**#Python script for the Transformer model**

%pip install transformers

%pip install torchinfo

%pip install transformers[torch] accelerate

%pip install loralib

from sklearn.model\_selection import StratifiedKFold, cross\_val\_score

import loralib as lora

from transformers import pipeline

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

import torch

from torchinfo import summary

from torch.utils.data import Dataset, DataLoader, TensorDataset, random\_split

from transformers import AutoModelForSequenceClassification, Trainer, TrainingArguments, DistilBertForSequenceClassification, get\_linear\_schedule\_with\_warmup

import torch.nn as nn

import torch.nn.functional as F

from transformers import AutoTokenizer, AdamW, TrainingArguments, Trainer

from sklearn.metrics import roc\_auc\_score, f1\_score, roc\_curve, auc, confusion\_matrix, accuracy\_score, make\_scorer

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

from transformers import DistilBertForSequenceClassification, DistilBertTokenizer

# create pipeline for sentiment analysis

classification = pipeline('sentiment-analysis')

print(type(classification))

# Load cleaned data for training

data = pd.read\_excel("/content/final ml dataframe.xlsx")

df = pd.DataFrame(data)

x = df['clean\_tokens2'].tolist()

y = df['target\_labels'].tolist()

label\_mapping = {"positive": 2, "negative": 1, "neutral": 0}

encoded\_labels = [label\_mapping[label] for label in y]

tokenizer = DistilBertTokenizer.from\_pretrained('distilbert-base-uncased')

tokenized\_data = tokenizer(list(x), truncation=True, padding=True, return\_tensors="pt")

x\_temp, x\_test, y\_temp, y\_test = train\_test\_split(tokenized\_data["input\_ids"], encoded\_labels, test\_size = 0.2, stratify = y, random\_state = 50)

x\_train, x\_val, y\_train, y\_val = train\_test\_split(x\_temp, y\_temp, test\_size = 0.5, stratify = y\_temp, random\_state = 50)

input\_ids\_list = tokenized\_data["input\_ids"]

attention\_mask\_list = tokenized\_data["attention\_mask"]

# Create a list of dictionaries

encoded\_inputs = [{'input\_ids': input\_ids, 'attention\_mask': attention\_mask} for input\_ids, attention\_mask in zip(input\_ids\_list, attention\_mask\_list)]

# Define device

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

num\_labels = 3

dropout\_rate = 0.3

# Load the pre-trained DistilBERT model with the specified number of labels

model = DistilBertForSequenceClassification.from\_pretrained('distilbert-base-uncased', num\_labels=num\_labels)

model.to(device) # Move model to the device

# Define a new classification head with a softmax activation

classification\_head = nn.Sequential(

lora.Linear(model.config.hidden\_size, num\_labels, r=16),

nn.ReLU(),

nn.Dropout(dropout\_rate) # Add dropout layer

)

classification\_head.to(device) # Move classification head to the same device

n\_splits = 10

cv = StratifiedKFold(n\_splits=n\_splits, shuffle=True, random\_state=50)

model.classifier = classification\_head # Replace the existing classification head with the new one

class EarlyStopping:

def \_\_init\_\_(self, patience=10, verbose=True, delta=0, path='checkpoint.pt', trace\_func=print):

self.patience = patience

self.verbose = verbose

self.counter = 0

self.best\_score = None

self.early\_stop = False

self.val\_loss\_min = np.Inf

self.delta = delta

self.path = path

self.trace\_func = trace\_func

def \_\_call\_\_(self, val\_loss, model):

score = -val\_loss

if self.best\_score is None:

self.best\_score = score

self.save\_checkpoint(val\_loss, model)

elif score < self.best\_score + self.delta:

self.counter += 1

self.trace\_func(f'EarlyStopping counter: {self.counter} out of {self.patience}')

if self.counter >= self.patience:

self.early\_stop = True

else:

self.best\_score = score

self.save\_checkpoint(val\_loss, model)

self.counter = 0

def save\_checkpoint(self, val\_loss, model):

if self.verbose:

self.trace\_func(f'Validation loss decreased ({self.val\_loss\_min:.6f} --> {val\_loss:.6f}). Saving model...')

torch.save(lora.lora\_state\_dict(model), self.path)

self.val\_loss\_min = val\_loss

# Define early stopping parameters

patience = 20

best\_metric = float("-inf")

# Define training hyperparameters

num\_epochs = 1000

batch\_size = 40

learning\_rate = 0.55e-2

class\_counts = torch.bincount(torch.tensor(y\_train))

total\_samples = len(y\_train)

class\_weights = total\_samples / (len(class\_counts) \* class\_counts.float())

class\_weights = class\_weights.to(device) # Move weights to device

y\_train = np.array(y\_train)

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

# Initialize lists to store cross-validation results

cross\_val\_scores = []

# Initialize best model state dictionary

best\_model\_state\_dict = None

best\_learning\_rate = None

lora.mark\_only\_lora\_as\_trainable(model)

#SentimentDataset is a custom dataset class

class SentimentDataset(Dataset):

def \_\_init\_\_(self, texts, attention\_masks, labels):

self.texts = texts

self.attention\_masks = attention\_masks

self.labels = labels

def \_\_len\_\_(self):

return len(self.labels)

def \_\_getitem\_\_(self, idx):

return {

'input\_ids': torch.tensor(self.texts[idx], dtype=torch.long).to(device),

'attention\_mask': torch.tensor(self.attention\_masks[idx], dtype=torch.long).to(device),

'labels': torch.tensor(self.labels[idx], dtype=torch.long).to(device)

}

for fold, (train\_idx, val\_idx) in enumerate(cv.split(x\_train, y\_train)):

print(f"Fold {fold + 1}/{n\_splits}")

# Split the data into training and validation sets for this fold

x\_train\_fold, x\_val\_fold = x\_train[train\_idx], x\_train[val\_idx]

y\_train\_fold, y\_val\_fold = y\_train[train\_idx], y\_train[val\_idx]

# Assuming attention\_mask\_list is defined and contains attention masks for your dataset

train\_dataset = SentimentDataset(x\_train\_fold, attention\_mask\_list[train\_idx], y\_train\_fold)

val\_dataset = SentimentDataset(x\_val\_fold, attention\_mask\_list[val\_idx], y\_val\_fold)

train\_dataloader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_dataloader = DataLoader(val\_dataset, batch\_size=batch\_size)

optimizer = AdamW(model.parameters(), lr=learning\_rate)

loss\_fn = nn.CrossEntropyLoss(weight=class\_weights)

early\_stopping = EarlyStopping(patience=patience, verbose=True)

# Define the number of training steps

total\_steps = len(train\_dataloader) \* num\_epochs

# Define the scheduler

scheduler = get\_linear\_schedule\_with\_warmup(optimizer,

num\_warmup\_steps=0,

num\_training\_steps=total\_steps)

# Initialize lists to store training and validation losses

train\_losses = []

val\_losses = []

for epoch in range(num\_epochs):

model.train()

total\_train\_loss = 0 # Rename total\_loss to avoid confusion with validation loss

for batch in train\_dataloader:

input\_ids = batch["input\_ids"].to(device)

attention\_mask = batch["attention\_mask"].to(device)

labels = batch["labels"].to(device)

optimizer.zero\_grad()

outputs = model(input\_ids, attention\_mask=attention\_mask, labels=labels)

loss = outputs.loss

total\_train\_loss += loss.item()

loss.backward()

optimizer.step()

scheduler.step() # Update the learning rate

avg\_train\_loss = total\_train\_loss / len(train\_dataloader)

train\_losses.append(avg\_train\_loss) # Append the average training loss for this epoch

# Validation

model.eval()

total\_val\_loss = 0

val\_correct = 0 # Initialize val\_correct

val\_total = 0 # Initialize val\_total

with torch.no\_grad():

for batch in val\_dataloader:

input\_ids = batch["input\_ids"].to(device)

attention\_mask = batch["attention\_mask"].to(device)

labels = batch["labels"].to(device)

outputs = model(input\_ids, attention\_mask=attention\_mask)

logits = outputs.logits

loss = loss\_fn(logits, labels)

total\_val\_loss += loss.item()

\_, predicted = torch.max(logits, 1)

val\_total += labels.size(0)

val\_correct += (predicted == labels).sum().item()

avg\_val\_loss = total\_val\_loss / len(val\_dataloader)

val\_losses.append(avg\_val\_loss) # Append the average validation loss for this epoch

val\_accuracy = val\_correct / val\_total if val\_total != 0 else 0

if val\_accuracy > best\_metric:

best\_metric = val\_accuracy

best\_model\_state\_dict = model.state\_dict()

early\_stopping(avg\_val\_loss, model)

if early\_stopping.early\_stop:

print("Early stopping")

break

print(f"Epoch {epoch + 1}/{num\_epochs}")

print(f"Train Loss: {avg\_train\_loss:.4f} | Val Loss: {avg\_val\_loss:.4f} | Val Acc: {val\_accuracy:.4f}")

cross\_val\_scores.append(val\_accuracy)

# Restore the best model state

if best\_model\_state\_dict is not None:

model.load\_state\_dict(best\_model\_state\_dict)

# Plot the training and validation losses

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

plt.plot(train\_losses, label='Training Loss')

plt.plot(val\_losses, label='Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.title('Training and Validation Loss Over Epochs')

plt.legend()

plt.show()

# Print the final learning rate that resulted in the best model

if best\_learning\_rate is not None:

print(f"The final learning rate for the best model is: {best\_learning\_rate}")

summary(model)

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

attention\_mask\_list\_test = np.array(attention\_mask\_list[len(x\_test):])

# Create test dataset and dataloader

test\_dataset = SentimentDataset(x\_test, attention\_mask\_list\_test, y\_test)

test\_dataloader = DataLoader(test\_dataset, batch\_size=batch\_size)

# Evaluation on the test data

model.eval() # Set the model to evaluation mode

total\_test\_loss = 0

test\_correct = 0

test\_total = 0

with torch.no\_grad():

for batch in test\_dataloader:

input\_ids = batch["input\_ids"].to(device)

attention\_mask = batch["attention\_mask"].to(device)

labels = batch["labels"].to(device)

outputs = model(input\_ids, attention\_mask=attention\_mask)

logits = outputs.logits

loss = loss\_fn(logits, labels)

total\_test\_loss += loss.item()

\_, predicted = torch.max(logits, 1)

test\_total += labels.size(0)

test\_correct += (predicted == labels).sum().item()

avg\_test\_loss = total\_test\_loss / len(test\_dataloader)

test\_accuracy = test\_correct / test\_total if test\_total != 0 else 0

print(f"Test Loss: {avg\_test\_loss:.4f} | Test Accuracy: {test\_accuracy:.4f}")

# Plot the training and validation losses

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

plt.plot(train\_losses, label='Training Loss')

plt.plot(val\_losses, label='Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.title('Training and Validation Loss Over Epochs')

plt.legend()

plt.show()

from sklearn.metrics import ConfusionMatrixDisplay, accuracy\_score, f1\_score, confusion\_matrix, roc\_auc\_score, precision\_score

import torch

import numpy as np

# Collect predictions and true labels

all\_labels = []

all\_predictions = []

all\_probs = []

model.eval()

with torch.no\_grad():

for batch in test\_dataloader:

input\_ids = batch["input\_ids"].to(device)

attention\_mask = batch["attention\_mask"].to(device)

labels = batch["labels"].to(device)

outputs = model(input\_ids, attention\_mask=attention\_mask)

logits = outputs.logits

probs = torch.softmax(logits, dim=1) # Ensure this is a 2D tensor

\_, predicted = torch.max(logits, 1)

all\_labels.extend(labels.cpu().numpy())

all\_predictions.extend(predicted.cpu().numpy())

all\_probs.extend(probs.cpu().numpy()) # Collect probabilities for all classes

# Convert to numpy arrays

all\_labels = np.array(all\_labels)

all\_predictions = np.array(all\_predictions)

all\_probs = np.array(all\_probs)

# Check the shape of all\_probs

print("Shape of all\_probs:", all\_probs.shape)

# Calculate metrics

accuracy = accuracy\_score(all\_labels, all\_predictions)

f1 = f1\_score(all\_labels, all\_predictions, average='macro') # Use 'macro' for multi-class

conf\_matrix = confusion\_matrix(all\_labels, all\_predictions)

roc\_auc = roc\_auc\_score(all\_labels, all\_probs, multi\_class='ovr') # Specify multi\_class parameter

precision = precision\_score(all\_labels, all\_predictions, average='macro') # Use 'macro' for multi-class

print(f"Accuracy: {accuracy:.5f}")

print(f"F1 Score: {f1:.5f}")

print(f"Confusion Matrix:\n{conf\_matrix}")

print(f"ROC-AUC: {roc\_auc:.5f}")

print(f"Precision: {precision:.5f}")

n\_classes = len(np.unique(all\_labels))

y\_true\_bin = label\_binarize(all\_labels, classes=np.arange(n\_classes))

y\_score\_bin = all\_probs

# Mapping of labels to class names

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

# Compute ROC curve and ROC AUC for each class

fpr = {}

tpr = {}

roc\_auc = {}

for i in range(n\_classes):

fpr[i], tpr[i], \_ = roc\_curve(y\_true\_bin[:, i], y\_score\_bin[:, i])

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

print(f"AUC-ROC for {label\_mapping\_after\_training[i]}: {roc\_auc[i]:.5f}")

# Compute macro-average ROC curve and ROC AUC

all\_fpr = np.linspace(0, 1, 100)

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

for i in range(n\_classes):

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

mean\_tpr /= n\_classes

fpr["macro"] = all\_fpr

tpr["macro"] = mean\_tpr

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

# Plot all ROC curves

plt.figure(figsize=(10, 8))

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

for i in range(n\_classes):

plt.plot(fpr[i], tpr[i], label=f'ROC curve for {label\_mapping\_after\_training[i]} (area = {roc\_auc[i]:.5f})')

plt.plot(fpr["macro"], tpr["macro"], label=f'Macro-average ROC curve (area = {roc\_auc["macro"]:.5f})', color='red', 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='best')

plt.grid(False)

plt.show()

#Extract class labels

labels = sorted(set(all\_labels))

stacked\_cm = confusion\_matrix(y\_test, all\_predictions)

print("Confusion Matrix:\n", stacked\_cm)

disp = ConfusionMatrixDisplay(confusion\_matrix=stacked\_cm, display\_labels=labels)

# Plot confusion matrix

fig, ax = plt.subplots(figsize=(8, 8)) # You can adjust the figure size if needed

disp.plot(cmap='Blues', ax=ax)

# Set font properties

plt.rcParams['font.family'] = 'Serif'

plt.rcParams['font.size'] = 12

# Update labels with Arial font and font size 12

ax.set\_xlabel('Predicted labels', fontsize=12, fontname='Serif')

ax.set\_ylabel('True labels', fontsize=12, fontname='Serif')

ax.set\_title('Confusion Matrix', fontsize=12, fontname='Serif')

# Show plot

plt.show()