**#Python code for the Multinomial Naïve Bayes model**

%pip install scikit-optimize

import numpy as np

import pandas as pd

from sklearn.naive\_bayes import MultinomialNB

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

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

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.feature\_selection import RFE

from sklearn.preprocessing import label\_binarize

from scipy.stats import randint

from gensim.models import Word2Vec

from matplotlib import pyplot as plt

from imblearn.over\_sampling import SMOTE

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.utils import class\_weight

import spacy

from tqdm import tqdm

import re

import time

import pickle

tfidf\_vectorizer = TfidfVectorizer(max\_features=50, ngram\_range=(1, 2))

x\_train\_tfidf = tfidf\_vectorizer.fit\_transform(x\_train)

x\_val\_tfidf = tfidf\_vectorizer.transform(x\_val)

x\_test\_tfidf = tfidf\_vectorizer.transform(x\_test)

smote = SMOTE(random\_state=50)

x\_train\_resampled, y\_train\_resampled = smote.fit\_resample(x\_train\_tfidf, y\_train\_mapped)

x\_test\_resampled, y\_test\_resampled = smote.fit\_resample(x\_test\_tfidf, y\_test\_mapped)

x\_val\_resampled, y\_val\_resampled = smote.fit\_resample(x\_val\_tfidf, y\_val\_mapped)

x\_train\_resampled\_dense =x\_train\_resampled.toarray()

x\_test\_resampled\_dense =x\_test\_resampled.toarray()

x\_val\_resampled\_dense =x\_val\_resampled.toarray()

class\_weights = class\_weight.compute\_class\_weight("balanced",

classes = np.unique(y\_train\_mapped),

y = y\_train\_mapped)

class\_labels = np.unique(y\_train\_mapped)

naive\_bayes\_classifier = MultinomialNB(class\_prior = class\_weights)

#random search hyperparameter tuning

parameter\_grid = {

'alpha': np.linspace(0.01, 5.0, 1000),

'fit\_prior': [True, False]}

random\_search = RandomizedSearchCV(

estimator = naive\_bayes\_classifier,

param\_distributions = parameter\_grid,

n\_iter = 1000,

cv = 10,

scoring = "accuracy",

random\_state = 50,

error\_score="raise"

# Fit the random search to your training data

random\_search.fit(x\_train\_dense , y\_train\_mapped)

# Print the best hyperparameters and corresponding score

print("Best Hyperparameters:", random\_search.best\_params\_)

print("Best Score:", random\_search.best\_score\_)

%pip install optuna

Import optuna

# Define the objective function

def objective(trial):

alpha = trial.suggest\_float('alpha', 0.0, 0.5 )

# Create Multinomial Naive Bayes classifier with the sampled hyperparameters

nb\_classifier = MultinomialNB(alpha=alpha)

# Define cross-validation strategy

cv = StratifiedKFold(n\_splits=10, shuffle=True, random\_state=50)

# Compute cross-validated accuracy

score = cross\_val\_score(nb\_classifier, x\_train\_resampled\_dense, y\_train\_resampled, cv=cv, scoring='accuracy')

# Optimize for the mean accuracy

return 1.0 - np.mean(score)

# Create Optuna study

study = optuna.create\_study(direction='minimize')

# Early stopping setup

best\_trial\_value = float('inf') # Initialize to positive infinity

early\_stopping\_counter = 0

# Perform optimization with early stopping

for \_ in range(10000):

study.optimize(objective, n\_trials=1)

# Check for improvement in the objective value

current\_best\_value = study.best\_value

if current\_best\_value < best\_trial\_value:

best\_trial\_value = current\_best\_value

early\_stopping\_counter = 0

else:

early\_stopping\_counter += 1

if early\_stopping\_counter >= 200: # Adjust consecutive runs for early stopping

print(f"Early stopping after {early\_stopping\_counter} consecutive runs without improvement.")

break

print('Best trial:')

best\_trial = study.best\_trial

print(f' Value: {1.0 - best\_trial.value}')

print(f' Params: {best\_trial.params}')

best\_params = study.best\_params

best\_alpha = best\_params['alpha']

# Create the final Naive Bayes classifier with the best hyperparameters

final\_nb\_classifier = MultinomialNB(alpha=best\_alpha)

# Fit the final model on the resampled training data

final\_nb\_classifier.fit(x\_train\_resampled\_dense, y\_train\_resampled)

# Evaluate on the val set

y\_val\_pred = final\_nb\_classifier.predict(x\_val\_resampled\_dense)

y\_val\_numeric = label\_binarize(y\_val\_resampled, classes=np.unique(y\_val\_resampled))

val\_accuracy = accuracy\_score(y\_val\_resampled, y\_val\_pred)

print("val Accuracy:", val\_accuracy)

val\_confusion\_matrix = confusion\_matrix(y\_val\_resampled, y\_val\_pred)

print("Confusion Matrix:\n", val\_confusion\_matrix)

val\_precision = precision\_score(y\_val\_resampled, y\_val\_pred, average = "macro"

zero\_division = 0)

print("Precision:", val\_precision)

val\_recall = recall\_score(y\_val\_resampled, y\_val\_pred, average = "macro")

print("Recall:", val\_recall)

val\_f1 = f1\_score(y\_val\_resampled, y\_val\_pred, average = "macro")

print("F1 score:", val\_f1)

bo\_decision\_prob =final\_nb\_classifier.predict\_proba(x\_val\_resampled\_dense)

val\_auc\_roc = roc\_auc\_score(y\_val\_numeric,

bo\_decision\_prob,average = "macro", multi\_class='ovr')

print("AUC-ROC:", val\_auc\_roc)

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

# 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\_val\_numeric[:, i], bo\_decision\_prob[:, 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.title('Receiver Operating Characteristic (ROC) Curve')

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

plt.grid(False)

plt.show()

#prediction

test\_pred = final\_nb\_classifier.predict(x\_test\_resampled\_dense)

bo\_test\_decision\_prob = final\_nb\_classifier.predict\_proba(x\_test\_resampled\_dense)

y\_test\_numeric = label\_binarize(y\_test\_resampled, classes=np.unique(y\_test\_mapped))

test\_accuracy = accuracy\_score(y\_test\_resampled, test\_pred)

print("test Accuracy:", test\_accuracy)

test\_confusion\_matrix = confusion\_matrix(y\_test\_resampled, test\_pred)

print("Confusion Matrix:\n", test\_confusion\_matrix)

test\_precision = precision\_score(y\_test\_resampled, test\_pred, average = "macro",

zero\_division = 0)

print("Precision:", test\_precision)

test\_recall = recall\_score(y\_test\_resampled, test\_pred, average = "macro")

print("Recall:", test\_recall)

test\_f1 = f1\_score(y\_test\_resampled, test\_pred, average = "macro")

print("F1 score:", test\_f1)

test\_auc\_roc = roc\_auc\_score(y\_test\_numeric, bo\_test\_decision\_prob,

average = "macro", multi\_class='ovr')

print("AUC-ROC:", test\_auc\_roc)

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

# 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\_numeric[:, i], bo\_test\_decision\_prob[:, 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.title('Receiver Operating Characteristic (ROC) Curve')

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

plt.grid(False)

plt.show()