**#Python script for the Random Forest Model**

%pip install scikit-optimize

%pip install pyswarm

from sklearn.ensemble import RandomForestClassifier

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

from pyswarm import pso

import time

from sklearn.utils import shuffle

from gensim.utils import simple\_preprocess

from sklearn.feature\_extraction.text import TfidfVectorizer

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

from sklearn.preprocessing import label\_binarize

import numpy as np

import pandas as pd

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)

print('Original class distribution:', Counter(y\_train\_mapped))

y\_train = np.array(y\_train).reshape(-1, 1)

y\_val = np.array(y\_val).reshape(-1, 1)

y\_test = np.array(y\_test).reshape(-1,1)

smote = SMOTE(random\_state=50)

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

# Shuffle the resampled data

x\_train\_tfidf\_resampled, y\_train\_resampled = shuffle(x\_train\_tfidf\_resampled, y\_train\_resampled, random\_state=50)

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

print('Resampled class distribution:', Counter(y\_train\_resampled))

# Convert to a DataFrame for easier plotting

df\_resampled = pd.DataFrame(x\_train\_dense\_resampled)

df\_resampled['label'] = y\_train\_resampled

# Optionally reduce dimensionality for visualization

from sklearn.decomposition import PCA

pca = PCA(n\_components=2)

x\_train\_pca = pca.fit\_transform(x\_train\_dense\_resampled)

# Create a DataFrame with PCA results

df\_pca = pd.DataFrame(x\_train\_pca, columns=['PC1', 'PC2'])

df\_pca['Sentiment label'] = y\_train\_resampled

# Scatter plot using PCA components

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

sns.scatterplot(data=df\_pca, x='PC1', y='PC2', hue='Sentiment label', palette='viridis')

plt.show()

def Random\_Diverse\_Rep(text, sentiment, num\_reps):

tweet = []

tweet\_sentiment = []

for \_ in range(num\_reps):

random\_index = np.random.randint(low = 0, high = len(text))

tweet.append(text[random\_index])

tweet\_sentiment.append(sentiment[random\_index])

return tweet, tweet\_sentiment

# Randomly shuffle the dataset

x\_train\_tfidf, y\_train\_mapped = shuffle(x\_train\_tfidf, y\_train\_mapped, random\_state = 50)

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

# Generate diverse representations

num\_reps = 150

text, sentiment = Random\_Diverse\_Rep(x\_train\_dense, y\_train\_mapped, num\_reps)

#Random search

rf\_classifier = RandomForestClassifier()

max\_features\_dist = randint(10, 100)

parameter\_grid = {

'n\_estimators': randint(10, 1000),

'max\_depth': randint(5, 50), #of the trees

'min\_samples\_split': randint(2, 20),

'min\_samples\_leaf': randint(2, 4),

'max\_features': max\_features\_dist}

random\_search = RandomizedSearchCV(

estimator = rf\_classifier,

param\_distributions = parameter\_grid,

n\_iter = 100,

cv = 10,

scoring = 'accuracy',

random\_state = 50)

# Fit the random search to your training data

random\_search.fit(text, sentiment)

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

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

best\_estimator = random\_search.best\_estimator\_

# Define the objective function to be minimized (negative accuracy)

def objective\_function(params):

    n\_estimators, max\_depth, min\_samples\_split, min\_samples\_leaf, max\_features = params

    rf\_classifier = RandomForestClassifier(

        n\_estimators=int(n\_estimators),

        max\_depth=int(max\_depth),

        min\_samples\_split=int(min\_samples\_split),

        min\_samples\_leaf=int(min\_samples\_leaf),

        max\_features=int(max\_features)

    )

    #cross-validation to evaluate the model

    scores = cross\_val\_score(rf\_classifier, text, sentiment, cv=10, scoring=make\_scorer(accuracy\_score))

    mean\_score = scores.mean()

    return -mean\_score

# Define the search space for PSO

lb = [5, 5, 2, 2, 10]

ub = [100, 100, 20, 4, 50]

#PSO optimization

best\_params, best\_score = pso(objective\_function, lb, ub, swarmsize=20, maxiter=100)

# Convert best\_params to integer values

best\_params = [int(param) for param in best\_params]

# Print the best hyperparameters and corresponding score

print("Best Hyperparameters:", dict(zip(['n\_estimators', 'max\_depth', 'min\_samples\_split', 'min\_samples\_leaf', 'max\_features'], best\_params)))

print("Best Score:", -best\_score)

# RandomForestClassifier with the optimized hyperparameters

optimized\_rf\_classifier = RandomForestClassifier(

n\_estimators=best\_params[0],

max\_depth=best\_params[1],

min\_samples\_split=best\_params[2],

min\_samples\_leaf=best\_params[3],

max\_features=best\_params[4])

# Training

optimized\_rf\_classifier.fit(text, sentiment)

#validation

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

# Use the trained model for sentiment analysis prediction

sentiment\_pred = optimized\_rf\_classifier.predict(x\_val\_dense)

label\_mapping = {

    'neutral': 0,

    'negative': 1,

    'positive': 2

}

# Convert the predictions to numeric labels

sentiment\_pred = [label\_mapping[pred] for pred in sentiment\_pred]

#y\_test\_onehot = np.zeros\_like(sentiment\_pred )

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

y\_val = np.array(y\_val\_mapped

#evaluate

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

rf\_accuracy = accuracy\_score(y\_val, sentiment\_pred)

print("Accuracy:", rf\_accuracy)

#Extract class labels

labels = sorted(set(y\_val))

rf\_cm = confusion\_matrix(y\_val, sentiment\_pred)

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

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

# Plot confusion matrix

fig, ax = plt.subplots(figsize=(8, 8))

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')

# Show plot

plt.show()

rf\_precision = precision\_score(y\_val, sentiment\_pred, average = "macro")

print("Precision:", rf\_precision)

rf\_recall = recall\_score(y\_val, sentiment\_pred, average = "macro")

print("Recall:", rf\_recall)

rf\_f1 = f1\_score(y\_val, sentiment\_pred, average = "macro")

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

decision\_prob = optimized\_rf\_classifier.predict\_proba(x\_val\_dense)

auc\_roc = roc\_auc\_score(y\_val\_bin, decision\_prob, average="macro", multi\_class="ovr")

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

# Assuming you have a label\_mapping dictionary that maps class indices to class labels

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], 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}")

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

# Plot macro-average ROC curve

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

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

color='deeppink', linestyle=':')

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", fontsize="small")

plt.grid(False)

plt.show()

# Use the trained model for sentiment analysis prediction

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

# Use the trained model for sentiment analysis prediction

test\_pred = optimized\_rf\_classifier.predict(x\_test\_dense)

label\_mapping = {

    'neutral': 0,

    'negative': 1,

    'positive': 2

}

# Convert the predictions to numeric labels

test\_pred = [label\_mapping[pred] for pred in test\_pred]

#y\_test\_onehot = np.zeros\_like(sentiment\_pred )

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

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

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

print("Accuracy:", rf\_accuracy)

labels = sorted(set(y\_test\_mapped))

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

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

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')

# Show plot

plt.show()

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

print("Precision:", rf\_precision)

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

print("Recall:", rf\_recall)

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

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

test\_decision\_prob = optimized\_rf\_classifier.predict\_proba(x\_test\_dense)

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

print("AUC-ROC:", 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\_bin[:, i], 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]:.4f})")

# Plot macro-average ROC curve

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

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

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()