VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Programming Intergration Project

Designing an AI-based Speaker Diarization System for subtitle generator software

Lecturer: Quan Thanh Tho
Students: Dang Mau Anh Quan – 2352983

HO CHI MINH CITY, DECEMBER 2025

TABLE OF CONTENTS

1 Introduction

1.1 Motivation

Automatic subtitle generation is commonly implemented as a multi-stage pipeline that includes audio extraction, speaker diarization, automatic speech recognition, and final subtitle formatting. Although modern ASR systems can achieve strong transcription accuracy, the overall subtitle quality is often limited by the diarization stage [1], which defines the temporal boundaries used for downstream transcription and segmentation. In practice, diarization errors such as over-segmentation at brief pauses (e.g., breathing) produce fragmented speech regions, leading to unstable subtitle timing, flickering entrys, and unnatural text breaks. This project is motivated by the need to improve the robustness and temporal stability of the diarization stage so that the subsequent transcription and `.srt` formatting steps can generate subtitles that are better synchronized and more readable under real viewing conditions.

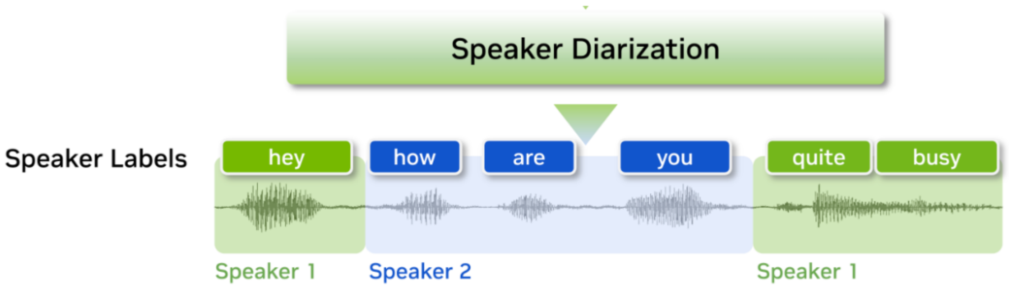


Figure 1: Illustration of speaker diarization

1.2 Problem Statement

This project considers the problem of generating a readable and temporally consistent subtitle file in SubRip (`.srt`) format from an input video. The system follows a pipeline in which audio is extracted, segmented by a speaker diarization module, transcribed by an automatic speech recognition (ASR) engine, and formatted into subtitle entries. In this setting, the diarization stage is a critical module of subtitle quality because it provides the time boundaries used for transcription. When diarization produces unstable boundaries or excessive fragmentation such as splitting a single utterance into multiple segments due to brief pauses or breathing, the resulting subtitles tend to exhibit unnatural breaks and inconsistent timing [2].

Accordingly, the problem addressed in this work is to improve the diarization-based segmentation so that speech regions are temporally stable, non-overlapping, and well-aligned with the underlying audio. The desired output is a set of coherent speech segments that supports accurate transcription and produces `.srt` files that are smoother, more readable, and closer to human-edited subtitle quality.

1.3 Objectives and Scope

The primary objective of this project is to improve the quality of automatically generated subtitles by enhancing the speaker diarization stage of the subtitle generation pipeline. Specifically, the system aims to produce cleaner and more temporally stable speech segments that reduce over-fragmentation caused by short pauses.

The scope of this report includes dataset preparation using time-annotated speech activity labels, model training and inference for diarization-based segmentation, post-processing strategies to stabilize segment boundaries, and evaluation using both segmentation-level metrics and subtitle-oriented quality criteria.

2 Methodology

2.1 System Overview

The proposed system follows a sequential pipeline to convert an input video into a finalized subtitle file in SubRip (`.srt`) format. First, the audio stream is extracted and normalized to a consistent sampling rate for downstream processing. Next, the extracted audio is passed through the speaker diarization module, which will classify it into speech segments or non-speech segments. The speech segments are then transcribed using a Whisper-based ASR module, providing transcription on a word-level timestamps. The transcription output is subsequently segmented into subtitle entries and refined by a post-processing stage that resolves overlaps and reduce grammar errors in the entries. Finally, the system exports the results as `.srt` files and which can be embedded back into the original video.

In this pipeline, the diarization stage is treated as the primary leverage point for improving subtitle quality. More stable and coherent diarization boundaries reduce unnecessary fragmentation before ASR, leading to smoother subtitle timing and more natural entry segmentation in the final `.srt` output.

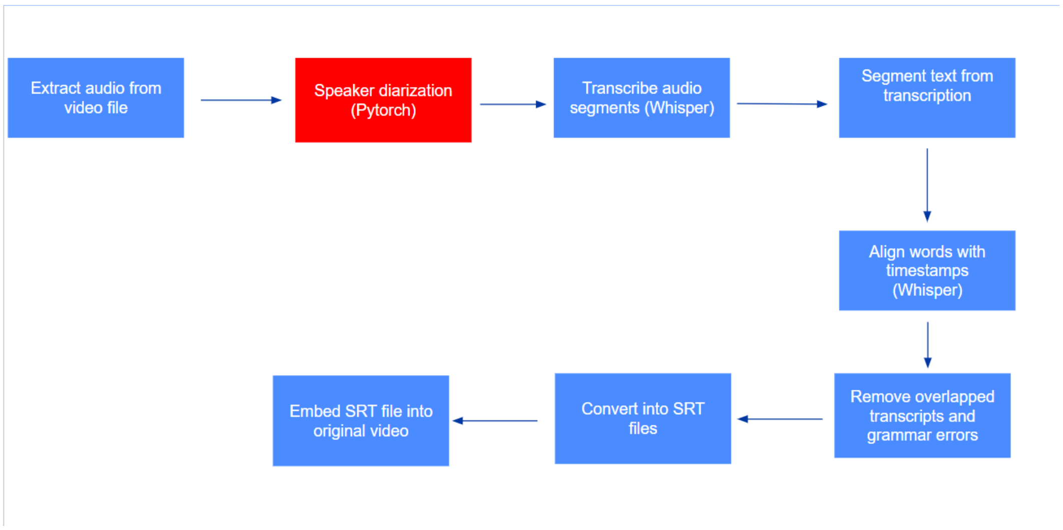

Extract audio from video file
Speaker diarization (Pytorch)
Transcribe audio segments (Whisper)
Segment text from transcription
Align words with timestamps (Whisper)
Remove overlapped transcripts and grammar errors
Convert into SRT files
Embed SRT file into original video

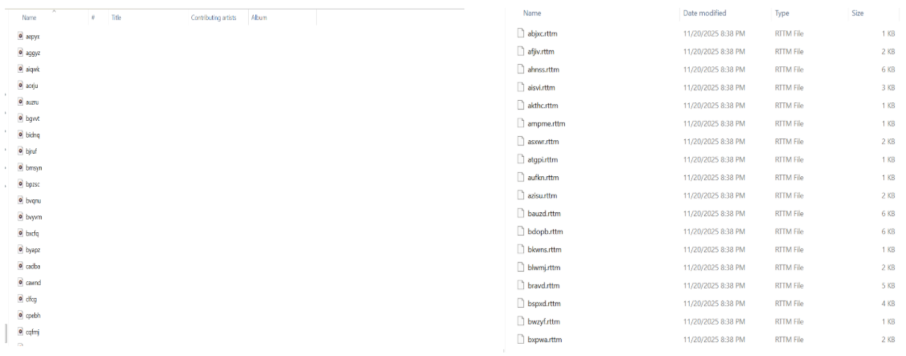
Figure 2: An overview of the system structure

2.2 Data Preparation

Data preparation was carried out to ensure that the training pipeline operates on a clean and consistent set of audio recordings and time-aligned annotations. The overall goal of this step is to verify that every annotation has a corresponding audio file, standardize audio to a uniform format required by the model, and organize the dataset into training and validation splits. Throughout the process, invalid or mismatched entries (e.g., missing audio for an RTTM file) are explicitly excluded, ensuring that subsequent training relies only on verified inputs.

2.2.1 Dataset Selection and Organization

VoxConverse is adopted as the primary resource due to its realistic conversational audio and high-quality time annotations provided in RTTM format. Each recording is represented by a waveform file (`.wav`) and a corresponding RTTM file describing speech regions. During preparation, all available pairs are enumerated and validated so that only samples with both audio and annotation are included. For convenience and reproducibility, the dataset is organized into subsets (e.g., development and test).



Name	#	Title	Contributing artists	Album
aepyx				
aggyz				
aiqwk				
aorju				
auzru				
bgwt				
bidnq				
bjruf				
bmsyn				
bpzsc				
bvqnu				
bvyvm				
bxcfq				
byapz				
cadba				
cawnd				
clfcg				
cpebh				
cqfmj				

Name	Date modified	Type	Size
abjxc.rttm	11/20/2025 8:38 PM	RTTM File	1 KB
afjiv.rttm	11/20/2025 8:38 PM	RTTM File	2 KB
ahnss.rttm	11/20/2025 8:38 PM	RTTM File	6 KB
aisvi.rttm	11/20/2025 8:38 PM	RTTM File	3 KB
akthc.rttm	11/20/2025 8:38 PM	RTTM File	1 KB
ampme.rttm	11/20/2025 8:38 PM	RTTM File	1 KB
asxwr.rttm	11/20/2025 8:38 PM	RTTM File	2 KB
atgpi.rttm	11/20/2025 8:38 PM	RTTM File	1 KB
aufkn.rttm	11/20/2025 8:38 PM	RTTM File	1 KB
azisu.rttm	11/20/2025 8:38 PM	RTTM File	2 KB
bauzd.rttm	11/20/2025 8:38 PM	RTTM File	6 KB
bdopb.rttm	11/20/2025 8:38 PM	RTTM File	6 KB
bkwns.rttm	11/20/2025 8:38 PM	RTTM File	1 KB
blwmj.rttm	11/20/2025 8:38 PM	RTTM File	2 KB
bravd.rttm	11/20/2025 8:38 PM	RTTM File	5 KB
bspxd.rttm	11/20/2025 8:38 PM	RTTM File	4 KB
bwzyf.rttm	11/20/2025 8:38 PM	RTTM File	1 KB
bxpwa.rttm	11/20/2025 8:38 PM	RTTM File	2 KB

Figure 3: Illustration of the VoxConverse dataset.

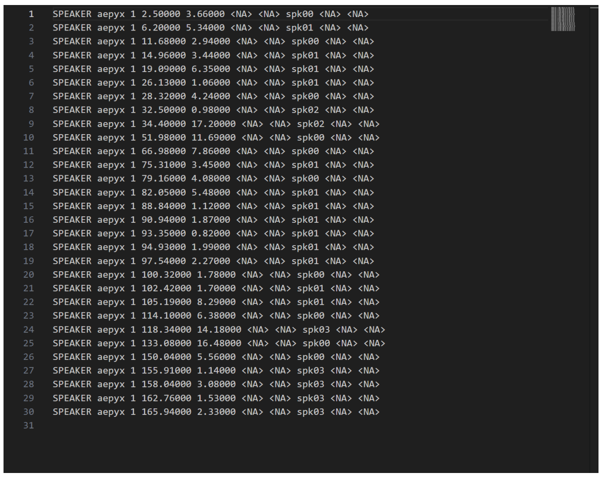
2.2.2 Audio Preprocessing

All audio recordings are standardized to a consistent format prior to training. In particular, recordings are converted to mono and resampled to 16 kHz, matching the model configuration and common practice for speech activity modeling. Each waveform is represented in floating-point format to ensure numerical stability during training. To support minibatch training, the audio is further segmented into fixed-length windows of 5.0 s with 50% overlap (i.e., 2.5 s hop size). Short trailing segments are zero-padded to maintain a uniform duration. This windowing strategy increases the number of training examples while preserving temporal context.

2.2.3 RTTM File Processing

RTTM annotations are used as the ground-truth metadata to describe speech activity within each recording. During data preparation, we first verify the consistency of the annotation set by keeping only recordings that have both RTTM annotations and the corresponding audio file.

For model development, the original development set is further partitioned into training and validation subsets using a fixed 80%/20% ratio. This results in a clean and reproducible set of artifacts for subsequent training and evaluation.



```
SPEAKER aepyx 1 2.50000 3.66000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 6.20000 5.34000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 11.68000 2.94000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 14.96000 3.44000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 19.09000 6.35000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 26.13000 1.06000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 28.32000 4.24000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 32.50000 0.98000 <NA> <NA> spk02 <NA> <NA>
SPEAKER aepyx 1 34.40000 17.20000 <NA> <NA> spk02 <NA> <NA>
SPEAKER aepyx 1 51.98000 11.69000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 66.98000 7.86000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 75.31000 3.45000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 79.16000 4.08000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 82.05000 5.48000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 88.84000 1.12000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 90.94000 1.87000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 93.35000 0.82000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 94.93000 1.99000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 97.54000 2.27000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 100.32000 1.78000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 102.42000 1.70000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 105.19000 8.29000 <NA> <NA> spk01 <NA> <NA>
SPEAKER aepyx 1 114.10000 6.38000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 118.34000 14.18000 <NA> <NA> spk03 <NA> <NA>
SPEAKER aepyx 1 133.08000 16.48000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 150.04000 5.56000 <NA> <NA> spk00 <NA> <NA>
SPEAKER aepyx 1 155.91000 1.14000 <NA> <NA> spk03 <NA> <NA>
SPEAKER aepyx 1 158.04000 3.08000 <NA> <NA> spk03 <NA> <NA>
SPEAKER aepyx 1 162.76000 1.53000 <NA> <NA> spk03 <NA> <NA>
SPEAKER aepyx 1 165.94000 2.33000 <NA> <NA> spk03 <NA> <NA>
```

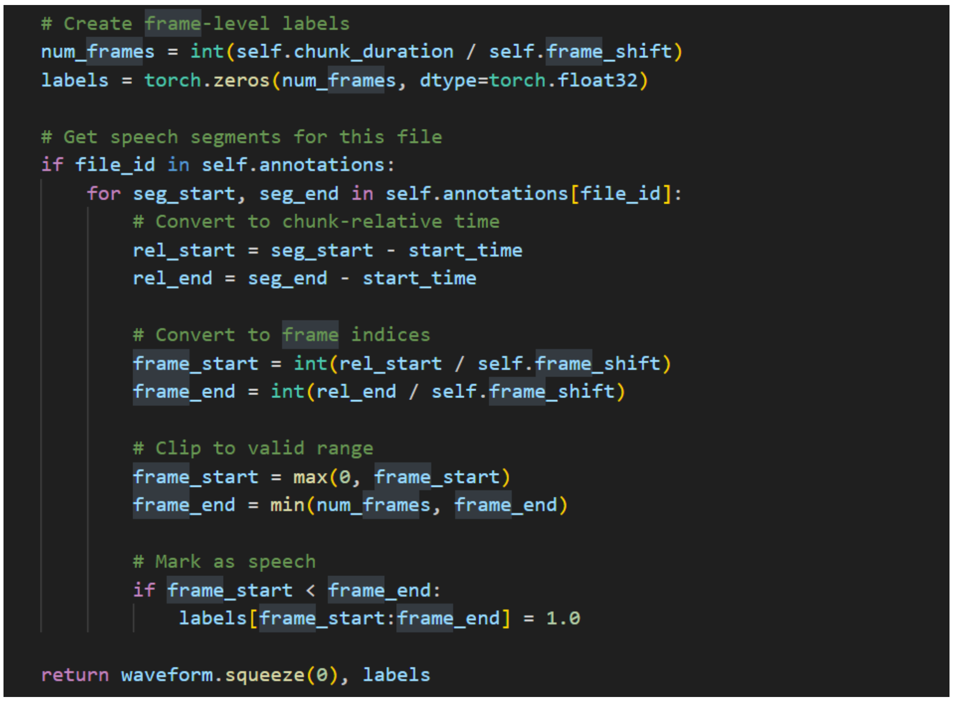
Figure 4: An example of a RTTM file

2.3 Training Pipeline

2.3.1 Overview

This project trains a voice activity detection (VAD) model to classify short time frames as either speech or non-speech. The training pipeline follows three main steps which are loading audio recordings and their RTTM speech annotations, creating fixed-length audio chunks with corresponding frame-level labels, and optimizing a neural network to predict speech activity over time.

At a high level, the model takes a raw waveform segment as input and outputs a sequence of speech probabilities (one value per frame). During training, the model is updated by minimizing a binary classification loss between predicted speech activity and the ground-truth labels derived from RTTM annotations. Standard training utilities are used to monitor validation performance and save the best-performing checkpoint for later use.



```python
# Create frame-level labels
num_frames = int(self.chunk_duration / self.frame_shift)
labels = torch.zeros(num_frames, dtype=torch.float32)

# Get speech segments for this file
if file_id in self.annotations:
    for seg_start, seg_end in self.annotations[file_id]:
        # Convert to chunk-relative time
        rel_start = seg_start - start_time
        rel_end = seg_end - start_time

        # Convert to frame indices
        frame_start = int(rel_start / self.frame_shift)
        frame_end = int(rel_end / self.frame_shift)

        # Clip to valid range
        frame_start = max(0, frame_start)
        frame_end = min(num_frames, frame_end)

        # Mark as speech
        if frame_start < frame_end:
            labels[frame_start:frame_end] = 1.0

return waveform.squeeze(0), labels
```

Figure 5: Frame probabilities processing

2.3.2 Model Architecture

The proposed VAD model processes raw audio for every time frame to decide if it contains speech. Figure 6 summarizes the main components and the flow of information from the waveform input to the final frame-level predictions.

The first stage is a SincNet-style convolutional front-end that learns a set of audio filters directly from the waveform [3]. This stage plays an important role in feature extraction. Instead of relying on hand-crafted acoustic features, the model learns frequency-selective patterns that are useful for separating speech from non-speech. The extracted features are then passed through batch normalization followed by a ReLU nonlinearity, which helps keep the feature scale stable and improves the effectiveness of subsequent processing.

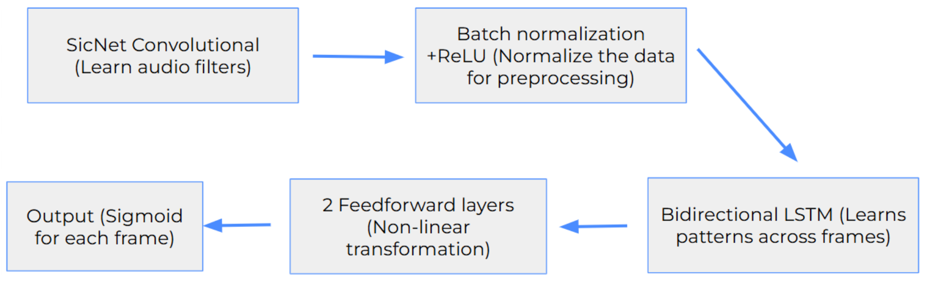

SicNet Convolutional (Learn audio filters)
Batch normalization +ReLU (Normalize the data for preprocessing)
Bidirectional LSTM (Learns patterns across frames)
2 Feedforward layers (Non-linear transformation)
Output (Sigmoid for each frame)

Figure 6: Overall model flow for voice activity detection.

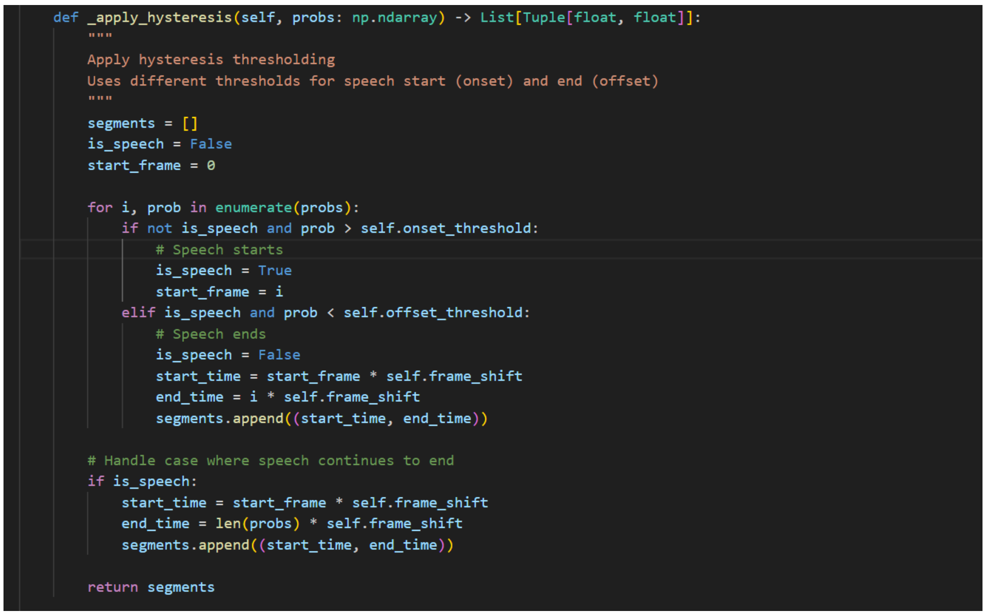
After feature normalization, the model uses a bidirectional LSTM to capture temporal context. Speech activity is inherently time-dependent, so decisions are more reliable when the model can observe neighboring frames rather than treating each frame independently. By reading the sequence in both directions, the bidirectional LSTM learns patterns such as speech continuity, pauses, and transitions between speech and silence, which reduces fragmented or noisy predictions.

Finally, the LSTM outputs are refined by two feedforward fully-connected layers that apply additional nonlinear transformations before producing the final output. The model ends with a sigmoid activation that converts the last layer output into a probability for each frame, where values closer to 1 indicate speech and values closer to 0 indicate non-speech. These frame-level probabilities are later converted into continuous speech segments using the post-processing method described in the inference subsection.

2.3.3 Inference and Post-processing

After training, inference applies the saved model to a new audio recording and returns a list of detected speech segments. The audio is first converted to the expected format such as mono waveform and 16 kHz sample rate. Then, the system runs the model over the recording in overlapping chunks and combines the chunk-level predictions into a single probability timeline.

To convert probabilities into human-readable speech intervals, the inference stage uses a simple post-processing strategy which is hysteresis thresholding followed by duration-based filtering. Very short speech bursts are removed, and segments separated by a short silence gap can be merged to form cleaner, more stable speech regions. The final output is a list of time ranges (start and end time) in seconds, which can be used directly by later modules.



```python
def _apply_hysteresis(self, probs: np.ndarray) -> List[Tuple[float, float]]:
    """
    Apply hysteresis thresholding
    Uses different thresholds for speech start (onset) and end (offset)
    """
    segments = []
    is_speech = False
    start_frame = 0

    for i, prob in enumerate(probs):
        if not is_speech and prob > self.onset_threshold:
            # Speech starts
            is_speech = True
            start_frame = i
        elif is_speech and prob < self.offset_threshold:
            # Speech ends
            is_speech = False
            start_time = start_frame * self.frame_shift
            end_time = i * self.frame_shift
            segments.append((start_time, end_time))

    # Handle case where speech continues to end
    if is_speech:
        start_time = start_frame * self.frame_shift
        end_time = len(probs) * self.frame_shift
        segments.append((start_time, end_time))

    return segments
```

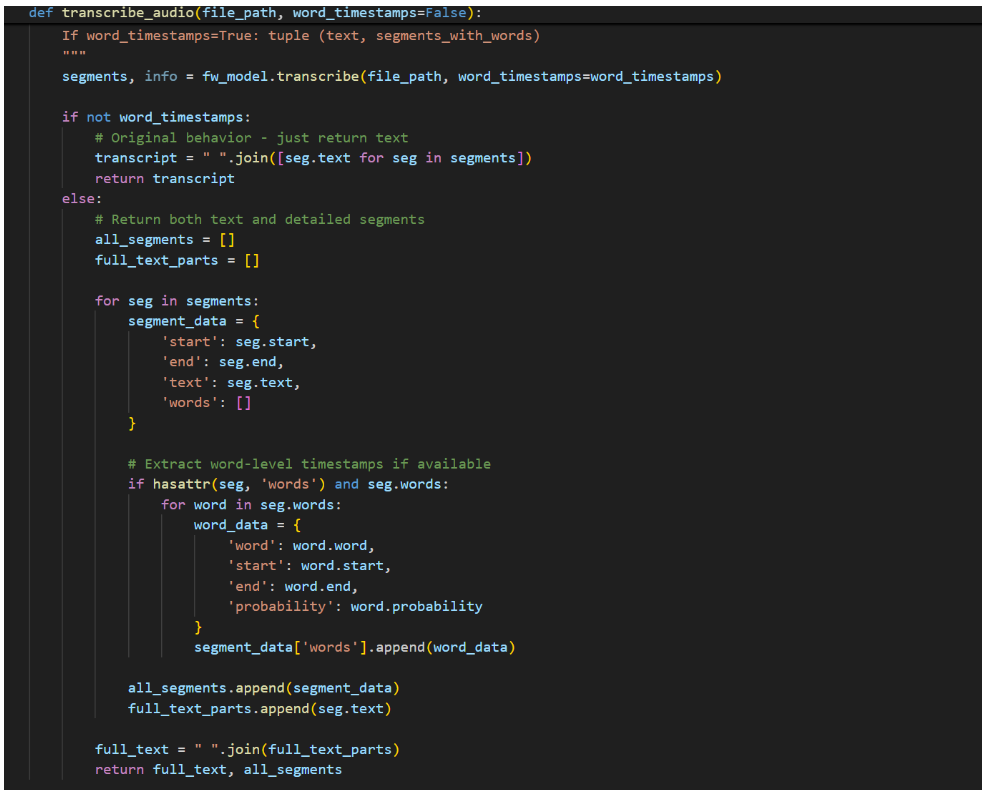
Figure 7: Illustration of hysteresis thresholding for speech segmentation.

2.4 ASR and Subtitle Generation

After speech regions are detected, the system generates subtitles by transcribing each region and converting the transcription into standard SRT entries. The goal of this stage is to produce readable subtitles with reliable timestamps, while keeping the pipeline lightweight and reproducible.

2.4.1 Faster-Whisper Transcription

We use Faster-Whisper to perform automatic speech recognition (ASR) on the extracted speech segments. In addition to producing the transcript text, Faster-Whisper can return word-level timestamps. These timestamps are especially useful for subtitle generation because they allow the system to place subtitle boundaries based on the actual timing of each word soken.



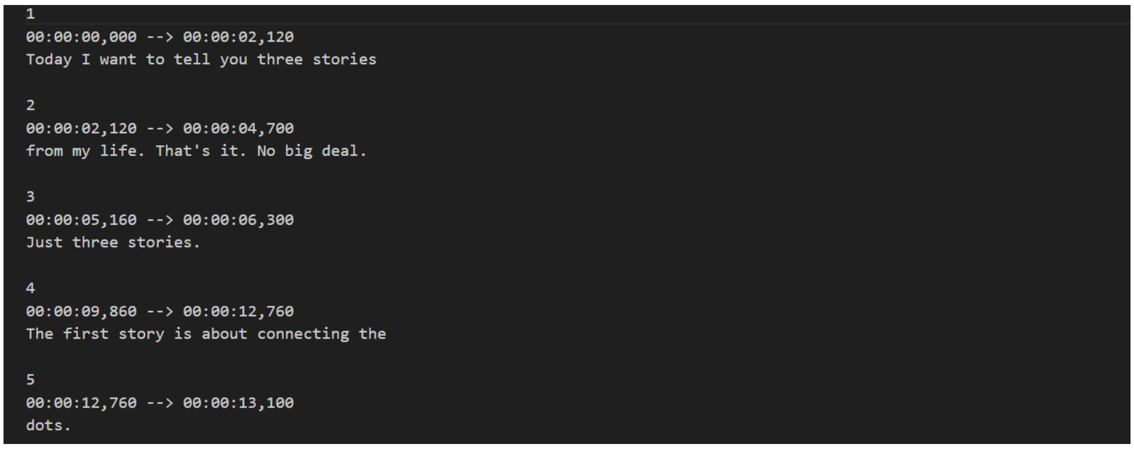
```
def transcribe_audio(file_path, word_timestamps=False):
    If word_timestamps=True: tuple (text, segments_with_words)
    """
    segments, info = fw_model.transcribe(file_path, word_timestamps=word_timestamps)

    if not word_timestamps:
        # Original behavior - just return text
        transcript = " ".join([seg.text for seg in segments])
        return transcript
    else:
        # Return both text and detailed segments
        all_segments = []
        full_text_parts = []

        for seg in segments:
            segment_data = {
                'start': seg.start,
                'end': seg.end,
                'text': seg.text,
                'words': []
            }

            # Extract word-level timestamps if available
            if hasattr(seg, 'words') and seg.words:
                for word in seg.words:
                    word_data = {
                        'word': word.word,
                        'start': word.start,
                        'end': word.end,
                        'probability': word.probability
                    }
                    segment_data['words'].append(word_data)

            all_segments.append(segment_data)
            full_text_parts.append(seg.text)

        full_text = " ".join(full_text_parts)
        return full_text, all_segments
```

Figure 8: Overall ASR and subtitle generation flow.

2.4.2 Timestamp Handling and Subtitle Segmentation

The subtitle timing is derived directly from the ASR timestamps. In practice, the system builds subtitles by scanning words in time order and grouping them into short chunks that are easy to read. A new subtitle chunk is started when a clear pause is observed between consecutive words, or when the current subtitle would become too long. This produces subtitle entries that follow natural speaking rhythm and avoids overly dense lines.

Because each speech segment is processed independently, the subtitle timestamps are adjusted to match their absolute position in the full audio (i.e., the segment start time is added back). After subtitles are written, a final pass checks for timestamp overlaps and slightly adjusts end times when needed to keep the SRT timeline valid and readable.



```
1
00:00:00,000 --> 00:00:02,120
Today I want to tell you three stories

2
00:00:02,120 --> 00:00:04,700
from my life. That's it. No big deal.

3
00:00:05,160 --> 00:00:06,300
Just three stories.

4
00:00:09,860 --> 00:00:12,760
The first story is about connecting the

5
00:00:12,760 --> 00:00:13,100
dots.
```

Figure 9: Example of speech chunks in SRT file.

3 Experiments and Results

3.1 Experimental Setup

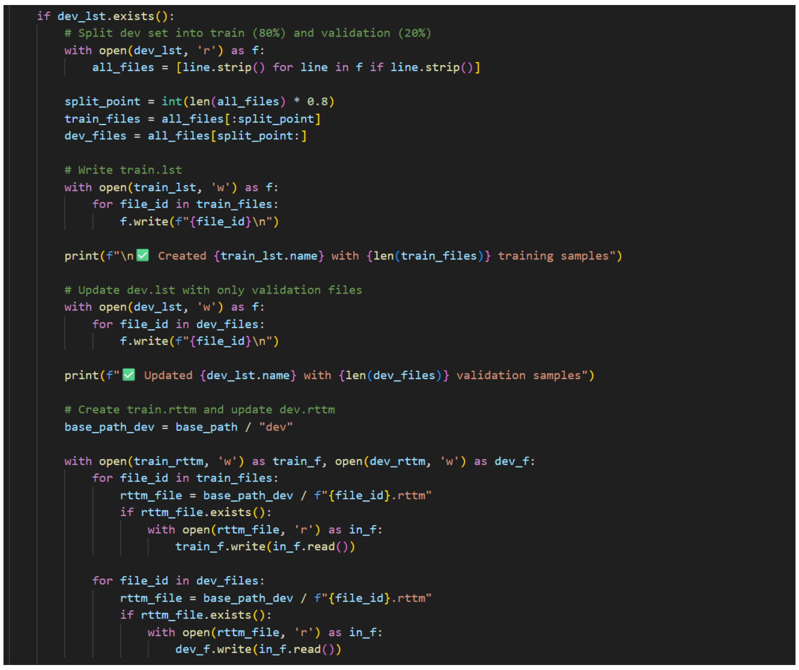
3.1.1 Data Splits and Preprocessing

All experiments follow a simple and reproducible data preparation procedure. We use the available development portion as the source for model learning, and split it into training (80%) and validation (20%) sets at the recording level to avoid overlap between subsets. A separate test split is kept for final checking. Ground-truth speech regions are provided in RTTM format, and split-specific RTTM files are consolidated so that each split has one consistent annotation file.

For preprocessing, audio is standardized to a single-channel (mono) waveform at 16 kHz sampling rate to match the model input requirements. During training, each recording is converted into fixed-length chunks (5 seconds) with overlap, and frame-level labels are created from RTTM speech intervals using a 10 ms frame step. This produces a clean set of input–label pairs for VAD learning.

3.1.2 Training Configuration

We train the VAD model using a lightweight configuration to keep the pipeline practical. The architecture follows a straightforward flow such that a SincNet-style convolutional front-end extracts waveform features, a bidirectional LSTM



```python
if dev_lst.exists():
    # Split dev set into train (80%) and validation (20%)
    with open(dev_lst, 'r') as f:
        all_files = [line.strip() for line in f if line.strip()]

    split_point = int(len(all_files) * 0.8)
    train_files = all_files[:split_point]
    dev_files = all_files[split_point:]

    # Write train.lst
    with open(train_lst, 'w') as f:
        for file_id in train_files:
            f.write(f"{file_id}\n")

    print(f"\n✅ Created {train_lst.name} with {len(train_files)} training samples")

    # Update dev.lst with only validation files
    with open(dev_lst, 'w') as f:
        for file_id in dev_files:
            f.write(f"{file_id}\n")

    print(f"✅ Updated {dev_lst.name} with {len(dev_files)} validation samples")

    # Create train.rttm and update dev.rttm
    base_path_dev = base_path / "dev"

    with open(train_rttm, 'w') as train_f, open(dev_rttm, 'w') as dev_f:
        for file_id in train_files:
            rttm_file = base_path_dev / f"{file_id}.rttm"
            if rttm_file.exists():
                with open(rttm_file, 'r') as in_f:
                    train_f.write(in_f.read())

        for file_id in dev_files:
            rttm_file = base_path_dev / f"{file_id}.rttm"
            if rttm_file.exists():
                with open(rttm_file, 'r') as in_f:
                    dev_f.write(in_f.read())
```

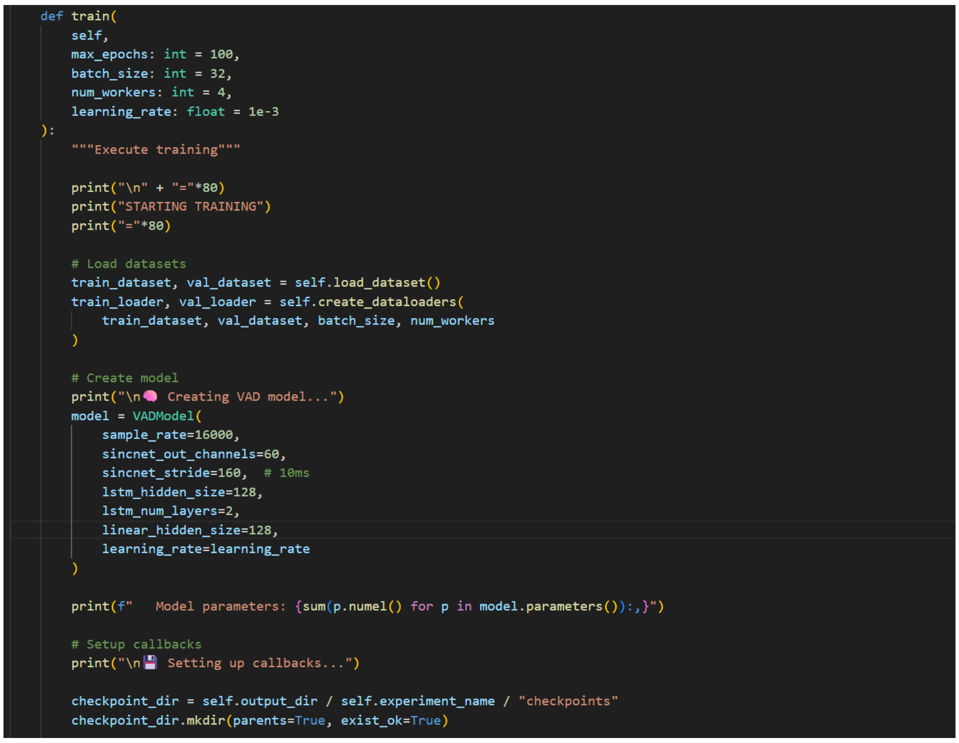
Figure 10: Dataset splits and consolidated RTTM annotations.

models temporal context, and two feedforward layers produce frame-level speech probabilities.

Optimization is done with the Adam optimizer and a standard learning-rate scheduling strategy based on validation loss. Training runs for up to a fixed number of epochs, while early stopping and checkpointing are used to prevent overfitting and to keep the best model for later inference. We also enable simple audio augmentation during training (e.g., random gain and additive noise) to improve robustness [4].

3.1.3 Compute Environment

Experiments are implemented with PyTorch and trained using PyTorch Lightning. When a CUDA-capable GPU is available, training is to speed up optimization. Otherwise, the same pipeline can run on CPU. In the full subtitle generation pipeline, ASR is performed using Faster-Whisper, which is also configured to use GPU when possible.



```python
def train(
    self,
    max_epochs: int = 100,
    batch_size: int = 32,
    num_workers: int = 4,
    learning_rate: float = 1e-3
):
    """Execute training"""

    print("\n" + "="*80)
    print("STARTING TRAINING")
    print("="*80)

    # Load datasets
    train_dataset, val_dataset = self.load_dataset()
    train_loader, val_loader = self.create_dataloaders(
        train_dataset, val_dataset, batch_size, num_workers
    )

    # Create model
    print("\n🧠 Creating VAD model...")
    model = VADModel(
        sample_rate=16000,
        sincnet_out_channels=60,
        sincnet_stride=160,  # 10ms
        lstm_hidden_size=128,
        lstm_num_layers=2,
        linear_hidden_size=128,
        learning_rate=learning_rate
    )

    print(f"   Model parameters: {sum(p.numel() for p in model.parameters()):,}")

    # Setup callbacks
    print("\n💾 Setting up callbacks...")

    checkpoint_dir = self.output_dir / self.experiment_name / "checkpoints"
    checkpoint_dir.mkdir(parents=True, exist_ok=True)
```

Figure 11: Training configuration used for the VAD model.

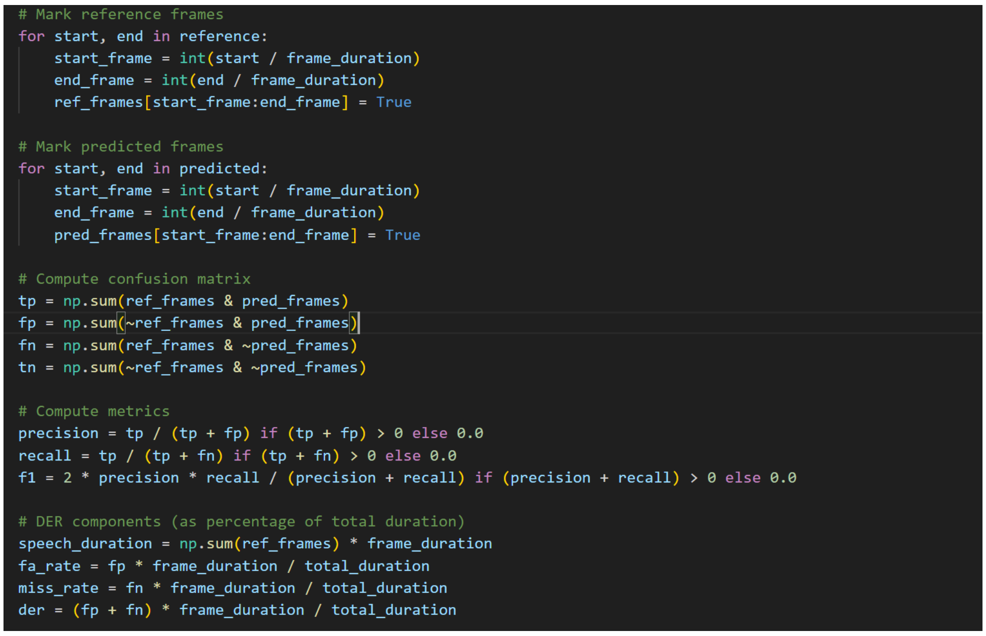
3.2 Evaluation Method

We evaluate the system in two parts. First, we measure how well the VAD model detects where speech happens in the audio. Second, we evaluate the generated subtitles by comparing them with a reference subtitle set, focusing on both transcription correctness and timestamp alignment.

3.2.1 VAD Evaluation

We evaluate the VAD module on the VoxConverse dataset by comparing the model's predicted speech regions with the reference speech annotations provided in RTTM format. In our setup, VoxConverse recordings are organized into a dedicated test split, and evaluation is performed only on the recordings listed for testing. For each test recording, the trained VAD model is run to produce a set of predicted speech segments (start time and end time).

To compute scores consistently, both the predicted segments and the reference segments are converted into a frame-level timeline using a fixed frame step of 10 ms. This produces two aligned binary sequences over time (speech vs. non-speech), which allows us to count true positives, false positives, false negatives, and true negatives across the full recording. From these counts, we report precision, recall, and F1-score to summarize how well the model detects speech frames. We also report a DER-style detection error rate that summarizes overall detection errors, together with simple components that reflect false alarms (predicting speech where there is none) and misses (failing to detect speech). Finally, all metrics are aggregated over the entire VoxConverse test set by reporting the average performance and variability across files.



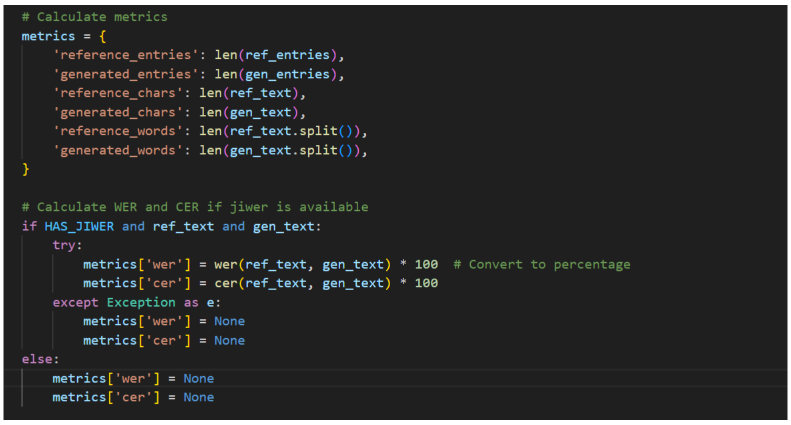
```python
# Mark reference frames
for start, end in reference:
    start_frame = int(start / frame_duration)
    end_frame = int(end / frame_duration)
    ref_frames[start_frame:end_frame] = True

# Mark predicted frames
for start, end in predicted:
    start_frame = int(start / frame_duration)
    end_frame = int(end / frame_duration)
    pred_frames[start_frame:end_frame] = True

# Compute confusion matrix
tp = np.sum(ref_frames & pred_frames)
fp = np.sum(~ref_frames & pred_frames)
fn = np.sum(ref_frames & ~pred_frames)
tn = np.sum(~ref_frames & ~pred_frames)

# Compute metrics
precision = tp / (tp + fp) if (tp + fp) > 0 else 0.0
recall = tp / (tp + fn) if (tp + fn) > 0 else 0.0
f1 = 2 * precision * recall / (precision + recall) if (precision + recall) > 0 else 0.0

# DER components (as percentage of total duration)
speech_duration = np.sum(ref_frames) * frame_duration
fa_rate = fp * frame_duration / total_duration
miss_rate = fn * frame_duration / total_duration
der = (fp + fn) * frame_duration / total_duration
```

Figure 12: Evaluation overview for the VAD module.

3.2.2 Subtitle Evaluation

Subtitle quality is evaluated by comparing the SRT files produced by our pipeline against reference SRT files prepared for the same recordings. In our experiments, we run the full subtitle generation pipeline on the 20 Ted Talk videos selected for evaluation and save the generated subtitles as standard SRT outputs. For each evaluated recording, we pair the generated SRT with its corresponding reference SRT and compute both text accuracy and timing consistency.

For text accuracy, each SRT file is parsed into subtitle entries, where every entry contains a start time, an end time, and the displayed text. We then concatenate the subtitle texts in time order and measure the difference between the generated text and the reference text using word error rate and character error rate. For timing quality, we estimate synchronization by checking subtitle entries that overlap in time and measuring the absolute difference between their the generated and reference SRT files.



```python
# Calculate metrics
metrics = {
    'reference_entries': len(ref_entries),
    'generated_entries': len(gen_entries),
    'reference_chars': len(ref_text),
    'generated_chars': len(gen_text),
    'reference_words': len(ref_text.split()),
    'generated_words': len(gen_text.split()),
}

# Calculate WER and CER if jiwer is available
if HAS_JIWER and ref_text and gen_text:
    try:
        metrics['wer'] = wer(ref_text, gen_text) * 100  # Convert to percentage
        metrics['cer'] = cer(ref_text, gen_text) * 100
    except Exception as e:
        metrics['wer'] = None
        metrics['cer'] = None
else:
    metrics['wer'] = None
    metrics['cer'] = None
```

Figure 13: Evaluation overview for the SRT module.

3.3 Results

3.3.1 VAD Results

We report VAD performance on the VoxConverse evaluation split by comparing predicted speech regions against the RTTM reference annotations. Overall, the model achieves high recall, meaning it detects most speech frames, while precision is slightly lower due to false alarms in challenging segments (e.g., background sounds). The final F1-score indicates a strong balance between detecting speech and avoiding non-speech, and the DER-style score summarizes the remaining detection errors over the whole test set. In this run, the evaluated duration is 0.18 hours of audio, containing 0.16 hours of speech.

Metric	Value
Precision	0.8921 ± 0.0369
Recall	0.9900 ± 0.0142
F1-score	0.9383 ± 0.0254
DER (speech detection)	0.1152 ± 0.0452
False alarm rate	0.1066
Miss rate	0.0086

Table 1: Summary VAD results (mean ± standard deviation across evaluated recordings).

3.3.2 Subtitle Results

Subtitle evaluation is conducted on the TED-talk dataset by pairing each generated SRT file with its corresponding reference SRT file and measuring both text accuracy and timing consistency. Overall, the system reaches an overall quality level of "Good" based on the evaluation criteria. On average, the generated subtitles contain slightly more words than the reference ($52698 \rightarrow 54514$), while the number of subtitle entries increases more clearly ($6912 \rightarrow 9016$). This confirms that the system tends to split subtitles into smaller chunks than the ground truth, which improves readability in some cases but can also lead to over-segmentation.

Regarding transcription correctness, the average WER is 23.2% and the average CER is 11.5%. For timing, the average start-time deviation is 0.864 seconds, meaning that subtitles are generally aligned with the reference timeline, but noticeable offsets still appear in some clips. Across the evaluated files, the quality labels are distributed as follows: 2 files are rated Excellent, 9 files are rated Good, 5 files are rated Fair, and 3 files are rated Poor. The strongest cases (Excellent/-Good) typically combine low WER/CER with moderate timing deviation, while the weaker cases are associated with large transcription errors and/or unstable segmentation, which increases the mismatch in subtitle entry counts.

File	Entries (Ref → Gen)	Words (Ref → Gen)	WER (%)	CER (%)	Timing Δ (s)	Quality
01.srt	128 → 134	828 → 791	27.7	13.6	1.100 ± 1.369	Fair
02.srt	427 → 550	3170 → 3226	24.4	12.7	0.805 ± 0.804	Fair
03.srt	428 → 601	3647 → 3669	17.2	7.3	0.792 ± 0.711	Good
04.srt	315 → 379	2277 → 2271	16.9	7.6	0.743 ± 0.714	Good
05.srt	324 → 434	2703 → 2664	16.9	7.4	1.156 ± 0.837	Good
06.srt	322 → 405	2760 → 2704	16.9	9.5	0.803 ± 0.752	Good
07.srt	12 → 14	87 → 93	42.5	16.5	1.035 ± 0.532	Poor
09.srt	229 → 281	1667 → 1655	11.3	5.2	0.679 ± 0.707	Good
10.srt	139 → 482	2005 → 2765	51.6	46.8	0.870 ± 0.795	Poor
11.srt	384 → 431	2921 → 2913	35.9	12.5	0.951 ± 0.658	Fair
12.srt	424 → 498	2856 → 2853	16.5	7.6	1.014 ± 0.854	Good
13.srt	468 → 563	3158 → 3152	7.9	2.4	0.701 ± 0.758	Excellent
14.srt	77 → 172	973 → 976	7.3	1.6	0.468 ± 0.630	Excellent
15.srt	410 → 486	2817 → 2798	28.6	8.6	0.878 ± 0.680	Fair
16.srt	216 → 304	1782 → 1764	15.8	5.2	0.781 ± 0.850	Good
17.srt	427 → 553	3170 → 3250	27.9	15.5	0.807 ± 0.823	Fair
18.srt	329 → 472	2732 → 2721	18.9	9.4	0.757 ± 0.837	Good
19.srt	1595 → 1986	11406 → 12512	46.6	26.5	0.967 ± 0.732	Poor
20.srt	258 → 271	1739 → 1737	10.9	2.9	1.107 ± 0.823	Good
Average	6912 → 9016	52698 → 54514	23.2	11.5	0.864	Good

Table 2: SRT evaluation results on the TED-talk dataset (reference vs. generated subtitles).

4 Conclusion and Future Work

4.1 Conclusion

This project developed an AI-based subtitle generation pipeline that combines voice activity detection and automatic speech recognition to produce time-aligned SRT subtitles. The VAD module uses a SincNet–BiLSTM architecture to detect speech regions from raw audio and achieved strong performance on VoxConverse, with precision 0.892, recall 0.990, and F1-score 0.938 (DER 0.115). Using these detected speech segments, the ASR stage generates subtitle text with timestamps and formats them into standard SRT outputs. On the evaluated subtitle set, the system reached an overall "Good" quality level, with average WER 23.2%, CER 11.5%, and an average timing deviation of 0.864 seconds. Overall, the results show that the pipeline can reliably identify speech and produce readable subtitles with reasonable alignment.

4.2 Limitations and Future Work

Despite promising results, several limitations remain. First, subtitle quality is sensitive to timing offsets and segmentation behavior; the system tends to produce more subtitle entries than the reference, indicating over-segmentation in some cases. Second, ASR errors still contribute noticeably to the final WER/CER, especially on difficult audio conditions. Third, the current evaluation was conducted on a limited amount of test audio, so broader testing is needed to confirm robustness across diverse domains.

Future work will focus on improving both timing and readability of subtitles by refining segmentation policies (e.g., stronger pause-based grouping and stricter length constraints), and by adding a lightweight alignment step to reduce systematic timestamp drift. In addition, robustness can be improved by extending augmentation and training with noisier conditions, as well as testing on more varied datasets. Finally, integrating speaker diarization would enable speaker-aware subtitles (e.g., speaker labels or per-speaker grouping), which is an important step toward cleaner subtitles for multi-speaker conversations.

5 References

References

[1] J. S. Chung, J. Huh, A. Nagrani, T. Afouras, and A. Zisserman, "Spot the Conversation: Speaker Diarisation in the Wild," in *Proc. Interspeech*, 2020, pp. 299–303, doi:10.21437/Interspeech.2020-2337.

[2] J. Ramirez, J. C. Segura, C. Benitez, A. de la Torre, and A. Rubio, "Improved voice activity detection combining noise reduction and subband divergence measures," in *Proc. Interspeech*, 2004, pp. 961–964, doi:10.21437/Interspeech.2004-347.

[3] M. Ravanelli and Y. Bengio, "Speaker Recognition from Raw Waveform with SincNet," *arXiv preprint* arXiv:1808.00158, 2018.

[4] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust Speech Recognition via Large-Scale Weak Supervision," in *Proc. ICML*, *PMLR*, vol. 202, 2023, pp. 28492–28518.