**MissForest**

!pip install missingpy

!pip install scikit-learn==1.1.2

import sklearn

print(sklearn.\_\_version\_\_)

import sys

import sklearn.neighbors.\_base

sys.modules['sklearn.neighbors.base'] = sklearn.neighbors.\_base

import pandas as pd

import numpy as np

from missingpy import MissForest

# Load the dataset

df = pd.read\_csv('travel-times.csv')

df.isna().sum()

# Save the ID variable in a separate variable

id\_var = df['id']

# Drop the ID variable from the dataset

df = df.drop('id', axis=1)

from missingpy import MissForest

# Impute the missing values in the relevant columns

mfimputer = MissForest(criterion='squared\_error', max\_features=None)

df\_pimputed = mfimputer.fit\_transform(df)

# Convert the imputed values back to a DataFrame

df\_pimputed = pd.DataFrame(df\_pimputed, columns=df.columns)

# Create a list of the categorical variable names

cat\_vars = ['DayOfWeek', 'GoingTo' ]

# Select the categorical variables

df\_cat = df\_pimputed[cat\_vars]

# Select the numerical variables

num\_vars = [col for col in df.columns if col not in cat\_vars]

df\_num = df\_pimputed[num\_vars]

# Round the values in df\_cat\_imputed to the nearest integer

df\_cat = df\_cat.round()

#concatenate

df\_imputed = pd.concat([df\_num, df\_cat], axis=1)

# Add the ID variable back to the imputed dataset

df\_imputed['id'] = id\_var

# Rearrange the columns in the imputed dataframe to match the original order

df\_imputed = df\_imputed.reindex(columns=['id', 'DayOfWeek', 'GoingTo', 'Distance', 'MaxSpeed', 'AvgSpeed', 'AvgMovingSpeed', 'FuelEco0my', 'TotalTime', 'MovingTime'])

# Check for missing values

df\_imputed.isna().sum()

# Export the imputed data to Stata format

df\_imputed.to\_csv('tt.csv', index=False)

**Model Building - LightGBM**

**import** numpy **as** np

**import** pandas **as** pd

**from** lightgbm **import** LGBMClassifier

**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.over\_sampling **import** SMOTE

**from** imblearn.pipeline **import** Pipeline **as** imbpipeline

**from** sklearn.neighbors **import** KNeighborsClassifier

**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** confusion\_matrix

**from** sklearn.metrics **import** roc\_curve

**from** sklearn.metrics **import** roc\_auc\_score

**from** sklearn.compose **import** ColumnTransformer

**from** joblib **import** dump, load

In [ ]:

**import** pandas **as** pd

*# Load the initial dataset*

df **=** pd**.**read\_csv('mortality\_small.csv')

*# Calculate the proportion of each category in the 'mortality' variable*

mortality\_proportion **=** df['mortality']**.**value\_counts(normalize**=True**)

*# Display the proportion*

print(mortality\_proportion)

In [ ]:

df **=** pd**.**read\_csv('mortality\_small.csv')

print('Class Ratio:',sum(df['mortality'])**/**len(df['mortality']))

In [ ]:

df**.**shape

In [ ]:

df**.**isna()**.**sum()

In [ ]:

drop\_cols **=** ['id']

df**.**drop(columns**=**drop\_cols, inplace**=True**)

df**.**select\_dtypes(exclude**=**'O')**.**columns

In [ ]:

num\_cols **=** ['age', 'bmi', 'sys\_bp', 'dias\_bp', 'hba1c', 'ldl']

cat\_cols **=** ['sex', 'ethnic', 'retinopathy', 'ihd', 'cevd', 'nephropathy']

In [ ]:

X **=** df**.**drop('mortality', axis**=**1)

y **=** df['mortality']

print(X**.**shape)

print(y**.**shape)

In [ ]:

n\_estimators **=** [100, 300]

max\_depth **=** [3,4,5]

min\_child\_weight **=** range(1,3,1)

booster **=** ['gbdt']

base\_score **=** [0.5,0.6]

learning\_rate **=** [0.1,0.2]

objective **=** ['binary']

seed **=** [27]

gamma**=** [0.7,0.8,0.9]

colsample\_bytree**=**[0.7,0.8,0.9]

subsample**=**[0.6,0.7,0.8]

reg\_alpha **=** [1e-5,0.01,0.03]

weights **=** np**.**linspace(0.3, 0.9, 2)

num\_leaves **=** [6]

In [ ]:

lgbm\_params **=** {'classifier\_\_n\_estimators': n\_estimators, 'classifier\_\_max\_depth': max\_depth,

'classifier\_\_learning\_rate' : learning\_rate, 'classifier\_\_min\_child\_weight' : min\_child\_weight,

'classifier\_\_boosting\_type' : booster, 'classifier\_\_seed':seed,'smote\_\_sampling\_strategy': weights,

'classifier\_\_reg\_alpha':reg\_alpha, 'classifier\_\_num\_leaves':num\_leaves}

In [ ]:

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', RobustScaler(), num\_cols),

],

remainder**=**'passthrough'

)

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

('scaler', preprocessor),

('classifier', LGBMClassifier())])

stratified\_kfold **=** StratifiedKFold(n\_splits**=**5,

shuffle**=True**,

random\_state**=**11)

param\_grid **=** lgbm\_params

grid\_search **=** GridSearchCV(estimator**=**pipeline,

param\_grid**=**param\_grid,

scoring**=**'roc\_auc',

cv**=**stratified\_kfold,

n\_jobs**=-**1)

In [ ]:

**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')

In [ ]:

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}')

In [ ]:

*# 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()

In [ ]:

best\_params **=** grid\_search**.**best\_params\_

best\_params

In [ ]:

*# 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)

*# Assign feature names*

feature\_names **=** df**.**drop('mortality', 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**)

*# 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)

plt**.**xlabel('Feature')

plt**.**ylabel('Importance')

plt**.**title('Feature Importances')

plt**.**show()

In [ ]:

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**from** sklearn.metrics **import** confusion\_matrix

*# Sample data for demonstration purposes (replace this with your own data)*

*# For instance, y\_test can be the actual labels and y\_test\_proba the predicted probabilities from your model*

*# Define a list of threshold values to check*

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

*# Create empty lists to store the results*

sensitivities **=** []

specificities **=** []

accuracies **=** []

youden\_indices **=** []

*# 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]

*# Calculate sensitivity, specificity, and accuracy*

sensitivity **=** tp **/** (tp **+** fn) **if** (tp **+** fn) **!=** 0 **else** 0

specificity **=** tn **/** (tn **+** fp) **if** (tn **+** fp) **!=** 0 **else** 0

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

*# Append the results to the lists*

sensitivities**.**append(sensitivity)

specificities**.**append(specificity)

accuracies**.**append(accuracy)

*# Calculate Youden's Index and append it to the list*

youden\_index **=** sensitivity **+** specificity **-** 1

youden\_indices**.**append(youden\_index)

*# Find the threshold that maximizes Youden's Index*

optimal\_idx **=** np**.**argmax(youden\_indices)

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

max\_youden\_index **=** youden\_indices[optimal\_idx]

*# Extract corresponding sensitivity, specificity, and accuracy for the optimal threshold*

optimal\_sensitivity **=** sensitivities[optimal\_idx]

optimal\_specificity **=** specificities[optimal\_idx]

optimal\_accuracy **=** accuracies[optimal\_idx]

*# Plot the results*

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

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

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

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

*# Mark the optimal threshold with a vertical dashed line*

plt**.**axvline(x**=**optimal\_threshold, color**=**'red', linestyle**=**'--')

plt**.**text(optimal\_threshold, 0.0, f'Optimal Threshold: {optimal\_threshold:.4f}', ha**=**'left', color**=**'red')

plt**.**text(optimal\_threshold, optimal\_sensitivity **-** 0.05, f'Sensitivity: {optimal\_sensitivity:.4f}', ha**=**'left', color**=**'blue')

plt**.**text(optimal\_threshold, optimal\_specificity **-** 0., f'Specificity: {optimal\_specificity:.4f}', ha**=**'left', color**=**'orange')

plt**.**text(optimal\_threshold, optimal\_accuracy **-** 0.05, f'Accuracy: {optimal\_accuracy:.4f}', ha**=**'left', color**=**'green')

plt**.**legend()

plt**.**xlabel('Threshold')

plt**.**ylabel('Score')

plt**.**title('Performance Metrics Across Different Thresholds')

plt**.**show()

print("Optimal Threshold:", optimal\_threshold)

print("Max Youden's Index:", max\_youden\_index)

print("Optimal Sensitivity:", optimal\_sensitivity)

print("Optimal Specificity:", optimal\_specificity)

print("Optimal Accuracy:", optimal\_accuracy)

In [ ]:

*# Set the desired threshold (e.g., 0.15)*

desired\_threshold **=** 0.15

*# 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)

*# Print the confusion matrix*

print("Confusion Matrix:")

print(conf\_matrix)

pd**.**crosstab(y\_test, y\_test\_pred, rownames**=**['True'], colnames**=**['Predicted'], margins**=True**)

In [ ]:

*# Save the model to disk*

dump(grid\_search, 'mortality.joblib')

In [ ]:

*# Load the model back after closing Jupyter*

grid\_search**=**load('mortality.joblib')

**Model Building – Random Forest**

**import** numpy **as** np

**import** pandas **as** pd

**from** lightgbm **import** LGBMClassifier

**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.over\_sampling **import** SMOTE

**from** imblearn.pipeline **import** Pipeline **as** imbpipeline

**from** sklearn.neighbors **import** KNeighborsClassifier

**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** confusion\_matrix

**from** sklearn.metrics **import** roc\_curve

**from** sklearn.metrics **import** roc\_auc\_score

**from** sklearn.compose **import** ColumnTransformer

**from** joblib **import** dump, load

**from** sklearn.ensemble **import** RandomForestClassifier

In [ ]:

df **=** pd**.**read\_csv('mortality\_small.csv')

print('Class Ratio:',sum(df['mortality'])**/**len(df['mortality']))

In [ ]:

drop\_cols **=** ['id']

df**.**drop(columns**=**drop\_cols, inplace**=True**)

df**.**select\_dtypes(exclude**=**'O')**.**columns

In [ ]:

num\_cols **=** ['age', 'bmi', 'sys\_bp', 'dias\_bp', 'hba1c', 'ldl']

cat\_cols **=** ['sex', 'ethnic', 'retinopathy', 'ihd', 'cevd', 'nephropathy']

In [ ]:

X **=** df**.**drop('mortality', axis**=**1)

y **=** df['mortality']

print(X**.**shape)

print(y**.**shape)

In [ ]:

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X,

y,

test\_size**=**0.2,

stratify**=**y,

random\_state**=**11)

*#weights = np.linspace(0.3, 0.9, 7)*

weights **=** [0.3, 0.5]

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':[7],

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

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

'classifier\_\_max\_features':[0.8],

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

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

'classifier\_\_n\_estimators':[200],

'classifier\_\_min\_samples\_leaf':[8]}

grid\_search **=** GridSearchCV(estimator**=**pipeline,

param\_grid**=**param\_grid,

scoring**=**'roc\_auc',

cv**=**stratified\_kfold,

n\_jobs**=-**1)

In [ ]:

**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')

In [ ]:

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}')

In [ ]:

best\_params **=** grid\_search**.**best\_params\_

best\_params

In [ ]:

*# 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()

In [ ]:

**from** sklearn.feature\_selection **import** RFE

*# Extract the classifier from the pipeline*

classifier **=** grid\_search**.**best\_estimator\_**.**named\_steps['classifier']

*# Create an RFE model, specifying the number of features to retain*

rfe **=** RFE(estimator**=**classifier, n\_features\_to\_select**=**3, step**=**1)

*# Fit the RFE model to the data*

rfe**.**fit(X\_train, y\_train)

*# Obtain the feature ranking*

ranking **=** rfe**.**ranking\_

*# Print the feature ranking*

print(ranking)

*# Create a DataFrame with the feature names and RFE rankings*

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

feature\_importance\_df **=** pd**.**DataFrame({'feature\_name': feature\_names, 'rfe\_ranking': rfe**.**ranking\_})

*# Sort the DataFrame by rfe\_ranking*

feature\_importance\_df**.**sort\_values(by**=**'rfe\_ranking', inplace**=True**)

*# Print the DataFrame*

print(feature\_importance\_df)

In [ ]:

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**from** sklearn.metrics **import** confusion\_matrix

*# Sample data for demonstration purposes (replace this with your own data)*

*# For instance, y\_test can be the actual labels and y\_test\_proba the predicted probabilities from your model*

*# Define a list of threshold values to check*

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

*# Create empty lists to store the results*

sensitivities **=** []

specificities **=** []

accuracies **=** []

youden\_indices **=** []

*# 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]

*# Calculate sensitivity, specificity, and accuracy*

sensitivity **=** tp **/** (tp **+** fn) **if** (tp **+** fn) **!=** 0 **else** 0

specificity **=** tn **/** (tn **+** fp) **if** (tn **+** fp) **!=** 0 **else** 0

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

*# Append the results to the lists*

sensitivities**.**append(sensitivity)

specificities**.**append(specificity)

accuracies**.**append(accuracy)

*# Calculate Youden's Index and append it to the list*

youden\_index **=** sensitivity **+** specificity **-** 1

youden\_indices**.**append(youden\_index)

*# Find the threshold that maximizes Youden's Index*

optimal\_idx **=** np**.**argmax(youden\_indices)

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

max\_youden\_index **=** youden\_indices[optimal\_idx]

*# Extract corresponding sensitivity, specificity, and accuracy for the optimal threshold*

optimal\_sensitivity **=** sensitivities[optimal\_idx]

optimal\_specificity **=** specificities[optimal\_idx]

optimal\_accuracy **=** accuracies[optimal\_idx]

*# Plot the results*

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

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

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

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

*# Mark the optimal threshold with a vertical dashed line*

plt**.**axvline(x**=**optimal\_threshold, color**=**'red', linestyle**=**'--')

plt**.**text(optimal\_threshold, 0.0, f'Optimal Threshold: {optimal\_threshold:.4f}', ha**=**'left', color**=**'red')

plt**.**text(optimal\_threshold, optimal\_sensitivity **-** 0.05, f'Sensitivity: {optimal\_sensitivity:.4f}', ha**=**'left', color**=**'blue')

plt**.**text(optimal\_threshold, optimal\_specificity **-** 0., f'Specificity: {optimal\_specificity:.4f}', ha**=**'left', color**=**'orange')

plt**.**text(optimal\_threshold, optimal\_accuracy **-** 0.05, f'Accuracy: {optimal\_accuracy:.4f}', ha**=**'left', color**=**'green')

plt**.**legend()

plt**.**xlabel('Threshold')

plt**.**ylabel('Score')

plt**.**title('Performance Metrics Across Different Thresholds')

plt**.**show()

*# In[36]:*

print("Optimal Threshold:", optimal\_threshold)

print("Max Youden's Index:", max\_youden\_index)

print("Optimal Sensitivity:", optimal\_sensitivity)

print("Optimal Specificity:", optimal\_specificity)

print("Optimal Accuracy:", optimal\_accuracy)

In [ ]: