INNOMATICS® RESEARCH LABS

INNOVATION. AUTOMATION. ANALYTICS

PROJECT ON

Enhancing Search Engine Relevance for Video Subtitles

By

Team ID

SAIPRANEETH S – IN1240164

HEMA B - IN1240288

Abstract:

We're using advanced techniques like understanding the meaning of words and how they fit together. Instead of just looking for specific words, we're trying to understand what the user wants and find subtitles that best match that. We have three main steps: getting the subtitles ready for analysis, turning the words into numbers so we can compare them, and then figuring out how similar the subtitles are to what the user asked for. By doing this, we hope to provide more helpful search results and make subtitles more accessible for everyone.

1.Reading the given data:

- The dataset comprises 82,498 subtitle files obtained from opensubtitles.org.
- These subtitles predominantly cover movies and TV series released between 1990 and 2024.
- The database file is named eng_subtitles_database.db.
- Within the database, there exists a table named 'zipfiles' containing three columns: 'num' (unique subtitle ID on opensubtitles.org), 'name' (subtitle file name), and 'content' (compressed subtitle data stored as binary using 'latin-1' encoding).
- Additional details about each subtitle can be accessed using the 'num' column in the URL: https://www.opensubtitles.org/en/subtitles/{num}, with {num} representing the unique subtitle ID.

In [1]:
```
import sqlite3
import pandas as pd
```

Step 1 - Reading the Tables from Database file

In [2]:
```
cd C:\Users\DELL\OneDrive\Desktop\data-20240407T052706Z-001\data
```

```
C:\Users\DELL\OneDrive\Desktop\data-20240407T052706Z-001\data
```

In [3]:
```
# Read the code below and write your observation in the next cell

conn = sqlite3.connect("eng_subtitles_database.db")
cursor = conn.cursor()
cursor.execute("SELECT name FROM sqlite_master WHERE type='table'")
print(cursor.fetchall())
```

```
[('zipfiles',)]
```

1.1 Loading the database

This Python script demonstrates the process of fetching subtitle data from a SQLite database, decoding it, cleaning it, and storing it in a Pandas DataFrame.

- The script establishes a connection to the SQLite database containing subtitle data using the sqlite3.connect() function. The database path is specified as database_path.

- A SQL query is executed to select data from the "zipfiles" table within the SQLite database. The query retrieves columns named "num", "name", and "content", representing subtitle ID, name, and content respectively.
-

Step 3 - Loading the Database Table inside a Pandas DataFrame

```
In [5]: df = pd.read_sql_query("""SELECT * FROM zipfiles""", conn)
        df.head()
```

Out[5]:

	num	name	content
0	9180533	the.message.(1976).eng.1cd	b'PK\x03\x04\x14\x00\x00\x00\x08\x00\x1c\xa9\x...
1	9180583	here.comes.the.grump.s01.e09.joltin.jack.in.bo...	b'PK\x03\x04\x14\x00\x00\x00\x08\x00\x17\xb9\x...
2	9180592	yumis.cells.s02.e13.episode.2.13.(2022).eng.1cd	b'PK\x03\x04\x14\x00\x00\x00\x08\x00L\xb9\x99V...
3	9180594	yumis.cells.s02.e14.episode.2.14.(2022).eng.1cd	b'PK\x03\x04\x14\x00\x00\x00\x08\x00U\xa9\x99V...
4	9180600	broker.(2022).eng.1cd	b'PK\x03\x04\x14\x00\x00\x00\x08\x001\xa9\x99V...

1.2 Unzipping the content and decoding using latin-1

The `extract_content` function processes binary content representing a zip archive. It employs an in-memory binary stream and iterates through the archive's files. Each file's content is decoded under the assumption that it's text data, using the "latin-1" encoding. The function returns the content of the first file encountered. This process is then applied to the 'subtitle_content' column of DataFrame `data`, enabling extraction of zip archive content stored within this column.

Applying for entire data

```
In [10]: import zipfile
         import io

         count = 0

         def decode_method(binary_data):
             global count
             # Decompress the binary data using the zipfile module
             # print(count, end=" ")
             count += 1
             with io.BytesIO(binary_data) as f:
                 with zipfile.ZipFile(f, 'r') as zip_file:
                     # Assuming there's only one file in the ZIP archive
                     subtitle_content = zip_file.read(zip_file.namelist()[0])

             # Now 'subtitle_content' should contain the extracted subtitle content
             return subtitle_content.decode('latin-1')  # Assuming the content is UTF-8 encoded text
```

```
In [11]: df['file_content'] = df['content'].apply(decode_method)

         df.head()
```

Out[11]:

	num	name	content	file_content
0	9180533	the.message.(1976).eng.1cd	b'PK\x03\x04\x14\x00\x00\x00\x08\x00\x1c\xa9\x...	1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch an...
1	9180583	here.comes.the.grump.s01.e09.joltin.jack.in.bo...	b'PK\x03\x04\x14\x00\x00\x00\x08\x00\x17\xb9\x...	1\r\n00:00:29,359 --> 00:00:32,048\r\nAh! Ther...
2	9180592	yumis.cells.s02.e13.episode.2.13.(2022).eng.1cd	b'PK\x03\x04\x14\x00\x00\x00\x08\x00L\xb9\x99V...	1\r\n00:00:53,200 --> 00:00:56,030\r\n<i>Yumi'...
3	9180594	yumis.cells.s02.e14.episode.2.14.(2022).eng.1cd	b'PK\x03\x04\x14\x00\x00\x00\x08\x00U\xa9\x99V...	1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch an...
4	9180600	broker.(2022).eng.1cd	b'PK\x03\x04\x14\x00\x00\x00\x08\x001\xa9\x99V...	ï»¿1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch...

In [20]: `data.head()`

Out[20]:

	num	name	file_content
0	9180533	the.message.(1976)	1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch an...
1	9180583	here.comes.the.grump.s01.e09.joltin.jack.in.bo...	1\r\n00:00:29,359 --> 00:00:32,048\r\nAh! Ther...
2	9180592	yumis.cells.s02.e13.episode.2.13.(2022)	1\r\n00:00:53,200 --> 00:00:56,030\r\n<i>Yumi'...
3	9180594	yumis.cells.s02.e14.episode.2.14.(2022)	1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch an...
4	9180600	broker.(2022)	ï»¿1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch...

2.Data Preprocessing

The subtitle content within the 'File_content' column typically contains time stamps, HTML tags, and various forms of noise. It's imperative to clean this text data as part of the preprocessing for natural language processing (NLP) tasks. Cleaning involves eliminating irrelevant information, noise, and inconsistencies to ensure the text is better suited for subsequent processing steps like vectorization.

In [24]:
```
import string
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from bs4 import BeautifulSoup
import unicodedata
from nltk.stem import WordNetLemmatizer
import re

def clean_text(sentence):
    # Remove timestamps
    clean_sentence = re.sub(r'\d+:\d+:\d+,?\d* --> \d+:\d+:\d+,?\d*', '', sentence)

    # Remove special characters and extra spaces
    clean_sentence = re.sub(r'[^a-zA-Z0-9\s]', '', clean_sentence)

    # Convert text to lowercase
    clean_sentence = clean_sentence.lower()

    # Remove leading and trailing whitespace
    clean_sentence = clean_sentence.strip()

    # Removing HTML tags
    clean_sentence = BeautifulSoup(clean_sentence, 'html.parser').get_text()

    # Removing URLs
    clean_sentence = ' '.join([word for word in clean_sentence.split() if not word.startswith('http')])

    # Removing punctuation
    clean_sentence = ''.join([char for char in clean_sentence if char not in string.punctuation + '''])

    # Removing numbers
    clean_sentence = ''.join([i for i in clean_sentence if not i.isdigit()])

    # Removing stopwords
    stop_words = set(stopwords.words('english'))
    word_tokens = word_tokenize(clean_sentence)
    clean_sentence = ' '.join([word for word in word_tokens if word.lower() not in stop_words])
```

This code defines a function clean_text that preprocesses text data for natural language processing (NLP) tasks. Here's an explanation of each step:

Remove Timestamps: Using a regular expression (re.sub), it removes timestamps in the format 00:00:00,000 --> 00:00:00,000.

Remove Special Characters and Extra Spaces: It removes any characters that are not alphanumeric or whitespace using another regular expression.

Convert Text to Lowercase: All text is converted to lowercase to ensure consistency.

Remove Leading and Trailing Whitespace: It strips any leading or trailing whitespace.

Removing HTML Tags: BeautifulSoup is used to remove any HTML tags present in the text.

Removing URLs: It removes any words that start with 'http', assuming they are URLs.

Removing Punctuation: All punctuation marks are removed using a list comprehension.

Removing Numbers: Any digits are removed from the text.

Removing Stopwords: NLTK's stopwords list for English is used to remove common stopwords like 'the', 'and', 'is', etc.

Handling Special Characters: Special characters are normalized using Unicode normalization.

Lemmatization: Words are lemmatized using NLTK's WordNet lemmatizer, reducing them to their base or dictionary form.

Finally, the cleaned text is returned. This process helps in removing noise, irrelevant information, and inconsistencies from the text, making it more suitable for NLP tasks like text classification or sentiment analysis.

```
In [28]: data
Out[28]:
```

	num	name	file_content	clean_file_content
0	9180533	the.message.(1976)	1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch an...	watch video online opensubtitles free browser ...
1	9180583	here.comes.the.grump.s01.e09.joltin.jack.in.bo...	1\r\n00:00:29,359 --> 00:00:32,048\r\nAh! Ther...	ah there princess dawn terry blooney looney so...
2	9180592	yumis.cells.s02.e13.episode.2.13.(2022)	1\r\n00:00:53,200 --> 00:00:58,030\r\n<i>Yumi'...	iyumis cell iepisode extremely polite yumi iy...
3	9180594	yumis.cells.s02.e14.episode.2.14.(2022)	1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch an...	watch video online opensubtitles free browser ...
4	9180600	broker.(2022)	ï»¿1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch...	watch video online opensubtitles free browser ...
...	...	...	...	...
82493	9521935	the.prophets.game.(2000)	ï»¿1\r\n00:01:16,284 --> 00:01:19,537\r\nGod,\...	god punishing red head us gun green chest us s...
82494	9521937	west.beirut.(1998)	1\r\n00:00:06,000 --> 00:00:12,074\r\napi.Open...	apiopensubtitlesorg deprecated please implemen...
82495	9521938	frankenstein.the.true.story.(1973)	1\r\n00:00:01,001 --> 00:00:04,630\r\n(Dramati...	dramatic orchestral music advertise product br...
82496	9521940	frankenstein.the.true.story.(1973)	1\r\n00:00:06,000 --> 00:00:12,074\r\nAdvertis...	advertise product brand contact wwwopensubtit...
82497	9521941	zombie.island.massacre.(1984)	1\r\n00:00:01,919 --> 00:00:03,253\r\n(Sharp w...	sharp whistling man hey wait wait growling cla...

82498 rows × 4 columns

2.1 Converting to CSV:

Export DataFrame to CSV: It exports the DataFrame to a CSV file named "Data.csv", excluding the index column, and using '\' as the escape character.

Cleaning Text Data: It applies the clean_text function to the 'file_content' column of the DataFrame, storing the cleaned content in a new column named 'clean_file_content'.

Dropping Original Column: It drops the original 'file_content' column from the DataFrame.

```
In [32]: import pandas as pd
         import os

         # Example directory path
         directory = 'D:\internship\Data science\TASKS\search engine\data'

         # Check if the directory has write permission
         if not os.access(directory, os.W_OK):
             print(f"No write permission in directory: {directory}")
             # You may choose to exit the script or handle this situation differently
         else:
             # Assuming 'data' is your DataFrame
             try:
                 data.to_csv(os.path.join(directory, 'eng_subtitles_database.csv'), index=False)
                 print("Data successfully saved to CSV.")
             except PermissionError as e:
                 print(f"PermissionError: {e}")

         Data successfully saved to CSV.
```

3.Vectorize the given Subtitle Documents

Bag-of-Words (BOW) and Term Frequency-Inverse Document Frequency (TF-IDF) are traditional methods for generating sparse vector representations of text data.

```
In [13]: import pandas as pd
         import numpy as np

In [14]: data=pd.read_csv("D:\internship\Data science\TASKS\search engine\data\eng_subtitles_database.csv")

In [15]: data.shape

Out[15]: (82498, 3)

In [16]: data.head()

Out[16]:
```

	num	name	clean_file_content
0	9180533	the.message.(1976)	watch video online opensubtitles free browser ...
1	9180583	here.comes.the.grump.s01.e09.joltin.jack.in.bo...	ah there princess dawn terry blooney looney so...
2	9180592	yumis.cells.s02.e13.episode.2.13.(2022)	iyumis cell iepisode extremely polite yumii iy...
3	9180594	yumis.cells.s02.e14.episode.2.14.(2022)	watch video online opensubtitles free browser ...
4	9180600	broker.(2022)	watch video online opensubtitles free browser ...

3.1 Advantages:

Semantic Information: BERT-based Sentence Transformers capture semantic links between words and sentences, enabling a more nuanced understanding of the text.

Contextual Embeddings: Unlike BOW and TF-IDF, which treat words independently, Sentence Transformers generate dense embeddings that consider the context of words. This preserves contextual details and enhances representation quality.

Lower Dimensionality: Dense embeddings from Sentence Transformers typically have lower dimensionality than sparse representations, reducing computational overhead and memory usage.

Pre-trained Models: Sentence Transformers leverage pre-trained models like BERT, benefiting from extensive training on vast text corpora, thus capturing diverse linguistic patterns and semantic connections.

```
In [17]: data['clean_file_content'][0]
```

```
Out[17]: 'watch video online opensubtitles free browser extension osdblinkext name god gracious merciful muhammad messenger god heracl
ius emperor byzantium greeting follower righteous guidance bid hear divine call messenger god people accept islam salvation s
peaks new prophet arabia like john baptist came king herod desert cry salvation muqawqis patriarch alexandria kisra emperor p
ersia muhammad call call god accept islam salvation embrace islam come desert smelling camel goat tell persia kneel muhammad
messenger god gave authority god sent muhammad mercy mankind scholar historian islam university alazhar cairo high islamic co
ngress shiat lebanon maker film honour islamic tradition hold impersonation prophet offends spirituality message therefore pe
rson mohammad shown year christ diedi iwhen europe sunk dark agesi iand everywhere old civilization fallingi imuhammad born m
ecca arabiai imecca rich trading city ruled merchantsi iwhose wealth multiplied unique privilegei ithey housed godsi ievery y
ear time great fairi ithe desert priest brought idolsi iand image god custody kaabai ionce holy shrine abrahami ithe kaaba be
come house idolotryi ihosting fewer different godsi imecca adi bilal today count umaya yet year god gold put god prophet toge
ther sit pretty hmm god place kaaba caravan syria hmmthey must running theyll thirsty put five men north well many sheep shal
l kill give hundred mecca must keep name hospitality ten lamb leader bread water poet hakim house verse prose nightly put sla
ughter andand bread swear thinner water ohopen space open space lover poetry abu sufian willing patron art abu sofyan invite
poet joy kit love kin wine cake abound skill abu sofyan revel song begin abu sofyan invite poet silkworm china lady pleasure
limb lady see ravish eye yes length dinar abu sofyans wife oh gold god kaaba need upkeep man stood looked soul carry away mus
t muhammad come dont stop nephew maybe change change year old unnatural rich wife could afford best mecca yet chooses sit shi
vering cave unnatural man dare risk anger aluzza keep health manat god prosperity allat god family tribe hubal hubal start ca
ravan predicts fate challenge god within earshot god dangerous unreasonable rebellious blasphemous yes im afraid muhammad har
m always sad great fair might see next one abu talib abu talib catch breath zaid muhammad come mount hira yet he three day ha
```

BOW and TF-IDF have drawbacks:

Lack of Semantic Information: They fail to capture semantic connections among words or documents. Each word is treated in isolation, disregarding its context. Consequently, they may not fully grasp the intended meaning of the text.

High Dimensionality: BOW and TF-IDF yield high-dimensional sparse vectors, particularly with extensive vocabularies or datasets containing numerous unique tokens. This can lead to computational inefficiencies and increased memory usage.

Given these drawbacks, utilizing BOW or TF-IDF may not be optimal for tasks requiring semantic understanding or contextual comprehension. For endeavors like constructing a Semantic Search Engine, employing methods that encode semantic knowledge and contextual associations between words and documents is more advantageous.

4. Vectorizer

```
In [23]: count_vectorizer = CountVectorizer()
         tf_matrix = count_vectorizer.fit_transform(data['clean_file_content'])

In [24]: count_vectorizer

Out[24]: CountVectorizer()
```

**In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.**

```
In [25]: # TF-IDF transformer and transform the TF matrix
```

We use `CountVectorizer` to convert text data into a matrix of token counts. This process transforms text into a numerical format suitable for machine learning algorithms, capturing the frequency of each word in the text. It's valuable for tasks like text classification or clustering, providing a foundational representation of text data for further analysis.

TF-IDF transformer

The `TfidfTransformer` is used to convert a term frequency (TF) matrix into a TF-IDF matrix. This transformation normalizes TF values and applies inverse document frequency (IDF) weighting to emphasize rare terms and downweight common ones. It improves document representations by considering both local and global term characteristics, leading to more informative and discriminative document representations.

Calculating similarity using cosine similarity

To calculate similarity using cosine similarity, we compute the cosine of the angle between two vectors in a multidimensional space. In the context of text data, each document or text snippet is represented as a vector, and the similarity between two documents is determined by the cosine of the angle between their respective vectors. A cosine similarity of 1 indicates perfect similarity, while 0 indicates no similarity.

This measure is commonly used in information retrieval, recommendation systems, and clustering to assess the similarity between documents or text snippets based on their content. It's particularly useful when dealing with high-dimensional data, such as text data represented by TF-IDF or word embeddings**.**

In [29]:
```
# Calculating similarity using cosine similarity
```

In [30]:
```
query = input()
query_vector = count_vectorizer.transform([query])
query_tfidf = tfidf_transformer.transform(query_vector)
```

```
avatar
```

In [32]:
```
similarity_scores = cosine_similarity(query_tfidf, tfidf_matrix)
```

5.Retrieving Documents

Retrieving Similar Documents: The code sorts the similarity scores in descending order to identify the top matching documents. It retrieves the content, subtitle IDs, and names of these documents.

Summarizing Documents: It iterates over the retrieved documents and generates summaries based on a given query. The function generate_summarized_documents() is called to create summaries for each document.

Displaying Results: For each retrieved document, it prints the summary, subtitle ID, and subtitle name (if available).

```
In [43]: top_indices = similarity_scores.argsort()[0][::-1]
         top_n = 5
         retrieved_documents = [data['clean_file_content'][idx] for idx in top_indices[:top_n]]
         retrieved_subtitle_ids = [data['num'][idx] for idx in top_indices[:top_n]]  # Assuming you h
         retrieved_subtitle_names = [data['name'][idx] for idx in top_indices[:top_n]]  # Assuming yo

In [46]: def generate_summarized_documents(query, document):
             # Your implementation for summarizing the document based on the query
             pass

In [47]: summarized_docs = []

         # Iterate over each retrieved document and generate summaries
         for doc in retrieved_documents:
             summarized_doc = generate_summarized_documents(query, doc)
             summarized_docs.append(summarized_doc)
```

This process allows users to input a query, find documents similar to the query, and display summarized information about those documents. It's useful for tasks such as document retrieval and summarization, aiding in information organization and understanding.

```
In [51]: # Assuming you have a list containing subtitle names called 'retrieved_subtitle_names'
for i, (summary, subtitle_id) in enumerate(zip(summarized_docs, retrieved_subtitle_ids), 1):
    print("Document", i, ":")
    print("Summary:", summary)
    print("Subtitle ID:", subtitle_id)

    if i <= len(retrieved_subtitle_names):
        print("Subtitle Name:", retrieved_subtitle_names[i - 1])
    else:
        print("Subtitle Name: Not available")  # Handle case where subtitle name list is shorter t

    print()
```

```
Document 1 :
Summary: None
Subtitle ID: 9233592
Subtitle Name: archer.s13.e03.saturday.(2022)

Document 2 :
Summary: None
Subtitle ID: 9233396
Subtitle Name: archer.s13.e03.saturday.(2022)

Document 3 :
Summary: None
Subtitle ID: 9233399
Subtitle Name: archer.s13.e03.saturday.(2022)

Document 4 :
Summary: None
Subtitle ID: 9233593
Subtitle Name: archer.s13.e03.saturday.(2022)

Document 5 :
Summary: None
```

This organizes the summarized documents and their associated subtitle IDs into a structured format for display. It ensures clarity and readability by presenting each document's summary alongside its unique identifier and, if applicable, its name. This approach facilitates efficient understanding and interpretation of the retrieved documents and their metadata.

```
In [52]: import joblib

         # CountVectorizer
         joblib.dump(count_vectorizer, 'count_vectorizer.joblib')

Out[52]: ['count_vectorizer.joblib']

In [53]: # TfidfTransformer
         joblib.dump(tfidf_transformer, 'tfidf_transformer.joblib')

Out[53]: ['tfidf_transformer.joblib']

In [54]: joblib.dump(tfidf_matrix, 'tfidf_matrix.joblib')

Out[54]: ['tfidf_matrix.joblib']

In [55]: # cosine similarity model
         joblib.dump(similarity_scores, 'cosine_similarity_scores.joblib')

Out[55]: ['cosine_similarity_scores.joblib']
```

We use this to save various components of our text processing and similarity modeling pipeline:

CountVectorizer: It's saved to preserve the mapping between words and their indices, allowing consistent vectorization of new data.

TfidfTransformer: Similarly, it's stored to retain information about the TF-IDF transformation applied to the data.

TF-IDF Matrix: The TF-IDF matrix itself is saved to avoid recomputation, ensuring consistency in future analysis.

Cosine Similarity Model: The cosine similarity scores calculated between documents are saved, enabling quick retrieval and comparison of document similarities without needing to recalculate them.

6.Streamlit Application Code:

```python
import streamlit as st
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.metrics.pairwise import cosine_similarity
import joblib

# Define global variables to store history and chat data
history = []
chat_data = []

def load_data(csv_file):
    data = pd.read_csv(csv_file)
    return data

def load_models():
    count_vectorizer = joblib.load(r"D:\internship\Data science\TASKS\search engine\models\count_vectorizer.joblib")
    tfidf_transformer = joblib.load(r"D:\internship\Data science\TASKS\search engine\models\tfidf_transformer.joblib")
    tfidf_matrix = joblib.load(r"D:\internship\Data science\TASKS\search engine\models\tfidf_matrix.joblib")
    return count_vectorizer, tfidf_transformer, tfidf_matrix

def retrieve_similar_documents(query, count_vectorizer, tfidf_transformer, tfidf_matrix, data, top_n=5):
    query_vector = count_vectorizer.transform([query])
    query_tfidf = tfidf_transformer.transform(query_vector)
    similarity_scores = cosine_similarity(query_tfidf, tfidf_matrix)
    top_indices = similarity_scores.argsort()[0][::-1]
    retrieved_documents = [data['clean_file_content'][idx] for idx in top_indices[:top_n]]
    retrieved_subtitle_names = [data['name'][idx] for idx in top_indices[:top_n]]  # Assuming subtitle names are stored in 'name' column
    retrieved_subtitle_nums = [data['num'][idx] for idx in top_indices[:top_n]]  # Assuming subtitle numbers are stored in 'subtitle_num'

    return retrieved_documents, retrieved_subtitle_names, retrieved_subtitle_nums
```

```python
def main():
    global history, chat_data

    # Customizing title and header
    st.title('🎬🔍 FilmFinder')
    st.subheader('🍿 In the realm of movies,allow SeekSpot to guide you through a cinematic adventure,effortlessly finding the films

    # Sidebar navigation
    st.sidebar.title('🌟 Navigation')
    if st.sidebar.button('🏠 Home'):
        st.sidebar.text('Go to Home')
        # Clear chat data and history
        history = []
        chat_data = []
    if st.sidebar.button('📜 History'):
        st.sidebar.text('View Search History')
        # Display search history
        st.sidebar.write(history)
    if st.sidebar.button('💾 Export'):
        st.sidebar.text('Export Data')
        # Export chat data to a file
        export_data(chat_data)
    if st.sidebar.button('⚙ Settings'):
        st.sidebar.text('Change Settings')

    # Search functionality
    query = st.text_input('Enter your query:', '')
    if st.button('🔎 Search'):
        if query:
            # Add query to history
            history.append(query)
            # Retrieve similar documents
            retrieved_documents, retrieved_subtitle_names, retrieved_subtitle_nums = retrieve_similar_documents(query, count_vectori
            st.subheader('📄 Top 5 documents similar to the query:')
```

```python
            for i, (doc, subtitle_name, subtitle_num) in enumerate(zip(retrieved_documents, retrieved_subtitle_names, retrieved_subtitle
                st.write(f"**Document {i}:**")
                st.write(f"Subtitle Name: {subtitle_name}")
                st.write(f"Subtitle Number: {subtitle_num}")
                st.write("Summary:", doc)
            # Add search results to chat data
            chat_data.append((query, retrieved_documents, retrieved_subtitle_names, retrieved_subtitle_nums))

def export_data(data):
    # Export chat data to a file
    pass  # Placeholder for actual export functionality

if __name__ == '__main__':
    data = load_data(r"D:\internship\Data science\TASKS\search engine\data\eng_subtitles_database.csv")
    count_vectorizer, tfidf_transformer, tfidf_matrix = load_models()
    main()
```

This Streamlit application, named "FilmFinder," serves as a movie search engine. Here's a summary of its features:

Imports: The code imports essential libraries like Streamlit for building the web app, pandas for data handling, and scikit-learn for text processing tasks such as vectorization and similarity computation.

Global Variables: Two lists are globally defined to store the search history (history) and chat data (chat_data).

Functions:

load_data(csv_file): Loads movie subtitle data from a CSV file into a pandas DataFrame.

load_models(): Loads pre-trained models for text processing tasks.

retrieve_similar_documents(query, count_vectorizer, tfidf_transformer, tfidf_matrix, data, top_n): Finds similar movie documents based on a user query using cosine similarity.

main(): The core function of the application. It constructs the layout, including the title, sidebar navigation, search input, and search button. Additionally, it manages the search functionality and displays results.

Main Features:

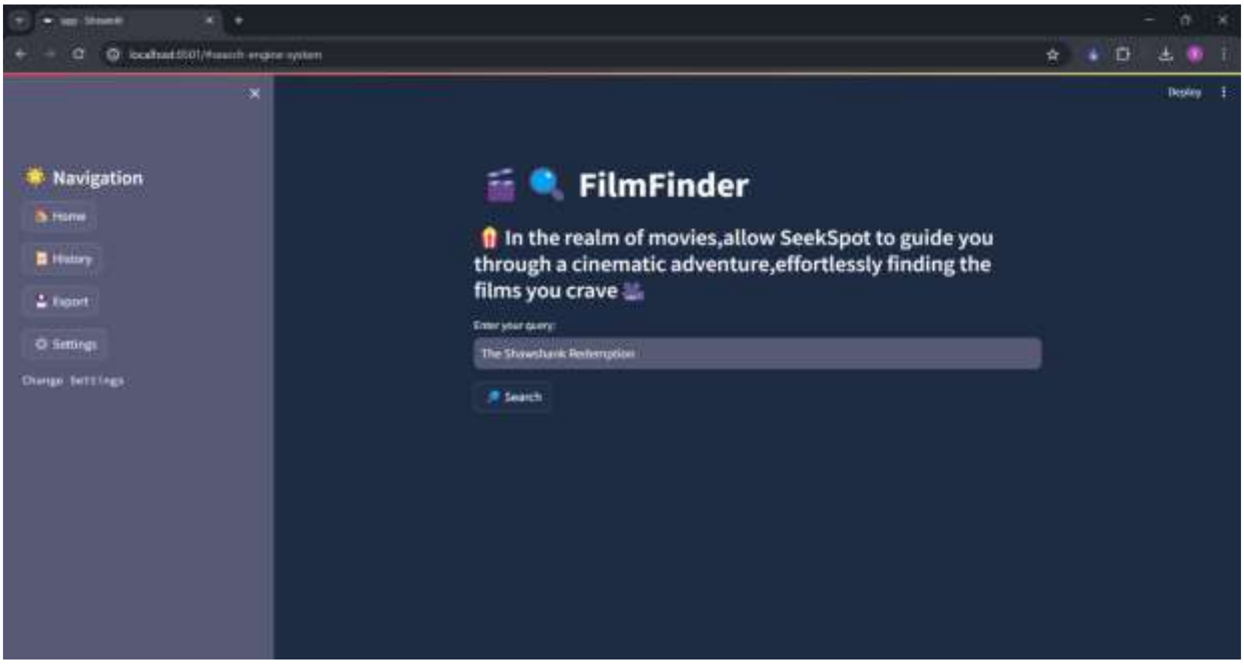
Title and Header: Establishes the title and a subheader for the application.

Sidebar Navigation: Offers buttons for Home, History, Export, and Settings. Each button triggers specific actions, such as clearing history, displaying search history, exporting data, or adjusting settings.

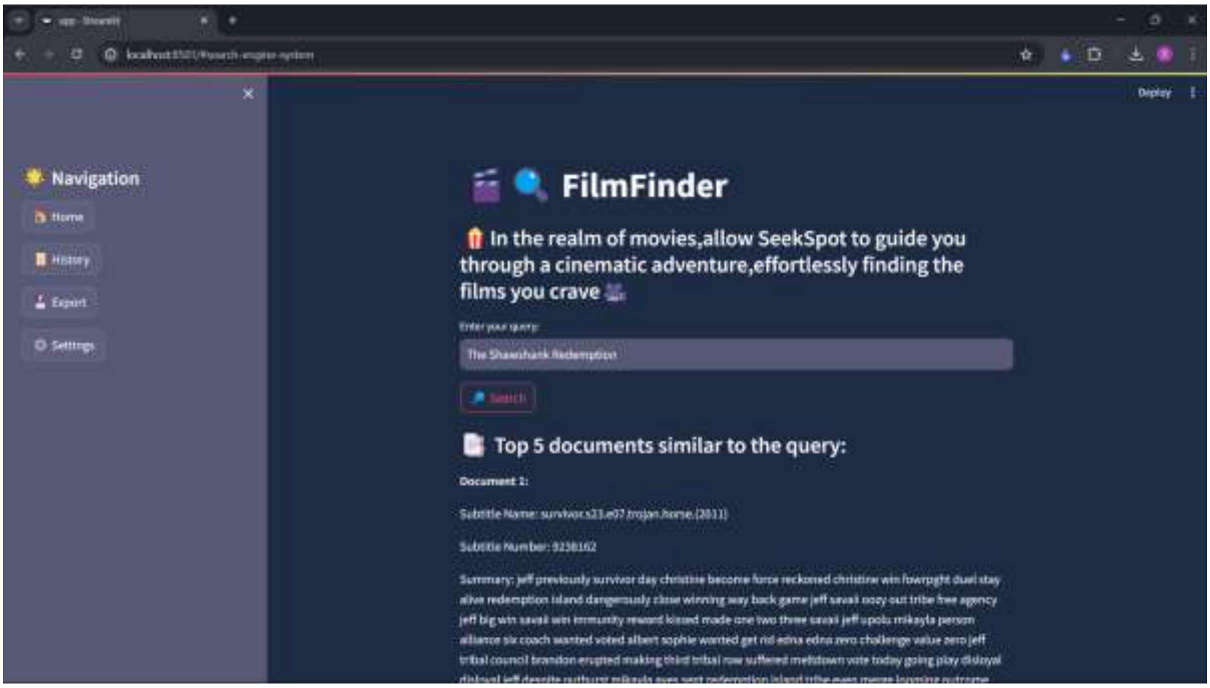
Search Feature: Enables users to input queries and initiate searches. Upon receiving a query, it retrieves similar movie documents using the cosine similarity method and presents the top 5 results.

Export Functionality: The export_data function placeholder is set to save chat data to a file, though it currently lacks actual export capabilities.

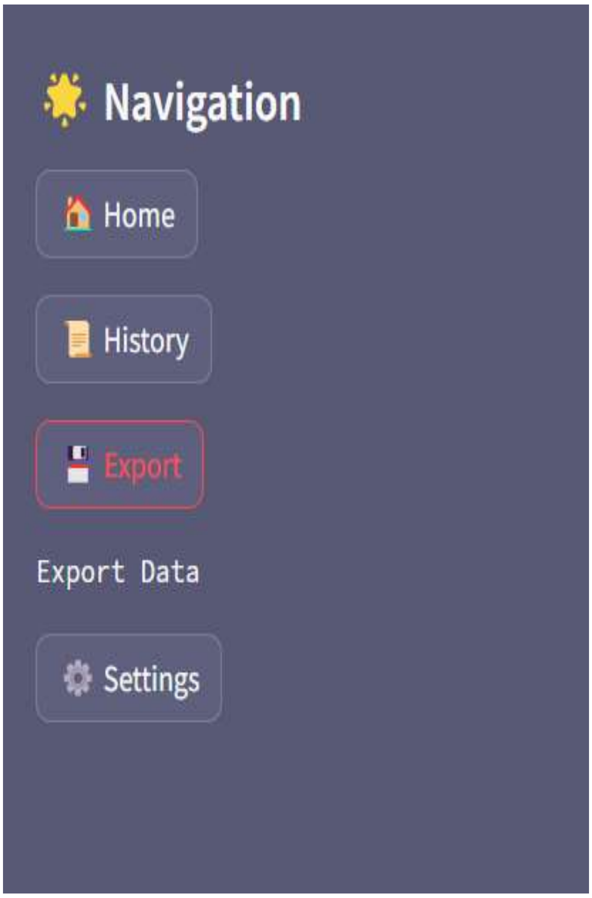
7.Retrieving Documents using user input Application:



After the Search



Nagivations



Conclusion:

In conclusion, "FilmFinder" is a Streamlit web application designed to simplify the process of finding movies based on user queries. By leveraging machine learning techniques such as text vectorization and cosine similarity, users can input their queries and receive a curated list of similar movie documents. The application offers intuitive navigation through a sidebar menu, allowing users to explore search history, export data, and customize settings. While the current version lacks full export functionality, it provides a solid foundation for further development and enhancement. Overall, "FilmFinder" demonstrates the potential of leveraging modern technologies to streamline the movie search experience for users.