END TERM REPORT

ANUSHKA TYAGI
24112023

SECTION I: ABSTRACT

The goal of my project is to build a **Subtitle Generator** that creates subtitles for videos, starting from scratch. The process begins with taking a YouTube video URL as input, downloading the video, extracting its audio, converting the audio into text, and then syncing the subtitles with the video. The main steps include audio extraction, converting audio into spectrograms, using a deep learning model for speech-to-text conversion, and finally adding the subtitles to the video.The goal is to design an end-to-end system capable of transcribing spoken words in videos into accurate subtitles, addressing the challenge of aligning audio and text data effectively.

The process begins with downloading the video using **YT-DLP**, a reliable tool for handling YouTube URLs. The downloaded video is processed with **FFmpeg**, which extracts the audio stream in formats like MP3. The extracted audio is then analyzed using **Librosa**, a Python library that generates **spectrograms**—graphical representations of audio frequencies over time. These spectrograms are essential for converting audio into a machine-readable format, enabling the system to process the audio data effectively.Next, the spectrograms are passed through a hybrid **CNN-LSTM model** to convert the audio into text. The **Convolutional Neural Network (CNN)** captures visual features from the spectrograms, while the **Long Short-Term Memory (LSTM)** network processes the sequential patterns of the audio data. The model is trained using the **LibriSpeech dataset**, which provides spectrograms and their corresponding text labels.The text labels of dataset are handled using one hot encoding to convert categorical character labels into a numerical format, allowing the model to predict characters independently. It simplifies multi-class classification by representing each character as a distinct vector, making it easier for the CNN-LSTM model to process the data.

This setup enables the model to learn and predict character sequences accurately. The entire neural network is built and trained using **Keras** and **TensorFlow**, which provide tools for deep learning and model optimization.Finally, the predicted subtitles are synchronized with the video and added using **FFmpeg** to generate a final subtitled

video.The model's performance is evaluated on metrics such as accuracy and **Word Error Rate (WER).**

SECTION II: METHODOLOGY

1)Problem Statement

Subtitle generators are systems that automatically convert spoken language from audio or video content into written text, producing subtitles that appear on-screen. These systems use speech recognition technology to analyze the audio, identify the words being spoken, and then transcribe them into a readable format.The background of subtitle generators traces back to the increasing demand for making multimedia content accessible to people who are deaf or hard of hearing, as well as to non-native speakers or people in noisy environments who rely on subtitles to better understand spoken content. With the growth of digital content platforms, such as YouTube, streaming services, and educational videos, subtitles have become essential for a broader audience.Subtitles were produced from the necessity of people to communicate further and deliver the art of cinema to other countries. It eventually expanded to television and has developed into incredible technology that enables us to utilize content despite its created language initially.Early systems for subtitling were manually created, which was time-consuming and costly. .Advancements in artificial intelligence (AI) have revolutionized the process of subtitle creation, making it more efficient and accurate. AI-driven subtitle generation involves automatic speech recognition (ASR), natural language processing (NLP), and machine learning algorithms to transcribe and synchronize spoken words with video content. These technologies can detect nuances in speech, accommodate various accents and dialects, and adapt to context for higher accuracy.

2)Data Collection and Preprocessing

- **Data Sources**

  LibriSpeech is a corpus of approximately 1000 hours of English speech, prepared by Vassil Panayotov with the assistance of Daniel Povey. The data is derived from read audiobooks from the LibriVox project, and has been carefully segmented and aligned.It was designed to aid the training and testing of Automatic Speech Recognition (ASR) systems.It is organized into three main folders: `dev`, `test`, and `train`, representing development, testing, and training

sets. Within each folder, there are subfolders for each speaker, containing audio files stored in smaller subfolders (chapters). Alongside the audio files (in FLAC format), there are `*.trans.txt` files containing the corresponding text transcripts of the spoken utterances. This structure allows easy access to both the audio and the transcripts for training and testing speech recognition models.

- **Preprocessing Steps**

  I have preprocessed the **LibriSpeech `train-clean-100`** for training.

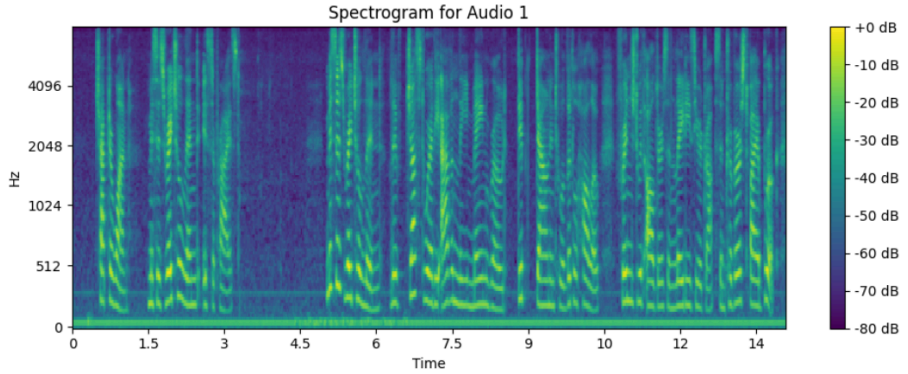
  - It starts by defining the base path to the dataset and specifying the path for the **`train-clean-100`** subset, which is a smaller, more manageable subset of LibriSpeech.
  - It then collects the paths of all **FLAC** audio files by recursively searching the folder structure, which is organized by speaker, chapter, and utterance, with each file identified by a unique file ID.
  - The **`find_sentence()`** function retrieves the transcript for each audio file by splitting the file ID into **`speaker_id`**, **`chapter_id`**, and **`utterance_id`**, and searching the corresponding `.trans.txt` file for the matching transcript.
  - A DataFrame is created to store the audio file metadata, including **`file_id`, `speaker_id`, `chapter_id`, `utterance_id`,** the full `audio_path`, and the corresponding transcription.

	file_id	speaker_id	chapter_id	utterance_id	audio_path	sentence
0	103-1240-0000	103	1240	0000	/kaggle/input/librispeech-clean/LibriSpeech/tr...	CHAPTER ONE MISSUS RACHEL LYNDE IS SURPRISED M...
1	103-1240-0001	103	1240	0001	/kaggle/input/librispeech-clean/LibriSpeech/tr...	THAT HAD ITS SOURCE AWAY BACK IN THE WOODS OF ...
2	103-1240-0002	103	1240	0002	/kaggle/input/librispeech-clean/LibriSpeech/tr...	FOR NOT EVEN A BROOK COULD RUN PAST MISSUS RAC...
3	103-1240-0003	103	1240	0003	/kaggle/input/librispeech-clean/LibriSpeech/tr...	AND THAT IF SHE NOTICED ANYTHING ODD OR OUT OF...
4	103-1240-0004	103	1240	0004	/kaggle/input/librispeech-clean/LibriSpeech/tr...	BUT MISSUS RACHEL LYNDE WAS ONE OF THOSE CAPAB...

Using the **`train-clean-100`** subset allows for a smaller, cleaner dataset, making it easier to manage while ensuring high-quality, labeled audio-text pairs for model training.

3) Algorithms and Techniques

- **Audio Preprocessing**
  - ➢ *Spectrogram Generation*: The audio data from the LibriSpeech dataset (FLAC files) is converted into spectrograms using **Librosa**.A **spectrogram** is a visual representation of sound that shows how the frequencies of a signal change over time. Its color gradient reflects how strong the signal is at specific points in time and frequency i.e. its amplitude or intensity.A spectrogram in Librosa is created by first applying the **Short-Time Fourier Transform (STFT)** to divide the audio into short, overlapping frames and extract frequency information.The frequencies are mapped to the Mel scale to align with human hearing, and the resulting values are converted to decibels for easier visualization and interpretation.



  - ➢ *Normalization*: The spectrograms are normalized to ensure consistent scaling across samples, which helps the model learn efficiently.By normalizing, all spectrograms are on the same scale, making it easier for the model to learn patterns and features efficiently.

- **One-Hot Encoding for Text Data**

One Hot Encoding is a method for converting categorical variables into a binary format. It creates new binary columns (0s and 1s) for each category in the original variable. Each category in the original column is represented as a separate column, where a value of 1 indicates the presence of that category, and 0 indicates its absence.The transcript data (text) is converted into one-hot encoded vectors at the character level. This method represents each character as a binary vector, making it easier for the model to predict characters sequentially.This encoding ensures that the text is in a format the model can use

for training and generating predictions.The matrix has dimensions **(length of text, number of unique characters)**, with each row corresponding to a character in the text and each column corresponding to a character.

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

- **Model Architecture**
  - ➢ *Convolutional Neural Networks (CNNs)*:

    CNN layers are used to process the spectrograms, capturing local patterns in the time-frequency domain (e.g., phonemes and sound features).These features help extract meaningful audio representations.

  - ➢ *Long Short-Term Memory Networks (LSTMs)*:

    LSTMs handle the sequential nature of audio and text data, learning **temporal relationships and context over time**.They predict each character in the transcript sequentially, maintaining the context of prior predictions.**Temporal relationships** focus on the sequential flow of data, while **context over time** ensures that the model retains and uses relevant information from earlier parts of the sequence to make better predictions.

    Bidirectional LSTM processes the input sequence in both forward and backward directions. This allows the model to understand context from both past and future states in the sequence.

  - ➢ *Combined CNN-LSTM Architecture*:

    The CNN extracts features from the spectrogram, which are then passed to the LSTM for sequential modeling of text data. This hybrid architecture ensures the model captures both spatial and temporal features of the data.

➢ *TimeDistributed Output Layer*

This layer outputs predictions for each time step in the input sequence.Using **softmax** as activation function,The result is a sequence of probability distributions, one for each time step, where each distribution represents the likelihood of each character at that step.

- **WHY THESE TECHNIQUES WERE USED OVER OTHER ALTERNATIVES**

➔ **Spectrograms**

Spectrograms transform audio signals into a time-frequency representation, making patterns in speech more visible and easier to process.Speech features like pitch, phonemes, and rhythm are more evident in the spectrogram than in **raw waveforms**.Raw waveforms are high-dimensional and harder for models to interpret.They require the model to learn frequency and time-based features from scratch, which increases complexity and training time.Spectrograms pre-process audio into a compact and meaningful representation, reducing the model's workload.

➔ **One Hot Encoding**

I chose to use one-hot encoding as it is a straightforward and simple approach that transforms each character into a unique vector, making it easy to implement and understand. Since I haven't delved into advanced techniques like **embeddings** yet, one-hot encoding provides a good foundation for my project without introducing unnecessary complexity. It allows me to focus on the core functionality of my subtitle generator before exploring more advanced methods.

Also **Label encoding** is not used because it maps each unique character or word to a distinct integer, which doesn't capture the underlying relationships between characters. In a subtitle generator, where predicting characters or words in sequence is crucial, label encoding would treat all characters as independent, ignoring the context and proximity of characters or words in the sequence. This would limit the model's ability to understand patterns in speech and generate accurate subtitles.

➔ **CNN**

CNNs are excellent for image processing and feature extraction so used here in time-frequency features of spectrograms. CNNs can detect important audio patterns (like phonemes) by convolving over small regions, making them more efficient .CNNs are used instead of **Fully connected layers** as they would treat the entire spectrogram as a single vector, losing local structure and increasing the number of parameters, which can lead to overfitting and poor performance.

➔ **LSTM**

LSTMs can handle the temporal relationships in audio and are well-suited for capturing long-term dependencies in sequential data like speech.

GRUs (Gated Recurrent Units) are simpler than LSTMs because they have fewer gates and lack separate memory cells, which makes them computationally less intensive.However, GRUs might struggle with very long sequences because they lack the more complex memory mechanism of LSTMs, which can help capture long-term dependencies in data.

Transformers, like **GPT** or **BERT**, are **high-level models**, requiring large amounts of data and substantial computational power to train effectively. LSTMs are simpler to implement and require less training time and resources, making them better suited for my project.

4)Model Development

- **Model Architecture**
  - ➢ The model takes in spectrograms with a specific shape,where the shape typically includes time steps and frequency bins.

```
# Define the model
inputs = Input(shape=spectrogram_shape, name="input_spectrogram")
```

  - ➢ **Layer1:**
    - ➔ **Conv2D**: A 2D convolutional layer with 64 filters, using a 3x3 kernel. The 'same' padding ensures the output size is the same as the input size.

- ➔ **LeakyReLU**: Activation function to introduce non-linearity into the model.It solves the "dying ReLU" problem by allowing a small, non-zero gradient for negative input values.
- ➔ **BatchNormalization**: Normalizes the output to improve convergence during training.
- ➔ **MaxPooling2D**: Downsample the data (reducing computational cost) and focus on dominant features.
- ➔ **Dropout**: Randomly drops 20% of neurons to prevent overfitting.

```
x = Conv2D(64, (3, 3), padding='same')(inputs)
x = LeakyReLU(alpha=0.1)(x)
x = BatchNormalization()(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Dropout(0.2)(x)
```

- ➢ The next three convolutional layers progressively increase the number of filters (128, 256, and 512). Each layer extracts higher-level features from the spectrograms.Multiple filters (e.g., 64, 128, 256, 512) are used to extract increasingly complex features at each layer.

```
x = Conv2D(128, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.1)(x)
x = BatchNormalization()(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Dropout(0.2)(x)

x = Conv2D(256, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.1)(x)
x = BatchNormalization()(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Dropout(0.3)(x)

x = Conv2D(512, (3, 3), padding='same')(x)
x = LeakyReLU(alpha=0.1)(x)
x = BatchNormalization()(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Dropout(0.3)(x)
```

After each convolution, we apply **LeakyReLU**, **BatchNormalization**, **MaxPooling2D**, and **Dropout** in a similar manner as in Layer 1.

- **Flatten**: Converts the 2D feature maps into a 1D vector so it can be passed to fully connected layers.
- **Dense**: The first dense layer has 1024 units with ReLU activation,it enables the model to learn abstract, high-dimensional representations of the data The second dense layer reshapes the output to match the desired sequence length (output_sequence_length) and feature size (512 units).
- **Reshape**: Reshapes the output into a 3D tensor with the shape (output_sequence_length, 512) to make it compatible with the following LSTM layers.

```
x = Flatten()(x)
x = Dense(1024, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(output_sequence_length * 512, activation='relu')(x)
x = Reshape((output_sequence_length, 512))(x)
```

- **Bidirectional LSTM**: These layers process the sequence in both directions (forward and backward).The first Bidirectional LSTM layer has 512 units, and the second one has 256 units.Randomly 30% of neurons are dropped to prevent overfitting.

```
x = Bidirectional(LSTM(512, return_sequences=True))(x)
x = Dropout(0.3)(x)
x = Bidirectional(LSTM(256, return_sequences=True))(x)
x = Dropout(0.3)(x)
```

- **TimeDistributed**: This allows applying the dense layer to each time step of the output sequence. It enables the model to make predictions for each time step independently, which is important for tasks like subtitle generation.
- **Dense Layer**: The final dense layer has as many units as the number of possible characters (num_characters), with a **softmax activation** to output probabilities for each character at each time step.

```
# Output layer
outputs = TimeDistributed(Dense(num_characters, activation='softmax'))(x)
```

- **Optimization**
  - *Loss Function*: A **categorical cross-entropy** loss function is used to optimize the character-level predictions, comparing the predicted and true characters at each timestep.It's suitable for multi-class classification problems like here where each time step outputs a probability distribution.
  - *Optimization Algorithm*: Optimizers like **Adam** are used for efficient gradient updates during training, ensuring faster convergence.It combines the benefits of accelerated convergence and adaptive learning rates as it automatically adjusts the learning rate for each parameter
  - *Sequence-to-Sequence Learning*: The model is trained to map the input spectrogram to the corresponding character sequence (transcript).
- **Callbacks**

A callback is an object that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc).

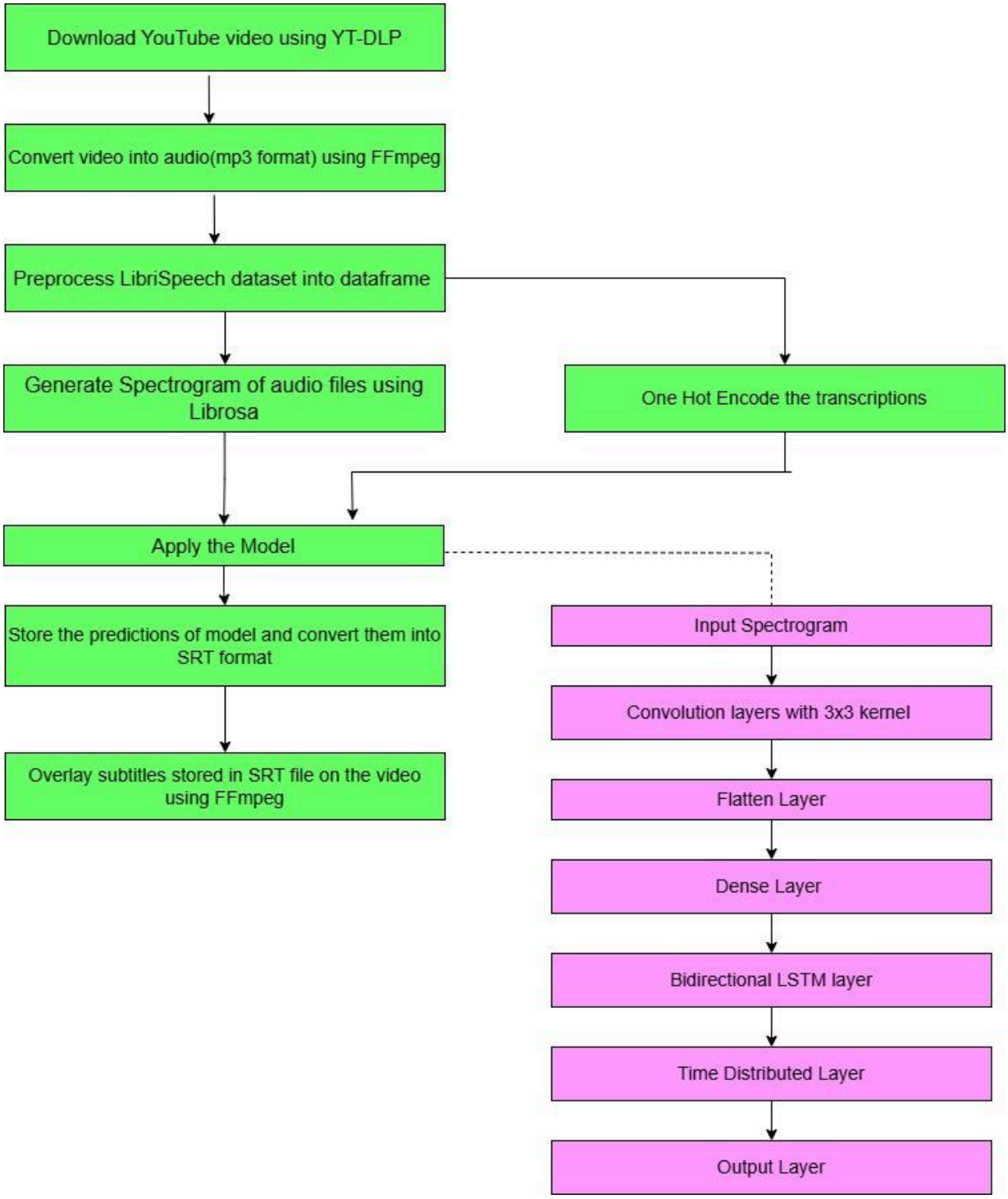
- ➢ **ReduceLROnPlateau:**Reduces the learning rate when the validation loss stops improving, allowing the model to converge more effectively during later stages of training.
- ➢ **EarlyStopping:**Stops training early if the model stops improving (based on validation loss), preventing overfitting and saving computational resources.

```
# Add callbacks for better training
callbacks = [
    ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=3,
        min_lr=1e-6,
        verbose=1
    ),
    EarlyStopping(
        monitor='val_loss',
        patience=10,
        restore_best_weights=True,
        verbose=1
    )
```

- **Training Parameters**
  - ➢ **Batch Size**:A smaller batch size (16) balances memory constraints and model performance.
  - ➢ **Epochs**:20 epochs allow the model to refine its understanding of the data over multiple passes.

- **Evaluation Metrics**
  - ➢ *Word Error Rate (WER)*: Evaluates the accuracy of entire words in the predicted transcript, providing a higher-level assessment of the subtitle quality.Accuracy is calculated with the **ground truth text** i.e. the actual transcriptions which were present in the dataset.

- **Implementation Workflow**



SECTION III: RESULTS

Despite extensive efforts and experimentation, my model's performance resulted in a Word Error Rate (WER) of 0.98, which indicates that the predictions were not accurate. I tried various approaches, including experimenting with different architectures, tuning the number of epochs, and incorporating additional data into the training process. However, these adjustments did not lead to a significant improvement in the model's accuracy and gave incorrect predictions.

SECTION IV: FINAL DELIVERABLES

The primary output of my project is a subtitle generation model designed to predict subtitles from audio inputs. The model aims to achieve low Word Error Rate (WER) by accurately transcribing speech into text. While I successfully built and trained the model, achieving accurate predictions with low WER remains a challenge, as the model currently produces inaccurate results with a WER of 0.98. The core objective of the project—to develop a system capable of generating accurate subtitles from audio—was partially met.

One of the planned features was the ability to convert a URL containing audio into a subtitle track and overlay these predicted subtitles onto the audio using FFmpeg. However, this functionality could not be achieved due to the model's prediction inaccuracies. Despite these setbacks, the project still provides a foundational subtitle generation system that can be further refined. Future improvements will focus on reducing WER, handling noisy backgrounds, and improving real-time subtitle overlay capabilities.

SECTION V: CHALLENGES FACED

- **Limited GPU resources:** Insufficient access to high-performance GPUs slowed down training and hindered experimentation.
- **Dataset limitations:** Due to low training dataset size, model could not be trained properly.
- **Model architecture selection:** Various CNN-LSTM configurations didn't yield desired performance, and tuning them proved difficult.
- **Word Error Rate (WER):** Achieving low WER while maintaining overall model performance was difficult.

SECTION VI: SECONDARY GOALS OR FUTURE SCOPE

1. **Achieving Near-Zero Word Error Rate (WER):** One of the primary future goals for the subtitle generator is to significantly reduce the Word Error Rate (WER) to near-zero levels. This can be accomplished by refining the model's architecture, leveraging advanced techniques such as hybrid models (e.g., Transformer-LSTM combinations) and also using better encoding decoding techniques such as embeddings for transcriptions.

2. **Enhancing Robustness Against Noisy Backgrounds:** A key challenge in real-world applications is handling audio with noisy or unpredictable backgrounds. To address this, future iterations of the model will be required such as training the model on noisy datasets.

3. **Incorporating Multilingual Translation Capabilities:** To expand the accessibility and utility of the subtitle generator, a future goal is to integrate multilingual translation functionality.

At the current stage, achieving these ambitious goals was not feasible, primarily due to my beginner-level expertise in the field and the limited time available for this project.As I continue to enhance my skills and dedicate more time to development, these objectives will become more attainable in future.

SECTION VII: REFERENCES

- **https://subtitlebee.com/blog/history-of-subtitles/#:~:text=These%20subtitles%20were%20produced%20from,despite%20its%20created%20language%20initially**
- **https://www.amberscript.com/en/blog/ai-for-subtitles-for-videos-guide/**
- **https://www.openslr.org/12**
- **https://khareanu1612.medium.com/audio-signal-processing-with-spectrograms-and-librosa-b66a0a6bc5cc**
- **https://www.geeksforgeeks.org/ml-one-hot-encoding/**
- **https://www.geeksforgeeks.org/sparse-matrix-in-machine-learning/**
- **https://medium.com/@prudhviraju.srivatsavaya/lstm-vs-gru-c1209b8ecb5a**
- **https://www.geeksforgeeks.org/regularization-in-machine-learning/**
- https://youtube.com/playlist?list=PLKnIA16_RmvYuZauWaPlRTC54KxSNLtNn&si=e2t_1lRMQ6aE16yk
- https://youtu.be/ZLlPkmmDJAc?si=wqpwAiOi7YF_51mP
- https://www.kaggle.com/code/heiswicked/libri-01
- https://github.com/nicknochnack/DeepAudioClassification/blob/main/AudioClassification.ipynb
- https://github.com/nicknochnack/DeepAudioClassification/blob/main/AudioClassification.ipynb
- **https://www.youtube.com/watch?v=dJYGatp4SvA**
- https://www.sciencedirect.com/science/article/pii/S1877050924006227?ref=pdf_download&fr=RR-2&rr=8f9cd116fad98e86

SECTION VII: APPENDIX

I also made my project using all pre trained models which worked completely well. The final result was able to make a well synchronized subtitled video. Following was my approach and the pre-trained models I used.

1. Downloading and Preprocessing Video and Audio

- It begins with installing the necessary tool **yt-dlp** to download video from YouTube.
- Function **download_and_play_youtube_audio** uses yt-dlp to extract and save video using its URL as MP3 for further processing.
- These preprocessing steps ensure the video/audio data is available for transcription.

```python
def download_and_play_youtube_audio(url, output_path="output.mp3"):
    try:
        # Use yt-dlp to download the audio
        subprocess.run([
            'yt-dlp',
            '-x', '--audio-format', 'mp3',
            '-o', 'temp_audio.%(ext)s',
            url
        ], check=True)

        temp_file = "temp_audio.mp3"

        # Convert to the desired output path if needed
        if output_path != temp_file:
            os.rename(temp_file, output_path)

        # Load the audio file using librosa
        audio_data, sr = librosa.load(output_path, sr=None)

        # Display audio player in Kaggle notebook
        print(f"Playing audio from: {output_path}")
        display(Audio(audio_data, rate=sr))

        return True

    except Exception as e:
        print(f"Error: {str(e)}")
```

2. Using Whisper for Transcription

- The pre-trained Whisper model from OpenAI is loaded using the library **whisper.** It is used to transcribe audio into text directly.
- The function **transcribe_audio_with_whisper** handles transcription and provides a structured output with timestamps and text segments.
- Whisper's advantage is its robustness and ability to handle various accents and noise levels, making it ideal for transcription tasks.

```python
def transcribe_audio_with_whisper(audio_path):
    try:
        # Load the Whisper model
        model = whisper.load_model("base")

        # Transcribe the audio directly using Whisper
        result = model.transcribe(audio_path)

        # Return the transcribed text instead of printing directly
        return result['text']

    except Exception as e:
        print(f"Error: {str(e)}")
        return None
```

3. Generating Subtitles in SRT Format

- A custom function **create_srt_file** takes the transcription results and converts them into a standard **SRT (SubRip Subtitle)** format.
- The function iterates over transcription segments and formats them with sequence numbers, start and end times, and transcribed text.
- The SRT format is widely supported by media players for displaying subtitles.

```python
def create_srt_file(transcription_result, srt_path="subtitles.srt"):
    try:
        segments = transcription_result['segments']  # Whisper provides timestamps here
        with open(srt_path, "w", encoding="utf-8") as srt_file:
            for idx, segment in enumerate(segments):
                start_time = segment['start']
                end_time = segment['end']
                text = segment['text']

                # Format time as hh:mm:ss,ms
                def format_time(seconds):
                    hours = int(seconds // 3600)
                    minutes = int((seconds % 3600) // 60)
                    seconds = int(seconds % 60)
                    milliseconds = int((seconds - int(seconds)) * 1000)
                    return f"{hours:02}:{minutes:02}:{seconds:02},{milliseconds:03}"

                srt_file.write(f"{idx + 1}\n")
                srt_file.write(f"{format_time(start_time)} --> {format_time(end_time)}\n")
                srt_file.write(f"{text.strip()}\n\n")

        print(f"Subtitle file saved to: {srt_path}")
        return srt_path

    except Exception as e:
        print(f"Error creating SRT file: {str(e)}")
        return None
```

4. Overlaying Subtitles onto the Video

- **FFmpeg i**s used for combining subtitles with video.
- A function **overlay_subtitles** applies the SRT file to the original video and creates an output video with embedded subtitles.

```python
def overlay_subtitles(video_path, srt_path, output_path="output_video.mp4"):
    try:
        # Use FFmpeg to overlay subtitles
        subprocess.run([
            'ffmpeg',
            '-i', video_path,
            '-vf', f"subtitles={srt_path}",
            '-c:a', 'copy',
            output_path
        ], check=True)
        print(f"Subtitled video saved as: {output_path}")
        return output_path
    except Exception as e:
        print(f"Error adding subtitles: {str(e)}")
        return None
```

5. Displaying the Results

- Finally, the generated video with embedded subtitles is displayed using **IPython.display.Video**, making it accessible directly in the notebook environment**.**

```python
from IPython.display import Video

# Path to your video file
video_path = "output_video.mp4"

# Display the video
Video(video_path, embed=True, width=640, height=360)
```

THANK YOU