**#Python script for the Logistic Regression Model**

from sklearn.feature\_extraction.text import TfidfVectorizer

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from skopt import BayesSearchCV

from sklearn.feature\_selection import RFE

from itertools import combinations

from sklearn.linear\_model import LogisticRegression

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

from skopt.space import Categorical, Real, Integer

from sklearn.metrics import classification\_report

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

tfidf\_vectorizer = TfidfVectorizer()

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)

#convert labels in numerical forms

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

y\_train\_mapped = [label\_mapping[label] for label in y\_train]

y\_val\_mapped = [label\_mapping[label] for label in y\_val]

y\_test\_mapped = [label\_mapping[label] for label in y\_test]

logreg = LogisticRegression()

param\_distributions = {

'C': [10, 20, 50, 80, 100, 150],

'penalty': ['l1', 'l2'],

'solver': ['liblinear', 'saga']

}

random\_search = RandomizedSearchCV(logreg,

param\_distributions,

n\_iter=500,

scoring='accuracy',

cv=10,

random\_state=50)

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

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

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

logreg = LogisticRegression()

param\_distributions = {

'C': Real(1e-3, 1e+2, prior='log-uniform'),

'penalty': Categorical(['l1', 'l2']),

'solver': Categorical(['liblinear', 'saga'])

}

bayes\_search = BayesSearchCV(logreg,

param\_distributions,

n\_iter=50,

scoring='accuracy',

cv=10,

n\_jobs=-1,

random\_state=50)

bayes\_search.fit(x\_train\_tfidf, y\_train\_mapped)

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

print("Best Accuracy:", bayes\_search.best\_score\_)

bo\_optimized\_model = LogisticRegression( C = bayes\_search.best\_params\_['C'],

penalty = bayes\_search.best\_params\_['penalty'],

solver = bayes\_search.best\_params\_['solver'])

bo\_optimized\_model.fit(x\_train\_tfidf, y\_train\_mapped)

bo\_y\_pred = bo\_optimized\_model.predict(x\_val\_tfidf)

bo\_y\_val\_bin = label\_binarize(y\_val\_mapped, classes=np.unique(y\_val\_mapped))

#evaluate

bo\_y\_val = np.array(y\_val\_mapped)

nb\_accuracy = accuracy\_score(bo\_y\_val , bo\_y\_pred )

print("Accuracy:", nb\_accuracy)

labels = sorted(set(bo\_y\_val))

nb\_confusion\_matrix = confusion\_matrix(bo\_y\_val , bo\_y\_pred )

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

disp = ConfusionMatrixDisplay(confusion\_matrix=nb\_confusion\_matrix, 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)

nb\_precision = precision\_score(bo\_y\_val , bo\_y\_pred , average = "macro")

print("Precision:", nb\_precision)

nb\_recall = recall\_score(bo\_y\_val , bo\_y\_pred , average = "macro")

print("Recall:", nb\_recall)

nb\_f1 = f1\_score(bo\_y\_val , bo\_y\_pred , average = "macro")

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

bo\_decision\_prob = bo\_optimized\_model.predict\_proba(x\_val\_tfidf)

auc\_roc = roc\_auc\_score(y\_val\_mapped, bo\_decision\_prob, average = "macro", multi\_class = "ovr")

print("Macro-average AUC-ROC:", auc\_roc)

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

# Compute 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\_bin[:, 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]:.4f}")

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

macro\_auc\_values = []

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

macro\_auc\_values.append(roc\_auc[i])

roc\_auc["macro"] = np.mean(macro\_auc\_values)

print(f"Macro-average AUC-ROC: {roc\_auc['macro']:.4f}")

# Calculate the macro-average ROC curve and AUC

fpr["macro"], tpr["macro"], \_ = roc\_curve(y\_val\_bin.ravel(), bo\_decision\_prob.ravel())

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

print(f"Macro-average AUC-ROC: {roc\_auc['macro']:.4f}")

# Plot the ROC curves

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

plt.plot(fpr["macro"], tpr["macro"], label=f"Macro-average ROC curve (AUC = {roc\_auc['macro']:.4f})", color='deeppink', linestyle=':')

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]:.4f})")

plt.plot([0, 1], [0, 1], color='navy', linestyle='--', label = “Random model”)

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")

plt.grid(False)

plt.show()

#prediction

test\_y\_pred = bo\_optimized\_model.predict(x\_test\_tfidf)

test\_bo\_decision\_prob = bo\_optimized\_model.predict\_proba(x\_test\_tfidf)

bo\_y\_test\_bin = label\_binarize(y\_test\_mapped, classes=np.unique(y\_test\_mapped))

#evaluate

bo\_y\_test = np.array(y\_test\_mapped)

nb\_accuracy = accuracy\_score(bo\_y\_test , test\_y\_pred )

print("Accuracy:", nb\_accuracy)

labels = sorted(set(bo\_y\_test))

test\_confusion\_matrix = confusion\_matrix(bo\_y\_test , test\_y\_pred)

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

disp = ConfusionMatrixDisplay(confusion\_matrix=test\_confusion\_matrix, 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)

nb\_precision = precision\_score(bo\_y\_test , test\_y\_pred , average = "macro")

print("Precision:", nb\_precision)

nb\_recall = recall\_score(bo\_y\_test , test\_y\_pred , average = "macro")

print("Recall:", nb\_recall)

nb\_f1 = f1\_score(bo\_y\_test , test\_y\_pred , average = "macro")

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

test\_bo\_decision\_prob = bo\_optimized\_model.predict\_proba(x\_test\_tfidf)

auc\_roc = roc\_auc\_score(y\_test\_mapped, test\_bo\_decision\_prob, average = "macro", multi\_class = "ovr")

print("Macro-average AUC-ROC:", auc\_roc)

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

# Compute 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(bo\_y\_test\_bin[:, i], test\_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]:.4f}")

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

macro\_auc\_values = []

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

macro\_auc\_values.append(roc\_auc[i])

roc\_auc["macro"] = np.mean(macro\_auc\_values)

print(f"Macro-average AUC-ROC: {roc\_auc['macro']:.4f}")

# Calculate the macro-average ROC curve and AUC

fpr["macro"], tpr["macro"], \_ = roc\_curve(bo\_y\_test\_bin.ravel(), test\_bo\_decision\_prob.ravel())

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

print(f"Test Set Macro-average AUC-ROC: {roc\_auc['macro']:.4f}")

# Plot the ROC curves

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

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

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})")

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")

plt.grid(False)

plt.show()