RMIT UNIVERSITY

ASSIGNMENT 1

Machine Learning Model For Sepsis Prediction

Nguyen Bao Tri - s3749560

I. Introduction

Sepsis is a life-threatening infecton that develops from staying in the Intensive Care Units (ICU), which happens when the infection-fighting processes damage its own organs. During this COVID-19 pandemic, the ICUs needs to focuses on this problems more than ever, therefore, the need to monitor patients status regarding their infection possibility considerably increases. With that in mind, this notebook goal is to build a machine learning model that can predict the infection propability of a patient based on a provided dataset.

Many libraries are used in this notebook including analytical and visualizing libraries such as *numpy, pandas, matplotlib and seaborn* and scikit-learn machine learning models. Since our object is predicting the Sepsis result, the models I use are **binary classifiers**, certainly, *LogisticRegression, RandomForest*.

II. Exploratory Data Analysis (EDA)

1. Import Necessary Libraries

- In this notebook, multiple libraries are used to do exploratory data analysis and visualize data.

In [1]:
```
#import libraries
#ignore warning
import warnings
warnings.filterwarnings("ignore")

#measure running time
from time import time

#for analytics and visualization
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

#data processing and machine learning algorithms
from sklearn.preprocessing import MinMaxScaler, StandardScaler, PolynomialFeatures
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split, KFold, cross_val_score, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score, classification_report, plot_confusion_matrix
```

2. Load dataset

Load the csv file to do some analysis prior to training the model.

In [2]:
```
#train set
patientsFrame_Train = pd.read_csv('Paitients_Files_Train.csv')
#test set
patientsFrame_Test = pd.read_csv('Paitients_Files_Test.csv')

s = patientsFrame_Train.shape
print('The data has %d rows and %d columns' %(s[0], s[1]))

patientsFrame_Train.head(10)
```

```
The data has 599 rows and 11 columns
```

Out[2]:

	ID	PRG	PL	PR	SK	TS	M11	BD2	Age	Insurance	Sepssis
0	ICU200010	6	148	72	35	0	33.6	0.627	50	0	Positive
1	ICU200011	1	85	66	29	0	26.6	0.351	31	0	Negative
2	ICU200012	8	183	64	0	0	23.3	0.672	32	1	Positive
3	ICU200013	1	89	66	23	94	28.1	0.167	21	1	Negative
4	ICU200014	0	137	40	35	168	43.1	2.288	33	1	Positive
5	ICU200015	5	116	74	0	0	25.6	0.201	30	1	Negative
6	ICU200016	3	78	50	32	88	31.0	0.248	26	0	Positive
7	ICU200017	10	115	0	0	0	35.3	0.134	29	1	Negative
8	ICU200018	2	197	70	45	543	30.5	0.158	53	1	Positive
9	ICU200019	8	125	96	0	0	0.0	0.232	54	1	Positive

- The data has *599 entries* and 11 columns for 11 different features with the last one is the result of sepssis.

Check for null/ missing value in each columns using the **isnull()**.

In [3]:
```
patientsFrame_Train.isnull().sum()
```

Out[3]:
```
ID           0
PRG          0
PL           0
PR           0
SK           0
TS           0
M11          0
BD2          0
Age          0
Insurance    0
Sepssis      0
dtype: int64
```

- As we can see, there are *no* null value found in any columns. Next, we check through the **info()** the data types of each feature to see whether the missing value is recorded as another format.

In [4]:
```
patientsFrame_Train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 599 entries, 0 to 598
Data columns (total 11 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   ID         599 non-null    object 
 1   PRG        599 non-null    int64  
 2   PL         599 non-null    int64  
 3   PR         599 non-null    int64  
 4   SK         599 non-null    int64  
 5   TS         599 non-null    int64  
 6   M11        599 non-null    float64
 7   BD2        599 non-null    float64
 8   Age        599 non-null    int64  
 9   Insurance  599 non-null    int64  
 10  Sepssis    599 non-null    object 
dtypes: float64(2), int64(7), object(2)
memory usage: 51.6+ KB
```

- With these simple examination through the dataset, there are no missing values and we can assume that the data set is cleaned and we can proceed to do the analytics.

Drop column ID and Insurance as they are unnecessary for the analysis.

In [5]:
```
patientsFrame_Train = patientsFrame_Train.drop(columns=['ID', 'Insurance'])
patientsFrame_Test =  patientsFrame_Test.drop(columns=['ID', 'Insurance'])
```

We can see that Sepssis value is binary (positive and negative), hence, for convenience upon further analysis, we convert Sepssis value to 0 for 'negative' and 1 for 'positive'

In [6]:
```
seps = {'Negative': 0, 'Positive': 1}
patientsFrame_Train['Sepssis'] = patientsFrame_Train['Sepssis'].map(seps)
#check the dataframe again
patientsFrame_Train.head()
```

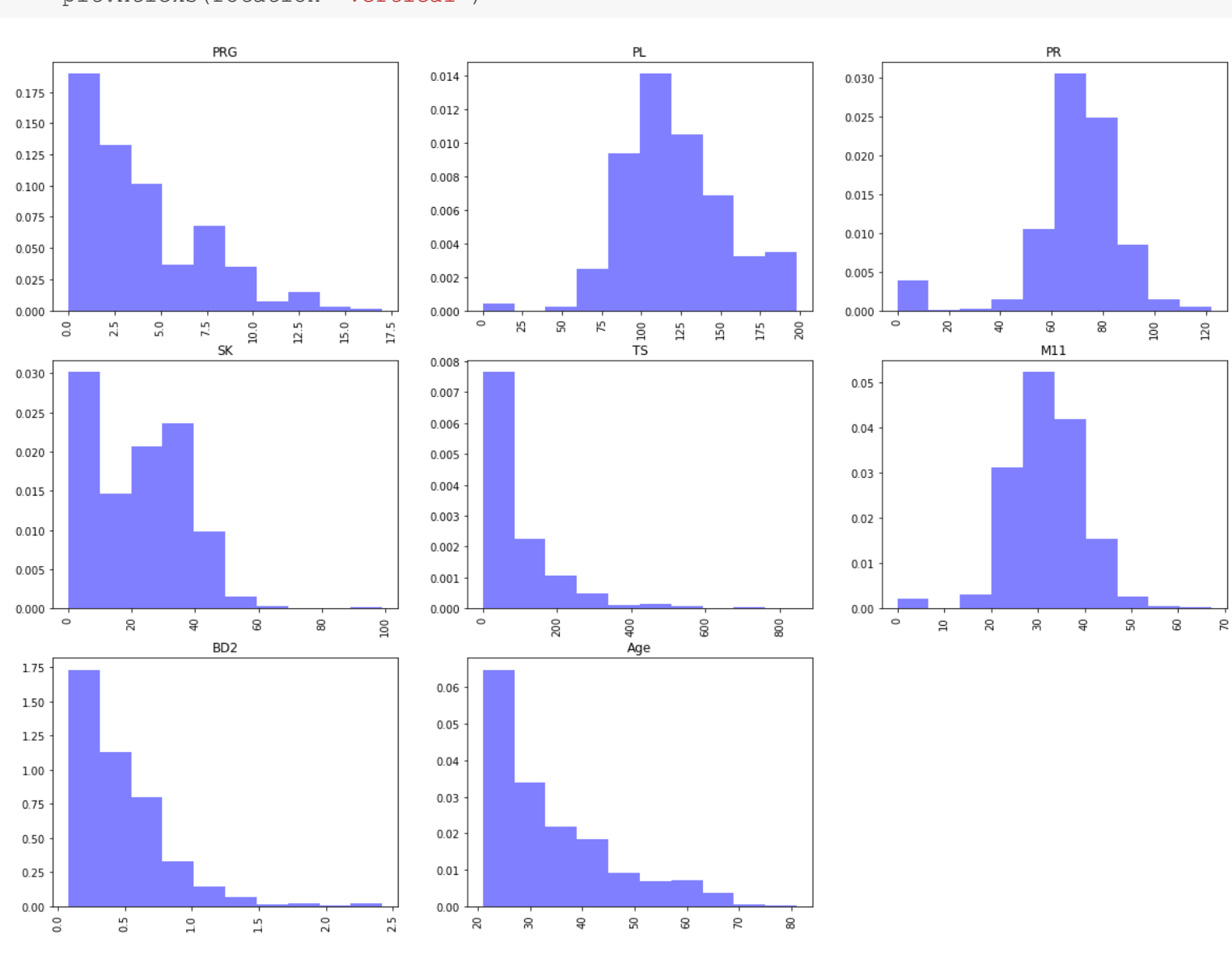
Out[6]:

	PRG	PL	PR	SK	TS	M11	BD2	Age	Sepssis
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

3. Plotting and Analysing

Start with plotting the histogram of every features to examine their distribution.

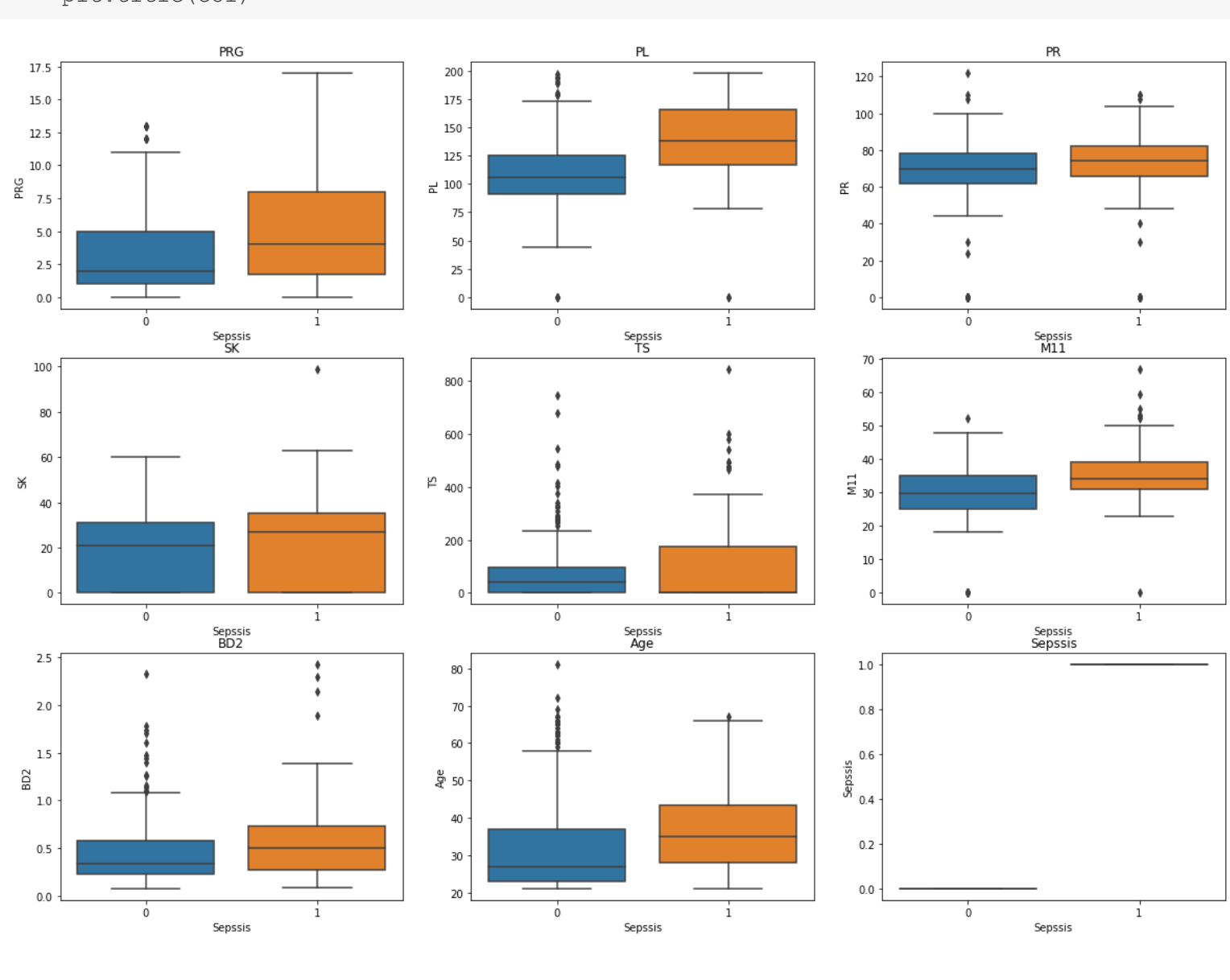
In [7]:
```
#create a dataframe that dropped Sepssis
patientsFrame_dropSep = patientsFrame_Train.drop(columns=['Sepssis'])
plt.figure(figsize = (20,20))
for i, col in enumerate(patientsFrame_dropSep.columns):
    plt.subplot(4,3,i+1)
    plt.hist(patientsFrame_dropSep[col], alpha =0.5, color = 'b', density =True)
    plt.title(col)
    plt.xticks(rotation='vertical')
```



- **Gaussian Distribution**: PL, PR, M11
- From observation, there seems to be outliers in PL, PR, and M11. These will be put into consideration.

Now, I use boxplots on the features with the division of Sepssis results to see the correlation between the caught rate and other attributes.

In [8]:
```
plt.figure(figsize = (20,20))
for i, col in enumerate(patientsFrame_Train.columns):
    plt.subplot(4,3,i+1)
    sns.boxplot(data=patientsFrame_Train, x='Sepssis', y=patientsFrame_Train[col])
    plt.title(col)
```



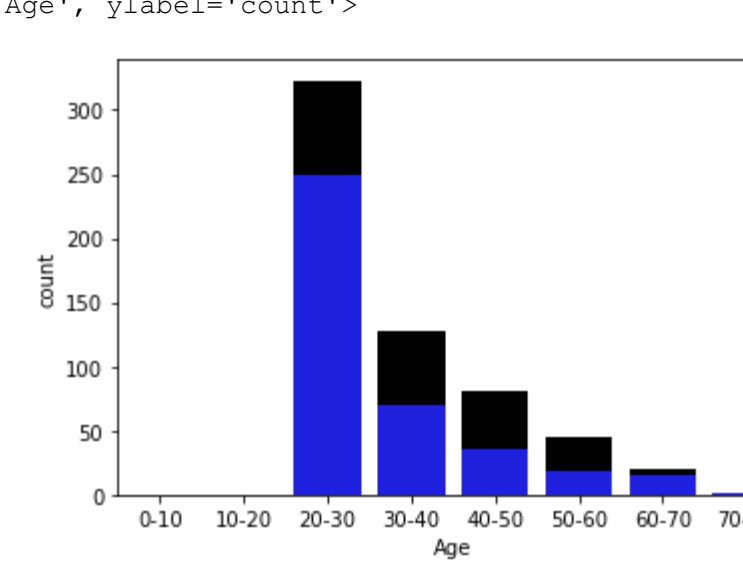
As shown in the graph, firstly, there are many outliers upon every divided features. These outliers are just outliers of the data as plotting out with two groups. However, there are some points that are considerable:

- For *SK*, the outliers are minor, which do not significantly affect our prediction later on. Moreover, the distribution of negative and positive results are equivalent, we can assume that **SK feature is not so important**.
- In *PL, PR and M11*, by comparing with the previous histogram, the under outliers may not have subtantial impact on the prediction results as they are evenly scattered upon both groups.
- On the other hand, also from observation, PL and PRG are the most correlated features.

More visualization to examine more insight.

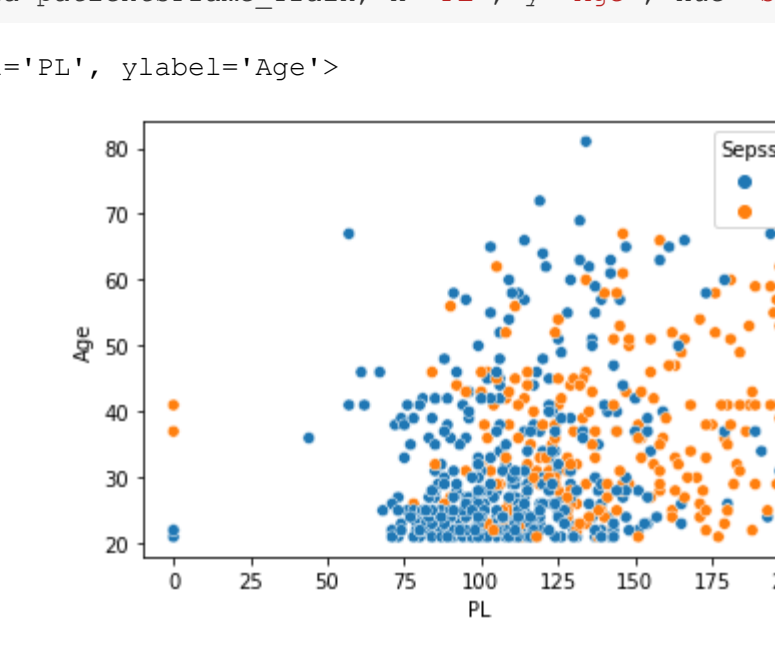
In [9]:
```
age_group = pd.cut(patientsFrame_Train.Age, bins=[0,10,20,30,40,50,60,70,80],
                   labels = ['0-10','10-20','20-30','30-40','40-50','50-60','60-70','70-80'])
sns.countplot(age_group, color = 'black')
sns.countplot(age_group[(patientsFrame_Train.Sepssis == 0)], color ='b')
```

Out[9]: <AxesSubplot:xlabel='Age', ylabel='count'>



In [10]:
```
sns.scatterplot(data=patientsFrame_Train, x='PL', y="Age", hue="Sepssis")
```

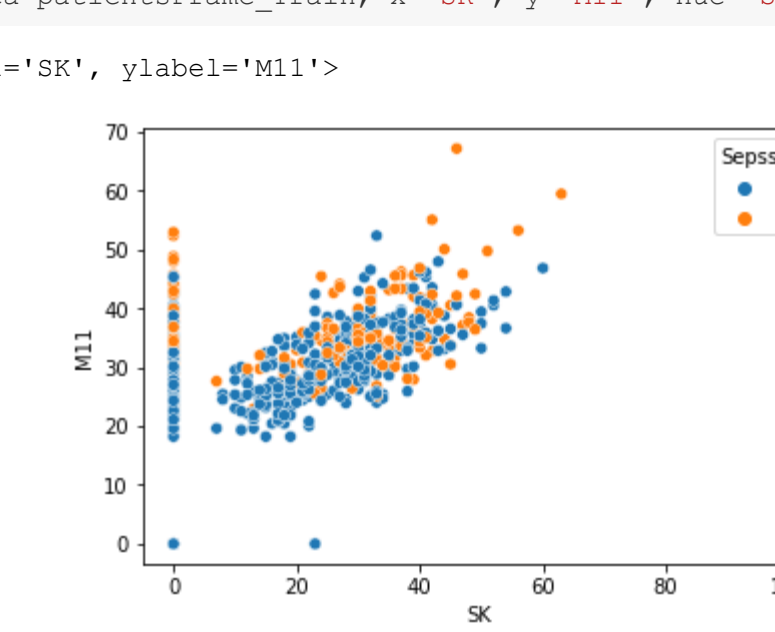
Out[10]: <AxesSubplot:xlabel='PL', ylabel='Age'>



- From the two above graphs, older patients have higher rate of getting the disease but this only happens in groups, that means in a group of older people, the positive percentage is higher compared to younger groups. Additionally, the higher the PL value is the more chance that patient is positive.

In [11]:
```
sns.scatterplot(data=patientsFrame_Train, x='SK', y="M11", hue="Sepssis")
```

Out[11]: <AxesSubplot:xlabel='SK', ylabel='M11'>

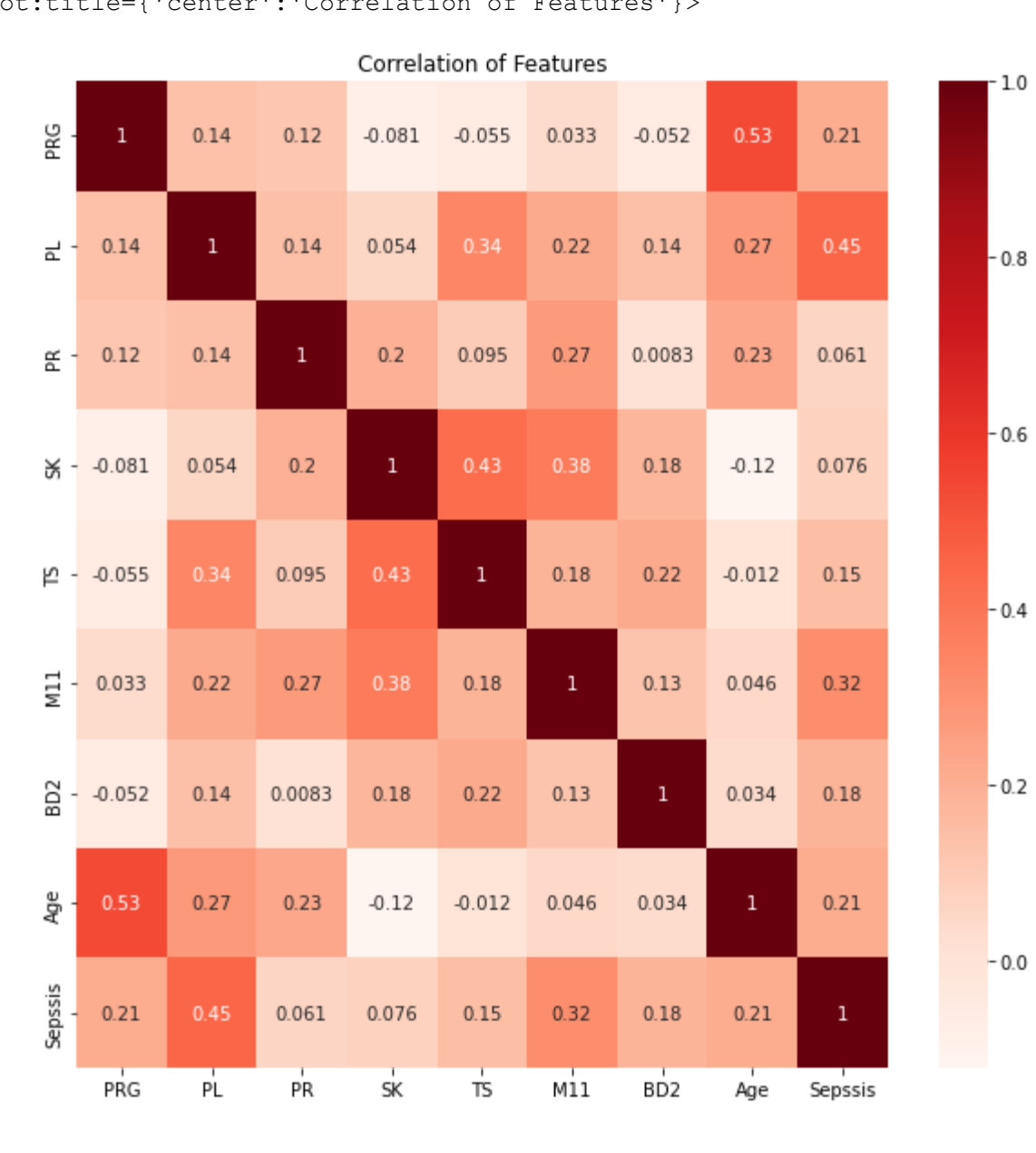


- Once again, SK does not show any huge impacts with its even allocation.
- Another information that can be seen from this is those with higher M11 also have higher chance of being positive.

Use heatmap correlation to get a general view

In [12]:
```
plt.figure(figsize=(10,10))
plt.title('Correlation of Features')
sns.heatmap(patientsFrame_Train.corr(), cbar=True, annot=True, cmap='Reds')
```

Out[12]: <AxesSubplot:title={'center':'Correlation of Features'}>



- Through this overall heatmap, the above assumptions are strengthen as PL has the highest correlation, M11 in the second place, Age and PRG comes next. These features will be meaningful for our model.
- In addition, PRG and Age seem to have high correlation.

III. Training models

1. Data Preprocessing

From the EDA step, we can see that our data has myriad of outliers upon the features. Some are insignificant, but some are crucial to our model based on the analysis, certainly, M11. M11 is body mass index, therefore, it can not be 0 (as below 18 is already considered malnutritous) so I did some conditional imputation.

M11

In [13]:
```
patientsFrame_Train['M11'].describe()
```

Out[13]:
```
count    599.000000
mean      31.920033
std        8.008227
min        0.000000
25%       27.100000
50%       32.000000
75%       36.550000
max       67.100000
Name: M11, dtype: float64
```

From the heatmap from EDA process, we can see that *M11* is highly correlate to *SK*, so I will examine the SK value of those outliers.

In [14]:
```
patientsFrame_Train[['M11','SK']].loc[patientsFrame_Train['M11'] == 0]
```

Out[14]:

	M11	SK
9	0.0	0
49	0.0	0
60	0.0	0
81	0.0	0
145	0.0	23
371	0.0	23
426	0.0	0
494	0.0	0
522	0.0	0

Most of the outliers has the SK values of 0 and some is 23. Therefore, I will impute those with the average M11 that has SK equal to 0, the same for 23. The technique that is used is initially replace the outliers with *nan* and use *fillna* to fill those spots with the calculated value.

In [15]:
```
patientsFrame_Train['M11'].loc[(patientsFrame_Train['SK'] == 0) &  (patientsFrame_Train['M1
1'] != 0)].mean()
patientsFrame_Train['M11'].loc[(patientsFrame_Train['SK'] == 23) & (patientsFrame_Train['M1
1'] != 0)].mean()
```

Out[15]:
```
31.686666666666664
```

In [16]:
```
#calculate the mean of M11 with SK =0 and SK =23, excluding those M11 = 0
m11_mean_sk23 = patientsFrame_Train['M11'].loc[(patientsFrame_Train['SK'] == 23) & (patients
Frame_Train['M11'] != 0)].mean()
m11_mean_sk0 = patientsFrame_Train['M11'].loc[(patientsFrame_Train['SK'] == 0) &  (patientsF
rame_Train['M11'] != 0)].mean()

#get the indices of those with M11 = 0
outliers = patientsFrame_Train['M11'].loc[(patientsFrame_Train['SK'] == 0) & (patientsFrame_
Train['M11'] == 0)].index
outliers1 = patientsFrame_Train['M11'].loc[(patientsFrame_Train['SK'] == 23) & (patientsFram
e_Train['M11'] == 0)].index

#impute
print(patientsFrame_Train['M11'][outliers])
print(patientsFrame_Train['M11'][outliers1])
patientsFrame_Train['M11'][outliers] = np.nan
patientsFrame_Train['M11'].fillna(round(m11_mean_sk0, 2), inplace = True)

patientsFrame_Train['M11'][outliers1] = np.nan
patientsFrame_Train['M11'].fillna(round(m11_mean_sk23, 2), inplace = True)

print(patientsFrame_Train['M11'][outliers])
print(patientsFrame_Train['M11'][outliers1])
```

```
9      0.0
49     0.0
60     0.0
81     0.0
426    0.0
494    0.0
522    0.0
Name: M11, dtype: float64
145    0.0
371    0.0
Name: M11, dtype: float64
9      31.59
49     31.59
60     31.59
81     31.59
426    31.59
494    31.59
522    31.59
Name: M11, dtype: float64
145    31.69
371    31.69
Name: M11, dtype: float64
```

PL

Although, as analysed above, the Sepssis result distribution in PL outliers are even, but they are still outliers and PL is a significant figure, it is a good idea to handle the outliers carefully so that the values are sufficiently standardized to train.

Imply the same technique to PL, the general mean is decent enough for this case as the Sepssis status of these outliers are balanced.

The value that is imputed to the outliers must be integer following the initial format of the dataset.

In [17]:
```
patientsFrame_Train['PL'].describe()
```

Out[17]:
```
count    599.000000
mean     120.153589
std       32.682364
min        0.000000
25%       99.000000
50%      116.000000
75%      140.000000
max      198.000000
Name: PL, dtype: float64
```

In [18]:
```
patientsFrame_Train['PL'].loc[patientsFrame_Train['PL'] == 0]
```

Out[18]:
```
75     0
182    0
342    0
349    0
502    0
Name: PL, dtype: int64
```

In [19]:
```
outliers2 = patientsFrame_Train['PL'].loc[patientsFrame_Train['PL'] == 0].index
pl_mean = patientsFrame_Train['PL'].mean()
patientsFrame_Train['PL'][outliers2] = np.nan
patientsFrame_Train['PL'].fillna(int(pl_mean), inplace = True)
print(patientsFrame_Train['PL'][outliers2])
```

```
75     120.0
182    120.0
342    120.0
349    120.0
502    120.0
Name: PL, dtype: float64
```

Set Up Training And Testing Data

In this notebook, I will use both univariate model and multivariate model so I will set up the data frame for both models. The picked split ratio is 7:3, this feeds more data to train the model which results in better accuracy.

Dependent Variable

In [20]:
```
patientsFrame_Train_Y = patientsFrame_Train['Sepssis']
```

Univariate Model

Undoubtedly, the independent variable is *PL* with the highest correlation with our prediction target.

In [21]:
```
patientsFrame_Test_X_U = patientsFrame_Test[['PL']]
patientsFrame_Train_X_U = patientsFrame_Train[['PL']]

trainX_U, valX_U, trainY_U, valY_U = train_test_split(patientsFrame_Train_X_U, patientsFrame
_Train_Y, test_size=0.2,shuffle=True)
#examine the split
print(trainX_U.shape)
print(valX_U.shape)
print(trainY_U.shape)
print(valY_U.shape)
```

```
(479, 1)
(120, 1)
(479,)
(120,)
```

Feature Selection For Multivariate Model

With the EDA process output, the dataframe that is put into model will have selective features, so that the model will be less resources-consuming, more efficient regarding accuracy and more comprehendable. In this case, I only keep *PL, M11, Age and PRG*.

In [22]:
```
patientsFrame_Train_X_M = patientsFrame_Train[['PL', 'M11', 'Age', 'PRG']]
patientsFrame_Test_X_M = patientsFrame_Test[['PL', 'M11', 'Age', 'PRG']]
trainX_M, valX_M, trainY_M, valY_M = train_test_split(patientsFrame_Train_X_M, patientsFrame
_Train_Y, test_size=0.3,shuffle=True)
#examine the split
print(trainX_M.shape)
print(valX_M.shape)
print(trainY_M.shape)
print(valY_M.shape)
```

```
(419, 4)
(180, 4)
(419,)
(180,)
```

Regarding the value distribution and range of the selected features of the processing data, the difference between features's range is significant and with the inclusion of later-used technique (Polynomial Features), it is necessary to perform data scaling. Moreover, this action can save computational resources and enhance accuracy due to the facts that LogisticRegression is sensitive to the range of data points.

In this notebook, both Logistic Regression and tree-based algorithms are used. The scaling step is helpful for the performance of LogisticRegression but is not effective for the tree-based. Furthermore, for classification tasks, the results of tree-based algorithms is class selected, no calculation involved, consequently, data scaling does not affect the output of tree-based classifier.

Throughout the 4 features we selected, there are two that have Gaussian Distribution and the other two are skewed. Thus, I will mix the scaling techniques to suit the distribution of each. **Normalization** to the skewed features and **Standardization** to the normal distribution.

In [23]:
```
#define the scaler using ColumnTransformer to conditionally transform the data
scaler = ColumnTransformer([("standard", StandardScaler(), ['PL', 'M11']), ('norm', MinMaxSc
aler(), ['Age', 'PRG'])])

#fit the data with the scaler
print(trainX_M)
print(valX_M)

scaler.fit(trainX_M)
trainX_M = scaler.transform(trainX_M)
valX_M = scaler.transform(valX_M)

print(trainX_M)
print(valX_M)
```

```
        PL   M11  Age  PRG
156   99.0  24.6   21    2
179  130.0  39.1   37    5
380  107.0  30.8   24    1
16   118.0  45.8   31    0
304  150.0  21.0   37    3
..     ...   ...  ...  ...
161  102.0  37.2   45    7
7    115.0  35.3   29   10
506  180.0  36.5   35    0
108   83.0  34.3   25    3
389  100.0  31.6   28    3

[419 rows x 4 columns]
        PL   M11  Age  PRG
582  121.0  26.5   62   12
99   122.0  49.7   31    1
65    99.0  29.0   32    5
381  105.0  20.0   22    0
43   171.0  45.4   54    9
..     ...   ...  ...  ...
476  105.0  33.7   29    2
583  100.0  38.7   42    8
328  102.0  45.5   23    2
253   86.0  35.8   25    0
28   145.0  22.2   57   13

[180 rows x 4 columns]
[[-0.71722982 -1.10374295  0.          0.11764706]
 [ 0.30685484  0.96860621  0.26666667  0.29411765]
 [-0.45294991 -0.21763504  0.05        0.05882353]
 ...
 [ 1.95860429  0.59701256  0.23333333  0.        ]
 [-1.24578965  0.28258717  0.06666667  0.17647059]
 [-0.68419483 -1.10374295  0.11666667  0.17647059]
 [ 0.00953994 -0.83219375  0.68333333  0.70588235]
 [-0.71722982 -0.47489217  0.18333333  0.05882353]
 [-0.28777496 -0.10329853  0.06666667  0.29411765]
 [-0.55205488 -1.98985086  0.1         0.        ]
 [ 1.66128939  1.86900618  0.55        0.52941176]
 [-1.21275466  1.02577446  0.06666667  0.23529412]
 [-1.11364969  0.36833955  0.18333333  0.35294118]
 [-1.31185963  0.68838899  0.06666667  0.11764706]
 [ 0.7363097   1.81813793  0.43333333  0.17647059]
 [ 2.52019911 -0.91794613  0.3         0.        ]
 [-0.58508987 -0.18949533  0.2         0.29411765]
 [-0.91543976  1.58316692  0.23333333  0.47058824]
 [-1.54310455 -0.10472774  0.01666667  0.23529412]
 [-1.37792296  0.98904644  0.01666667  0.        ]
 [-1.21275466  0.49696812  0.1         0.11764706]
 [-1.04757971 -0.60352078  0.          0.05882353]
 [ 2.2559192   0.28258717  0.33333333  0.        ]
 [-0.45294991  0.61130463  0.06666667  0.05882353]
 [ 1.72735937  0.86856176  0.06666667  0.        ]
 [-0.28777496  1.0114824   0.05        0.17647059]
 [-1.11364969  0.69705701  0.01666667  0.11764706]
 [-0.84936978 -0.889362    0.01666667  0.05882353]
 [-0.12260002  0.25400305  0.28333333  0.11764706]
 [-2.10469936  0.06820623  0.23333333  0.29411765]
 [-1.0806147  -1.13232708  0.15        0.52941176]
 [ 0.10864449 -0.51776836  0.51666667  0.29411765]
 [ 1.00058961 -0.889362    0.16666667  0.47058824]
 [ 1.13272957 -0.81790616  0.1         0.05882353]
 [-1.37792296 -0.04613028  0.06666667  0.29411765]
 [ 0.60416974 -0.53206042  0.08333333  0.11764706]
 [-0.68419483  0.78280938  0.05        0.35294118]
 [-0.61812486  0.48267606  0.11666667  0.05882353]
 [-0.91543976 -1.40387628  0.06666667  0.05882353]
 [ 0.60416974  1.19727922  0.          0.23529412]
 [ 0.7363097   1.44024429  0.06666667  0.05882353]
 [ 0.67023972 -0.67498106  0.31666667  0.41176471]
 [-0.05653004  0.45409193  0.06666667  0.11764706]
 [ 2.18984922  0.76851732  0.23333333  0.47058824]
 [ 1.23183453 -0.10329853  0.15        0.41176471]
 [ 2.42109414 -0.889362    0.76666667  0.        ]
 [ 0.50506477 -0.90365407  0.5         0.05882353]
 [-0.88240477 -0.10472774  0.06666667  0.05882353]
 [-1.0806147  -0.3462636   0.03333333  0.64705882]
 [ 1.44220952  0.70420706  0.01666667  0.11764706]
 [ 0.57113475  0.12537448  0.23333333  0.11764706]
 [ 2.52019911  1.25585763  0.68333333  0.35294118]
 [ 0.24078486  1.09723477  0.05        0.05882353]
 [ 0.76934469  0.22541892  0.31666667  0.23529412]
 [-0.68419483  0.08254102  0.35        0.        ]
 [ 0.14167989 -0.48918423  0.4         0.        ]
 [ 0.83541466  0.79710145  0.11666667  0.05882353]
 [-0.68419483 -0.21763504  0.          0.35294118]
 [ 0.40595981  0.06820623  0.4         0.17647059]
 [-0.94847475 -1.77546992  0.11666667  0.23529412]
 [-1.31185963 -0.68927312  0.01666667  0.17647059]
 [-1.17971967 -0.64639693  0.11666667  0.23529412]
 [ 1.82646434  0.1968348   0.61666667  0.47058824]
 [ 0.76934469  0.88285383  0.26666667  0.23529412]
 [ 1.9255693   0.39692368  0.26666667  0.58823529]
 [-0.98150974  1.08294271  0.06666667  0.        ]
 [ 2.5532341   1.28303159  0.11666667  0.        ]
 [-0.05653004  0.36833955  0.03333333  0.        ]
 [-0.15563501 -0.70356518  0.          0.05882353]
 [ 2.38805915  0.36833955  0.06666667  0.17647059]
 [-0.02349505 -0.66068899  0.          0.05882353]
 [ 0.00953994  0.28258717  0.2         0.        ]
 [-0.71722982  0.43979987  0.48333333  0.47058824]
 [ 0.83541466 -0.16046678  0.66666667  0.23529412]
 [ 0.04257493  0.33975543  0.2         0.29411765]
 [-1.11364969 -0.48918423  0.06666667  0.11764706]
 [-1.0806147  -0.47489217  0.01666667  0.11764706]
 [ 0.70327471  0.01103798  0.7         0.17647059]
 [-0.7832998  -2.01843499  0.          0.05882353]
 [-0.68419483  1.16869509  0.06666667  0.11764706]
 [-0.58508987  0.96860621  0.16666667  0.41176471]
 [ 0.63720473  0.98289827  0.61666667  0.70588235]
 [-2.53415422 -1.0465747   0.25        0.29411765]
 [-1.57613954 -0.78931756  0.1         0.29411765]
 [ 0.7363097   0.36833955  0.33333333  0.47058824]
 [ 1.23183453 -0.10329853  0.11666667  0.17647059]
 [-1.57613954 -1.60396516  0.06666667  0.        ]
 [-0.12260002  0.21112686  0.38333333  0.        ]
 [-0.05653004 -1.01799057  0.26666667  0.41176471]
 [ 0.24078486 -0.04613028  0.2         0.05882353]
 [-0.05653004 -0.47489217  0.13333333  0.52941176]
 [ 1.82646434  0.13966654  0.51666667  0.17647059]
 [-0.68419483 -1.24666358  0.08333333  0.05882353]
 [ 0.7363097  -0.87506994  0.          0.05882353]
 [-0.25473998 -0.18905091  0.          0.        ]
 [-0.65115984  1.89759031  0.28333333  0.58823529]
 [-0.32080995 -0.31767948  0.03333333  0.05882353]
 [ 0.14167989  0.21112686  0.55        0.35294118]
 [-0.91543976 -0.51776836  0.01666667  0.        ]
 [-0.05653004 -1.4324604   0.05        0.05882353]
 [-0.4859849  -0.83219375  0.13333333  0.41176471]
 [ 1.66128939  0.13966654  0.05        0.17647059]
 [ 1.49611445  1.91188237  0.1         0.29411765]
 [-0.51901989 -0.38913979  0.41666667  0.        ]
 [-0.81633479 -1.08945089  0.3         0.17647059]
 [-0.84936978  0.63988875  0.6         0.47058824]
 [-0.41991492 -0.76073343  0.51666667  0.11764706]
 [-0.55205488 -0.64639693  0.03333333  0.        ]
 [ 0.60416974 -1.46104453  0.          0.        ]
 [ 0.07560992  0.11108242  0.31666667  0.52941176]
 [ 0.1086449   0.23971099  0.28333333  0.29411765]
 [ 1.00058961  0.45409193  0.11666667  0.35294118]
 [-1.54310455  0.42550781  0.3         0.47058824]
 [ 0.83541466  0.83997764  0.13333333  0.11764706]
 [ 0.57113475  0.56842844  0.06666667  0.        ]
 [ 1.13272957  0.23971099  0.41666667  0.47058824]
 [ 1.9255693   1.68320936  0.03333333  0.        ]
 [-1.04757971 -0.16046678  0.03333333  0.05882353]
 [ 0.86844965  0.1968348   0.73333333  0.29411765]
 [ 1.3309395  -0.97511438  0.43333333  0.58823529]
 [ 1.95860429  1.56887286  0.33333333  0.05882353]
 [-0.55205488 -0.3462636   0.33333333  0.35294118]
 [-0.71722982 -0.7750255   0.18333333  0.35294118]
 [-1.17971967  1.04006652  0.1         0.11764706]
 [ 2.52019911  0.62559669  0.16666667  0.23529412]
 [ 1.39700948  0.95431414  0.2         0.05882353]
 [-0.38687993 -1.0465747   0.1         0.35294118]
 [ 0.14167989  0.75422526  0.5         0.41176471]
 [ 1.46307946  2.85515854  0.03333333  0.        ]
 [-0.15563501 -1.46104453  0.26666667  0.23529412]
 [-0.84936978 -1.20378739  0.01666667  0.05882353]
 [ 0.76934469  1.28303159  0.11666667  0.05882353]
 [ 1.43004447  0.06820623  0.48333333  0.05882353]
 [ 0.70327471 -1.08945089  0.          0.11764706]
 [ 0.14167989 -0.10329853  0.05        0.17647059]
 [ 0.80237968 -0.28909535  0.53333333  0.52941176]
 [-0.28777496  0.29687924  0.06666667  0.05882353]
 [-1.64220952 -0.6178128   0.01666667  0.11764706]
 [ 0.07560992  0.18254273  0.21666667  0.35294118]
 [ 1.00058961  1.3973681   0.          0.        ]
 [-0.88240477 -0.90365407  0.          0.11764706]
 [ 0.27381985 -0.68927312  0.16666667  0.23529412]
 [-0.71722982 -0.96082232  0.11666667  0.23529412]
 [ 0.04257493 -1.33241596  0.31666667  0.17647059]
 [ 1.69432438  1.44024429  0.11666667  0.05882353]
 [ 0.04257493  0.38263162  0.13333333  0.23529412]
 [ 0.27381985  0.51126019  0.3         0.58823529]
 [ 0.57113475  0.32546336  0.          0.        ]
 [ 0.83541466  1.16869509  0.38333333  0.        ]
 [-1.21275466  0.65418082  0.11666667  0.05882353]
 [ 1.85949933  0.32546336  0.          0.        ]
 [-0.58508987 -1.14661914  0.13333333  0.35294118]
 [-0.35384494  0.75422526  0.15        0.23529412]
 [-0.35384494  0.01103798  0.1         0.11764706]
 [-0.02349505 -0.38913979  0.21666667  0.23529412]
 [ 1.09969458  1.96905063  0.1         0.35294118]
 [ 2.05770926 -1.28953977  0.18333333  0.47058824]
 [ 0.27381985  0.5827205   0.18333333  0.17647059]
 [-0.41991492 -1.00369851  0.01666667  0.11764706]
 [-0.61812486 -0.21763504  0.25        0.35294118]
 [-0.58508987  1.56887286  0.2         0.05882353]
 [-0.65115984  0.48267606  0.08333333  0.        ]
 [-0.41991492  0.02533004  0.01666667  0.11764706]
 [ 0.17471488  0.88285383  0.46666667  0.47058824]
 [ 0.14167989 -0.10472774  0.55        0.47058824]
 [ 0.14167989 -1.40387628  0.          0.        ]
 [-1.11364969  0.75422526  0.05        0.05882353]
 [-0.32080995 -0.31767948  0.15        0.17647059]
 [-1.27882464 -1.5896731   0.03333333  0.05882353]
 [ 0.53809976  1.54028873  0.2         0.        ]
 [ 1.0336246   2.52644109  0.25        0.41176471]
 [-1.51901989  1.1968348   0.13333333  0.11764706]
 [-0.68419483  0.91145795  0.35        0.47058824]
 [-0.61812486  1.88328625  0.03333333  0.11764706]
 [-1.14668468  0.49696812  0.06666667  0.        ]
 [ 0.80237968 -1.44675247  0.6         0.76470588]]
```

2. Univariate Polynomial LogisticRegression

The first model to train will be the Univariate Polynomial Logistic Regression.

Polynomialize the independent variable:

In [24]:
```
#define the poly transformer
poly = PolynomialFeatures(3)
poly.fit(trainX_U)

trainX_U = poly.transform(trainX_U)
valX_U = poly.transform(valX_U)
```

Since the range of PG is high, I do some data scaling in order to enhance the model performance. Use standard scaler as PG has *Gaussian distribution*.

In [25]:
```
ss = StandardScaler()
ss.fit(trainX_U)

trainX_U = ss.transform(trainX_U)
valX_U = ss.transform(valX_U)
```

Check the model performance without tuning first. Then we will use f1 score to examine

In [26]:
```
clf1 = LogisticRegression()
clf1.fit(trainX_U, trainY_U.to_numpy().ravel())
```

Out[26]:
```
LogisticRegression()
```

In [27]:
```
train_U_pred = clf1.predict(trainX_U)
train_U_f1 = f1_score(trainY_U, train_U_pred, average = 'binary')
print("Train F1-Score score: {:.3f}".format(train_U_f1))
```

```
Train F1-Score score: 0.556
```

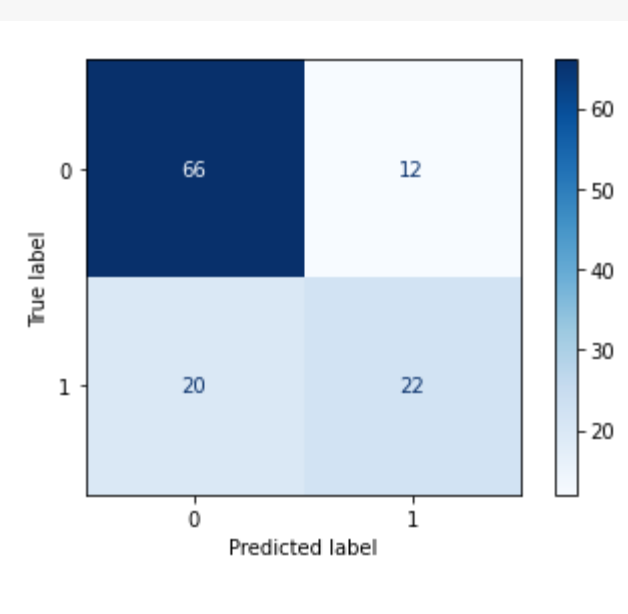
In [28]:
```
val_U_pred = clf1.predict(valX_U)
val_U_f1 = f1_score(valY_U, val_U_pred, average = 'binary')
print("Validate F1-Score score: {:.3f}".format(val_U_f1))
```

```
Validate F1-Score score: 0.579
```

As expected, the univariate, although in the form of polynomial model has low accuracy due to the fair correlation between the independent variable and the dependent. In addition, the validate score and the train are similar, hence, the model does not have to be tuned.

- Plot Confusion Matrix to validate the prediction result

In [29]:
```
cfs_matrix1 = plot_confusion_matrix(clf1, valX_U, valY_U, cmap=plt.cm.Blues)
plt.show()
```



The corrected prediction for negative is fairly high and the prediction for positive is inaccurate.

- Use KFold cross validation to evaluate the model accuracy accross the dataset

In [30]:
```
cv = KFold(n_splits=5, random_state=1, shuffle=True)
scores_U = cross_val_score(clf1, patientsFrame_Train_X_U, patientsFrame_Train_Y, scoring =
'f1', cv=cv, n_jobs=-1)
scores_U_mean = scores_U.mean()
print(scores_U)
print('Average KFold cross validation f1 score:', scores_U_mean)
```

```
[0.52777778 0.56756757 0.56716418 0.5974026  0.51515152]
Average KFold cross validation f1 score: 0.555012727400787
```

3. Multivariate Generalized Logistic Regression

The second model to train will be the Multivariate Generalized Logistic Regression.

In [31]:
```
clf2 = LogisticRegression()
clf2.fit(trainX_M, trainY_M)
```

Out[31]:
```
LogisticRegression()
```

In [32]:
```
train_M_pred = clf2.predict(trainX_M)
train_M_f1 = f1_score(trainY_M, train_M_pred, average = 'binary')
print("Train F1-Score score: {:.3f}".format(train_M_f1))
```

```
Train F1-Score score: 0.657
```

In [33]:
```
val_M_pred = clf2.predict(valX_M)
val_M_f1 = f1_score(valY_M, val_M_pred, average = 'binary')
print("Validate F1-Score score: {:.3f}".format(val_M_f1))
```

```
Validate F1-Score score: 0.609
```

The above score show an overfitting pattern and the accuracy is higher than the previous model due to being more complex. Now, I will try to tune the hyperparameter and find the best params set.

Perform GridSearchCV to get the best parameters for the model. GridSearchCV is the method that runs through all the combination of provided parameters then give back the best result through cross validation. As GridSearchCV is an exhaustive search, hence it takes a lot of computational resources, therefore, I limit the number of grid to lower the total amount of combinations that need to be calculated.

- Solver will not have the 'sag' or 'saga' because the problem is not multiclass.
- l1 is not added as only liblinear solver supports it.

Moreover, the aimed approach of this GridSearchCV is only a greedy method and its goal is to find the relative best.

The data that I use to run in this search will be the overall data to avoid the overfitting upon the train set.

In [34]:
```
#get the full train set after scalling
trainX_M_all = np.concatenate((trainX_M, valX_M))
trainY_M_all = np.concatenate((trainY_M, valY_M))
#lists of values for params
solvers = ['newton-cg', 'lbfgs', 'liblinear']
penalty = ['l2']
c_values = np.arange(1,100,1)
fit_intercept = [True, False]
# define grid search
grid = dict(solver=solvers,penalty=penalty,C=c_values, fit_intercept = fit_intercept)
grid_search = GridSearchCV(estimator=clf2, param_grid=grid, n_jobs=-1, cv=5, scoring='f1',er
ror_score=0)
grid_result = grid_search.fit(trainX_M_all, trainY_M_all)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# these code are for examination
# means = grid_result.cv_results_['mean_test_score']
# stds = grid_result.cv_results_['std_test_score']
# params = grid_result.cv_results_['params']
# for mean, stdev, param in zip(means, stds, params):
#     print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.641768 using {'C': 2, 'fit_intercept': False, 'penalty': 'l2', 'solver': 'newton-c
g'}
```

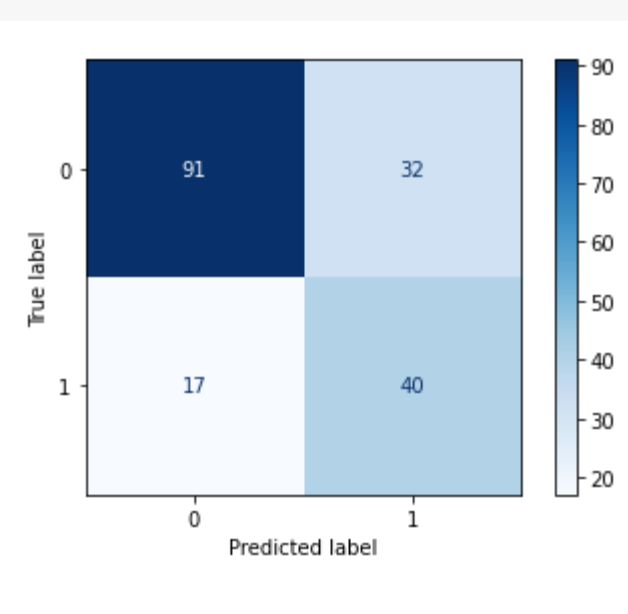
In [35]:
```
clf2 = LogisticRegression(C= 2, fit_intercept= False, penalty= 'l2', solver='newton-cg')
clf2.fit(trainX_M, trainY_M)

train_M_pred = clf2.predict(trainX_M)
train_M_f1 = f1_score(trainY_M, train_M_pred, average = 'binary')
print("Train F1-Score score: {:.3f}".format(train_M_f1))

start = time()
val_M_pred = clf2.predict(valX_M)
val_M_f1 = f1_score(valY_M, val_M_pred, average = 'binary')
end = time()
time2 = end - start
print('Processing time', time2,'s')
print("Validate F1-Score score: {:.3f}".format(val_M_f1))
```

```
Train F1-Score score: 0.660
Processing time 0.0031731128692626953 s
Validate F1-Score score: 0.620
```

In [36]:
```
cfs_matrix2 = plot_confusion_matrix(clf2, valX_M, valY_M, cmap=plt.cm.Blues)
plt.show()
```



The number of corrected prediction on negative is high with a fair amount of false, meanwhile the result on positive is pretty decent.

- KFold Cross Validation Test:

In [37]:
```
cv = KFold(n_splits=5, random_state=1, shuffle=True)
scores_M = cross_val_score(clf2, trainX_M_all, trainY_M_all, scoring = 'f1', cv=cv, n_jobs=-
1)
print(scores_M)
print('Average KFold cross validation f1 score:', scores_M.mean())
```

```
[0.54347826 0.63414634 0.5952381  0.66666667 0.70212766]
Average KFold cross validation f1 score: 0.6283314047624419
```

4. Random Forest Classifier

The third model is a tree-based algorithm, RandomForestClassifier.

The training procedure is the same as the previous processes.

- Initially run with the data with default parameters

In [38]:
```
clf3 = RandomForestClassifier(random_state=42)
clf3.fit(trainX_M, trainY_M)
```

Out[38]:
```
RandomForestClassifier(random_state=42)
```

In [39]:
```
train_M_pred_rf = clf3.predict(trainX_M)
train_M_f1_rf = f1_score(trainY_M, train_M_pred_rf, average = 'binary')
print("Train F1-Score score: {:.3f}".format(train_M_f1_rf))
```

```
Train F1-Score score: 1.000
```

In [40]:
```
val_M_pred_rf = clf3.predict(valX_M)
val_M_rf_f1 = f1_score(valY_M, val_M_pred_rf, average = 'binary')
print("Validate F1-Score score: {:.3f}".format(val_M_rf_f1))
```

```
Validate F1-Score score: 0.645
```

There's currently a overfitting pattern, so first, check the parameters that are in use

In [41]:
```
print('Parameters currently in use:\n')
print(clf3.get_params())
```

```
Parameters currently in use:

{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_dept
h': None, 'max_features': 'auto', 'max_leaf_nodes': None, 'max_samples': None, 'min_impuri
ty_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1, 'min_samples_split':
2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 100, 'n_jobs': None, 'oob_score': Fals
e, 'random_state': 42, 'verbose': 0, 'warm_start': False}
```

Hyperparameter Tuning

The approach of my tuning process of this model is also a greedy approach, the main goal is to find the relative best.

In [42]:
```
#number of trees in random forest
n_estimators = [100, 200, 500]
#number of features to consider at every split
max_features = ['auto', 'sqrt']
#minimum samples to split a node
min_samples_split = [2, 5, 10]
#minimum samples required at each leaf node
min_samples_leaf = [1, 2, 5]
#maximum number of levels in tree, our first run see an overfitting so the depth should not
 be high
max_depth = [5, 8, 12, 15, 20]

# define grid search
grid = dict(n_estimators=n_estimators,max_features=max_features, min_samples_split=min_sampl
es_split, max_depth=max_depth, min_samples_leaf=min_samples_leaf)
grid_search = GridSearchCV(estimator=clf3, param_grid=grid, n_jobs=-1, cv=5, scoring='f1',er
ror_score=0)
grid_result = grid_search.fit(trainX_M_all, trainY_M_all)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# these code are for examination
# means = grid_result.cv_results_['mean_test_score']
# stds = grid_result.cv_results_['std_test_score']
# params = grid_result.cv_results_['params']
# for mean, stdev, param in zip(means, stds, params):
#     print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.637847 using {'max_depth': 15, 'max_features': 'auto', 'min_samples_leaf': 2, 'min
_samples_split': 10, 'n_estimators': 100}
```

Apply the best params to the model with a subtle modification on the max_depth to lower the propability of overfitting.

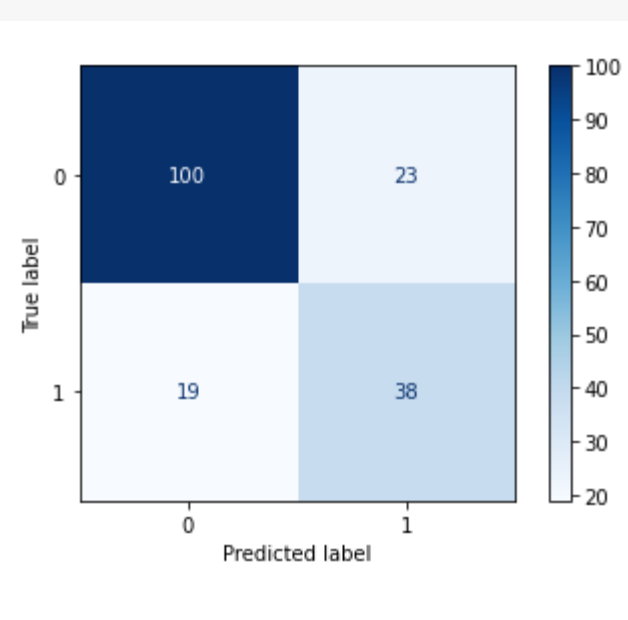
In [50]:
```
clf3 = RandomForestClassifier(max_depth=5, n_estimators=100, random_state=42,  max_features=
'auto', min_samples_leaf = 2, min_samples_split=10)
clf3.fit(trainX_M, trainY_M)

train_M_pred_rf = clf3.predict(trainX_M)
train_M_rf_f1 = f1_score(trainY_M, train_M_pred_rf, average = 'binary')
print("Train F1-Score score: {:.3f}".format(train_M_rf_f1))

start = time()
val_M_pred_rf = clf3.predict(valX_M)
val_M_rf_f1 = f1_score(valY_M, val_M_pred_rf, average = 'binary')
print("Validate F1-Score score: {:.3f}".format(val_M_rf_f1))
end = time()
time3 = end-start
print('Processing time', time3,'s')
```

```
Train F1-Score score: 0.763
Validate F1-Score score: 0.644
Processing time 0.01697516441345215 s
```

In [51]:
```
cfs_matrix3 = plot_confusion_matrix(clf3, valX_M, valY_M, cmap=plt.cm.Blues)
plt.show()
```



The result of negative prediction is extremly high and the prediction on positive is average.

- KFold CV test

In [52]:
```
cv = KFold(n_splits=5, random_state=1, shuffle=True)
scores_rf = cross_val_score(clf3, trainX_M_all, trainY_M_all, scoring = 'f1', cv=cv, n_jobs=
-1)
print(scores_rf)
print('Average KFold cross validation f1 score:', scores_rf.mean())
```

```
[0.61176471 0.61538462 0.64102564 0.71232877 0.68292683]
Average KFold cross validation f1 score: 0.6526861117368379
```

5. Model Comparision

Here is the summary of each model performance.

In [53]:
```
print('Average KFold cross validation f1 score of Univariate Polynomial Logistic Regression:
 ', scores_U_mean)
print('Average KFold cross validation f1 score of Multivariate Logistic Regression: ', score
s_M.mean())
print('Average KFold cross validation f1 score of Random Forest Classifier: ', scores_rf.mea
n())
```

```
Average KFold cross validation f1 score of Univariate Polynomial Logistic Regression:  0.5
55012727400787
Average KFold cross validation f1 score of Multivariate Logistic Regression:  0.6283314047
624419
Average KFold cross validation f1 score of Random Forest Classifier:  0.6526861117368379
```

Regarding the two models that use Logistic Regression, the *Multivariate* comes up with a significantly better results, since it complexity is higher. Moreover, with the provided dataset, the independent variables for the Sepssis attributes just fairly correlated with our target, recorded highest was only 0.45 for PL, which is also my chosen independent variable for the *Univariate* model (with the least accuracy and the worst performance). This statement also strengthen the idea that our model is hard to get higher score with simple algorithm such as Logistic Regression. So, there are only two trained models left that can be put into consideration.

Comparing the Multivariate LogisticRegression and RandomForestClassifier(both use the same processed dataset), we can see that although the RandomForest takes longer to process but it gives a more decent result, that can be seen from the f1 score and the two confusion matrices above. The RandomForest has a better prediction on the negative of the *val*, and similar accuracy on the positive estimation.

Hence, the best model is the **RandomForestClassifier**.

6. Predicting The Test Set

In [56]:
```
patientsFrame_Test_X_M = scaler.transform(patientsFrame_Test_X_M)
prediction = clf3.predict(patientsFrame_Test_X_M)
print(prediction)

prediction_df = pd.read_csv('s3749560_predictions.csv', delimiter=',')

#input the prediction to the Sepssis
prediction_df.loc[:, 'Sepssis'] = prediction

#convert back to Negative and Positive
seps_reverse = {0: 'Negative', 1: 'Positive'}
prediction_df['Sepssis'] = prediction_df['Sepssis'].map(seps_reverse)

prediction_df.to_csv('s3749560_predictions.csv', index=False)
prediction_df.head()
```

```
[0 0 0 0 1 1 0 1 0 1 0 0 1 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 1 0 1 1 1 1 1 0 1 0 0 1 1 0 0
 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 1 1 0 0 0 0 0 1 1 0 0 0 0 1 0
 1 0 1 0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 1 1 1
 0 1 1 0 0 0 1 1 1 1 1 0 1 0 1 0 0 0 0 1 0]
```

Out[56]:

	ID\t	Sepssis
0	ICU200609\t\t\t\t\t	Negative
1	ICU200610\t\t\t\t\t	Negative
2	ICU200611\t\t\t\t\t	Negative
3	ICU200612\t\t\t\t\t	Negative
4	ICU200613\t\t\t\t\t	Positive

IV. Conclusion

In general, this notebook goal has been achieved with the **clf3** RandomForestClassifier model.

On the other hand, the method that is used in this notebook to search the best parameters for each algorithm is relative, hence, for different run the results may vary but the average performance of the model will fairly change.

Regarding real worlds problem, these two approaches is not decent enough for many reasons. Personally, I, myself do not have enough knowledge yet to capable of coming up with the best solution for these algorithms yet. Secondly, the dataset as we have seen EDA is not strong enough (weak correlation) for these models to calculate the best results. Therefore, more advanced models and techniques should be implemented if we want to handle this problems sufficiently.

In [ ]: