Information Retrival | Assingment 3 - Search Engine Implementation Using Vector Space Modelling

Group 16: Pranay Dhareshwar, Paridhi Gupta, Saverro Suseno, Zayan Adam Shareef

This Jupyter Notebook outlines the development of a search engine based on the Vector Space Model (VSM). We aim to showcase the implementation of it including indexing, querying, and ranking mechanisms within the VSM framework. This project is part of Group 16's assignment for the Information Retrieval course.

1.0 Data Preparation

In this section, we set up our environment and prepare our dataset for the search engine:

- **Library Imports**: Load the necessary Python libraries for data manipulation, regular expressions, and date handling.
- **Data Loading**: Read our dataset into a `pandas` DataFrame from a CSV file.
- **Null Value Replacement**: Substitute missing data with placeholder text to ensure data consistency.

In order for the search engine to produce accurate and effective search results, it needs clean, consistent, and meaningful data, which is why data preparation is so important. This stage is crucial in the context of the given search engine architecture since it determines the quality of data that will be matched against user queries and feeds into the indexing and query processing components.

```
import pandas as pd
import re
from datetime import datetime
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext

# Load the CSV file into a pandas DataFrame
df = pd.read_csv('goodbooks_dataset.csv')

# Replacing the null values with the given placeholders
df['title'].fillna('[NO TITLE]', inplace=True)
df['authors'].fillna('[NO AUTHOR]', inplace=True)
df['publisher'].fillna('[NO PUBLISHER]', inplace=True)
df['isbn'].fillna('[NO ISBN]', inplace=True)
```

2.0 Text Preprocessing Function

The `prepro` function standardises the text data. It does the following:

- **Convert to String**: Ensure all data is in string format for consistency.
- **Remove Unwanted Characters**: Strip out hashtags, mentions, and URLs.
- **Case Normalisation**: Convert all text to lowercase.
- **Whitespace Management**: Trim extra spaces for clean text.

The process of text preprocessing is essential for streamlining and standardising the input data, eliminating noise, and enabling the search engine to concentrate on the important textual information, hence increasing the relevancy of search results. It guarantees that both user queries and stored data are in a format that facilitates efficient comparison and matching, hence supporting the search engine's query processing.

```
def prepro(text):
    text = str(text)                            # Ensure text is a string
    text = re.sub(r'[#@]\w+', ' ', text)        # Delete hashtags and mentions
    text = re.sub(r'[^a-zA-Z0-9\s]', ' ', text) # Keep letters and numbers
    text = text.lower()                         # Change to lower case
    text = re.sub(r' +', ' ', text).strip()     # Remove extra spaces
    text = re.sub(r'http\S+|www\S+', '', text)  # Remove URLs
    text = re.sub(r'[!?\']', '', text)          # Remove specific punctuation
    text = re.sub(r'</br>', ' ', text)          # Remove breaks
    text = re.sub(r'-', ' ', text)              # Remove hypens
    return text

# Applying the preprocessing
df['title'] = df['title'].map(prepro)
df['authors'] = df['authors'].map(prepro)
df['publisher'] = df['publisher'].map(prepro)
```

3.0 Date Formatting

- Convert string dates to datetime objects with the original format `MM/DD/YYYY`, handling errors gracefully.
- Convert datetime objects back to strings with the desired format `DD/MM/YYYY` for uniformity and improved readability.

In order to facilitate time-based searches and the chronological organisation of results, date formatting standardises time data and makes it possible for search engines to accurately read and compare dates. Appropriate date handling within the architecture of the search engine makes possible more advanced capabilities like ranking search results according to timeliness or filtering by publication date.

```
df['publication_date'] = pd.to_datetime(df['publication_date'], format='%m/%d/%Y', errors='coerce')
df['publication_date'] = df['publication_date'].dt.strftime('%d/%m/%Y')

print('The publication dates after the format is changed:')
df['publication_date']
```

4.0 Search Engine Logic

The function named `search_engine` defines the core search logic and does the following:

- **Exact Match**: Look for an exact match in a specific column if the user has chosen to filter by that column.
- **Vector Space Model**: Use TF-IDF to vectorize the text and cosine similarity for ranking documents relative to the query.

This code defines a `search_engine` function that processes a search query against a dataframe `df` to find relevant information. It allows for two types of searches: column-specific search, if the `search_type` parameter matches a dataframe column, and a general TF-IDF vectorizer-based search across multiple text columns.

The function preprocesses the query, then either filters the dataframe for matching entries in a specified column or calculates cosine similarities between the query and combined text fields to rank and return the most relevant entries.

Strong search logic is necessary to swiftly offer appropriate results. The search engine logic is the central component of the system that defines how user queries are understood and what results are retrieved. This logic directly affects the indexing and ranking procedures in the overall architecture, which in turn affects how quickly the system can parse queries and obtain relevant data.

```
def search_engine(query, search_type='All'):
    preprocessed_query = prepro(query)
    if search_type != 'All' and search_type.lower() in df.columns:
        column_specific_df = df[df[search_type.lower()].str.contains(preprocessed_query, case=False, na=False)]
        if column_specific_df.empty:
            return None, None
        rankings = range(1, len(column_specific_df) + 1)
        return column_specific_df, rankings
    else:
        tfidf_vectorizer = TfidfVectorizer()
        combined_text = df['title'] + " " + df['authors'] + " " + df['publisher'] + " " + df['isbn']
        tfidf_matrix = tfidf_vectorizer.fit_transform(combined_text)
        query_vector = tfidf_vectorizer.transform([preprocessed_query])
        cosine_similarities = cosine_similarity(query_vector, tfidf_matrix).flatten()
        relevant_indices = cosine_similarities.argsort()[::-1]
        if len(relevant_indices) == 0:
            return None, None
        rankings = cosine_similarities[relevant_indices]
        return df.iloc[relevant_indices], rankings
```

5.0 Event Handling for Search

- `on_search_click` is an event handler that triggers when the user clicks the 'Search' button:
  - **Get Query**: Retrieve the user's query from the input field.
  - **Execute Search**: Call the `search_engine` function to find matching results.
  - **Display Results**: Update the Treeview with the search outcomes.
  - **No Results Feedback**: Inform the user when no results are found.

Event handling for search is necessary to ensure a responsive user experience, capturing and processing user input seamlessly, which is fundamental to user satisfaction. It's a critical component in the GUI of the search engine architecture, where the interaction between the user's input and the system's output is managed, providing immediate feedback on search actions.

```
def on_search_click():
    query = entry.get()
    search_option = search_option_combobox.get()
    result, rankings = search_engine(query, search_option)
    if result is not None and not result.empty:
        dataframe_to_treeview(result, rankings, result_tree)
    else:
        for i in result_tree.get_children():
            result_tree.delete(i)
        status_label.config(text="No results found.")
```

6.0 Search Ranking

- **Function:** The function `dataframe_to_treeview` iterates through the search results stored in a DataFrame, and for each result, it assigns a rank and displays this information in a hierarchical view (TreeView), which allows users to see a sorted list of results based on their relevance to the search query.
- **Status Label:** It also updates a status label with the total number of results displayed, providing immediate feedback to the user on the quantity of data that has been ranked and is currently viewable, enhancing the transparency of the search process.

A search engine's ranking system is crucial because it ranks the results according to how relevant they are to the user's query, improving the search experience by displaying the most relevant content first. In order to guarantee that the user sees the best match to their search query first, ranking functions as the last filter that the processed data must travel through in the search engine's architecture before being presented. Since it indicates how well the search algorithm understands and responds to user intent, it strongly ties in with search engine logic and is essential to end-user happiness.

```
def dataframe_to_treeview(df, rankings, tree):
    for i in tree.get_children():
        tree.delete(i)
    for index, (row, rank) in enumerate(zip(df.iterrows(), rankings), start=1):
        _, row_data = row
        row_values = (index, row_data['title'], row_data['authors'], row_data['average_rating'],
                      row_data['publication_date'], row_data['publisher'], row_data['isbn'])
        tree.insert("", "end", values=row_values)
    status_label.config(text=f"Shows {len(df)} results with rankings")
```

7.0 General User Interface Setup

The GUI setup code plays a crucial role in creating the user interface. It sets the interactive features and layout that users would interact with the search engine, which in turn impacts their satisfaction and experience. It makes the process of entering and displaying search queries and results easier, which is crucial for the search engine's overall efficiency and functionality in the described architecture.

Setting up the Graphical User Interface using Tkinter:

- **Main Window**: Create the primary application window with a title.
- **Search Frame**: Construct a frame to hold search-related widgets.
- **Input Fields and Labels**: Add input fields for query text and dropdown menus for specifying the search type.
- **Result Display**: Set up a Treeview to show the search results with a scrollbar for navigation.

- **Application Launch**: Enter the Tkinter main event loop to start the application.

This script sets up a graphical user interface (GUI) for a search engine using Tkinter, where users can enter a search query and choose from specific search options through a dropdown menu. The on_search_click function retrieves the user's input, executes the search using the search_engine function (defined separately), and displays the results in a Treeview widget as a formatted table.

If no results are found, it clears the table and updates a status label to notify the user. The GUI layout includes a title label, entry fields for the query and search options, a search button, and a Treeview table with a scrollbar for displaying search results. The application is structured to initiate with the main window setup, including widget configurations for user interactions and result display.

```
# GUI Setup
root = tk.Tk()
root.title("Group 16 Search Engine using Vector Space Modelling")

# Title Label
title_label = tk.Label(root, text="Group 16 Search Engine using Vector Space Modelling", font=("Helvetica", 16))
title_label.pack(pady=10)

# Creating a frame for the query and search type
query_frame = tk.Frame(root)
query_frame.pack(pady=10)

# Creating widgets for specific search options
entry_label = tk.Label(query_frame, text="Enter your query:")
entry_label.pack(side=tk.LEFT, padx=(0, 10))

entry = tk.Entry(query_frame, width=50)
entry.pack(side=tk.LEFT)

search_option_label = tk.Label(query_frame, text="Search by:")
search_option_label.pack(side=tk.LEFT, padx=(20, 10))

search_options = ['All', 'Title', 'Authors', 'ISBN', 'Publisher']
search_option_combobox = ttk.Combobox(query_frame, values=search_options, width=15)
search_option_combobox.current(0)
search_option_combobox.pack(side=tk.LEFT)

search_button = tk.Button(query_frame, text="Search", command=on_search_click)
search_button.pack(side=tk.LEFT, padx=(20, 0))

status_label = tk.Label(root, text="")
status_label.pack(pady=10)

# Defining the columns
columns = ('Rank', 'Title', 'Authors', 'Average Rating', 'Publication Date', 'Publisher', 'ISBN')

# Setting up the Treeview widget for the table inside
frame = tk.Frame(root)
frame.pack(expand=True, fill='both', padx=10, pady=10)

result_tree = ttk.Treeview(frame, columns=columns, show='headings')
result_tree.pack(side=tk.LEFT, expand=True, fill='both')

# scrollbar
scrollbar = ttk.Scrollbar(frame, orient="vertical", command=result_tree.yview)
scrollbar.pack(side=tk.RIGHT, fill='y')
result_tree.configure(yscrollcommand=scrollbar.set)

for col in columns:
    result_tree.heading(col, text=col)
    result_tree.column(col, width=100, anchor=tk.CENTER)

# Running the application
root.mainloop()
```