**Enhancing Search Engine Relevance for Video Subtitles**

**By: SAIPRANEETH ADIGOPULA**

**SANJANA KUDUDULA**

Table of Content

**[Abstract: 1](#_Toc10950)**

[1. Read the given data: 1](#_Toc19846)

[2. Cleaning: 4](#_Toc30141)

[3. Document chunking: 5](#_Toc21160)

[4. Vectorize the given Subtitle Documents: 6](#_Toc26211)

**[4 (b). Advantages: 7](#_Toc27180)**

**[5. Store embeddings in a ChromaDB database: 8](#_Toc9876)**

[6. Retrieving Documents 9](#_Toc19457)

[7. Retrieving Documents using user input Application: 10](#_Toc30204)

**[8. Conclusion: 13](#_Toc27319)**

Abstract**:**

In the rapidly evolving landscape of digital content consumption, accessibility to video content is increasingly important. This project addresses the need for enhancing video accessibility by improving the search relevance of subtitles, particularly focusing on the content within subtitles. The objective is to develop an advanced search engine algorithm that efficiently retrieves subtitles based on user queries, emphasizing natural language processing and machine learning techniques to enhance relevance and accuracy.

This project uses between semantic search engines over traditional keyword-based search engines. As keyword-based engines rely on exact matches between user queries and indexed documents, semantic search engines delve deeper into understanding the meaning and context of queries and documents.

The core logic involves three key steps: data preprocessing, vectorization of subtitle documents and user queries, and cosine similarity calculation. Data preprocessing includes cleaning steps such as removing timestamps to ensure accurate analysis. Vectorization converts textual data into numerical representations, facilitating comparison. Cosine similarity calculation measures the relevance of documents to the user's query based on vector representations.

By employing semantic search techniques, this project aims to provide more meaningful and relevant search results for video subtitles, enhancing the accessibility of subtitle content for all users.

## Read the given data:

Database contains a sample of 82498 subtitle files from opensubtitles.org. Most of the subtitles are of movies and tv-series which were released after 1990 and before 2024.

Database File Name: eng\_subtitles\_database.db

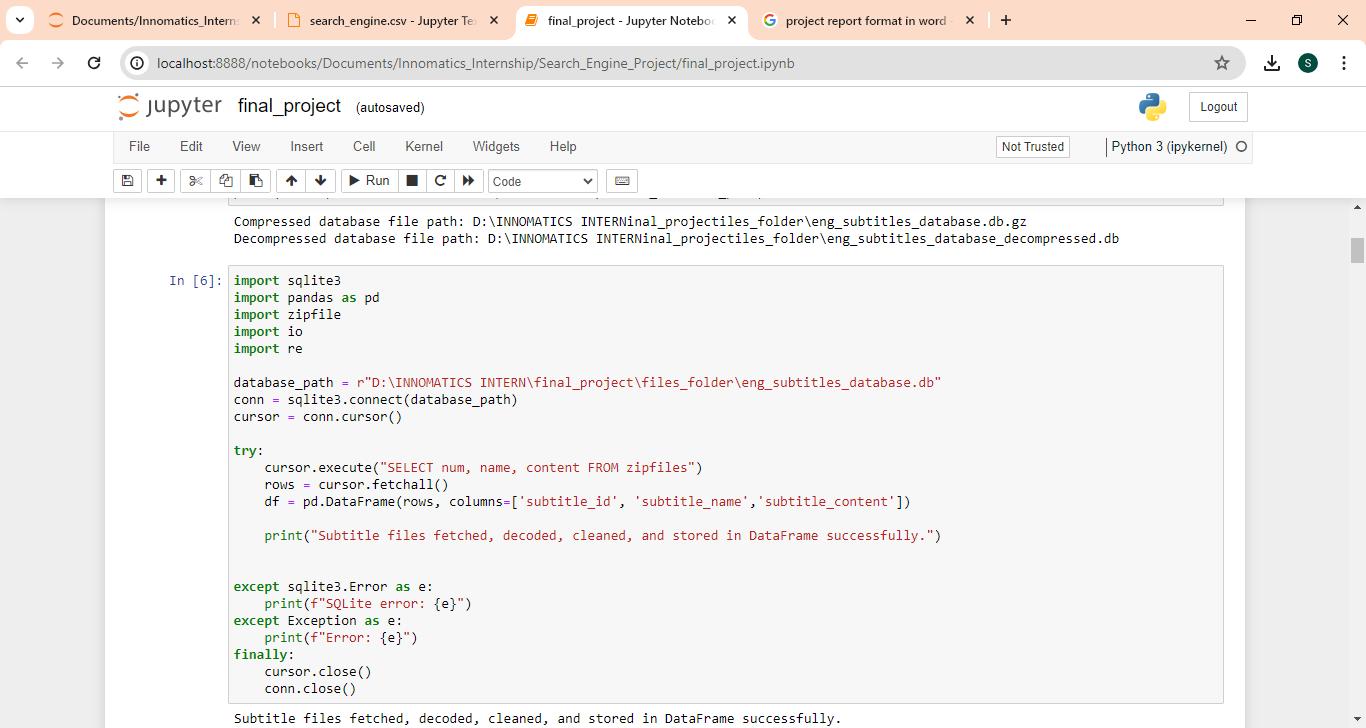
Database contains a table called 'zipfiles' with three columns.

1. num: Unique Subtitle ID reference for www.opensubtitles.org

2. name: Subtitle File Name

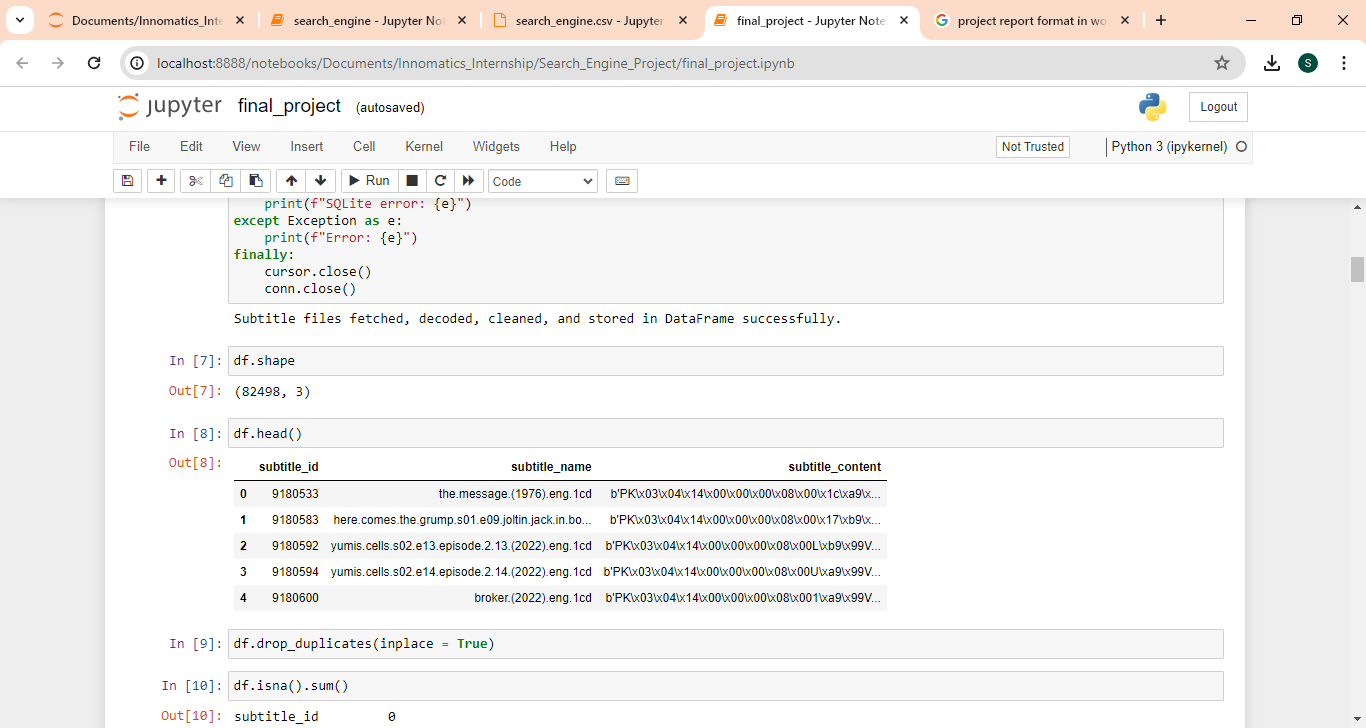
3. content: Subtitle file were compressed and stored as a binary using 'latin-1' encoding.

You can use 'num' to get more details about each subtitle by going to the following link: [https://www.opensubtitles.org/en/subtitles/{num}](https://www.opensubtitles.org/en/subtitles/%7bnum%7d) \*\*Replace {num} with Unique Subtitle ID.

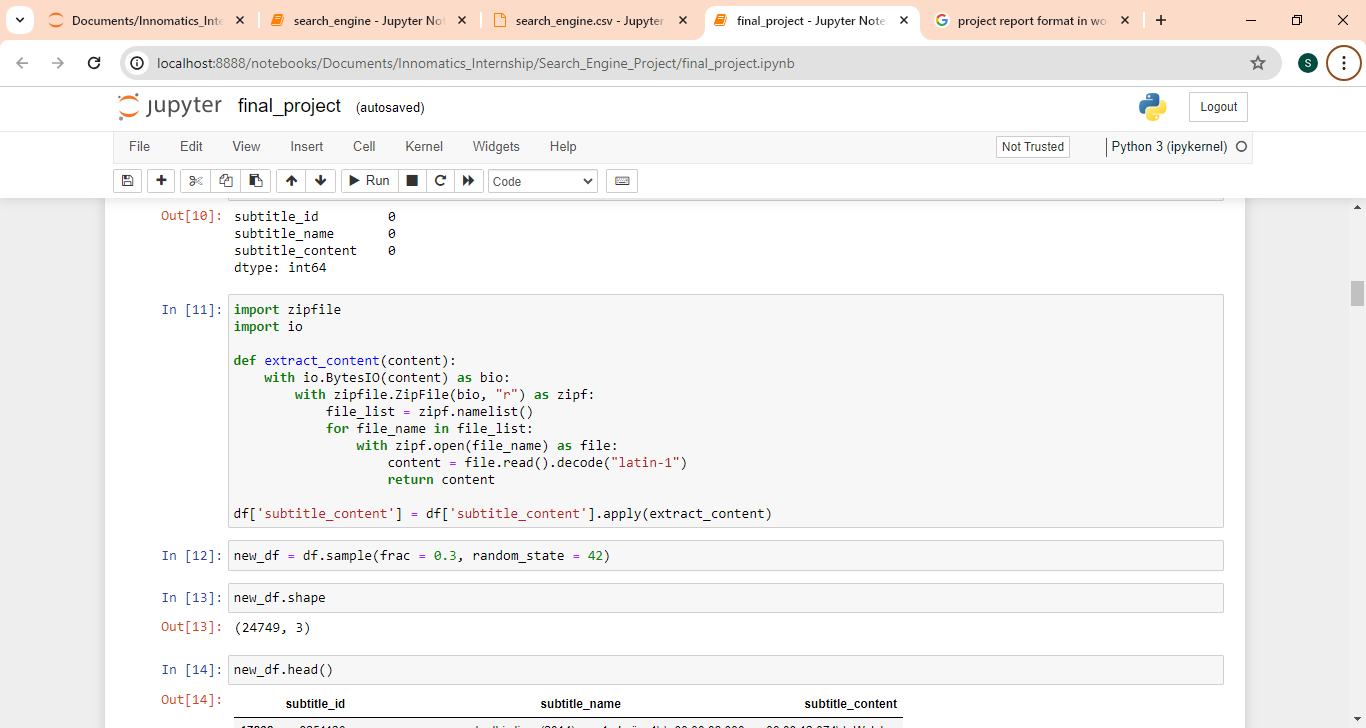


This Python script demonstrates the process of fetching subtitle data from a SQLite database, decoding it, cleaning it, and storing it in a Pandas DataFrame.

* 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.
* The fetched rows from the query result are stored in variable named rows.
* Using the Pandas library, a DataFrame named df is created from the fetched rows. The DataFrame consists of three columns: "subtitle\_id", "subtitle\_name", and "subtitle\_content", corresponding to the subtitle data fetched from the database.
* Upon successful retrieval, decoding, cleaning, and storage of subtitle data in the DataFrame, a success message is printed to indicate the completion of the process.
* Exception handling is implemented to catch any potential errors during the execution of the script. This includes handling SQLite errors (sqlite3.Error) and other general exceptions (Exception).
* Finally, the SQLite cursor and connection are closed using the cursor.close() and conn.close() functions respectively, ensuring proper cleanup and release of database resources.

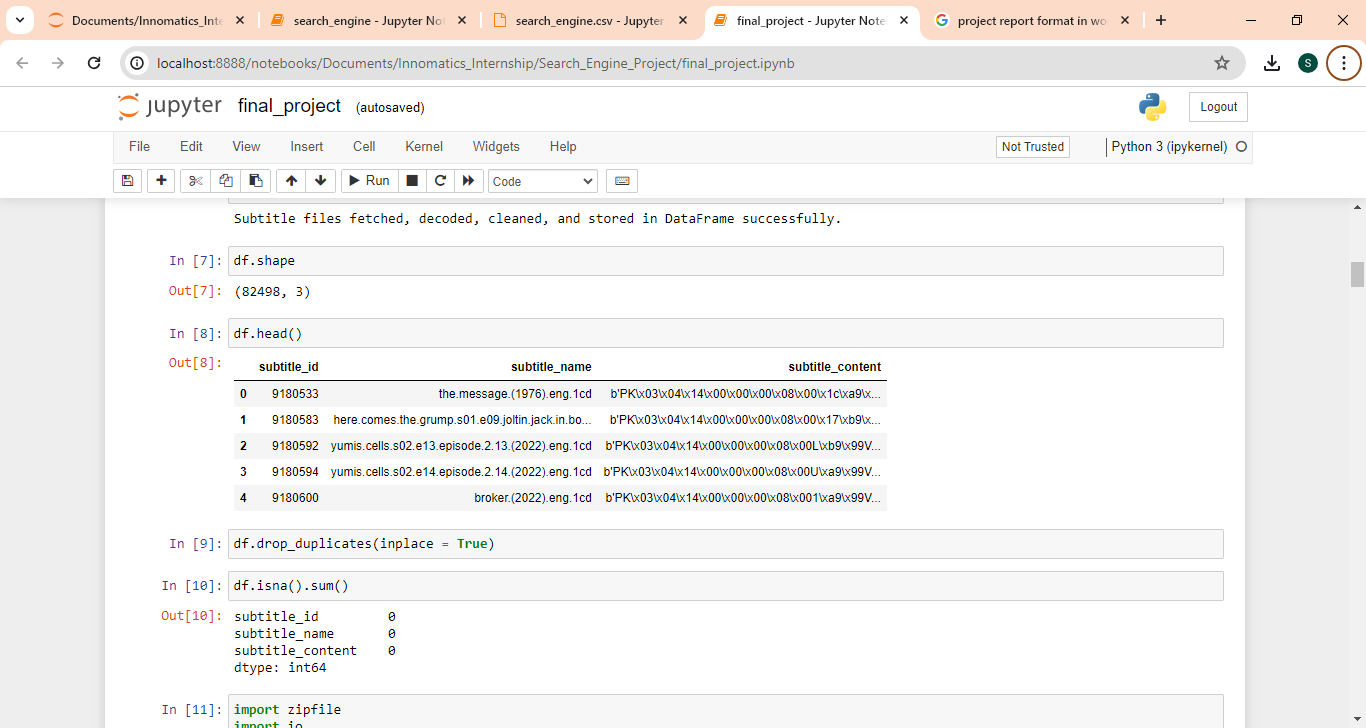


After fetching the data from the SQLite database and storing it in a Pandas DataFrame. The DataFrame df has 82,498 rows and 3 columns.subtitle\_id: Represents the unique identifier for each subtitle. subtitle\_name: Contains the name or title of each subtitle file. subtitle\_content: Holds the content or text of each subtitle. The subtitle\_content column contains the binary content of the subtitle files, as indicated by the prefix 'b'. This content likely needs further processing or decoding to extract meaningful text information.

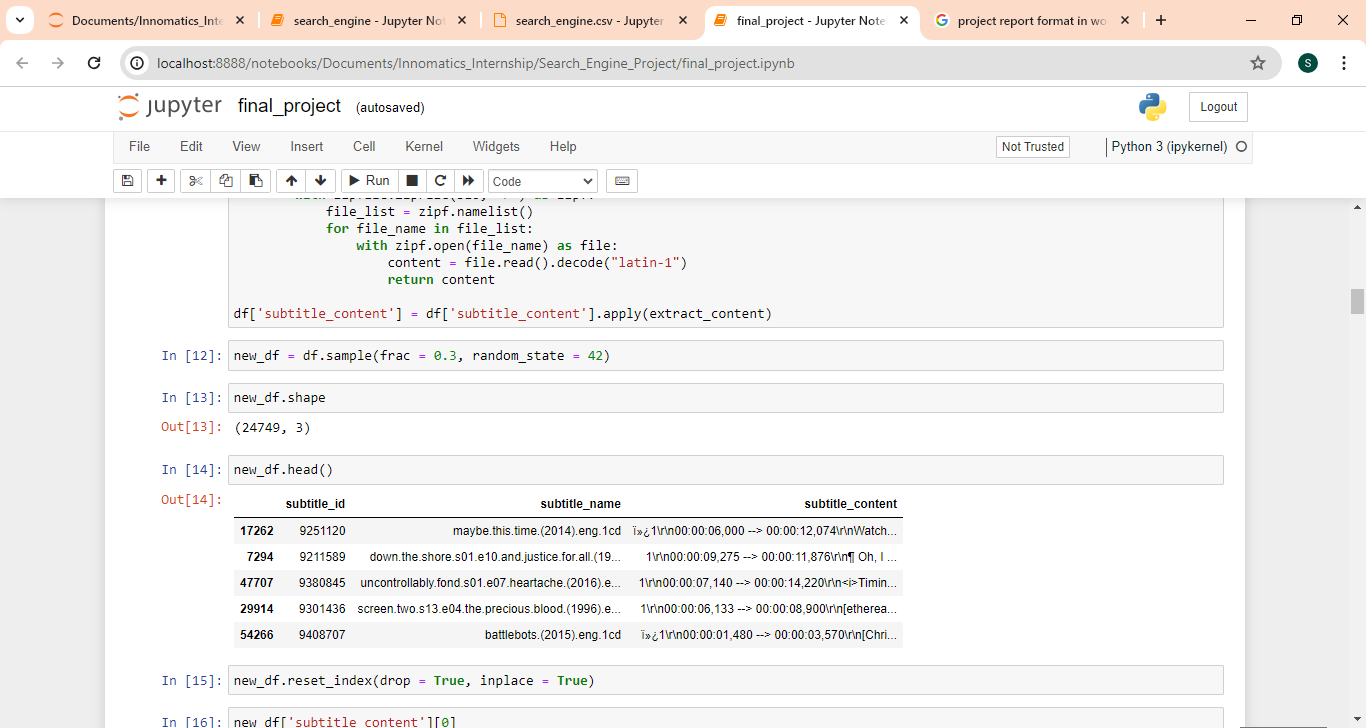


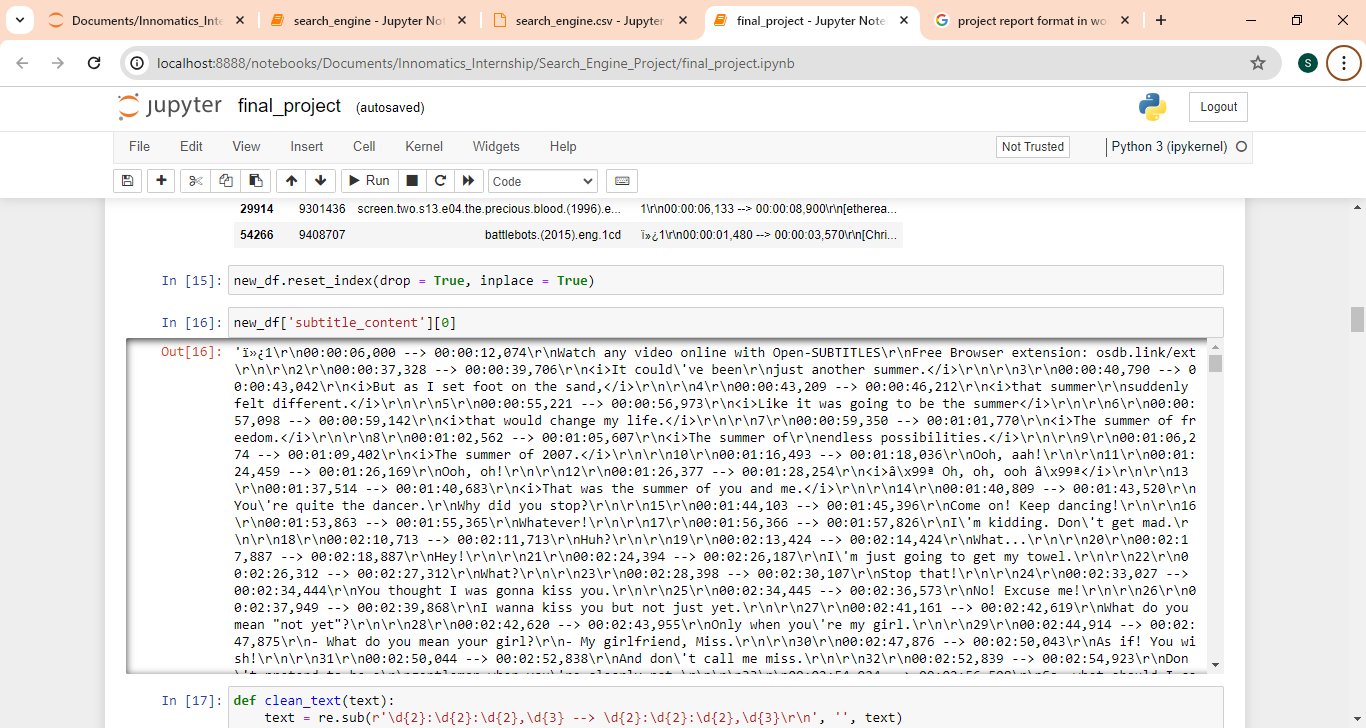
### 1(a). Unzipping the content and decoding using latin-1

The extract\_content function is defined with one parameter, content, representing the binary content of a zip archive containing subtitle files. The function utilizes the io.BytesIO class to create an in-memory binary stream (bio) from the provided content. Within the context of a zipfile.ZipFile object (zipf), created from the binary stream, the function iterates through the list of file names (file\_list) contained in the zip archive. For each file in the archive, the function opens it (with zipf.open(file\_name) as file) and reads its content (content = file.read().decode("latin-1")). The content is decoded using the "latin-1" encoding, assuming it represents text data. The extracted content of the first file encountered in the zip archive is returned. The extract\_content function is applied to the subtitle\_content column of the DataFrame df using the apply method. This enables the function to be executed on each row of the Data Frame, extracting the content of zip archives stored in the subtitle\_content column.



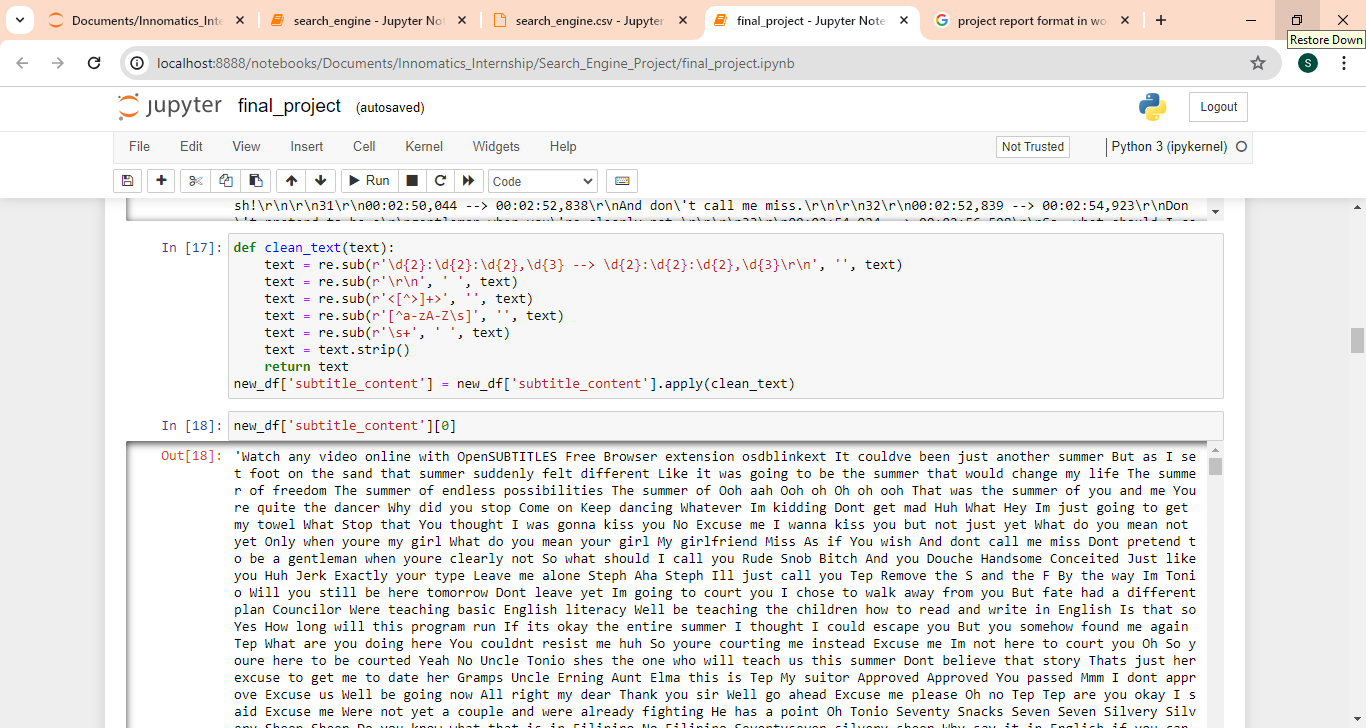
Working with a Data Frame of shape (824925, 3) where one of the row contains more than 100,000 tokens can consume a significant amount of computer resources, especially in terms of memory and processing power. Because of limited compute resources, only random 30% of the data is used.





## Cleaning:

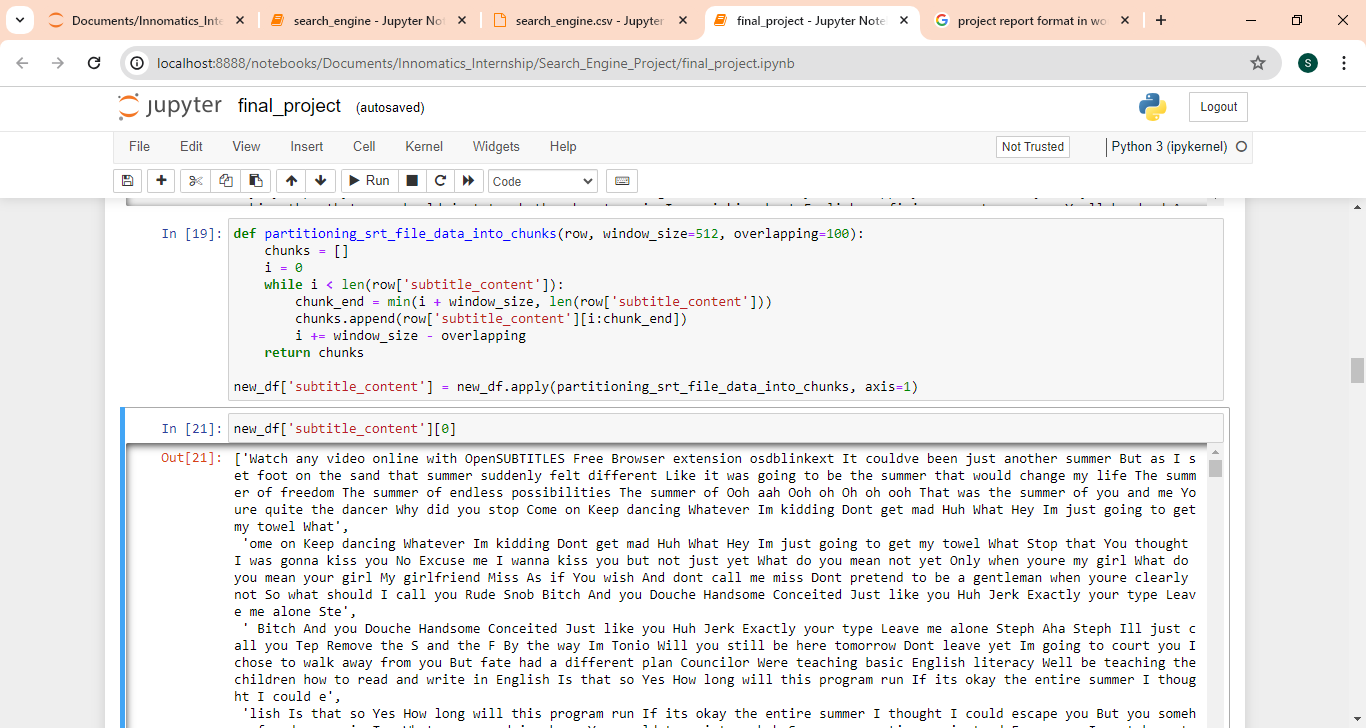
Text data is a subtitle contect which has time stamps, HTML tags and lots of noises. Cleaning has to be done on the text data, as cleaning text data is an essential preprocessing step in natural language processing (NLP) tasks, especially before vectorization. Cleaning the text aims to remove noise, irrelevant information, and inconsistencies from the text, making it more suitable for downstream processing.



1. The code defines a function clean\_text(text) that applies several cleaning steps to the text data:
2. re.sub(r'\d{2}:\d{2}:\d{2},\d{3} --> \d{2}:\d{2}:\d{2},\d{3}\r\n', '', text) This regular expression removes timestamps in the format "hh:mm:ss,mmm --> hh:mm:ss,mmm" along with the carriage return and newline characters (\r\n).
3. re.sub(r'\r\n', ' ', text) This line replaces carriage return and newline characters (\r\n) with a space. This step helps ensure that the text remains coherent and well-structured after removing the timestamps.
4. re.sub(r'<[^>]+>', '', text) This regular expression removes HTML tags from the text. HTML tags are often present in text data obtained from web sources and may not carry meaningful information for NLP tasks.
5. re.sub(r'[^a-zA-Z\s]', '', text) This line removes any characters that are not alphabetic or whitespace. It effectively eliminates punctuation and special characters from the text, retaining only letters and spaces.
6. re.sub (r'\s+', ' ', text) This regular expression replaces multiple consecutive whitespace characters with a single space. It helps normalize the text and ensures consistent spacing between words.
7. text.strip() Finally, this line removes leading and trailing whitespace from the text.

## Document chunking:

As the text documents are very large, Consider the challenge of embedding large documents there will be Information Loss. The document chunking technique is used as a crucial step in handling large documents effectively, particularly in scenarios where embedding entire documents as single vectors is impractical due to the risk of information loss or computational constraints.



* Embedding entire long documents as single vectors can lead to information loss due to their diverse topics, sentiments, and themes, necessitating significant computational resources.
* Addresses this by dividing large documents into smaller, manageable chunks for individual processing and embedding, improving efficiency.
* Overlapping windows are used to prevent splitting important text between chunks. Chunks are based on a token window size, ensuring each contains just below a specified number of tokens, and overlap to retain context between adjacent chunks.

## 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. While they are widely used and can be effective for certain tasks, they also have some disadvantages:

##### 4 (a). Disadvantages of BOW / TF-IDF:

No Semantic Information: BOW and TF-IDF representations do not capture semantic relationships between words or documents. They treat each word independently and ignore the context in which words appear. This can lead to limitations in capturing the true meaning or intent of the text.

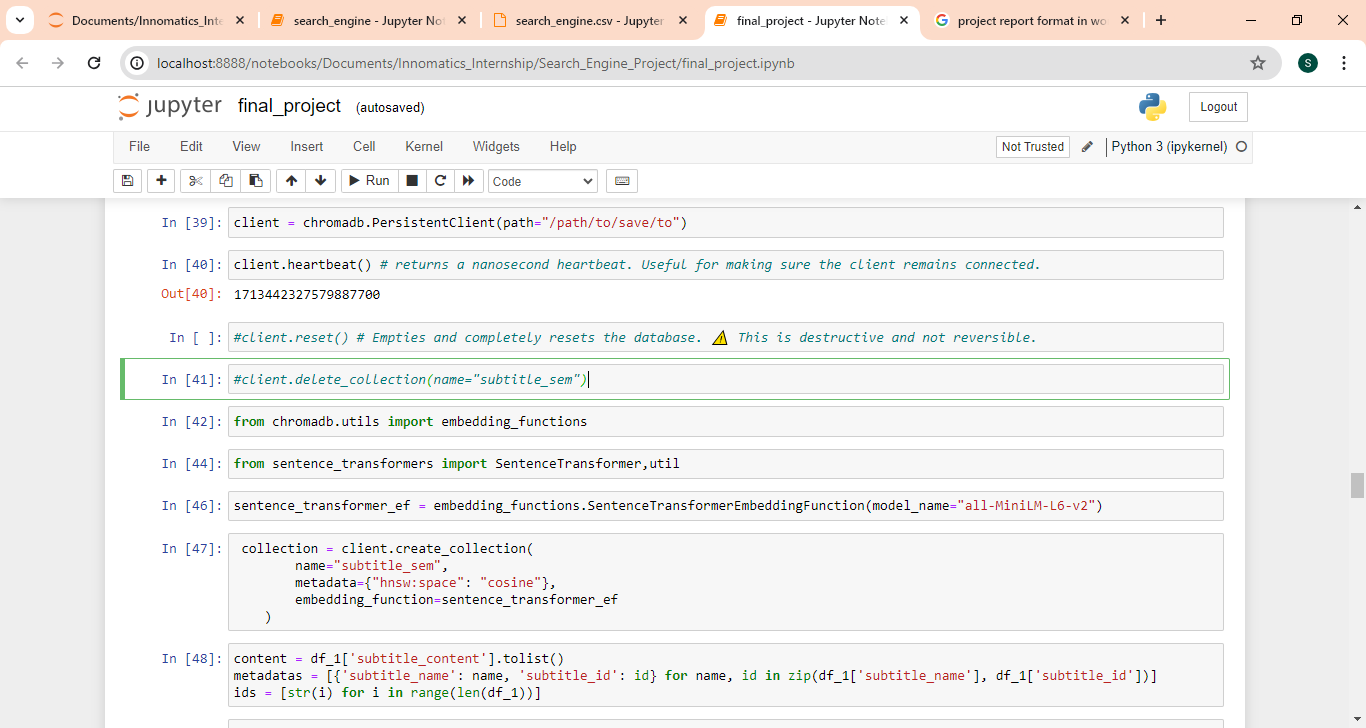
High Dimensionality: BOW and TF-IDF representations result in high-dimensional sparse vectors, especially for large vocabularies or datasets with many unique tokens. This high dimensionality can lead to computational inefficiencies and increased memory requirements.

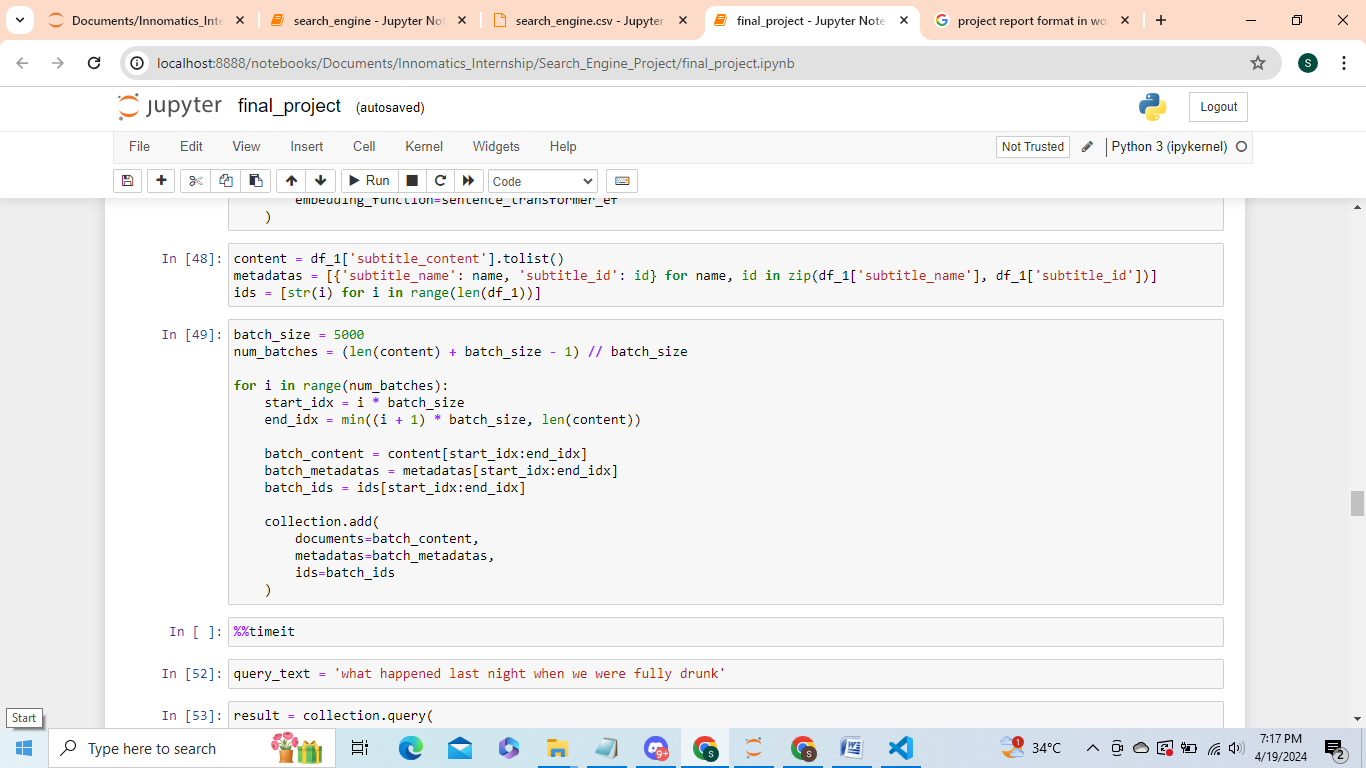
Given these disadvantages, using BOW or TF-IDF for text vectorization may not be ideal for tasks that require capturing semantic information or understanding the context of the text. Instead, for tasks like building a Semantic Search Engine, it's more beneficial to use methods that encode semantic information and capture contextual relationships between words and documents.

Usage of BERT-based Sentence Transformers: BERT-based Sentence Transformers, such as those provided by the Sentence Transformers library, offer a solution to the limitations of BOW and TF-IDF. These models leverage pre-trained language representations, such as BERT, to generate dense vector embeddings that encode semantic information and capture contextual relationships between words and sentences. By using deep contextualized embeddings, Sentence Transformers can produce representations that better reflect the meaning and intent of the text.

4 (b). Advantages:

* Semantic Information: BERT-based Sentence Transformers capture semantic relationships between words and sentences, allowing for more nuanced understanding and interpretation of the text.
* Contextual Embeddings: Unlike BOW and TF-IDF, which treat each word independently, Sentence Transformers generate dense embeddings that take into account the context in which words appear. This helps preserve contextual information and improve the quality of representations.
* Lower Dimensionality: Dense embeddings produced by Sentence Transformers typically have lower dimensionality compared to sparse representations, reducing computational overhead and memory requirements.
* Pre-trained Models: Sentence Transformers leverage pre-trained models like BERT, which have been trained on large text corpora, capturing a wide range of linguistic patterns and semantic relationships.



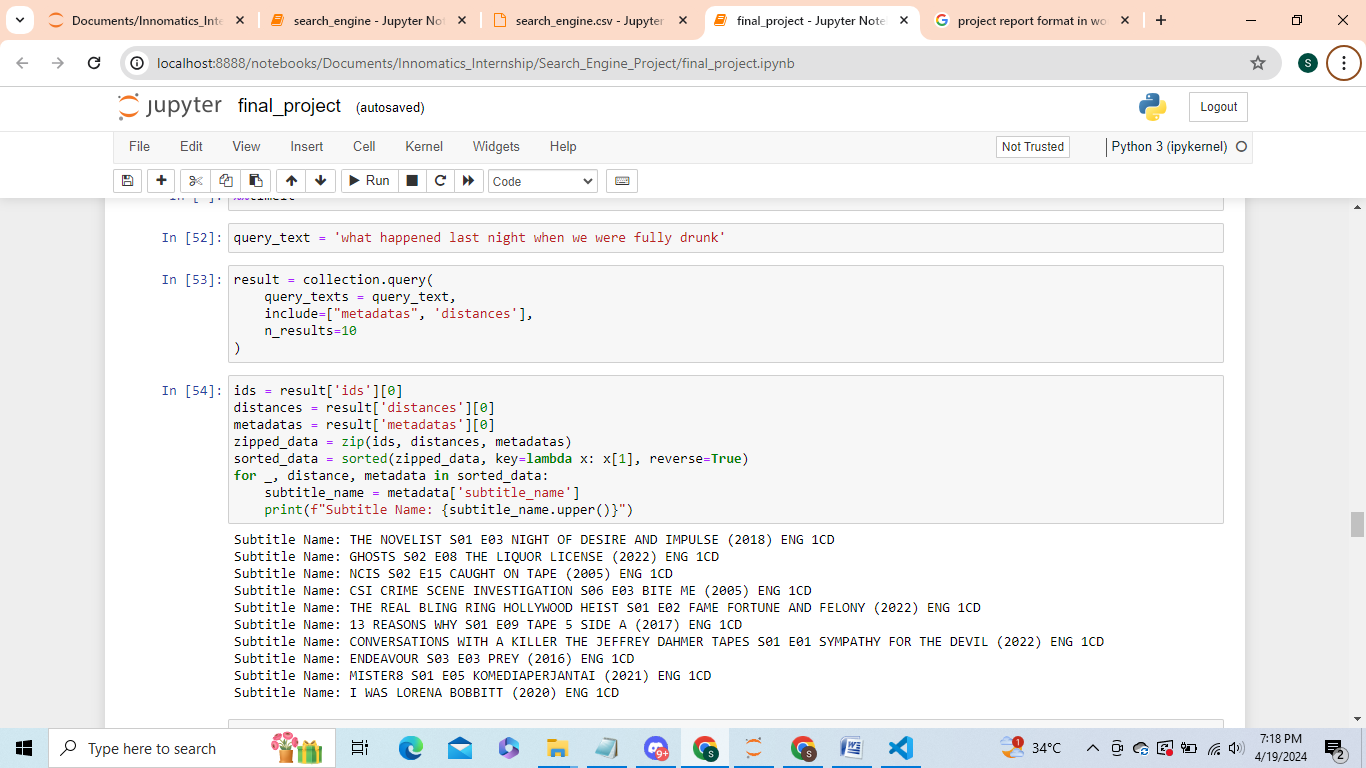


1. Store embeddings in a ChromaDB database**:**

ChromaDB is used for storing text embeddings because of its specialized features and capabilities that make it well-suited for managing and querying high-dimensional embeddings efficiently. The process of creating and populating a collection in the ChromaDB database with embeddings generated using a Sentence Transformer model. Here's a brief explanation of each step:

* Import necessary libraries including chromadb for interacting with the ChromaDB database, SentenceTransformer for generating sentence embeddings, and embedding\_functions from chromadb.utils for defining the embedding function.
* It initializes a persistent client for interacting with the ChromaDB database, specifying the path where the database will be saved.
* An embedding function named ‘sentence\_transformer\_ef’ is defined using the SentenceTransformerEmbeddingFunction class from embedding\_functions. This function utilizes a pre-trained SentenceTransformer model named "all-MiniLM-L6-v2" to generate embeddings for input sentences. "all-MiniLM-L6-v2" refers to a specific variant of the MiniLM model, which is optimized for speed, performance, and efficiency.
* A collection named "subtitle\_sem" is created within the ChromaDB database. The collection is configured with metadata specifying the space as "cosine" to indicate cosine similarity for nearest neighbor search, and the defined embedding function sentence\_transformer\_ef is associated with the collection.
* The text content of subtitles (df\_1['subtitle\_content']) and corresponding metadata (df\_1['subtitle\_name'] and df\_1['subtitle\_id']) are retrieved from DataFrame df\_1. The content and metadata are then split into batches for efficient insertion into the collection.
* The data is inserted into the collection in batches to avoid memory constraints and optimize performance. Each batch consists of a subset of the subtitle content, metadata, and unique IDs.The insertion process is repeated for each batch until all subtitle content, metadata, and IDs are processed.

## Retrieving Documents



The process of retrieving documents based on a user's search query involves several key steps process of performing a semantic search using embeddings stored in a ChromaDB collection.

The user's search query is taken as input. In this case, the query text is "what happened last night when we were fully drunk".

While performing a query in the ChromaDB collection, query text is provided directly to ChromaDB, already a defined embedding function sentence\_transformer\_ef is associated with the collection , ChromaDB internally uses the embedding function to generate embeddings for the query text.

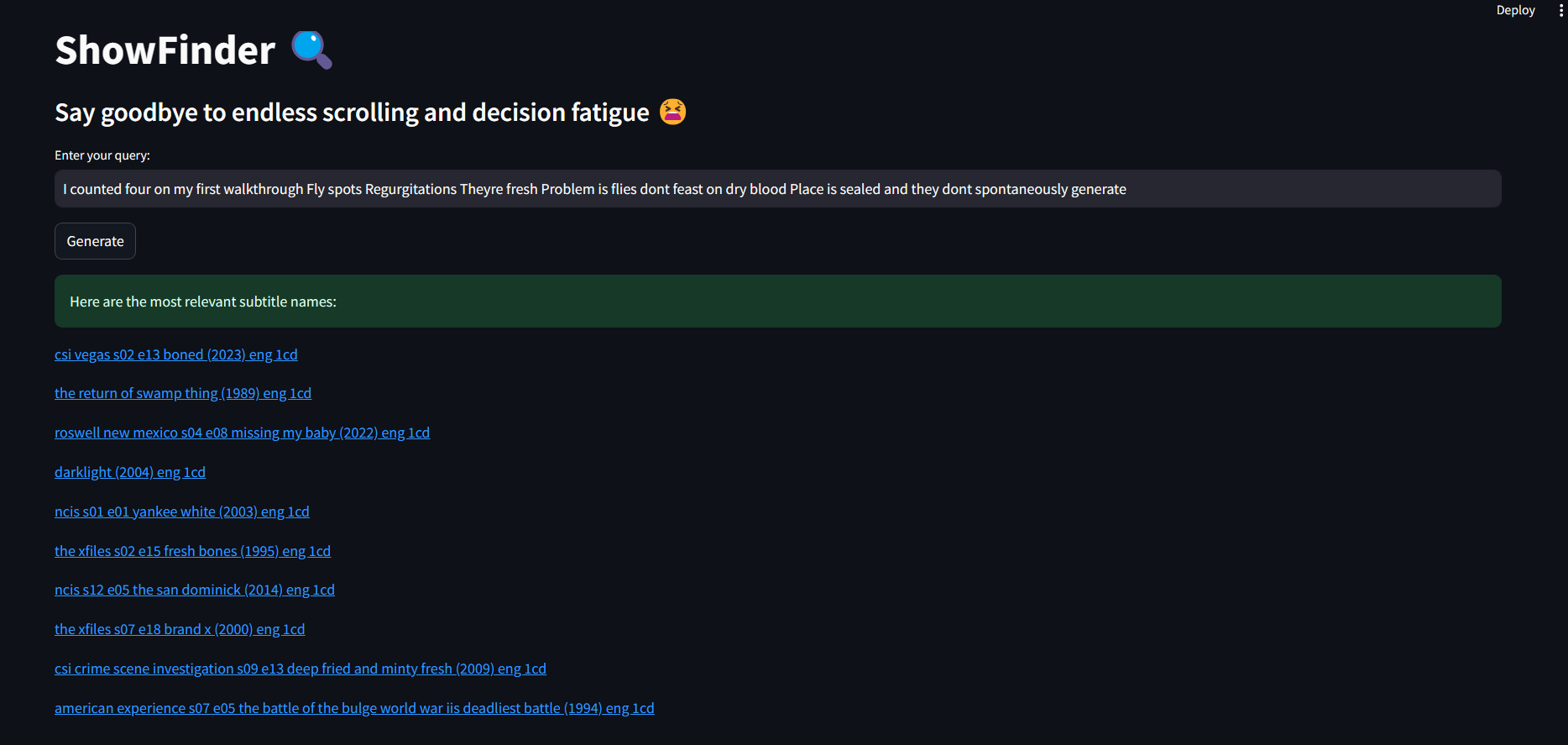
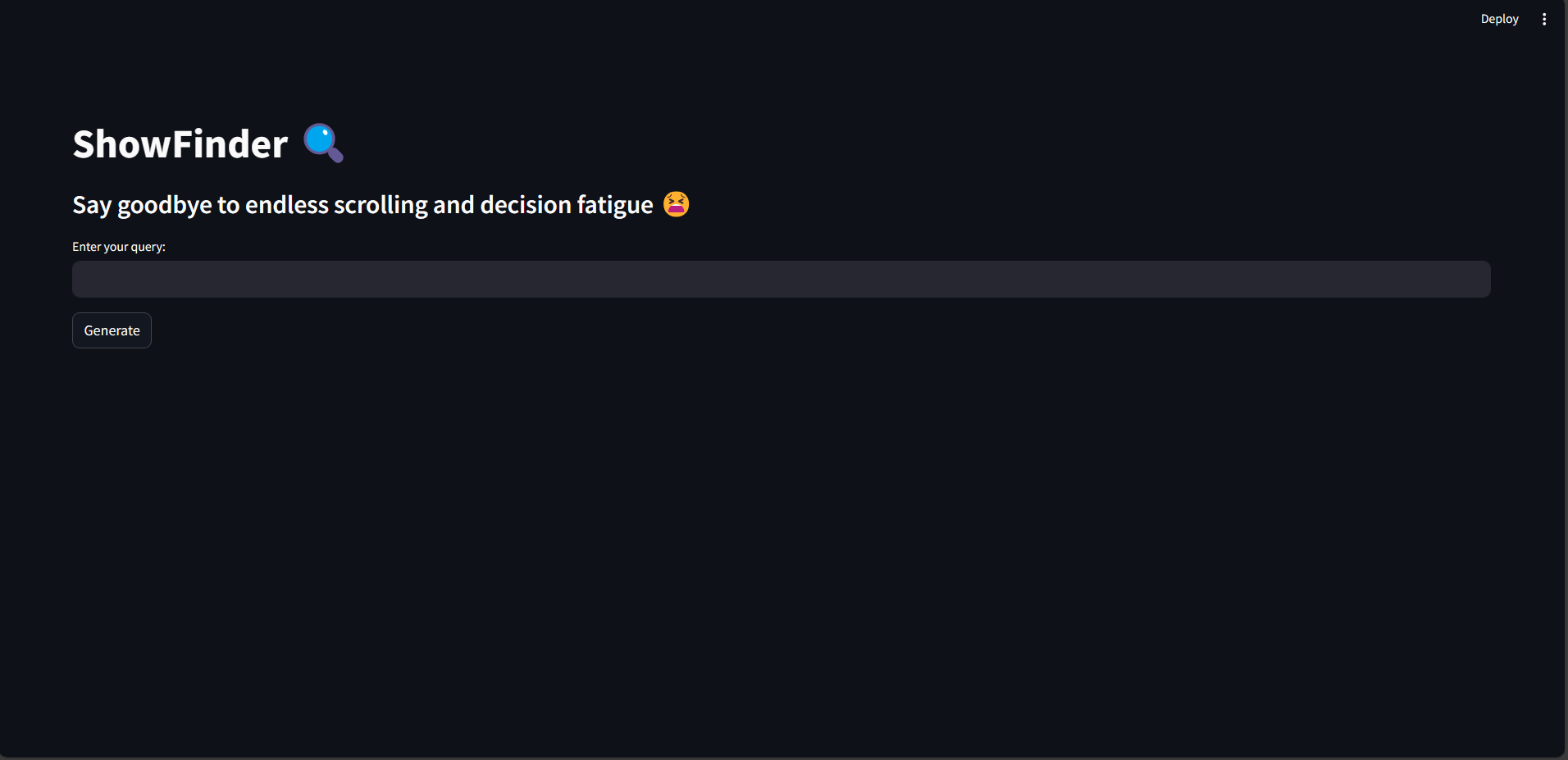
ChromaDB then calculates the similarity scores between the query embedding and the embeddings of documents stored in the collection using a cosine distance as similarity measure.

The cosine similarity scores are used to rank the candidate documents based on their relevance to the user's search query. The code retrieves the top 10 most relevant documents from the collection, along with their corresponding distances (cosine similarity scores) and metadata.

The sorted list of candidate documents, along with their metadata (e.g., subtitle name), is printed to the console. The documents are sorted based on their cosine similarity scores in descending order, with the most relevant documents appearing first.

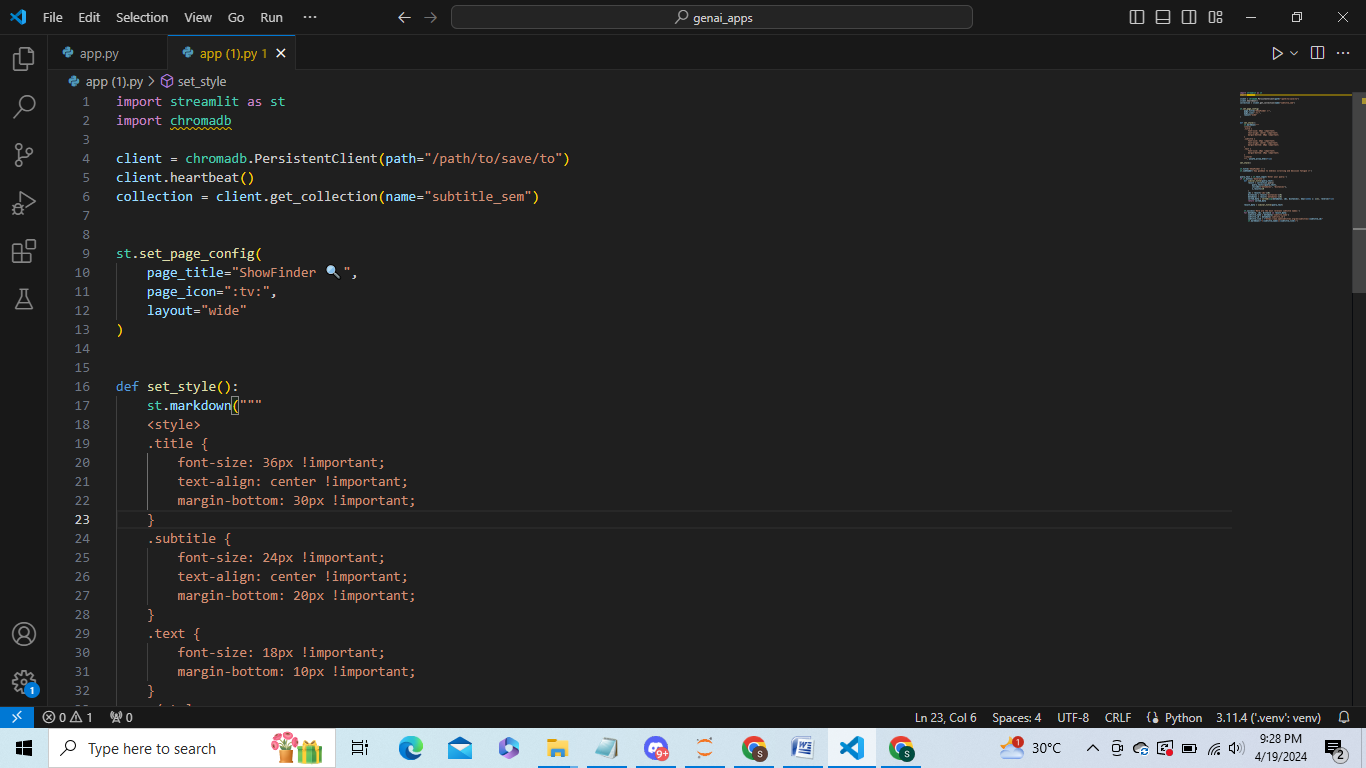
This approach allows you to leverage the pre-trained embedding model directly within ChromaDB, simplifies the process of performing similarity searches without the need to generate embeddings externally.

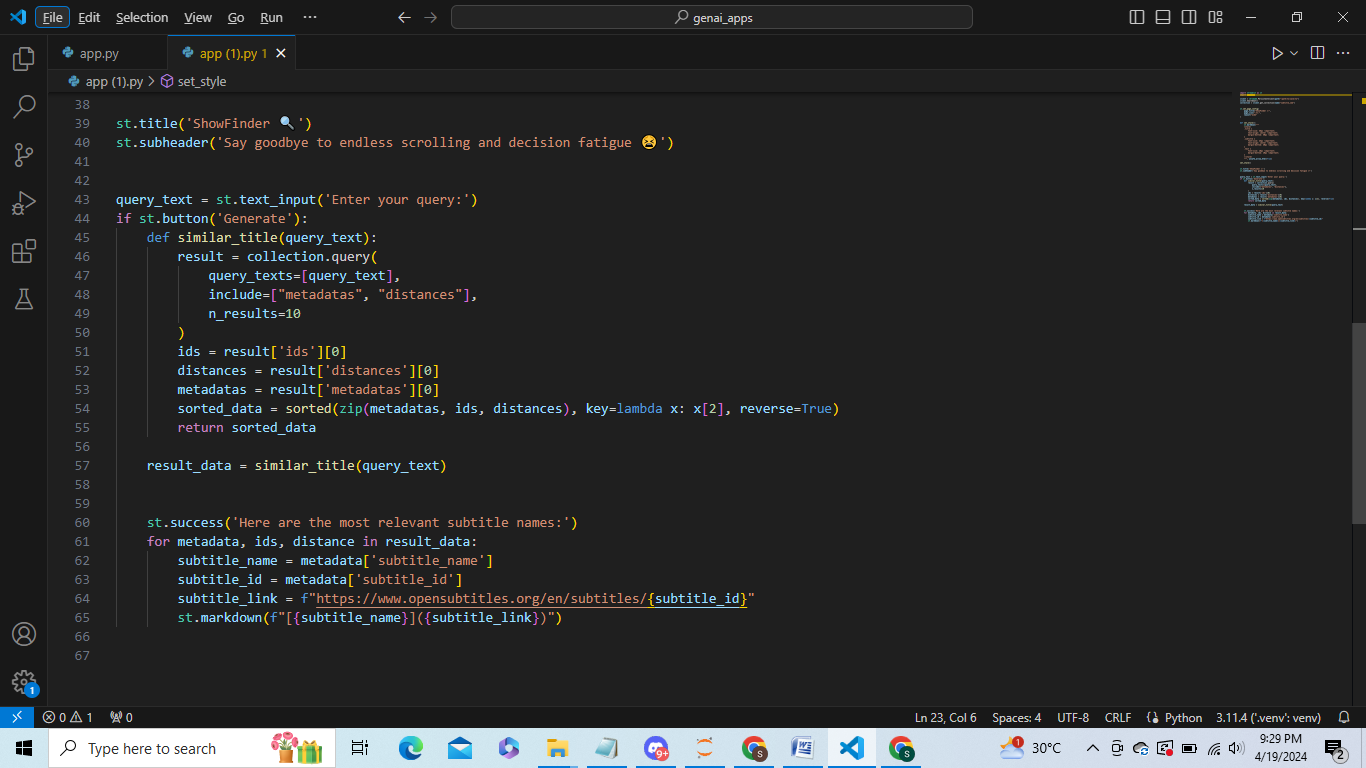
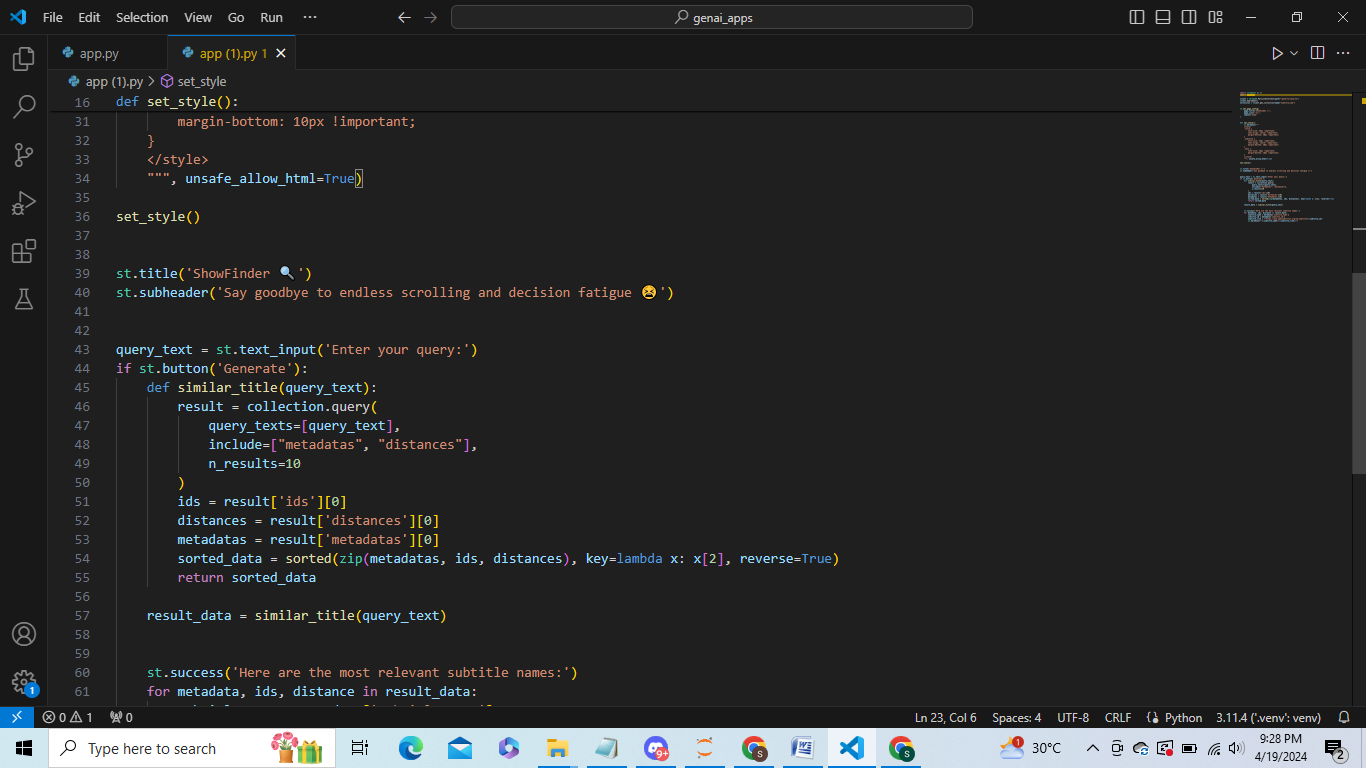
## Retrieving Documents using user input Application:



This Streamlit app, titled "ShowFinder," is designed to help users find relevant subtitles based on a search query.

Backend application code:





* The app imports necessary libraries including streamlit for creating web applications and chromadb for interacting with the ChromaDB database.
* A persistent client for interacting with ChromaDB is initialized, specifying the path where the database will be saved.
* Streamlit page configuration settings are set, including page title, icon, and layout.
* Custom CSS styling is applied to the app using Streamlit's set\_page\_config and markdown functions to adjust font sizes and alignment for titles, subtitles, and text.
* The UI elements are defined using Streamlit's st functions. Users can input their search query using a text input field, and a button allows them to trigger the search.
* When the user clicks the "Generate" button, the search query is passed to the similar\_title function, which queries the ChromaDB collection for similar subtitles based on the query text. The results include metadata such as subtitle name and ID, along with the cosine similarity distances.
* The most relevant subtitle names are displayed using Streamlit's st.success and st.markdown functions. Each subtitle name is presented as a clickable link that redirects to the corresponding subtitle page on OpenSubtitles.org.
* Overall, this Streamlit app provides an intuitive interface for users to search for subtitles based on their queries, leveraging embeddings stored in the ChromaDB collection for efficient and accurate search results.

1. Conclusion**:**

In conclusion, the documentation encompasses essential aspects of working with subtitle data, NLP models, and search engine development. It outlines techniques for retrieving and preprocessing subtitle data from a SQLite database, as well as extracting content from zip archives containing subtitle files. The choice of the "all-MiniLM-L6-v2" model for embedding generation is based on considerations of accuracy, speed, performance, model size, and compatibility. Additionally, the integration of NLP techniques into a search engine framework facilitates efficient information retrieval based on user queries. Overall, the documentation provides a comprehensive guide for developing NLP-based search systems tailored to subtitle data, enabling effective extraction of insights from text data in the context of video content.