**#Python script for the** **Stacked Ensemble**

from tensorflow.keras.preprocessing.text import Tokenizer

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

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

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.optimizers import Adam

import numpy as np

import pandas as pd

from sklearn.linear\_model import LogisticRegression

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv1D, MaxPooling2D, Flatten, Dense

from tensorflow.keras.utils import to\_categorical

from scipy.stats import randint

from sklearn.ensemble import RandomForestClassifier

from sklearn.utils import class\_weight

from sklearn.naive\_bayes import MultinomialNB

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

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.preprocessing import StandardScaler

from sklearn.utils import shuffle

from sklearn.model\_selection import KFold

import optuna

import xgboost as xgb

from sklearn.preprocessing import label\_binarize

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

import matplotlib.pyplot as plt

import optuna.visualization as vis

from sklearn import svm

#Data processing for non-neural network models

# Initial split into training and test sets

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, stratify=y, random\_state=50)

# Further split training data into training and meta training sets

x\_base\_train, x\_meta\_train, y\_base\_train, y\_meta\_train = train\_test\_split(x\_train, y\_train, test\_size=0.2, stratify=y\_train, random\_state=50)

# Convert labels to numerical forms

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

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

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

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

tfidf\_vectorizer = TfidfVectorizer()

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

x\_meta\_train\_tfidf = tfidf\_vectorizer.transform(x\_meta\_train)

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

#data processing for neural network model

# Convert labels to one-hot encoding for CNN

y\_base\_train\_one\_hot = to\_categorical(y\_base\_train\_mapped, num\_classes=3)

y\_meta\_train\_one\_hot = to\_categorical(y\_meta\_train\_mapped, num\_classes=3)

y\_test\_one\_hot = to\_categorical(y\_test\_mapped, num\_classes=3)

vocab\_size = 30

# Initialize and fit the tokenizer on x\_train

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

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

# Tokenize x\_base\_train, x\_meta\_train, and x\_test

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

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

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

# Pad sequences

seq\_length = 20

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

x\_meta\_train\_padded = pad\_sequences(x\_meta\_train\_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\_base\_train\_np = np.array(x\_base\_train\_padded)

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

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

***For stacked ensemble 1:***

lr\_model = LogisticRegression(C = 999648.1844014695,penalty = 'l2', solver = 'saga',)

svm\_model = svm.SVC(C= 2.1644511010807355,kernel = 'sigmoid',gamma = 'scale',probability = True)

nb\_model = MultinomialNB(alpha = 0.49988)

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

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

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

lr\_val\_preds = lr\_model.predict\_proba(x\_val\_tfidf)[:, 1]

svm\_val\_preds = svm\_model.predict\_proba(x\_val\_tfidf)[:, 1]

nb\_val\_preds = nb\_model.predict\_proba(x\_val\_tfidf)[:, 1]

stacked\_predictions = np.column\_stack((svm\_val\_preds, lr\_val\_preds, nb\_val\_preds ))

print(stacked\_predictions.shape)

def objective(trial):

params = {

'objective': 'multi:softmax',

'num\_class': 3,

'random\_state': 50,

'max\_depth': trial.suggest\_int('max\_depth', 60, 65),

'learning\_rate': trial.suggest\_float('learning\_rate', 0.3, 0.5),

'subsample': trial.suggest\_float('subsample', 0.3, 0.7),

'colsample\_bytree': trial.suggest\_float('colsample\_bytree', 0.3, 0.6),

}

boosted\_rf\_classifier = xgb.XGBClassifier(\*\*params)

boosted\_rf\_classifier.fit(x\_train\_tfidf , y\_train\_mapped)

y\_val\_pred = boosted\_rf\_classifier.predict(x\_val\_tfidf)

# Evaluate F1 score

f1 = f1\_score(y\_val\_mapped, y\_val\_pred, average='macro')

return 1.0 - f1 # Optimize for accuracy (minimize 1 - accuracy)

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

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

early\_stopping\_counter = 0

for \_ in range(10000):

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

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 >= 100: # Adjust the number of consecutive runs for early stopping

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

break

best\_params = study.best\_params

print("Best Hyperparameters:", best\_params)

# Train the final model with the best hyperparameters

xgb\_model = xgb.XGBClassifier()

xgb\_model.fit(stacked\_predictions, y\_val\_mapped)

val\_svm\_preds = svm\_model.predict\_proba(x\_test\_tfidf)[:, 1]

val\_lr\_preds = lr\_model.predict\_proba(x\_test\_tfidf)[:, 1]

val\_nb\_preds = nb\_model.predict\_proba(x\_test\_tfidf)[:, 1]

val\_stacked\_predictions = np.column\_stack((val\_svm\_preds, val\_lr\_preds, val\_nb\_preds ))

# final test pred

val\_preds = xgb\_model.predict(val\_stacked\_predictions)

val\_preds\_prob = xgb\_model.predict\_proba(val\_stacked\_predictions)

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

stacked\_accuracy = accuracy\_score(y\_test\_mapped, val\_preds)

print("Accuracy:", stacked\_accuracy)

stacked\_precision = precision\_score(y\_test\_mapped ,val\_preds, average='macro')

print("Precision:", stacked\_precision)

stacked\_recall = recall\_score(y\_test\_mapped , val\_preds, average='macro')

print("Recall:", stacked\_recall)

stacked\_f1 = f1\_score(y\_test\_mapped , val\_preds, average='macro')

print("F1 Score:", stacked\_f1)

stacked\_cm = confusion\_matrix(y\_test\_mapped , val\_preds)

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

stacked\_auc\_roc = roc\_auc\_score(y\_test\_numeric, val\_preds\_prob,average = "macro", multi\_class='ovr')

print("AUC-ROC:", stacked\_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\_test\_numeric[:, i], val\_preds\_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

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)

fpr["macro"] = all\_fpr

tpr["macro"] = mean\_tpr

# Plot the ROC curves

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

plt.plot([0, 1], [0, 1], color='navy', 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})")

# Plot the macro-average ROC curve

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

***For stacked ensemble 2:***

xgb\_model = xgb.XGBClassifier(objective='multi:softmax', num\_class=3, random\_state=50, max\_depth= 69, learning\_rate = 0.052862, subsample = 0.63021, colsample\_bytree = 0.42458, gamma = 0.11968, reg\_alpha = 0.44536, reg\_lambda = 2.2946)

svm\_model = svm.SVC(C=2.1644511010807355, kernel='sigmoid', gamma='scale', probability=True)

nb\_model = MultinomialNB(alpha=0.49988)

xgb\_model.fit(x\_base\_train\_tfidf, y\_base\_train\_mapped)

svm\_model.fit(x\_base\_train\_tfidf, y\_base\_train\_mapped)

nb\_model.fit(x\_base\_train\_tfidf, y\_base\_train\_mapped)

# Generate predictions on the meta training set

meta\_svm\_preds = svm\_model.predict\_proba(x\_meta\_train\_tfidf)

meta\_xgb\_preds = xgb\_model.predict\_proba(x\_meta\_train\_tfidf)

meta\_nb\_preds = nb\_model.predict\_proba(x\_meta\_train\_tfidf)

# Stack predictions horizontally to create meta features

stacked\_meta\_features = np.column\_stack((meta\_svm\_preds, meta\_xgb\_preds, meta\_nb\_preds))

# Define the objective function

def objective(trial):

# Define the hyperparameter search space

C = trial.suggest\_float('C', 3.7, 3.9, log=True)

penalty = trial.suggest\_categorical('penalty', ['l1', 'l2'])

# Combined search space for solver and penalty

solver\_penalty = trial.suggest\_categorical('solver\_penalty', [

('liblinear', 'l1'),

('liblinear', 'l2'),

('saga', 'l1'),

('saga', 'l2'),

('newton-cg', 'l2'),

('lbfgs', 'l2'),

('sag', 'l2')

])

solver, penalty = solver\_penalty

lr\_model = LogisticRegression(C=C, penalty=penalty, solver=solver, max\_iter=1000, random\_state=50)

lr\_model.fit(stacked\_meta\_features, y\_meta\_train\_mapped)

y\_val\_pred = lr\_model.predict(stacked\_meta\_features)

f1 = f1\_score(y\_meta\_train\_mapped, y\_val\_pred, average='macro')

return 1.0 - f1 # Minimize 1 - F1 score

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

study.optimize(objective, n\_trials=150)

best\_params = study.best\_params

print("Best Hyperparameters:", best\_params)

lr\_model = LogisticRegression(

C=best\_params['C'],

penalty=best\_params['solver\_penalty'][1],

solver=best\_params['solver\_penalty'][0],

max\_iter=1000,

random\_state=50

)

lr\_model.fit(stacked\_meta\_features, y\_meta\_train\_mapped)

# Generate predictions on the test set

test\_svm\_preds = svm\_model.predict\_proba(x\_test\_tfidf)

test\_xgb\_preds = xgb\_model.predict\_proba(x\_test\_tfidf)

test\_nb\_preds = nb\_model.predict\_proba(x\_test\_tfidf)

# Stack predictions horizontally to create test meta features

stacked\_test\_features = np.column\_stack((test\_svm\_preds, test\_xgb\_preds, test\_nb\_preds))

fig\_optimization\_history = vis.plot\_optimization\_history(study)

fig\_optimization\_history.show()

fig\_param\_importance = vis.plot\_param\_importances(study)

fig\_param\_importance.show()

fig\_parallel\_coordinates = vis.plot\_parallel\_coordinate(study)

fig\_parallel\_coordinates.show()

fig\_slice = vis.plot\_slice(study)

fig\_slice.show()

# Final test predictions

final\_preds = lr\_model.predict(stacked\_test\_features)

final\_preds\_prob = lr\_model.predict\_proba(stacked\_test\_features)

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

stacked\_accuracy = accuracy\_score(y\_test\_mapped, final\_preds)

print("Accuracy:", stacked\_accuracy)

stacked\_precision = precision\_score(y\_test\_mapped, final\_preds, average='macro')

print("Precision:", stacked\_precision)

stacked\_recall = recall\_score(y\_test\_mapped, final\_preds, average='macro')

print("Recall:", stacked\_recall)

stacked\_f1 = f1\_score(y\_test\_mapped, final\_preds, average='macro')

print("F1 Score:", stacked\_f1)

stacked\_cm = confusion\_matrix(y\_test\_mapped, final\_preds)

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

stacked\_auc\_roc = roc\_auc\_score(y\_test\_numeric, final\_preds\_prob, average='macro', multi\_class='ovr')

print("AUC-ROC:", stacked\_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\_test\_numeric[:, i], final\_preds\_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

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)

fpr["macro"] = all\_fpr

tpr["macro"] = mean\_tpr

# Plot the ROC curves

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

plt.plot([0, 1], [0, 1], color='navy', 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})")

# Plot the macro-average ROC curve

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

***For stacked ensemble 3:***

cnn\_model = Sequential([

Embedding(input\_dim=vocab\_size, output\_dim=60, input\_length=seq\_length), # Embedding layer

Conv1D(filters=80, kernel\_size=5, activation='relu'), # 1D Convolutional layer

GlobalMaxPooling1D(), # Global Max Pooling layer

Dense(200, activation='relu'), # Dense layer

Dropout(0.3), # Dropout layer

Dense(3, activation='softmax') # Output layer

])

early\_stopping = EarlyStopping(monitor='val\_loss', patience=20, restore\_best\_weights=True)

# Define the learning rate

learning\_rate = 0.55e-4

# Create an optimizer with the desired learning rate

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

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

# Print the model summary

cnn\_model.summary()

# Train the model

cnn\_model.fit(x\_base\_train\_np, y\_base\_train\_one\_hot, epochs=1000, batch\_size=30, validation\_split=0.2, callbacks=[early\_stopping])

svm\_model = svm.SVC(probability=True, random\_state = 50)

rf\_model = RandomForestClassifier(random\_state = 50)

nb\_model = MultinomialNB()

svm\_model.fit(x\_base\_train\_tfidf, y\_base\_train)

rf\_model.fit(x\_base\_train\_tfidf, y\_base\_train)

nb\_model.fit(x\_base\_train\_tfidf, y\_base\_train)

# Get predictions for meta training and test data

cnn\_meta\_train\_preds = cnn\_model.predict(x\_meta\_train\_np)

svm\_meta\_train\_preds = svm\_model.predict\_proba(x\_meta\_train\_tfidf)

rf\_meta\_train\_preds = rf\_model.predict\_proba(x\_meta\_train\_tfidf)

nb\_meta\_train\_preds = nb\_model.predict\_proba(x\_meta\_train\_tfidf)

cnn\_test\_preds = cnn\_model.predict(x\_test\_np)

svm\_test\_preds = svm\_model.predict\_proba(x\_test\_tfidf)

rf\_test\_preds = rf\_model.predict\_proba(x\_test\_tfidf)

nb\_test\_preds = nb\_model.predict\_proba(x\_test\_tfidf)

stacked\_meta\_features = np.hstack((cnn\_meta\_train\_preds, svm\_meta\_train\_preds, rf\_meta\_train\_preds, nb\_meta\_train\_preds))

stacked\_test\_features = np.hstack((cnn\_test\_preds, svm\_test\_preds, rf\_test\_preds, nb\_test\_preds))

# Define the objective function

def objective(trial):

# Define the hyperparameter search space

C = trial.suggest\_float('C', 3.9, 4.9, log=True)

penalty = trial.suggest\_categorical('penalty', ['l1', 'l2'])

# Combined search space for solver and penalty

solver\_penalty = trial.suggest\_categorical('solver\_penalty', [

('liblinear', 'l1'),

('liblinear', 'l2'),

('saga', 'l1'),

('saga', 'l2'),

('newton-cg', 'l2'),

('lbfgs', 'l2'),

('sag', 'l2')

])

solver, penalty = solver\_penalty

lr\_model = LogisticRegression(C=C, penalty=penalty, solver=solver, max\_iter=1000, random\_state=50)

lr\_model.fit(stacked\_meta\_features, y\_meta\_train\_mapped)

y\_val\_pred = lr\_model.predict(stacked\_meta\_features)

f1 = f1\_score(y\_meta\_train\_mapped, y\_val\_pred, average='macro')

return 1.0 - f1 # Minimize 1 - F1 score

# Create the study and optimize the objective function

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

study.optimize(objective, n\_trials=300)

# Get the best hyperparameters

best\_params = study.best\_params

print("Best Hyperparameters:", best\_params)

meta\_learner = LogisticRegression(C=best\_params['C'],

penalty=best\_params['solver\_penalty'][1],

solver=best\_params['solver\_penalty'][0],

max\_iter=1000,

random\_state=50)

meta\_learner.fit(stacked\_meta\_features, y\_meta\_train)

fig\_optimization\_history = vis.plot\_optimization\_history(study)

fig\_optimization\_history.show()

fig\_param\_importance = vis.plot\_param\_importances(study)

fig\_param\_importance.show()

fig\_parallel\_coordinates = vis.plot\_parallel\_coordinate(study)

fig\_parallel\_coordinates.show()

fig\_slice = vis.plot\_slice(study)

fig\_slice.show()

# Evaluate the meta-learner on the test data

final\_preds = meta\_learner.predict(stacked\_test\_features)

final\_preds\_prob = meta\_learner.predict\_proba(stacked\_test\_features)

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

stacked\_accuracy = accuracy\_score(y\_test, final\_preds)

print("Accuracy:", stacked\_accuracy)

stacked\_precision = precision\_score(y\_test, final\_preds, average='macro')

print("Precision:", stacked\_precision)

stacked\_recall = recall\_score(y\_test, final\_preds, average='macro')

print("Recall:", stacked\_recall)

stacked\_f1 = f1\_score(y\_test, final\_preds, average='macro')

print("F1 Score:", stacked\_f1)

stacked\_cm = confusion\_matrix(y\_test, final\_preds)

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

stacked\_auc\_roc = roc\_auc\_score(y\_test\_numeric, final\_preds\_prob, average='macro', multi\_class='ovr')

print("AUC-ROC:", stacked\_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\_test\_numeric[:, i], final\_preds\_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

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)

fpr["macro"] = all\_fpr

tpr["macro"] = mean\_tpr

# Plot the ROC curves

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

plt.plot([0, 1], [0, 1], color='navy', 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})")

# Plot the macro-average ROC curve

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

***For stacked ensemble 4:***

cnn\_model = Sequential([

Embedding(input\_dim=vocab\_size, output\_dim=60, input\_length=seq\_length), # Embedding layer

Conv1D(filters=80, kernel\_size=5, activation='relu'), # 1D Convolutional layer

GlobalMaxPooling1D(), # Global Max Pooling layer

Dense(200, activation='relu'), # Dense layer

Dropout(0.3), # Dropout layer

Dense(3, activation='softmax') # Output layer

])

early\_stopping = EarlyStopping(monitor='val\_loss', patience=20, restore\_best\_weights=True)

# Define the learning rate

learning\_rate = 0.55e-4

# Create an optimizer with the desired learning rate

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

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

# Print the model summary

cnn\_model.summary()

# Train the model

cnn\_model.fit(x\_base\_train\_np, y\_base\_train\_one\_hot, epochs=1000, batch\_size=30, validation\_split=0.2, callbacks=[early\_stopping])

rf\_model = RandomForestClassifier(n\_estimators = 9, max\_depth = 82, min\_samples\_split = 10, minsamples\_leaf = 2, max\_features = 42, random\_state= 50)

nb\_model = MultinomialNB(alpha = 0.49988)

lr\_model = LogisticRegression(C = 999648.1844014695,penalty = 'l2', solver = 'saga',random\_state = 50)

rf\_model.fit(x\_base\_train\_tfidf, y\_base\_train)

nb\_model.fit(x\_base\_train\_tfidf, y\_base\_train)

lr\_model.fit(x\_base\_train\_tfidf, y\_base\_train)

# Get predictions for meta training and test data

cnn\_meta\_train\_preds = cnn\_model.predict(x\_meta\_train\_np)

lr\_meta\_train\_preds = lr\_model.predict\_proba(x\_meta\_train\_tfidf)

rf\_meta\_train\_preds = rf\_model.predict\_proba(x\_meta\_train\_tfidf)

nb\_meta\_train\_preds = nb\_model.predict\_proba(x\_meta\_train\_tfidf)

cnn\_test\_preds = cnn\_model.predict(x\_test\_np)

lr\_test\_preds = lr\_model.predict\_proba(x\_test\_tfidf)

rf\_test\_preds = rf\_model.predict\_proba(x\_test\_tfidf)

nb\_test\_preds = nb\_model.predict\_proba(x\_test\_tfidf)

stacked\_meta\_features = np.hstack((cnn\_meta\_train\_preds, lr\_meta\_train\_preds, rf\_meta\_train\_preds, nb\_meta\_train\_preds))

stacked\_test\_features = np.hstack((cnn\_test\_preds, lr\_test\_preds, rf\_test\_preds, nb\_test\_preds))

# Create the scaler

scaler = StandardScaler()

# Fit the scaler on the meta-features and transform them

stacked\_meta\_features\_standardized = scaler.fit\_transform(stacked\_meta\_features)

stacked\_test\_features\_standardized = scaler.transform(stacked\_test\_features)

def objective(trial):

# Define the hyperparameter space for the SVM model

params = {

'C': trial.suggest\_float('C', 0.1, 10.0),

'kernel': trial.suggest\_categorical('kernel', ['linear', 'poly', 'rbf', 'sigmoid']),

'degree': trial.suggest\_int('degree', 2, 5) if trial.params['kernel'] == 'poly' else 3, # Only used if kernel is 'poly'

'gamma': trial.suggest\_categorical('gamma', ['scale', 'auto'])

}

# Initialize the SVM model with the suggested parameters

svm\_model = svm.SVC(\*\*params, probability=True, random\_state=50)

# Train the SVM model

svm\_model.fit(stacked\_meta\_features\_standardized, y\_meta\_train\_mapped)

# Predict on the training set (you can use cross-validation for more robust evaluation)

y\_val\_pred = svm\_model.predict(stacked\_meta\_features\_standardized)

# Calculate the F1 score

f1 = f1\_score(y\_meta\_train\_mapped, y\_val\_pred, average='macro')

# Return the value to be minimized (1.0 - f1\_score)

return 1.0 - f1

# Create the Optuna study

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

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

early\_stopping\_counter = 0

# Perform the optimization

for \_ in range(10000):

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

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 >= 100: # Adjust the number of consecutive runs for early stopping

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

break

# Get the best hyperparameters

best\_params = study.best\_params

print("Best Hyperparameters:", best\_params)

fig\_optimization\_history = vis.plot\_optimization\_history(study)

fig\_optimization\_history.show()

fig\_param\_importance = vis.plot\_param\_importances(study)

fig\_param\_importance.show()

fig\_parallel\_coordinates = vis.plot\_parallel\_coordinate(study)

fig\_parallel\_coordinates.show()

fig\_slice = vis.plot\_slice(study)

fig\_slice.show()

svm\_model = svm.SVC(\*\*best\_params, probability=True, random\_state = 50)

svm\_model.fit(stacked\_meta\_features\_standardized, y\_meta\_train\_mapped)

# Evaluate the meta-learner on the test data

final\_preds = svm\_model.predict(stacked\_test\_features)

final\_preds\_prob = svm\_model.predict\_proba(stacked\_test\_features)

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

# Convert y\_test\_mapped to one-hot encoding

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

# Compute accuracy

stacked\_accuracy = accuracy\_score(y\_test\_mapped, final\_preds)

print("Accuracy:", stacked\_accuracy)

# Compute precision

stacked\_precision = precision\_score(y\_test\_mapped, final\_preds, average='macro', zero\_division=0)

print("Precision:", stacked\_precision)

# Compute recall

stacked\_recall = recall\_score(y\_test\_mapped, final\_preds, average='macro')

print("Recall:", stacked\_recall)

# Compute F1 score

stacked\_f1 = f1\_score(y\_test\_mapped, final\_preds, average='macro')

print("F1 Score:", stacked\_f1)

# Compute confusion matrix

stacked\_cm = confusion\_matrix(y\_test\_mapped, final\_preds)

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

# Compute AUC-ROC

stacked\_auc\_roc = roc\_auc\_score(y\_test\_one\_hot, final\_preds\_prob, average='macro', multi\_class='ovr')

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

# Compute ROC curve and AUC for each class

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

fpr = dict()

tpr = dict()

roc\_auc = dict()

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

fpr[i], tpr[i], \_ = roc\_curve(y\_test\_one\_hot[:, i], final\_preds\_prob[:, i])

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

print(f"AUC-ROC for {label\_mapping\_after\_training[i]}: {roc\_auc[i]:.4f}")

# Compute macro-average AUC-ROC

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}")

# Compute macro-average ROC curve

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)

fpr["macro"] = all\_fpr

tpr["macro"] = mean\_tpr

# Plot ROC curves

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

plt.plot([0, 1], [0, 1], color='navy', 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})")

# Plot macro-average ROC curve

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