BUILDING A SEMANTIC BASED SEARCH ENGINE

Enhancing search engine relevance for video subtitles

BY

SARTHAK AGRAWAL
(IN1240235)

&

HANNAH IGBOKE

(IN1240353)

SUBMITTED TO INNOMATICS RESEARCH LABS

HYDERABAD, INDIA

APRIL, 2024

Table of contents

Background

In the fast-evolving landscape of digital content, effective search engines play a pivotal role in connecting users with relevant information. Example, 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.

Problem

There is a difficulty of finding relevant video content through traditional keyword-based search methods when searching for subtitles. Currently, most search engines rely on keywords within video titles, descriptions, or closed captions. However, this approach is not ideal for finding specific content within a video based on dialogue or spoken information.

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 by building a semantic search engine.

About the data

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.

Project steps

Below is an outline of the steps taken to meet the project's objective:

1. Reading the Data from the Database – decompressing and decoding
2. Data cleaning
3. Data chunking
4. Generating text embedding
5. Storing data in a vector database (vector stores)
6. Frontend application to access the Search Engine

Tech stack

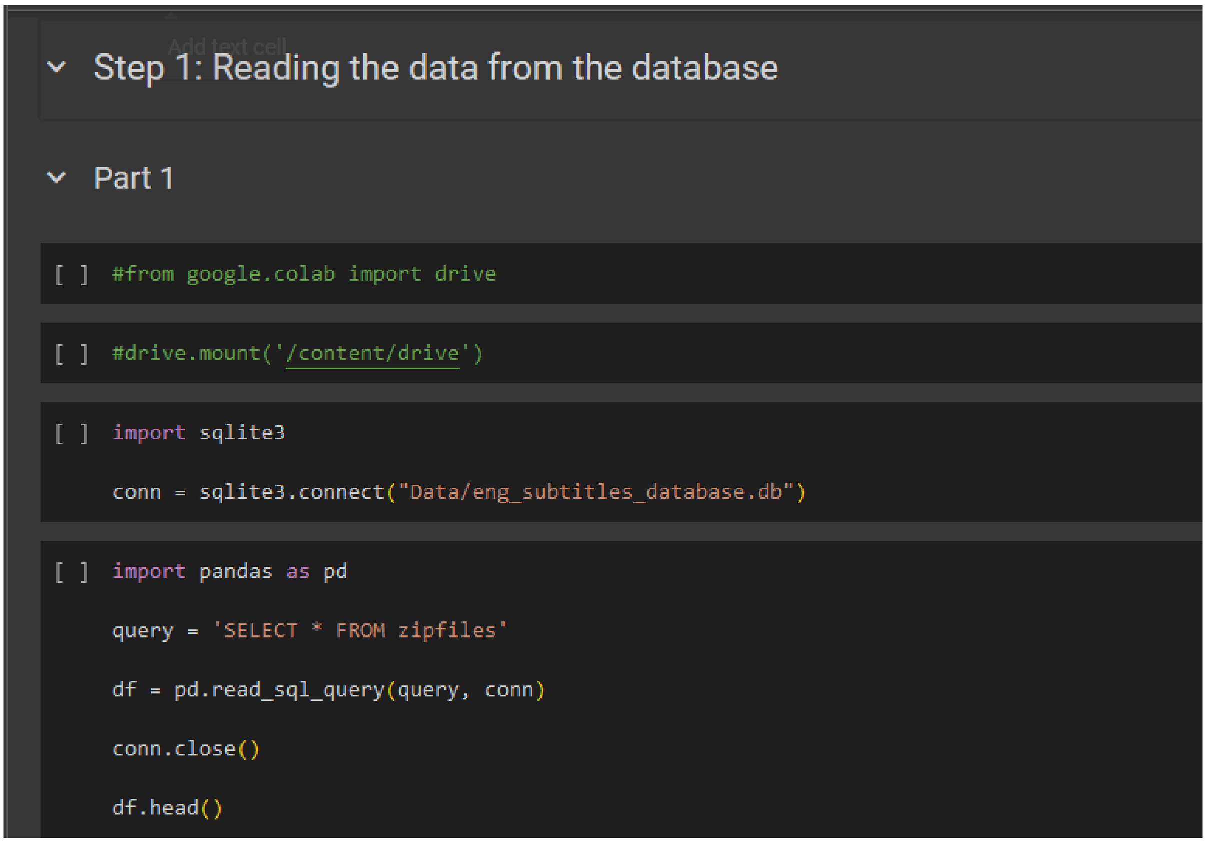
In the course of this project the following tools and languages were utilized:

- Python
- ChromaDB
- Natural Language Processing
- Streamlit
- Google Colab

Step 1: Reading the Data from the Database

Part 1: Connect to database and retrieve the data

The data, as already mentioned above, was provided in a database file (.db) compressed and encoded in **latin-1**. In order to begin, we connected to the SQLite database extracting data from the '**zipfiles**' table. The entire data was retrieved from the table and stored in a pandas dataframe called **df** after which the connection to the database was closed. The code snippet is as seen below.



Step 1: Reading the data from the database

Part 1

```
[ ] #from google.colab import drive
```

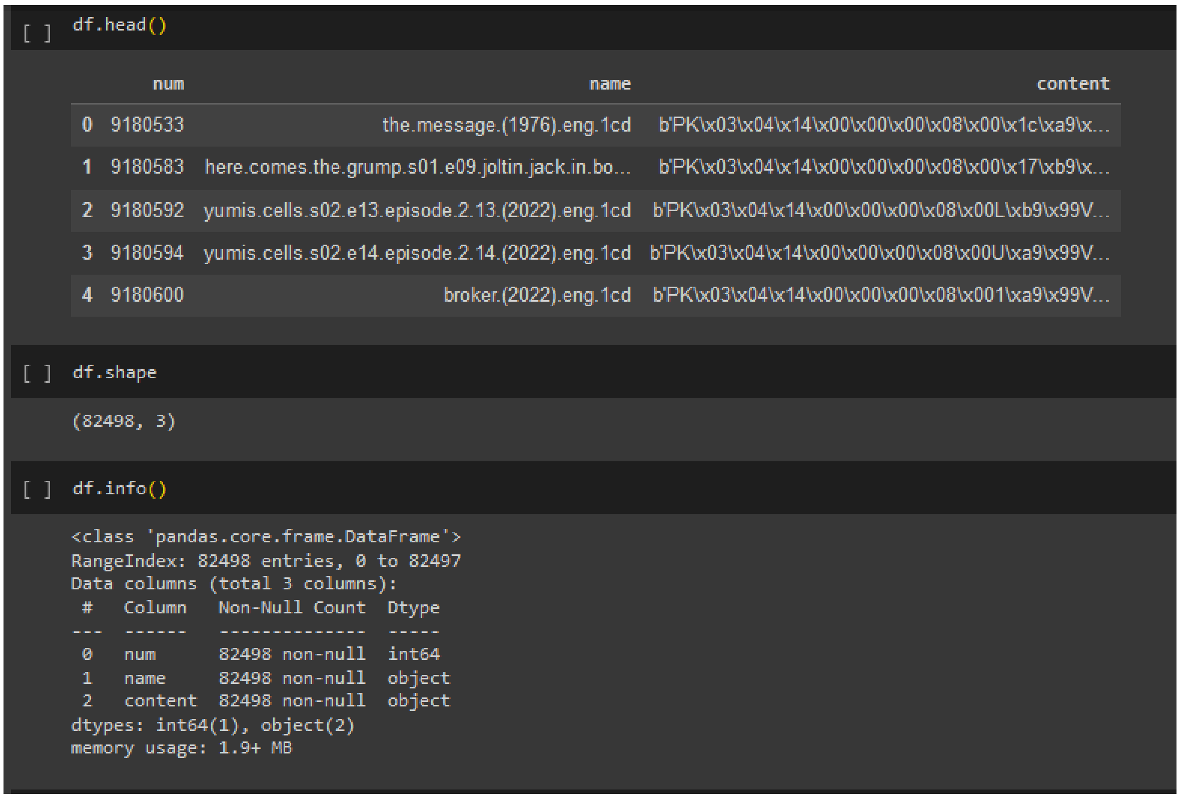
```
[ ] #drive.mount('/content/drive')
```

```
[ ] import sqlite3

    conn = sqlite3.connect("Data/eng_subtitles_database.db")
```

```
[ ] import pandas as pd

    query = 'SELECT * FROM zipfiles'

    df = pd.read_sql_query(query, conn)

    conn.close()

    df.head()
```

Figure 1: Connecting to database

The snapshot below gives us an overview of our data after retrieval. The ‘content’ column is still compressed and encoded.



```
[ ] df.head()
```

	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...

```
[ ] df.shape

    (82498, 3)
```

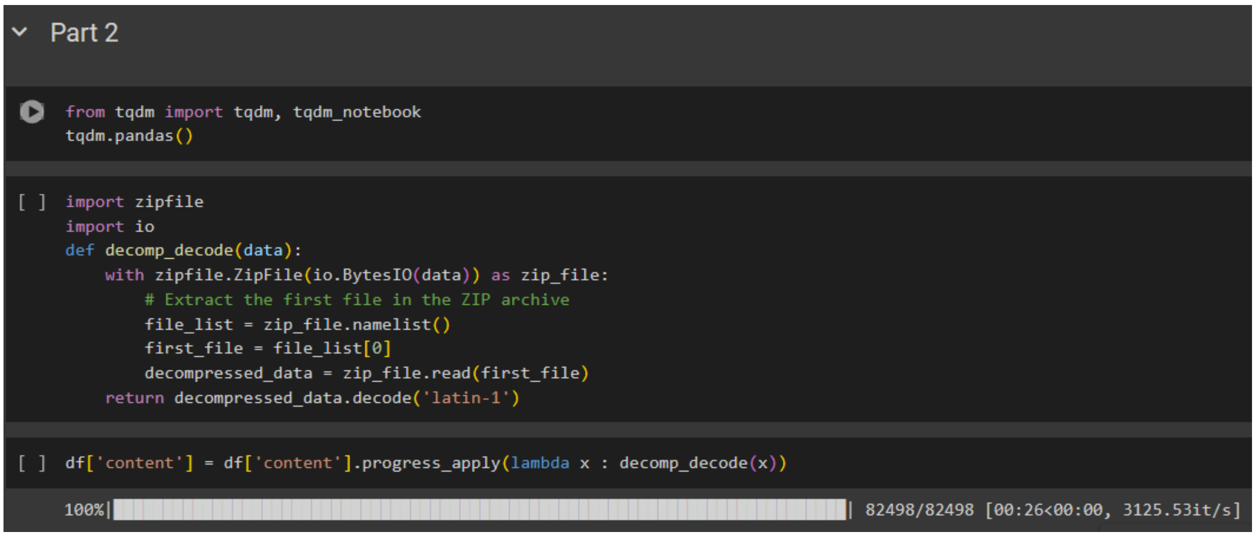
```
[ ] df.info()

    <class 'pandas.core.frame.DataFrame'>
    RangeIndex: 82498 entries, 0 to 82497
    Data columns (total 3 columns):
     #   Column   Non-Null Count  Dtype 
    ---  ------   --------------  ----- 
     0   num      82498 non-null  int64 
     1   name     82498 non-null  object
     2   content  82498 non-null  object
    dtypes: int64(1), object(2)
    memory usage: 1.9+ MB
```

Figure 2: Exploratory data analysis

Part 2: Creating a function to decompress and decode the text data

Here, we imported the **zipfile** and **io** libraries to allow us to work with zip files in python. A function **decomp_decode** is created that takes a compressed data as input and returns the decompressed and decoded text. Here is a breakdown of what the code does:

1. **Decompressing:**
   - First, it uses the zipfile library to handle the compressed data.
   - It creates a virtual in-memory file object using **io.BytesIO(data)**, essentially treating the compressed data as a file.
   - Then, it opens this virtual file as a ZIP archive using *zipfile.ZipFile.*
   - It assumes the archive contains only one file and extracts it using *zip_file.read(first_file).*
2. **Decoding:**
   - Finally, it decodes the extracted (but still encoded) data using *.decode('latin-1').* This translates the bytes from the compressed file into human-readable text, since the encoding used is **'latin-1'**.



Part 2

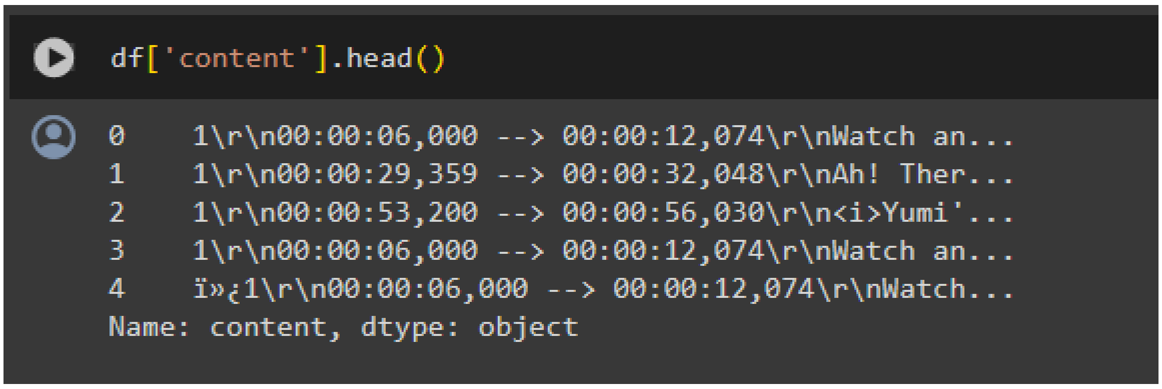
```
from tqdm import tqdm, tqdm_notebook
tqdm.pandas()
```

```
[ ] import zipfile
    import io
    def decomp_decode(data):
        with zipfile.ZipFile(io.BytesIO(data)) as zip_file:
            # Extract the first file in the ZIP archive
            file_list = zip_file.namelist()
            first_file = file_list[0]
            decompressed_data = zip_file.read(first_file)
        return decompressed_data.decode('latin-1')
```

```
[ ] df['content'] = df['content'].progress_apply(lambda x : decomp_decode(x))

    100%|██████████████████████████████████████████████| 82498/82498 [00:26<00:00, 3125.53it/s]
```

Figure 3: Creating the decomp_decode function

Decomp_decode is used to decompress and decode the data in the content column and replace the data in the content column.



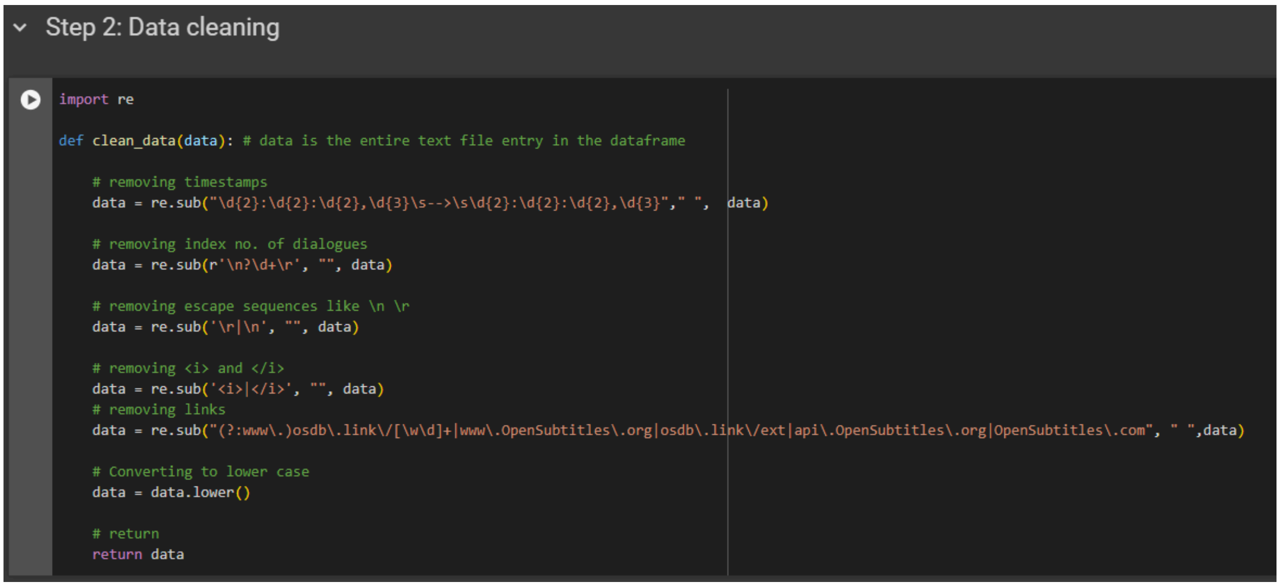
```
df['content'].head()
```

```
0    1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch an...
1    1\r\n00:00:29,359 --> 00:00:32,048\r\nAh! Ther...
2    1\r\n00:00:53,200 --> 00:00:56,030\r\n<i>Yumi'...
3    1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch an...
4    ï»¿1\r\n00:00:06,000 --> 00:00:12,074\r\nWatch...
Name: content, dtype: object
```

Figure 4: The content column

Step 2: Data cleaning

The project is focused on building a semantic search engine and as such, the **BERT** model will be used to generate text embeddings for the subtitle files. There are two types of cleaning that can be performed – light cleaning and heavy cleaning. Light cleaning involves surface level text formatting and removal of irrelevant features in our data while heavy cleaning involves lemmatization, stop word removal, removal of special characters and punctuation. In order to decide which cleaning type to apply, we considered the effects each type will have on the data. For heavy cleaning, we will strip the subtitle data down to basic words devoid of context and sentence demarcations which will in turn affect the efficiency of our semantic search engine in returning relevant results. On the other side, a light cleaning helps us remove just enough features that are not useful in the search results. To this effect we performed **light cleaning** on our data to remove features like timestamps, line breaks, links, italics tags and converted the text to lower case letters.



```python
import re

def clean_data(data): # data is the entire text file entry in the dataframe

    # removing timestamps
    data = re.sub("\d{2}:\d{2}:\d{2},\d{3}\s-->\s\d{2}:\d{2}:\d{2},\d{3}"," ",  data)

    # removing index no. of dialogues
    data = re.sub(r'\n?\d+\r', "", data)

    # removing escape sequences like \n \r
    data = re.sub('\r|\n', "", data)

    # removing <i> and </i>
    data = re.sub('<i>|</i>', "", data)
    # removing links
    data = re.sub("(?:www\.)osdb\.link\/[\w\d]+|www\.OpenSubtitles\.org|osdb\.link\/ext|api\.OpenSubtitles\.org|OpenSubtitles\.com", " ",data)

    # Converting to lower case
    data = data.lower()

    # return
    return data
```

Figure 5: Data cleaning

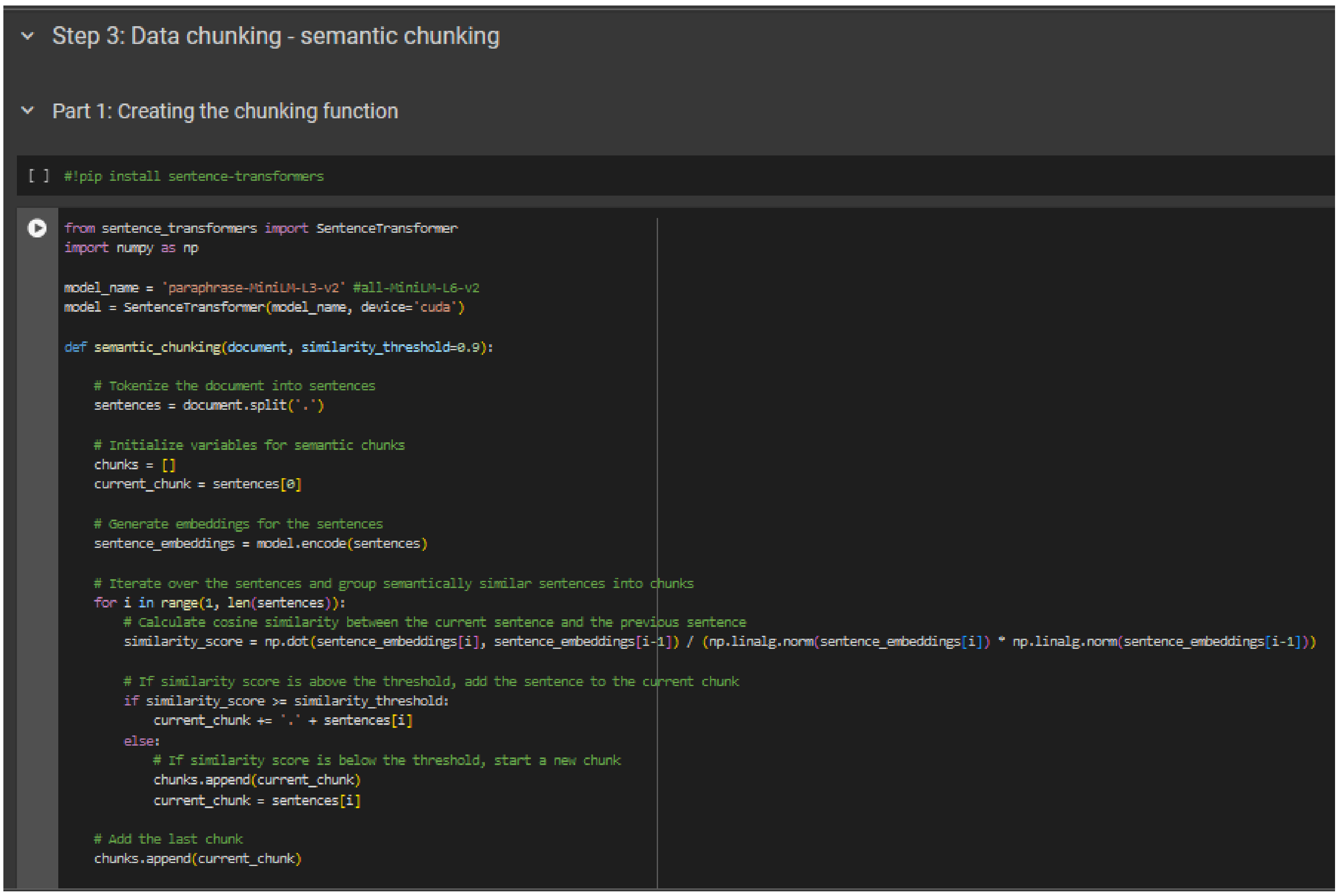
This code defines a function called clean_data that takes a text entry (a subtitle file) and returns a cleaner version of the text. Here's a breakdown of what each step does:

1. **Cleaning Formatting:**
   - It uses regular expressions (re) to perform various cleaning tasks:
     - Removes timestamps using a pattern that matches the format *HH:MM:SS,SSS* --> *HH:MM:SS,SSS* and replaces them with spaces.
     - Removes dialogue line numbers by searching for a number followed by newline characters (*\n or \r*).
     - Removes unnecessary newline characters *(\n or \r).*
     - Deletes italic formatting tags *(<i> and </i>).*
   - Additionally, it replaces website links related to OpenSubtitles.org with spaces, potentially removing irrelevant information.
2. **Normalization:**
   - Finally, it converts the entire text to lowercase using *data.lower()*. This standardizes the text and makes it easier for further processing.

Step 3: Data chunking

Part 1: Creating the chunking function

We observed that the subtitle files are long and embedding such files in one single vector will lead to information loss. It was therefore not practical to embed an entire document as a single vector. Before generating embeddings, we carried out a unique **semantic chunking** process.



Step 3: Data chunking - semantic chunking

Part 1: Creating the chunking function

```
[ ] #!pip install sentence-transformers
```

```
from sentence_transformers import SentenceTransformer
import numpy as np

model_name = 'paraphrase-MiniLM-L3-v2' #all-MiniLM-L6-v2
model = SentenceTransformer(model_name, device='cuda')

def semantic_chunking(document, similarity_threshold=0.9):

    # Tokenize the document into sentences
    sentences = document.split('.')

    # Initialize variables for semantic chunks
    chunks = []
    current_chunk = sentences[0]

    # Generate embeddings for the sentences
    sentence_embeddings = model.encode(sentences)

    # Iterate over the sentences and group semantically similar sentences into chunks
    for i in range(1, len(sentences)):
        # Calculate cosine similarity between the current sentence and the previous sentence
        similarity_score = np.dot(sentence_embeddings[i], sentence_embeddings[i-1]) / (np.linalg.norm(sentence_embeddings[i]) * np.linalg.norm(sentence_embeddings[i-1]))

        # If similarity score is above the threshold, add the sentence to the current chunk
        if similarity_score >= similarity_threshold:
            current_chunk += '.' + sentences[i]
        else:
            # If similarity score is below the threshold, start a new chunk
            chunks.append(current_chunk)
            current_chunk = sentences[i]

    # Add the last chunk
    chunks.append(current_chunk)
```

Figure 6: Semantic chunking

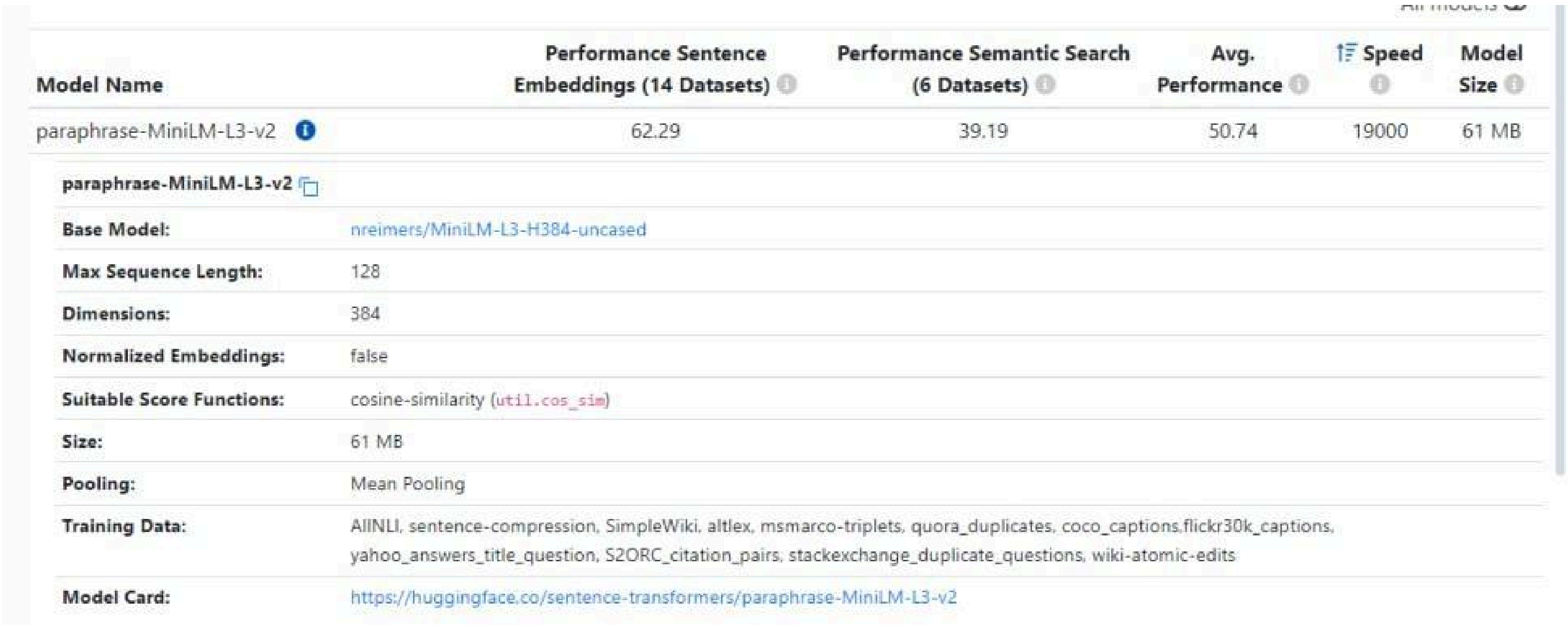
The code snippet above defines a function called **'semantic_chunking'** that takes a document (text) as input and outputs a list of semantically similar sentence groups, called chunks. Below is a breakdown of our code:

Importing Libraries:

- We imported the SentenceTransformer class from the sentence_transformers library in order to create sentence embeddings (numerical representations of sentences) that capture their meaning.
- And the numpy library, to carry out tasks like calculating cosine similarity.

Model Selection:

- We used the **'paraphrase-MiniLM-L3-v2'** model for faster computation.



Model Name	Performance Sentence Embeddings (14 Datasets)	Performance Semantic Search (6 Datasets)	Avg. Performance	Speed	Model Size
paraphrase-MiniLM-L3-v2	62.29	39.19	50.74	19000	61 MB

paraphrase-MiniLM-L3-v2	
Base Model:	nreimers/MiniLM-L3-H384-uncased
Max Sequence Length:	128
Dimensions:	384
Normalized Embeddings:	false
Suitable Score Functions:	cosine-similarity (`util.cos_sim`)
Size:	61 MB
Pooling:	Mean Pooling
Training Data:	AllNLI, sentence-compression, SimpleWiki, altlex, msmarco-triplets, quora_duplicates, coco_captions,flickr30k_captions, yahoo_answers_title_question, S2ORC_citation_pairs, stackexchange_duplicate_questions, wiki-atomic-edits
Model Card:	https://huggingface.co/sentence-transformers/paraphrase-MiniLM-L3-v2

Figure 7: A summary of relevant features of the **'paraphrase-MiniLM-L3-v2'** model

Loading the Model:

- Next, we create an instance of the SentenceTransformer class using the specified model name (‘**paraphrase-MiniLM-L3-v2**’). The **device='cuda'** argument utilizes the graphics card (GPU) for faster computations during sentence encoding.

Function Definition:

- We created a function called ‘**semantic_chunking**’ that takes two arguments:
  - **document**: the text we want to split into semantically similar chunks.
  - **similarity_threshold**: This is a value between 0 and 1 that determines how similar two sentences need to be to be grouped in the same chunk. We chose a threshold of **0.9** for stricter grouping and better results.

Chunking Process:

- *sentences = document.split('.')*: This line splits the document into sentences using the period (.) as the delimiter.
- *chunks = []:* This line initializes an empty list called chunks to store the final semantic chunks.
- *current_chunk = sentences[0]:* This line sets the current_chunk variable to the first sentence in the document.
- *sentence_embeddings = model.encode(sentences):* This line encodes all the sentences in the document using the loaded sentence transformer model. This creates a numpy array where each row represents the numerical embedding of a sentence.

Iterating Through Sentences:

- The for loop iterates over all sentences in the document starting from the second sentence (index = 1)
- Inside the loop:
    - The similarity score line of code calculates the cosine similarity between the current sentence embedding (i) and the previous sentence embedding (i-1). Cosine similarity utilizes cosine distance, that serves as a distance metric to measure how similar two vectors are.
    - The conditional statement checks if the cosine similarity between the two sentences is greater than or equal to the similarity_threshold.
        - If True: The current sentence is appended to the current_chunk variable, separated by a period (.)
        - Else: if the cosine similarity is below the threshold, it signifies a semantic shift in the text.
        - The current chunk is appended to the chunks list as a separate chunk.
        - The current_chunk variable is then reset to the current sentence.
- After the loop finishes iterating through all sentences:
    - The final current_chunk is appended to the chunks list to capture the last group of sentences.

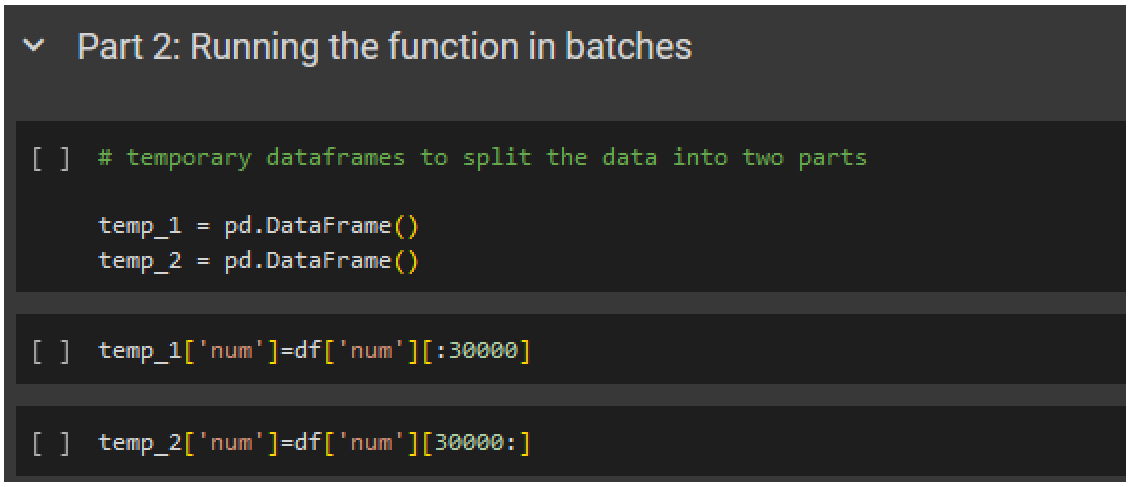
Returning the Results:

- The function returns the chunks list, containing a grouping of the original document's sentences into semantically similar chunks.

Part 2: Running the function in batches

The function has been created and is ready to be used. However some challenges were encountered while trying to perform the chunking process on the entire dataframe at a go. We had limited compute resources on Google Colab and a large file to work with. To resolve this challenge we created two temporary dataframes : temp_1 and temp_2 to house two parts of the data. By splitting the data we reduce the workload on the GPU and RAM, helping it interact with a smaller subset of the data at a time. It is also important to note that this splitted data came in handy in other steps of this project.

In the code snippet below, we populate the first and second dataframe. The first part contained the first 30,000 rows of data (for the *num* column) while the second part contained the remaining 52,498 rows.



Part 2: Running the function in batches

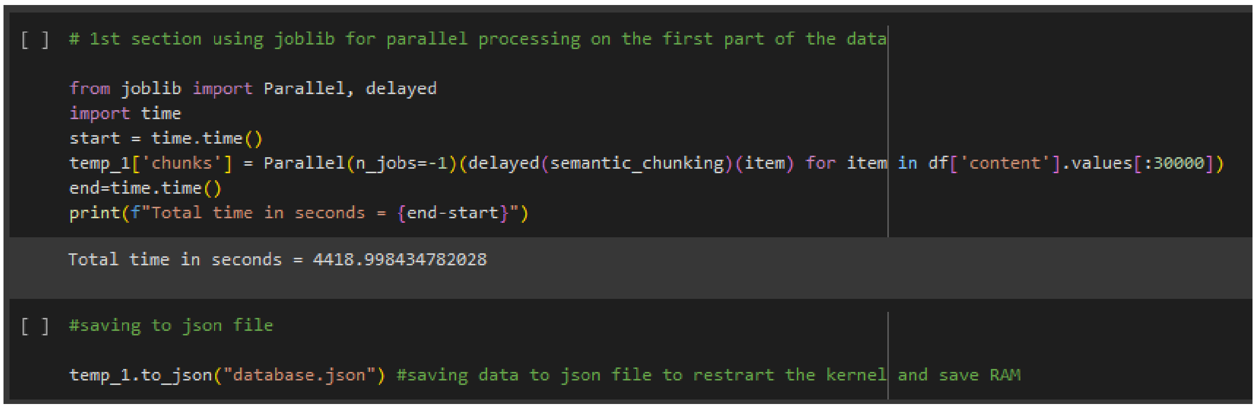
```
[ ] # temporary dataframes to split the data into two parts

    temp_1 = pd.DataFrame()
    temp_2 = pd.DataFrame()
```

```
[ ] temp_1['num']=df['num'][:30000]
```

```
[ ] temp_2['num']=df['num'][30000:]
```

Figure 8: Creating the temporary dataframes

The next challenge we encountered was the amount of time taken for the chunking process. We tackled this by utilizing **'joblib'**, a python library that allows us to leverage *parallel processing* on our python code to achieve significant reduction in execution time of our process. For more context, when we applied semantic_chunking directly to our entire data, the estimated execution time was about **<u>160 hours</u>**! Alternatively, with parallel processing, the execution time was cut down to **<u>4.5 hours</u>** to complete.



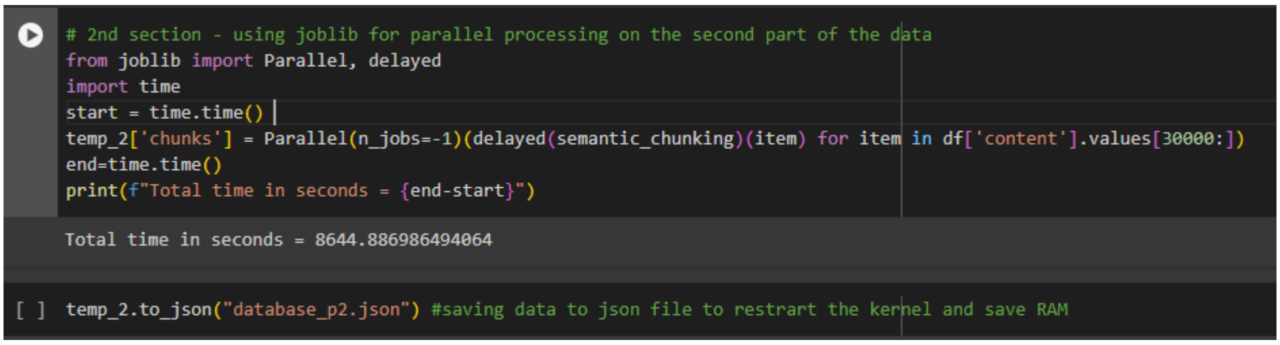
```
[ ] # 1st section using joblib for parallel processing on the first part of the data

    from joblib import Parallel, delayed
    import time
    start = time.time()
    temp_1['chunks'] = Parallel(n_jobs=-1)(delayed(semantic_chunking)(item) for item in df['content'].values[:30000])
    end=time.time()
    print(f"Total time in seconds = {end-start}")

    Total time in seconds = 4418.998434782028
```

```
[ ] #saving to json file

    temp_1.to_json("database.json") #saving data to json file to restrart the kernel and save RAM
```

Figure 9: Running the chunking function on the first part of the dataframe

Again, to maximize the use of compute resources and reduce time taken, the parallel processing is carried out for each part of the dataframe. In the code snippet above, using parallel processing we performed the chunking process on the content column of our original dataframe, remembering to subset the first 30,000 rows of data and then created a new column called chunks which is added to the temp_1 dataframe. The end result is converted to a .json file. Why .json file? By saving to a .json file we preserve the data as a list (this is because .json file format has the ability to retain all the properties of a file). An alternative would have been in a csv format but this instead formats and stores our data as a string.

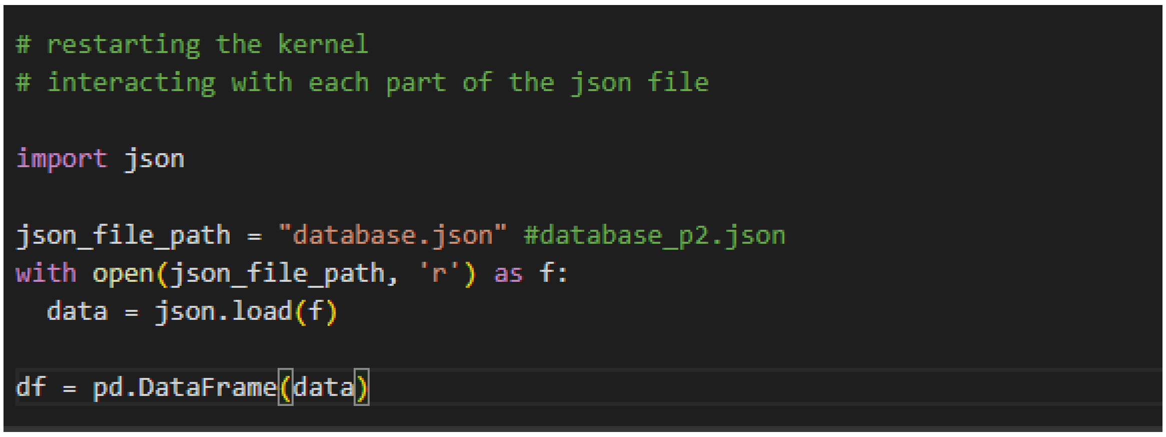
This same process is repeated for the second part of the data as can be seen from the snapshot below. It is important to **note** that in order to reduce the workload on the RAM, we restarted the kernel in order to proceed with this second step.



```
# 2nd section - using joblib for parallel processing on the second part of the data
from joblib import Parallel, delayed
import time
start = time.time()
temp_2['chunks'] = Parallel(n_jobs=-1)(delayed(semantic_chunking)(item) for item in df['content'].values[30000:])
end=time.time()
print(f"Total time in seconds = {end-start}")

Total time in seconds = 8644.886986494064

[ ] temp_2.to_json("database_p2.json") #saving data to json file to restrart the kernel and save RAM
```

Figure 10: Running the chunking function on the second part of the dataframe

Before we can proceed with generating the embeddings for the text files, we load each of the json files - *database.json* and *database_p2.json* one at a time into a dataframe *df*. As indicated in the code snippet below, the kernel is restarted to minimize the load on the RAM.



```
# restarting the kernel
# interacting with each part of the json file

import json

json_file_path = "database.json" #database_p2.json
with open(json_file_path, 'r') as f:
  data = json.load(f)

df = pd.DataFrame(data)
```

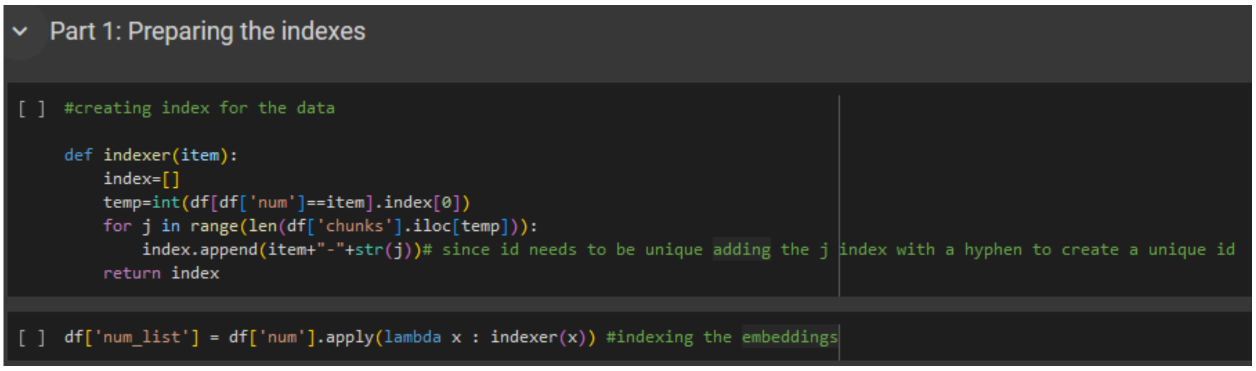
Figure 11: Loading the .json files into df

Step 4: Generating text embeddings

In this step, we prepare our data to be stored in a vector store. This preparation involves two parts.

Part 1: Preparing indexes

In order to store our embeddings we need to have a unique identifier for each row. In this case, since we can have as much as 1000 chunks for one subtitle file, we find that we cannot use one identifier for these 1000 rows of data in our vector database or store but still we want that upon search by a user query for any part of a subtitle file, that it correctly retrieves the data from the database. To solve this we created a function called **indexer** that takes an item and finds the corresponding row in the dataframe based on the 'num' column. Then it iterates through the chunks column value in that row (which is a list) and creates unique identifiers by combining the item with a sequential index (j) separated by a hyphen(-). This serves the purpose of generating unique identifiers for elements within a chunk represented by the item.


Part 1: Preparing the indexes

```
[ ]  #creating index for the data

     def indexer(item):
         index=[]
         temp=int(df[df['num']==item].index[0])
         for j in range(len(df['chunks'].iloc[temp])):
             index.append(item+"-"+str(j))# since id needs to be unique adding the j index with a hyphen to create a unique id
         return index
```

```
[ ]  df['num_list'] = df['num'].apply(lambda x : indexer(x)) #indexing the embeddings
```

Figure 12: Preparing the indexes

An example of what the final result of the indexing is something like the image below:

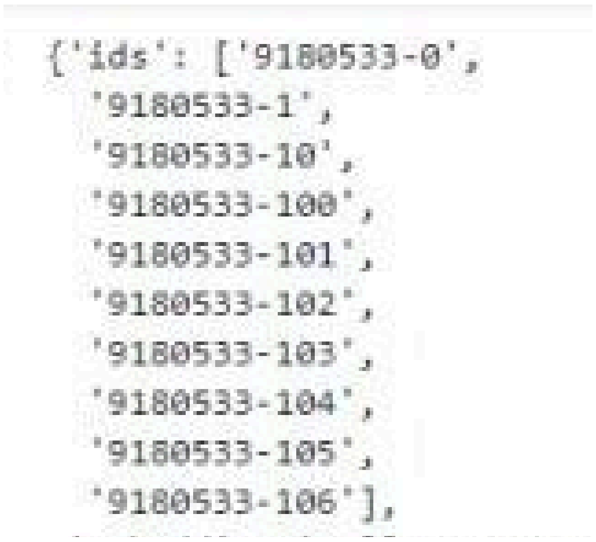
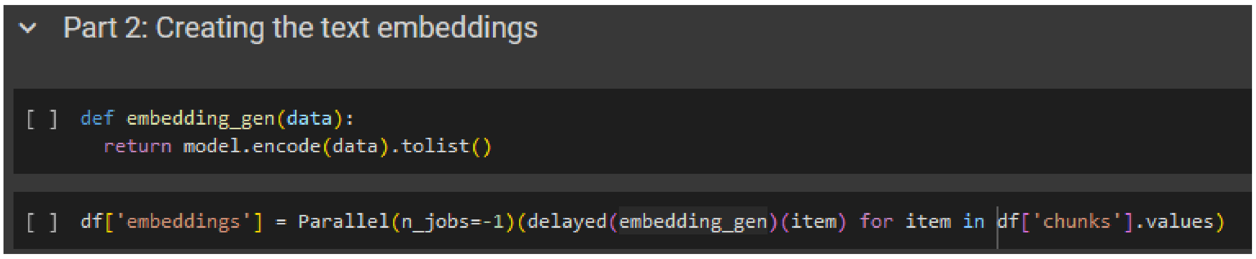

```
{'ids': ['9180533-0',
  '9180533-1',
  '9180533-10',
  '9180533-100',
  '9180533-101',
  '9180533-102',
  '9180533-103',
  '9180533-104',
  '9180533-105',
  '9180533-106'],
```

Figure 13: A sample of the indexes

Part 2: Creating text embeddings

In the second part, we create a function, ‘**embedding_gen**’ that can take an input argument (data), encode it and convert the results into a list. Next, we carry out the embedding process on the chunks column of the dataframe (df) using parallel processing.



Part 2: Creating the text embeddings

```
[ ]  def embedding_gen(data):
       return model.encode(data).tolist()
```

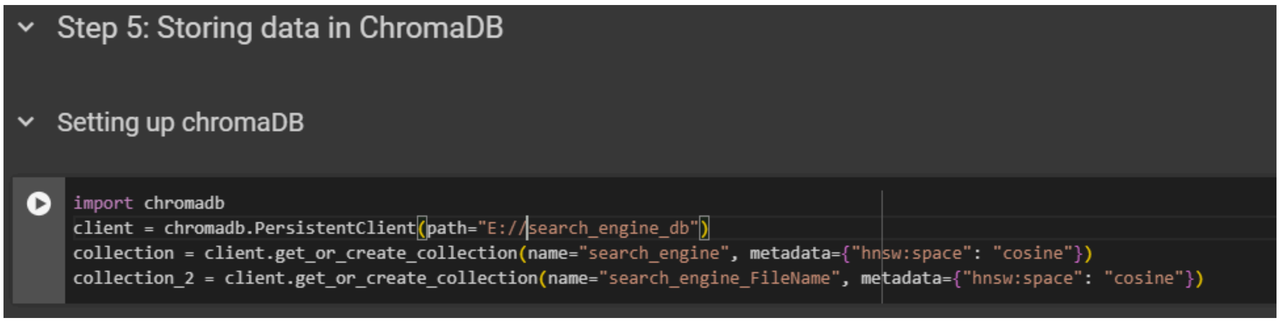
```
[ ]  df['embeddings'] = Parallel(n_jobs=-1)(delayed(embedding_gen)(item) for item in df['chunks'].values)
```

Figure 14: Creating text embeddings

Step 5: Storing data in vector database

Unlike traditional databases, vector databases or stores are specialized types of databases designed to store and manage information represented as vectors. These vectors are in turn used to perform similarity searches. In this project we used ChromaDB. Other alternatives include Pinecone, and Apache OpenSearch.

The snapshot below shows how we set up the ChromaDB for use.



Step 5: Storing data in ChromaDB

Setting up chromaDB

```
import chromadb
client = chromadb.PersistentClient(path="E://search_engine_db")
collection = client.get_or_create_collection(name="search_engine", metadata={"hnsw:space": "cosine"})
collection_2 = client.get_or_create_collection(name="search_engine_FileName", metadata={"hnsw:space": "cosine"})
```

Figure 15: Setting up chromaDB

We:

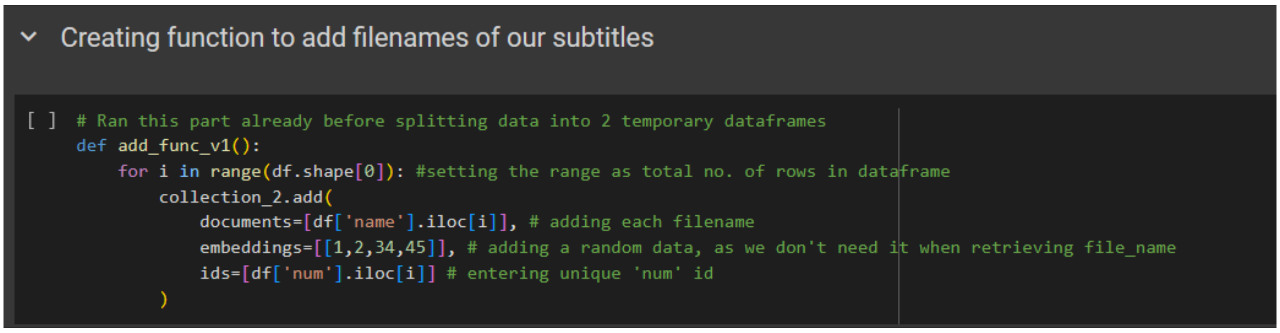
- Imported the **‘chromadb’** library,
- Created a persistent client to ensure long-lasting connection with the database during the program’s execution. This is created to prevent the RAM on Google Colab from crashing. It helps save the collection data in the local machine rather than the data being stored on RAM and being volatile.
- Created a collection called **‘search_engine’** where we stored the text embeddings (of the subtitle files) and unique identifiers or indexes. The metadata argument

(optional) provides additional information about our collection. ChromaDB uses an approximate nearest neighbor (ANN) search algorithm called Hierarchical Navigable Small World (HNSW) to find the most similar items to a given query. In this case we indicate that this collection uses the HNSW algorithm with cosine similarity for nearest neighbor searches.

- The second collection (collection_2) "**search_engine_FileName**" is created to store the corresponding file names of our subtitles document. It follows the same logic as the first collection.

Note that cosine similarity is used here in order to check for the similarity between vectors. Cosine similarity is particularly useful for tasks like ours involving semantic understanding and information retrieval in high-dimensional spaces.

Now that our collections are fully set up, Next we create two functions. The first one below is used to extract the **name** column from the dataframe as well as its unique identifier **num** and adding these documents into **collection_2** which we created earlier.

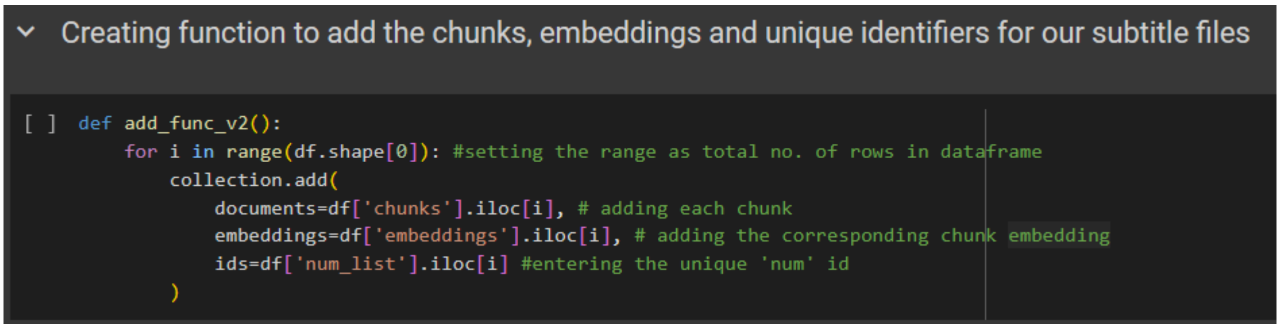


Creating function to add filenames of our subtitles

```
# Ran this part already before splitting data into 2 temporary dataframes
def add_func_v1():
    for i in range(df.shape[0]): #setting the range as total no. of rows in dataframe
        collection_2.add(
            documents=[df['name'].iloc[i]], # adding each filename
            embeddings=[[1,2,34,45]], # adding a random data, as we don't need it when retrieving file_name
            ids=[df['num'].iloc[i]] # entering unique 'num' id
        )
```

Figure 16: Creating a function to add filenames to collection_2

Similar to the function created above, the second one below adds the subtitle chunks created in the early stages of this project, alongside its embeddings and unique identifiers.



Creating function to add the chunks, embeddings and unique identifiers for our subtitle files

```
def add_func_v2():
    for i in range(df.shape[0]): #setting the range as total no. of rows in dataframe
        collection.add(
            documents=df['chunks'].iloc[i], # adding each chunk
            embeddings=df['embeddings'].iloc[i], # adding the corresponding chunk embedding
            ids=df['num_list'].iloc[i] #entering the unique 'num' id
        )
```

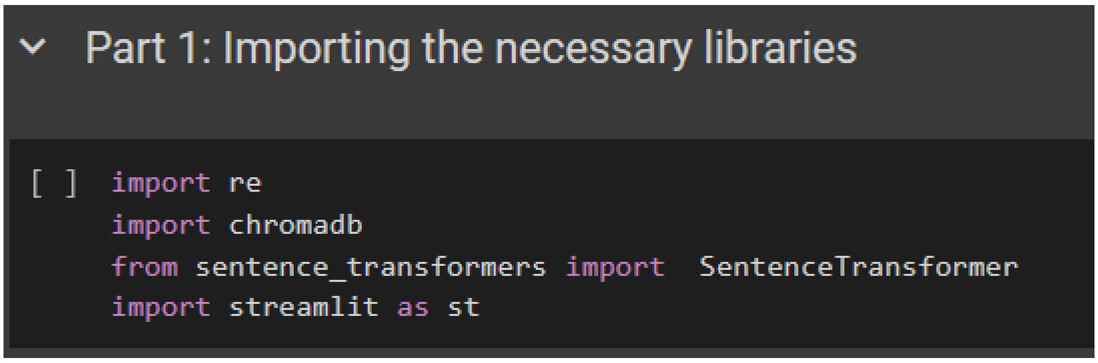
Figure 17: Function to add the chunks, embeddings and unique identifiers to our chromaDB collection

The reason for using a loop to iterate over each row of the dataframe is that chromaDB is restricted to accept an item as large as 5461 in length at a time. So if we pass the entire dataframe at once it will raise "ValueError: Batch size {size_of_df}exceeds maximum batch size 5461". That's why we iterate over each row and add it separately to avoid the ValueError. The entire process of adding the chunks and embeddings is estimated to take 50 hours. With the project deadline closeby and at the point of documentation we have successfully saved 30% of the 82,000+ files we started with, in **chromadb**. Aside from this we are ready to move to the next section.

Step 6: Web application creation

Using Streamlit we created a web application that allows a user to enter a part of a subtitle, click the search button and then get the 10 most relevant subtitle files for that query. Taking it a step further by linking each filename to a link to the OpenSubtitles page where the subtitle can be downloaded. The following code snippets show how we made this happen. Before going on, it is important for you to note that in order to get the expected results, the same actions performed on the subtitle files should be performed on the user query. More of that in the sections below.

First, we imported the needed libraries for this task.

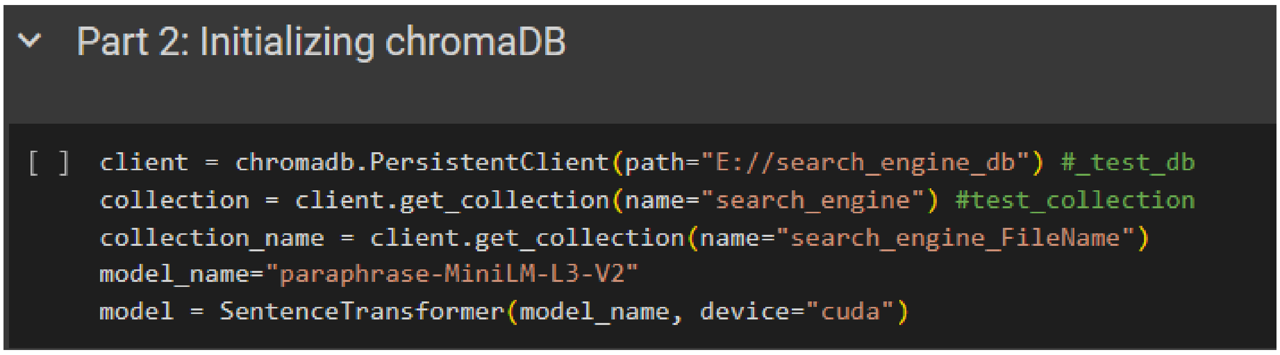


Part 1: Importing the necessary libraries

```
import re
import chromadb
from sentence_transformers import  SentenceTransformer
import streamlit as st
```

Figure 18: Importing libraries

Second, we establish a persistent connection. The path specifies the location of our database which in this case is **"E://search_engine_db"**. Next we retrieve the collections **"search_engine"** and **"search_engine_FileName"** to the variables *collection* and *collection_name* respectively. Lastly we define a variable containing the name of the pre-trained sentence transformer model (“**paraphrase-MiniLM-L3-V2**”) we used for text embeddings, and created a SentenceTransformer object with the device name set to cuda to indicate that the model will be loaded into GPU for faster processing

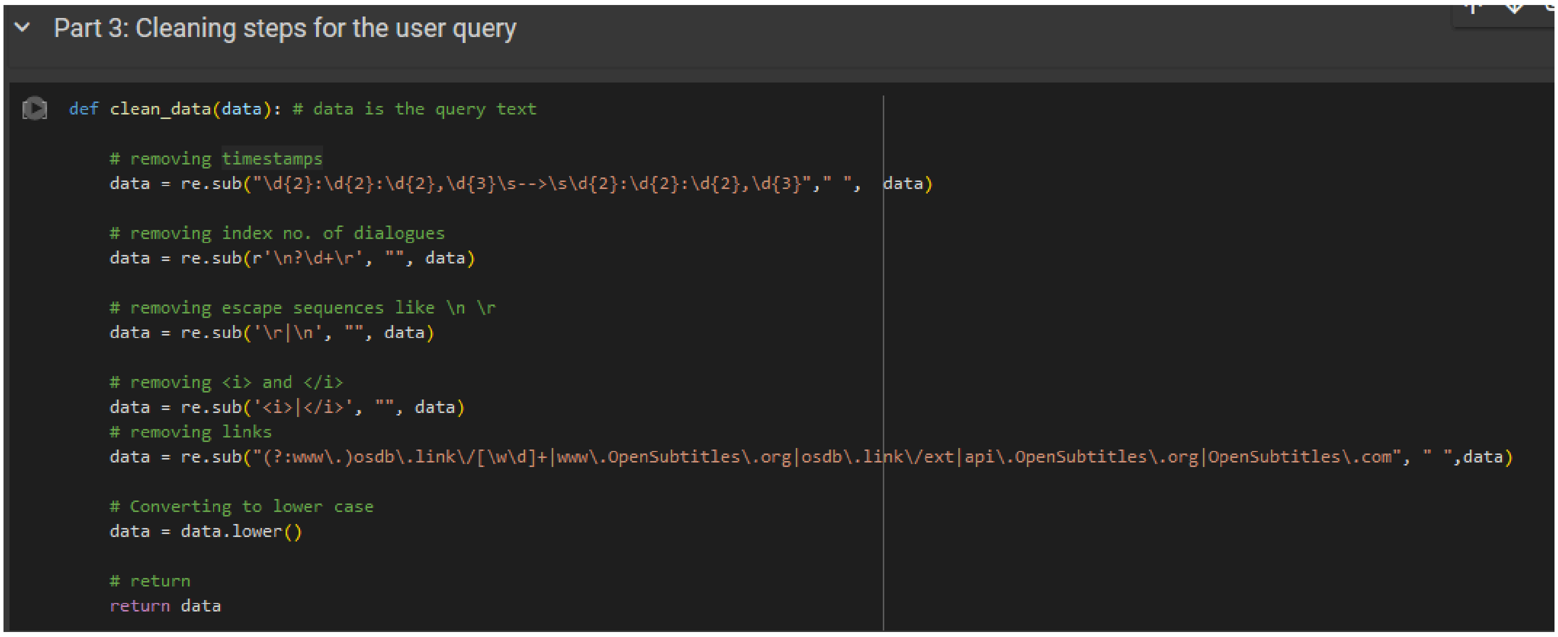


Part 2: Initializing chromaDB

```
[ ] client = chromadb.PersistentClient(path="E://search_engine_db") #_test_db
    collection = client.get_collection(name="search_engine") #test_collection
    collection_name = client.get_collection(name="search_engine_FileName")
    model_name="paraphrase-MiniLM-L3-V2"
    model = SentenceTransformer(model_name, device="cuda")
```

Figure 19: Initializing ChromaDB

Third, like mentioned earlier, the same cleaning step carried out in the subtitles database file is also performed on the user query.

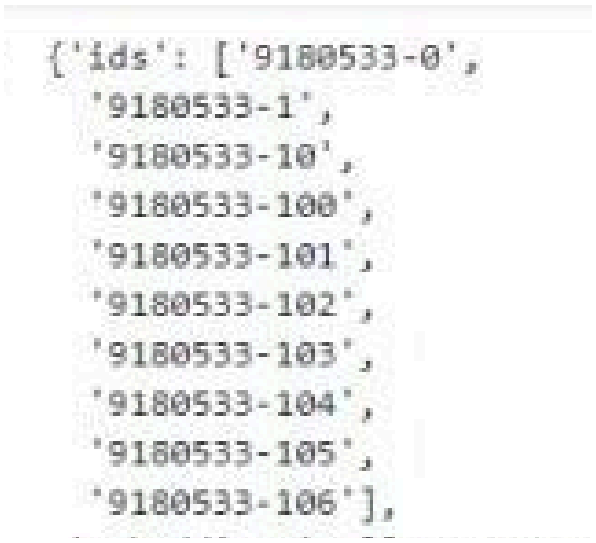


Part 3: Cleaning steps for the user query

```
def clean_data(data): # data is the query text

    # removing timestamps
    data = re.sub("\d{2}:\d{2}:\d{2},\d{3}\s-->\s\d{2}:\d{2}:\d{2},\d{3}"," ",  data)

    # removing index no. of dialogues
    data = re.sub(r'\n?\d+\r', "", data)

    # removing escape sequences like \n \r
    data = re.sub('\r|\n', "", data)

    # removing <i> and </i>
    data = re.sub('<i>|</i>', "", data)
    # removing links
    data = re.sub("(?:www\.)osdb\.link\/[\w\d]+|www\.OpenSubtitles\.org|osdb\.link\/ext|api\.OpenSubtitles\.org|OpenSubtitles\.com", " ",data)

    # Converting to lower case
    data = data.lower()

    # return
    return data
```

Figure 20: Cleaning steps

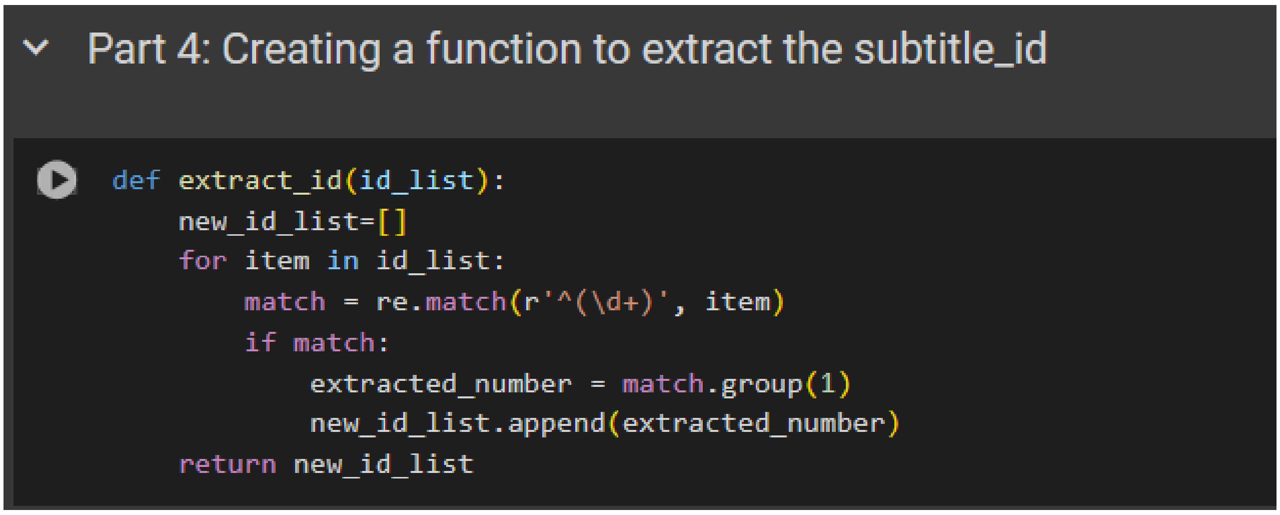
When the cleaning step is all done, the next step is to extract the subtitle id or unique identifier. Earlier in this report is a snapshot of what the modified ids looked like after chunking and text embeddings. Below is the snapshot once more. It shows some of the unique identifiers for the subtitle file chunks with the id **'9180533'.**



```
{'ids': ['9180533-0',
  '9180533-1',
  '9180533-10',
  '9180533-100',
  '9180533-101',
  '9180533-102',
  '9180533-103',
  '9180533-104',
  '9180533-105',
  '9180533-106'],
```

Figure 21: A view of the unique identifiers for subtitle chunks

The code block below is a function definition created to extract the digits before the hyphen. This extracted id is used to retrieve the name of the subtitle file in our second collection.

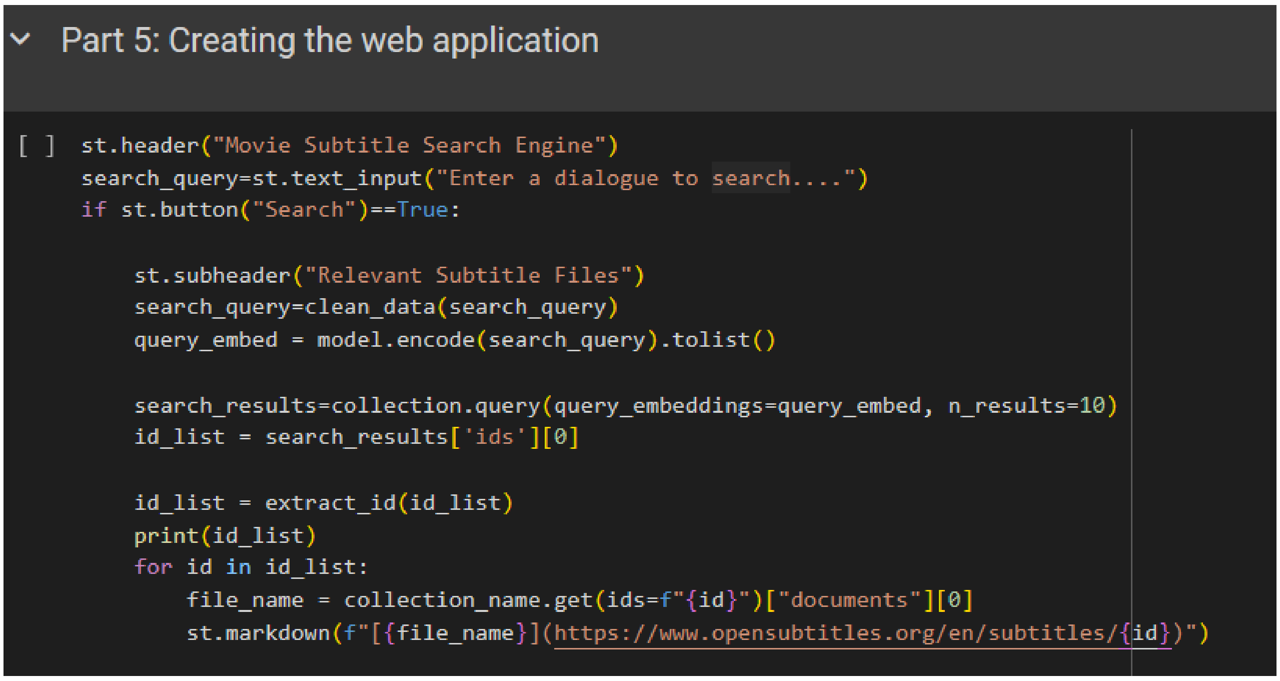


Part 4: Creating a function to extract the subtitle_id

```
def extract_id(id_list):
    new_id_list=[]
    for item in id_list:
        match = re.match(r'^(\d+)', item)
        if match:
            extracted_number = match.group(1)
            new_id_list.append(extracted_number)
    return new_id_list
```

Figure 22: Creating a function to extract the subtitle id

The last part of the application can be seen below:


Part 5: Creating the web application

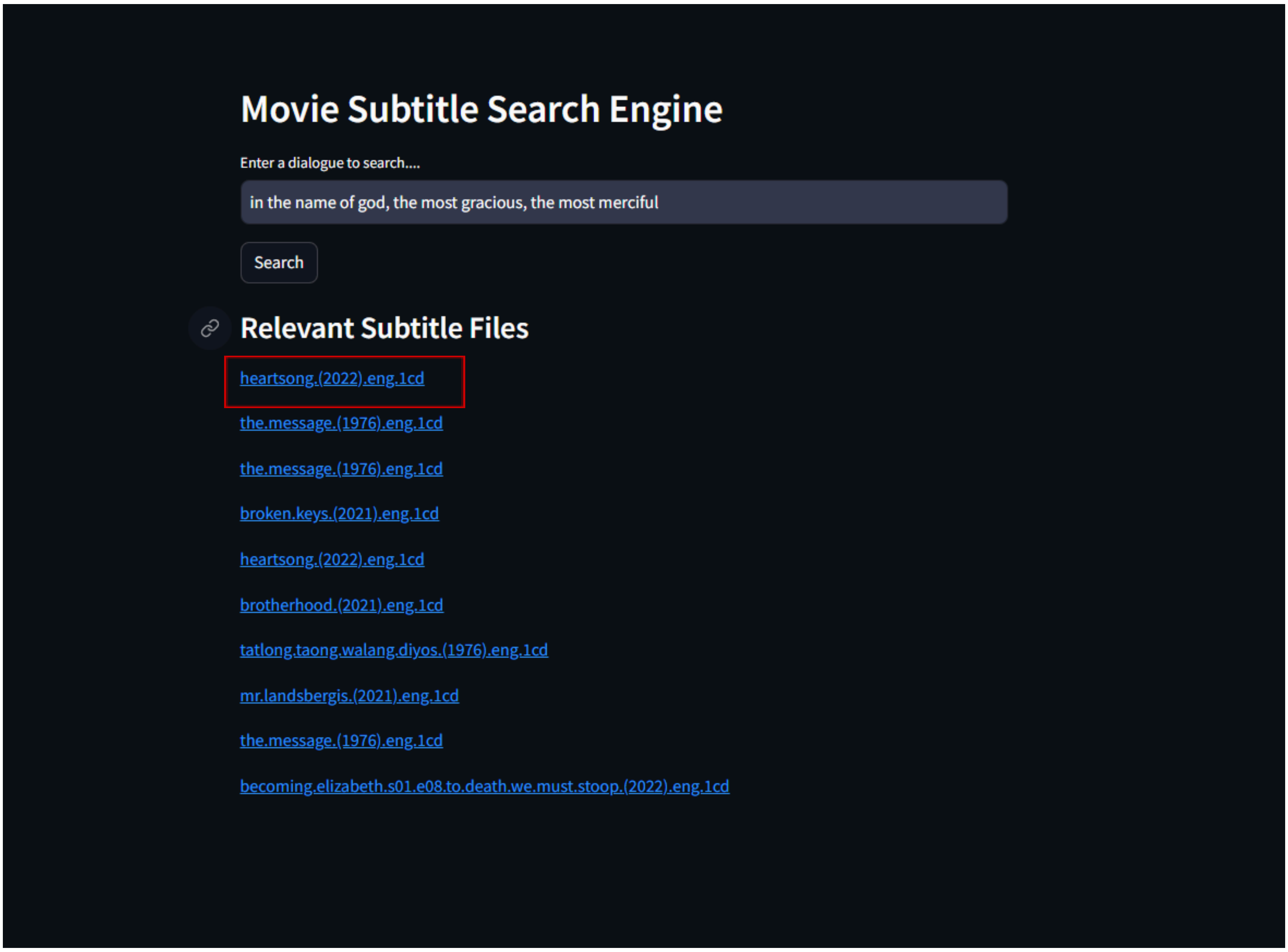
```
[ ]  st.header("Movie Subtitle Search Engine")
     search_query=st.text_input("Enter a dialogue to search....")
     if st.button("Search")==True:

         st.subheader("Relevant Subtitle Files")
         search_query=clean_data(search_query)
         query_embed = model.encode(search_query).tolist()

         search_results=collection.query(query_embeddings=query_embed, n_results=10)
         id_list = search_results['ids'][0]

         id_list = extract_id(id_list)
         print(id_list)
         for id in id_list:
             file_name = collection_name.get(ids=f"{id}")["documents"][0]
             st.markdown(f"[{file_name}](https://www.opensubtitles.org/en/subtitles/{id})")
```

Figure 23: Creating the web app using streamlit

We used the streamlit library to create an interactive web application. We defined a header for our web application and provided a text box for users to enter their query, in this case a dialogue from a movie or a TV show for which they would like to retrieve the subtitles related to it. The if conditional code block is set to true to indicate that the processes below it are to be carried out if and only if the user clicks the search button.

When the user clicks the search button, the text query is cleaned and converted into text embeddings. Thereafter cosine similarity is used to compute the cosine distance between the user's query vector and all the subtitle vectors in our collection and the top 10 closest files will be selected from the collection. This metric helps to identify subtitles that are semantically close to the user's intent, even if they don't contain the exact keywords.

The ids are retrieved from the first collection and used in the next part of the code to retrieve the corresponding file names in collection_name. This part of the code is created to dynamically attach the download links for each of the 10 results from the **OpenSubtitles.org** website. Below is a snapshot of the final web application we built.



Movie Subtitle Search Engine

Enter a dialogue to search....

in the name of god, the most gracious, the most merciful

Relevant Subtitle Files

heartsong.(2022).eng.1cd

the.message.(1976).eng.1cd

the.message.(1976).eng.1cd

broken.keys.(2021).eng.1cd

heartsong.(2022).eng.1cd

brotherhood.(2021).eng.1cd

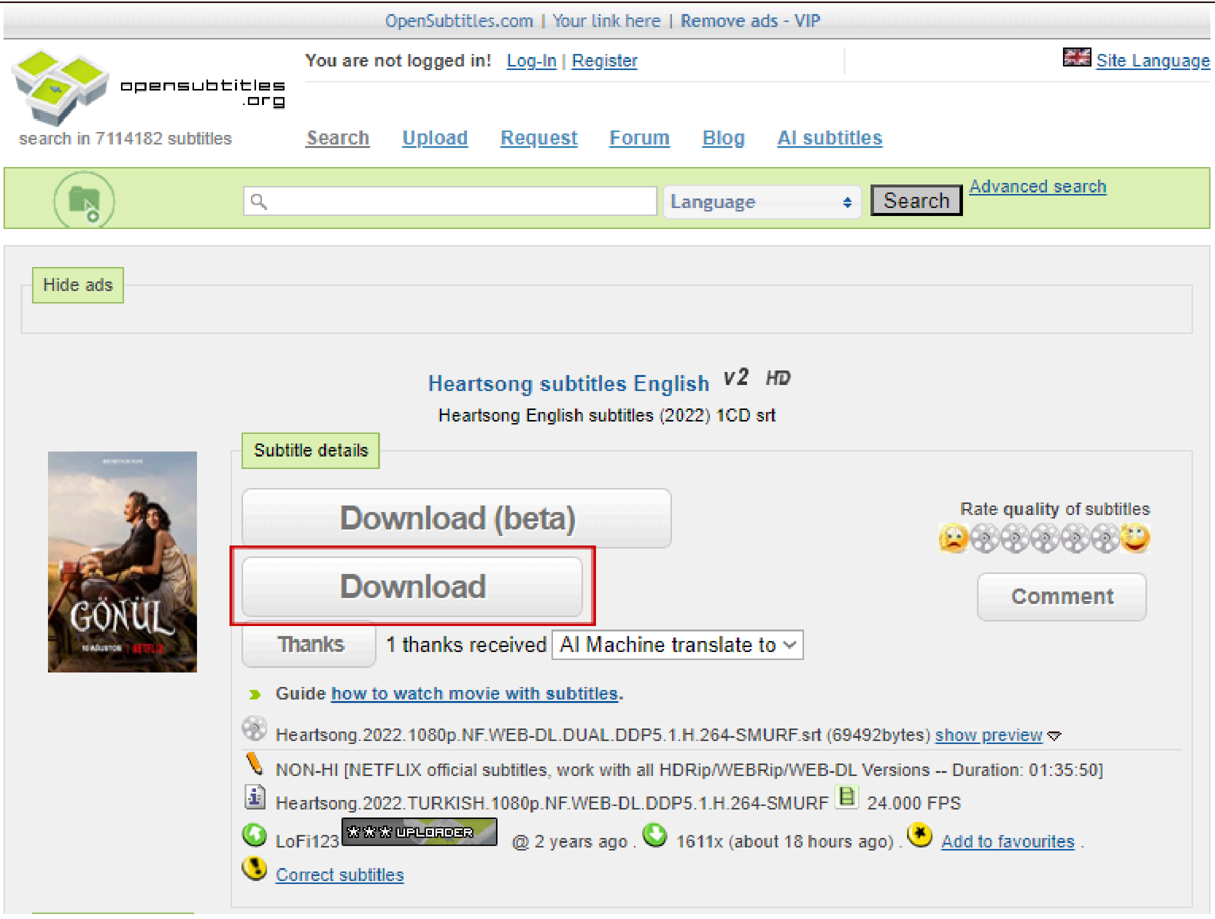
tatlong.taong.walang.diyos.(1976).eng.1cd

mr.landsbergis.(2021).eng.1cd

the.message.(1976).eng.1cd

becoming.elizabeth.s01.e08.to.death.we.must.stoop.(2022).eng.1cd

Figure 24: The Final application


OpenSubtitles.com | Your link here | Remove ads - VIP

You are not logged in! Log-In | Register

Site Language

opensubtitles.org

search in 7114182 subtitles

Search Upload Request Forum Blog AI subtitles

Language Search Advanced search

Hide ads

Heartsong subtitles English v2 HD

Heartsong English subtitles (2022) 1CD srt

Subtitle details

Download (beta)

Download

Thanks 1 thanks received AI Machine translate to

Rate quality of subtitles

Comment

Guide how to watch movie with subtitles.

Heartsong.2022.1080p.NF.WEB-DL.DUAL.DDP5.1.H.264-SMURF.srt (69492bytes) show preview

NON-HI [NETFLIX official subtitles, work with all HDRip/WEBRip/WEB-DL Versions -- Duration: 01:35:50]

Heartsong.2022.TURKISH.1080p.NF.WEB-DL.DDP5.1.H.264-SMURF 24.000 FPS

LoFi123 UPLOADER @ 2 years ago . 1611x (about 18 hours ago) . Add to favourites .

Correct subtitles

Figure 25: Redirection to OpenSubtitles.org page

Conclusion

In this project we built an MVP (**M**inimum **V**iable **P**roduct) that bridges the gap between a user's search intent and the content of video subtitles. We did this by utilizing techniques like light cleaning to preserve contextual meanings, semantic chunking on our subtitle files to mitigate information loss and created an iterative web app to test our solution. This is a glimpse of what can be done in building a semantic based search engine for video subtitles. We encourage exploring this concept deeper in building a more robust search engine.

References

The following articles, videos and documentations were instrumental during this project:

Pradip Nichite. (2023). *Semantic Search with Open-Source Vector DB: Chroma DB | Pinecone Alternative | Code*. Www.youtube.com. https://www.youtube.com/watch?v=eCCHDxMaFIk

Bakharia, A. (2020, January 19). *Quick Semantic Search using Siamese-BERT Networks*. Medium. https://towardsdatascience.com/quick-semantic-search-using-siamese-bert-networks-1052e7b4df1

Bricken, A. (2021a). *Does BERT Need Clean Data? Part 1: Data Cleaning.* Medium. https://towardsdatascience.com/part-1-data-cleaning-does-bert-need-clean-data-6a50c9c6e9fd

Chroma. *the AI-native open-source embedding database*. Www.trychroma.com. https://www.trychroma.com/

Bricken, A. (2021b). *Does BERT Need Clean Data? Part 2: Classification.* Medium. https://towardsdatascience.com/does-bert-need-clean-data-part-2-classification-d29adf9f745a

Hoffman, H. (2023). *Embeddings and Vector Databases With ChromaDB – Real Python*. Realpython.com. https://realpython.com/chromadb-vector-database/

Kamradt, G. *ChunkViz*. Chunkviz.up.railway.app. Retrieved 2024, from https://chunkviz.up.railway.app/#explanation

NeuralNine. (2023). *Parallelize Python Tasks with Joblib*. Www.youtube.com. https://www.youtube.com/watch?v=Dm4up8_zJdo

SBERT.net. (2024). *Pretrained Models — Sentence-Transformers documentation*. Www.sbert.net. https://www.sbert.net/docs/pretrained_models.html

Todeschini, S. (2023). *How to Chunk Text Data — A Comparative Analysis*. Medium. https://towardsdatascience.com/how-to-chunk-text-data-a-comparative-analysis-3858c4a0997a