Among us

- Detecting spoofed audio samples in a multi-party conversation -

Project Report

Vigneshwar Anandhamurugan

Aalborg University
Cyber Security

AALBORG UNIVERSITY
STUDENT REPORT

Cyber Security
Aalborg University
http://www.aau.dk

Title:
Among us - Detecting spoofed audio samples in a multi-party conversation

Theme:
Scientific Theme

Project Period:
Fall Semester 2025

Project Group:
1017

Participant(s):
Vigneshwar Anandhamurugan

Supervisor(s):
Ashutosh Dhar Dwivedi
Hannes Künstner

Page Numbers: 42

Date of Completion:
June 3, 2025

Abstract:

As voice technologies become more prevalent, it becomes vital to improve spoofing detection algorithms especially when multiple speakers are involved. This thesis presents a novel pipeline combining speech separation with spoof detection to address the challenge of identifying spoofed audio samples in overlapped speech. The system uses a SOTA spoof detection model which achieves an EER of 2% on ASVspoof 2019 dataset. However, the EER drops to 24% when tested on a custom- generated mixed audio dataset, as a result of modifications of the audio artifacts by the speech separation process. The system when trained on the custom dataset saw an increase in EER to 22% highlighting the importance of domain adaptation for spoof detection in complex acoustic environments and provides a foundation for future research in real-world scenarios.

Contents

Preface

Aalborg University, 2025

[illegible]
01/06/25

Vigneshwar Anandhamurugan
vanand23@student.aau.dk

Abbreviations

A list of the abbreviations used in this report, sorted in alphabetical order:

Abbreviation	Definition
AI	Artificial Intelligence
AUC	Area Under the Curve
AUROC	Area Under the Receiver Operating Characteristic curve
BB	Bonafide-Bonafide
CNN	Convolutional Neural Network
CQCC	Constant Q Cepstral Coefficients
CPU	Central Processing Unit
CSV	Comma-Separated Values
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Network
EER	Equal Error Rate
FBI	Federal Bureau of Investigation
GAN	Generative Adversarial Network
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
kHz	Kilohertz
LCNN	Light Convolutional Neural Network
LR	Learning Rate
MFCC	Mel Frequency Cepstral Coefficients
ML	Machine Learning
RNN	Recurrent Neural Network
SB	Spoof-Bonfide
SOTA	State Of The Art
SS	Spoof-Spoof
STFT	Short-Time Fourier Transform
US	United States
VSDC	Voice Spoofing Detection Corpus (audio dataset)
ZCR	Zero-Crossing Rate

Chapter 1

Introduction

Audio Deepfakes are AI generated or edited/synthesized to create fake audio that seems real [1]. They often use sophisticated machine learning algorithms such as deep neural networks (DNNs), generative adversarial networks (GANs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs)[2]. Despite the different architectures of these machine learning algorithms, they primarily process audio clips to discern unique patterns in a person's voice such as vocabulary, accent, and speech patterns, which are then used to create audio clips that resemble the original voice note.

Audio deepfakes have become a growing concern with the recent boom in AI and AI associated deepfake generating techniques. In the United States, FBI has issued warnings about threat actors using sophisticated AI techniques to mimic the voices of senior US officials. These scammers use artificially crafted voice notes to deceive individuals, into providing personally identifiable information, credentials, and/or access to personal accounts. The impersonations appear highly realistic, making it difficult to discern the fraud [3]. On the other hand, an Argentinian woman was deceived by scammers impersonating George Clooney. The victim was convinced that she was in contact with the "real" actor for over six weeks, eventually transferred approximately £10,000 to the fraudsters. [4].

Despite promising strides in the development of techniques that identify spoofed audio samples, most of the work in this realm has been conducted with static audio clips. Such datasets are compromised of audio clips that last anywhere between 2-10 seconds, with little or no attention given to variations in acoustic background and recording conditions. Furthermore, these clips usually belong to a single person. [5] tests the reproducibility of such models in a real-time scenario, such as a Teams group call, and reported that these models failed to accurately classify intermittently induced spoofed audio streams. This report addresses the issue by identifying the possible cause of this drop in performance and proposes possible solutions to address this research gap. This report is organized in the following manner: Background, Related work, Methodology, Results, Discussion and

Conclusion.

1.1 Problem Statement

With the increasing sophistication of deepfake audio generation, detecting spoofed speech has become a critical challenge, particularly in multiparty conversations where multiple speakers interact dynamically. While research in the domain of audio spoof detection is primarily based on efficiently identifying individual audio clips as spoofed or not, this project will delve into scenarios where multiple speakers are involved. This system primarily aims to deal with observations reported by [5], that traditional audio spoof detection algorithms struggle to accurately predict audio streams, as spoofed or bonafide, in a multi-party environment. The system tries to determine the reason behind the drop in performance of spoof detection algorithms and possible solutions and/or strategies to help improve the detection accuracy. The proposed system will analyze multiparty conversations to identify if manipulated speech forms are present in a conversation, contributing to advances in the domain of audio deepfake detection.

Chapter 2

Background

This section explains the different basic concepts associated, based on which this system is built, and concepts that align closely with the system.

2.1 Methods of generation of spoofed audio

Spoofed audio refers to any kind of manipulated or fabricated speech designed to deceive a target victim or an automated system. Machine Learning algorithms could also be used to generate spoofed audio clips.

2.1.1 Replay Attacks

Replay attacks involve recording a person's voice, which is later used to impersonate the said person to deceive the target victim. Detection of these systems prove quite difficult as these audio clips do not undergo synthesis or modification, and thereby the underlying characteristics of the audio clips are maintained. Autonomous audio spoof detection systems rely on these modified characteristics to differentiate spoofed audio samples from bonafide samples. These characteristics will be discussed in later sections.

Replay attacks are of two different types: single hop and multi hop attacks. The single hop replay attack uses a direct playback of a recorded audio clip while a multi-hop replays audio through multiple devices. The characteristics of the audio clip degrade over several hops and, therefore, become more difficult to detect. ASVspoof 2019[6] compromises of single-hop audio clips and VSDC[7] proposes multi-hop replay attacks.

2.1.2 Speech Synthesis

Speech thesis technique generates artificial speech from text inputs, with the goal of producing realistic sounding audio that mimics a target speaker's voice. Generating speech from text is referred to as Text-to-speech synthesis while modifying the voice's characteristics to resemble an attacker is called voice conversion. To accurately replicate the target speaker, the underlying text-to-speech ML model has to be trained on the person's speech samples to capture vocal nuances. This enables the ML model to accurately adjust the pitch, tone, speaking rate, and accent of the synthesized audio clip.

This technique often uses models such as Tacotron, which uses a Griffin-Lim algorithm over an RNN to generate raw waveforms, and a Multi-SpectroGAN that uses a conditional discriminator to eliminate reconstruction loss between ground truth and the generated mel-spectogram [8]. Audio clips that have undergone similar syntheses form part of the [6] data set. Most spoof detection algorithms work on similar datasets to hone their prediction capabilities, to better ascertain a given input audio as either spoof or bonafide.

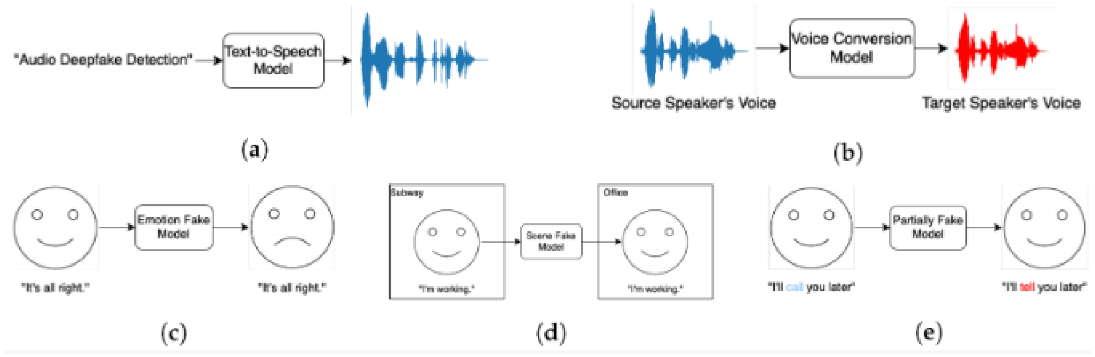

Figure 2.1: Common Spoofing Techniques [9]

2.1.3 Adversarial Attacks

To carry out such attacks, attackers introduce subtle changes to an audio waveform that are imperceptible to deceive human listeners. These modifications can significantly alter the output of deep learning detection models [10]. Attackers could have varying levels of knowledge about the underlying spoof detection model. If the attack has complete knowledge of the underlying ML algorithm, then the attack is termed a white-box attack. If the attacker has little knowledge, then it is a gray-box attack, while if the attacker has zero knowledge, then the attack is called a black-box attack [11].

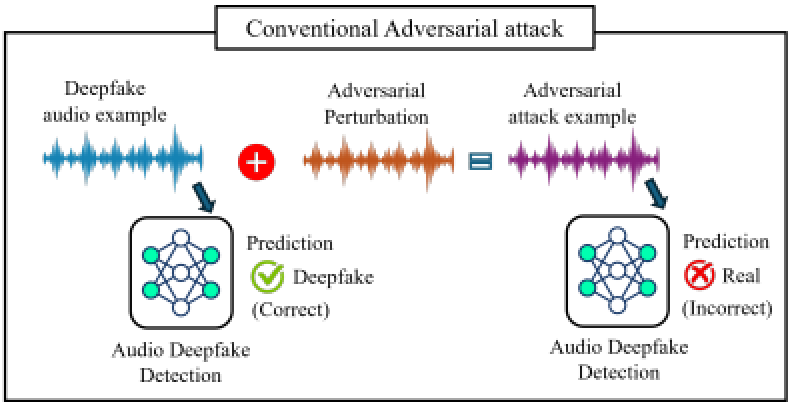

Figure 2.2: Adversarial Attack [11]

Apart from differentiating adversarial attacks on the basis of the knowledge gathered about the underlying ML algorithm, they could also be classified in terms of techniques used:

- Signal Processing: Attackers could manipulate the different characteristics of an audio waveform such as the frequency spectrum or the phase of the audio signal. Time stretching and resampling can also be used to carry out this attack.
- Generative Adversarial Networks (GAN) attacks: in this attack, an audio generator uses a discriminator to align the spoofed sample more in line with the genuine sample [11]

2.2 Evaluation metrics

Evaluation metrics are standardized ways to assess how well an audio spoof detection algorithm performs. They help assess the reliability and accuracy of the system in question.

Equal Error Rate (EER) is the most commonly used evaluation metric in the domain of audio spoof detection. It represents a threshold where the value of the false rejection rate and the value of false acceptance rate coincide. In this context, the false acceptance rate represents the possibility of a spoof audio clip being classified as genuine. On the other hand, false rejection rate is the possibility that a genuine audio clip is falsely classified as spoofed. Spoof detection algorithms with a low EER perform better than their counterparts with a higher EER.

Accuracy signifies the percentage of audio samples correctly classified as being spoofed or genuine. Although accuracy is a common indicator of an ML algorithms' predictions, in the realm of audio spoof detection, EER tends to be a little more important than accuracy. This is the result of the datasets, used in the training of these algorithms, being heavily unbalanced.

F1-score also has a role to play in an audio spoof detection algorithms' performance evaluation. It is the harmonic mean of precision and recall and is useful when dealing with datasets that are inherently biased, as it considers both the value of the false acceptance rate as well as the value of the false rejection rate. The difference between F1-score and EER is that the former has a fixed threshold, i.e. clear positive/negative labels assigned to predictions, whereas the latter deals with a variable threshold and is therefore better suited to deal with audio spoof detection.

Other commonly used evaluation metrics are precision and recall. Precision refers to how often the model's predictions of spoofed audio clips are actually spoofed. On the other hand, recall refers to how often actual spoofed clips are correctly classified as spoofed by audio spoof detection algorithms.

2.3 Audio Spoof Detection

As discussed above, Audio Spoof Detection is the process of classifying any given audio clip as spoofed or bonafide based on the features extracted from the said audio clip. In general, an audio spoof detection pipeline has four components, namely the Raw Audio Input, a Feature Extractor, A Classifier, and the prediction.

2.3.1 Raw Audio Input

The pipeline starts with a raw audio input, which can either be authentic or spoofed using techniques like text-to-speech, voice conversion or adversarial deepfake techniques as discussed previously. This input could either be in the form of a waveform or a digital audio file with extensions '.flac'.

2.3.2 Feature Extractor

Raw audio is fed into a feature extractor, where it is transformed into a meaningful representation that ML algorithms could utilize. These meaningful representations could be in the form of hand-crafted features or learned features. The former are designed by professionals based on established principles in audio signal processing and are usually based on properties such as frequency and/or energy. The latter is automatically learned by deep learning models, such as CNNs and Transformers, directly from the raw audio input. These models tend to be self-supervised. Some common hand-crafted features include:

1. Mel-Frequency Cepstral Coefficients (MFCC) - these help capture short-term frequency and timbral characteristics of speech, which can highlight the differences

between spoofed and genuine audio samples [12].

2. Constant Q Cepstral Coefficients (CQCC) - these provide a time-frequency breakdown with pitch based spacing, making them useful in differentiating subtle cues that might reveal a spoofed artifact [12]

3. Spectrograms - Visual representations of the frequency spectrum over time, often used as visual input to CNNs to detect patterns or anomalies [12]

2.3.3 Classifier

These features that have been extracted above are then passed to a classifier which attempts to learn subtle differences between bonafide and spoofed audio samples. CNNs, ResNet, and Transformers are some of the most popular classifier architectures..

Convolutional Neural Networks (CNNs) are deep learning models that are designed to automatically learn spatial patterns in data. Although they were originally developed for image processing, they are well suited for audio tasks as well. These systems use convolutional filters that slide over the input such as a spectrogram, with each filter learning to detect specific patterns in the audio frequency-time space. While early layers capture simple patterns, deeper layers learn more complex structures making these systems robust to small variations and effective in extracting meaningful features from audio data [13].

Residual Network (ResNet) is a deep CNN architecture introduced by Microsoft that uses 'skip connections' to help train very deep networks. These connections allow the network to pass information across layers without modifications efficiently dealing with vanishing gradient problem. They have better feature extraction compared to CNNs, as they have higher depth. They also have better accuracy compared to standard CNNs [14].

Transformers are deep learning models originally designed for sequence data, but are now commonly used in audio tasks. They use a mechanism called self-attention to understand how different parts of the input relate to each other, no matter how far apart they are in the sequence. Unlike CNNs, transformers use layers of attention to evaluate the importance of each segment, making them effective at long-range patterns and contextual information in audio.

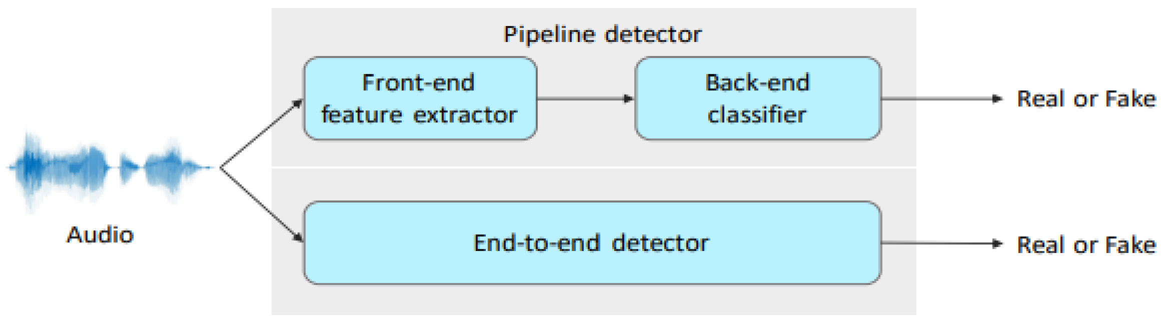

Figure 2.3: Spoof Detection Pipeline [15]

2.3.4 Prediction

The classifier finally outputs a prediction i.e. if the raw audio input is either spoofed or genuine. Typically, audio spoof detection is a binary classification problem. but could also provide confidence scores associated with the prediction or segment-level decisions for an in-depth analysis. This report presents audio spoof detection as a multi-class problem and will be explained in detail in the later sections.

2.4 Speech Separation

Speech separation is the process of isolating individual speakers' voices from a single audio recording where multiple people are talking at the same time. This is often called the 'cocktail party problem'. The system receives a mixed audio signal containing overlapping voices from two or more speakers. The system processes the input and aims to reconstruct the original speech signals for each individual speaker, producing separate audio channels for each.

Similar to Spoof Detection, the raw input undergoes pre-processing, where it is converted into a suitable representation. Most often, raw input is converted into waveforms or spectrograms. The former is a one-dimensional signal that shows how the air pressure changes over time i.e. how loud the sound is at each moment. It is a plot between the amplitude and time. The latter is a visual representation, as discussed above. The input is processed using the Short-Time Fourier Transform(STFT). The raw input is sliced into segments and each of these segments is subjected to STFT. It is a plot between time and frequency [16].

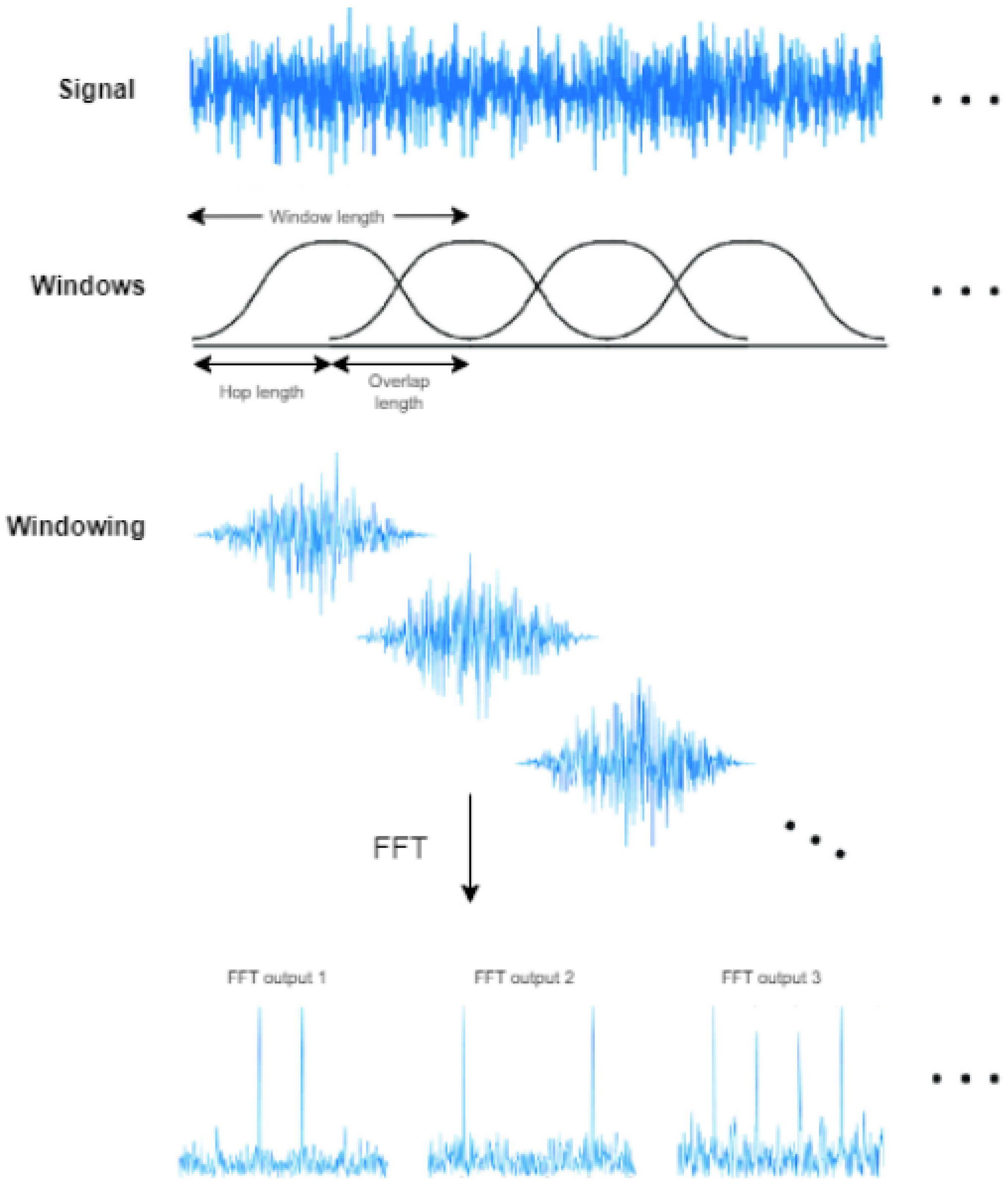

Figure 2.4: Spoof Detection Pipeline [16]

After the preprocessing is complete, the transformed input is fed into a feature extractor. In the case of transformed input being a spectrogram, the model learns to pick up patterns in time-frequency segments, which often carry speaker-speaker artifacts. In the case of transformed input being a waveform, the feature extractor works in a similar fashion by breaking down the waveform into a set of blocks.

Once the features are extracted, the deep learning models try to figure out which parts of the transformed input belong to which speaker. Some techniques used include

- Mask Estimation - the model learns to create a 'mask' for each speaker. These masks indicate which segments of the input belong to which speaker. Applying these masks to the mixed audio allows the system to isolate individual speakers[17].
- Deep Clustering - the model projects time-frequency bins into an embedding space. In this space, bins that belong to the same speaker are grouped together. A clustering algorithm, then assigns each group to a specific speaker, thereby effectively differentiating the individual voices[18].
- Encoder-Separator-Decoder Architecture - these models employ an encoder which converts the raw waveform into a set of learnable segments. The separator module assigns speaker-specific weights to these components, after which the decoder reconstructs the waveform. This approach is commonly used in models such as Conv-TasNet, where a feature extractor step is unnecessary, as the encoder acts as a substitute within the larger separation architecture[18].

Unlike spoof detection models, speech separation models have a post-processing step. In this step, the separated features or masked signals are converted back into clean audio waveforms for each speaker. In the case of the use of spectrograms, the inverse STFT function is used over the segments to reconstruct the audio signal. On the other hand, for time-domain methods, the decoder reconstructs the waveform directly from the separated segments[18].

2.5 Transfer Learning

Transfer learning, a machine learning technique in which a model, initially trained to deal with a source task, is leveraged to deal with a target task. Traditionally, machine learning algorithms are built for each specific application, which would ideally have large amounts of associated data. They would also require substantial computational resources to deal with the size of the dataset. When a situation arises where an ML model has to be trained for a target task but lacks sufficient data or computational resources, Transfer Learning proves useful. A general Transfer Learning pipeline includes 3 steps:

1. Pre-training - In this step, a model is trained on a large, either a general purpose or a task-specific dataset. The model learns patterns and other parameters that could be used to solve a generic task[16].
2. Knowledge Transfer - The learned parameters i.e. the weights, from the pre-trained models are transferred to a new model designed for the target task[16].
3. Adaptation - The transferred model is either used as a feature extractor or fine-tuned using a smaller, task-specific dataset. While the former involves using learned

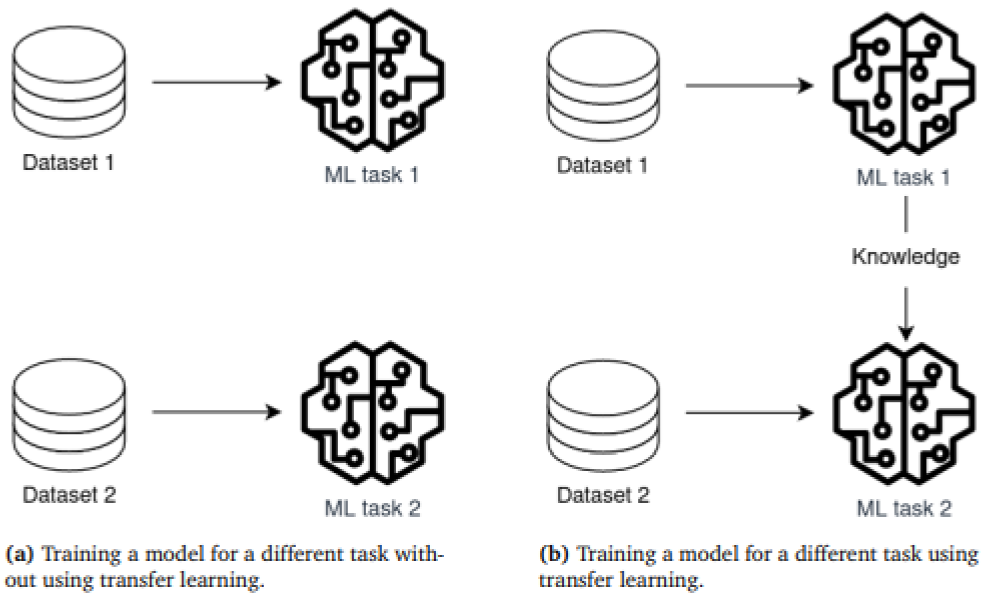
Dataset 1
ML task 1
Dataset 2
ML task 2
Dataset 1
ML task 1
Knowledge
Dataset 2
ML task 2
(a) Training a model for a different task without using transfer learning.
(b) Training a model for a different task using transfer learning.

Figure 2.5: Transfer Learning Usage[16]

representations as input for a new classifier or regressor, the latter allows some or all of the model's parameters to be fine-tuned to better fit the task at hand[16].

Transfer learning proves useful in scenarios where the availability of labeled data for a target task is limited. This is because this technique makes use of existing knowledge to reduce the time taken in training the model for the target task, and reduce the use of computational resources. This technique has resulted in increased accuracy and generalization[19] especially in computer vision, natural language processing, and medical imaging.

However, the effectiveness of this technique depends on the similarity between the original and target tasks. If the domains are too disjoint, the transferred knowledge may not be effective and can even reduce the performance of these models, commonly known as negative transfer[20]. Despite these challenges, transfer learning remains a useful tool for building useful ML pipelines with limited data resources.

Chapter 3

Related Works

This section compromises some of the most closely related scholarly articles and their findings have been summarized.

In one study [5], Arjun Pankajaskhan built a software tool designed to identify deepfake audio during live conversations across different communication platforms. Their approach tested ResNet and LCNN models on the ASVspoof 2019 dataset. Although the models performed well in static or pre-recorded environments, the tool struggled when dealing with the unpredictability of live, real-time calls like those on Microsoft Teams. In addition, they had voluntarily circumvented the 'cocktail party problem' by allotting time frames for each individual speaker in the call. The research highlights the gap between the performance of these models in static environments and in real-time dynamic environments.

Gustav A.P. Bonvang's work[16], focused on applying transfer learning with ResNet50 to detect audio deepfakes in real-world conditions. The model was trained on Mel spectrograms, derived from the In-The-Wild dataset. His results showed impressive gains, achieving almost 97% accuracy and 95% f1 score, outperforming non-transfer baselines by 22% accuracy and 44% f1 score respectively. The study illustrates how transfer learning can cut down training costs and data needs while maintaining strong detection results.

Govind Mittal et al.[21], developed an AI-powered verification system named PITCH, which uses a set of 20 audio-based challenges designed to test speaker authenticity. These include changes in tonal shifts, complexity of spoken phrases, and other variations. The system was tested on a massive dataset of deepfake audio and achieved an AUROC of 89% with machine-only detection, while human evaluators scored 72%. Combining human intuition with machine pre=screening increased the overall accuracy to 85%. The results support the idea that the combination of machine analysis with human judgment can strengthen voice authentication systems in practical scenarios.

A separate study [22], demonstrated how detectors can be manipulated using adversarial examples generated by GANs. These attacks were able to reduce the detection accuracy in both the digital and physical domains. The findings raise concerns about model vulnerability and call for strategies like adversarial training and input preprocessing(transformed representations are less sensitive to subtle, malicious changes) to defend against increasingly sophisticated spoofing methods.

He and Whitehill proposed a thorough survey of recent progress in end-to-end models for multi-speaker automatic speech recognition using single channel (monaural) audio[18]. The survey categorizes models into single-input-single-output and single-input-multiple-output paradigms, examining design trade-offs and recent architectural improvements. While the former deals better with an unknown number of speakers, it requires advanced decoding for complex overlaps, the latter requires fixed number of speakers, and they tend to work better under controlled settings. The study also emphasizes the differences between real-world datasets, such as LibriCSS, and simulated datasets, such as LibriMix. The former has better ecological validity but is more difficult to annotate accurately, while the latter enables better large-scale systematic experimentation but does not capture real-world scenarios accurately. The study highlights key challenges in this domain, such as overlapping speech, limited annotated data, and the difficulty of attributing words to individual speakers without special cues.

Chapter 4

Methodology

This Section provides a detailed explanation on how the system was formulated and adapted to the domain of identifying spoofed audio samples in a multi-party conversation.

4.1 Custom Dataset Creation

This project aims to develop a pipeline that is capable of identifying spoofed audio sources, in a multi-party environment which is often referred to as a cocktail party problem. As discussed in Section 2, the cocktail party problem refers to the challenge of distinguishing and focusing on individual audio sources, potentially spoofed in this context, where multiple speakers are present simultaneously. The primary challenge is the absence of relevant data. While existing resources such as the ASVspoof 2019 dataset consist of a mix of bonafide and spoofed audio samples, they are primarily designed for single-speaker verification tasks. They do not host audio samples from real or simulated multi-party conversations, where overlapping speech and/or interference is present. To address this limitation, this project proposes the creation of a novel dataset in an attempt to replicate the audio characteristics of a multi-party environment. This custom generated dataset includes mixtures of bonafide samples, spoofed samples, and an overlap of bonafide and spoofed audio samples.

To create a dataset of overlapping audio samples with desired characteristics for this project, a custom novel mixing process was implemented to generate 30,000 unique mixed files. The ASVspoof 2019 dataset was parsed, and the spoofed and bonafide audio samples were segregated into two arrays. The process of mixing begins by randomly selecting files which are then subsequently passed to the 'overlap_audio' function. To make the selection truly random, the cryptographically secure Python's 'secrets' module was used. This ensures the dataset isn't biased and is also resistant to patterns that might otherwise skew the learning efficiency.

The ’overlap_audio’ function takes two input audio files- each labeled as either spoof or bonafide and generates a single output audio file. The overlapping technique induces realistic variations that could ideally be expected to take place in the presence of multiple speakers. The goal is to create a training dataset that better reflects the complexity of real-world multi-party conversational scenarios. The function performs the following steps:

1. During the pre-processing step, the audio files are loaded using the librosa package from python, preserving its original sampling rate. While loading, the audio clips are subjected to a check to ensure that the audio files use the same sampling rate, as mismatched rates can lead to distortion or misalignment during mixing.

2. A random delay is introduced to mimic the natural variability of the speaker’s timings in audio calls. Here, one of the audio signals is randomly delayed by up to one second. This is done by padding the chosen audio with silence, introducing a forced delay.

3. Audio samples are then subject to length trimming, where the audio signals are trimmed to the same duration, to avoid mismatches when they are added together. The length of the shortened audio sample is chosen as the length of the resultant mixed audio sample.

4. Following this a gain adjustment is carried out where, each audio sample is scaled by a random factor between 0.7 and 1.3 to emulate variations in loudness between speakers. This difference in loudness could be the result of a difference in quality of microphone used, distance from their respective microphones, and the vocal intensity of the speakers.

5. The two audio samples are then mixed to produce a transformed waveform. This signal is also normalized to prevent clipping thereby maintaining consistent output levels.

6. Finally the output is labeled. A function creates a label for each resultant audio waveform. If the input audio samples are both bonafide, the output audio waveform is given a ’BB’ label. Similarly, if both the audio samples are spoofed, the resultant audio waveform is given the ’SS’ label. If a mixture of two is present, the output audio waveform is given a ’SB’ label regardless of how the audios are mixed.

7. The resultant mixed audio is saved locally in the flac format, similar to the ASVspoof 2019 dataset, with a filename that includes the filenames of the individual audio samples.

The function outputs a mixed audio sample, along with metadata such as the filenames of the individual audio clips, their individual true labels, and the label of the combined audio waveform. This information is logged into a dictionary which is then used to create a single metadata file hosting all the information about all the audio samples generated.

4.2 Evaluation of SOTA spoof detection model on ASVspoof 2019

Before constructing a pipeline and running inference on our custom dataset, the state-of-the-art spoof detection model is tested on the original dataset. This is done to facilitate comparison between the results of this model on the individual audio samples and the audio samples that are a result of the speech separation process.

To perform spoof detection, we use a pre-trained WAVLM-based sequence classification model, fine-tuned on the ASVspoof 2019 dataset. The model, 'abhishtagatya/wavlm-base-960h-asv19-deepfake,' is loaded from the Hugging Face repository from the Hugging Face Transformers framework. The following components have been loaded:

- AutoConfig -this component loads the correct configuration class for a given pre-trained model. It contains information such as the number of transformer layers present, number of attention heads present, type and size of the convolutional feature encoder used, number of output classes, pooling method for aggregating sequence outputs, and dropout rates.
- Wav2Vec2FeatureExtractor - this component is used to convert raw audio waveforms into input features.
- WavLMForSequenceClassification - the component is the classification model itself. This component predicts whether the input audio sample is bonafide or if it has been spoofed.

```
config = AutoConfig.from_pretrained("abhishtagatya/wavlm-base-960h-asv19-
    deepfake")
feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained("abhishtagatya/
    wavlm-base-960h-asv19-deepfake")
spoof_model = WavLMForSequenceClassification.from_pretrained("abhishtagatya/
    wavlm-base-960h-asv19-deepfake", config=config).to(device)
```

Listing 4.1: Loading the spoof detection model

A metadata file pertaining to the evaluation dataset of the ASVspoof 2019 dataset is parsed. For each audio file name that is parsed, an audio path, for the audio associated audio file, is constructed by appending '.flac' to the file name. This path is subsequently fed to the predict spoof function, which utilizes the in-built load_audio function to load the audio sample. The loaded raw waveform is passed to a SepFormer model, which separates the waveform into a two dimensional tensor which is split and stored into two individual arrays. Each of these arrays is passed to the 'Wav2vec2FeatureExtractor' essentially converting the audio signal into a transformed input which the model can process. The 'return_tensors="pt"' attribute is used to ensure that the output is a PyTorch tensor, while the 'Padding=True' attribute is used to handle variable length of the inputs by zero padding them. Finally, the 'torch.no_grad()' function is utilized to disable the gradient tracking, ultimately leading to an increase in the efficiency of the inference process.

```python
# Part 2: Read Metadata from a .txt file

def load_metadata(metadata_file):
    metadata = []
    count = 0
    metadata_count = {}
    with open(metadata_file, "r") as f:
        for line in f:
            count += 1
            parts = line.strip().split(' ')
            filename, _, _, _, _, label = parts
            metadata.append((filename, label))
            if label not in metadata_count:
                metadata_count[label] = 1
            else:
                metadata_count[label] += 1

    print(count, metadata_count)
    return metadata

metadata = load_metadata(metadata_file)
print(metadata[:5])  # Display first 5 entries
```

Listing 4.2: Loading of custom metadata file

```python
def predict_spoof(audio_path):
    audio = load_audio(audio_path)
    inputs = feature_extractor(audio, sampling_rate=16000, return_tensors="pt
    ", padding=True)
    with torch.no_grad():
        logits = spoof_model(**inputs.to(spoof_model.device)).logits
    probs = torch.softmax(logits, dim=-1).cpu().numpy()[0]
    predicted_label = "spoof" if probs[1] > probs[0] else "bonafide"
    return predicted_label, probs
```

Listing 4.3: Prediction function for spoof detection model

The model outputs raw logits for each class, which are subsequently converted into probabilities using the softmax function. Finally, the final labels are assigned based on which class has the higher probability. If the probabilities are equal, then the audio sample is classified as bonafide. Finally, the performance metrics accuracy, AUC score and EER of the inference is calculated for comparison purposes.

4.3 Evaluation of SOTA spoof detection model on the custom dataset

Following the construction of the overlapped audio samples, simulating the cocktail party problem, a testing pipeline is designed, combining a speech separation model and a spoof detection model to evaluate the feasibility of identifying spoofed audio sources in complex audio environments.

The system initally checks for GPU availability using Pytorch and assigns the computation to a CUDA-enabled GPU, for faster inference and better scalability when processing large batches of audio.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Listing 4.4: Choosing CUDA-enabled GPU

To perform spoof detection, 'abhishtagatya/wavlm-base-960h-asv19-deepfake,' is loaded from the Hugging Face repository from the Hugging Face Transformers framework. The model has been loaded similar to the previous step. To handle overlapping speech, a state-of-the-art Speech separation model called the Sepformer has been incorporated from the Speech brain toolkit. This model is pre-trained on the WSJ0-2Mix dataset and has proven to provide high quality separation.

```
from speechbrain.pretrained import SepformerSeparation
model1 = SepformerSeparation.from_hparams("speechbrain/sepformer-wsj02mix")
```

Listing 4.5: Loading Speech Separation Model

The pipeline execution starts with the loading of the metadata files created in association with the mixed audio clips generation. The execution of the pipeline starts with a function iterating over the entire metadata file. The metadata file contains the mixed audio files' names and their associated true labels. The loop is set to 10,000 iterations as a result of a lack of memory and computational resources.

Each mixed audio sample is then passed through the previously loaded sepformer model which separates the mixtures into two speaker channels. The output is a tensor containing 2 clean speech waveforms. These waveforms are subsequently moved to the CPU, where they are individually resampled from 8Khz to 16Khz as the spoof detection model expects a audio sample at 16Khz. Each resampled audio waveform is then fed to a 'predict spoof' function, similar to the function described above, which outputs a label prediction i.e. spoof or bonafide and a probability score for each class. The individual predicted labels are converted into a string to resemble the true labels of the mixed audio clips. In other words, the output of the individual audio waveforms is combined to replicate the 'SS', 'BB', and 'SB' labels.

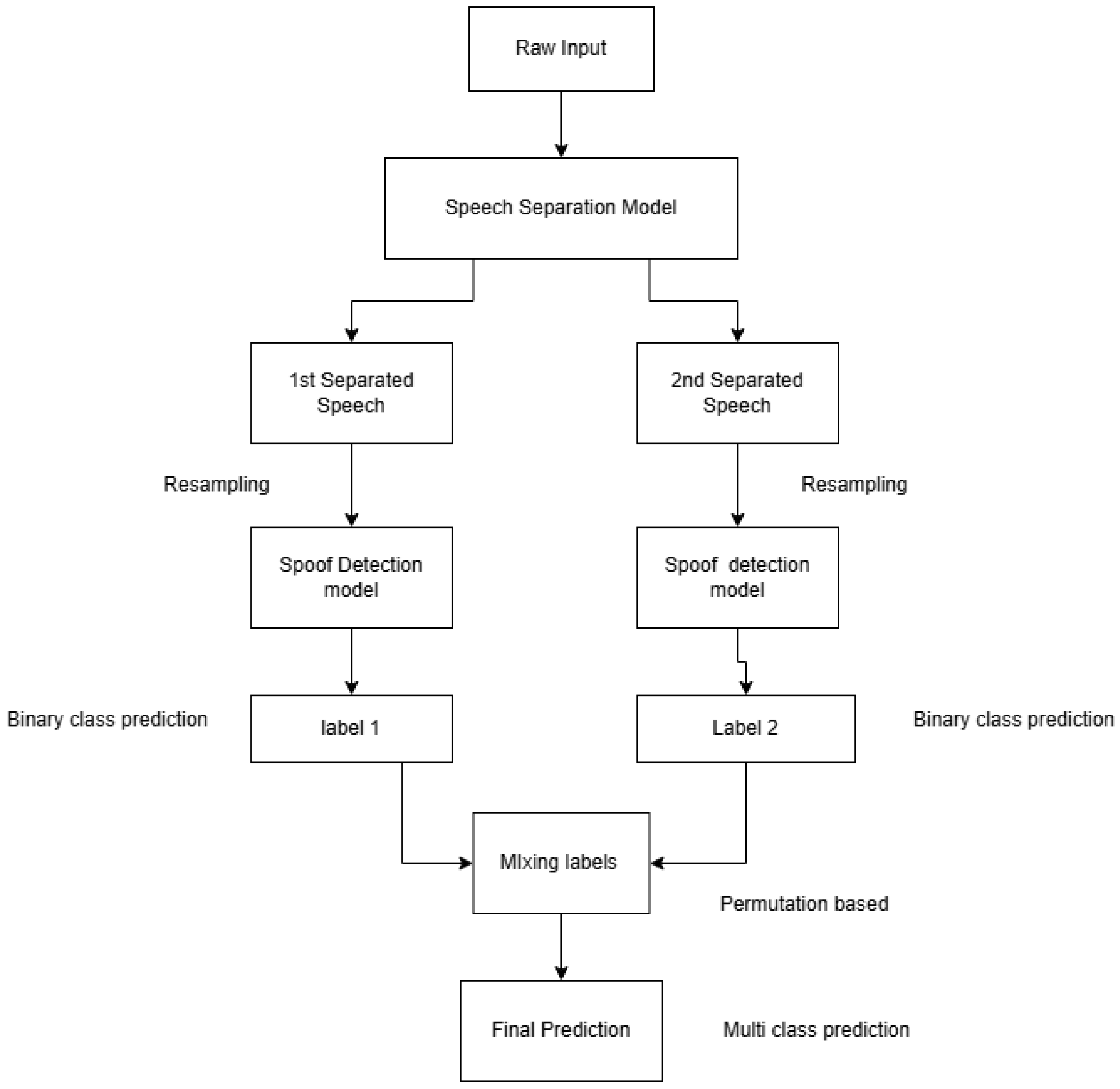

Figure 4.1: Architecture of the developed System

Since the order of the speakers in separation is not guaranteed, the pipeline introduces a permutation-invariant comparison step:

- it calculates the total prediction error using both the original and flipped order of speaker scores
- the permutataion with the lower error is chosen for the evaluation. This ensures a fair assessment regardless of which speaker was assigned to which output channel by the SepFormer.

This was performed to compare the predictions of the individual waveforms with their true values and not just the predictions of the combined audio sample. For efficient memory management and prevention of GPU overflow, temporary tensors and the CUDA memory cache are periodically deleted. At the end of the pipeline, the performance characteristics of the classification of mixed audio samples are calculated as well as the performance characteristics of spoof detection of individual audio waveforms post-speech separation. These evaluation metrics are calculated using the accuracy_score, precision_score, recall_score, and f1_score from the 'sklearn.metrics' package. The simplified architecture of the system developed has been illustrated in Figure 4.1.

4.4 Training of the Pipeline

In this section, the process of training the pipeline constructed in the previous section is discussed in detail. In the dataset creation section, it has been mentioned that only the evaluation dataset of the ASVspoof 2019 was used to create a custom evaluation dataset. For the purpose of training the pipeline, a custom development dataset and a custom training dataset were created from Development dataset and Training dataset from the ASVspoof 2019 dataset respectively. 30,000 mixed audio samples were created as part of the custom training dataset while 20,000 mixed audio samples were created as part of the custom training dataset. While the custom development training data was used as part of the validation part of an epoch, the custom training data was used to initially fine-tune the spoof detection model.

Following the dataset creation, the state-of-the-art spoof detection model and an equally competent speech separation model are used, similar to the process used in the above section. To mimic the training of the original spoof detection model, training parameters are used to train the spoof detection model as part of the pipeline. Here, the speech separation model has been frozen and only the spoof detection model has been targeted for fine-tuning, hence the use of the spoof model's learning parameters. The speech separation model has been frozen in order to facilitate the training of the spoof detection model when the other factors of the system are kept constant. This is to ensure that, the training solely focuses on adapting the spoof detection model to the characteristics of the mixed dataset, preventing potential degradation of separation quality.

The AdamW optimizer, a variant of Adam that decouples weight decay from the gradient update has been used as it has consistently shown to improve the generalization capabilities of transformer-based models. The learning rate, Batch size and gradient accumulation are used in line with the training parameters of the original spoof detection model[23]. The parameters used are

- The learning rate is set to 1e-6 to prevent catastrophic forgetting and ensure gradual adaptation during fine-tuning.
- Batch size is set 1 due to memory constraints from large model size and input length
- Gradient accumulation is set to 2 so that gradients are accumulated over 2 steps before performing an optimizer step, to effectively simulate a larger batch size and stable updates
- Learning rate Scheduler is set to linear, therefore the learning rate will decrease linearly with increase in epoch number.

An epoch starts with the custom training data shuffled and sliced. Due to restrictions in computational availabilities, only 50% of the training data is used i.e. 15,000 custom training data is being used. The random seed of 42 is chosen for reproducibility reasons. The metadata is shuffled using this randomness, to prevent the loop from training on the same portion of the dataset, thereby eliminating redundant training. This increases the generalization capabilities of the pipeline.

```
random.seed(42)
random.shuffle(metadata_train)
metadata_train_subset = metadata_train[:15000]
```

Listing 4.6: Shuffling metadata array

When an audio sample passes through a sepformer, it undergoes separation and a resultant output with two isolated speakers is produced. The output is separated into two individual variables with each undergoing a squeeze function to ensure that the dimensions of the tensor and the input requirements of the spoof detection function match. Similar to the above section, the input waveform undergoes resampling and is converted into input features using the pre-trained Wav2vec2 feature extractor. The transformed input is fed to the spoof detection model with the true label (obtained from the metadata file) and the output loss is computed. Gradients are accumulated over multiple steps to simulate a large batch size, after which the following processes are carried out:

1. clipping of gradients to avoid exploding gradients.

2. modification of model weights by the optimizer
3. change in learning rate scheduler
4. resetting of gradients for the next batch

To prevent the system hosted on Google Colab from crashing, intermediate variables are deleted after the processing of each individual audio clip and CUDA memory is explicitly cleared every 100 files.

After each epoch, the model is evaluated on a held-out development set to measure the model's generalization capability during training. Due to restrictions in the availability of computational resources, a fixed-size subset of 10,000 audio samples is selected after the custom development metadata array is shuffled. This is to ensure an unbiased evaluation. Each audio file name in the metadata file is joined with an appropriate path that fetches the mixed audio sample from the custom development dataset. This audio sample is then fed into a SepFormer model which subsequently outputs two individual audio waveforms. Since the order of separated speakers is not guaranteed, all possible speaker-label alignments are evaluated. The permutation with lesser loss value is chosen as the final prediction. This ensures that label assignments aren't carried out random and/or mistakenly flipped. After determining the optimal permutation, the spoof detection model predicts labels for both audio samples, which are then compared to their true values. Classification losses are accumulated to compute the average loss per prediction.

Due to time constraints on how long a colab program could be run, the optimized parameters of the pipeline are saved, so they could be utilized in subsequent epochs. The code below is used to save the trained parameters. To enable fetching of the spoof model and feature_extractor components for training, the spoof model and feature_extractor components are saved. In addition, the current states of the optimizer and learning rate schedulers are saved to enable continuity in the training regime.

```python
from transformers import AutoConfig

save_path = "/content/drive/My Drive/My notes - cyber/Semester 4/
    finetuned_spoof_model6"

# Save model, feature extractor, and config
spoof_model.save_pretrained(save_path)
feature_extractor.save_pretrained(save_path)
spoof_model.config.save_pretrained(save_path)


# Save optimizer and scheduler
torch.save(optimizer.state_dict(), os.path.join(save_path, "optimizer.pt"))
torch.save(lr_scheduler.state_dict(), os.path.join(save_path, "scheduler.pt")
    )

print(f"Model, feature extractor, and config saved to {save_path}")
```

Listing 4.7: Saving learned parameters

Finally, to make the performance metrics of the epoch available, they are summed together as part of the metrics_history array and are then subsequently saved. This enables us to visualize the training metrics of the pipeline over several epochs.

```python
metrics_history = {
    "epoch": [],
    "train_loss": [],
    "val_loss": [],
    "val_accuracy": []
    # Add more if needed
}
metrics_history["epoch"].append(epoch + 1)
metrics_history["train_loss"].append(total_loss)
metrics_history["val_loss"].append(val_loss)
metrics_history["val_accuracy"].append(val_acc)
import os
import pandas as pd

metrics_file = "/content/drive/My Drive/My notes - cyber/Semester 4/
    training_metrics.csv"

# Check if file exists and load old metrics
if os.path.exists(metrics_file):
    existing_df = pd.read_csv(metrics_file)
    combined_df = pd.concat([existing_df, pd.DataFrame(metrics_history)],
    ignore_index=True)
else:
    combined_df = pd.DataFrame(metrics_history)

# Save the updated file
combined_df.to_csv(metrics_file, index=False)
```

Listing 4.8: Saving performance metrics

For subsequent epochs, majority of the execution remains the same except for the loading of the spoof detection model and its parameters. Unlike the first epoch, where the model and its parameters are fetched from Hugging Face, the rest of the epochs utilize the parameters saved during the previous epochs. The parameters are loaded using the following code:

```python
# Load optimizer and scheduler
optimizer.load_state_dict(torch.load(os.path.join(saved_model_path, "
    optimizer.pt")))
```

```python
lr_scheduler.load_state_dict(torch.load(os.path.join(saved_model_path, "
    scheduler.pt")))

learning_rate = 3e-4
batch_size = 1
num_epochs = 4
gradient_accumulation_steps = 2
```

Listing 4.9: Loading saved paramters from previous epochs

The learning rate was increased in the final epoch alone, to speed up the learning process. This was a result of the lack of resources to carry out more epochs. Similar to the parameters, the model in itself was loaded from a locally saved instance.

```python
# Check if CUDA is available for using GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

from speechbrain.pretrained import SepformerSeparation
model1 = SepformerSeparation.from_hparams(
  "speechbrain/sepformer-wsj02mix"
)
saved_model_path = "/content/drive/My Drive/My notes - cyber/Semester 4/
    finetuned_spoof_model5"

config = AutoConfig.from_pretrained(saved_model_path)
feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained(saved_model_path
    )
spoof_model = WavLMForSequenceClassification.from_pretrained(saved_model_path
    , config=config).to(device)
spoof_model.train()

spoof_model = spoof_model.to(device)
```

Listing 4.10: Loading saved model

4.5 Evaluation of the Final Trained Pipeline

Once training of the pipeline is complete, the trained pipeline is evaluated against the custom evaluation dataset. This is performed to calculate performance characteristics between the trained and pre-trained pipeline. To ensure an unbiased comparison, the same custom dataset has been used in both inference processes.

Although most of the pipeline characteristics were similar, the trained pipeline has small changes to how the models were loaded. While the speech separation model was loaded exactly the same way, it had been loaded during the inference of the pretrained pipeline, the loading of the spoof detection model differed. Instead of the spoof detection model being loaded directly from Hugging face, the newly fine-tuned spoof detection model stored in the Google Drive was put to use. The code used to load this custom fine-tuned model is presented below:

```python
# Check if CUDA is available for using GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

saved_model_path = "/content/drive/My Drive/My notes - cyber/Semester 4/
    finetuned_spoof_model6"

config = AutoConfig.from_pretrained(saved_model_path)
feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained(saved_model_path
    )
spoof_model = WavLMForSequenceClassification.from_pretrained(saved_model_path
    , config=config).to(device)
spoof_model.eval()

spoof_model = spoof_model.to(device)
```

Listing 4.11: Loading final trained model

Similar to the pre-trained pipeline's execution, the system checks for the availability of a GPU. If a CUDA enabled GPU exists, all of the heavy computation is assigned to this device. The rest of the pipeline follows the traditional practice of loading the associated metadata file and iterating through it. Each file name encountered is appended to a local path to generate the associated local path of the mixed audio sample.

```python
audio_path = os.path.join(audio_folder, filename)
```

Listing 4.12: Generating path for mixed audio files

This path is then used to fetch the mixed audio sample, which is fed to the SepFormer, effectively resulting in a two-dimensional tensor. Each tensor is unsqueezed, to resemble the input requirements of the spoof detection model, which transforms the same into processable units. These processable units are used by the spoof detection model to provide predictions of the individual audio sample. The predictions in the form of raw output logits are converted into human-readable probabilities, which are used to determine the predicted labels of the individual as well as the mixed audio sample. These labels are used to calculate the performance characteristics, as mentioned in the above subsection, for comparison purposes.

4.6 Ethical Considerations

Despite the role this spoof detection pipeline plays in multiparty spoofed scenarios, developing and testing such a pipeline also gives room for certain ethical issues that have to be addressed.

It is a prerequisite that the speech data, even if publicly available, must be governed by stringent privacy standards. This is to ensure that the unauthorized use of voice recordings, even in a synthetic or an anonymized form is reduced. This prevents possible infringement of privacy rights. All data sets used in this project originate from thoroughly vetted sources, who have also provided consent for use in research environments. This research does not collect any kind of personally identifiable information.

Automated systems have the tendency of inheriting biases from the data, they have been trained on. In the case of spoof detection models, the system could learn biases from the dataset, such as familiarity with certain accents, languages or speaker demographics. This could result in catastrophic outcomes with the system misclassifying certain groups more often. While this work, does not explicitly deal with these biases, maximum effort has been undertaken to create an even and unbiased custom dataset from the original ASVspoof 2019 dataset to avoid this misclassification problem.

As synthetic and deepfake speech technologies evolve, the boundaries between legitimate and malicious use blur. While building systems for the purpose of spoof detection isn't ethically wrong, the irresponsible deployment of them is. Often times, adversaries learn from existing spoof detection mechanisms, to further improve the spoofing techniques. To prevent such a scenario from occurring, the built pipeline has not been shared irresponsibly.

As a cyber security researcher, it is of utmost importance that the research is conducted ethically and it does not further endanger the existing digital space. The actions, mentioned above, were undertaken consciously to create a safer research environment and to ensure that the research does not, in any way, impede GDPR regulations or unwritten ethical rules.

Chapter 5

Results

The results obtained from the experiments in section chapter 4 is discussed in detail in this section.

5.1 Testing SOTA on the original dataset

Prior to the implementation of the spoof detection algorithm, the state-of-the-art spoof detection model was tested on the original ASVspoof 2019 for the sake of comparison between the model's performance across this dataset and the custom-generated evaluation dataset. The model had an impeccable accuracy at 0.9831, a precision of 0.9829, a recall of 0.9983, and an f1-score of 0.9905. The high accuracy indicates that the model correctly classified almost 99% of the data, showing a strong performance. The precision means that when the model predicts a sample as spoofed, then the sample actually belongs to the spoofed class. Higher recall means that the model minimizes false negatives. The confusion matrix of the spoof detection algorithm is visualized below:

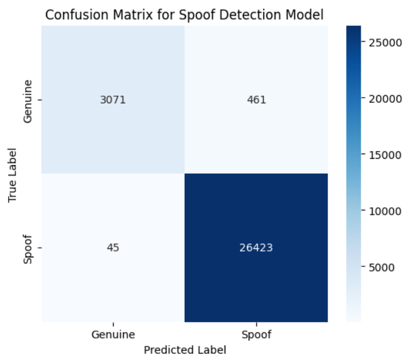

Figure 5.1: Confusion matrix of SOTA spoof detection model on ASVspoof 2019

The model has an EER of 0.0157 which is exceptionally good. The EER represents the equilibrium point between the acceptance of false positives and false negatives. This low eer indicates that the model achieves a very low balance error between misclassifying spoofed audio samples as genuine or genuine samples as spoofed. This EER confirms the model's reliable performance in spoof detection.

5.2 Evaluation of Pre-trained Pipeline

To assess the performance of the pre-trained pipeline that incorporates the state-of-the-art spoof detection and speech separation models, we evaluated it on a custom-generated evaluation dataset. As discussed in Section chapter 4, this custom dataset compromises of overlapping speeches from, either two spoofed audio sources, two bonafide sources or a mix of one spoofed and one bonafide source. These overlapped audio samples were appropriately named 'SS,"SB,' and 'BB.' The classification system attempted to label each input mixed into one of these classes.

Class	Precision	Recall	F1-score	Support
SS	0.35	0.99	0.52	2535
SB	0.50	0.19	0.27	4979
BB	0.93	0.35	0.50	2486

Table 5.1: Classification metrics of the pre-trained pipeline

A class-wise analysis has been included below

- SS (spoof-spoof) - This class has a precision of only 0.35, thereby indicating that of all the samples predicted as SS, only 35% of them actually had SS as a true label. A recall of 0.99 suggests that nearly all audio samples of the SS class were correctly identified, hence the model is highly inclusive. This recall could be misleading as many of those predictions could be false positives. The f1 score suggests that while SS detection is inclusive, its precision is lacking.
- SB (spoof-bonafide) - This class has a precision of 0.50, so half of the samples predicted as SB were correct. The model has a recall of just 0.19, i.e. only 19% of the actual audio samples belonging to the SB class were correctly identified, indicating a major under-recognition of this class, i.e the model is selective. Finally its f1 score of 0.27 signifies difficulty in handling heterogenous cases.
- BB (bonafide-bonafide) - This class has a higher precision, compared to the other classes, at 0.93. This signifies that the model performs exceptional in avoiding false positives. But the model suffers with a 0.35 recall, which means that it misses most of the actual BB samples.

Looking at the bigger picture, the model has an overall accuracy of 42.98%. A random prediction for a model with 3 output classes is at 33.3%, so for the 10,000 samples, this model performs just better than a model that randomly predicts. The overall weighted precision, weighted recall, and weighted F1-score of the model is a mere 0.59, 0.51 and 0.43 respectively. These values take into account the support variable, which indicates the number of instances for each class. The lower weighted f1 score suggests that the high misclassification of the more prevalent SB class significantly drags down the average.

To provide further insight, a confusion matrix was designed based on the classification performance across the three mixed-label categories. The confusion matrix further strengthens the inference that the model has a higher tendency to classify audio as SS. They also underscore the necessity of fine-tuning the model on the specific structure of the custom dataset.

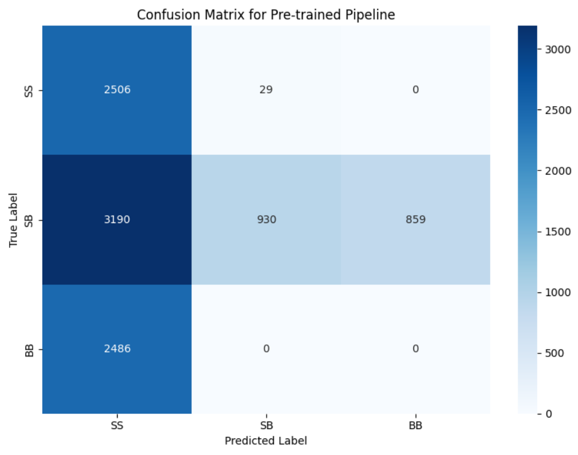

Figure 5.2: Confusion matrix of pre-trained pipeline - multiclassification task

Apart from evaluating the model on the mixed label classification task, we assessed its performance on the individual audio streams as well. This evaluation reflects the models binary classification capabilities as the resultant audio stream could either be classified as spoofed or bonafide only. The model had an accuracy of 67.92%, a precision of 61.06%, a recall of 99.04%, and an f1 score of 75.62%. The high recall means that the model correctly identified nearly all spoofed audio samples, demonstrating extreme sensitivity towards cues associated with spoofed speeches. The precision is noticeably lower, indicating that a significant number of bonafide audio samples were misclassified as spoof. The f1 score balances the trade-off between the above two metrics, indicating that the model performs decently on the binary classification task. The Equal Error rate of the system was poor 23.90% at a threshold of 0.9999 suggesting that the model is heavily skewed toward avoiding spoof acceptance. Ideally, systems deployed for spoof detection have an EER of less than 10%. Therefore, this system is not suitable for deployment yet and requires further fine tuning.

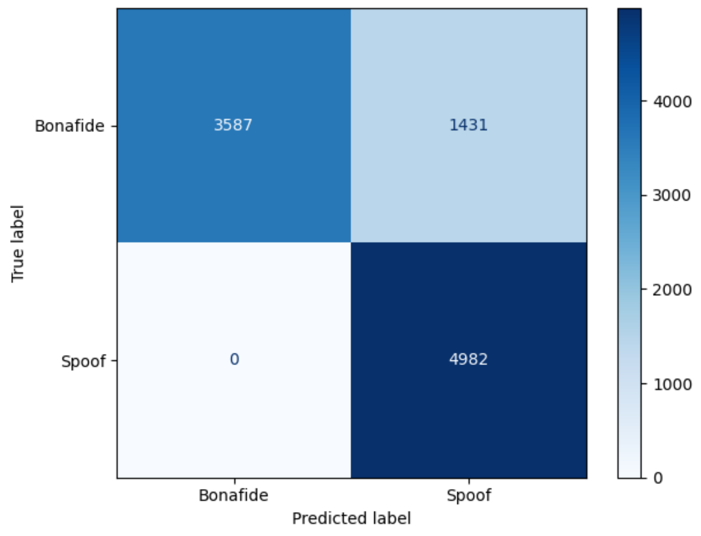

Figure 5.3: Confusion of Pretained pipeline - binary classification

The model seems to correctly predict spoofed audio samples at the expense of misclassifying legitimate audio samples as spoof. Although most models in the high-stake audio verification follow a conservative approach similar to this pipeline, the pipeline still seems to massively under-perform. The discrepancy in performance metrics between multi-class, and binary class classification indicates that while the spoof detection model performs average, an error is post-separation label assignment.

5.3 Train pipeline

In an attempt to increase the efficiency and the capability of the pipeline, the pipeline was trained on the custom generated mixed train and mixed development data. As discussed in Section chapter 4, the speech separation was frozen and only the parameters associated with the spoof detection model were fine tuned. For a total of 6 epochs, the models were trained over 15,000 mixed audio samples. The model was then evaluated over 10,000 mixed audio samples from the custom-generated development dataset.

The average training loss has evidently been decreasing across the epochs, indicating that the model has inferred patterns and is effectively learning and converging. The validation loss improved quickly during the first two epochs and plateaued after epoch the third epoch. This suggests that the model has learned generalizable patterns early on. In an attempt to push the model towards betterment, the learning rate was modified for the last epoch from 1e-6 to 3e-4. The validation loss increased infinitesimally but so did the validation accuracy. This analysis supports that the spoof detection model trained on separated audio streams was able to learn meaningful patterns that ultimately helped the

Epoch	Cumulative Train Loss	Average Train Loss	Validation loss	Validation accuracy
1	7731.96	0.5155	0.4292	0.8028
2	5657.80	0.3772	0.4028	0.8195
3	5120.00	0.3413	0.3836	0.8280
4	4750.49	0.3167	0.3826	0.8292
5	4626.12	0.3084	0.3826	0.8292
6	4915.02	0.3277	0.3831	0.8300

Table 5.2: Training metrics of the pipeline

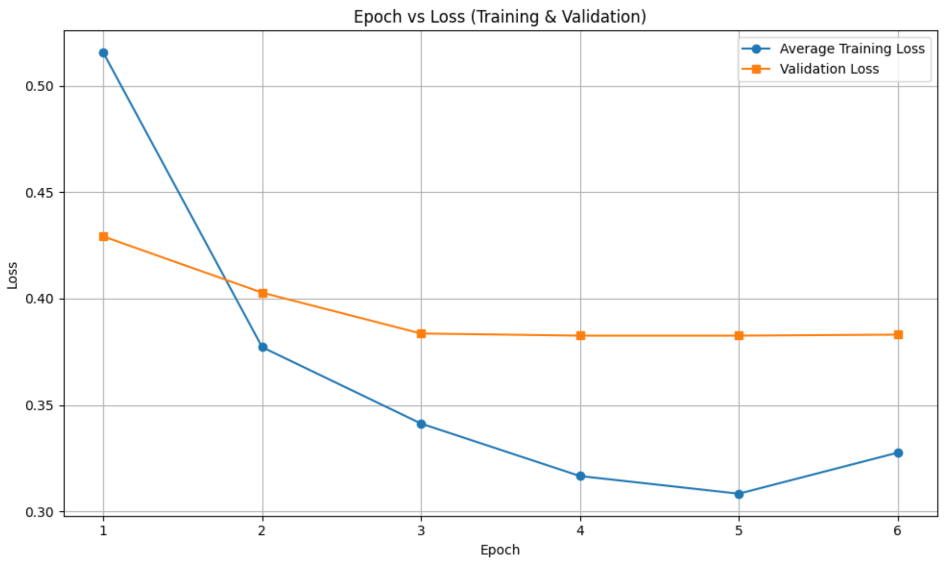

Figure 5.4: Training loss vs validation loss

model generalize better on the validation data.

5.4 Trained pipeline

Following the horrendous results of the pre-trained pipeline, the spoof detection pipeline was trained. To be more precise, the spoof detection model was fine tuned while the speech separation model was frozen. This trained pipeline was subsequently evaluated on the custom dataset compromising of 10,000 mixed audio samples. The evaluation was performed similar to the how the pre-trained pipeline was evaluated.

Class wise analysis says that:

- SS (spoof-spoof) - the recall of the trained pipeline dropped from 99% to a mere 45% while the precision and f1 have increased slightly with the metrics being 55% and 49% respectively. This drop in recall suggests that the model is not skewed towards

Class	Precision	Recall	F1-score	Support
SS	0.55	0.45	0.49	2535
SB	0.63	0.38	0.47	4979
BB	0.47	0.93	0.62	2486

Table 5.3: Classification Report of Trained pipeline

flagging everything as spoofed and a balance in predictions across classes have been attained.

- SB (spoof-bonafide) - The model attained an improvement in precision. The precision rose from 50% to 63% between pre-trained and trained pipelines. The recall also seems have to risen from 19% to 38%. The model's f1-score has also risen from 27% to a 47% indicating the model's fine tuning has proven useful. The model has learned better at predicting the instances from the SB class, which is more representative of the real-world cocktail party scenarios.
- BB (bonafide-bonafide) - The model had an increase across the recall and f1-score characteristics with the recall increasing from 35% to an impeccable 93% while the f1-score increasing from 50% to 62%. The fine-tuning of the model has improved it's ability to better recognize audio samples belonging to the BB class, and has also learned to avoid false positives on the spoof samples.

To provide further insights, a confusion matrix was formulated for the trained pipeline, similar to the pre-trained pipeline. The confusion matrix reasserts the previous inferences showing that the model shows a strong ability to correctly identify bonafide samples and that the SS class' precision and recall also seem to have improved.

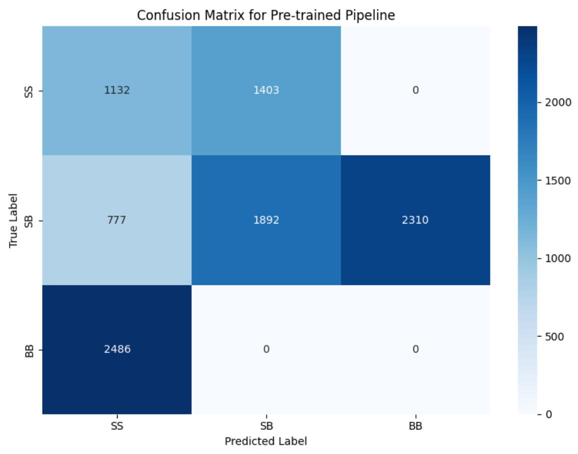

Figure 5.5: Confusion matrix of trained pipeline - multiclassification

Apart from evaluating the model's performance on the mixed label classification task, the model's performance on individual audio streams retrieved after speech separation was also assessed. This analysis provides a clearer view of the underlying binary classification task in a different setting. The model had an accuracy of 74.41%, precision of 84.72%, recall of 59.87% and an F1-score of 70.16%. The model's high precision indicates that the model rarely raises a false alarm i.e. bonafide speech is likely to be misclassified as spoofed speech. The recall value has dropped miles in comparison to the pre-trained, indicating that the model's ability to identify all spoofed samples as spoof has reduced. An EER of 21.98% indicates that the model seems to have marginally better performance compared to the pre-trained pipeline. It indicates that at a threshold of 0.1935, the model misclassifies a spoofed sample as bonafide or vice versa, 21.98% of the time. The extremely low decision threshold implies that the model classifies inputs as spoof only if it is highly confident.

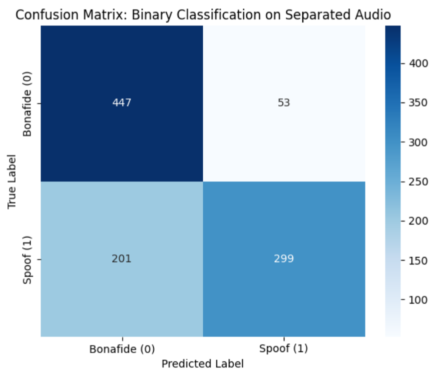

Figure 5.6: Confusion matrix of trained pipeline - binary classification

The discrepancy in performance metrics between multi class and binary class classification indicates that while the spoof detection model performs average, an error is post-separation label assignment is present.

Chapter 6

Discussion

This section discusses the impact of the research, the limitations faced during the study and the possible future works that can be done to capitalize on the results produced in this study.

6.1 Evaluation of SOTA spoof detection model on ASVspoof 2019 vs Custom dataset

In this study, it can be observed that the spoof's detection model's performance underwent a noticeable drop when transitioning from evaluating individual audio samples to mixed samples. When tested individually on the ASVspoof 2019 dataset, the model had an impeccable performance with an accuracy of 98% and an EER of 1.15%. These results clearly indicate the the model is only suited for clean, controlled and well-structured spoof detection tasks where the samples do not have any kind of interference from other speakers and acoustic condition. In comparison the spoof detection model had an accuracy of 42.98% over mixed audio samples and an accuracy of 67.92% on individual audio streams that were a result of mixed audio stream undergoing speech separation. In addition, the spoof detection model had an EER of 23.9%. These results signify that the model underwent a clear degradation in performance and the associated decision boundary sharpness compared to its counterpart. This degradation dropped further when the predictions were mixed to form the final mixed label prediction where the accuracy of the model dropped to a stooping 42.98%.

Most of these performance drops could be attributed to the speech separation process. When an audio sample undergoes speech separation, the temporal-spectral features of the audio sample are modified. The Zero-Crossing Rate(ZCR), which measures how often a signal crosses zero amplitude, is altered. Voiced speech tend to have a low ZCR, while Unvoiced Speech have high ZCR[24]. When an audio sample undergoes speech separation, high ZCR regions are suppressed, which causes erasure of unvoiced spoofing artifacts. Certain spoof detection algorithms use Average ZCR as a cue for detecting spoofed audio samples[25]. So this erasure of key spoofing artifacts could have been the reason for the drop in performance of the spoof detection problem. In addition, it has been found that most queues for spoof detection lie in the range of 0 and 4kHz in normal voiced speech and in the range of 4-8khz in quiet parts or unvoiced sounds[26]. [27] quotes that their proposed speech separation method retains dominant peaks but the modulation effects in the frequency range of 50-200Hz are excluded which leads to a residual 'buzz' artifact. Since spoof detection also depends on these artifacts, the removal of the same affects its prediction abilities. Apart from these interferences, the audio sample also undergoes a resampling process as the audio sample that is output from the SepFormer is of a different sampling rate and therefore cannot be fed into the spoof detection model. This resampling process could have introduced artifacts that ultimately hinder the spoof detection model's capabilities.

As the ASVspoof 2019 dataset is designed specifically with spoofed samples containing spoofing artifacts that are a result of known spoofing techniques, a model trained on this dataset struggles to generalize on a mixed dataset with overlapped speech samples because of the elimination or inclusion of spoofing artifacts caused either by the mixing of speech samples and/or separation of mixed audio samples. The difference in results between the labeling of the overall mixed audio sample and the labeling of individual audio streams post-separation is a result of one of the speech samples being identified incorrectly, out of the two audio clips produced by the SepFormer. This inherently lowers the accuracy of the final pipeline.

6.2 Evaluation of Pre-trained vs Trained pipeline

To enhance the performance characteristics of the pre-trained model, the pipeline was fine-tuned to the custom-generated mixed dataset. The trained pipeline did witness an increase in performance as compared to its counterpart. This signifies that the training did help the model learn measurable and meaningful representations that ultimately boosted the pipeline's performance. While the overall gains appear moderate, they highlight important aspects of domain adaptation and robustness of speech-based ML systems.

While the f1 score for the SS class slightly declined for the trained pipeline in comparison to the pretrained pipeline, the f1 scores associated with the SB and BB class showed an improvement. The SB class showed the largest increase, indicating that the model learned key artifacts that help deal with inter-class variations introduced by the overlap algorithm. The binary classification also showed an improvement with the accuracy increasing by almost 7% in comparison to the pre-trained pipeline. There was an increase in precision but a decrease in recall, which ultimately led to a reduced EER.

This increase in performance could be attributed to the consistency of spoofing artifacts introduced in the speech separation process. As the speech separation model, SepFormer, was frozen, the separation artifacts introduced across the dataset would be consistent. The pipeline was able to grasp on to these consistent artifacts as noise in the classification task. These artifacts are considered noise, as they are predictable and are present across all the audio samples regardless of their true labels being spoof or bonafide. The possibility that the spoof model also adapted to the suppressed low-frequency artifacts is possible. The trained model could also recalibrate its ability to detect noise residuals that are a common modification to audio samples introduced by the speech separation process.

The reduction in EER by 2%, an increase in the binary classification's accuracy by 7% and an increase in multi classification's accuracy by 11% shows promising potential towards the development of a better system, which is better equipped in handling the cocktail party problem with spoofed audio samples.

6.3 Limitations

In the process of conducting this research, the system faced certain logistical and technical limitations. The primary challenge being the lack of time in furthering the research. Training over 15,000 data items and the model's validation over 10,000 data items over 6 epochs consumed $\tilde{1}00$ hours. Besides the running time of the pipeline, analysis of the artifacts induced by speech separation that affect spoof detection is also heavily time consuming. Although the research indicates that these artifacts induce a noticeable performance reduction of the integrated spoof detection model, due to a lack of time, the research couldn't delve into which of these artifacts play a bigger role in performance reduction and by how much. Despite the use of A100 GPUs, an extremely powerful hardware, the training of the model took over 100 hours. In addition to these, inference on the evaluation dataset was also time consuming.

Due to hardware limitations and processing constraints, the batch size was reduced to just 1 which increases the risk of overfitting and may lead to poor generalization to unseen data. In addition, despite the generation of 30,000 training data and 20,000 validation data, only 15,000 training data and 10,000 validation were used during the training process. While 30,000 in itself isn't enough for the training process, further slicing of the dataset hindered the training process. Furthermore, the custom dataset generation was conducted solely based on the LA subset of the ASVspoof 2019 dataset. This could result in the pipeline overfitting to a specific type of spoofing attack, thus reducing it's generalization to other spoofing attacks. Due to hardware limitations, training of the entire pipeline was not conducted. In this research, the speech separation pipeline had to be frozen to save compute resources. While necessary, this technique has restricted the end-to-end learning of the pipeline where the separation model could have worked hand-in-hand with the spoof detection model to improve the overall performance of the pipeline.

Another challenge faced by the pipeline is the adaptation of EER over a multi classification task. In the domain of spoof detection, EER plays a vital role in the evaluating the performance of the system. While most systems use EER over a binary classification task, this research attempted at adapting the EER over a multi classification task and failed. Although EER was still used in the system to evaluate its performance, EER was calculated based on the model's ability to ascertain a label to the audio samples that were derived from the speech separation task and not on the multi classification task of predicting if the audio sample belonged to the 'SS,''SB,' and 'BB.'

6.4 Future Works

This research while demonstrating the promise and limitation of the current setup, in an overlapped speech scenario, it also opens up opportunities for advancements in the fine-tuning of the system as well as certain other explorable avenues.

6.4.1 Fine-tuning the Speech Separation model

As discussed in the the previous section, the research presents with an opportunity to hone the speech separation model, in addition to the spoof detection model. In this research, the speech separation system was frozen to reduce complexity of the implemented system, requirement of computational resources and to isolate the fine-tuning of the spoof detection model. However, future studies could investigate end-to-end learning, where both modules are trained together simultaneously. This may allow the separation model to become spoof-aware, learning to retain discriminative artifacts necessary for spoof detection.

6.4.2 Increase in speakers and Integration of Speaker diarization

The current pipeline assumes only two speakers with binary spoof labels to be present in the custom generated dataset. Future work could be done on increasing the number of speakers involved in the cocktail party problem and investigating how an increase in speaker audio samples could affect the system's prediction capabilities. The system also works only on identifying the number of spoofed samples i.e. one or two spoofed samples. Future work could integrate a speech diarization model that would help the system attribute the labels to its specific speakers, which could prove useful in security contexts.

6.4.3 Dataset Generation

The system to its best capability avoided including inherent bias during dataset generation. Although cryptographic randomness was introduced to alleviate inclinations of dataset towards a specific class, the final dataset generated was not explored for the existence of possible prejudices. A promising direction is to investigate a better technology or process with recent and more diverse spoofing methods, including real-time recordings and AI-generated deepfakes to test the robustness of the system in more adversarial conditions. The custom generated mixed audio samples also lacks replication of real-world acoustic variations. Real-world variations such as reverberation, environmental noise, inconsistencies with the microphone in use or adversarial attacks could provide additional insights that the system could adapt to, making it relevant to the requirements of the deployable system.

6.4.4 Latency of the pipeline

The goal of the project was limited to just exploring the relevance of state-of-the-art spoof detection algorithms, in a overlapped audio conversation scenario which are representative of group calls in platforms such as Teams and Discord, and podcasts that could be subject to spoofing attacks. The system does not delve into increasing the efficiency of the training and inference processes. The system clearly required excessive time periods during the training and inference process, which makes it not so suitable for real-time spoof detection deployments. Future work could explore methods and techniques that help optimize the inference algorithm and provide low-latency models that are better suited for real-world situations.

Chapter 7

Conclusion

This chapter summarizes the experiments carried out in this research, the scope of the research, and the results produced by the research.

The research began with the goal of analyzing the performance of spoof detection model, which was trained on the simple ASVspoof 2019 dataset with individual audio samples, across a custom generated dataset which was more representative of a cocktail party problem with spoofed audio samples. The spoof detection model was initially tested on the ASVspoof 2019 dataset. The model performed exceptionally well with an accuracy of 98% and an EER of 1.15%. The model was later integrated into a pipeline with a speech separation model, SepFormer, which helped split the mixed audio samples. The resultant audio clips were tested for the spoofing artifacts by the spoof detection model. The untrained pipeline had an accuracy of 43% and an EER of 23%. In an attempt, the pipeline was fine-tuned on the custom generated dataset, with the speech separation model kept frozen. The fine-tuned pipeline was evaluated on the custom-generated evaluation dataset and the model returned an accuracy of 53%.

The research clearly indicates a drop in the spoof detection model's performance as a result of the introduction and erasure of key artifacts by the speech separation model. The report also suggests that further work could be done to identify which of these artifacts play a more important role in the performance of the pipeline. The report also emphasizes the need for additional time and hardware resources for furthering the research in this domain. The report also necessitates the need for a speech diarization model that could attribute which of the speakers in the conversation are spoofed.

The report concludes that, although spoof detection models with excellent prediction capabilities exist, these models struggle in a scenario that includes multiple speakers. This task requires the need for a domain-specific pipeline, such as the one implemented in the scenario, to deal better with the cocktail party problem with possible spoofed speakers.

Bibliography

[1] Zahra Khanjani, Gabrielle Watson, and Vandana P. Janeja. "Audio deepfakes: A survey". In: *Frontiers in Big Data* Volume 5 - 2022 (2023). ISSN: 2624-909X. DOI: 10.3389/fdata.2022.1001063. URL: https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2022.1001063.

[2] Mehmet Karakose et al. "A New Approach for Deepfake Detection with the Choquet Fuzzy Integral". eng. In: *Applied sciences* 14.16 (2024), pp. 7216–. ISSN: 2076-3417.

[3] April Rubin. *Scams use AI to mimic senior officials' voices, FBI warms*. https://www.axios.com/2025/05/15/artificial-intelligence-voice-scams-government-officials?utm_source=chatgpt.com. 2025.

[4] Times of India. *Scammers use AI George Clooney to steal over Rs. 11 lakhs from Facebook user*. https://timesofindia.indiatimes.com/technology/artificial-intelligence/scammers-use-ai-george-clooney-to-steal-over-rs-11-lakhs-from-facebook-user/articleshow/121171381.cms?utm_source=chatgpt.com. 2025.

[5] Jonat John Mathew et al. "Towards the Development of a Real-Time Deepfake Audio Detection System in Communication Platforms". eng. In: (2024).

[6] Xin Wang et al. "ASVspoof 2019: A large-scale public database of synthesized, converted and replayed speech". eng. In: *Computer speech language* 64 (2020), pp. 101114–. ISSN: 0885-2308.

[7] "Voice spoofing detection corpus for single and multi-order audio replays". eng. In: *Computer speech language* 65 (2021), pp. 101132–. ISSN: 0885-2308.

[8] Rishabh Ranjan, Mayank Vatsa, and Richa Singh. "Uncovering the Deceptions: An Analysis on Audio Spoofing Detection and Future Prospects". eng. In: (2023).

[9] Bowen Zhang et al. "Audio Deepfake Detection: What Has Been Achieved and What Lies Ahead". eng. In: *Sensors (Basel, Switzerland)* 25.7 (2025), pp. 1989–. ISSN: 1424-8220.

[10] Piotr Kawa, Marcin Plata, and Piotr Syga. "Defense Against Adversarial Attacks on Audio DeepFake Detection". eng. In: (2022).

[11] Muhammad Umar Farooq et al. “Transferable Adversarial Attacks on Audio Deepfake Detection”. eng. In: (2025).

[12] Taiba Majid Wani et al. “Detecting Audio Deepfakes: Integrating CNN and BiLSTM with Multi-Feature Concatenation”. eng. In: *Proceedings of the 2024 ACM Workshop on Information Hiding and Multimedia Security*. New York, NY, USA: ACM, 2024, pp. 271–276. ISBN: 9798400706370.

[13] Shuhui Qu et al. “Understanding Audio Pattern Using Convolutional Neural Network From Raw Waveforms”. eng. In: (2016).

[14] Noussaiba Djeffal et al. “Transfer Learning-Based Deep Residual Learning for Speech Recognition in Clean and Noisy Environments”. eng. In: (2025).

[15] Jiangyan Yi et al. “Audio Deepfake Detection: A Survey”. eng. In: (2023).

[16] Gustav Arnt Palmelund Bonvang. *Listening Beyond Words: Transfer Learning for Audio Deepfake Detection*. eng. 2023.

[17] Peter Ochieng. “Speech Separation Based on Pre-trained Model and Deep Modularization”. In: *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2025, pp. 1–5. DOI: 10.1109/ICASSP49660.2025.10888747.

[18] Xinlu He and Jacob Whitehill. *Survey of End-to-End Multi-Speaker Automatic Speech Recognition for Monaural Audio*. 2025. arXiv: 2505.10975 [cs.CL]. URL: https://arxiv.org/abs/2505.10975.

[19] Javier Canales Luna. *What is Transfer Learning in AI? An Introductory Guide with Examples*. https://www.datacamp.com/blog/what-is-transfer-learning-in-ai-an-introductory-guide. 2024.

[20] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. “A survey of transfer learning”. In: *Journal of Big data* 3 (2016), pp. 1–40.

[21] Govind Mittal et al. “PITCH: AI-assisted Tagging of Deepfake Audio Calls using Challenge-Response”. eng. In: (2024).

[22] Mouna Rabhi, Spiridon Bakiras, and Roberto Di Pietro. “Audio-deepfake detection: Adversarial attacks and countermeasures”. eng. In: *Expert systems with applications* 250 (2024), pp. 123941–. ISSN: 0957-4174.

[23] abhishtagatya. *abhishtagatya/wavlm-base-960h-asv19-deepfake*. https://huggingface.co/abhishtagatya/wavlm-base-960h-asv19-deepfake/blob/main/README.md?code=true. 2024.

[24] Rafizah Mohd Hanifa et al. “Voiced and unvoiced separation in malay speech using zero crossing rate and energy”. eng. In: *Indonesian Journal of Electrical Engineering and Computer Science* 16.2 (2019), pp. 775–. ISSN: 2502-4752.

[25] Jinyu Zhan et al. "Detecting Spoofed Speeches via Segment-Based Word CQCC and Average ZCR for Embedded Systems". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.11 (2022), pp. 3862–3873. DOI: 10.1109/TCAD.2022.3197531.

[26] Menglu Li, Yasaman Ahmadiadli, and Xiao-Ping Zhang. "Audio Anti-Spoofing Detection: A Survey". eng. In: (2024).

[27] Pejman Mowlaee, Mads Græsbøll Christensen, and Søren Holdt Jensen. *Sinusoidal masks for single channel speech separation*. eng. 2010.

Appendix A

Appendix

```python
# -*- coding: utf-8 -*-
"""Overlap generation.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1jOeyaZrMKOUAE2LkPO-fL9haJb05cTgQ
"""

from google.colab import drive
import os
import librosa
import numpy as np
import soundfile as sf
import secrets  # Cryptographic randomness
import random  # General randomness
import torchaudio
import torchaudio.transforms as T

drive.mount('/content/drive')

# Step 2: Define dataset paths
base_path = "/content/drive/My Drive/My notes - cyber/Semester 4"
metadata_path = os.path.join(base_path, "ASVspoof2019_LA_cm_protocols")
output_path = os.path.join(base_path, "overlapped_audio_2")
metadata_output_path = os.path.join(base_path, "overlapped_audio_metadata_2.txt")

# Ensure output directory exists
os.makedirs(output_path, exist_ok=True)

# Step 3: Read metadata to classify audio files
spoof_files = []
bonafide_files = []
```

```python
file_status = {}  # Dictionary to store file name -> bonafide/spoof

# Read the metadata file
metadata_file = os.path.join(metadata_path, "ASVspoof2019.LA.cm.eval.trl.txt"
    )

with open(f"{metadata_file}", "r") as f:
    for line in f:
        parts = line.strip().split()
        if len(parts) < 5:
            continue  # Skip malformed lines
        _, filename, _, _, label = parts  # Extract relevant columns
        file_path = os.path.join(base_path, "ASVspoof2019_LA_eval/flac",
    filename+’.flac’)  # Adjust path
        if os.path.exists(file_path):  # Ensure file exists before adding
            if label == "bonafide":
                bonafide_files.append(file_path)
            else:
                spoof_files.append(file_path)


print(f"Loaded {len(bonafide_files)} bonafide and {len(spoof_files)} spoof
    files.")

import torch

# Step 4: Function to create a realistic overlap and log metadata
def overlap_audio(file1, file2, label1, label2, output_dir):
    audio1, sr1 = librosa.load(file1, sr=None)
    audio2, sr2 = librosa.load(file2, sr=None)

    if sr1 != sr2:
        raise ValueError("Sampling rates do not match!")

    # Generate a random time offset (up to 1 second)
    max_delay = int(sr1 * np.random.uniform(0, 1))  # Up to 1 second shift
    if secrets.randbelow(2) == 0:  # Randomly decide who starts first
        audio1 = np.pad(audio1, (max_delay, 0))  # Delay first audio
    else:
        audio2 = np.pad(audio2, (max_delay, 0))  # Delay second audio

    # Match length of both signals (cut longer one)
    min_len = min(len(audio1), len(audio2))
    audio1 = audio1[:min_len]
    audio2 = audio2[:min_len]

    # Apply random gain variation (loudness change)
    gain1 = np.random.uniform(0.7, 1.3)  # Random loudness for audio1
    gain2 = np.random.uniform(0.7, 1.3)  # Random loudness for audio2
    audio1 *= gain1
    audio2 *= gain2

    # # Apply reverberation (simulates room acoustics)
    # reverb = T.Reverberate()
    # audio1 = reverb(torch.tensor(audio1).unsqueeze(0)).squeeze(0).numpy()
    # audio2 = reverb(torch.tensor(audio2).unsqueeze(0)).squeeze(0).numpy()

    # Mix both audio signals
    mixed_audio = audio1 + audio2

    # Normalize to prevent clipping
    mixed_audio = mixed_audio / np.max(np.abs(mixed_audio))

    # Define mixed audio status
    if label1 == "spoof" and label2 == "spoof":
        mixed_label = "SS"  # Spoof-Spoof
    elif label1 == "bonafide" and label2 == "bonafide":
        mixed_label = "BB"  # Bonafide-Bonafide
    else:
        mixed_label = "SB"  # Spoof-Bonafide

    # Save the mixed audio file
    new_filename = f"{os.path.basename(file1).split('.')[0]}_{os.path.basename(file2).split('.')[0]}.flac"
    output_filepath = os.path.join(output_dir, new_filename)
    sf.write(output_filepath, mixed_audio, sr1)

    return new_filename, file1, label1, file2, label2, mixed_label

# Step 5: Use cryptographic randomness to create 25,000 mixed audio files and log metadata
mix_log = []
num_pairs = 30000  # Target number of mixed audio files

for i in range(num_pairs):
    if i % 5000==0:
      print(i)
    # Cryptographic randomness for true unpredictability
    if secrets.randbelow(2) == 0:  # 50% chance of spoof-bonafide mix
        file1 = secrets.choice(spoof_files)
        file2 = secrets.choice(bonafide_files)
        label1 = "spoof"
        label2 = "bonafide"
    else:
        # Randomly choose bonafide-bonafide or spoof-spoof
        if secrets.randbelow(2) == 0 and len(bonafide_files) > 1:
            file1, file2 = random.sample(bonafide_files, 2)
            label1 = label2 = "bonafide"
        else:
            file1, file2 = random.sample(spoof_files, 2)
            label1 = label2 = "spoof"

    # Overlap the chosen files
    mixed_file, file1_name, label1, file2_name, label2, mixed_label = overlap_audio(file1, file2, label1, label2, output_path)
    mix_log.append(f"{mixed_file} {os.path.basename(file1_name)} {label1} {os.path.basename(file2_name)} {label2} {mixed_label}")


# Step 6: Save the metadata for the new dataset
with open(metadata_output_path, "w") as f:
    for entry in mix_log:
        f.write(entry + "\n")

print(f"    Successfully created {len(mix_log)} mixed audio files!")
print(f"      Metadata saved at: {metadata_output_path}")

counter = {}
for line in mix_log:
  parts = line.strip().split(' ')
  if parts[-1] not in counter:
    counter[parts[-1]]=0
  counter[parts[-1]]+=1

print(counter)
```

Listing A.1: Complete code for Overlap Generation

```
# -*- coding: utf-8 -*-
"""Pretrained sota spoof.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1wtqimvr7pbYLldir1ucYycy6MR5-3iDH
"""

!pip install torch torchaudio transformers scikit-learn soundfile asteroid

# Part 2: Load pretrained models and dependencies
import torch
from asteroid.models import ConvTasNet
from transformers import AutoProcessor, AutoModelForPreTraining, AutoModelForAudioClassification, AutoConfig
from transformers import Wav2Vec2Processor, Wav2Vec2ForSequenceClassification, Wav2Vec2FeatureExtractor, WavLMForSequenceClassification
import soundfile as sf
import os
import torchaudio
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
```

```
import numpy as np
from collections import Counter

import torchaudio.transforms as T

# Part 1: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Set paths for the audio and metadata directories
audio_folder = "/content/drive/My Drive/My notes - cyber/Semester 4/
    ASVspoof2019_LA_eval/flac"
metadata_file = "/content/drive/My Drive/My notes - cyber/Semester 4/
    ASVspoof2019_LA_cm_protocols/ASVspoof2019.LA.cm.eval.trl.txt"

# Check if CUDA is available for using GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Load the pretrained separation model (ConvTasNet)
sepformer = ConvTasNet.from_pretrained('mpariente/ConvTasNet_WHAM!_sepclean')
sepformer = sepformer.to(device)

# config = AutoConfig.from_pretrained("abhishtagatya/wavlm-base-960h-itw-
    deepfake")
# feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained("abhishtagatya
    /wavlm-base-960h-itw-deepfake")
# spoof_model = WavLMForSequenceClassification.from_pretrained("abhishtagatya
    /wavlm-base-960h-itw-deepfake", config=config).to(device)


config = AutoConfig.from_pretrained("abhishtagatya/wavlm-base-960h-asv19-
    deepfake")
feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained("abhishtagatya/
    wavlm-base-960h-asv19-deepfake")
spoof_model = WavLMForSequenceClassification.from_pretrained("abhishtagatya/
    wavlm-base-960h-asv19-deepfake", config=config).to(device)


spoof_model = spoof_model.to(device)

from asteroid.models import BaseModel

model = BaseModel.from_pretrained("JorisCos/ConvTasNet_Libri2Mix_sepnoisy_16k
    ")
model = model.to(device)

!pip install speechbrain

from speechbrain.inference.separation import SepformerSeparation as separator

```

```python
model1 = separator.from_hparams(source="speechbrain/sepformer-wsj02mix", savedir='pretrained_models/sepformer-wsj02mix',run_opts={"device": "cuda"})
model1 = model1.to(device)

import pandas as pd

# Load file, specify only needed columns
df = pd.read_csv(metadata_file, header=None, delim_whitespace=True, usecols=[1, 4], names=["file", "label"])

# Map labels to numeric (spoof = 1, bonafide = 0)
label_map = {"spoof": 1, "bonafide": 0}
df["label"] = df["label"].map(label_map)

df.shape

def load_audio(audio_path):
    waveform, sample_rate = torchaudio.load(audio_path)
    if sample_rate != 16000:
        waveform = torchaudio.transforms.Resample(sample_rate, 16000)(waveform)
    return waveform.squeeze().numpy()

def predict_spoof(audio_path):
    audio = load_audio(audio_path)
    inputs = feature_extractor(audio, sampling_rate=16000, return_tensors="pt", padding=True)
    with torch.no_grad():
        logits = spoof_model(**inputs.to(spoof_model.device)).logits
    probs = torch.softmax(logits, dim=-1).cpu().numpy()[0]
    predicted_label = "spoof" if probs[1] > probs[0] else "bonafide"
    return predicted_label, probs

from tqdm import tqdm
from sklearn.metrics import accuracy_score, confusion_matrix, roc_auc_score

y_true = []
y_pred = []
y_score = []
count = 0

for _, row in tqdm(df.iterrows(), total=len(df)):
    count += 1
    if count > 30000:
        break


    audio_path = os.path.join(audio_folder, row["file"])
    audio_path = audio_path + '.flac'

    if not os.path.exists(audio_path):
        print(f"Missing: {audio_path}")
        continue

    pred, score = predict_spoof(audio_path)

    y_true.append(row["label"])
    y_pred.append(pred)
    y_score.append(score)

print(y_true)
print(y_pred)
print(y_score)

# === Step 6: Evaluate ===

y_true_label = ["spoof" if label == 1 else 0 for label in y_true]
cm = confusion_matrix(y_true_label, y_pred)


acc = accuracy_score(y_true_label, y_pred)

y_score_modified = [score[1] for score in y_score]
roc = roc_auc_score(y_true, y_score_modified)

print(f"\nAccuracy: {acc:.4f}")
print("Confusion Matrix:\n", cm)
print(f"ROC AUC Score: {roc:.4f}")

from sklearn.metrics import roc_curve
import numpy as np

fpr, tpr, thresholds = roc_curve(y_true, y_score_modified, pos_label=1)

# Find the point where FPR     1 - TPR
eer = fpr[np.nanargmin(np.absolute((1 - tpr) - fpr))]

print(f"Equal Error Rate (EER): {eer:.4f}")
```

Listing A.2: Complete code for Testing SOTA spoof detection model

```
# -*- coding: utf-8 -*-
"""pretrained pipeline.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1HveSqt1YMWgGNFNq05qH9QpwqR8YXB9w
"""

!pip install torch torchaudio transformers scikit-learn soundfile asteroid
```

```

!pip install speechbrain

# Part 2: Load pretrained models and dependencies
import torch
from asteroid.models import ConvTasNet
from transformers import AutoProcessor, AutoModelForPreTraining,
    AutoModelForAudioClassification, AutoConfig
from transformers import Wav2Vec2Processor, Wav2Vec2ForSequenceClassification
    , Wav2Vec2FeatureExtractor, WavLMForSequenceClassification
import soundfile as sf
import os
import torchaudio
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
import numpy as np
from collections import Counter
import torchaudio.transforms as T

# Part 1: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Set paths for the audio and metadata directories
audio_folder = "/content/drive/My Drive/My notes - cyber/Semester 4/
    overlapped_audio_2/"
metadata_file = "/content/drive/My Drive/My notes - cyber/Semester 4/
    overlapped_audio_metadata_2.txt"

# Check if CUDA is available for using GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Load the pretrained separation model (ConvTasNet)
sepformer = ConvTasNet.from_pretrained('mpariente/ConvTasNet_WHAM!_sepclean')
sepformer = sepformer.to(device)

config = AutoConfig.from_pretrained("abhishtagatya/wavlm-base-960h-asv19-
    deepfake")
feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained("abhishtagatya/
    wavlm-base-960h-asv19-deepfake")
spoof_model = WavLMForSequenceClassification.from_pretrained("abhishtagatya/
    wavlm-base-960h-asv19-deepfake", config=config).to(device)


spoof_model = spoof_model.to(device)

# # Check if CUDA is available for using GPU
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```python
# print(f"Using device: {device}")

# saved_model_path = "/content/drive/My Drive/My notes - cyber/Semester 4/
    finetuned_spoof_model6"

# config = AutoConfig.from_pretrained(saved_model_path)
# feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained(
    saved_model_path)
# spoof_model = WavLMForSequenceClassification.from_pretrained(
    saved_model_path, config=config).to(device)
# spoof_model.eval()

# spoof_model = spoof_model.to(device)

from speechbrain.pretrained import SepformerSeparation
model1 = SepformerSeparation.from_hparams(
  "speechbrain/sepformer-wsj02mix"
)

# Part 2: Read Metadata from a .txt file

def load_metadata(metadata_file):
    metadata = []
    count = 0
    metadata_count = {}
    with open(metadata_file, "r") as f:
        for line in f:
            count += 1
            parts = line.strip().split(' ')
            filename, _, _, _, _, label = parts
            metadata.append((filename, label))
            if label not in metadata_count:
                metadata_count[label] = 1
            else:
                metadata_count[label] += 1

    print(count, metadata_count)
    return metadata

# Example usage: Load metadata
metadata = load_metadata(metadata_file)
print(metadata[:5])  # Display first 5 entries

# Part 3: Function to load audio
def load_audio(audio_path):
    """
    Function to load audio using torchaudio.
    Converts audio to a waveform tensor.
    """
    waveform, sample_rate = torchaudio.load(audio_path, normalize=True)
    return waveform
```

```

# Part 5: Function to predict spoof or bonafide using Wav2Vec2

def predict_spoof(audio):
    inputs = feature_extractor(audio, sampling_rate=16000, return_tensors="pt", padding=True)
    # if audio.ndimension() == 1:
    #     # If it's 1D, add an extra dimension to make it 2D (e.g., batch dimension)
    #     audio = audio.unsqueeze(0)
    # if audio.size(1) < 100:
    #     audio = F.pad(audio, (0, 100 - audio.size(1)))
    # with torch.no_grad():
    #     logits = spoof_model(**inputs.to(spoof_model.device)).logits

    # print(f"Waveform size: {audio.size()}")

    # probs = torch.softmax(logits, dim=-1).cpu().numpy()[0]
    # predicted_label = "spoof" if probs[1] > probs[0] else "bonafide"
    # return predicted_label, probs

    inputs = feature_extractor(audio, sampling_rate=16000, return_tensors="pt", padding=True)
    with torch.no_grad():
        logits = spoof_model(**inputs.to(spoof_model.device)).logits
    probs = torch.softmax(logits, dim=-1).cpu().numpy()[0]
    predicted_label = "spoof" if probs[1] > probs[0] else "bonafide"
    return predicted_label, probs

from scipy.io.wavfile import write


output_dir = '/content/drive/My Drive/My notes - cyber/Semester 4/Separated_overlapped_audio_1'

# Create directory if it doesn't exist
os.makedirs(output_dir, exist_ok=True)

# Part 8: Loop Through Audio Files and Perform Inference
def run_pipeline_for_all_files(audio_folder, metadata):
    """
    Loop through all audio files, separate the speakers, and perform spoof detection.
    """
    predictions = []
    y_true = []
    y_pred = []
    y_score = []
    all_true = []
    all_pred = []
    all_score = []
```

```python
    count = 0
    for filename, true_label in metadata:
        audio_path = os.path.join(audio_folder, filename)  # Path to the audio file

        # Step 1: Load the audio
        try:
            waveform = load_audio(audio_path)
        except Exception as e:
            print(f"Error loading {filename}: {e}")
            continue  # Skip this file if there's an issue

        count += 1
        if count > 10000:
            break

        # Step 2: Separate speakers

        speaker_waveform = model1.separate_file(audio_path)

        speaker1 = speaker_waveform[:, :, 0].squeeze(0).detach().cpu()
        speaker2 = speaker_waveform[:, :, 1].squeeze(0).detach().cpu()


        # # Save directly from tensor (expected shape: [channels, time])
        # output1_path = os.path.join(output_dir, "output" + str(count + 1) +".wav")
        # output2_path = os.path.join(output_dir, "output" + str(count + 2) +".wav")
        # torchaudio.save(output1_path, speaker1, 8000)
        # torchaudio.save(output2_path, speaker2, 8000)
        # count += 2


        # Step 3: Predict spoof/bona fide for each speaker

        resample = torchaudio.transforms.Resample(orig_freq=8000, new_freq=16000)
        speaker1_resampled = resample(speaker1)
        speaker2_resampled = resample(speaker2)

        label1, prob1 = predict_spoof(speaker1_resampled)
        label2, prob2 = predict_spoof(speaker2_resampled)

        # Combine the normalized scores by averaging
        combined_score = (prob1 + prob2) / 2

        # Step 4: Format prediction (e.g., "sb" or "bb")
        if label1 == "spoof" and label2 == "bonafide":
            prediction = "SB"
        elif label2 == "spoof" and label1 == "bonafide":
            prediction = "SB"
        elif label1 == "bonafide" and label2 == "bonafide":
            prediction = "BB"
        else:
            prediction = "SS"

        temp1, temp2 = true_label[0], true_label[1]
        if temp1 == "S":
            individual_true_value1 = 1
        else:
            individual_true_value1 = 0
        if temp2 == "S":
            individual_true_value2 = 1
        else:
            individual_true_value2 = 0

        binary_labels = [individual_true_value1, individual_true_value2]

        pred_scores = [prob1[1], prob2[1]]

        temp_all_pred = [label1, label2]


        # Permutation 1: scores as-is
        error1 = np.sum(np.abs(np.array(pred_scores) - np.array(binary_labels
    )))

        # Permutation 2: flipped scores
        flipped_scores = pred_scores[::-1]
        error2 = np.sum(np.abs(np.array(flipped_scores) - np.array(
    binary_labels)))

        if error1 <= error2:
            all_true.extend(binary_labels)
            all_score.extend(pred_scores)
            all_pred.extend(temp_all_pred)
        else:
            all_true.extend(binary_labels)
            all_score.extend(flipped_scores)
            all_pred.extend(temp_all_pred[::-1])

        y_true.append(true_label)
        y_pred.append(prediction)
        y_score.append(combined_score)

        # print(y_true, y_pred, y_score, all_true, all_score, prob1, prob2)

        # Delete tensors to free memory
        del waveform, speaker1, speaker2, speaker_waveform

        # Empty CUDA cache periodically
        if count % 100 == 0:  # Adjust frequency as needed
            torch.cuda.empty_cache()


    return y_true, y_pred, y_score, all_true, all_score, all_pred

def calculate_eer(y_true, y_pred, y_score, all_true, all_score):
    """
    Calculate the Equal Error Rate (EER) from the predictions.
    """
    # Separate the combined scores and true labels
    combined_scores = np.array(y_score)
    true_labels = np.array(y_true)

    # Convert true labels to binary (spoof = 1, bonafide = 0)
    # y_true_binary = np.array([0 if label == "ss" else 1 if label == "sb" else 2 for label in true_labels])

    y_true_binary = np.array([0 if label == "ss" else 1 for label in true_labels])

    # # Calculate ROC curve and EER
    # fpr, tpr, thresholds = roc_curve(y_true_binary, combined_scores)

    # # FRR = 1 - TPR
    # frr = 1 - tpr

    # # Find EER (Equal Error Rate), where FAR equals FRR
    # eer_index = np.argmin(np.abs(fpr - frr))
    # eer_threshold = thresholds[eer_index]
    # eer = fpr[eer_index]  # EER is where FAR equals FRR

    fpr, tpr, thresholds = roc_curve(all_true, all_score, pos_label=1)
    fnr = 1 - tpr
    eer_index = np.nanargmin(np.abs(fnr - fpr))
    eer = (fpr[eer_index] + fnr[eer_index]) / 2
    threshold = thresholds[eer_index]
    return eer, threshold

    # Plot FAR and FRR
    plt.plot(fpr, label='FAR')
    plt.plot(frr, label='FRR')
    plt.axvline(x=eer_index, linestyle='--', color='r', label=f'EER at threshold {eer_threshold:.2f}')
    plt.legend()
    plt.xlabel('Threshold')
    plt.ylabel('Rate')
    plt.title('FAR and FRR')
    plt.show()

    # Print EER
    print(f"EER: {eer:.4f} at threshold {eer_threshold:.2f}")

    return eer

from sklearn.metrics import classification_report, accuracy_score
import pandas as pd

def evaluate_multiclass_metrics(y_true, y_pred, y_score):

    report_dict = classification_report(y_true, y_pred, target_names=["SS", "SB", "BB"], labels=["SS", "SB", "BB"], output_dict=True)

    print("        Classification Report:")
    print(report_dict)

    acc = accuracy_score(y_true, y_pred)
    print(f"    Accuracy: {acc * 100:.2f}%")


    report_df = pd.DataFrame(report_dict).transpose()

    # Save to CSV
    report_df.to_csv("/content/drive/My Drive/My notes - cyber/Semester 4/classification_report_pretrained.csv")


    return acc

# Part 10: Run the Full Pipeline

# Load metadata
metadata = load_metadata(metadata_file)

# Run the pipeline for all files in the audio folder
y_true1, y_pred1, y_score1, all_true, all_score, all_pred = run_pipeline_for_all_files(audio_folder, metadata)
y_score1_modified = [score[0] for score in y_score1]

# Calculate EER
# eer = calculate_eer(y_true1, y_pred1, y_score1_modified, all_true, all_score)
# print(f"Equal Error Rate (EER): {eer * 100:.2f}%")

accuracy = evaluate_multiclass_metrics(y_true1, y_pred1, y_score1_modified)

acc1 = accuracy_score(y_true1, y_pred1)
print(f"Accuracy: {acc1 * 100:.2f}%")

eer, threshold = calculate_eer(y_true1, y_pred1, y_score1_modified, all_true, all_score)
print(f"Equal Error Rate (EER): {eer * 100:.2f}%")
print(f"Threshold at EER: {threshold:.6f}")

from sklearn.metrics import accuracy_score, precision_score, recall_score,
    f1_score

binary_all_pred = [1 if label=='spoof' else 0 for label in all_pred]

# # Calculate metrics
# all_pred = []
# for score in all_score:
#     if score > 0.5:
#         all_pred.append(1)
#     else:
#         all_pred.append(0)

accuracy_all = accuracy_score(all_true, binary_all_pred)
precision_all = precision_score(all_true, binary_all_pred)
recall_all = recall_score(all_true, binary_all_pred)
f1_all = f1_score(all_true, binary_all_pred)

# Display results
print(f"Accuracy: {accuracy_all:.2f}")
print(f"Precision: {precision_all:.2f}")
print(f"Recall: {recall_all:.2f}")
print(f"F1 Score: {f1_all:.2f}")

# Assuming labels = ['ss', 'sb', 'bb']
y_true_combined_binary = []
for label in y_true1:
    if label in ['ss', 'sb']:
        y_true_combined_binary.append(1)  # spoof
    else:
        y_true_combined_binary.append(0)  # bonafide

# from sklearn.metrics import roc_curve
# from scipy.optimize import brentq
# from scipy.interpolate import interp1d

# # Compute ROC
# print(y_true_combined_binary[0], y_score1_modified[0])
# fpr, tpr, thresholds = roc_curve(y_true_combined_binary, y_score1_modified)
# fnr = 1 - tpr
# print(np.unique(fpr), np.unique(tpr))
# # Interpolate and find EER
# eer_index_combined = np.nanargmin(np.absolute(fnr - fpr))
# eer_combined = (fpr[eer_index_combined] + fnr[eer_index_combined]) / 2

# print(f"EER: {eer_combined:.4f}")

metrics_history = {
    "name" : '',
    "eer": [],
    "threshold" :[],
    "total_acc": [],
    "individual_acc": [],
    "precision": [],
    "recall": [],
    "f1_score": []
    # Add more if needed
}
metrics_history["name"] = 'Pre-Trained Metrics'
metrics_history["eer"].append(eer)
metrics_history["threshold"].append(threshold)
metrics_history["total_acc"].append(acc1)  # Replace with actual value from `
    evaluate_on_dev`
metrics_history["individual_acc"].append(accuracy_all)
metrics_history["precision"].append(precision_all)
metrics_history["recall"].append(recall_all)
metrics_history["f1_score"].append(f1_all)

# Replace with actual value

import os
import pandas as pd

metrics_file = "/content/drive/My Drive/My notes - cyber/Semester 4/
    pipeline_metrics.csv"

# Check if file exists and load old metrics
if os.path.exists(metrics_file):
    existing_df = pd.read_csv(metrics_file)
    combined_df = pd.concat([existing_df, pd.DataFrame(metrics_history)],
    ignore_index=True)
else:
    combined_df = pd.DataFrame(metrics_history)

# Save the updated file
combined_df.to_csv(metrics_file, index=False)
```

Listing A.3: Complete code for Testing pre-trained Pipeline and Trained Pipeline

```
# -*- coding: utf-8 -*-
"""Train pipeline.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1kCq_6_QEprfoFkK0yyM-t3lyycD6DzFg
"""

!pip install torch torchaudio transformers scikit-learn soundfile asteroid
    speechbrain
```

```

# Part 2: Load pretrained models and dependencies
import torch
from asteroid.models import ConvTasNet
from transformers import AutoProcessor, AutoModelForPreTraining,
    AutoModelForAudioClassification, AutoConfig
from transformers import Wav2Vec2Processor, Wav2Vec2ForSequenceClassification
    , Wav2Vec2FeatureExtractor, WavLMForSequenceClassification
import soundfile as sf
import os
import torchaudio
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
import numpy as np
from collections import Counter
import torchaudio.transforms as T
from torch.nn.utils import clip_grad_norm_

# Part 1: Mount Google Drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Set paths for the audio and metadata directories
audio_folder = "/content/drive/My Drive/My notes - cyber/Semester 4/
    overlapped_audio_2/"
metadata_file = "/content/drive/My Drive/My notes - cyber/Semester 4/
    overlapped_audio_metadata_2.txt"
audio_folder_train = "/content/drive/My Drive/My notes - cyber/Semester 4/
    overlapped_audio_train/"
audio_folder_dev = "/content/drive/My Drive/My notes - cyber/Semester 4/
    overlapped_audio_dev/"
metadata_file_train = "/content/drive/My Drive/My notes - cyber/Semester 4/
    overlapped_audio_metadata_train.txt"
metadata_file_dev = "/content/drive/My Drive/My notes - cyber/Semester 4/
    overlapped_audio_metadata_dev.txt"

import random
from torch.optim import AdamW
from transformers import get_scheduler


# Set random seed for reproducibility
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

set_seed(42)
```

```

# # Check if CUDA is available for using GPU
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# print(f"Using device: {device}")

# from speechbrain.pretrained import SepformerSeparation
# model1 = SepformerSeparation.from_hparams(
#   "speechbrain/sepformer-wsj02mix"
# )

# config = AutoConfig.from_pretrained("abhishtagatya/wavlm-base-960h-asv19-
    deepfake")
# feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained("abhishtagatya
    /wavlm-base-960h-asv19-deepfake")
# spoof_model = WavLMForSequenceClassification.from_pretrained("abhishtagatya
    /wavlm-base-960h-asv19-deepfake", config=config).to(device)
# spoof_model.train()

# spoof_model = spoof_model.to(device)

# Check if CUDA is available for using GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

from speechbrain.pretrained import SepformerSeparation
model1 = SepformerSeparation.from_hparams(
  "speechbrain/sepformer-wsj02mix"
)
saved_model_path = "/content/drive/My Drive/My notes - cyber/Semester 4/
    finetuned_spoof_model5"

config = AutoConfig.from_pretrained(saved_model_path)
feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained(saved_model_path
    )
spoof_model = WavLMForSequenceClassification.from_pretrained(saved_model_path
    , config=config).to(device)
spoof_model.train()

spoof_model = spoof_model.to(device)

# Part 2: Read Metadata from a .txt file

def load_metadata(metadata_file):
    metadata = []
    count = 0
    metadata_count = {}
    with open(metadata_file, "r") as f:
        for line in f:
            count += 1
            parts = line.strip().split(' ')
            filename, _, _, _, _, label = parts
            metadata.append((filename, label))
            if label not in metadata_count:
                metadata_count[label] = 1
            else:
                metadata_count[label] += 1

    print(count, metadata_count)
    return metadata

# Example usage: Load metadata
metadata = load_metadata(metadata_file)
metadata_dev = load_metadata(metadata_file_dev)
metadata_train = load_metadata(metadata_file_train)
print(metadata[:5])  # Display first 5 entries
print(metadata_dev[:5])
print(metadata_train[:5])

# Part 3: Function to load audio
def load_audio(audio_path):
    """
    Function to load audio using torchaudio.
    Converts audio to a waveform tensor.
    """
    waveform, sample_rate = torchaudio.load(audio_path, normalize=True)
    return waveform

# Part 5: Function to predict spoof or bonafide using Wav2Vec2

def predict_spoof(audio):
    inputs = feature_extractor(audio, sampling_rate=16000, return_tensors="pt
    ", padding=True)
    with torch.no_grad():
        logits = spoof_model(**inputs.to(spoof_model.device)).logits
    probs = torch.softmax(logits, dim=-1).cpu().numpy()[0]
    predicted_label = "spoof" if probs[1] > probs[0] else "bonafide"
    return predicted_label, probs

# Optimizer and scheduler
learning_rate = 1e-6
optimizer = AdamW(spoof_model.parameters(), lr=learning_rate)
batch_size = 1
num_epochs = 4
gradient_accumulation_steps = 2

# Count training steps
total_training_steps = (len(metadata_train) // gradient_accumulation_steps) *
     num_epochs
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=total_training_steps,
)

# Load optimizer and scheduler
optimizer.load_state_dict(torch.load(os.path.join(saved_model_path, "optimizer.pt")))
lr_scheduler.load_state_dict(torch.load(os.path.join(saved_model_path, "scheduler.pt")))

learning_rate = 3e-4
batch_size = 1
num_epochs = 4
gradient_accumulation_steps = 2

# Evaluate on dev set
def evaluate_on_dev(dev_metadata, dev_audio_folder):
    spoof_model.eval()
    total_loss = 0
    correct = 0
    total = 0
    count1 = 0
    random.shuffle(dev_metadata)
    dev_metadata_subset = dev_metadata[:100]
    with torch.no_grad():
        for filename, label in tqdm(dev_metadata_subset, desc="Evaluating"):
            audio_path = os.path.join(dev_audio_folder, filename)

            label_map = {'s': 1, 'b': 0}
            labels_pair = [label_map[label[0].lower()], label_map[label[1].lower()]]

            try:
                waveform = load_audio(audio_path)
                separated = model1.separate_file(audio_path)  # Assuming model1 is defined
                s1 = separated[:, :, 0].squeeze(0).cpu()
                s2 = separated[:, :, 1].squeeze(0).cpu()
            except Exception as e:
                print(f"Skipping {filename} due to error: {e}")
                continue

            best_loss = float('inf')
            best_perm = None

            for perm in [(s1, s2, labels_pair), (s2, s1, labels_pair[::-1])]:
              loss_total = 0
              predictions = []
              for i, spk in enumerate(perm[:2]):
                  inputs = feature_extractor(spk, sampling_rate=16000, return_tensors="pt", padding=True)
                  inputs = {k: v.to(spoof_model.device) for k, v in inputs.
```

```
items()}
                  labels_tensor = torch.tensor([perm[2][i]]).to(spoof_model.device)
                  outputs = spoof_model(**inputs, labels=labels_tensor)
                  loss_total += outputs.loss.item()
              if loss_total < best_loss:
                  best_loss = loss_total
                  best_perm = perm

            for i, spk in enumerate(best_perm[:2]):
              inputs = feature_extractor(spk, sampling_rate=16000, return_tensors="pt", padding=True)
              inputs = {k: v.to(spoof_model.device) for k, v in inputs.items()}
              labels_tensor = torch.tensor([best_perm[2][i]]).to(spoof_model.device)
              outputs = spoof_model(**inputs, labels=labels_tensor)
              logits = outputs.logits
              preds = torch.argmax(logits, dim=-1)
              correct += (preds == labels_tensor).sum().item()
              total += 1
              total_loss += outputs.loss.item()

            # for i, spk in enumerate([s1, s2]):
            #     true_label = 1 if label[i].lower() == "s" else 0
            #     inputs = feature_extractor(spk, sampling_rate=16000, return_tensors="pt", padding=True)
            #     inputs = {k: v.to(spoof_model.device) for k, v in inputs.items()}
            #     labels = torch.tensor([true_label]).to(spoof_model.device)

            #     outputs = spoof_model(**inputs, labels=labels)
            #     loss = outputs.loss
            #     logits = outputs.logits
            #     preds = torch.argmax(logits, dim=-1)
            #     correct += (preds == labels).sum().item()
            #     total += 1
            #     total_loss += loss.item()

    accuracy = correct / total if total > 0 else 0
    avg_loss = total_loss / total if total > 0 else 0
    print(f"\n[DEV] Loss: {avg_loss:.4f} | Accuracy: {accuracy * 100:.2f}%\n")
    spoof_model.train()  # set back to training mode
    return avg_loss, accuracy

# Training loop
from tqdm import tqdm
global_step = 0


```

```python
for epoch in range(num_epochs):
    epoch_loss = 0.0
    print(f"\nEpoch {epoch + 1}/{num_epochs}")
    total_loss = 0.0
    count = 0
    random.seed(42)  # For reproducibility
    random.shuffle(metadata_train)
    spoof_model.train()
    metadata_train_subset = metadata_train[:150]
    step = 0
    for idx, (filename, label) in tqdm(enumerate(metadata_train_subset), desc="Training", total=len(metadata_train_subset)):

        audio_path = os.path.join(audio_folder_train, filename)

        label_map = {'s': 1, 'b': 0}
        labels = [label_map[label[0].lower()], label_map[label[1].lower()]]

        try:
            # Load and separate audio
            mixed = load_audio(audio_path)
            separated = model1.separate_file(audio_path)  # Assuming model1 is defined
            s1 = separated[:, :, 0].squeeze(0).cpu()
            s2 = separated[:, :, 1].squeeze(0).cpu()
        except Exception as e:
            print(f"Skipping {filename} due to error: {e}")
            continue

        permutations = [(s1, s2, labels),(s2, s1, labels[::-1])]
        best_loss = float('inf')
        best_perm = None

        for perm_s1, perm_s2, perm_labels in permutations:
          loss_total = 0
          for i, spk in enumerate([perm_s1, perm_s2]):
              inputs = feature_extractor(spk, sampling_rate=16000, return_tensors="pt", padding=True)
              inputs = {k: v.to(spoof_model.device) for k, v in inputs.items()}
              labels_tensor = torch.tensor([perm_labels[i]]).to(spoof_model.device)
              outputs = spoof_model(**inputs, labels=labels_tensor)
              loss_total += outputs.loss.item()

          if loss_total < best_loss:
              best_loss = loss_total
              best_perm = (perm_s1, perm_s2, perm_labels)

        for i, spk in enumerate(best_perm[:2]):
          inputs = feature_extractor(spk, sampling_rate=16000, return_tensors
="pt", padding=True)
          inputs = {k: v.to(spoof_model.device) for k, v in inputs.items()}
          labels_tensor = torch.tensor([best_perm[2][i]]).to(spoof_model.device)
          outputs = spoof_model(**inputs, labels=labels_tensor)
          loss = outputs.loss / gradient_accumulation_steps
          loss.backward()
          total_loss += loss.item()
          epoch_loss += loss.item()


        # for i, spk in enumerate([s1, s2]):
        #     # Determine true label (0 = bonafide, 1 = spoof)
        #     true_label = 1 if label[i].lower() == "s" else 0
        #     inputs = feature_extractor(spk, sampling_rate=16000, return_tensors="pt", padding=True)
        #     inputs = {k: v.to(spoof_model.device) for k, v in inputs.items()}
        #     labels = torch.tensor([true_label]).to(spoof_model.device)

        #     # Forward pass
        #     outputs = spoof_model(**inputs, labels=labels)
        #     loss = outputs.loss / gradient_accumulation_steps
        #     loss.backward()
        #     total_loss += loss.item()
        #     epoch_loss += loss.item()

        if (idx * 2 + i + 1) % gradient_accumulation_steps == 0:

            # In training loop, after loss.backward()
            clip_grad_norm_(spoof_model.parameters(), max_norm=1.0)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()
            global_step += 1


        # Free memory for the processed clip
        del s1, s2, labels
        del inputs, outputs

        if (idx + 1) % 100 == 0:
            print(f"Processed {idx + 1} files. Avg loss: {total_loss / (idx + 1):.4f}")
            torch.cuda.empty_cache()

    print(f"Epoch {epoch+1} finished. Total loss: {total_loss:.4f}")

    # Evaluate on dev set
    val_loss, val_acc = evaluate_on_dev(metadata_dev, audio_folder_dev)
    break
```

```

from transformers import AutoConfig

save_path = "/content/drive/My Drive/My notes - cyber/Semester 4/
    finetuned_spoof_model6"

# Save model, feature extractor, and config
spoof_model.save_pretrained(save_path)
feature_extractor.save_pretrained(save_path)
spoof_model.config.save_pretrained(save_path)


# Save optimizer and scheduler
torch.save(optimizer.state_dict(), os.path.join(save_path, "optimizer.pt"))
torch.save(lr_scheduler.state_dict(), os.path.join(save_path, "scheduler.pt")
    )

print(f"Model, feature extractor, and config saved to {save_path}")

metrics_history = {
    "epoch": [],
    "train_loss": [],
    "val_loss": [],
    "val_accuracy": []
}
metrics_history["epoch"].append(epoch + 1)
metrics_history["train_loss"].append(total_loss)
metrics_history["val_loss"].append(val_loss)
metrics_history["val_accuracy"].append(val_acc)

import os
import pandas as pd

metrics_file = "/content/drive/My Drive/My notes - cyber/Semester 4/
    training_metrics.csv"

# Check if file exists and load old metrics
if os.path.exists(metrics_file):
    existing_df = pd.read_csv(metrics_file)
    combined_df = pd.concat([existing_df, pd.DataFrame(metrics_history)],
    ignore_index=True)
else:
    combined_df = pd.DataFrame(metrics_history)

# Save the updated file
combined_df.to_csv(metrics_file, index=False)
```

Listing A.4: Complete Code for training the Pipeline

```
# -*- coding: utf-8 -*-
"""confusion matrix.ipynb
```

```

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/10g90ZBfFwJTyffs9hkEzhlK4dg9Tl9GT
"""

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.metrics import confusion_matrix

# Class labels
labels = ['SS', 'SB', 'BB']
y_true = ['SS'] * 2535 + ['SB'] * 4979 + ['BB'] * 2486

# Simulating predictions based on recall distribution (approximate)
y_pred = (
    ['SS'] * int(0.9889 * 2535) + ['SB'] * int(0.1869 * 4979) + ['BB'] * int
    (0.3459 * 2486) +
    ['SB'] * (2535 - int(0.9889 * 2535)) +  # SS misclassified as SB
    ['SS'] * (4979 - int(0.1869 * 4979)) +  # SB misclassified as SS
    ['SS'] * (2486 - int(0.3459 * 2486))    # BB misclassified as SS
)

# Generate confusion matrix
cm = confusion_matrix(y_true, y_pred, labels=labels)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels,
    yticklabels=labels)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for Pre-trained Pipeline')
plt.tight_layout()
plt.show()

# Re-import necessary libraries after code execution environment reset
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, confusion_matrix,
    ConfusionMatrixDisplay

# Simulated data based on description
# Ground truth labels (0 = bonafide, 1 = spoof)
y_true_individual = np.random.choice([0, 1], size=10000, p=[0.5, 0.5])

# Simulated predicted probabilities for the positive class (spoof)
# Assume high recall, moderate precision
y_scores_individual = y_true_individual * np.random.uniform(0.5, 1.0, size
    =10000) + \
                        (1 - y_true_individual) * np.random.uniform(0.0, 0.7,
    size=10000)

# Predicted labels based on a threshold of 0.5
y_pred_individual = (y_scores_individual > 0.5).astype(int)

# ROC Curve
fpr, tpr, thresholds = roc_curve(y_true_individual, y_scores_individual)
roc_auc = auc(fpr, tpr)

# Confusion Matrix
cm = confusion_matrix(y_true_individual, y_pred_individual)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["Bonafide"
    , "Spoof"])

# Plotting
# fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# # ROC curve
# ax[0].plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc
    :.2f})')
# ax[0].plot([0, 1], [0, 1], color='grey', lw=1, linestyle='--')
# ax[0].set_xlim([0.0, 1.0])
# ax[0].set_ylim([0.0, 1.05])
# ax[0].set_xlabel('False Positive Rate')
# ax[0].set_ylabel('True Positive Rate')
# ax[0].set_title('ROC Curve - Individual Audio Classification')
# ax[0].legend(loc="lower right")

# Confusion matrix
disp.plot(cmap=plt.cm.Blues)

plt.tight_layout()
plt.show()

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.metrics import confusion_matrix

# Class labels
labels = ['SS', 'SB', 'BB']
y_true = ['SS'] * 2535 + ['SB'] * 4979 + ['BB'] * 2486

# Simulating predictions based on recall distribution (approximate)
y_pred = (
    ['SS'] * int(0.4469 * 2535) + ['SB'] * int(0.3801 * 4979) + ['BB'] * int
    (0.9296 * 2486) +
    ['SB'] * (2535 - int(0.4469 * 2535)) +  # SS misclassified as SB
    ['SS'] * (4979 - int(0.3801 * 4979)) +  # SB misclassified as SS
    ['SS'] * (2486 - int(0.9296 * 2486))    # BB misclassified as SS
)

# Generate confusion matrix
cm = confusion_matrix(y_true, y_pred, labels=labels)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels,
    yticklabels=labels)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for Pre-trained Pipeline')
plt.tight_layout()
plt.show()

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# Simulated predictions and true labels for binary classification
# Assuming "1" = spoof, "0" = bonafide
# Using the derived metrics: Accuracy = 0.7441, Precision = 0.8472, Recall =
    0.5987
# Let's create a synthetic example with 1000 samples for visualization

# Estimate positives and negatives
n_samples = 1000
recall = 0.5987
precision = 0.8472
true_positives = int(recall * 500)  # assuming 500 actual spoofs
false_negatives = 500 - true_positives
false_positives = int((true_positives / precision) - true_positives)
true_negatives = 500 - false_positives

# Construct labels
y_true = [1] * true_positives + [1] * false_negatives + [0] * false_positives
     + [0] * true_negatives
y_pred = [1] * true_positives + [0] * false_negatives + [1] * false_positives
     + [0] * true_negatives

# Create confusion matrix
cm = confusion_matrix(y_true, y_pred)
labels = ['Bonafide (0)', 'Spoof (1)']

# Plot confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels,
    yticklabels=labels)
plt.xlabel("Predicted Label")
```

```python
plt.ylabel("True Label")
plt.title("Confusion Matrix: Binary Classification on Separated Audio")
plt.tight_layout()
plt.show()

cm

import matplotlib.pyplot as plt

# Epoch data
epochs = [1, 2, 3, 4, 5, 6]
train_loss_cumulative = [
    7731.957299,
    5657.795003,
    5119.997936,
    4750.488601,
    4626.121368,
    4915.015645
]
val_loss = [
    0.4292,
    0.4027894042,
    0.3836123315,
    0.3826045701,
    0.3826045701,
    0.3830747771
]
val_accuracy = [
    0.8028,
    0.81945,
    0.82795,
    0.8292,
    0.8292,
    0.83
]

# Average training loss
data_items = 15000
train_loss_avg = [loss / data_items for loss in train_loss_cumulative]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(epochs, train_loss_avg, label='Average Training Loss', marker='o')
plt.plot(epochs, val_loss, label='Validation Loss', marker='s')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Epoch vs Loss (Training & Validation)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```python

from numpy import array, float32
# y_pred =loaded from the testing code
# y_true =loaded from the testing code
# y_score= loaded from the testing code

from sklearn.metrics import accuracy_score, precision_score, recall_score,
    f1_score, roc_curve, confusion_matrix


y_true_numeric = [1 if label == 'spoof' else 0 for label in y_true]

# Step 2: Extract predicted probabilities for positive class ("spoof")
# y_score is list of arrays: [prob_class_0, prob_class_1]
y_score_positive = [prob[1] for prob in y_score]

# Step 3: Calculate metrics
accuracy = accuracy_score(y_true_numeric, y_pred)
precision = precision_score(y_true_numeric, y_pred)
recall = recall_score(y_true_numeric, y_pred)
f1 = f1_score(y_true_numeric, y_pred)
cm = confusion_matrix(y_true_numeric, y_pred)

fpr, tpr, thresholds = roc_curve(y_true_numeric, y_score_positive)

# Step 4: Print results
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("Confusion Matrix:\n", cm)
print("ROC Curve data:")

import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Confusion matrix values
cm = np.array([[3071, 461],
               [45, 26423]])

# Labels for the confusion matrix
labels = ['Genuine', 'Spoof']
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels,
    yticklabels=labels)

plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for Spoof Detection Model')
plt.show()
```

Listing A.5: Complete Code for Calculating Confusion matrix for pre-trained pipeline