ELEC 6601: Digital Signal Processing

Wavelet Multiresolution Analysis Based Speech Emotion Recognition System using 1D CNN LSTM Networks

Submitted by:

Talha Jabbar (40256009)

Vijay Kumar (40221480)

Professor: Dr. Khaled Humadi

A report submitted in partial fulfillment of the requirement of ELEC6601 Concordia University

Fall 2023

A. Abstract

Speech Emotion Recognition (SER) is the process of discerning a speaker's emotional state through speech signals, finding applications in human-computer interaction and psychological evaluation. Commonly, SER systems utilize time-frequency representations such as spectrograms, mel-frequency cepstrum coefficients (MFCCs), and mel-spectrograms, all derived from the Fast Fourier Transform (FFT) conversion of the time domain signal to the frequency domain. The FFT, however, faces a fundamental issue due to the uncertainty principle, which it does not allow a good resolution in both time and frequency domains.

To address this limitation, this paper explores the efficacy of wavelet transforms in SER, leveraging its multiresolution feature for improved localization in both frequency and time domains. The proposed method introduces the Wavelet-based Deep Emotion Recognition (WaDER) technique, which integrates long short-term memory (LSTM) networks, autoencoders, and 1D convolutional neural networks (CNNs). The study employs Monte-Carlo K-fold validation on an audio dataset which has a diverse set of emotional states, such as calm, neutral, sad, happy, angry, disgust, fearful and surprised.

The results reveal that the continuous wavelet transform features enable effective differentiation between various emotions, demonstrating an accurate and robust way to process audio signals and identify the emotional state of the speaker.

B. Introduction

Emotion recognition holds significant importance in human-computer interaction, and Speech Emotion Recognition (SER) serves as a key element in enhancing conversational intelligence between machines and humans. SER, a speech processing activity, aims to discern and categorize emotions conveyed through spoken language, analyzing speech patterns to identify a speaker's emotional state, encompassing sentiments like happiness, sadness, or frustration.

The impact of SER extends to healthcare, offering valuable insights into detecting mental health issues and mitigating the risk of suicidal behaviors. Additionally, SER goes beyond mere emotion detection, contributing to the realm of speech emotion generation, where it becomes a tool for generating human-like emotional speech. Notably, companies such as DeepZen, based in London, have developed deep learning models capable of creating emotionally expressive speech for applications like audiobooks.

Recognizing the acoustic variations inherent in speech emotions, the initial step involves identifying distinguishable and salient features within a voice segment to enhance SER's recognition rate. Traditionally, researchers have utilized features like Mel-frequency cepstrum coefficients (MFCC) and Mel-spectrograms. A Mel Spectrogram graphically represents the frequency spectrum's changes over time in an audio signal, while MFCCs capture the spectral properties of the audio signal through coefficients. Various methodologies have been employed for speech emotion recognition, including an approach proposed by Xu et al. at Head Fusion, utilizing a multi-head attention mechanism in conjunction with MFCC features. Another researcher employed Mel-spectrograms as features, training them with an attention-based model focusing on relevant input parts for decision-making.

In the pursuit of SER, some methods leverage raw audio, but recognizing the human auditory system's sensitivity to sound frequency and amplitude, this paper advocates for an approach utilizing wavelet transformations of an audio signal. The proposed method, employing continuous wavelet transform,

divides frequency bands into multiple levels, utilizing these levels as features to discern and recognize emotional states.

C. Problem Statement

The features (MFCC, mel-spectogram) which were discussed before uses Fast Fourier Transform (FFT) to transform the time domain signal to frequency domain signal. However, due to the uncertainty principle, FFT cannot obtain a good resolution in both the time and frequency domains at the same time. Heisenberg's uncertainty principle states the following:

1. A narrow window will localize the signal in time, but the frequency will be highly uncertain.
2. When the window is sufficiently large, the time uncertainty increases.

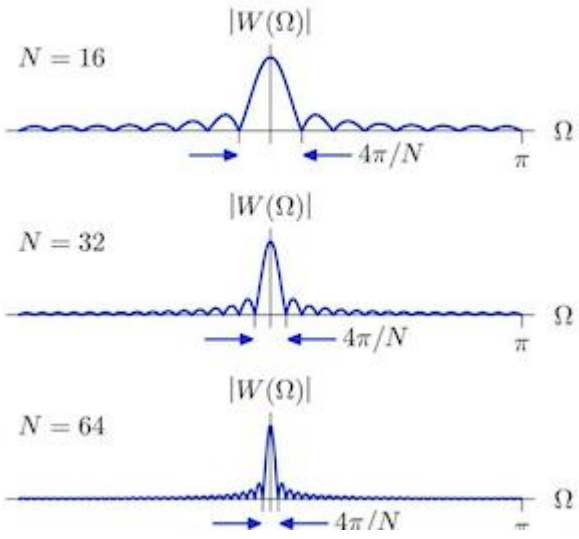


Figure 1. Low and High frequency signals

To capture different frequencies, FFT's employ a fixed-size window, necessitating a smaller window for higher frequencies and a larger one for lower frequencies. In contrast, the multiresolution characteristic of wavelet transforms allows for simultaneous localization in both time and frequency domains. Multiresolution Analysis (MRA), exemplified by Wavelet Transform, involves analyzing a signal at various resolution levels. Researchers have explored wavelet packets in Speech Emotion Recognition (SER), as these packets constitute a generalized multiresolution decomposition categorizing frequency bands into different levels. Wavelet packets also decompose high-frequency sections not segmented in traditional multiresolution analysis. Certain researchers have integrated deep learning algorithms to investigate wavelets for SER. Despite the high dimensional nature of wavelet transform features, advancements in computer resources now facilitate the development of neural networks using such data, overcoming previous computational constraints.

D. Proposed Solution

The proposed solution is to implement continuous wavelet transform (CWT) on the given audio signal which provides an overcomplete representation of a signal by letting the translation and scale parameter of a wavelet to vary continuously, this variation results in a output signal that captures certain frequencies at different given scales.

A wavelet by definition, is a mathematical function used in signal processing. It oscillates like a wave starting from 0 reaching up to its peak and return back to zero. Wavelets have the ability to be localized in both time and frequency domains, they capture low and high frequency components of a signal simultaneously due to their multiresolution properties. There exist multiple types of wavelet transforms, our focus will be on continuous wavelet transforms. Expanding more on wavelets, they have two essential parameters which are known as the scale and position. Scale describes how "stretched" or "compressed" a wavelet is and the position is a point in time. The CWT of a signal is represented by

$$W(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{+\infty} f(t)\ \tilde{\psi}\left(\frac{t-b}{a}\right) dt \tag{1}$$

The scale parameter denoted by "a", "b" is the translation parameter, $\psi(t)$ is the mother wavelet, $\tilde{\psi}$ (t) is the complex conjugate of the mother wavelet, $f(t)$ is the continuous time signal, and "t" is the time.

The scale parameter "a" of the wavelet transform can be varied such that different frequencies of the signal can be captured. Wavelets that have a compressed shape have the capability to capture high frequencies whereas wavelets that are stretched can capture low frequencies. **Figure 2** shown below represents the compressed and stretched wavelets that are created based on the scaling parameter.

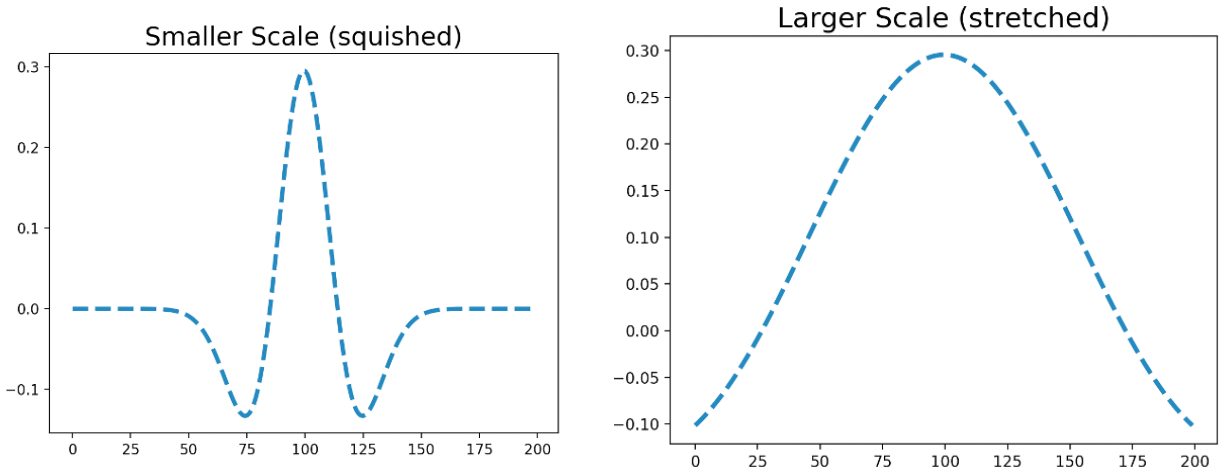


Figure 2. Compressed Wavelet (Left) - Stretched Wavelet (Right)

The benefit of applying wavelet transform instead of FFT, is that it can record spectral and temporal data at the same time. In simple terms, a signal is convolved with a series of wavelets of varying sizes and locations which in turn captures the frequencies identified by that scaling factor parameter. As the parameters "a" and "b" vary, different wavelets are generated from the mother wavelet and those generated wavelets are denoted as the daughter wavelets. There exists plenty of mother wavelets and each wavelet is used for a different task, but for this application the real-valued morlet wavelet is used, **Figure 3** represents the real-valued morlet wavelet and the wavelet is defined as such:

$$\psi(t) = \cos(\xi t)\, e^{\frac{-t^2}{2\sigma^2}} \tag{2}$$

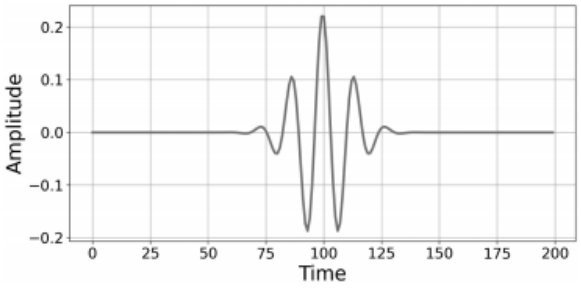


Figure 3. The real-valued morlet wavelet

Defining the wavelet function and its output, the algorithm requires discretization. Wavelet frequencies F_a are expressed as $\frac{F_c}{a\delta}$, where δ is the sampling period and F_c is the central wavelet frequency (set to 1 Hz). Considering the human hearing range from 20 Hz to 20 kHz, with most speech falling between 20 Hz and 4 kHz, discrete scales 'a' are chosen as positive integers from {1, 2, 3, ...N}. To align with the lower limit of human hearing, the choice of N ensures that the frequency corresponding to scale N exceeds 20 Hz. Additionally, translation values 'b' in the Continuous Wavelet Transform (CWT) are discretized as positive integers representing timesteps, denoted as 'b' in the set {1, 2, 3, ..., T}, where T is the total number of timesteps in the signal. This discretization process is crucial for effective algorithm implementation, aligning with human hearing frequencies and the input data's characteristics.

In dealing with continuous wavelet transformations, the generated data is highly non-linear. Since the application involves classifying the emotional state of audio samples, a neural network is employed. However, for effective classification, the data should be linearly separable. To achieve this, an autoencoder network is used to linearize and compress data dimensions. This process enhances the network's capacity to accurately identify and categorize emotional characteristics within the audio samples.

E. Neural Network Architecture used for Emotion Recognition

With the signal processing part defined and how the process of extracting wavelet features highlighted in the above chapter, the data obtained which as mentioned is highly non-linear. **Figure 4** shown below is a representation of the CWT features. Principle Component Analysis is applied to the CWT features and the figure shown below highlights that features are non-linear as there is a significant spread in multiple directions. One approach to solve the non-linearity of data is to propose a deep learning model that consists of an autoencoder model first and then followed by a classification model. The autoencoder will be used to reduce the dimensions of the features while keeping timesteps the same.

After optimizing the autoencoder, its latent space serves as the input for the classifier model. The overall network architecture, as illustrated in **Figure 5** below, features an autoencoder comprising an encoder and decoder structure. The encoder's role is compression, while the decoder's function is to decompress the data and reconstructing the compressed data to obtain the original input. Post-training and optimization, the latent space with dimensions (4000 x 8) becomes the input layer for the classifier model. This classifier model incorporates multiple convolutional layers to extract and construct feature maps, aiding in the final decision-making process to identify the emotional state of the provided audio sample.

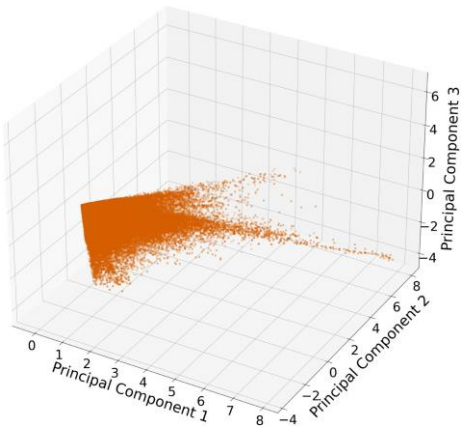
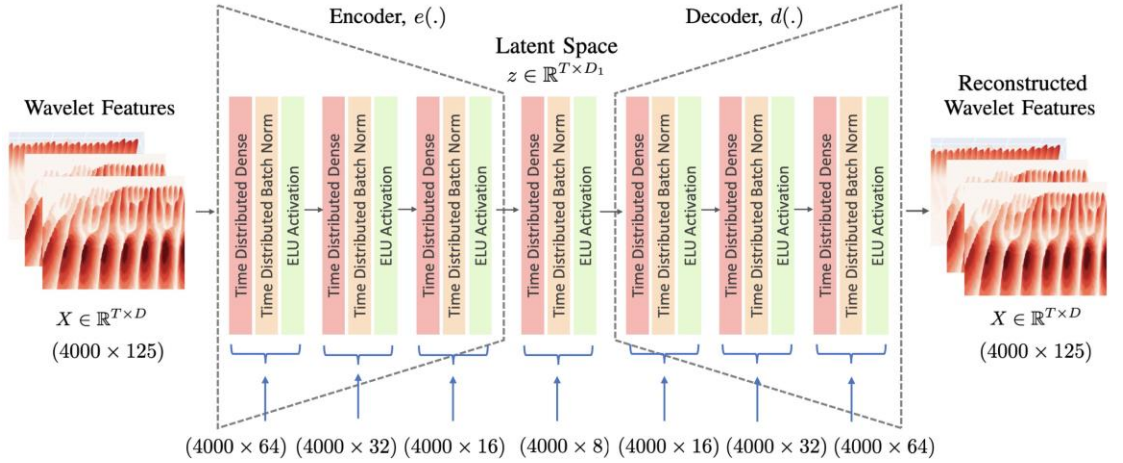
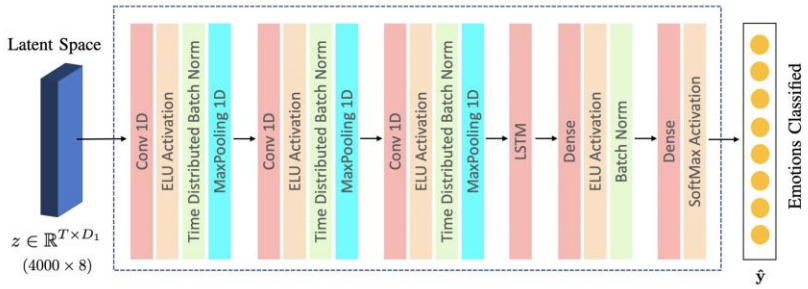


Figure 4. Scatter plot of the first two principal components. The PCA applied on the CWT features.



(a) Autoencoder Architecture. The shape of each layer's output is represented in the format (x × y), where x is the number of timesteps and y is the dimensionality of the feature maps.



(b) Classifier Architecture

Figure 5. Neural Network Architecture - Autoencoder with classifier architecture [1]

Talking about the architecture proposed a brief explanation of the layers used will be provided.

1. 1D CNN is a 1D convolutional layer that is used to extract the sequential data and learn the local features at each timestep
2. ELU Activation function is a function that can produce negative outputs which alleviates the effect of vanishing gradient during the weight optimization process of the network. The function itself has shown to lead to better generalization performance and is defined as such:

$$ELU(x) = \begin{cases} x & if\ x > 0 \\ \alpha(e^x - 1) & if\ x \leq 0 \end{cases} \quad (3)$$

3. Batch Normalization layer adjusts the means and variances of layer inputs by normalizing them. The addition of this layer makes the training process go faster and makes sure that output is not saturated.
4. Max Pooling layer is used to reduce the resolution of the CWT features. In other words, it performs down sampling operation.
5. The LSTM layer is a layer that excels in capturing and learning long term data within a sequential stream of data, making them useful for temporal pattern retention.

F. Dataset & Tools used

The dataset used to evaluate and test the proposed algorithm is the RAVDESS speech dataset which consists of 1440 audio files containing the following emotional tones, neutral, calm, happy, sad, angry, fearful, disgust, and surprised. Each recording has either the following two sentences spoken, “Dogs are sitting by the door” and “Kids are talking by the door” spoken in a different emotional tone. Due to computational limitations, 5% of the data (72 audio samples) were processed because converting the audio files into wavelet spectrums resulted in a large number of files being created since for each audio sample a wavelet transformation is obtained with the following file dimensions of 1x4000x125. Given this constraint the implementation will be done with 72 audio samples.

To implement the proposed solution, python programming language is used. Python has an extensive library for signal processing and building neural network models, the results of the audio processing and results obtained can be seen in the upcoming chapters. For signal processing the library librosa and pywt are used mainly since it already has built-in tools that simplify the process of computing the wavelet transform of the given audio signal. For building the model, Keras is used which operates with TensorFlow as the backend to train and construct models.

G. Audio Signal processing

Figure 6 shown above highlights the audio sample that has been preprocessed. The preprocessing consists of loading the audio signal, trimming trailing and leading silence, normalizing the audio file, and finally creating a new audio copy with AWGN added with an SNR between 15dB to 30dB.

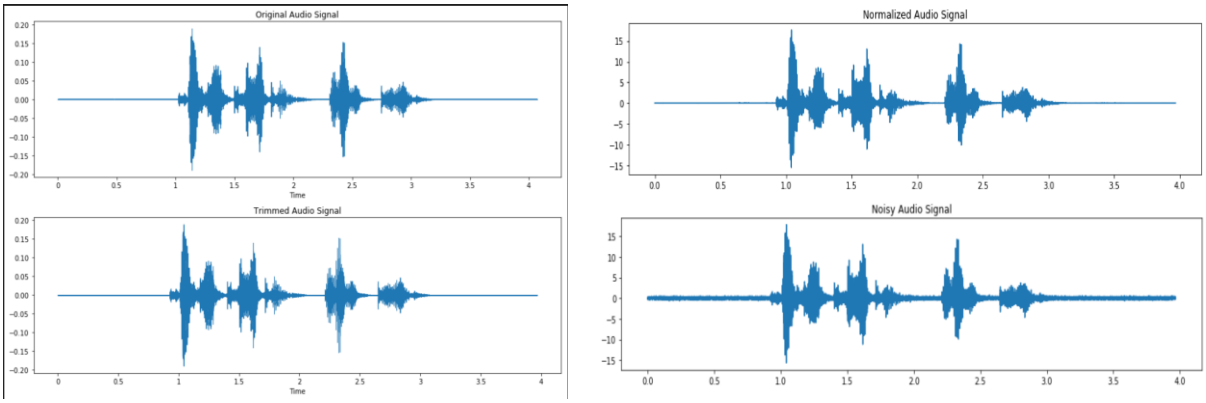


Figure 6. Audio Sample Preprocessed

The scalogram shown below in **Figure 7** highlights the fact that at a high scale low frequencies are being captured and at low level of the scales the high frequencies are being captured.

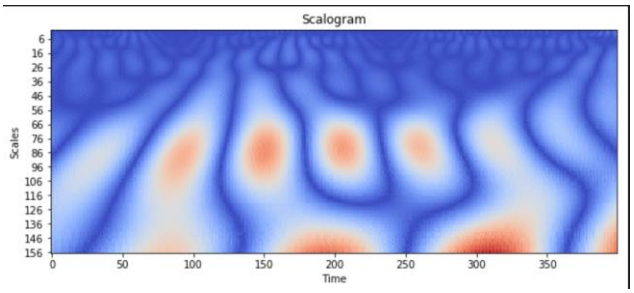


Figure 7. Audio Sample Scalogram

Figure 8 shown below is the CWT of multiple audio samples of different emotions, and as mentioned previously different level of scales capture different ranges of frequencies and that can be observed in the CWT obtained.

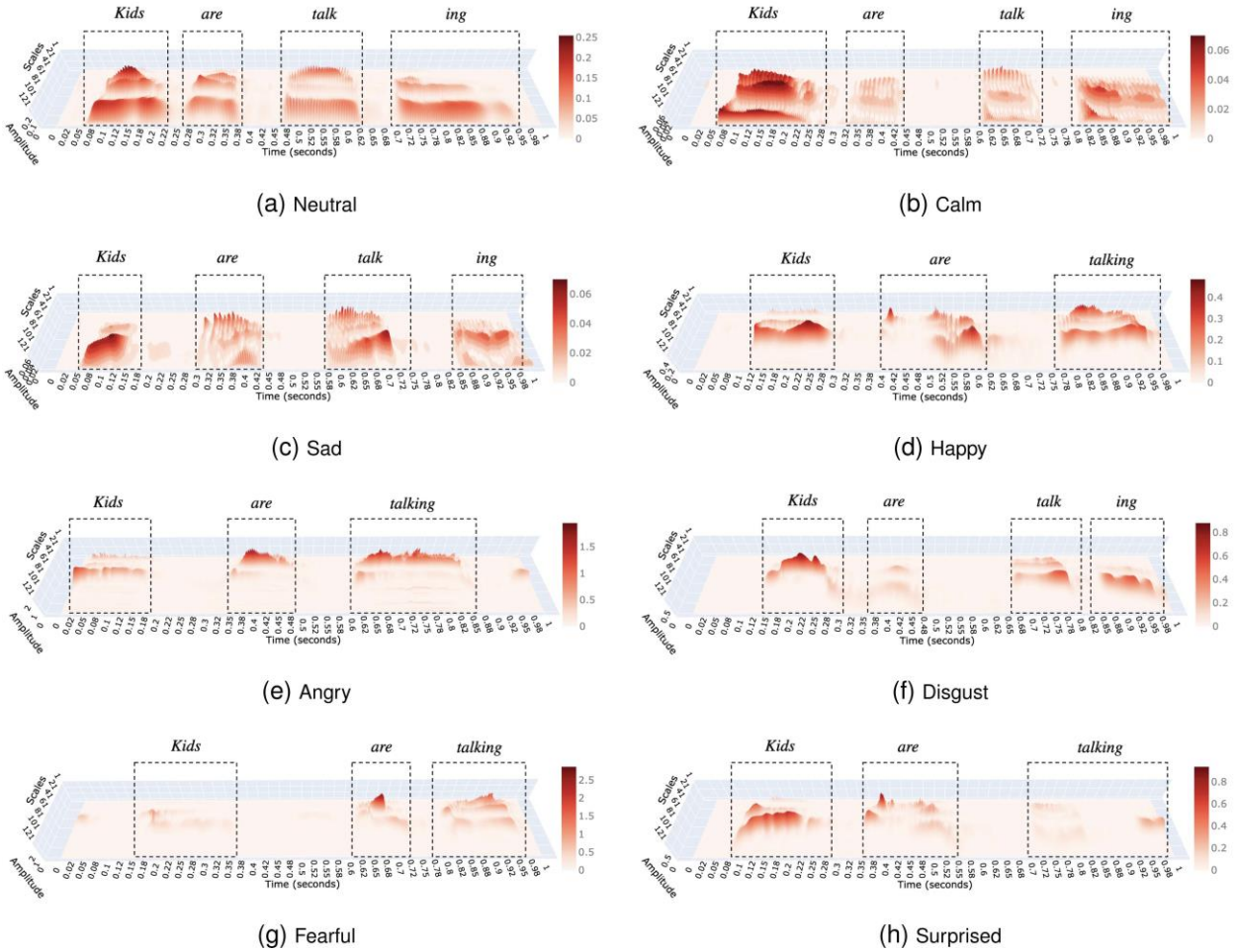


Figure 8. Continuous Wavelet Transform features of different emotions

Figure 9 shown below is the frequency distribution of the CWT and when comparing the two audio samples we can observe that majority of the distinct data lies between the range of 80Hz to 2000Hz, and

everything outside this range can be neglected as it has no significance. By applying a bandpass filter to filter out the selected range of frequencies we are able to reduce the size of the audio file and prioritize feature extraction and learning in the selected bandpass region.

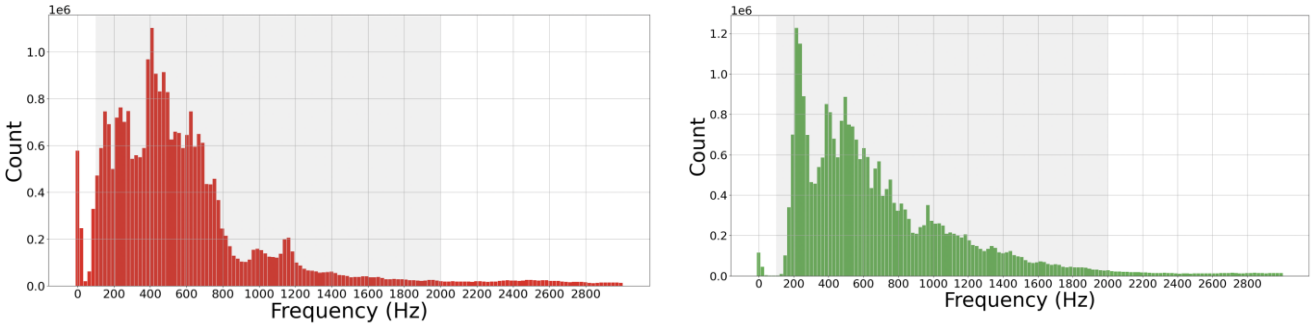


Figure 9. Histogram of 2 audio samples with frequencies weighted by their amplitudes

H. Results

Upon training the autoencoder model and the classifier model, we can observe the following results, The autoencoder has a mean Error of (1.405 ± 0.205) and a mean R^2 score of (0.84 ± 0.02). upon simple observation it can be said that the compression and decompression of data is successful with 84% of the data being able to be reconstructed. Results obtained in the paper have a slightly better performance of 92% success rate in reconstruction of the original audio features.

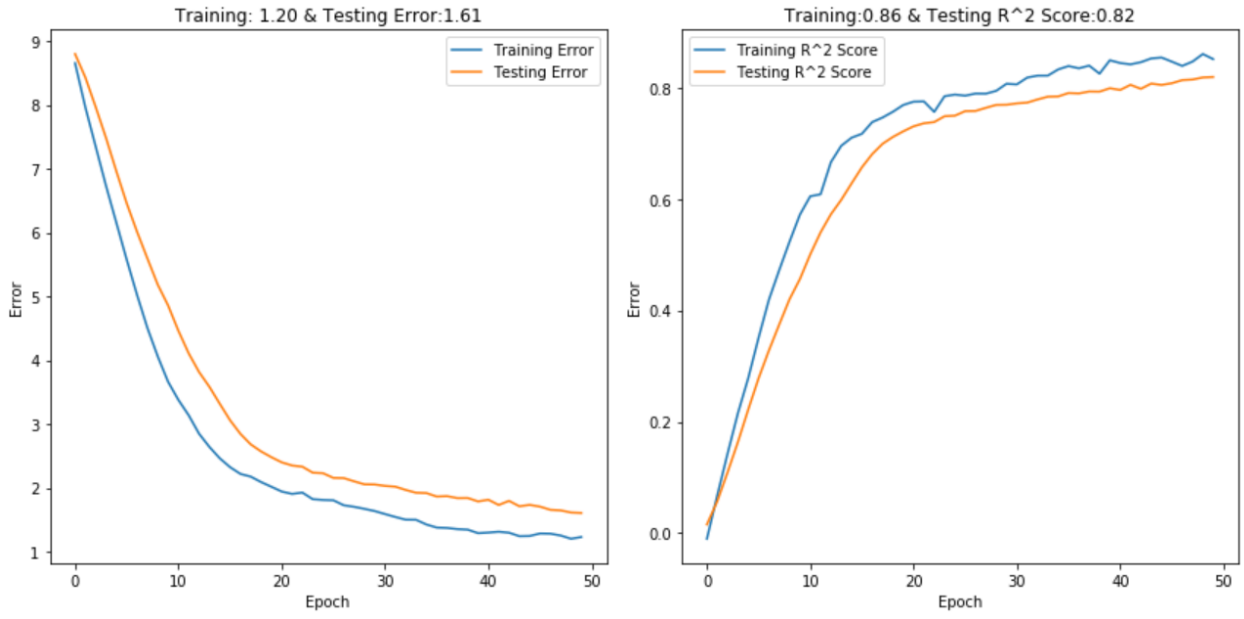


Figure 8. Autoencoder Error vs R^2 score

Moving onto the classifier which is the main model responsible for identifying the emotional state of the speaker, **Figure 11** shown below is a confusion matrix for the implemented model we can observe that it for the majority of the emotional states it is correctly classified but there seems to be a misclassification

between the fearful and disgust states. The classifier model has a weighted accuracy of 74% and unweighted accuracy of 75.78%. by simple comparison our implementation underperformed by 10%.

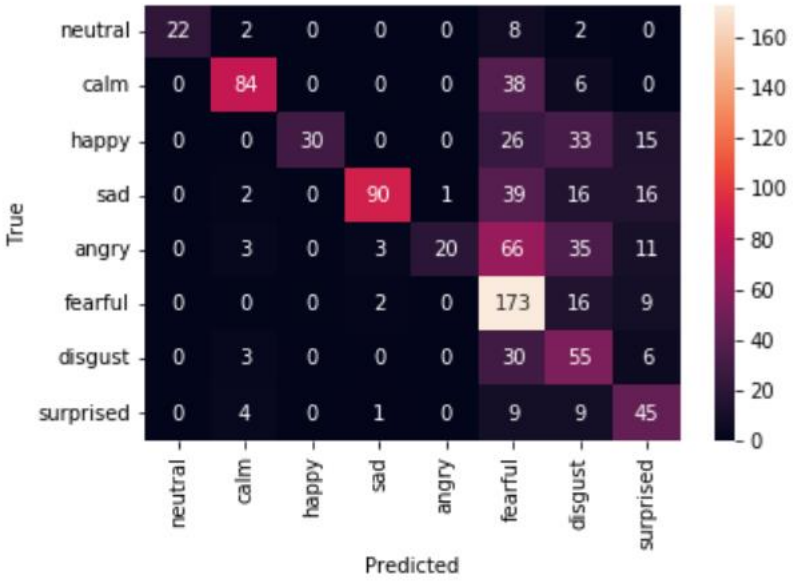


Figure 9. Classifier confusion matrix

I. Conclusion

The implementation of the proposed solution uses continuous wavelet transform as features to perform speech emotion recognition, for our implementation due to computational limitations using 5% of the audio samples to solve this problem wasn't feasible since when dealing with recognition applications you need sufficient amount of data to extract features from and with such limitation our results showcased that even with such constraint majority of the audio files where being labeled correctly but due to missing and reduced data some features were not able to be extracted.

The project covered different aspects of how to analyze audio signals using wavelet which proved to be effective in capturing a large range of frequencies where in the time or frequency domain you weren't able to do so. In addition, understanding and visualizing audio signals helped understand that certain ranges can be disregarded and other can be kept which in turn allows reduction in audio size, and finally the process of building a model to recognize the emotional states based on features extract proved to be a difficult task which took a lot of time but eventually we got a favorable outcome from the project.

J. References

[1] A. Dutt, "Wavelet Multiresolution Analysis Based Speech Emotion Recognition System using 1D CNN LSTM Networks," *IEEE/ACM TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING,* 2023.

K. Appendix A

```
import os
import numpy as np
import matplotlib.pyplot as plt

import pywt
import librosa
import librosa.display

from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split

import random
from random import seed, random, randint, sample
from scipy.signal import hilbert, chirp
from scipy.io import wavfile
from tqdm import tqdm
from scipy.interpolate import interp2d

get_ipython().run_line_magic('matplotlib', 'inline')

# using speech data
speech_folder_name = './Audio_Speech_Actors_01-24/'
actors_folder_name = [os.path.join(speech_folder_name, actor) for
actor in os.listdir(speech_folder_name)]
audio_files_path = [os.path.join(actor_num, file) for actor_num in
actors_folder_name for file in os.listdir(actor_num)]
data = np.array([[file_path, int(file_path.split('\\')[-1].split('-
')[2])-1] for file_path in audio_files_path])

labels = ['neutral', 'calm', 'happy', 'sad', 'angry', 'fearful',
'disgust', 'surprised']
print(f"{len(audio_files_path)} Audio files fetched...\n")
labels_idx, count = np.unique(data[:, -1], return_counts=True)
for i in range(len(count)):
    print(f"{labels[int(labels_idx[i])]} -> {count[i]} samples.")

def add_awgn(audio):
    snr_db = np.random.uniform(15, 30)
    noise_std = np.sqrt(np.var(audio) / (10 ** (snr_db / 10)))
    gaussian_noise = np.random.normal(0, noise_std, len(audio))
    return audio + gaussian_noise
```

```
def preprocess_audio(audio):
    trimmed, idx = librosa.effects.trim(audio)
    norm_seq = (trimmed - np.mean(trimmed)) / np.std(trimmed)
    noisy = add_awgn(norm_seq)

    return norm_seq, noisy

def compute_wavelet_features(audio, label):
    wavelet = 'morl'
    sr = 16000
    widths = np.arange(1, 256)
    #print(f"Scales using: {widths}")

    dt = 1/sr
    frequencies = pywt.scale2frequency(wavelet=wavelet, scale=widths)
/ dt
    #print(f"Frequencies associated with the scales: {frequencies}")

    #creating filter to select frequencies between 20Hz and 5Khz -
this is where most speech lies
    upper = [x for x in range(len(widths)) if frequencies[x] > 2000][-
1]
    lower = [x for x in range(len(widths)) if frequencies[x] < 100][0]

    widths = widths[upper:lower]

    #computing wavelet transform
    wavelet_coefs, freqs = pywt.cwt(audio, widths, wavelet=wavelet,
sampling_period=dt)
    #print(f"shape of wavelet transform: {wavelet_coefs.shape}")

    # Fixed Segment Generation
    start = 0
    end = wavelet_coefs.shape[1]
    frames = []
    frame_size = 4000
    count = 0

    while start + frame_size <= end -1:
        f = (wavelet_coefs)[:, start:start+frame_size]
        assert f.shape[1] == frame_size
        frames.append(f)
        start += frame_size

    frames = np.array(frames)
    frames = frames.reshape((len(frames), frame_size,
wavelet_coefs.shape[0]))
```

```
    labels = np.ones(shape=(len(frames), 1))* int(label)

    return frames, labels

#data = np.array([[file, int(file.split('\\')[-1].split('-')[2])-1]
for file in audio_files_path])

x_train, x_, y_train, y_ = train_test_split(data[:60, 0], data[:60, -
1], test_size=0.3, random_state=42)
x_val, x_test, y_val, y_test = train_test_split(x_, y_,
test_size=0.25, random_state=42)
labels = ['neutral', 'calm', 'happy', 'sad', 'angry', 'fearful',
'disgust', 'surprised']

print(f"Training: {x_train.shape}, labels: {y_train.shape}")
print(f"Validation: {x_val.shape}, labels: {y_val.shape}")
print(f"Testing: {x_test.shape}, labels: {y_test.shape}")

print(np.unique(y_train, return_counts=True))
print(np.unique(y_val, return_counts=True))
print(np.unique(y_test, return_counts=True))

# Training data saving
# Set a seed for reproducibility
seed(42)

# Initialize lists to store data
x_train_wavelet = []
y_train_wavelet = []
uniq_id = []

# Iterate over individual labels
count = 0
num_rand_samp = 100

for label_index in range(len(labels)):
    label_indices = np.where(y_train == str(label_index))[0]
    selected_indices = sample(label_indices.tolist(),
min(num_rand_samp, len(label_indices)))

    for audio_index in tqdm(selected_indices, desc=f"Label
{label_index}"):
        current_sample = x_train[audio_index]
        seq, _ = librosa.load(current_sample, sr=16000)
        normalised_audio, noisy_audio = preprocess_audio(audio=seq)

        for audio_type, audio_data in enumerate([normalised_audio,
noisy_audio]):
```

```
            features, labelss =
compute_wavelet_features(audio=audio_data, label=label_index)

            # Randomly sample from features
            indices = np.arange(len(features))
            selected_indices = sample(indices.tolist(),
min(num_rand_samp, len(indices)))
            selected_features = features[selected_indices]

            # Update lists
            uniq_id += [count] * len(selected_features)
            y_train_wavelet.extend(labelss)

            if count == 0:
                x_train_wavelet = selected_features
            else:
                x_train_wavelet = np.concatenate((x_train_wavelet,
selected_features), axis=0)

            count += 1

print(f"X: {x_train_wavelet.shape}")

y_train_wavelet = np.array(y_train_wavelet)
print("Y: ", y_train_wavelet.shape, " unique: ",
np.unique(y_train_wavelet, return_counts=True))

# Write all features to a .npz file
np.savez_compressed(os.getcwd()+"/training_features",
a=x_train_wavelet, b=y_train_wavelet)


# ### Validation Data saving...

# validation data saving
# Set a seed for reproducibility
seed(42)

# Initialize lists to store data
x_val_wavelet = []
y_val_wavelet = []
uniq_id = []

# Iterate over individual labels
count = 0
num_rand_samp = 100

for label_index in range(len(labels)):
```

```
    label_indices = np.where(y_val == str(label_index))[0]
    selected_indices = sample(label_indices.tolist(),
min(num_rand_samp, len(label_indices)))

    for audio_index in tqdm(selected_indices, desc=f"Label
{label_index}"):
        current_sample = x_val[audio_index]
        seq, _ = librosa.load(current_sample, sr=16000)
        normalised_audio, noisy_audio = preprocess_audio(audio=seq)

        for audio_type, audio_data in enumerate([normalised_audio,
noisy_audio]):
            features, labelss =
compute_wavelet_features(audio=audio_data, label=label_index)

            # Randomly sample from features
            indices = np.arange(len(features))
            selected_indices = sample(indices.tolist(),
min(num_rand_samp, len(indices)))
            selected_features = features[selected_indices]

            # Update lists
            uniq_id += [count] * len(selected_features)
            y_val_wavelet.extend(labelss)

            if count == 0:
                x_val_wavelet = selected_features
            else:
                x_val_wavelet = np.concatenate((x_val_wavelet,
selected_features), axis=0)

            count += 1

print(f"X: {x_val_wavelet.shape}")

y_val_wavelet = np.array(y_val_wavelet)
print("Y: ", y_val_wavelet.shape, " unique: ",
np.unique(y_val_wavelet, return_counts=True))
# Write all features to a .npz file
np.savez_compressed(os.getcwd()+"/validation_features",
a=x_val_wavelet, b=y_val_wavelet)


# ### Testing data saving...

# validation data saving
# Set a seed for reproducibility
seed(42)
```

```
# Initialize lists to store data
x_test_wavelet = []
y_test_wavelet = []
uniq_id = []

# Iterate over individual labels
count = 0
num_rand_samp = 100

for label_index in range(len(labels)):
    label_indices = np.where(y_test == str(label_index))[0]
    selected_indices = sample(label_indices.tolist(),
min(num_rand_samp, len(label_indices)))

    for audio_index in tqdm(selected_indices, desc=f"Label
{label_index}"):
        current_sample = x_test[audio_index]
        seq, _ = librosa.load(current_sample, sr=16000)
        normalised_audio, noisy_audio = preprocess_audio(audio=seq)

        for audio_type, audio_data in enumerate([normalised_audio,
noisy_audio]):
            features, labelss =
compute_wavelet_features(audio=audio_data, label=label_index)

            # Randomly sample from features
            indices = np.arange(len(features))
            selected_indices = sample(indices.tolist(),
min(num_rand_samp, len(indices)))
            selected_features = features[selected_indices]

            # Update lists
            uniq_id += [count] * len(selected_features)
            y_test_wavelet.extend(labelss)

            if count == 0:
                x_test_wavelet = selected_features
            else:
                x_test_wavelet = np.concatenate((x_test_wavelet,
selected_features), axis=0)

            count += 1

print(f"X: {x_test_wavelet.shape}")
```

```
y_test_wavelet = np.array(y_test_wavelet)
print("Y: ", y_test_wavelet.shape, " unique: ",
np.unique(y_test_wavelet, return_counts=True))
# Write all features to a .npz file
np.savez_compressed(os.getcwd()+"/testing_features", a=x_test_wavelet,
b=y_test_wavelet)
```

```
import os
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split
import seaborn as sns

from random import seed, random, randint, sample

from keras.models import Model, save_model, load_model
from keras.layers import Input, Dense, BatchNormalization,
TimeDistributed, ELU
from keras.optimizers import Adam, SGD
from keras import backend as K
from keras.utils import to_categorical

import pickle as pkl

get_ipython().run_line_magic('matplotlib', 'inline')

def r_squared(y_true, y_pred):
    SS_res =  K.sum(K.square(y_true - y_pred))
    SS_tot = K.sum(K.square(y_true - K.mean(y_true)))
    return 1 - SS_res/(SS_tot + K.epsilon())

# Input layer
input_layer = Input(shape=(4000, 125))

# Encoder
encoder_layer_1 = TimeDistributed(Dense(64))(input_layer)
encoder_layer_1 =
TimeDistributed(BatchNormalization())(encoder_layer_1)
encoder_layer_1 = TimeDistributed(ELU())(encoder_layer_1)

encoder_layer_2 = TimeDistributed(Dense(16))(encoder_layer_1)
```

```
encoder_layer_2 =
TimeDistributed(BatchNormalization())(encoder_layer_2)
encoder_layer_2 = TimeDistributed(ELU())(encoder_layer_2)

encoder_layer_3 = TimeDistributed(Dense(16))(encoder_layer_2)
encoder_layer_3 =
TimeDistributed(BatchNormalization())(encoder_layer_3)
encoder_layer_3 = TimeDistributed(ELU())(encoder_layer_3)

# Latent Space
latent_space = TimeDistributed(Dense(8))(encoder_layer_3)
latent_space = TimeDistributed(BatchNormalization())(latent_space)
latent_space = TimeDistributed(ELU(),
name="latent_space_out")(latent_space)

# Decoder
decoder_layer_1 = TimeDistributed(Dense(16))(latent_space)
decoder_layer_1 =
TimeDistributed(BatchNormalization())(decoder_layer_1)
decoder_layer_1 = TimeDistributed(ELU())(decoder_layer_1)

decoder_layer_2 = TimeDistributed(Dense(32))(decoder_layer_1)
decoder_layer_2 =
TimeDistributed(BatchNormalization())(decoder_layer_2)
decoder_layer_2 = TimeDistributed(ELU())(decoder_layer_2)

decoder_layer_3 = TimeDistributed(Dense(64))(decoder_layer_2)
decoder_layer_3 =
TimeDistributed(BatchNormalization())(decoder_layer_3)
decoder_layer_3 = TimeDistributed(ELU())(decoder_layer_3)

# Output layer
output_layer = TimeDistributed(Dense(125))(decoder_layer_3)

# Autoencoder Model
autoencoder = Model(inputs=input_layer, outputs=output_layer)

# Display the model summary
autoencoder.summary()

autoencoder.compile(optimizer=Adam(lr=0.001),
loss='mean_squared_error', metrics=[r_squared])

training_data = np.load('training_features.npz')
validation_data = np.load('validation_features.npz')
testing_data = np.load('testing_features.npz')

x_train, y_train = training_data['a'], training_data['b']
```

```
x_val, y_val = validation_data['a'], validation_data['b']
x_test, y_test = testing_data['a'], testing_data['b']
x_test = np.concatenate((x_val, x_test), axis=0)
y_test = np.concatenate((y_val, y_test), axis=0)

print(f"Training-> {x_train.shape},  {y_train.shape}")
#print(f"Validation-> {x_val.shape},  {y_val.shape}")
print(f"Testing-> {x_test.shape},  {y_test.shape}")
# label shape formatting
labels = ['neutral', 'calm', 'happy', 'sad', 'angry', 'fearful',
'disgust', 'surprised']
num_classes = len(labels)
y_train = to_categorical(y_train, num_classes)
#y_val = to_categorical(y_val, num_classes)
y_test = to_categorical(y_test, num_classes)
print(f"Training hot label-> {y_train.shape}")
#print(f"Validation hot label-> {y_val.shape}")
print(f"Testing hot label-> {y_test.shape}")

history1 = autoencoder.fit(x_train, x_train, epochs=50,
batch_size=128, shuffle=True, validation_data=(x_test, x_test))

with open('./autoencoder_history.pkl', 'wb') as file:
    pkl.dump(history1.history, file)
with open('./autoencoder_history.pkl', 'rb') as f:
    data = pkl.load(f)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
min_loss = min(data['loss'])
min_val_loss = min(data['val_loss'])
plt.plot(data['loss'], label='Training Error')
plt.plot(data['val_loss'], label='Testing Error')
plt.title(f'Training: {min_loss:.2f} & Testing
Error:{min_val_loss:.2f}')
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.legend()

max_r = max(data['r_squared'])
max_val_r = max(data['val_r_squared'])
plt.subplot(1, 2, 2)
plt.plot(data['r_squared'], label='Training R^2 Score')
plt.plot(data['val_r_squared'], label='Testing R^2 Score')
plt.title(f'Training:{max_r:.2f} & Testing R^2 Score:{max_val_r:.2f}')
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.legend()
```

```
plt.tight_layout()
plt.show()
save_model(autoencoder, './autoencoder_model.h5')
print("Autoencoder Model Saved...")
#autoencoder = load_model('./autoencoder_model.h5')

from keras.models import Model
from keras.layers import Input, Conv1D, ELU, BatchNormalization,
MaxPooling1D, LSTM, Dense, Softmax
from keras.metrics import categorical_accuracy, Precision, Recall

def unweighted_accuracy(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)),
axis=0)
    class_samples = K.sum(y_true, axis=0)

    unweighted_accuracy = K.sum(true_positives /
K.maximum(class_samples, 1)) / K.sum(K.cast(class_samples > 0,
'float32'))
    return unweighted_accuracy

def weighted_accuracy(y_true, y_pred):
    class_weights = K.sum(y_true, axis=0)  # Assuming one-hot encoded
labels
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)),
axis=0)
    total_samples = K.sum(class_weights)

    weighted_accuracy = K.sum((true_positives /
K.maximum(class_weights, 1)) * (class_weights / total_samples))
    return weighted_accuracy

# Assuming 'autoencoder' is your trained autoencoder model
# Assuming 'latent_space_layer_name' is the name of the latent space
layer in your autoencoder

# Extracting the encoder part from the autoencoder model
encoder_model = Model(inputs=autoencoder.input,
outputs=autoencoder.get_layer("latent_space_out").output,
name='encoder_model')

# Freeze the layers of the encoder during classification
for layer in encoder_model.layers:
    layer.trainable = False

# Classifier architecture on top of the encoder
classifier_input = encoder_model.output
```

```
# Convolutional layers
classifier_output = Conv1D(128, kernel_size=3, padding='same',
name='conv1')(classifier_input)
classifier_output = ELU(name='elu1')(classifier_output)
classifier_output =
TimeDistributed(BatchNormalization(name='batch_norm1'))(classifier_out
put)
classifier_output = MaxPooling1D(pool_size=2,
name='maxpool1')(classifier_output)

classifier_output = Conv1D(256, kernel_size=3, padding='same',
name='conv2')(classifier_output)
classifier_output = ELU(name='elu2')(classifier_output)
classifier_output =
TimeDistributed(BatchNormalization(name='batch_norm2'))(classifier_out
put)
classifier_output = MaxPooling1D(pool_size=2,
name='maxpool2')(classifier_output)

classifier_output = Conv1D(512, kernel_size=3, padding='same',
name='conv3')(classifier_output)
classifier_output = ELU(name='elu3')(classifier_output)
classifier_output =
TimeDistributed(BatchNormalization(name='batch_norm3'))(classifier_out
put)
classifier_output = MaxPooling1D(pool_size=2,
name='maxpool3')(classifier_output)

# LSTM layer
classifier_output = LSTM(128, name='lstm')(classifier_output)

# Dense layers
classifier_output = Dense(128, name='dense1')(classifier_output)
classifier_output = ELU(name='elu_dense1')(classifier_output)
classifier_output =
BatchNormalization(name='batch_norm_dense1')(classifier_output)

classifier_output = Dense(8, activation='softmax',
name='output')(classifier_output)

# Create the classifier model
classifier_model = Model(inputs=encoder_model.input,
outputs=classifier_output, name='classifier_model')

# Compile the classifier model
classifier_model.compile(optimizer=SGD(lr=0.0001),
loss='categorical_crossentropy', metrics=['accuracy',
```

```
weighted_accuracy, unweighted_accuracy])

# Display the classifier model summary
classifier_model.summary()
# Train the final model
history2 = classifier_model.fit(
    x_train, y_train,
    epochs=50,
    batch_size=64,
    shuffle=True,
    validation_data=(x_test, y_test))

with open('./classifier_history.pkl', 'wb') as file:
    pkl.dump(history2.history, file)

save_model(classifier_model, './classifier_model.h5')
print("classifier Model Saved...")


y_pred_train = classifier_model.predict(x_train)
#y_pred_val = classifier_model.predict(x_val)
y_pred_test = classifier_model.predict(x_test)

y_pred_train_labels = np.argmax(y_pred_train, axis=1)
#y_pred_val_labels = np.argmax(y_pred_val, axis=1)
y_pred_test_labels = np.argmax(y_pred_test, axis=1)

cm1 = confusion_matrix(np.argmax(y_train, axis=1),
y_pred_train_labels)
#cm2 = confusion_matrix(np.argmax(y_val, axis=1), y_pred_val_labels)
cm3= confusion_matrix(np.argmax(y_test, axis=1), y_pred_test_labels)

sns.heatmap(cm1, annot=True, fmt='d', xticklabels=labels,
yticklabels=labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
sns.heatmap(cm3, annot=True, fmt='d', xticklabels=labels,
yticklabels=labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```