import numpy as np

import pandas as pd

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split, GridSearchCV, StratifiedKFold, cross\_val\_score, RepeatedStratifiedKFold

from sklearn.preprocessing import MinMaxScaler, RobustScaler, StandardScaler

from imblearn.pipeline import Pipeline

from sklearn.neighbors import KNeighborsClassifier

from sklearn.preprocessing import OneHotEncoder

from sklearn.svm import SVC

from xgboost.sklearn import XGBClassifier

import xgboost as xgb

import matplotlib.pyplot as plt

import seaborn as sns

from seaborn import heatmap

from sklearn.metrics import classification\_report, confusion\_matrix, roc\_curve, roc\_auc\_score

from sklearn.compose import ColumnTransformer

from sklearn.utils import compute\_class\_weight

from joblib import dump, load

from keras.models import Sequential

from keras.layers import LSTM, Dense

from keras.optimizers import Adam

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from imblearn.pipeline import Pipeline as imbpipeline

from imblearn.over\_sampling import SMOTE

--------------------------

df = pd.read\_csv('Test Analysis 1 25-5-24.csv')

--------------------------

# Preprocessing

df['Year'] = pd.to\_datetime(df['Year'], format='%Y')

df['Month'] = pd.to\_datetime(df['Month'], format='%m')

df['month'] = df['Month'].dt.month

df['year'] = df['Year'].dt.year

df.drop(['Year', 'Month'], axis=1, inplace=True)

--------------------------

# Feature and target definitions # Prepare for LGBM

num\_cols = ['P', 'T', 'month', 'year']

X = df[num\_cols]

y = df['H']

--------------------------

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, stratify=y, random\_state=11)

--------------------------

preprocessor = ColumnTransformer(

 transformers=[

 ('num', StandardScaler(), num\_cols),

 ],

 remainder='passthrough'

)

weights = [0.8, 0.9]

pipeline = imbpipeline(steps = [['smote', SMOTE(random\_state=11)],

 ['scaler', preprocessor],

 ['classifier', SVC(probability=True)]])

param\_grid = {'classifier\_\_C':[0.2],

 'smote\_\_sampling\_strategy': weights,

 'classifier\_\_kernel':['rbf'],

 'classifier\_\_gamma':[0.03, 1]}

grid\_search = GridSearchCV(estimator=pipeline,

 param\_grid=param\_grid,

 scoring='roc\_auc',

 n\_jobs=-1)

--------------------------

#Execute Training

import time

start\_time = time.time()

--------------------------

grid\_search.fit(X\_train, y\_train)

--------------------------

#To monitor training time

end\_time = time.time()

print(f'Time taken: {end\_time - start\_time:.3f} seconds')

--------------------------

cv\_score = grid\_search.best\_score\_

test\_score = grid\_search.score(X\_test, y\_test)

print(f'Cross-validation score: {cv\_score}\nTest score: {test\_score}')

--------------------------

# Get the best estimator from the GridSearchCV object

best\_estimator = grid\_search.best\_estimator\_

# Get the predicted probabilities for the test set

y\_test\_proba = best\_estimator.predict\_proba(X\_test)[:, 1]

# Compute the fpr, tpr, and thresholds for the ROC curve

fpr, tpr, thresholds = roc\_curve(y\_test, y\_test\_proba)

# Plot the ROC curve

plt.plot(fpr, tpr, label='ROC curve')

plt.plot([0, 1], [0, 1], 'k--', label='Random guess')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curve')

# Compute the AUC

auc = roc\_auc\_score(y\_test, y\_test\_proba)

# Add the AUC score to the graph

plt.annotate(f'AUC = {auc:.4f}', xy=(0.8, 0.2), xycoords='axes fraction')

plt.legend(loc='best')

plt.show()

--------------------------

# Get the best estimator from the GridSearchCV object

best\_estimator = grid\_search.best\_estimator\_

# Get the predicted probabilities for the test set

y\_test\_proba = best\_estimator.predict\_proba(X\_test)[:, 1]

# Define a list of threshold values to check

thresholds = np.linspace(0.0005, 1, 1000)

#[0.0005, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

# Create empty lists to store the results

sensitivities = []

specificities = []

accuracies = []

precisions = []

recalls = []

f1\_scores = []

# Iterate over the threshold values

for threshold in thresholds:

 # Modify the predicted probabilities based on the threshold

 y\_test\_pred = [1 if prob >= threshold else 0 for prob in y\_test\_proba]

 # Compute the confusion matrix

 conf\_matrix = confusion\_matrix(y\_test, y\_test\_pred)

 # Extract true positives, true negatives, false positives, and false negatives

 tp = conf\_matrix[1,1]

 tn = conf\_matrix[0,0]

 fp = conf\_matrix[0,1]

 fn = conf\_matrix[1,0]

 sensitivity = tp / (tp + fn)

 specificity = tn / (tn + fp)

 accuracy = (tp + tn) / (tp + tn + fp + fn)

 precision = (tp+1) / (tp + fp+1) # Add a small value to both numerator and denominator

 recall = sensitivity

 f1\_score = 2 \* (precision \* recall) / (precision + recall)

 # Append the results to the lists

 sensitivities.append(sensitivity)

 specificities.append(specificity)

 accuracies.append(accuracy)

 precisions.append(precision)

 recalls.append(recall)

 f1\_scores.append(f1\_score)

# Plot the results

plt.plot(thresholds, sensitivities, label='Sensitivity')

plt.plot(thresholds, specificities, label='Specificity')

plt.plot(thresholds, accuracies, label='Accuracy')

#plt.plot(thresholds, precisions, label='Precision')

#plt.plot(thresholds, f1\_scores, label='F1-score')

plt.legend()

plt.xlabel('Threshold')

plt.ylabel('Score')

#plt.title('H')

plt.show()

--------------------------

# Set the desired threshold

desired\_threshold = 0.5

# Modify the predicted probabilities based on the desired threshold

y\_test\_pred = [1 if prob >= desired\_threshold else 0 for prob in y\_test\_proba]

# Compute the confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_test\_pred)

# Extract true positives, true negatives, false positives, and false negatives

tp = conf\_matrix[1, 1]

tn = conf\_matrix[0, 0]

fp = conf\_matrix[0, 1]

fn = conf\_matrix[1, 0]

# Calculate precision, recall, and F1-score

sensitivity = tp / (tp + fn)

specificity = tn / (tn + fp)

accuracy = (tp + tn) / (tp + tn + fp + fn)

precision = (tp) / (tp + fp)

recall = sensitivity

f1\_score = 2 \* (precision \* recall) / (precision + recall)

print("Sensitivity:", sensitivity)

print("Specificity:", specificity)

print("Accuracy:", accuracy)

print("Precision:", precision)

print("Recall:", recall)

print("F1-score:", f1\_score)