Speech Emotion Recognition

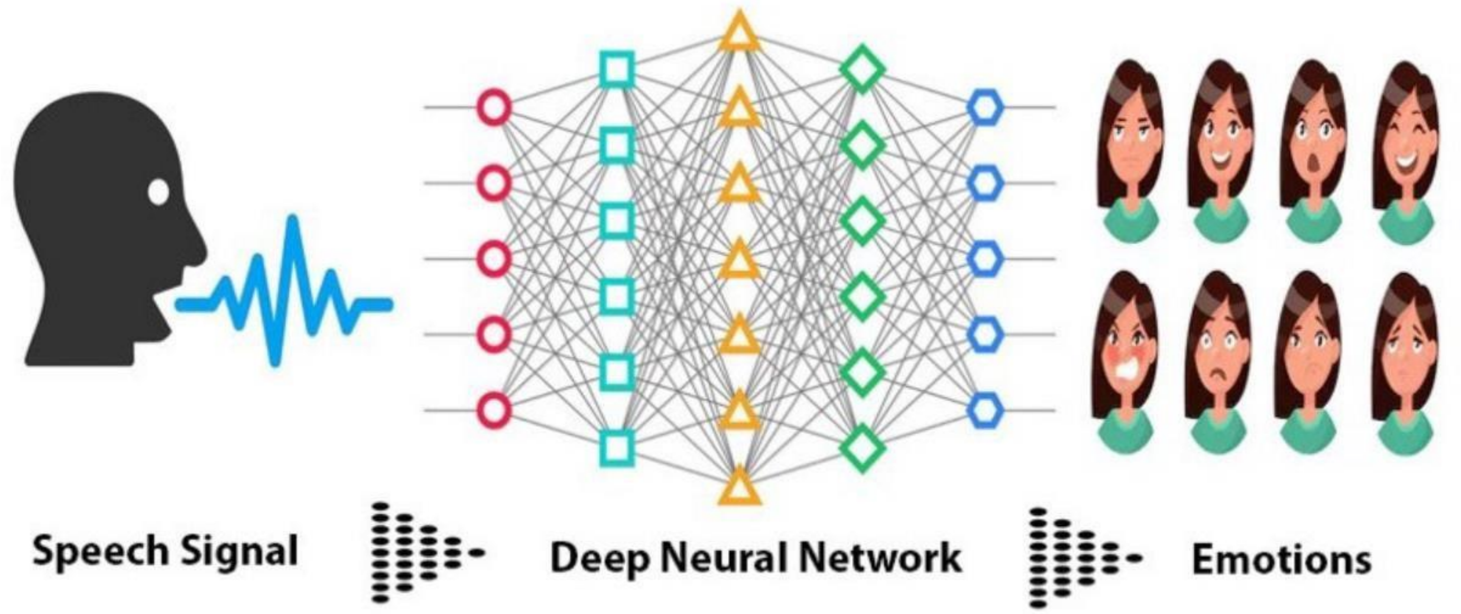


Table of Contents

1.Crema Dataset of Audios ____________________________________2

2.Imports ____2

3.Download the Dataset and Understand the Format _______________2

4. Load an audio and Listen to instances of the classes you have and plot the waveform of the audio____3

5.Split the Dataset into Training and Test sets______________________4

5.1. Split the data into 70% training and validation and 30% testing

5.2. Use 5% of the training and validation data for validation

6. Data Augmentation____5

6.1. What is data augmentation

6.2. Benefits of data augmentation

6.3. Time Stretch

6.4. Pitch Shift

7. Fix Length of audios____6

7.1. Get target length

7.2. Fix Length

7.3. Check audio length

8. Create the feature space_____________________________________8

8.1. Get first factor

7.2. 1D Feature Space

7.3. 2D Feature Space

7.4. Display mel_spectogram

7.5. Create the feature space for specific indices in the dataset

9. Encoding____12

10.Model ____12

10. 1.What is CNN

10.2. 1D CNN

10.2.1.Why is 1D CNN suitable

10.2.2.Our implemented 1D Model

10.2.3.Explanation

10.2. 4.Hyperparameters

10.3. 2D CNN

10.3.1.Why is 2D CNN suitable

10.3.2.Our implemented 1D Model

10.3.3.Explanation

10.3.4.Hyperparameters

11.Measures of accuracy ______________________________________19

12.Runs____20

13.Comparison between 1D, 2D according to measures obtained______27

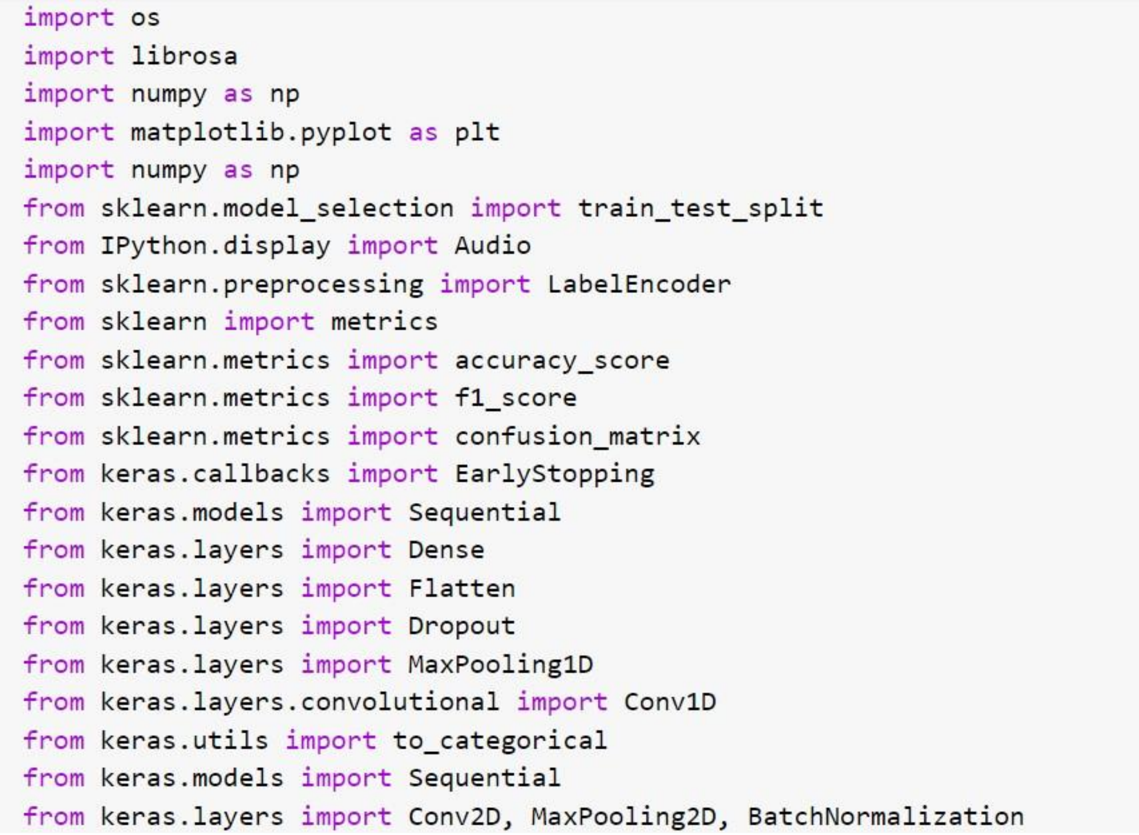
14.Referenced paper____28

Google CoLab Notebook link:
https://colab.research.google.com/drive/1IP1dqElzHnssMYlivaxPhLPRUHWgaeJ-?usp=sharing

1. Crema Dataset of Audios

The CREMA-D (Crowd-Sourced Emotional Multimodal Actors Dataset) is a commonly referenced dataset for Speech Emotion Recognition (SER) tasks. It contains a collection of audio and video recordings of actors portraying various emotions. The CREMA-D dataset includes 6 emotions: Anger, Disgust, Fear, Happy, Neutral, and Sad. It consists of 7,442 audio files from 91 actors, providing a diverse set of emotional expressions. Each audio file is approximately 4 seconds long and is labelled with the corresponding emotion category.

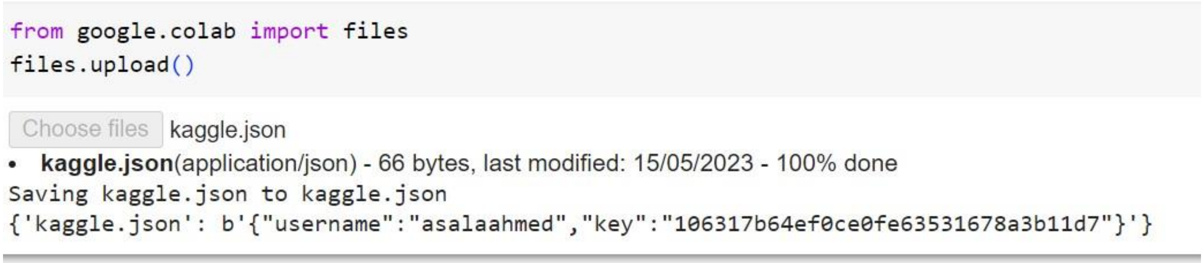
2.Imports



```
import os
import librosa
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from IPython.display import Audio
from sklearn.preprocessing import LabelEncoder
from sklearn import metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix
from keras.callbacks import EarlyStopping
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import MaxPooling1D
from keras.layers.convolutional import Conv1D
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
```

3.Download the Dataset and Understand the Format

this code snippet sets up the Kaggle API in Google Colab by uploading the kaggle.json file, installs the Kaggle API client, downloads the CREMA dataset using the Kaggle API, and unzips the downloaded dataset for further use in the Colab environment.

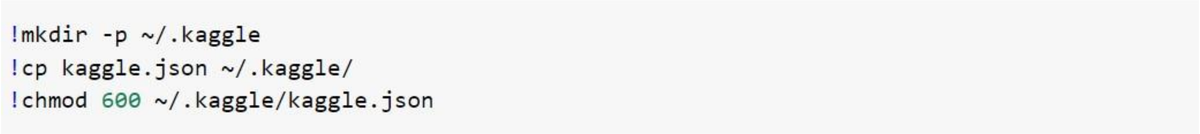


```
from google.colab import files
files.upload()
```

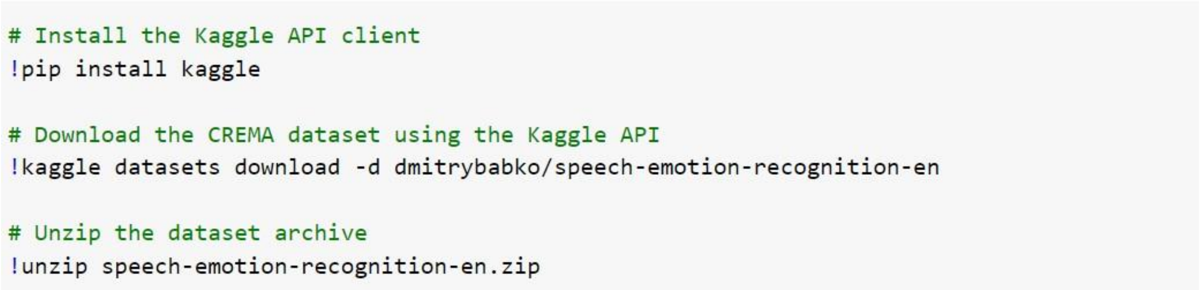
Choose files kaggle.json

- **kaggle.json**(application/json) - 66 bytes, last modified: 15/05/2023 - 100% done

```
Saving kaggle.json to kaggle.json
{'kaggle.json': b'{"username":"asalaahmed","key":"106317b64ef0ce0fe63531678a3b11d7"}'}
```

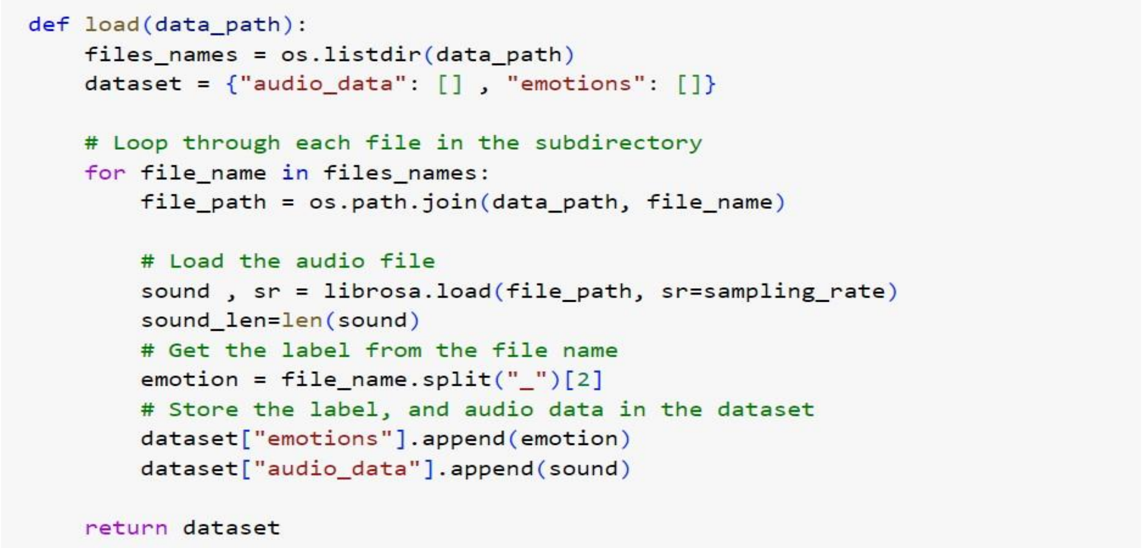


```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
```

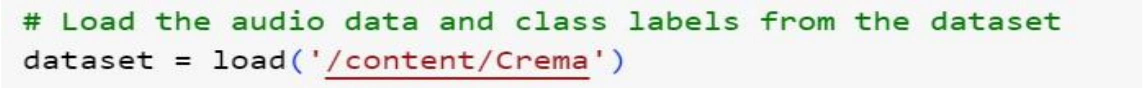


```
# Install the Kaggle API client
!pip install kaggle

# Download the CREMA dataset using the Kaggle API
!kaggle datasets download -d dmitrybabko/speech-emotion-recognition-en

# Unzip the dataset archive
!unzip speech-emotion-recognition-en.zip
```

4. Load, Listen, and plot the waveform of the audio

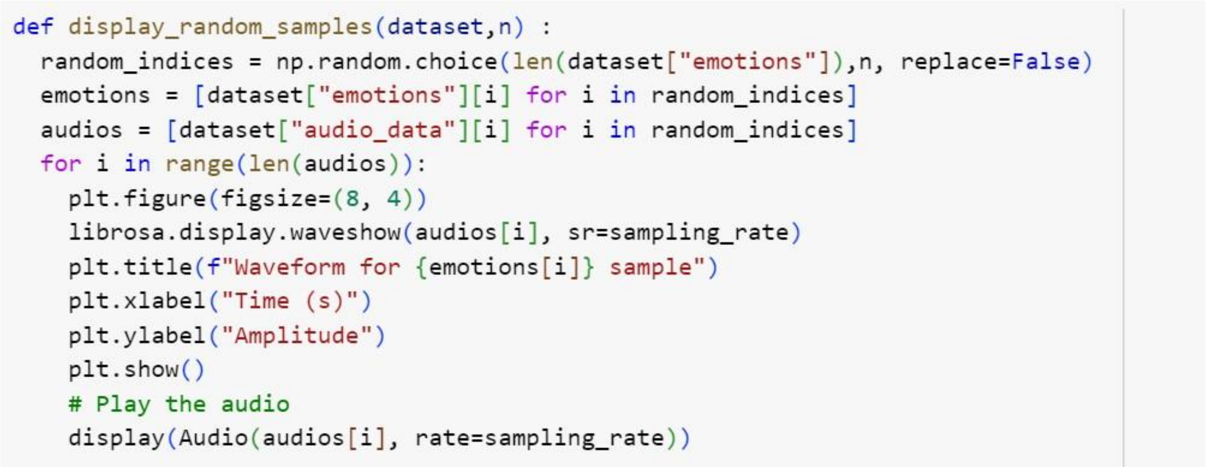


```
def load(data_path):
    files_names = os.listdir(data_path)
    dataset = {"audio_data": [] , "emotions": []}

    # Loop through each file in the subdirectory
    for file_name in files_names:
        file_path = os.path.join(data_path, file_name)

        # Load the audio file
        sound , sr = librosa.load(file_path, sr=sampling_rate)
        sound_len=len(sound)
        # Get the label from the file name
        emotion = file_name.split("_")[2]
        # Store the label, and audio data in the dataset
        dataset["emotions"].append(emotion)
        dataset["audio_data"].append(sound)

    return dataset
```



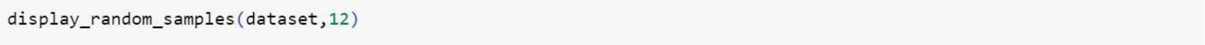
```
# Load the audio data and class labels from the dataset
dataset = load('/content/Crema')
```

you need to provide the data_path, which is the path to the directory containing the audio files. The function then loads each audio file using librosa.load() and extracts the emotion label from the file name. The audio data with sr=16000 and emotion labels are stored in the dataset dictionary, which is returned at the end.

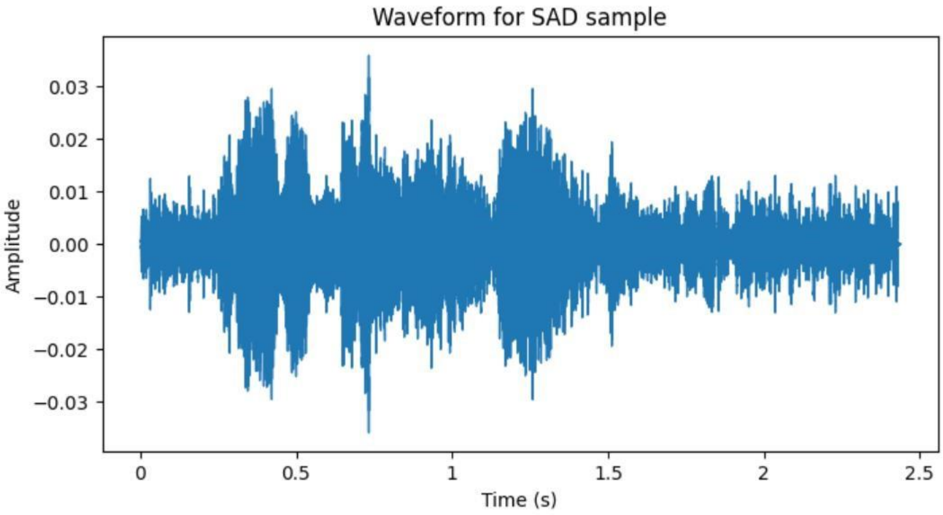


```
def display_random_samples(dataset,n) :
  random_indices = np.random.choice(len(dataset["emotions"]),n, replace=False)
  emotions = [dataset["emotions"][i] for i in random_indices]
  audios = [dataset["audio_data"][i] for i in random_indices]
  for i in range(len(audios)):
    plt.figure(figsize=(8, 4))
    librosa.display.waveshow(audios[i], sr=sampling_rate)
    plt.title(f"Waveform for {emotions[i]} sample")
    plt.xlabel("Time (s)")
    plt.ylabel("Amplitude")
    plt.show()
    # Play the audio
    display(Audio(audios[i], rate=sampling_rate))
```

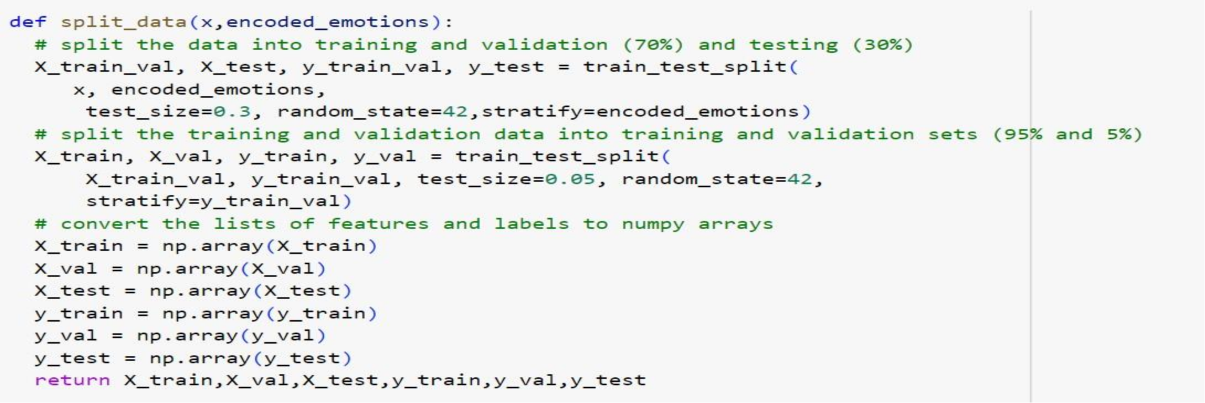
The **display_random_samples** function allows you to randomly select **n** audio samples from a dataset and display their waveforms along with playing the audio. It provides a visual and auditory representation of the selected samples.



```
display_random_samples(dataset,12)
```



5.Split the Dataset into Training and Test sets



```
def split_data(x,encoded_emotions):
  # split the data into training and validation (70%) and testing (30%)
  X_train_val, X_test, y_train_val, y_test = train_test_split(
     x, encoded_emotions,
      test_size=0.3, random_state=42,stratify=encoded_emotions)
  # split the training and validation data into training and validation sets (95% and 5%)
  X_train, X_val, y_train, y_val = train_test_split(
      X_train_val, y_train_val, test_size=0.05, random_state=42,
      stratify=y_train_val)
  # convert the lists of features and labels to numpy arrays
  X_train = np.array(X_train)
  X_val = np.array(X_val)
  X_test = np.array(X_test)
  y_train = np.array(y_train)
  y_val = np.array(y_val)
  y_test = np.array(y_test)
  return X_train,X_val,X_test,y_train,y_val,y_test
```

The function train_test_split() returns random non-repeatable indices from the given dataset according to the given ratio

For the first split: The function performs the first split, dividing the data and labels into a 70% training/validation set (**X_train_val**, **y_train_val**) and a 30% testing set (**X_test**, **y_test**). The **test_size** parameter is set to 0.3 to specify the desired ratio. The **stratify** parameter is used to ensure that the split maintains the proportional distribution of the encoded emotions in the original dataset.

For the second split: The second split further splits the training/validation set (**X_train_val**, **y_train_val**) into a 95% training set (**X_train**, **y_train**) and a 5% validation set (**X_val**, **y_val**), again using stratified sampling.

After the splits, the function converts the lists of features and labels to numpy arrays. Finally, it prints the shapes of the resulting datasets.

6. Data Augmentation

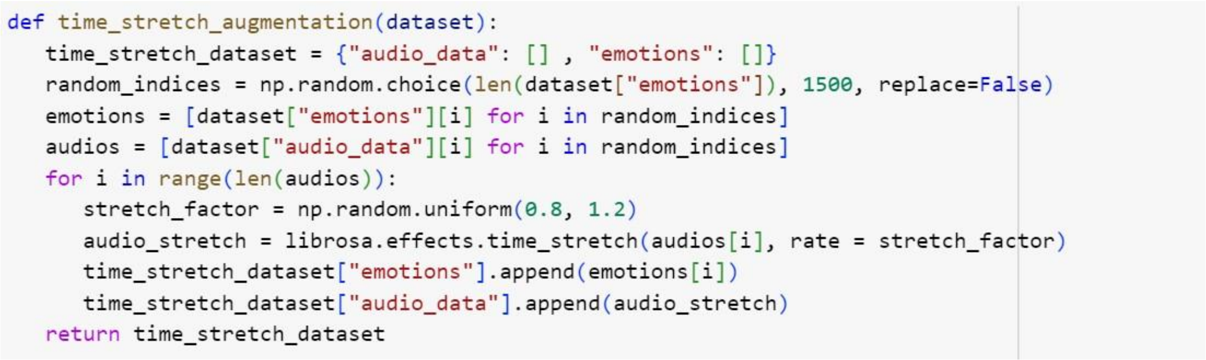
6.1. What is data augmentation

Data augmentation refers to techniques that are used to add slightly edited versions of existing data or create synthetic data by using existing data, thereby increasing the actual amount of data available.

6.2. Benefits of data augmentation

- It increases the model's ability to generalize.
- It adds variability to the data and minimizes data overfitting.
- It saves on the cost of collecting and labeling additional data.
- It improves the accuracy of the deep learning model's predictions. **6.3.**

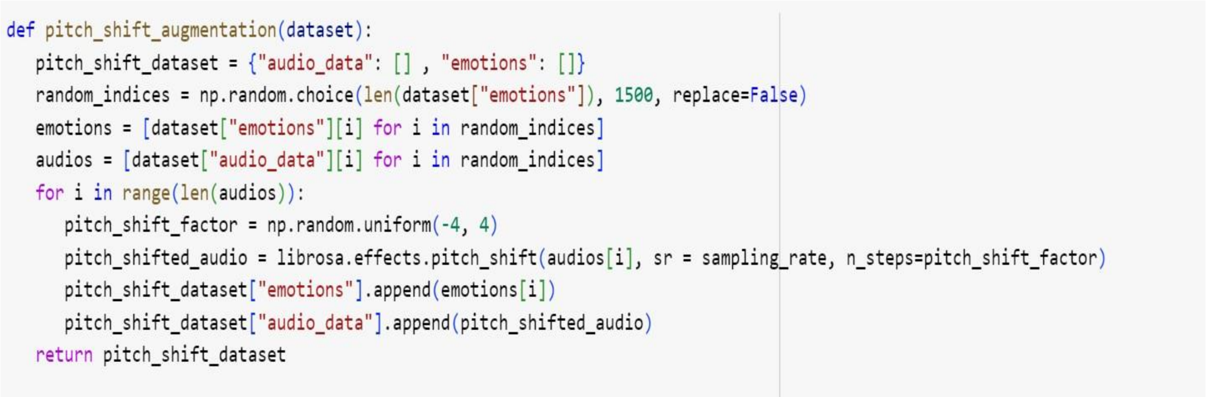
Time Stretch



```
def time_stretch_augmentation(dataset):
    time_stretch_dataset = {"audio_data": [] , "emotions": []}
    random_indices = np.random.choice(len(dataset["emotions"]), 1500, replace=False)
    emotions = [dataset["emotions"][i] for i in random_indices]
    audios = [dataset["audio_data"][i] for i in random_indices]
    for i in range(len(audios)):
        stretch_factor = np.random.uniform(0.8, 1.2)
        audio_stretch = librosa.effects.time_stretch(audios[i], rate = stretch_factor)
        time_stretch_dataset["emotions"].append(emotions[i])
        time_stretch_dataset["audio_data"].append(audio_stretch)
    return time_stretch_dataset
```

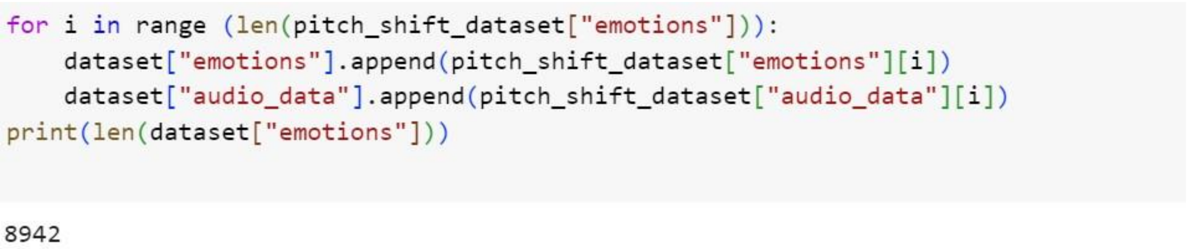
The **time_stretch_augmentation** function applies time stretching augmentation to a dataset by randomly selecting audio samples, generating random time stretch factors, and stretching the audio accordingly. This process creates an augmented dataset with time-stretched audio samples and their corresponding emotions

6.4. Phase Shift



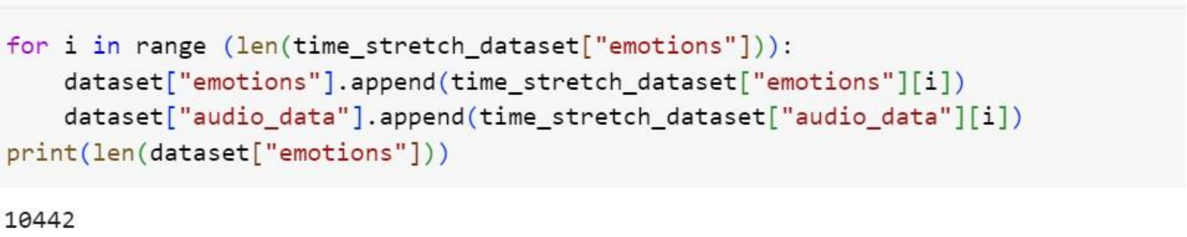
```
def pitch_shift_augmentation(dataset):
   pitch_shift_dataset = {"audio_data": [] , "emotions": []}
   random_indices = np.random.choice(len(dataset["emotions"]), 1500, replace=False)
   emotions = [dataset["emotions"][i] for i in random_indices]
   audios = [dataset["audio_data"][i] for i in random_indices]
   for i in range(len(audios)):
      pitch_shift_factor = np.random.uniform(-4, 4)
      pitch_shifted_audio = librosa.effects.pitch_shift(audios[i], sr = sampling_rate, n_steps=pitch_shift_factor)
      pitch_shift_dataset["emotions"].append(emotions[i])
      pitch_shift_dataset["audio_data"].append(pitch_shifted_audio)
   return pitch_shift_dataset
```

The **pitch_shift_augmentation** function applies pitch shifting augmentation to a dataset by randomly selecting audio samples, generating random pitch shift factors, and shifting the pitch of the audio accordingly. This process creates an augmented dataset with pitch-shifted audio samples and their corresponding emotions.



```
for i in range (len(pitch_shift_dataset["emotions"])):
    dataset["emotions"].append(pitch_shift_dataset["emotions"][i])
    dataset["audio_data"].append(pitch_shift_dataset["audio_data"][i])
print(len(dataset["emotions"]))
```

```
8942
```



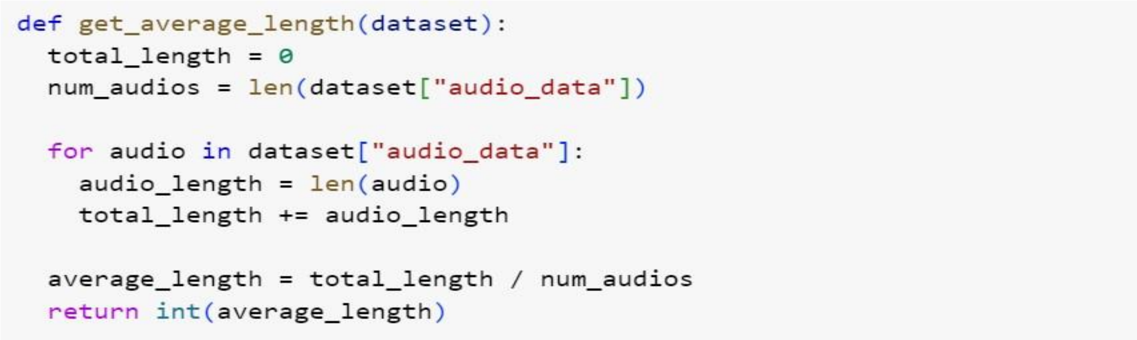
```
for i in range (len(time_stretch_dataset["emotions"])):
    dataset["emotions"].append(time_stretch_dataset["emotions"][i])
    dataset["audio_data"].append(time_stretch_dataset["audio_data"][i])
print(len(dataset["emotions"]))
```

```
10442
```

Concatenate the augmented datasets to our dataset.

7. Fix Length of audios

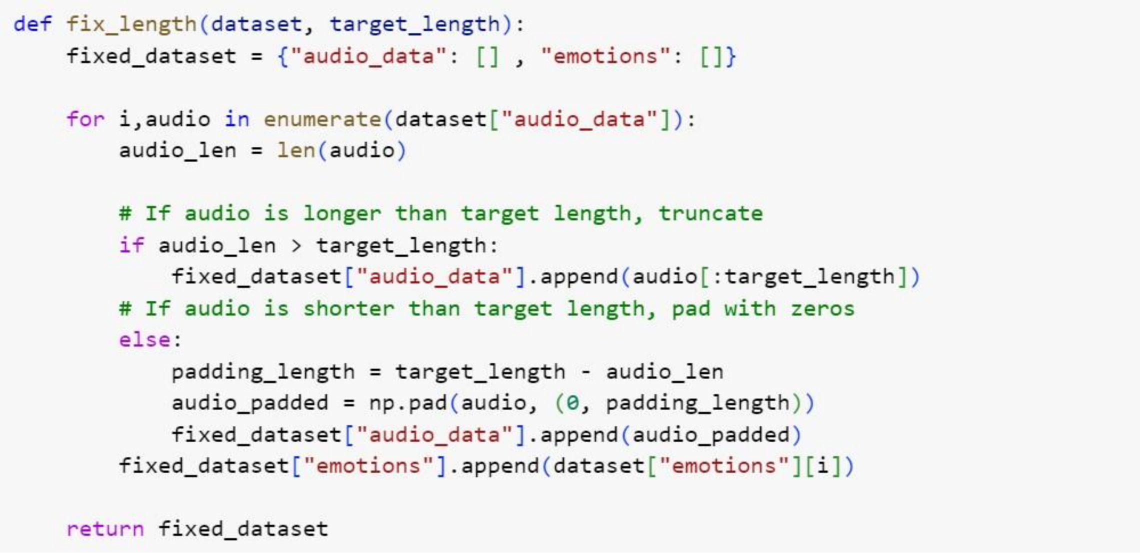
7.1. Get target length



```
def get_average_length(dataset):
  total_length = 0
  num_audios = len(dataset["audio_data"])

  for audio in dataset["audio_data"]:
    audio_length = len(audio)
    total_length += audio_length

  average_length = total_length / num_audios
  return int(average_length)
```

The **get_average_length** function calculates the average length of audio samples in a dataset.

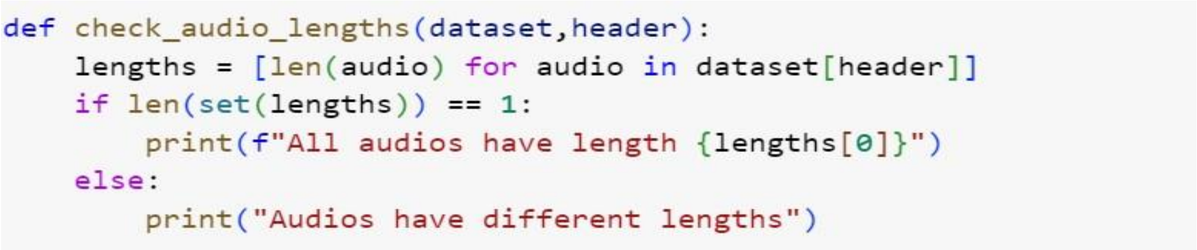
7.2. Fix Length



```python
def fix_length(dataset, target_length):
    fixed_dataset = {"audio_data": [] , "emotions": []}

    for i,audio in enumerate(dataset["audio_data"]):
        audio_len = len(audio)

        # If audio is longer than target length, truncate
        if audio_len > target_length:
            fixed_dataset["audio_data"].append(audio[:target_length])
        # If audio is shorter than target length, pad with zeros
        else:
            padding_length = target_length - audio_len
            audio_padded = np.pad(audio, (0, padding_length))
            fixed_dataset["audio_data"].append(audio_padded)
        fixed_dataset["emotions"].append(dataset["emotions"][i])

    return fixed_dataset
```

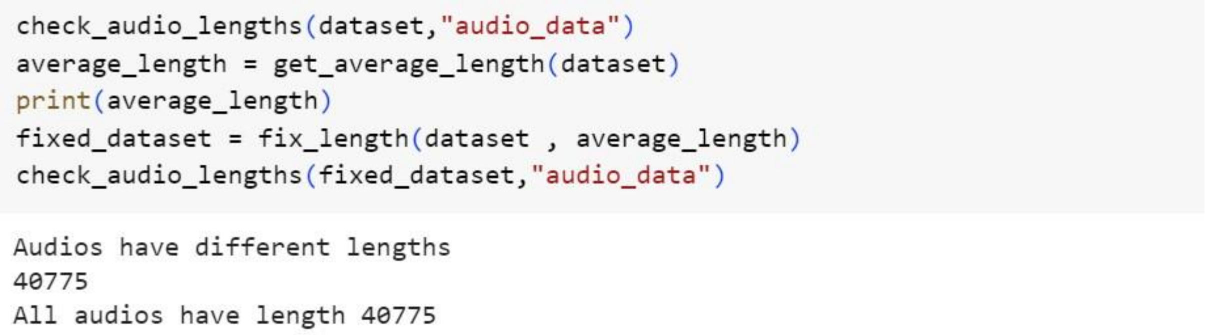
the **fix_length** function adjusts the length of audio samples in a dataset to a specified target length by either truncating or padding the audio. These functions are useful for ensuring that all audio samples in a dataset have a consistent length, which can be beneficial for certain machine learning models that require fixed-length inputs.

7.3. Check audio length



```python
def check_audio_lengths(dataset,header):
    lengths = [len(audio) for audio in dataset[header]]
    if len(set(lengths)) == 1:
        print(f"All audios have length {lengths[0]}")
    else:
        print("Audios have different lengths")
```

This function checks the length of all audios and check if they are all the same length or not, this function is to ensure that fix length function works efficiently.

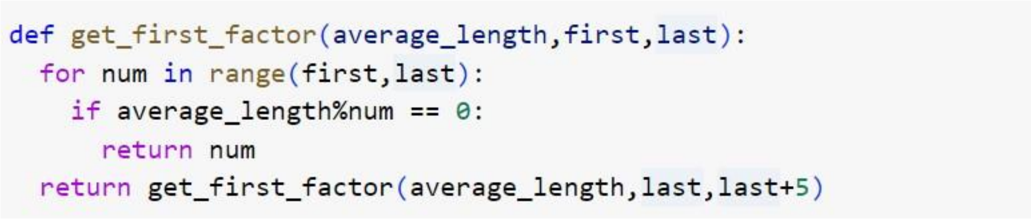


```python
check_audio_lengths(dataset,"audio_data")
average_length = get_average_length(dataset)
print(average_length)
fixed_dataset = fix_length(dataset , average_length)
check_audio_lengths(fixed_dataset,"audio_data")
```

```
Audios have different lengths
40775
All audios have length 40775
```

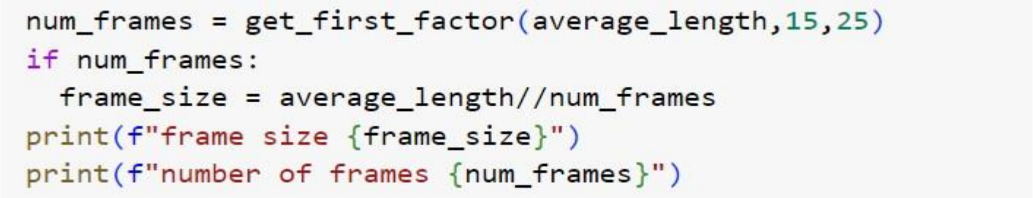
8. Create the feature space

8.1. Get first factor



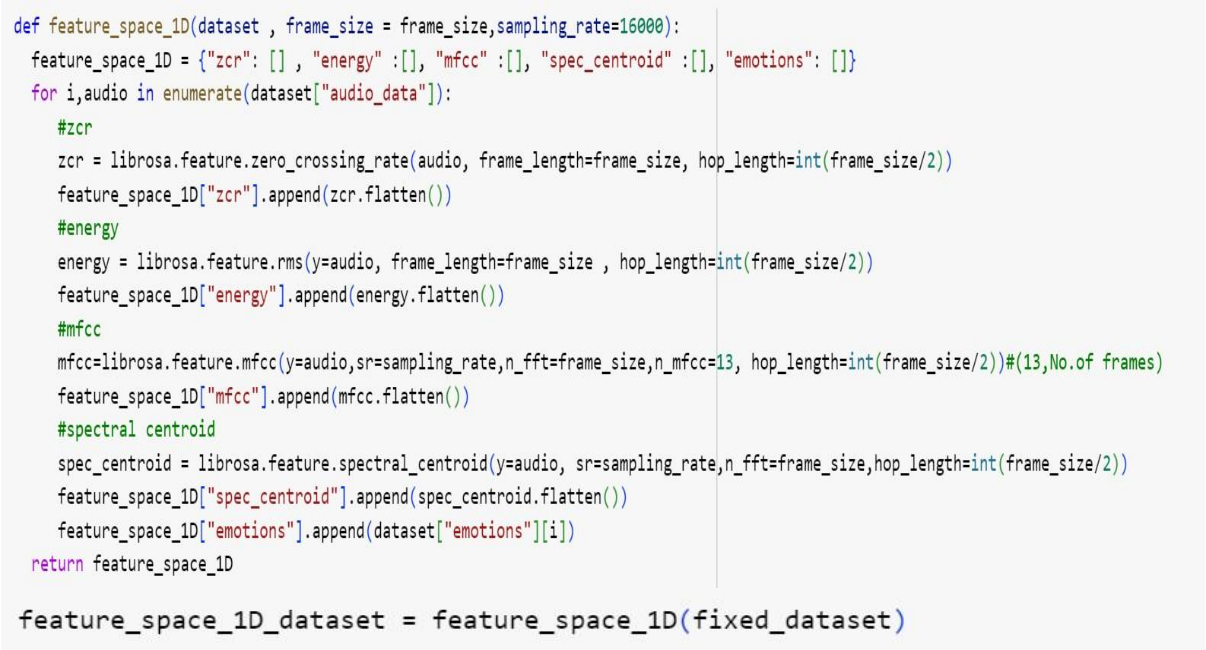
```python
def get_first_factor(average_length,first,last):
  for num in range(first,last):
    if average_length%num == 0:
      return num
  return get_first_factor(average_length,last,last+5)
```

This function is to get the first factor (within a specific range) of the audio length so it could be divided to frames.



```python
num_frames = get_first_factor(average_length,15,25)
if num_frames:
  frame_size = average_length//num_frames
print(f"frame size {frame_size}")
print(f"number of frames {num_frames}")
```

8.2. 1D Feature Space



```python
def feature_space_1D(dataset , frame_size = frame_size,sampling_rate=16000):
  feature_space_1D = {"zcr": [] , "energy" :[], "mfcc" :[], "spec_centroid" :[], "emotions": []}
  for i,audio in enumerate(dataset["audio_data"]):
    #zcr
    zcr = librosa.feature.zero_crossing_rate(audio, frame_length=frame_size, hop_length=int(frame_size/2))
    feature_space_1D["zcr"].append(zcr.flatten())
    #energy
    energy = librosa.feature.rms(y=audio, frame_length=frame_size , hop_length=int(frame_size/2))
    feature_space_1D["energy"].append(energy.flatten())
    #mfcc
    mfcc=librosa.feature.mfcc(y=audio,sr=sampling_rate,n_fft=frame_size,n_mfcc=13, hop_length=int(frame_size/2))#(13,No.of frames)
    feature_space_1D["mfcc"].append(mfcc.flatten())
    #spectral centroid
    spec_centroid = librosa.feature.spectral_centroid(y=audio, sr=sampling_rate,n_fft=frame_size,hop_length=int(frame_size/2))
    feature_space_1D["spec_centroid"].append(spec_centroid.flatten())
    feature_space_1D["emotions"].append(dataset["emotions"][i])
  return feature_space_1D

feature_space_1D_dataset = feature_space_1D(fixed_dataset)
```

The function initializes a new dataset called feature_space_1D, which will store the extracted audio features. Iterate Through Audio Samples: The function iterates over each audio sample in the input dataset.

Extract Features:

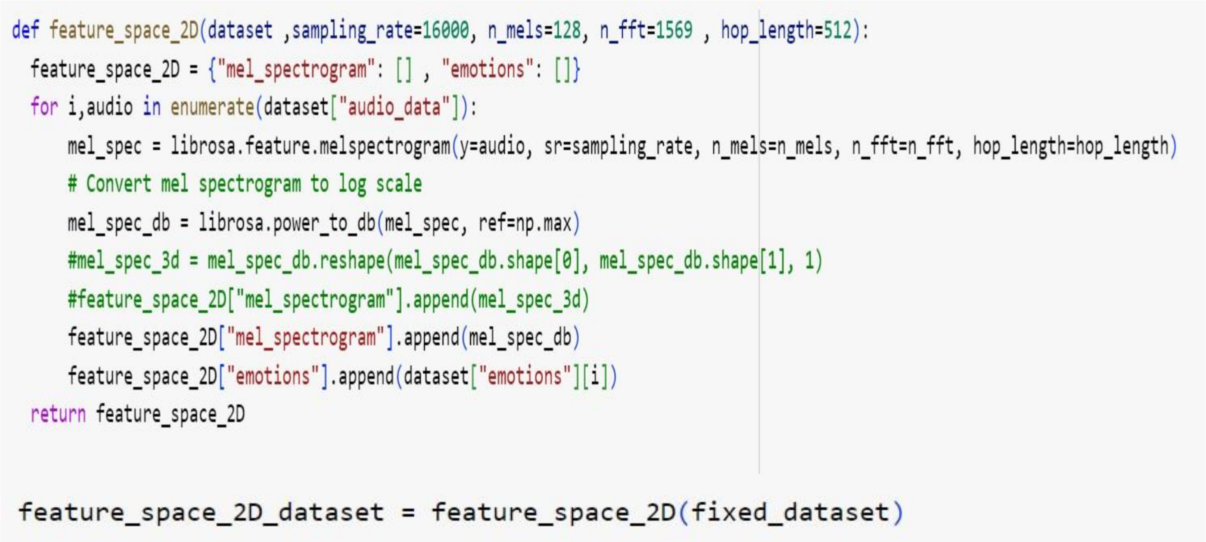
- Zero Crossing Rate (ZCR): The function computes the zero crossing rate using librosa.feature.zero_crossing_rate. It specifies the frame size and hop length to calculate the ZCR for each frame of the audio. The resulting ZCR values are flattened and added to the feature_space_1D.

- Energy: The function computes the root mean square (RMS) energy using librosa.feature.rms. It specifies the frame size and hop length to calculate the energy for each frame of the audio. The resulting energy values are flattened and added to the feature_space_1D.
- MFCC: The function computes the Mel-frequency cepstral coefficients (MFCC) using librosa.feature.mfcc. It specifies the frame size, sampling rate, number of MFCC coefficients, and hop length to calculate the MFCC for each frame of the audio. The resulting MFCC values are flattened and added to the feature_space_1D.
- Spectral Centroid: The function computes the spectral centroid using librosa.feature.spectral_centroid. It specifies the frame size, sampling rate, and hop length to calculate the spectral centroid for each frame of the audio. The resulting spectral centroid values are flattened and added to the feature_space_1D.

The corresponding emotion for each audio sample is added to the feature_space_1D.
After extracting features for all audio samples, the function returns the feature_space_1D dataset containing the extracted features and their corresponding emotions.

In summary, the feature_space_1D function computes and flattens multiple audio features (ZCR, energy, MFCC, spectral centroid) for each audio sample in a dataset and stores them along with the corresponding emotions in a new dataset. This function is useful for preparing audio data for machine learning tasks that require feature-based representations of audio samples.

8.3. 2D Feature Space



```
def feature_space_2D(dataset ,sampling_rate=16000, n_mels=128, n_fft=1569 , hop_length=512):
  feature_space_2D = {"mel_spectrogram": [] , "emotions": []}
  for i,audio in enumerate(dataset["audio_data"]):
      mel_spec = librosa.feature.melspectrogram(y=audio, sr=sampling_rate, n_mels=n_mels, n_fft=n_fft, hop_length=hop_length)
      # Convert mel spectrogram to log scale
      mel_spec_db = librosa.power_to_db(mel_spec, ref=np.max)
      #mel_spec_3d = mel_spec_db.reshape(mel_spec_db.shape[0], mel_spec_db.shape[1], 1)
      #feature_space_2D["mel_spectrogram"].append(mel_spec_3d)
      feature_space_2D["mel_spectrogram"].append(mel_spec_db)
      feature_space_2D["emotions"].append(dataset["emotions"][i])
  return feature_space_2D

feature_space_2D_dataset = feature_space_2D(fixed_dataset)
```

The function initializes a new dataset called feature_space_2D, which will store the extracted Mel spectrogram feature. Iterate Through Audio Samples: The function iterates over each audio sample in the input dataset.

Extract Mel Spectrogram:

- Mel Spectrogram: The function computes the Mel spectrogram using librosa.feature.melspectrogram. It takes the audio sample, sampling rate, number of Mel filters (n_mels), size of the FFT window (n_fft), and hop length (hop_length) as parameters. The resulting Mel spectrogram is a 2D array.

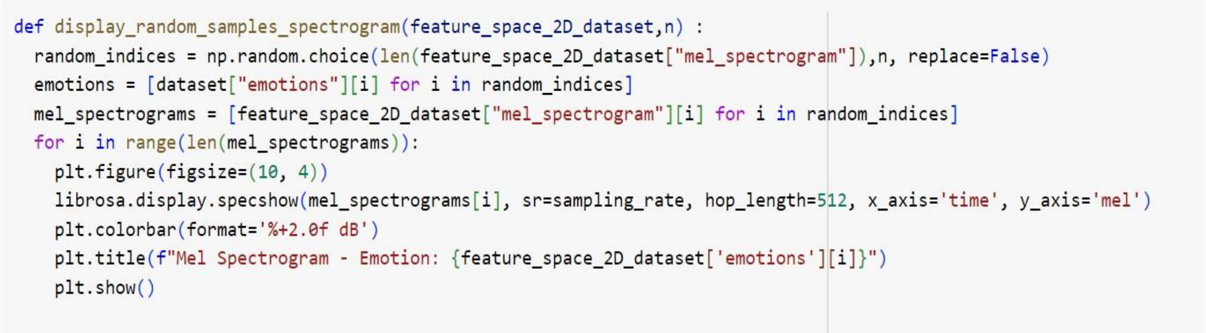
Convert to Log Scale:

- Log Scale Conversion: The Mel spectrogram is converted to a log scale using librosa.power_to_db. This transformation enhances the visualization and emphasizes smaller values.

The corresponding emotion for each audio sample is added to the feature_space_2D. After extracting the Mel spectrogram and emotions for all audio samples, the function returns the feature_space_2D dataset containing the Mel spectrogram and their corresponding emotions.

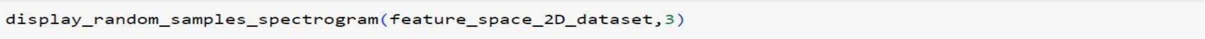
In summary, the feature_space_2D function computes the Mel spectrogram feature for each audio sample in a dataset, converts it to a log scale, and stores it along with the corresponding emotions in a new dataset. This function is useful for tasks that require 2D representations of audio, such as deep learning models that process spectrogram-like inputs.

7.4. Display mel_spectogram

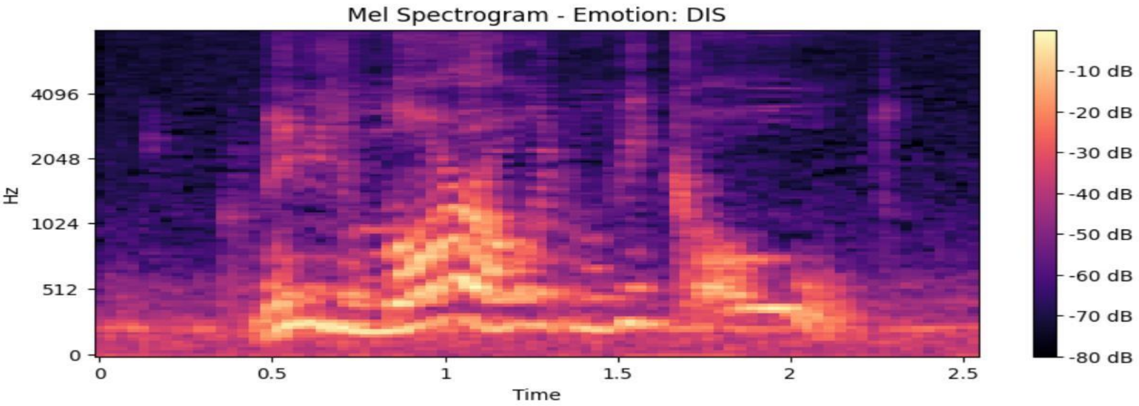


```
def display_random_samples_spectrogram(feature_space_2D_dataset,n) :
  random_indices = np.random.choice(len(feature_space_2D_dataset["mel_spectrogram"]),n, replace=False)
  emotions = [dataset["emotions"][i] for i in random_indices]
  mel_spectrograms = [feature_space_2D_dataset["mel_spectrogram"][i] for i in random_indices]
  for i in range(len(mel_spectrograms)):
    plt.figure(figsize=(10, 4))
    librosa.display.specshow(mel_spectrograms[i], sr=sampling_rate, hop_length=512, x_axis='time', y_axis='mel')
    plt.colorbar(format='%+2.0f dB')
    plt.title(f"Mel Spectrogram - Emotion: {feature_space_2D_dataset['emotions'][i]}")
    plt.show()
```

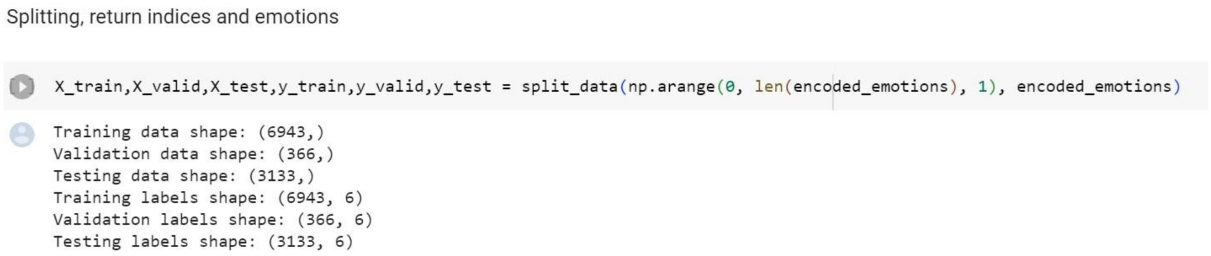
The **display_random_samples_spectrogram** function displays a random selection of Mel spectrograms from a dataset. It chooses n random samples from the dataset and retrieves the corresponding Mel spectrograms and emotions. Each Mel spectrogram is then visualized using **librosa.display.specshow()**, showing the spectrogram with color coding. The function provides a quick way to visually inspect the Mel spectrograms and their associated emotions.



```
display_random_samples_spectrogram(feature_space_2D_dataset,3)
```



8.5. Create the feature space for specific indices in the dataset

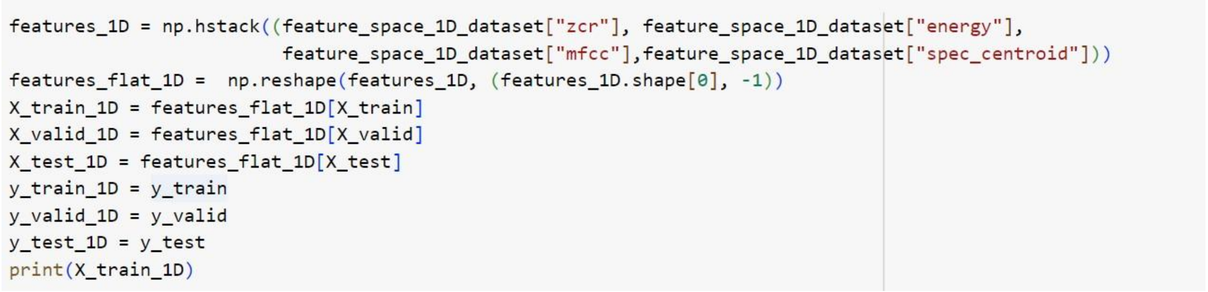


Splitting, return indices and emotions

```
X_train,X_valid,X_test,y_train,y_valid,y_test = split_data(np.arange(0, len(encoded_emotions), 1), encoded_emotions)
```

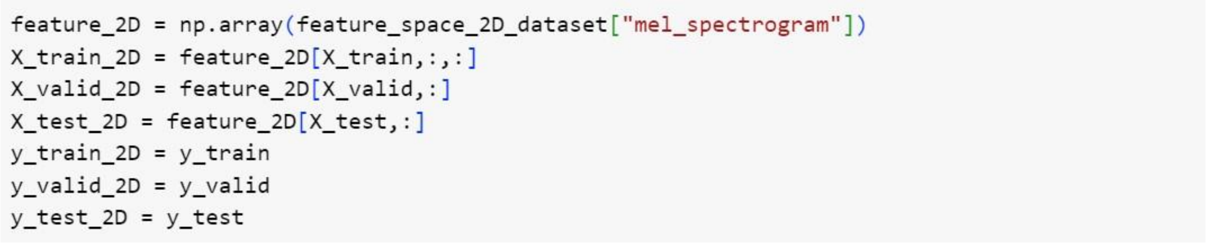
```
Training data shape: (6943,)
Validation data shape: (366,)
Testing data shape: (3133,)
Training labels shape: (6943, 6)
Validation labels shape: (366, 6)
Testing labels shape: (3133, 6)
```

Get the indices of the training, test, and validations data and their corresponding emotions.



```
features_1D = np.hstack((feature_space_1D_dataset["zcr"], feature_space_1D_dataset["energy"],
                         feature_space_1D_dataset["mfcc"],feature_space_1D_dataset["spec_centroid"]))
features_flat_1D =  np.reshape(features_1D, (features_1D.shape[0], -1))
X_train_1D = features_flat_1D[X_train]
X_valid_1D = features_flat_1D[X_valid]
X_test_1D = features_flat_1D[X_test]
y_train_1D = y_train
y_valid_1D = y_valid
y_test_1D = y_test
print(X_train_1D)
```

The code horizontally stacks the 1D features extracted from the audio dataset using **np.hstack()**. The features include zero crossing rate (**zcr**), energy, MFCC, and spectral centroid. This creates a single array called **features_1D**, and then reshapes the **features_1D** array using **np.reshape()** to have a 2D shape. The first dimension corresponds to the number of samples, while the second dimension is automatically determined based on the size of the feature array. Finally, The code assigns the flattened 1D features to training, validation, and testing sets based on the indices provided by **X_train**, **X_valid**, and **X_test**. These indices determine the samples used for each set and assigns the corresponding labels (**y_train**, **y_valid**, **y_test**) to the training, validation, and testing sets.



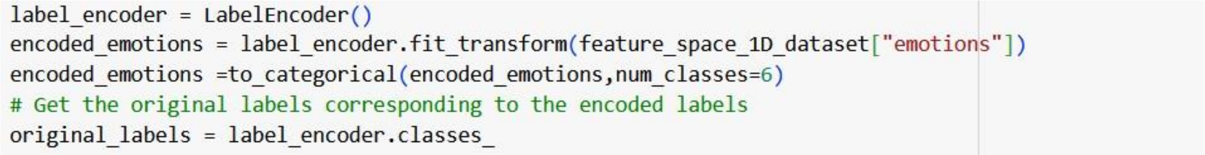
```
feature_2D = np.array(feature_space_2D_dataset["mel_spectrogram"])
X_train_2D = feature_2D[X_train,:,:]
X_valid_2D = feature_2D[X_valid,:]
X_test_2D = feature_2D[X_test,:]
y_train_2D = y_train
y_valid_2D = y_valid
y_test_2D = y_test
```

The code creates a single array called **feature_2D**. Then, it assigns the 2D feature to training, validation, and testing sets based on the indices provided by **X_train**, **X_valid**, and **X_test**. These indices determine the samples used for each set and assigns the corresponding labels (**y_train**, **y_valid**, **y_test**) to the training, validation, and testing sets.

9.Encoding

One-hot encoding represents each emotion as a binary vector where only one element is "hot" (1) indicating the class, and the rest are "cold" (0) indicating the absence of that class. This is necessary for the softmax activation function to work effectively.

The softmax function takes an input vector and normalizes it into a probability distribution over the classes. It ensures that the predicted class probabilities sum up to 1. By using the one-hot encoded vectors, the softmax function can easily determine the most probable class by selecting the element with the highest value (probability).



```
label_encoder = LabelEncoder()
encoded_emotions = label_encoder.fit_transform(feature_space_1D_dataset["emotions"])
encoded_emotions =to_categorical(encoded_emotions,num_classes=6)
# Get the original labels corresponding to the encoded labels
original_labels = label_encoder.classes_
```

10.Model

10.1.What is CNN

CNN stands for Convolutional Neural Network. It is a type of deep learning model that is widely used for image recognition, computer vision tasks, and other tasks involving grid-like data such as audio and time series data.

CNNs are particularly effective in capturing spatial relationships and patterns in data. They consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Here's a brief overview of these layers:

1. Convolutional Layers: These layers perform convolution operations on the input data. Convolution involves sliding a small window (filter/kernel) over the input data and performing element-wise multiplication and summation to extract features. The filters learn to detect different patterns or features in the data, such as edges, textures, or shapes.
2. Pooling Layers: Pooling layers are used to reduce the spatial dimensions of the data. They help in reducing the number of parameters and computational complexity while retaining important features. Max pooling is a common type of pooling used, where the maximum value within each pooling region is selected.
3. Fully Connected Layers: After the convolutional and pooling layers, the resulting features are flattened into a vector and passed through fully connected layers. These layers are similar to the layers in traditional neural networks, where each neuron is connected to every neuron in the previous and subsequent layers. Fully connected layers are responsible for making predictions based on the learned features.

10.2. 1D CNN

10.2.1.Why is 1D CNN suitable

A 1D CNN can indeed be suitable for this Crema dataset due to several reasons:

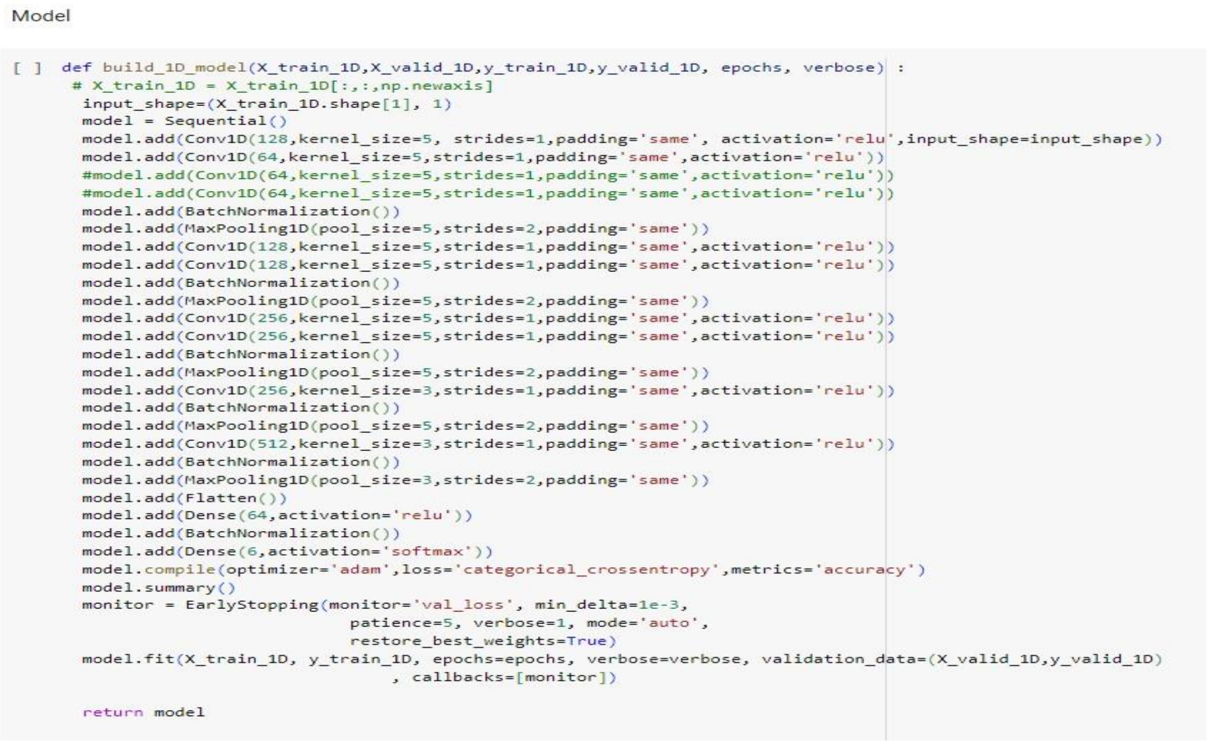
Local Pattern Detection: Similar to image-based tasks, 1D CNNs are adept at capturing local patterns and dependencies in sequential data. In the case of audio features, a 1D CNN can learn to detect relevant patterns or combinations of features that contribute to different emotions. For example, variations in zero crossing rate, MFCC coefficients, or energy levels may carry useful information about the underlying emotional content.

Hierarchical Feature Extraction: Just like in image classification, a 1D CNN can learn hierarchical representations from the input audio features. By stacking convolutional layers with increasing receptive fields, the network can extract low-level features at earlier layers (e.g., detecting shortterm patterns) and higher-level features at deeper layers (e.g., capturing longer-term dependencies). This hierarchical feature extraction enables the model to understand complex patterns and relationships in the audio data.

Translation Invariance: While 1D CNNs do not exhibit translation invariance in the same manner as 2D CNNs for images, they can still be effective at capturing local patterns that are position-invariant within the audio sequence. This property allows the model to identify meaningful features regardless of their precise location within the audio.

Parameter Sharing and Efficiency: 1D CNNs also leverage parameter sharing, which reduces the number of parameters and makes the model more efficient. By sharing weights across different regions of the input audio, the network can detect similar patterns or features at various positions. This parameter efficiency is particularly useful when working with large-scale datasets like CREMA-D.

10.2.2.Our implemented 1D Model



Model

```
[ ]  def build_1D_model(X_train_1D,X_valid_1D,y_train_1D,y_valid_1D, epochs, verbose) :
      # X_train_1D = X_train_1D[:,:,np.newaxis]
       input_shape=(X_train_1D.shape[1], 1)
       model = Sequential()
       model.add(Conv1D(128,kernel_size=5, strides=1,padding='same', activation='relu',input_shape=input_shape))
       model.add(Conv1D(64,kernel_size=5,strides=1,padding='same',activation='relu'))
       #model.add(Conv1D(64,kernel_size=5,strides=1,padding='same',activation='relu'))
       #model.add(Conv1D(64,kernel_size=5,strides=1,padding='same',activation='relu'))
       model.add(BatchNormalization())
       model.add(MaxPooling1D(pool_size=5,strides=2,padding='same'))
       model.add(Conv1D(128,kernel_size=5,strides=1,padding='same',activation='relu'))
       model.add(Conv1D(128,kernel_size=5,strides=1,padding='same',activation='relu'))
       model.add(BatchNormalization())
       model.add(MaxPooling1D(pool_size=5,strides=2,padding='same'))
       model.add(Conv1D(256,kernel_size=5,strides=1,padding='same',activation='relu'))
       model.add(Conv1D(256,kernel_size=5,strides=1,padding='same',activation='relu'))
       model.add(BatchNormalization())
       model.add(MaxPooling1D(pool_size=5,strides=2,padding='same'))
       model.add(Conv1D(256,kernel_size=3,strides=1,padding='same',activation='relu'))
       model.add(BatchNormalization())
       model.add(MaxPooling1D(pool_size=5,strides=2,padding='same'))
       model.add(Conv1D(512,kernel_size=3,strides=1,padding='same',activation='relu'))
       model.add(BatchNormalization())
       model.add(MaxPooling1D(pool_size=3,strides=2,padding='same'))
       model.add(Flatten())
       model.add(Dense(64,activation='relu'))
       model.add(BatchNormalization())
       model.add(Dense(6,activation='softmax'))
       model.compile(optimizer='adam',loss='categorical_crossentropy',metrics='accuracy')
       model.summary()
       monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                             patience=5, verbose=1, mode='auto',
                             restore_best_weights=True)
       model.fit(X_train_1D, y_train_1D, epochs=epochs, verbose=verbose, validation_data=(X_valid_1D,y_valid_1D)
                                  , callbacks=[monitor])

       return model
```

10.2.3.Explanation:

1.Convolutional Layers:

- The model begins with two sets of consecutive convolutional layers, each followed by batch normalization and ReLU activation.

- The first Conv1D layer has 128 filters with a kernel size of 5, and the second Conv1D layer has 64 filters with a kernel size of 5.
- These layers are responsible for extracting local features from the input sequence.

2.Max Pooling Layers:

- After each set of convolutional layers, a max pooling layer is added.
- The max pooling layer reduces the dimensionality of the output, capturing the most important features while discarding irrelevant information.
- The pooling layers have a pool size of 5, a stride of 2, and padding set to 'same', ensuring that the spatial dimensions are preserved.

3.Additional Convolutional Layers:

- The model adds another set of Conv1D layers, each followed by batch normalization and ReLU activation.
- These layers further extract features from the input sequence.
- The first Conv1D layer has 128 filters with a kernel size of 5, and the second Conv1D layer has 128 filters with a kernel size of 5.

4.Additional Max Pooling Layers:

- Similar to earlier layers, max pooling layers are inserted after the additional Conv1D layers.
- These pooling layers have the same configuration as the previous ones.

5.Further Convolutional Layers:

- The model includes one more Conv1D layer with 256 filters and a kernel size of 3, followed by batch normalization.
- Another max pooling layer follows this Conv1D layer.

6.Dense Layers:

- The output of the final max pooling layer is flattened to a 1D vector using the Flatten layer.
- The flattened output is then fed into a dense layer with 64 units and ReLU activation, followed by batch normalization.
- Finally, a dense layer with 6 units (equal to the number of classes) and softmax activation is added to produce the final class probabilities.

7.Model Compilation:

- The model is compiled with the Adam optimizer, categorical cross-entropy loss (since it's a multiclass classification problem), and accuracy as the evaluation metric.

8.Model Training:

- The fit method is used to train the model on the provided training data (X_train_1D and y_train_1D).
- The training is performed for a specified number of epochs, with verbose output during training.
- The validation_data argument is used to monitor the performance on the validation data (X_valid_1D and y_valid_1D).

- The EarlyStopping callback is applied to monitor the validation loss and stop training if it doesn't improve after a certain number of epochs.

10.2.4.Hyperparameters

Different hyperparameters were tested, until the ones that give highest results were chosen

1.kernel_size: The size of the convolutional kernel/filter. In our model, convolutional layers use kernel sizes of 5 and 3.

2.strides: The stride length used in convolutional and pooling operations. In our model, the stride length is set to 1 for the convolutional layers and 2 for the pooling layers.

3.padding: The padding strategy applied in convolutional and pooling operations. In our model, padding is set to 'same', which pads the input sequence with zeros to preserve the spatial dimensions after the convolution or pooling.

4.pool_size: The size of the pooling window. The max pooling layers in our model use a pool size of 5.

5.activation: The activation function applied to the output of the convolutional and dense layers. Our model uses the ReLU activation function for most convolutional layers and the softmax activation function for the final dense layer.

6.optimizer: The optimization algorithm used during training. The model uses the Adam optimizer, which is a popular choice for gradient-based optimization.

7.loss: The loss function used to measure the difference between the predicted and true labels. The model uses categorical cross-entropy loss since it's a multiclass classification problem.

8.metrics: The evaluation metric used to monitor the model's performance during training. The model uses accuracy as the evaluation metric.

9.epochs: The number of epochs, or complete passes through the training data, during training. This is a hyperparameter that needs to be set based on the specific dataset and model convergence.

10.3. 2D CNN

10.3.1.Why is 2D CNN suitable

A 2D CNN is suitable for multiclass classification on the CREMA-D dataset using spectrograms because of the following reasons:

1. Spatial Information: Spectrograms are 2D representations of audio signals that capture both frequency and time information. A 2D CNN is well-suited for processing spectrograms as it can effectively leverage the spatial structure present in the data. The CNN's convolutional layers can detect local patterns and features in different regions of the spectrogram, allowing the model to learn relevant representations for emotion classification.

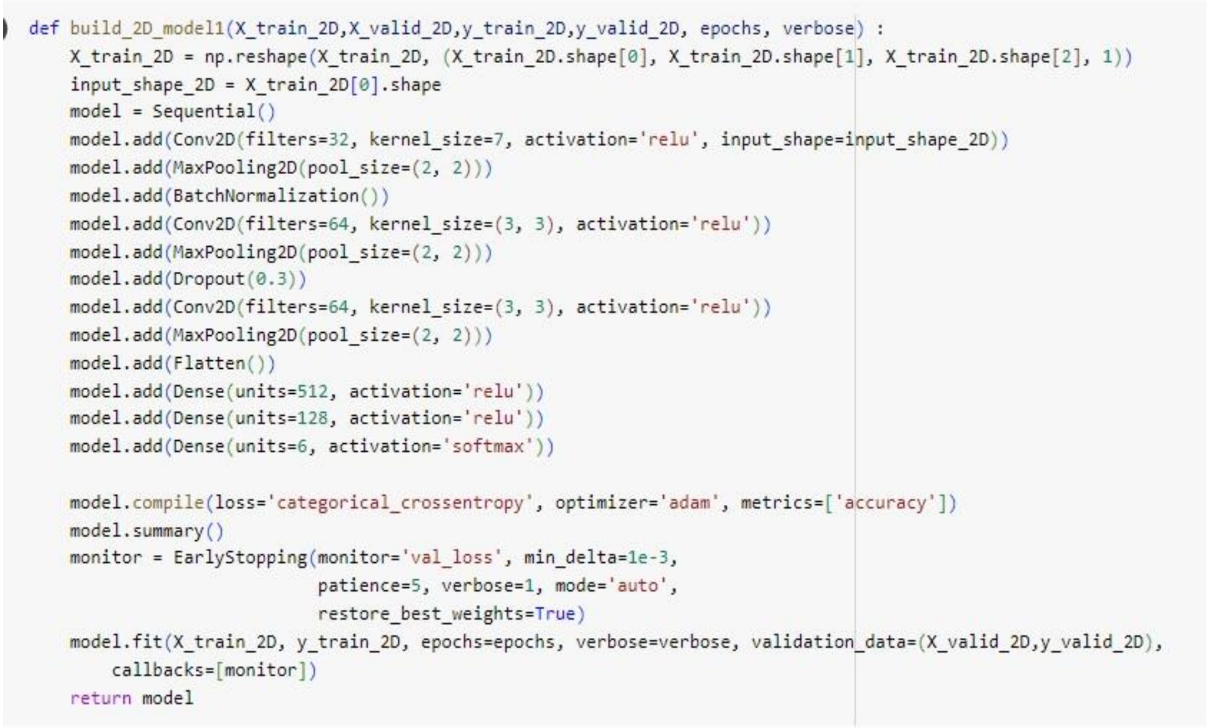
2. Hierarchical Feature Extraction: Similar to images, spectrograms contain hierarchical structures. A 2D CNN can learn to extract low-level features, such as edges or frequency patterns, at earlier convolutional layers, and then progressively learn higher-level features, such as combinations of these patterns, at deeper layers. This hierarchical feature extraction enables the model to capture complex relationships and representations in the spectrogram data.

3. Translation Invariance: CNNs are known for their ability to capture spatial invariance, meaning they can identify patterns regardless of their exact location within the input data. In the case of spectrograms, the position of different features or patterns might vary within the time-frequency domain. A 2D CNN can learn to be invariant to these translations, allowing it to effectively recognize important features regardless of their precise location in the spectrogram.

4. Parameter Sharing and Efficiency: CNNs leverage parameter sharing, meaning that the same set of weights is applied across different spatial locations in the input data. This sharing of parameters makes the model more efficient, as it significantly reduces the number of parameters that need to be learned. This is particularly useful when working with large spectrograms, where the number of parameters can quickly become overwhelming.

5. State-of-the-Art Performance: CNNs have been widely successful in various image-based tasks, such as image classification and object detection. They have demonstrated state-ofthe-art performance on many benchmarks. Applying CNNs to spectrogram-based audio classification tasks, including emotion classification, has shown promising results and can potentially achieve high accuracy in identifying different emotions from the spectrogram representations.

10.3.2.Our implemented 2D Model



```python
def build_2D_model1(X_train_2D,X_valid_2D,y_train_2D,y_valid_2D, epochs, verbose) :
    X_train_2D = np.reshape(X_train_2D, (X_train_2D.shape[0], X_train_2D.shape[1], X_train_2D.shape[2], 1))
    input_shape_2D = X_train_2D[0].shape
    model = Sequential()
    model.add(Conv2D(filters=32, kernel_size=7, activation='relu', input_shape=input_shape_2D))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(BatchNormalization())
    model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.3))
    model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(units=512, activation='relu'))
    model.add(Dense(units=128, activation='relu'))
    model.add(Dense(units=6, activation='softmax'))

    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    model.summary()
    monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                            patience=5, verbose=1, mode='auto',
                            restore_best_weights=True)
    model.fit(X_train_2D, y_train_2D, epochs=epochs, verbose=verbose, validation_data=(X_valid_2D,y_valid_2D),
        callbacks=[monitor])
    return model
```

10.3.3.Explanation

1.Convolutional layers: The model starts with a Conv2D layer with 32 filters, a kernel size of 7x7, and a ReLU activation function. This layer is responsible for extracting features from the input spectrogram. Max pooling with a pool size of 2x2 is applied after this convolutional layer to reduce the spatial dimensions of the output.

2.Batch normalization: After each convolutional layer, a BatchNormalization() layer is added. Batch normalization helps in normalizing the activations of the previous layer, making the training process more stable and efficient.

3.Dropout: To prevent overfitting, a Dropout() layer with a rate of 0.3 is added after the second convolutional layer. Dropout randomly sets a fraction of input units to 0 during training, which helps in reducing over-reliance on specific features and promotes better generalization.

4.Flattening: The output from the last convolutional layer is flattened using the Flatten() layer. This converts the 2D feature maps into a 1D vector, preparing the data for the fully connected layers.

5.Fully connected layers: Two dense (fully connected) layers are added. The first dense layer has 512 units with a ReLU activation function, followed by a second dense layer with 128 units and ReLU activation. These layers are responsible for learning high-level representations and capturing complex relationships in the data.

6.Output layer: The final dense layer has 6 units, corresponding to the number of classes in the classification task. The softmax activation function is used in the output layer to produce probability distributions over the classes, indicating the predicted probability of each class.

7.Compilation: The model is compiled with the categorical cross-entropy loss function, the Adam optimizer, and the accuracy metric.

8.Training: The model is trained using the fit() function. The training data (X_train_2D and y_train_2D) is used for training, and the validation data (X_valid_2D and y_valid_2D) is used for validation during the training process. The number of epochs and verbosity level are specified as inputs.

9.Early stopping: The training process includes an early stopping callback (EarlyStopping) that monitors the validation loss. If the validation loss does not improve for a certain number of epochs (patience), the training is stopped early, and the model weights are restored to the best performing weights.

10.3.4.Hyperparameters

Different hyperparameters were tested, until the ones that give highest results were chosen

1.Filters: The number of filters determines the depth or the number of feature maps produced by a convolutional layer. More filters allow the model to learn more diverse and complex features from the input data. In the code, the first Conv2D layer uses 32 filters, and the second Conv2D layer uses 64 filters.

2.Kernel size: The kernel size specifies the dimensions of the sliding window used for convolution. It determines the receptive field or the region of the input that each filter considers. In the code, the first Conv2D layer uses a kernel size of 7x7, and the second Conv2D layer uses a kernel size of 3x3. Larger kernel sizes capture more spatial information, but they also increase the computational complexity. Smaller kernel sizes can capture more local patterns.

3.Pooling: Pooling layers downsample the feature maps, reducing the spatial dimensions. The MaxPooling2D layers in the code use a pool size of 2x2, meaning the feature maps are downsampled by a factor of 2 in each dimension. Pooling helps in reducing the spatial resolution while retaining the most relevant features.

4.Dropout: Dropout is a regularization technique that helps prevent overfitting. It randomly sets a fraction of input units to 0 during training, which reduces the reliance on specific features and encourages the model to learn more robust and generalized representations. In the code, a Dropout layer is added after the second Conv2D layer with a dropout rate of 0.3. This means that 30% of the input units will be randomly set to 0 during training.

5.Dense layers: Dense layers, also known as fully connected layers, are responsible for learning highlevel representations and making predictions. The first dense layer in the code has 512 units, followed by a second dense layer with 128 units. The number of units determines the dimensionality of the learned representations. More units can provide a higher capacity to capture complex relationships in the data, but they also increase the number of parameters and computational complexity.

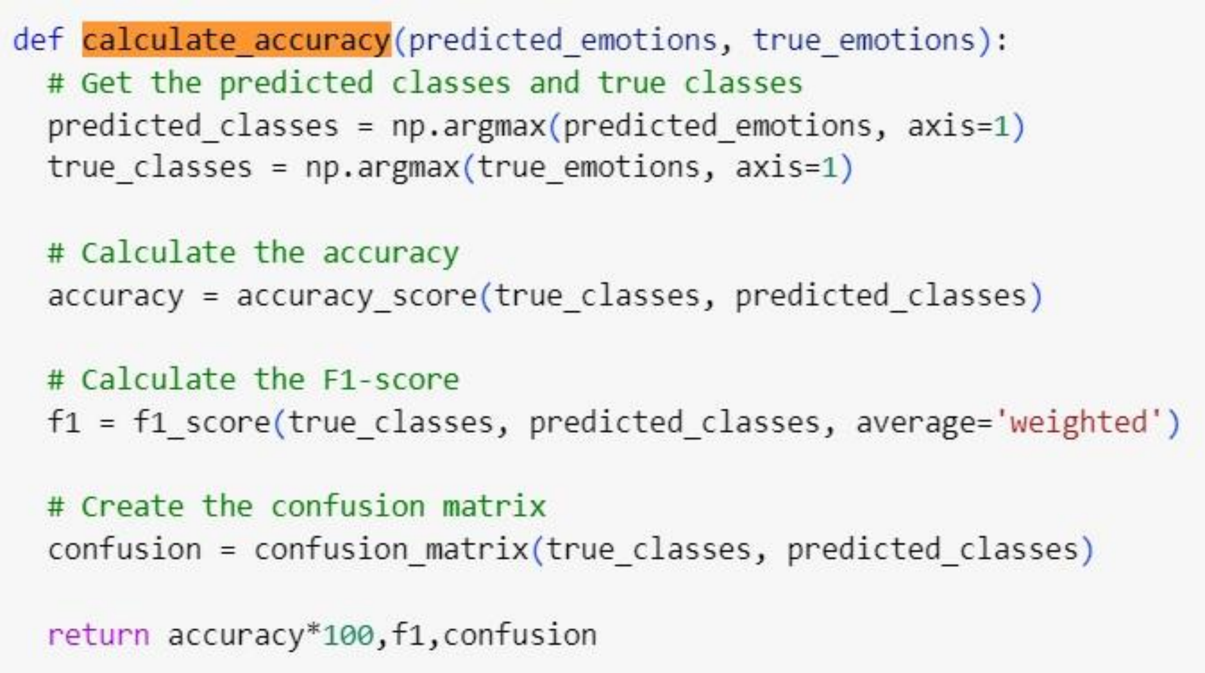
6.Activation functions: Activation functions introduce non-linearity to the model and help it learn complex patterns. The code uses the ReLU (Rectified Linear Unit) activation function for the convolutional and dense layers. ReLU is widely used in CNN models due to its ability to alleviate the vanishing gradient problem and promote faster convergence. The output layer uses the softmax activation function, which produces a probability distribution over the classes, allowing the model to make predictions for multiclass classification.

7.Loss function: The loss function measures the model's performance and guides the learning process. In the code, the categorical cross-entropy loss function is used, which is commonly used for multiclass classification tasks. It compares the predicted probabilities to the true labels and calculates the cross-entropy loss.

8.Optimizer: The optimizer determines how the model's weights are updated during training to minimize the loss function. The code uses the Adam optimizer, which is an adaptive learning rate optimization algorithm. Adam combines the benefits of both AdaGrad and RMSprop by adapting the learning rate based on the estimated first and second moments of the gradients. Adam is known for its efficiency and effectiveness in training deep neural networks.

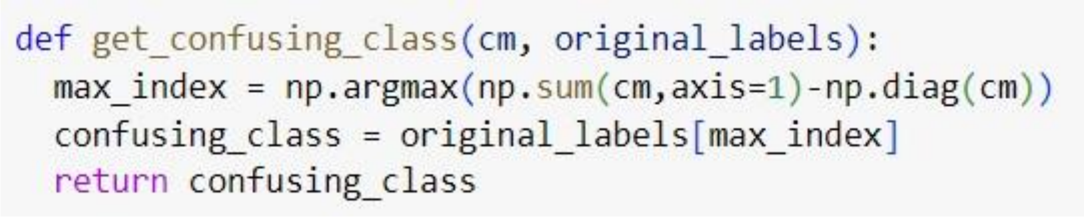
11.Measures of accuracy

11.1 accuracy and f1_score



```
def calculate_accuracy(predicted_emotions, true_emotions):
  # Get the predicted classes and true classes
  predicted_classes = np.argmax(predicted_emotions, axis=1)
  true_classes = np.argmax(true_emotions, axis=1)

  # Calculate the accuracy
  accuracy = accuracy_score(true_classes, predicted_classes)

  # Calculate the F1-score
  f1 = f1_score(true_classes, predicted_classes, average='weighted')

  # Create the confusion matrix
  confusion = confusion_matrix(true_classes, predicted_classes)

  return accuracy*100,f1,confusion
```

11.2 Get most confusing class



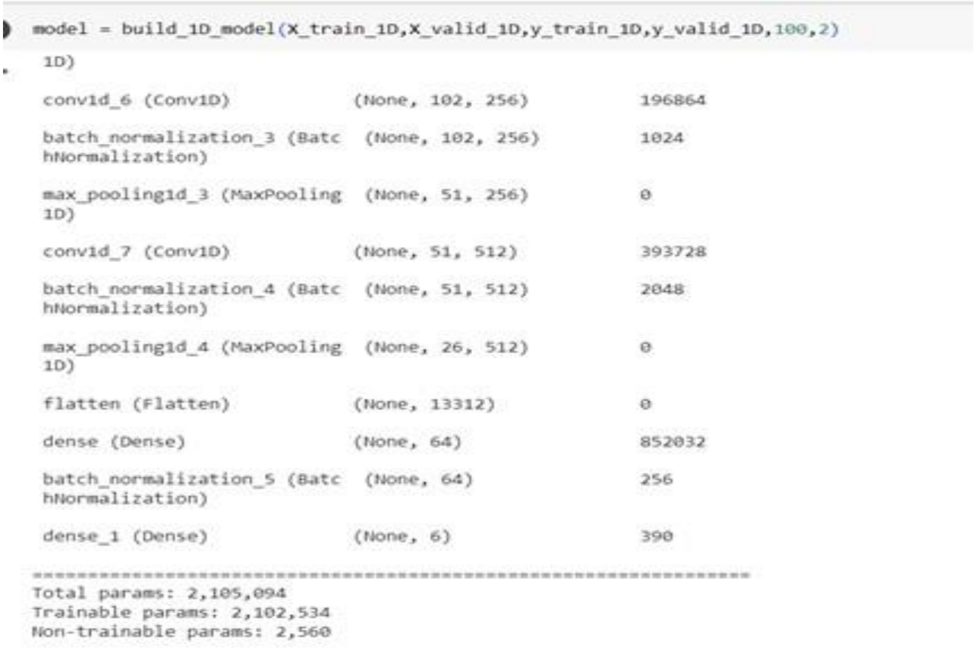
```
def get_confusing_class(cm, original_labels):
  max_index = np.argmax(np.sum(cm,axis=1)-np.diag(cm))
  confusing_class = original_labels[max_index]
  return confusing_class
```

The function sum each row of the confusion matrix except diagonal elements and get the index of the maximum element. It returns the most confusing class among the original labels.

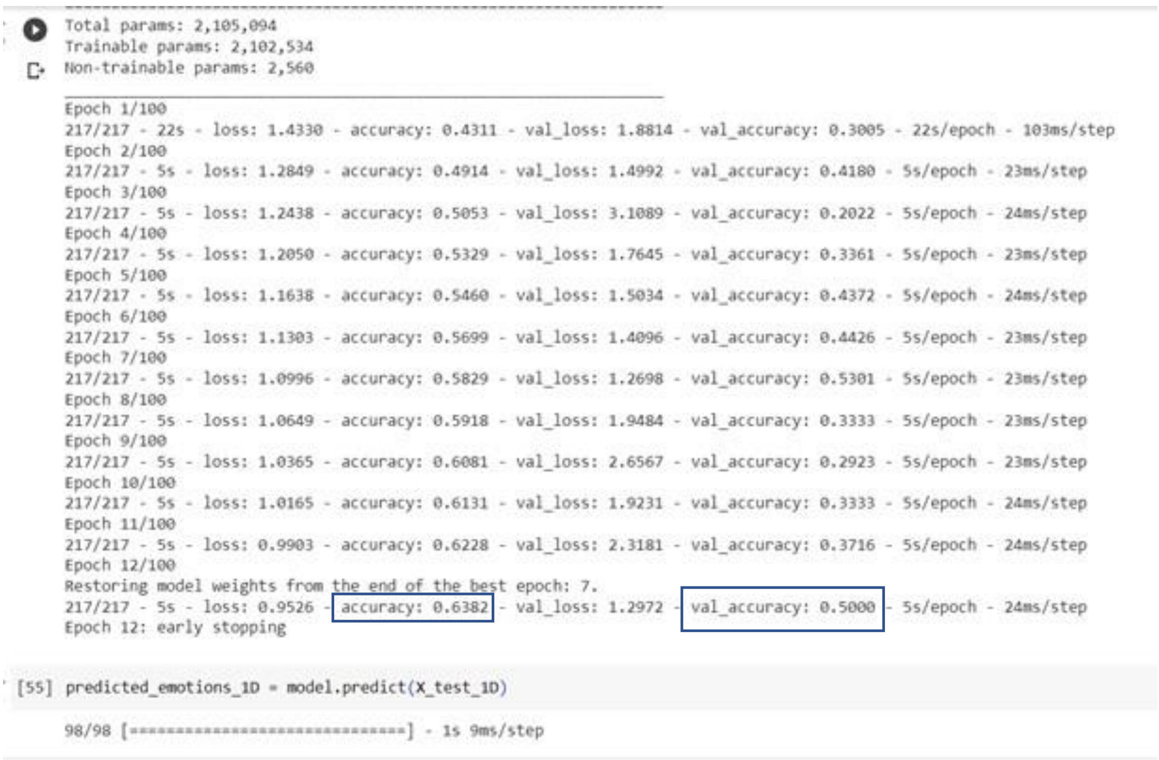
12.Runs

12.1 1D model

a)

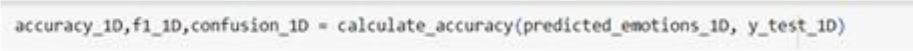


```
model = build_1D_model(X_train_1D,X_valid_1D,y_train_1D,y_valid_1D,100,2)

 1D)

 conv1d_6 (Conv1D)           (None, 102, 256)          196864

 batch_normalization_3 (Batc  (None, 102, 256)         1024
 hNormalization)

 max_pooling1d_3 (MaxPooling  (None, 51, 256)          0
 1D)

 conv1d_7 (Conv1D)           (None, 51, 512)           393728

 batch_normalization_4 (Batc  (None, 51, 512)          2048
 hNormalization)

 max_pooling1d_4 (MaxPooling  (None, 26, 512)          0
 1D)

 flatten (Flatten)           (None, 13312)             0

 dense (Dense)               (None, 64)                852032

 batch_normalization_5 (Batc  (None, 64)               256
 hNormalization)

 dense_1 (Dense)             (None, 6)                 390

=================================================================
Total params: 2,105,094
Trainable params: 2,102,534
Non-trainable params: 2,560
```

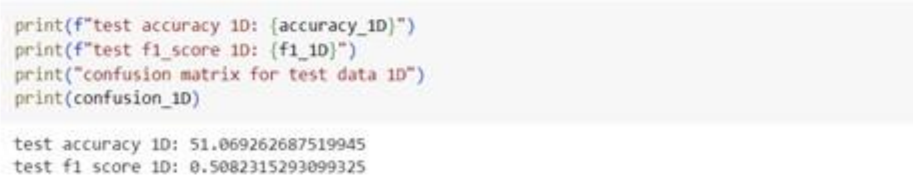


```
_________________________________________________________________
Total params: 2,105,094
Trainable params: 2,102,534
Non-trainable params: 2,560
_________________________________________________________________
Epoch 1/100
217/217 - 22s - loss: 1.4330 - accuracy: 0.4311 - val_loss: 1.8814 - val_accuracy: 0.3005 - 22s/epoch - 103ms/step
Epoch 2/100
217/217 - 5s - loss: 1.2849 - accuracy: 0.4914 - val_loss: 1.4992 - val_accuracy: 0.4180 - 5s/epoch - 23ms/step
Epoch 3/100
217/217 - 5s - loss: 1.2438 - accuracy: 0.5053 - val_loss: 3.1089 - val_accuracy: 0.2022 - 5s/epoch - 24ms/step
Epoch 4/100
217/217 - 5s - loss: 1.2050 - accuracy: 0.5329 - val_loss: 1.7645 - val_accuracy: 0.3361 - 5s/epoch - 23ms/step
Epoch 5/100
217/217 - 5s - loss: 1.1638 - accuracy: 0.5460 - val_loss: 1.5034 - val_accuracy: 0.4372 - 5s/epoch - 24ms/step
Epoch 6/100
217/217 - 5s - loss: 1.1303 - accuracy: 0.5699 - val_loss: 1.4096 - val_accuracy: 0.4426 - 5s/epoch - 23ms/step
Epoch 7/100
217/217 - 5s - loss: 1.0996 - accuracy: 0.5829 - val_loss: 1.2698 - val_accuracy: 0.5301 - 5s/epoch - 23ms/step
Epoch 8/100
217/217 - 5s - loss: 1.0649 - accuracy: 0.5918 - val_loss: 1.9484 - val_accuracy: 0.3333 - 5s/epoch - 23ms/step
Epoch 9/100
217/217 - 5s - loss: 1.0365 - accuracy: 0.6081 - val_loss: 2.6567 - val_accuracy: 0.2923 - 5s/epoch - 23ms/step
Epoch 10/100
217/217 - 5s - loss: 1.0165 - accuracy: 0.6131 - val_loss: 1.9231 - val_accuracy: 0.3333 - 5s/epoch - 24ms/step
Epoch 11/100
217/217 - 5s - loss: 0.9903 - accuracy: 0.6228 - val_loss: 2.3181 - val_accuracy: 0.3716 - 5s/epoch - 24ms/step
Epoch 12/100
Restoring model weights from the end of the best epoch: 7.
217/217 - 5s - loss: 0.9526 - accuracy: 0.6382 - val_loss: 1.2972 - val_accuracy: 0.5000 - 5s/epoch - 24ms/step
Epoch 12: early stopping

[55] predicted_emotions_1D = model.predict(X_test_1D)

98/98 [==============================] - 1s 9ms/step
```

Function build_1D_model takes training and validation datasets and their corresponding emotions along with specified verbose and epochs, compile model, fit the datasets provided and returns the model, which predicts the test validation, where the returned predicted emotions will be examined to determine performance of the model.

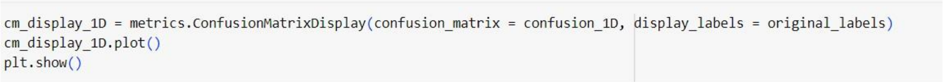


```
accuracy_1D,f1_1D,confusion_1D = calculate_accuracy(predicted_emotions_1D, y_test_1D)
```

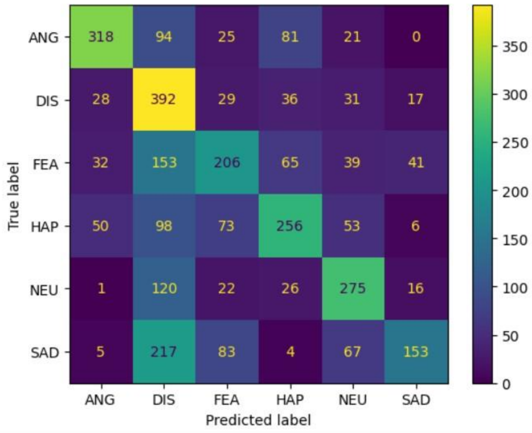


```
print(f"test accuracy 1D: {accuracy_1D}")
print(f"test f1_score 1D: {f1_1D}")
print("confusion matrix for test data 1D")
print(confusion_1D)
```

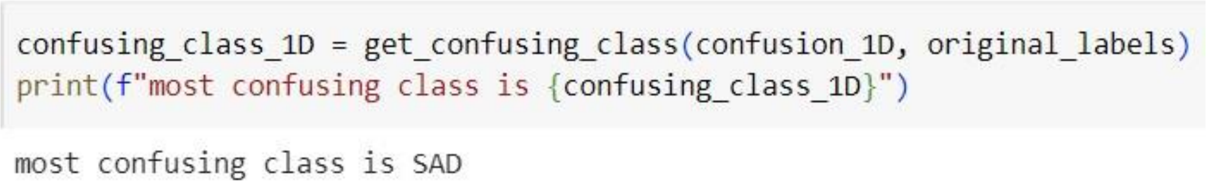
```
test accuracy 1D: 51.069262687519945
test f1 score 1D: 0.5082315293099325
```



```
cm_display_1D = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_1D, display_labels = original_labels)
cm_display_1D.plot()
plt.show()
```



True label	ANG	DIS	FEA	HAP	NEU	SAD
ANG	318	94	25	81	21	0
DIS	28	392	29	36	31	17
FEA	32	153	206	65	39	41
HAP	50	98	73	256	53	6
NEU	1	120	22	26	275	16
SAD	5	217	83	4	67	153

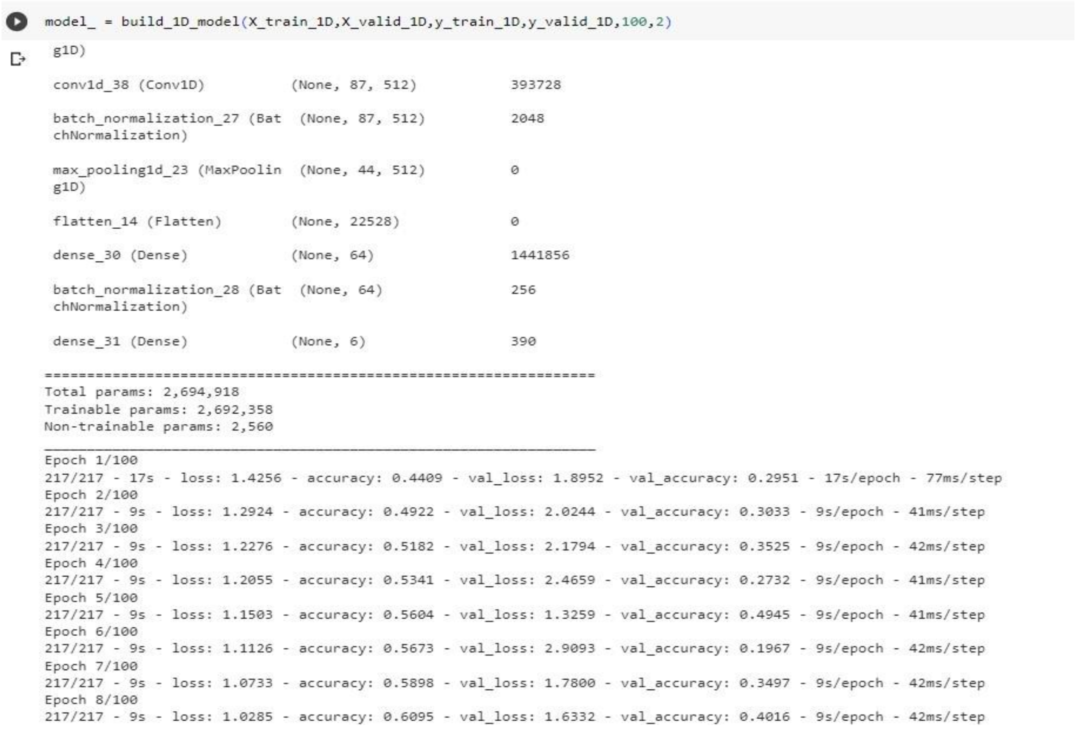


```
confusing_class_1D = get_confusing_class(confusion_1D, original_labels)
print(f"most confusing class is {confusing_class_1D}")
```

```
most confusing class is SAD
```

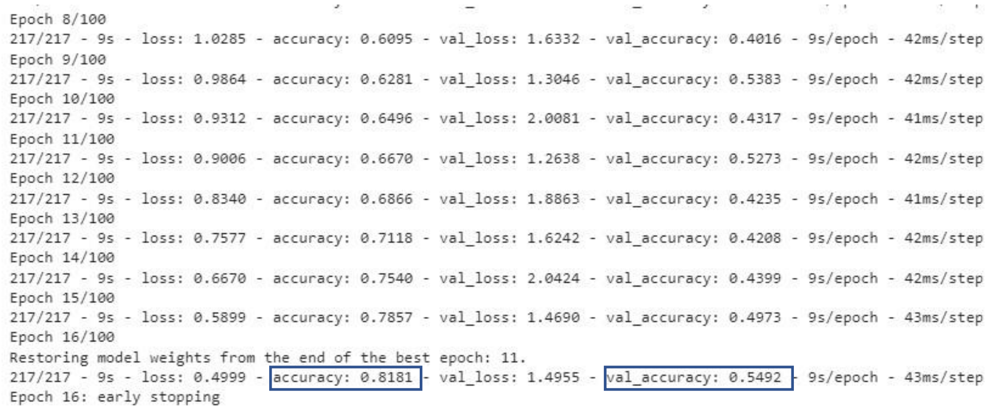
b)

Another run was conducted to have better indication on the model’s performance.

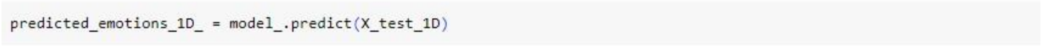


```
model_ = build_1D_model(X_train_1D,X_valid_1D,y_train_1D,y_valid_1D,100,2)
```

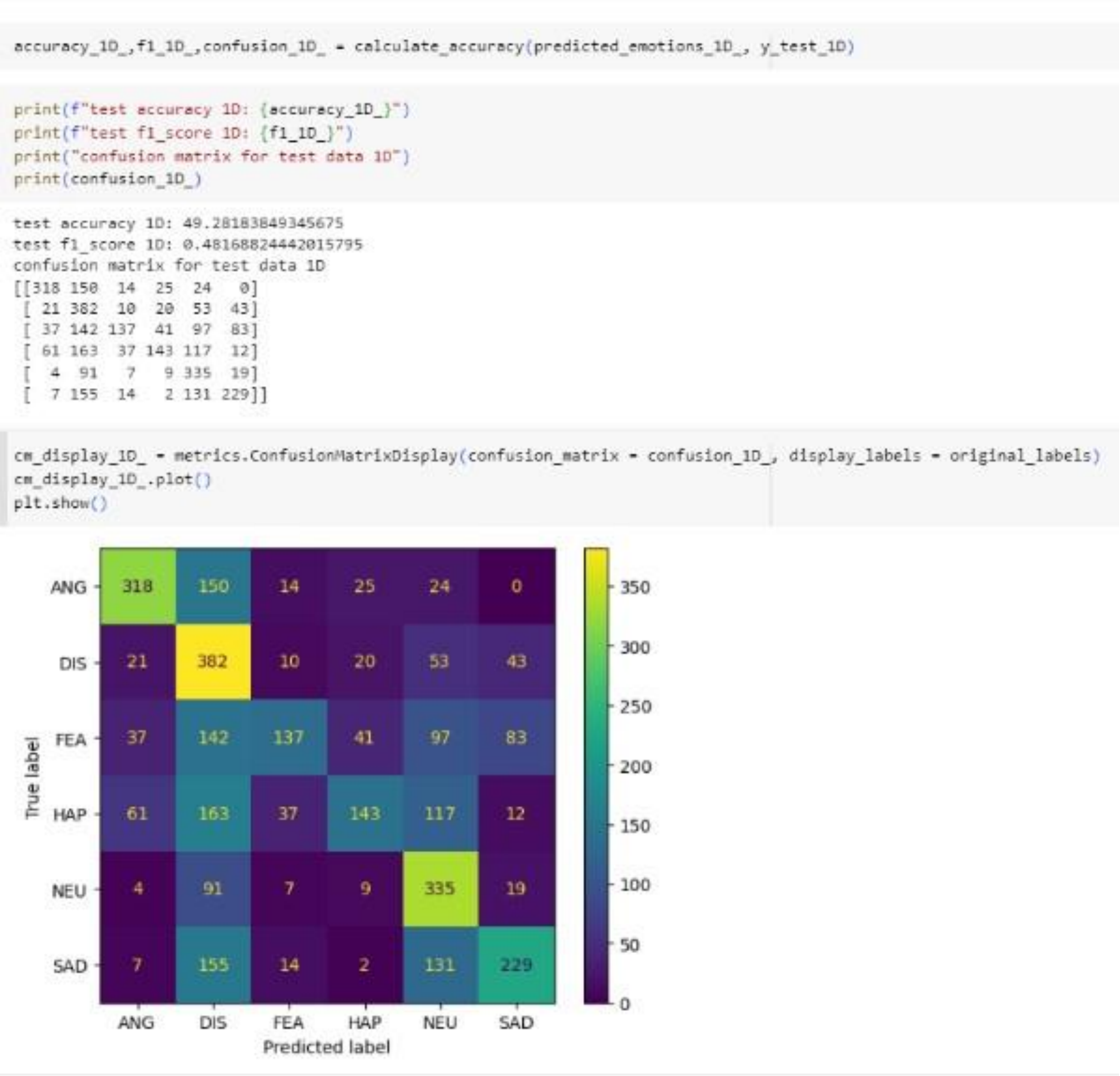
```
 g1D)

 conv1d_38 (Conv1D)          (None, 87, 512)           393728

 batch_normalization_27 (Bat  (None, 87, 512)          2048
 chNormalization)

 max_pooling1d_23 (MaxPoolin  (None, 44, 512)          0
 g1D)

 flatten_14 (Flatten)        (None, 22528)             0

 dense_30 (Dense)            (None, 64)                1441856

 batch_normalization_28 (Bat  (None, 64)               256
 chNormalization)

 dense_31 (Dense)            (None, 6)                 390

=================================================================
Total params: 2,694,918
Trainable params: 2,692,358
Non-trainable params: 2,560
_________________________________________________________________
Epoch 1/100
217/217 - 17s - loss: 1.4256 - accuracy: 0.4409 - val_loss: 1.8952 - val_accuracy: 0.2951 - 17s/epoch - 77ms/step
Epoch 2/100
217/217 - 9s - loss: 1.2924 - accuracy: 0.4922 - val_loss: 2.0244 - val_accuracy: 0.3033 - 9s/epoch - 41ms/step
Epoch 3/100
217/217 - 9s - loss: 1.2276 - accuracy: 0.5182 - val_loss: 2.1794 - val_accuracy: 0.3525 - 9s/epoch - 42ms/step
Epoch 4/100
217/217 - 9s - loss: 1.2055 - accuracy: 0.5341 - val_loss: 2.4659 - val_accuracy: 0.2732 - 9s/epoch - 41ms/step
Epoch 5/100
217/217 - 9s - loss: 1.1503 - accuracy: 0.5604 - val_loss: 1.3259 - val_accuracy: 0.4945 - 9s/epoch - 41ms/step
Epoch 6/100
217/217 - 9s - loss: 1.1126 - accuracy: 0.5673 - val_loss: 2.9093 - val_accuracy: 0.1967 - 9s/epoch - 42ms/step
Epoch 7/100
217/217 - 9s - loss: 1.0733 - accuracy: 0.5898 - val_loss: 1.7800 - val_accuracy: 0.3497 - 9s/epoch - 42ms/step
Epoch 8/100
217/217 - 9s - loss: 1.0285 - accuracy: 0.6095 - val_loss: 1.6332 - val_accuracy: 0.4016 - 9s/epoch - 42ms/step
```



```
Epoch 8/100
217/217 - 9s - loss: 1.0285 - accuracy: 0.6095 - val_loss: 1.6332 - val_accuracy: 0.4016 - 9s/epoch - 42ms/step
Epoch 9/100
217/217 - 9s - loss: 0.9864 - accuracy: 0.6281 - val_loss: 1.3046 - val_accuracy: 0.5383 - 9s/epoch - 42ms/step
Epoch 10/100
217/217 - 9s - loss: 0.9312 - accuracy: 0.6496 - val_loss: 2.0081 - val_accuracy: 0.4317 - 9s/epoch - 41ms/step
Epoch 11/100
217/217 - 9s - loss: 0.9006 - accuracy: 0.6670 - val_loss: 1.2638 - val_accuracy: 0.5273 - 9s/epoch - 42ms/step
Epoch 12/100
217/217 - 9s - loss: 0.8340 - accuracy: 0.6866 - val_loss: 1.8863 - val_accuracy: 0.4235 - 9s/epoch - 41ms/step
Epoch 13/100
217/217 - 9s - loss: 0.7577 - accuracy: 0.7118 - val_loss: 1.6242 - val_accuracy: 0.4208 - 9s/epoch - 42ms/step
Epoch 14/100
217/217 - 9s - loss: 0.6670 - accuracy: 0.7540 - val_loss: 2.0424 - val_accuracy: 0.4399 - 9s/epoch - 42ms/step
Epoch 15/100
217/217 - 9s - loss: 0.5899 - accuracy: 0.7857 - val_loss: 1.4690 - val_accuracy: 0.4973 - 9s/epoch - 43ms/step
Epoch 16/100
Restoring model weights from the end of the best epoch: 11.
217/217 - 9s - loss: 0.4999 - accuracy: 0.8181 - val_loss: 1.4955 - val_accuracy: 0.5492 - 9s/epoch - 43ms/step
Epoch 16: early stopping
```



```
predicted_emotions_1D_ = model_.predict(X_test_1D)
```

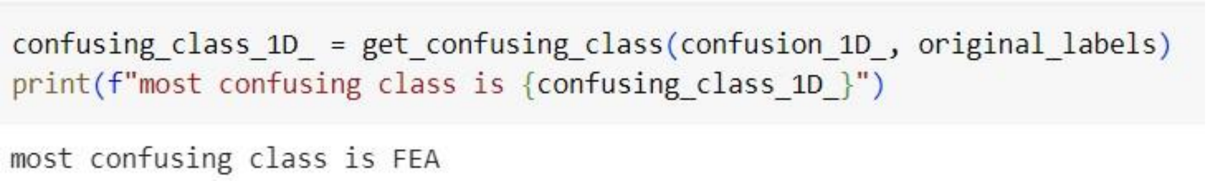


```
accuracy_1D_,f1_1D_,confusion_1D_ = calculate_accuracy(predicted_emotions_1D_, y_test_1D)
```

```
print(f"test accuracy 1D: {accuracy_1D_}")
print(f"test f1_score 1D: {f1_1D_}")
print("confusion matrix for test data 1D")
print(confusion_1D_)
```

```
test accuracy 1D: 49.28183849345675
test f1_score 1D: 0.48168824442015795
confusion matrix for test data 1D
[[318 150  14  25  24   0]
 [ 21 382  10  20  53  43]
 [ 37 142 137  41  97  83]
 [ 61 163  37 143 117  12]
 [  4  91   7   9 335  19]
 [  7 155  14   2 131 229]]
```

```
cm_display_1D_ = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_1D_, display_labels = original_labels)
cm_display_1D_.plot()
plt.show()
```

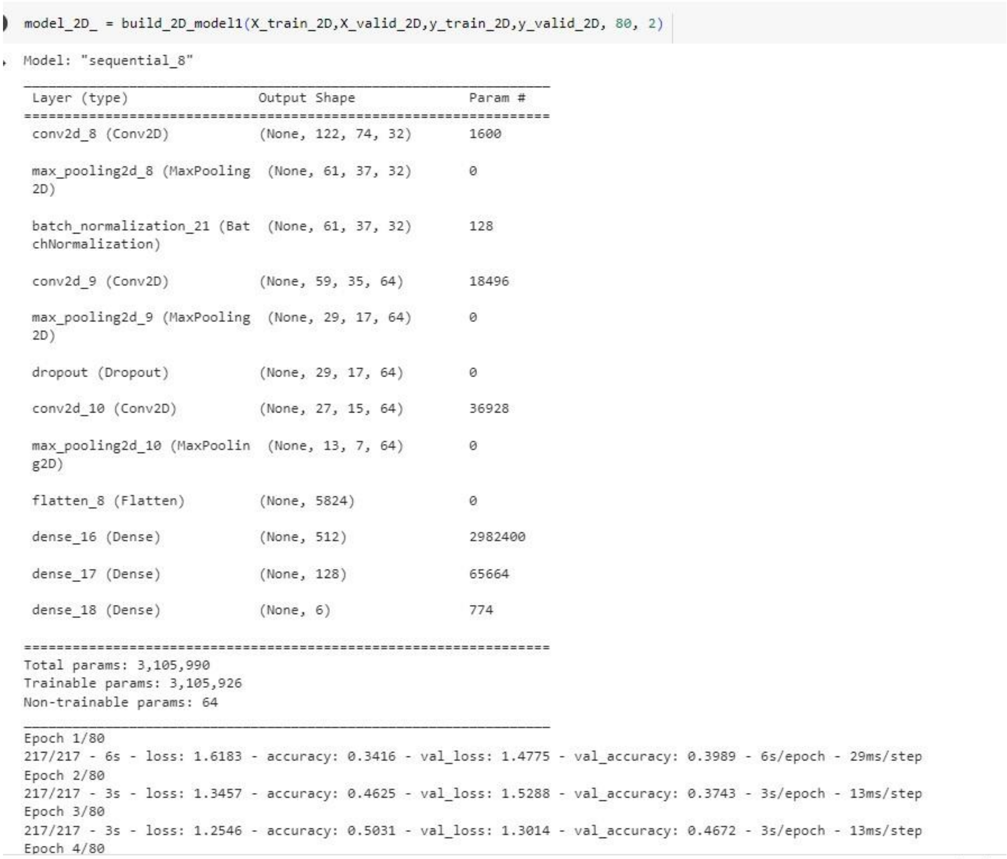


```
confusing_class_1D_ = get_confusing_class(confusion_1D_, original_labels)
print(f"most confusing class is {confusing_class_1D_}")
```

```
most confusing class is FEA
```

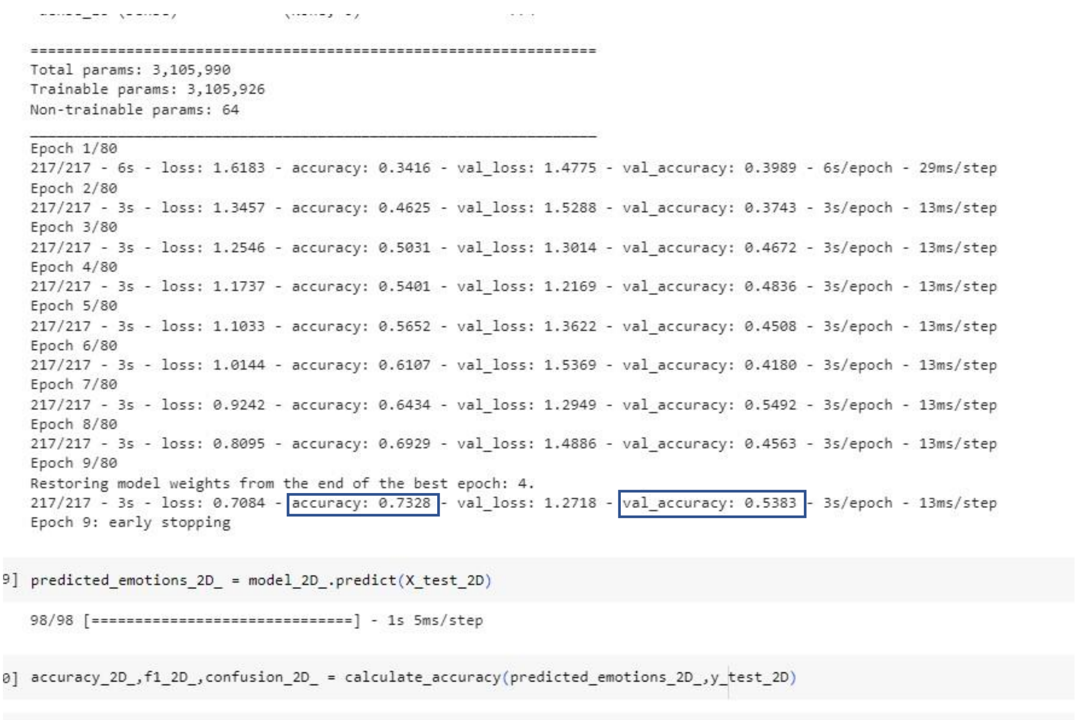
12.2 2D model

a)



```
model_2D_ = build_2D_model1(X_train_2D,X_valid_2D,y_train_2D,y_valid_2D, 80, 2)
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 122, 74, 32)       1600

 max_pooling2d_8 (MaxPooling  (None, 61, 37, 32)       0
 2D)

 batch_normalization_21 (Bat  (None, 61, 37, 32)       128
 chNormalization)

 conv2d_9 (Conv2D)           (None, 59, 35, 64)        18496

 max_pooling2d_9 (MaxPooling  (None, 29, 17, 64)       0
 2D)

 dropout (Dropout)           (None, 29, 17, 64)        0

 conv2d_10 (Conv2D)          (None, 27, 15, 64)        36928

 max_pooling2d_10 (MaxPoolin  (None, 13, 7, 64)        0
 g2D)

 flatten_8 (Flatten)         (None, 5824)              0

 dense_16 (Dense)            (None, 512)               2982400

 dense_17 (Dense)            (None, 128)               65664

 dense_18 (Dense)            (None, 6)                 774

=================================================================
Total params: 3,105,990
Trainable params: 3,105,926
Non-trainable params: 64
_________________________________________________________________
Epoch 1/80
217/217 - 6s - loss: 1.6183 - accuracy: 0.3416 - val_loss: 1.4775 - val_accuracy: 0.3989 - 6s/epoch - 29ms/step
Epoch 2/80
217/217 - 3s - loss: 1.3457 - accuracy: 0.4625 - val_loss: 1.5288 - val_accuracy: 0.3743 - 3s/epoch - 13ms/step
Epoch 3/80
217/217 - 3s - loss: 1.2546 - accuracy: 0.5031 - val_loss: 1.3014 - val_accuracy: 0.4672 - 3s/epoch - 13ms/step
Epoch 4/80
```

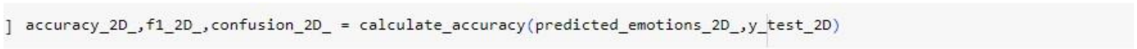


```
=================================================================
Total params: 3,105,990
Trainable params: 3,105,926
Non-trainable params: 64
_________________________________________________________________
Epoch 1/80
217/217 - 6s - loss: 1.6183 - accuracy: 0.3416 - val_loss: 1.4775 - val_accuracy: 0.3989 - 6s/epoch - 29ms/step
Epoch 2/80
217/217 - 3s - loss: 1.3457 - accuracy: 0.4625 - val_loss: 1.5288 - val_accuracy: 0.3743 - 3s/epoch - 13ms/step
Epoch 3/80
217/217 - 3s - loss: 1.2546 - accuracy: 0.5031 - val_loss: 1.3014 - val_accuracy: 0.4672 - 3s/epoch - 13ms/step
```

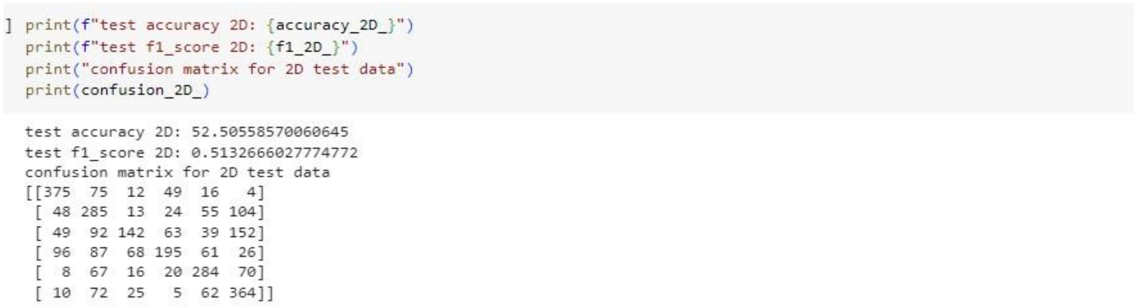
```
Epoch 4/80
217/217 - 3s - loss: 1.1737 - accuracy: 0.5401 - val_loss: 1.2169 - val_accuracy: 0.4836 - 3s/epoch - 13ms/step
Epoch 5/80
217/217 - 3s - loss: 1.1033 - accuracy: 0.5652 - val_loss: 1.3622 - val_accuracy: 0.4508 - 3s/epoch - 13ms/step
Epoch 6/80
217/217 - 3s - loss: 1.0144 - accuracy: 0.6107 - val_loss: 1.5369 - val_accuracy: 0.4180 - 3s/epoch - 13ms/step
Epoch 7/80
217/217 - 3s - loss: 0.9242 - accuracy: 0.6434 - val_loss: 1.2949 - val_accuracy: 0.5492 - 3s/epoch - 13ms/step
Epoch 8/80
217/217 - 3s - loss: 0.8095 - accuracy: 0.6929 - val_loss: 1.4886 - val_accuracy: 0.4563 - 3s/epoch - 13ms/step
Epoch 9/80
Restoring model weights from the end of the best epoch: 4.
217/217 - 3s - loss: 0.7084 - accuracy: 0.7328 - val_loss: 1.2718 - val_accuracy: 0.5383 - 3s/epoch - 13ms/step
Epoch 9: early stopping
```

```
9] predicted_emotions_2D_ = model_2D_.predict(X_test_2D)

   98/98 [==============================] - 1s 5ms/step
```

```
0] accuracy_2D_,f1_2D_,confusion_2D_ = calculate_accuracy(predicted_emotions_2D_,y_test_2D)
```

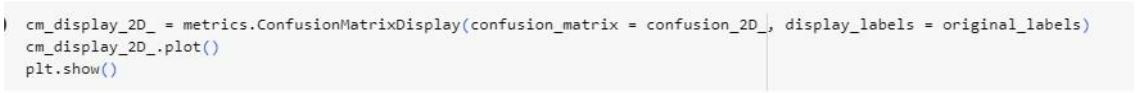


```
] accuracy_2D_,f1_2D_,confusion_2D_ = calculate_accuracy(predicted_emotions_2D_,y_test_2D)
```

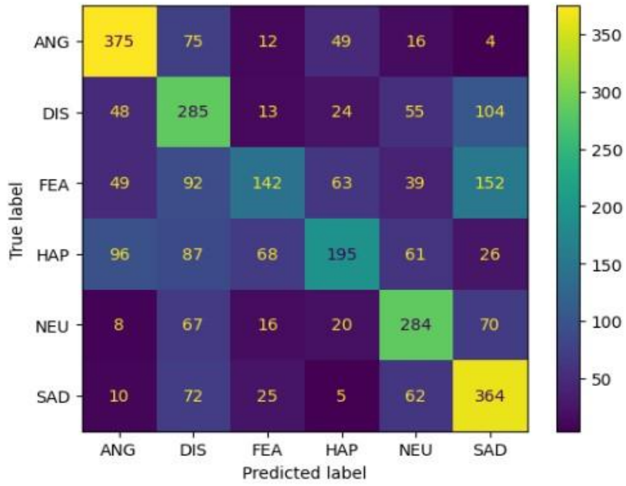


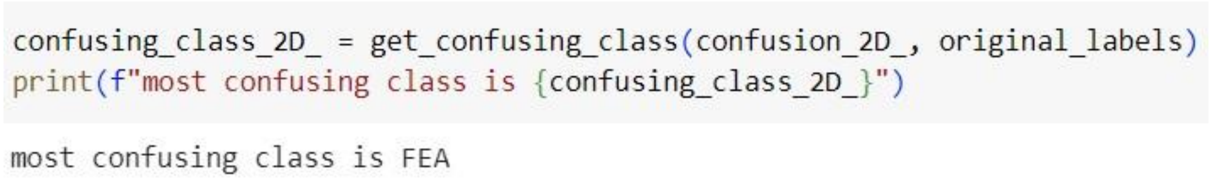
```
] print(f"test accuracy 2D: {accuracy_2D_}")
  print(f"test f1_score 2D: {f1_2D_}")
  print("confusion matrix for 2D test data")
  print(confusion_2D_)
```

```
test accuracy 2D: 52.50558570060645
test f1_score 2D: 0.5132666027774772
confusion matrix for 2D test data
[[375  75  12  49  16   4]
 [ 48 285  13  24  55 104]
 [ 49  92 142  63  39 152]
 [ 96  87  68 195  61  26]
 [  8  67  16  20 284  70]
 [ 10  72  25   5  62 364]]
```



```
cm_display_2D_ = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_2D_, display_labels = original_labels)
cm_display_2D_.plot()
plt.show()
```

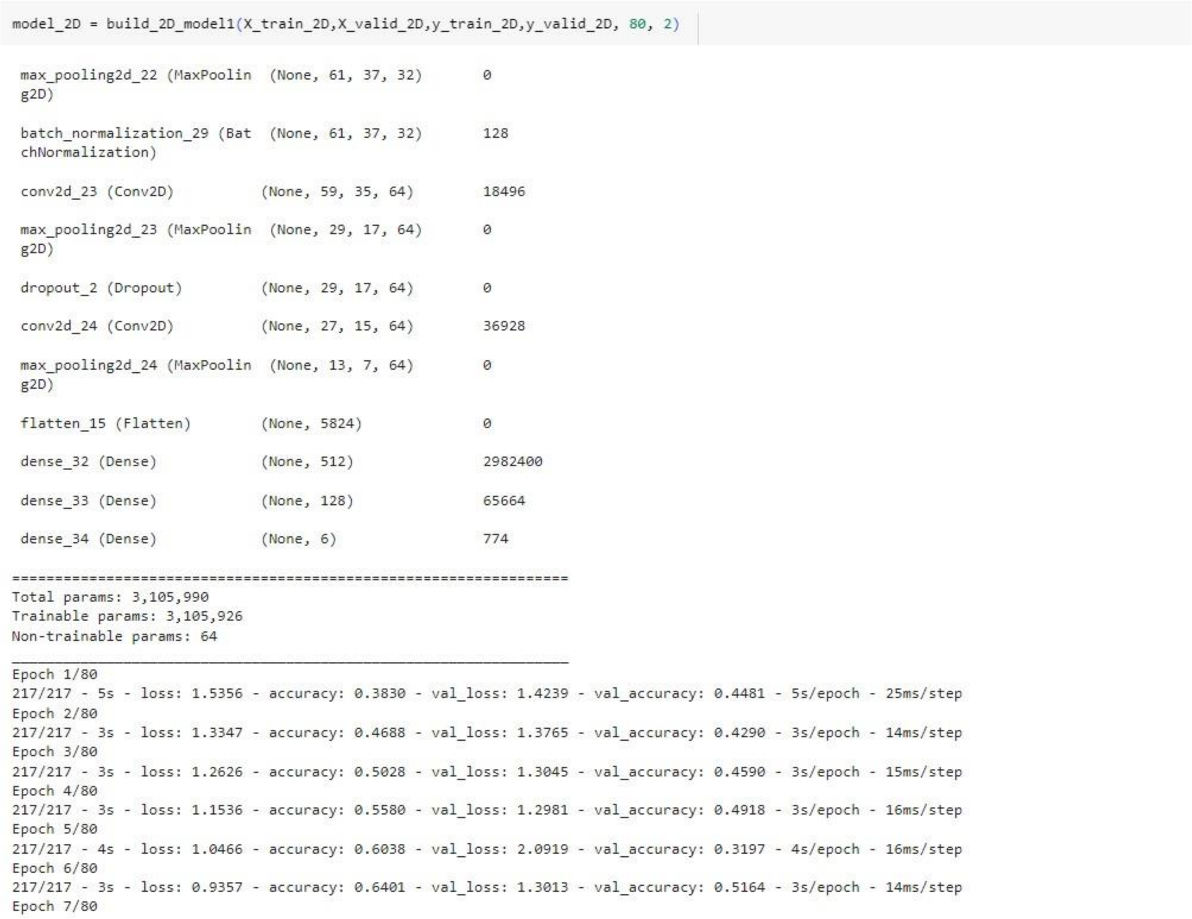




```
confusing_class_2D_ = get_confusing_class(confusion_2D_, original_labels)
print(f"most confusing class is {confusing_class_2D_}")
```

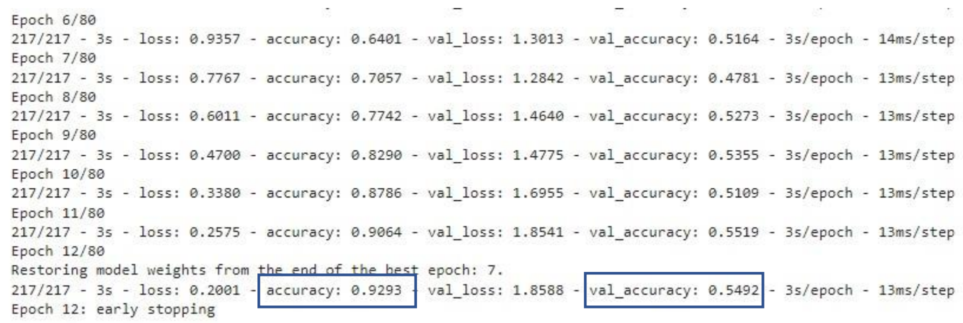
```
most confusing class is FEA
```

b)

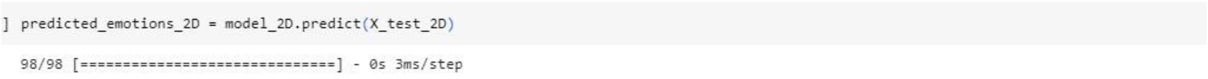


```
model_2D = build_2D_model1(X_train_2D,X_valid_2D,y_train_2D,y_valid_2D, 80, 2)
```

```
 max_pooling2d_22 (MaxPoolin  (None, 61, 37, 32)       0
 g2D)

 batch_normalization_29 (Bat  (None, 61, 37, 32)       128
 chNormalization)

 conv2d_23 (Conv2D)          (None, 59, 35, 64)       18496

 max_pooling2d_23 (MaxPoolin  (None, 29, 17, 64)       0
 g2D)

 dropout_2 (Dropout)         (None, 29, 17, 64)       0

 conv2d_24 (Conv2D)          (None, 27, 15, 64)       36928

 max_pooling2d_24 (MaxPoolin  (None, 13, 7, 64)        0
 g2D)

 flatten_15 (Flatten)        (None, 5824)             0

 dense_32 (Dense)            (None, 512)              2982400

 dense_33 (Dense)            (None, 128)              65664

 dense_34 (Dense)            (None, 6)                774

=================================================================
Total params: 3,105,990
Trainable params: 3,105,926
Non-trainable params: 64
_________________________________________________________________
Epoch 1/80
217/217 - 5s - loss: 1.5356 - accuracy: 0.3830 - val_loss: 1.4239 - val_accuracy: 0.4481 - 5s/epoch - 25ms/step
Epoch 2/80
217/217 - 3s - loss: 1.3347 - accuracy: 0.4688 - val_loss: 1.3765 - val_accuracy: 0.4290 - 3s/epoch - 14ms/step
Epoch 3/80
217/217 - 3s - loss: 1.2626 - accuracy: 0.5028 - val_loss: 1.3045 - val_accuracy: 0.4590 - 3s/epoch - 15ms/step
Epoch 4/80
217/217 - 3s - loss: 1.1536 - accuracy: 0.5580 - val_loss: 1.2981 - val_accuracy: 0.4918 - 3s/epoch - 16ms/step
Epoch 5/80
217/217 - 4s - loss: 1.0466 - accuracy: 0.6038 - val_loss: 2.0919 - val_accuracy: 0.3197 - 4s/epoch - 16ms/step
Epoch 6/80
217/217 - 3s - loss: 0.9357 - accuracy: 0.6401 - val_loss: 1.3013 - val_accuracy: 0.5164 - 3s/epoch - 14ms/step
Epoch 7/80
```

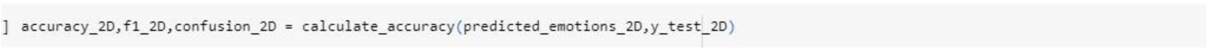


```
Epoch 6/80
217/217 - 3s - loss: 0.9357 - accuracy: 0.6401 - val_loss: 1.3013 - val_accuracy: 0.5164 - 3s/epoch - 14ms/step
Epoch 7/80
217/217 - 3s - loss: 0.7767 - accuracy: 0.7057 - val_loss: 1.2842 - val_accuracy: 0.4781 - 3s/epoch - 13ms/step
Epoch 8/80
217/217 - 3s - loss: 0.6011 - accuracy: 0.7742 - val_loss: 1.4640 - val_accuracy: 0.5273 - 3s/epoch - 13ms/step
Epoch 9/80
217/217 - 3s - loss: 0.4700 - accuracy: 0.8290 - val_loss: 1.4775 - val_accuracy: 0.5355 - 3s/epoch - 13ms/step
Epoch 10/80
217/217 - 3s - loss: 0.3380 - accuracy: 0.8786 - val_loss: 1.6955 - val_accuracy: 0.5109 - 3s/epoch - 13ms/step
Epoch 11/80
217/217 - 3s - loss: 0.2575 - accuracy: 0.9064 - val_loss: 1.8541 - val_accuracy: 0.5519 - 3s/epoch - 13ms/step
Epoch 12/80
Restoring model weights from the end of the best epoch: 7.
217/217 - 3s - loss: 0.2001 - accuracy: 0.9293 - val_loss: 1.8588 - val_accuracy: 0.5492 - 3s/epoch - 13ms/step
Epoch 12: early stopping
```

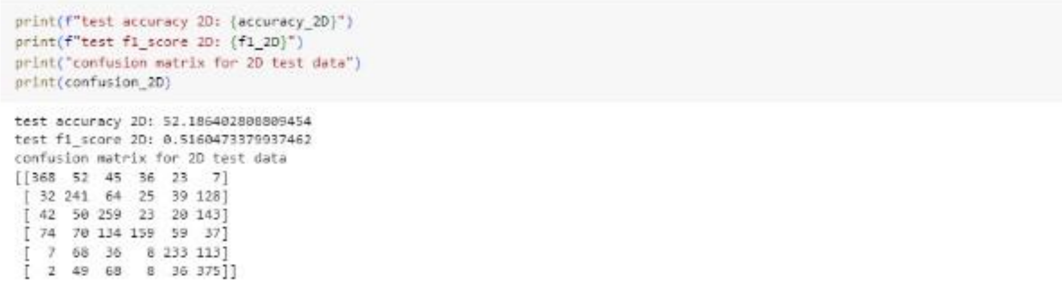


```
] predicted_emotions_2D = model_2D.predict(X_test_2D)
```

```
98/98 [==============================] - 0s 3ms/step
```

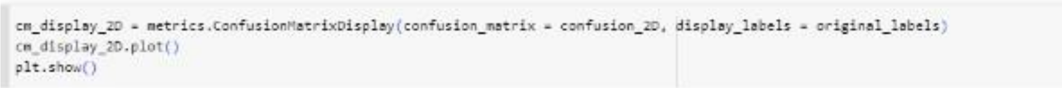


```
] accuracy_2D,f1_2D,confusion_2D = calculate_accuracy(predicted_emotions_2D,y_test_2D)
```

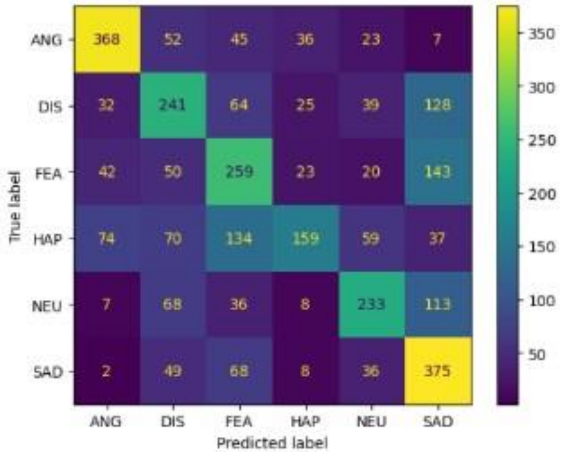


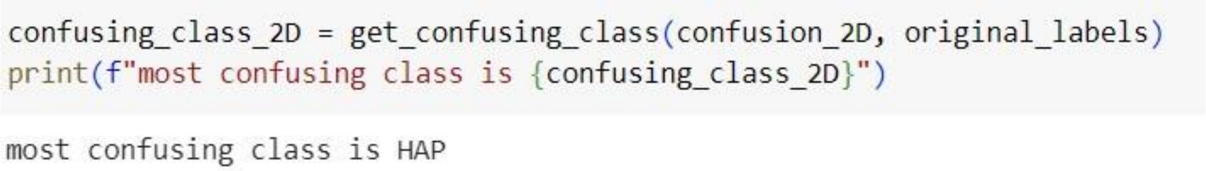
```
print(f"test accuracy 2D: {accuracy_2D}")
print(f"test f1_score 2D: {f1_2D}")
print("confusion matrix for 2D test data")
print(confusion_2D)
```

```
test accuracy 2D: 52.186402808809454
test f1_score 2D: 0.5160473379937462
confusion matrix for 2D test data
[[368  52  45  36  23   7]
 [ 32 241  64  25  39 128]
 [ 42  50 259  23  20 143]
 [ 74  70 134 159  59  37]
 [  7  68  36   8 233 113]
 [  2  49  68   8  36 375]]
```



```
cm_display_2D = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_2D, display_labels = original_labels)
cm_display_2D.plot()
plt.show()
```

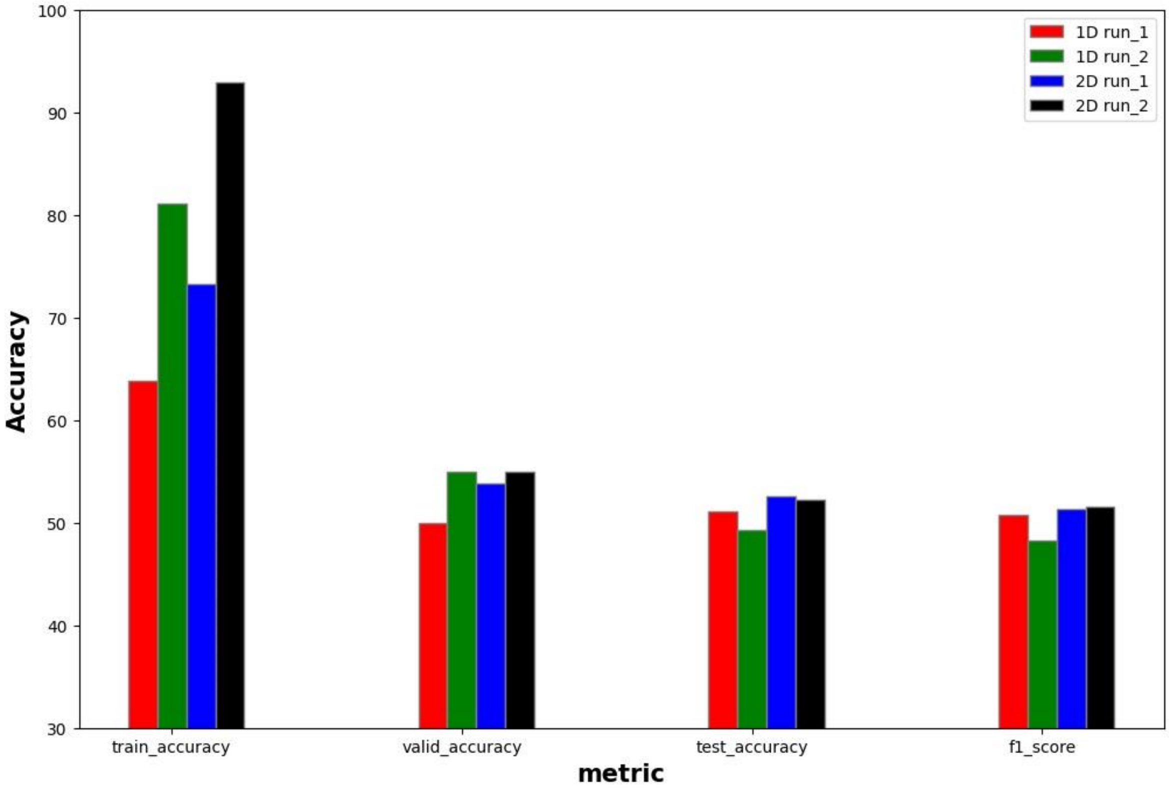




```
confusing_class_2D = get_confusing_class(confusion_2D, original_labels)
print(f"most confusing class is {confusing_class_2D}")
```

```
most confusing class is HAP
```

13.Comparison between 1D, 2D according to measures obtained

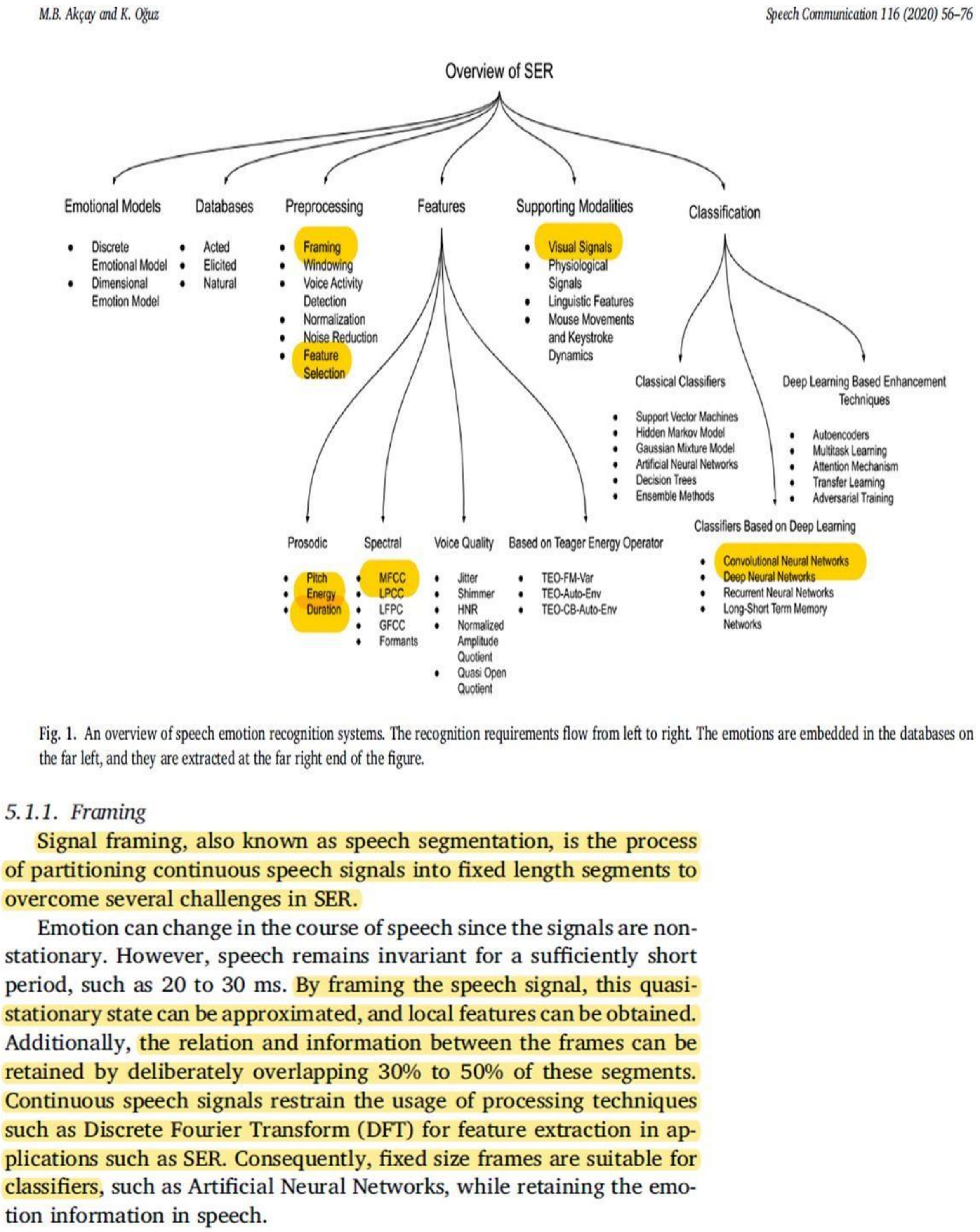


It can be noticed that measures of 2D model are better than 1D model. For the 'train_accuracy', 'valid_accuracy', and 'test_accuracy' measures, the 2D CNN models (measures_2D and measures_2D_1) achieved higher accuracy scores than the 1D CNN models (measures_1D and measures_1D_1) for all three sets (train, validation, and test). This suggests that the 2D CNN models are better at generalizing to new data than the 1D CNN models.

For the 'f1_score' measure, the 2D CNN models also achieved higher scores than the 1D CNN models. This measure takes into account both precision and recall, and is often used as a summary measure of a model's classification performance.

14.Referenced Paper

Akçay, M. B., & Oğuz, K. (2020). Speech emotion recognition: Emotional models, databases, features, preprocessing methods, supporting modalities, and classifiers



M.B. Akçay and K. Oğuz

Speech Communication 116 (2020) 56–76

Overview of SER
Emotional Models
Discrete Emotional Model
Dimensional Emotion Model
Databases
Acted
Elicited
Natural
Preprocessing
Framing
Windowing
Voice Activity Detection
Normalization
Noise Reduction
Feature Selection
Features
Supporting Modalities
Visual Signals
Physiological Signals
Linguistic Features
Mouse Movements and Keystroke Dynamics
Classification
Classical Classifiers
Support Vector Machines
Hidden Markov Model
Gaussian Mixture Model
Artificial Neural Networks
Decision Trees
Ensemble Methods
Deep Learning Based Enhancement Techniques
Autoencoders
Multitask Learning
Attention Mechanism
Transfer Learning
Adversarial Training
Classifiers Based on Deep Learning
Convolutional Neural Networks
Deep Neural Networks
Recurrent Neural Networks
Long-Short Term Memory Networks
Prosodic
Pitch
Energy
Duration
Spectral
MFCC
LPCC
LFPC
GFCC
Formants
Voice Quality
Jitter
Shimmer
HNR
Normalized Amplitude Quotient
Quasi Open Quotient
Based on Teager Energy Operator
TEO-FM-Var
TEO-Auto-Env
TEO-CB-Auto-Env

Fig. 1. An overview of speech emotion recognition systems. The recognition requirements flow from left to right. The emotions are embedded in the databases on the far left, and they are extracted at the far right end of the figure.

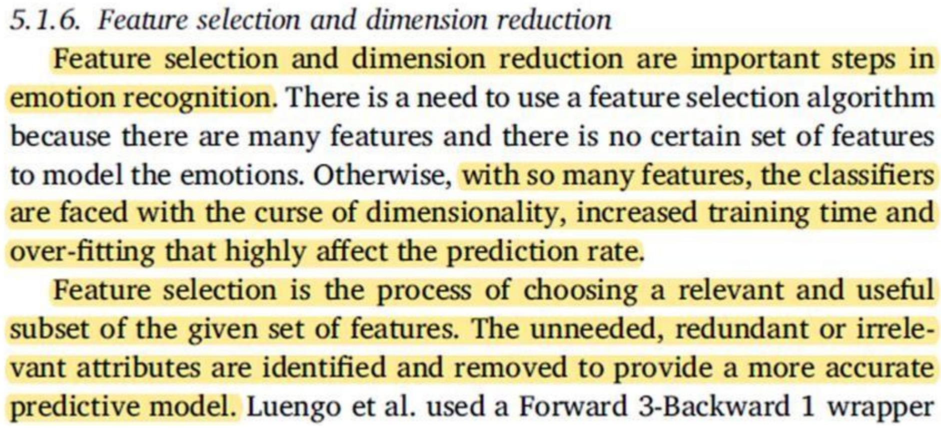
5.1.1. Framing

Signal framing, also known as speech segmentation, is the process of partitioning continuous speech signals into fixed length segments to overcome several challenges in SER.

Emotion can change in the course of speech since the signals are non-stationary. However, speech remains invariant for a sufficiently short period, such as 20 to 30 ms. By framing the speech signal, this quasi-stationary state can be approximated, and local features can be obtained. Additionally, the relation and information between the frames can be retained by deliberately overlapping 30% to 50% of these segments. Continuous speech signals restrain the usage of processing techniques such as Discrete Fourier Transform (DFT) for feature extraction in applications such as SER. Consequently, fixed size frames are suitable for classifiers, such as Artificial Neural Networks, while retaining the emotion information in speech.

.

This highlighted part has inspired us to use equal frames, and extract features for each single frame on its own.

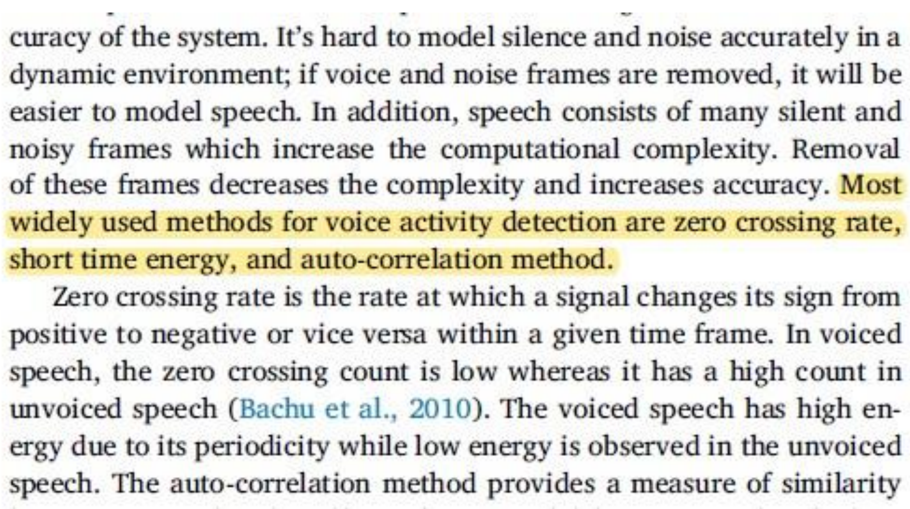


5.1.6. Feature selection and dimension reduction

Feature selection and dimension reduction are important steps in emotion recognition. There is a need to use a feature selection algorithm because there are many features and there is no certain set of features to model the emotions. Otherwise, with so many features, the classifiers are faced with the curse of dimensionality, increased training time and over-fitting that highly affect the prediction rate.

Feature selection is the process of choosing a relevant and useful subset of the given set of features. The unneeded, redundant or irrelevant attributes are identified and removed to provide a more accurate predictive model. Luengo et al. used a Forward 3-Backward 1 wrapper

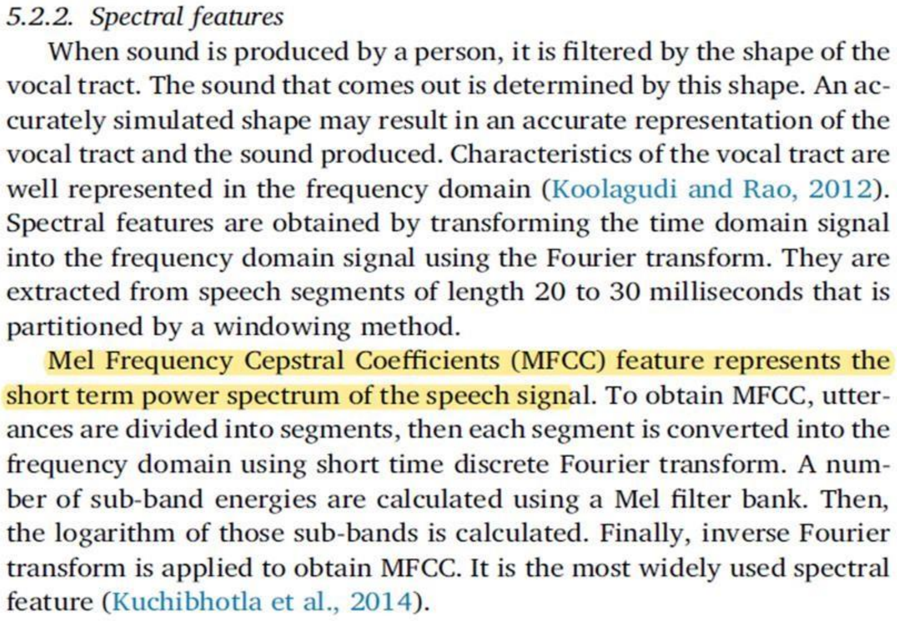
Instead of having audios with thousands of dimensions, we have reduced them by dividing the total audio into frames, and selecting only representative features for each frame.



curacy of the system. It's hard to model silence and noise accurately in a dynamic environment; if voice and noise frames are removed, it will be easier to model speech. In addition, speech consists of many silent and noisy frames which increase the computational complexity. Removal of these frames decreases the complexity and increases accuracy. Most widely used methods for voice activity detection are zero crossing rate, short time energy, and auto-correlation method.

Zero crossing rate is the rate at which a signal changes its sign from positive to negative or vice versa within a given time frame. In voiced speech, the zero crossing count is low whereas it has a high count in unvoiced speech (Bachu et al., 2010). The voiced speech has high energy due to its periodicity while low energy is observed in the unvoiced speech. The auto-correlation method provides a measure of similarity

Zero crossing rate was used as a feature for each frame, as the paper suggested that it is most widely used for activity detection.



5.2.2. Spectral features

When sound is produced by a person, it is filtered by the shape of the vocal tract. The sound that comes out is determined by this shape. An accurately simulated shape may result in an accurate representation of the vocal tract and the sound produced. Characteristics of the vocal tract are well represented in the frequency domain (Koolagudi and Rao, 2012). Spectral features are obtained by transforming the time domain signal into the frequency domain signal using the Fourier transform. They are extracted from speech segments of length 20 to 30 milliseconds that is partitioned by a windowing method.

Mel Frequency Cepstral Coefficients (MFCC) feature represents the short term power spectrum of the speech signal. To obtain MFCC, utterances are divided into segments, then each segment is converted into the frequency domain using short time discrete Fourier transform. A number of sub-band energies are calculated using a Mel filter bank. Then, the logarithm of those sub-bands is calculated. Finally, inverse Fourier transform is applied to obtain MFCC. It is the most widely used spectral feature (Kuchibhotla et al., 2014).

MFCC was used as a frequency feature space to get spectral audio features