Assignment #3
Speech Emotion Recognition

Names	ID
Sally Kamel Soliman Armanyous	19015762
Salma Saeed Mahmoud Ghareb	19015779
Esraa Hassan Mokhtar Aboshady	19015407

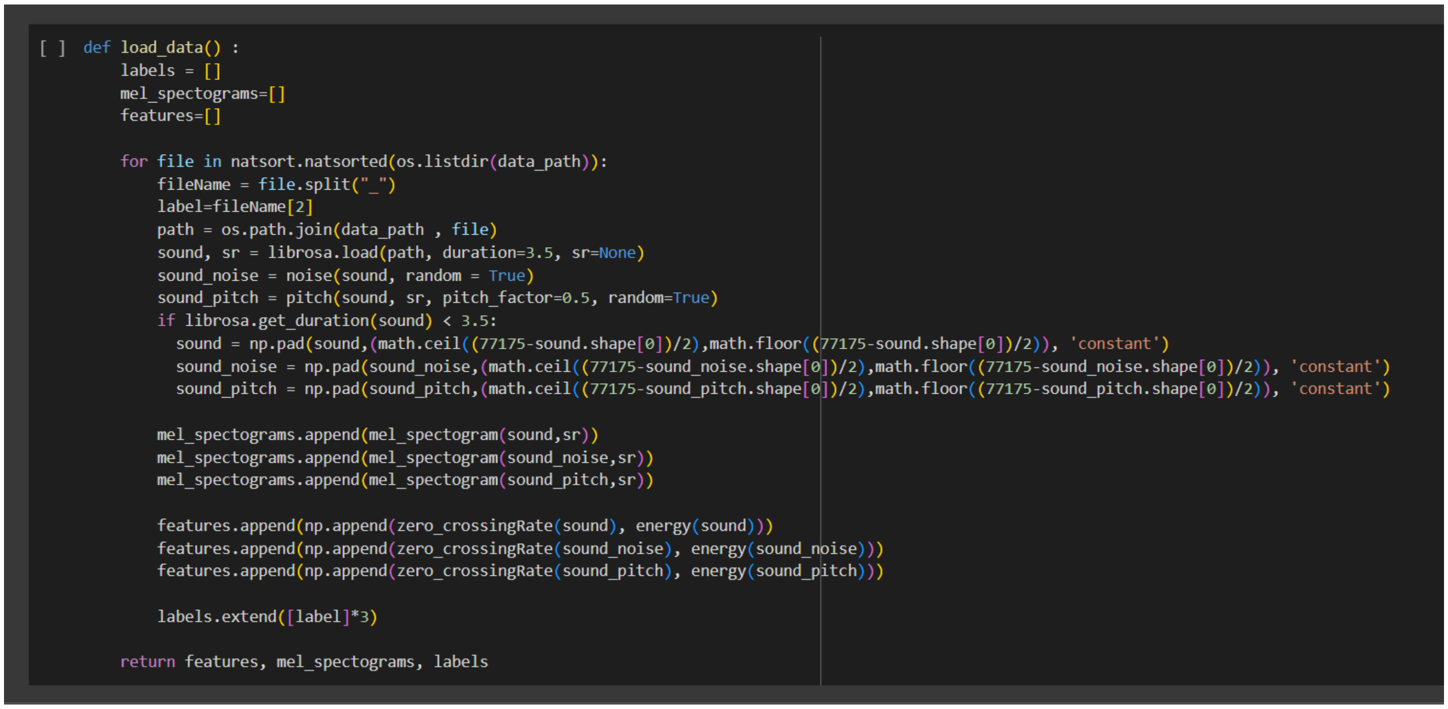
Problem Statement:

Speech is the most natural way of expressing ourselves as humans. It is only natural then to extend this communication medium to computer applications. We define speech emotion recognition (SER) systems as a collection of methodologies that process and classify speech signals to detect the embedded emotions. Below we will show the needed steps to achieve the goal of the assignment.

1. Download the Dataset and Understand the Format

We will use CREMA dataset and load it through the "load_data" function that:

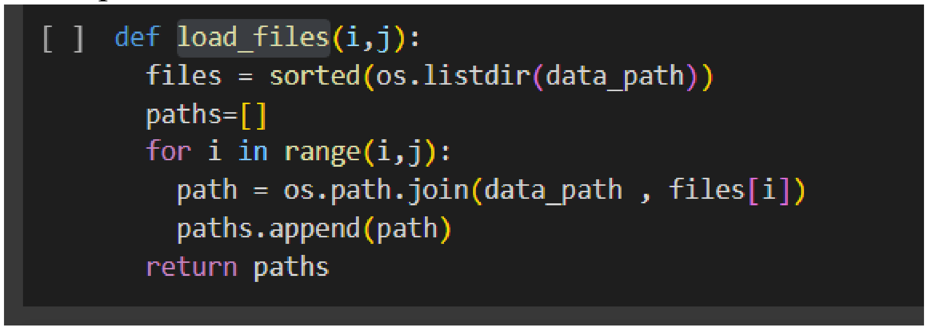
- Iterates through the files in the directory specified by data_path, sorted in natural order.
- For each file, the file name is split by underscores and the third element (index 2) is extracted and stored in label.
- The full path of the file is created using os.path.join() and loaded into a numpy array sound using the librosa.load() function. The duration parameter specifies the maximum duration of the audio in seconds, and sr=None uses the native sampling rate of the audio file.
- Two modified versions of sound are created: sound_noise and sound_pitch. sound_noise adds random noise to sound, and sound_pitch changes the pitch of sound by a factor of 0.5.
- If the duration of sound is less than 3.5 seconds, it is padded with zeros on both sides to make its length 77175 (the expected length of the audio signal after preprocessing).
- The mel spectrograms of sound, sound_noise, and sound_pitch are computed and appended to the mel_spectograms list.
- The zero-crossing rate and energy features of sound, sound_noise, and sound_pitch are computed using the zero_crossingRate() and energy() functions, respectively. The two feature vectors are concatenated using np.append() and appended to the features list.



```
[ ]  def load_data() :
         labels = []
         mel_spectograms=[]
         features=[]

         for file in natsort.natsorted(os.listdir(data_path)):
             fileName = file.split("_")
             label=fileName[2]
             path = os.path.join(data_path , file)
             sound, sr = librosa.load(path, duration=3.5, sr=None)
             sound_noise = noise(sound, random = True)
             sound_pitch = pitch(sound, sr, pitch_factor=0.5, random=True)
             if librosa.get_duration(sound) < 3.5:
               sound = np.pad(sound,(math.ceil((77175-sound.shape[0])/2),math.floor((77175-sound.shape[0])/2)), 'constant')
               sound_noise = np.pad(sound_noise,(math.ceil((77175-sound_noise.shape[0])/2),math.floor((77175-sound_noise.shape[0])/2)), 'constant')
               sound_pitch = np.pad(sound_pitch,(math.ceil((77175-sound_pitch.shape[0])/2),math.floor((77175-sound_pitch.shape[0])/2)), 'constant')

             mel_spectograms.append(mel_spectogram(sound,sr))
             mel_spectograms.append(mel_spectogram(sound_noise,sr))
             mel_spectograms.append(mel_spectogram(sound_pitch,sr))

             features.append(np.append(zero_crossingRate(sound), energy(sound)))
             features.append(np.append(zero_crossingRate(sound_noise), energy(sound_noise)))
             features.append(np.append(zero_crossingRate(sound_pitch), energy(sound_pitch)))

             labels.extend([label]*3)

         return features, mel_spectograms, labels
```

Here's the "load_files" function:

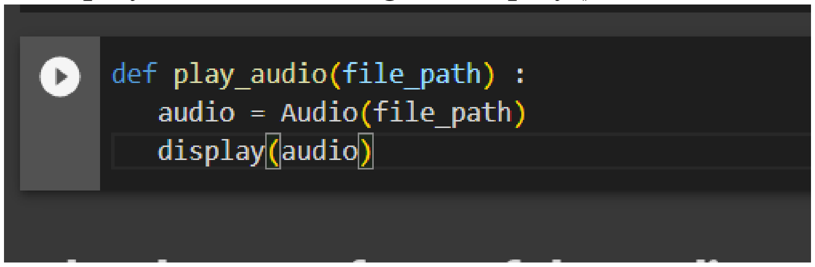
- It takes two indices as input, representing the start and end of a range of files to be loaded, obtains the paths of these files using os.path.join(), and returns a list of the file paths.



```
[ ]  def load_files(i,j):
       files = sorted(os.listdir(data_path))
       paths=[]
       for i in range(i,j):
         path = os.path.join(data_path , files[i])
         paths.append(path)
       return paths
```

The "play_audio" function:

- It creates an Audio object audio from the file_path using the Audio() function then displays the audio using the display() function.

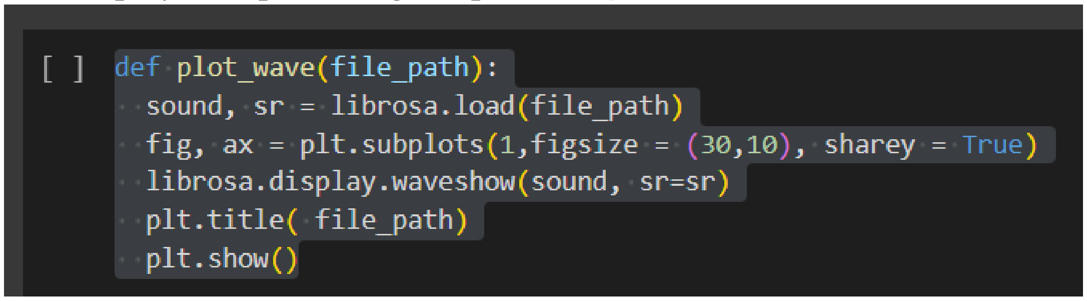


```
def play_audio(file_path) :
   audio = Audio(file_path)
   display(audio)
```

The "plot_wave" function:

- It loads the audio data and sampling rate of the file specified by file_path, and assigns them to the variables sound and sr, respectively.

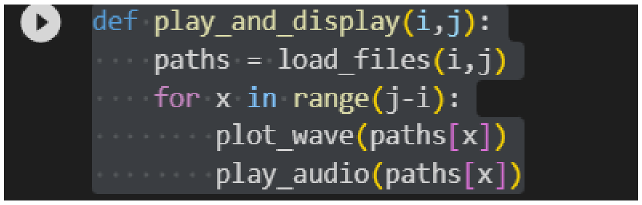
- It displays the waveform of the audio signal using the librosa.display.waveshow() function with arguments sound and sr for the audio data and sampling rate, respectively.
- It displays the plot using the plt.show() function.



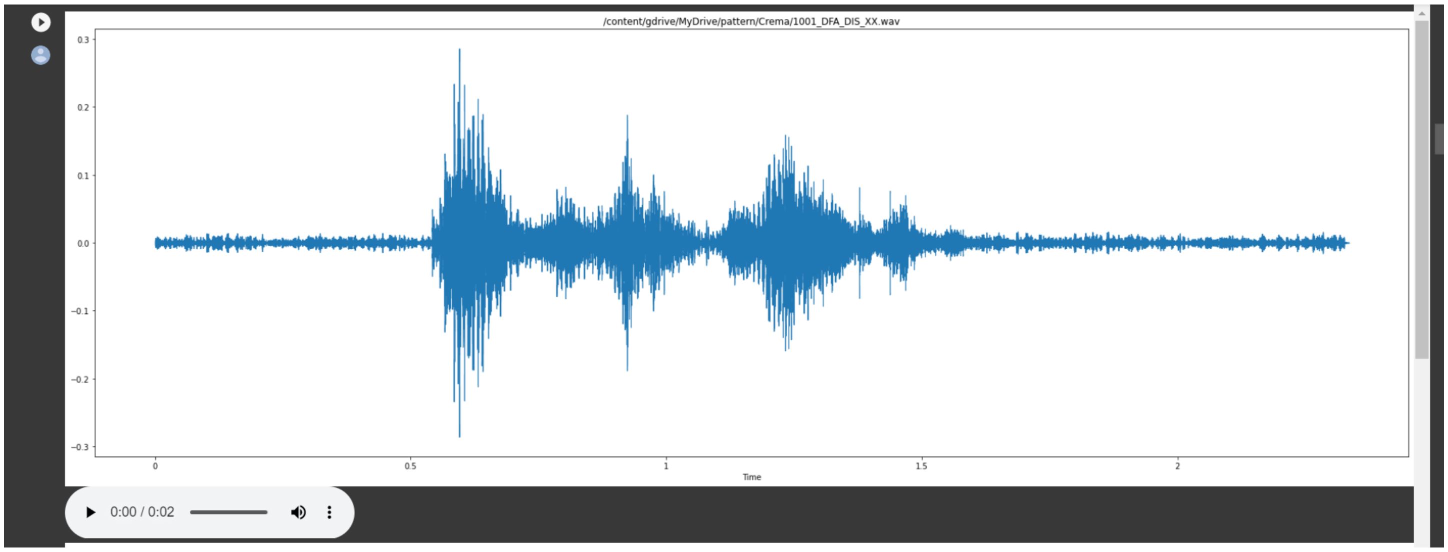
```
[ ]  def plot_wave(file_path):
       sound, sr = librosa.load(file_path)
       fig, ax = plt.subplots(1,figsize = (30,10), sharey = True)
       librosa.display.waveshow(sound, sr=sr)
       plt.title( file_path)
       plt.show()
```

The “play_and_display” function:

- It calls the load_files(i,j) function to obtain a list of file paths for files in the specified range [i,j).
- It uses a for loop to iterate over each file path in the paths list.
- Within the loop, it calls the plot_wave() function with the current file path to plot the waveform of the audio signal and display it in a new figure.
- It then calls the play_audio() function with the current file path to play and display the audio file in the notebook.



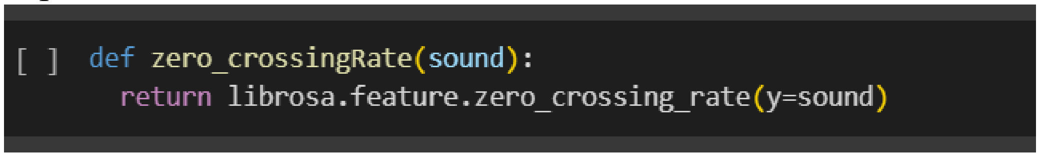
```
def play_and_display(i,j):
    paths = load_files(i,j)
    for x in range(j-i):
        plot_wave(paths[x])
        play_audio(paths[x])
```



2. Create the Feature Space:

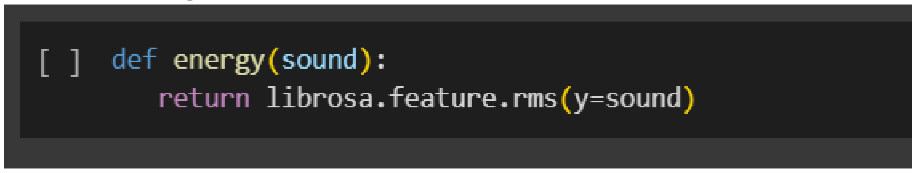
We used the time domain and here are multiple of features that can help improving the model:

- **Zero crossing rate:** The rate of sign-changes of the signal during the duration of a particular frame.



```
[ ]  def zero_crossingRate(sound):
       return librosa.feature.zero_crossing_rate(y=sound)
```

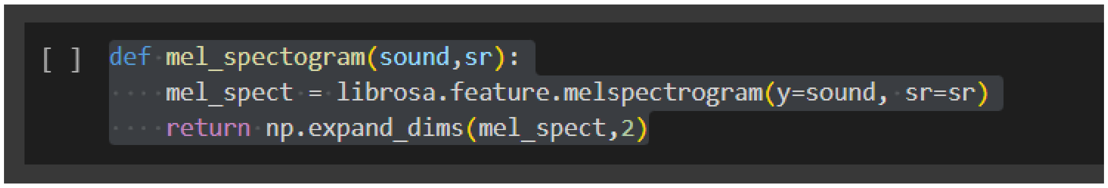
- **Energy:** The sum of squares of the signal values, normalized by the respective frame length.



```
[ ]  def energy(sound):
        return librosa.feature.rms(y=sound)
```

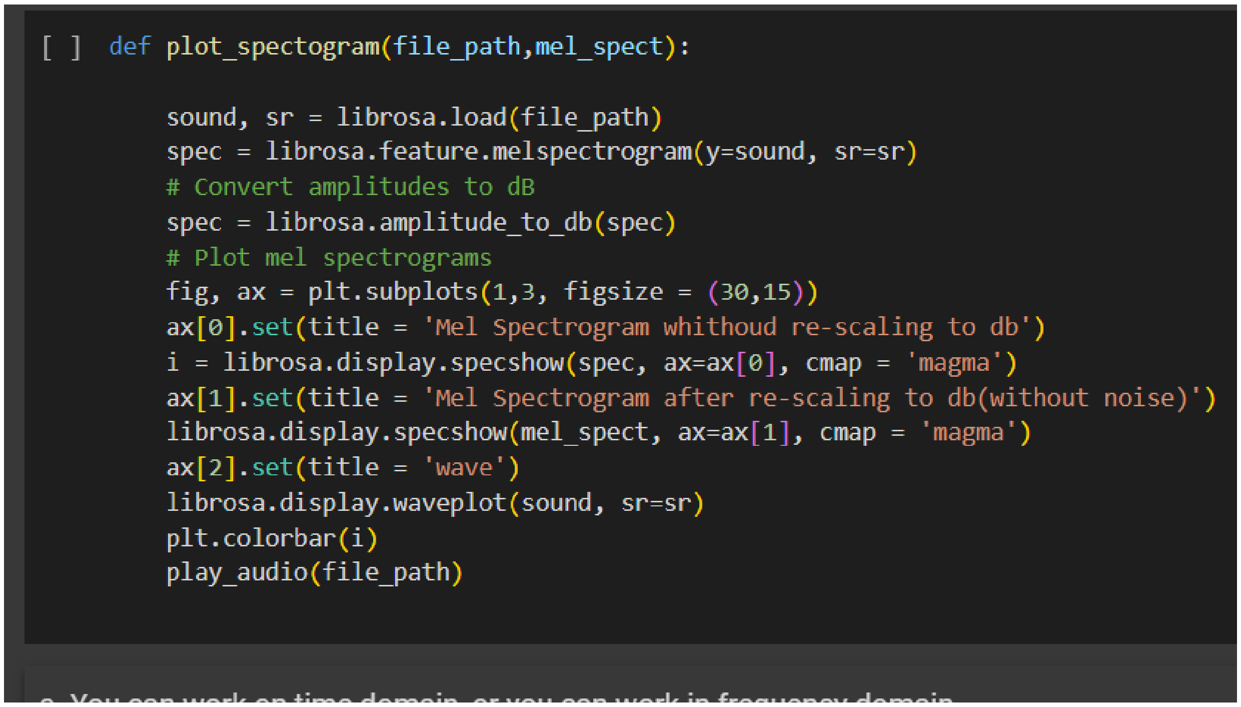
Then convert the audio waveform to mel-spectrogram and use this as the feature space:

- **mel_spect: This line uses the librosa library to compute the Mel spectrogram of the input sound signal, where y is the sound signal and sr is the sample rate.**
- **np.expand_dims: This line adds an extra dimension to the computed Mel spectrogram. This is done so that the spectrogram can be fed as an input to a convolutional neural network (CNN) which requires a 3D input tensor, where the first two dimensions represent the width and height of the input image (spectrogram), and the third dimension represents the number of channels.**



```
[ ]  def mel_spectogram(sound,sr):
         mel_spect = librosa.feature.melspectrogram(y=sound, sr=sr)
         return np.expand_dims(mel_spect,2)
```

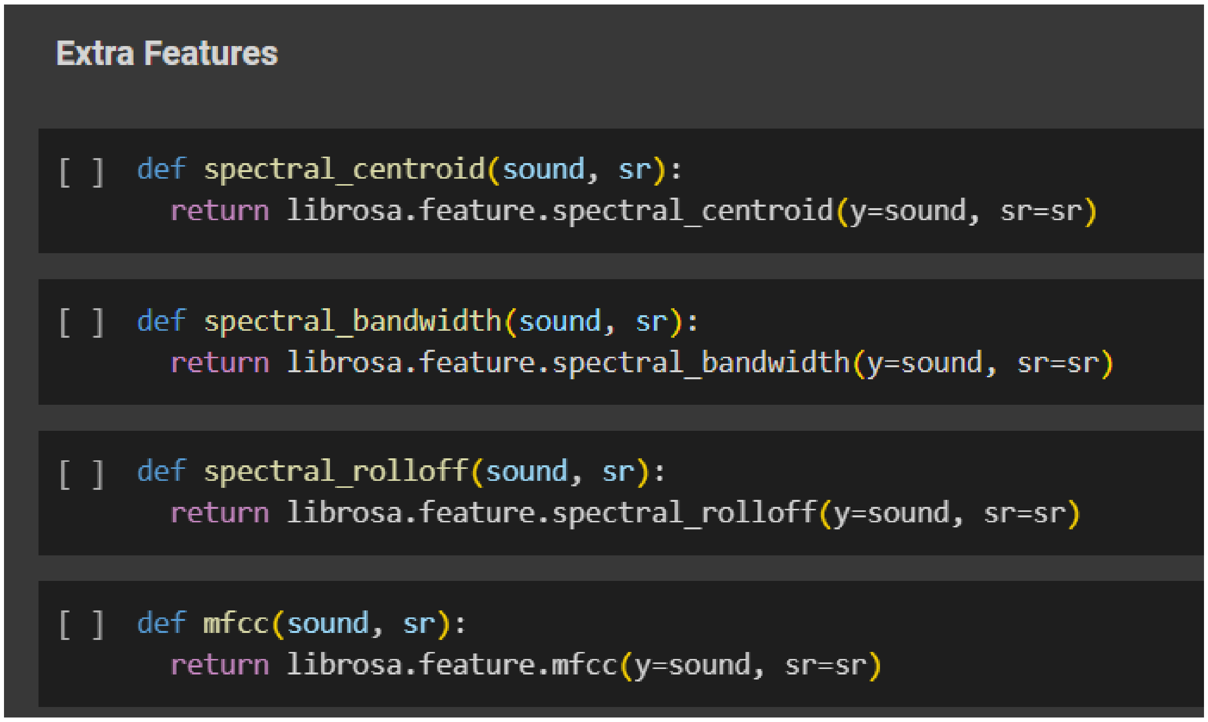
- **Then plots a mel spectrogram of a given audio file and displays it along with the waveform of the audio file. It takes two arguments: file_path which is the path to the audio file and mel_spect which is the mel spectrogram of the audio file.**



```
[ ]  def plot_spectogram(file_path,mel_spect):

         sound, sr = librosa.load(file_path)
         spec = librosa.feature.melspectrogram(y=sound, sr=sr)
         # Convert amplitudes to dB
         spec = librosa.amplitude_to_db(spec)
         # Plot mel spectrograms
         fig, ax = plt.subplots(1,3, figsize = (30,15))
         ax[0].set(title = 'Mel Spectrogram whithoud re-scaling to db')
         i = librosa.display.specshow(spec, ax=ax[0], cmap = 'magma')
         ax[1].set(title = 'Mel Spectrogram after re-scaling to db(without noise)')
         librosa.display.specshow(mel_spect, ax=ax[1], cmap = 'magma')
         ax[2].set(title = 'wave')
         librosa.display.waveplot(sound, sr=sr)
         plt.colorbar(i)
         play_audio(file_path)
```

o You can work on time domain, or you can work in frequency domain

Extra features:



Extra Features

```
[ ]  def spectral_centroid(sound, sr):
       return librosa.feature.spectral_centroid(y=sound, sr=sr)
```

```
[ ]  def spectral_bandwidth(sound, sr):
       return librosa.feature.spectral_bandwidth(y=sound, sr=sr)
```

```
[ ]  def spectral_rolloff(sound, sr):
       return librosa.feature.spectral_rolloff(y=sound, sr=sr)
```

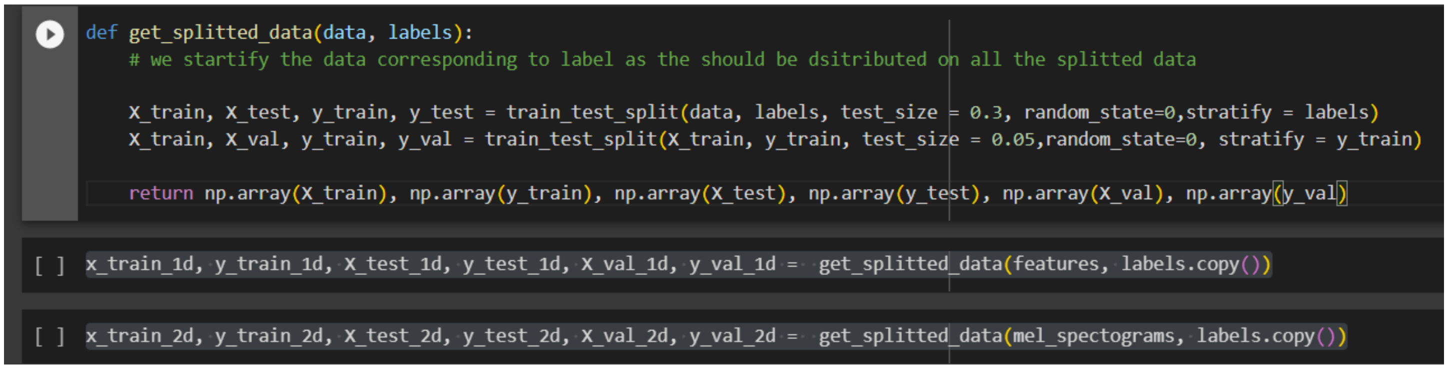
```
[ ]  def mfcc(sound, sr):
       return librosa.feature.mfcc(y=sound, sr=sr)
```

3. Building the Model:

Split the data into 70% training and validation and 30% testing. Use 5% of the training and validation data for validation.

The function **“get_splitted_data”** takes in two arguments data and labels. These are the features and corresponding labels respectively:

- The function splits the data and labels into three sets - training, validation and testing sets.
- The training set is further split into training and validation sets, with a ratio of 95:5 respectively.
- The stratify parameter in train_test_split ensures that the labels are distributed in the same ratio across all the sets, maintaining the same class distribution.

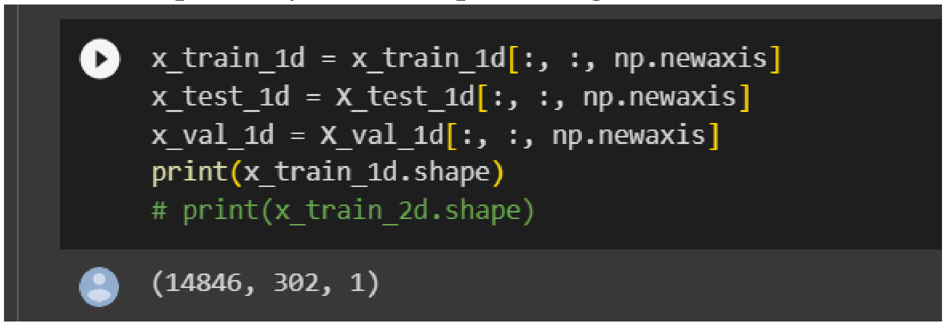


```
def get_splitted_data(data, labels):
    # we startify the data corresponding to label as the should be dsitributed on all the splitted data

    X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size = 0.3, random_state=0,stratify = labels)
    X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size = 0.05,random_state=0, stratify = y_train)

    return np.array(X_train), np.array(y_train), np.array(X_test), np.array(y_test), np.array(X_val), np.array(y_val)
```

```
x_train_1d, y_train_1d, X_test_1d, y_test_1d, X_val_1d, y_val_1d =  get_splitted_data(features, labels.copy())
```

```
x_train_2d, y_train_2d, X_test_2d, y_test_2d, X_val_2d, y_val_2d =  get_splitted_data(mel_spectograms, labels.copy())
```

Then we create new arrays with an additional dimension using np.newaxis to represent the channel required by some deep learning frameworks.

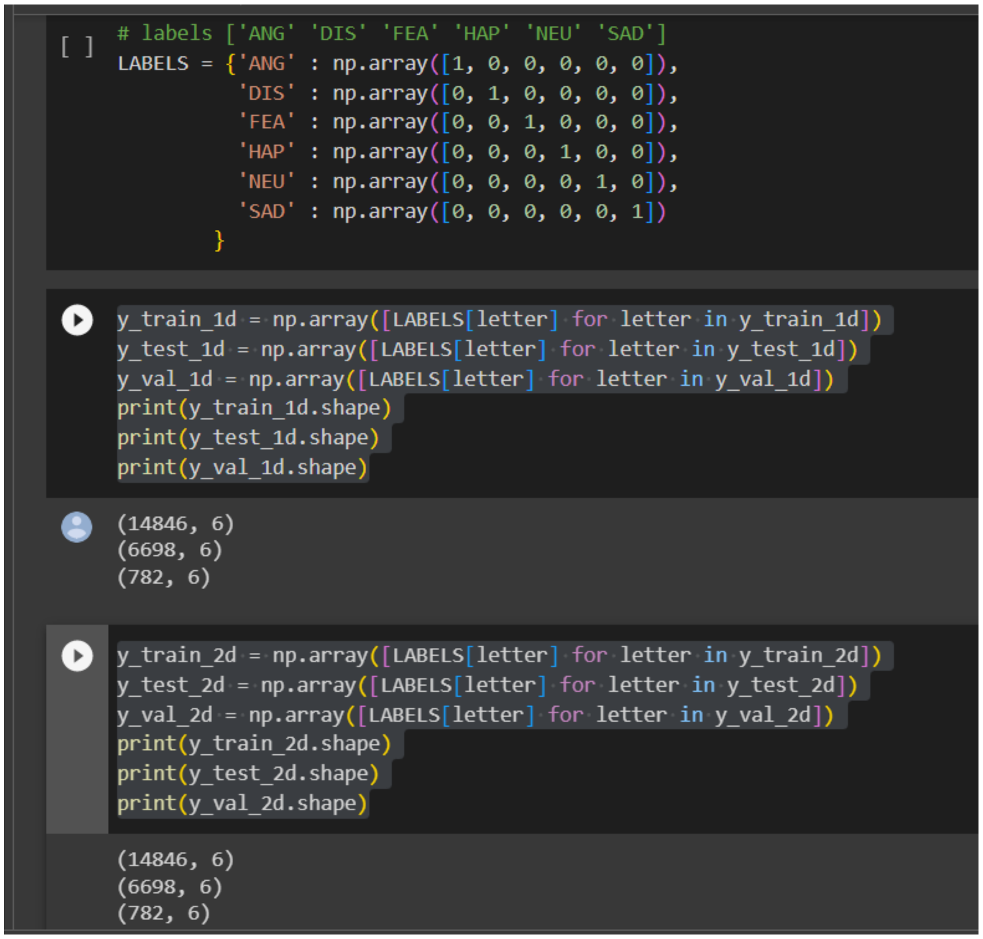


```
x_train_1d = x_train_1d[:, :, np.newaxis]
x_test_1d = X_test_1d[:, :, np.newaxis]
x_val_1d = X_val_1d[:, :, np.newaxis]
print(x_train_1d.shape)
# print(x_train_2d.shape)
```

```
(14846, 302, 1)
```

Then we create a dictionary **LABELS** that maps each label to a one-hot encoded vector representing that label. Then, the training, test, and validation labels are converted from their original string format to one-hot encoded vectors using this dictionary.

- The variable **y_train_1d** is a one-dimensional numpy array of shape (n_samples, n_classes), where n_samples is the number of training samples and n_classes is the number of possible classes (6 in this case).
- Same for **y_test_1d** and **y_val_1d**
- The same process is repeated for the two-dimensional data, resulting in the variables **y_train_2d**, **y_test_2d**, and **y_val_2d**, which are all two-dimensional numpy arrays of shape (n_samples, n_classes).



```
# labels ['ANG' 'DIS' 'FEA' 'HAP' 'NEU' 'SAD']
LABELS = {'ANG' : np.array([1, 0, 0, 0, 0, 0]),
          'DIS' : np.array([0, 1, 0, 0, 0, 0]),
          'FEA' : np.array([0, 0, 1, 0, 0, 0]),
          'HAP' : np.array([0, 0, 0, 1, 0, 0]),
          'NEU' : np.array([0, 0, 0, 0, 1, 0]),
          'SAD' : np.array([0, 0, 0, 0, 0, 1])
         }
```

```
y_train_1d = np.array([LABELS[letter] for letter in y_train_1d])
y_test_1d = np.array([LABELS[letter] for letter in y_test_1d])
y_val_1d = np.array([LABELS[letter] for letter in y_val_1d])
print(y_train_1d.shape)
print(y_test_1d.shape)
print(y_val_1d.shape)
```

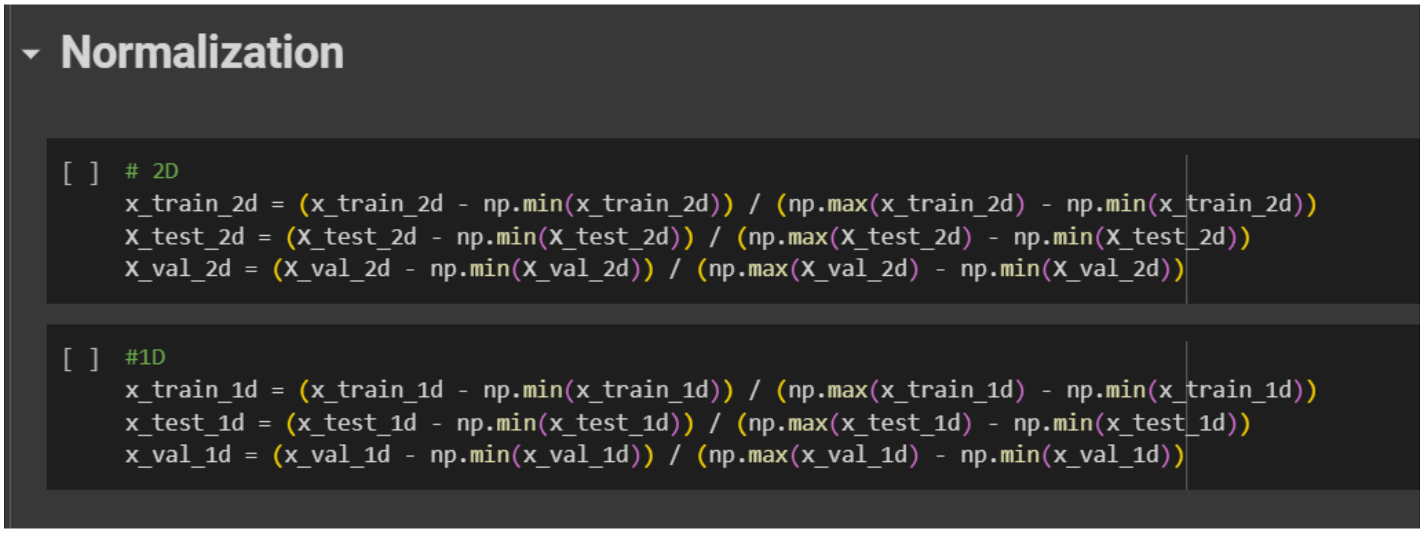
```
(14846, 6)
(6698, 6)
(782, 6)
```

```
y_train_2d = np.array([LABELS[letter] for letter in y_train_2d])
y_test_2d = np.array([LABELS[letter] for letter in y_test_2d])
y_val_2d = np.array([LABELS[letter] for letter in y_val_2d])
print(y_train_2d.shape)
print(y_test_2d.shape)
print(y_val_2d.shape)
```

```
(14846, 6)
(6698, 6)
(782, 6)
```

Then we have made a **normalization** step :

- The code performs data normalization on the input data. The formula used for normalization is (x - np.min(x)) / (np.max(x) - np.min(x)), where x represents the input data.
- The goal of normalization is to scale the input data to a range between 0 and 1. This process can be helpful for training deep learning models because it helps to ensure that the model's weights are updated in a consistent and efficient manner.
- In this case, the normalization is applied separately to the 2D and 1D data arrays, which were previously split into training, validation, and testing sets.



Normalization

```
# 2D
x_train_2d = (x_train_2d - np.min(x_train_2d)) / (np.max(x_train_2d) - np.min(x_train_2d))
x_test_2d = (x_test_2d - np.min(x_test_2d)) / (np.max(x_test_2d) - np.min(x_test_2d))
x_val_2d = (x_val_2d - np.min(x_val_2d)) / (np.max(x_val_2d) - np.min(x_val_2d))
```

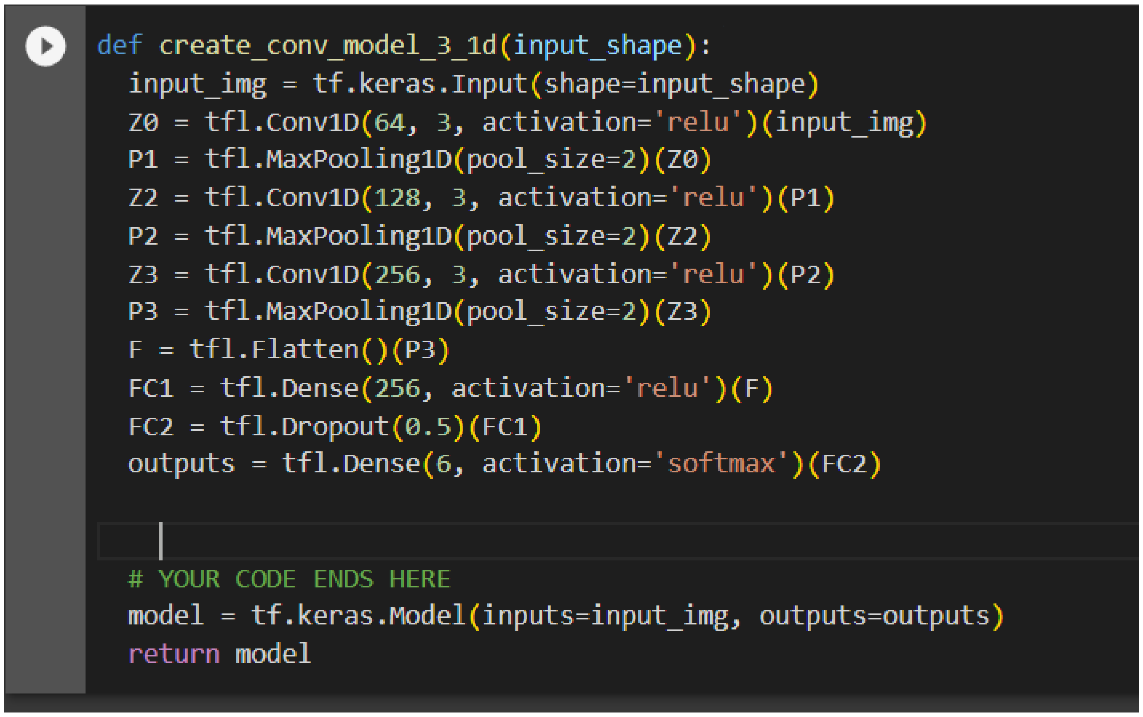
```
#1D
x_train_1d = (x_train_1d - np.min(x_train_1d)) / (np.max(x_train_1d) - np.min(x_train_1d))
x_test_1d = (x_test_1d - np.min(x_test_1d)) / (np.max(x_test_1d) - np.min(x_test_1d))
x_val_1d = (x_val_1d - np.min(x_val_1d)) / (np.max(x_val_1d) - np.min(x_val_1d))
```

CNN model:

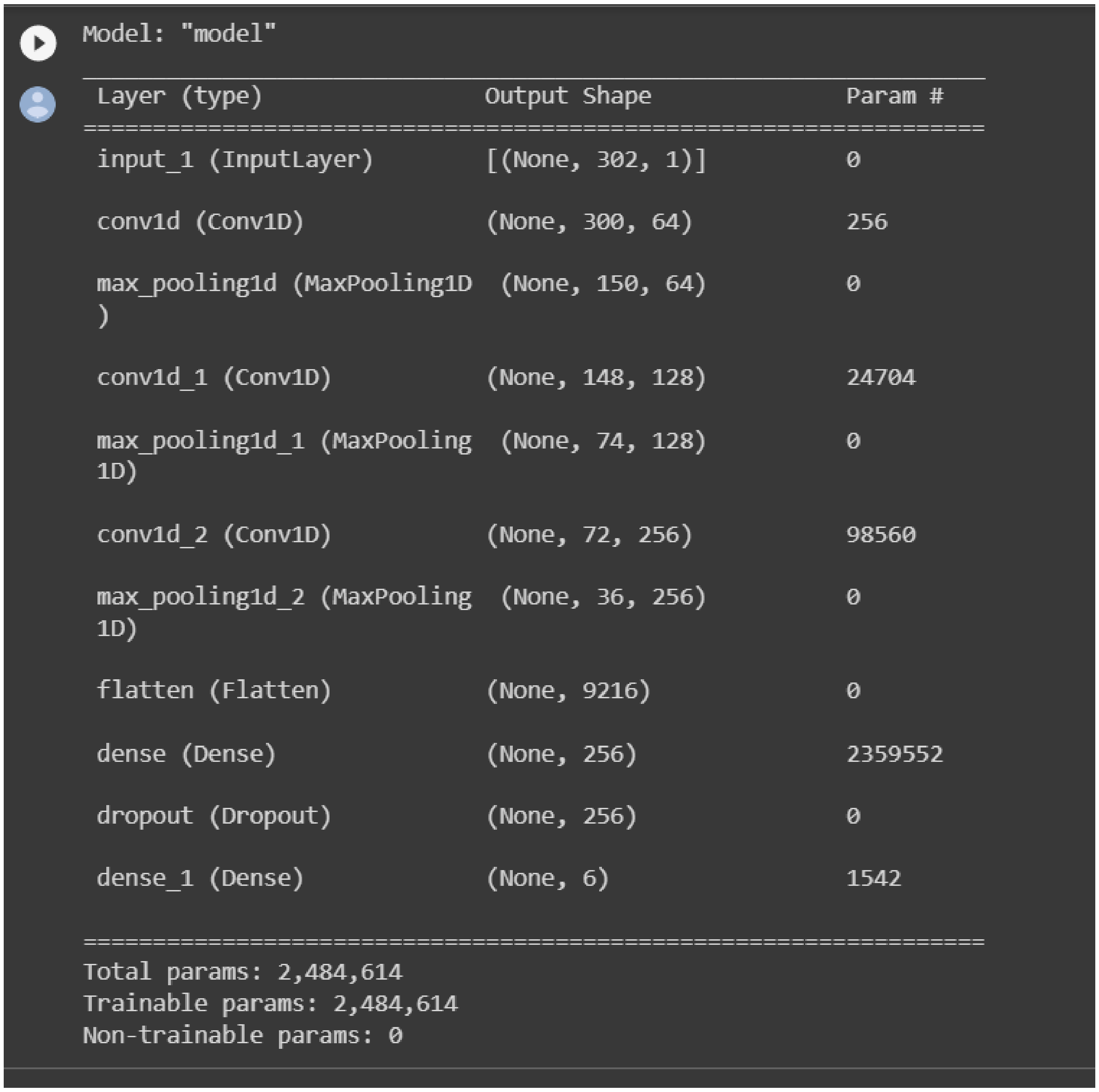
1D model:

The function **“create_conv_model_3_1d”** creates a convolutional neural network (CNN) model for 1D data:

- It applies a 1D convolutional layer with 64 filters, a filter size of 3, and ReLU activation function to the input_img layer.
- Then applies a 1D max-pooling layer with a pool size of 2 to Z0 to reduce its spatial dimensions.
- Then applies another 1D convolutional layer with 128 filters, a filter size of 3, and ReLU activation function to P1.
- Then applies max-pooling to Z2 with a pool size of 2, reducing its spatial dimensions.
- Then applies a third 1D convolutional layer with 256 filters, a filter size of 3, and ReLU activation function to P2.
- Then applies max-pooling to Z3 with a pool size of 2, further reducing its spatial dimensions.
- Flattens the output of the max-pooling layer P3 into a 1D vector.
- Then applies a fully connected (dense) layer with 256 units and ReLU activation function to F. The result is assigned to FC1.
- And applies dropout regularization to the FC1 layer with a rate of 0.5, randomly setting half of the layer's outputs to 0 during training.
- Then applies a final dense layer with 6 units and softmax activation function to FC2. The softmax activation is suitable for multi-class classification tasks.
- Creates a Keras Model object by specifying the input and output layers.
- Finally, the created model is returned as the output of the function.



```
def create_conv_model_3_1d(input_shape):
  input_img = tf.keras.Input(shape=input_shape)
  Z0 = tfl.Conv1D(64, 3, activation='relu')(input_img)
  P1 = tfl.MaxPooling1D(pool_size=2)(Z0)
  Z2 = tfl.Conv1D(128, 3, activation='relu')(P1)
  P2 = tfl.MaxPooling1D(pool_size=2)(Z2)
  Z3 = tfl.Conv1D(256, 3, activation='relu')(P2)
  P3 = tfl.MaxPooling1D(pool_size=2)(Z3)
  F = tfl.Flatten()(P3)
  FC1 = tfl.Dense(256, activation='relu')(F)
  FC2 = tfl.Dropout(0.5)(FC1)
  outputs = tfl.Dense(6, activation='softmax')(FC2)


  # YOUR CODE ENDS HERE
  model = tf.keras.Model(inputs=input_img, outputs=outputs)
  return model
```

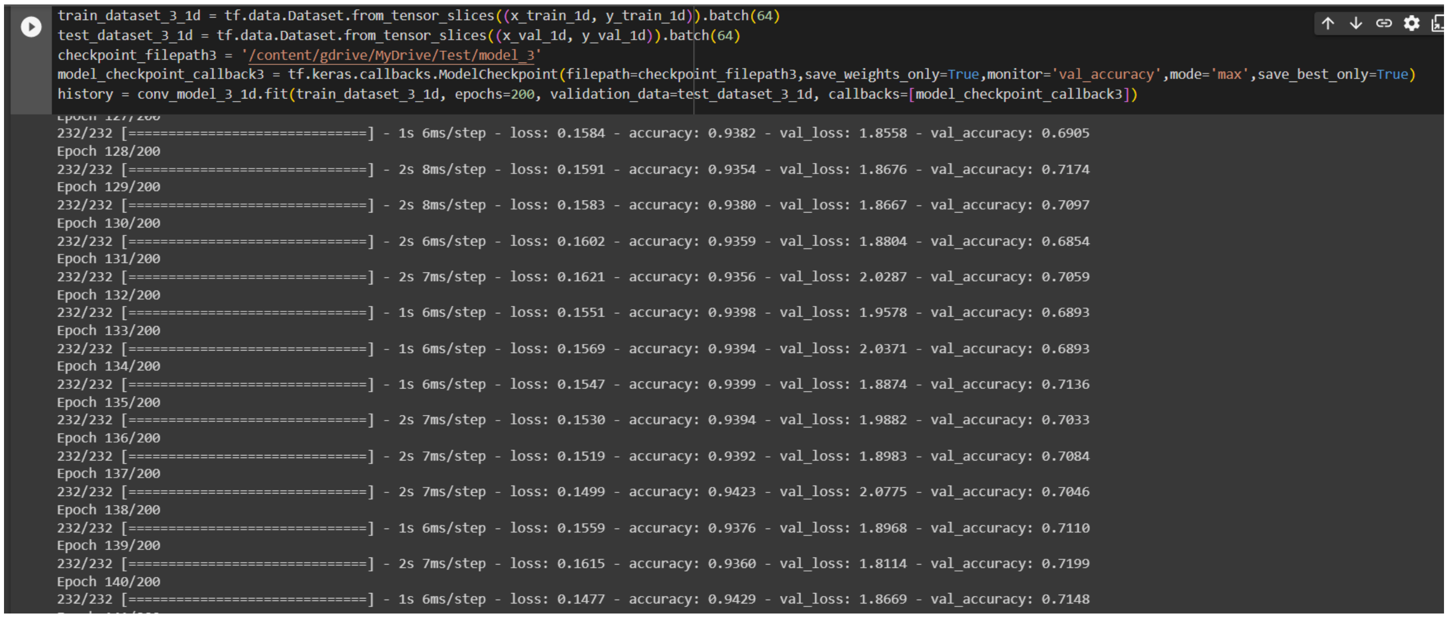


```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 302, 1)]          0

 conv1d (Conv1D)             (None, 300, 64)           256

 max_pooling1d (MaxPooling1D  (None, 150, 64)          0
 )

 conv1d_1 (Conv1D)           (None, 148, 128)          24704

 max_pooling1d_1 (MaxPooling  (None, 74, 128)          0
 1D)

 conv1d_2 (Conv1D)           (None, 72, 256)           98560

 max_pooling1d_2 (MaxPooling  (None, 36, 256)          0
 1D)

 flatten (Flatten)           (None, 9216)              0

 dense (Dense)               (None, 256)               2359552

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 6)                 1542

=================================================================
Total params: 2,484,614
Trainable params: 2,484,614
Non-trainable params: 0
```

After that we create a TensorFlow dataset (train_dataset_3_1d) from the training data (x_train_1d and y_train_1d). The **“from_tensor_slices”** function is used to create a dataset from individual tensors, and the batch function is used to group the data into batches of size 64.
Then creates a TensorFlow dataset (test_dataset_3_1d) from the validation data (x_val_1d and y_val_1d). The data is batched into size 64 as well.

And we specify the filepath where the model checkpoints will be saved during training. Then create a callback (model_checkpoint_callback3) for model checkpointing. The callback saves the weights of the model to the specified filepath (checkpoint_filepath3) whenever the validation accuracy improves.
By setting save_weights_only=True, only the model weights are saved instead of the entire model.

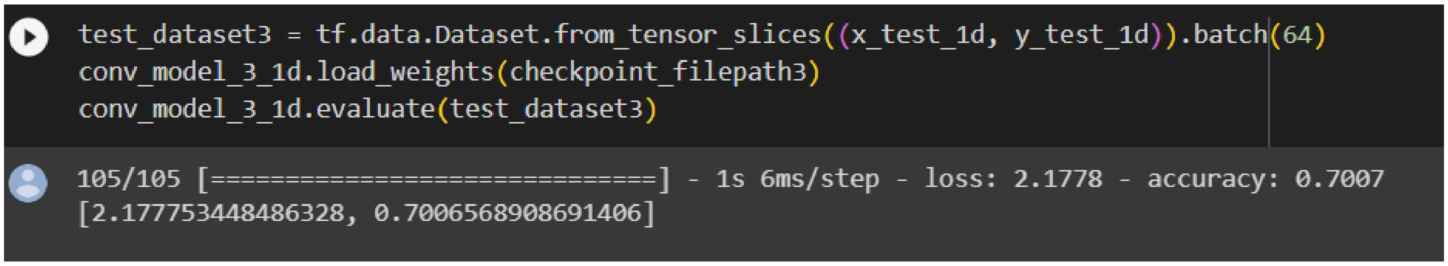
Then trains the model (conv_model_3_1d) using the provided training dataset (train_dataset_3_1d) and validation dataset (test_dataset_3_1d). The model is trained for 200 epochs, and the **ModelCheckpoint** callback (model_checkpoint_callback3) is used during training. The training history is saved to the history variable.



```
train_dataset_3_1d = tf.data.Dataset.from_tensor_slices((x_train_1d, y_train_1d)).batch(64)
test_dataset_3_1d = tf.data.Dataset.from_tensor_slices((x_val_1d, y_val_1d)).batch(64)
checkpoint_filepath3 = '/content/gdrive/MyDrive/Test/model_3'
model_checkpoint_callback3 = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_filepath3,save_weights_only=True,monitor='val_accuracy',mode='max',save_best_only=True)
history = conv_model_3_1d.fit(train_dataset_3_1d, epochs=200, validation_data=test_dataset_3_1d, callbacks=[model_checkpoint_callback3])
```

```
232/232 [==============================] - 1s 6ms/step - loss: 0.1584 - accuracy: 0.9382 - val_loss: 1.8558 - val_accuracy: 0.6905
Epoch 128/200
232/232 [==============================] - 2s 8ms/step - loss: 0.1591 - accuracy: 0.9354 - val_loss: 1.8676 - val_accuracy: 0.7174
Epoch 129/200
232/232 [==============================] - 2s 8ms/step - loss: 0.1583 - accuracy: 0.9380 - val_loss: 1.8667 - val_accuracy: 0.7097
Epoch 130/200
232/232 [==============================] - 2s 6ms/step - loss: 0.1602 - accuracy: 0.9359 - val_loss: 1.8804 - val_accuracy: 0.6854
Epoch 131/200
232/232 [==============================] - 2s 7ms/step - loss: 0.1621 - accuracy: 0.9356 - val_loss: 2.0287 - val_accuracy: 0.7059
Epoch 132/200
232/232 [==============================] - 1s 6ms/step - loss: 0.1551 - accuracy: 0.9398 - val_loss: 1.9578 - val_accuracy: 0.6893
Epoch 133/200
232/232 [==============================] - 1s 6ms/step - loss: 0.1569 - accuracy: 0.9394 - val_loss: 2.0371 - val_accuracy: 0.6893
Epoch 134/200
232/232 [==============================] - 1s 6ms/step - loss: 0.1547 - accuracy: 0.9399 - val_loss: 1.8874 - val_accuracy: 0.7136
Epoch 135/200
232/232 [==============================] - 2s 7ms/step - loss: 0.1530 - accuracy: 0.9394 - val_loss: 1.9882 - val_accuracy: 0.7033
Epoch 136/200
232/232 [==============================] - 2s 7ms/step - loss: 0.1519 - accuracy: 0.9392 - val_loss: 1.8983 - val_accuracy: 0.7084
Epoch 137/200
232/232 [==============================] - 2s 7ms/step - loss: 0.1499 - accuracy: 0.9423 - val_loss: 2.0775 - val_accuracy: 0.7046
Epoch 138/200
232/232 [==============================] - 1s 6ms/step - loss: 0.1559 - accuracy: 0.9376 - val_loss: 1.8968 - val_accuracy: 0.7110
Epoch 139/200
232/232 [==============================] - 2s 7ms/step - loss: 0.1615 - accuracy: 0.9360 - val_loss: 1.8114 - val_accuracy: 0.7199
Epoch 140/200
232/232 [==============================] - 1s 6ms/step - loss: 0.1477 - accuracy: 0.9429 - val_loss: 1.8669 - val_accuracy: 0.7148
```

After executing this code, the model will be trained on the provided datasets, and the weights will be saved to the specified filepath whenever the validation accuracy improves. The training progress and metrics can be accessed from the history object.



```
test_dataset3 = tf.data.Dataset.from_tensor_slices((x_test_1d, y_test_1d)).batch(64)
conv_model_3_1d.load_weights(checkpoint_filepath3)
conv_model_3_1d.evaluate(test_dataset3)
```

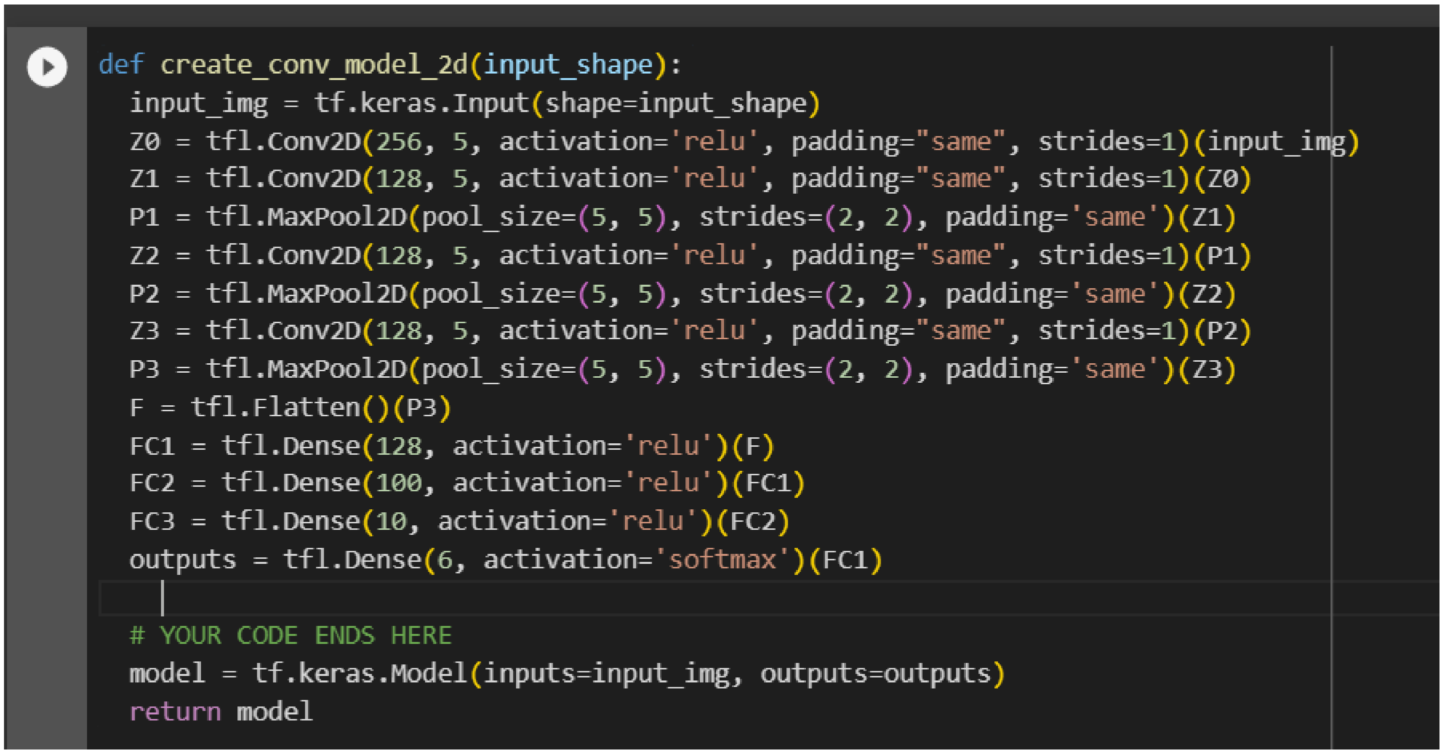
```
105/105 [==============================] - 1s 6ms/step - loss: 2.1778 - accuracy: 0.7007
[2.177753448486328, 0.7006568908691406]
```

2D model:

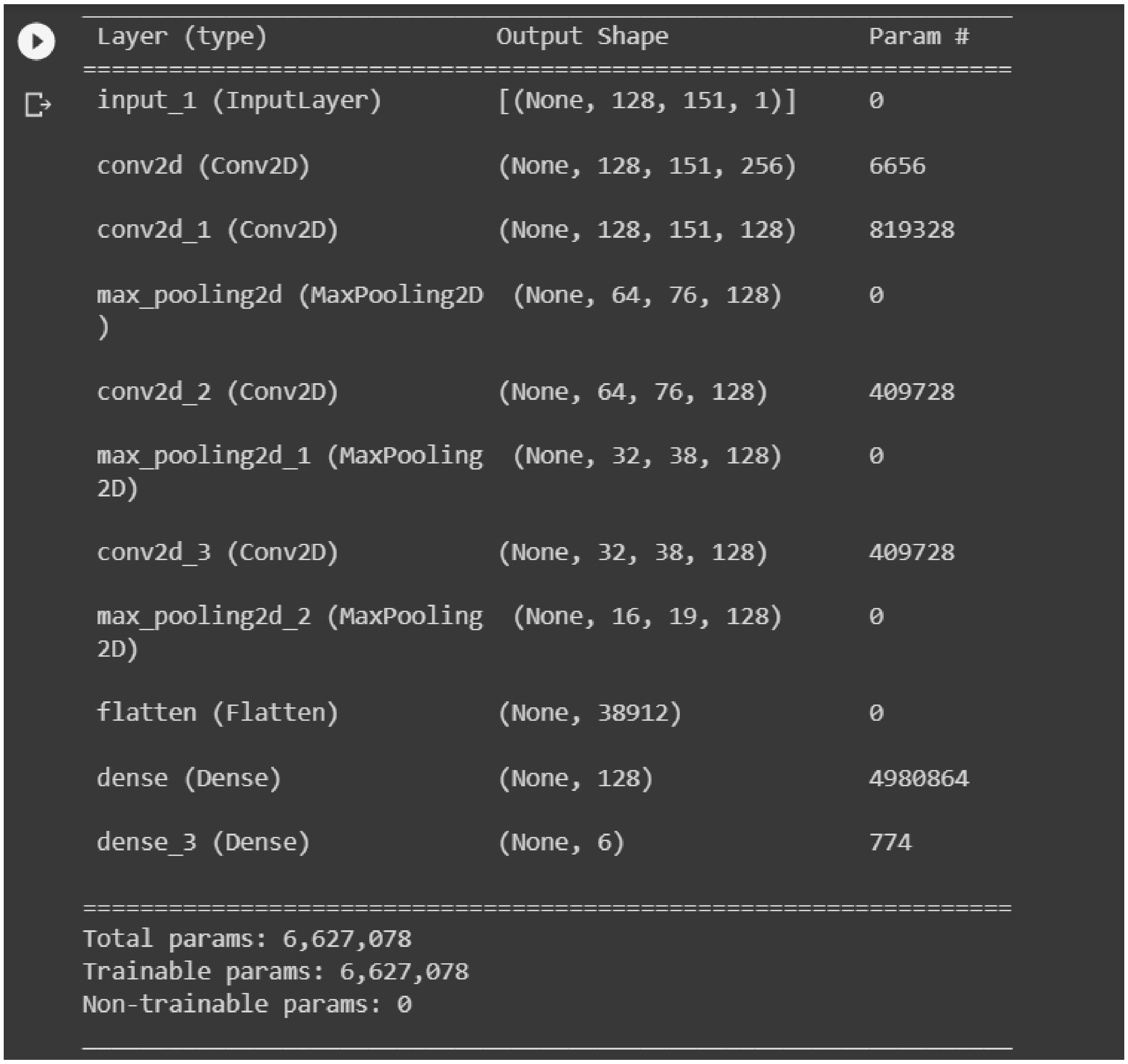
The function **“create_conv_model_2d”** that creates a convolutional neural network (CNN) model for 2D image data:

- The function creates an input layer for the model with the specified input_shape, which should be suitable for 2D image data.
- It applies a 2D convolutional layer with 256 filters, a kernel size of 5x5, ReLU activation function, "same" padding, and stride of 1 to the input_img layer.
- It applies another 2D convolutional layer with 128 filters, a kernel size of 5x5, ReLU activation function, "same" padding, and stride of 1 to Z0.
- Then applies max-pooling to Z1 with a pool size of 5x5, strides of 2x2, and "same" padding.
- The above pattern of applying convolutional layers, followed by max-pooling layers, is repeated for two more blocks.

- Flattens the output of the last max-pooling layer (P3) into a 1D vector.
- It applies a fully connected (dense) layer with 128 units and ReLU activation function to F. The result is assigned to FC1.
- Then applies another fully connected layer with 100 units and ReLU activation function to FC1. The result is assigned to FC2.
- Then applies yet another fully connected layer with 10 units and ReLU activation function to FC2. The result is assigned to FC3.
- Then applies a final dense layer with 6 units and softmax activation function to FC1. The softmax activation is suitable for multi-class classification tasks.
- It creates a Keras Model object by specifying the input and output layers. The input layer is input_img, and the output layer is outputs.
- Finally, the created model is returned as the output of the function.



```
def create_conv_model_2d(input_shape):
  input_img = tf.keras.Input(shape=input_shape)
  Z0 = tfl.Conv2D(256, 5, activation='relu', padding="same", strides=1)(input_img)
  Z1 = tfl.Conv2D(128, 5, activation='relu', padding="same", strides=1)(Z0)
  P1 = tfl.MaxPool2D(pool_size=(5, 5), strides=(2, 2), padding='same')(Z1)
  Z2 = tfl.Conv2D(128, 5, activation='relu', padding="same", strides=1)(P1)
  P2 = tfl.MaxPool2D(pool_size=(5, 5), strides=(2, 2), padding='same')(Z2)
  Z3 = tfl.Conv2D(128, 5, activation='relu', padding="same", strides=1)(P2)
  P3 = tfl.MaxPool2D(pool_size=(5, 5), strides=(2, 2), padding='same')(Z3)
  F = tfl.Flatten()(P3)
  FC1 = tfl.Dense(128, activation='relu')(F)
  FC2 = tfl.Dense(100, activation='relu')(FC1)
  FC3 = tfl.Dense(10, activation='relu')(FC2)
  outputs = tfl.Dense(6, activation='softmax')(FC1)

  # YOUR CODE ENDS HERE
  model = tf.keras.Model(inputs=input_img, outputs=outputs)
  return model
```



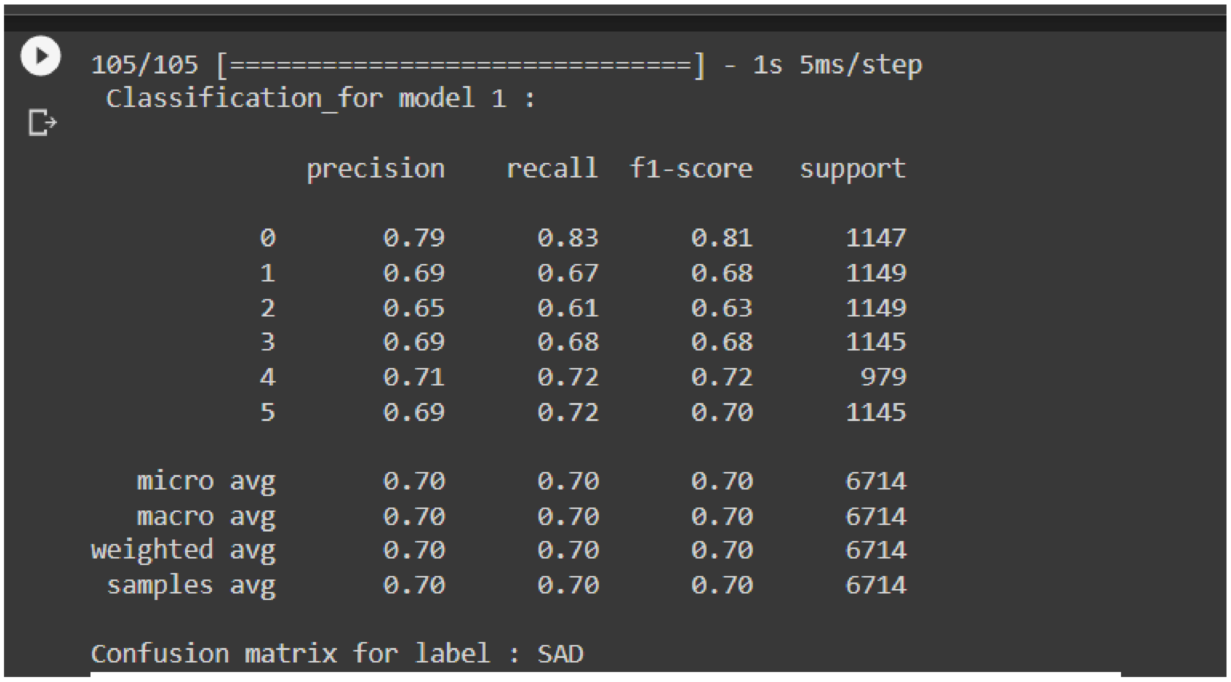
```
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 128, 151, 1)]     0         
                                                                 
 conv2d (Conv2D)             (None, 128, 151, 256)     6656      
                                                                 
 conv2d_1 (Conv2D)           (None, 128, 151, 128)     819328    
                                                                 
 max_pooling2d (MaxPooling2D  (None, 64, 76, 128)      0         
 )                                                               
                                                                 
 conv2d_2 (Conv2D)           (None, 64, 76, 128)       409728    
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 32, 38, 128)      0         
 2D)                                                             
                                                                 
 conv2d_3 (Conv2D)           (None, 32, 38, 128)       409728    
                                                                 
 max_pooling2d_2 (MaxPooling  (None, 16, 19, 128)      0         
 2D)                                                             
                                                                 
 flatten (Flatten)           (None, 38912)             0         
                                                                 
 dense (Dense)               (None, 128)               4980864   
                                                                 
 dense_3 (Dense)             (None, 6)                 774       
                                                                 
=================================================================
Total params: 6,627,078
Trainable params: 6,627,078
Non-trainable params: 0
_________________________________________________________________
```

After that we provide a code to train a model using TensorFlow datasets for 2D image data. Here's a breakdown of the code:

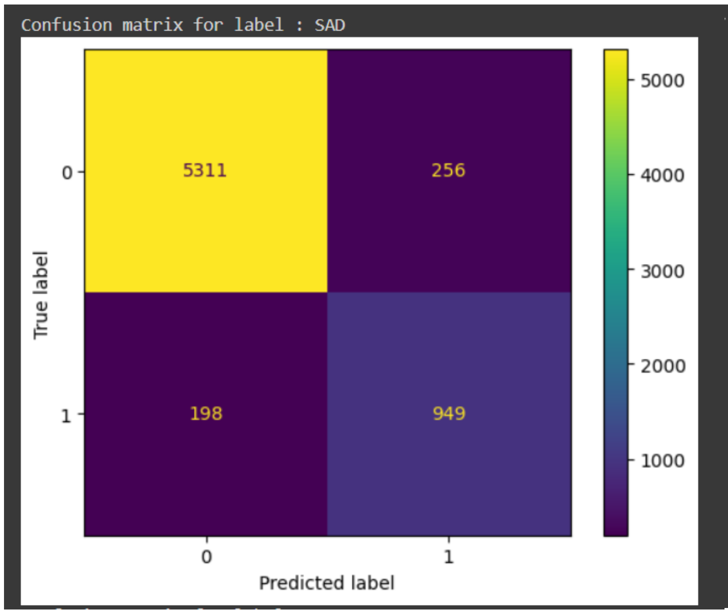
- This line creates a TensorFlow dataset (**train_dataset_2d**) from the training data (**x_train_2d** and **y_train_2d**). The **from_tensor_slices** function is used to create a dataset from individual tensors, and the **batch** function is used to group the data into batches of size 64.
- Similarly, creates a TensorFlow dataset (**test_dataset_2d**) from the validation data (**X_val_2d** and **y_val_2d**). The data is batched into size 64 as well.
- This line creates a callback (**model_checkpoint_callback_2d**) for model checkpointing. The callback saves the weights of the model to the specified filepath (**checkpoint_filepath_2d**) whenever the validation accuracy improves. By setting **save_weights_only=True**, only the model weights are saved instead of the entire model.

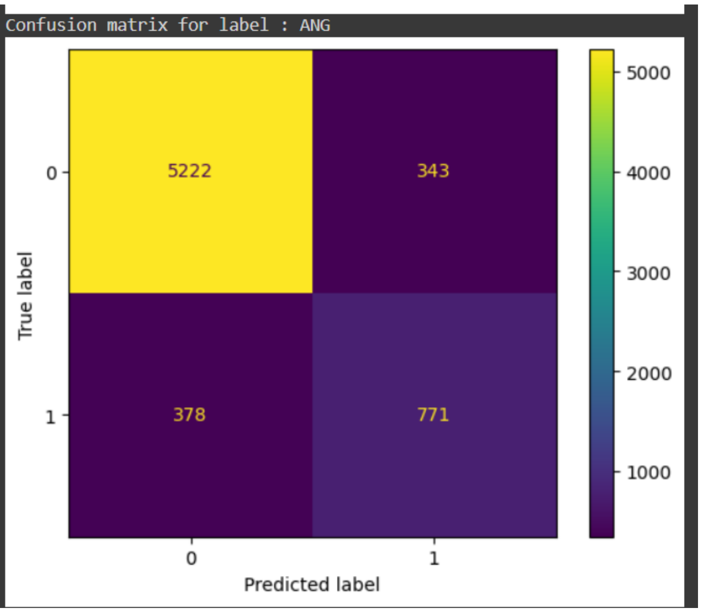
4. Big picture:

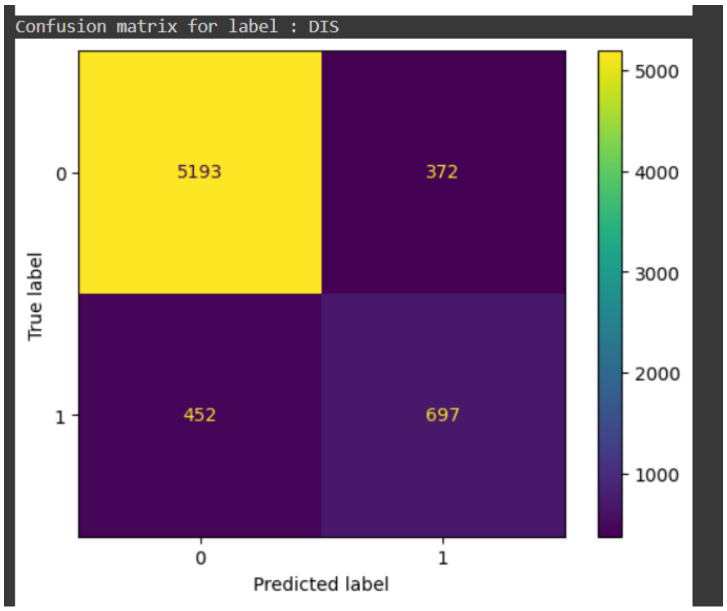
For 1D model:

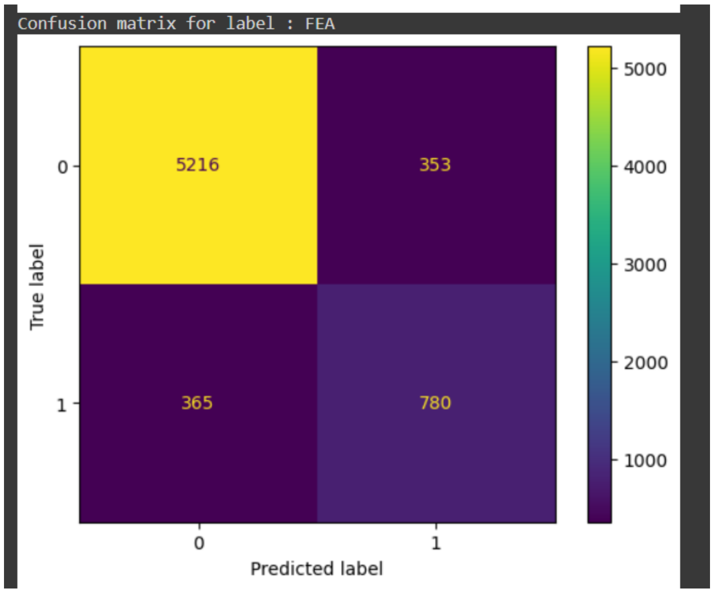


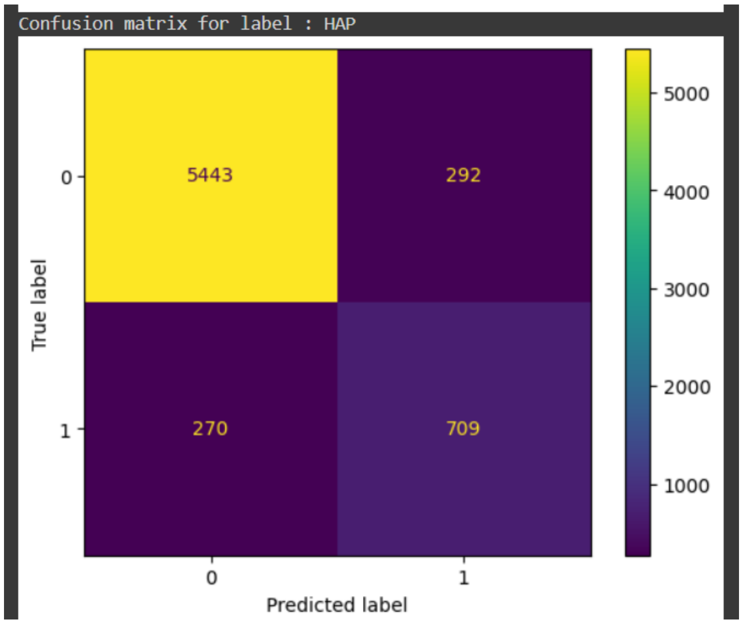
```
105/105 [==============================] - 1s 5ms/step
 Classification_for model 1 :

              precision    recall  f1-score   support

           0       0.79      0.83      0.81      1147
           1       0.69      0.67      0.68      1149
           2       0.65      0.61      0.63      1149
           3       0.69      0.68      0.68      1145
           4       0.71      0.72      0.72       979
           5       0.69      0.72      0.70      1145

   micro avg       0.70      0.70      0.70      6714
   macro avg       0.70      0.70      0.70      6714
weighted avg       0.70      0.70      0.70      6714
 samples avg       0.70      0.70      0.70      6714

Confusion matrix for label : SAD
```

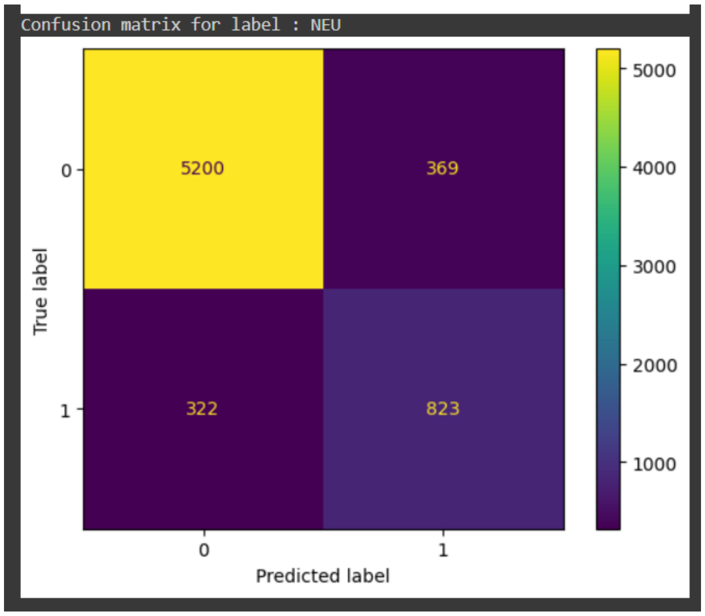








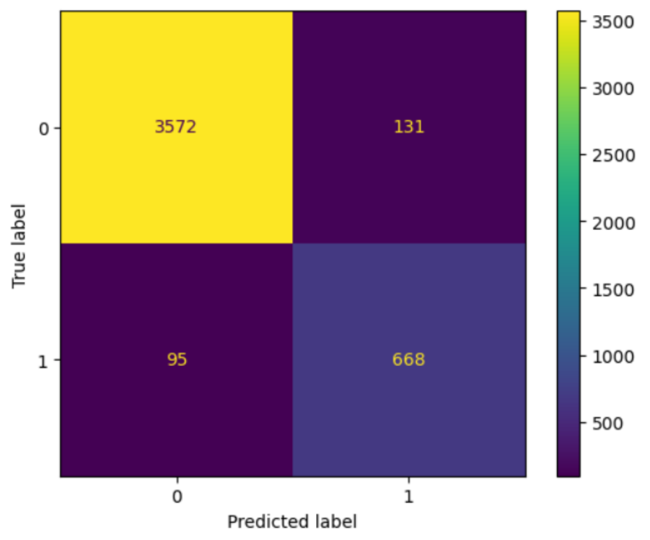




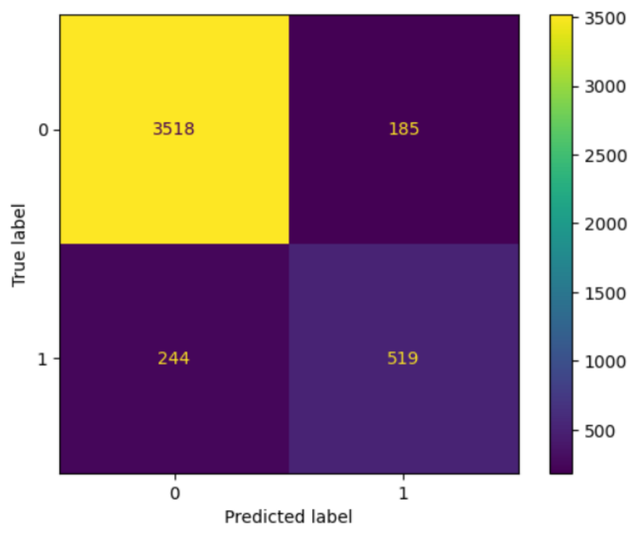
For 2D model:

```
70/70 [==============================] - 11s 159ms/step
 Classification_for model 2 :

              precision    recall  f1-score   support

           0       0.84      0.88      0.86       763
           1       0.74      0.68      0.71       763
           2       0.70      0.71      0.70       762
           3       0.80      0.75      0.78       763
           4       0.75      0.81      0.78       652
           5       0.70      0.70      0.70       763

   micro avg       0.75      0.75      0.75      4466
   macro avg       0.75      0.76      0.75      4466
weighted avg       0.75      0.75      0.75      4466
 samples avg       0.75      0.75      0.75      4466
```

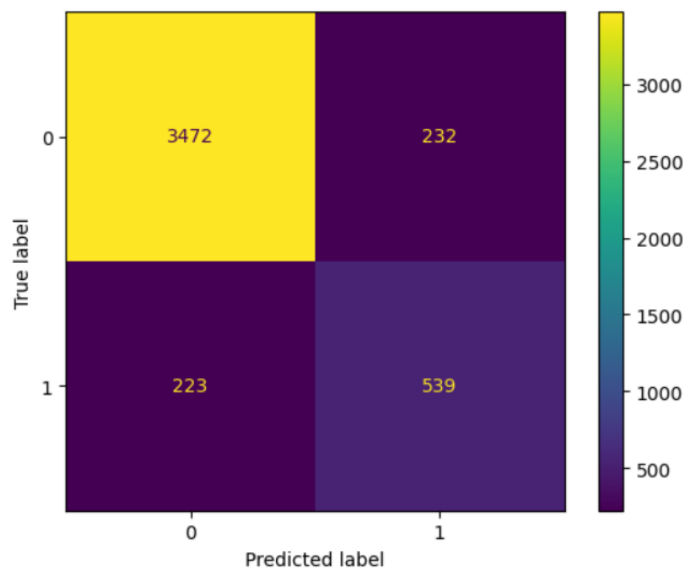
Confusion matrix for label : SAD



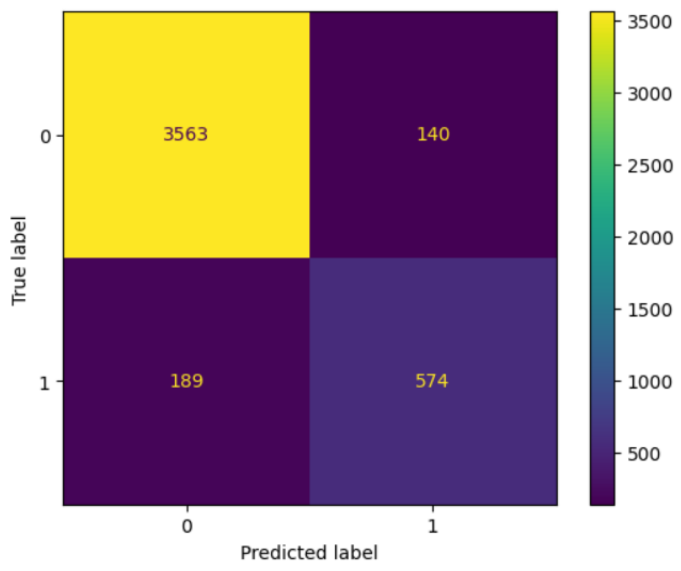
Confusion matrix for label : ANG



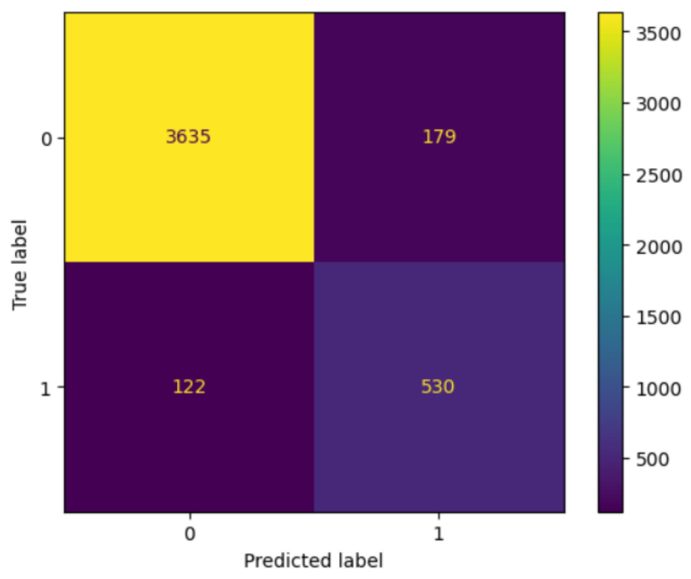
Confusion matrix for label : DIS



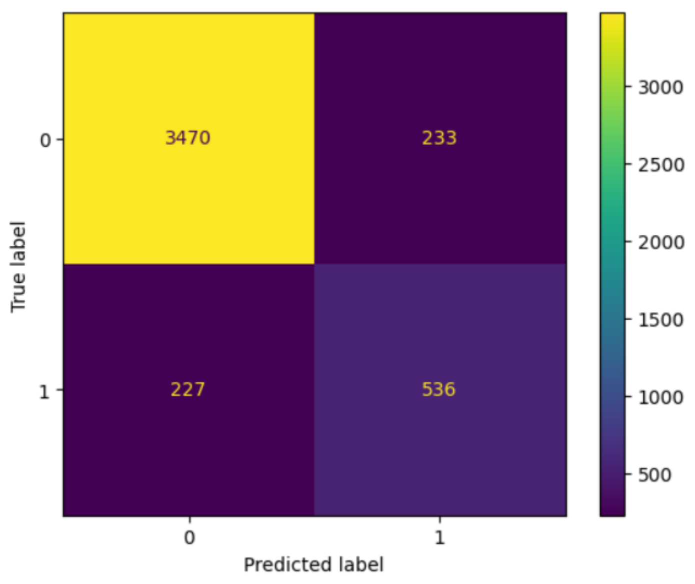
Confusion matrix for label : FEA



Confusion matrix for label : HAP



Confusion matrix for label : NEU



Colab link:
https://colab.research.google.com/drive/1LTaxLK7dqPH-uuvvbCee6FLq95ZIawSq?usp=sharing

Kaggle link:

https://www.kaggle.com/code/sallykamel/notebook5b424d1b15