First Data Pipeline

I. Introduction

If you were born into a Pakistani family, you would know what it means to be raised by a village. In Pakistan, families are big - 6 people in an average household kind of big. Even that statistic does not fully capture the depth and width of Pakistani families because our definition of family is not limited to the people we share our living quarters with. Rather, it extends far and wide to include each of our grandparents' households, their siblings, our parents' siblings and their spouses, our siblings and their spouses, neices, nephews, our parents' cousins, our cousins' and their spouses and children, friends, families of friends, so on and so forth.

The thing about being born in a large family is that since birth, you are hard-wired to remember the names, faces, voices of, and relations between each of your family members. I didn't realize how naturally and effortlessly it came to me until I hosted my Minerva friends over the summer. Their shock at my ability to navigate through the intricate web of my family was delightfully amusing.

"How can you keep track of so many people around you on any given day?"

"Just like that."

But really, *how*?

This Machine Learning project aims to see if keeping track of so many people is indeed an extraordinary feat undertaken only by ordinary Pakistani brain or is it something an ordinary Machine Learning model can do too.

Can a ML model, like me, pick out exactly who is speaking and what at a casual Pakistani family gathering when a bunch of people talk to and over each other all at once?

To tackle this big question, I start with a smaller, more tractable one first.

Can a ML model at least distinguish between the voices of different family members and friends?

To answer this question, I tapped into the voice messages sent by eight of my family members over Whatsapp. The dataset includes four voice messages for each of the eight family members, including that of my paternal grandfather, maternal grandmother, mother, sister, two roommates at Minerva, and two friends from back home. The total number of audio files amounted to 32.

II. Data Preparation: Converting my Family's Voices in a Format that Python can Hear

To build a Machine Learning model for classifying my family's voices, I downloaded the most recent voice message in each of the eight most recent chats on my Whatsapp. Unsurprisingly, because Pakistanis have a lot to say to their loved ones, each voice message ranged from a minute to three minutes long. I trimmed each message into four chunks of approximately ten seconds each and converted them from OPUS to wav format to be compatible with Python libraries.

Next, I arranged audio files in subfolders labeled with the name of the speaker within a common "data" folder. Then, I created a metadata csv file using a Python script copying the following important details for each audio file:

- File Path: The path to access the file in the directory.
- File Name: The name of the audio file in the directory.
- Label: The name of the subfolder housing the audio file (also the name of the speaker).
- Duration: The duration of the audio.
- The Sample Rate: The sample rate of the audio file i.e. the number of samples (data points) captured or played per second in the audio.

Later, I added a Target column to the csv to assign an integer label (0-7) to each of the eight audio files. I used integers as labels instead of one-hot encoding to minimze memory required and achieve computational efficiency.

In [ ]:

```
# Import necessary libraries
import math, random
import torch
import torchaudio
from torchaudio import transforms
from torch.utils.data import DataLoader, Dataset, random_split
import librosa
import librosa.display
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import IPython.display as ipd
from pathlib import Path
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init
import seaborn as sns
from sklearn.metrics import classification_report, confusion_matrix
```

I used Pandas to load the metadata.csv file as a dataframe for Python to read. The following code cell visualizes what this looks like.

In [ ]:
```
# The path where the metadata is stored
metadata_file = Path.cwd()/'metadata.csv'

# Read metadata file
df = pd.read_csv(metadata_file)

# Add a Target column with a unique integer label
df['Target'] = pd.factorize(df['Label'])[0]

# Print the dataframe
df.head(len(df))
```

Out[ ]:

	File Path	File Name	Label	Duration (seconds)	Sample Rate (Hz)	Target
0	C:\Users\alina\Downloads\assignment\data\Azam\...	Azam-3.wav	Azam	10.437833	48000	0
1	C:\Users\alina\Downloads\assignment\data\Azam\...	Azam-4.wav	Azam	10.397833	48000	0
2	C:\Users\alina\Downloads\assignment\data\Azam\...	Azam1.wav	Azam	10.397833	48000	0
3	C:\Users\alina\Downloads\assignment\data\Azam\...	Azam2.wav	Azam	10.397833	48000	0
4	C:\Users\alina\Downloads\assignment\data\Giova...	giovanna1.wav	Giovanna	10.377833	48000	1
5	C:\Users\alina\Downloads\assignment\data\Giova...	giovanna2.wav	Giovanna	10.217833	48000	1
6	C:\Users\alina\Downloads\assignment\data\Giova...	giovanna3.wav	Giovanna	10.397833	48000	1
7	C:\Users\alina\Downloads\assignment\data\Giova...	giovanna4.wav	Giovanna	10.417833	48000	1
8	C:\Users\alina\Downloads\assignment\data\Hassa...	hassanadnan1.wav	HassanAdnan	10.377833	48000	2
9	C:\Users\alina\Downloads\assignment\data\Hassa...	hassanadnan2.wav	HassanAdnan	10.357833	48000	2
10	C:\Users\alina\Downloads\assignment\data\Hassa...	hassanadnan3.wav	HassanAdnan	10.297833	48000	2
11	C:\Users\alina\Downloads\assignment\data\Hassa...	hassanadnan4.wav	HassanAdnan	10.637833	48000	2
12	C:\Users\alina\Downloads\assignment\data\Hassa...	hassan-1.wav	HassanBukhari	10.417833	48000	3
13	C:\Users\alina\Downloads\assignment\data\Hassa...	hassan-2.wav	HassanBukhari	11.077833	48000	3
14	C:\Users\alina\Downloads\assignment\data\Hassa...	hassan-3.wav	HassanBukhari	10.377833	48000	3
15	C:\Users\alina\Downloads\assignment\data\Hassa...	hassan-4.wav	HassanBukhari	11.057833	48000	3
16	C:\Users\alina\Downloads\assignment\data\Mahee...	Maheen1.wav	Maheen	10.597833	48000	4
17	C:\Users\alina\Downloads\assignment\data\Mahee...	Maheen2.wav	Maheen	10.797833	48000	4
18	C:\Users\alina\Downloads\assignment\data\Mahee...	Maheen3.wav	Maheen	10.457833	48000	4
19	C:\Users\alina\Downloads\assignment\data\Mahee...	Maheen4.wav	Maheen	10.457833	48000	4
20	C:\Users\alina\Downloads\assignment\data\Muniz...	muniza-1.wav	Muniza	10.737833	48000	5
21	C:\Users\alina\Downloads\assignment\data\Muniz...	muniza-2.wav	Muniza	10.377833	48000	5
22	C:\Users\alina\Downloads\assignment\data\Muniz...	Muniza-3.wav	Muniza	10.257833	48000	5

	File Path	File Name	Label	Duration (seconds)	Sample Rate (Hz)	Target
23	C:\Users\alina\Downloads\assignment\data\Muniz...	muniza-4.wav	Muniza	10.437833	48000	5
24	C:\Users\alina\Downloads\assignment\data\Mussa...	Mussarat1.wav	Mussarat	10.357833	48000	6
25	C:\Users\alina\Downloads\assignment\data\Mussa...	mussarat2.wav	Mussarat	10.077833	48000	6
26	C:\Users\alina\Downloads\assignment\data\Mussa...	mussarat3.wav	Mussarat	10.397833	48000	6
27	C:\Users\alina\Downloads\assignment\data\Mussa...	Mussarat4.wav	Mussarat	10.497833	48000	6
28	C:\Users\alina\Downloads\assignment\data\Sana\...	sana1.wav	Sana	10.297833	48000	7
29	C:\Users\alina\Downloads\assignment\data\Sana\...	sana2.wav	Sana	10.417833	48000	7
30	C:\Users\alina\Downloads\assignment\data\Sana\...	sana3.wav	Sana	10.297833	48000	7
31	C:\Users\alina\Downloads\assignment\data\Sana\...	sana4.wav	Sana	10.397833	48000	7

To hear what I would be working with, I used the Librosa library to sample five audio files.

In [ ]:

```
samples = df.sample(5)[['File Name','Label']]

for i, idx in enumerate(samples.index):
    # Loop through the index of 'samples'

    # Load the audio file and grab the category
    filename = df['File Path'][idx]
    category = df['Label'][idx]

    # Load the audio data and sampling rate using the librosa library
    y, sr = librosa.load(filename)

    # Create an audio display for the loaded audio data
    audio_display = ipd.Audio(data=y, rate=sr)

    # Display the audio
    display(audio_display)

    # Print the category or class associated with the audio
    print(f"Class: {category}")
```

0:00 / 0:10

```
Class: Giovanna
```

0:00 / 0:10

```
Class: Maheen
```

0:00 / 0:10

```
Class: HassanAdnan
```

0:00 / 0:10

```
Class: Maheen
```

0:00 / 0:10

```
Class: Azam
```

Unlike me, computers cannot hear the audio files. Rather, they visualize it. Therefore, I used the Librosa library to visualize audio files in five different ways Python would want to see them.

The following code cell displays five different representations for each of the five samples taken above using various visualization functions from the librosa library.

For each audio sample:

- It displays the raw audio signal of the sample.
- Computes the mel spectrogram (melspec) in decibels (dB): A representation of how different frequency components in an audio signal change over time, with colors indicating their intensity.
- Calculates the Mel-frequency cepstral coefficients (MFCCs): A representation of audio as a set of coefficients, typically comprising 10 to 20 features, encapsulating the fundamental characteristics of the general shape or contour of an audio signal's frequency spectrum such as characterizing the overall timbre or tonal qualities of the sound.

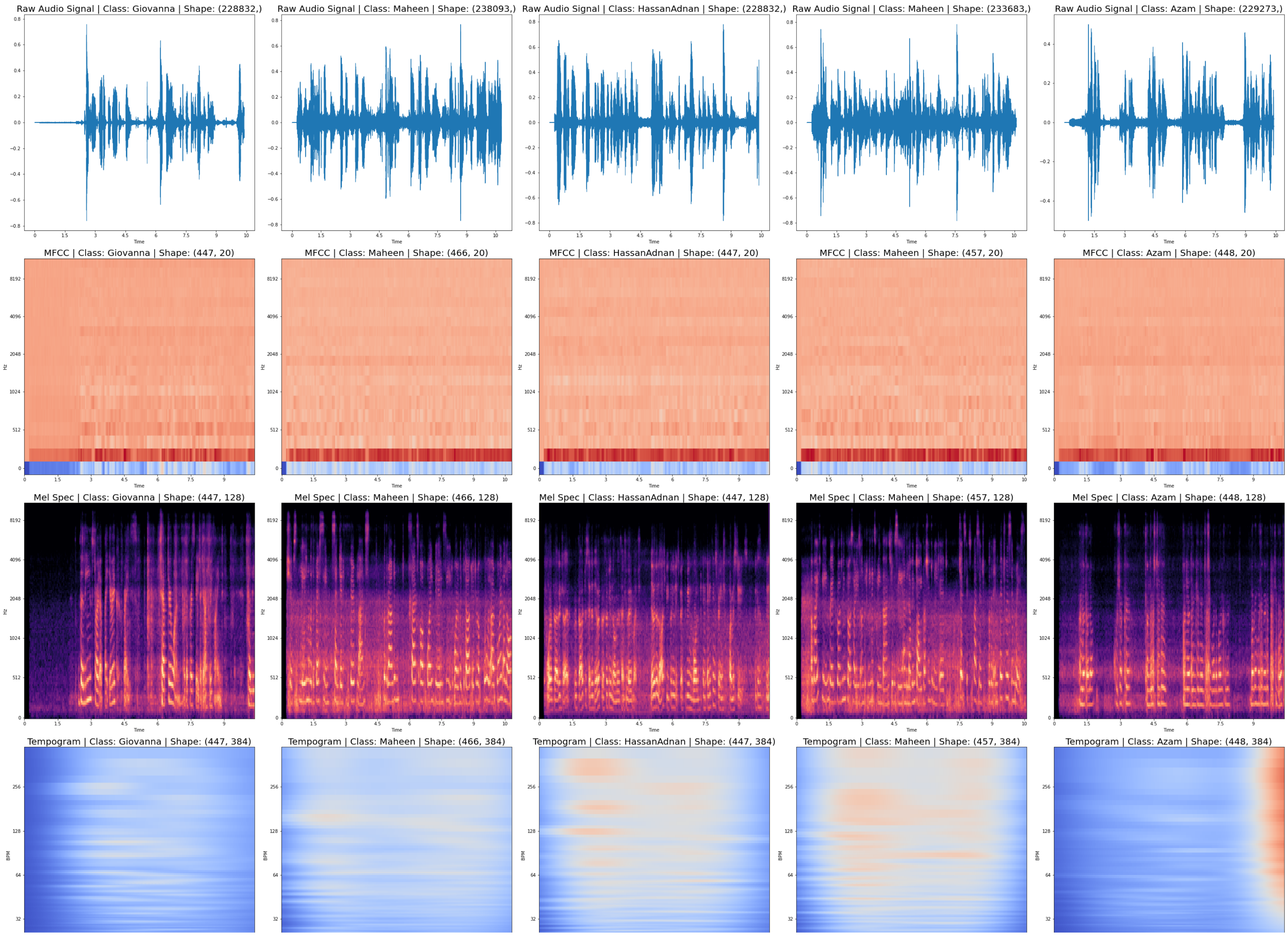
- Computes the tempogram: A representation of an audio signal emphasizing rhythmic patterns and tempo variations by measuring how tempo evolves over time.
- Computes the chromagram: A representation of the pitch content of an audio signal, typically divided into 12 bins to capture the presence and intensity of different musical pitches.

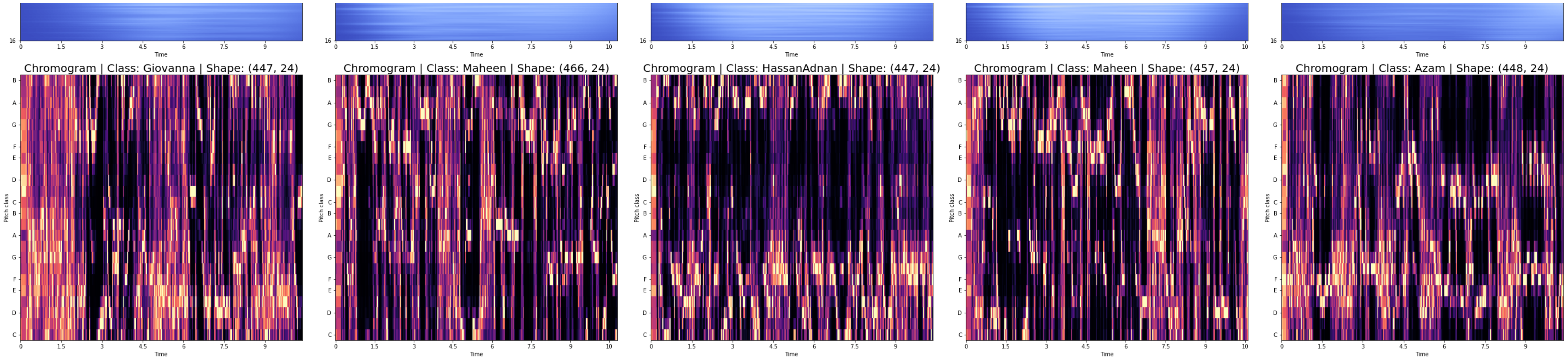
In [ ]:

```
# Create a 5x5 grid of subplots
fig, ax = plt.subplots(nrows=5, ncols=5, figsize=(40, 40))

# Iterate through the audio samples
for i, idx in enumerate(samples.index):
    # Load the audio file and obtain its category
    filename = df['File Path'][idx]
    category = df['Label'][idx]

    # Load the audio data and compute the mel spectrogram in decibels
    y, sr = librosa.load(filename)
    melspec = librosa.feature.melspectrogram(y=y, sr=sr, n_fft=2048, hop_length=512, n_mels=128)
    melspec = librosa.power_to_db(melspec, ref=np.max)

    # Calculate the Mel-frequency cepstral coefficients (MFCCs)
    mfcc = librosa.feature.mfcc(y=y, sr=sr, n_fft=2048, hop_length=512, n_mels=128)

    # Compute the tempogram to represent tempo variations
    oenv = librosa.onset.onset_strength(y=y, sr=sr, hop_length=512)
    tempogram = librosa.feature.tempogram(onset_envelope=oenv, sr=sr, hop_length=512, norm=None)

    # Generate the chromagram for pitch content
    chromagram = librosa.feature.chroma_stft(y=y, sr=sr, n_fft=2048, hop_length=512, n_chroma=24)

    # Plot the raw audio waveform
    librosa.display.waveshow(y, sr=sr, ax=ax[0][i])

    # Plot the MFCCs
    librosa.display.specshow(mfcc, sr=sr, ax=ax[1][i], y_axis='mel', x_axis='time')

    # Plot the mel spectrogram
    librosa.display.specshow(melspec, sr=sr, ax=ax[2][i], y_axis='mel', x_axis='time')

    # Plot the tempogram
    librosa.display.specshow(tempogram, sr=sr, hop_length=512, x_axis='time', y_axis='tempo', ax=ax[3][i])
```

```python
    # Plot the chromagram
    librosa.display.specshow(chromagram, sr=sr, hop_length=512, x_axis='time', y_axis='chroma', ax=ax[4][i])

    # Set titles for each subplot
    ax[0][i].set_title(f"Raw Audio Signal | Class: {category} | Shape: {y.shape}", fontsize=20)
    ax[1][i].set_title(f"MFCC | Class: {category} | Shape: {mfcc.T.shape}", fontsize=20)
    ax[2][i].set_title(f"Mel Spec | Class: {category} | Shape: {melspec.T.shape}", fontsize=20)
    ax[3][i].set_title(f"Tempogram | Class: {category} | Shape: {tempogram.T.shape}", fontsize=20)
    ax[4][i].set_title(f"Chromogram | Class: {category} | Shape: {chromagram.T.shape}", fontsize=20)

# Adjust subplot spacing
plt.subplots_adjust(wspace=0.4)

# Set the main title for the figure
fig.suptitle('Five Audio Samples, 5 Different Representations\n\n', fontsize=40)

# Adjust the layout of the subplots
fig.tight_layout()
```

Five Audio Samples, 5 Different Representations





III. Pre-Processing: Preparing my Family's Voices for Python to Understand

I crafted a dedicated class, AudioProcessing, to facilitate the essential steps of audio data preprocessing for machine learning applications. This class streamlines the preparation of audio data to ensure it is compatible with deep learning models.

Here's an overview of the key functionalities and their significance:

- Accessing audio files:

open(audio_file): This method loads an audio file, extracting the audio signal as a tensor and the sample rate. It forms the foundation for further processing.

- Augmenting raw audio data for a robust, generalizable model:

time_shift(audio, shift_limit): Time shifting is applied to the audio signal, introducing variations for enhanced model training. This helps the model generalize better by exposing it to shifted versions of the audio, enabling it to learn invariances.

- Converting audio file into a Python readable format:

create_melspec(audio, n_mels, n_fft, hop_len): This function creates Mel spectrograms from audio signals. These spectrograms serve as crucial inputs for deep learning models, revealing the audio's frequency characteristics over time.

- Augmenting Mel spectrograms to diversify audio data:

spec_augment(melspec, max_mask, num_freq_masks, num_time_masks): Data augmentation is performed on Mel spectrograms, increasing the diversity of training data by generating different variation. Frequency and time masking are applied to enhance the model's robustness and generalizability.

- Standardizing characteristics of audio files

standardize_channel(audio, num_channels): To ensure uniformity in the dataset, this method standardizes the number of audio channels. It either converts audio to mono or duplicates channels as needed for stereo.

standardize_sr(audio, new_sr): This function resamples the audio to match the desired sample rate, maintaining data consistency and enabling model compatibility.

pad_trunc(audio, max_ms): This method either pads or truncates the audio signal to achieve the desired maximum length, ensuring uniformity in the dataset.

I exclusively worked with PyTorch as it is better suited for end-to-end creation of models using tensors than the general purpose Librosa library.

In [ ]:

```
# Create a class for audio processing
class AudioProcessing():
  """
  A class for audio processing utilities.
  """
  @staticmethod
  def open(audio_file):
      """
    Load an audio file and return the signal as a tensor and the sample rate.

    Args:
        audio_file (str): The path to the audio file to be loaded.

    Returns:
        tuple: A tuple containing two elements:
            - torch.Tensor: The audio signal loaded from the file.
            - int: The sample rate of the audio.
        """
    # Load an audio file using torchaudio.load
    signal, sr = torchaudio.load(audio_file)
    # Return the audio signal (as a tensor) and the sample rate
    return (signal, sr)

  # Data Augmentation on Raw Data
```

```python
@staticmethod
def time_shift(audio, shift_limit):
    """
  Apply time shifting to an audio signal.

  Args:
      audio (tuple): A tuple containing the audio signal as a tensor and the sample rate.
      shift_limit (float): The maximum allowable time shift as a fraction of the signal's length.

  Returns:
      tuple: A tuple containing two elements:
          - torch.Tensor: The time-shifted audio signal.
          - int: The sample rate of the audio.
      """
    # Unpack the audio parameter into signal and sample rate (sr)
    signal, sr = audio
    # Calculate the length of the signal in terms of time steps
    _, signal_length = signal.shape
    # Calculate the amount by which the audio signal will be shifted
    shift_amount = int(random.random() * shift_limit * signal_length)
    # Apply time shifting to the audio signal using roll
    shifted_signal = signal.roll(shift_amount)
    # Return the shifted audio signal along with the original sample rate
    return (shifted_signal, sr)

@staticmethod
def create_melspec(audio, n_mels=128, n_fft=2048, hop_len=512):
    """
  Create a Mel spectrogram from an audio signal.

  Args:
      audio (tuple): A tuple containing the audio signal as a tensor and the sample rate.
      n_mels (int, optional): The number of mel bands to generate in the spectrogram. Defaults to 128.
      n_fft (int, optional): The number of samples in each short-time Fourier transform (STFT). Defaults to 2048.
      hop_len (int, optional): The hop length (stride) for the STFT. Defaults to 512.

  Returns:
      torch.Tensor: The Mel spectrogram of the audio signal converted to decibels.

  """
    # Unpack the audio parameter into signal and sample rate (sr)
    signal, sr = audio
```

```
    # Define the desired top dB value for AmplitudeToDB conversion
    top_db = 80

    # Calculate the Mel spectrogram from the audio signal
    mel_spec = transforms.MelSpectrogram(sr, n_fft=n_fft, hop_length=hop_len, n_mels=n_mels)(signal)

    # Convert the Mel spectrogram to decibels (AmplitudeToDB transformation)
    mel_spec = transforms.AmplitudeToDB(top_db=top_db)(mel_spec)

    # Return the resulting Mel spectrogram
    return (mel_spec)



# Data Augmentation on MelSpec


@staticmethod
def spec_augment(melspec, max_mask=0.1, num_freq_masks = 1, num_time_masks=1):
  """
  Apply frequency and time masking to a Mel spectrogram.
  """


  _, n_mels, n_steps = melspec.shape
  mask_value = melspec.mean()
  aug_spec = melspec

  freq_mask_param = max_mask * n_mels
  for _ in range(num_freq_masks):
    aug_spec = transforms.FrequencyMasking(freq_mask_param)(aug_spec, mask_value)

  time_mask_param = max_mask * n_steps
  for _ in range(num_time_masks):
    aug_spec = transforms.TimeMasking(time_mask_param)(aug_spec, mask_value)

  return aug_spec

@staticmethod
def standardize_channel(audio, num_channels):
  """
  Standardize the number of channels in an audio signal.

  Args:
```

```
        audio (tuple): A tuple containing the audio signal as a tensor and the sample rate.
        num_channels (int): The desired number of audio channels (1 for mono, 2 for stereo).

    Returns:
        tuple: A tuple containing the standardized audio signal and sample rate.
    """

    signal, sr = audio

    if signal.shape[0] == num_channels:
        # If the number of channels matches the desired num_channels, return the original audio
        return audio

    if num_channels == 1:
        # If num_channels is 1, convert to mono
        new_signal = signal[:1,:]
    else:
       new_signal = torch.cat([signal,signal])

    return ((new_signal, sr))

  @staticmethod
  def standardize_sr(audio, new_sr):
     """
     tandardize the sample rate of an audio signal.

    Args:
        audio (tuple): A tuple containing the audio signal as a tensor and the current sample rate.
        new_sr (int): The desired sample rate.

    Returns:
        tuple: A tuple containing the audio signal with the standardized sample rate and the new sample rate.
     """
     signal, sr = audio

     if sr == new_sr:
        return audio

     num_channels = signal.shape[0]
     new_signal = torchaudio.transforms.Resample(sr,new_sr)(signal[:1:])
     if num_channels >1:
        resample_two = torchaudio.transforms.Resample(sr,new_sr)(signal[1:,:])
        new_signal = torch.cat([new_signal,resample_two])
```

```
        return ((new_signal,new_sr))

    @staticmethod
    def pad_trunc(audio, max_ms):
        """
        Pad or truncate an audio signal to match a specified duration.

        Args:
            audio (tuple): A tuple containing the audio signal as a tensor and the sample rate.
            max_ms (int): The maximum duration in milliseconds.

        Returns:
            tuple: A tuple containing the padded or truncated audio signal tensor and sample rate.
        """
        signal, sr = audio
        num_rows, signal_len = signal.shape
        max_len = sr//1000 * max_ms

        if (signal_len > max_len):
          # Truncate the signal to the given length
          signal = signal[:,:max_len]

        elif (signal_len < max_len):
          # Length of padding to add at the beginning and end of the signal
          pad_begin_len = random.randint(0, max_len - signal_len)
          pad_end_len = max_len - signal_len - pad_begin_len

          # Pad with 0s
          pad_begin = torch.zeros((num_rows, pad_begin_len))
          pad_end = torch.zeros((num_rows, pad_end_len))

          signal = torch.cat((pad_begin, signal, pad_end), 1)

        return (signal, sr)
```

Complementing the AudioProcessing class is the AudioDataset class, designed to streamline data management. It prepares the audio data for model training, handling tasks such as loading, standardization, duration adjustment, time shifting, and data augmentation. It ensures that audio data is not only ready for analysis but also optimized to enhance model performance and robustness.

In [ ]:
```
class AudioDataset(Dataset):
    def __init__(self, df):
        """
        Initialize an AudioDataset.

        Args:
            df (DataFrame): A DataFrame containing audio file information.

        Initializes an AudioDataset with provided parameters for sample rate, number of audio channels, audio duration,
        and data augmentation shift percentage.
        """
        self.df = df
        self.sr = 44100  # Sample rate
        self.channel = 2  # Number of audio channels (2 for stereo)
        self.duration = 10_000  # Audio duration in milliseconds
        self.shift_pct = 0.4  # Data augmentation shift percentage

    def __len__(self):
        """
        Return the number of samples in the dataset.

        Returns:
            int: The number of samples in the dataset.
        """
        return len(self.df)

    def __getitem__(self, index):
        """
        Get a sample from the dataset.

        Args:
            index (int): The index of the sample to retrieve.

        Returns:
            tuple: A tuple containing two elements:
                - torch.Tensor: The augmented Mel spectrogram of the audio sample.
                - int: The target label associated with the sample.

        """
        audio_file = self.df.loc[index, 'File Path']
        target = self.df.loc[index, 'Target']
```

```python
        # Access audio file
        audio = AudioProcessing.open(audio_file)

        # Standardize sample rate
        standardize_sr = AudioProcessing.standardize_sr(audio, self.sr)

        # Standardize channel
        standardize_channel = AudioProcessing.standardize_channel(standardize_sr, self.channel)

        # Standardize duration
        standardize_duration = AudioProcessing.pad_trunc(standardize_channel, self.duration)

        # Augment raw data
        shift_audio = AudioProcessing.time_shift(standardize_duration, self.shift_pct)

        # Create a Mel spectrogram
        mel_spectrogram = AudioProcessing.create_melspec(shift_audio, n_mels=64, n_fft=1024, hop_len=None)

        # Augment Mel spectrogram
        augmented_spec = AudioProcessing.spec_augment(mel_spectrogram, max_mask=0.1, num_freq_masks=2, num_time_masks=2)

        return augmented_spec, target
```

IV. Model Creation: Creating a Deep Learning Model to Classify my Family's Voices

The task of choice is that of classification. I want to see if a Machine Learning model is able to correctly distinguish between the voices of different family members, classifying the audio files based on their speakers.

I decided to use Convolutional Neural Networks (CNNs) to undertake the task at hand, simply because they are more powerful than traditional Feedforward Neural Networks. Because of their ability to learn patterns that are translation invariant and have spatial hierarchies (F. Chollet, 2018), CNNs do a great job at classifying images and are therefore best-suited for classifying audio files based on their Mel spectrograms.

There are four types of layers in Convolutional Neural Networks (CNN):

1. Convolution Layers:

These puts the input image through convolutional filters, each filter activating certain features such as edges, colors, or objects. The idea is to extract features from the input image via a kernel (filter) and generate feature maps.

A kernel is a small matrix which has height and width smaller than the input image. The kernel moves across the height and width of the input image and dot product of the kernel and the image are computed for every spatial position, as shown by the formula below, where f is the input image, h is the kernel, and m and n are the indexes of rows and columns of the resulting matrix:

$$G[m,n] = fh[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k]$$

The result is a convolved feature. The kernel size and stride length are parameters used to generate the convolved feature.

Here, the network learns the features and activates when they see a specific type of feature at a given spatial location in the input image.

2. Rectified Linear Activation Function

This layer consists of nodes or units that implement an activation function known as ReLU. The activation function is a nonlinear function with all the desirable properties of linear function, enabling optimization wth gradient-based methods. It is a piecewise functionwhich outputs the input if greater than 0.0 and outputs 0.0 if the input is 0.0 or less:

$$g(z) = max\{0, z\}$$

This is where activation takes place i.e. only the features activated by the convolution layer re carried forward into the next layer.

3. Batch Normalization (between Activation and Pooling layer)

This layer stablizes the network during training by noralizing the data to zero mean and unit variance. For each feature column, the mean an dvariance of all the samples in the dataset are computed and then normalized:

$$\hat{A}_i = \frac{A_i - \mu_i}{\sigma_i}$$

This ensures that all feature values are on the same scale so the network can learn weights for each feature on the same scale and produce a linear combination of each feature vector much more efficiently.

The Kaiming normal initialization method is used to initialize the weights of the first convolution layer. The bias terms of the first convolution layer are intiailized to zero to align with the weight initialization done by Kaiming initialization.

These layers are stored in a list to apply them in a sequential manner.

4. Adaptive Pooling Layer

This layer flattens the feature map i.e. reduces the spatial dimensions while retaining the most important features. Themain purpose is to reduce the size of the tensor and speed up calculations. The output size of this layer would be (1,1).

5. Linear Layer

The linear layer outputs one prediction per class, completing the classification problem.

In [ ]:
```
class AudioClassifier (nn.Module):
    # Build the model architecture
    def __init__(self):
        """
        Initialize the AudioClassifier model.

        The model architecture consists of several convolutional layers followed by an adaptive average pooling layer
        and a linear classifier.

        The convolutional layers use ReLU activation functions and batch normalization. Kaiming initialization is
        applied to the convolutional layer weights.

        The final output is a classification result with 8 classes.

        """
        super().__init__()
        conv_layers = []

        # First Convolution Block

        # Create a 2D convolutional layer with 2 input channels, 8 output channels, a 5x5 kernel, and specified stride and padding.
        self.conv1 = nn.Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
        # Apply the Rectified Linear Unit (ReLU) activation function to introduce non-linearity.
        self.relu1 = nn.ReLU()
        # Apply Batch Normalization with 8 features.
        self.bn1 = nn.BatchNorm2d(8)
        # Initialize the weights of the convolution layer using Kaiming initialization
        init.kaiming_normal_(self.conv1.weight, a=0.1)
        # Initialize the bias terms of the convolution layer to zeros.
        self.conv1.bias.data.zero_()
        # Add the layers (convolution, activation, and batch norm) to the list 'conv_layers'.
        conv_layers += [self.conv1, self.relu1, self.bn1]

        # Second Convolution Block
```

```python
        self.conv2 = nn.Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(16)
        init.kaiming_normal_(self.conv2.weight, a=0.1)
        self.conv2.bias.data.zero_()
        conv_layers += [self.conv2, self.relu2, self.bn2]

        # Third Convolution Block
        self.conv3 = nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu3 = nn.ReLU()
        self.bn3 = nn.BatchNorm2d(32)
        init.kaiming_normal_(self.conv3.weight, a=0.1)
        self.conv3.bias.data.zero_()
        conv_layers += [self.conv3, self.relu3, self.bn3]

        # Fourth Convolution Block
        self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu4 = nn.ReLU()
        self.bn4 = nn.BatchNorm2d(64)
        init.kaiming_normal_(self.conv4.weight, a=0.1)
        self.conv4.bias.data.zero_()
        conv_layers += [self.conv4, self.relu4, self.bn4]

        # Pooling and Linear Classifier
        self.ap = nn.AdaptiveAvgPool2d(output_size=1)
        self.lin = nn.Linear(in_features=64, out_features=8)

        # Wrap the Convolutional Blocks
        self.conv = nn.Sequential(*conv_layers)

    # Forward pass computations
    def forward(self, x):
        """
        Forward pass computations for the AudioClassifier model.

        Args:
            x (torch.Tensor): Input tensor.

        Returns:
            torch.Tensor: Output tensor representing classification results.

        """
        # Run the convolutional blocks
```

```
        x = self.conv(x)

        # Adaptive pool and flatten for input to linear layer
        x = self.ap(x)
        x = x.view(x.shape[0], -1)

        # Linear layer
        x = self.lin(x)

        # Final output
        return x
```

V. Training: Learning the Voices of My Family

I used torch's DataLoader to load my dataset in batches of 2 for the training of the model. In batches, the DataLoder applies the pre-processing methods from the AudioProcessing class on the AudioDataset object, outputing transformed data which can directly be fed to the CNN model. I randomly split my dataset into training (80%) and validation (20%) set.

For training, I used Cross Entropy Loss, Adam optimization algorithm, and One Cycle Learning Rate Policy:

- Cross-Entropy Loss: Sparse Categorical Cross Entropy is used a loss function. The purpose of this function is to minimize the difference between predicted class probabilities and actual class labels. During training, the "loss" i.e. the difference between predicted probabilities and actual class labels is minimzed. Cross Entropy Loss is used to output a probability over the 8 classes for each Mel spectrogram. Cross Entropy Loss decreases as the predicted labels converge towards the true label. The equation used to calculate cCross Entropy Loss is as follows:

$$CE = -\sum_{i=1}^{N} y_{true_i} \cdot log(y_{pred_i})$$

- Adam Optimizer: Adaptive Moment Estomation optimizer extends the stochastic gradience descent (SDG) algorithm to update the weights during training. The ky difference between SDG and Adam Optimizer is that the SDG maintains a single learning rate throughout training but Adam optimizer computes and adjusts individul learning rate for each network weight individually. It operates the on the following formula, where β_1 and β_2 are the decay rate of gradient average:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)[\frac{\delta L}{\delta w_t}] v_t = \beta_2 v_{t-1} + (1-\beta_2)[\frac{\delta L}{\delta w_t}]^2$$

- One-Cycle Learning Rate Policy: Cyclic learning rate is used to get the learning rate just right for optimization.

In [ ]:
```
# Define the audio dataset using the provided DataFrame
dataset = AudioDataset(df)
```

In [ ]:
```
test_file = Path.cwd()/'test.csv'
test = pd.read_csv(test_file)
test['Target'] = pd.factorize(test['Label'])[0]

test_dataset = AudioDataset(test)
test_dl = torch.utils.data.DataLoader(test_dataset, batch_size=2,shuffle=False)
```

In [ ]:
```
# Calculate the total number of items in the dataset
num_items = len(dataset)
# Determine the number of items for training (80% of the dataset)
num_train = round(num_items * 0.8)
# Calculate the number of items for validation (remaining 20%)
num_validation = num_items - num_train

tdata, vdata = random_split(dataset, [num_train, num_validation])

# Create data loaders for training and validation

# Create a data loader for training with batch size 8
train_dl = torch.utils.data.DataLoader(tdata, batch_size=8, shuffle=True)
# Create a data loader for validation with batch size 8
val_dl = torch.utils.data.DataLoader(vdata, batch_size=8, shuffle=False)
```

In [ ]:
```
# Create the model and put it on the GPU if available
myModel = AudioClassifier()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
myModel = myModel.to(device)
```

In [ ]:
```
def training(model, train_dl, num_epochs):
  # Loss Function, Optimizer and Scheduler
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(),lr=0.001)
  scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.001,
                                                  steps_per_epoch=int(len(train_dl)),
                                                  epochs=num_epochs,
                                                  anneal_strategy='linear')
  train_acc_history = []
  # Repeat for each epoch
```

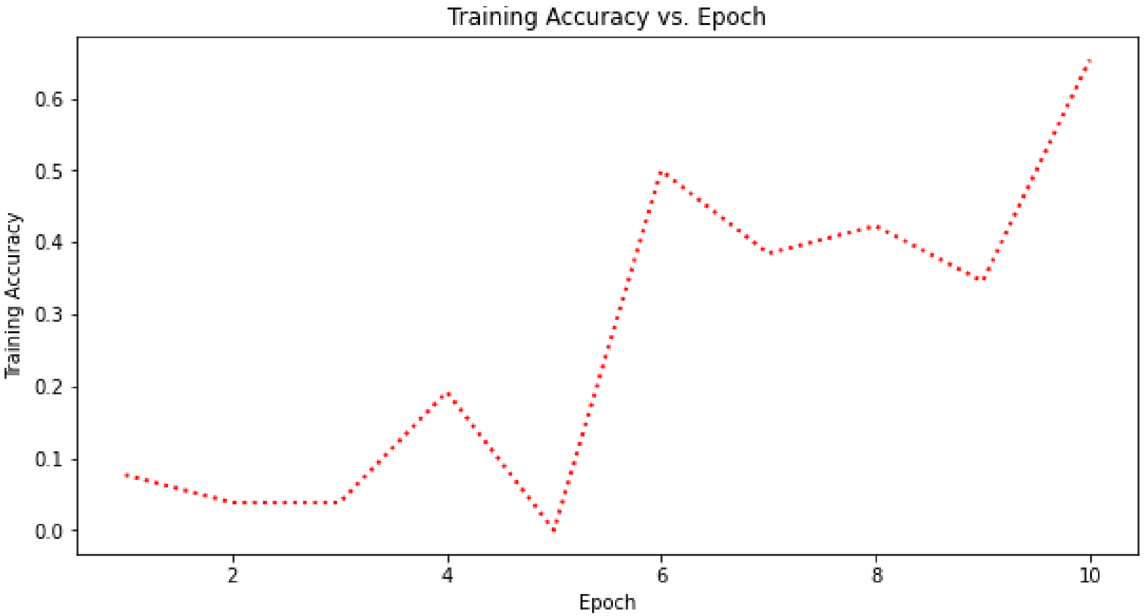
```python
for epoch in range(num_epochs):
  running_loss = 0.0
  correct_prediction = 0
  total_prediction = 0

  # Repeat for each batch in the training set
  for i, data in enumerate(train_dl):
      # Get the input features and target labels, and put them on the GPU
      inputs, labels = data[0].to(device), data[1].to(device)

      # Normalize the inputs
      inputs_m, inputs_s = inputs.mean(), inputs.std()
      inputs = (inputs - inputs_m) / inputs_s

      # Zero the parameter gradients
      optimizer.zero_grad()

      # forward + backward + optimize
      outputs = model(inputs)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()
      scheduler.step()

      # Keep stats for Loss and Accuracy
      running_loss += loss.item()

      # Get the predicted class with the highest score
      _, prediction = torch.max(outputs,1)
      # Count of predictions that matched the target label
      correct_prediction += (prediction == labels).sum().item()
      total_prediction += prediction.shape[0]

  # Print stats at the end of the epoch
  num_batches = len(train_dl)
  avg_loss = running_loss / num_batches
  acc = correct_prediction/total_prediction
  train_acc_history.append(acc)
  print(f'Epoch: {epoch+1}, Loss: {avg_loss:.2f}, Accuracy: {acc:.2f}')

print('Finished Training')
plt.figure(figsize=(10,5))
plt.plot(range(1, num_epochs + 1), train_acc_history, lw=2, linestyle='dotted',color='red')
```

```
plt.xlabel('Epoch')
plt.ylabel('Training Accuracy')
plt.title('Training Accuracy vs. Epoch')
plt.show()
```

The model is trained for 10 epochs. By the end, the accuracy achieved is 65%. Given the small dataset, 10 epochs are suitable as a greater number of epochs would result in the model learning the data or overfitting.

In [ ]:
```
num_epochs= 10
training(myModel, train_dl, num_epochs)
```

```
Epoch: 1, Loss: 2.09, Accuracy: 0.08
Epoch: 2, Loss: 2.07, Accuracy: 0.04
Epoch: 3, Loss: 2.08, Accuracy: 0.04
Epoch: 4, Loss: 2.05, Accuracy: 0.19
Epoch: 5, Loss: 2.02, Accuracy: 0.00
Epoch: 6, Loss: 1.96, Accuracy: 0.50
Epoch: 7, Loss: 2.01, Accuracy: 0.38
Epoch: 8, Loss: 1.99, Accuracy: 0.42
Epoch: 9, Loss: 1.94, Accuracy: 0.35
Epoch: 10, Loss: 1.94, Accuracy: 0.65
Finished Training
```



After training, the model is validated on validation dataset (20% of the original split). The accuracy achieved is 33%, a very poor score.

In [ ]:
```
def predict(model, val_dl):
  correct_prediction = 0
  total_prediction = 0

  # Disable gradient updates
  with torch.no_grad():
    for data in val_dl:
      # Get the input features and target labels, and put them on the GPU
      inputs, labels = data[0].to(device), data[1].to(device)

      # Normalize the inputs
      inputs_m, inputs_s = inputs.mean(), inputs.std()
      inputs = (inputs - inputs_m) / inputs_s

      # Get predictions
      outputs = model(inputs)

      # Get the predicted class with the highest score
      _, prediction = torch.max(outputs,1)
      # Count of predictions that matched the target label
      correct_prediction += (prediction == labels).sum().item()
      total_prediction += prediction.shape[0]

  acc = correct_prediction/total_prediction
  print(f'Accuracy: {acc:.2f}, Total items: {total_prediction}')

# Run inference on trained model with the validation set
predict(myModel, val_dl)
```

```
Accuracy: 0.33, Total items: 6
```

VI. Testing: The Moment of Truth

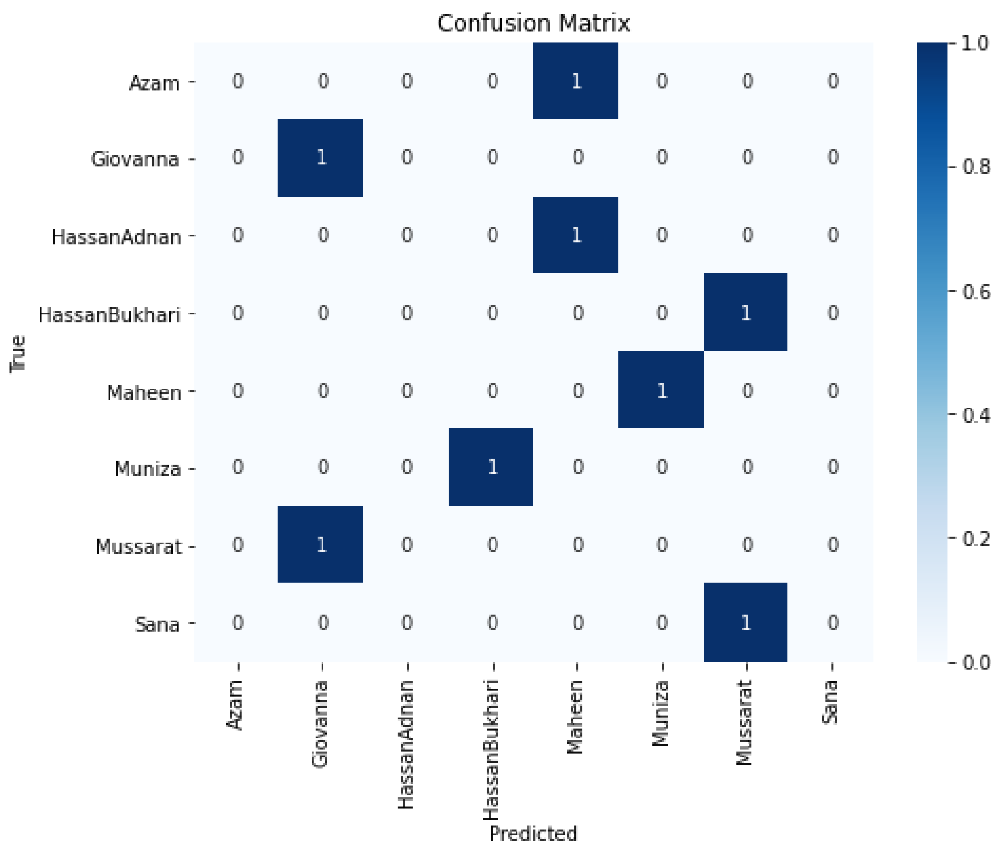
The testing of the model revealed that my model, in fact, is not as good as me in distinguishing the voices of my beloved family and friends. Where I have an accuracy close to 100%, my model barely achieved 12% of accuracy.

It is important to note though that the comparison is not fair. While my mind had trained on the sounds of my family's voices since I was in my mother's womb and spent the next 21 years learning the inflections and emotions in each, the model had only four 10-second samples per person to learn from and only one sample per class to try it's expertise on. My hypothesis was that the very low accuracy of the model can hence easily be attributed to the small training dataset and even smaller testing dataset.

Since I draw inspiration for this reasoning from cognitive scientists who test the effectivenss of their methods and measures to gain confidence in their findings about cognition, I found it fit to test my model on an existing sound dataset. I tried downloading the 'UrbanSoundDataset8k' which is a publicly available dataset of city sounds but it took excruciatingly long to get everything downloaded. In the interest of time, I have left the trial and perfection of this model for upcoming data pipelines, hoping to build an excellent audio classifier with sentiment analysis for the second pipeline, and a generative audio model for the final project.

In [ ]:
```
def test(model, test_dl, device):
    correct_predictions = 0
    total_predictions = 0
    true_labels = []
    predicted_labels = []

    # Disable gradient updates
    model.eval()  # Set the model to evaluation mode
    with torch.no_grad():
        for data in test_dl:
            inputs, labels = data[0].to(device), data[1].to(device)

            # Normalize the inputs
            inputs_m, inputs_s = inputs.mean(), inputs.std()
            inputs = (inputs - inputs_m) / inputs_s

            # Get predictions
            outputs = model(inputs)

            # Get the predicted class with the highest score
            _, predictions = torch.max(outputs, 1)

            # Count of predictions that matched the target label
            correct_predictions += (predictions == labels).sum().item()
            total_predictions += predictions.shape[0]

            true_labels.extend(labels.cpu().numpy())
            predicted_labels.extend(predictions.cpu().numpy())


    cm = confusion_matrix(true_labels, predicted_labels)
```

```python
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=target_names, yticklabels=target_names)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.show()

    accuracy = correct_predictions / total_predictions
    print(f'Accuracy: {accuracy:.2f}, Total items: {total_predictions}')

    # Calculate precision, recall, and F1 score
    classification_rep = classification_report(true_labels, predicted_labels, target_names=target_names,zero_division=0)
    print(classification_rep)

# You'll need to set the target_names to the names of your classes.
target_names = df['Label'].unique()
target_names = [str(target) for target in target_names]


# Run testing on your trained model with the test set
test(myModel, test_dl, device)
```




```
Accuracy: 0.12, Total items: 8
               precision    recall  f1-score   support

         Azam       0.00      0.00      0.00         1
     Giovanna       0.50      1.00      0.67         1
  HassanAdnan       0.00      0.00      0.00         1
HassanBukhari       0.00      0.00      0.00         1
       Maheen       0.00      0.00      0.00         1
       Muniza       0.00      0.00      0.00         1
     Mussarat       0.00      0.00      0.00         1
         Sana       0.00      0.00      0.00         1

     accuracy                           0.12         8
    macro avg       0.06      0.12      0.08         8
 weighted avg       0.06      0.12      0.08         8
```

VII. Executive Summary

The Machine Learning project started with an aim to classify voices of my family and friends based on voice messages I received from them on Whatsapp. I used Librosa library to visualize the various representations of the raw audio data and used Pytorch and torchaudio to pre-process the data into Python readable format. I converted audio files into Mel spectrograms and trained a Convolutional Neural Network on the Mel spectrograms. The testing accuracy achieved by the CNN is quite low and would benefit from a larger dataset.

IX. References

Nandi, P. (2021, December 10). CNNs for audio classification. Medium. https://towardsdatascience.com/cnns-for-audio-classification-6244954665ab

Doshi, K. (2021, May 21). Audio deep learning made simple: Sound classification, step-by-step. Medium. https://towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-cebc936bbe5

Gupta, A. (2021, October 7). A comprehensive guide on optimizers in deep learning. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/

One-cycle learning rate schedulers. (n.d.). https://kaggle.com/code/residentmario/one-cycle-learning-rate-schedulers

OpenAI. (2023). ChatGPT (September 25 Version) [Large language model]. https://chat.openai.com

Shakhadri, S. A. G. (2022, April 25). Guide to audio classification using deep learning. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2022/04/guide-to-audio-classification-using-deep-learning/

Skalski, P. (2019, April 14). Gentle dive into math behind convolutional neural networks. Medium. https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9

Tanksale, N. (2022, October 19). Finding good learning rate and the one cycle policy. Medium. https://towardsdatascience.com/finding-good-learning-rate-and-the-one-cycle-policy-7159fe1db5d6

Urban Sound Datasets. (n.d.). Urban Sound Datasets. https://urbansounddataset.weebly.com/download-urbansound8k.html

AI Use: ChatGPT September 25 Version was used to help comment the code and provide most of the docstrings. It was also used to explain complicated CNN terms when encountered in dense readings.