**#Python script for the CNN model**

import numpy as np

import pandas as pd

import keras

import tensorflow as tf

from scipy.stats import randint, uniform

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, Conv1D, GlobalMaxPooling1D, Dense, Dropout

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

import matplotlib.pyplot as plt

from tensorflow.keras.optimizers import Adam

from scikeras.wrappers import KerasClassifier

from tensorflow.keras.optimizers import SGD

import tensorflow\_addons as tfa

from tensorflow.keras.callbacks import EarlyStopping

from sklearn.metrics import accuracy\_score, precision\_score, f1\_score, roc\_auc\_score, confusion\_matrix, recall\_score, classification\_report

from tensorflow.keras.callbacks import Callback

# Convert y\_train\_mapped, y\_val\_mapped, and y\_test\_mapped to one-hot encoded format

num\_classes = 3

y\_train\_encoded = to\_categorical(y\_train\_mapped, num\_classes=num\_classes)

y\_val\_encoded = to\_categorical(y\_val\_mapped, num\_classes=num\_classes)

y\_test\_encoded = to\_categorical(y\_test\_mapped, num\_classes=num\_classes)

vocab\_size = 100

# Initialize and fit the tokenizer on x\_train

tokenizer = Tokenizer(num\_words=vocab\_size)

tokenizer.fit\_on\_texts(x\_train)

# Tokenize x\_train, x\_val, and x\_test

x\_train\_sequences = tokenizer.texts\_to\_sequences(x\_train)

x\_val\_sequences = tokenizer.texts\_to\_sequences(x\_val)

x\_test\_sequences = tokenizer.texts\_to\_sequences(x\_test)

# Convert sequences to NumPy arrays

x\_train\_np = np.array(x\_train\_sequences)

x\_val\_np = np.array(x\_val\_sequences)

x\_test\_np = np.array(x\_test\_sequences)

# Pad sequences

seq\_length = 60

x\_train\_padded = pad\_sequences(x\_train\_sequences, maxlen=seq\_length, padding='post')

x\_val\_padded = pad\_sequences(x\_val\_sequences, maxlen=seq\_length, padding='post')

x\_test\_padded = pad\_sequences(x\_test\_sequences, maxlen=seq\_length, padding='post')

# Convert sequences to NumPy arrays

x\_train\_np = np.array(x\_train\_padded)

x\_val\_np = np.array(x\_val\_padded)

x\_test\_np = np.array(x\_test\_padded)

model = Sequential()

model.add(Embedding(vocab\_size, 60, input\_length = seq\_length))

model.add(Conv1D(100, 5, activation = 'relu'))

model.add(GlobalMaxPooling1D())

model.add(Dense(50, activation = 'relu'))

model.add(Dropout(0.2))

model.add(Dense(len(label\_mapping), activation = 'softmax'))

early\_stopping = EarlyStopping(monitor = 'val\_loss', patience = 200, restore\_best\_weights = True)

# Define the learning rate

learning\_rate = 2e-6

# Create an optimizer with the desired learning rate

optimizer = Adam(learning\_rate=learning\_rate)

model.compile(optimizer = optimizer, loss = 'categorical\_crossentropy', metrics=['accuracy'])

model.summary()

class MetricsCallback(Callback):

def \_\_init\_\_(self, validation\_data):

super(MetricsCallback, self).\_\_init\_\_()

self.validation\_data = validation\_data

self.best\_f1 = 0

self.best\_auc\_roc = 0

def on\_epoch\_end(self, epoch, logs=None):

x\_val, y\_val = self.validation\_data

y\_val\_pred = self.model.predict(x\_val)

# Calculate F1 Score

f1 = f1\_score(y\_val.argmax(axis=1), y\_val\_pred.argmax(axis=1), average='weighted')

print(f"Epoch {epoch + 1} - Training F1 Score: {f1}")

# Calculate AUC-ROC

auc\_roc = roc\_auc\_score(y\_val, y\_val\_pred, multi\_class='ovr')

print(f"Epoch {epoch + 1} - Training AUC-ROC: {auc\_roc}")

# Update best F1 score and AUC-ROC

if f1 > self.best\_f1:

self.best\_f1 = f1

if auc\_roc > self.best\_auc\_roc:

self.best\_auc\_roc = auc\_roc

metrics\_callback = MetricsCallback(validation\_data=(x\_val\_np, y\_val\_encoded))

history = model.fit(x\_train\_np,

y\_train\_encoded,

epochs = 1000,

batch\_size = 30,

validation\_data = (x\_val\_np, y\_val\_encoded),

callbacks=[early\_stopping, metrics\_callback])

best\_val\_loss = min(history.history['val\_loss'])

print(f"Best Validation Loss: {best\_val\_loss}")

best\_val\_accuracy = min(history.history['val\_accuracy'])

print(f"Best Validation Accuracy: {best\_val\_accuracy}")

# Access the best F1 score and AUC-ROC through the same instance of MetricsCallback used during training

print(f"Best Training F1 Score: {metrics\_callback.best\_f1}")

print(f"Best Training AUC-ROC: {metrics\_callback.best\_auc\_roc}")

# Plot training loss

plt.figure(figsize=(10, 6))

plt.plot(history.history['loss'], label='Training Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.show

# validation loss

plt.figure(figsize=(10, 6))

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.show

# Predict on validation data

y\_test\_pred = model.predict(x\_test\_np)

# Convert one-hot encoded labels back to integers (if needed)

y\_test\_true = np.argmax(y\_test\_encoded, axis=1)

y\_test\_pred\_classes = np.argmax(y\_test\_pred, axis=1)

# Calculate accuracy

accuracy = accuracy\_score(y\_test\_true, y\_test\_pred\_classes)

print(f'Accuracy: {accuracy}')

# Calculate AUC-ROC score using macro averaging

roc\_auc\_macro = roc\_auc\_score(y\_test\_encoded, y\_test\_pred, average='macro')

print(f'AUC-ROC (macro): {roc\_auc\_macro}')

# Calculate precision, recall, and F1 score

classification\_rep = classification\_report(y\_test\_true, y\_test\_pred\_classes, target\_names=["neutral", "negative", "positive"])

print("Classification Report:\n", classification\_rep)

label\_mapping\_after\_training = {0: 'Neutral', 1: 'Negative', 2: 'Positive'}

y\_test\_pred = model.predict(x\_test\_np)

# Calculate the ROC curve and AUC for each class (for multi-class AUC-ROC)

fpr = dict()

tpr = dict()

roc\_auc = dict()

for i in range(len(label\_mapping\_after\_training)):

fpr[i], tpr[i], \_ = roc\_curve(y\_test\_encoded[:, i], y\_test\_pred[:, i])

roc\_auc[i] = auc(fpr[i], tpr[i])

print(f"AUC-ROC for {label\_mapping\_after\_training[i]}: {roc\_auc[i]:.2f}")

# Calculate macro-average AUC by averaging AUC scores for individual classes

all\_fpr = np.unique(np.concatenate([fpr[i] for i in range(len(label\_mapping\_after\_training))]))

mean\_tpr = np.zeros\_like(all\_fpr)

for i in range(len(label\_mapping\_after\_training)):

mean\_tpr += np.interp(all\_fpr, fpr[i], tpr[i])

mean\_tpr /= len(label\_mapping\_after\_training)

roc\_auc["macro"] = auc(all\_fpr, mean\_tpr)

print(f"Macro-average AUC-ROC: {roc\_auc['macro']:.2f}")

# Plot the ROC curves

plt.figure(figsize=(8, 6))

for i in range(len(label\_mapping\_after\_training)):

plt.plot(fpr[i],

tpr[i], label=f"ROC curve for {label\_mapping\_after\_training[i]} (AUC = {roc\_auc[i]:.2f})")

# Plot macro-average ROC curve

plt.plot(all\_fpr, mean\_tpr,

label=f"Macro-average ROC curve (AUC = {roc\_auc['macro']:.2f})", color='deeppink',

linestyle=':')

plt.plot([0, 1], [0, 1], color='navy', linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.legend(loc="lower right", fontsize="small")

plt.grid(False)

plt.show()