**Celebrity Audio dataset**

**Objective:**

To build a clean and structured Indian celebrity voice dataset for training a voice cloning model. The dataset needed to be speaker-specific, validated, and of high enough quality to support model training for encoder and vocoder systems.

**What We Did:**

1. Data Collection
   * Gathered names of celebrities (actors, politicians, cricketers) from Wikipedia and IMDb.
   * Categorized them and created a master list in Excel.
2. YouTube Scraping
   * Wrote a Python script using the YouTube Data API v3 to fetch ~45 filtered videos per celebrity.
   * Excluded music or unrelated content using EXCLUDE\_KEYWORDS, and prioritized speeches, interviews, etc., using INCLUDE\_KEYWORDS.
   * Saved output as individual CSVs with video title, URL, and duration.
3. Voice Validation
   * Manually and using Grok AI, verified whether the voice in each video actually belonged to the celebrity.
   * Added yes/no labels in the CSV under a verification column (e.g., isspeaking).
4. Data Management
   * Managed team submissions, collected validated data, and maintained proper folder hierarchy (actor/politician/cricketer-wise).
   * Uploaded organized data to Mobaxterm server under structured IDs (e.g., ID10001 to ID12626).
5. Data Analysis
   * Wrote scripts to:
     + Calculate total valid speaking duration per celebrity.
     + Flag speakers with <2 hrs of data for further scraping.
     + Generate a summary report with CSV name, ID, speaking duration, and audio counts.
6. Audio Download & Final Dataset Creation
   * Developed a script to download YouTube audio using yt-dlp only for videos verified as “yes.”
   * Each audio file was saved in a subfolder named after a unique speaker ID.
7. Final Analysis
   * Analyzed cloned .wav files:
     + Counted per-speaker .wav files.
     + Calculated total duration.
     + Generated a report (cloning\_analysis\_report.csv) summarizing this data.

**Expected Outcome:**

* A clean, labelled, and validated dataset of over 2,600+ Indian public figures, each with verified speech samples.
* Each speaker has a dedicated folder of .wav files, suitable for voice cloning model training.
* Final outputs enable model performance for voice cloning to improve due to:
  + Consistent speaker labelling
  + High-quality, verified samples
  + Balanced class distribution

--------------------------------------------------------------------------------------------------------------------------------------

**Step-1. Name Collection**

* Began by extracting a comprehensive list of Indian celebrities (actors, politicians, and cricketers) from **Wikipedia and IMDb**.  
  <https://www.imdb.com/list/ls068010962/>

<https://en.wikipedia.org/wiki/Lists_of_Indian_actors>

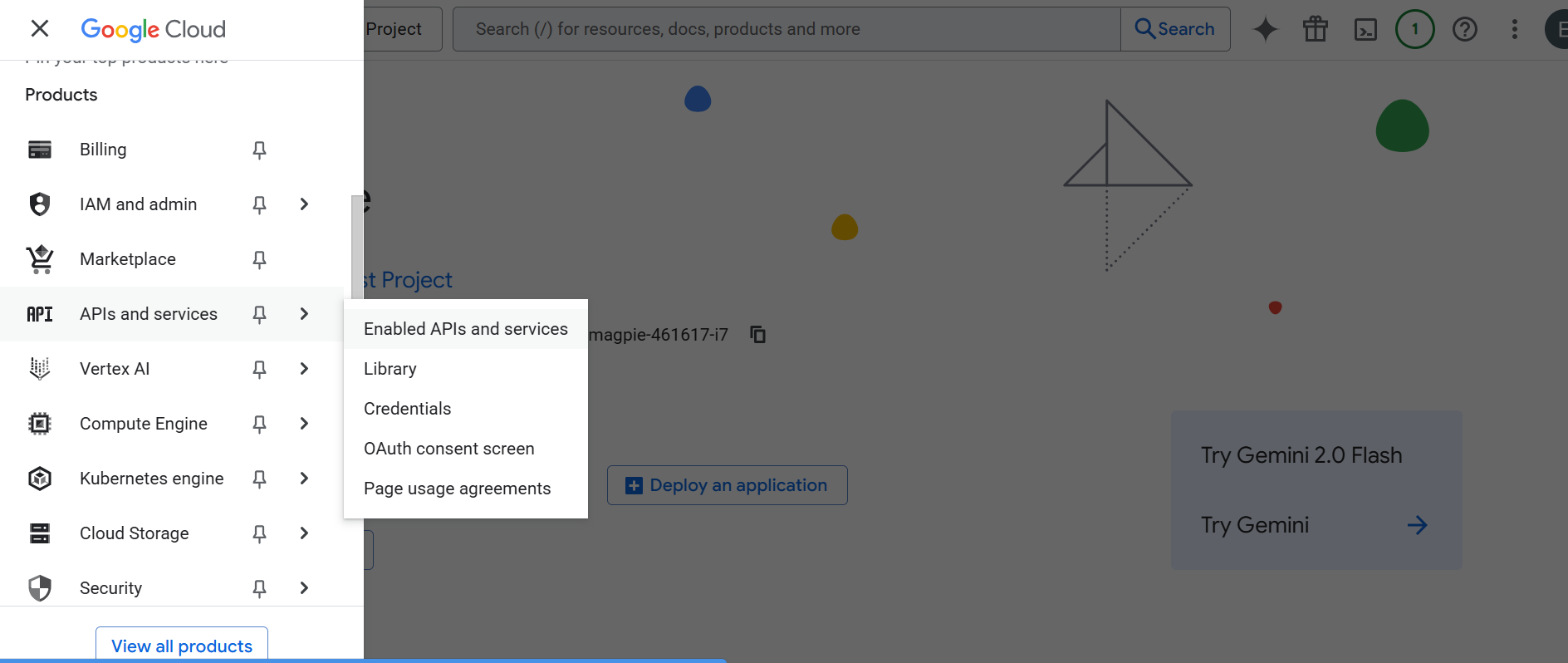
* This list was compiled into an **Excel sheet** for initial organization.

**Step-2. Categorization**

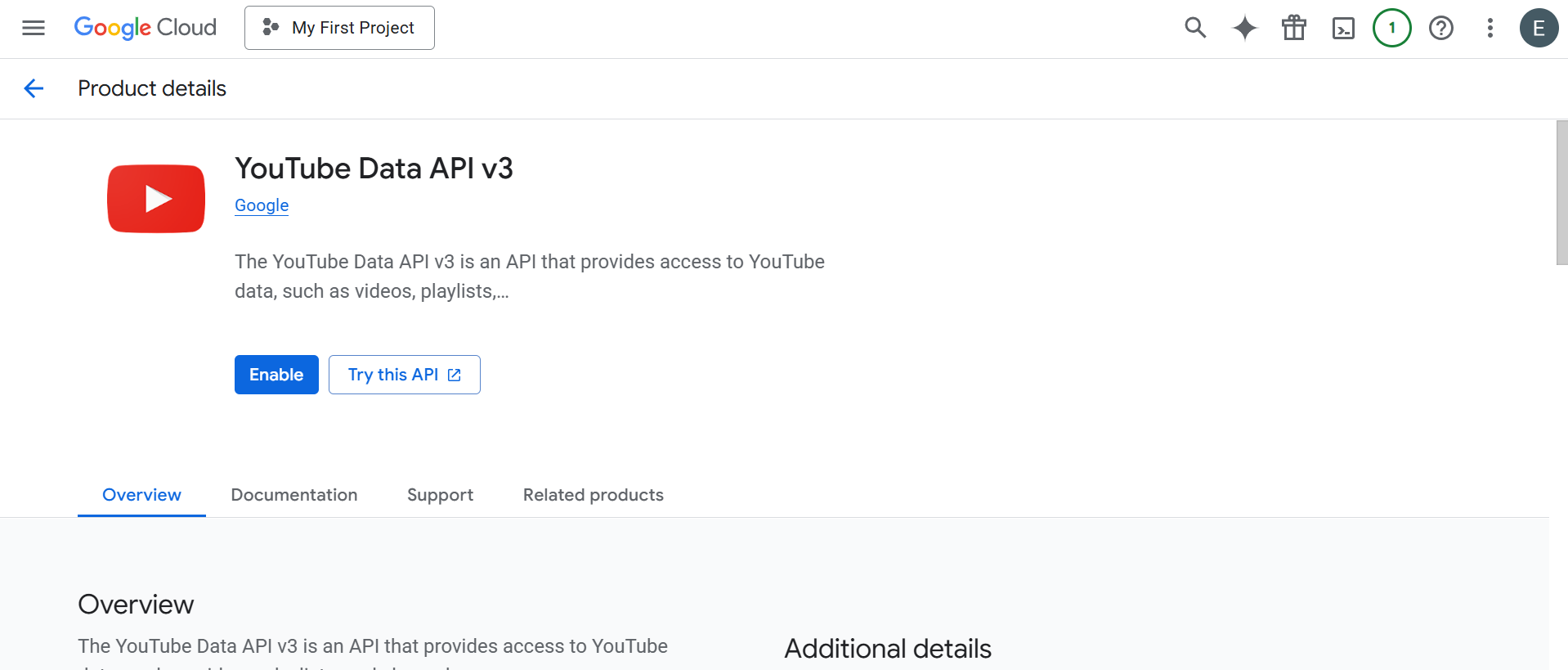
* All collected names were **manually categorized** into main groups and the subgroups:
  + **Actors:**
  + **Politicians:**
  + **Cricketers:**

**Step-3: Getting Youtube API**

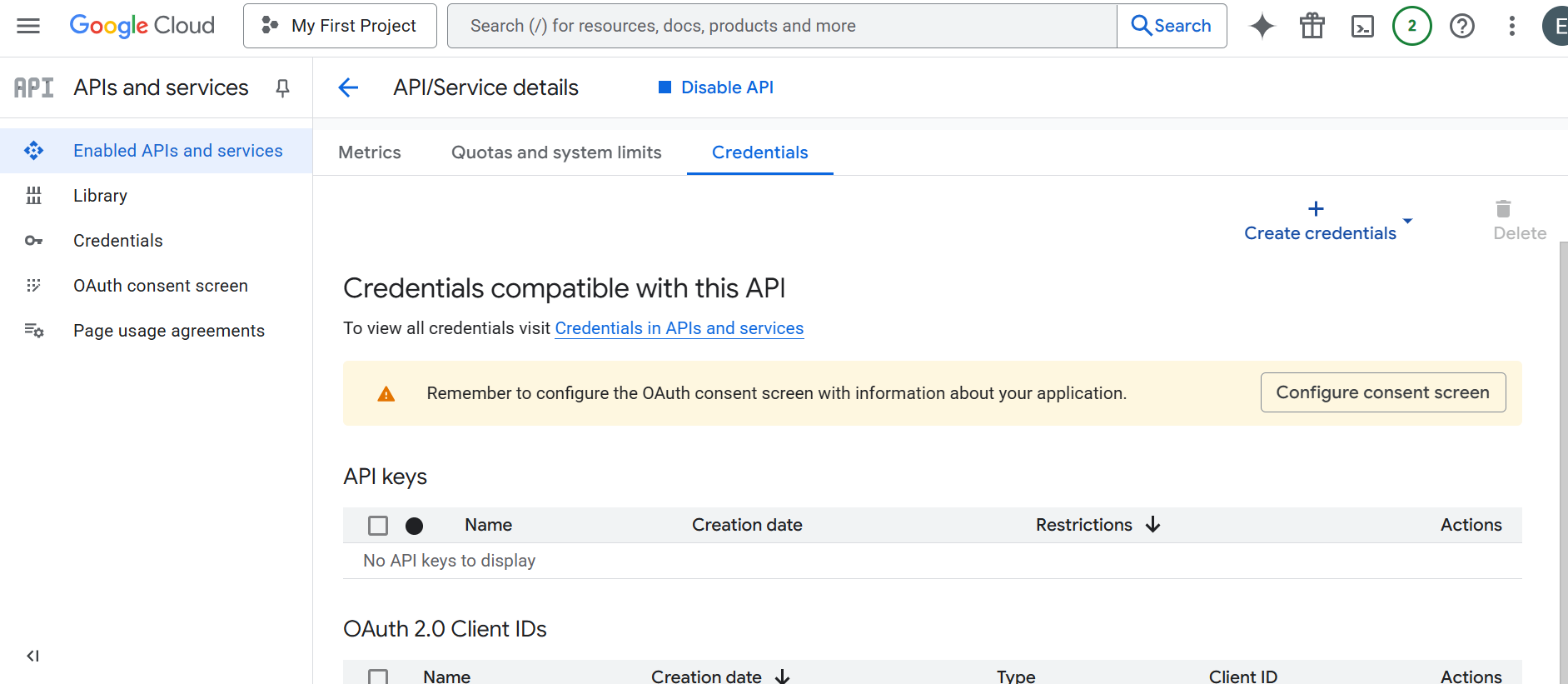
* <https://console.cloud.google.com/>



* Now select Enabled API and services.
* Search YouTube Data API v3



* Enable and Go to Credentials.



* Create Credentials and API Key.

**Step-4. YouTube Video Links Scraping Script**

* Python script that:
  + Read each name from the categorized list.
  + Queried the YouTube Data API to fetch up to ~50 videos per person.
  + Filtered the results using keywords (to exclude music videos and prioritize interviews/speeches).
  + Fetched metadata like video URL, title, and duration.
  + Stored the result for each individual in a separate CSV file.

Below Code was used to Fetch youtube videos for each Celebrity:

youtube\_fetch.py:

import csv

import time

import re

from googleapiclient.discovery import build

import isodate  # ✅ Converts ISO 8601 duration to HH:MM:SS

API\_KEY = "AIzaSyADqWkDCaTPEHvVMCDf5EMrt0c\_i-U3M1k"

# ✅ Define keywords to filter out song-related content

EXCLUDE\_KEYWORDS = ["song", "lyric", "music", "audio", "official video", "album", "track"]

# ✅ Define keywords to prioritize movies, interviews, and shows

INCLUDE\_KEYWORDS = ["full movie", "interview", "episode", "scene", "speech", "talk"]

# ✅ Fetch video durations

def get\_video\_details(video\_ids, youtube):

    video\_request = youtube.videos().list(

        part="contentDetails",

        id=",".join(video\_ids)

    )

    video\_response = video\_request.execute()

    video\_durations = {}

    for item in video\_response.get("items", []):

        video\_id = item["id"]

        duration\_iso = item["contentDetails"]["duration"]

        duration = str(isodate.parse\_duration(duration\_iso))  # Convert to HH:MM:SS

        video\_durations[video\_id] = duration

    return video\_durations

# ✅ Fetch YouTube videos while filtering song-related content

def get\_youtube\_videos(query, max\_results=15):

    youtube = build("youtube", "v3", developerKey=API\_KEY)

    all\_videos = []

    next\_page\_token = None

    while len(all\_videos) < max\_results:

        request = youtube.search().list(

            part="snippet",

            q=query,

            type="video",

            maxResults=min(15, max\_results - len(all\_videos)),

            pageToken=next\_page\_token

        )

        response = request.execute()

        video\_ids = []

        filtered\_videos = []

        for item in response["items"]:

            if item["id"]["kind"] == "youtube#video":

                title = item["snippet"]["title"].lower()

                description = item["snippet"].get("description", "").lower()

                # ✅ Exclude song-related videos

                if any(re.search(rf"\b{word}\b", title) for word in EXCLUDE\_KEYWORDS):

                    continue  # Skip song-related videos

                # ✅ Prioritize movies, interviews, and shows

                if any(re.search(rf"\b{word}\b", title) for word in INCLUDE\_KEYWORDS) or \

                   any(re.search(rf"\b{word}\b", description) for word in INCLUDE\_KEYWORDS):

                    video\_id = item["id"]["videoId"]

                    video\_ids.append(video\_id)

                    filtered\_videos.append(item)

        # ✅ Get durations for the filtered videos

        video\_durations = get\_video\_details(video\_ids, youtube)

        for item in filtered\_videos:

            video\_id = item["id"]["videoId"]

            title = item["snippet"]["title"]

            video\_url = f"https://www.youtube.com/watch?v={video\_id}"

            duration = video\_durations.get(video\_id, "N/A")

            all\_videos.append({"title": title, "url": video\_url, "duration": duration})

        next\_page\_token = response.get("nextPageToken")

        if not next\_page\_token:

            break

        time.sleep(1)  # Avoid hitting API rate limits

    return all\_videos

# ✅ Save to CSV

def save\_to\_csv(videos, filename):

    with open(filename, "w", newline="", encoding="utf-8") as file:

        writer = csv.DictWriter(file, fieldnames=["title", "url", "duration"])

        writer.writeheader()

        writer.writerows(videos)

# ✅ Read celebrity names from a file

def read\_celebrities\_from\_file(filename):

    with open(filename, "r", encoding="utf-8") as file:

        return [line.strip() for line in file if line.strip()]

# ✅ Process multiple celebrities in batches

def process\_batches(celeb\_list, batch\_size=60):

    for i in range(0, len(celeb\_list), batch\_size):

        batch = celeb\_list[i:i + batch\_size]

        print(f"Processing batch {i//batch\_size + 1}: {batch}")

        for celeb in batch:

            video\_list = get\_youtube\_videos(celeb, max\_results=45)

            csv\_filename = f"{celeb.replace(' ', '\_')}\_videos.csv"

            save\_to\_csv(video\_list, csv\_filename)

            print(f"✅ Saved: {csv\_filename}")

# ✅ Load celebrity names and process

celebrities = read\_celebrities\_from\_file("temporary.txt")

process\_batches(celebrities, batch\_size=60)

**Code Explanation: YouTube Video Scraper and Filter for Celebrity Content**

This Python script automates the process of retrieving and filtering YouTube video data for a list of celebrity names. It is especially designed to **exclude music content** and **prioritize interviews, speeches, and full movies**—making it useful for research, media analysis, or training datasets (e.g., for voice/speech models).

1. **Modules Used**

* csv: To write output into structured CSV files.
* time: Adds delay to avoid exceeding YouTube API request limits.
* re: Regular expressions to filter video titles and descriptions.
* googleapiclient.discovery: To access YouTube Data API v3.
* isodate: Converts YouTube’s video durations (ISO 8601 format) to HH:MM:SS.

1. **API Key Setup**

API\_KEY = "AIzaSy..."

* The script uses a YouTube Data API key to authenticate and perform search and video detail operations.

1. **Filtering Logic**

Two sets of keywords are defined:

EXCLUDE\_KEYWORDS = ["song", "lyric", "music", "audio", "official video", "album", "track"]

INCLUDE\_KEYWORDS = ["full movie", "interview", "episode", "scene", "speech", "talk"]

* **Excludes** music-related videos (e.g., songs, tracks, lyrics).
* **Includes** videos related to interviews, speeches, and full-length media.

1. **Function Overview**

**get\_video\_details(video\_ids, youtube)**

* Accepts a list of video IDs and returns a dictionary of video durations (HH:MM:SS format).

**get\_youtube\_videos(query, max\_results=15)**

* Searches YouTube for videos matching a **celebrity name**.
* Filters out irrelevant (music-related) results.
* Fetches metadata (title, URL, duration) for prioritized videos.
* Returns a list of dictionaries.

**save\_to\_csv(videos, filename)**

* Saves the filtered video data to a .csv file with columns: title, url, duration.

**read\_celebrities\_from\_file(filename)**

* Reads celebrity names line-by-line from a given file (temporary.txt).

**process\_batches(celeb\_list, batch\_size=60)**

* Processes celebrity names in batches to avoid overloading the API.
* For each celebrity, it:
  + Searches and filters videos.
  + Saves results to a unique CSV file (named after the celebrity)

1. **Main Execution**

celebrities = read\_celebrities\_from\_file("temporary.txt")

process\_batches(celebrities, batch\_size=60)

* The script reads **INPUT** as celebrity names from temporary.txt and processes them in groups of 60.
* **OUTPUT** is saved as multiple CSV files like Shah\_Rukh\_Khan\_videos.csv.

**Step-5. Validity Check of Videos**

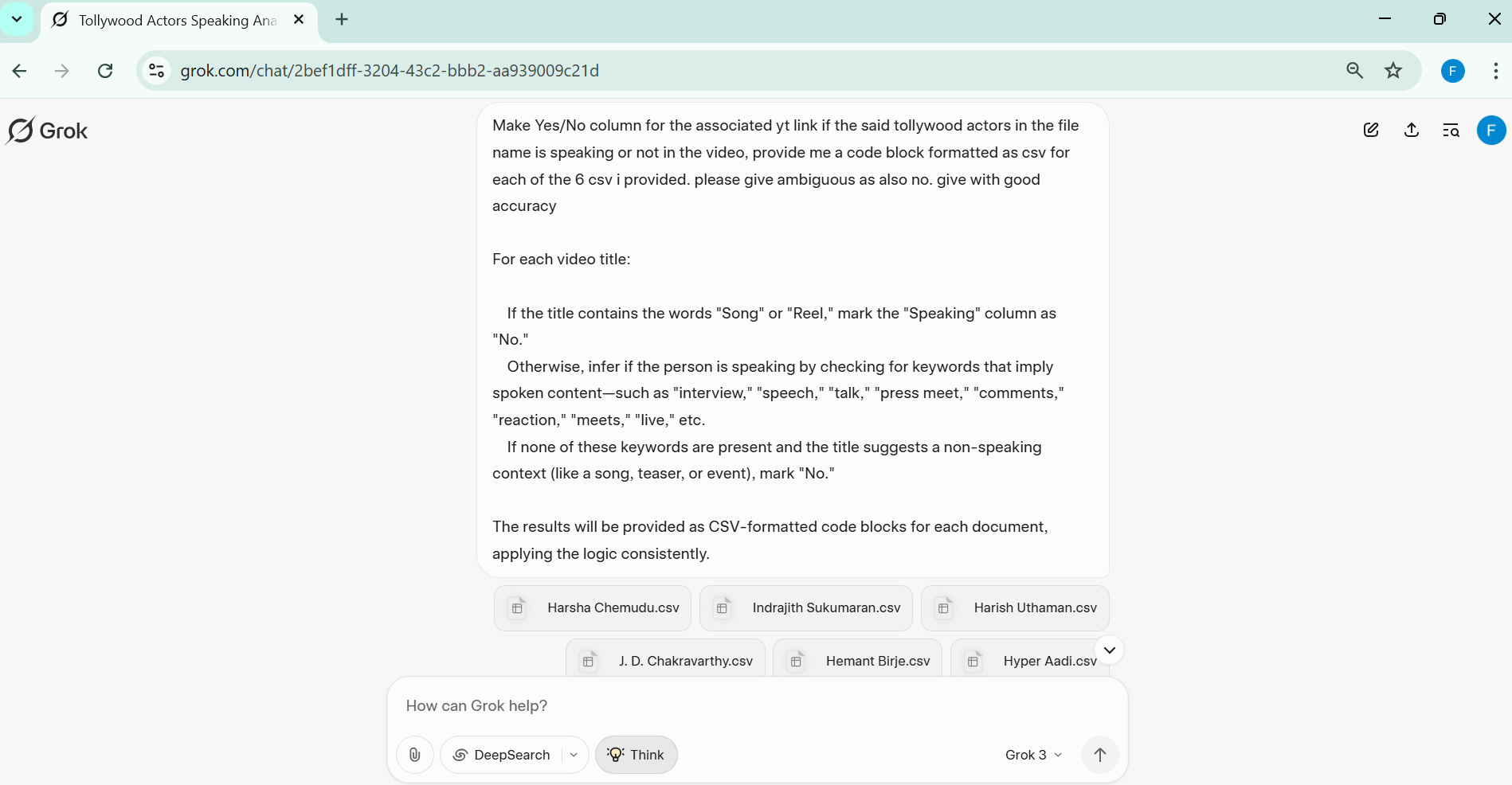
* Each video entry was validated to ensure the presence of the celebrity’s voice:
  + This step was carried out both manually and using Grok AI.
  + For each video, a label (yes or no) was assigned depending on whether the person's voice was clearly present.
* For Grok AI verification:
* Open <https://x.ai/grok>
* Give below given prompt and attach the Celebrity CSV files from your folder:

“Make Yes/No column for the associated yt link if the said tollywood actors in the file name is speaking or not in the video, provide me a code block formatted as csv for each of the 6 csv i provided. please give ambiguous as also no. give with good accuracy

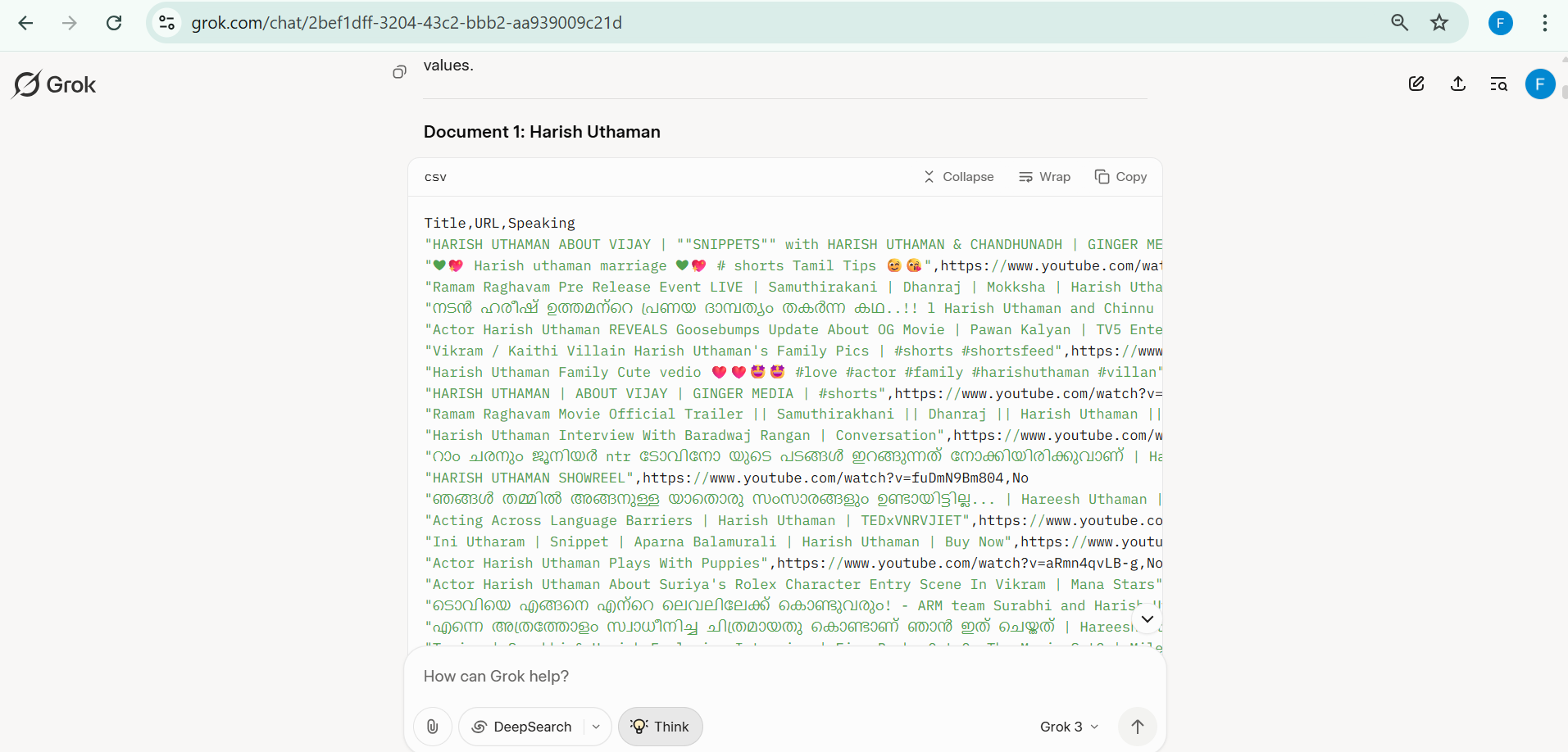
For each video title:

If the title contains the words "Song" or "Reel," mark the "Speaking" column as "No." Otherwise, infer if the person is speaking by checking for keywords that imply spoken content—such as "interview," "speech," "talk," "press meet," "comments," "reaction," "meets," "live," etc. If none of these keywords are present and the title suggests a non-speaking context (like a song, teaser, or event), mark "No."

The results will be provided as CSV-formatted code blocks for each document, applying the logic consistently.”



* Copy and paste CSV Blocks in VS Code for that respective celebrity



**Step-6. Centralized Data Collection**

* Each team member worked on different subsets of celebrities and I:
  + Responsibility for collecting, consolidating, and organizing all the validated CSVs.
  + Maintaining folders for each celebrity under their respective categories (Actor, Politician, Cricketer).
  + Ensured data consistency, file naming conventions, and folder hierarchy across contributions from the team.

**Step- 6. The Duration of Total Data Collected Script**

* Script to perform duration of data collected on the validated dataset:
  + The script computed the total speaking time per celebrity using video durations.
  + If a person had more than 2–3 hours of validated audio duration, they were marked as “done”.
  + For those with insufficient data, more videos were scraped and validated again.

Below Code was used for Data Analysis of the duration of videos for each Celebrity:

count.py:

import pandas as pd

import os

import requests

import isodate

import re

# Specify the folder path containing your CSV files.

folder\_path = r"C:\Users\lenovo\Desktop\CDAC Python\Celebrity Audio dataset\CONTRI\final\Famous Cricketers(Final)1\Famous Cricketers(Final)1" # Update this path if needed

# Insert your valid YouTube API key here.

api\_key = "AIzaSyB7LVIvQjtZg2FunF\_LrCbokARbOljtrm8"

def extract\_video\_id(url: str) -> str:

    """

    Extracts a YouTube video ID from a given URL using a regex pattern.

    Returns None if no valid 11-character video ID is found.

    """

    if not isinstance(url, str):

        print(f"Invalid URL format (not a string): {url}")

        return None

    match = re.search(r"(?:v=|youtu\.be/|embed/|shorts/|/v/|/e/|watch\\?v=)([a-zA-Z0-9\_-]{11})", url)

    if match:

        video\_id = match.group(1)

        print(f"Extracted video ID: {video\_id} from URL: {url}")

        return video\_id

    else:

        print(f"No valid video ID found in URL: {url}")

        return None

def get\_video\_duration(video\_ids, api\_key):

    """

    Given a list of YouTube video IDs and an API key, returns a dictionary

    mapping each video ID to its duration in seconds.

    """

    durations = {}

    if not video\_ids:

        print("No video IDs provided to fetch durations.")

        return durations

    print(f"Fetching durations for video IDs: {video\_ids}")

    url = (

        "https://www.googleapis.com/youtube/v3/videos"

        f"?part=contentDetails&id={','.join(video\_ids)}&key={api\_key}"

    )

    try:

        response = requests.get(url)

        response.raise\_for\_status()

        data = response.json()

        if "items" in data:

            for item in data["items"]:

                vid\_id = item["id"]

                duration\_str = item["contentDetails"]["duration"]

                duration\_seconds = isodate.parse\_duration(duration\_str).total\_seconds()

                durations[vid\_id] = duration\_seconds

                print(f"Video ID {vid\_id} duration: {duration\_seconds} seconds")

        else:

            print("No items found in API response:", data)

    except Exception as e:

        print(f"Error fetching video durations: {e}")

    return durations

def try\_read\_csv(file\_path):

    """

    Attempt to read CSV with multiple encodings.

    Returns DataFrame if successful, None if all attempts fail.

    """

    encodings = ['utf-8', 'latin-1', 'windows-1252', 'iso-8859-1']

    for encoding in encodings:

        try:

            df = pd.read\_csv(file\_path, on\_bad\_lines='skip', engine='python', encoding=encoding)

            print(f"Successfully read {os.path.basename(file\_path)} with {encoding} encoding")

            return df

        except Exception as e:

            print(f"Failed to read {os.path.basename(file\_path)} with {encoding}: {str(e)}")

    return None

# List to hold summary information and problematic files

summary\_data = []

problematic\_files = []

# Loop through each file in the folder.

for filename in os.listdir(folder\_path):

    if filename.endswith('.csv'):

        file\_path = os.path.join(folder\_path, filename)

        print(f"\nProcessing file: {filename}")

        # Try to read the CSV with multiple encodings

        df = try\_read\_csv(file\_path)

        if df is None:

            print(f"Skipping {filename} due to persistent encoding issues")

            problematic\_files.append(filename)

            continue

        try:

            # Find all columns that contain "manual" or "cross" (case-insensitive).

            target\_columns = [col for col in df.columns if any(x in col.lower() for x in ["manual", "cross", "isspeaking"])]

            print(f"Columns containing 'manual' or 'cross' found: {target\_columns}")

            # Check for any column that matches "url" (case-insensitive)

            url\_column = next((col for col in df.columns if col.lower() in ["url"]), None)

            if url\_column:

                print(f"Url column found: {url\_column}")

            else:

                print("No Url column found in this file.")

            # Initialize counts and duration

            yes\_count = 0

            total\_duration\_seconds = 0

            # If no matching column is found, set yes\_count to 0.

            if not target\_columns:

                print(f"No column containing 'manual' or 'cross' found in {filename}. Setting Total\_YES = {yes\_count}")

            else:

                # Count YES and gather video IDs in all matching columns.

                all\_video\_ids = []

                for col in target\_columns:

                    # Debug: Show unique values in the column

                    unique\_values = df[col].dropna().unique()

                    print(f"Unique values in '{col}' column: {unique\_values}")

                    # Filter rows where this column indicates affirmation (case-insensitive)

                    yes\_rows = df[df[col].astype(str).str.strip().str.lower().isin(['yes', 'y', 'true', '1'])]

                    count = len(yes\_rows)

                    yes\_count += count

                    print(f"In file {filename}, column '{col}' has {count} affirmative values (yes/y/true/1)")

                    # If Url column exists, extract video IDs

                    if url\_column and count > 0:

                        col\_video\_ids = yes\_rows[url\_column].dropna().apply(extract\_video\_id).dropna().tolist()

                        print(f"Video IDs extracted from column '{col}': {col\_video\_ids}")

                        all\_video\_ids.extend(col\_video\_ids)

                    else:

                        print(f"No video IDs extracted from column '{col}' (Url column missing or no affirmative rows).")

                # Calculate duration if we have video IDs

                if all\_video\_ids:

                    print(f"All video IDs collected: {all\_video\_ids}")

                    unique\_video\_ids = list(set(all\_video\_ids))

                    video\_durations = get\_video\_duration(unique\_video\_ids, api\_key)

                    total\_duration\_seconds = sum(video\_durations.get(vid, 0) for vid in all\_video\_ids)

                    print(f"Total duration for {filename}: {total\_duration\_seconds} seconds")

                else:

                    print("No video IDs found for duration calculation.")

            # Convert duration to formatted string

            hours = int(total\_duration\_seconds // 3600)

            minutes = int((total\_duration\_seconds % 3600) // 60)

            seconds = int(total\_duration\_seconds % 60)

            formatted\_duration = f"{hours}h {minutes}m {seconds}s"

            # Append the file name, YES count, and duration to the summary list.

            summary\_data.append({"FileName": filename, "Total\_YES": yes\_count, "Total\_Duration": formatted\_duration})

        except Exception as e:

            print(f"Error processing {filename} after reading: {e}")

            problematic\_files.append(filename)

# Create a summary DataFrame.

summary\_df = pd.DataFrame(summary\_data)

# Define the summary CSV file path.

summary\_file\_path = os.path.join(folder\_path, "summary.csv")

# Remove the existing summary file if it exists.

if os.path.exists(summary\_file\_path):

    try:

        os.remove(summary\_file\_path)

    except Exception as e:

        print(f"Could not remove existing summary file: {e}")

# Save the summary DataFrame to a CSV file.

try:

    summary\_df.to\_csv(summary\_file\_path, index=False)

    print(f"Summary saved to {summary\_file\_path}")

except Exception as e:

    print(f"Error saving summary CSV: {e}")

# Report problematic files

if problematic\_files:

    print("\nProblematic files that couldn't be fully processed:")

    for file in problematic\_files:

        print(f"- {file}")

**Code Explanation:**

This Python script processes a collection of .csv files containing metadata for YouTube videos linked to Celebrity category (Actor, Politician or Cricketer). The script identifies how many samples are marked as **"YES"** (i.e., valid speech samples) and calculates the **total duration** of corresponding YouTube videos. It then generates a **summary CSV file** with this data.

1. **Modules Used**

* **pandas**: For reading and processing tabular .csv data.
* **os**: For file and path handling.
* **requests**: To send HTTP requests to the YouTube API.
* **isodate**: To convert YouTube video duration from ISO 8601 to seconds.
* **re**: For extracting video IDs using regular expressions.

1. **Inputs & Setup**

folder\_path = r"..."

api\_key = "AIzaSy..."

* **INPUT** folder\_path: Directory containing .csv files for each cricketer.
* api\_key: Your YouTube Data API key used for fetching video metadata.

1. **Function Breakdown**

**extract\_video\_id(url: str)**

* Extracts the **11-character video ID** from various YouTube URL formats using regular expressions.

**get\_video\_duration(video\_ids, api\_key)**

* Sends a request to YouTube’s Data API to fetch contentDetails for a list of video IDs.
* Extracts duration in ISO format and converts it to **seconds** using isodate.

**try\_read\_csv(file\_path)**

* Attempts to read each .csv file using multiple fallback encodings (utf-8, latin-1, etc.).
* Handles encoding errors gracefully to improve robustness.

1. **Main Script Workflow**
2. **Iterate overall .csv files** in the folder.
3. **Read each file** using robust encoding strategies.
4. **Identify target columns** that may indicate speech sample labels:
   * Column names containing "manual", "cross", or "isspeaking" (case-insensitive).
5. **Identify the YouTube URL column** (url).
6. For each matched column:
   * **Count rows marked as “yes”** (or equivalent: "yes", "y", "true", "1").
   * **Extract YouTube video IDs** from the URL column for these rows.
7. **Calculate total duration** of valid videos (those marked “yes”).
8. **Format duration** to Xh Xm Xs format.
9. **Log summary data**: File name, count of YES-marked entries, and total duration.
10. **Save summary** to a file named summary.csv inside the same folder.
11. **Output**

A CSV file named summary.csv is created with the following columns:

* FileName: Name of the original data file.
* Total\_YES: Number of "yes" samples indicating usable speech content.
* Total\_Duration: Combined duration of associated YouTube videos.

**Step-8. Data Upload**

* After processing, cleaning, and verification:
  + The final organized dataset (CSV files + audio samples) was uploaded to the Mobaxterm server under a central directory maintained by Hruturaj Nikam Sir.
* Category and speaker count downloading done till now:

1. Male cricket captains= ID10001-ID10035 (Total are 35)

2. Marathi Actors= ID10036-ID10209 (Total are 174)

3. Famous cricketers= ID10210-ID10451 (Total are 242)

4. Ministers= ID10452-ID10498 (Total are 47)

5. Indian Actress= ID10499-ID11022 (Total are 524)

6. Tollywood Actress= ID11023-ID11485 (Total are 463)

7. Tollywood Actors= ID11486-ID11921 (Total are 436)

8. Marathi actress= ID11922-ID12166 (Total are 245)

9. Indian Actors= ID12167-ID12626 (Total are 459)

Total No. = 2,625

**Step-9: Doing download\_mapping**

downloadassir\_mapping.py:

#!/usr/bin/env python3

import os

import csv

import subprocess

import chardet # Install using: pip install chardet

import hashlib

FFMPEG\_LOCATION = "" # Leave empty if ffmpeg is in your system PATH

COOKIES\_FILE = os.path.expanduser("cookies.txt")

MASTER\_FILE = "downloaded\_links.txt"

FILENAME\_MAX\_LENGTH = 100

def detect\_encoding(file\_path, sample\_size=10000):

with open(file\_path, "rb") as f:

raw\_data = f.read(sample\_size)

result = chardet.detect(raw\_data)

return result["encoding"] if result["encoding"] else "utf-8"

def sanitize\_filename(filename):

return "".join(c if c.isalnum() or c in (" ", "-", "\_") else "" for c in filename)[:FILENAME\_MAX\_LENGTH]

def load\_downloaded\_links():

if os.path.exists(MASTER\_FILE):

with open(MASTER\_FILE, "r", encoding="utf-8") as f:

return set(line.strip() for line in f)

return set()

def save\_downloaded\_link(url):

with open(MASTER\_FILE, "a", encoding="utf-8") as f:

f.write(url + "\n")

print(f"Recorded link to {MASTER\_FILE}: {url}")

def extract\_video\_id(url):

if "youtube.com/watch?v=" in url:

return url.split("v=")[1].split("&")[0]

elif "youtu.be/" in url:

return url.split("youtu.be/")[1].split("?")[0]

else:

return None

def download\_youtube\_audio(url, output\_folder, title, downloaded\_links):

if url in downloaded\_links:

print(f"Skipping (already recorded): {url}")

return

video\_id = extract\_video\_id(url)

if video\_id is None:

print(f"Skipping invalid YouTube URL: {url}")

return

if not title or title.strip() == "":

title = "download\_" + hashlib.md5(url.encode()).hexdigest()

video\_folder = os.path.join(output\_folder, video\_id)

os.makedirs(video\_folder, exist\_ok=True)

output\_filepath = os.path.join(video\_folder, f"{video\_id}.wav") # Rename as folder name

if os.path.exists(output\_filepath):

print(f"Skipping (file exists): {output\_filepath}")

save\_downloaded\_link(url)

return

command = [

"yt-dlp",

"--no-check-certificate",

"--extract-audio",

"--audio-format", "wav",

"-o", os.path.join(video\_folder, f"{video\_id}.%(ext)s"), # Output named as folder (video ID)

"--cookies", COOKIES\_FILE,

url,

"--ppa", "ffmpeg:-ac 1 -ar 16000"

]

if FFMPEG\_LOCATION:

command.insert(1, "--ffmpeg-location")

command.insert(2, FFMPEG\_LOCATION)

print(f"Downloading audio: {title} from {url}")

try:

result = subprocess.run(command, check=True, text=True, capture\_output=True)

if "ERROR: Unable to open cookies file" in result.stderr:

print("Cookies file error detected. Terminating script.")

exit(1)

print(f"Successfully downloaded: {title}")

save\_downloaded\_link(url)

except subprocess.CalledProcessError as e:

print(f"Error downloading {title}: {e.stderr}")

def process\_csv\_file(csv\_path, output\_parent\_folder, speaker\_id, downloaded\_links):

encoding = detect\_encoding(csv\_path)

#print(f"Processing {csv\_path} | Detected Encoding: {encoding}")

output\_folder = os.path.join(output\_parent\_folder, speaker\_id)

os.makedirs(output\_folder, exist\_ok=True)

try:

with open(csv\_path, "r", newline="", encoding=encoding, errors="replace") as f:

reader = csv.reader(f)

header = next(reader, None)

#print("debug1")

if not header:

print(f"Skipping empty CSV file: {csv\_path}")

return

header\_lower = [h.strip().lower() for h in header]

try:

url\_index = header\_lower.index("url")

except ValueError:

print(f"Skipping {csv\_path}: Missing 'url' column")

return

title\_index = header\_lower.index("title") if "title" in header\_lower else -1

# Expanded list and flexible matching for Manual\_Verif

possible\_verify\_columns = ["cross-verify", "manual verif", "isspeaking", "cross\_verify", "manual\_verif", "Manual\_Verif", "Manual Verif", "manualverif", "ManualVerif"]

cross\_verify\_index = None

for col in header\_lower:

if "manual\_verif" in col or "manualverif" in col:

cross\_verify\_index = header\_lower.index(col)

print(f"Verification column found: {col}") # Debug: Confirm detection

break

if cross\_verify\_index is None:

for col in possible\_verify\_columns:

if col in header\_lower:

cross\_verify\_index = header\_lower.index(col)

print(f"Verification column found: {col}") # Debug: Confirm detection

break

if cross\_verify\_index is None:

print(f"Skipping {csv\_path}: Missing verification column. Headers found: {header\_lower}")

return

#possible\_verify\_columns = ["cross-verify", "manual verif","isspeaking", "cross\_verify", "Manual\_Verif", "Manual Verif"]

#cross\_verify\_index = None

#for col in possible\_verify\_columns:

#if col in header\_lower:

#cross\_verify\_index = header\_lower.index(col)

#break

#if cross\_verify\_index is None:

# print(f"Skipping {csv\_path}: Missing verification column")

# return

for row in reader:

needed\_index = max(url\_index, cross\_verify\_index)

if len(row) <= needed\_index:

print(f"Skipping malformed row in {csv\_path}: {row}")

continue

cross\_value = row[cross\_verify\_index].strip().lower()

if cross\_value == "yes" or cross\_value == "y" or cross\_value == "Y" or cross\_value == "Yes":

url = row[url\_index].strip()

if not url:

continue

title = row[title\_index].strip() if title\_index >= 0 and len(row) > title\_index else ""

download\_youtube\_audio(url, output\_folder, title, downloaded\_links)

except Exception as e:

print(f"Error reading {csv\_path}: {e}")

def process\_csv\_directory(csv\_directory, start\_id=10001):

output\_parent\_folder = os.path.join(csv\_directory + "\_cloning", "wav")

os.makedirs(output\_parent\_folder, exist\_ok=True)

downloaded\_links = load\_downloaded\_links()

csv\_files = sorted([f for f in os.listdir(csv\_directory) if f.lower().endswith(".csv")])

speaker\_map\_path = os.path.join(output\_parent\_folder, "speaker\_mapping.txt")

with open(speaker\_map\_path, "a+", encoding="utf-8") as mapping\_file:

for idx, filename in enumerate(csv\_files):

csv\_path = os.path.join(csv\_directory, filename)

speaker\_id = f"id{start\_id + idx:05d}"

speaker\_name = os.path.splitext(filename)[0] # Get filename without extension

mapping\_file.write(f"{speaker\_id}\t{speaker\_name}\n") # Write speaker mapping

process\_csv\_file(csv\_path, output\_parent\_folder, speaker\_id, downloaded\_links)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

csv\_directory = r"TOLLYWOOD ACTORS" # Change this to your actual folder path

starting\_id = 11486 # Change this to your desired starting ID

process\_csv\_directory(csv\_directory, start\_id=starting\_id)

**Code Explanation:**

1. **Dependencies and Setup:**
   * Utilizes os, csv, subprocess, and hashlib from Python's standard library, along with third-party libraries chardet (for encoding detection) and yt-dlp (for YouTube audio downloading).
   * Requires ffmpeg for audio processing, with its path optionally specified in FFMPEG\_LOCATION.
   * Uses a cookies file (cookies.txt) for authenticated downloads and a tracking file (downloaded\_links.txt) to record processed URLs.
2. **Core Functions:**
   * **detect\_encoding(file\_path):** Detects the encoding of a CSV file using chardet by analyzing a sample of its content, defaulting to UTF-8 if detection fails.
   * **sanitize\_filename(filename):** Cleans a filename by retaining only alphanumeric characters, spaces, hyphens, and underscores, and limits length to 100 characters.
   * **load\_downloaded\_links() and save\_downloaded\_link(url):** Manages a set of previously downloaded URLs stored in downloaded\_links.txt to prevent redundant downloads.
   * **extract\_video\_id(url):** Parses YouTube URLs (e.g., youtube.com/watch?v= or youtu.be/) to extract the unique video ID.
   * **download\_youtube\_audio(url, output\_folder, title, downloaded\_links):**
     + Skips already downloaded or invalid URLs.
     + Extracts the video ID and creates a dedicated folder for each video.
     + Uses yt-dlp to download audio in WAV format (mono, 16kHz sample rate) via the command:  
       yt-dlp --no-check-certificate --extract-audio --audio-format wav -o <output\_path> --cookies cookies.txt --ppa "ffmpeg:-ac 1 -ar 16000" <url>
     + Saves the downloaded file as <video\_id>.wav in a folder named after the video ID.
     + Logs the URL to downloaded\_links.txt upon success.
   * **process\_csv\_file(csv\_path, output\_parent\_folder, speaker\_id, downloaded\_links):**
     + Detects CSV file encoding and reads it using the csv module.
     + Checks for required columns: url and a verification column (e.g., manual\_verif, cross-verify, etc.).
     + Processes rows where the verification column contains "yes", "y", "Y", or "Yes".
     + Calls download\_youtube\_audio for each valid URL, storing output in a speaker-specific folder.
   * **process\_csv\_directory(csv\_directory, start\_id):**
     + Creates an output directory structure: <csv\_directory>\_cloning/wav.
     + Iterates through all CSV files in the specified directory.
     + Assigns each CSV a unique speaker ID (e.g., id10001, id10002, etc.) starting from the provided start\_id.
     + Logs speaker IDs and corresponding CSV filenames (without extension) to speaker\_mapping.txt.
     + Processes each CSV file to download audio.
3. **Execution:**
   * The script runs via the if \_\_name\_\_ == "\_\_main\_\_": block.
   * **INPUT:** Takes a CSV directory path (e.g., TOLLYWOOD ACTORS) and a starting speaker ID (e.g., 11486).
   * Calls process\_csv\_directory to process all CSV files in the directory.

**Output:**

* Audio files are saved as <video\_id>.wav in subfolders named by video ID within <csv\_directory>\_cloning/wav/<speaker\_id>.
* A speaker\_mapping.txt file maps speaker IDs to CSV filenames for reference.
* Downloaded URLs are tracked in downloaded\_links.txt to avoid reprocessing.

**Error Handling:**

* Skips empty or malformed CSV files, missing columns, or invalid URLs.
* Handles download errors from yt-dlp and exits if the cookies file is inaccessible.
* Uses encoding detection to handle varied CSV file formats.

**Usage Example:**

* Directory: TOLLYWOOD ACTORS
* Starting ID: 11486
* CSV files in the directory are processed, audio is downloaded, and organized by speaker ID (e.g., id11486, id11487, etc.).

**Dependencies:**

* Python 3, yt-dlp, chardet, and ffmpeg must be installed.
* A cookies.txt file is required for authenticated downloads.

**Notes:**

* Ensure ffmpeg is in the system PATH or specify its location in FFMPEG\_LOCATION.
* The script assumes CSV files contain at least a url column and a verification column (e.g., manual\_verif) to filter valid entries.

**Step-10. Do Final Data Analysis of downloaded CSVs:**

Directanalysis.py:

import os

import csv

from datetime import timedelta

from pydub.utils import mediainfo

from pydub import AudioSegment

def get\_audio\_duration(filepath):

try:

info = mediainfo(filepath)

duration = float(info['duration'])

return duration

except:

return 0

def format\_seconds\_to\_hhmmss(seconds):

return str(timedelta(seconds=int(seconds)))

def analyze\_cloning\_with\_local\_audio(input\_csv\_dir, cloning\_wav\_dir, output\_csv\_path="cloning\_analysis\_report.csv", start\_id=10499):

report\_data = []

max\_wav\_columns = 0

input\_csv\_files = sorted([f for f in os.listdir(input\_csv\_dir) if f.lower().endswith(".csv")])

folder\_names = sorted([

f for f in os.listdir(cloning\_wav\_dir)

if os.path.isdir(os.path.join(cloning\_wav\_dir, f))

])

id\_counter = start\_id

for idx, folder\_name in enumerate(folder\_names):

id\_folder\_path = os.path.join(cloning\_wav\_dir, folder\_name)

subfolders = [

subfolder\_name for subfolder\_name in sorted(os.listdir(id\_folder\_path))

if os.path.isdir(os.path.join(id\_folder\_path, subfolder\_name))

]

wav\_count = len(subfolders)

max\_wav\_columns = max(max\_wav\_columns, wav\_count)

total\_duration = 0

for subfolder in subfolders:

audio\_path = os.path.join(id\_folder\_path, subfolder, f"{subfolder}.wav")

if os.path.isfile(audio\_path):

total\_duration += get\_audio\_duration(audio\_path)

else:

print(f"⚠️ File not found: {audio\_path}")

formatted\_duration = format\_seconds\_to\_hhmmss(total\_duration)

unique\_id = f"id{id\_counter:05d}"

id\_counter += 1

csv\_filename = os.path.splitext(input\_csv\_files[idx])[0] if idx < len(input\_csv\_files) else "Unknown"

row = {

"CSV Filename (Folder Name)": csv\_filename,

"Unique ID": unique\_id,

"WAV Files Count": wav\_count,

"Duration": formatted\_duration,

}

for i, subfolder in enumerate(subfolders):

row[f"{i+1}.wav"] = subfolder

report\_data.append(row)

fieldnames = ["CSV Filename (Folder Name)", "Unique ID", "WAV Files Count", "Duration"] + [

f"{i}.wav" for i in range(1, max\_wav\_columns + 1)

]

with open(output\_csv\_path, "w", newline="", encoding="utf-8") as f:

writer = csv.DictWriter(f, fieldnames=fieldnames)

writer.writeheader()

writer.writerows(report\_data)

print(f"✅ Report generated: {output\_csv\_path}")

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

input\_csv\_dir = "/nlsasfs/home/acoustic-cdac/hruturaj/voice\_cloning/INDIAN ACTRESS"

cloning\_dir = "/nlsasfs/home/acoustic-cdac/hruturaj/voice\_cloning/INDIAN ACTRESS\_cloning/wav"

output\_csv\_path="/nlsasfs/home/acoustic-cdac/hruturaj/voice\_cloning/INDIAN ACTRESS\_cloning/cloning\_analysis\_report.csv"

analyze\_cloning\_with\_local\_audio(input\_csv\_dir, cloning\_dir, start\_id=10499)

**Code Explanation:**

1. **Import Libraries**
   * os, csv, timedelta: Handle file operations and formatting.
   * pydub.utils.mediainfo: Extract metadata including duration of audio.
   * pydub.AudioSegment: Audio handling (used indirectly via mediainfo).
2. **Core Functions**

**get\_audio\_duration(filepath)**

Returns the duration (in seconds) of a WAV file using mediainfo.

**format\_seconds\_to\_hhmmss(seconds)**

Formats total seconds into HH:MM:SS string format for readability in reports.

1. **Main Function: analyze\_cloning\_with\_local\_audio()**

**Parameters:**

* input\_csv\_dir: Directory of original CSV files (used for matching).  
  Eg.- "/nlsasfs/home/acoustic-cdac/hruturaj/voice\_cloning/INDIAN ACTRESS"
* cloning\_wav\_dir: Folder with cloned audio (organized in subfolders).

Eg.- "/nlsasfs/home/acoustic-cdac/hruturaj/voice\_cloning/INDIAN ACTRESS\_cloning/wav"

* output\_csv\_path: Output report file path.

Eg.- "/nlsasfs/home/acoustic-cdac/hruturaj/voice\_cloning/INDIAN ACTRESS\_cloning/cloning\_analysis\_report.csv"

* start\_id: Initial ID number to tag each data entry.  
  Eg.- ID10499

**Logic:**

1. **Read CSV Files & Cloned Audio Folders**:
   * Scans both CSV directory and WAV cloning directory to ensure parallel analysis.
2. **Iterate Over Each Speaker Folder**:
   * Each folder in cloning\_wav\_dir is expected to represent one speaker or ID.
   * Subfolders within represent individual cloned audio samples.
3. **Process Each WAV File**:
   * Calculates duration for each WAV.
   * Accumulates total duration for that speaker.
   * Logs missing files for debugging.
4. **Create a Report Entry**:
   * Includes folder name, generated unique ID, count of WAV files, total duration, and filenames of individual WAVs.
5. **Export to CSV**:
   * Dynamically adjusts the number of columns depending on the max number of WAV files found in any speaker folder.
   * Outputs a single consolidated report.

