STATISTICS FOR DATA ANALYSIS

Final Project

Higher Diploma in Science in Data Analytics

Name: Eduardo Campos

Contents

Figures Index

Introduction

This project offers a systematic analysis of "MIMIC II Surgical Intensive Care" data, involving meticulous steps like library importation, data loading, scatterplot generation, outlier identification, and thorough cleaning. Advanced techniques, graphics, and models will be employed to construct a Logistic Regression predictive model, assessing its accuracy against required standards.

Step by Step of the Analysis of MIMIC II Surgical Intensive Care Unit Dataset

Step.1 Import Libraries

1) Import Libraries ¶

```
[951]: import pandas as pd
       import matplotlib.pyplot as plt
       from pandas.plotting import scatter_matrix
       import matplotlib.pyplot as plt
       import seaborn as sns
       import numpy as np
       from sklearn.model_selection import train_test_split
       from sklearn.preprocessing import StandardScaler
       import copy
       from sklearn.linear_model import LogisticRegression
       from sklearn.metrics import accuracy_score, classification_report
       from sklearn.model_selection import KFold, cross_val_score
       from sklearn.metrics import confusion_matrix
```

Figure 1The import of the libraries.

In this phase, I use essential libraries for analysis:

1. **pandas (pd):** Facilitates data manipulation and analysis with DataFrame structures.
2. **matplotlib.pyplot (plt):** Creates various plots and charts for visual data representation.
3. **pandas.plotting.scatter_matrix:** Part of pandas, aids in constructing scatter plots for variable relationships.
4. **seaborn (sns):** Extends Matplotlib, providing extra plot types and themes for enhanced visualization.
5. **numpy (np):** Supports numerical operations, handling large arrays and matrices.
6. **sklearn.model_selection.train_test_split:** Splits data into training and testing sets for machine learning model assessment.
7. **sklearn.preprocessing.StandardScaler:** Standardizes features in scikit-learn by centering them on zero and scaling to unit variance.
8. **sklearn.linear_model.LogisticRegression:** Scikit-learn's logistic regression implementation for classification tasks.
9. **sklearn.metrics.accuracy_score:** Measures classification model accuracy.
10. **sklearn.metrics.classification_report**: Generates a detailed text report with key classification metrics.

11. **sklearn.model_selection.KFold**: Executes k-fold cross-validation in scikit-learn.
12. **sklearn.model_selection.cross_val_score:** Evaluates a model's performance using cross-validated scoring.
13. **sklearn.metrics.confusion_matrix**: Computes a confusion matrix, offering insights into classification performance.

Step 2. Load Data

2) Load the Data

```
[952]: df = pd.read_excel("MIMIC II Surgical Intensive Care Unit Data.xlsx")
pd.set_option('display.max_columns', None)
#pd.reset_option('display.max_columns')
df
```

[952]:

	aline_flg	icu_los_day	hospital_los_day	age	gender_num	weight_first	bmi	sapsi_first	sofa_first	service_unit	service_num	day_icu_intime	day_icu_inti
0	1	7.63	13	72.36841	1	75.0	29.912791	15	9	SICU	1	Friday	
1	1	0.58	3	44.49191	0	0.0	0.000000	21	7	SICU	1	Saturday	
2	1	1.75	5	23.74217	1	95.2	28.464563	18	7	SICU	1	Saturday	
3	0	1.38	9	36.54657	1	72.0	23.982402	14	5	SICU	1	Sunday	
4	1	7.06	27	24.64717	1	90.0	25.474850	15	6	SICU	1	Saturday	
...	...	...	...	...	...	...	...	...	...	...	...	...	
977	0	2.27	2	50.15744	0	0.0	0.000000	14	7	SICU	1	Saturday	
978	0	1.08	3	77.96057	0	0.0	0.000000	16	5	SICU	1	Friday	
979	0	2.45	3	33.04643	1	84.0	0.000000	16	6	SICU	1	Saturday	
980	1	3.70	4	72.44020	0	60.0	0.000000	14	5	SICU	1	Saturday	
981	1	22.63	23	53.74423	1	124.4	0.000000	18	9	SICU	1	Wednesday	

982 rows × 46 columns

Figure 2. Load the data using Pandas

Using 'pandas,' I imported data from the Excel file 'MIMIC II Surgical Intensive Care Unit Data.xlsx.' The command 'pd.read_excel("MIMIC II Surgical Intensive Care Unit Data.xlsx")' loaded the file into the 'df' variable for analysis, resulting in a table with 982 rows and 46 columns.

2.1. Information of the data

With the 'df.info()' function, I can identify any null values in my dataset and review the data types. The output reveals that there are no null values in my dataset.

```
[955]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 982 entries, 0 to 981
Data columns (total 46 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   aline_flg           982 non-null    int64
 1   icu_los_day         982 non-null    float64
 2   hospital_los_day    982 non-null    int64
 3   age                 982 non-null    float64
 4   gender_num          982 non-null    int64
 5   weight_first        982 non-null    float64
 6   bmi                 982 non-null    float64
 7   sapsi_first         982 non-null    int64
 8   sofa_first          982 non-null    int64
 9   service_unit        982 non-null    object
 10  service_num         982 non-null    int64
 11  day_icu_intime      982 non-null    object
 12  day_icu_intime_num  982 non-null    int64
 13  hour_icu_intime     982 non-null    int64
 14  hosp_exp_flg        982 non-null    int64
 15  icu_exp_flg         982 non-null    int64
 16  day_28_flg          982 non-null    int64
 17  mort_day_censored   982 non-null    float64
 18  censor_flg          982 non-null    int64
 19  sepsis_flg          982 non-null    int64
 20  chf_flg             982 non-null    int64
 21  afib_flg            982 non-null    int64
 22  renal_flg           982 non-null    int64
 23  liver_flg           982 non-null    int64
 24  copd_flg            982 non-null    int64
 25  cad_flg             982 non-null    int64
 26  stroke_flg          982 non-null    int64
 27  mal_flg             982 non-null    int64
 28  resp_flg            982 non-null    int64
 29  map_1st             982 non-null    float64
 30  hr_1st              982 non-null    int64
 31  temp_1st            982 non-null    float64
 32  spo2_1st            982 non-null    int64
```

Figure 3. Information about the status of my dataset

Step 3. Create pairwise scatterplots of variables of interest

2) Create pairwise scatterplots of variables of interest

```
[956]: # Selecting the numerical variables of interest
num_vars = df.select_dtypes(include='number')
num_vars_ints = num_vars[["icu_los_day", "hospital_los_day", "age", "weight_first", "bmi", "sapsi_first", "sofa_first",
                          "map_1st", "hr_1st", "temp_1st", "spo2_1st", "abg_count", "wbc_first", "hgb_first",
                          "platelet_first", "sodium_first", "potassium_first", "tco2_first", "chloride_first",
                          "bun_first", "creatinine_first", "po2_first", "pco2_first", "iv_day_1"]]

# Assigning colors based on variables, where 0 represents no deaths in the ICU, and 1 represents deaths.
colors = df['icu_exp_flg'].map({0: 'blue', 1: 'red'})

# Creating scatter matrix
scatter_matrix(num_vars_ints, alpha=0.8, figsize=(15, 15), diagonal='hist', c=colors)

for ax in plt.gcf().axes:
    ax.xaxis.label.set_rotation(45)
    ax.yaxis.label.set_rotation(0)
    ax.yaxis.label.set_ha('right')

plt.show()
```

Figure 4. Creating a scatter matrix

I've chosen numerical values in the dataset to explore relationships using a scatter matrix. This graphical approach helps visualize correlations, and by utilizing the 'icu_exp_flg' column, I can distinguish points on the graph based on 'Deaths' or 'No Deaths' in the ICU, marking them in blue for 'No Death' and red for 'Death.'

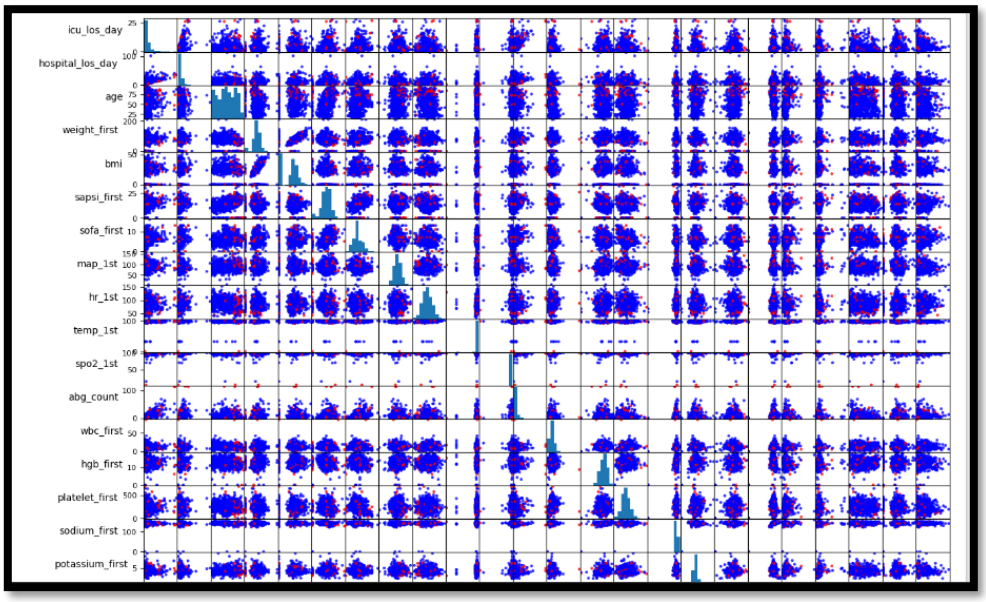


Figure 5. Displaying a Scatter Matrix to Find Relations Between Variables

I use a heatmap for a numerical assessment of correlations, providing scores to complement graphical insights. This enhances the depth of understanding by offering a quantitative measure of observed relationships in the graphics.

3.1. Heatmap of Numerical Variables

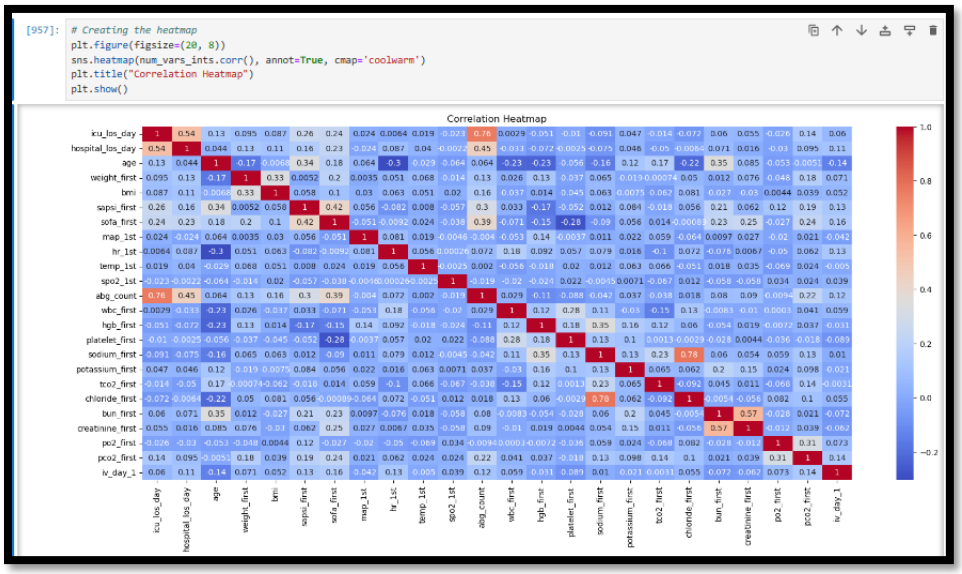
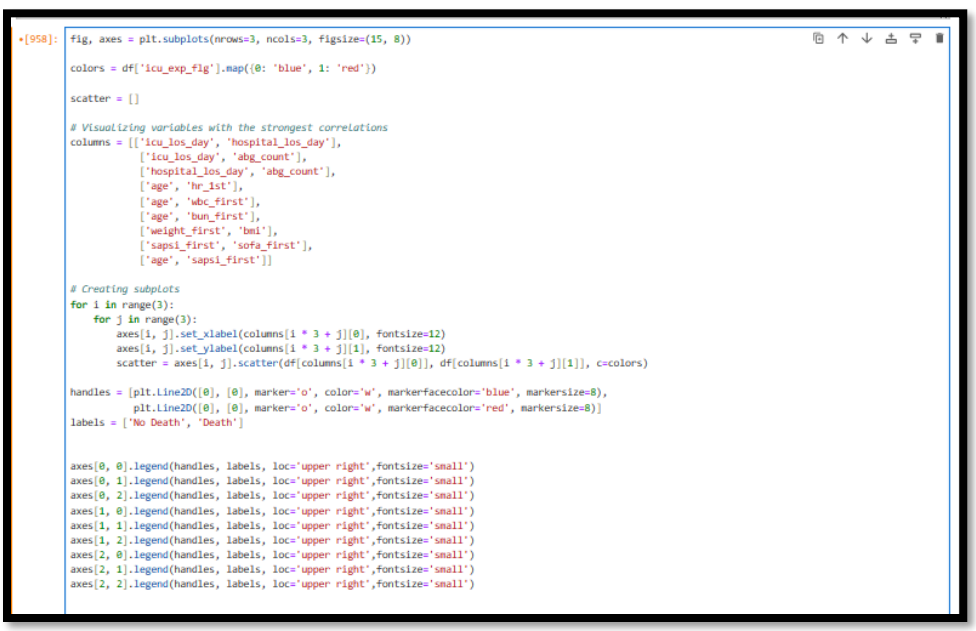


Figure 6. Heatmap of Numerical Variables

Now I proceed to graphic all those variables using matplotlib and creating subplots of the variables with strongest correlations.



```
[958]: fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(15, 8))

colors = df['icu_exp_flg'].map({0: 'blue', 1: 'red'})

scatter = []

# Visualizing variables with the strongest correlations
columns = [['icu_los_day', 'hospital_los_day'],
           ['icu_los_day', 'abg_count'],
           ['hospital_los_day', 'abg_count'],
           ['age', 'hr_1st'],
           ['age', 'wbc_first'],
           ['age', 'bun_first'],
           ['weight_first', 'bmi'],
           ['sapsi_first', 'sofa_first'],
           ['age', 'sapsi_first']]

# Creating subplots
for i in range(3):
    for j in range(3):
        axes[i, j].set_xlabel(columns[i * 3 + j][0], fontsize=12)
        axes[i, j].set_ylabel(columns[i * 3 + j][1], fontsize=12)
        scatter = axes[i, j].scatter(df[columns[i * 3 + j][0]], df[columns[i * 3 + j][1]], c=colors)

handles = [plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='blue', markersize=8),
           plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='red', markersize=8)]
labels = ['No Death', 'Death']


axes[0, 0].legend(handles, labels, loc='upper right',fontsize='small')
axes[0, 1].legend(handles, labels, loc='upper right',fontsize='small')
axes[0, 2].legend(handles, labels, loc='upper right',fontsize='small')
axes[1, 0].legend(handles, labels, loc='upper right',fontsize='small')
axes[1, 1].legend(handles, labels, loc='upper right',fontsize='small')
axes[1, 2].legend(handles, labels, loc='upper right',fontsize='small')
axes[2, 0].legend(handles, labels, loc='upper right',fontsize='small')
axes[2, 1].legend(handles, labels, loc='upper right',fontsize='small')
axes[2, 2].legend(handles, labels, loc='upper right',fontsize='small')
```

Figure 7. Creating Scatterplots for the Variables with the Best Correlations

3.2. Analysis per each Correlation between Variables

.

1- **Days Stay in ICU vs Arterial Blood Gas:** The visualization indicates higher mortality probability with extended ICU stays, especially when coupled with frequent Arterial Blood Gas (ABG) tests exceeding 50, particularly beyond 25 days. Prolonged ICU stays and increased ABG tests signal a higher likelihood of adverse outcomes, potentially leading to mortality.
2- **Age vs Heart Rate:** As depicted in this graph, as age increases, the heart rate slows down, correlating with higher mortality rates, especially after the age of 40.

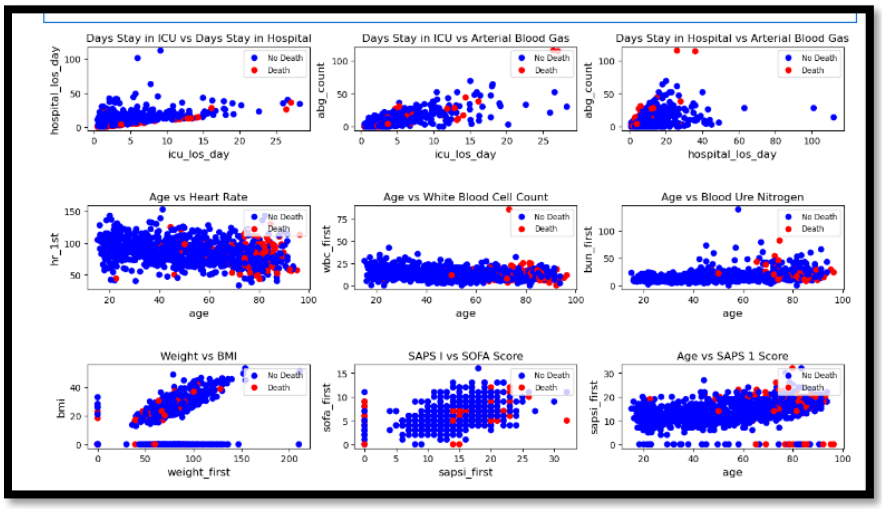


Figure 8. Initial Analysis of Variables Through Scatter Plots

3- **Distribution of SAPS I Score vs Arterial Blood Gas:** A higher SAPS I (Simplified Acute Physiology Score I) score, exceeding 15, coupled with Arterial Blood Gas tests conducted more than 50 times, indicates an increased likelihood of mortality. Review of Medical and Pharmacological Sciences, 2023).
4- **Distribution of Hemoglobin vs Sodium:** Both hemoglobin and sodium levels are crucial indicators of health. Adult reference ranges are around 12 to 16 g/dL for hemoglobin and 135 to 145 mmol/L for sodium. Abnormal levels may indicate underlying health conditions. (See Figure 12).

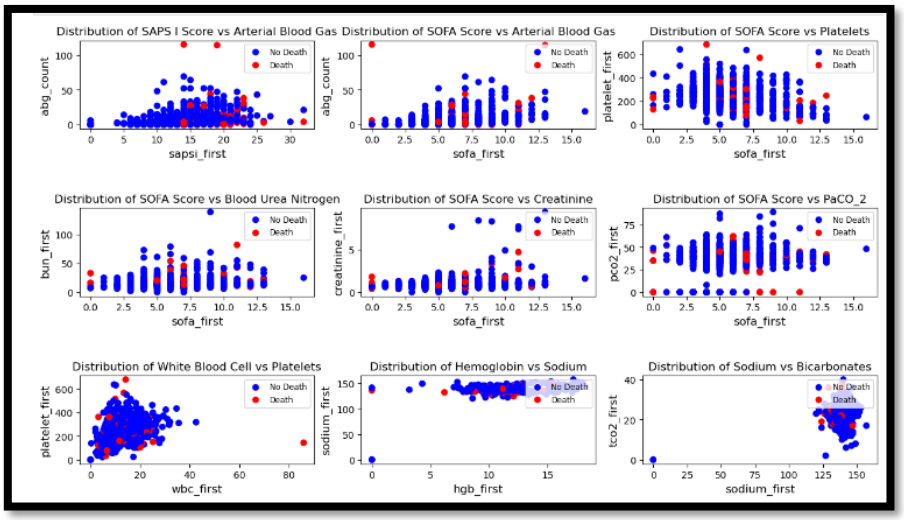


Figure 9. Second Analysis of Variables Through Scatter Plots

5- **Distribution of Sodium vs Chloride:** Sodium and chloride balance is crucial for overall health, with abnormal levels indicating potential medical conditions. Normal ranges are around 135-145 mmol/L for sodium and 95-105 mmol/L for chloride.

6- **Distribution of Blood Urea Nitrogen vs Creatinine:** BUN and creatinine levels, measured for kidney function assessment, help detect kidney disorders or dehydration. Normal ranges are approximately 7-20 mg/dL for BUN and 0.6-1.2 mg/dL for creatinine.

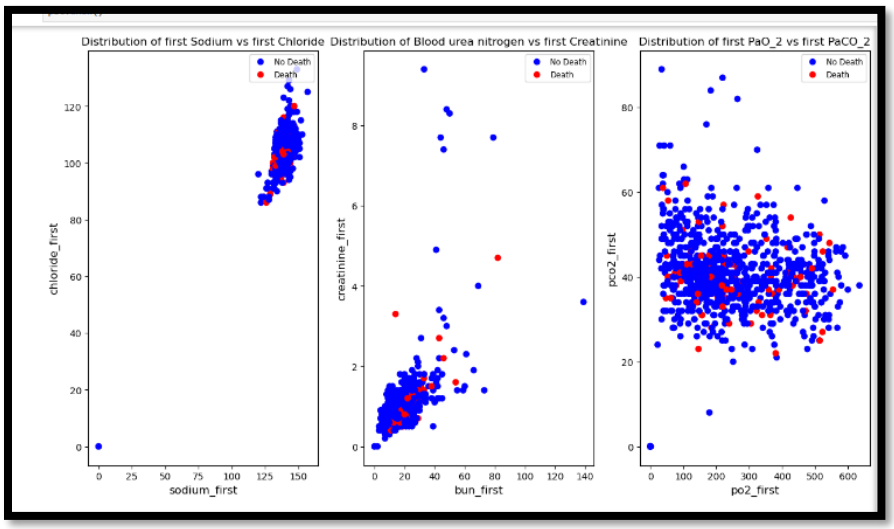


Figure 10. Third Analysis of Variables Through Scatter Plots

Step 3. Create box plots to identify Outliers

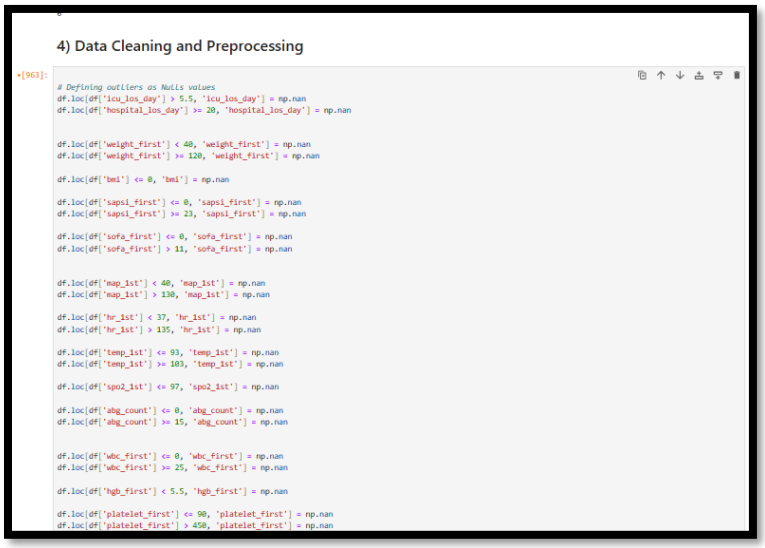
After analyzing each relationship, I proceed to create box plots using matplotlib to better identify outliers in my dataset. The variable 'num_vars_ints' contains all the previously mentioned columns, allowing me to utilize it to present all the data that could impact my analysis through modeling.

```
fig, axes = plt.subplots(nrows=6, ncols=4, figsize=(20, 15))

count = 0

for variable in num_vars_ints.columns:
    count += 1
    ax = axes.flatten()[count - 1]
    sns.boxplot(x=df[variable].dropna(), y=df['service_unit'], ax=ax, color='blue')

    ax.set_title(variable)
    ax.set_xlabel(variable)
    ax.set_ylabel('service_unit')

plt.tight_layout()
plt.show()
```

Figure 11. Identifying Outliers Through Box Plots

Step 4. Data Cleaning and Preprocessing

In the process of analyzing the output of the boxplots, the decision was made to fill the outliers in each column with null values. A range of elements was defined in this process to be converted to nulls based on the variables in the dataset.



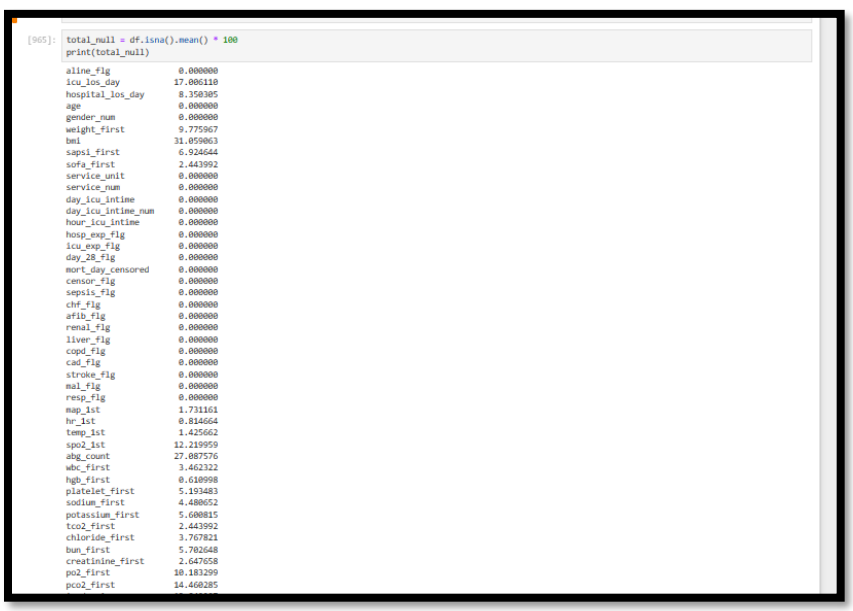
4) Data Cleaning and Preprocessing

```
[963]:
# Defining outliers as Nulls values
df.loc[df['icu_los_day'] > 5.5, 'icu_los_day'] = np.nan
df.loc[df['hospital_los_day'] >= 20, 'hospital_los_day'] = np.nan


df.loc[df['weight_first'] < 40, 'weight_first'] = np.nan
df.loc[df['weight_first'] >= 120, 'weight_first'] = np.nan

df.loc[df['bmi'] <= 0, 'bmi'] = np.nan

df.loc[df['sapsi_first'] <= 0, 'sapsi_first'] = np.nan
df.loc[df['sapsi_first'] >= 23, 'sapsi_first'] = np.nan

df.loc[df['sofa_first'] <= 0, 'sofa_first'] = np.nan
df.loc[df['sofa_first'] > 11, 'sofa_first'] = np.nan


df.loc[df['map_1st'] < 40, 'map_1st'] = np.nan
df.loc[df['map_1st'] > 130, 'map_1st'] = np.nan

df.loc[df['hr_1st'] < 37, 'hr_1st'] = np.nan
df.loc[df['hr_1st'] > 135, 'hr_1st'] = np.nan

df.loc[df['temp_1st'] <= 93, 'temp_1st'] = np.nan
df.loc[df['temp_1st'] >= 103, 'temp_1st'] = np.nan

df.loc[df['spo2_1st'] <= 97, 'spo2_1st'] = np.nan

df.loc[df['abg_count'] <= 0, 'abg_count'] = np.nan
df.loc[df['abg_count'] >= 15, 'abg_count'] = np.nan


df.loc[df['wbc_first'] <= 0, 'wbc_first'] = np.nan
df.loc[df['wbc_first'] >= 25, 'wbc_first'] = np.nan

df.loc[df['hgb_first'] < 5.5, 'hgb_first'] = np.nan

df.loc[df['platelet_first'] <= 90, 'platelet_first'] = np.nan
df.loc[df['platelet_first'] > 450, 'platelet_first'] = np.nan
```

Figure 12. Cleaning and Preprocessing the Data

4.1. Filling Nulls Values with the Mean

Now, I identify columns with null values using the 'df.isna()' function, and I obtain the percentage for each one.



```
[965]:
total_null = df.isna().mean() * 100
print(total_null)

aline_flg            0.000000
icu_los_day         17.006110
hospital_los_day     8.358305
age                  0.000000
gender_num           0.000000
weight_first         9.775967
bmi                 31.059063
sapsi_first          6.926644
sofa_first           2.443992
service_unit         0.000000
service_num          0.000000
day_icu_intime       0.000000
day_icu_intime_num   0.000000
hour_icu_intime      0.000000
hosp_exp_flg         0.000000
icu_exp_flg          0.000000
day_28_flg           0.000000
mort_day_censored    0.000000
censor_flg           0.000000
sepsis_flg           0.000000
chf_flg              0.000000
afib_flg             0.000000
renal_flg            0.000000
liver_flg            0.000000
copd_flg             0.000000
cad_flg              0.000000
stroke_flg           0.000000
mal_flg              0.000000
resp_flg             0.000000
map_1st              1.731161
hr_1st               0.814664
temp_1st             1.425662
spo2_1st            12.219959
abg_count           27.087576
wbc_first            3.462322
hgb_first            0.610998
platelet_first       5.193483
sodium_first         4.480652
potassium_first      5.608815
tco2_first           2.443992
chloride_first       3.767821
bun_first            5.702648
creatinine_first     2.647658
po2_first           10.183299
pco2_first          14.460285
```

Figure 13. Identify the new nulls values

I filled null entries with means in specific columns to address missing values and maintain data distribution. Introducing an "Age Range" categorization in 12-year increments, I applied the mean for each range to enhance accuracy in imputing values for age-related differences and metabolic variations in the dataset.

The steps for this process were:

1. **Create an Age Ranges:** I added a new column that helped to determinate the “ageRange”.

```
[968]: # Step 1: Create age ranges
       df['ageRange'] = pd.cut(df['age'], bins=np.arange(1, 101, 11), right=False)
       df
```

[968]:

_count	wbc_first	hgb_first	platelet_first	sodium_first	potassium_first	tco2_first	chloride_first	bun_first	creatinine_first	po2_first	pco2_first	iv_day_1	ageRange
NaN	8.1	14.1	354.0	138.0	4.6	15.0	109.0	NaN	1.6	196.0	39.0	2230.875000	[67, 78)
4.0	7.1	12.6	262.0	139.0	4.2	31.0	100.0	16.0	0.5	146.0	23.0	NaN	[34, 45)
9.0	4.8	10.7	NaN	146.0	3.4	19.0	110.0	10.0	1.0	134.0	30.0	2358.244141	[23, 34)
NaN	12.1	14.4	182.0	145.0	3.6	26.0	110.0	10.0	0.7	NaN	NaN	1877.560059	[34, 45)
NaN	21.6	13.4	130.0	143.0	3.8	32.0	104.0	17.0	1.3	38.0	NaN	648.599976	[23, 34)
...	...	...	...	...	...	...	...	...	...	...	...	...	...
1.0	11.7	11.2	159.0	139.0	4.0	22.0	103.0	22.0	1.2	144.0	36.0	650.000000	[45, 56)
NaN	13.7	12.2	312.0	142.0	3.3	24.0	107.0	16.0	0.8	NaN	NaN	150.000000	[67, 78)
1.0	NaN	10.2	177.0	138.0	3.7	15.0	113.0	15.0	1.2	158.0	45.0	1836.659912	[23, 34)
4.0	7.6	14.6	363.0	138.0	NaN	22.0	104.0	20.0	0.8	154.0	45.0	289.950012	[67, 78)
NaN	19.9	15.1	374.0	137.0	4.6	21.0	99.0	13.0	1.1	84.0	31.0	21.770000	[45, 56)

Figure 14. Creation of the column 'ageRange".

2. **Group by ‘ageRange’**: I group each age range established and got the mean of each one. As you can see the values are grouped in the variable ‘group_data’ (See figure 18).

```
[969]: # Step 2: Group by AgeRange
       grouped_data = df.groupby('ageRange', observed=False)

       #Print each group separately
       for name, group in grouped_data:
           print(f"Group: {name}")
           print(group[columns_to_fill])

Group: [1, 12)
Empty DataFrame
Columns: [sapsi_first, icu_los_day, hospital_los_day, bmi, weight_first, abg_count, po2_first, pco2_first, iv_day_1, spo2_1st, sofa_first, map_1st, h
r_1st, temp_1st, wbc_first, hgb_first, platelet_first, sodium_first, tco2_first, chloride_first, bun_first, creatinine_first, potassium_first]
Index: []
Group: [12, 23)
     sapsi_first  icu_los_day  hospital_los_day        bmi  weight_first  \
7           10.0          NaN              18.0  21.827509          52.4
14           9.0         1.44               1.0  28.210895          85.0
22           6.0         0.64               1.0  24.223355          75.0
31          13.0         1.71               NaN  22.555273          80.0
52          11.0         2.48               4.0  30.795974          83.0
..           ...          ...               ...        ...           ...
935          7.0         1.46               1.0        NaN          77.7
956         12.0         0.72               2.0        NaN          65.0
958         11.0         0.70               7.0        NaN         112.0
966         17.0         2.39               3.0        NaN          76.0
970         10.0          NaN               NaN        NaN          75.5
```

Figure 15. Grouped 'ageRange' values to get the mean of each one.

3. **Fill NaN values with corresponding mean for multiple numeric columns:** Now, utilizing the variable 'grouped_data' that I defined for the columns to fill

null values, I proceed to fill the nulls with the mean based on the age range. In this step, I also deleted the variable 'ageRange' as it won't be used in the model analysis.

```
[972]: # Drop the temporary AgeRange column if you don't need it
       df.drop('ageRange', axis=1, inplace=True)

[973]: df.head()
```

[973]:

	aline_flg	icu_los_day	hospital_los_day	age	gender_num	weight_first	bmi	sapsi_first	sofa_first	service_unit	service_num	day_icu_intime	day_icu_intim
0	1	2.400648	13.000000	72.36841	1	75.000000	29.912791	15.0	9.0	SICU	1	Friday	
1	1	0.580000	3.000000	44.49191	0	80.924576	27.834917	21.0	7.0	SICU	1	Saturday	
2	1	1.750000	5.000000	23.74217	1	95.200000	28.464563	18.0	7.0	SICU	1	Saturday	
3	0	1.380000	9.000000	36.54657	1	72.000000	23.982402	14.0	5.0	SICU	1	Sunday	
4	1	2.038264	6.008403	24.64717	1	90.000000	25.474850	15.0	6.0	SICU	1	Saturday	

```
[974]: df[columns_to_fill].isna().sum()

[974]: sapsi_first        0
       icu_los_day        0
       hospital_los_day   0
       bmi                0
       weight_first       0
       abg_count          0
       po2_first          0
       pco2_first         0
       iv_day_1           0
       spo2_1st           0
       sofa_first         0
       map_1st            0
       hr_1st             0
       temp_1st           0
       wbc_first          0
       hgb_first          0
       platelet_first     0
       sodium_first       0
       tco2_first         0
       chloride_first     0
       bun_first          0
       creatinine_first   0
       potassium_first    0
       dtype: int64
```

Figure 16. Filling the missing values with the means of the age range.

Step 5. Conduct Data Analysis

Comprehensive analysis revealed a close relationship between median and mean, indicating symmetric data distribution. Outliers, notably in 'iv_day_1' and 'po2_first,' emphasized substantial variability, reflecting ICU patient diversity. While high standard deviation provides insights, it requires careful consideration in predictive model development for accurate and meaningful predictions.

5) Conduct data analysis

```
[1025]: num_vars_ints = num_vars[['sapsi_first', 'icu_los_day', 'hospital_los_day','bmi','weight_first', 'abg_count',
                  'po2_first', 'pco2_first', 'iv_day_1', 'spo2_1st', 'sofa_first', 'map_1st', 'hr_1st',
                  'temp_1st', 'wbc_first', 'hgb_first', 'platelet_first', 'sodium_first', 'tco2_first',
                  'chloride_first', 'bun_first', 'creatinine_first', 'potassium_first']]
        num_vars_ints.agg(['median','mean','std'])
```

[1025]:

	sapsi_first	icu_los_day	hospital_los_day	bmi	weight_first	abg_count	po2_first	pco2_first	iv_day_1	spo2_1st	sofa_first	map_1st	hr_1st	te
median	14.000000	2.295000	7.000000	22.898715	76.900000	4.000000	198.000000	39.000000	1134.629944	100.000000	6.000000	88.00000	85.000000	98
mean	13.628310	3.534155	9.007128	18.620344	75.667546	7.133401	221.572301	36.917515	1624.942100	98.947047	5.827902	88.79294	85.759674	97
std	4.957858	3.669347	8.837931	13.557910	26.059355	10.039551	153.155829	14.891737	1654.865304	4.653021	2.182760	17.18805	18.256784	5

Figure 17. Median, Mean and Standard Deviation of Variables of Interests

Step 6. Create New Box Plots and Analyze Variables After Data Imputation

Post-imputation, gender-stratified box plots showed aligned medians and reduced outliers, preparing the data for predictive models. However, identifying and addressing irrelevant variables (overfitting, redundancy, multicollinearity) precedes model development. Detailed insights on multicollinearity will be provided shortly.

```
[1026]: fig, axes = plt.subplots(nrows=6, ncols=4, figsize=(20, 15))

        count = 0

        for variable in num_vars_ints.columns:
            count += 1
            ax = axes.flatten()[count - 1]  # Get the current subplot

            # Use seaborn's boxplot for better categorical plotting
            sns.boxplot(x='gender_num', y= variable, data=df, ax=ax, palette={0: 'blue', 1: 'purple'})

            ax.set_title(variable)
            ax.set_xlabel('Gender (0: Male, 1: Female)')
            ax.set_ylabel(variable)


        plt.tight_layout()
        plt.show()


        plt.tight_layout()
        plt.show()
```

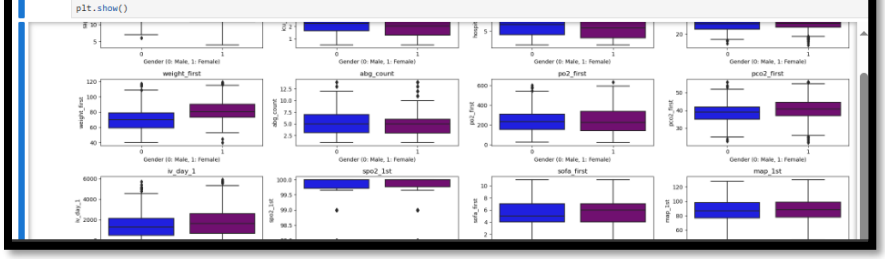


Figure 18. Box Plots After Data Imputation

6.1. Dropping Columns

I excluded columns based on their characteristics for being unnecessary:

- Service_unit: Irrelevant, as all units are associated with 'SICU.'
- Service_num: Numerical representation of 'service_unit,' also deemed irrelevant.
- day_icu_intime: Redundant, as a numerical column provides the same information.
- bmi: With 31% missing values and weight used to calculate BMI, its inclusion introduces redundancy.
- 'day_28_flg,' 'mort_day_censored,' 'censor_flg,' and 'aline_flg': Removed to prevent multicollinearity and redundancy.
- 'aline_flg': Redundant with ICU length of stay, thus removed from the dataset. By removing these columns, I aim to streamline the dataset and enhance the effectiveness of the analysis.

```
[26]: # Columns to drop
columns_to_drop = ['service_num', 'service_unit','day_icu_intime','bmi','day_28_flg',
                   'mort_day_censored','censor_flg','aline_flg']

df = df.drop(columns=columns_to_drop)


df
```

[26]:

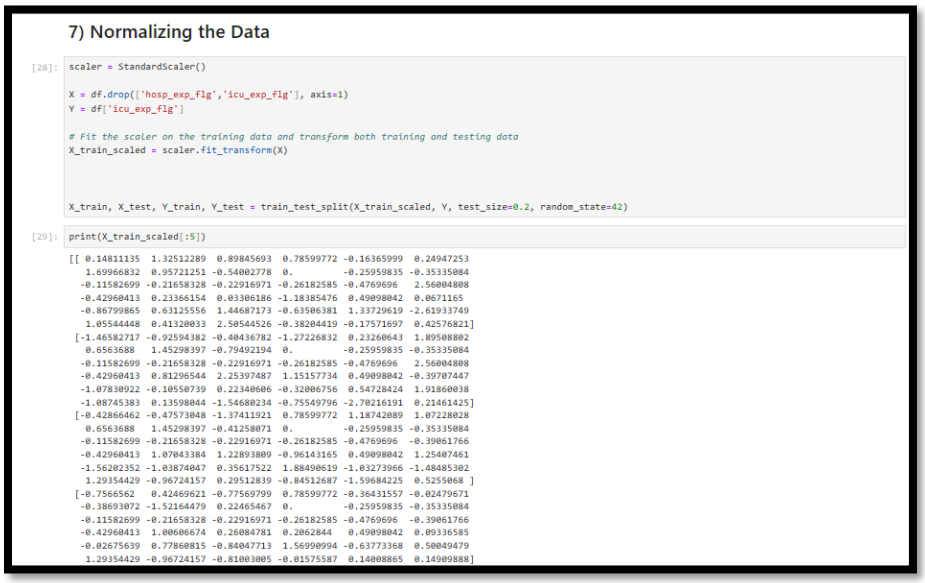
	icu_los_day	hospital_los_day	age	gender_num	weight_first	sapsi_first	sofa_first	day_icu_intime_num	hour_icu_intime	hosp_exp_flg	icu_exp_flg	sepsis_flg
0	2.400648	13.000000	72.36841	1	75.000000	15.0	9.0	6	6	1	0	0
1	0.580000	3.000000	44.49191	0	80.924576	21.0	7.0	7	4	1	1	0
2	1.750000	5.000000	23.74217	1	95.200000	18.0	7.0	7	7	0	0	0
3	1.380000	9.000000	36.54657	1	72.000000	14.0	5.0	1	12	0	0	0
4	2.038264	6.008403	24.64717	1	90.000000	15.0	6.0	7	22	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...
977	2.270000	2.000000	50.15744	0	79.826875	14.0	7.0	7	18	1	1	0
978	1.080000	3.000000	77.96057	0	77.417647	16.0	5.0	6	12	1	0	0
979	2.450000	3.000000	33.04643	1	84.000000	16.0	6.0	7	11	0	0	0
980	3.700000	4.000000	72.44020	0	60.000000	14.0	5.0	7	23	1	1	0
981	2.322571	8.112583	53.74423	1	79.826875	18.0	9.0	4	1	0	0	0

982 rows × 38 columns

Figure 19. Excluding Columns with Potential Impact on Analysis.

Step 7. Normalizing the Data

I divided my data into variables X (features) and Y (target). Excluding 'icu_exp_flg' and 'hosp_exp_flg' from X, I used Standard Scaler to standardize features, ensuring a mean of 0 and a standard deviation of 1. Employing 'train_test_split,' I created datasets for independent variables (X) and the target variable (y), split for model evaluation.



7) Normalizing the Data

```
[28]: scaler = StandardScaler()

      X = df.drop(['hosp_exp_flg','icu_exp_flg'], axis=1)
      Y = df['icu_exp_flg']

      # Fit the scaler on the training data and transform both training and testing data
      X_train_scaled = scaler.fit_transform(X)



      X_train, X_test, Y_train, Y_test = train_test_split(X_train_scaled, Y, test_size=0.2, random_state=42)

[29]: print(X_train_scaled[:5])

[[ 0.14811135  1.32512289  0.89845693  0.78599772 -0.16365999  0.24947253
   1.69966832  0.95721251 -0.54002778  0.         -0.25959835 -0.35335084
  -0.11582699 -0.21658328 -0.22916971 -0.26182585 -0.4769696   2.56004808
  -0.42960413  0.23366154  0.83306186 -1.18385476  0.49098042  0.0671165
  -0.86799865  0.63125556  1.44687173 -0.63506381  1.33729619 -2.61933749
   1.05544448  0.41320033  2.50544526 -0.38204419 -0.17571697  0.42576821]
 [-1.46582717 -0.92594382 -0.40436782 -1.27226832  0.23260643  1.89508802
   0.6563688   1.45298397 -0.79492194  0.         -0.25959835 -0.35335084
  -0.11582699 -0.21658328 -0.22916971 -0.26182585 -0.4769696   2.56004808
  -0.42960413  0.81296544  2.25397487  1.15157734  0.49098042 -0.39707447
  -1.07830922 -0.10550739  0.22340606 -0.32006756  0.54728424  1.91860038
  -1.08745383  0.13598044 -1.54680234 -0.75549796 -2.70216191  0.21461425]
 [-0.42866462 -0.47573048 -1.37411921  0.78599772  1.18742089  1.07228028
   0.6563688   1.45298397 -0.41258071  0.         -0.25959835 -0.35335084
  -0.11582699 -0.21658328 -0.22916971 -0.26182585 -0.4769696  -0.39061766
  -0.42960413  1.07043384  1.22893809 -0.96143165  0.49098042  1.25407461
  -1.56202352 -1.03874047  0.35617522  1.88490619 -1.03273966 -1.48485302
   1.29354429 -0.96724157  0.29512839 -0.84512687 -1.59684225  0.5255068 ]
 [-0.7566562   0.42469621 -0.77569799  0.78599772 -0.36431557 -0.02479671
  -0.38693072 -1.52164479  0.22465467  0.         -0.25959835 -0.35335084
  -0.11582699 -0.21658328 -0.22916971 -0.26182585 -0.4769696  -0.39061766
  -0.42960413  1.00606674  0.26084781  0.2062844   0.49098042  0.09336585
  -0.02675639  0.77860815 -0.84047713  1.56990994 -0.63773368  0.50049479
   1.29354429 -0.96724157 -0.81003005 -0.01575587  0.14008865  0.14909888]
```

Figure 20. Normalizing data and defining training and testing data.

Step 8. Multicollinearity Concept and PCA

Multicollinearity occurs when regression model variables are highly correlated, complicating interpretation and coefficient estimation. Principal Component Analysis (PCA) tackles multicollinearity by transforming correlated variables into uncorrelated components, using sklearn PCA on scaled train data to create two components for enhanced understanding and visualization.

```
from sklearn.decomposition import PCA

class_labels = ['No Death', 'Death in ICU']
colors = ['blue', 'red']

# Define X_r as the scaled data
X_r = X_train_scaled

pca = PCA(n_components=2)
pca_result = pca.fit_transform(X_r)

# Plot each class separately with different colors
for label_value, color in zip(np.unique(Y), colors):
    indices = np.where(Y == label_value)
    plt.scatter(pca_result[indices, 0], pca_result[indices, 1], label=class_labels[label_value], color=color)

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA Principal Component Analysis')
plt.legend()
plt.show()
```

Figure 21. Defining the PCA

8.1. PCA Graphic

Principal components, representing linear combinations of the dataset, reveal a noticeable separation in the PCA plot based on ICU outcome (deaths or no deaths). Blue points lean positive, while red points tend negative, suggesting systematic differences in contributing variables. Variables with positive loadings for no deaths and negative loadings for deaths play a crucial role, hinting at structural differences providing potential insights into ICU mortality factors.

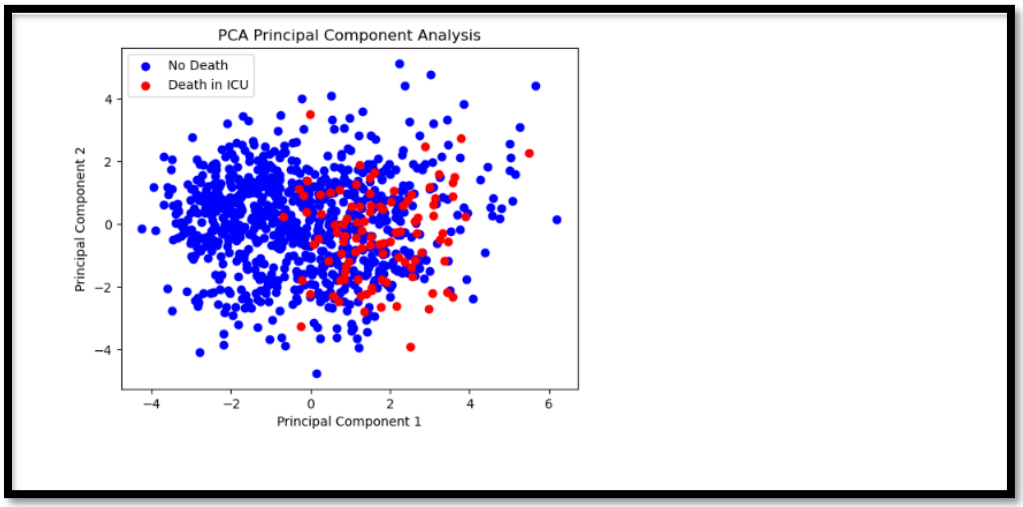


Figure 22. Scatter Plots of PCA

8.2. Data Frame of PCA Component

The PCA Data Frame below displays components alongside the 'icu_exp_flg' target variable, predicting ICU deaths. Binary values (1 and 0) conventionally represent the positive class ('deaths in the ICU') and negative class ('no deaths in the ICU'). These values offer clear indicators of predicted outcomes in binary classification scenarios, distinguishing between cases with and without ICU deaths.

```
[31]: # Create a DataFrame with principal components and target variable
pc_df = pd.DataFrame(data=pca_result, columns=['PC1', 'PC2'])
pc_df['icu_exp_flg'] = Y

pc_df
```

[31]:

	PC1	PC2	icu_exp_flg
0	0.614088	1.355427	0
1	0.760816	-0.948288	1
2	-1.533892	-0.316138	0
3	-2.074523	-0.441329	0
4	-0.636772	2.292636	0
...	...	...	...
977	1.356675	0.092266	1
978	1.427444	-1.581672	0
979	-1.411090	0.020767	0

Figure 23. Data frame of PCA Components

Step 9. Using Logistic Regression

Data is prepared for Logistic Regression modeling using Scikit-learn, configuring 'log_reg_model' with 1000 iterations and 'liblinear' solver. After training on features (X_train) and target (Y_train), predictions are made on the test set (X_test) and stored in 'y_pred.' Accuracy is assessed using 'accuracy_score' and a detailed 'classification_report,' and K-Fold Cross-Validation with 5 splits offers robust performance evaluation across diverse training data subsets.

```
[33]:

# Initialize the Logistic Regression model with an increased max_iter
log_reg_model = LogisticRegression(max_iter=1000, solver='liblinear')

# Train the model on the training set
log_reg_model.fit(X_train, Y_train)

# Make predictions on the testing set
y_pred = log_reg_model.predict(X_test)


# Evaluate the model
accuracy = accuracy_score(Y_test, y_pred)
classification_rep = classification_report(Y_test, y_pred)
acc_log_reg_model = round(log_reg_model.score(X_train, Y_train) * 100, 2)

# Results
print("Accuracy:", accuracy)
print("Classification Report:\n", classification_rep)

print(" ")
# Set up K-Fold Cross-Validation with 5 splits
k_folds = KFold(n_splits=5)

# Perform cross-validation
scores = cross_val_score(log_reg_model, X_train, Y_train, cv=k_folds, scoring='accuracy')

# Print the Cross-Validation scores
print("Cross Validation Scores: ", scores)
print("Average CV Score: ", scores.mean())
print("Number of CV Scores used in Average: ", len(scores))
```

Figure 24. Performing a Logistic Regression Model

9.1. Results

The Logistic Regression Model exhibits a high accuracy of 93%, supported by a detailed classification report showcasing high precision, recall, F1-score, and support. Cross-validation results reveal consistent accuracy per split (87%-94%), averaging 91%, confirming the model's robustness and strong predictive capabilities.

```
Accuracy: 0.9390862944162437
Classification Report:
               precision    recall  f1-score   support

           0       0.95      0.99      0.97       178
           1       0.82      0.47      0.60        19

    accuracy                           0.94       197
   macro avg       0.88      0.73      0.78       197
weighted avg       0.93      0.94      0.93       197


Cross Validation Scores:  [0.9044586  0.89171975 0.94267516 0.9044586  0.92356688]
Average CV Score:  0.913375796178344
Number of CV Scores used in Average:  5
```

Figure 25. Results of the Logistic Regression Model

9.2. Confusion Matrix

We can obtain specific insights that describe the model's robustness through the confusion matrix, which numerically presents the accuracy breakdown. The results are as follows:

- 175 patients were correctly predicted as 'No Death in the ICU,' matching the labels in the “Y_test” file derived from the cleaned data.
- 3 patients were incorrectly predicted as 'No Death in the ICU' when, in the dataset, they are defined as 'Death.'
- 10 patients were incorrectly predicted as 'Death in the ICU' when, in the dataset, they are defined as 'No Death.'
- 9 patients were correctly predicted as 'Death in the ICU,' aligning with the labels in the dataset.
-

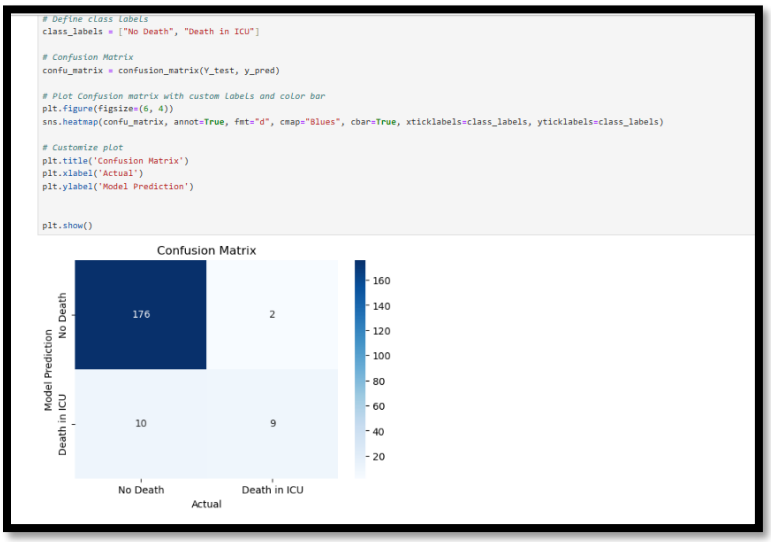


Figure 26. Confusion Matrix Result

Conclusion

In conclusion, this project transforms raw data into valuable insights, enabling ICU mortality prediction with essential tools like pandas, matplotlib, seaborn, numpy, and sklearn. These tools identify and handle redundant or anomalous data, crucial for predicting outcomes like mortality in the ICU. Their strategic application streamlines analysis, enhancing precision and reliability in predictive models.