EE698R Project Report

Speaker Diarization Pipeline with AssemblyAI and Pyannote

Introduction

Speaker diarization is the process of partitioning an audio stream into homogeneous segments according to speaker identity. It helps in applications like meeting transcription, customer service call analysis, and multimedia indexing. This report presents a complete pipeline for performing speaker diarization on both real-time microphone inputs and pre-recorded audio files using AssemblyAI and Pyannote.

Real-world Applications:

- **Automatic meeting minutes generation**
  This application uses diarization to accurately transcribe who said what in a meeting, enabling automated note-taking and documentation. This helps organizations improve meeting efficiency and provides reference materials for attendees and absentees. In the corporate world, where detailed minutes are crucial for accountability, this capability not only reduces manual effort but also minimizes human errors. It further allows searchable archives of discussions, which is invaluable for knowledge management.

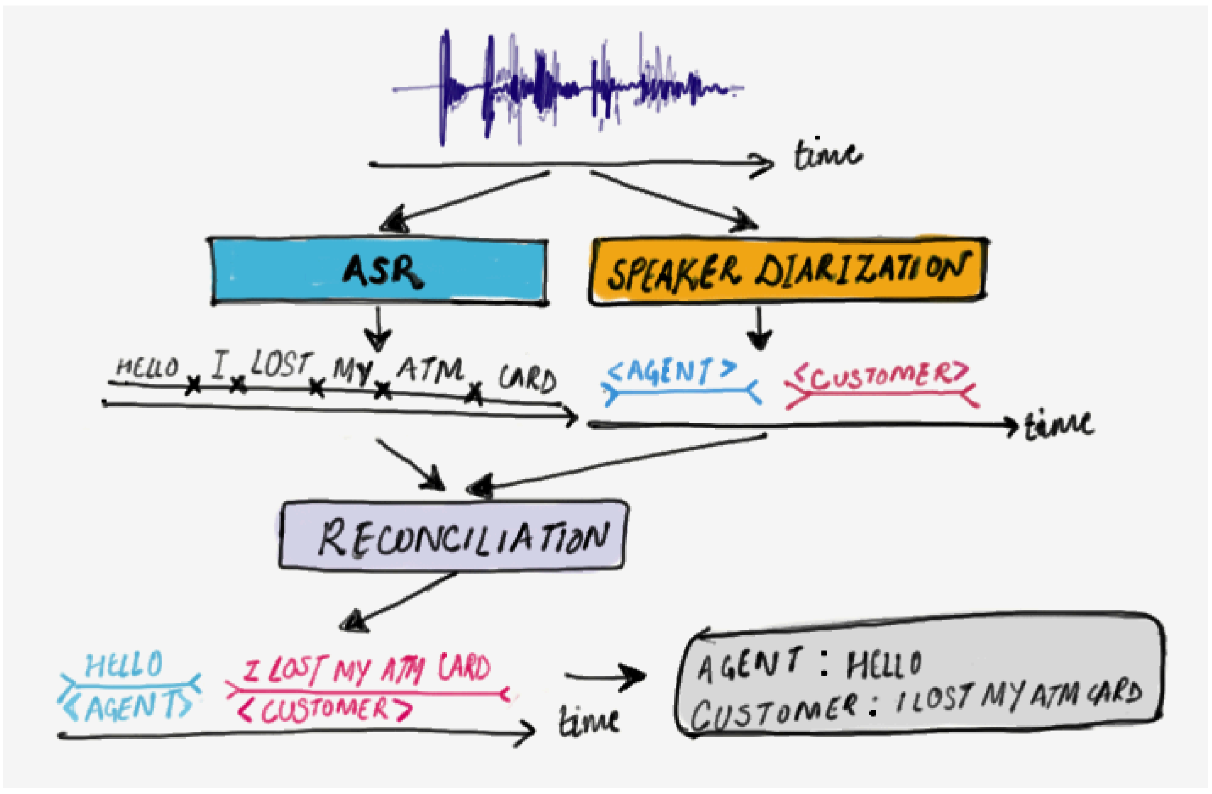
- **Analyzing podcast or interview participants**
  By segmenting speakers, it becomes easier to study individual speaking patterns, derive insights, or create speaker-specific summaries in interviews and podcast recordings. This facilitates better understanding of conversation flow and speaker engagement. Content creators and researchers can benefit by identifying dominant voices, frequent interrupters, or emotional tones from specific participants, thereby refining production or conducting qualitative analysis.

- **Monitoring customer service calls for compliance and analytics**
  Diarization can isolate agent and customer speech, enabling quality analysis, sentiment tracking, and adherence to service protocols, which is crucial for customer relationship management. Companies can use these insights to train staff better, enhance customer satisfaction, and even identify recurring issues or compliance risks through automated review of large volumes of calls.

- **Indexing and searching video/audio content**
  With speaker diarization, media files can be annotated with speaker turns, allowing better searchability and easier navigation across large datasets for video or audio content. Educational platforms, legal bodies, and content archives can tag and retrieve precise moments from lengthy materials. This level of detailed indexing

  revolutionizes content accessibility, offering faster and more intelligent data retrieval.

Overview of Speaker Diarization



Speaker diarization involves identifying “who spoke when” in an audio recording. The typical pipeline involves:

- **Speech-to-text transcription**
  This step converts spoken words in the audio into textual format, forming the base for further diarization processing. It is essential because raw audio is not easily interpretable by machines. A robust transcription process ensures that further tasks like sentiment analysis, keyword spotting, or topic detection are grounded in accurate data, making the entire pipeline more effective.

- **Voice activity detection (VAD)**
  VAD identifies segments in the audio where speech is present, filtering out silence and noise for focused analysis. This allows for efficient processing by reducing unnecessary computational load. VAD also improves precision by ensuring that only the relevant audio is passed to the next stages of diarization, especially important

when dealing with long recordings with intermittent speech.

- **Speaker embedding extraction**
  During this step, the system generates a vector representation of a speaker's voice, capturing unique vocal characteristics. These embeddings act like fingerprints for each speaker's voice and are crucial for distinguishing between different individuals. A well-designed embedding extractor can improve clustering accuracy, even in complex multi-speaker environments.

- **Clustering speaker embeddings**
  The embeddings are grouped based on similarity, ideally forming clusters corresponding to individual speakers. This unsupervised learning step is at the core of diarization, enabling separation of speakers without prior knowledge of their identities. Effective clustering ensures the integrity of the final speaker-attributed segments.

- **Generating time-aligned speaker-labeled transcriptions**
  The clustered segments are matched with timestamps to generate readable transcripts that identify speakers across the timeline. This final output is highly useful for end-users who wish to understand or search conversations based on speaker turns, particularly in multi-party discussions like interviews, panels, or customer support calls.

Illustration 1: Speaker Diarization Pipeline

Audio Input → VAD → Embeddings → Clustering → Speaker Labels

AssemblyAI: An Overview

AssemblyAI is a robust speech-to-text API that provides advanced features such as speaker labels, topic detection, sentiment analysis, summarization, and more. It enables easy integration of transcription services into custom pipelines using simple RESTful APIs or SDKs.

Features:

- **Accurate transcription with punctuation and formatting**
  AssemblyAI ensures high-quality transcripts with readable punctuation and proper sentence structures, suitable for official documentation. It not only helps users comprehend long audio files more efficiently but also supports automatic report generation in formats suitable for business, legal, or academic use.

- **Speaker labeling with high temporal precision**
  The tool can accurately distinguish and label multiple speakers within a conversation, enhancing the utility of the transcript. This feature is crucial for diarization, as it allows users to track dialogue dynamics and identify each participant, which is particularly beneficial in complex, multi-speaker audio.

- **Language detection across over 80 languages**
  This enables the system to automatically adapt to different languages without manual configuration. Such flexibility allows global applications and supports multilingual environments where language switching happens within the same audio.

- **Sentiment and topic analysis for higher-level understanding**
  These insights are crucial for applications like feedback evaluation, emotion detection, and trend analysis. Businesses can automatically gauge customer satisfaction or detect emotional distress, adding strategic value to their voice analytics pipelines.

- **Custom vocabulary support for domain-specific terms**
  Specialized vocabularies can be provided to improve accuracy for jargon or brand-specific language, especially useful in technical or business contexts. This enhances transcription quality where standard speech models may struggle, such as in healthcare, finance, or legal industries.

How It Works:

1. **Upload audio to AssemblyAI API**
   Audio files are sent to the API endpoint for processing via SDK or HTTP requests. The process is seamless and supports large-scale asynchronous uploads.

2. **Configure transcription parameters such as speaker labels=True, language model, etc.**
   These parameters control the behavior and features of the transcription. Configurations help tailor the process to specific needs like diarization, sentiment detection, or summarization.

3. **Audio is transcribed with optional features like speaker labeling**
   The system applies deep learning models to generate the transcript and metadata. These features enhance the raw transcript with contextual and structural information, boosting its interpretability.

4. **Transcription and metadata are returned in JSON format**
   Results include timestamps, speaker IDs, text, confidence scores, and optional sentiment or topic tags. This structured output makes it easy to integrate AssemblyAI into other analytics pipelines or visualization tools.

Diagram 1: AssemblyAI Workflow

Audio → AssemblyAI → JSON Transcript + Metadata

Pyannote: Pre-trained Speaker Embedding Model

In this pipeline, we used the pre-trained speaker embedding model from Pyannote, which provides a robust method for extracting speaker-specific embeddings from audio data. Pyannote's speaker embedding model is built upon deep learning techniques and fine-tuned to capture the unique vocal traits of different speakers in a conversation.

Pre-trained Model

The pre-trained model from Pyannote is based on state-of-the-art neural networks trained on large datasets of conversational speech. It generates speaker embeddings — numerical representations that uniquely describe the voice of a speaker. These embeddings are essential for clustering the audio segments into groups that correspond to different speakers.

How It Works:

- The model extracts embeddings from the pre-processed audio features (e.g., Mel spectrograms).
- These embeddings are then used to perform speaker clustering. In our case, these embeddings are passed to the clustering algorithm after being extracted.
- The embeddings help to distinguish speakers even in noisy environments by leveraging robust features learned during the model's training phase.
- The pre-trained model can be fine-tuned for specific domains or accents to further improve diarization accuracy.

AssemblyAI Integration and Workflow

In our implementation, AssemblyAI is used to perform transcription and provide speaker-labeled utterances. The following steps were involved:

1. **Configure the TranscriptionConfig with speaker labels=True, punctuate=True, formatText=True**
   These flags ensure that the transcription is properly formatted and speakers are identified. The configuration plays a key role in ensuring usability of the output for diarization, providing coherent sentences with speaker mapping.

2. **Send audio file to AssemblyAI's API using its Python SDK**
   The audio file is uploaded via an API call for processing. The SDK abstracts complex API interactions, making integration fast and error-free.

3. **Receive speaker-attributed utterances with timestamps and transcribed text in JSON**
   This structured output is parsed to identify speech segments. Timestamps help align text with audio segments, while speaker IDs enable separation.

4. **Parse and extract segments with timestamps and speaker info**
   The JSON is traversed to map segments to speakers and time ranges. This stage bridges AssemblyAI and the Pyannote pipeline by providing speaker-stamped inputs for further refinement.

5. **Store or pass these segments into further processing**
   These segments are later used for embedding, clustering, or analysis. Depending on the use-case, the outputs can be visualized, stored in databases, or analyzed for decision-making.

Advantages and Limitations of AssemblyAI

Advantages:

- **Highly accurate speech-to-text conversion**
  The transcriptions produced by AssemblyAI are precise, improving downstream processing and user trust. This high accuracy is especially beneficial for applications involving decision-making, such as legal or medical transcription, where misinterpretation can have serious implications.

- **Easy to use with Python SDK**
  Developers can quickly integrate transcription into their applications using a few lines of code. This simplicity accelerates development cycles and allows rapid prototyping of audio processing applications.

- **Fast processing time for short files (under 10 mins)**
  This ensures quick turnaround in real-time or near-real-time applications. It is highly advantageous for users dealing with urgent analyses, such as emergency response or live broadcasting.

- **Integrated speaker labeling and sentiment analysis**
  Multifunctional capabilities in one API reduce the need for separate models. This modular approach lowers costs and minimizes integration challenges, making it attractive for startups and enterprises alike.

- **Secure and scalable for large datasets**
  AssemblyAI can handle high-volume uploads with encryption, making it enterprise-ready. Its cloud infrastructure ensures horizontal scalability and meets industry standards for data security.

Limitations:

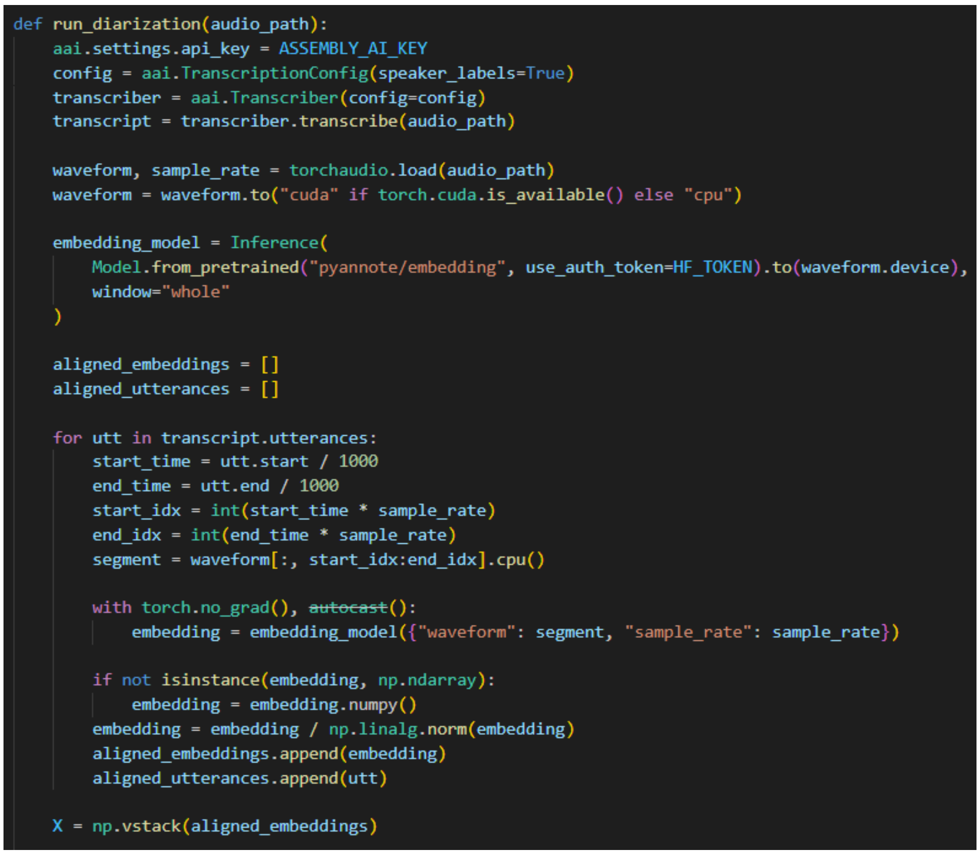
- **Requires internet connectivity**
  Since AssemblyAI is cloud-based, it cannot function in offline environments. This limits its use in remote or high-security locations without network access.

- **Performance may degrade in noisy or echo-rich environments**
  Background sounds may interfere with speech recognition and speaker differentiation. This is particularly challenging in open office settings or outdoor recordings.

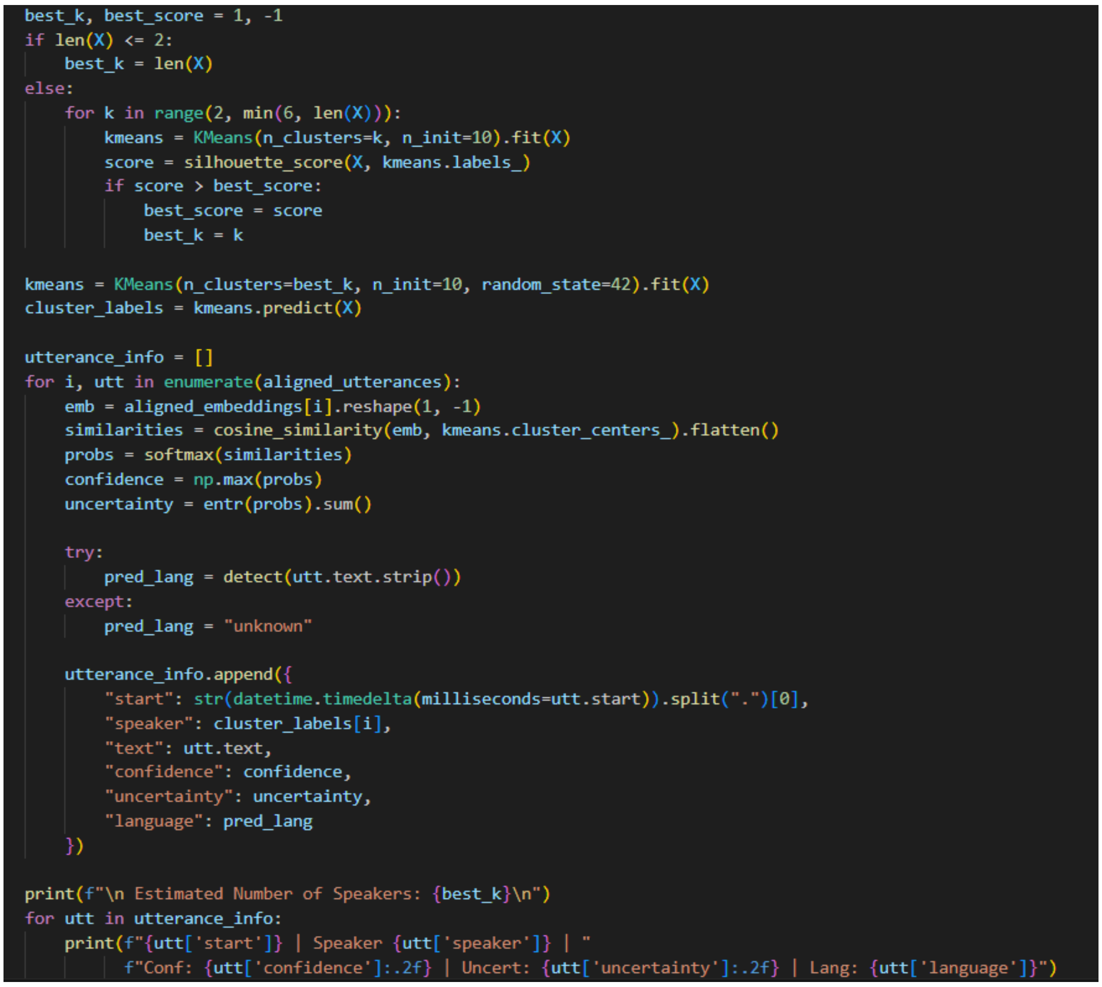
- **Speaker diarization is limited to distinguishable voice segments**
  It struggles with overlapping or very short utterances. This limitation affects the granularity and accuracy of the final transcript in fast-paced or interruptive conversations.

- **Cost considerations for long audio**
  Although affordable, AssemblyAI's usage fees can add up when processing large volumes of audio or long files. Large-scale enterprises might need to assess long-term costs for continuous processing at scale.

RUN_DIARIZATION FUNCTION:



```python
def run_diarization(audio_path):
    aai.settings.api_key = ASSEMBLY_AI_KEY
    config = aai.TranscriptionConfig(speaker_labels=True)
    transcriber = aai.Transcriber(config=config)
    transcript = transcriber.transcribe(audio_path)

    waveform, sample_rate = torchaudio.load(audio_path)
    waveform = waveform.to("cuda" if torch.cuda.is_available() else "cpu")

    embedding_model = Inference(
        Model.from_pretrained("pyannote/embedding", use_auth_token=HF_TOKEN).to(waveform.device),
        window="whole"
    )

    aligned_embeddings = []
    aligned_utterances = []

    for utt in transcript.utterances:
        start_time = utt.start / 1000
        end_time = utt.end / 1000
        start_idx = int(start_time * sample_rate)
        end_idx = int(end_time * sample_rate)
        segment = waveform[:, start_idx:end_idx].cpu()

        with torch.no_grad(), ~~autocast~~():
            embedding = embedding_model({"waveform": segment, "sample_rate": sample_rate})

        if not isinstance(embedding, np.ndarray):
            embedding = embedding.numpy()
        embedding = embedding / np.linalg.norm(embedding)
        aligned_embeddings.append(embedding)
        aligned_utterances.append(utt)

    X = np.vstack(aligned_embeddings)
```



```python
best_k, best_score = 1, -1
if len(X) <= 2:
    best_k = len(X)
else:
    for k in range(2, min(6, len(X))):
        kmeans = KMeans(n_clusters=k, n_init=10).fit(X)
        score = silhouette_score(X, kmeans.labels_)
        if score > best_score:
            best_score = score
            best_k = k

kmeans = KMeans(n_clusters=best_k, n_init=10, random_state=42).fit(X)
cluster_labels = kmeans.predict(X)

utterance_info = []
for i, utt in enumerate(aligned_utterances):
    emb = aligned_embeddings[i].reshape(1, -1)
    similarities = cosine_similarity(emb, kmeans.cluster_centers_).flatten()
    probs = softmax(similarities)
    confidence = np.max(probs)
    uncertainty = entr(probs).sum()

    try:
        pred_lang = detect(utt.text.strip())
    except:
        pred_lang = "unknown"

    utterance_info.append({
        "start": str(datetime.timedelta(milliseconds=utt.start)).split(".")[0],
        "speaker": cluster_labels[i],
        "text": utt.text,
        "confidence": confidence,
        "uncertainty": uncertainty,
        "language": pred_lang
    })

print(f"\n Estimated Number of Speakers: {best_k}\n")
for utt in utterance_info:
    print(f"{utt['start']} | Speaker {utt['speaker']} | "
          f"Conf: {utt['confidence']:.2f} | Uncert: {utt['uncertainty']:.2f} | Lang: {utt['language']}")
```

Transcription with Speaker Labels

- The `run_diarization(audio_path)` function starts by using the **AssemblyAI API** for transcription. It is configured to include **speaker labels**, so each spoken segment (utterance) includes metadata such as **speaker ID** and **timestamps**.

Loading Audio and Creating Embeddings

- The audio is loaded using **torchaudio**.
- Then, for each utterance, the corresponding audio segment is extracted and passed through a pre-trained **pyannote/embedding** model from **Hugging Face** to obtain **speaker embeddings**.
- These embeddings are **L2-normalized** and collected for further processing.

Clustering to Identify Unique Speakers

- The code uses **KMeans clustering** to group similar embeddings, thus identifying distinct speakers.
- It dynamically selects the best number of clusters (i.e., speakers) using the **Silhouette Score**, which measures how well samples are clustered.

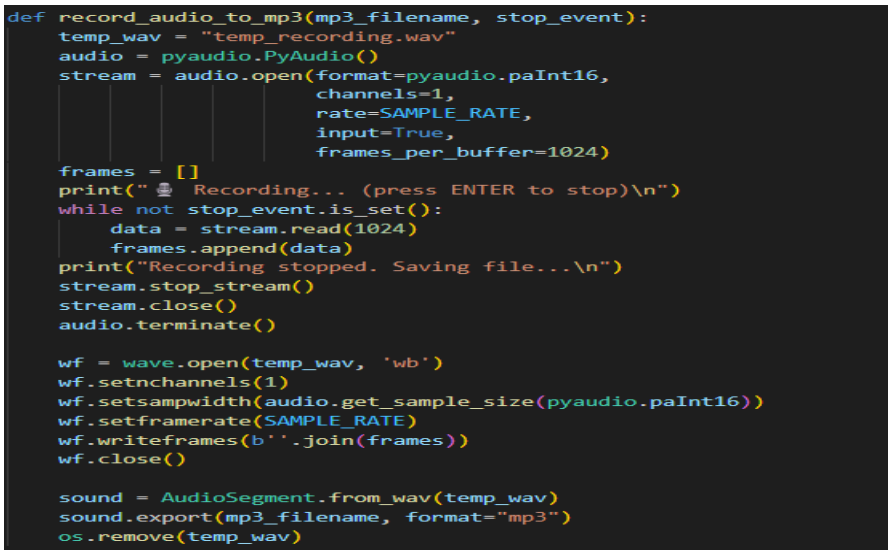
Utterance Analysis and Language Detection

For each utterance, it calculates:

- **Confidence:** Based on the maximum softmax probability of similarity with cluster centers.
- **Uncertainty:** Using **entropy** to assess ambiguity in speaker assignment.
- **Language Detection:** Tries to detect the spoken language using the **langdetect** library; defaults to "unknown" on failure.

Result Formatting and Display

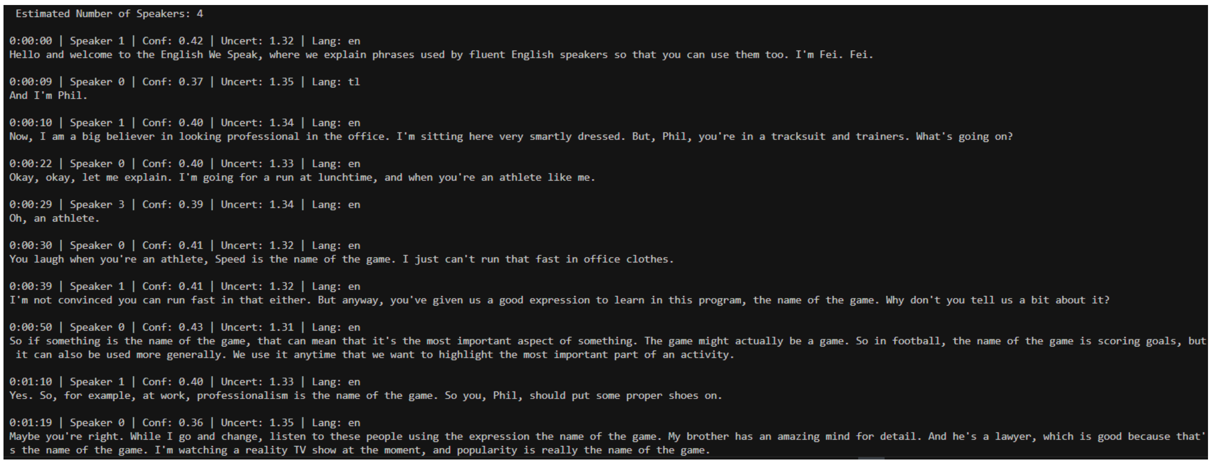
- The code stores and prints each utterance's **start time**, **speaker label**, **text**, **confidence**, **uncertainty**, and **predicted language** in a human-readable format.



```python
def record_audio_to_mp3(mp3_filename, stop_event):
    temp_wav = "temp_recording.wav"
    audio = pyaudio.PyAudio()
    stream = audio.open(format=pyaudio.paInt16,
                        channels=1,
                        rate=SAMPLE_RATE,
                        input=True,
                        frames_per_buffer=1024)
    frames = []
    print("🎙 Recording... (press ENTER to stop)\n")
    while not stop_event.is_set():
        data = stream.read(1024)
        frames.append(data)
    print("Recording stopped. Saving file...\n")
    stream.stop_stream()
    stream.close()
    audio.terminate()

    wf = wave.open(temp_wav, 'wb')
    wf.setnchannels(1)
    wf.setsampwidth(audio.get_sample_size(pyaudio.paInt16))
    wf.setframerate(SAMPLE_RATE)
    wf.writeframes(b''.join(frames))
    wf.close()

    sound = AudioSegment.from_wav(temp_wav)
    sound.export(mp3_filename, format="mp3")
    os.remove(temp_wav)
```

The `record_audio_to_mp3(mp3_filename, stop_event)` function is designed to record audio from a microphone, save it as a temporary **WAV** file, and then convert it to **MP3** format. The process begins by initializing a temporary filename `temp_recording.wav` for storing the audio. Using the **PyAudio** library, it opens a stream with specific audio parameters like **mono channel**, a predefined **sample rate**, and **16-bit PCM audio format**. The function continuously reads audio data in chunks (buffers of **1024 frames**) and appends them to a list until a **stop event** is triggered (usually by user input). Once the recording is stopped, the stream is closed, and the collected audio frames are saved into a **.wav** file using the **wave** module by setting appropriate audio metadata like **channel count**, **sample width**, and **frame rate**. After successfully writing the **WAV file**, the **pydub** library is used to convert it to **MP3** format. Finally, the temporary **WAV** file is deleted to clean up and save storage. This function is particularly useful for applications involving voice recording, such as **speech processing**, **audio logging**, or creating **voice notes**.

OUTPUTS:



```
 Estimated Number of Speakers: 4

0:00:00 | Speaker 1 | Conf: 0.42 | Uncert: 1.32 | Lang: en
Hello and welcome to the English We Speak, where we explain phrases used by fluent English speakers so that you can use them too. I'm Fei. Fei.

0:00:09 | Speaker 0 | Conf: 0.37 | Uncert: 1.35 | Lang: tl
And I'm Phil.

0:00:10 | Speaker 1 | Conf: 0.40 | Uncert: 1.34 | Lang: en
Now, I am a big believer in looking professional in the office. I'm sitting here very smartly dressed. But, Phil, you're in a tracksuit and trainers. What's going on?

0:00:22 | Speaker 0 | Conf: 0.40 | Uncert: 1.33 | Lang: en
Okay, okay, let me explain. I'm going for a run at lunchtime, and when you're an athlete like me.

0:00:29 | Speaker 3 | Conf: 0.39 | Uncert: 1.34 | Lang: en
Oh, an athlete.

0:00:30 | Speaker 0 | Conf: 0.41 | Uncert: 1.32 | Lang: en
You laugh when you're an athlete, Speed is the name of the game. I just can't run that fast in office clothes.

0:00:39 | Speaker 1 | Conf: 0.41 | Uncert: 1.32 | Lang: en
I'm not convinced you can run fast in that either. But anyway, you've given us a good expression to learn in this program, the name of the game. Why don't you tell us a bit about it?

0:00:50 | Speaker 0 | Conf: 0.43 | Uncert: 1.31 | Lang: en
So if something is the name of the game, that can mean that it's the most important aspect of something. The game might actually be a game. So in football, the name of the game is scoring goals, but
 it can also be used more generally. We use it anytime that we want to highlight the most important part of an activity.

0:01:10 | Speaker 1 | Conf: 0.40 | Uncert: 1.33 | Lang: en
Yes. So, for example, at work, professionalism is the name of the game. So you, Phil, should put some proper shoes on.

0:01:19 | Speaker 0 | Conf: 0.36 | Uncert: 1.35 | Lang: en
Maybe you're right. While I go and change, listen to these people using the expression the name of the game. My brother has an amazing mind for detail. And he's a lawyer, which is good because that'
s the name of the game. I'm watching a reality TV show at the moment, and popularity is really the name of the game.
```

Conclusion

Speaker diarization is a powerful tool for segmenting and labeling audio streams based on speaker identity, which is highly useful in real-world applications such as **meeting transcription**, **customer service analytics**, and **multimedia indexing**. This report presented a comprehensive pipeline that combines the strengths of **AssemblyAI** for transcription and speaker labeling, and the pre-trained **Pyannote model** for speaker embedding extraction and clustering. Together, these tools offer a reliable, scalable solution for speaker diarization across various contexts.

References

- **Pyannote documentation**: https://pyannote.github.io/
- **AssemblyAI documentation**: https://www.assemblyai.com/
- **"Deep Clustering and Conventional Networks for Music Separation: An Overview,"** J. R. Hershey, et al., **ICASSP 2017**.