

```
# Uninstall any existing versions of TensorFlow and Keras (to avoid version conflicts)
!pip uninstall -y tensorflow keras tf-keras

# Install TensorFlow 2.17.0, ensuring compatibility with tf-keras 2.17.0; resamy: used to resample audio files
!pip install tensorflow==2.17.0 resamy librosa matplotlib

# Libraries required for converting zipfile
import zipfile
import os

# Import necessary libraries
import pandas as pd
import seaborn as sns
import librosa #audio processing
import librosa.display #audio visualisations
import numpy as np #numerical operations
import matplotlib.pyplot as plt #plotting
import glob #to find files in directories using patterns
from sklearn.model_selection import train_test_split #splits dataset into train and test
from sklearn.preprocessing import LabelEncoder #to encode categorical emotion labels as integers
from tensorflow.keras.utils import to_categorical #converts int labels to one hot encoded format(converts to binary vector represen.) for c1
from tensorflow.keras.models import Sequential #to define a linear stack of layers
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, Dropout, Input #different types of layers for building CNN
from tensorflow.keras.optimizers import Adam #optimizer for training the model
```

```
WARNING: Skipping tensorflow as it is not installed.
WARNING: Skipping keras as it is not installed.
WARNING: Skipping tf-keras as it is not installed.
Collecting tensorflow==2.17.0
  Using cached tensorflow-2.17.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (4.2 kB)
Requirement already satisfied: resamy in /usr/local/lib/python3.10/dist-packages (0.4.3)
Requirement already satisfied: librosa in /usr/local/lib/python3.10/dist-packages (0.10.2.post1)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.8.0)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!<0.5.1,!<0.5.2,>=0.2.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (0.2.0)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (0.2.0)
Requirement already satisfied: h5py>=3.10.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (3.12.1)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (18.1.1)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.3.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (0.4.1)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (3.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (24.2)
Requirement already satisfied: protobuf!=4.21.0,!<4.21.1,!<4.21.2,!<4.21.3,!<4.21.4,!<4.21.5,<5.0.0dev,>=3.20.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (2.32.3)
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (2.32.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (75.1.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (2.5.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (4.12.2)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (1.16.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (1.67.1)
Requirement already satisfied: tensorboard<2.18,>=2.17 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (2.17.1)
Collecting keras>=3.2.0 (from tensorflow==2.17.0)
  Using cached keras-3.6.0-py3-none-any.whl.metadata (5.8 kB)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (0.37.0)
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (1.26.4)
Requirement already satisfied: numba>=0.53 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (0.60.0)
Requirement already satisfied: audioread>=2.1.9 in /usr/local/lib/python3.10/dist-packages (from librosa) (3.0.1)
Requirement already satisfied: scipy>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.13.1)
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.5.2)
Requirement already satisfied: joblib>=0.14 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.4.2)
Requirement already satisfied: decorator>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (4.4.2)
Requirement already satisfied: soundfile>=0.12.1 in /usr/local/lib/python3.10/dist-packages (from librosa) (0.12.1)
Requirement already satisfied: pooch>=1.1 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.8.2)
Requirement already satisfied: soxr>=0.3.2 in /usr/local/lib/python3.10/dist-packages (from librosa) (0.5.0.post1)
Requirement already satisfied: lazy-loader>=0.1 in /usr/local/lib/python3.10/dist-packages (from librosa) (0.4)
Requirement already satisfied: msgpack>=1.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.1.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.54.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.7)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (11.0.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow==2.17.0) (0.43.0)
Requirement already satisfied: rich in /usr/local/lib/python3.10/dist-packages (from keras>=3.2.0->tensorflow==2.17.0) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.10/dist-packages (from keras>=3.2.0->tensorflow==2.17.0) (0.0.8)
Requirement already satisfied: optree in /usr/local/lib/python3.10/dist-packages (from keras>=3.2.0->tensorflow==2.17.0) (0.13.1)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/python3.10/dist-packages (from numba>=0.53->resamy) (0.43.0)
```

Requirement already satisfied: platformdirs>=2.5.0 in /usr/local/lib/python3.10/dist-packages (from pooch>=1.1->librosa) (4.3.6)
 Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorf

```
zip_path = '/content/ravdess file.zip' #defining the zip path
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall() #converting zipfile
```

DATA PREPROCESSING

```
# Feature extraction function
def extract_features(file_name):
    audio, sample_rate = librosa.load(file_name, res_type='kaiser_fast')
    # Using Mel-spectrogram as feature
    mel_spectrogram = librosa.feature.melspectrogram(y=audio, sr=sample_rate, n_mels=128, fmax=8000)
    log_mel_spectrogram = librosa.power_to_db(mel_spectrogram)
    return log_mel_spectrogram

# Path to the extracted audio files
data_path = '/content' # Adjust if the folder name is different

# Define emotion label dictionary
emotion_labels = {
    '01': 'neutral',
    '02': 'calm',
    '03': 'happy',
    '04': 'sad',
    '05': 'angry',
    '06': 'fearful',
    '07': 'disgust',
    '08': 'surprised'
}

# Lists to store features and corresponding emotion labels
features, labels = [], []

# Loop through each audio file in the dataset directory
for file in glob.glob("/content/**/*.wav", recursive=True):
    # Extract the filename and retrieve emotion label based on filename format
    file_name = os.path.basename(file)
    emotion_code = file_name.split("-")[2] # The emotion is the 3rd element in the filename

    # Check if emotion code exists in our emotion_labels dictionary
    if emotion_code in emotion_labels:
        emotion = emotion_labels[emotion_code]
        labels.append(emotion) # Append the correct emotion label
    else:
        print(f"Unrecognized emotion code in file: {file_name}")

# Convert labels to a DataFrame to inspect the emotion counts
labels_df = pd.DataFrame(labels, columns=['Emotions'])
print(labels_df['Emotions'].value_counts())
```

```
Emotions
angry      192
fearful    192
disgust     192
surprised  192
happy       192
calm        192
sad         192
neutral     96
Name: count, dtype: int64
```

```
# Use glob to get the list of all .wav files in the directory
files = glob.glob(f"/content/**/*.wav", recursive=True)

# Check the length of the dataset
dataset_length = len(files)
print(f"Number of audio files in the dataset: {dataset_length}")
```

```
Number of audio files in the dataset: 1440
```

```
files[:5]
```

```
↳ ['/content/Actor_17/03-01-05-02-02-01-17.wav',
    '/content/Actor_17/03-01-06-02-01-01-17.wav',
    '/content/Actor_17/03-01-07-01-01-01-17.wav',
    '/content/Actor_17/03-01-08-01-01-02-17.wav',
    '/content/Actor_17/03-01-08-01-02-02-17.wav']
```

```
# Import necessary libraries
from IPython.display import Audio
import warnings
warnings.filterwarnings('ignore')
```

```
labels[:15]
```

```
↳ ['angry',
    'fearful',
    'disgust',
    'surprised',
    'surprised',
    'neutral',
    'surprised',
    'happy',
    'calm',
    'sad',
    'surprised',
    'disgust',
    'disgust',
    'disgust',
    'sad']
```

```
# Convert labels list into a DataFrame
labels_df = pd.DataFrame(labels, columns=['Emotions'])
```

```
# Plot the count of each emotion
plt.figure(figsize=(10, 6))
ax = sns.countplot(x='Emotions', data=labels_df, order=labels_df['Emotions'].value_counts().index, palette='Set2')
```

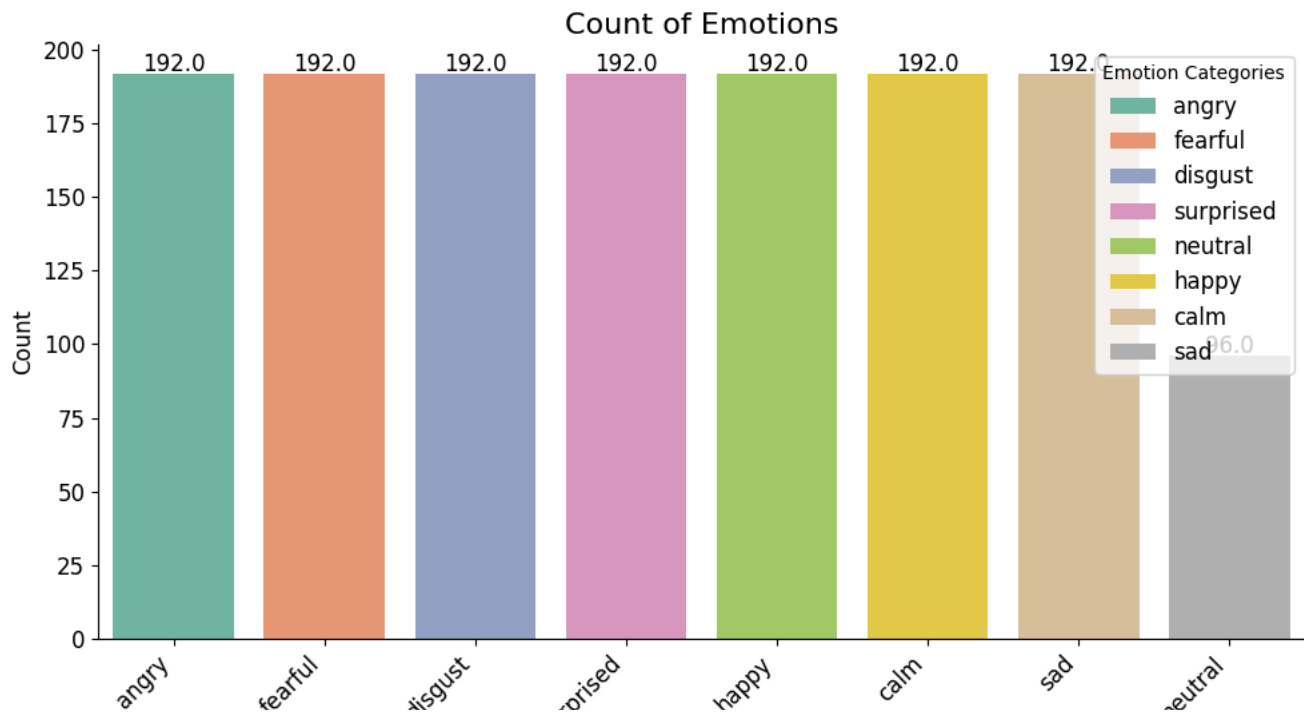
```
# Add title and labels
plt.title('Count of Emotions', size=16)
plt.xlabel('Emotions', size=12)
plt.ylabel('Count', size=12)
```

```
# Add counts on top of each bar
for p in ax.patches:
    ax.annotate(f'{p.get_height()}', (p.get_x() + p.get_width() / 2., p.get_height()),
               ha='center', va='center', size=12, xytext=(0, 5), textcoords='offset points')
```

```
# Adjust plot aesthetics
sns.despine(top=True, right=True, left=False, bottom=False) # Removes or adjusts borders of plot
plt.xticks(rotation=45, ha='right', fontsize=12) # Rotate x labels for better readability
plt.yticks(fontsize=12)
```

```
# Add a legend
plt.legend(title="Emotion Categories", labels=labels_df['Emotions'].unique(), loc='upper right', fontsize=12)
```

```
plt.tight_layout() # Ensure everything fits without overlap
plt.show()
```



```
# List of emotions to process
emotions = ['fearful', 'calm', 'neutral', 'happy', 'sad', 'disgust', 'surprised']

# Define a function to plot the waveform and spectrogram
def create_waveform_and_spectrogram(data, sr, e, ax_wave, ax_spec):
    # Plot waveform
    ax_wave.set_title(f'Waveform - {e}', fontsize=12)
    librosa.display.waveshow(data, sr=sr, ax=ax_wave)

    # Compute and plot spectrogram
    X = librosa.stft(data)
    Xdb = librosa.amplitude_to_db(abs(X))
    ax_spec.set_title(f'Spectrogram - {e}', fontsize=12)
    img = librosa.display.specshow(Xdb, sr=sr, x_axis='time', y_axis='hz', ax=ax_spec)
    plt.colorbar(img, ax=ax_spec, format='%2.0f dB')

# Initialize the figure for a vertical layout with larger figsize
fig, axes = plt.subplots(len(emotions), 2, figsize=(10, 20))
fig.suptitle('Waveforms and Spectrograms for Each Emotion', fontsize=16)

# Iterate over emotions and plot
for idx, emotion in enumerate(emotions):
    emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

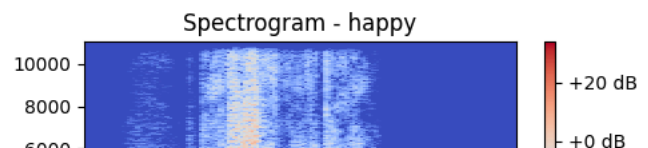
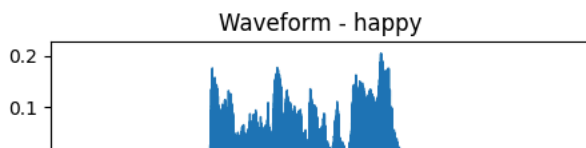
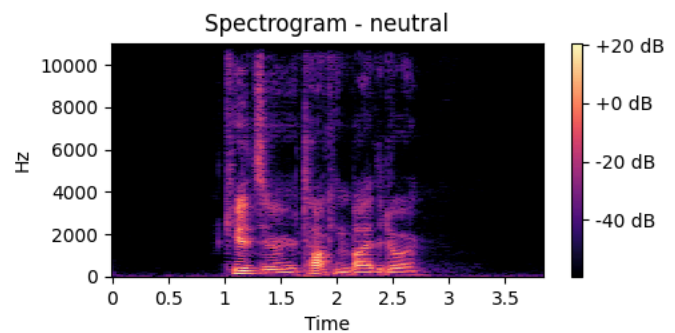
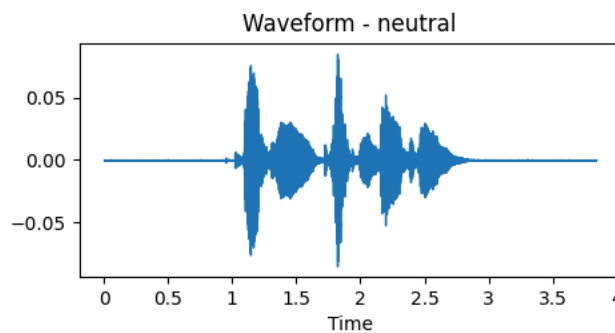
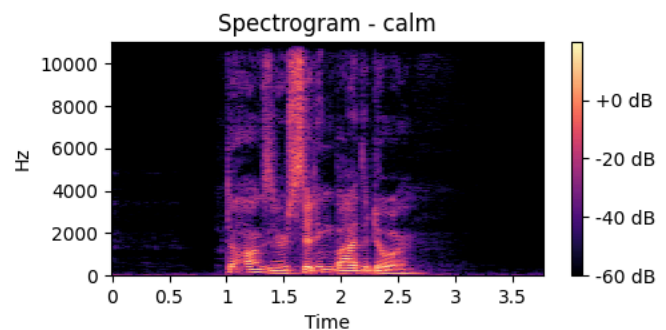
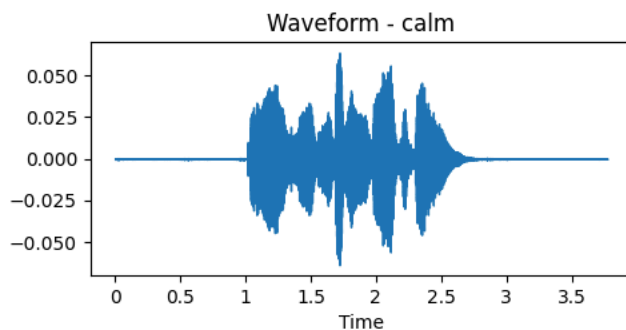
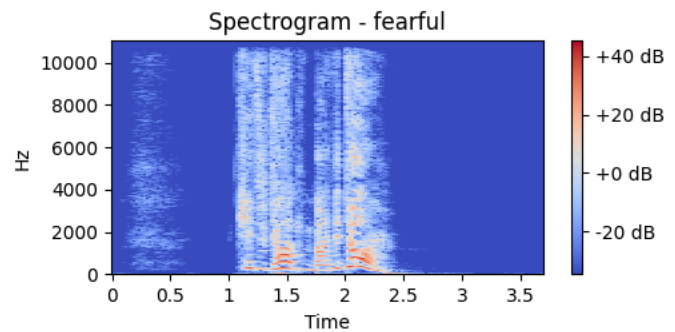
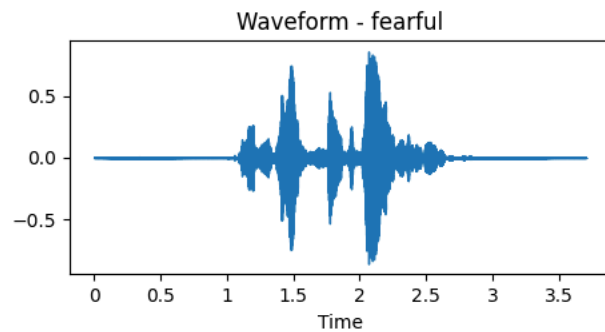
    if emotion_files:
        path = emotion_files[0] # Take the first matching file for each emotion
        data, sampling_rate = librosa.load(path)

        # Plot on respective subplots in vertical layout
        create_waveform_and_spectrogram(data, sampling_rate, emotion, axes[idx, 0], axes[idx, 1])

# Adjust layout for space between subplots
plt.subplots_adjust(hspace=0.6)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



Waveforms and Spectrograms for Each Emotion



DATA AUGMENTATION

```
import librosa

# Example path to the audio file
emotion = 'sad' # Change as needed for different emotions
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    path = emotion_files[1] # Use the second file from the emotion-matched files
    data, sample_rate = librosa.load(path)

    # Check if the audio is stereo (more than one channel)
    if len(data.shape) > 1:
        print("Stereo audio detected. Converting to mono.")
        data = librosa.to_mono(data) # Convert stereo to mono
        print(f"Converted to mono. Shape of data: {data.shape}")
    else:
        print("Mono audio detected.")

    # Optionally, play the audio (if you want to check manually)
    from IPython.display import Audio
```

```
display(Audio(data=data, rate=sample_rate))

else:
    print(f"No audio files found for emotion: {emotion}")
```

↻ Mono audio detected.



```
import librosa
import numpy as np
import glob
from IPython.display import Audio, display

# Augmentation Functions

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.time_stretch(y=data, rate=default_rate)

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal
    """
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

# Path to the audio files
emotion = 'sad'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    path = emotion_files[1] # Use the second file from the emotion-matched files
    data, sample_rate = librosa.load(path)

    # Check if the audio is stereo (more than one channel)
    if len(data.shape) > 1:
        print("Stereo audio detected. Converting to mono.")
        data = librosa.to_mono(data) # Convert stereo to mono
        print(f"Converted to mono. Shape of data: {data.shape}")
    else:
        print("Mono audio detected.")

    # Apply augmentation techniques

    # Add noise
    noisy_data = noise(data)
    print("Noise added to audio.")

    # Time-stretching (adjust tempo)
    stretched_data = stretch(noisy_data) # Using default rate in the function
```

```

print("Time-stretching applied.")

# Shifting the audio
shifted_data = shift(stretched_data)
print("Audio shifted.")

# Pitch shifting
pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
print("Pitch shifting applied.")

# Optionally, visualize or listen to the augmented audio
display(Audio(data=pitched_data, rate=sample_rate))

else:
    print(f"No audio files found for emotion: {emotion}")

```

↩ Mono audio detected.
 Noise added to audio.
 Time-stretching applied.
 Audio shifted.
 Pitch shifting applied.

```

import librosa
import numpy as np
import glob
from IPython.display import Audio, display

```

Augmentation Functions FOR SAD

```

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    # `time_stretch` function requires two arguments: `y` (audio data) and `rate` (stretch factor)
    return librosa.effects.time_stretch(y=data, rate=default_rate)

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal
    """
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

# Path to the audio files
emotion = 'sad'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    for file_path in emotion_files: # Loop through all files of the emotion
        print(f"Processing file: {file_path}")

        data, sample_rate = librosa.load(file_path)

```

```

# Check if the audio is stereo (more than one channel)
if len(data.shape) > 1:
    print("Stereo audio detected. Converting to mono.")
    data = librosa.to_mono(data) # Convert stereo to mono
    print(f"Converted to mono. Shape of data: {data.shape}")
else:
    print("Mono audio detected.")

# Apply augmentation techniques

# Add noise
noisy_data = noise(data)
print("Noise added to audio.")

# Time-stretching (adjust tempo)
stretched_data = stretch(noisy_data) # Using default rate in the function
print("Time-stretching applied.")

# Shifting the audio
shifted_data = shift(stretched_data)
print("Audio shifted.")

# Pitch shifting
pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
print("Pitch shifting applied.")

# Optionally, visualize or listen to the augmented audio
display(Audio(data=pitched_data, rate=sample_rate))

else:
    print(f"No audio files found for emotion: {emotion}")

```

 Show hidden output

Augmentation Functions FOR NEUTRAL

```

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    # `time_stretch` function requires two arguments: `y` (audio data) and `rate` (stretch factor)
    return librosa.effects.time_stretch(y=data, rate=default_rate)

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal
    """
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

# Path to the audio files
emotion = 'neutral'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

```



```

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    for file_path in emotion_files: # Loop through all files of the emotion
        print(f"Processing file: {file_path}")

        data, sample_rate = librosa.load(file_path)

        # Check if the audio is stereo (more than one channel)
        if len(data.shape) > 1:
            print("Stereo audio detected. Converting to mono.")
            data = librosa.to_mono(data) # Convert stereo to mono
            print(f"Converted to mono. Shape of data: {data.shape}")
        else:
            print("Mono audio detected.")

        # Apply augmentation techniques

        # Add noise
        noisy_data = noise(data)
        print("Noise added to audio.")

        # Time-stretching (adjust tempo)
        stretched_data = stretch(noisy_data) # Using default rate in the function
        print("Time-stretching applied.")

        # Shifting the audio
        shifted_data = shift(stretched_data)
        print("Audio shifted.")

        # Pitch shifting
        pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
        print("Pitch shifting applied.")

        # Optionally, visualize or listen to the augmented audio
        display(Audio(data=pitched_data, rate=sample_rate))

else:
    print(f"No audio files found for emotion: {emotion}")

```

 [Show hidden output](#)

Augmentation Functions FOR CALM

```

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    # `time_stretch` function requires two arguments: `y` (audio data) and `rate` (stretch factor)
    return librosa.effects.time_stretch(y=data, rate=default_rate)

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal
    """
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

```

```

# Path to the audio files
emotion = 'calm'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    for file_path in emotion_files: # Loop through all files of the emotion
        print(f"Processing file: {file_path}")

        data, sample_rate = librosa.load(file_path)

        # Check if the audio is stereo (more than one channel)
        if len(data.shape) > 1:
            print("Stereo audio detected. Converting to mono.")
            data = librosa.to_mono(data) # Convert stereo to mono
            print(f"Converted to mono. Shape of data: {data.shape}")
        else:
            print("Mono audio detected.")

        # Apply augmentation techniques

        # Add noise
        noisy_data = noise(data)
        print("Noise added to audio.")

        # Time-stretching (adjust tempo)
        stretched_data = stretch(noisy_data) # Using default rate in the function
        print("Time-stretching applied.")

        # Shifting the audio
        shifted_data = shift(stretched_data)
        print("Audio shifted.")

        # Pitch shifting
        pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
        print("Pitch shifting applied.")

        # Optionally, visualize or listen to the augmented audio
        display(Audio(data=pitched_data, rate=sample_rate))
    else:
        print(f"No audio files found for emotion: {emotion}")

```

 [Show hidden output](#)

Augmentation Functions FOR HAPPY

```

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    # `time_stretch` function requires two arguments: `y` (audio data) and `rate` (stretch factor)
    return librosa.effects.time_stretch(y=data, rate=default_rate)

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal

```

```

"""
if len(data.shape) > 1: # If stereo, convert to mono
    data = librosa.to_mono(data)
return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

# Path to the audio files
emotion = 'happy'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    for file_path in emotion_files: # Loop through all files of the emotion
        print(f"Processing file: {file_path}")

        data, sample_rate = librosa.load(file_path)

        # Check if the audio is stereo (more than one channel)
        if len(data.shape) > 1:
            print("Stereo audio detected. Converting to mono.")
            data = librosa.to_mono(data) # Convert stereo to mono
            print(f"Converted to mono. Shape of data: {data.shape}")
        else:
            print("Mono audio detected.")

        # Apply augmentation techniques

        # Add noise
        noisy_data = noise(data)
        print("Noise added to audio.")

        # Time-stretching (adjust tempo)
        stretched_data = stretch(noisy_data) # Using default rate in the function
        print("Time-stretching applied.")

        # Shifting the audio
        shifted_data = shift(stretched_data)
        print("Audio shifted.")

        # Pitch shifting
        pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
        print("Pitch shifting applied.")

        # Optionally, visualize or listen to the augmented audio
        display(Audio(data=pitched_data, rate=sample_rate))

else:
    print(f"No audio files found for emotion: {emotion}")

```

 [Show hidden output](#)

Augmentation Functions FOR ANGRY

```

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    # `time_stretch` function requires two arguments: `y` (audio data) and `rate` (stretch factor)
    return librosa.effects.time_stretch(y=data, rate=default_rate)

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)

```

```

    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal
    """
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

# Path to the audio files
emotion = 'angry'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    for file_path in emotion_files: # Loop through all files of the emotion
        print(f"Processing file: {file_path}")

        data, sample_rate = librosa.load(file_path)

        # Check if the audio is stereo (more than one channel)
        if len(data.shape) > 1:
            print("Stereo audio detected. Converting to mono.")
            data = librosa.to_mono(data) # Convert stereo to mono
            print(f"Converted to mono. Shape of data: {data.shape}")
        else:
            print("Mono audio detected.")

        # Apply augmentation techniques

        # Add noise
        noisy_data = noise(data)
        print("Noise added to audio.")

        # Time-stretching (adjust tempo)
        stretched_data = stretch(noisy_data) # Using default rate in the function
        print("Time-stretching applied.")

        # Shifting the audio
        shifted_data = shift(stretched_data)
        print("Audio shifted.")

        # Pitch shifting
        pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
        print("Pitch shifting applied.")

        # Optionally, visualize or listen to the augmented audio
        display(Audio(data=pitched_data, rate=sample_rate))
    else:
        print(f"No audio files found for emotion: {emotion}")

```

 [Show hidden output](#)

Augmentation Functions FOR FEARFUL

```

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    # `time_stretch` function requires two arguments: `y` (audio data) and `rate` (stretch factor)
    return librosa.effects.time_stretch(y=data, rate=default_rate)

```

```

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal
    """
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

# Path to the audio files
emotion = 'fearful'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    for file_path in emotion_files: # Loop through all files of the emotion
        print(f"Processing file: {file_path}")

        data, sample_rate = librosa.load(file_path)

        # Check if the audio is stereo (more than one channel)
        if len(data.shape) > 1:
            print("Stereo audio detected. Converting to mono.")
            data = librosa.to_mono(data) # Convert stereo to mono
            print(f"Converted to mono. Shape of data: {data.shape}")
        else:
            print("Mono audio detected.")

        # Apply augmentation techniques

        # Add noise
        noisy_data = noise(data)
        print("Noise added to audio.")

        # Time-stretching (adjust tempo)
        stretched_data = stretch(noisy_data) # Using default rate in the function
        print("Time-stretching applied.")

        # Shifting the audio
        shifted_data = shift(stretched_data)
        print("Audio shifted.")

        # Pitch shifting
        pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
        print("Pitch shifting applied.")

        # Optionally, visualize or listen to the augmented audio
        display(Audio(data=pitched_data, rate=sample_rate))

else:
    print(f"No audio files found for emotion: {emotion}")

```

 [Show hidden output](#)

Augmentation Functions FOR DISGUST

```

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate

```

```

if len(data.shape) > 1: # If stereo, convert to mono
    data = librosa.to_mono(data)
# `time_stretch` function requires two arguments: `y` (audio data) and `rate` (stretch factor)
return librosa.effects.time_stretch(y=data, rate=default_rate)

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal
    """
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

# Path to the audio files
emotion = 'disgust'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    for file_path in emotion_files: # Loop through all files of the emotion
        print(f"Processing file: {file_path}")

        data, sample_rate = librosa.load(file_path)

        # Check if the audio is stereo (more than one channel)
        if len(data.shape) > 1:
            print("Stereo audio detected. Converting to mono.")
            data = librosa.to_mono(data) # Convert stereo to mono
            print(f"Converted to mono. Shape of data: {data.shape}")
        else:
            print("Mono audio detected.")

        # Apply augmentation techniques

        # Add noise
        noisy_data = noise(data)
        print("Noise added to audio.")

        # Time-stretching (adjust tempo)
        stretched_data = stretch(noisy_data) # Using default rate in the function
        print("Time-stretching applied.")

        # Shifting the audio
        shifted_data = shift(stretched_data)
        print("Audio shifted.")

        # Pitch shifting
        pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
        print("Pitch shifting applied.")

        # Optionally, visualize or listen to the augmented audio
        display(Audio(data=pitched_data, rate=sample_rate))

else:
    print(f"No audio files found for emotion: {emotion}")

```

 [Show hidden output](#)

Augmentation Functions FOR SUPRISED

```

def noise(data):
    """
    Add random noise to the audio signal
    """
    noise_amp = 0.035 * np.random.uniform() * np.amax(data)
    data = data + noise_amp * np.random.normal(size=data.shape[0])
    return data

```

```

def stretch(data):
    """
    Time-stretching of the audio signal using a default rate.
    """
    default_rate = 0.8 # Default time-stretching rate
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    # `time_stretch` function requires two arguments: `y` (audio data) and `rate` (stretch factor)
    return librosa.effects.time_stretch(y=data, rate=default_rate)

def shift(data):
    """
    Shift the audio signal by a random amount
    """
    shift_range = int(np.random.uniform(low=-5, high=5) * 1000)
    return np.roll(data, shift_range)

def pitch(data, sampling_rate, pitch_factor=0.7):
    """
    Shift the pitch of the audio signal
    """
    if len(data.shape) > 1: # If stereo, convert to mono
        data = librosa.to_mono(data)
    return librosa.effects.pitch_shift(y=data, sr=sampling_rate, n_steps=pitch_factor)

# Path to the audio files
emotion = 'surprised'
emotion_files = [file for file, label in zip(glob.glob("/content/**/*.wav", recursive=True), labels) if label == emotion]

# Ensure at least one file with the specified emotion exists
if len(emotion_files) > 0:
    for file_path in emotion_files: # Loop through all files of the emotion
        print(f"Processing file: {file_path}")

        data, sample_rate = librosa.load(file_path)

        # Check if the audio is stereo (more than one channel)
        if len(data.shape) > 1:
            print("Stereo audio detected. Converting to mono.")
            data = librosa.to_mono(data) # Convert stereo to mono
            print(f"Converted to mono. Shape of data: {data.shape}")
        else:
            print("Mono audio detected.")

        # Apply augmentation techniques

        # Add noise
        noisy_data = noise(data)
        print("Noise added to audio.")

        # Time-stretching (adjust tempo)
        stretched_data = stretch(noisy_data) # Using default rate in the function
        print("Time-stretching applied.")

        # Shifting the audio
        shifted_data = shift(stretched_data)
        print("Audio shifted.")

        # Pitch shifting
        pitched_data = pitch(shifted_data, sample_rate, pitch_factor=0.7)
        print("Pitch shifting applied.")

        # Optionally, visualize or listen to the augmented audio
        display(Audio(data=pitched_data, rate=sample_rate))
    else:
        print(f"No audio files found for emotion: {emotion}")

```

 [Show hidden output](#)

FEATURE EXTRACTION

```

# Feature Extraction Function
def extract_features(data, sample_rate):

```

```

result = np.array([])

# Zero Crossing Rate (ZCR)
zcr = np.mean(librosa.feature.zero_crossing_rate(y=data).T, axis=0)
result = np.hstack((result, zcr)) # stacking horizontally

# Chroma STFT
stft = np.abs(librosa.stft(data))
chroma_stft = np.mean(librosa.feature.chroma_stft(S=stft, sr=sample_rate).T, axis=0)
result = np.hstack((result, chroma_stft)) # stacking horizontally

# Mel-Frequency Cepstral Coefficients (MFCC)
mfcc = np.mean(librosa.feature.mfcc(y=data, sr=sample_rate).T, axis=0)
result = np.hstack((result, mfcc)) # stacking horizontally

# Root Mean Square (RMS) Energy
rms = np.mean(librosa.feature.rms(y=data).T, axis=0)
result = np.hstack((result, rms)) # stacking horizontally

# Mel Spectrogram
mel = np.mean(librosa.feature.melspectrogram(y=data, sr=sample_rate).T, axis=0)
result = np.hstack((result, mel)) # stacking horizontally

return result

# Function to apply augmentations and extract features
def get_features(path):
    # Load audio data, setting duration and offset to avoid silent sections
    data, sample_rate = librosa.load(path, duration=2.5, offset=0.6)

    # Feature extraction without augmentation
    res1 = extract_features(data, sample_rate)
    result = np.array(res1)

    # Feature extraction with noise augmentation
    noisy_data = noise(data)
    res2 = extract_features(noisy_data, sample_rate)
    result = np.vstack((result, res2)) # stacking vertically

    # Feature extraction with stretching and pitching
    stretched_data = stretch(data)
    stretched_pitched_data = pitch(stretched_data, sample_rate)
    res3 = extract_features(stretched_pitched_data, sample_rate)
    result = np.vstack((result, res3)) # stacking vertically

    return result

# Create a DataFrame with file paths and corresponding labels
data_path = pd.DataFrame({
    'files': files,
    'Emotions': labels
})

# Initialize empty lists for features and labels
X, Y = [], []

# Loop through each file path and emotion label in the dataset
for file_path, emotion in zip(data_path.files, data_path.Emotions):
    # Extract features using the get_features function
    features = get_features(file_path)

    # Append each feature vector from the augmented dataset to X, and corresponding emotion label to Y
    for feature_vector in features:
        X.append(feature_vector)
        Y.append(emotion) # Each augmentation (3x) will correspond to the same emotion label

# Convert X and Y to numpy arrays for compatibility with machine learning frameworks
X = np.array(X)
Y = np.array(Y)

# Check lengths of X, Y and the shape of the Path column in the data_path DataFrame
print(len(X), len(Y), data_path['files'].shape)

```



```
↳ 4320 4320 (1440,)
```

```
# Check if files and labels have the same length
print(f"Number of files: {len(files)}")
print(f"Number of labels: {len(labels)}")
```

```
↳ Number of files: 1440
Number of labels: 1440
```

```
# Convert features (X) and labels (Y) to a DataFrame
Features = pd.DataFrame(X)
```

```
# Add labels as a new column in the DataFrame
Features['labels'] = Y
```

```
# Save the DataFrame to a CSV file
Features.to_csv('features.csv', index=False)
```

```
# Display the first few rows of the DataFrame
print(Features.head())
```

```
↳
```

	0	1	2	3	4	5	6	\
0	0.165071	0.670164	0.649708	0.698891	0.693580	0.715213	0.701987	
1	0.339446	0.762483	0.760767	0.831211	0.804191	0.807244	0.720037	
2	0.150796	0.705705	0.633148	0.629354	0.679170	0.636790	0.671458	
3	0.074617	0.587358	0.651982	0.696176	0.741251	0.716236	0.643976	
4	0.246116	0.691675	0.741865	0.782159	0.818721	0.808975	0.725570	

	7	8	9	...	153	154	155	156	\
0	0.651738	0.752112	0.763794	...	0.003117	0.002914	0.003016	0.002015	
1	0.684648	0.754255	0.765908	...	0.018077	0.018517	0.017949	0.017331	
2	0.653568	0.643654	0.738486	...	0.000488	0.000626	0.000497	0.000776	
3	0.607775	0.666376	0.688183	...	0.004640	0.006766	0.009339	0.006497	
4	0.643200	0.680619	0.699865	...	0.016344	0.018865	0.021209	0.017973	

	157	158	159	160	161	labels
0	0.001758	0.002185	0.002028	0.001309	0.000103	angry
1	0.017101	0.017503	0.017911	0.017948	0.015984	angry
2	0.000639	0.000355	0.000380	0.000252	0.000022	angry
3	0.005714	0.004967	0.004475	0.002107	0.000166	fearful
4	0.017634	0.016599	0.016135	0.013545	0.011951	fearful

```
[5 rows x 163 columns]
```

DATA PREPARATION FOR CNN MODEL

```
# Separate the features (all columns except 'labels')
X = Features.iloc[:, :-1].values # Select all rows, all columns except the last one ('labels')
```

```
# Separate the labels (the 'labels' column)
Y = Features['labels'].values # Extract the 'labels' column as the target variable
```

```
label_encoder = LabelEncoder()
Y_encoded = to_categorical(label_encoder.fit_transform(labels))
```

```
num_augmentations = 3 # Adjust based on the actual number of augmentations you used
```

```
# Repeat each label in Y_encoded to match the number of augmentations
Y_expanded = np.repeat(Y_encoded, num_augmentations, axis=0)
```

```
print("Shape of X:", X.shape) # X should already include augmented features
print("Shape of Y_expanded:", Y_expanded.shape)
```

```
↳ Shape of X: (4320, 162)
Shape of Y_expanded: (4320, 8)
```

```
print("Labels:", labels[:10]) # Display a sample of original labels
print("Y_encoded shape:", Y_encoded.shape if 'Y_encoded' in locals() else "Not defined")
```

```
↳ Labels: ['angry', 'fearful', 'disgust', 'surprised', 'surprised', 'neutral', 'surprised', 'happy', 'calm', 'sad']
Y_encoded shape: (1440, 8)
```

```
# Now split the data with consistent lengths for X and Y
x_train, x_test, y_train, y_test = train_test_split(X, Y_expanded, random_state=0, shuffle=True)

print("Shapes - x_train:", x_train.shape, "y_train:", y_train.shape, "x_test:", x_test.shape, "y_test:", y_test.shape)

➡ Shapes - x_train: (3240, 162) y_train: (3240, 8) x_test: (1080, 162) y_test: (1080, 8)

from sklearn.preprocessing import OneHotEncoder # Import the necessary class

# Ensure Y is a 1D array of labels
Y = np.array(Y).reshape(-1, 1) # Reshaping to a column vector (necessary for fit_transform)

# Perform one-hot encoding on the labels
encoder = OneHotEncoder() # Instantiate the encoder
Y_encoded = encoder.fit_transform(Y).toarray() # Transform and convert to a dense array

from sklearn.preprocessing import OneHotEncoder

# Convert labels list to a numpy array and reshape for encoding
Y = np.array(labels).reshape(-1, 1)

# One-hot encode the labels
encoder = OneHotEncoder(sparse_output=False)
Y_encoded = encoder.fit_transform(Y) # Y_encoded will have a shape matching the number of emotions

print("One-hot encoded labels shape:", Y_encoded.shape)

➡ One-hot encoded labels shape: (1440, 8)

from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Fit and transform the training data
x_train = scaler.fit_transform(x_train)

# Transform the test data based on the training data scaling
x_test = scaler.transform(x_test)

# Print the shapes of the scaled data
x_train.shape, y_train.shape, x_test.shape, y_test.shape

➡ ((3240, 162), (3240, 8), (1080, 162), (1080, 8))

# Reshaping the data to match the input requirements for a Conv1D model
x_train = np.expand_dims(x_train, axis=2) # Adding an additional dimension (channels)
x_test = np.expand_dims(x_test, axis=2) # Adding an additional dimension (channels)

# Check the new shapes of the data
x_train.shape, y_train.shape, x_test.shape, y_test.shape

➡ ((3240, 162, 1), (3240, 8), (1080, 162, 1), (1080, 8))
```

CNN MODEL FOR SER

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Dropout, Flatten, Dense

# Define the model architecture
model = Sequential()

# Add 1D Convolutional layers with MaxPooling
model.add(Conv1D(64, kernel_size=3, strides=1, padding='same', activation='relu', input_shape=(x_train.shape[1], 1)))
model.add(MaxPooling1D(pool_size=2, strides=2, padding='same'))
model.add(Dropout(0.3)) # Adding dropout for regularization

model.add(Conv1D(128, kernel_size=3, strides=1, padding='same', activation='relu'))
```

```

model.add(MaxPooling1D(pool_size=2, strides=2, padding='same'))
model.add(Dropout(0.3))

model.add(Conv1D(256, kernel_size=3, strides=1, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2, strides=2, padding='same'))
model.add(Dropout(0.4)) # Adding higher dropout for deeper layers

model.add(Conv1D(256, kernel_size=3, strides=1, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2, strides=2, padding='same'))
model.add(Dropout(0.5)) # Final dropout for strong regularization

# Flatten the output from convolutional layers
model.add(Flatten())

# Add a Dense layer with 256 units
model.add(Dense(units=256, activation='relu'))
model.add(Dropout(0.5)) # Dropout layer for regularization

# Add the final Dense layer for multiclass classification
num_classes = len(np.unique(Y)) # Get the number of unique emotions (number of classes)
model.add(Dense(units=num_classes, activation='softmax')) # Use softmax for multiclass classification

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Summarize the model architecture
model.summary()

```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 162, 64)	256
max_pooling1d (MaxPooling1D)	(None, 81, 64)	0
dropout (Dropout)	(None, 81, 64)	0
conv1d_1 (Conv1D)	(None, 81, 128)	24,704
max_pooling1d_1 (MaxPooling1D)	(None, 41, 128)	0
dropout_1 (Dropout)	(None, 41, 128)	0
conv1d_2 (Conv1D)	(None, 41, 256)	98,560
max_pooling1d_2 (MaxPooling1D)	(None, 21, 256)	0
dropout_2 (Dropout)	(None, 21, 256)	0
conv1d_3 (Conv1D)	(None, 21, 256)	196,864
max_pooling1d_3 (MaxPooling1D)	(None, 11, 256)	0
dropout_3 (Dropout)	(None, 11, 256)	0
flatten (Flatten)	(None, 2816)	0
dense (Dense)	(None, 256)	721,152
dropout_4 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 8)	2,056

Total params: 1,043,592 (3.98 MB)

Trainable params: 1,043,592 (3.98 MB)

```

from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping # Import necessary callbacks

```

```

# Callbacks
rlrp = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, min_lr=1e-5)
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

```

```

# Train the model
history = model.fit(
    x_train, y_train,
    batch_size=64,
    epochs=50,
    validation_data=(x_test, y_test),
    callbacks=[rlrp, early_stopping]
)

```

```

Epoch 1/50
51/51 ————— 11s 178ms/step - accuracy: 0.1726 - loss: 2.0303 - val_accuracy: 0.2648 - val_loss: 1.9312 - learning_rate
Epoch 2/50
51/51 ————— 8s 131ms/step - accuracy: 0.2407 - loss: 1.8821 - val_accuracy: 0.3111 - val_loss: 1.8543 - learning_rate
Epoch 3/50
51/51 ————— 10s 126ms/step - accuracy: 0.3153 - loss: 1.7824 - val_accuracy: 0.3602 - val_loss: 1.7038 - learning_rate
Epoch 4/50
51/51 ————— 11s 130ms/step - accuracy: 0.3398 - loss: 1.7261 - val_accuracy: 0.3704 - val_loss: 1.6594 - learning_rate
Epoch 5/50
51/51 ————— 8s 164ms/step - accuracy: 0.3625 - loss: 1.6787 - val_accuracy: 0.3907 - val_loss: 1.6250 - learning_rate
Epoch 6/50
51/51 ————— 8s 126ms/step - accuracy: 0.3615 - loss: 1.6285 - val_accuracy: 0.3685 - val_loss: 1.6373 - learning_rate
Epoch 7/50
51/51 ————— 10s 126ms/step - accuracy: 0.3899 - loss: 1.6179 - val_accuracy: 0.4111 - val_loss: 1.5672 - learning_rate
Epoch 8/50
51/51 ————— 12s 160ms/step - accuracy: 0.4180 - loss: 1.5365 - val_accuracy: 0.4287 - val_loss: 1.5387 - learning_rate
Epoch 9/50
51/51 ————— 11s 172ms/step - accuracy: 0.3994 - loss: 1.5629 - val_accuracy: 0.4343 - val_loss: 1.5086 - learning_rate
Epoch 10/50
51/51 ————— 6s 126ms/step - accuracy: 0.4163 - loss: 1.4955 - val_accuracy: 0.4417 - val_loss: 1.4900 - learning_rate
Epoch 11/50
51/51 ————— 12s 154ms/step - accuracy: 0.4250 - loss: 1.4966 - val_accuracy: 0.4472 - val_loss: 1.4343 - learning_rate
Epoch 12/50
51/51 ————— 11s 177ms/step - accuracy: 0.4400 - loss: 1.4670 - val_accuracy: 0.4500 - val_loss: 1.4542 - learning_rate
Epoch 13/50
51/51 ————— 7s 126ms/step - accuracy: 0.4478 - loss: 1.4548 - val_accuracy: 0.4704 - val_loss: 1.3920 - learning_rate
Epoch 14/50
51/51 ————— 8s 163ms/step - accuracy: 0.4657 - loss: 1.3925 - val_accuracy: 0.4870 - val_loss: 1.3631 - learning_rate
Epoch 15/50
51/51 ————— 9s 142ms/step - accuracy: 0.4772 - loss: 1.3643 - val_accuracy: 0.4935 - val_loss: 1.3674 - learning_rate
Epoch 16/50
51/51 ————— 9s 126ms/step - accuracy: 0.4853 - loss: 1.3641 - val_accuracy: 0.5074 - val_loss: 1.3280 - learning_rate
Epoch 17/50
51/51 ————— 8s 165ms/step - accuracy: 0.5009 - loss: 1.3537 - val_accuracy: 0.4917 - val_loss: 1.3364 - learning_rate
Epoch 18/50
51/51 ————— 8s 130ms/step - accuracy: 0.4992 - loss: 1.3141 - val_accuracy: 0.4926 - val_loss: 1.3235 - learning_rate
Epoch 19/50
51/51 ————— 10s 126ms/step - accuracy: 0.5116 - loss: 1.3034 - val_accuracy: 0.5222 - val_loss: 1.2902 - learning_rate
Epoch 20/50
51/51 ————— 8s 165ms/step - accuracy: 0.5032 - loss: 1.2922 - val_accuracy: 0.5111 - val_loss: 1.2832 - learning_rate
Epoch 21/50
51/51 ————— 9s 131ms/step - accuracy: 0.5271 - loss: 1.2658 - val_accuracy: 0.5417 - val_loss: 1.2335 - learning_rate
Epoch 22/50
51/51 ————— 10s 127ms/step - accuracy: 0.5408 - loss: 1.2288 - val_accuracy: 0.5231 - val_loss: 1.2633 - learning_rate
Epoch 23/50
51/51 ————— 11s 131ms/step - accuracy: 0.5268 - loss: 1.2362 - val_accuracy: 0.5509 - val_loss: 1.1968 - learning_rate
Epoch 24/50
51/51 ————— 9s 171ms/step - accuracy: 0.5558 - loss: 1.2128 - val_accuracy: 0.5509 - val_loss: 1.2035 - learning_rate
Epoch 25/50
51/51 ————— 8s 126ms/step - accuracy: 0.5769 - loss: 1.1397 - val_accuracy: 0.5370 - val_loss: 1.2132 - learning_rate
Epoch 26/50
51/51 ————— 10s 126ms/step - accuracy: 0.5725 - loss: 1.1423 - val_accuracy: 0.5713 - val_loss: 1.1619 - learning_rate
Epoch 27/50
51/51 ————— 12s 161ms/step - accuracy: 0.5832 - loss: 1.1238 - val_accuracy: 0.5667 - val_loss: 1.1651 - learning_rate
Epoch 28/50
51/51 ————— 14s 244ms/step - accuracy: 0.5905 - loss: 1.1126 - val_accuracy: 0.5796 - val_loss: 1.1388 - learning_rate
Epoch 29/50

```

Double-click (or enter) to edit

```

# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=1)

# Print the accuracy
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

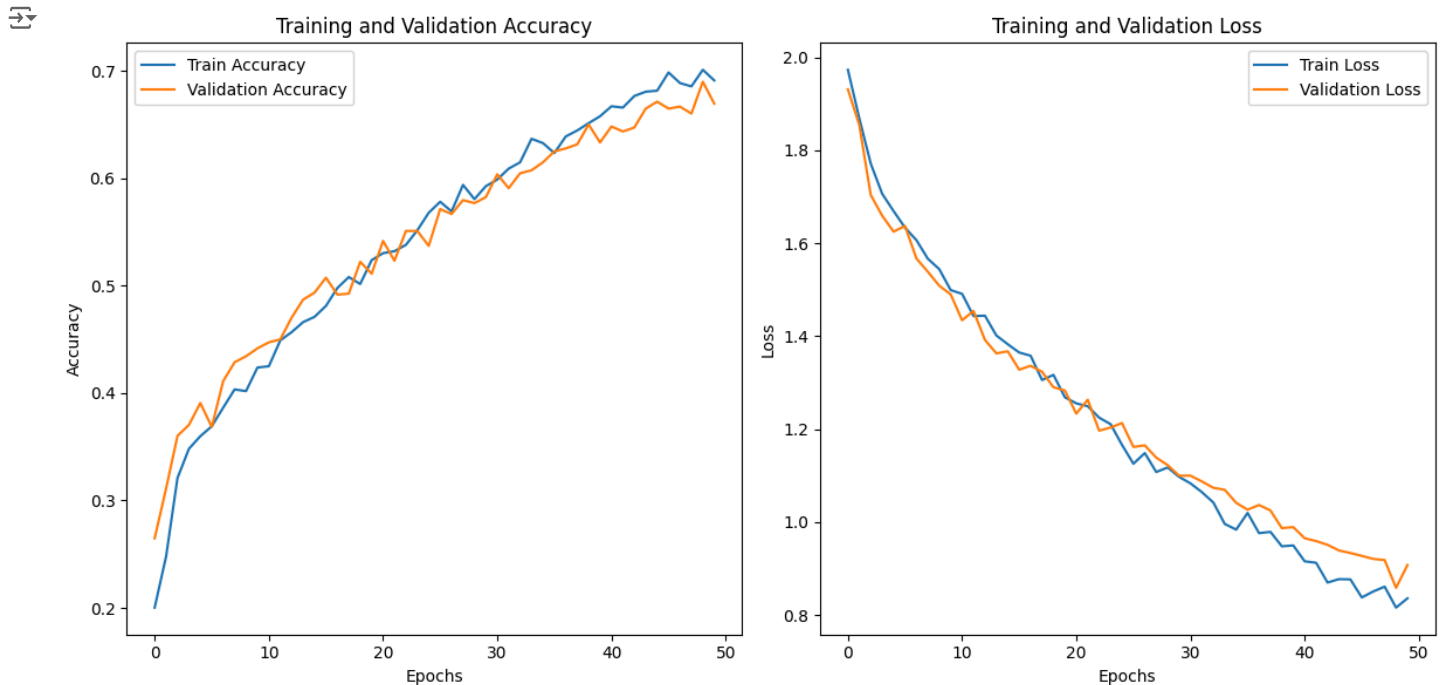
34/34 ————— 1s 15ms/step - accuracy: 0.6839 - loss: 0.8550
Test Accuracy: 68.98%

# Plot the training and validation accuracy
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

```

```
# Plot the training and validation loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Show the plots
plt.tight_layout()
plt.show()
```



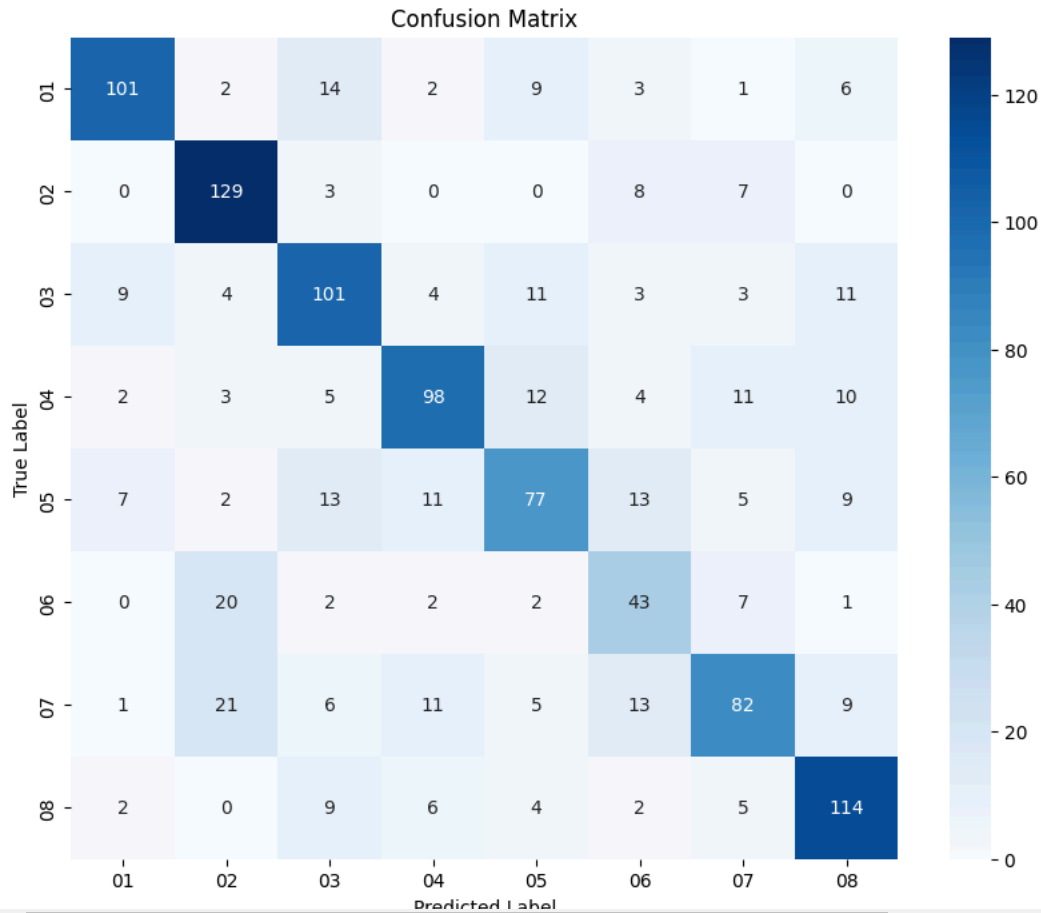
```
from sklearn.metrics import confusion_matrix

# Get predictions on the test set
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1) # Get class predictions from softmax output

# Confusion Matrix
cm = confusion_matrix(np.argmax(y_test, axis=1), y_pred_classes)

# Plot the Confusion Matrix using seaborn
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=emotion_labels, yticklabels=emotion_labels)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

34/34 1s 15ms/step



```

from sklearn.metrics import classification_report

# Get classification report for precision, recall, and f1-score
y_true = np.argmax(y_test, axis=1) # True labels
y_pred = np.argmax(model.predict(x_test), axis=1) # Predicted labels

# Print classification report
print(classification_report(y_true, y_pred, target_names=emotion_labels))

```

34/34 1s 27ms/step

	precision	recall	f1-score	support
01	0.83	0.73	0.78	138
02	0.71	0.88	0.79	147
03	0.66	0.69	0.68	146
04	0.73	0.68	0.70	145
05	0.64	0.56	0.60	137
06	0.48	0.56	0.52	77
07	0.68	0.55	0.61	148
08	0.71	0.80	0.75	142
accuracy			0.69	1080
macro avg	0.68	0.68	0.68	1080
weighted avg	0.69	0.69	0.69	1080

```

# Predict the probabilities for each class
y_pred_bin = model.predict(x_test)

# Verify the shape of y_pred_bin
print(y_pred_bin.shape) # It should be (num_samples, num_classes)

```

34/34 1s 27ms/step
(1080, 8)

```

from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt

emotion_labels = ['happy', 'sad', 'angry', 'fearful', 'surprise', 'neutral', 'disgust']

# Number of classes
num_classes = len(emotion_labels)

# Binarize the true labels for multi-class ROC
y_test_bin = label_binarize(np.argmax(y_test, axis=1), classes=np.arange(num_classes))

# Predict the probabilities for each class
y_pred_bin = model.predict(x_test)

# Verify the shape of y_pred_bin
print(f"y_pred_bin shape: {y_pred_bin.shape}")

# Plot ROC curves for each class
plt.figure(figsize=(10, 8))
for i in range(num_classes):
    fpr, tpr, _ = roc_curve(y_test_bin[:, i], y_pred_bin[:, i])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'Class {emotion_labels[i]} (AUC = {roc_auc:.2f})')

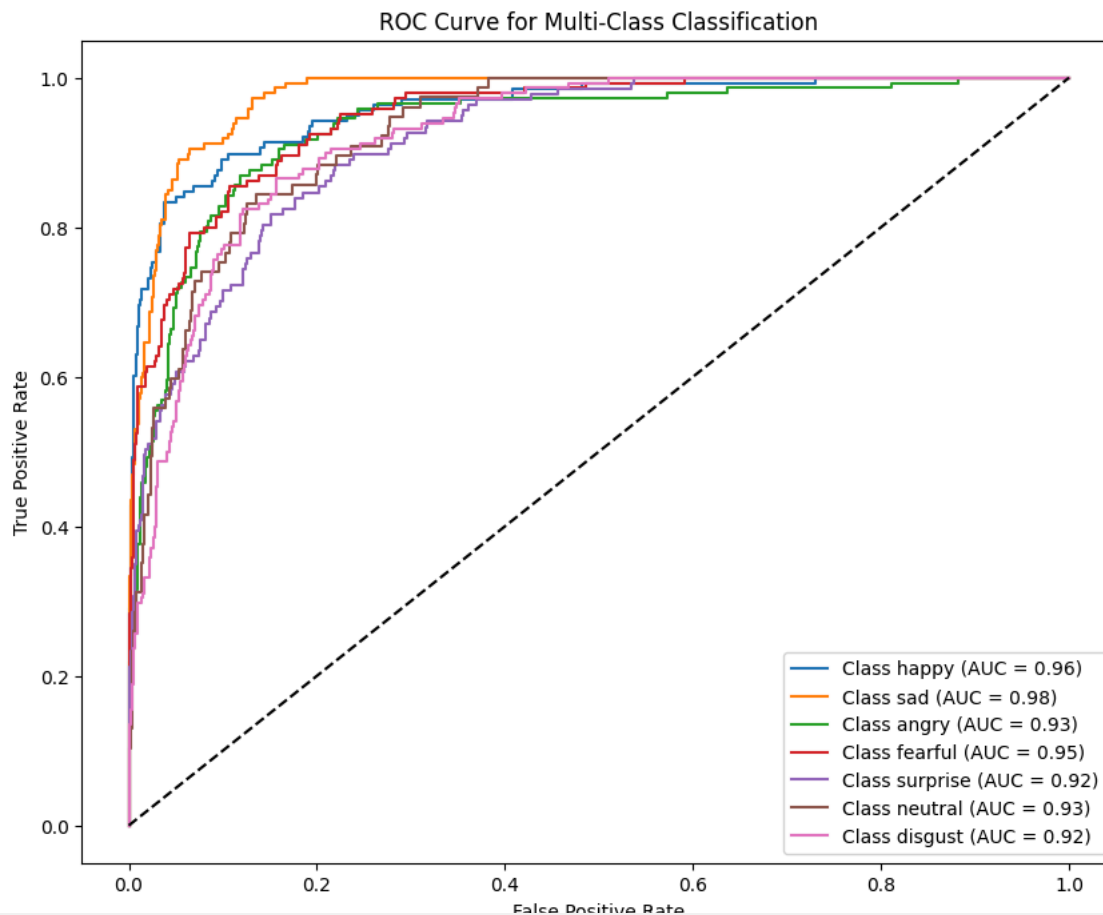
# Plot a random classifier line (diagonal line)
plt.plot([0, 1], [0, 1], 'k--')

# Title and labels
plt.title('ROC Curve for Multi-Class Classification')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc='lower right')

# Show the plot
plt.show()

```

34/34 — 0s 14ms/step
y_pred_bin shape: (1080, 8)



```

from sklearn.metrics import precision_recall_curve
from sklearn.metrics import average_precision_score

# Predict probabilities
y_pred_bin = model.predict(x_test)

# Binarize true labels
y_test_bin = label_binarize(np.argmax(y_test, axis=1), classes=np.arange(num_classes))

# Plot Precision-Recall curve for each class
plt.figure(figsize=(10, 8))

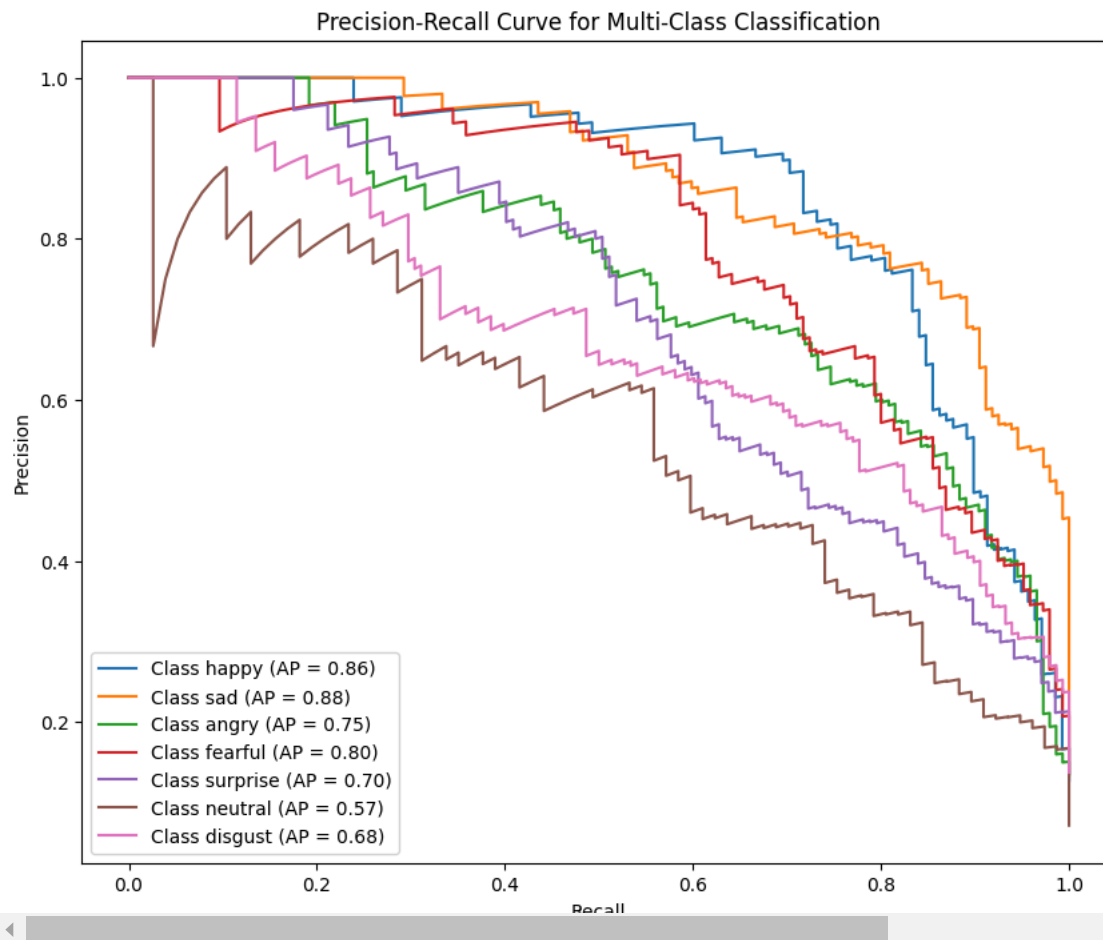
for i in range(num_classes):
    precision, recall, _ = precision_recall_curve(y_test_bin[:, i], y_pred_bin[:, i])
    average_score = average_precision_score(y_test_bin[:, i], y_pred_bin[:, i])
    plt.plot(recall, precision, label=f'Class {emotion_labels[i]} (AP = {average_precision:.2f})')

# Title and labels
plt.title('Precision-Recall Curve for Multi-Class Classification')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend(loc='lower left')

# Show the plot
plt.show()

```

34/34 1s 15ms/step



```

import numpy as np
import matplotlib.pyplot as plt

# Get predicted labels
y_pred_classes = np.argmax(y_pred_bin, axis=1)

# Get true labels
y_true_classes = np.argmax(y_test_bin, axis=1)

# Calculate class-wise accuracy
class_accuracies = []

```

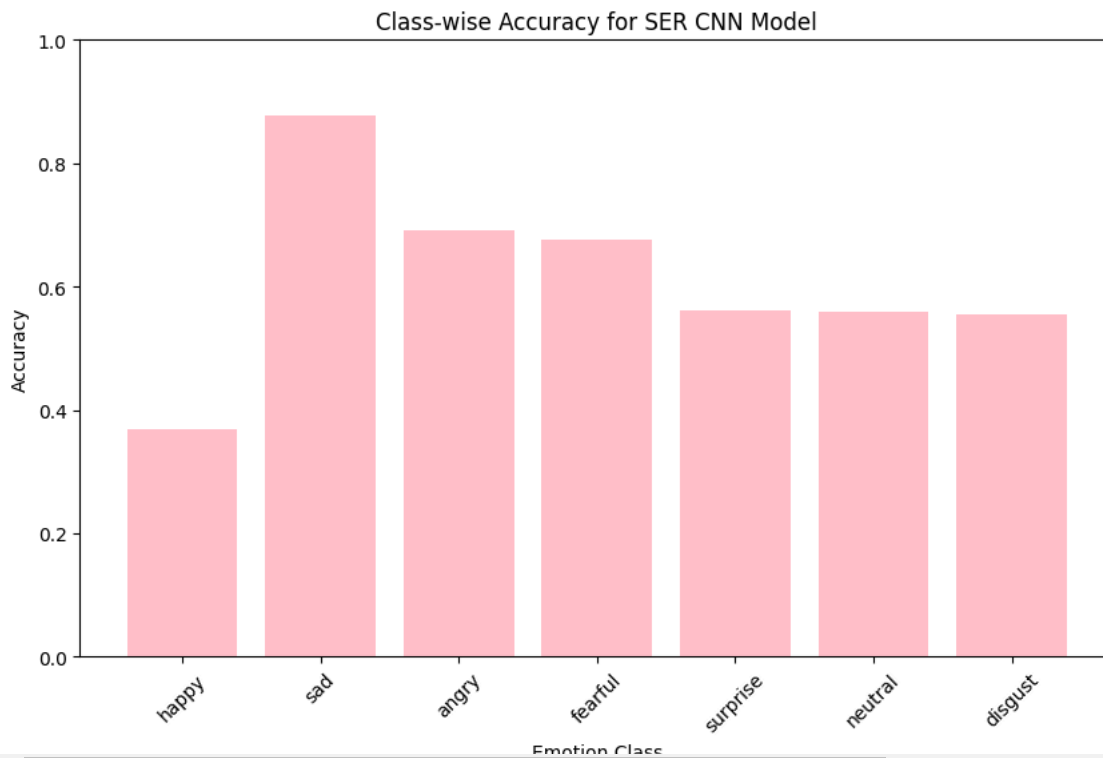


```

for i in range(num_classes):
    class_correct = np.sum((y_pred_classes == i) & (y_true_classes == i))
    class_total = np.sum(y_true_classes == i)
    class_accuracy = class_correct / class_total if class_total > 0 else 0
    class_accuracies.append(class_accuracy)

# Plot the class-wise accuracy bar chart
plt.figure(figsize=(10, 6))
plt.bar(emotion_labels, class_accuracies, color='pink')
plt.xlabel('Emotion Class')
plt.ylabel('Accuracy')
plt.title('Class-wise Accuracy for SER CNN Model')
plt.xticks(rotation=45)
plt.ylim(0, 1)
plt.show()

```



DEPLOYMENT

```
pip install gradio
```



Show hidden output

```
pip install tensorflow keras librosa gradio numpy scikit-learn matplotlib
```



Show hidden output

```
model.save('my_model.h5')
```



WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is consi

```

import gradio as gr
import numpy as np
import librosa
import os
from tensorflow.keras.models import load_model

```

```

# Load your trained model
try:
    print("Loading the model...")

```

```

model = load_model('/content/my_model.h5')
print("Model loaded successfully!")
print(f"Model expected input shape: {model.input_shape}")
except Exception as e:
    print(f"Error loading model: {e}")
    raise e

# Emotion labels (update based on your final label mapping)
emotion_labels = ['angry', 'fearful', 'disgust', 'surprised', 'happy', 'calm', 'neutral', 'sad']
print(f"Emotion labels: {emotion_labels}")

# Feature extraction function
def extract_features(audio_path):
    try:
        print(f"Extracting features from: {audio_path}")
        if not os.path.exists(audio_path):
            raise FileNotFoundError(f"File not found: {audio_path}")

        # Load the audio file
        data, sr = librosa.load(audio_path, duration=2.5, offset=0.6)
        print(f"Audio loaded successfully! Data shape: {data.shape}, Sample rate: {sr}")

        # Extract features
        zcr = np.mean(librosa.feature.zero_crossing_rate(y=data).T, axis=0)
        chroma = np.mean(librosa.feature.chroma_stft(y=data, sr=sr).T, axis=0)
        mfcc = np.mean(librosa.feature.mfcc(y=data, sr=sr, n_mfcc=13).T, axis=0)
        rms = np.mean(librosa.feature.rms(y=data).T, axis=0)
        mel = np.mean(librosa.feature.melspectrogram(y=data, sr=sr).T, axis=0)

        features = np.hstack([zcr, chroma, mfcc, rms, mel])
        print(f"Extracted features shape: {features.shape}")
        return features
    except Exception as e:
        print(f"Error during feature extraction: {e}")
        return None

# Prediction function
def predict_emotion(audio_file):
    try:
        print(f"Received audio file path: {audio_file}")

        if not os.path.exists(audio_file):
            error_message = f"Error: File not found at {audio_file}"
            print(error_message)
            return error_message

        # Extract features
        features = extract_features(audio_file)
        if features is None:
            return "Error: Unable to extract features from audio."

        print(f"Extracted features shape: {features.shape}")

        # Fix: Adjust feature shape to match the model
        expected_shape = 2816
        current_shape = features.shape[0]

        if current_shape < expected_shape:
            # Pad with zeros if too small
            features = np.pad(features, (0, expected_shape - current_shape), mode='constant')
            print(f"Features padded to shape: {features.shape}")
        elif current_shape > expected_shape:
            # Trim if too large
            features = features[:expected_shape]
            print(f"Features trimmed to shape: {features.shape}")

        # Prepare features for model input
        input_data = np.expand_dims(features, axis=0)
        input_data = np.expand_dims(input_data, axis=-1)
        print(f"Input data prepared for model. Shape: {input_data.shape}")

        # Predict emotion
        predictions = model.predict(input_data)
        print(f"Raw predictions: {predictions}")


        predicted_label = np.argmax(predictions)
        print(f"Predicted label index: {predicted_label}")

```

```
    predicted_emotion = emotion_labels[predicted_label]
    print(f"Predicted emotion: {predicted_emotion}")
    return predicted_emotion
except Exception as e:
    print(f"Error during prediction: {e}")
    return f"Error: {e}"

# Gradio Interface
iface = gr.Interface(
    fn=predict_emotion,
    inputs=gr.Audio(type="filepath"),
    outputs="text",
    title="Speech Emotion Recognition",
    description="Upload an audio file to predict the emotion."
)

print("Launching Gradio interface...")
iface.launch(share=True)
```

 WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you t
Loading the model...
Model loaded successfully!
Model expected input shape: (None, 162, 1)
Emotion labels: ['angry', 'fearful', 'disgust', 'surprised', 'happy', 'calm', 'neutral', 'sad']