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 sklearn.pipeline import Pipeline

from sklearn.neighbors import KNeighborsClassifier

from imblearn.pipeline import Pipeline as imbpipeline

from xgboost.sklearn import XGBClassifier

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import ColumnTransformer

import xgboost as xgb

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier

from lightgbm import LGBMClassifier

import lightgbm as lgb

from sklearn.metrics import classification\_report

from imblearn.over\_sampling import SMOTE

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

from sklearn.utils import compute\_class\_weight

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

from joblib import dump, load

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

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

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

# Convert Year and Month to datetime

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

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

# Extract month and year components

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

df['year'] = df['Year'].dt.year # Extract the year

# Drop old columns (if you don't need them anymore)

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

# Prepare for LGBM

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

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

weights = [0.3, 0.5, 0.7, 0.9]

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

 ['classifier', RandomForestClassifier()]])

stratified\_kfold = StratifiedKFold(n\_splits=10,

 shuffle=True,

 random\_state=11)

param\_grid = {'classifier\_\_max\_depth':[3, 5, 7],

 'classifier\_\_min\_samples\_split':[2, 5, 10],

 'classifier\_\_max\_features':[0.1, 0.2, 0.4],

 'classifier\_\_criterion':["gini", "entropy"],

 'classifier\_\_bootstrap':[True, False],

 'classifier\_\_n\_estimators':[10, 50, 100],

 'classifier\_\_min\_samples\_leaf':[1, 2, 4, 6]}

grid\_search = GridSearchCV(estimator=pipeline,

 param\_grid=param\_grid,

 scoring='roc\_auc',

 cv=stratified\_kfold,

 n\_jobs=-1)

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

#Execute Training

import time

start\_time = time.time()

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

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

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

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()

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

# retrieve the best estimator from the grid search

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

# extract the XGBClassifier from the pipeline

xgb\_clf = best\_estimator.named\_steps['classifier']

# get the feature importances

importances = xgb\_clf.feature\_importances\_

print(importances)

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

df = df.rename(columns={'P': 'Precipitation','T': 'Land Surface Temperature', 'month': 'Month'})

feature\_names = df.drop('H', axis=1).columns

print(len(feature\_names))

print(len(importances))

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

# Assign feature names

#feature\_names = df.drop('H', axis=1).columns

# create a DataFrame with feature importances and feature names as columns

importance\_df = pd.DataFrame(data={'feature\_names': feature\_names, 'importances': importances})

importance\_df.sort\_values(by='importances', ascending=False, inplace=True)

feature\_importances\_timeseries = np.array(importance\_df)

# Create a bar chart of feature importances

plt.figure(figsize=(12,6))

plt.bar(x=np.arange(importance\_df.shape[0]), height=importance\_df['importances'])

plt.xticks(np.arange(importance\_df.shape[0]), importance\_df['feature\_names'], rotation=90) # Key line

plt.xlabel('Features')

plt.ylabel('Importance Score')

plt.title('Feature Importances')

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)

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