Enhancing Search Engine Relevance for Video Subtitles

Submitted to

Innomatics Research Labs

By

Sanhita Thangallapalli

**BACKGROUND:**

In today's digital world, it's important for people to easily find the information they need, especially when it comes to video content like movies and online videos. However, searching for specific content within videos, such as subtitles, can sometimes be challenging. This project aims to solve that problem by creating a smart search engine specifically for video subtitles.

**OBJECTIVE:**

The main goal of this project is to make it easier for users to find subtitles for videos by developing an advanced search algorithm. Unlike typical search engines that focus on things like titles and keywords, our algorithm will pay more attention to the actual content of the subtitles. We'll use advanced techniques from natural language processing and machine learning to make sure our search results are accurate and relevant to what users are looking for.

**DATA DETAILS:**

The database provided contained a sample of 82,498 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.

**i. Num:** Unique Subtitle ID reference for www.opensubtitles.org

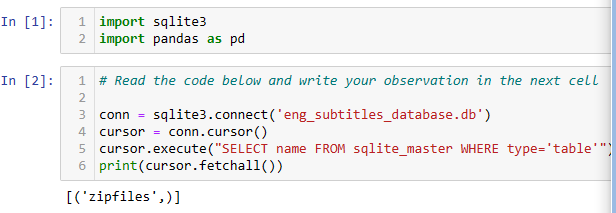
**ii. Name:** Subtitle File Name

**iii**. **Content:** Subtitle files were compressed and stored as a binary using 'latin-1' encoding.

**PART 1: IMPORTING OF DATA:**

Firstly we need to extract the data from the database “ eng\_subtitles\_database.db”. Here this database contains a table called ‘zipfiles’ with three columns. Namely ‘Num’,’Name’,’Content’.To extract the code below are the steps involved:

1. The code is using the **sqlite3 module in Python**, which provides a lightweight disk-based database. The code establishes a connection to an SQLite database file named **'eng\_subtitles\_database.db'** using **sqlite3.connect().** It creates a cursor object to interact with the database using **conn.cursor().** The **SQL query SELECT name FROM sqlite\_master WHERE type='table'** is executed to fetch the names of all tables in the **database.cursor.fetchall()** is used to fetch all the results from the executed query. Finally, the code prints the names of all the tables in the database.



1. **i. Retrieving Column Details:**The code executes a special command called "PRAGMA" to gather information about the columns in the 'zipfiles' table.

**ii. Fetching the Details:** After executing the PRAGMA command, all the information about the columns is fetched.

**iii. Displaying Column Names:** Then, it goes through each piece of information (each column) fetched from the PRAGMA query. For each column, the code prints its name.In the PRAGMA table\_info query, each result contains several details about the columns, but here, we are interested only in the name of each column. The column name is accessed using col[1].

**iv.** **Querying the 'zipfiles' Table:**

The code executes a SQL query to retrieve all the data from the **'zipfiles'** table in the SQLite database file **(eng\_subtitles\_database.db),** and stores the result in a DataFrame called ‘df’.

**v. Displaying the First Five Rows:** The **‘df.head()**’ command is used to display the first five rows of the DataFrame **‘df’**.



1. **i. Decompressing ZIP File Content:**

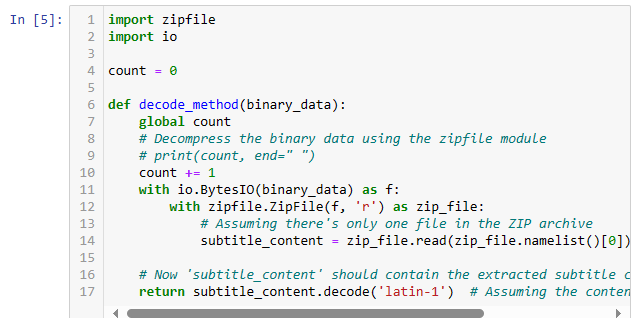
* The function **decode\_method()** is defined to decompress binary data that is assumed to be a ZIP file.
* It uses the **‘zipfile’** module to decompress the binary data.
* A counter, **count**, is incremented each time the function is called.

**ii. Decompression Process:**

* The binary data is treated as a ZIP archive and decompressed using the **‘zipfile.ZipFile()’** function.
* **‘io.BytesIO(binary\_data)’** is used to create a binary stream from the binary data.
* The extracted content is read from the ZIP file using **zip\_file.read(zip\_file.namelist()[0]).**This assumes there's only one file in the ZIP archive.

**iii. Text Decoding:**

* The decoded subtitle content is then returned as a string using .decode**('latin-1').**
* It assumes the content is UTF-8 encoded text. However, **'latin-1'** is used here instead of **'utf-8'**. There may be a reason for this. The original author might have chosen **'latin-1'** because it is more permissive in terms of character encodings, whereas **'utf-8'** might raise errors if it encounters characters it cannot interpret.



1. **i. Progress Bar Setup:**

* The from tqdm import tqdm line imports the tqdm library for displaying a progress bar.
* **tqdm.pandas()** is used to enable progress\_apply for pandas operations.

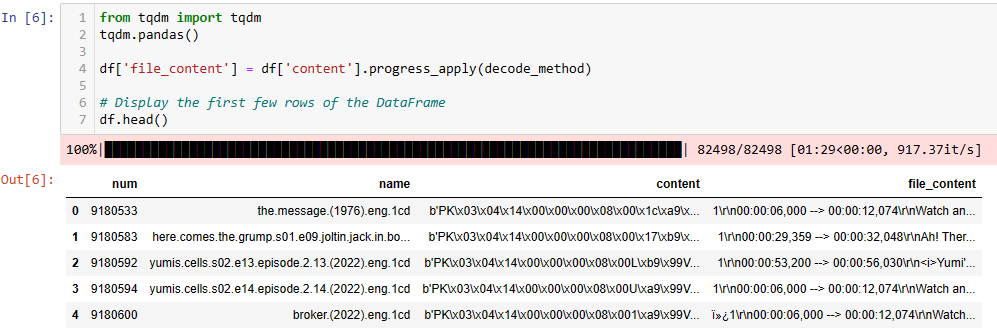
**ii. Applying the Decoding Function:**

* The **df['content'].progress\_apply(decode\_method)** line applies the **decode\_method** function to each element in the **'content'** column of the DataFrame **df.**
* The result of the decoding is stored in a new column called **'file\_content'**

**iii. Displaying the First Few Rows:**

* **df.head()** is used to display the first few rows of the DataFrame df.

This code applies the decode\_method function to each element in the 'content' column of the DataFrame df, storing the result in a new column called 'file\_content'. Then, it displays the first few rows of the DataFrame.



1. **i. Importing Necessary Libraries and Setting Up Progress Bar:**

* The code imports the **tqdm library** to set up a progress bar for the cleaning process.

**ii.Randomly Selecting 30% of the Rows:**

* **df\_sample = df.sample(frac=0.3)** randomly selects **30%** of the rows from the DataFrame df and assigns them to the DataFrame **df\_sample.**

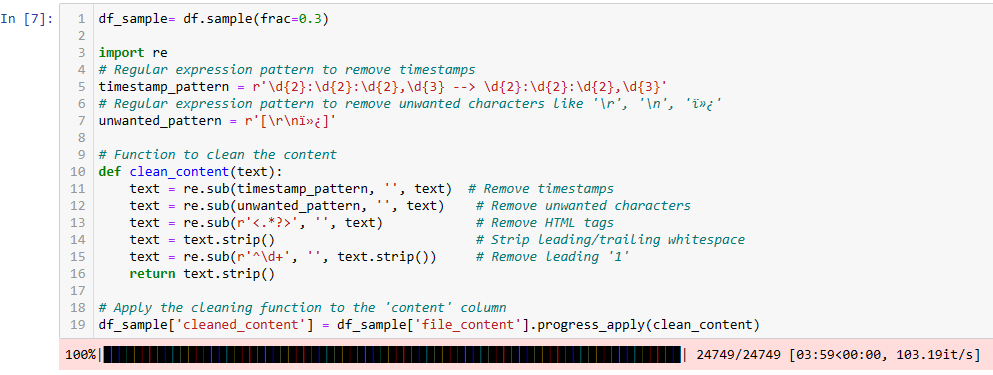
**iii.Importing the 're' Module for Regular Expressions:**

* The **re** module is imported to work with regular expressions.

**iv.Defining Regular Expression Patterns:**

* **Two regular expression patterns are defined:**

1. **timestamp\_pattern:** This pattern is used to remove timestamps from the content.
2. **unwanted\_pattern**: This pattern is used to remove unwanted characters such as '\r', '\n', and 'ï»¿' from the content.

****

**ii.Defining the Cleaning Function:**

* The **clean\_content()** function is defined to clean the content using regular expressions.
* The function performs the following tasks:

1. Removes timestamps from the content using **re.sub(timestamp\_pattern, '', text).**
2. Removes unwanted characters like '\r', '\n', and 'ï»¿' from the content using **re.sub(unwanted\_pattern, '', text).**
3. Removes HTML tags from the content using **re.sub(r'<.\*?>', '', text**).
4. Strips leading/trailing whitespace from the content using **text.strip().**
5. Removes a leading '1' from the text using **re.sub(r'^\d+', '', text.strip()).**

**iii.Applying the Cleaning Function:**

* The **clean\_content()** function is applied to the **'file\_content'** column of the DataFrame **df\_sample.**
* The cleaned content is stored in a new column called **'cleaned\_content'.**

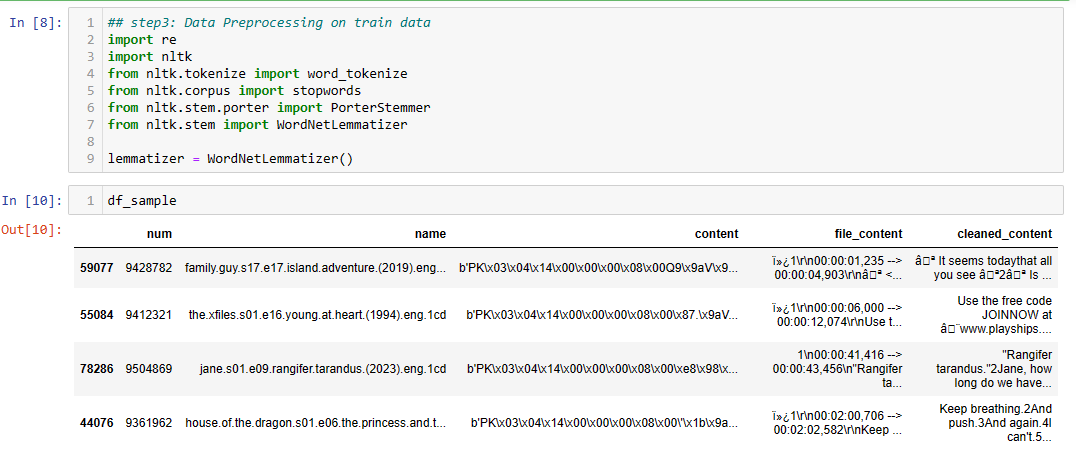
**F.** Here

**i.Importing required Libraries:**

* The code imports the required libraries for text preprocessing, including **regular expression, NLTK, and specific NLTK modules**.
* **re**: Provides support for regular expressions.
* **nltk:** Natural Language Toolkit, a powerful library for symbolic and statistical natural language processing.
* **word\_tokenize:** Tokenizes sentences into words.
* **stopwords:** A set of common words to be ignored, like "the", "is", "at", "which", etc.
* **PorterStemmer:** A stemming algorithm to remove morphological affixes from words, leaving only the word stem.
* **WordNetLemmatizer:** A lemmatization tool to reduce words to their base or root form.

**ii. Initializing WordNet Lemmatizer:**

* **lemmatizer = WordNetLemmatizer()** initializes the **WordNet Lemmatizer** for use in the **text preprocessing.**



1. **Preprocessing Function:**

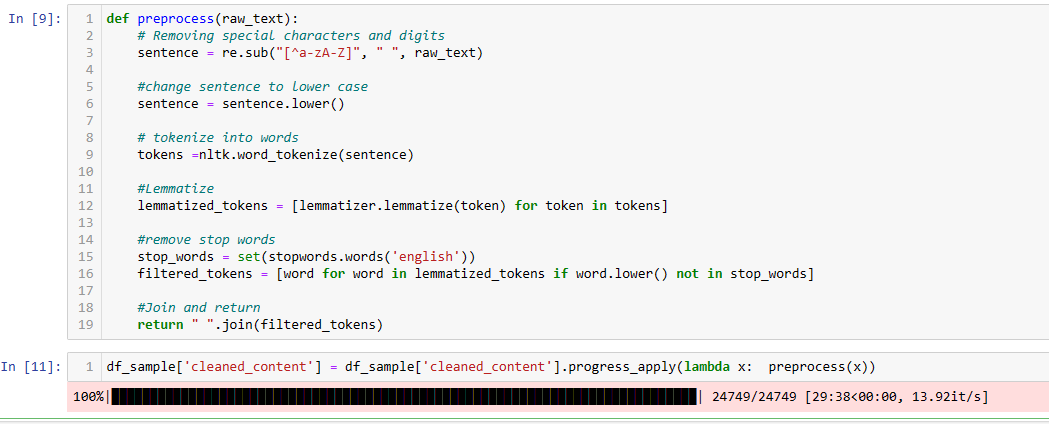
* The **preprocess()** function takes **raw\_text** as input and performs several preprocessing steps on it.

1. **Removing Special Characters and Digits:**

* *re.sub("[^a-zA-Z]", " ", raw\_text):* Removes special characters and digits, replacing them with a space.

1. **Changing Sentence to Lower Case:**

* *sentence = sentence.lower():* Converts the text to lower case.



1. **Tokenization:**

* *tokens = nltk.word\_tokenize(sentence***):** Tokenizes the text into words.

1. **Lemmatization:**

* *lemmatized\_tokens = [lemmatizer.lemmatize(token) for token in tokens]:* Lemmatizes the tokens, reducing them to their base form.

1. **Removing Stop Words:**

* *stop\_words = set(stopwords.words('english')):* Creates a set of English stop words.
* *filtered\_tokens = [word for word in lemmatized\_tokens if word.lower() not in stop\_words]:* Removes stop words from the lemmatized tokens.

1. J**oining and Returning:**

* *return " ".join(filtered\_tokens):* Joins the filtered tokens back into a string and returns the preprocessed text.

**PROCESS OF CHUNKING:**

1. **Document Chunking:**
2. Challenge: Information Loss:
3. Consider the challenge of embedding large documents: there's a risk of information loss.
4. It is often not practical to embed an entire document as a single vector, particularly when dealing with long documents.

b. Solution: Divide Documents into Smaller Chunks:

* To mitigate the challenge of embedding large documents, it's advisable to divide the document into smaller, more manageable chunks for embedding.

c.Another Problem: Context Splitting:

* Let's say we set the token window to be 500, then we'd expect each chunk to be just below 500 tokens.
* One common concern of this method is that we might accidentally cut off some important text between chunks, splitting up the context.

**d. Overlapping Windows**

* To mitigate the problem of splitting up the context, we can set overlapping windows with a specified amount of tokens to overlap so we have tokens shared between chunks. This helps in maintaining the context.

**Code Explanation:**

1. **Chunking Documents:**

* The chunk\_document function takes three parameters**: corpou**s (list of documents), **id\_col** (list of IDs), and **chunk\_size** (default set to 300).

1. **Loop Through Each Document:**

* The function loops through each document and its corresponding ID using **zip(corpous, id\_col).**

1. **Splitting the Document into Words:**

* The document is split into words using **words = doc.split().**

1. **Chunking:**

* For each document, the function creates chunks of the specified size **(chunk\_size).**
* It starts at the beginning of the document and moves in steps of chunk\_size until the end of the document is reached.
* For each chunk, it extracts words within the chunk size, joins them into a string, and appends them to the data list.
* **DataFrame Creation:**
* After all the chunks are created, the data list is converted into a DataFrame, where each row contains an ID and a chunk of the document.
* **Returning the DataFrame:**
* Finally, the function returns the DataFrame containing the document chunks with their corresponding IDs.



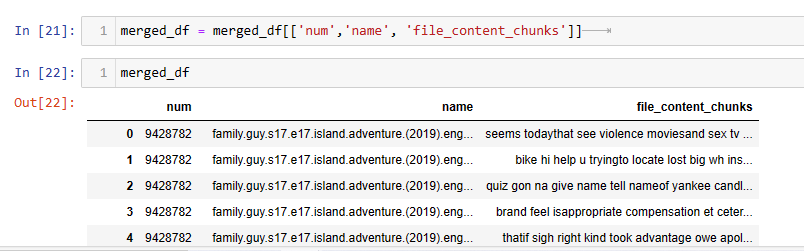
1. **Chunking Documents:**

* The *chunk\_document()* function is called to chunk the documents.
* *df\_sample['cleaned\_content']:* The cleaned content of the DataFrame *df\_sample* is passed as the corpus.
* *df\_sample['num']:* The 'num' column of the DataFrame df\_sample is passed as the ID column.
* The resulting *DataFrame* contains the document chunks with their corresponding IDs.



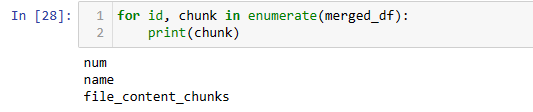
1. **Renaming Columns:**

* The code renames the columns of the DataFrame chuncked\_df.
* The first column is renamed to 'num'.
* The second column is renamed to 'file\_content\_chunks'.



* **Looping through DataFrame:**

The code iterates through each element in the DataFrame merged\_df. id and chunk are the variables that represent the index and the content of each element, respectively. enumerate() is used to loop through each element along with its index.



* **Printing Chunk:** Inside the loop, chunk is printed, representing each element of the DataFrame.

**PART B : INTRODUCTION TO BERT-based SentenceTransformers:**

**Embeddings :**

BERT generates contextual embeddings, it takes as input a sequence (usually a sentence) rather than a single word. BERT needs to be shown the context that surrounding words provide before it can generate a word embedding. With BERT, you do need to have the actual model as the vector representations of words will vary based on the specific sequences you’re inputting. The output is a fixed-length vector representation of the input sentence.

BERT or Bidirectional Encoder Representations from Transformers, is a technique that allows for bidirectional training of Transformers for natural language modeling tasks. Language models which are bidirectionally trained can learn deeper context from language than single-direction models. BERT generates context aware embeddings that allow for multiple representations (each representation, in this case, is a vector) of each word based on a given word’s context.

**Word Ordering:**

BERT model explicitly takes as input the position (index) of each word in the sentence before calculating its embedding.

**Out-of-Vocabulary:**

BERT learns representations at the subword level, so a BERT model will have a smaller vocabulary space than the number of unique words in its training corpus. In turn, BERT is able to generate embeddings for words outside of its vocabulary space giving it a near infinite vocabulary.

**SENTENCE TRANSFORMER:**

The Sentence Transformer is a natural language processing model designed to generate fixed-sized numerical representations (embeddings) of sentences or short paragraphs. These embeddings capture the semantic meaning of the input text, enabling various downstream tasks such as semantic similarity, text classification, and clustering.

**Key Features:**

1. Semantic Understanding:

* Sentence Transformer is trained to understand the semantic meaning of sentences and paragraphs, capturing their context in the embedding.

1. Transfer Learning:

* It employs transfer learning, utilizing large pre-trained transformer-based models (such as BERT, RoBERTa) fine-tuned on large text corpora. This enables it to achieve superior performance on various natural language processing (NLP) tasks.

1. Fixed-Size Embeddings:

* It generates fixed-sized numerical representations (embeddings) of sentences or short paragraphs, making it suitable for input to various machine learning models.

1. Usage Flexibility:

* Sentence Transformer can be used for a variety of NLP tasks, including semantic textual similarity, text classification, clustering, and many more.

**How it Works:**

1. **Pre-Trained Models:**

* Sentence Transformer utilizes pre-trained transformer-based models, such as BERT, RoBERTa, and DistilBERT, which are fine-tuned on large text corpora to capture the semantic meaning of sentences effectively.

1. **Embedding Generation:**

* Given a sentence or a short paragraph, Sentence Transformer generates a fixed-sized numerical representation (embedding) that captures the semantic context of the input text.

1. **Transfer Learning:**

* The model employs transfer learning, where the pre-trained transformer-based model is fine-tuned on a specific downstream task, enabling it to generate high-quality embeddings that can be used for various NLP tasks.

**Common Applications:**

1. **Semantic Textual Similarity:**

* Sentence Transformer can be used to compute the similarity between two sentences or short paragraphs.

1. **Text Classification:**

* It can be used for various text classification tasks, including sentiment analysis, topic classification, and more.

1. **Clustering:**

* Sentence Transformer can be utilized for clustering similar documents or sentences together.

1. **Question Answering:**

* It can be applied to question-answering tasks, where the model generates embeddings for both the question and the context, and then matches them to find the answer.

**MODEL OVERVIEW:**

**Model – ‘all-MiniLM-L6-v2’**

* Description: All-round model tuned for many use-cases. Trained on a large and diverse dataset of over 1 billion training pairs.
* Dimensions: 384
* Suitable Score Functions: dot-product (util.dot\_score), cosine-similarity (util.cos\_sim), euclidean distance
* Size: 80 MB
* Training Data: 1B+ training pairs. For details, see model card.

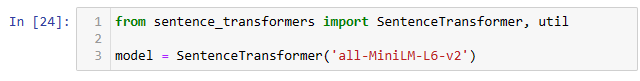
**HOW WE USING THE BERT-SENTENCE TRANSFORMER:**

We're going to use a powerful tool called "BERT-based SentenceTransformers" to generate embeddings. These embeddings are like numerical representations of text that capture the meaning or semantics of sentences. By using SentenceTransformers, we can create embeddings that encode semantic information, allowing us to build a Semantic Search Engine.

In simpler terms, SentenceTransformers helps us understand the meaning of sentences in a way that computers can work with. This is important because it allows us to search for text based on its meaning rather than just specific words or phrases. So, instead of just matching keywords, our search engine will be able to find relevant content based on the overall meaning of the text. This makes the search results much more accurate and useful for users.

**Sentence Transformer with 'all-MiniLM-L6-v2'**

The Sentence Transformer is a natural language processing model designed to generate fixed-sized numerical representations (embeddings) of sentences or short paragraphs. Specifically, the 'all-MiniLM-L6-v2' model is a variant of the MiniLM model, fine-tuned on a large amount of data to capture rich semantic representations effectively.

****

**PART C: INTRODUCTION TO CHROMADB:**

ChromaDB is a Python package that helps with the extraction, management, and analysis of text data, providing an efficient way to store and retrieve documents. It's a convenient tool for Natural Language Processing (NLP) tasks and allows for the integration of various NLP models.

**Key Features:**

1. **Text Data Management:**

* ChromaDB offers efficient management of text data, allowing easy storage and retrieval of documents.

1. **Integration with NLP Models:**

* It seamlessly integrates with NLP models such as Sentence Transformer, enabling efficient processing and analysis of text data.

1. **Data Retrieval:**

* ChromaDB provides an easy-to-use interface for fetching data from databases, making it convenient for NLP tasks.

**How it Works:**

1. **Text Data Storage:**

* ChromaDB facilitates the storage of text data, making it easily accessible for NLP tasks.

1. **Integration with NLP Models:**

* It seamlessly integrates with popular NLP models like Sentence Transformer, allowing for the efficient processing of text data.

1. **Data Retrieval:**

* ChromaDB provides an interface to retrieve data from databases, which can be used directly with NLP models for various tasks.

**Common Applications:**

1. **Document Storage and Retrieval:**

* ChromaDB can be used to store and retrieve large volumes of text data efficiently.

1. **Semantic Textual Similarity:**

* It can be used to compute the similarity between two sentences or short paragraphs.

1. **Text Classification:**

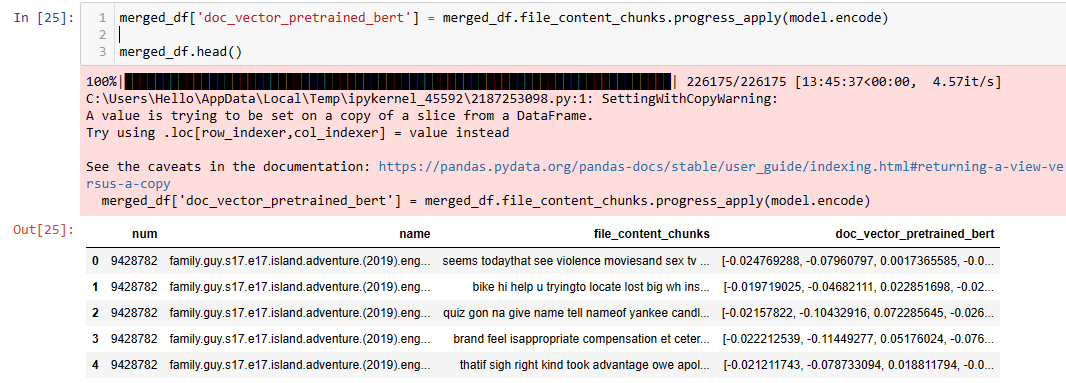
* ChromaDB can be utilized for various text classification tasks, including sentiment analysis, topic classification, and more.

1. **Clustering:**

* It can be used to cluster similar documents or sentences together

**Applying Sentence Transformer to Generate Document Vectors:**

* **merged\_df['file\_content\_chunks']:** Accesses the 'file\_content\_chunks' column in the DataFrame, which contains the text data.
* .**progress\_apply():** Applies a function to each element of the 'file\_content\_chunks' column, displaying a progress bar.
* **model.encode:** Encodes the text using the Sentence Transformer model.



* **merged\_df['doc\_vector\_pretrained\_bert']:** Creates a new column named 'doc\_vector\_pretrained\_bert' in the DataFrame merged\_df to store the document vectors generated by the Sentence Transformer.

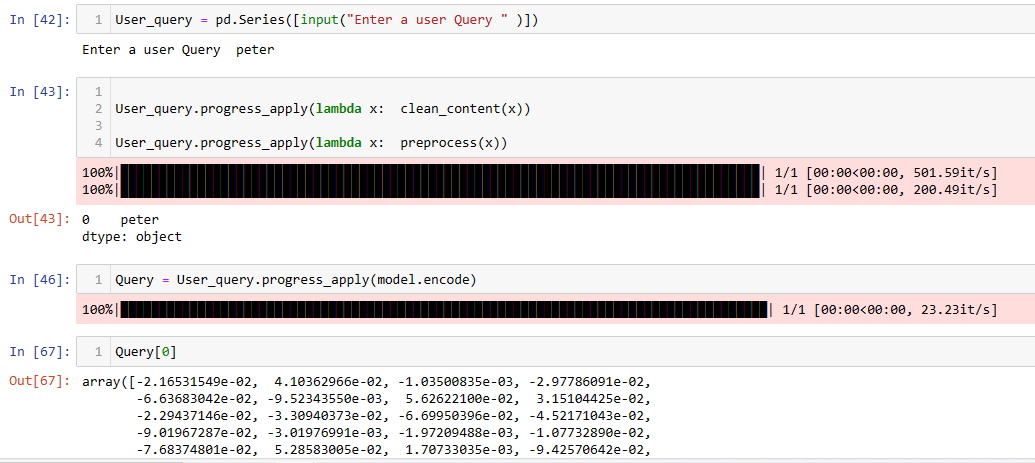
This code essentially uses Sentence Transformer to encode the text in the **'file\_content\_chunks'** column and stores the resulting document vectors in a new column named **'doc\_vector\_pretrained\_bert**' in the DataFrame **merged\_df.** Finally, it displays the first few rows of the DataFrame.

1. **USER QUERY, CLEANS, PREPROCESSES IT:**
2. **Inputting User Query:**

* This line takes a user input query and stores it in a pandas Series named User\_query.

1. **Cleaning User Query:**

* clean\_content() is a function to clean the content.
* User\_query.progress\_apply(lambda x: clean\_content(x)) applies the clean\_content() function to the user query to clean it.
* However, the .progress\_apply() is unnecessary here as we are working with only one input. It's typically used to display a progress bar while applying a function to each element of a DataFrame or Series.



1. **Preprocessing User Query:**

* preprocess() is a function for text preprocessing.
* User\_query.progress\_apply(lambda x: preprocess(x)) applies the preprocess() function to the user query for preprocessing.
* As with the previous line, .progress\_apply() is not necessary here due to a single input.

1. **Encoding User Query:**

* model.encode is used to encode the user query using the Sentence Transformer model.
* User\_query.progress\_apply(model.encode) applies the Sentence Transformer model to encode the user query.
* Again, the use of .progress\_apply() here is unnecessary since there is only one input.

Overall, the code takes a user query, cleans, preprocesses it, and then encodes it using the Sentence Transformer model.

**COSINE SIMILARITY:**

1. **Computing Cosine Similarity:**

* CODE:cos\_sim=util.cos\_sim(Query,merged\_df['doc\_vector\_pretrained\_bert'])
* util.cos\_sim(Query, merged\_df['doc\_vector\_pretrained\_bert']): Computes the cosine similarity between the user query vector (Query) and the document vectors stored in the DataFrame merged\_df['doc\_vector\_pretrained\_bert'].

1. **Sorting the Results:**

* Copy code: a = cos\_sim.argsort()[:][::].tolist()
* cos\_sim.argsort(): Returns the indices that would sort the cosine similarity array.
* [:][::]: Selects all elements from the sorted indices.
* .tolist(): Converts the sorted indices into a list.



1. **Flattening the List:**

Copy code : single\_list = [] for sublist in a: single\_list.extend(sublist)

* This loop is used to flatten the list of lists obtained in the previous step into a single list.

1. **Selecting Top Similar Documents:**

* Copy code: top\_sub = merged\_df.iloc[single\_list[::-1]]
* merged\_df.iloc[single\_list[::-1]]: Retrieves the top similar documents from the DataFrame merged\_df based on the cosine similarity. The [::-1] reverses the order to get the most similar documents first.

1. **Displaying the Top K Documents:**

* Copy code: k = int(input("No\_of docs you want"))

top\_sub[:k]

* int(input("No\_of docs you want")): Takes input from the user to specify the number of documents they want to retrieve.
* top\_sub[:k]: Displays the top K documents from the sorted list.

This code computes the cosine similarity between the user query and the document vectors, sorts the results, selects the top similar documents, and displays the top K documents based on user input.

**MERGING OF DATA:**

* Merged\_df.to\_csv("Sub\_Titles.csv", index=False): Saves the DataFrame merged\_df to a CSV file named "Sub\_Titles.csv" without including the index column.



**CONCLUSION:**

In this conversation, we loaded the data using ChromaDB and encoded the text data using the Sentence Transformer model 'all-MiniLM-L6-v2'. Then, we demonstrated how to input a user query, preprocess it, and compute cosine similarity with the pre-encoded document vectors to find the most relevant documents. Finally, we exported the top similar documents to a CSV file named "Sub\_Titles.csv".