directory.
- Starts a background thread that continuously watches for changes in the file directory.
- Provides options for content search or filename search, allowing both exact and pattern-based queries.
- If a file is modified or deleted, the monitoring handler triggers an update in the relevant indexes.



2. Project 2: Document Ranking System

2.1. Goals of the system

“The project is a Document Ranking System that helps users find the most relevant documents from a collection based on their queries.”

- Allow users to query data even if the query does not exactly match.
- Process documents to extract important terms.
- Rank the retrieved documents by relevance to query.
- Provide a user-friendly interface for query input and result display.

2.2. Proposed Solution

2.2.1. Query Handling

Match incomplete queries with existing terms and show relevant documents to the user.

2.2.2. Process Documents

Process the documents and query to extract only important terms so searching is made easier.

2.2.3. Document Ranking

Uses mathematical techniques like Term frequency-inverse document frequency(TF-IDF) and Inverse to rank documents by relevance.

2.2.4. Techniques Used

Keyword matching: Simple comparison of words in the query and documents.
TF-IDF scoring: A mathematical method to measure how important a word is to a document relative to all documents.

2.3. Implementation

2.3.1. Data Gathering/Selection

A folder “data” is a dedicated directory where all the documents are to be collected, our application will look for documents and subdirectories in this folder and create indexes for all documents.

2.3.2. Development

We used Python for coding and VS Code as the code editor.
Python is like a Swiss army knife for programming – it has tools (libraries) for almost anything you need.
Think of VS Code as a comfortable desk where you can organize and write your work efficiently.

2.3.3. Searching

Imagine a treasure hunt!

- The query is like asking, "Where is my treasure (document)?"
- Each document is like a treasure chest filled with words (terms).

Steps in the search:

- Break words apart: We split the query and all documents into smaller parts (tokens/terms).
- Score the treasure chests: We use math (TF-IDF) to decide how closely each document matches your query.
- Rank treasures: We sort the documents, showing the best matches first!

2.3.4. TF-IDF

TF (Term Frequency):

- “ How often does a word appear? “
- More frequent words are more important.
- Example: If "cat" appears 10 times, it's more important
- than "dog," which appears once.

IDF (Inverse Document Frequency):

- “How special is this word? ”
- If a word like "the" appears in all documents, it’s less useful.
- Rare words matter more.
- The "+1" avoids dividing by zero.

TF-IDF:

- Combine both! A word is important if:
- It appears often in the document (TF is high).
- It’s rare across all documents (IDF is high).

Why These Formulas?

- TF is in the numerator:
  A word's importance grows as it appears more in a document.
- IDF is in the numerator (inside a log):
  Rare words are more meaningful, so we emphasize them.
- Why multiply?
  Both frequency (TF) and rarity (IDF) are important! Multiplication combines them.

2.3.5. Vectorization and Cosine Similarity

Vectorization (Turning words into numbers):

1. Imagine we have two documents:
   - Doc 1: "Cats are cute."
   - Doc 2: "Dogs are cute too."
   - Query: "Cute cats."
2. First, we make a list of all unique words: [cats, are, cute, dogs, too]. Each document becomes a "list of numbers" (a vector).

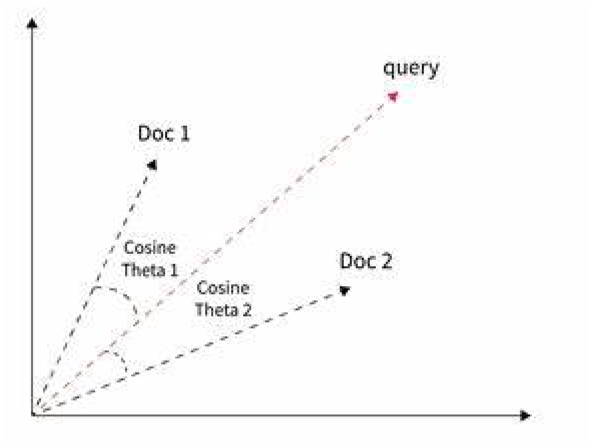
Example:

- Doc 1 → [1, 1, 1, 0, 0] (1 means the word is present, 0 means it isn't).
- Doc 2 → [0, 1, 1, 1, 1].
- Query → [1, 0, 1, 0, 0].

3. Why do we do this?
   Computers understand numbers, not words.
   Turning text into vectors allows mathematical comparisons.

Cosine Similarity (Finding how "close" vectors are):

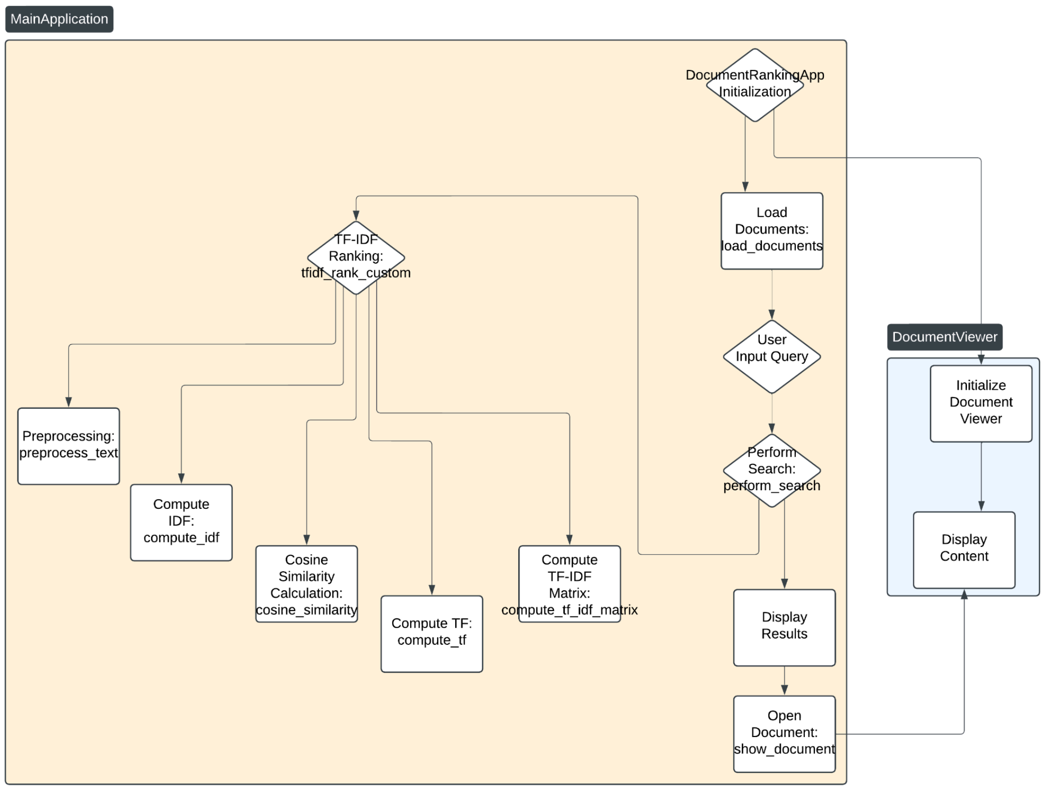
- Cosine similarity measures the angle between the arrows:
- Smaller angle = more similar.



*Current dataset consists of data from 10 newsgroups each containing 100 files. Resource(Kaggle): (10)Dataset Text Document Classification

2.4. Design and Workflow

- User runs the app, input a query and clicks the "Search" button.
- The query is preprocessed, the documents are retrieved after passing through TF-IDF calculations.
- The results undergo a cosine similarity process with respect to the query.
- The finalized results are displayed to the user


MainApplication
DocumentRankingApp Initialization
Load Documents: load_documents
User Input Query
Perform Search: perform_search
Display Results
Open Document: show_document
TF-IDF Ranking: tfidf_rank_custom
Preprocessing: preprocess_text
Compute IDF: compute_idf
Cosine Similarity Calculation: cosine_similarity
Compute TF: compute_tf
Compute TF-IDF Matrix: compute_tf_idf_matrix
DocumentViewer
Initialize Document Viewer
Display Content

3. Project 3: Probabilistic, Non-Overlapped and Proximal Nodes Model

3.1. Goals of the system

“The project is a Document Retrieving System that helps users find the most relevant documents from a collection based on their queries, using different retrieval models.”

- Allow users to query data even if the query does not exactly match.
- Process documents to extract important terms.
- Allow users to retrieve documents by selecting a specific retrieval model.
- Provide a user-friendly interface for query input and result display.

3.2. Proposed Solution

3.2.1. Query Handling

Match incomplete queries with existing terms and show relevant documents to the user.

3.2.2. Process Documents

Process the documents and query to extract only important terms so searching is made easier.

3.2.3. Document Retrieval

Implement all the three aforementioned models so users can select and see the results for that specific model.

3.3. Implementation

3.3.1. Data Gathering/Selection

A folder “data” is a dedicated directory where all the documents are to be collected, our application will look for documents and subdirectories in this folder and create indexes for all documents.

3.3.2. Development

We used Python for coding and VS Code as the code editor.
Python is like a Swiss army knife for programming – it has tools (libraries) for almost anything you need.
Think of VS Code as a comfortable desk where you can organize and write your work efficiently.

3.3.3. Preprocessing the query and documents

Think of it as cleaning up a messy room before you can find the toys you want to play with. Preprocessing helps clean up the text in documents and queries, so we can find the best answers more easily.

Example:

Original Sentence:
"I love space exploration and space missions."

Step 1: Tokenization
Split into words: ["I", "love", "space", "exploration", "and", "space", "missions"]
Step 2: Removing Stop Words
Remove words like "I", "and": ["love", "space", "exploration", "space", "missions"]
Step 3: Lowercasing
Convert everything to lowercase: ["love", "space", "exploration", "space", "missions"]
Step 4: Stemming (remove prefixes/suffixes to return a world to base level)
Change "missions" to "mission": ["love", "space", "exploration", "space", "mission"]

Why Do We Clean the Text?

- Clean Text = Better Results
  Without cleaning the text, we might look at "Space exploration" and "space exploration" as two different things. But after cleaning, they are treated the same, helping us find the best documents.
- Faster and Easier Comparisons
  With fewer unnecessary words, it's faster to find the books that answer your question.

3.3.4. Document Retrieval Models

Imagine you have a giant library with thousands of books. When you ask a question, these models help find the best books that answer your question.

We'll learn about three models that help us find the best books:

- Probabilistic Retrieval (BIM)
- Non-Overlapped List
- Proximal Nodes

1. Probabilistic Retrieval (Binary Independence Model):
Imagine you're playing a game where you get a point if you find the right answer (1) and no points (0) if you don't. BIM uses this idea to figure out which documents are relevant based on whether certain words are present (or not).

Steps in BIM:

1. Preprocessing:
   Clean the words in the books and your question (like removing "is" or "the").
2. Binary Weights:
   Words are either "there" (1) or "not there" (0). No in-between!
3. Query Representation:
   You ask a question, and we turn it into a list of 1s and 0s based on what words are in the library.
4. Document Scoring:
   We score each book based on how many matching 1s (words) it has with your question.

5. Ranking:
   Books with the most matches get ranked higher.
6. Retrieve Top-K Documents:
   We show you the best books (top K) with the most matches.

Formula used in BIM for Document Scoring:
Jaccard Score = Number of matching words / Total unique words in both the query and document
- Matching words: Count of words that appear in both the document and query.
- Total unique words: All the unique words in both the document and query combined.

Example:
Query: "machine learning"
Document 1: "Introduction to machine learning techniques"
Document 2: "The history of space exploration"

Step 1: Preprocessing
Clean words in documents and query.

Step 2: Binary Weights
Query: machine(1), learning(1)
Document 1: machine(1), learning(1)
Document 2: machine(0), learning(0)

Step 3: Document Scoring
Document 1: 2 matches
Document 2: 0 matches

Step 4: Ranking
Document 1 is more relevant to the query, so it gets a higher rank.

2. Non-Overlapping Lists Model:
Think of it as collecting all the books that talk about certain words or topics and combining them into one big list without repeating any book.

Steps in Non-Overlapped List explained with an example:
Terms: "machine learning" and "data visualization"
Document 1: "machine learning techniques"
Document 2: "data visualization in Python"
Document 3: "introduction to programming"

1. Identify Terms of Interest:
   Choose words you care about, like “machine learning” and “data visualization”.
2. Retrieve Documents per Term:
   Find documents that have "machine learning" and books that have "data visualization" and add them in their own lists respectively.
3. Combine Lists for Non-Overlapping Results:

Put all the documents from both lists together but don’t repeat any.
4. Present Results:
   This combined list of results only contains documents containing either of the terms.

3. Proximal Nodes Model
Imagine a network where words or topics are connected. Proximal nodes are closely related words or concepts. When you ask a question, we look for words connected to your topic and find relevant books based on that.
Steps in Proximal Nodes Model:
1. Define Proximal Nodes:
   Identify related topics to your query.
2. Explore Network Relationships:
   Think of words as points connected to each other. We find books connected to your main topic.
3. Retrieve Connected Documents:
   Books that are connected to the related topics are considered relevant.
4. Present Results:
   Show the documents connected to the nodes (related topics).

Example:
Query: "space exploration"
Proximal Nodes: "NASA", "astronauts", "moon mission"

Step 1: Define Proximal Nodes
We choose "NASA", "astronauts", and "moon mission" as related topics to "space exploration".

Step 2: Explore Network
Documents connected to "NASA" and "astronauts" are relevant.

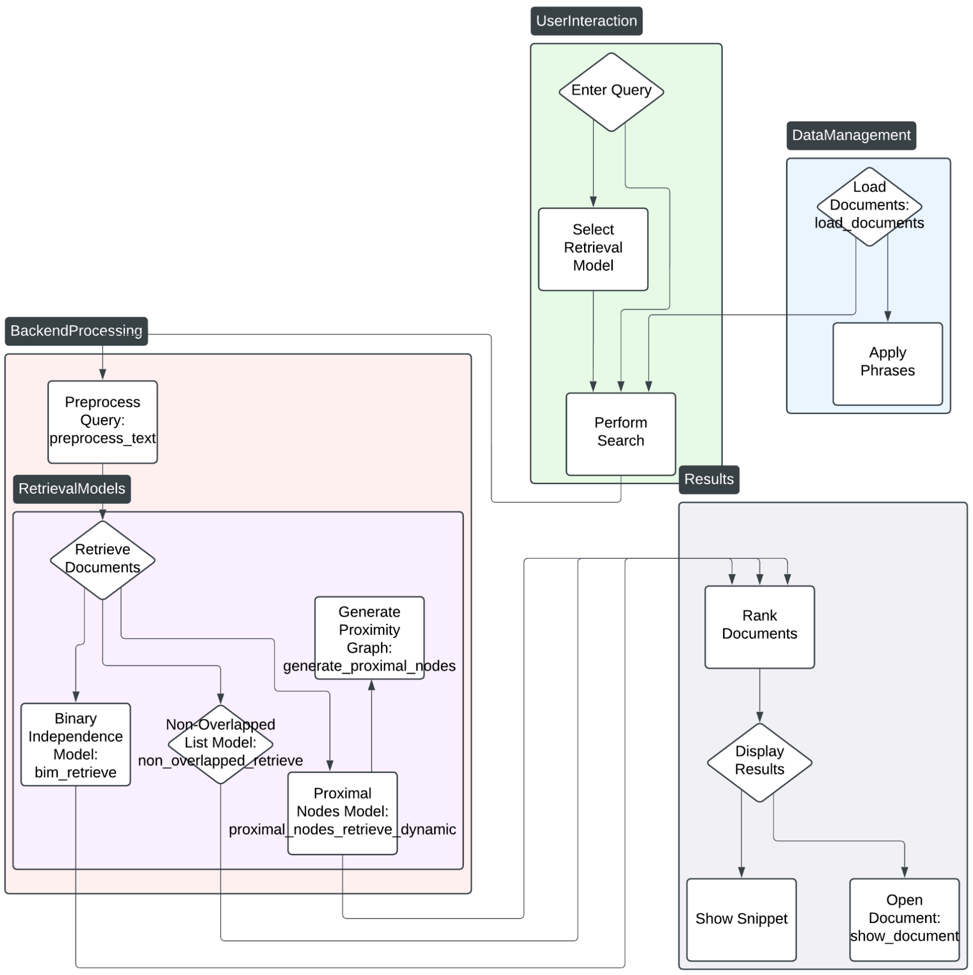
Step 3: Retrieve Connected Documents
Retrieve documents that talk about "NASA" and "astronauts".

Step 4: Present Results
Show the documents related to "NASA" and "astronauts".

*Current dataset consists of data from 10 newsgroups each containing 100 files. Resource(Kaggle): (10)Dataset Text Document Classification

3.4. Design and Workflow

- User enters the query, selects the retrieval model and clicks the search button.
- The dataset is loaded into the memory.
- The query is preprocessed along with the documents, and the selected model is run.
- The finalized results are ranked and then displayed to the user.


UserInteraction
Enter Query
Select Retrieval Model
Perform Search
DataManagement
Load Documents: load_documents
Apply Phrases
BackendProcessing
Preprocess Query: preprocess_text
RetrievalModels
Retrieve Documents
Generate Proximity Graph: generate_proximal_nodes
Binary Independence Model: bim_retrieve
Non-Overlapped List Model: non_overlapped_retrieve
Proximal Nodes Model: proximal_nodes_retrieve_dynamic
Results
Rank Documents
Display Results
Show Snippet
Open Document: show_document

4. Project 4: Structure Guided Browsing and Hypertext Model

4.1. Goals of the system

"Create an application that allows users to interact with a dataset through Structure Guided Browsing and can also navigate using Hypertext Model."

- Allow users to interact with the dataset in a structure guided environment that includes sections and planes.
- Allow users to navigate between different documents based upon links that are created using Hypertext Model.

4.2. Implementation

4.2.1. Choose field of application

We have a dataset that contains information from 10 different newsgroups, we will be using this dataset to implement our Guided Structure and Hypertext models.

4.2.2. Development

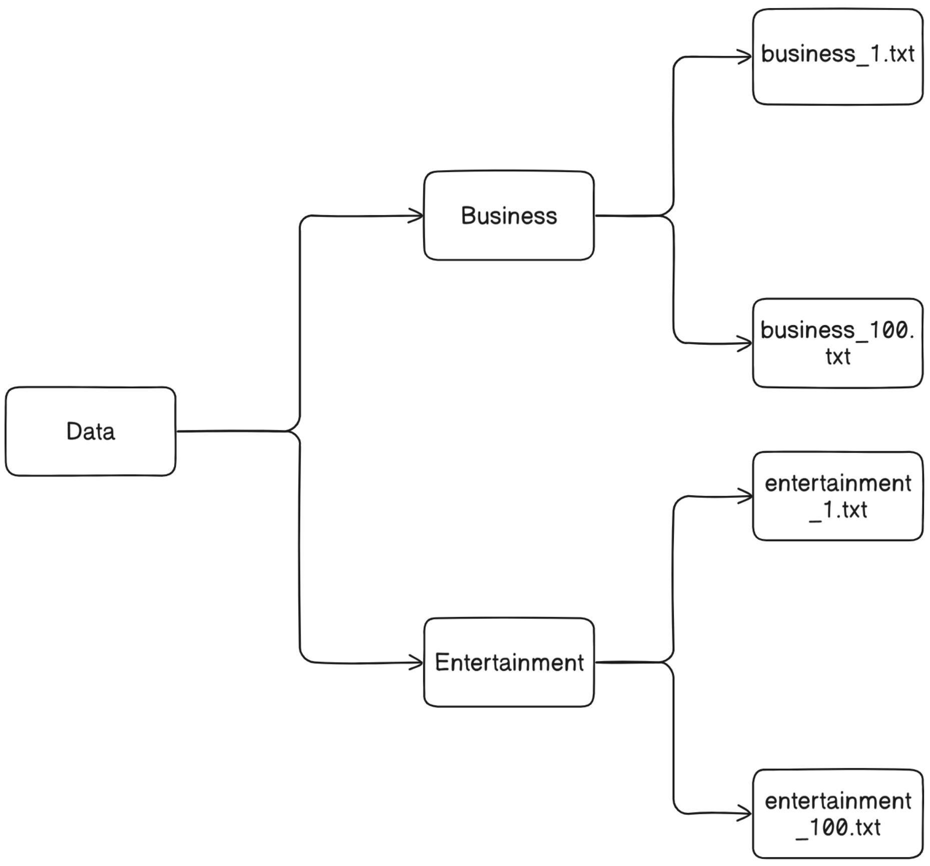
We used Python for coding and VS Code as the code editor.
Python is like a Swiss army knife for programming – it has tools (libraries) for almost anything you need.
Think of VS Code as a comfortable desk where you can organize and write your work efficiently.

4.2.3. Content Hierarchy

We have divided the data files from each news group in a separate folder grouping specific files together.

4.2.4. Visual Map

Visual representation of this structure is as such:



4.2.5. Assign Content Links

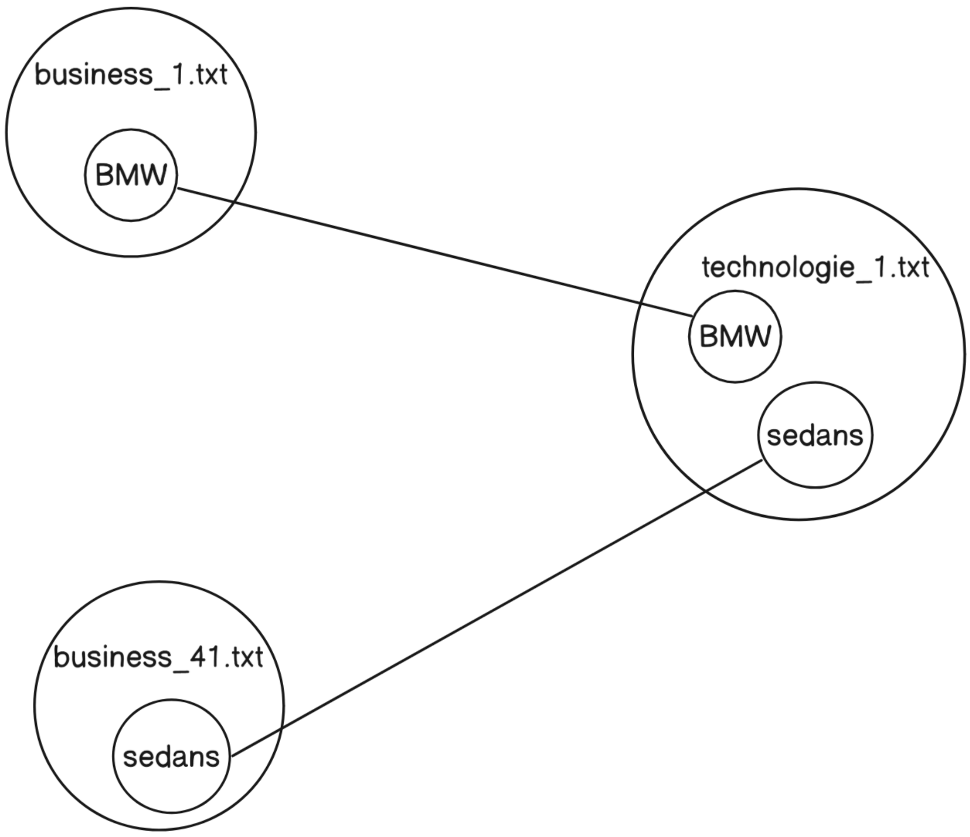
We assign links between documents to navigate through them. This part is handled in Hypertext model implementation.

User Interactivity Components
We add collapsible sections and the content in a selection pane where users can easily view and interact with all files in the dataset.

4.2.6. Hypertext Model

Define Nodes:
We use words that are occuring between only two documents at the moment as linking nodes. These words are most probably nouns and allow users to see in which other documents and where they are included again.

Create Links:
We create links between these words which allow us to navigate to the location where they are used again in the whole dataset.



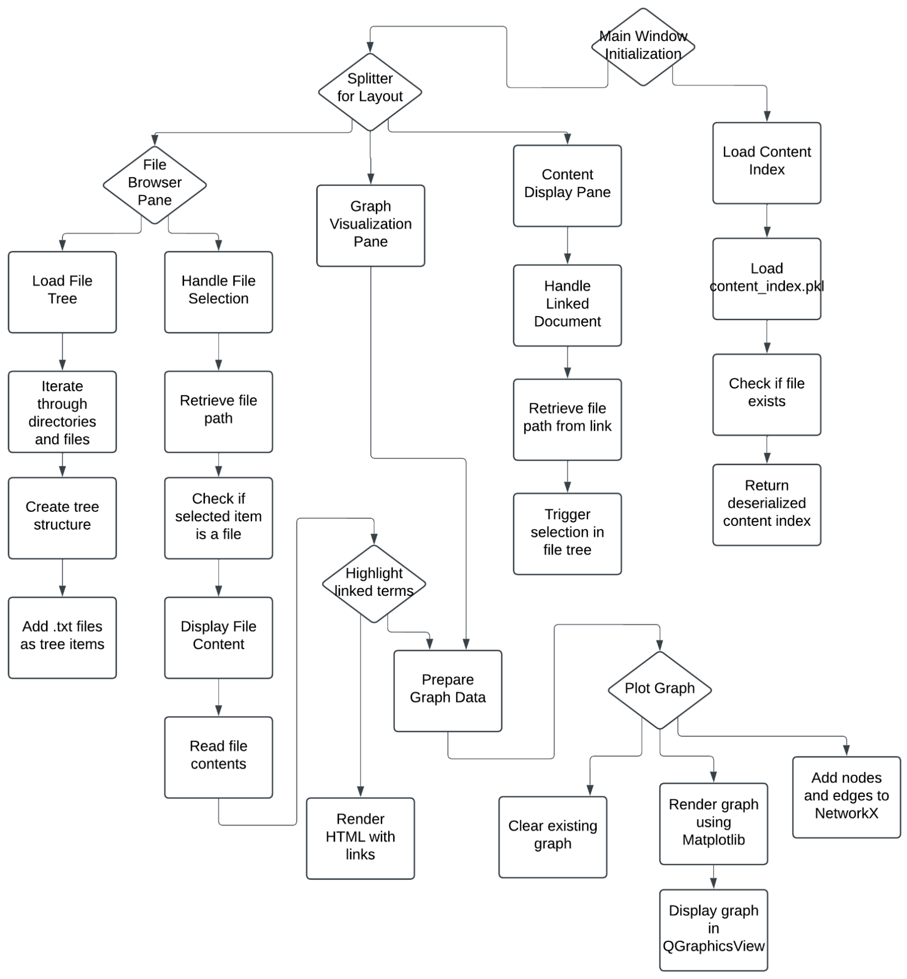
Implement Hyperlinks
We use python HyperText Markup Language function from pyQT which allows us to create hyperlinks between documents on the selected words.

User Interaction
The Hyperlinks created are shown in the universal blue underline depicting that they are urls, when user clicks on them they are redirected to the connected document.

*Current dataset consists of data from 10 newsgroups each containing 100 files. Resource(Kaggle): (10)Dataset Text Document Classification

4.4. Design and Workflow

- The app sets up a GUI with a file browser, content display, and graph visualization panes.
- Files and directories are loaded into a tree view, allowing users to browse .txt files.
- Clicking a file shows its content, highlighting terms linked to other files, and enabling navigation via hyperlinks.
- Linked terms are plotted as a graph, showing relationships between documents and terms dynamically.


Main Window Initialization
Splitter for Layout
Load Content Index
Load content_index.pkl
Check if file exists
Return deserialized content index
File Browser Pane
Graph Visualization Pane
Content Display Pane
Handle Linked Document
Retrieve file path from link
Trigger selection in file tree
Load File Tree
Iterate through directories and files
Create tree structure
Add .txt files as tree items
Handle File Selection
Retrieve file path
Check if selected item is a file
Display File Content
Read file contents
Highlight linked terms
Render HTML with links
Prepare Graph Data
Plot Graph
Clear existing graph
Render graph using Matplotlib
Display graph in QGraphicsView
Add nodes and edges to NetworkX

5. Project 5: Set Theoretic IR Models

5.1. Goals of the system

“The project is to create a search engine that uses 1 of the 4 Set Theoretic IR models; Fuzzy, Boolean Extended, Generalized Vector, or Latent Semantic Indexing.”

- To implement the Generalized Vector Model (GVM) for ranking documents based on relevance to a user query.
- To calculate relevance scores using cosine similarity between vector representations of documents and queries.
- To provide an interactive application for query-based document retrieval.

5.2. Proposed Solution

5.2.1. Query Handling

Match incomplete queries with existing terms and show relevant documents to the user.

5.2.2. Document Retrieval

Use a vector space representation for documents and queries, with term weights computed using TF-IDF.

5.2.3. Document Ranking

Rank documents based on cosine similarity between query and document vectors.

5.3. Implementation

5.3.1. Data Gathering/Selection

A folder “data” is a dedicated directory where all the documents are to be collected, our application will look for documents and subdirectories in this folder and create indexes for all documents.

5.3.2. Choose IR Model

We'll use the Generalized Vector Model, as it provides robust functionality with term weighting and cosine similarity for ranking results.

5.3.3. Development

We used Python for coding and VS Code as the code editor.
Python is like a Swiss army knife for programming – it has tools (libraries) for almost anything you need.
Think of VS Code as a comfortable desk where you can organize and write your work efficiently.

5.3.4. Generalized Vector Model

Imagine each document and query is like a treasure chest, and each word is a gem. The Generalized Vector Model (GVM) helps us:

Put gems in order:
Turn words into numbers based on how important they are.
Example: In a document about "apples," the word "apple" is a bigger gem than "the."

Measure similarity:
Compare how many similar gems two chests (document and query) have.
Example: If the query chest has "apple, fruit," and the document chest has "apple, orange," they're a good match!

Find the best match:
The closer the match, the more relevant the document.
Search Criteria:
How do we find matching documents?

- Turn the query into a treasure chest:
  - Example: If someone asks, "Where are apples grown?" we keep important words like "apples" and "grown."
- Compare chests:
  - Use the math (TF-IDF and cosine similarity) to see which document treasure chest matches the query chest best.
- Show the best results first:
  - Example: If one document talks about apple farms and another talks about apple phones, the first one is a better match.

Representation Scheme:
How do we turn words into numbers?

- TF (Term Frequency):
  - Count how many times a word appears in a document.
  - Example: In "Apples are apples," the word "apples" has a frequency of 2.
- IDF (Inverse Document Frequency):
  - Words that appear everywhere (like "the") are less important.
  - Example: If "apple" is in 2 of 10 documents, it's more special than "is," which is in all 10.
- TF-IDF:
  - Combine TF and IDF to get a score for each word.
  - Example: "Apple" in a document about farming gets a higher score than in a generic article.

Query Processing
How do we process a question?

- Break the query into words:
  - Example: The query "Where are apples grown?" becomes ["where," "apples," "grown"].
- Calculate the query's TF-IDF:
  - Assign importance to each word based on how rare it is in the dataset.

- Compare with each document:
    - Use cosine similarity to see which document matches best.
- Real-life Example:
  If the query is "How to grow apples," and the documents are:
    - "Apple farming techniques."
    - "Top 10 apple recipes."

  The system will rank #1 higher because it's about growing apples.

Ranking Algorithm

How do we rank documents?

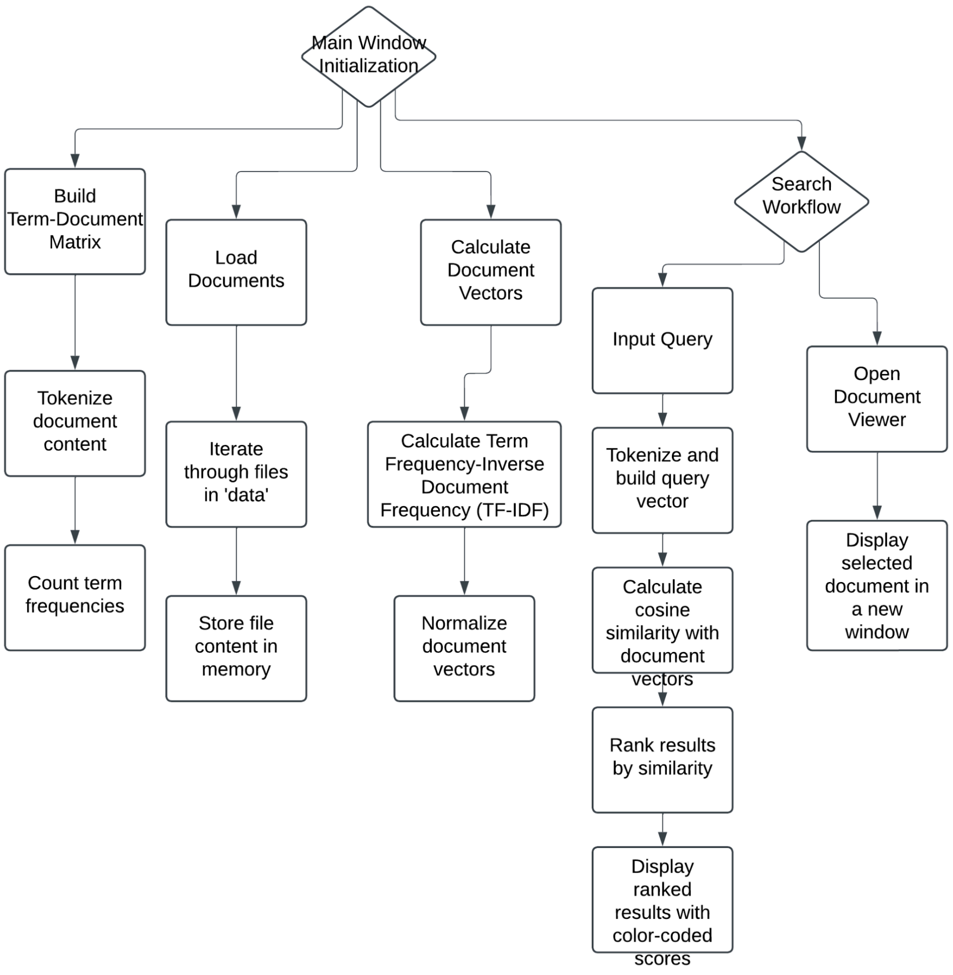
1. Cosine similarity:
    a. Think of it like measuring the angle between two arrows (one for the query, one for the document).
    b. Smaller angle = better match.
2. Sort by scores:
    a. Example: If Document A has a score of 0.9 and Document B has 0.7, show A first.

   Example Output:
   Query: "apple farming"
    b. Document A: "Apple farming techniques" (Score: 0.9)
    c. Document B: "Best apple varieties" (Score: 0.7)

*Current dataset consists of data from 10 newsgroups each containing 100 files. Resource(Kaggle): (10)Dataset Text Document Classification

5.4. Design and Workflow

- Loads documents and builds a term-document matrix and document vectors for search operations.
- Loads documents and builds a term-document matrix and document vectors for search operations.
- Ranks and displays results with color-coded similarity scores, along with document snippets.
- Allows users to open and view the full content of selected documents in a new window.


Main Window Initialization
Build Term-Document Matrix
Tokenize document content
Count term frequencies
Load Documents
Iterate through files in 'data'
Store file content in memory
Calculate Document Vectors
Calculate Term Frequency-Inverse Document Frequency (TF-IDF)
Normalize document vectors
Search Workflow
Input Query
Tokenize and build query vector
Calculate cosine similarity with document vectors
Rank results by similarity
Display ranked results with color-coded scores
Open Document Viewer
Display selected document in a new window

Project 6: Neural Network IR Model

6.1. Goals of the search engine

“Build an application that uses semantic understanding to improve search accuracy, making it easier for users to find relevant articles.”

- Expands queries using related terms for better search results.
- Ranks articles based on relevance using similarity scores.
- Provides an easy-to-use interface for searching and viewing articles.

6.2. Implementation

Neural Network IR Model:
This system acts like a helpful librarian who doesn’t just look for the exact words you say but understands what you mean.

Example:
Imagine you want a book about "cats."
Instead of looking only for "cats," the librarian also checks for books with "kittens" or "felines."
They rank books that mention "cats" a lot higher than those that mention it once.

Why is this smart?
Because it understands that words like "kitten" and "feline" mean the same thing as "cat"!

6.2.1. Data Preparation

- Dataset:
  - Articles: A collection of 5 short articles about health, exercise, and fitness.
- Example:
  - "Article 1": "Exercise improves physical fitness and promotes well-being."
  - "Article 2": "Benefits of regular workouts include better health and increased energy levels."
- Vocabulary:
  - A predefined dictionary linking terms to related words.
- Example:
  - "exercise" → ["workouts", "physical activity", "training"]
  - "health" → ["wellness", "well-being", "fitness"]

6.2.2. Neural Network Model

Tokenization:
“ Splits text into words and converts to lowercase. ”
Example:
Input: "Benefits of exercise" → Output: ["benefits", "of", "exercise"]

Semantic Expansion:
“ Expands query with related terms from the vocabulary. “

Example:
Query: ["exercise", "health"] → Expanded Query: ["exercise", "health", "workouts", "fitness", "training", "well-being"]

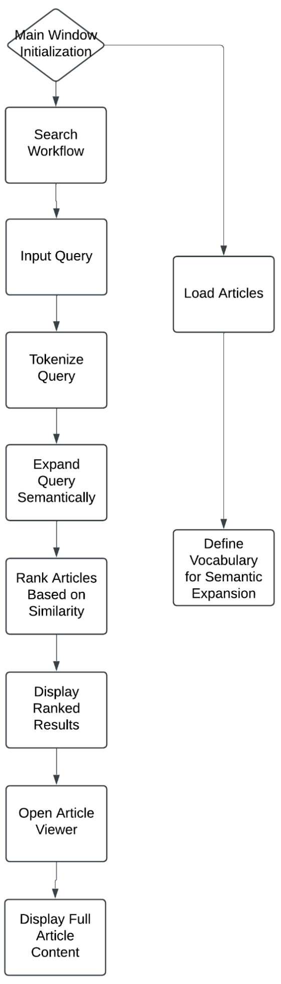
Similarity Calculation:
“ Finds overlap between the expanded query and each article. “
Example:
Query: ["exercise", "workouts"]
Article: "Exercise improves fitness through regular workouts."
Shared Terms: ["exercise", "workouts"] → High similarity.

Ranking and display
“ Articles are ranked by similarity scores and displayed with snippets. ”
Example:
Query: "exercise benefits"
Results:
Article 2: "Benefits of regular workouts" (Score: 0.9)
Article 1: "Exercise improves physical fitness" (Score: 0.8)

*Current dataset consists of data from 10 newsgroups each containing 100 files. Resource(Kaggle): (10)Dataset Text Document Classification

6.3. Design and Workflow

- Loads predefined articles and vocabulary for semantic expansion of queries.
- Tokenizes the query, expands it semantically, and calculates similarity scores between the query and articles.
- Displays ranked articles with similarity scores and snippets.
- Opens a new window to display the full content of a selected article.


Main Window Initialization
Search Workflow
Input Query
Tokenize Query
Expand Query Semantically
Rank Articles Based on Similarity
Display Ranked Results
Open Article Viewer
Display Full Article Content
Load Articles
Define Vocabulary for Semantic Expansion

7. Project 7: Interference and Belief Network IR Models

7.1. Goals of the system

“Create an application that demonstrates implementation and working of interference and belief network models.”

- Build two models (Interference Model and Belief Network) to rank documents based on a query.
- Understand how probabilities help decide which documents are most relevant.

7.2. Implementation

7.2.1. Interference Model

Imagine you are looking for a book in a library:

- Assign probabilities:
    - Each book (document) gets a score based on how likely it matches your query.
- Match query and books:
    - If your query is "cats," books with "cats" in their title or content will get higher scores.
- Sort the matches:
    - Books with higher scores are more relevant and appear at the top of the list. Example: Query: "cats"
    - Book 1: "Cats and their habits" (Score: 0.9)
    - Book 2: "How to care for cats" (Score: 0.8)
    - Book 3: "Dogs and their habits" (Score: 0.1)

7.2.2. Belief Network Model

A Belief Network is like solving a mystery by piecing together clues:

- Nodes: Represent things we care about (e.g., documents, queries, features).
    - Example: "Does this document talk about cats?"
- Edges: Show relationships between nodes.
    - Example: "If a document mentions cats, it’s likely about animals."
- Use probabilities:
    - Combine clues to decide if a document matches the query.
- Example:
    - Query: "space exploration"
    - Evidence: Document mentions "rocket," "NASA," and "Mars."
    - Belief: High chance this document is about space exploration.

7.2.3. Data Preparation

What we are working with:
Documents:

- A collection of 20 short text files about topics like animals, space, fitness, and more.
    - Example: "Cats prefer quiet places to sleep and enjoy warm sunlight on a lazy afternoon."
- Queries:
    - Simple questions or keywords, such as "cat," "dog," "sunlight," and "space exploration."
- Relevance Judgments:
    - Predefined scores indicating which documents are good matches for each query.
- Example:
    - Query: "cat"
    - Relevant Documents: doc1, doc3.

7.2.4. Interference Model

Assign Probabilities:

- Calculate how likely each query is to match each document.
- Formula: $P(\text{Query} \mid \text{Document}) = P(\text{Query}) \cdot P(\text{Document})$ $\quad P :=$ Probability

Sort Documents:

- Higher probabilities mean a better match.

Example: Query: "cat"
Document 1: $P=0.4$
Document 3: $P=0.3$
Document 2: $P=0.1$

Output: doc1 > doc3 > doc2.

7.2.5. Belief Network Model

Set up the network:

- Nodes: Query, documents, features (e.g., words like "Mars" or "planets").
- Edges: Relationships between nodes.

Calculate probabilities:

- Use Bayes' theorem to find $P(\text{Relevance} \mid \text{Query})$
- $P(\text{Relevance} \mid \text{Query}) = P(\text{Query} \mid \text{Relevance}) \cdot P(\text{Relevance}) / P(\text{Query})$

Example: Query: "space exploration"

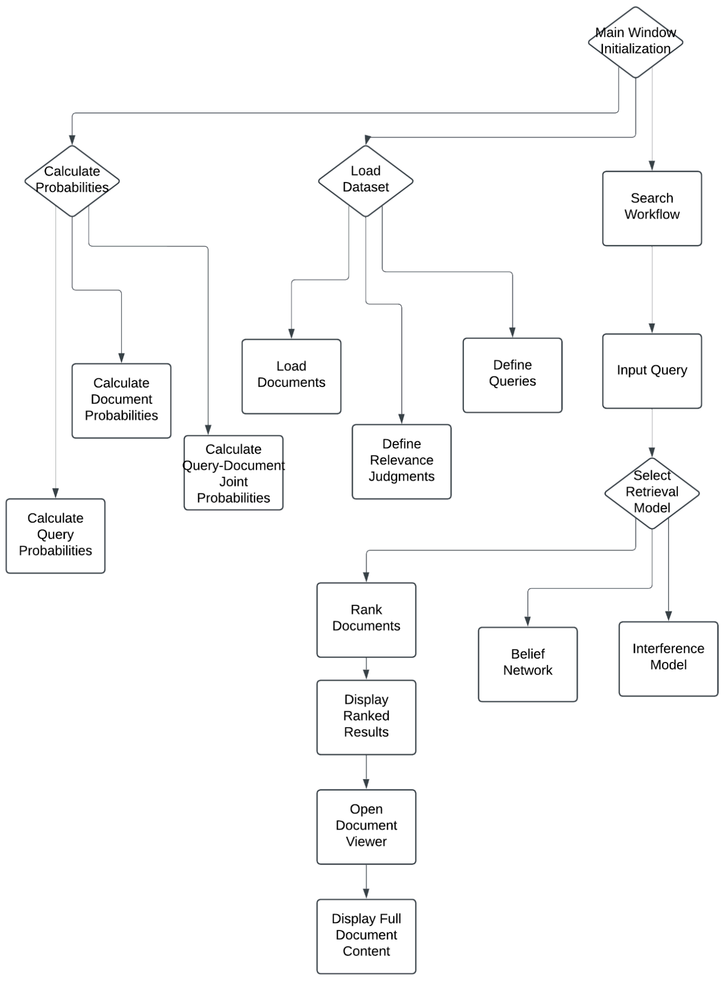
- Evidence: Document mentions "rockets" and "NASA."
- Belief: $P=0.9$, so this document is likely relevant.

*Current dataset consists of data from 10 newsgroups each containing 100 files. Resource(Kaggle): (10)Dataset Text Document Classification

7.3. Design and Workflow

- Loads documents, queries, and relevance judgments into memory and calculates query, document, and joint probabilities.Loads documents, queries, and relevance judgments into memory and calculates query, document, and joint probabilities.
- Accepts a query, allows model selection (Interference Model or Belief Network), and ranks documents based on relevance scores.

- Displays ranked results with snippets, scores, and links to full document content.
- Opens a new window to show the full content of a selected document.


Main Window Initialization
Calculate Probabilities
Load Dataset
Search Workflow
Calculate Document Probabilities
Calculate Query-Document Joint Probabilities
Calculate Query Probabilities
Load Documents
Define Relevance Judgments
Define Queries
Input Query
Select Retrieval Model
Rank Documents
Belief Network
Interference Model
Display Ranked Results
Open Document Viewer
Display Full Document Content