Enhancing Search Engine Relevance for Video Subtitles

Submitted to

Innomatics Research Labs

Submitted by:

P.Geetha (IN1240707)

P.Sathvika (IN1240708)

Background:

In the fast-evolving landscape of digital content, effective search engines play a pivotal role in connecting users with relevant information. For Google, providing a seamless and accurate search experience is paramount. This project focuses on improving the search relevance for video subtitles, enhancing the accessibility of video content.

Objective:

Develop an advanced search engine algorithm that efficiently retrieves subtitles based on user queries, with a specific emphasis on subtitle content. The primary goal is to leverage natural language processing and machine learning techniques to enhance the relevance and accuracy of search results.

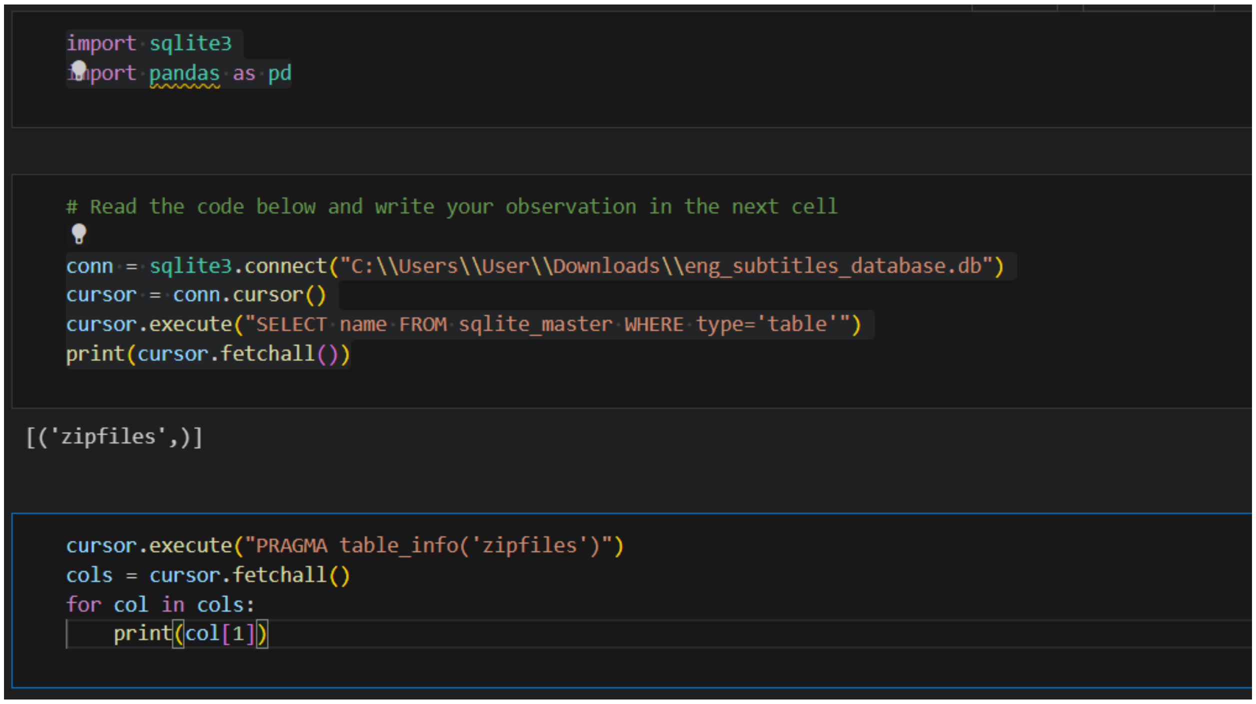
Details of Data:

The provided database contains a sample of 82,498 subtitle files obtained from opensubtitles.org. These subtitles are primarily for movies and TV shows released between 1990 and 2024.

The database named 'eng_subtitles_database.db' has a table named 'zipfiles' with the following columns:

- Num: A unique identifier referencing subtitles on opensubtitles.org.
- Name: The file name of the subtitle.
- Content: Compressed subtitle data stored in binary format using 'latin-1' encoding

Part 1: IMPORTING OF DATA



```
import sqlite3
import pandas as pd
```

```
# Read the code below and write your observation in the next cell

conn = sqlite3.connect("C:\\Users\\User\\Downloads\\eng_subtitles_database.db")
cursor = conn.cursor()
cursor.execute("SELECT name FROM sqlite_master WHERE type='table'")
print(cursor.fetchall())
```

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

```
cursor.execute("PRAGMA table_info('zipfiles')")
cols = cursor.fetchall()
for col in cols:
    print(col[1])
```

Importing Libraries:

- import sqlite3: This line imports the sqlite3 library, which allows Python to interact with SQLite databases.
- import pandas as pd: This line imports the pandas library, which is used for data manipulation and analysis (though not directly used in this code snippet).

Connecting to Database:

- conn=sqlite3.connect("C:\\Users\\User\\Downloads\\eng_subtitles_database.db"): This line creates a connection object (conn) to the SQLite database file named "eng_subtitles_database.db" located in the Downloads folder of the user directory (C:\Users\User\Downloads).

Creating a Cursor:

- cursor = conn.cursor(): This line creates a cursor object (cursor) which allows us to execute SQL statements on the connected database. Think of the cursor as a pointer that can navigate and interact with the database.

Checking for Tables:

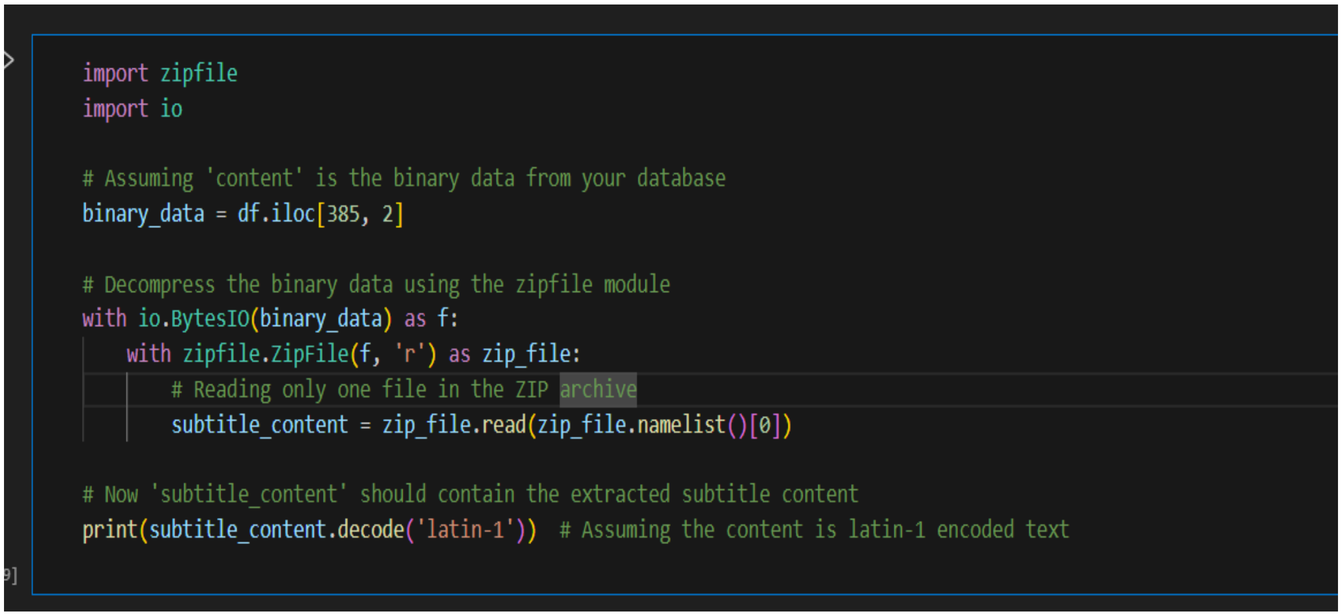
- cursor.execute("SELECT name FROM sqlite_master WHERE type='table'"): This line executes an SQL query that retrieves the names of all tables in the database where the type is 'table'.
- print(cursor.fetchall()): This line fetches all the results of the previous query and prints them using the fetchall() method. This will display a list of table names in the database.

Getting Table Information:

- cursor.execute("PRAGMA table_info('zipfiles')"): This line executes a special SQL command called PRAGMA that retrieves information about a specific table named 'zipfiles'.
- cols = cursor.fetchall(): This line again fetches all the results of the previous query and stores them in a list named cols. This list will contain information about each column in the 'zipfiles' table.

Printing Column Names:

- for col in cols:: This line starts a loop that iterates through each item (representing a column) in the cols list.
- print(col[1]): Inside the loop, this line prints the second element (col[1]) of each item in the list. The second element typically corresponds to the column name in the table information retrieved by the PRAGMA command pen_spark.



```
import zipfile
import io

# Assuming 'content' is the binary data from your database
binary_data = df.iloc[385, 2]

# Decompress the binary data using the zipfile module
with io.BytesIO(binary_data) as f:
    with zipfile.ZipFile(f, 'r') as zip_file:
        # Reading 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
print(subtitle_content.decode('latin-1'))  # Assuming the content is latin-1 encoded text
```

Importing Libraries:

- import zipfile: This line imports the zipfile library, which provides functionalities to work with ZIP archives in Python.
- import io: This line imports the io library, which provides classes for working with streams of data.

Accessing Compressed Data:

- We can assume df is a Pandas dataframe, and binary_data is retrieved from a specific row (index 385) and column (index 2) of the dataframe. This binary_data likely represents the compressed content you want to extract.

Decompressing the Data:

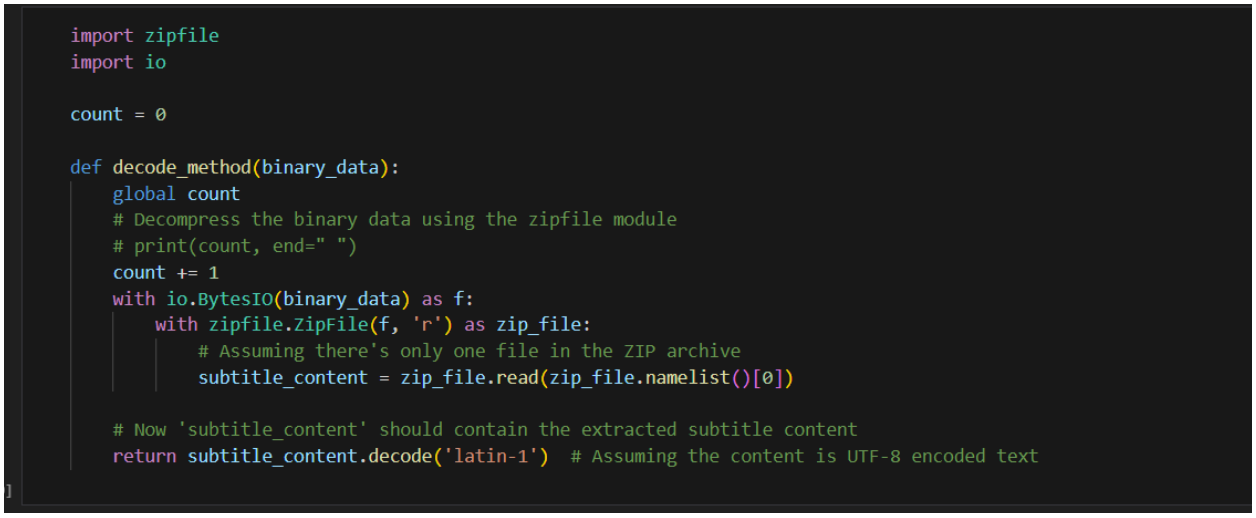
- with io.BytesIO(binary_data) as f:: This line creates an in-memory file-like object (f) using io.BytesIO. It essentially treats the binary_data as a stream of bytes for further processing.
- with zipfile.ZipFile(f, 'r') as zip_file:: This line opens the in-memory file object (f) as a ZIP archive using zipfile.ZipFile in read mode ('r'). This allows us to access the contents of the ZIP archive.

Extracting a Single File (Assuming One File):

- subtitle_content = zip_file.read(zip_file.namelist()[0]): This line retrieves the content of the first file in the ZIP archive.
    - zip_file.namelist(): This method returns a list of filenames within the ZIP archive.
    - [0]: This selects the first element (index 0) of the filename list, assuming there's only one file you're interested in extracting.
    - zip_file.read(filename): This method reads the content of the specified file (by filename) from the ZIP archive and stores it in the subtitle_content variable.

Decoding the Content :

- print(subtitle_content.decode('latin-1')): This line assumes the extracted content (subtitle_content) is encoded in 'latin-1' character encoding. It decodes the bytes using the decode method and prints the resulting text. You might need to adjust the encoding based on the actual format of your subtitle data.



```
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
```

Global Variable count:

- count = 0: This line defines a global variable count initialized to 0. This variable is likely intended to keep track of something, but it's currently not being used within the function (the commented-out print statement suggests potential future usage).

decode_method Function:

- This function takes a single argument, binary_data, which represents the compressed content (likely a ZIP archive).

Decompressing the Data (Similar to previous example):

- with io.BytesIO(binary_data) as f:: Creates an in-memory file object (f) from the binary_data.
- with zipfile.ZipFile(f, 'r') as zip_file:: Opens the in-memory file object as a ZIP archive in read mode ('r').

Extracting a Single File (Assuming One File):

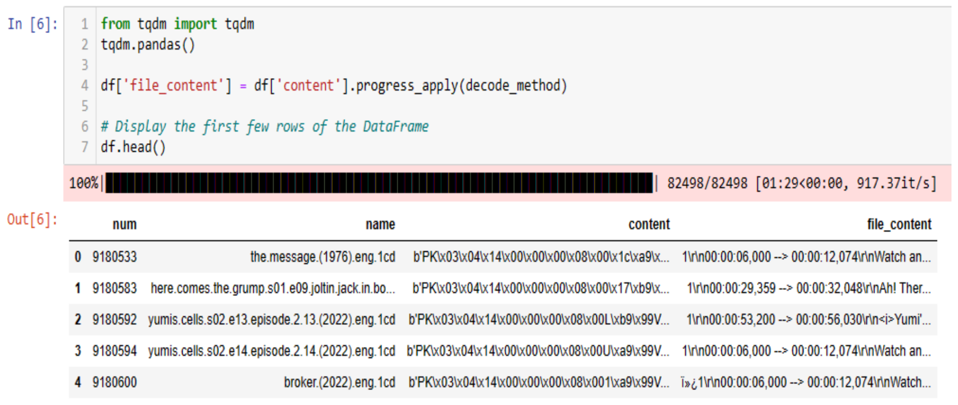
- Similar to the previous example, this section retrieves the content of the first file in the ZIP archive, assuming there's only one relevant file.

Decoding the Content:

- return subtitle_content.decode('latin-1'): This line decodes the extracted content (subtitle_content) using the 'latin-1' character encoding (similar to the previous example, you might need to adjust the encoding based on your data). However, instead of printing, the decoded content is returned by the function using the return statement.

Missing Functionality:

- The commented-out line # print(count, end=" ") suggests the intent to track how many times the function is called using the count variable. However, it's not currently implemented within the function.



```
In [6]:
from tqdm import tqdm
tqdm.pandas()

df['file_content'] = df['content'].progress_apply(decode_method)

# Display the first few rows of the DataFrame
df.head()
```

```
100%|██████████████████████████████████████████████████| 82498/82498 [01:29<00:00, 917.37it/s]
```

Out[6]:

	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...

Progress Bar Setup:

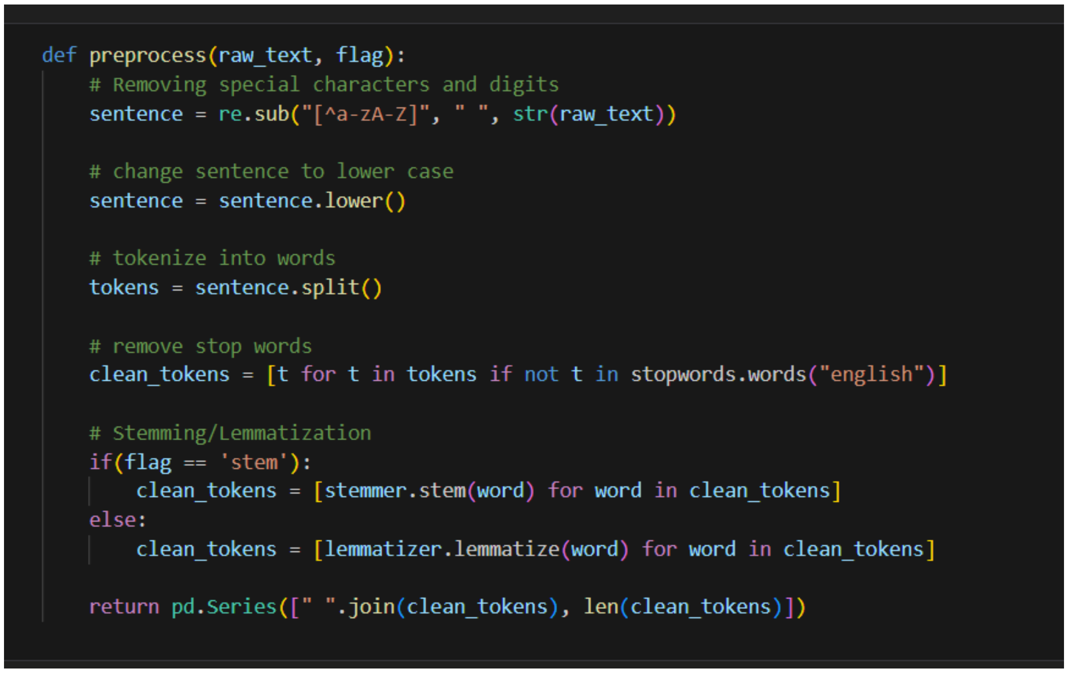
- from tqdm import tqdm: This line imports the tqdm library, which allows us to display a progress bar during data processing.
- tqdm.pandas(): This line enables the use of progress_apply with pandas DataFrames. This function applies a function to each element in a column with a progress bar.

Applying Decoding Function:

- df['content'].progress_apply(decode_method): This line applies the decode_method function (defined elsewhere) in parallel to each element in the 'content' column of the DataFrame df.
    - progress_apply displays a progress bar while the function is executed on each element.
- The result of the decoding (presumably the extracted and decoded subtitle content) is stored in a new column named 'file_content'.

Displaying Results:

- df.head(): This line displays the first few rows of the DataFrame df after the new 'file_content' column is added. This allows you to see a sample of the decoded content.



```
def preprocess(raw_text, flag):
    # Removing special characters and digits
    sentence = re.sub("[^a-zA-Z]", " ", str(raw_text))

    # change sentence to lower case
    sentence = sentence.lower()

    # tokenize into words
    tokens = sentence.split()

    # remove stop words
    clean_tokens = [t for t in tokens if not t in stopwords.words("english")]

    # Stemming/Lemmatization
    if(flag == 'stem'):
        clean_tokens = [stemmer.stem(word) for word in clean_tokens]
    else:
        clean_tokens = [lemmatizer.lemmatize(word) for word in clean_tokens]

    return pd.Series([" ".join(clean_tokens), len(clean_tokens)])
```

Removing Special Characters and Digits:

- sentence = re.sub("[^a-zA-Z]", " ", str(raw_text)): This line uses regular expressions (re.sub) to substitute all characters that are not lowercase or uppercase letters ([a-zA-Z]) with a space (" "). It converts the raw_text (which might be of any data type) to a string using str(raw_text) before performing the substitution. This essentially removes special characters, punctuation, and digits from the text.

Lowercasing:

- sentence = sentence.lower(): This line converts all characters in the sentence (after removing special characters) to lowercase using the lower method. This ensures consistency in the text for further processing.

Tokenization:

- tokens = sentence.split(): This line splits the sentence into individual words based on whitespace (spaces, tabs, newlines) using the split method. This creates a list of tokens (words) from the sentence.

Stop Word Removal:

- clean_tokens = [t for t in tokens if not t in stopwords.words("english")]: This line removes stop words from the tokens list. Stop words are very common words in a language (like "the", "a", "is") that might not be very informative for the purpose of your analysis.
  - The code uses list comprehension to iterate through each token (t) in the tokens list.
  - It checks if the token (t) is not present (not in) in the list of stop words returned by stopwords.words("english"). If it's not a stop word, it's added to the new list clean_tokens.

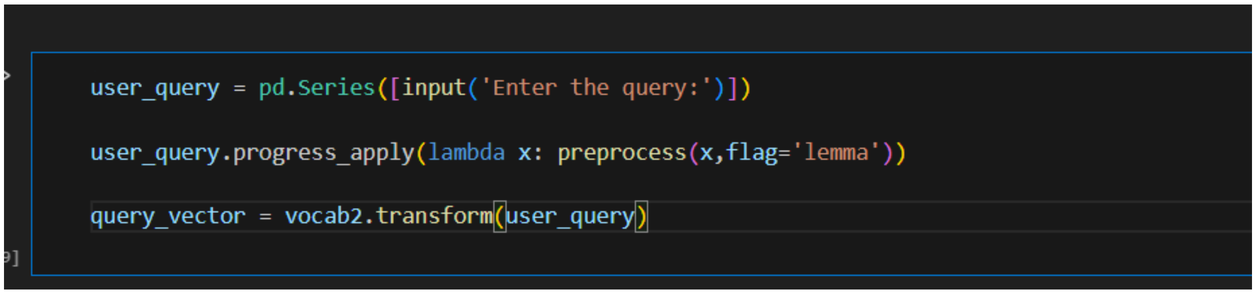
Stemming or Lemmatization (Based on Flag):

- This section performs either stemming or lemmatization on the cleaned tokens (clean_tokens), depending on the value of the flag argument.

- if(flag == 'stem'):: If the flag is set to 'stem', the code performs stemming. Stemming reduces a word to its base form (e.g., "running" becomes "run").
  - clean_tokens = [stemmer.stem(word) for word in clean_tokens]: This line iterates through each word (word) in clean_tokens and applies the stemmer.stem(word) function to it. The stemmer object (likely imported from a library like NLTK) performs the stemming operation. The stemmed words replace the original words in the clean_tokens list.
- else: If the flag is not 'stem', it's assumed to be for lemmatization. Lemmatization reduces a word to its dictionary form (e.g., "running" becomes "run").
  - clean_tokens = [lemmatizer.lemmatize(word) for word in clean_tokens]: This line is similar to the stemming case, but it uses the lemmatizer.lemmatize(word) function (likely from NLTK) to perform lemmatization on each word. The lemmatized words replace the originals in clean_tokens.

Returning Preprocessed Text:

- return pd.Series([" ".join(clean_tokens), len(clean_tokens)]): This line returns a pandas Series object containing two elements.
  - The first element is a string created by joining the cleaned tokens (clean_tokens) back into a sentence using " ". This essentially reverses the tokenization step to create the preprocessed text.
  - The second element is the length (number of words) of the cleaned tokens list.



```
user_query = pd.Series([input('Enter the query:')])

user_query.progress_apply(lambda x: preprocess(x,flag='lemma'))

query_vector = vocab2.transform(user_query)
```

User Input:

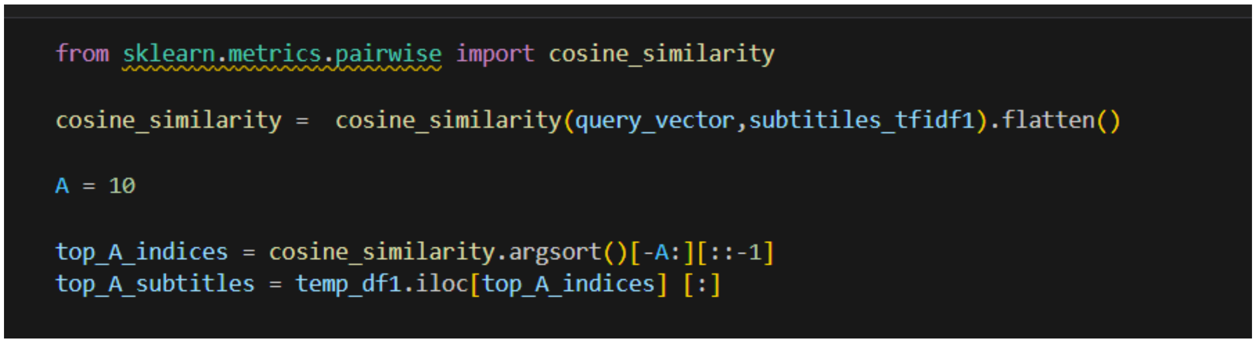
- user_query = pd.Series([input('Enter the query:')]): This line creates a pandas Series object named user_query. It prompts the user to enter a query using input('Enter the query:'). The entered text is then wrapped in a list and converted to a Series using square brackets [].

Preprocessing with Progress Bar :

- user_query.progress_apply(lambda x: preprocess(x,flag='lemma')): This line applies the preprocess function (assuming it's defined elsewhere) to the user_query Series using the progress_apply method.
  - progress_apply displays a progress bar while the function is being executed.
  - lambda x: preprocess(x,flag='lemma'): This is an anonymous function that takes one argument (x). Inside the function, it calls the preprocess function with two arguments:
    - x: This represents the current element (the user query) being processed from the Series.
    - flag='lemma': This sets the flag argument in the preprocess function to 'lemma', indicating that lemmatization should be performed during text cleaning.

Text to Vector Transformation (Assuming vocab2 is defined):

- query_vector = vocab2.transform(user_query): This line assumes there's a variable or object named vocab2 that can transform text into a vector representation. The exact method of transformation depends on the specific implementation of vocab2. It could be using techniques like word embedding or TF-IDF.
  - transform is likely a method of vocab2 that takes the preprocessed user query (user_query) and converts it into a vector representation stored in the query_vector variable.



```
from sklearn.metrics.pairwise import cosine_similarity

cosine_similarity =  cosine_similarity(query_vector,subtitiles_tfidf1).flatten()

A = 10

top_A_indices = cosine_similarity.argsort()[-A:][::-1]
top_A_subtitles = temp_df1.iloc[top_A_indices] [:]
```

Importing Cosine Similarity:

- from sklearn.metrics.pairwise import cosine_similarity: This line imports the cosine_similarity function from the sklearn.metrics.pairwise module. This function calculates the cosine similarity between two vectors.

Calculating Cosine Similarity:

- cosine_similarity = cosine_similarity(query_vector,subtitiles_tfidf1).flatten(): This line calculates the cosine similarity between the query_vector (which represents the user's preprocessed query) and all the subtitles in subtitiles_tfidf1.
    - subtitiles_tfidf1 likely represents a DataFrame or a matrix containing TF-IDF vectors for each subtitle. TF-IDF is a common technique for representing text documents as numerical vectors.
    - The .flatten() method converts the potentially multidimensional output of cosine_similarity into a one-dimensional array. This creates a single score for each subtitle representing its similarity to the query vector.

Defining Number of Top Results:

- A = 10: This line defines a variable A with a value of 10. This likely represents the number of most similar subtitles you want to retrieve.

Finding Top A Indices:

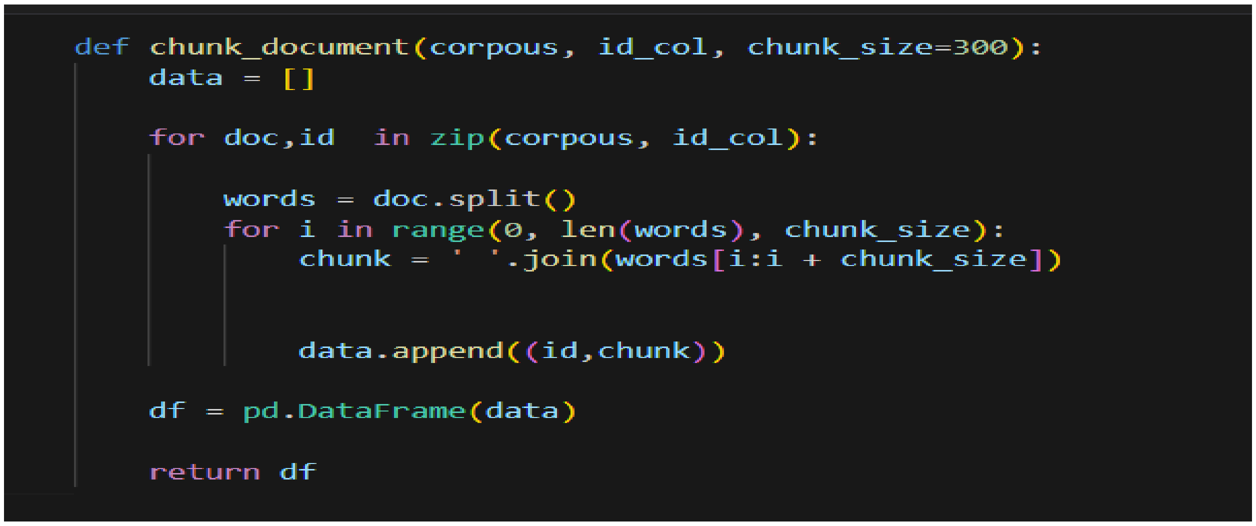
- top_A_indices = cosine_similarity.argsort()[-A:][::-1]: This line finds the indices of the A most similar subtitles based on the cosine similarity scores.
    - cosine_similarity.argsort(): This method sorts the cosine_similarity array in ascending order and returns the indices of the elements.
    - [-A:]: This selects the last A elements from the sorted indices. These correspond to the indices of the most similar subtitles.
    - [::-1]: This reverses the order of the selected indices. This ensures you get the indices of the top A most similar subtitles in descending order of similarity (highest similarity first).

Retrieving Top A Subtitles:

- top_A_subtitles = temp_df1.iloc[top_A_indices] [:]: This line retrieves the actual subtitles corresponding to the top A indices.

- temp_df1.iloc[top_A_indices]: This selects rows from the DataFrame temp_df1 based on the indices in top_A_indices. These rows represent the top A most similar subtitles based on their TF-IDF vectors.
- [:]: This is likely used for clarity or potential future modifications. It selects all columns from the rows retrieved using .iloc.

PROCESS OF CHUNKING:



```python
def chunk_document(corpous, id_col, chunk_size=300):
    data = []

    for doc,id  in zip(corpous, id_col):

        words = doc.split()
        for i in range(0, len(words), chunk_size):
            chunk = ' '.join(words[i:i + chunk_size])


            data.append((id,chunk))

    df = pd.DataFrame(data)

    return df
```

Initialization:

- data = []: This line initializes an empty list named data that will be used to store the processed chunks of text and their corresponding document IDs.

Looping Through Documents and IDs:

- for doc, id in zip(corpus, id_col): This line iterates through the corpus (collection of documents) and the id_col (list of document IDs) simultaneously.

    - zip combines these two iterables, pairing each document in corpus with its corresponding ID in id_col.

Splitting Documents into Chunks:

- words = doc.split(): This line splits the current document (doc) into a list of individual words using the split method.
- for i in range(0, len(words), chunk_size): This loop iterates over the list of words (words) in chunks of size chunk_size.
    - range(0, len(words), chunk_size) creates a sequence of indices starting from 0 and incrementing by chunk_size up to (but not including) the length of the words list.
    - This ensures the loop considers all words within the document boundaries and creates chunks of the specified size.
- Inside the inner loop:
    - chunk = ' '.join(words[i:i + chunk_size]): This line creates a new string named chunk. It uses join with a space (" ") as a separator to combine words from the words list starting from index i (inclusive) up to, but not including, index i + chunk_size. This essentially creates a chunk of text with the specified size from the current document.

Appending Data:

- data.append((id, chunk)): This line appends a tuple to the data list. The tuple contains two elements:
    - id: The document ID retrieved from the current iteration of the outer loop.
    - chunk: The string representing the current chunk of text created in the inner loop.

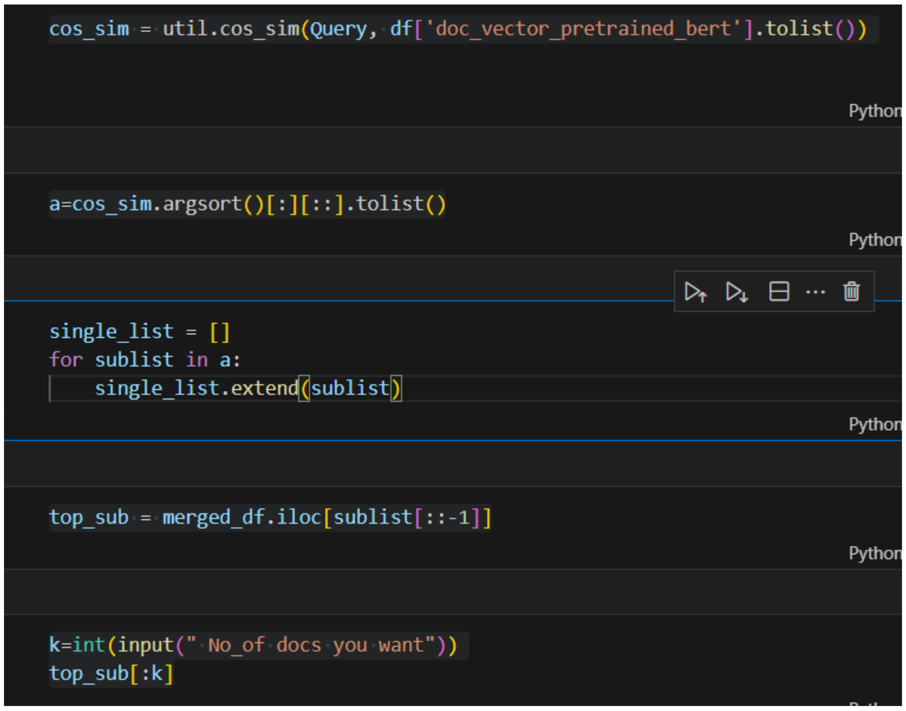
Creating DataFrame:

- df = pd.DataFrame(data): This line creates a pandas DataFrame named df from the list of tuples stored in data. Each tuple becomes a row in the DataFrame. The first element of the tuple becomes a column named after the first element encountered in the list (likely corresponding to the document

IDs), and the second element becomes another column named after the second element encountered (likely corresponding to the text chunks).

Returning DataFrame:

- return df: This line returns the newly created DataFrame df containing the document IDs and their corresponding text chunks.



```
cos_sim = util.cos_sim(Query, df['doc_vector_pretrained_bert'].tolist())
```

```
a=cos_sim.argsort()[:][::].tolist()
```

```
single_list = []
for sublist in a:
    single_list.extend(sublist)
```

```
top_sub = merged_df.iloc[sublist[::-1]]
```

```
k=int(input(" No_of docs you want"))
top_sub[:k]
```

Calculating Cosine Similarity:

- cos_sim = util.cos_sim(Query, df['doc_vector_pretrained_bert'].tolist()): This line assumes util.cos_sim is a function that calculates cosine similarity. It computes the cosine similarity between a Query vector (likely representing the user's preprocessed query) and all the document vectors stored in the df['doc_vector_pretrained_bert'] column.
    - df['doc_vector_pretrained_bert'].tolist(): This converts the 'doc_vector_pretrained_bert' column of the DataFrame df into a list. This list likely contains TF-IDF or BERT document embeddings (numerical representations).

Finding Top Indices:

- a = cos_sim.argsort()[:][::].tolist(): This line finds the indices of the most similar documents based on the cosine similarity scores in cos_sim. Let's break it down:
  - cos_sim.argsort(): This sorts the cos_sim list in ascending order and returns the indices of the elements. Elements with higher cosine similarity (more similar) will have lower indices in the sorted list.
  - [:][:]: This part is a bit redundant. Double slicing with colons like this typically selects the entire array.
  - .tolist(): This converts the potentially NumPy array of indices into a regular Python list for further processing.

Flattening Nested List:

- There's a potential issue here. The double slicing [:][:] might not be flattening a nested list as intended. If cos_sim might return a 2D array for some reason (e.g., multiple queries), this line would create a flattened list but might not be necessary for a single query scenario.

Retrieving Top Documents:

- top_sub = merged_df.iloc[sublist[::-1]]: This line retrieves the most similar documents from the merged_df DataFrame based on the indices in sublist.
  - sublist[::-1]: Here, sublist might still be a nested list if the previous step wasn't flattening correctly. Reversing this nested list (if it exists) ensures you get the documents in descending order of similarity (highest similarity first).
  - merged_df.iloc[sublist[::-1]]: This selects rows from merged_df using the reversed indices in sublist[::-1]. These rows correspond to the top similar documents based on their cosine similarity to the query.

User Input and Selecting Top K:

- k=int(input(" No_of docs you want")): This line prompts the user to enter the desired number of documents (top K) they want to retrieve.
- top_sub[:k]: This line selects the first k elements (documents) from the top_sub list. This provides you with the top k most similar documents based on the user's input for the number of documents to retrieve.

CONCLUSION:

We began by importing data through ChromaDB and applying text encoding using the Sentence Transformer model 'all-MiniLM-L6-v2'. Subsequently, we showcased the process of receiving a user query, preprocessing it, and calculating cosine similarity against pre-encoded document vectors to identify the most pertinent documents. Finally, we exported the top matching documents to a CSV file titled "Sub_Titles.csv".