# Vector Space Model

**Dilawer Khan Cs_06**

**Bilal Hassan Cs_37**

## Table of Content:

## Contents

## Introduction:

The Vector Space Model (VSM) is a popular information retrieval technique used to represent and rank documents based on their relevance to a given query. It provides a foundation for building search engines and recommendation systems by calculating the similarity between documents and queries. By employing the VSM, we can convert textual information into a numerical representation that captures the semantic meaning and importance of terms within documents. This enables efficient comparison and ranking of documents according to their similarity to user queries, leading to improved search accuracy and user satisfaction. By this implementation, the VSM determines the similarity between documents and queries, enabling effective ranking and retrieval of relevant information.

## Aim:

- The project's objective is to create a document evaluation system based on the vector space model. Users will be able to search through a collection of text documents using the system, and the results will be ranked according to how closely the papers match their search terms. By implementing the VSM, the project aims to significantly reduce the time and effort required for users to find the most relevant documents. The system will swiftly process user queries and rank documents based on their similarity to the query, ensuring a more efficient and responsive search experience. The project aims to enhance information retrieval by delivering more precise and meaningful search results by utilizing the VSM's principles.

Key objectives of the project include:

- Building a graphical user interface (GUI) to facilitate user interaction and query input.
- Implementing the VSM algorithm to calculate TF-IDF values for terms and documents.
- Computing cosine similarity between queries and documents to determine their relevance.
- Displaying the ranked results to the user in an intuitive and user-friendly manner.
- Allowing users to view the content of the ranked documents for further exploration and analysis.

By achieving these objectives, the project aims to provide an efficient and effective document retrieval system that can be applied in various domains, such as information retrieval, document search, recommendation systems, and more.

## Vector Space Model (VSM)

The algorithm implemented in this project is the Vector Space Model (VSM). The VSM is a widely used and effective algorithm for document ranking and retrieval in information retrieval systems. It offers a flexible and efficient approach to represent and compare documents and queries based on their semantic content.

### Why VSM:

The Vector Space Model (VSM) is chosen as the algorithm for the document ranking system due to the following reasons:

- The VSM is relatively simple to implement and computationally efficient, making it suitable for processing large document collections and providing real-time ranking results.
- The VSM can efficiently handle large-scale document collections, making it suitable for systems dealing with a significant amount of textual data.
- The VSM produces a natural ranking based on the cosine similarity between vectors, allowing users to quickly identify the most relevant documents.
- The VSM is specifically designed for handling textual data, capturing the importance of words and terms within documents, and facilitating tasks such as document retrieval and search engines.
- The VSM allows for the inclusion of various features and attributes, enabling the representation and comparison of documents and queries in a versatile manner.

## Psuedo Code:

```
Function Main_Function():
    directory = "D:\Aoa Project final"
    Content_Dictionary = {} // Dictionary to store document content
    files_name_list = []

    For each filename in directory:
        If filename ends with ".txt":
            file_path = join(directory, filename)
            content = read_file(file_path)
            file_name = remove_extension(filename)
            Content_Dictionary[file_name] = content
            files_name_list.append(file_name)
        End If
    End For

    user_query = get_user_query()
    clear_text_widget()

    d = length(files_name_list)
    str = ""
    content_lst = []

    For each file_name in files_name_list:
        str = concatenate(str, Content_Dictionary[file_name])
        content_lst.append(Content_Dictionary[file_name])
    End For

    words = split(str, " ")
    wordslst = []

    For each word in words:
        If word not equal to ‘.’:
            wordslst.append(word)
        End If
    End For

    wordslst = set(wordslst)
    wordslst = sort(wordslst)
```

```
List_2D = create_2D_list(length(wordslst), length(content_lst) * 2 + 5)

For i = 0 to length(wordslst) - 1:
    If wordslst[i] is in user_query:
        List_2D[i][0] = 1
    End If

    For j = 0 to length(content_lst) - 1:
        newstr = split(content_lst[j], " ")

        For each k in newstr:
            If wordslst[i] is equal to k:
                List_2D[i][j + 1] += 1

                If List_2D[i][j + 1] > 1:
                    List_2D[i][d + 1] += -1
                End If

                List_2D[i][d + 1] += 1
                List_2D[i][d + 2] = d / List_2D[i][d + 1]
                List_2D[i][d + 3] = logarithm(List_2D[i][d + 2], 10)
                d1 = d + 4
            End If
        End For
    End For
End For

For l = 0 to length(content_lst):
    List_2D[i][d1 + l] = List_2D[i][l] * List_2D[i][d + 3]
End For

// Calculation of Cosine Similarity
Doc_name = []
Doc_ranking = []

For i = 0 to length(content_lst) - 1:
    num2 = 0

    For j = 0 to length(wordslst) - 1:
        num2 += List_2D[j][d1] * List_2D[j][d1 + i + 1]
    End For

    Doc_name.append(files_name_list[i])

    If num2 == 0:
        Doc_ranking.append(num2)
    Else:
        num3 = lstr[0] * lstr[i + 1]
        Doc_ranking.append(num2 / num3)
    End If
End For

// Sorting Document Rankings
sorted_indices = sort_indices(Doc_ranking, descending=True)
Doc_ranking1 = get_elements_by_indices(Doc_ranking, sorted_indices)
Doc_name1 = get_elements_by_indices(Doc_name, sorted_indices)
```

```
    Display_Rankings(Doc_name1, Doc_ranking1)

End Function

// Function to calculate Cosine Similarity
Function cosine_similarity(vector1, vector2):
    dot_product = dot_product(vector1, vector2)
    magnitude1 = calculate_magnitude(vector1)
    magnitude2 = calculate_magnitude(vector2)

    similarity = dot_product / (magnitude1 * magnitude2)
    Return similarity
End Function

window = create_GUI_window()
entry_query = create_input_field()
calculate_button = create_button("Search Document", Main_Function)
text_widget = create_text_widget(window)
window.mainloop()
```

## Algorithm Design:

### Read Document Files:

This step involves reading the text documents from a specified directory. It ensures that the system can access and retrieve the content of each document for further processing. By storing the content in a dictionary with file names as keys, it allows for easy retrieval and association between documents and their content.

### Preprocess Documents:

Preprocessing the document content is crucial for standardization and consistency. By removing punctuation, converting text to lowercase, and handling stop words (if necessary), the system prepares the documents for further analysis. Preprocessing helps eliminate noise and reduces the impact of irrelevant or common words on the ranking process.

### Create Vocabulary:

Creating a vocabulary involves extracting unique words from preprocessed documents. Sorting the words alphabetically ensures consistency and facilitates efficient indexing and searching. The vocabulary serves as the foundation for representing and comparing documents based on their shared words.

### Calculate Term Frequencies:

The calculation of term frequencies involves counting the occurrence of each word in the vocabulary within each document. By storing these frequencies in a 2D list or matrix, the system establishes a numerical representation of the documents, where each element represents the importance of a word in a specific document.

### Calculate Document Frequencies (DF):

The document frequencies indicate how many documents contain a particular word from the vocabulary. By counting the number of documents that include each word, the system obtains the document frequency (DF) values. This information helps assess the significance and uniqueness of words across the document collection.

## Calculate Inverse Document Frequencies (IDF):

The inverse document frequencies (IDF) measure the rarity and importance of words in the document collection. Calculated using the logarithmic function, IDF values assign higher weights to words that appear less frequently across documents. This emphasizes the uniqueness and discriminative power of such words in the ranking process.

## Calculate TF-IDF Weights:

By multiplying the term frequencies with their respective IDF values, the system computes the TF-IDF weights. This process highlights words that are frequent in a particular document but rare across the collection. The resulting weights represent the importance of words within each document and serve as the basis for ranking.

## Process User Query:

The system allows users to input queries for document retrieval or ranking. Like document preprocessing, the query undergoes the same text normalization steps, removing punctuation and converting it to lowercase. This ensures consistency and compatibility between the user query and the document representation.

## Calculate Query Vector:

Converting the preprocessed query into a vector representation allows for direct comparison with document vectors. The query vector is constructed using the vocabulary and IDF values. It sets the vector components to 1 if the corresponding word is present in the query, and 0 otherwise. This binary representation captures the relevance of words in the query.

## Calculate Cosine Similarity:

The cosine similarity measure determines the similarity between the query vector and each document vector. Computed using the dot product and vector norms, cosine similarity captures the angle between the two vectors and indicates their similarity. Higher cosine similarity values imply greater relevance and rank documents accordingly.

## Rank Documents:

Sorting the documents based on their cosine similarity scores enables the system to present the most relevant documents to the user. By arranging the documents in descending order, the system emphasizes the highest-ranking documents that closely match the user query. This facilitates efficient retrieval and improves the user experience.

## Display Results:

Presenting the ranked list of documents and their cosine similarity scores to the user is the final step. This display enables users to quickly identify and access the most relevant documents based on their search query. Providing options to view the content of each document enhances the user experience and enables further exploration.

## Algorithm Analysis:

### Time Complexity:

The total time complexity of the document ranking algorithm based on the Vector Space Model (VSM) can be estimated as follows:

Reading Document Files: O(n * k) - where n is the number of documents and k is the average length of the documents.

Preprocessing Documents: O(n * k) - similar to the reading step, as it operates on each document individually.

Creating Vocabulary: O(n * k) - again, iterating through each document to extract unique words.

Calculating Term Frequencies: O(n * m) - where m is the number of unique words in the vocabulary.

Calculating Document Frequencies: O(n * m) - similar to the term frequency calculation, as it iterates through the documents and vocabulary.

Calculating IDF Values: O(m) - as it involves a single pass through the unique words in the vocabulary.

Calculating TF-IDF Weights: O(n * m) - similar to the previous steps, as it operates on each document and the vocabulary.

Processing User Query: O(k) - where k is the length of the query.

Calculating Query Vector: O(m) - as it involves a single pass through the vocabulary.

Calculating Cosine Similarity: O(n * m) - similar to the previous steps, as it operates on each document and the vocabulary.

Ranking Documents: O(n * log(n)) - as it involves sorting the documents based on cosine similarity scores.

Displaying Results: O(n) - as it requires iterating through the ranked documents to present them to the user.

Therefore, the total time complexity can be approximated as O (n * m) + O(n * log(n)) + O(k), where n represents the number of documents, m represents the number of unique words in the vocabulary, and k represents the length of the query.

### Space Complexity:

Storing Document Content: O(n * k) - where n is the number of documents and k is the average length of the documents.

Creating Vocabulary: O(m) - where m is the number of unique words in the vocabulary.

Storing Term Frequencies, Document Frequencies, IDF Values, and TF-IDF Weights: O(n * m) - using a 2D list or matrix to represent the relationship between words and documents.

Storing Query Vector and Cosine Similarity Scores: O(n) - as it requires storing the results for each document.

The space complexity is mainly influenced by the number of documents (n), the number of unique words in the vocabulary (m), and the average length of the documents (k).

To summarize, the space complexity for the VSM-based document ranking algorithm is O(n * k) + O(m) + O(n * m) + O(n), where n represents the number of documents, m represents the number of unique words in the vocabulary, and k represents the average length of the documents.

## Limitation:

The algorithm assumes that the document collection is representative and covers the relevant domain adequately.

Preprocessing steps, such as stop word removal, may inadvertently eliminate important keywords, affecting the ranking accuracy.

The algorithm treats each word as an independent feature, disregarding the context and semantic relationships between words.

## Experimental set up:

| Package/Class/Function | Description |
|---|---|
| os | Used for interacting with the operating system to access and manipulate files and directories. |
| tkinter | Used for creating the graphical user interface (GUI) for the document ranking system. |
| scrolledtext | A widget from the tkinter package used for displaying and scrolling text in the GUI. |
| math | Used for mathematical calculations, such as logarithm calculations. |
| abspath | A function from the os package used to get the absolute path of the directory. |
| listdir | A function from the os package used to list the files in a directory. |
| join | A function from the os package used to join the directory path and file name. |
| open | A function used to open a file for reading. |
| read | A method used to read the contents of a file. |
| splitext | A function from the os.path module used to split the file name and extension. |
| Text | A class from the tkinter package used for creating a text widget in the GUI. |
| Button | A class from the tkinter package used for creating buttons in the GUI. |
| Entry | A class from the tkinter package used for creating an input field in the GUI. |
| Label | A class from the tkinter package used for creating labels in the GUI. |
| Toplevel | A class from the tkinter package used for creating additional windows in the GUI. |
| Main_Function | A user-defined function that serves as the main entry point for the document ranking algorithm. |
| getcontent | A user-defined function that retrieves and displays the content of a selected document. |
| clear_labels | A user-defined function that clears the labels and results displayed in the GUI. |
| create_labels | A user-defined function that creates labels to display the ranking results in the GUI. |

The implementation incorporates the Python programming language and its built-in packages, along with the tkinter package for the graphical user interface. The os package is utilized for directory operations, file access, and manipulation. The scrolledtext widget is used to display and scroll the text in the GUI. Mathematical calculations, such as logarithm calculations, are performed using the math package.

The GUI components, including text widgets, buttons, labels, and input fields, are created using the classes provided by the tkinter package. Additional windows for displaying document content and clearing results are created using the Toplevel class.

The experimental setup also involves the usage of user-defined functions, such as Main_Function, getcontent, clear_labels, and create_labels, to implement specific functionalities and enhance the interactivity of the document ranking system.

## Result:

After execution, it gives the following output on screen. It demands a query to be searched in query box. Output at first of code will be like as shown in figure 1,

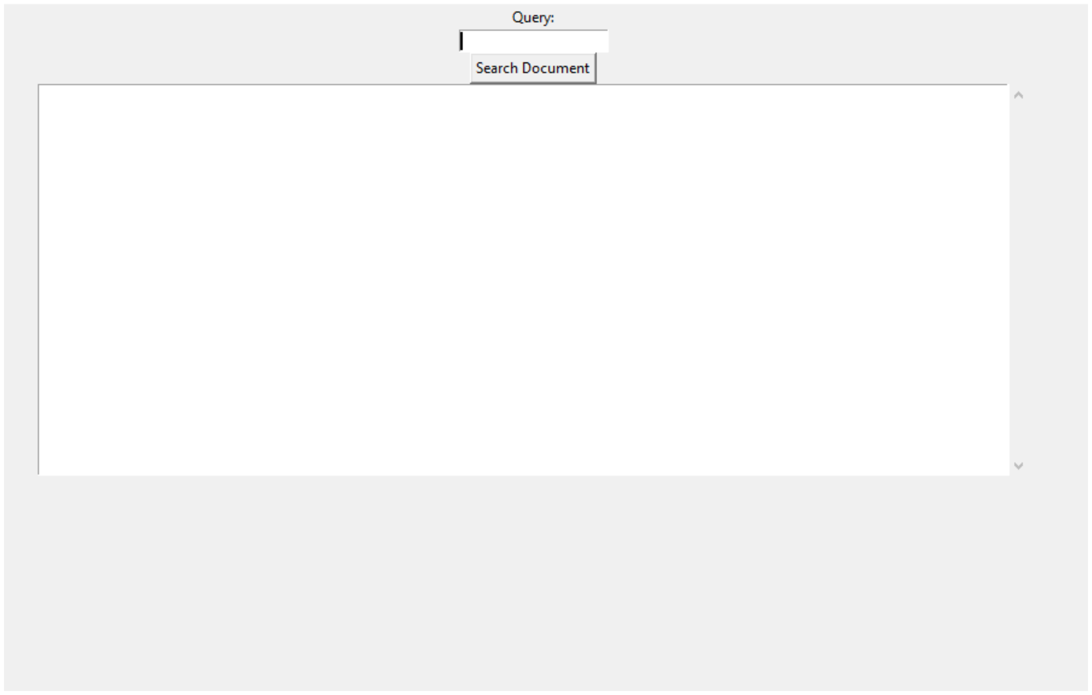
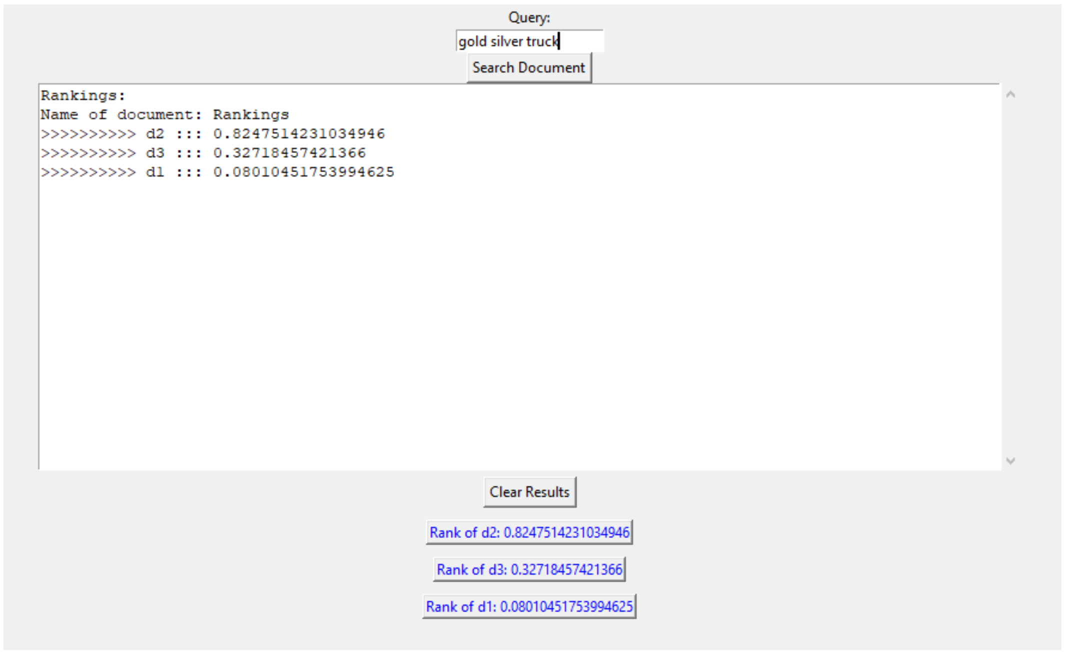


*Figure 1*

After searching given query it gives ranking of documents as shown in figure 2,

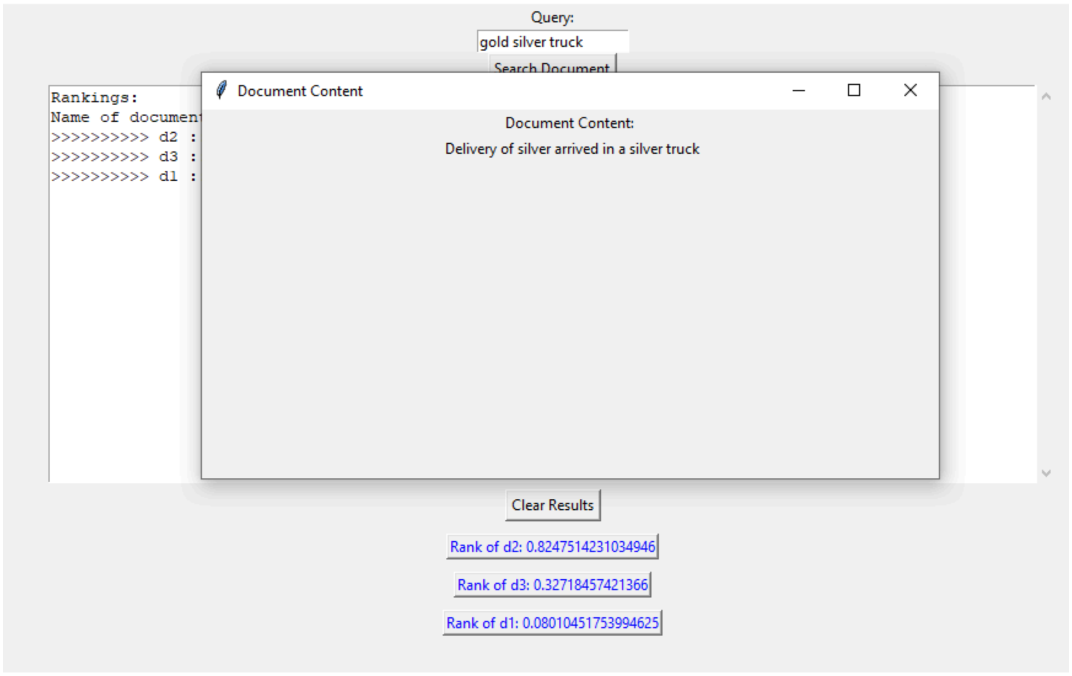


```
Rankings:
Name of document: Rankings
>>>>>>>>>> d2 ::: 0.8247514231034946
>>>>>>>>>> d3 ::: 0.32718457421366
>>>>>>>>>> d1 ::: 0.08010451753994625
```

Rank of d2: 0.8247514231034946
Rank of d3: 0.32718457421366
Rank of d1: 0.08010451753994625

*Figure 2*

As documents are in sorted form now, user can view document which he wants by just clicking documents shown in blue. For example if we want to view d2, I will simply click on it and it's content will be shown as given in figure 3.



```
Rankings:
Name of documen
>>>>>>>>>> d2 :
>>>>>>>>>> d3 :
>>>>>>>>>> d1 :
```

Document Content:
Delivery of silver arrived in a silver truck

Rank of d2: 0.8247514231034946
Rank of d3: 0.32718457421366
Rank of d1: 0.08010451753994625

*Figure 3*

## Code:

```python
import math
import os
import tkinter as tk
from tkinter import scrolledtext


def Main_Function():
    # give directory folder path where text documents are present
    directory = r"D:\Aoa Project final"
    directory = os.path.abspath(directory)
    # making dictionary to store its content by its key "text file name"
    Content_Dictionary = {}
    files_name_list = []
    for filename in os.listdir(directory):
        if filename.endswith(".txt"):  # Filter only text files
            file_path = os.path.join(directory, filename)
            with open(file_path, "r") as file:
                content = file.read()
                file_name = os.path.splitext(filename)[0]
                Content_Dictionary[file_name] = content
                files_name_list.append(file_name)
    # user_query="gold silver truck"
    user_query = entry_query.get()
    text_widget.delete(1.0, tk.END)
    # d means total documents
    d = len(files_name_list)
    str = ""
    content_lst = []
    for i in files_name_list:
        str += Content_Dictionary[i]
        content_lst.append(Content_Dictionary[i])
    words = str.split()
    wordslst = []
    # print(words)
    # ignoring dot in documents
    for word in words:
        if word != '.':
            # print(word)
            wordslst.append(word)
    # for making unique words
    wordslst = set(wordslst)
    # for sorting words alphabetically
    wordslst = sorted(wordslst, key=lambda x: x.lower())
    print(wordslst)
    # maing 2d list for storing calculation
    # It will not store words or items in that 2d list.Just start from Q
    List_2D = []
    rows = len(content_lst) * 2 + 5
    coloumns = len(wordslst)
    for i in range(coloumns):
        row = [0] * rows
        List_2D.append(row)
    # Performing calculation and storing them in 2d list
    for i in range(len(wordslst)):
        # Here i means checking each word
        # Storing Q first
        if wordslst[i] in user_query:
            List_2D[i][0] = 1
        # This loop is for iterating through words of documents sequentally
        for j in range(len(content_lst)):
            newstr = content_lst[j].split()
            for k in newstr:
                if wordslst[i] == k:
                    # D1,D2,D3,.....
                    List_2D[i][j + 1] += 1
                    # calculating DFi
                    if List_2D[i][j + 1] > 1:
                        List_2D[i][d + 1] += -1
                    List_2D[i][d + 1] += 1
                    # D/DFi
                    List_2D[i][d + 2] = d / List_2D[i][d + 1]
                    # IDFi
                    List_2D[i][d + 3] = math.log(List_2D[i][d + 2], 10)
                    d1 = d + 4
        # Calulating Weight,tfi*IDFi
        for l in range(0, len(content_lst) + 1):
            List_2D[i][d1 + l] = List_2D[i][l] * List_2D[i][d + 3]
    # print(List_2D)
    # printing 2d list after calculation
    for j in range(len(List_2D)):
        print(List_2D[j])

    print("Now Rank")
    lstr = []
    # Calculating |Q|,|D1|,|D2|,|D3|,........
    for l in range(len(content_lst) + 1):
        num2 = 0
        for k in range(len(wordslst)):
            # print(List_2D[k][d1+l])
            num1 = List_2D[k][d1 + l] * List_2D[k][d1 + l]
            num2 += num1
        num2 = math.sqrt(num2)
        lstr.append(num2)
    lstr2 = []
    # Calculating Q*Di
    for i in range(len(content_lst)):
        num2 = 0
        for j in range(len(wordslst)):
            # Q*Di
            num2 += List_2D[j][d1] * List_2D[j][d1 + i + 1]
        lstr2.append(num2)
    # print(lstr)
    # print(lstr2)
    # print("so finally based on text,here is following Doc_ranking")
    Doc_name = []
    Doc_ranking = []
    # calculating Cosine Di
    for i in range(len(lstr2)):
        num2 = lstr[0] * lstr[i + 1]
        Doc_name.append(files_name_list[i])
        if num2 == 0:
            Doc_ranking.append(num2)
        else:
            Doc_ranking.append(lstr2[i] / num2)
    # print(Doc_name)
    # print(Doc_ranking)
    # Sorting Document based on their Ranking
    sorted_indices = sorted(range(len(Doc_ranking)), key=lambda x:
Doc_ranking[x], reverse=True)
    Doc_ranking1 = [Doc_ranking[i] for i in sorted_indices]
    Doc_name1 = [Doc_name[i] for i in sorted_indices]
    # GUIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
    text_widget.insert(tk.END, "Rankings:\n")
    text_widget.insert(tk.END, "Name of document: Rankings\n")
    for i in range(len(Doc_ranking1)):
        if Doc_ranking1[i] > 0:
            text_widget.insert(tk.END, f">>>>>>>>>> {Doc_name1[i]} :::
{Doc_ranking1[i]}\n")

    def getcontent(name):
        content = Content_Dictionary.get(name, "")
        content_window = tk.Toplevel(window)
        content_window.title("Document Content")
        content_label = tk.Label(content_window, text="Document
Content:").pack()
        content_label = tk.Label(content_window, text=content).pack()
        content_window.geometry("600x300")
        content_window.geometry("+350+20")
        # content_label.pack(padx=10, pady=10)

    label_list = []

    def clear_labels():
        for label in window.winfo_children():
            if isinstance(label, tk.Label):
                label.destroy()
                text_widget.delete(1.0, tk.END)
                # text_widget.clear()
        if next_button:
            next_button.destroy()

    next_button = tk.Button(window, text="Clear Results", command=clear_labels)
    # clear_labels()
    next_button.pack(pady=5)

    def create_labels():
        for i in range(len(Doc_ranking1)):
            if Doc_ranking1[i] > 0:
                label_text = f"Rank of {Doc_name1[i]}: {Doc_ranking1[i]}"
                label = tk.Label(window, text=label_text, cursor="hand2",
fg="blue", relief="raised")
                label.pack(pady=5)
                label.bind("<Button-1>", lambda e, name=Doc_name1[i]:
```

```
getcontent(name))
                label_list.append(label)

    create_labels()


window = tk.Tk()
window.title("Document Doc_ranking System")
# Create query input label and entry
label_query = tk.Label(window, text="Query:")
label_query.pack()
entry_query = tk.Entry(window)
entry_query.pack()
calculate_button = tk.Button(window, text="Search Document",
command=Main_Function)
calculate_button.pack()
text_widget = scrolledtext.ScrolledText(window, width=100, height=20)
window.geometry("1200x600")
# Set the window position
window.geometry("+50+20")
text_widget.pack()
window.mainloop()
```

## References:

https://openai.com