**1) 72-hour fatality rate by serum phosphorus concentration**

This program analyzes the fatality rates stratified by serum inorganic phosphate levels for training and validation datasets. It performs chi-square tests to compare the fatality rates between different strata and visualizes the results using bar graphs with error bars.

## Features

- Load training and validation datasets.

- Stratify serum phosphate levels into predefined bins.

- Calculate stratified counts and overall counts for death and alive outcomes.

- Compute fatality rates and their associated error bars.

- Perform chi-square tests for each stratum and overall.

- Plot fatality rates as bar graphs with error bars.

- Save the generated plot as a PNG file.

## Program Name: ‘serum\_phosphate\_fatality\_rate\_analysis.py’

# Import necessary libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from scipy.stats import chi2\_contingency

# Load data

df\_train = pd.read\_csv("/training dataset.csv")

df\_test = pd.read\_csv("/test dataset.csv")

# Extract relevant columns

serum\_phosphate\_train = df\_train.iloc[:, 27].astype(float) # AB row

outcome\_train = df\_train.iloc[:, 5].astype(int) # F row

serum\_phosphate\_test = df\_test.iloc[:, 27].astype(float) # AB row

outcome\_test = df\_test.iloc[:, 5].astype(int) # F row

# Define new bins for stratification

bins = [10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 35.0]

labels = ["10.0", "10.1 - 11.0", "11.1 - 12.0", "12.1 - 13.0", "13.1 - 14.0", "14.1 -"]

# Stratify data

train\_stratified = pd.cut(serum\_phosphate\_train, bins=bins, labels=labels, right=False)

test\_stratified = pd.cut(serum\_phosphate\_test, bins=bins, labels=labels, right=False)

# Function to calculate stratified counts

def calculate\_stratified\_counts(outcome, stratified):

stratified\_counts = pd.DataFrame(index=labels, columns=['Death', 'Alive', 'Total'])

stratified\_counts['Death'] = stratified[outcome == 1].value\_counts().reindex(labels, fill\_value=0)

stratified\_counts['Alive'] = stratified[outcome == 0].value\_counts().reindex(labels, fill\_value=0)

stratified\_counts['Total'] = stratified\_counts['Death'] + stratified\_counts['Alive']

return stratified\_counts

# Calculate counts for training data

train\_counts = calculate\_stratified\_counts(outcome\_train, train\_stratified)

# Calculate counts for test data

test\_counts = calculate\_stratified\_counts(outcome\_test, test\_stratified)

# Calculate overall counts

overall\_train\_counts = pd.DataFrame({

'Death': [outcome\_train[outcome\_train == 1].count()],

'Alive': [outcome\_train[outcome\_train == 0].count()],

'Total': [len(outcome\_train)]

}, index=['Overall'])

overall\_test\_counts = pd.DataFrame({

'Death': [outcome\_test[outcome\_test == 1].count()],

'Alive': [outcome\_test[outcome\_test == 0].count()],

'Total': [len(outcome\_test)]

}, index=['Overall'])

# Calculate fatality rates with error bars

def calculate\_fatality\_rate\_and\_error(counts):

fatality\_rate = counts.copy()

fatality\_rate['Fatality Rate'] = fatality\_rate['Death'] / fatality\_rate['Total']

fatality\_rate['Error'] = np.sqrt(fatality\_rate['Fatality Rate'] \* (1 - fatality\_rate['Fatality Rate']) / fatality\_rate['Total'])

return fatality\_rate

train\_fatality\_rate = calculate\_fatality\_rate\_and\_error(train\_counts)

test\_fatality\_rate = calculate\_fatality\_rate\_and\_error(test\_counts)

overall\_train\_fatality\_rate = calculate\_fatality\_rate\_and\_error(overall\_train\_counts)

overall\_test\_fatality\_rate = calculate\_fatality\_rate\_and\_error(overall\_test\_counts)

# Combine overall fatality rates with stratified rates

train\_fatality\_rate = pd.concat([train\_fatality\_rate, overall\_train\_fatality\_rate])

test\_fatality\_rate = pd.concat([test\_fatality\_rate, overall\_test\_fatality\_rate])

# Chi-square test for each stratum

chi2\_results = pd.DataFrame(index=labels + ['Overall'], columns=['Chi2', 'p-value'])

for label in labels + ['Overall']:

if label == 'Overall':

train\_death = overall\_train\_counts.loc[label, 'Death']

train\_alive = overall\_train\_counts.loc[label, 'Alive']

test\_death = overall\_test\_counts.loc[label, 'Death']

test\_alive = overall\_test\_counts.loc[label, 'Alive']

else:

train\_death = train\_counts.loc[label, 'Death']

train\_alive = train\_counts.loc[label, 'Alive']

test\_death = test\_counts.loc[label, 'Death']

test\_alive = test\_counts.loc[label, 'Alive']

contingency\_table = np.array([[train\_death, train\_alive], [test\_death, test\_alive]])

chi2, p, \_, \_ = chi2\_contingency(contingency\_table)

chi2\_results.loc[label] = [chi2, p]

# Display results

print("Training Data Stratified Counts:")

print(train\_counts)

print("\nValidation Data Stratified Counts:")

print(test\_counts)

print("\nOverall Training Data Counts:")

print(overall\_train\_counts)

print("\nOverall Validation Data Counts:")

print(overall\_test\_counts)

print("\nTraining Data Fatality Rate:")

print(train\_fatality\_rate)

print("\nValidation Data Fatality Rate:")

print(test\_fatality\_rate)

print("\nChi-square Test Results:")

print(chi2\_results)

# Plotting the Fatality Rates as bar graphs

x = np.arange(len(labels) + 1) # the label locations including 'Overall'

width = 0.35 # the width of the bars

fig, ax = plt.subplots(figsize=(14, 8))

rects1 = ax.bar(x - width/2, train\_fatality\_rate['Fatality Rate'], width, yerr=train\_fatality\_rate['Error'], label='Training Data', color='lightgrey', capsize=5)

rects2 = ax.bar(x + width/2, test\_fatality\_rate['Fatality Rate'], width, yerr=test\_fatality\_rate['Error'], label='Validation Data', color='darkgrey', capsize=5)

# Add some text for labels, title and custom x-axis tick labels, etc.

ax.set\_xlabel('Serum Phosphate (mg/dL)')

ax.set\_ylabel('Fatality Rate')

ax.set\_title('Serum Phosphate Stratified Fatality Rates')

ax.set\_xticks(x)

ax.set\_xticklabels(labels + ['Overall'])

ax.legend()

# Customize the grid

ax.grid(True, which='both', linestyle='--', linewidth=0.5)

# Function to add labels on bars

def autolabel(rects, errors):

"""Attach a text label below each bar in \*rects\*, displaying its height."""

for rect, error in zip(rects, errors):

height = rect.get\_height()

ax.annotate(f'{height:.2%}',

xy=(rect.get\_x() + rect.get\_width() / 2, height - error - 0.03),

xytext=(0, -3), # 3 points vertical offset below

textcoords="offset points",

ha='center', va='top')

autolabel(rects1, train\_fatality\_rate['Error'])

autolabel(rects2, test\_fatality\_rate['Error'])

fig.tight\_layout()

# Save and show the plot

plt.savefig('serum\_phosphate\_fatality\_rate\_bar\_plot.png')

plt.show()

**2) ROC curve, Bootstrap, and Calibration plot for MLRA model (Model 1)**

This program analyzes the prediction of 72-hour fatality in patients using inorganic phosphorus levels along with other biochemical markers. It evaluates the model performance using various metrics, performs internal validation using bootstrap resampling, and visualizes the results with ROC and calibration plots.

## Features

- Load and preprocess training and validation datasets.

- Calculate prediction probabilities using logistic regression.

- Evaluate model performance using metrics such as accuracy, precision, recall, F1 score, AUC, log loss, MCC, and Cohen's Kappa.

- Perform bootstrap resampling to calculate 95% confidence intervals for AUC and other metrics.

- Generate ROC and calibration plots.

- Display regression statistics for calibration.

## Program Name: ‘serum\_phosphate\_fatality\_prediction\_mlra.py’

import pandas as pd

import numpy as np

from sklearn.metrics import (roc\_curve, auc, accuracy\_score, f1\_score,

precision\_score, recall\_score, log\_loss,

matthews\_corrcoef, cohen\_kappa\_score, roc\_auc\_score)

from sklearn.utils import resample

from sklearn.calibration import calibration\_curve

import matplotlib.pyplot as plt

from statsmodels.tools.tools import add\_constant

from scipy import stats

# Load data

df\_train = pd.read\_csv("/training dataset.csv")

df\_test = pd.read\_csv("/test dataset.csv")

# Remove rows with missing values

df\_train = df\_train.dropna()

df\_test = df\_test.dropna()

# Set explanatory and target variables

X\_train = df\_train[['Age', 'AST', 'Albumin', 'K', 'Mg']]

X\_test = df\_test[['Age', 'AST', 'Albumin', 'K', 'Mg']]

y\_train = df\_train['Outcome']

y\_test = df\_test['Outcome']

# Add intercept to explanatory variables

X\_train = add\_constant(X\_train)

X\_test = add\_constant(X\_test)

# Calculate prediction probabilities

def predict\_proba(X):

beta = np.array([-5.83600, 0.03131, 0.48150, -1.05000, 0.29570, 1.53000])

z = np.dot(X, beta)

p = 1 / (1 + np.exp(-z))

return p

# Calculate prediction probabilities

y\_train\_pred\_proba = predict\_proba(X\_train)

y\_test\_pred\_proba = predict\_proba(X\_test)

# ROC curve and AUC for training data

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

auc\_score\_train = auc(fpr\_train, tpr\_train)

optimal\_idx\_train = np.argmax(tpr\_train - fpr\_train)

optimal\_threshold\_train = thresholds\_train[optimal\_idx\_train]

sensitivity\_train = tpr\_train[optimal\_idx\_train]

specificity\_train = 1 - fpr\_train[optimal\_idx\_train]

# ROC curve and AUC for test data

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

auc\_score\_test = auc(fpr\_test, tpr\_test)

optimal\_idx\_test = np.argmax(tpr\_test - fpr\_test)

optimal\_threshold\_test = thresholds\_test[optimal\_idx\_test]

sensitivity\_test = tpr\_test[optimal\_idx\_test]

specificity\_test = 1 - fpr\_test[optimal\_idx\_test]

# Calculate evaluation metrics

y\_train\_pred = (y\_train\_pred\_proba > optimal\_threshold\_train).astype(int)

y\_test\_pred = (y\_test\_pred\_proba > optimal\_threshold\_test).astype(int)

accuracy\_train = accuracy\_score(y\_train, y\_train\_pred)

f1\_train = f1\_score(y\_train, y\_train\_pred)

precision\_train = precision\_score(y\_train, y\_train\_pred)

recall\_train = recall\_score(y\_train, y\_train\_pred)

accuracy\_test = accuracy\_score(y\_test, y\_test\_pred)

f1\_test = f1\_score(y\_test, y\_test\_pred)

precision\_test = precision\_score(y\_test, y\_test\_pred)

recall\_test = recall\_score(y\_test, y\_test\_pred)

# Calculate Log Loss

log\_loss\_train = log\_loss(y\_train, y\_train\_pred\_proba)

log\_loss\_test = log\_loss(y\_test, y\_test\_pred\_proba)

# Calculate MCC

mcc\_train = matthews\_corrcoef(y\_train, y\_train\_pred)

mcc\_test = matthews\_corrcoef(y\_test, y\_test\_pred)

# Calculate Cohen's Kappa

kappa\_train = cohen\_kappa\_score(y\_train, y\_train\_pred)

kappa\_test = cohen\_kappa\_score(y\_test, y\_test\_pred)

# Calculate 95% CI for AUC using Bootstrap

def calculate\_auc\_ci(y\_true, y\_pred\_proba, n\_bootstraps=1000, alpha=0.95):

rng = np.random.RandomState(42)

bootstrapped\_scores = []

for \_ in range(n\_bootstraps):

indices = rng.randint(0, len(y\_pred\_proba), len(y\_pred\_proba))

if len(np.unique(y\_true[indices])) < 2:

continue

score = roc\_auc\_score(y\_true[indices], y\_pred\_proba[indices])

bootstrapped\_scores.append(score)

sorted\_scores = np.array(bootstrapped\_scores)

sorted\_scores.sort()

lower = sorted\_scores[int((1.0 - alpha) / 2.0 \* len(sorted\_scores))]

upper = sorted\_scores[int((alpha + (1.0 - alpha) / 2.0) \* len(sorted\_scores))]

return lower, upper

ci\_train = calculate\_auc\_ci(y\_train, y\_train\_pred\_proba)

ci\_test = calculate\_auc\_ci(y\_test, y\_test\_pred\_proba)

# Function to calculate 95% CI for any metric

def calculate\_metric\_ci(metric\_values, alpha=0.95):

sorted\_scores = np.array(metric\_values)

sorted\_scores.sort()

lower = sorted\_scores[int((1.0 - alpha) / 2.0 \* len(sorted\_scores))]

upper = sorted\_scores[int((alpha + (1.0 - alpha) / 2.0) \* len(sorted\_scores))]

return lower, upper

# Bootstrap internal validation

n\_bootstraps = 1000

bootstrapped\_auc\_scores = []

bootstrapped\_accuracy\_scores = []

bootstrapped\_sensitivity\_scores = []

bootstrapped\_specificity\_scores = []

bootstrapped\_precision\_scores = []

bootstrapped\_f1\_scores = []

example\_fpr = None

example\_tpr = None

for i in range(n\_bootstraps):

X\_train\_resampled, y\_train\_resampled = resample(X\_train, y\_train)

y\_train\_pred\_proba\_resampled = predict\_proba(X\_train\_resampled)

fpr\_resampled, tpr\_resampled, thresholds\_resampled = roc\_curve(y\_train\_resampled, y\_train\_pred\_proba\_resampled)

auc\_score\_resampled = auc(fpr\_resampled, tpr\_resampled)

optimal\_idx\_resampled = np.argmax(tpr\_resampled - fpr\_resampled)

optimal\_threshold\_resampled = thresholds\_resampled[optimal\_idx\_resampled]

y\_train\_pred\_resampled = (y\_train\_pred\_proba\_resampled > optimal\_threshold\_resampled).astype(int)

bootstrapped\_auc\_scores.append(auc\_score\_resampled)

bootstrapped\_accuracy\_scores.append(accuracy\_score(y\_train\_resampled, y\_train\_pred\_resampled))

bootstrapped\_sensitivity\_scores.append(recall\_score(y\_train\_resampled, y\_train\_pred\_resampled))

bootstrapped\_specificity\_scores.append(1 - fpr\_resampled[optimal\_idx\_resampled])

bootstrapped\_precision\_scores.append(precision\_score(y\_train\_resampled, y\_train\_pred\_resampled))

bootstrapped\_f1\_scores.append(f1\_score(y\_train\_resampled, y\_train\_pred\_resampled))

if i == 0:

example\_fpr = fpr\_resampled

example\_tpr = tpr\_resampled

mean\_bootstrap\_auc = np.mean(bootstrapped\_auc\_scores)

ci\_bootstrap\_auc = calculate\_metric\_ci(bootstrapped\_auc\_scores)

mean\_bootstrap\_accuracy = np.mean(bootstrapped\_accuracy\_scores)

ci\_bootstrap\_accuracy = calculate\_metric\_ci(bootstrapped\_accuracy\_scores)

mean\_bootstrap\_sensitivity = np.mean(bootstrapped\_sensitivity\_scores)

ci\_bootstrap\_sensitivity = calculate\_metric\_ci(bootstrapped\_sensitivity\_scores)

mean\_bootstrap\_specificity = np.mean(bootstrapped\_specificity\_scores)

ci\_bootstrap\_specificity = calculate\_metric\_ci(bootstrapped\_specificity\_scores)

mean\_bootstrap\_precision = np.mean(bootstrapped\_precision\_scores)

ci\_bootstrap\_precision = calculate\_metric\_ci(bootstrapped\_precision\_scores)

mean\_bootstrap\_f1 = np.mean(bootstrapped\_f1\_scores)

ci\_bootstrap\_f1 = calculate\_metric\_ci(bootstrapped\_f1\_scores)

apparent\_performance = auc\_score\_train

optimism = mean\_bootstrap\_auc - auc\_score\_train

corrected\_performance = auc\_score\_train - optimism

# Output results

print('\*\*Training Data\*\*')

print('Sensitivity:', sensitivity\_train)

print('Specificity:', specificity\_train)

print('Accuracy:', accuracy\_train)

print('F1 Score:', f1\_train)

print('Precision:', precision\_train)

print('Recall:', recall\_train)

print('AUC:', auc\_score\_train)

print('95% CI (AUC):', ci\_train)

print('Log Loss:', log\_loss\_train)

print('MCC:', mcc\_train)

print('Cohen\'s Kappa:', kappa\_train)

print('\*\*Test Data\*\*')

print('Sensitivity:', sensitivity\_test)

print('Specificity:', specificity\_test)

print('Accuracy:', accuracy\_test)

print('F1 Score:', f1\_test)

print('Precision:', precision\_test)

print('Recall:', recall\_test)

print('AUC:', auc\_score\_test)

print('95% CI (AUC):', ci\_test)

print('Log Loss:', log\_loss\_test)

print('MCC:', mcc\_test)

print('Cohen\'s Kappa:', kappa\_test)

print('\*\*Bootstrap Results\*\*')

print('Bootstrap Mean AUC:', mean\_bootstrap\_auc)

print('Bootstrap 95% CI (AUC):', ci\_bootstrap\_auc)

print('Bootstrap Mean Accuracy:', mean\_bootstrap\_accuracy)

print('Bootstrap 95% CI (Accuracy):', ci\_bootstrap\_accuracy)

print('Bootstrap Mean Sensitivity:', mean\_bootstrap\_sensitivity)

print('Bootstrap 95% CI (Sensitivity):', ci\_bootstrap\_sensitivity)

print('Bootstrap Mean Specificity:', mean\_bootstrap\_specificity)

print('Bootstrap 95% CI (Specificity):', ci\_bootstrap\_specificity)

print('Bootstrap Mean Precision:', mean\_bootstrap\_precision)

print('Bootstrap 95% CI (Precision):', ci\_bootstrap\_precision)

print('Bootstrap Mean F1 Score:', mean\_bootstrap\_f1)

print('Bootstrap 95% CI (F1 Score):', ci\_bootstrap\_f1)

print('Apparent performance:', apparent\_performance)

print('Optimism:', optimism)

print('Corrected performance:', corrected\_performance)

# Plot ROC curve

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

plt.plot(fpr\_train, tpr\_train, 'k-', label='Training AUC = %0.3f' % auc\_score\_train)

plt.plot(fpr\_test, tpr\_test, 'k--', label='Validation AUC = %0.3f' % auc\_score\_test)

plt.xlabel('1 - specificity')

plt.ylabel('Sensitivity')

plt.title('ROC Curve')

plt.legend(loc='lower right')

plt.show()

# Plot Bootstrap AUC distribution

plt.figure(figsize=(10, 6))

plt.hist(bootstrapped\_auc\_scores, bins=50, color='lightgrey', edgecolor='k', alpha=0.7)

plt.xlabel('AUC')

plt.ylabel('Frequency')

plt.title('Bootstrap AUC Distribution')

plt.axvline(x=mean\_bootstrap\_auc, color='k', linestyle='--', label='Mean AUC = %0.3f' % mean\_bootstrap\_auc)

plt.legend(loc='lower right')

plt.show()

# Plot Calibration plot (Training and Test data)

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

prob\_true\_train, prob\_pred\_train = calibration\_curve(y\_train, y\_train\_pred\_proba, n\_bins=10)

prob\_true\_test, prob\_pred\_test = calibration\_curve(y\_test, y\_test\_pred\_proba, n\_bins=10)

# Calculate and plot regression lines

slope\_train, intercept\_train, r\_value\_train, p\_value\_train, std\_err\_train = stats.linregress(prob\_pred\_train, prob\_true\_train)

slope\_test, intercept\_test, r\_value\_test, p\_value\_test, std\_err\_test = stats.linregress(prob\_pred\_test, prob\_true\_test)

plt.plot(prob\_pred\_train, prob\_true\_train, 'ko', label='Training Data')

plt.plot(prob\_pred\_test, prob\_true\_test, 'k^', label='Validation Data')

plt.plot([0, 1], [0, 1], 'k:', linewidth=1, label='Perfect calibration')

plt.plot(prob\_pred\_train, intercept\_train + slope\_train \* np.array(prob\_pred\_train), 'k-', label=f'Train fit (slope={slope\_train:.3f}, intercept={intercept\_train:.3f})')

plt.plot(prob\_pred\_test, intercept\_test + slope\_test \* np.array(prob\_pred\_test), 'k--', label=f'Validation fit (slope={slope\_test:.3f}, intercept={intercept\_test:.3f})')

plt.xlabel('Mean predicted probability')

plt.ylabel('Fraction of positives')

plt.title('Calibration Plot')

plt.legend(loc='lower right')

plt.show()

# Output regression statistics

print('\*\*Training Data Regression Statistics\*\*')

print(f'Intercept: {intercept\_train}')

print(f'Slope: {slope\_train}')

print(f'R-squared: {r\_value\_train\*\*2}')

print(f'P-value: {p\_value\_train}')

print(f'Standard Error: {std\_err\_train}')

print('\*\*Validation Data Regression Statistics\*\*')

print(f'Intercept: {intercept\_test}')

print(f'Slope: {slope\_test}')

print(f'R-squared: {r\_value\_test\*\*2}')

print(f'P-value: {p\_value\_test}')

print(f'Standard Error: {std\_err\_test}')

# Plot Bootstrap ROC curve (example of the first bootstrap sample)

plt.figure(figsize=(10, 6))

plt.plot(example\_fpr, example\_tpr, 'k-', label='Bootstrap Sample ROC Curve')

plt.xlabel('1 - specificity')

plt.ylabel('Sensitivity')

plt.title('Bootstrap ROC Curve Example')

plt.legend(loc='lower right')

plt.show()

# Output binning range and number of bins

bins = np.linspace(0, 1, 11)

print('Binning range:', bins)

print('Number of bins:', len(bins) - 1)

# Output actual observed probabilities in each bin (Training data)

print('\*\*Binning information for Training Data\*\*')

for i in range(len(prob\_true\_train)):

print(f'Bin {i+1}: Predicted probability range = [{bins[i]}, {bins[i+1]}], Observed probability = {prob\_true\_train[i]}')

# Output actual observed probabilities in each bin (Test data)

print('\*\*Binning information for Test Data\*\*')

for i in range(len(prob\_true\_test)):

print(f'Bin {i+1}: Predicted probability range = [{bins[i]}, {bins[i+1]}], Observed probability = {prob\_true\_test[i]}')

**3) ROC curves, and Calibration plot for Prediction OneTM (Model 2)**

This program evaluates the prediction performance of Model 2 developed by Prediction One using ROC curves, calibration plots, and various evaluation metrics. It works with provided training and validation datasets, calculates optimal thresholds, and visualizes the results.

## Features

- Load and preprocess training and validation datasets.

- Calculate ROC curves and AUC for training and validation data.

- Compute various evaluation metrics including accuracy, precision, recall, F1 score, log loss, MCC, and Cohen's Kappa.

- Determine optimal thresholds for classification.

- Plot ROC curves and calibration plots with regression lines.

- Calculate and display 95% confidence intervals for AUC.

## Program Name: ‘prediction\_performance\_prediction\_one\_analysis.py’

# Importing necessary libraries

import pandas as pd

import numpy as np

from sklearn.metrics import (

roc\_curve, auc, accuracy\_score, f1\_score, precision\_score, recall\_score,

log\_loss, matthews\_corrcoef, cohen\_kappa\_score

)

from sklearn.calibration import calibration\_curve

import matplotlib.pyplot as plt

from scipy.stats import linregress

# Load data

df\_train = pd.read\_csv("/training dataset.csv")

df\_test = pd.read\_csv("/testing dataset.csv")

# Define target variable and predicted probabilities

y\_train = df\_train.iloc[:, 7] # H row "Outcome"

y\_train\_pred\_proba = df\_train.iloc[:, 1] # B row "1"

y\_test = df\_test.iloc[:, 15] # P row "Outcome"

y\_test\_pred\_proba = df\_test.iloc[:, 3] # D row "1"

# ROC curve and AUC for training data

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

auc\_score\_train = auc(fpr\_train, tpr\_train)

# ROC curve and AUC for validation data

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

auc\_score\_test = auc(fpr\_test, tpr\_test)

# Calculate evaluation metrics

optimal\_idx\_train = np.argmax(tpr\_train - fpr\_train)

optimal\_threshold\_train = thresholds\_train[optimal\_idx\_train]

y\_train\_pred = (y\_train\_pred\_proba >= optimal\_threshold\_train).astype(int)

metrics\_train = {

'sensitivity': tpr\_train[optimal\_idx\_train],

'specificity': 1 - fpr\_train[optimal\_idx\_train],

'accuracy': accuracy\_score(y\_train, y\_train\_pred),

'f1\_score': f1\_score(y\_train, y\_train\_pred),

'precision': precision\_score(y\_train, y\_train\_pred),

'recall': recall\_score(y\_train, y\_train\_pred),

'log\_loss': log\_loss(y\_train, y\_train\_pred\_proba),

'mcc': matthews\_corrcoef(y\_train, y\_train\_pred),

'kappa': cohen\_kappa\_score(y\_train, y\_train\_pred)

}

optimal\_idx\_test = np.argmax(tpr\_test - fpr\_test)

optimal\_threshold\_test = thresholds\_test[optimal\_idx\_test]

y\_test\_pred = (y\_test\_pred\_proba >= optimal\_threshold\_test).astype(int)

metrics\_test = {

'sensitivity': tpr\_test[optimal\_idx\_test],

'specificity': 1 - fpr\_test[optimal\_idx\_test],

'accuracy': accuracy\_score(y\_test, y\_test\_pred),

'f1\_score': f1\_score(y\_test, y\_test\_pred),

'precision': precision\_score(y\_test, y\_test\_pred),

'recall': recall\_score(y\_test, y\_test\_pred),

'log\_loss': log\_loss(y\_test, y\_test\_pred\_proba),

'mcc': matthews\_corrcoef(y\_test, y\_test\_pred),

'kappa': cohen\_kappa\_score(y\_test, y\_test\_pred)

}

# Calculate AUC confidence intervals

def calculate\_auc\_ci(auc, y\_true, y\_pred\_proba, confidence=0.95):

n1 = sum(y\_true)

n2 = len(y\_true) - n1

q1 = auc / (2 - auc)

q2 = 2 \* auc\*\*2 / (1 + auc)

se\_auc = np.sqrt((auc \* (1 - auc) + (n1 - 1) \* (q1 - auc\*\*2) + (n2 - 1) \* (q2 - auc\*\*2)) / (n1 \* n2))

lower = auc - 1.96 \* se\_auc

upper = auc + 1.96 \* se\_auc

return lower, upper

ci\_train = calculate\_auc\_ci(auc\_score\_train, y\_train, y\_train\_pred\_proba)

ci\_test = calculate\_auc\_ci(auc\_score\_test, y\_test, y\_test\_pred\_proba)

# Output results

def print\_metrics(title, metrics, auc\_score, ci):

print(f'\*\*{title}\*\*')

for key, value in metrics.items():

print(f'{key.capitalize()}: {value}')

print('AUC:', auc\_score)

print('95% CI (AUC):', ci)

print\_metrics('Training Data', metrics\_train, auc\_score\_train, ci\_train)

print\_metrics('Validation Data', metrics\_test, auc\_score\_test, ci\_test)

# Plot ROC curve

plt.plot(fpr\_train, tpr\_train, 'k-', label='Training AUC = %0.3f' % auc\_score\_train) # Solid black line

plt.plot(fpr\_test, tpr\_test, 'k--', label='Validation AUC = %0.3f' % auc\_score\_test) # Dashed black line

plt.xlabel('1 - Specificity')

plt.ylabel('Sensitivity')

plt.title('ROC Curve')

plt.legend(loc='lower right')

plt.show()

# Calibration plot with regression lines

def plot\_calibration\_curve(y\_true, y\_pred\_proba, label, marker, linestyle, line\_label):

prob\_true, prob\_pred = calibration\_curve(y\_true, y\_pred\_proba, n\_bins=10)

plt.plot(prob\_pred, prob\_true, marker, label=label)

slope, intercept, r\_value, p\_value, std\_err = linregress(prob\_pred, prob\_true)

line = slope \* np.array(prob\_pred) + intercept

plt.plot(prob\_pred, line, linestyle, label=f'{line\_label} (slope={slope:.3f}, intercept={intercept:.3f})')

return slope, intercept, r\_value, p\_value, std\_err

plt.figure(figsize=(8, 8)) # Make the plot square

train\_stats = plot\_calibration\_curve(y\_train, y\_train\_pred\_proba, 'Training Data', 'ko', 'k-', 'Train fit')

test\_stats = plot\_calibration\_curve(y\_test, y\_test\_pred\_proba, 'Validation Data', 'k^', 'k--', 'Validation fit')

plt.plot([0, 1], [0, 1], 'k:', label='Perfect calibration')

plt.xlabel('Mean Predicted Probability')

plt.ylabel('Fraction of Positives')

plt.title('Calibration Plot')

plt.legend(loc='lower right')

plt.show()

# Output regression statistics

def print\_regression\_stats(title, stats):

print(f'\*\*{title} Regression Statistics\*\*')

print('Slope:', stats[0])

print('Intercept:', stats[1])

print('Correlation coefficient:', stats[2])

print('Determination coefficient:', stats[2]\*\*2)

print('p-value:', stats[3])

print('Standard error:', stats[4])

print\_regression\_stats('Training Data', train\_stats)

print\_regression\_stats('Validation Data', test\_stats)

**4) ROC curves, SHAP values, and Calibration plot for LightGBM (Model 3)**

This program analyzes the prediction performance of Model 3 developed by LightGPT for predicting 72-hour fatality in patients using inorganic phosphorus levels along with other biochemical markers. It evaluates the model performance using various metrics, performs SHAP analysis, and visualizes the results with ROC and calibration plots.

## Features

- Load and preprocess training and validation datasets.

- Train a LightGBM model with parameter tuning and early stopping.

- Calculate ROC curves, AUC, and evaluation metrics for training and validation data.

- Generate calibration plots with regression lines.

- Perform SHAP analysis for feature importance and dependence.

- Visualize results with plots.

## Program Name: ‘serum\_phosphate\_prediction\_analysis\_lightgbm.py’

# Install required libraries

!pip install shap

!pip install lightgbm

!pip install scikit-learn

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import shap

import lightgbm as lgb

from sklearn.metrics import (roc\_auc\_score, roc\_curve, accuracy\_score, f1\_score, log\_loss,

matthews\_corrcoef, cohen\_kappa\_score, precision\_score, recall\_score, precision\_recall\_curve,

average\_precision\_score)

from sklearn.calibration import calibration\_curve

from scipy.stats import linregress

# Load training data

train\_data = pd.read\_csv("/training dataset.csv", encoding="utf-8")

# Load validation data

test\_data = pd.read\_csv("/test dataset.csv", encoding="utf-8")

# Replace spaces in feature names with underscores

train\_data.columns = train\_data.columns.str.replace(' ', '\_')

test\_data.columns = test\_data.columns.str.replace(' ', '\_')

# Convert object columns to numeric, forcing errors to NaN

train\_data = train\_data.apply(pd.to\_numeric, errors='coerce')

test\_data = test\_data.apply(pd.to\_numeric, errors='coerce')

# Select features and target variable

X\_train = train\_data.iloc[:, 4:42] # E row to AP row

y\_train = (train\_data.iloc[:, 3] == 1).astype(int) # D row "Outcome" "1"

X\_test = test\_data.iloc[:, 4:42] # E row to AP row

y\_test = (test\_data.iloc[:, 3] == 1).astype(int) # D row "Outcome" "1"

# Ensure 'P' column exists

if 'P' in train\_data.columns:

interaction\_feature = 'P'

else:

raise ValueError("'P' column not found in the dataset")

# Define model with parameter tuning

model = lgb.LGBMClassifier(max\_depth=2, learning\_rate=0.01, n\_estimators=500, subsample=0.7, colsample\_bytree=0.7,

lambda\_l1=0.1, lambda\_l2=0.1, force\_col\_wise=True)

# Train the model with early stopping

callbacks = [lgb.early\_stopping(stopping\_rounds=50)]

model.fit(X\_train, y\_train, eval\_set=[(X\_train, y\_train), (X\_test, y\_test)], eval\_metric='auc', callbacks=callbacks)

# Calculate predicted probabilities

y\_train\_pred\_proba = model.predict\_proba(X\_train, num\_iteration=model.best\_iteration\_)[:, 1]

y\_test\_pred\_proba = model.predict\_proba(X\_test, num\_iteration=model.best\_iteration\_)[:, 1]

# Calculate ROC curve

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

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

# Calculate AUROC

auc\_train = roc\_auc\_score(y\_train, y\_train\_pred\_proba)

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

# Calculate standard error of AUC

def auc\_se(auc, y\_true):

n1 = np.sum(y\_true == 1)

n0 = np.sum(y\_true == 0)

q1 = auc / (2 - auc)

q2 = 2 \* auc\*\*2 / (1 + auc)

se = np.sqrt((auc \* (1 - auc) + (n1 - 1) \* (q1 - auc\*\*2) + (n0 - 1) \* (q2 - auc\*\*2)) / (n1 \* n0))

return se

# Calculate 95% confidence interval of AUC

auc\_train\_se = auc\_se(auc\_train, y\_train)

auc\_train\_ci\_lower = auc\_train - 1.96 \* auc\_train\_se

auc\_train\_ci\_upper = auc\_train + 1.96 \* auc\_train\_se

auc\_test\_se = auc\_se(auc\_test, y\_test)

auc\_test\_ci\_lower = auc\_test - 1.96 \* auc\_test\_se

auc\_test\_ci\_upper = auc\_test + 1.96 \* auc\_test\_se

# Plot ROC curve

plt.figure(figsize=(10, 6))

plt.plot(fpr\_train, tpr\_train, label=f'Training ROC curve (AUC = {auc\_train:.3f})', color='black', linestyle='-')

plt.plot(fpr\_test, tpr\_test, label=f'Validation ROC curve (AUC = {auc\_test:.3f})', color='black', linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('1 - Specificity')

plt.ylabel('Sensitivity')

plt.title('ROC Curve')

plt.legend(loc="lower right")

plt.show()

# Calculate sensitivity and specificity at optimal threshold

optimal\_idx\_train = np.argmax(tpr\_train - fpr\_train)

sensitivity\_train = tpr\_train[optimal\_idx\_train]

specificity\_train = 1 - fpr\_train[optimal\_idx\_train]

optimal\_idx\_test = np.argmax(tpr\_test - fpr\_test)

sensitivity\_test = tpr\_test[optimal\_idx\_test]

specificity\_test = 1 - fpr\_test[optimal\_idx\_test]

# Function to calculate evaluation metrics

def calculate\_metrics(y\_true, y\_pred\_proba, threshold=0.5):

y\_pred = (y\_pred\_proba >= threshold).astype(int)

accuracy = accuracy\_score(y\_true, y\_pred)

f1 = f1\_score(y\_true, y\_pred)

mcc = matthews\_corrcoef(y\_true, y\_pred)

kappa = cohen\_kappa\_score(y\_true, y\_pred)

log\_loss\_value = log\_loss(y\_true, y\_pred\_proba)

precision = precision\_score(y\_true, y\_pred)

recall = recall\_score(y\_true, y\_pred)

return accuracy, f1, mcc, kappa, log\_loss\_value, precision, recall

# Calculate Precision-Recall AUC

precision\_train, recall\_train, \_ = precision\_recall\_curve(y\_train, y\_train\_pred\_proba)

pr\_auc\_train = average\_precision\_score(y\_train, y\_train\_pred\_proba)

precision\_test, recall\_test, \_ = precision\_recall\_curve(y\_test, y\_test\_pred\_proba)

pr\_auc\_test = average\_precision\_score(y\_test, y\_test\_pred\_proba)

# Evaluation metrics for training dataset

accuracy\_train, f1\_train, mcc\_train, kappa\_train, log\_loss\_train, precision\_train, recall\_train = calculate\_metrics(y\_train, y\_train\_pred\_proba)

# Evaluation metrics for validation dataset

accuracy\_test, f1\_test, mcc\_test, kappa\_test, log\_loss\_test, precision\_test, recall\_test = calculate\_metrics(y\_test, y\_test\_pred\_proba)

# Output results

def print\_metrics(title, metrics, auc\_score, ci):

print(f'\*\*{title}\*\*')

for key, value in metrics.items():

print(f'{key.capitalize()}: {value}')

print('AUC:', auc\_score)

print('95% CI (AUC):', ci)

metrics\_train = {

'sensitivity': sensitivity\_train,

'specificity': specificity\_train,

'accuracy': accuracy\_train,

'f1\_score': f1\_train,

'precision': precision\_train,

'recall': recall\_train,

'log\_loss': log\_loss\_train,

'mcc': mcc\_train,

'kappa': kappa\_train,

'precision-recall auc': pr\_auc\_train

}

metrics\_test = {

'sensitivity': sensitivity\_test,

'specificity': specificity\_test,

'accuracy': accuracy\_test,

'f1\_score': f1\_test,

'precision': precision\_test,

'recall': recall\_test,

'log\_loss': log\_loss\_test,

'mcc': mcc\_test,

'kappa': kappa\_test,

'precision-recall auc': pr\_auc\_test

}

print\_metrics('Training Data', metrics\_train, auc\_train, (auc\_train\_ci\_lower, auc\_train\_ci\_upper))

print\_metrics('Validation Data', metrics\_test, auc\_test, (auc\_test\_ci\_lower, auc\_test\_ci\_upper))

# Calibration plot with regression lines

plt.figure(figsize=(8, 8)) # Make the plot square

def plot\_calibration\_curve(y\_true, y\_pred\_proba, label, color, linestyle, marker):

prob\_true, prob\_pred = calibration\_curve(y\_true, y\_pred\_proba, n\_bins=10)

plt.plot(prob\_pred, prob\_true, marker=marker, linestyle='none', color=color, label=label)

slope, intercept, r\_value, p\_value, std\_err = linregress(prob\_pred, prob\_true)

line = slope \* np.array(prob\_pred) + intercept

plt.plot(prob\_pred, line, linestyle, color=color, label=f'{label} fit (slope={slope:.3f}, intercept={intercept:.3f})')

return slope, intercept, r\_value, p\_value, std\_err

plot\_calibration\_curve(y\_train, y\_train\_pred\_proba, 'Training', 'black', '-', 'o')

plot\_calibration\_curve(y\_test, y\_test\_pred\_proba, 'Validation', 'black', '--', '^')

plt.plot([0, 1], [0, 1], 'k:', label='Perfect calibration')

plt.xlabel('Mean Predicted Probability')

plt.ylabel('Fraction of Positives')

plt.title('Calibration Plot')

plt.legend(loc='lower right')

plt.tight\_layout()

plt.show()

# Output regression statistics

def print\_regression\_stats(title, y\_true, y\_pred\_proba):

print(f'\*\*{title} Regression Statistics\*\*')

prob\_true, prob\_pred = calibration\_curve(y\_true, y\_pred\_proba, n\_bins=10)

slope, intercept, r\_value, p\_value, std\_err = linregress(prob\_pred, prob\_true)

print('Slope:', slope)

print('Intercept:', intercept)

print('Correlation coefficient:', r\_value)

print('Determination coefficient:', r\_value\*\*2)

print('p-value:', p\_value)

print('Standard error:', std\_err)

print\_regression\_stats('Training Data', y\_train, y\_train\_pred\_proba)

print\_regression\_stats('Validation Data', y\_test, y\_test\_pred\_proba)

# SHAP analysis

explainer = shap.Explainer(model, X\_train)

shap\_values = explainer(X\_train)

# Summary plot

shap.summary\_plot(shap\_values, X\_train)

# Output mean SHAP value of features

mean\_shap\_values = np.abs(shap\_values.values).mean(axis=0)

shap\_importance\_df = pd.DataFrame({'feature': X\_train.columns, 'mean\_SHAP\_value': mean\_shap\_values})

shap\_importance\_df = shap\_importance\_df.sort\_values(by='mean\_SHAP\_value', ascending=False)

print(shap\_importance\_df)

# SHAP dependence plot for each feature

for feature in X\_train.columns:

shap.dependence\_plot(feature, shap\_values.values, X\_train, interaction\_index=interaction\_feature)

# Bar plot of mean SHAP values

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

bars = plt.barh(shap\_importance\_df['feature'], shap\_importance\_df['mean\_SHAP\_value'], color='gray')

plt.xlabel('Mean SHAP Value (Impact on Model Output)')

plt.title('Mean SHAP Values of Features')

plt.gca().invert\_yaxis()

# Add SHAP values to the right of the bars

for bar in bars:

plt.text(bar.get\_width(), bar.get\_y() + bar.get\_height()/2, f'{bar.get\_width():.3f}', va='center')

plt.tight\_layout()

plt.show()