predicting-diabetes07-06

June 7, 2024

1 PREDICTING DIABETES IN PIMA INDIAN WOMEN USING MACHINE LEARNING MODELS

Capstone Project-I submitted to Imarticus Learning

Submitted By Dr. DILEEP KUMAR SHETTY

A population of women who were at least 21 years old, of Pima Indian heritage and living near Phoenix, Arizona, was tested for diabetes according to World Health Organization criteria. The data were collected by the US National Institute of Diabetes and Digestive and Kidney Diseases. Some values are not within the expected range and should be treated as missing values. What method would be more effective for filling this type of missing value? Additionally, how should further classification proceed? npreg: Number of times pregnant

glu: Plasma glucose concentration a 2 hours in an oral glucose tolerance test

bp: Diastolic blood pressure (mm Hg)

skin:Skin Thickness(Triceps skin fold thickness (mm))

insulin: 2-Hour serum insulin (mu U/ml)

bmi: Body mass index (weight in kg/(height in m)^2)

ped: Diabetes pedigree function

age: Age (years)

type –yes or no for diabetic according to WHO criteria

2 OBJECTIVES

This project aims to examine the factors influencing diabetes among Pima Indian women. The main objectives are as follows:

i. To analyze the data on Pima Indian women using various ML models.

ii. To explore whether specific characteristics or factors within the studied population are linked to a higher likelihood of having diabetes. Identifying such sub-groups is valuable for understanding risk factors and tailoring interventions or treatments more effectively.

3 1.1 Libraries and modules commonly used in data analysis and machine learning in Python

[1]:
```
#Pandas is a powerful data manipulation library for Python.
import pandas as pd

#NumPy is a numerical computing library for Python.
import numpy as np

#Matplotlib is a plotting library for creating static, interactive, and␣
↪animated visualizations in Python.
import matplotlib.pyplot as plt

#ListedColormap is a class in Matplotlib used to create a colormap from a list␣
↪of colors.
from matplotlib.colors import ListedColormap

#Seaborn is a statistical data visualization library based on Matplotlib.
import seaborn as sns

#is_string_dtype is a function from Pandas used to check if a dtype is of␣
↪object type.
from pandas.api.types import is_string_dtype

#StandardScaler is a preprocessing technique used to standardize features by␣
↪removing the mean and scaling to unit variance.
from sklearn.preprocessing import StandardScaler

#train_test_split is a function in scikit-learn used for splitting a dataset␣
↪into training and testing sets.
from sklearn.model_selection import train_test_split
```

[2]:
```
#The metrics module in scikit-learn provides various metrics for evaluating␣
↪model performance.
from sklearn import metrics

#LogisticRegression is a class in scikit-learn used for logistic regression␣
↪modeling.
from sklearn.linear_model import LogisticRegression

#classification_report is a function in scikit-learn that generates a text␣
↪report showing the main classification metrics.
from sklearn.metrics import classification_report

#cohen_kappa_score is a function in scikit-learn used for calculating the␣
↪Cohen's kappa statistic.
```

```
from sklearn.metrics import cohen_kappa_score

#confusion_matrix is a function in scikit-learn that computes the confusion␣
↪matrix to evaluate the accuracy of a classification.
from sklearn.metrics import confusion_matrix

#roc_auc_score is a function in scikit-learn used for computing the area under␣
↪the ROC AUC.
from sklearn.metrics import roc_auc_score

#roc_curve is a function in scikit-learn used for generating receiver operating␣
↪characteristic (ROC) curves.
from sklearn.metrics import roc_curve

#SGDClassifier is a class in scikit-learn implementing linear classifiers with␣
↪Stochastic Gradient Descent training.
from sklearn.linear_model import SGDClassifier

#DecisionTreeClassifier is a class in scikit-learn for building decision tree␣
↪models.
from sklearn.tree import DecisionTreeClassifier

#GridSearchCV is a class in scikit-learn for hyperparameter tuning using grid␣
↪search.
from sklearn.model_selection import GridSearchCV

#The tree module in scikit-learn provides tools for working with decision trees.
from sklearn import tree

#export_graphviz is a function in scikit-learn for exporting decision tree␣
↪models to Graphviz format.
from sklearn.tree import export_graphviz
```

[3]:
```
#Statsmodels is a library for estimating and testing statistical models.
import statsmodels
import statsmodels.api as sm

#SVC is a class in scikit-learn implementing Support Vector Classification.
from sklearn.svm import SVC

#GaussianNB is a class in scikit-learn implementing Gaussian Naive Bayes␣
↪classification.
from sklearn.naive_bayes import GaussianNB

#KNeighborsClassifier is a class in scikit-learn for k-nearest neighbors␣
↪classification.
```

[4]:
```
#Ignore Warnings:
import warnings
from warnings import filterwarnings
filterwarnings('ignore')

#Adjust Figure Size for Matplotlib:
plt.rcParams['figure.figsize'] = [10,4]
```

[5]:
```
#Adjusting some display and print options for Pandas and NumPy
#max_columns to None, Pandas not to truncate the display of columns.
pd.options.display.max_columns = None

##max_rows to None, Pandas not to truncate the display of rows.
pd.options.display.max_rows = None

# To see the full numeric values without exponential notation.
np.set_printoptions(suppress=True)
```

[6]:
```
#The os.chdir function is used to change the current working directory to the␣
↪specified path.
import os
os.chdir("C:\DKS\Machine_Learning\Extra_Projects")

##Load the Dataset
data = pd.read_csv('diabetes.csv')

#The sample(15) method is used to display a random sample of 15 rows from the␣
↪loaded DataFrame
data.sample(15)
```

[6]:
```
     npreg  glu  bp  skin  insulin   bmi    ped  age  type
25      10  125  70    26      115  31.1  0.205   41     1
152      9  156  86    28      155  34.3  1.189   42     1
466      0   74  52    10       36  27.8  0.269   22     0
323     13  152  90    33       29  26.8  0.731   43     1
465      0  124  56    13      105  21.8  0.452   21     0
603      7  150  78    29      126  35.2  0.692   54     1
172      2   87   0    23        0  28.9  0.773   25     0
372      0   84  64    22       66  35.8  0.545   21     0
320      4  129  60    12      231  27.5  0.527   31     0
233      4  122  68     0        0  35.0  0.394   29     0
701      6  125  78    31        0  27.6  0.565   49     1
585      1   93  56    11        0  22.5  0.417   22     0
750      4  136  70     0        0  31.2  1.182   22     1
397      0  131  66    40        0  34.3  0.196   22     1
755      1  128  88    39      110  36.5  1.057   37     1
```

```
[7]: # Display summary statistics
     summary_stats = data.describe()
     summary_stats
```

```
[7]:             npreg         glu          bp        skin     insulin         bmi  \
     count  768.000000  768.000000  768.000000  768.000000  768.000000  768.000000
     mean     3.845052  120.894531   69.105469   20.536458   79.799479   31.992578
     std      3.369578   31.972618   19.355807   15.952218  115.244002    7.884160
     min      0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
     25%      1.000000   99.000000   62.000000    0.000000    0.000000   27.300000
     50%      3.000000  117.000000   72.000000   23.000000   30.500000   32.000000
     75%      6.000000  140.250000   80.000000   32.000000  127.250000   36.600000
     max     17.000000  199.000000  122.000000   99.000000  846.000000   67.100000

                   ped         age        type
     count  768.000000  768.000000  768.000000
     mean     0.471876   33.240885    0.348958
     std      0.331329   11.760232    0.476951
     min      0.078000   21.000000    0.000000
     25%      0.243750   24.000000    0.000000
     50%      0.372500   29.000000    0.000000
     75%      0.626250   41.000000    1.000000
     max      2.420000   81.000000    1.000000
```

In the variables Glucose, BP, Skin, Insulin, and BMI, the minimum values are zero. In reality, these factors are unlikely to have a minimum value of zero; instead, these zero values are indicative of missing data. Therefore, we need to replace these zeros with NA to accurately represent the absence of valid information.

```
[8]: #This lines selects specific columns from the original DataFrame (data)
     data_Correct=data.iloc[:,[0,6,7,8]]
     data_missing=data.iloc[:,1:6]

     #This line replaces all occurrences of 0 in the data_missing DataFrame with NaN.
       ↪
     #The inplace=True argument modifies the DataFrame in place.
     data_missing.replace(0, np.nan, inplace=True)

     #Concatenates (pd.concat) the two DataFrames (data_Correct and data_missing)␣
       ↪along the columns (axis=1).
     #The result is stored in the variable data.
     data= pd.concat([data_Correct,data_missing], axis=1)
     data.head()
```

```
[8]:    npreg    ped  age  type    glu    bp  skin  insulin   bmi
     0      6  0.627   50     1  148.0  72.0  35.0      NaN  33.6
     1      1  0.351   31     0   85.0  66.0  29.0      NaN  26.6
```

```
2      8  0.672   32     1  183.0  64.0   NaN      NaN  23.3
3      1  0.167   21     0   89.0  66.0  23.0     94.0  28.1
4      0  2.288   33     1  137.0  40.0  35.0    168.0  43.1
```

[9]:
```
#The dtypes attribute in Pandas is used to display the data types of each␣
↪column in a DataFrame.
data.dtypes
```

[9]:
```
npreg        int64
ped        float64
age          int64
type         int64
glu        float64
bp         float64
skin       float64
insulin    float64
bmi        float64
dtype: object
```

[10]:
```
# Check the info
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   npreg    768 non-null    int64
 1   ped      768 non-null    float64
 2   age      768 non-null    int64
 3   type     768 non-null    int64
 4   glu      763 non-null    float64
 5   bp       733 non-null    float64
 6   skin     541 non-null    float64
 7   insulin  394 non-null    float64
 8   bmi      757 non-null    float64
dtypes: float64(6), int64(3)
memory usage: 54.1 KB
```

[11]:
```
#Splitting the DataFrame into feature variables (data_x) and the target␣
↪variable (data_y).
data_x = data.iloc[:, data.columns != 'type']
data_y = data.iloc[:,data.columns == 'type']
print(data_y.head(2))
print(data_x.head(2))
```

```
   type
```

```
0     1
1     0
   npreg    ped  age    glu    bp  skin  insulin   bmi
0      6  0.627   50  148.0  72.0  35.0      NaN  33.6
1      1  0.351   31   85.0  66.0  29.0      NaN  26.6
```

[12]:
```
#Summary statistics table where the features are listed as rows, and the␣
↪summary statistics are columns.
data_x.describe().T
```

[12]:
```
         count        mean         std     min       25%       50%        75%  \
npreg    768.0    3.845052    3.369578   0.000   1.00000    3.0000    6.00000
ped      768.0    0.471876    0.331329   0.078   0.24375    0.3725    0.62625
age      768.0   33.240885   11.760232  21.000  24.00000   29.0000   41.00000
glu      763.0  121.686763   30.535641  44.000  99.00000  117.0000  141.00000
bp       733.0   72.405184   12.382158  24.000  64.00000   72.0000   80.00000
skin     541.0   29.153420   10.476982   7.000  22.00000   29.0000   36.00000
insulin  394.0  155.548223  118.775855  14.000  76.25000  125.0000  190.00000
bmi      757.0   32.457464    6.924988  18.200  27.50000   32.3000   36.60000

            max
npreg     17.00
ped        2.42
age       81.00
glu      199.00
bp       122.00
skin      99.00
insulin  846.00
bmi       67.10
```

4 Interpretation:

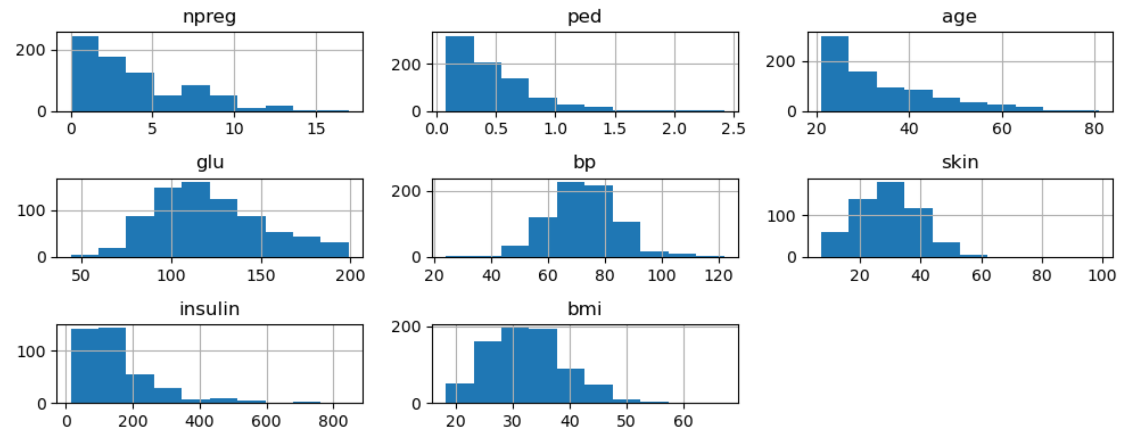
Glucose Level: The mean glucose level in the blood is approximately 119 mg/dL. It's worth noting that when fasting blood glucose falls between 100 to 125 mg/dL (5.6 to 6.9 mmol/L), it is typically recommended to consider lifestyle changes and monitor glycemia closely.

Blood Pressure (BP): The average blood pressure (BP) in our dataset is around 71 mmHg. It's important to mention that a normal blood pressure level is typically less than 120/80 mmHg.

BMI (Body Mass Index): The mean BMI in our dataset is 33.24. According to BMI categories, a BMI of 30.0 or higher falls within the obese range. Therefore, in our study, the average BMI suggests that participants are in the overweight range.

Age: The mean age in our study is 31 years, with the maximum age recorded being 67 years.

[13]:
```
# plot the histogram of numeric features variables
data_x.hist()
plt.tight_layout()
plt.show()
```



Interpretation: In this dataset, the variables age, ped, and npreg follow right-skewed distributions. BP and BMI, on the other hand, exhibit approximately symmetric distributions, while the variable skin displays a bimodal distribution.

Conclusions: Age: The variable “age” in our dataset is characterized by a right-skewed distribution. This means that the majority of individuals in our sample tend to be younger, with a few outliers who are older than the average age.

Ped: Similarly, the variable “ped” also displays a right-skewed distribution. This implies that most observations have relatively low values, while a smaller proportion of the data have higher values for this variable.

Npreg: The variable “npreg” exhibits a right-skewed distribution as well. This suggests that the majority of individuals in our dataset have a low number of pregnancies, while a smaller subset of the data includes individuals with a higher number of pregnancies.

BP: In contrast to the aforementioned variables, “BP” follows an approximately symmetric distribution. This implies that the values of blood pressure (BP) are evenly distributed around the mean, with a similar number of individuals having both high and low blood pressure readings.

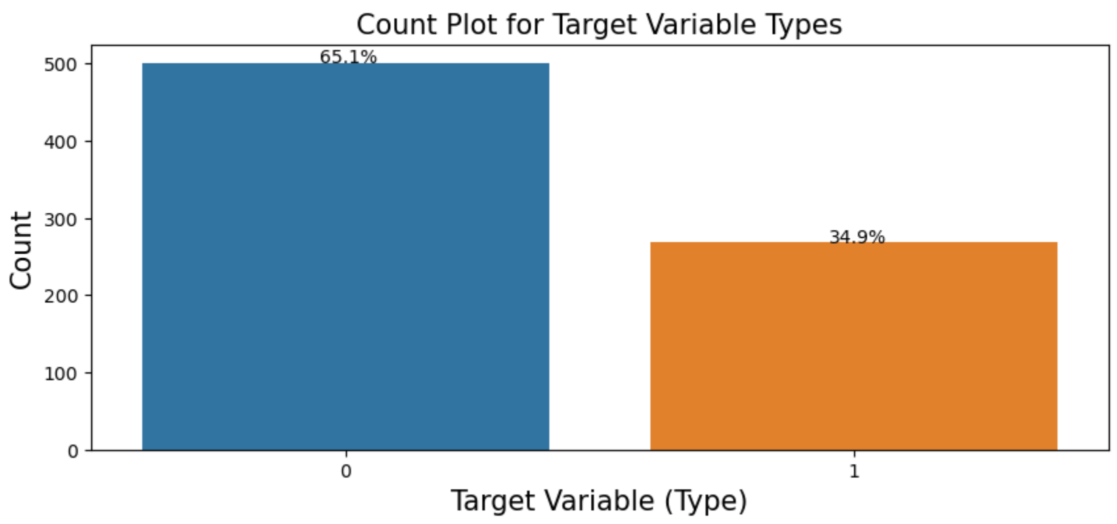
BMI: Like BP, the variable “BMI” also displays an approximately symmetric distribution. This suggests that the body mass index (BMI) values in our dataset are evenly spread around the mean, with a similar number of individuals having both low and high BMI values.

Skin: The variable “skin” is characterized by a bimodal distribution. This means that there are two distinct peaks or modes in the data, indicating that there are two groups of individuals in our sample with different values for this variable. These groups may have different characteristics related to skin measurements.

[14]:
```
class_frequency =data_y.value_counts()
class_frequency
```

[14]:
```
type
0       500
1       268
dtype: int64
```

[15]:
```
class_frequency =data_y.value_counts()
class_frequency
sns.countplot(data=data_y,x ="type")
plt.text(x = -0.05, y =data_y.value_counts()[0]+1, s =␣
  ↪str(round((class_frequency[0])*100/len(data_y),2)) + '%')
plt.text(x = 0.95, y =data_y.value_counts()[1]+1, s =␣
  ↪str(round((class_frequency[1])*100/len(data_y),2)) + '%')
plt.title('Count Plot for Target Variable Types', fontsize = 15)
plt.xlabel('Target Variable (Type)', fontsize = 15)
plt.ylabel('Count', fontsize = 15)
plt.show()
```



Interpretation:In our study, the target variable is "type," which signifies whether a person has diabetes or not. Here, the value 0 denotes non-diabetic individuals, indicating that they do not have diabetes, while the value 1 represents diabetics, indicating that they have been diagnosed with diabetes. Our study reveals that 32.83% of the participants are classified as diabetics, while 67.17% are classified as non-diabetics.

Conclusion: In our research study, we have a target variable called "type," which serves as an indicator of an individual's diabetes status. This variable takes on two distinct values:

When "type" is assigned the value 0, it signifies that the individual is categorized as non-diabetic. In other words, individuals with this value do not have diabetes.

When "type" is assigned the value 1, it indicates that the individual is categorized as diabetic. This means they have been diagnosed with diabetes.

Based on our analysis of the data, we have found that 34.90% of the study participants fall into the diabetic category (type = 1), indicating that they have diabetes. In contrast, 65.1% of the participants fall into the non-diabetic category (type = 0), signifying that they do not have diabetes.

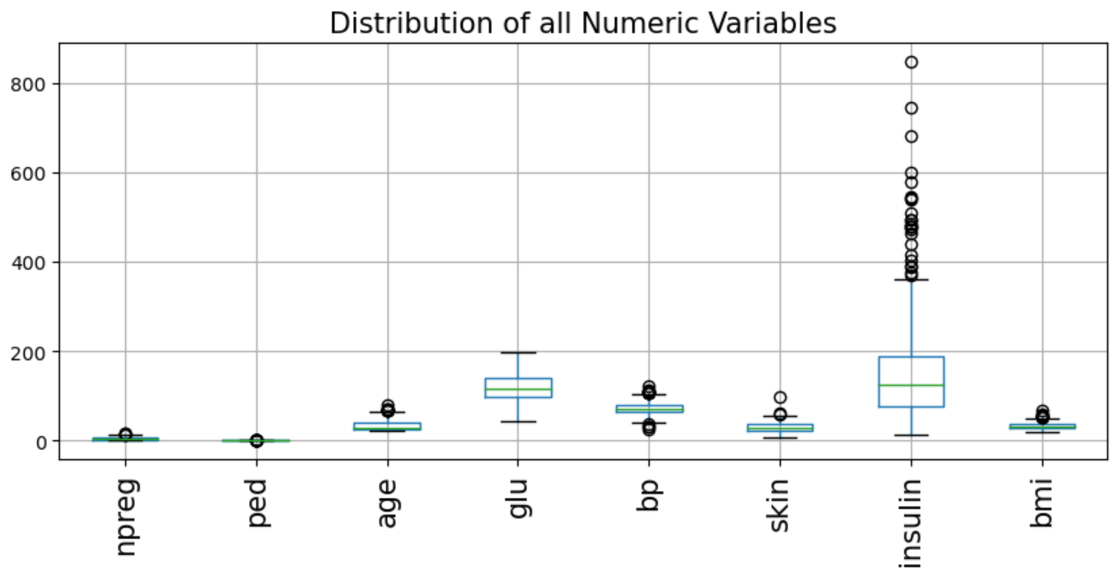
These percentages are important findings as they provide insights into the prevalence of diabetes within our study population. It suggests that a significant portion of the individuals in our study have been diagnosed with diabetes (34.90%), while the majority are non-diabetic (65.10%). This information is crucial for understanding the distribution of diabetes within the studied group and may have implications for healthcare interventions or further research related to diabetes management and prevention.

[ ]:

```
```

[16]:

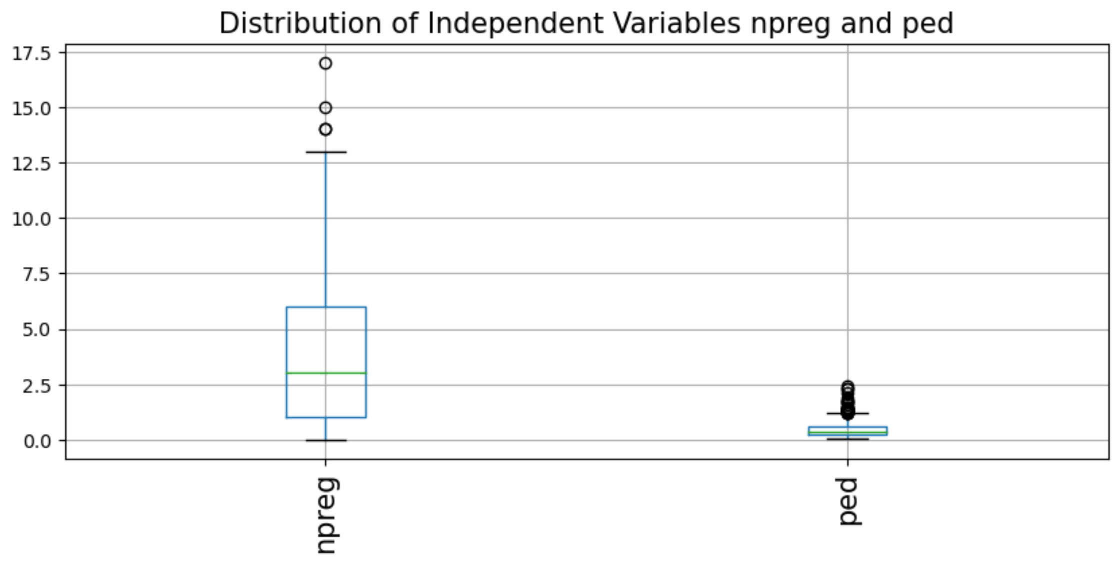
```
data_x.boxplot()
plt.title('Distribution of all Numeric Variables', fontsize = 15)
plt.xticks(rotation = 'vertical', fontsize = 15)
plt.show()
```



Notice that the variables ‘npreg’ and ‘ped’ has a quite small range as compared to the other variables. Thus, it is difficult to see the outliers for these variables. So, we plot the boxplot only for the variables ‘npreg’, ‘ped’.

[17]:

```
variables = [ 'npreg','ped']
data_x[variables].boxplot()
plt.title('Distribution of Independent Variables npreg and ped', fontsize = 15)
plt.xticks(rotation = 'vertical', fontsize = 15)
plt.show()
```



In our study, we've presented two plots that illustrate the distribution of data for various variables. These plots have revealed some interesting insights:

Outliers in all the Variables: For most of the variables we've analyzed, we've noticed the presence of data points that exceed the upper extreme of the distribution. This indicates that there are some values in our dataset that are significantly higher than the majority of the data. These extreme values are commonly referred to as outliers.

Outliers in Blood Pressure (BP): In the case of blood pressure (BP), we've identified a different scenario. Here, we have not only identified data points above the upper extreme but also some below the lower extreme. This suggests that there are both high and low extreme values in the BP variable.

Now, the decision we've made regarding these outliers is crucial and context-dependent. Given that our study is related to medical research, these extreme values may carry important clinical significance. Outliers in medical data can sometimes represent critical health conditions or unusual responses to treatment. Therefore, we have chosen not to remove these outliers from our dataset.

By retaining these extreme values, our analysis may capture the full spectrum of possible medical scenarios, including rare or unusual cases. This decision aligns with the goal of thorough medical research, which seeks to understand and account for all possible variations and conditions within the studied population.

```
[18]: Total = data.isnull().sum().sort_values(ascending = False)
      Percentage = (data.isnull().sum()*100/data.isnull().count()).
       ↪sort_values(ascending = False)
      Missing_Values = pd.concat([Total, Percentage], axis = 1, keys = ['Total',␣
       ↪'Percentage of missing observations'])
      Missing_Values
```

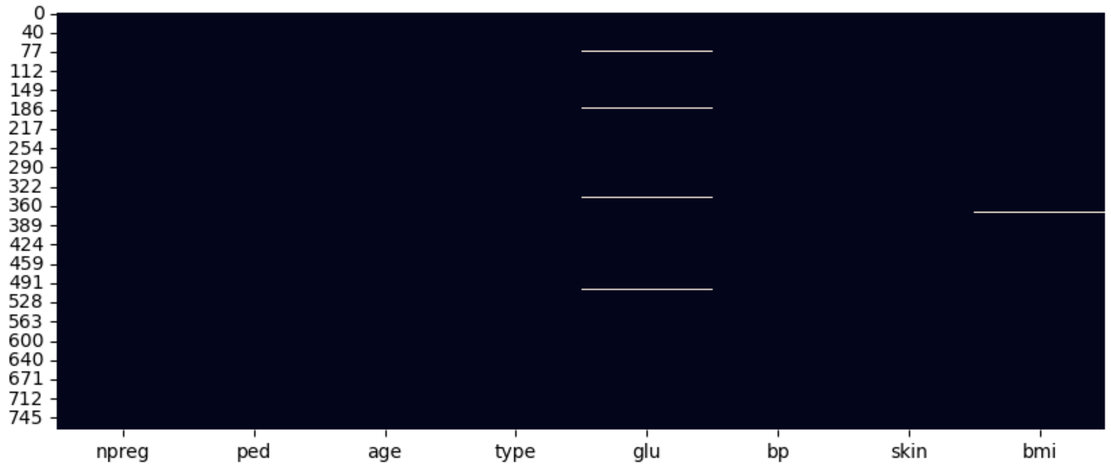
[18]:
```
         Total  Percentage of missing observations
insulin    374                           48.697917
skin       227                           29.557292
bp          35                            4.557292
bmi         11                            1.432292
glu          5                            0.651042
npreg        0                            0.000000
ped          0                            0.000000
age          0                            0.000000
type         0                            0.000000
```

Interpretation:With insulin having nearly 50% missing values in the dataset, it is prudent to consider removing this variable instead of filling or dropping missing values. Filling missing values with any central tendency measure risks losing originality, and dropping them results in losing almost half of the data. For other variables like skin, blood pressure (bp), and body mass index (bmi), excluding glucose (glu), filling missing values with the median is suitable due to the presence of outliers. For glu, filling missing values with the mean is a reasonable approach.

[19]:
```python
data.drop(['insulin'],axis=1, inplace=True)
data.shape
```

[19]:
```
(768, 8)
```

[20]:
```python
# Drop missing values for a specific feature

data.dropna(subset=['skin'], inplace=True)
sns.heatmap(data.isnull(), cbar=False)
plt.show()
```



[21]:
```python
data['glu'].fillna(data["glu"].mean() , inplace = True)
data['bp'].fillna(data["bp"].median() , inplace = True)
```

```
data['bmi'].fillna(data["bmi"].median() , inplace = True)
```

[22]:
```
Total = data.isnull().sum().sort_values(ascending = False)
Percentage = (data.isnull().sum()*100/data.isnull().count()).
 ↪sort_values(ascending = False)
Missing_Values = pd.concat([Total, Percentage], axis = 1, keys = ['Total',␣
 ↪'Percentage of missing observations'])
Missing_Values
```

[22]:
```
       Total  Percentage of missing observations
npreg      0                                 0.0
ped        0                                 0.0
age        0                                 0.0
type       0                                 0.0
glu        0                                 0.0
bp         0                                 0.0
skin       0                                 0.0
bmi        0                                 0.0
```

[23]:
```
data.shape
```

[23]:
```
(541, 8)
```

5 4.1 Univariate Analysis

1.glu-Plasma glucose concentration in an oral glucose tolerance test

[24]:
```
data.glu.describe()
```

[24]:
```
count    541.000000
mean     120.940299
std       30.787626
min       56.000000
25%       99.000000
50%      116.000000
75%      140.000000
max      199.000000
Name: glu, dtype: float64
```

The average plasma glucose concentration in an oral glucose tolerance test is 120.89 mg/dL, ranging from a minimum of 56 mg/dL to a maximum of 199 mg/dL. Among the observations, 25% show a glucose level less than or equal to 99 mg/dL, and 50% have a level less than or equal to 117 mg/dL. A 2-hour plasma glucose level below 140 mg/dL is considered normal. The range of 140-199 mg/dL indicates impaired glucose tolerance, while a level of 200 mg/dL or higher indicates diabetes.

6 Skewness and Kurtosis

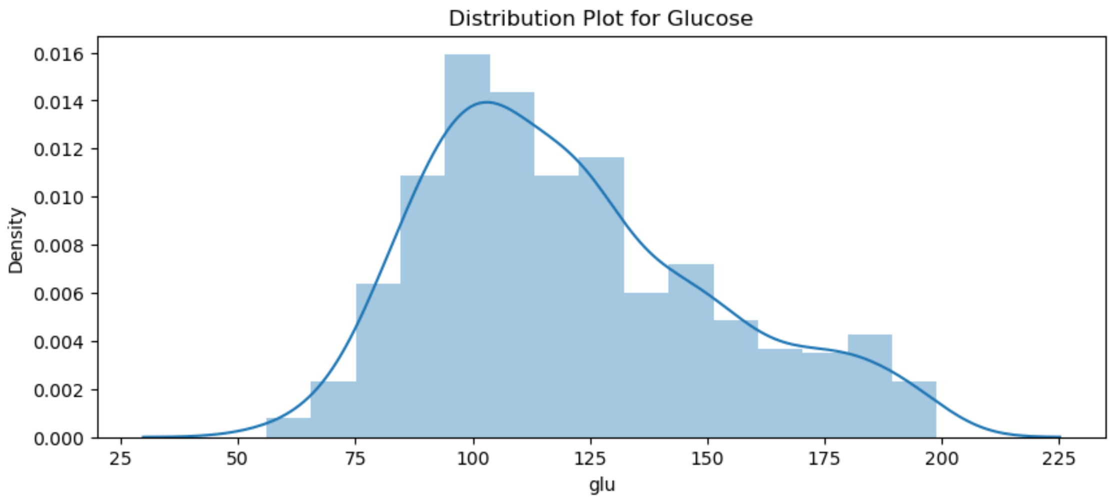
[25]:
```
print("Skewness: %f" % data['glu'].skew())
print("Kurtosis: %f" % data['glu'].kurt())
```

```
Skewness: 0.617323
Kurtosis: -0.294508
```

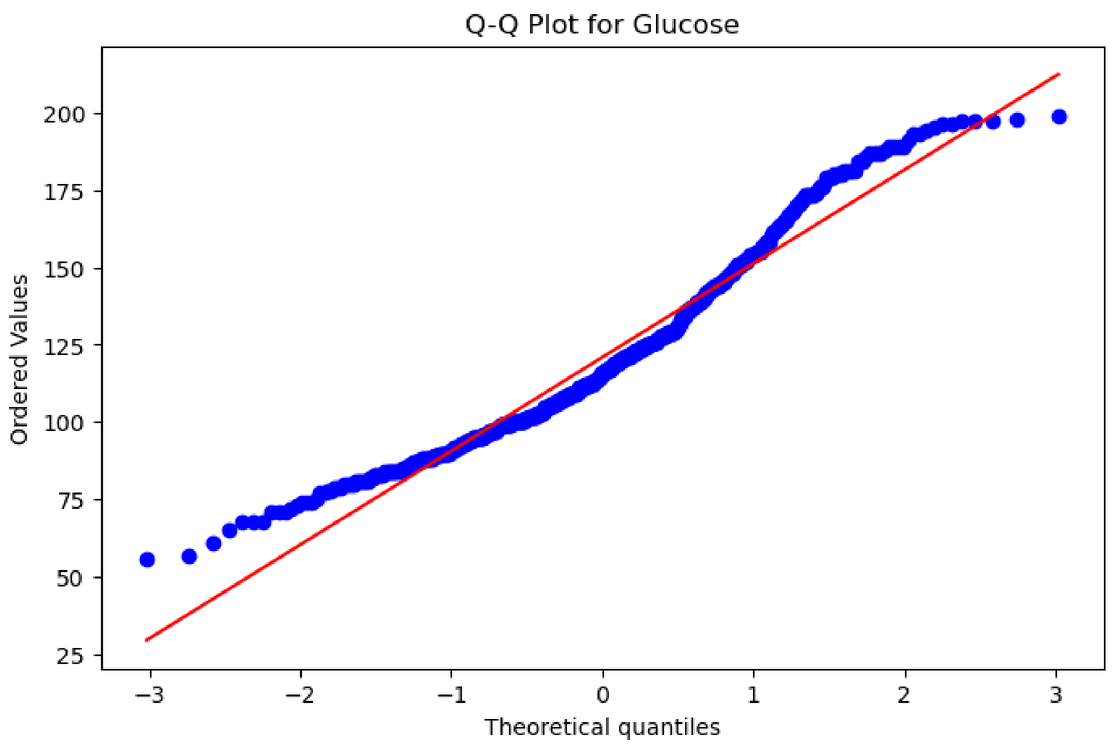
In summary, the skewness value suggests a minor asymmetry to the right in the distribution of plasma glucose concentration, while the kurtosis value indicates that the tails of the distribution are relatively close to those of a normal distribution. These results suggest that the distribution may be roughly symmetric with moderate tails, but a more detailed examination and statistical tests may be needed for a comprehensive assessment of normality.

[26]:
```
sns.distplot(data.glu)
plt.title("Distribution Plot for Glucose")
```

[26]:
```
Text(0.5, 1.0, 'Distribution Plot for Glucose')
```



[27]:
```
# q-q plot:q-q plot is used to compare the quantiles of two distributions
# p-p plot:p-p plot is the way to visual comparison of cdf of the two␣
  ↪distributions
import scipy.stats as stats
plt.figure(figsize = (8,5))
stats.probplot(data["glu"],plot=plt)
plt.title("Q-Q Plot for Glucose")
plt.show()
```



[28]:
```python
import numpy as np
from scipy.stats import jarque_bera


# Perform Jarque-Bera test
statistic, p_value = jarque_bera(data.glu)

# Display the results
print(f"Jarque-Bera statistic: {statistic}")
print(f"P-value: {p_value}")

# Check the null hypothesis
if p_value < 0.05:
    print("The glu does not come from a normal distribution (reject the null␣
 ↪hypothesis).")
else:
    print("The glu comes from a normal distribution (fail to reject the null␣
 ↪hypothesis).")
```

```
Jarque-Bera statistic: 36.23866327547381
P-value: 1.351681139466535e-08
The glu does not come from a normal distribution (reject the null hypothesis).
```

The confirmation of non-normal distribution for plasma glucose concentration in an oral glucose tolerance test is supported by the density plot, Q-Q plot, and Jarque-Bera test.

7 2.BP

```
[29]: data.bp.describe()
```

```
[29]: count    541.000000
      mean      71.463956
      std       12.261997
      min       24.000000
      25%       64.000000
      50%       72.000000
      75%       80.000000
      max      110.000000
      Name: bp, dtype: float64
```

The average blood pressure is approximately 71.46, reflecting a central tendency, but there is notable variability in the dataset. Quartile values offer insights into blood pressure distribution across different percentiles. The minimum value of 24 suggests potential risks, as extremely low blood pressure is associated with conditions such as severe hypotension, stiff arteries in the elderly, diabetes, arteriovenous malformation, and aortic dissection. The mean value of 71.46 falls within the normal range for blood pressure. Monitoring and understanding such variations are crucial for assessing potential health risks and conditions within the population.

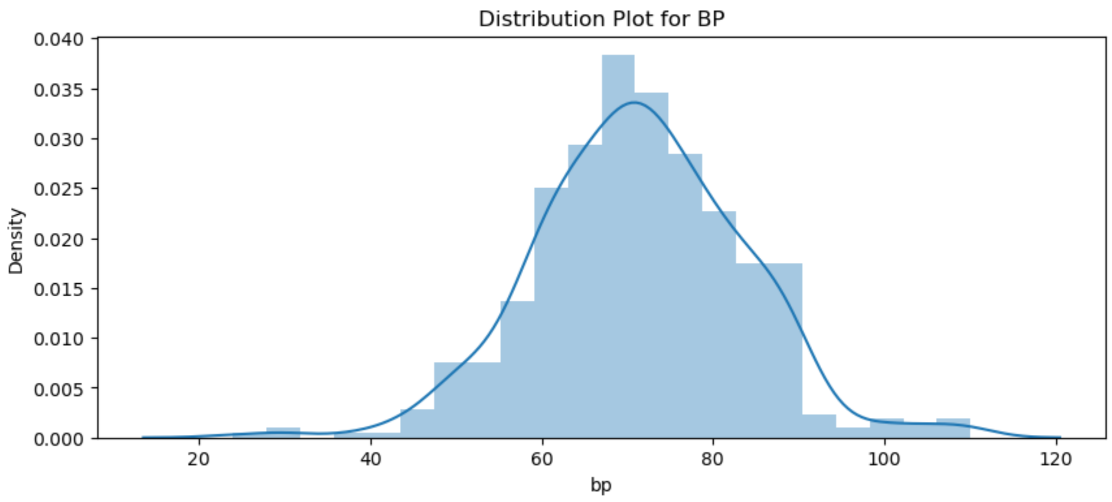
```
[30]: print("Skewness: %f" % data['bp'].skew())
      print("Kurtosis: %f" % data['bp'].kurt())
```

```
Skewness: -0.007283
Kurtosis: 0.823193
```

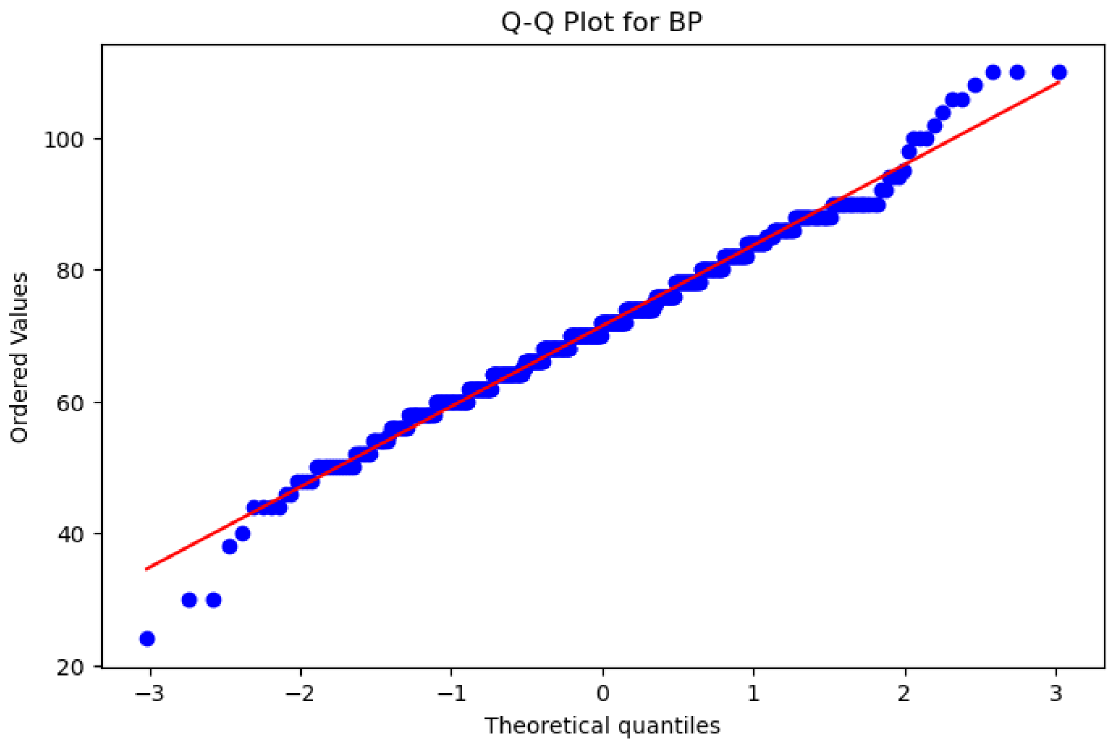
The skewness near zero suggests a nearly symmetric distribution of blood pressure, and the mesokurtic kurtosis indicates a moderate peak and tails, similar to what would be expected in a normal distribution. These characteristics provide insights into the shape and symmetry of the blood pressure distribution in your dataset.

```
[31]: sns.distplot(data.bp)
      plt.title("Distribution Plot for BP")
```

```
[31]: Text(0.5, 1.0, 'Distribution Plot for BP')
```



[32]:
```
plt.figure(figsize = (8,5))
stats.probplot(data["bp"],plot=plt)
plt.title("Q-Q Plot for BP")
plt.show()
```



[33]:
```
# Perform Jarque-Bera test
statistic, p_value = jarque_bera(data.bp)

# Display the results
print(f"Jarque-Bera statistic: {statistic}")
print(f"P-value: {p_value}")

# Check the null hypothesis
if p_value < 0.05:
    print("The BP does not come from a normal distribution (reject the null␣
 ↪hypothesis).")
else:
    print("The BP comes from a normal distribution (fail to reject the null␣
 ↪hypothesis).")
```

```
Jarque-Bera statistic: 14.595424282189091
P-value: 0.0006770860819314788
The BP does not come from a normal distribution (reject the null hypothesis).
```

The confirmation of non-normal distribution for Blood Pressure is supported by the density plot, Q-Q plot, and Jarque-Bera test.

8 3.Skin Type

[34]:
```
data.skin.describe()
```

[34]:
```
count    541.000000
mean      29.153420
std       10.476982
min        7.000000
25%       22.000000
50%       29.000000
75%       36.000000
max       99.000000
Name: skin, dtype: float64
```

The “skin thickness” values range from 7 to 99, with an average around 29.15. The data shows some variability, with a moderate spread of values.

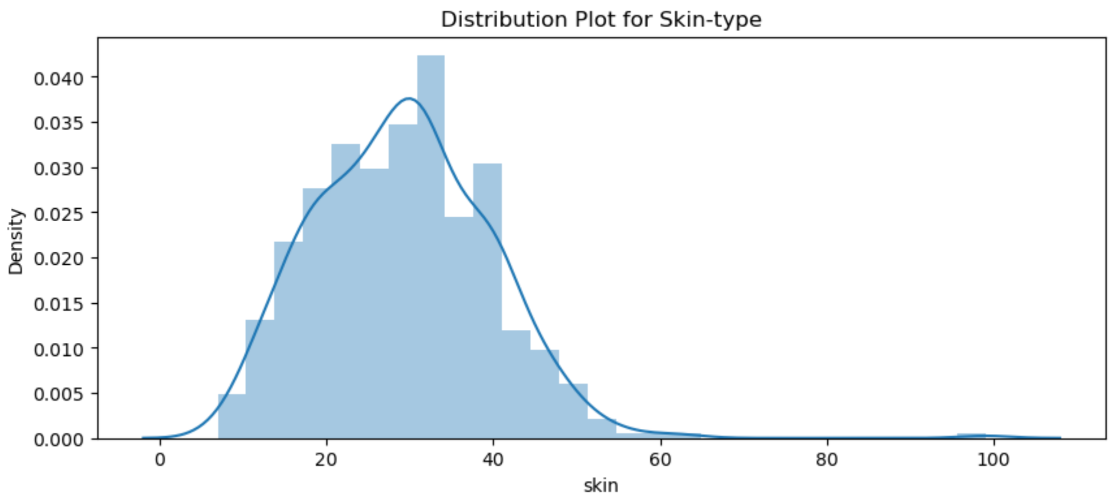
[35]:
```
print("Skewness: %f" % data['skin'].skew())
print("Kurtosis: %f" % data['skin'].kurt())
```

```
Skewness: 0.690619
Kurtosis: 2.935491
```

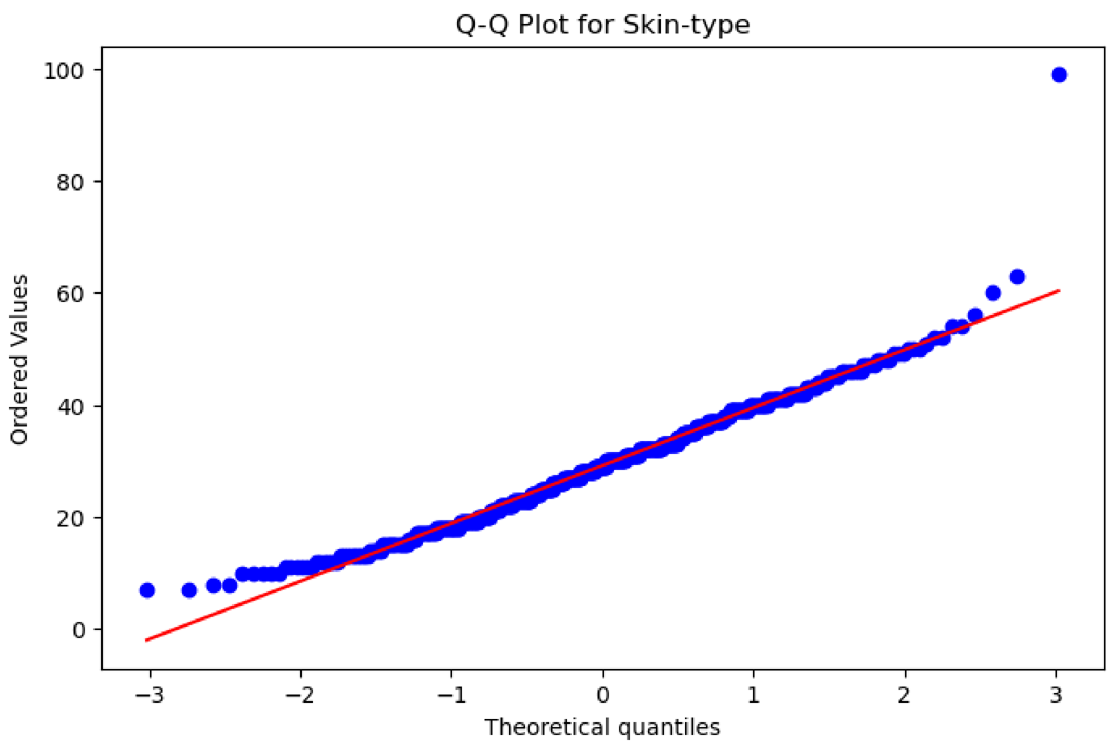
The positive skewness suggests a right-skewed distribution of skin thickness, and the leptokurtic kurtosis indicates heavier tails and a sharper peak. These characteristics provide insights into the asymmetry and concentration of values in the skin thickness distribution in your dataset.

```
[36]: sns.distplot(data.skin)
      plt.title("Distribution Plot for Skin-type")
```

```
[36]: Text(0.5, 1.0, 'Distribution Plot for Skin-type')
```



```
[37]: plt.figure(figsize = (8,5))
      stats.probplot(data["skin"],plot=plt)
      plt.title("Q-Q Plot for Skin-type")
      plt.show()
```



[38]:
```
# Perform Jarque-Bera test
statistic, p_value = jarque_bera(data.skin)

# Display the results
print(f"Jarque-Bera statistic: {statistic}")
print(f"P-value: {p_value}")

# Check the null hypothesis
if p_value < 0.05:
    print("The Skin-type does not come from a normal distribution (reject the␣
 ↪null hypothesis).")
else:
    print("The Skin-type comes from a normal distribution (fail to reject the␣
 ↪null hypothesis).")
```

```
Jarque-Bera statistic: 231.99762905260965
P-value: 4.191359802100761e-51
The Skin-type does not come from a normal distribution (reject the null
hypothesis).
```

The confirmation of non-normal distribution for Skin-type is supported by the density plot, Q-Q plot, and Jarque-Bera test.

9 4.Age

```
[39]: data.age.describe()
```

```
[39]: count    541.000000
      mean      31.558226
      std       10.743768
      min       21.000000
      25%       23.000000
      50%       28.000000
      75%       38.000000
      max       81.000000
      Name: age, dtype: float64
```

The average age in the study is 31.50 years, ranging from a minimum of 21 years to a maximum of 81 years. Twenty-five percent of women are younger than 23 years, and 50% of women are 28 years old.

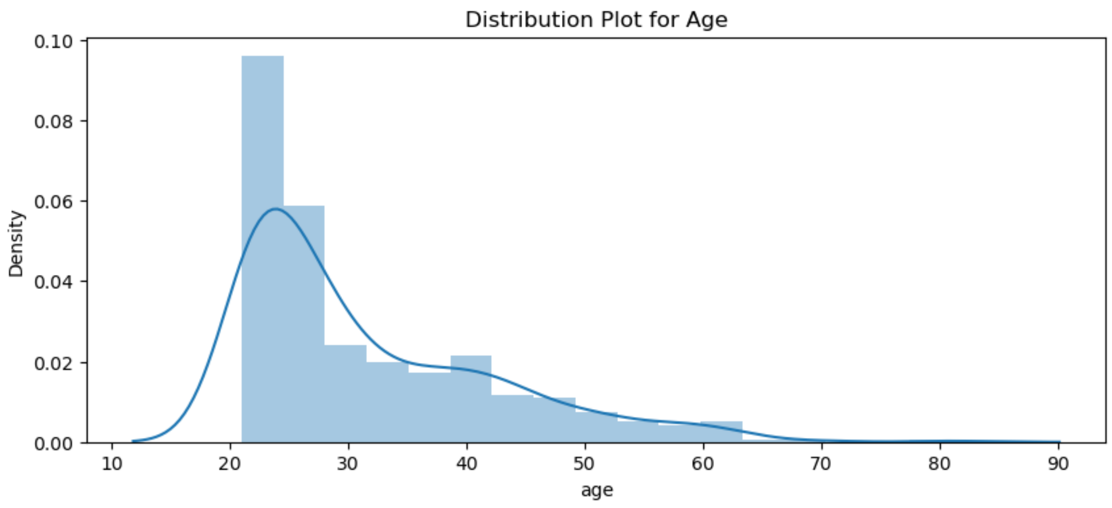
```
[40]: print("Skewness: %f" % data['age'].skew())
      print("Kurtosis: %f" % data['age'].kurt())
```

```
Skewness: 1.269905
Kurtosis: 1.180848
```

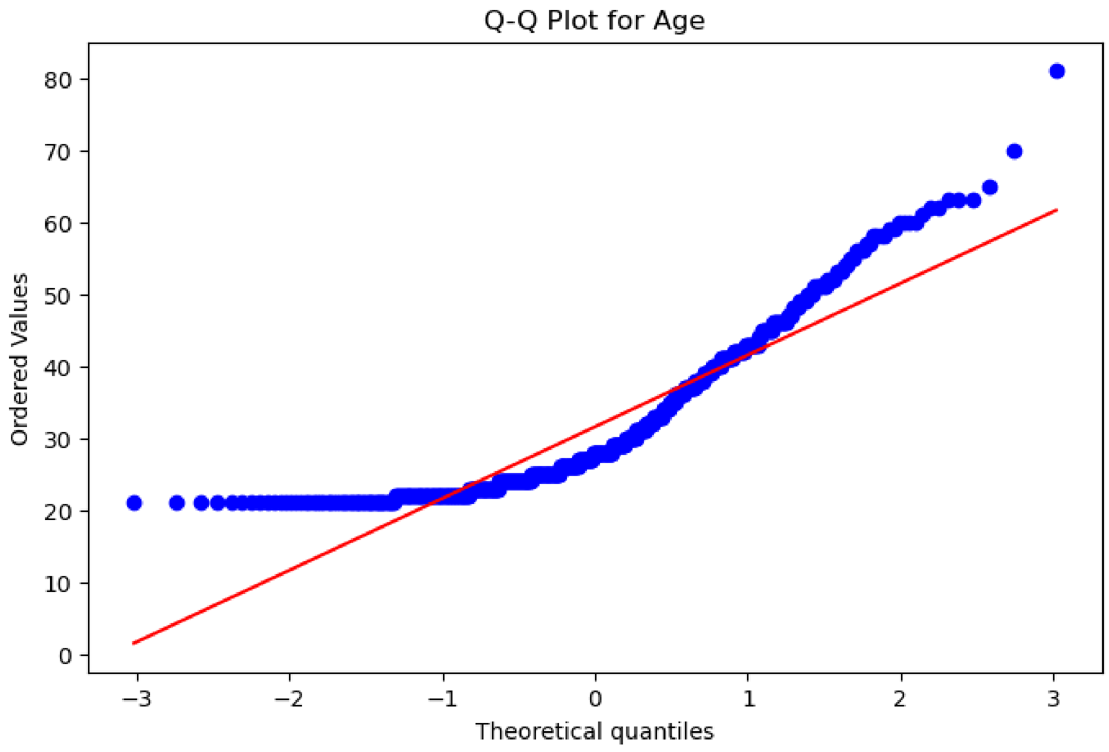
Based on these values, the age distribution in our study is likely to be somewhat skewed to the right with a few older individuals having ages considerably higher than the mean. Additionally, the distribution may exhibit heavier tails and a slightly more peaked shape compared to a normal distribution. Further analysis and visualization of the age distribution, such as a histogram or a density plot, could provide a clearer picture of the data distribution.

```
[41]: sns.distplot(data.age)
      plt.title("Distribution Plot for Age")
```

```
[41]: Text(0.5, 1.0, 'Distribution Plot for Age')
```



[42]:
```
plt.figure(figsize = (8,5))
stats.probplot(data["age"],plot=plt)
plt.title("Q-Q Plot for Age")
plt.show()
```



```
[43]: # Perform Jarque-Bera test
statistic, p_value = jarque_bera(data.age)

# Display the results
print(f"Jarque-Bera statistic: {statistic}")
print(f"P-value: {p_value}")

# Check the null hypothesis
if p_value < 0.05:
    print("The Age does not come from a normal distribution (reject the null␣
 ↪hypothesis).")
else:
    print("The Age comes from a normal distribution (fail to reject the null␣
 ↪hypothesis).")
```

```
Jarque-Bera statistic: 174.87643785709037
P-value: 1.061852055024055e-38
The Age does not come from a normal distribution (reject the null hypothesis).
```

The confirmation of non-normal distribution for age is supported by the density plot, Q-Q plot, and Jarque-Bera test.

10 5.BMI

```
[44]: data.bmi.describe()
```

```
[44]: count    541.000000
mean      32.895379
std        6.859116
min       18.200000
25%       27.900000
50%       32.800000
75%       36.900000
max       67.100000
Name: bmi, dtype: float64
```

```
[45]: # Calculate mode
mode_result = stats.mode(data.bmi)

print("Mode:", mode_result.mode[0])
```

```
Mode: 32.0
```

The mean Body Mass Index (BMI), computed as the ratio of weight in kilograms to the square of height in meters, is 32.89. The BMI ranges from a minimum of 18.20 to a maximum of 67.10. Notably, approximately 50% of women exhibit a BMI below 32.80, a value that aligns closely with both the mean and mode of the BMI. This indicates a close similarity between the mean, median, and mode, suggesting a distribution with a central tendency.

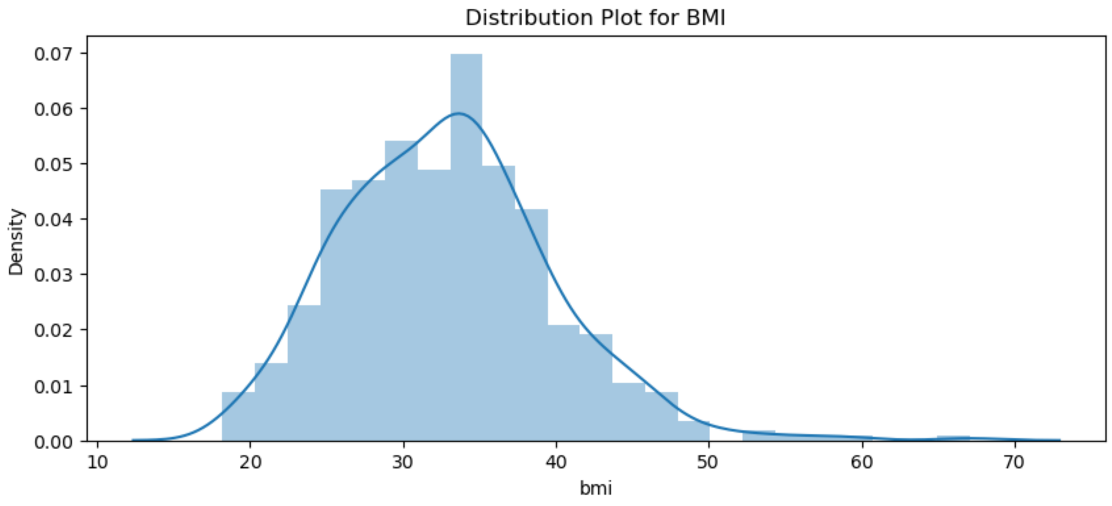
[46]:
```
print("Skewness: %f" % data['bmi'].skew())
print("Kurtosis: %f" % data['bmi'].kurt())
```

```
Skewness: 0.626830
Kurtosis: 1.272595
```

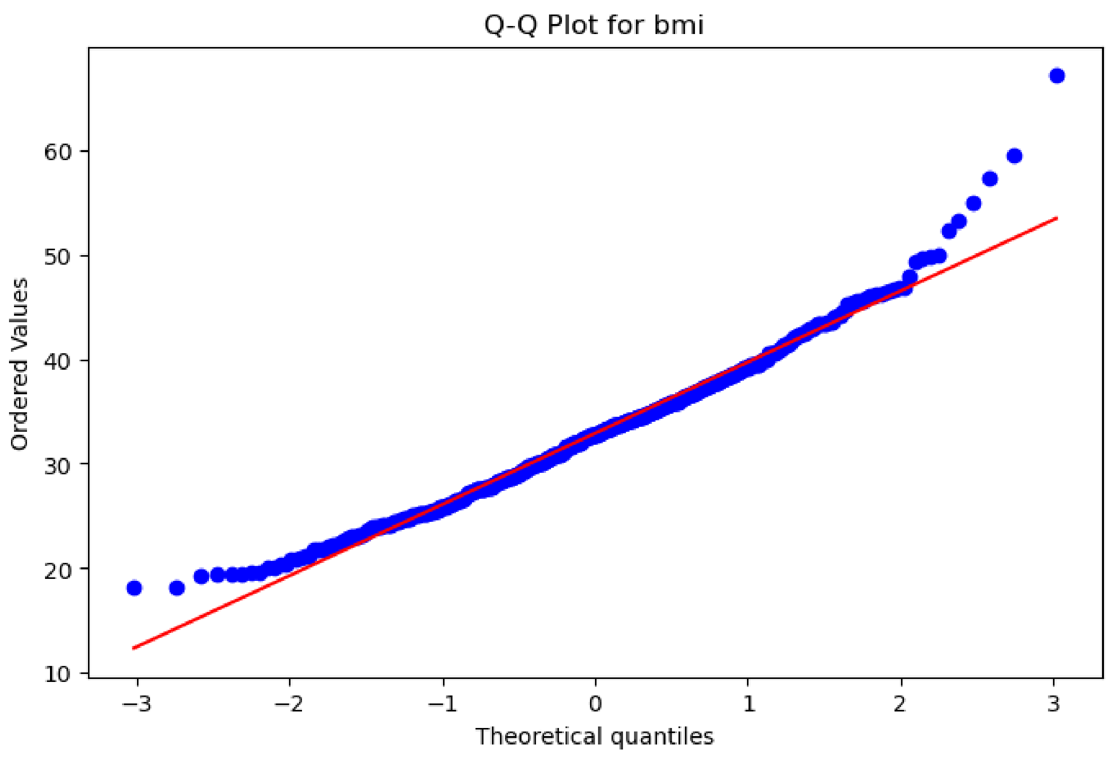
The distribution is right-skewed, suggesting a possible concentration of individuals with higher BMI values. The kurtosis indicates that the distribution has heavier tails and is more peaked than a normal distribution, possibly indicating the presence of more extreme BMI values.

[47]:
```
sns.distplot(data.bmi)
plt.title("Distribution Plot for BMI")
```

[47]:
```
Text(0.5, 1.0, 'Distribution Plot for BMI')
```



[48]:
```
plt.figure(figsize = (8,5))
stats.probplot(data["bmi"],plot=plt)
plt.title("Q-Q Plot for bmi")
plt.show()
```



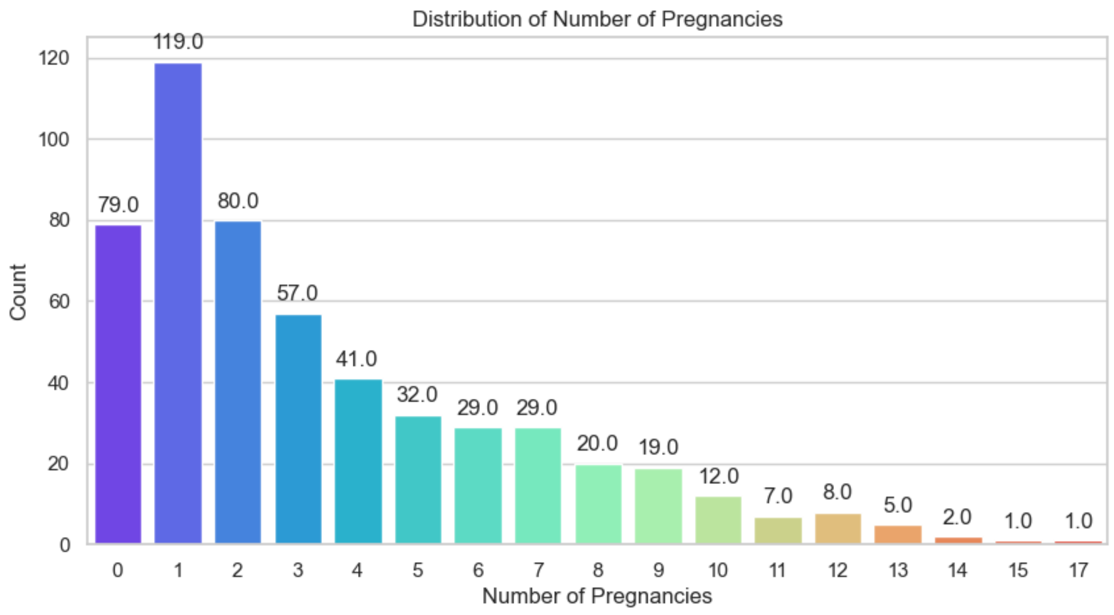
[49]:
```
# Perform Jarque-Bera test
statistic, p_value = jarque_bera(data.bmi)

# Display the results
print(f"Jarque-Bera statistic: {statistic}")
print(f"P-value: {p_value}")

# Check the null hypothesis
if p_value < 0.05:
    print("The BMI does not come from a normal distribution (reject the null␣
 ↪hypothesis).")
else:
    print("The BMI comes from a normal distribution (fail to reject the null␣
 ↪hypothesis).")
```

```
Jarque-Bera statistic: 70.44127573826687
P-value: 5.056748150849164e-16
The BMI does not come from a normal distribution (reject the null hypothesis).
```

The confirmation of non-normal distribution for BMI is supported by the density plot, Q-Q plot, and Jarque-Bera test.

11 6. Number of Pregnancies

```
[50]: sns.set(style="whitegrid")
      plt.figure(figsize=(10, 5))

      # Create the countplot
      ax = sns.countplot(x=data.npreg, palette="rainbow")

      # Add legends, labels, and values on each bar
      for p in ax.patches:
          ax.annotate(f'{p.get_height()}', (p.get_x() + p.get_width() / 2., p.
       ↪get_height()),
                      ha='center', va='center', xytext=(0, 10), textcoords='offset␣
       ↪points')

      plt.xlabel("Number of Pregnancies")
      plt.ylabel("Count")
      plt.title("Distribution of Number of Pregnancies")

      plt.show()
```

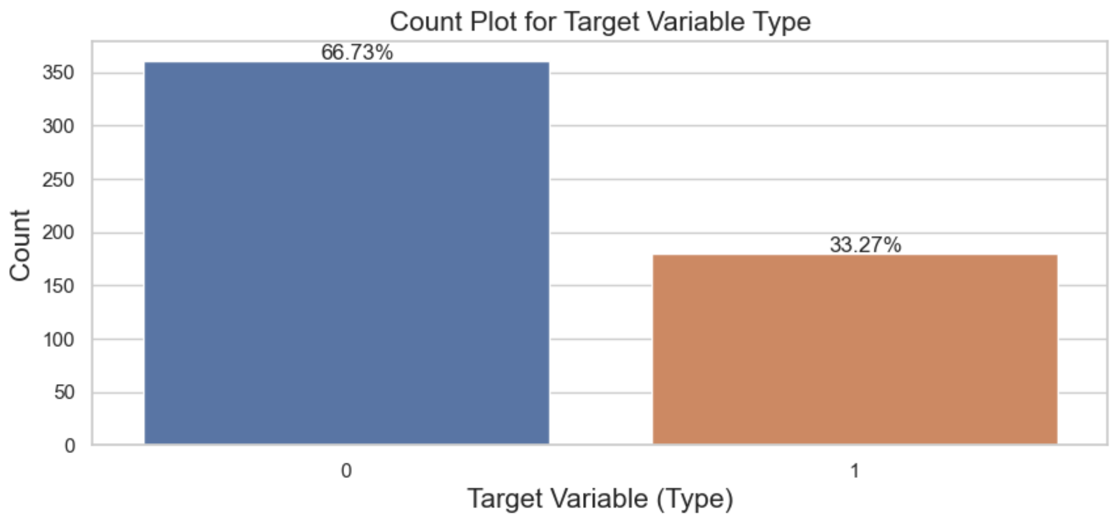


Where 79 women have no children and 119 women have one child, the mode of the distribution is determined to be 1.

12 7. Type (outcome): Diabetic (Yes or No)

```
[51]: #Splitting the DataFrame into feature variables (data_x) and the target␣
      ↪variable (data_y).
      data_x = data.iloc[:, data.columns != 'type']
      data_y = data.iloc[:,data.columns == 'type']
```

```
[52]: class_frequency =data_y.type.value_counts()
      sns.countplot(data=data_y,x ="type")
      plt.text(x = -0.05, y =data_y.value_counts()[0]+1, s =␣
       ↪str(round((class_frequency[0])*100/len(data_y),2)) + '%')
      plt.text(x = 0.95, y =data_y.value_counts()[1]+1, s =␣
       ↪str(round((class_frequency[1])*100/len(data_y),2)) + '%')
      plt.title('Count Plot for Target Variable Type', fontsize = 15)
      plt.xlabel('Target Variable (Type)', fontsize = 15)
      plt.ylabel('Count', fontsize = 15)
      plt.show()
```



Conclusion: In our research study, we have a target variable called "type," which serves as an indicator of an individual's diabetes status. This variable takes on two distinct values:

When "type" is assigned the value 0, it signifies that the individual is categorized as non-diabetic. In other words, individuals with this value do not have diabetes.

When "type" is assigned the value 1, it indicates that the individual is categorized as diabetic. This means they have been diagnosed with diabetes.

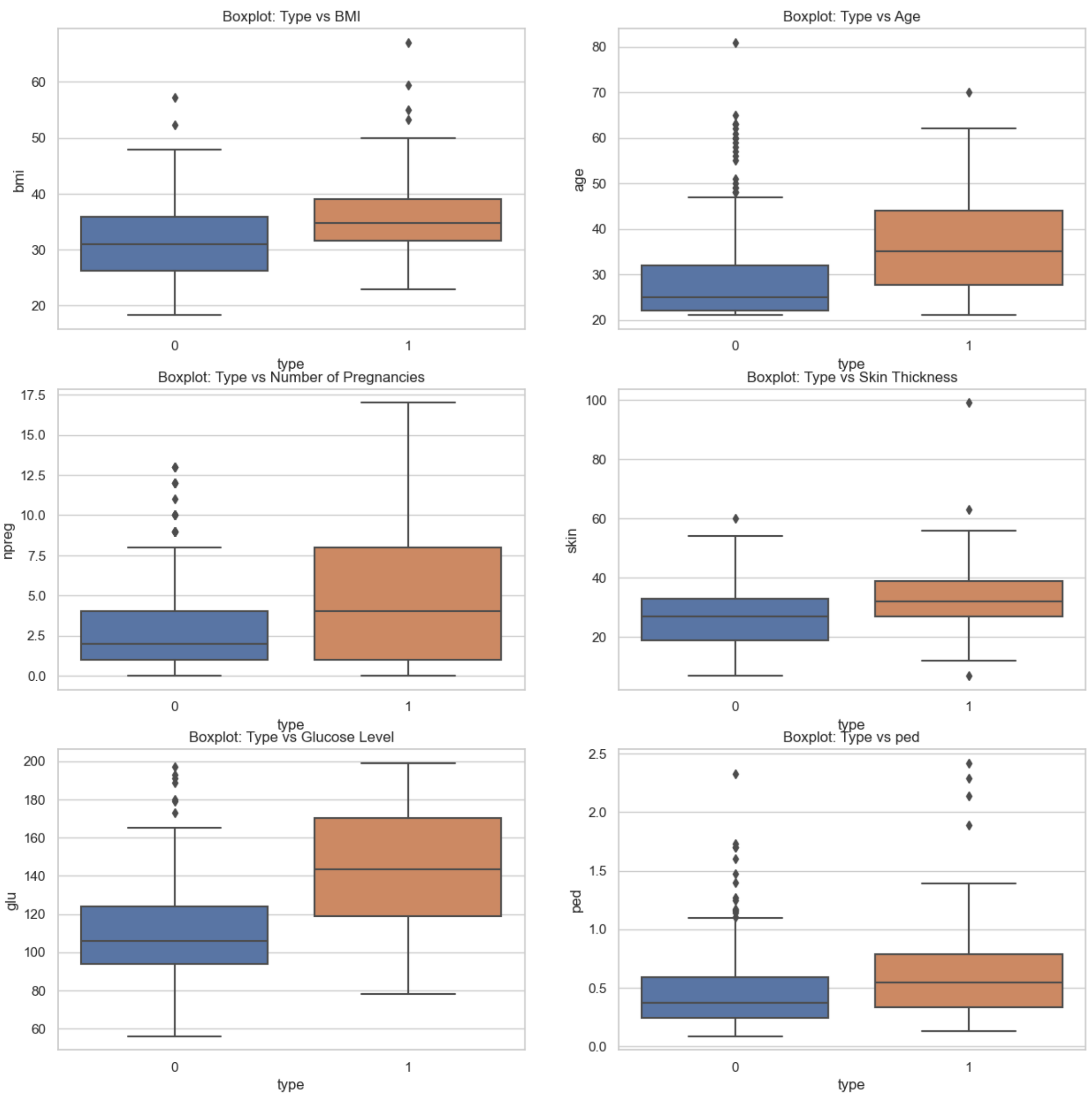
Based on our analysis of the data, we have found that 33.27% of the study participants fall into the diabetic category (type = 1), indicating that they have diabetes. In contrast, 66.73% of the participants fall into the non-diabetic category (type = 0), signifying that they do not have diabetes.

These percentages are important findings as they provide insights into the prevalence of diabetes within our study population. It suggests that a significant portion of the individuals in our study have been diagnosed with diabetes (33.27%), while the majority are non-diabetic (66.73%). This information is crucial for understanding the distribution of diabetes within the studied group and may have implications for healthcare interventions or further research related to diabetes management and prevention.

13 4.2 Multivariate Analysis

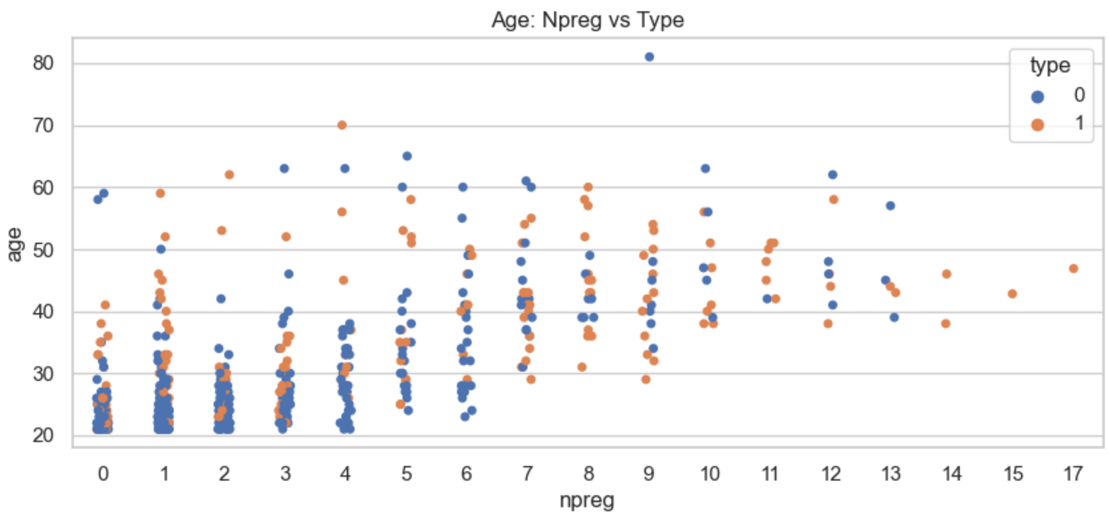
14 1.Box Plots for Target Variable (Type) with Different Features

[53]:
```python
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming 'data' is your DataFrame containing the variables
# 'type', 'bmi', 'age', 'npreg', 'skin', 'glu', 'bp'

# Set up the figure with subplots
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(15, 15))

# Boxplot for 'type' vs 'bmi'
sns.boxplot(x='type', y='bmi', data=data, ax=axes[0, 0])
axes[0, 0].set_title('Boxplot: Type vs BMI')

# Boxplot for 'type' vs 'age'
sns.boxplot(x='type', y='age', data=data, ax=axes[0, 1])
axes[0, 1].set_title('Boxplot: Type vs Age')

# Boxplot for 'type' vs 'npreg'
sns.boxplot(x='type', y='npreg', data=data, ax=axes[1, 0])
axes[1, 0].set_title('Boxplot: Type vs Number of Pregnancies')

# Boxplot for 'type' vs 'skin'
sns.boxplot(x='type', y='skin', data=data, ax=axes[1, 1])
axes[1, 1].set_title('Boxplot: Type vs Skin Thickness')

# Boxplot for 'type' vs 'glu'
sns.boxplot(x='type', y='glu', data=data, ax=axes[2, 0])
axes[2, 0].set_title('Boxplot: Type vs Glucose Level')

# Boxplot for 'type' vs 'ped'
sns.boxplot(x='type', y='ped', data=data, ax=axes[2, 1])
axes[2, 1].set_title('Boxplot: Type vs ped')

# For example, if using matplotlib
plt.savefig('my_plot.png', bbox_inches='tight')
```



From the multiple boxplots presented above, a notable observation emerges: individuals with diabetes exhibit significantly higher mean values for BMI, Glucose, Pedigree Function (ped), and age compared to those without diabetes. Specifically, the boxplots reveal a clear distinction in these variables between the two groups. The higher mean values in BMI, Glucose, ped, and age for individuals with diabetes suggest potential associations or characteristics that differentiate the diabetic and non-diabetic populations in the dataset.

[54]:
```
sns.stripplot(x="npreg",y="age",data=data,hue="type")
plt.title('Age: Npreg vs Type')
plt.show()
```

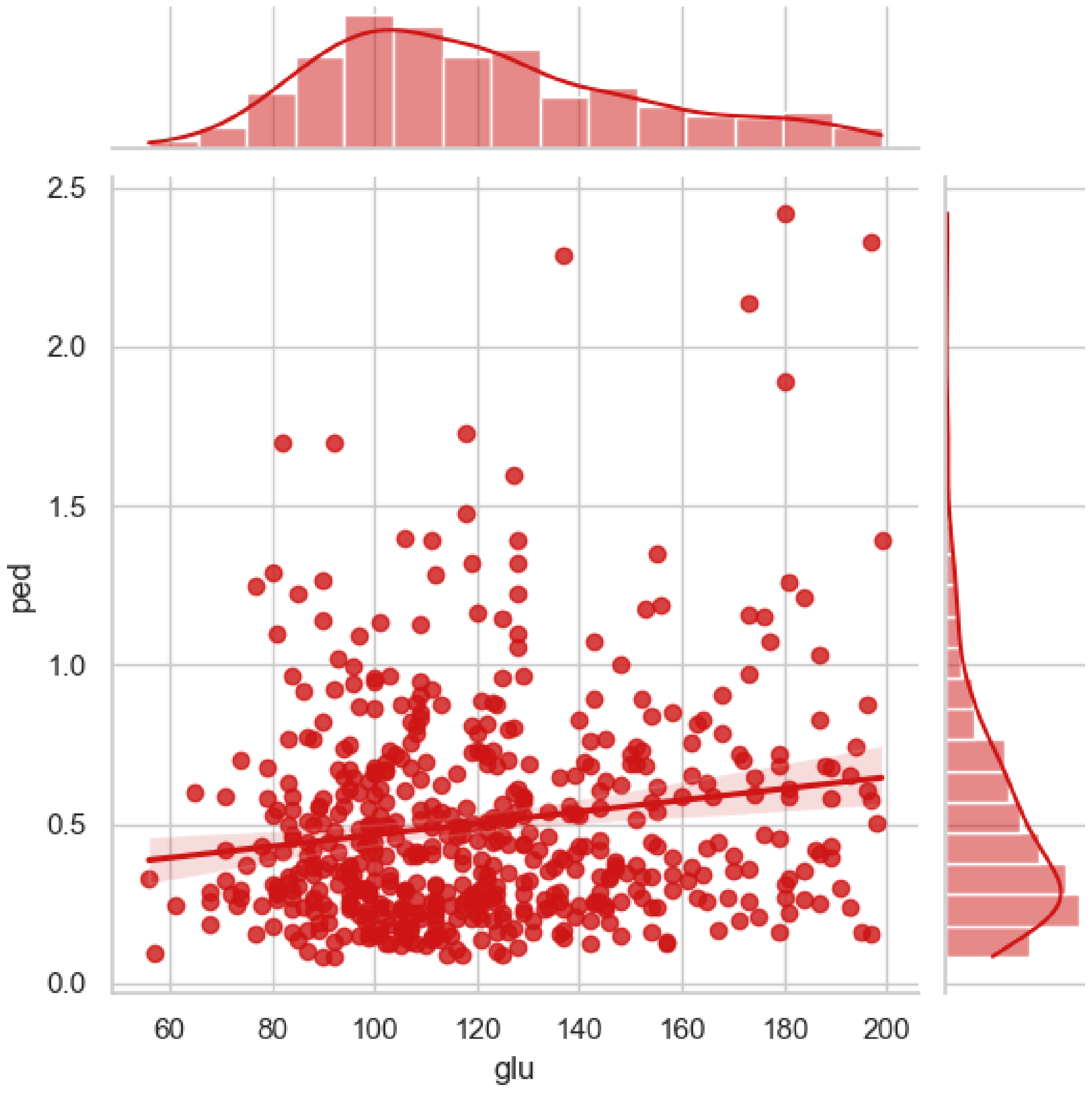


The likelihood of diabetes appears to be higher in cases of a greater number of pregnancies than in instances with fewer pregnancies. This observation implies a potential correlation between the number of pregnancies and the risk of diabetes among women. Further analysis and statistical testing would be needed to establish a conclusive relationship and determine potential contributing factors.

15 2. Analysis of ped with glucose

[55]:
```
x = data[['ped','glu']]
sns.jointplot(x=x.loc[:,'glu'], y=x.loc[:,'ped'],kind="reg", color="#ce1414")
```

[55]:
```
<seaborn.axisgrid.JointGrid at 0x19f20f38c70>
```



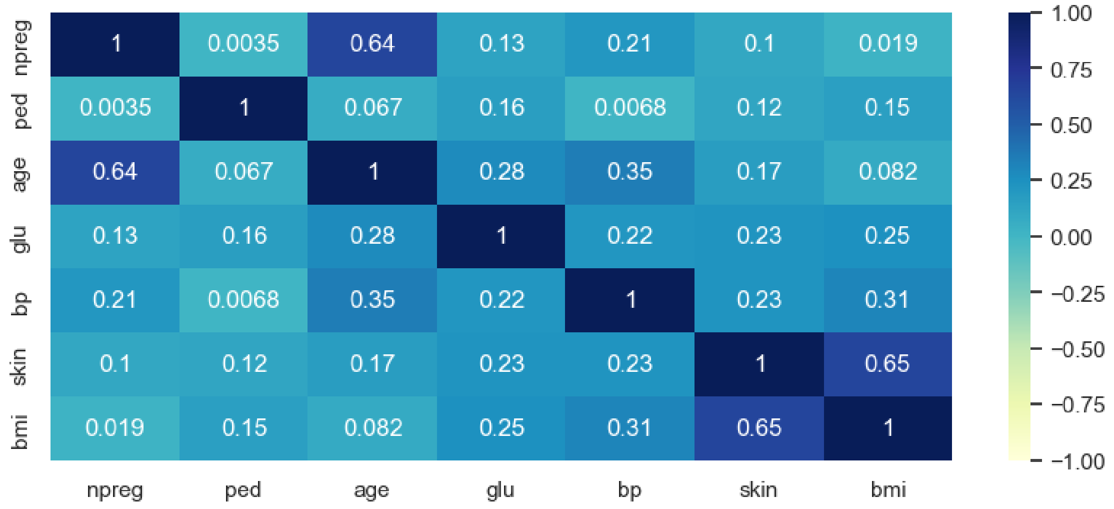
[56]:
```
data_x.corr()
```

[56]:
```
          npreg       ped       age       glu        bp      skin       bmi
npreg  1.000000  0.003539  0.644686  0.127319  0.205066  0.100239  0.019495
ped    0.003539  1.000000  0.066834  0.160832  0.006823  0.115016  0.150885
age    0.644686  0.066834  1.000000  0.279718  0.347147  0.166816  0.081580
glu    0.127319  0.160832  0.279718  1.000000  0.217581  0.227369  0.247116
bp     0.205066  0.006823  0.347147  0.217581  1.000000  0.226723  0.309508
skin   0.100239  0.115016  0.166816  0.227369  0.226723  1.000000  0.647828
bmi    0.019495  0.150885  0.081580  0.247116  0.309508  0.647828  1.000000
```

[57]:
```
corr=data_x.corr()
```

```
sns.heatmap(corr, cmap = 'YlGnBu', vmax = 1.0, vmin = -1.0, annot = True,␣
↪annot_kws = {"size": 12})
```

[57]:
```
<Axes: >
```



Age and number of pregancy: There is a positive correlation between Age and number of pregancy . This means that as those who are having higher age more number of children. BMI and Skin thickness: There is a positive correlation between BMI and Skin thickness. This means that as those who are having more BMI with higher skin thickness.

Remaining Variables: For the other variables in our dataset, there are weak or no significant correlations between them. This implies that changes in one of these variables do not strongly influence or predict changes in the others. They are relatively independent of each other in terms of their relationships within the dataset.

Understanding these correlations is important in various fields, such as medicine or statistics, as it helps us identify potential relationships between variables. In this case, the strong positive correlation between diastolic and systolic blood pressure might indicate that they are both influenced by similar factors or share a common physiological basis. Conversely, the weak or non-existent correlations among the remaining variables suggest that they are relatively unrelated in the context of this dataset. Please correct the sentence

[58]:
```
data.shape
```

[58]:
```
(541, 8)
```

[59]:
```
data_numeric = data.select_dtypes(include=np.number)
print(data_numeric.columns)
data_categoric = data.select_dtypes(include = object)
print(data_categoric.columns)
```

```
Index(['npreg', 'ped', 'age', 'type', 'glu', 'bp', 'skin', 'bmi'],
dtype='object')
Index([], dtype='object')
```

[60]:
```
#dummy_variables = pd.get_dummies(data_categoric, drop_first = True)
```

[61]:
```
#data_dummy = pd.concat([data_numeric, dummy_variables], axis=1)
data_dummy=data
```

[62]:
```
data_dummy.head(1)
```

[62]:
```
   npreg    ped  age  type    glu    bp  skin   bmi
0      6  0.627   50     1  148.0  72.0  35.0  33.6
```

[63]:
```
X = data_dummy.drop(['type'], axis = 1)
X=sm.add_constant(X)
y = pd.DataFrame(data_dummy['type'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 1)
```

[64]:
```
def get_test_report(model):
    return(classification_report(y_test,y_pred))
```

[65]:
```
def kappa_score(model):
    return(cohen_kappa_score(y_test,y_pred))
```

[66]:
```
def plot_confusion_matrix(model):
    cm = confusion_matrix(y_test, y_pred)
    conf_matrix = pd.DataFrame(data = cm,columns = ['Predicted:0','Predicted:
 ↪1'], index = ['Actual:0','Actual:1'])
    sns.heatmap(conf_matrix, annot = True, fmt = 'd', cmap =␣
 ↪ListedColormap(['lightskyblue']),cbar = False, linewidths = 0.1, annot_kws =␣
 ↪{'size':25})
    plt.xticks(fontsize = 20)
    plt.yticks(fontsize = 20)
    plt.show()
```

[67]:
```
def plot_roc(model):
    fpr,tpr,_=roc_curve(y_test,y_pred_prob)
    plt.plot(fpr,tpr)
    plt.xlim([0.0,1.0])
    plt.ylim([0.0,1.0])
    plt.plot([0,1],[0,1],"r--")
    plt.title("ROC Curve",fontsize=15)
    plt.xlabel("False positive",fontsize=15)
    plt.ylabel("True positive",fontsize=15)
    plt.text(x=0.02,y=0.9,s=("AUC Score:
↪",round(roc_auc_score(y_test,y_pred_prob),4)))
    plt.grid(True)
```

[68]:
```
score_card=pd.DataFrame(columns=["Model","AUC Score","Precision Score","Recall␣
↪Score","Accuracy Score","Kappa Score","f1-Score"])
def update_score_card(model_name):
    global score_card
    score_card=score_card.append({"Model":model_name,"AUC Score":
↪roc_auc_score(y_test,y_pred_prob),"Precision Score":metrics.
↪precision_score(y_test,y_pred),"Recall Score":metrics.
↪accuracy_score(y_test,y_pred),'Accuracy Score': metrics.
↪accuracy_score(y_test, y_pred),"Kappa Score":
↪cohen_kappa_score(y_test,y_pred),"f1-Score":metrics.
↪f1_score(y_test,y_pred)},ignore_index=True)
    return(score_card)
```

After completing data cleaning and certain exploratory data analysis (EDA) steps, we partitioned the data into two sets: a training set comprising 80% of the observations and a test set with 20% of the observations to assess the model's accuracy.

In this phase, we applied various machine learning models, namely Logistic Regression, Decision Tree, Naive Bayes, and Support Vector Machine. Subsequently, we compared the accuracy of these different models, selecting the best-performing ones for deployment.

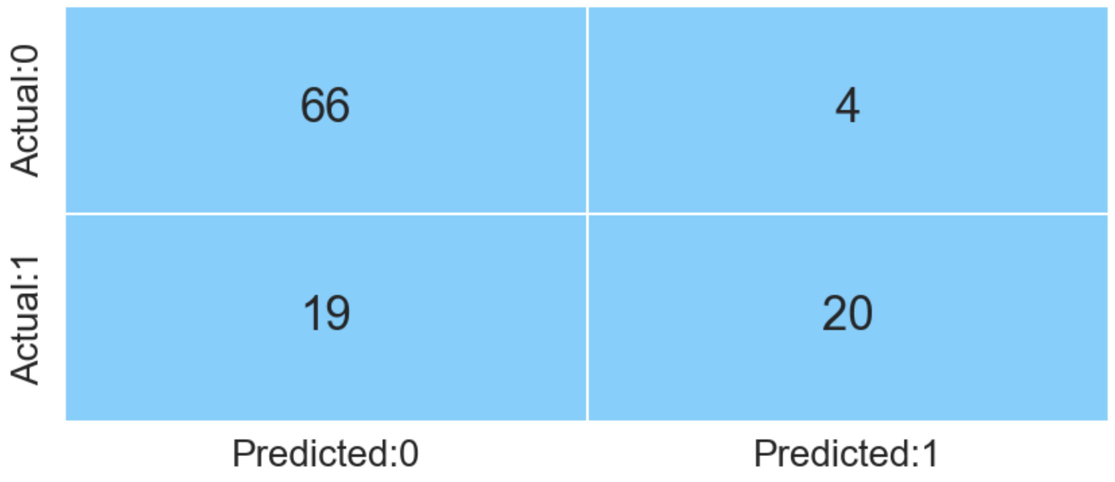
[69]:
```
Log_Reg_Full_Model=sm.Logit(y_train,X_train).fit()
print(Log_Reg_Full_Model.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.436427
         Iterations 7
                           Logit Regression Results
==============================================================================
Dep. Variable:                   type   No. Observations:                  432
Model:                          Logit   Df Residuals:                      424
Method:                           MLE   Df Model:                            7
Date:                Sun, 28 Jan 2024   Pseudo R-squ.:                  0.3090
Time:                        16:32:22   Log-Likelihood:                -188.54
converged:                       True   LL-Null:                       -272.85
Covariance Type:            nonrobust   LLR p-value:                 4.899e-33
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -9.7352      1.123     -8.667      0.000     -11.937      -7.534
npreg          0.1335      0.047      2.824      0.005       0.041       0.226
ped            1.3995      0.401      3.491      0.000       0.614       2.185
age            0.0261      0.015      1.692      0.091      -0.004       0.056
glu            0.0369      0.005      7.616      0.000       0.027       0.046
```

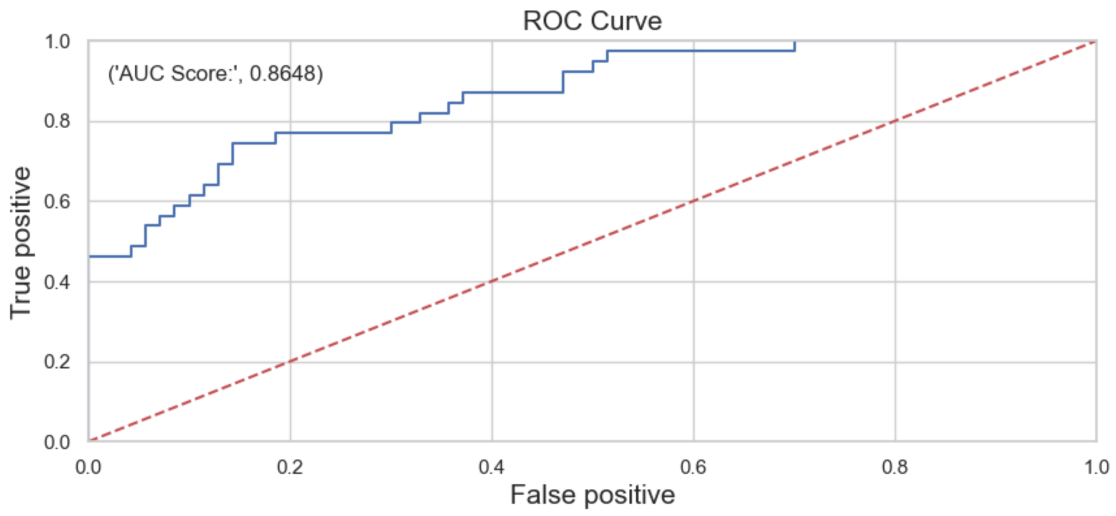
```
bp              -0.0126      0.012     -1.088      0.277      -0.035       0.010
skin             0.0034      0.017      0.204      0.838      -0.030       0.036
bmi              0.0905      0.026      3.491      0.000       0.040       0.141
==============================================================================
```

Except for BP and skin thickness, all variables are deemed statistically significant, as their p-values exceed 0.05 for intercept, npreg, ped, age, glu, and bmi. This implies significance at the 5% level. However, the inclusion of non-significant variables raises concerns about potential overfitting. To mitigate this, we intend to select only significant variables using forward or backward elimination in feature selection methods and subsequently assess the model's accuracy.

[70]:
```
y_pred_prob=Log_Reg_Full_Model.predict(X_test)
y_pred=["0" if x<0.5 else "1" for x in y_pred_prob]
y_pred=np.array(y_pred,dtype=np.float32)
y_pred[0:5]
plot_confusion_matrix(Log_Reg_Full_Model)
plot_roc(Log_Reg_Full_Model)
update_score_card(model_name="Logistic_Regression with Full Model")
```



[70]:
```
                                 Model  AUC Score  Precision Score  \
0  Logistic_Regression with Full Model   0.864835         0.833333

   Recall Score  Accuracy Score  Kappa Score  f1-Score
0      0.788991        0.788991     0.498098  0.634921
```



The confusion matrix indicates a 17.43% false negative and a 3.6% false positive, resulting in a total accuracy of 78.89%.

the AUC score of 0.8648 indicates that there is a high probability that the model will assign a higher predicted probability to a randomly chosen positive instance compared to a randomly chosen negative instance. The closer the AUC score is to 1, the better the model's ability to distinguish between positive and negative instances.

```
[71]: #SGDC Classifier with constant(intercept term alpha)
      SGD = SGDClassifier(loss = 'log', random_state = 10)
      Log_Reg_with_SGD = SGD.fit(X_train, y_train)
```

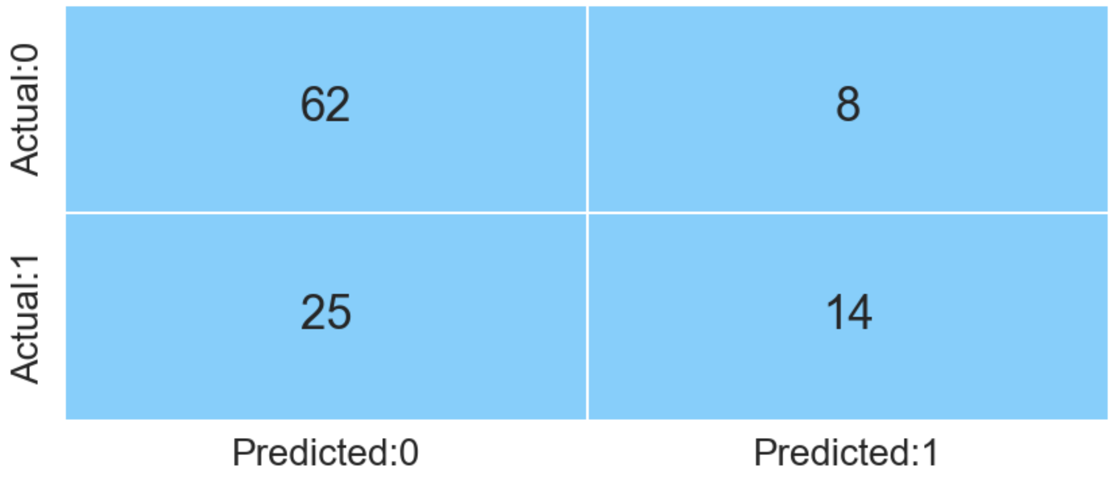
```
[72]: y_pred_prob =Log_Reg_with_SGD.predict_proba(X_test)[:,1]
      y_pred_prob
```

```
[72]: array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 1., 0., 1., 0., 0.,
             0., 0., 1., 0., 0., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
             0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1.,
             0., 1., 0., 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., 1., 0., 0., 0., 0.,
             0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 1.,
             0., 0., 0., 1., 0., 0., 0.])
```

```
[73]: y_pred =Log_Reg_with_SGD.predict(X_test)
      y_pred
```

```
[73]: array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0,
             1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
             0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 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, 1, 0, 0, 0, 0, 0, 0, 0,
         1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0],
        dtype=int64)
```

[74]:
```python
plot_confusion_matrix(Log_Reg_with_SGD)
```



The confusion matrix reveals a 22.93% false negative rate and a 7.3% false positive rate, leading to an overall accuracy of 69.72%. This accuracy is comparatively lower than that of the previous model.

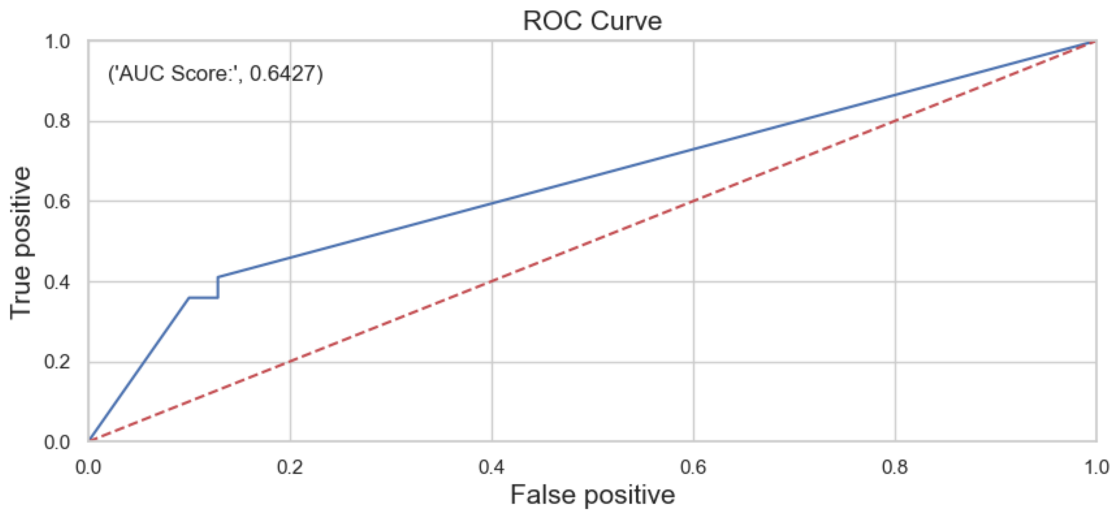
[75]:
```python
test_report = get_test_report(Log_Reg_with_SGD)
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.71      0.89      0.79        70
           1       0.64      0.36      0.46        39

    accuracy                           0.70       109
   macro avg       0.67      0.62      0.62       109
weighted avg       0.69      0.70      0.67       109
```

[76]:
```python
kappa_value = kappa_score(Log_Reg_with_SGD)
print(kappa_value)
```

```
0.2708291100750051
```

[77]:
```python
plot_roc(Log_Reg_with_SGD)
```



An Area Under the Curve (AUC) score of 0.6427 on the Receiver Operating Characteristic (ROC) curve suggests a moderate discriminatory performance of the model. The ROC curve illustrates the trade-off between the true positive rate (sensitivity) and the false positive rate (1-specificity) across various threshold values.

```
[78]: update_score_card(model_name = 'Logistic Regression (SGD)')
```

```
[78]:                                    Model  AUC Score  Precision Score  \
      0  Logistic_Regression with Full Model   0.864835         0.833333
      1            Logistic Regression (SGD)   0.642674         0.636364

         Recall Score  Accuracy Score  Kappa Score  f1-Score
      0      0.788991        0.788991     0.498098  0.634921
      1      0.697248        0.697248     0.270829  0.459016
```

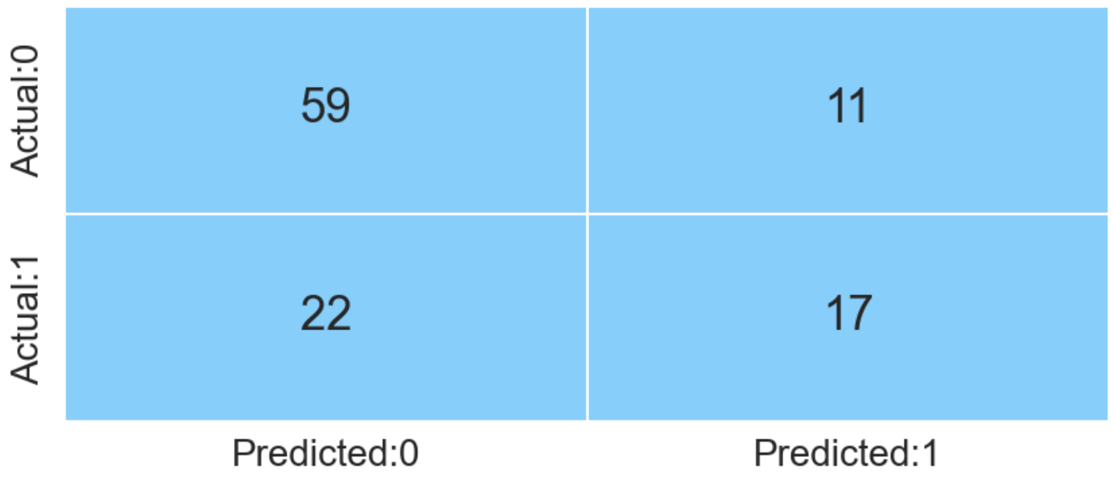
```
[79]: X = data_dummy.drop(['type'], axis = 1)
      y = pd.DataFrame(data_dummy['type'])
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
       ↪random_state = 1)
```

```
[80]: Log_Reg_Without_Intercept=sm.Logit(y_train,X_train).fit()
      print(Log_Reg_Without_Intercept.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.560074
         Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:                   type   No. Observations:                  432
Model:                          Logit   Df Residuals:                      425
```

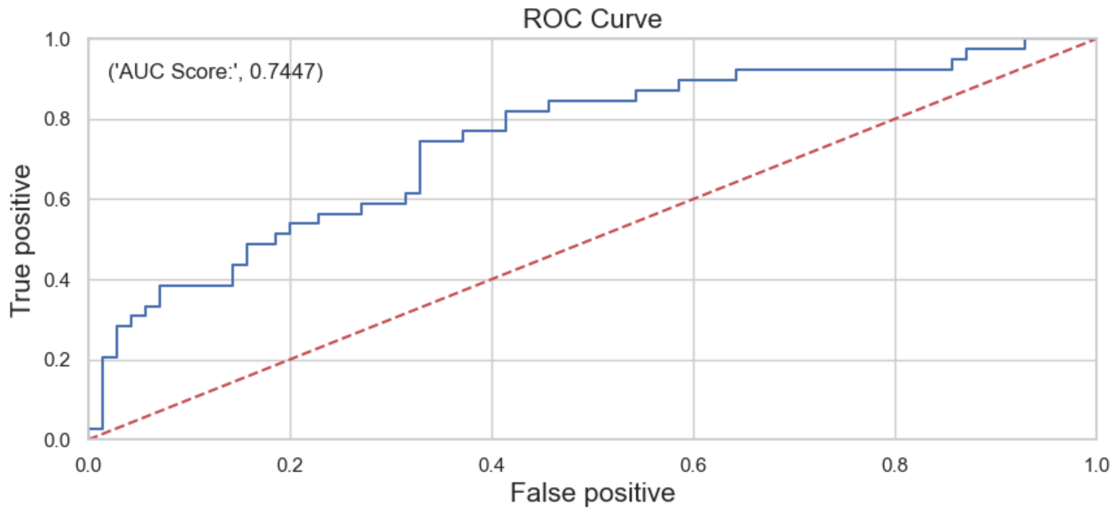
```
Method:                           MLE   Df Model:                            6
Date:                Sun, 28 Jan 2024   Pseudo R-squ.:                  0.1132
Time:                        16:32:28   Log-Likelihood:                -241.95
converged:                       True   LL-Null:                       -272.85
Covariance Type:            nonrobust   LLR p-value:                 1.946e-11
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
npreg          0.1354      0.043      3.168      0.002       0.052       0.219
ped            0.6483      0.341      1.899      0.058      -0.021       1.317
age            0.0054      0.014      0.378      0.705      -0.023       0.033
glu            0.0206      0.004      5.159      0.000       0.013       0.028
bp            -0.0593      0.010     -6.229      0.000      -0.078      -0.041
skin           0.0152      0.015      1.004      0.315      -0.014       0.045
bmi           -0.0114      0.022     -0.528      0.598      -0.054       0.031
==============================================================================
```

[81]:
```python
y_pred_prob=Log_Reg_Without_Intercept.predict(X_test)
y_pred=["0" if x<0.5 else "1" for x in y_pred_prob]
y_pred=np.array(y_pred,dtype=np.float32)
y_pred[0:5]
plot_confusion_matrix(Log_Reg_Without_Intercept)
plot_roc(Log_Reg_Without_Intercept)
update_score_card(model_name="Log_Reg_Without_Intercept")
```



[81]:
```
                                 Model  AUC Score  Precision Score  \
0  Logistic_Regression with Full Model   0.864835         0.833333
1             Logistic Regression (SGD)   0.642674         0.636364
2             Log_Reg_Without_Intercept   0.744689         0.607143

   Recall Score  Accuracy Score  Kappa Score  f1-Score
```

```
0       0.788991          0.788991      0.498098   0.634921
1       0.697248          0.697248      0.270829   0.459016
2       0.697248          0.697248      0.297324   0.507463
```



The confusion matrix reveals a 20.18% false negative rate and a 10.09% false positive rate, resulting in an overall accuracy of 69.72%. This accuracy is comparatively lower than that of the first model, suggesting a potential need to consider adding the intercept term. An AUC score of 0.74 implies a fair discriminatory performance, and further analysis or adjustments may be considered to potentially enhance the model's effectiveness.

16 Backward Model Selection Using Univariate Statistical Testing

[82]:
```
import statsmodels.api as sm
import pandas as pd

# Assume 'df' is your DataFrame with the target variable ('y') and predictor␣
 ↪variables

# Backward elimination function
def backward_elimination(data, target):
    features = list(data.columns)
    features.remove(target)

    while len(features) > 0:
        model = sm.Logit(data[target], sm.add_constant(data[features]))
        result = model.fit(disp=False)
        max_pvalue = result.pvalues.idxmax()
        # If the highest p-value is greater than a threshold (e.g., 0.05),␣
↪remove the corresponding feature
        if result.pvalues[max_pvalue] > 0.05:
            features.remove(max_pvalue)
        else:
            break  # If all p-values are below the threshold, stop

    return features

# Example usage
target_variable = 'type'
selected_features_backward = backward_elimination(data_dummy, target_variable)

print("Selected Features (Backward):", selected_features_backward)
```

```
Selected Features (Backward): ['npreg', 'ped', 'glu', 'bmi']
```

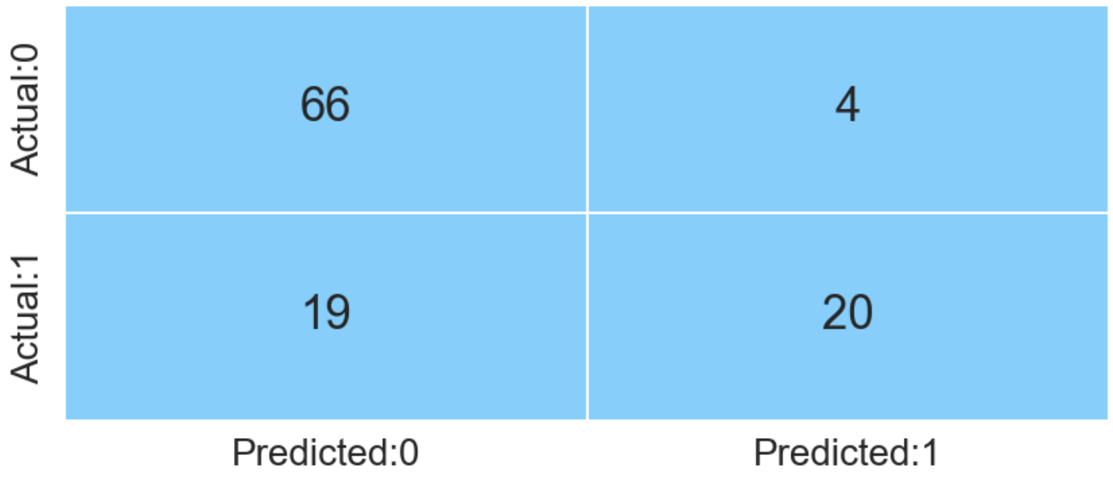
[83]:
```
X = data_dummy.drop(['type',"skin","age","bp"], axis = 1)
X=sm.add_constant(X)
y = pd.DataFrame(data_dummy['type'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 1)
```

[84]:
```
Log_Reg_Backward_Model_Selection=sm.Logit(y_train,X_train).fit()
print(Log_Reg_Backward_Model_Selection.summary())
```

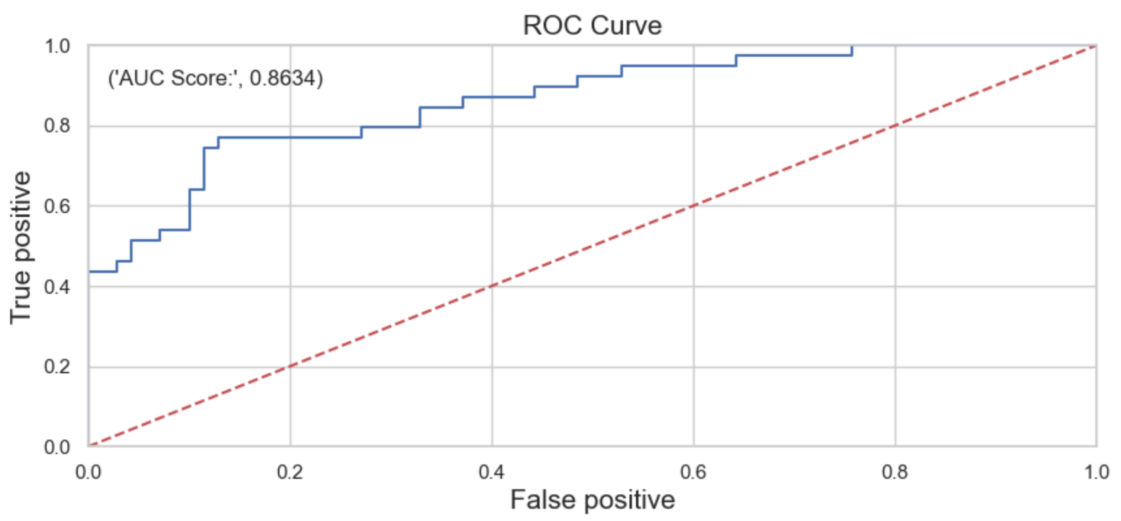
```
Optimization terminated successfully.
         Current function value: 0.440269
         Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:                   type   No. Observations:                  432
Model:                          Logit   Df Residuals:                      427
Method:                           MLE   Df Model:                            4
Date:                Sun, 28 Jan 2024   Pseudo R-squ.:                  0.3029
Time:                        16:32:34   Log-Likelihood:                -190.20
converged:                       True   LL-Null:                       -272.85
Covariance Type:            nonrobust   LLR p-value:                 1.065e-34
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -9.8095      1.004     -9.766      0.000     -11.778      -7.841
npreg          0.1795      0.037      4.836      0.000       0.107       0.252
ped            1.4627      0.395      3.705      0.000       0.689       2.236
glu            0.0377      0.005      8.004      0.000       0.028       0.047
bmi            0.0856      0.020      4.293      0.000       0.047       0.125
==============================================================================
```

In the model summary table, all variables are found to be significant at any given level of significance. Models of this type are less prone to overfitting. If the model performs well under the performance metrics, we can consider finalizing it for real-world predictions of diabetes

[85]:
```
y_pred_prob=Log_Reg_Backward_Model_Selection.predict(X_test)
y_pred=["0" if x<0.5 else "1" for x in y_pred_prob]
y_pred=np.array(y_pred,dtype=np.float32)
y_pred[0:5]
plot_confusion_matrix(Log_Reg_Backward_Model_Selection)
plot_roc(Log_Reg_Backward_Model_Selection)
update_score_card(model_name="Log_Reg_Backward_Model_Selection")
```



[85]:
```
                               Model  AUC Score  Precision Score  \
0  Logistic_Regression with Full Model   0.864835         0.833333
1            Logistic Regression (SGD)   0.642674         0.636364
2            Log_Reg_Without_Intercept   0.744689         0.607143
3     Log_Reg_Backward_Model_Selection   0.863370         0.833333

   Recall Score  Accuracy Score  Kappa Score  f1-Score
0      0.788991        0.788991     0.498098  0.634921
1      0.697248        0.697248     0.270829  0.459016
2      0.697248        0.697248     0.297324  0.507463
3      0.788991        0.788991     0.498098  0.634921
```



The confusion matrix reveals a 17.43% false negative rate and a 3.6% false positive rate, leading to an overall accuracy of 78.89%. The performance of this model is comparable to that of the first model. An AUC score of 0.8648 suggests that the model has a strong ability to discriminate between classes, making it a promising indicator of its overall performance.

[86]:
```
X = data_dummy.drop(['type',"skin","bp","npreg"], axis = 1)
X=sm.add_constant(X)
y = pd.DataFrame(data_dummy['type'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 1)
Log_Reg_With_Domain_Knowledge=sm.Logit(y_train,X_train).fit()
print(Log_Reg_With_Domain_Knowledge.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.448043
         Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:                   type   No. Observations:                  432
Model:                          Logit   Df Residuals:                      427
Method:                           MLE   Df Model:                            4
Date:                Sun, 28 Jan 2024   Pseudo R-squ.:                  0.2906
Time:                        16:32:37   Log-Likelihood:                -193.55
converged:                       True   LL-Null:                       -272.85
Covariance Type:            nonrobust   LLR p-value:                 2.939e-33
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -10.1884      1.033     -9.866      0.000     -12.212      -8.164
ped            1.2914      0.389      3.316      0.001       0.528       2.055
```

```
age              0.0483      0.012      4.198      0.000      0.026      0.071
glu              0.0348      0.005      7.421      0.000      0.026      0.044
bmi              0.0832      0.020      4.258      0.000      0.045      0.122
==============================================================================
```

[87]:
```
y_pred_prob=Log_Reg_With_Domain_Knowledge.predict(X_test)
y_pred=["0" if x<0.5 else "1" for x in y_pred_prob]
y_pred=np.array(y_pred,dtype=np.float32)
#plot_confusion_matrix(Log_Reg_With_Domain_Knowledge)
#plot_roc(Log_Reg_With_Domain_Knowledge)
#update_score_card(model_name="Log_Reg_With_Domain_Knowledge")
```

17 Decision Tree Classifiaction

[88]:
```
X = data_dummy.drop(['type'], axis = 1)
y = pd.DataFrame(data_dummy['type'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
 ↪random_state = 1)
```

[89]:
```
tuned_parameters=[{"criterion":["gini","entropy"],"min_samples_split":
 ↪[10,20,30],"max_depth":[3,5,7,9],"min_samples_leaf":
 ↪[15,20,25,30,35],"max_leaf_nodes":[5,10,15,20,25]}]
```

[90]:
```
decision_tree_classification=DecisionTreeClassifier(random_state=10)
grid=GridSearchCV(estimator=decision_tree_classification,param_grid=tuned_parameters,cv=10)
dt_grid=grid.fit(X_train,y_train)
print("Best parameters for DT:",dt_grid.best_params_,"\n")
```

```
Best parameters for DT: {'criterion': 'entropy', 'max_depth': 5,
'max_leaf_nodes': 10, 'min_samples_leaf': 15, 'min_samples_split': 10}
```

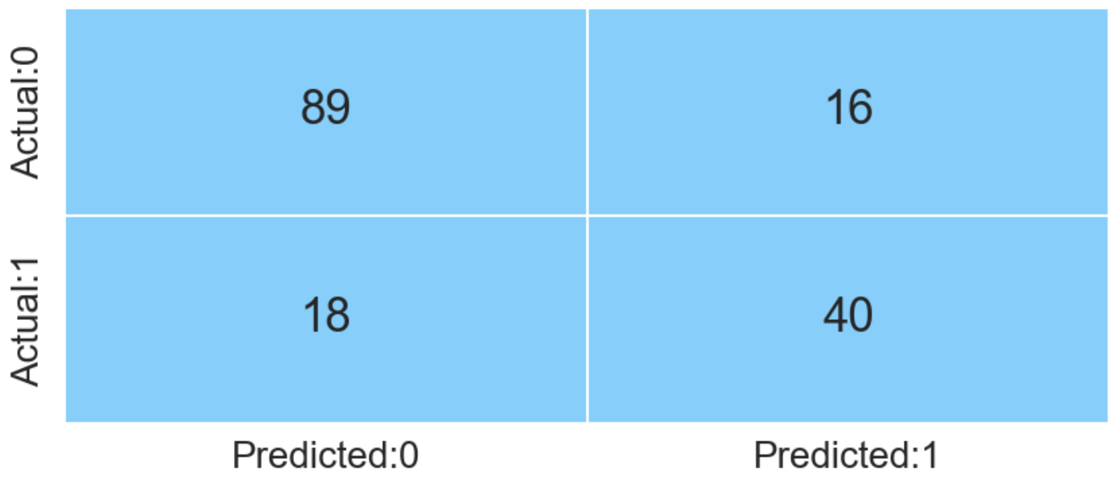
[91]:
```
dt_grid_model=DecisionTreeClassifier(criterion=dt_grid.best_params_.
 ↪get("criterion"),max_depth=dt_grid.best_params_.
 ↪get("max_depth"),max_leaf_nodes=dt_grid.best_params_.
 ↪get("max_leaf_nodes"),min_samples_leaf=dt_grid.best_params_.
 ↪get("min_samples_leaf"),min_samples_split=dt_grid.best_params_.
 ↪get("min_samples_split"))
```

[92]:
```
decision_tree_grid=dt_grid_model.fit(X_train,y_train)
```

[93]:
```
y_pred_prob=decision_tree_grid.predict_proba(X_test)[:,1]
```

[94]:
```
y_pred=decision_tree_grid.predict(X_test)
```

[95]:
```
plot_confusion_matrix(decision_tree_grid)
```



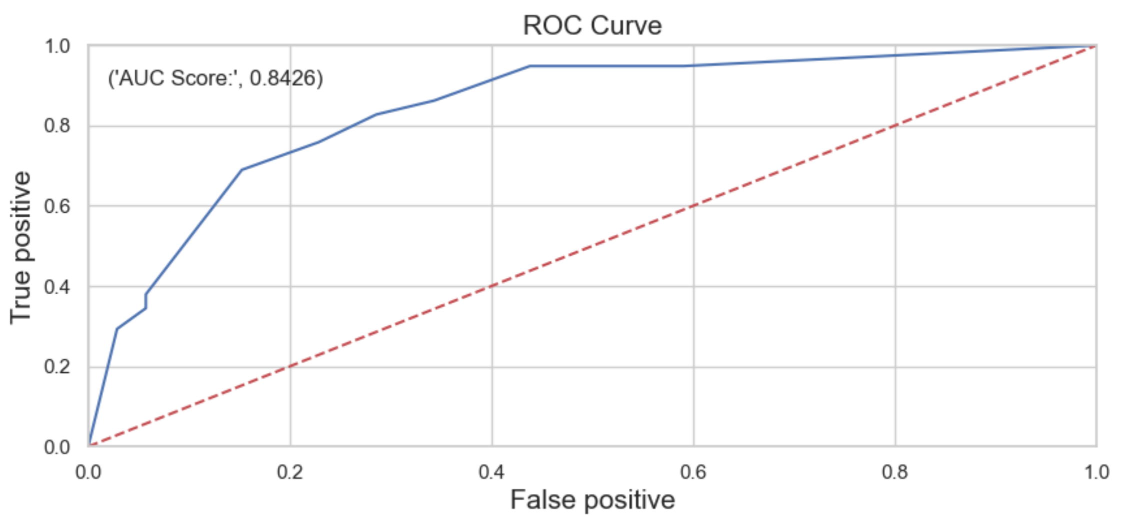
[96]:
```
test_report = get_test_report(decision_tree_grid)

# print the performace measures
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.83      0.85      0.84       105
           1       0.71      0.69      0.70        58

    accuracy                           0.79       163
   macro avg       0.77      0.77      0.77       163
weighted avg       0.79      0.79      0.79       163
```

[97]:
```
kappa_value = kappa_score(decision_tree_grid)

# print the kappa value
print(kappa_value)
```

```
0.5414529207347345
```

[98]:
```
plot_roc(decision_tree_grid)
```



[99]:
```
update_score_card(model_name = 'decision_tree_grid')
```

[99]:
```
                                Model  AUC Score  Precision Score  \
0  Logistic_Regression with Full Model   0.864835         0.833333
1             Logistic Regression (SGD)   0.642674         0.636364
2             Log_Reg_Without_Intercept   0.744689         0.607143
3         Log_Reg_Backward_Model_Selection 0.863370         0.833333
4                    decision_tree_grid   0.842611         0.714286

   Recall Score  Accuracy Score  Kappa Score  f1-Score
0      0.788991        0.788991     0.498098  0.634921
1      0.697248        0.697248     0.270829  0.459016
2      0.697248        0.697248     0.297324  0.507463
3      0.788991        0.788991     0.498098  0.634921
4      0.791411        0.791411     0.541453  0.701754
```

[100]:
```
X = data_dummy.drop(['type'], axis = 1)
y = pd.DataFrame(data_dummy['type'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
 ↪random_state = 1)
```

[101]:
```
from sklearn.naive_bayes import GaussianNB
```

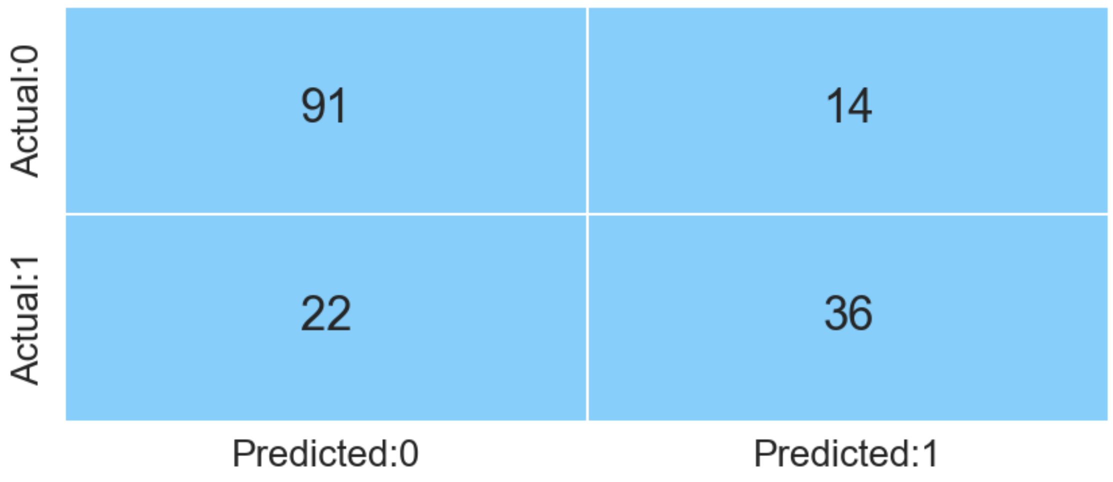
[102]:
```
Naive_Bayes_Model =GaussianNB().fit(X_train, y_train)
```

[103]:
```
y_pred_prob =Naive_Bayes_Model .predict_proba(X_test)[:,1]
```

[104]:
```
y_pred = Naive_Bayes_Model.predict(X_test)
y_pred[0:11]
```

[104]: array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1], dtype=int64)

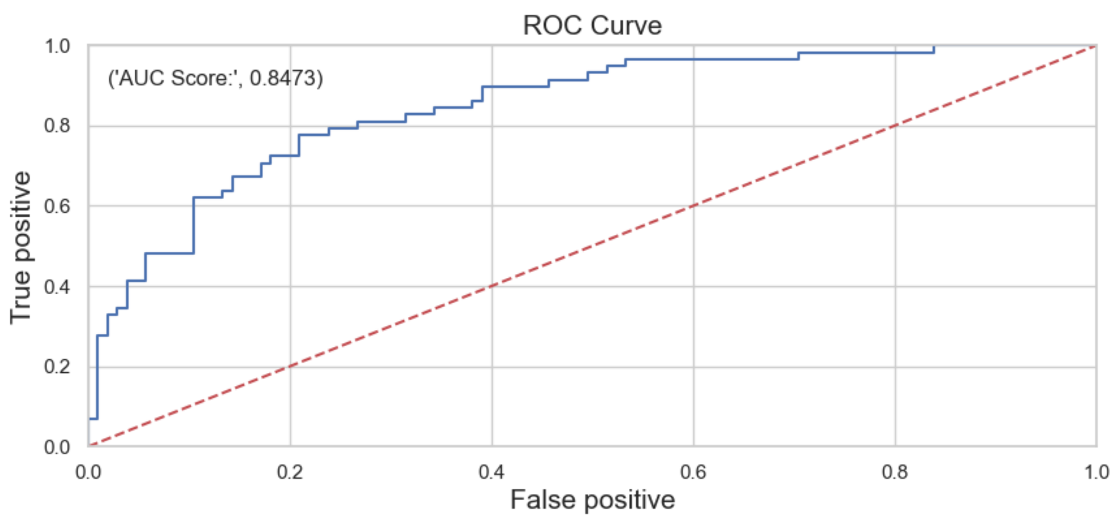
[105]:
```
plot_confusion_matrix(Naive_Bayes_Model)
```



[106]:
```
test_report = get_test_report(Naive_Bayes_Model)
print(test_report)
```

```
              precision    recall  f1-score   support

           0       0.81      0.87      0.83       105
           1       0.72      0.62      0.67        58

    accuracy                           0.78       163
   macro avg       0.76      0.74      0.75       163
weighted avg       0.77      0.78      0.78       163
```

[107]:
```
plot_roc(Naive_Bayes_Model)
```



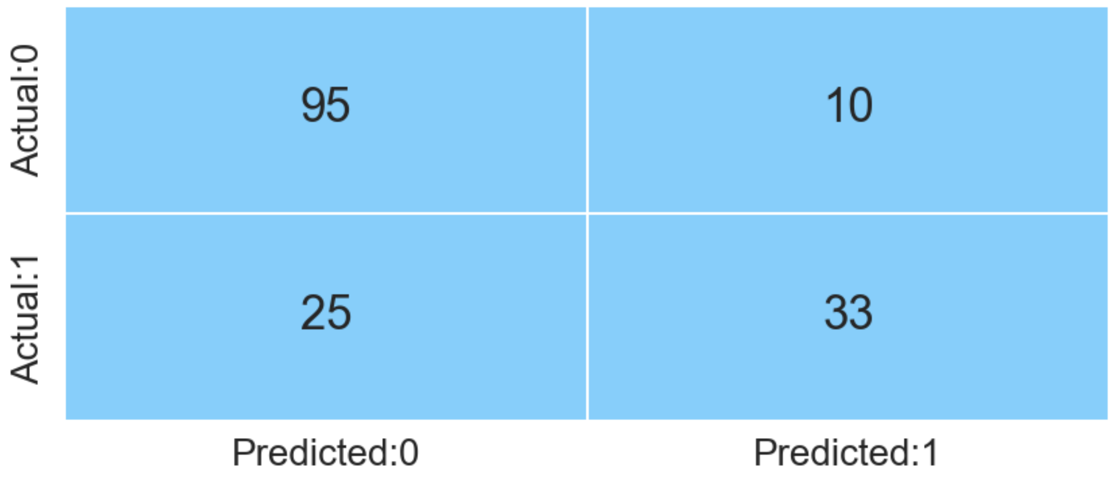
```
[108]: update_score_card(model_name = 'Naive_Bayes_Model')
```

```
[108]:                                  Model  AUC Score  Precision Score  \
       0  Logistic_Regression with Full Model   0.864835         0.833333
       1            Logistic Regression (SGD)   0.642674         0.636364
       2            Log_Reg_Without_Intercept   0.744689         0.607143
       3       Log_Reg_Backward_Model_Selection   0.863370         0.833333
       4                    decision_tree_grid   0.842611         0.714286
       5                     Naive_Bayes_Model   0.847291         0.720000

          Recall Score  Accuracy Score  Kappa Score  f1-Score
       0      0.788991        0.788991     0.498098  0.634921
       1      0.697248        0.697248     0.270829  0.459016
       2      0.697248        0.697248     0.297324  0.507463
       3      0.788991        0.788991     0.498098  0.634921
       4      0.791411        0.791411     0.541453  0.701754
       5      0.779141        0.779141     0.502880  0.666667
```

```
[109]: from sklearn.svm import SVC
```

```
[110]: svc_linear = SVC(kernel='linear', probability=True)  # Specify␣
         ↪'probability=True' to enable probability estimates
       svm_linear=svc_linear.fit(X_train, y_train)
       y_pred_prob =svm_linear.predict_proba(X_test)[:,1]
       y_pred =svm_linear .predict(X_test)
       plot_confusion_matrix(svm_linear)
       test_report = get_test_report(svm_linear)
       print(test_report)
       plot_roc(svm_linear)
```

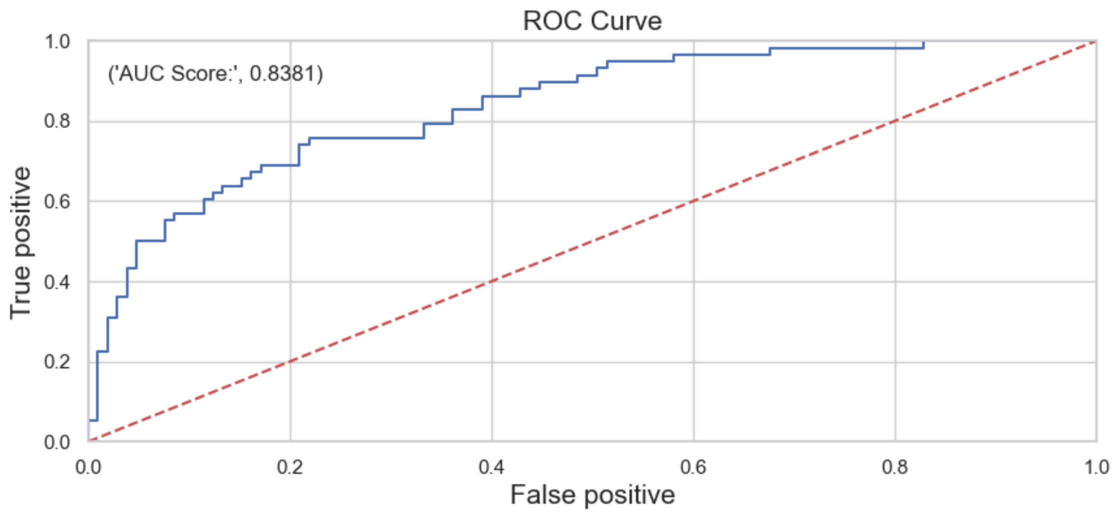
```
update_score_card(model_name = 'svm_linear')
```



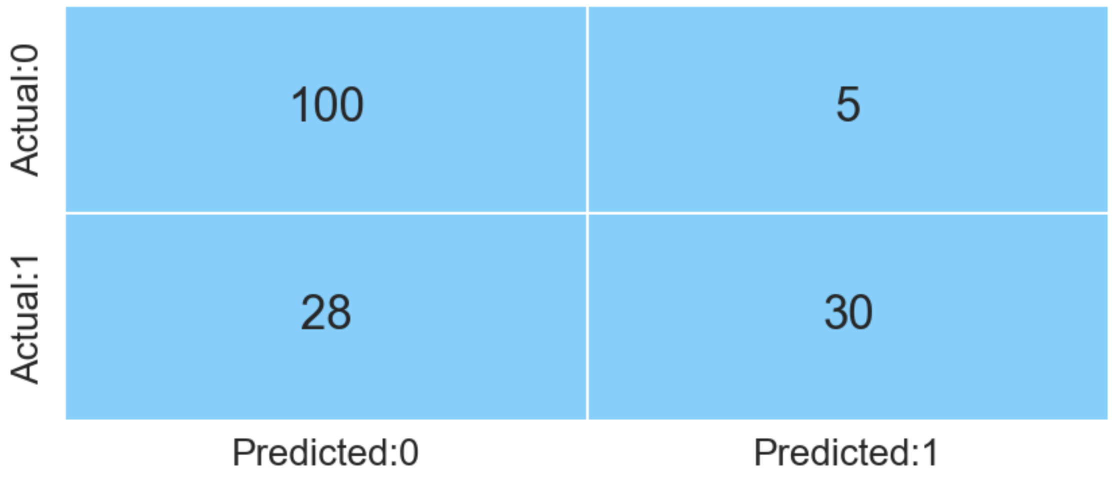
```
              precision    recall  f1-score   support

           0       0.79      0.90      0.84       105
           1       0.77      0.57      0.65        58

    accuracy                           0.79       163
   macro avg       0.78      0.74      0.75       163
weighted avg       0.78      0.79      0.78       163
```

[110]:

```
                                Model  AUC Score  Precision Score  \
0  Logistic_Regression with Full Model   0.864835         0.833333
1            Logistic Regression (SGD)   0.642674         0.636364
2            Log_Reg_Without_Intercept   0.744689         0.607143
3        Log_Reg_Backward_Model_Selection   0.863370         0.833333
4                   decision_tree_grid   0.842611         0.714286
5                    Naive_Bayes_Model   0.847291         0.720000
6                           svm_linear   0.838095         0.767442

   Recall Score  Accuracy Score  Kappa Score  f1-Score
0      0.788991        0.788991     0.498098  0.634921
1      0.697248        0.697248     0.270829  0.459016
2      0.697248        0.697248     0.297324  0.507463
3      0.788991        0.788991     0.498098  0.634921
4      0.791411        0.791411     0.541453  0.701754
5      0.779141        0.779141     0.502880  0.666667
6      0.785276        0.785276     0.502832  0.653465
```



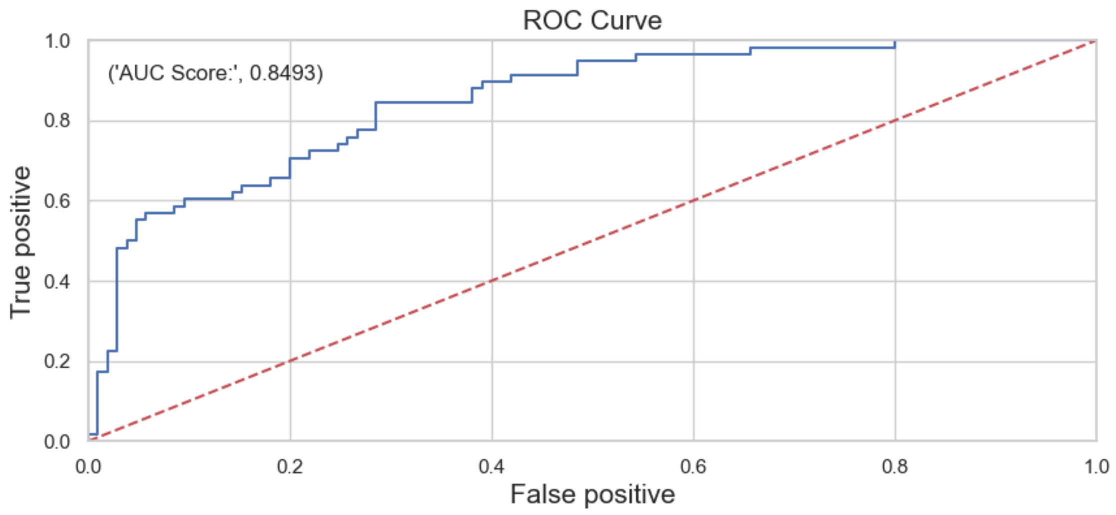
[111]:
```
svc_poly = SVC(kernel='poly', probability=True)  # Specify 'probability=True'␣
 ↪to enable probability estimates
svm_poly=svc_poly.fit(X_train, y_train)
y_pred_prob =svm_poly.predict_proba(X_test)[:,1]
y_pred =svm_poly .predict(X_test)
plot_confusion_matrix(svm_poly)
test_report = get_test_report(svm_poly)
print(test_report)
plot_roc(svm_poly)
update_score_card(model_name = 'svm_poly')
```



```
              precision    recall  f1-score   support
```

```
           0       0.78      0.95      0.86       105
           1       0.86      0.52      0.65        58

    accuracy                           0.80       163
   macro avg       0.82      0.73      0.75       163
weighted avg       0.81      0.80      0.78       163
```

[111]:
```
                                Model  AUC Score  Precision Score  \
0  Logistic_Regression with Full Model   0.864835         0.833333
1            Logistic Regression (SGD)   0.642674         0.636364
2            Log_Reg_Without_Intercept   0.744689         0.607143
3        Log_Reg_Backward_Model_Selection   0.863370         0.833333
4                   decision_tree_grid   0.842611         0.714286
5                    Naive_Bayes_Model   0.847291         0.720000
6                           svm_linear   0.838095         0.767442
7                             svm_poly   0.849261         0.857143

   Recall Score  Accuracy Score  Kappa Score  f1-Score
0      0.788991        0.788991     0.498098  0.634921
1      0.697248        0.697248     0.270829  0.459016
2      0.697248        0.697248     0.297324  0.507463
3      0.788991        0.788991     0.498098  0.634921
4      0.791411        0.791411     0.541453  0.701754
5      0.779141        0.779141     0.502880  0.666667
6      0.785276        0.785276     0.502832  0.653465
7      0.797546        0.797546     0.515362  0.645161
```



The table above compares the performance of seven models. Among these models, except for the Logistic model with SGD classifier, the remaining models exhibit nearly equal performance. Some models, such as decision tree, SVM, and naive Bayes, are complex (high variance) and may be prone to overfitting. When simple models (low variance) perform nearly as well as complex models, it is preferable to choose the simpler ones. We are selecting Logistic Regression with Backward Elimination method because all variables are significant, and it involves fewer features. To mitigate the risk of high variance, opting for simpler models is advisable, especially when the model does not exhibit signs of underfitting.

```
[112]: data_dummy.head(2)
```

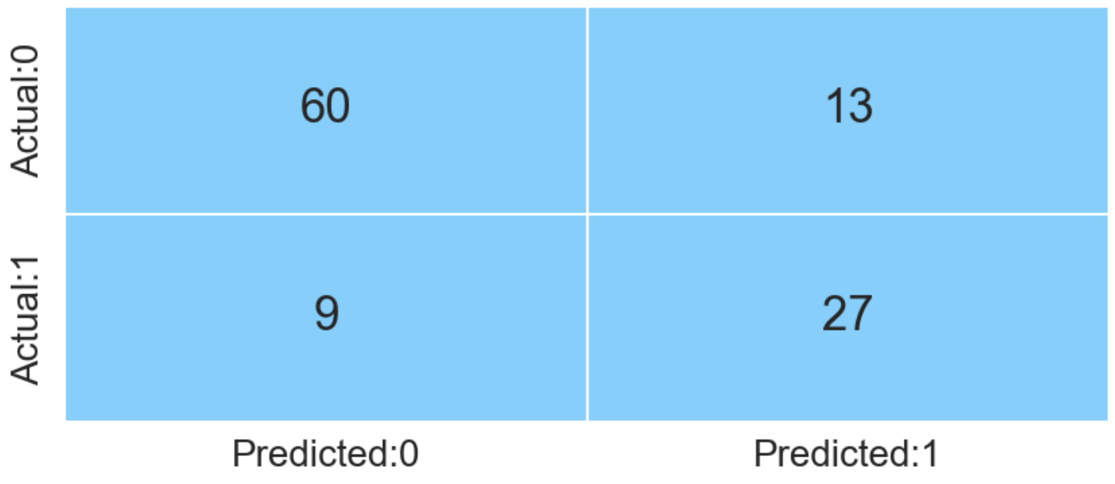
```
[112]:    npreg    ped  age  type    glu    bp  skin   bmi
       0      6  0.627   50     1  148.0  72.0  35.0  33.6
       1      1  0.351   31     0   85.0  66.0  29.0  26.6
```

```
[113]: # Drop the target variable 'type' from X
       X = data_dummy.drop(['type'], axis=1)

       # Standardize the features
       scale = StandardScaler().fit(X)
       features = scale.transform(X)
       features_scaled = pd.DataFrame(features, columns=["npreg", "ped", "age", "glu",␣
        ↪"bp", "skin", "bmi"])

       # Create the target variable 'y'
       y = pd.DataFrame(data_dummy['type'])

       # Add a constant term to the features
       X_scaled = sm.add_constant(features_scaled)

       # Split the data into training and testing sets
       X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,␣
        ↪random_state=10)


       # Ensure the indices are aligned
       X_train.reset_index(drop=True, inplace=True)
       y_train.reset_index(drop=True, inplace=True)
```

```
[114]: Log_Reg_Model_Std_Scalar=sm.Logit(y_train,X_train).fit()
       print(Log_Reg_Model_Std_Scalar.summary())
       y_pred_prob=Log_Reg_Model_Std_Scalar.predict(X_test)
       y_pred=["0" if x<0.5 else "1" for x in y_pred_prob]
       y_pred=np.array(y_pred,dtype=np.float32)
```

```
Optimization terminated successfully.
         Current function value: 0.429189
         Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:                   type   No. Observations:                  432
Model:                          Logit   Df Residuals:                      424
Method:                           MLE   Df Model:                            7
Date:                Sun, 28 Jan 2024   Pseudo R-squ.:                  0.3257
Time:                        16:34:04   Log-Likelihood:                -185.41
converged:                       True   LL-Null:                       -274.97
Covariance Type:            nonrobust   LLR p-value:                 2.975e-35
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.9423      0.137     -6.901      0.000      -1.210      -0.675
npreg          0.7030      0.175      4.024      0.000       0.361       1.045
ped            0.4401      0.138      3.182      0.001       0.169       0.711
age            0.1710      0.175      0.976      0.329      -0.172       0.514
glu            1.1143      0.153      7.286      0.000       0.815       1.414
bp            -0.1328      0.140     -0.948      0.343      -0.408       0.142
skin          -0.0001      0.176     -0.001      1.000      -0.345       0.345
bmi            0.6937      0.179      3.876      0.000       0.343       1.044
==============================================================================
```

[115]:
```
plot_confusion_matrix(Log_Reg_Model_Std_Scalar)
test_report = get_test_report(Log_Reg_Model_Std_Scalar)
print(test_report)
plot_roc(Log_Reg_Model_Std_Scalar)
update_score_card(model_name = 'Log_Reg_Model_Std_Scalar')
```

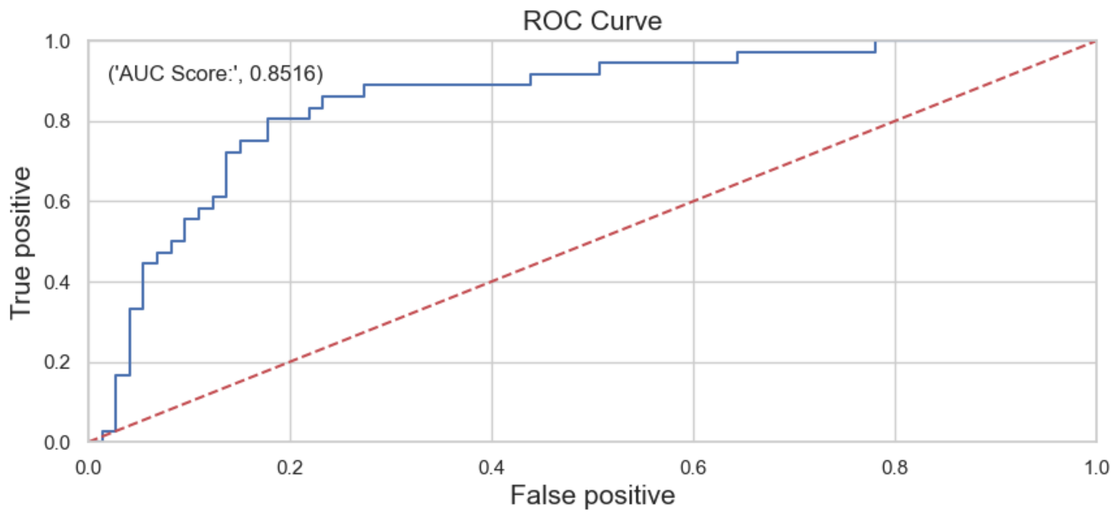


```
              precision    recall  f1-score   support

           0       0.87      0.82      0.85        73
           1       0.68      0.75      0.71        36
```

```
    accuracy                           0.80       109
   macro avg       0.77      0.79      0.78       109
weighted avg       0.81      0.80      0.80       109
```

[115]:

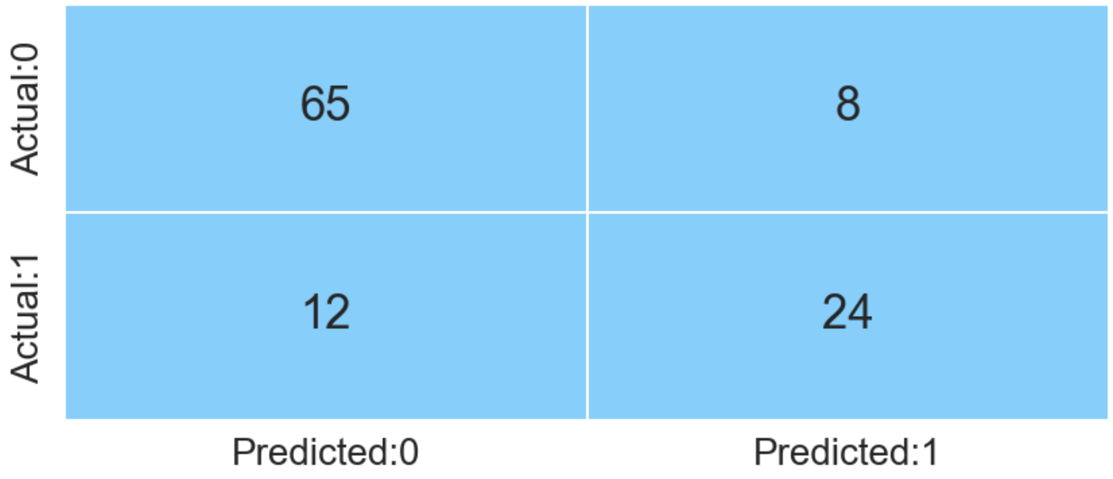
```
                              Model  AUC Score  Precision Score  \
0  Logistic_Regression with Full Model   0.864835         0.833333
1            Logistic Regression (SGD)   0.642674         0.636364
2            Log_Reg_Without_Intercept   0.744689         0.607143
3         Log_Reg_Backward_Model_Selection   0.863370         0.833333
4                   decision_tree_grid   0.842611         0.714286
5                    Naive_Bayes_Model   0.847291         0.720000
6                           svm_linear   0.838095         0.767442
7                             svm_poly   0.849261         0.857143
8               Log_Reg_Model_Std_Scalar   0.851598         0.675000

   Recall Score  Accuracy Score  Kappa Score  f1-Score
0      0.788991        0.788991     0.498098  0.634921
1      0.697248        0.697248     0.270829  0.459016
2      0.697248        0.697248     0.297324  0.507463
3      0.788991        0.788991     0.498098  0.634921
4      0.791411        0.791411     0.541453  0.701754
5      0.779141        0.779141     0.502880  0.666667
6      0.785276        0.785276     0.502832  0.653465
7      0.797546        0.797546     0.515362  0.645161
8      0.798165        0.798165     0.556255  0.710526
```



[116]:

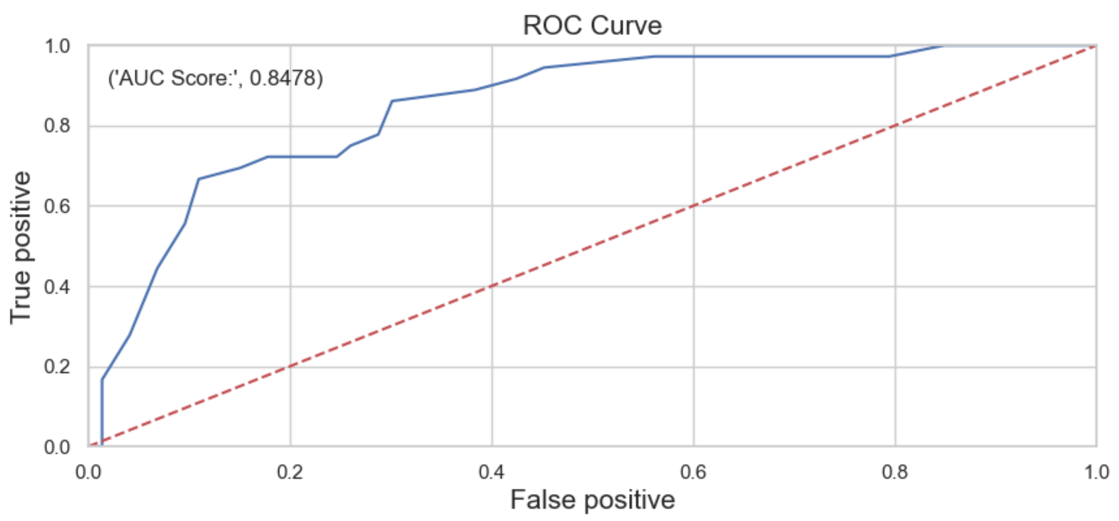
```
# Drop the target variable 'type' from X
X = data_dummy.drop(['type'], axis=1)
```

```
# Standardize the features
scale = StandardScaler().fit(X)
features = scale.transform(X)
features_scaled = pd.DataFrame(features, columns=["npreg", "ped", "age", "glu",␣
↪"bp", "skin", "bmi"])

# Create the target variable 'y'
y = pd.DataFrame(data_dummy['type'])

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features_scaled, y,␣
↪test_size=0.2, random_state=10)


# Ensure the indices are aligned
X_train.reset_index(drop=True, inplace=True)
y_train.reset_index(drop=True, inplace=True)

from sklearn.neighbors import KNeighborsClassifier

KN_Classifier =KNeighborsClassifier(n_neighbors=28,p=2,metric="euclidean")␣
↪#SVC(kernel='poly', probability=True)  # Specify 'probability=True' to␣
↪enable probability estimates
KN_Classifier_st=KN_Classifier.fit(X_train, y_train)
y_pred_prob =KN_Classifier_st.predict_proba(X_test)[:,1]
y_pred =KN_Classifier_st .predict(X_test)
plot_confusion_matrix(KN_Classifier_st)
test_report = get_test_report(KN_Classifier_st)
print(test_report)
plot_roc(KN_Classifier_st)
update_score_card(model_name = 'KN_Classifier_Standard_Scaler')
```



```
              precision    recall  f1-score   support

           0       0.84      0.89      0.87        73
           1       0.75      0.67      0.71        36

    accuracy                           0.82       109
   macro avg       0.80      0.78      0.79       109
weighted avg       0.81      0.82      0.81       109
```

[116]:
```
                                 Model  AUC Score  Precision Score  \
0  Logistic_Regression with Full Model   0.864835         0.833333
1             Logistic Regression (SGD)   0.642674         0.636364
2             Log_Reg_Without_Intercept   0.744689         0.607143
3      Log_Reg_Backward_Model_Selection   0.863370         0.833333
4                    decision_tree_grid   0.842611         0.714286
5                     Naive_Bayes_Model   0.847291         0.720000
6                            svm_linear   0.838095         0.767442
7                              svm_poly   0.849261         0.857143
8              Log_Reg_Model_Std_Scalar   0.851598         0.675000
9        KN_Classifier_Standard_Scaler   0.847793         0.750000

   Recall Score  Accuracy Score  Kappa Score  f1-Score
0      0.788991        0.788991     0.498098  0.634921
1      0.697248        0.697248     0.270829  0.459016
2      0.697248        0.697248     0.297324  0.507463
3      0.788991        0.788991     0.498098  0.634921
4      0.791411        0.791411     0.541453  0.701754
5      0.779141        0.779141     0.502880  0.666667
6      0.785276        0.785276     0.502832  0.653465
7      0.797546        0.797546     0.515362  0.645161
8      0.798165        0.798165     0.556255  0.710526
9      0.816514        0.816514     0.573218  0.705882
```



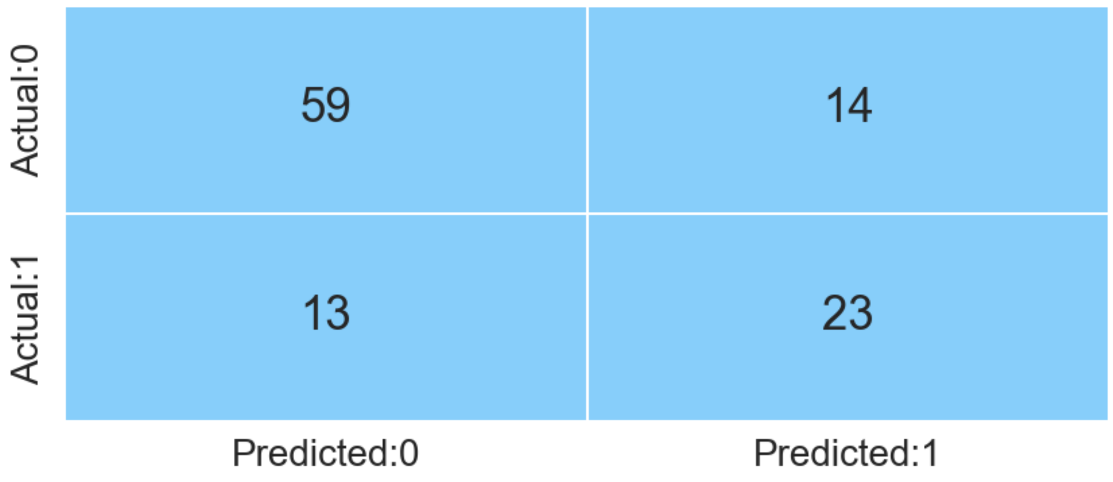
[117]:
```
from sklearn.ensemble import RandomForestClassifier
#intantiate the regressor
rf_cls = RandomForestClassifier(n_estimators=100, random_state=10)

# fit the regressor with training dataset
rf_cls.fit(X_train, y_train)
```

[117]:
```
RandomForestClassifier(random_state=10)
```

[118]:
```
# predict the values on test dataset using predict()
y_pred = rf_cls.predict(X_test)
y_pred
```

[118]:
```
array([0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1,
       1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0,
       0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0],
      dtype=int64)
```

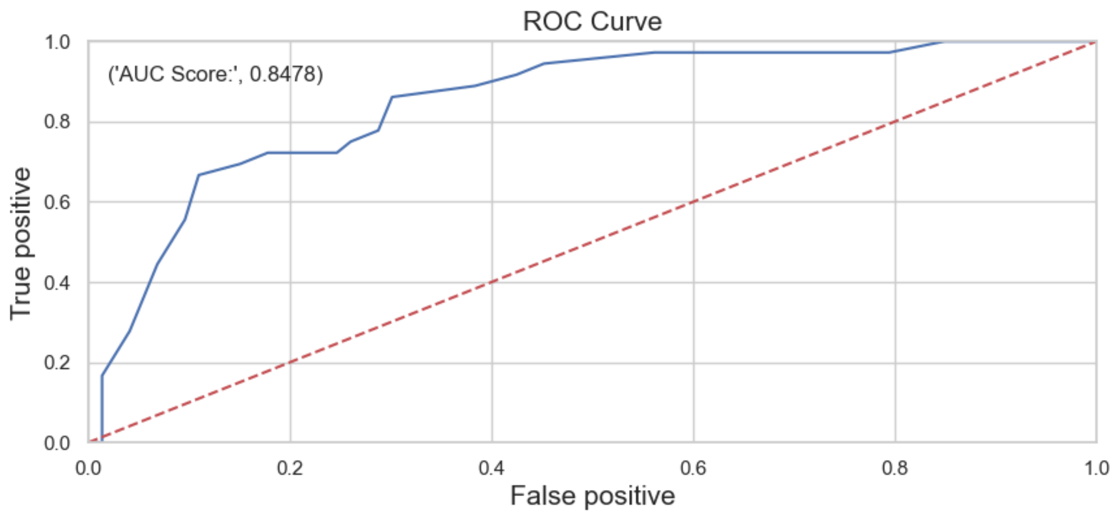
[119]:
```
plot_confusion_matrix(rf_cls)
test_report = get_test_report(rf_cls)
print(test_report)
plot_roc(rf_cls)
update_score_card(model_name = 'rf_cls')
```



```
              precision    recall  f1-score   support

           0       0.82      0.81      0.81        73
           1       0.62      0.64      0.63        36

    accuracy                           0.75       109
   macro avg       0.72      0.72      0.72       109
weighted avg       0.75      0.75      0.75       109
```

[119]:
```
                                Model  AUC Score  Precision Score  \
0   Logistic_Regression with Full Model   0.864835         0.833333
1              Logistic Regression (SGD)   0.642674         0.636364
2              Log_Reg_Without_Intercept   0.744689         0.607143
3           Log_Reg_Backward_Model_Selection   0.863370         0.833333
4                     decision_tree_grid   0.842611         0.714286
5                      Naive_Bayes_Model   0.847291         0.720000
6                             svm_linear   0.838095         0.767442
7                               svm_poly   0.849261         0.857143
8                Log_Reg_Model_Std_Scalar   0.851598         0.675000
9           KN_Classifier_Standard_Scaler   0.847793         0.750000
10                                rf_cls   0.847793         0.621622

    Recall Score  Accuracy Score  Kappa Score  f1-Score
0       0.788991        0.788991     0.498098  0.634921
1       0.697248        0.697248     0.270829  0.459016
2       0.697248        0.697248     0.297324  0.507463
3       0.788991        0.788991     0.498098  0.634921
4       0.791411        0.791411     0.541453  0.701754
5       0.779141        0.779141     0.502880  0.666667
```

```
6         0.785276          0.785276      0.502832  0.653465
7         0.797546          0.797546      0.515362  0.645161
8         0.798165          0.798165      0.556255  0.710526
9         0.816514          0.816514      0.573218  0.705882
10        0.752294          0.752294      0.443983  0.630137
```



```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

18 Multiple Regression Model Building for ped using PCA

```
[137]: from numpy.linalg import eig
       from sklearn.decomposition import PCA
```

```
[138]: data_1 = data_dummy.copy(deep = True)
       data_1.head()
```

```
[138]:    npreg    ped  age  type    glu    bp  skin   bmi
       0      6  0.627   50     1  148.0  72.0  35.0  33.6
       1      1  0.351   31     0   85.0  66.0  29.0  26.6
       3      1  0.167   21     0   89.0  66.0  23.0  28.1
       4      0  2.288   33     1  137.0  40.0  35.0  43.1
       6      3  0.248   26     1   78.0  50.0  32.0  31.0
```

[139]:
```
for feature in ["type"]:
    data_1[feature] = data_1[feature].astype('object')
```

[140]:
```
data.dtypes
```

[140]:
```
npreg      int64
ped      float64
age        int64
type       int64
glu      float64
bp       float64
skin     float64
bmi      float64
dtype: object
```

19 Compute Principal Components (from scratch)

20 Perform PCA with the following steps:

1. Filter the numerical variables
2. Scale the data to get variables on the same scale
3. Compute covariance matrix
4. Calculate eigenvalues and eigenvectors of the covariance matrix
5. Decide the number of principal components
6. Obtain principal components

[141]:
```
df_num_features =data_1.select_dtypes(include=[np.number])
data_num = df_num_features.drop('ped',axis=1)
data_num.head()
```

[141]:
```
   npreg  age    glu    bp  skin   bmi
0      6   50  148.0  72.0  35.0  33.6
1      1   31   85.0  66.0  29.0  26.6
3      1   21   89.0  66.0  23.0  28.1
4      0   33  137.0  40.0  35.0  43.1
6      3   26   78.0  50.0  32.0  31.0
```

[142]:
```
data_num_std = StandardScaler().fit_transform(data_num)
print(data_num_std)
```

```
[[ 0.74901305  1.71809765  0.87972826  0.04375637  0.55855696  0.10282275]
 [-0.7562472  -0.05200616 -1.16844217 -0.4460132  -0.01465704 -0.91866136]
 [-0.7562472  -0.98363975 -1.03839961 -0.4460132  -0.58787104 -0.69977191]
 …
 [-0.45519515 -0.4246596   0.03445158 -0.11950015 -0.20572838  0.56978691]
 [ 0.447961   -0.14516952  0.00194093  0.04375637 -0.58787104 -0.97703189]
 [-0.7562472  -0.79731303 -0.90835704 -0.11950015  0.17641429 -0.36414142]]
```

[143]:
```
cov_mat = np.cov(data_num_std.T)
print(cov_mat[0:5])
```

```
[[1.00185185 0.64587984 0.12755508 0.20544545 0.1004247  0.01953158]
 [0.64587984 1.00185185 0.28023555 0.34778955 0.16712469 0.08173092]
 [0.12755508 0.28023555 1.00185185 0.21798405 0.22778961 0.24757349]
 [0.20544545 0.34778955 0.21798405 1.00185185 0.22714303 0.31008079]
 [0.1004247  0.16712469 0.22778961 0.22714303 1.00185185 0.649028  ]]
```

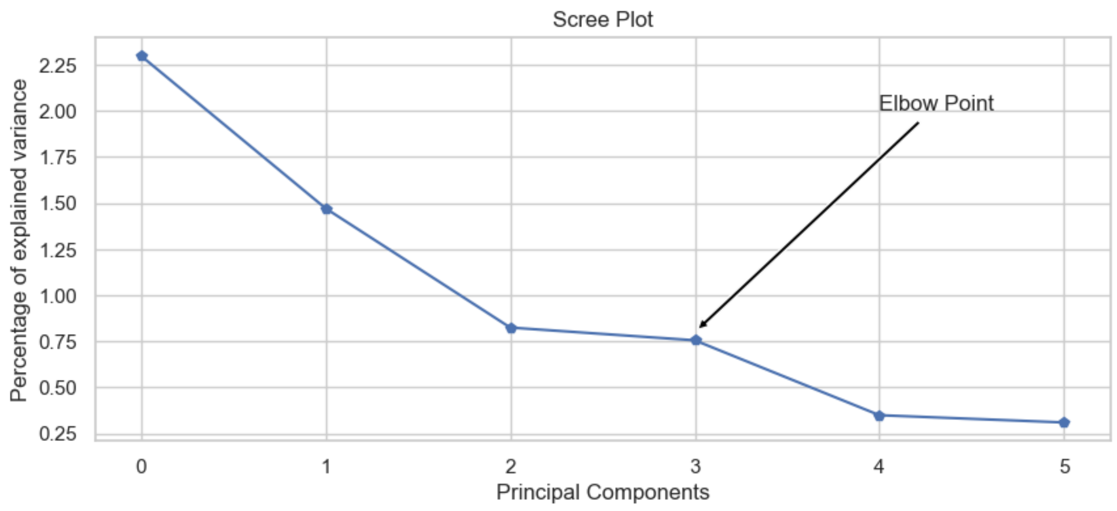
[144]:
```
eig_values, eig_vector = np.linalg.eig(cov_mat)
print('Eigen values:','\n','\n', eig_values,"\n")
print('Eigen vectors:','\n','\n',eig_vector,'\n')
```

```
Eigen values:

 [2.2991948  1.47162588 0.31010656 0.34945851 0.82499884 0.75572652]

Eigen vectors:

 [[-0.36607157 -0.54358524 -0.44030593  0.47421335  0.2818529  -0.26892012]
 [-0.45281473 -0.4839029   0.61269163 -0.41773334  0.0524138  -0.0903184 ]
 [-0.35885404  0.0456002  -0.13289832  0.07190307 -0.8978727  -0.2003422 ]
 [-0.41488941 -0.01663631 -0.20873069 -0.02897317 -0.00391307  0.88496709]
 [-0.43110956  0.44434121 -0.40069046 -0.53421526  0.27934406 -0.30452184]
 [-0.4173554   0.52015173  0.45711743  0.55608014  0.18327011 -0.05905298]]
```

[145]:
```
eig_values = list(eig_values)
eig_values.sort(reverse = True)
print(eig_values)
```

```
[2.299194798088914, 1.4716258844527035, 0.8249988427720018, 0.7557265225211692,
0.34945850631472203, 0.3101065569616022]
```

[146]:
```
plt.plot(eig_values,'bp')
plt.plot(eig_values)
plt.xlabel('Principal Components')
plt.ylabel('Percentage of explained variance')
plt.annotate(text ='Elbow Point', xy=(3,0.8), xytext=(4, 2),␣
 ↪arrowprops=dict(facecolor='black', arrowstyle = 'simple'))
plt.title('Scree Plot')
plt.show()
```



Using the scree plot alone, it is challenging to make a decision on the number of principal components to select for extracting maximum variations in the data. Therefore, we employed a manual approach, calculating the percentage of variance explained by the first p eigenvalues. In our case, the first four eigenvalues exceed 90%. Consequently, we have selected the first four components for further analysis.

```
[147]: eigenvector = eig_vector[:,0:4]
       eigenvector
```

```
[147]: array([[-0.36607157, -0.54358524, -0.44030593,  0.47421335],
              [-0.45281473, -0.4839029 ,  0.61269163, -0.41773334],
              [-0.35885404,  0.0456002 , -0.13289832,  0.07190307],
              [-0.41488941, -0.01663631, -0.20873069, -0.02897317],
              [-0.43110956,  0.44434121, -0.40069046, -0.53421526],
              [-0.4173554 ,  0.52015173,  0.45711743,  0.55608014]])
```

```
[148]: pca_data = pd.DataFrame(data_num_std.dot(eigenvector), columns=␣
        ↪['PC1','PC2','PC3','PC4'])
```

```
[149]: # Reset indices before concatenation
       pca_data_reset = pca_data.reset_index(drop=True)
       data_reset = data[['ped', 'type']].reset_index(drop=True)

       # Concatenate along the index axis
       data_pca = pd.concat([pca_data_reset, data_reset], axis=1)

       # Now check for missing values
       print("Missing values in 'data_pca':")
       print(data_pca.isnull().sum())
```

```
Missing values in 'data_pca':
PC1     0
PC2     0
PC3     0
PC4     0
ped     0
type    0
dtype: int64
```

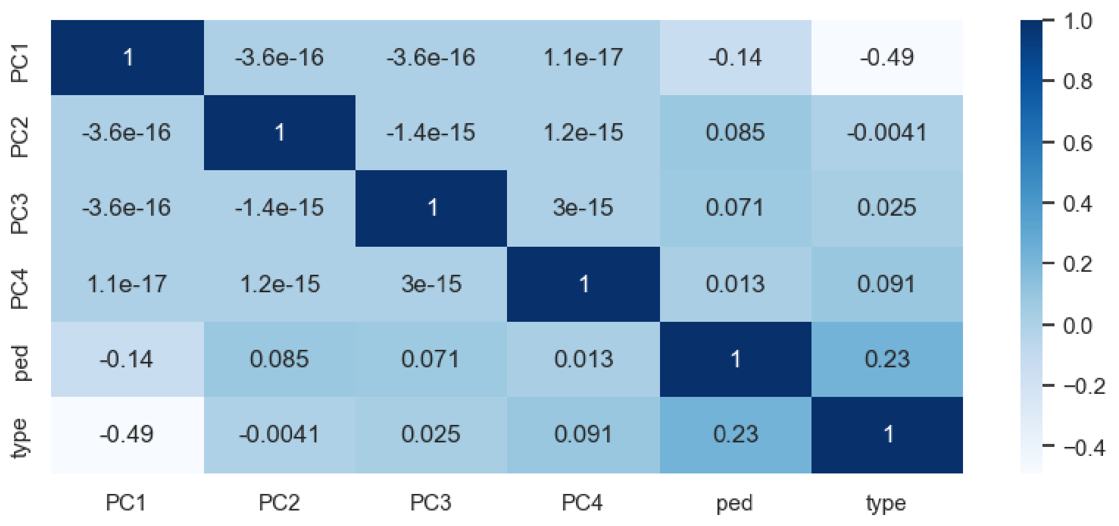
[150]:
```
data2=data_pca.copy(True)
data2.head()
```

[150]:
```
        PC1       PC2       PC3       PC4    ped  type
0 -1.669733 -0.897484  0.420015 -0.541739  0.627     1
1  1.294463 -0.093966  0.135434 -0.911009  0.351     0
2  1.825418  0.221937 -0.122913 -0.084545  0.167     0
3  0.342151  1.599030  1.471433  0.084141  2.288     1
4  1.516979  0.276827  0.066443 -0.205483  0.248     1
```

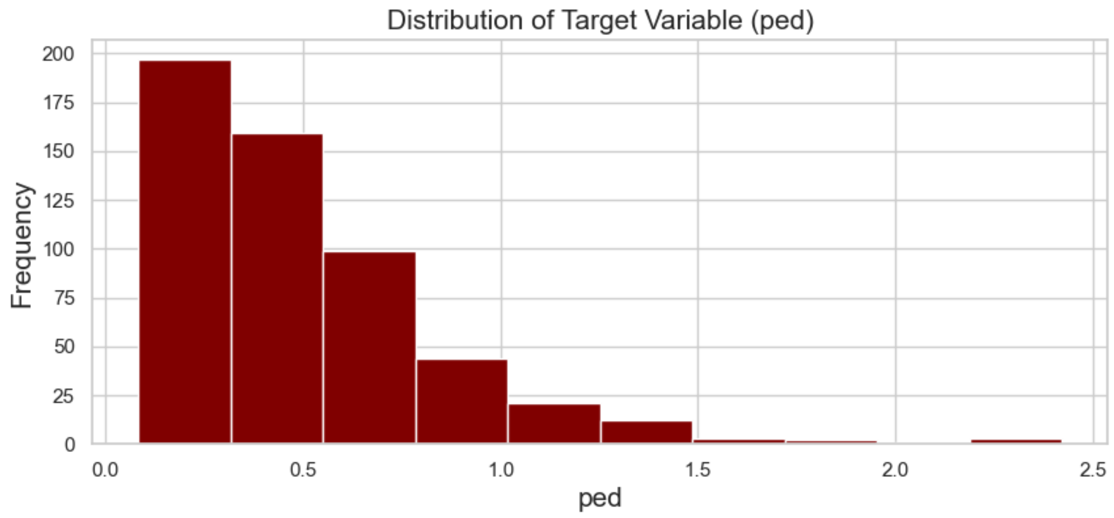
[151]:
```
data2.corr()
```

[151]:
```
               PC1           PC2           PC3           PC4       ped  \
PC1   1.000000e+00 -3.640388e-16 -3.574367e-16  1.123898e-17 -0.135099
PC2  -3.640388e-16  1.000000e+00 -1.386431e-15  1.190572e-15  0.084609
PC3  -3.574367e-16 -1.386431e-15  1.000000e+00  3.031616e-15  0.070958
PC4   1.123898e-17  1.190572e-15  3.031616e-15  1.000000e+00  0.012846
ped  -1.350988e-01  8.460946e-02  7.095778e-02  1.284589e-02  1.000000
type -4.891232e-01 -4.075420e-03  2.480845e-02  9.079427e-02  0.225474

          type
PC1  -0.489123
PC2  -0.004075
PC3   0.024808
PC4   0.090794
ped   0.225474
type  1.000000
```

[152]:
```
sns.heatmap(data2.corr(), cmap='Blues', annot=True)
plt.show()
```



[153]:
```python
data2.ped.hist(color = 'maroon')
plt.title('Distribution of Target Variable (ped)', fontsize = 15)
plt.xlabel('ped', fontsize = 15)
plt.ylabel('Frequency', fontsize = 15)

# display the plot
plt.show()
```



The distribution of the target variable is right-skewed. However, the assumption for the target variable in linear regression is a normal distribution. To address the right-skewed nature of the variable, a log transformation is applied to achieve normality.

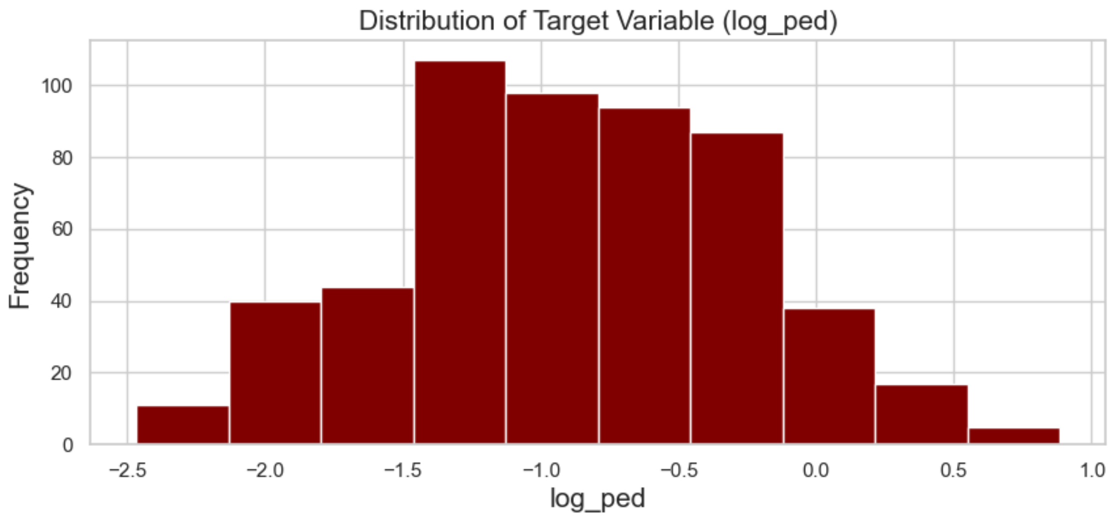
```
[154]: import numpy as np
       data2['log_ped'] = np.log(data2['ped'])

       # display the top 5 rows of data
       data2.head()
```

```
[154]:        PC1       PC2       PC3       PC4    ped  type   log_ped
       0 -1.669733 -0.897484  0.420015 -0.541739  0.627     1 -0.466809
       1  1.294463 -0.093966  0.135434 -0.911009  0.351     0 -1.046969
       2  1.825418  0.221937 -0.122913 -0.084545  0.167     0 -1.789761
       3  0.342151  1.599030  1.471433  0.084141  2.288     1  0.827678
       4  1.516979  0.276827  0.066443 -0.205483  0.248     1 -1.394327
```

```
  [1]: #data_numeric = data2.select_dtypes(include=np.number)
       #print(data_numeric.columns)
       #data_categoric = data2.select_dtypes(include = object)
       #dummy_variables = pd.get_dummies(data_categoric, drop_first = True)
```

```
  [2]: #data_2 = pd.concat([data_numeric, dummy_variables], axis=1)
       #data_2.head()
```

```
[317]: data_2.log_ped.hist(color = 'maroon')
       plt.title('Distribution of Target Variable (log_ped)', fontsize = 15)
       plt.xlabel('log_ped', fontsize = 15)
       plt.ylabel('Frequency', fontsize = 15)

       # display the plot
       plt.show()
```



[318]:
```
from sklearn.model_selection import train_test_split
X = data_2.drop(['ped',"log_ped"], axis = 1)
import statsmodels
import statsmodels.api as sm
X=sm.add_constant(X)
y = pd.DataFrame(data_2[['ped','log_ped']])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 1)
print("The shape of y_test is:",y_test.shape)
```

```
The shape of y_test is: (109, 2)
```

[319]:
```
from statsmodels.compat import lzip
import statsmodels.stats.api as sms
import statsmodels.formula.api as smf
from statsmodels.formula.api import ols
from statsmodels.tools.eval_measures import rmse
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn import metrics
```

[320]:
```
#Build model using sm.OLS().fit()
linreg_logmodel_full_pca = sm.OLS(y_train['log_ped'], X_train).fit()

# print the summary output
print(linreg_logmodel_full_pca.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                log_ped   R-squared:                       0.071
Model:                            OLS   Adj. R-squared:                  0.060
Method:                 Least Squares   F-statistic:                     6.512
Date:                Mon, 18 Dec 2023   Prob (F-statistic):           7.52e-06
Time:                        14:30:51   Log-Likelihood:                -398.47
No. Observations:                 432   AIC:                             808.9
Df Residuals:                     426   BIC:                             833.3
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -1.0074      0.037    -27.174      0.000      -1.080      -0.935
PC1           -0.0009      0.022     -0.042      0.967      -0.045       0.043
PC2            0.0159      0.024      0.664      0.507      -0.031       0.063
PC3            0.1405      0.053      2.672      0.008       0.037       0.244
PC4           -0.0214      0.053     -0.405      0.685      -0.125       0.082
```

```
type_1          0.3080      0.072      4.307      0.000       0.167       0.449
==============================================================================
Omnibus:                        3.248   Durbin-Watson:                   1.977
Prob(Omnibus):                  0.197   Jarque-Bera (JB):                2.585
Skew:                           0.058   Prob(JB):                        0.275
Kurtosis:                       2.639   Cond. No.                         3.97
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

The R-squared and adjusted R-squared values are considerably low, indicating poor performance of the model.

```
[321]: linreg_logmodel_full_predictions_pca = linreg_logmodel_full_pca.predict(X_test)
```

```
[322]: predicted_ped_pca = np.exp(linreg_logmodel_full_predictions_pca)

       # extract the 'Property_Sale_Price' values from the test data
       actual_ped= y_test['ped']
```

```
[323]: linreg_logmodel_full_rmse_pca = rmse(actual_ped, predicted_ped_pca)

       # calculate R-squared using rsquared
       linreg_logmodel_full_rsquared_pca = linreg_logmodel_full_pca.rsquared

       # calculate Adjusted R-Squared using rsquared_adj
       linreg_logmodel_full_rsquared_adj_pca = linreg_logmodel_full_pca.rsquared_adj
```

```
[324]: cols = ['Model', 'RMSE', 'R-Squared', 'Adj. R-Squared']
       result_tabulation=pd.DataFrame(columns=cols)
       linreg_logmodel_full_metrics = pd.Series({'Model': "LR model with log of target␣
        ↪variable_PCA ",
                             'RMSE':linreg_logmodel_full_rmse_pca,
                             'R-Squared': linreg_logmodel_full_rsquared_pca,
                             'Adj. R-Squared': linreg_logmodel_full_rsquared_adj_pca
                            })
       result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
        ↪ignore_index = True)
       result_tabulation
```

```
[324]:                                      Model     RMSE  R-Squared  \
       0  LR model with log of target variable_PCA   0.40599   0.071001

          Adj. R-Squared
       0        0.060097
```

```
[327]: from sklearn.linear_model import SGDRegressor
       SGD_model_pca=SGDRegressor(loss="squared_error",alpha=0.1,max_iter=1000)
       SGD_model_pca.fit(X_train,y_train["log_ped"])
       y_pred_SGD_pca=SGD_model_pca.predict(X_test)
       X_train.shape
```

```
[327]: (432, 6)
```

```
[328]: r_square_SGD_pca=SGD_model_pca.score(X_train,y_train["log_ped"])
       n=432
       p=6
       adj_R2_SGD_pca=1-(1-r_square_SGD_pca)*(n-1)/(n-p-1)
       from sklearn.metrics import mean_squared_error
       from math import sqrt
       rmse_SGD_pca=sqrt(mean_squared_error(y_test["log_ped"],y_pred_SGD_pca))
```

```
[329]: linreg_logmodel_full_metrics = pd.Series({'Model': "SGD_Model_pca",'RMSE':
        ↪rmse_SGD_pca,'R-Squared':r_square_SGD_pca,'Adj. R-Squared':adj_R2_SGD_pca })
       result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
        ↪ignore_index = True)
       result_tabulation
```

```
[329]:                                    Model      RMSE  R-Squared  \
       0  LR model with log of target variable_PCA  0.405990   0.071001
       1                           SGD_Model_pca  0.661159   0.053627

          Adj. R-Squared
       0        0.060097
       1        0.040267
```

```
[330]: data.head()
```

```
[330]:    npreg    ped  age type    glu    bp  skin   bmi
       0      6  0.627   50    1  148.0  72.0  35.0  33.6
       1      1  0.351   31    0   85.0  66.0  29.0  26.6
       3      1  0.167   21    0   89.0  66.0  23.0  28.1
       4      0  2.288   33    1  137.0  40.0  35.0  43.1
       6      3  0.248   26    1   78.0  50.0  32.0  31.0
```

```
[331]: data.shape
```

```
[331]: (541, 8)
```

```
[332]: for feature in ["type"]:
           data[feature] = data[feature].astype('object')
```

[333]:
```
import numpy as np
data['log_ped'] = np.log(data['ped'])

# display the top 5 rows of the data
data.head()
```

[333]:
```
   npreg    ped  age  type    glu    bp  skin   bmi   log_ped
0      6  0.627   50     1  148.0  72.0  35.0  33.6 -0.466809
1      1  0.351   31     0   85.0  66.0  29.0  26.6 -1.046969
3      1  0.167   21     0   89.0  66.0  23.0  28.1 -1.789761
4      0  2.288   33     1  137.0  40.0  35.0  43.1  0.827678
6      3  0.248   26     1   78.0  50.0  32.0  31.0 -1.394327
```

[334]:
```
data_numeric = data.select_dtypes(include=np.number)
print(data_numeric.columns)
data_categoric = data.select_dtypes(include = object)
print(data_categoric.columns)
```

```
Index(['npreg', 'ped', 'age', 'glu', 'bp', 'skin', 'bmi', 'log_ped'],
dtype='object')
Index(['type'], dtype='object')
```

[335]:
```
dummy_variables = pd.get_dummies(data_categoric, drop_first = True)
data_dummy = pd.concat([data_numeric, dummy_variables], axis=1)
data_dummy
```

[335]:
```
     npreg    ped  age         glu    bp  skin   bmi   log_ped  type_1
0        6  0.627   50  148.000000  72.0  35.0  33.6 -0.466809       1
1        1  0.351   31   85.000000  66.0  29.0  26.6 -1.046969       0
3        1  0.167   21   89.000000  66.0  23.0  28.1 -1.789761       0
4        0  2.288   33  137.000000  40.0  35.0  43.1  0.827678       1
6        3  0.248   26   78.000000  50.0  32.0  31.0 -1.394327       1
8        2  0.158   53  197.000000  70.0  45.0  30.5 -1.845160       1
13       1  0.398   59  189.000000  60.0  23.0  30.1 -0.921303       1
14       5  0.587   51  166.000000  72.0  19.0  25.8 -0.532730       1
16       0  0.551   31  118.000000  84.0  47.0  45.8 -0.596020       1
18       1  0.183   33  103.000000  30.0  38.0  43.3 -1.698269       0
19       1  0.529   32  115.000000  70.0  30.0  34.6 -0.636767       1
20       3  0.704   27  126.000000  88.0  41.0  39.3 -0.350977       0
23       9  0.263   29  119.000000  80.0  35.0  29.0 -1.335601       1
24      11  0.254   51  143.000000  94.0  33.0  36.6 -1.370421       1
25      10  0.205   41  125.000000  70.0  26.0  31.1 -1.584745       1
27       1  0.487   22   97.000000  66.0  15.0  23.2 -0.719491       0
28      13  0.245   57  145.000000  82.0  19.0  22.2 -1.406497       0
30       5  0.546   60  109.000000  75.0  26.0  36.0 -0.605136       0
31       3  0.851   28  158.000000  76.0  36.0  31.6 -0.161343       1
32       3  0.267   22   88.000000  58.0  11.0  24.8 -1.320507       0
```

34	10	0.512	45	122.000000	78.0	31.0	27.6	-0.669431	0
35	4	0.966	33	103.000000	60.0	33.0	24.0	-0.034591	0
37	9	0.665	46	102.000000	76.0	37.0	32.9	-0.407968	1
38	2	0.503	27	90.000000	68.0	42.0	38.2	-0.687165	1
39	4	1.390	56	111.000000	72.0	47.0	37.1	0.329304	1
40	3	0.271	26	180.000000	64.0	25.0	34.0	-1.305636	0
42	7	0.235	48	106.000000	92.0	18.0	22.7	-1.448170	0
43	9	0.721	54	171.000000	110.0	24.0	45.4	-0.327116	1
45	0	1.893	25	180.000000	66.0	39.0	42.0	0.638163	1
47	2	0.586	22	71.000000	70.0	27.0	28.0	-0.534435	0
48	7	0.344	31	103.000000	66.0	32.0	39.1	-1.067114	1
50	1	0.491	22	103.000000	80.0	11.0	19.4	-0.711311	0
51	1	0.526	26	101.000000	50.0	15.0	24.2	-0.642454	0
52	5	0.342	30	88.000000	66.0	21.0	24.4	-1.072945	0
53	8	0.467	58	176.000000	90.0	34.0	33.7	-0.761426	1
54	7	0.718	42	150.000000	66.0	42.0	34.7	-0.331286	0
55	1	0.248	21	73.000000	50.0	10.0	23.0	-1.394327	0
56	7	0.254	41	187.000000	68.0	39.0	37.7	-1.370421	1
57	0	0.962	31	100.000000	88.0	60.0	46.8	-0.038741	0
59	0	0.173	22	105.000000	64.0	41.0	41.5	-1.754464	0
63	2	0.699	24	141.000000	58.0	34.0	25.4	-0.358105	0
65	5	0.203	32	99.000000	74.0	27.0	29.0	-1.594549	0
66	0	0.855	38	109.000000	88.0	30.0	32.5	-0.156654	1
68	1	0.334	25	95.000000	66.0	13.0	19.6	-1.096614	0
69	4	0.189	27	146.000000	85.0	27.0	28.9	-1.666008	0
70	2	0.867	28	100.000000	66.0	20.0	32.9	-0.142716	1
71	5	0.411	26	139.000000	64.0	35.0	28.6	-0.889162	0
73	4	0.231	23	129.000000	86.0	20.0	35.1	-1.465338	0
74	1	0.396	22	79.000000	75.0	30.0	32.0	-0.926341	0
75	1	0.140	22	120.940299	48.0	20.0	24.7	-1.966113	0
77	5	0.370	27	95.000000	72.0	33.0	37.7	-0.994252	0
79	2	0.307	24	112.000000	66.0	22.0	25.0	-1.180908	0
80	3	0.140	22	113.000000	44.0	13.0	22.4	-1.966113	0
82	7	0.767	36	83.000000	78.0	26.0	29.3	-0.265268	0
83	0	0.237	22	101.000000	65.0	28.0	24.6	-1.439695	0
85	2	0.698	27	110.000000	74.0	29.0	32.4	-0.359536	0
86	13	0.178	45	106.000000	72.0	54.0	36.6	-1.725972	0
87	2	0.324	26	100.000000	68.0	25.0	38.5	-1.127012	0
88	15	0.153	43	136.000000	70.0	32.0	37.1	-1.877317	1
89	1	0.165	24	107.000000	68.0	19.0	26.5	-1.801810	0
91	4	0.443	34	123.000000	80.0	15.0	32.0	-0.814186	0
92	7	0.261	42	81.000000	78.0	40.0	46.7	-1.343235	0
94	2	0.761	21	142.000000	82.0	18.0	24.7	-0.273122	0
95	6	0.255	40	144.000000	72.0	27.0	33.9	-1.366492	0
96	2	0.130	24	92.000000	62.0	28.0	31.6	-2.040221	0
97	1	0.323	22	71.000000	48.0	18.0	20.4	-1.130103	0
98	6	0.356	23	93.000000	50.0	30.0	28.7	-1.032825	0

```
99       1  0.325   31  122.000000   90.0  51.0  49.7 -1.123930       1
103      1  0.283   24   81.000000   72.0  18.0  26.6 -1.262308       0
105      1  0.801   21  126.000000   56.0  29.0  28.7 -0.221894       0
107      4  0.287   37  144.000000   58.0  28.0  29.5 -1.248273       0
108      3  0.336   25   83.000000   58.0  31.0  34.3 -1.090644       0
109      0  0.247   24   95.000000   85.0  25.0  37.4 -1.398367       1
110      3  0.199   24  171.000000   72.0  33.0  33.3 -1.614450       1
111      8  0.543   46  155.000000   62.0  26.0  34.0 -0.610646       1
112      1  0.192   23   89.000000   76.0  34.0  31.2 -1.650260       0
114      7  0.588   39  160.000000   54.0  32.0  30.5 -0.531028       1
118      4  0.443   22   97.000000   60.0  23.0  28.2 -0.814186       0
119      4  0.223   21   99.000000   76.0  15.0  23.2 -1.500584       0
120      0  0.759   25  162.000000   76.0  56.0  53.2 -0.275754       1
121      6  0.260   24  111.000000   64.0  39.0  34.2 -1.347074       0
122      2  0.404   23  107.000000   74.0  30.0  33.6 -0.906340       0
125      1  0.496   26   88.000000   30.0  42.0  55.0 -0.701179       1
126      3  0.452   30  120.000000   70.0  30.0  42.9 -0.794073       0
127      1  0.261   23  118.000000   58.0  36.0  33.3 -1.343235       0
128      1  0.403   40  117.000000   88.0  24.0  34.5 -0.908819       1
130      4  0.361   33  173.000000   70.0  14.0  29.7 -1.018877       1
132      3  0.356   30  170.000000   64.0  37.0  34.5 -1.032825       1
133      8  0.457   39   84.000000   74.0  31.0  38.3 -0.783072       0
134      2  0.647   26   96.000000   68.0  13.0  21.1 -0.435409       0
135      2  0.088   31  125.000000   60.0  20.0  33.8 -2.430418       0
136      0  0.597   21  100.000000   70.0  26.0  30.8 -0.515838       0
137      0  0.532   22   93.000000   60.0  25.0  28.7 -0.631112       0
139      5  0.159   28  105.000000   72.0  29.0  36.9 -1.838851       0
141      5  0.286   38  106.000000   82.0  30.0  39.5 -1.251763       0
142      2  0.318   22  108.000000   52.0  26.0  32.5 -1.145704       0
144      4  0.237   23  154.000000   62.0  31.0  32.8 -1.439695       0
145      0  0.572   21  102.000000   75.0  23.0  32.8 -0.558616       0
146      9  0.096   41   57.000000   80.0  37.0  32.8 -2.343407       0
147      2  1.400   34  106.000000   64.0  35.0  30.5  0.336472       0
149      2  0.085   22   90.000000   70.0  17.0  27.3 -2.465104       0
150      1  0.399   24  136.000000   74.0  50.0  37.4 -0.918794       0
152      9  1.189   42  156.000000   86.0  28.0  34.3  0.173113       1
153      1  0.687   23  153.000000   82.0  42.0  40.6 -0.375421       0
155      7  0.337   36  152.000000   88.0  44.0  50.0 -1.087672       1
156      2  0.637   21   99.000000   52.0  15.0  24.6 -0.450986       0
157      1  0.833   23  109.000000   56.0  21.0  25.2 -0.182722       0
158      2  0.229   22   88.000000   74.0  19.0  29.0 -1.474033       0
159     17  0.817   47  163.000000   72.0  41.0  40.9 -0.202116       1
160      4  0.294   36  151.000000   90.0  38.0  29.7 -1.224176       0
161      7  0.204   45  102.000000   74.0  40.0  37.2 -1.589635       0
162      0  0.167   27  114.000000   80.0  34.0  44.2 -1.789761       0
163      2  0.368   21  100.000000   64.0  23.0  29.7 -0.999672       0
165      6  0.722   41  104.000000   74.0  18.0  29.9 -0.325730       1
```

166	3	0.256	22	148.000000	66.0	25.0	32.5	-1.362578	0
169	3	0.495	29	111.000000	90.0	12.0	28.4	-0.703198	0
171	6	0.542	29	134.000000	70.0	23.0	35.4	-0.612489	1
172	2	0.773	25	87.000000	72.0	23.0	28.9	-0.257476	0
173	1	0.678	23	79.000000	60.0	42.0	43.5	-0.388608	0
174	2	0.370	33	75.000000	64.0	24.0	29.7	-0.994252	0
175	8	0.719	36	179.000000	72.0	42.0	32.7	-0.329894	1
177	0	0.319	26	129.000000	110.0	46.0	67.1	-1.142564	1
181	0	0.725	23	119.000000	64.0	18.0	34.9	-0.321584	0
182	1	0.299	21	120.940299	74.0	20.0	27.7	-1.207312	0
185	7	0.745	41	194.000000	68.0	28.0	35.9	-0.294371	1
186	8	0.615	60	181.000000	68.0	36.0	30.1	-0.486133	1
187	1	1.321	33	128.000000	98.0	41.0	32.0	0.278389	1
188	8	0.640	31	109.000000	76.0	39.0	27.9	-0.446287	1
189	5	0.361	25	139.000000	80.0	35.0	31.6	-1.018877	1
191	9	0.374	40	123.000000	70.0	44.0	33.1	-0.983499	0
194	8	0.136	42	85.000000	55.0	20.0	24.4	-1.995100	0
195	5	0.395	29	158.000000	84.0	41.0	39.4	-0.928870	1
197	3	0.678	23	107.000000	62.0	13.0	22.9	-0.388608	1
198	4	0.905	26	109.000000	64.0	44.0	34.8	-0.099820	1
199	4	0.150	29	148.000000	60.0	27.0	30.9	-1.897120	1
200	0	0.874	21	113.000000	80.0	16.0	31.0	-0.134675	0
202	0	0.787	32	108.000000	68.0	20.0	27.3	-0.239527	0
203	2	0.235	27	99.000000	70.0	16.0	20.4	-1.448170	0
204	6	0.324	55	103.000000	72.0	32.0	37.7	-1.127012	0
205	5	0.407	27	111.000000	72.0	28.0	23.9	-0.898942	0
206	8	0.605	57	196.000000	76.0	29.0	37.5	-0.502527	1
208	1	0.289	21	96.000000	64.0	27.0	33.2	-1.241329	0
209	7	0.355	41	184.000000	84.0	33.0	35.5	-1.035637	1
210	2	0.290	25	81.000000	60.0	22.0	27.7	-1.237874	0
211	0	0.375	24	147.000000	85.0	54.0	42.8	-0.980829	0
212	7	0.164	60	179.000000	95.0	31.0	34.2	-1.807889	0
213	0	0.431	24	140.000000	65.0	26.0	42.6	-0.841647	1
214	9	0.260	36	112.000000	82.0	32.0	34.2	-1.347074	1
215	12	0.742	38	151.000000	70.0	40.0	41.8	-0.298406	1
216	5	0.514	25	109.000000	62.0	41.0	35.8	-0.665532	1
217	6	0.464	32	125.000000	68.0	30.0	30.0	-0.767871	0
218	5	1.224	32	85.000000	74.0	22.0	29.0	0.202124	1
220	0	1.072	21	177.000000	60.0	29.0	34.6	0.069526	1
223	7	0.687	61	142.000000	60.0	33.0	28.8	-0.375421	0
224	1	0.666	26	100.000000	66.0	15.0	23.6	-0.406466	0
225	1	0.101	22	87.000000	78.0	27.0	34.6	-2.292635	0
227	3	0.652	24	162.000000	52.0	38.0	37.2	-0.427711	1
228	4	2.329	31	197.000000	70.0	39.0	36.7	0.845439	0
229	0	0.089	24	117.000000	80.0	31.0	45.2	-2.419119	0
231	6	0.238	46	134.000000	80.0	37.0	46.2	-1.435485	1
232	1	0.583	22	79.000000	80.0	25.0	25.4	-0.539568	0

234	3	0.293	23	74.000000	68.0	28.0	29.7	-1.227583	0
236	7	0.586	51	181.000000	84.0	21.0	35.9	-0.534435	1
237	0	0.686	23	179.000000	90.0	27.0	44.1	-0.376878	1
238	9	0.831	32	164.000000	84.0	21.0	30.8	-0.185125	1
240	1	0.192	21	91.000000	64.0	24.0	29.2	-1.650260	0
241	4	0.446	22	91.000000	70.0	32.0	33.1	-0.807436	0
243	6	1.318	33	119.000000	50.0	22.0	27.1	0.276115	1
244	2	0.329	29	146.000000	76.0	35.0	38.2	-1.111698	0
245	9	1.213	49	184.000000	85.0	15.0	30.0	0.193097	1
247	0	0.427	23	165.000000	90.0	33.0	52.3	-0.850971	0
248	9	0.282	34	124.000000	70.0	33.0	35.4	-1.265848	0
249	1	0.143	23	111.000000	86.0	19.0	30.1	-1.944911	0
252	2	0.249	24	90.000000	80.0	14.0	24.4	-1.390302	0
253	0	0.238	25	86.000000	68.0	32.0	35.8	-1.435485	0
254	12	0.926	44	92.000000	62.0	7.0	27.6	-0.076881	1
255	1	0.543	21	113.000000	64.0	35.0	33.6	-0.610646	1
256	3	0.557	30	111.000000	56.0	39.0	30.1	-0.585190	0
257	2	0.092	25	114.000000	68.0	22.0	28.7	-2.385967	0
258	1	0.655	24	193.000000	50.0	16.0	25.9	-0.423120	0
259	11	1.353	51	155.000000	76.0	28.0	33.3	0.302324	1
260	3	0.299	34	191.000000	68.0	15.0	30.9	-1.207312	0
262	4	0.612	24	95.000000	70.0	32.0	32.1	-0.491023	0
263	3	0.200	63	142.000000	80.0	15.0	32.4	-1.609438	0
265	5	0.997	43	96.000000	74.0	18.0	33.6	-0.003005	0
267	2	1.101	24	128.000000	64.0	42.0	40.0	0.096219	0
270	10	1.136	38	101.000000	86.0	37.0	45.6	0.127513	1
271	2	0.128	21	108.000000	62.0	32.0	25.2	-2.055725	0
273	1	0.422	21	71.000000	78.0	50.0	33.2	-0.862750	0
275	2	0.677	25	100.000000	70.0	52.0	40.5	-0.390084	0
276	7	0.296	29	106.000000	60.0	24.0	26.5	-1.217396	1
277	0	0.454	23	104.000000	64.0	23.0	27.8	-0.789658	0
279	2	0.881	22	108.000000	62.0	10.0	25.3	-0.126698	0
281	10	0.280	39	129.000000	76.0	28.0	35.9	-1.272966	0
282	7	0.262	37	133.000000	88.0	15.0	32.4	-1.339411	0
285	7	0.647	51	136.000000	74.0	26.0	26.0	-0.435409	0
286	5	0.619	34	155.000000	84.0	44.0	38.7	-0.479650	0
287	1	0.808	29	119.000000	86.0	39.0	45.6	-0.213193	1
288	4	0.340	26	96.000000	56.0	17.0	20.8	-1.078810	0
289	5	0.263	33	108.000000	72.0	43.0	36.1	-1.335601	0
290	0	0.434	21	78.000000	88.0	29.0	36.9	-0.834711	0
291	0	0.757	25	107.000000	62.0	30.0	36.6	-0.278392	1
292	2	1.224	31	128.000000	78.0	37.0	43.3	0.202124	1
293	1	0.613	24	128.000000	48.0	45.0	40.5	-0.489390	1
295	6	0.692	28	151.000000	62.0	31.0	35.5	-0.368169	0
296	2	0.337	29	146.000000	70.0	38.0	28.0	-1.087672	1
297	0	0.520	24	126.000000	84.0	29.0	30.7	-0.653926	0
298	14	0.412	46	100.000000	78.0	25.0	36.6	-0.886732	1

301	2	0.422	25	144.000000	58.0	33.0	31.6	-0.862750	1
302	5	0.156	35	77.000000	82.0	41.0	35.8	-1.857899	0
305	2	0.215	29	120.000000	76.0	37.0	39.7	-1.537117	0
306	10	0.326	47	161.000000	68.0	23.0	25.5	-1.120858	1
307	0	0.143	21	137.000000	68.0	14.0	24.8	-1.944911	0
308	0	1.391	25	128.000000	68.0	19.0	30.5	0.330023	1
309	2	0.875	30	124.000000	68.0	28.0	32.9	-0.133531	1
310	6	0.313	41	80.000000	66.0	30.0	26.2	-1.161552	0
311	0	0.605	22	106.000000	70.0	37.0	39.4	-0.502527	0
312	2	0.433	27	155.000000	74.0	17.0	26.6	-0.837018	1
313	3	0.626	25	113.000000	50.0	10.0	29.5	-0.468405	0
314	7	1.127	43	109.000000	80.0	31.0	35.9	0.119559	1
315	2	0.315	26	112.000000	68.0	22.0	34.1	-1.155183	0
316	3	0.284	30	99.000000	80.0	11.0	19.3	-1.258781	0
318	3	0.150	28	115.000000	66.0	39.0	38.1	-1.897120	0
320	4	0.527	31	129.000000	60.0	12.0	27.5	-0.640555	0
321	3	0.197	25	112.000000	74.0	30.0	31.6	-1.624552	1
322	0	0.254	36	124.000000	70.0	20.0	27.4	-1.370421	1
323	13	0.731	43	152.000000	90.0	33.0	26.8	-0.313342	1
324	2	0.148	21	112.000000	75.0	32.0	35.7	-1.910543	0
325	1	0.123	24	157.000000	72.0	21.0	25.6	-2.095571	0
326	1	0.692	30	122.000000	64.0	32.0	35.1	-0.368169	1
328	2	0.127	23	102.000000	86.0	36.0	45.5	-2.063568	1
329	6	0.122	37	105.000000	70.0	32.0	30.8	-2.103734	0
330	8	1.476	46	118.000000	72.0	19.0	23.1	0.389336	0
331	2	0.166	25	87.000000	58.0	16.0	32.7	-1.795767	0
334	1	0.260	22	95.000000	60.0	18.0	23.9	-1.347074	0
335	0	0.259	26	165.000000	76.0	43.0	47.9	-1.350927	0
338	9	0.893	33	152.000000	78.0	34.0	34.2	-0.113169	1
340	1	0.472	22	130.000000	70.0	13.0	25.9	-0.750776	0
341	1	0.673	36	95.000000	74.0	21.0	25.9	-0.396010	0
342	1	0.389	22	120.940299	68.0	35.0	32.0	-0.944176	0
345	8	0.349	49	126.000000	88.0	36.0	38.5	-1.052683	0
346	1	0.654	22	139.000000	46.0	19.0	28.7	-0.424648	0
348	3	0.279	26	99.000000	62.0	19.0	21.8	-1.276543	0
349	5	0.346	37	120.940299	80.0	32.0	41.0	-1.061317	1
352	3	0.243	46	61.000000	82.0	28.0	34.4	-1.414694	0
353	1	0.580	24	90.000000	62.0	12.0	27.2	-0.544727	0
356	1	0.962	28	125.000000	50.0	40.0	33.3	-0.038741	1
357	13	0.569	44	129.000000	72.0	30.0	39.9	-0.563875	1
358	12	0.378	48	88.000000	74.0	40.0	35.3	-0.972861	0
359	1	0.875	29	196.000000	76.0	36.0	36.5	-0.133531	1
360	5	0.583	29	189.000000	64.0	33.0	31.2	-0.539568	1
362	5	0.305	65	103.000000	108.0	37.0	39.2	-1.187444	0
364	4	0.385	30	147.000000	74.0	25.0	34.9	-0.954512	0
365	5	0.499	30	99.000000	54.0	28.0	34.0	-0.695149	0
367	0	0.252	21	101.000000	64.0	17.0	21.0	-1.378326	0

368	3	0.306	22	81.000000	86.0	16.0	27.5	-1.184170	0
369	1	0.234	45	133.000000	102.0	28.0	32.8	-1.452434	1
370	3	2.137	25	173.000000	82.0	48.0	38.4	0.759403	1
371	0	1.731	21	118.000000	64.0	23.0	32.8	0.548699	0
372	0	0.545	21	84.000000	64.0	22.0	35.8	-0.606969	0
373	2	0.225	25	105.000000	58.0	40.0	34.9	-1.491655	0
374	2	0.816	28	122.000000	52.0	43.0	36.2	-0.203341	0
375	12	0.528	58	140.000000	82.0	43.0	39.2	-0.638659	1
376	0	0.299	22	98.000000	82.0	15.0	25.2	-1.207312	0
377	1	0.509	22	87.000000	60.0	37.0	37.2	-0.675307	0
379	0	1.021	35	93.000000	100.0	39.0	43.4	0.020783	0
380	1	0.821	24	107.000000	72.0	30.0	30.8	-0.197232	0
381	0	0.236	22	105.000000	68.0	22.0	20.0	-1.443923	0
382	1	0.947	21	109.000000	60.0	8.0	25.4	-0.054456	0
383	1	1.268	25	90.000000	62.0	18.0	25.1	0.237441	0
384	1	0.221	25	125.000000	70.0	24.0	24.3	-1.509593	0
385	1	0.205	24	119.000000	54.0	13.0	22.3	-1.584745	0
386	5	0.660	35	116.000000	74.0	29.0	32.3	-0.415515	1
387	8	0.239	45	105.000000	100.0	36.0	43.3	-1.431292	1
388	5	0.452	58	144.000000	82.0	26.0	32.0	-0.794073	1
389	3	0.949	28	100.000000	68.0	23.0	31.6	-0.052346	0
390	1	0.444	42	100.000000	66.0	29.0	32.0	-0.811931	0
392	1	0.389	21	131.000000	64.0	14.0	23.7	-0.944176	0
393	4	0.463	37	116.000000	72.0	12.0	22.1	-0.770028	0
395	2	1.600	25	127.000000	58.0	24.0	27.7	0.470004	0
396	3	0.944	39	96.000000	56.0	34.0	24.7	-0.057629	0
397	0	0.196	22	131.000000	66.0	40.0	34.3	-1.629641	1
399	3	0.241	25	193.000000	70.0	31.0	34.9	-1.422958	1
402	5	0.286	35	136.000000	84.0	41.0	35.0	-1.251763	1
403	9	0.280	38	72.000000	78.0	25.0	31.6	-1.272966	0
405	2	0.520	26	123.000000	48.0	32.0	42.1	-0.653926	0
409	1	0.702	28	172.000000	68.0	49.0	42.4	-0.353822	1
410	6	0.674	28	102.000000	90.0	39.0	35.7	-0.394525	0
411	1	0.528	25	112.000000	72.0	30.0	34.4	-0.638659	0
412	1	1.076	22	143.000000	84.0	23.0	42.4	0.073250	0
413	1	0.256	21	143.000000	74.0	22.0	26.2	-1.362578	0
414	0	0.534	21	138.000000	60.0	35.0	34.6	-0.627359	1
415	3	0.258	22	173.000000	84.0	33.0	35.7	-1.354796	1
416	1	1.095	22	97.000000	68.0	21.0	27.2	0.090754	0
417	4	0.554	37	144.000000	82.0	32.0	38.5	-0.590591	1
419	3	0.219	28	129.000000	64.0	29.0	26.4	-1.518684	1
420	1	0.507	26	119.000000	88.0	41.0	45.3	-0.679244	0
421	2	0.561	21	94.000000	68.0	18.0	26.0	-0.578034	0
422	0	0.496	21	102.000000	64.0	46.0	40.6	-0.701179	0
423	2	0.421	21	115.000000	64.0	22.0	30.8	-0.865122	0
424	8	0.516	36	151.000000	78.0	32.0	42.9	-0.661649	1
425	4	0.264	31	184.000000	78.0	39.0	37.0	-1.331806	1

427	1	0.328	38	181.000000	64.0	30.0	34.1	-1.114742	1
428	0	0.284	26	135.000000	94.0	46.0	40.6	-1.258781	0
429	1	0.233	43	95.000000	82.0	25.0	35.0	-1.456717	1
431	3	0.551	38	89.000000	74.0	16.0	30.4	-0.596020	0
432	1	0.527	22	80.000000	74.0	11.0	30.0	-0.640555	0
434	1	1.138	36	90.000000	68.0	8.0	24.5	0.129272	0
436	12	0.244	41	140.000000	85.0	33.0	37.4	-1.410587	0
438	1	0.147	21	97.000000	70.0	15.0	18.2	-1.917323	0
440	0	0.435	41	189.000000	104.0	25.0	34.3	-0.832409	1
441	2	0.497	22	83.000000	66.0	23.0	32.2	-0.699165	0
442	4	0.230	24	117.000000	64.0	27.0	33.2	-1.469676	0
444	4	0.380	30	117.000000	62.0	12.0	29.7	-0.967584	1
445	0	2.420	25	180.000000	78.0	63.0	59.4	0.883768	1
446	1	0.658	28	100.000000	72.0	12.0	25.3	-0.418550	0
447	0	0.330	26	95.000000	80.0	45.0	36.5	-1.108663	0
448	0	0.510	22	104.000000	64.0	37.0	33.6	-0.673345	1
449	0	0.285	26	120.000000	74.0	18.0	30.5	-1.255266	0
450	1	0.415	23	82.000000	64.0	13.0	21.2	-0.879477	0
452	0	0.381	25	91.000000	68.0	32.0	39.9	-0.964956	0
454	2	0.498	24	100.000000	54.0	28.0	37.8	-0.697155	0
455	14	0.212	38	175.000000	62.0	30.0	33.6	-1.551169	1
457	5	0.364	24	86.000000	68.0	28.0	30.2	-1.010601	0
458	10	1.001	51	148.000000	84.0	48.0	37.6	0.001000	1
459	9	0.460	81	134.000000	74.0	33.0	25.9	-0.776529	0
460	9	0.733	48	120.000000	72.0	22.0	20.8	-0.310610	0
462	8	0.705	39	74.000000	70.0	40.0	35.3	-0.349557	0
463	5	0.258	37	88.000000	78.0	30.0	27.6	-1.354796	0
465	0	0.452	21	124.000000	56.0	13.0	21.8	-0.794073	0
466	0	0.269	22	74.000000	52.0	10.0	27.8	-1.313044	0
467	0	0.600	25	97.000000	64.0	36.0	36.8	-0.510826	0
469	6	0.571	27	154.000000	78.0	41.0	46.1	-0.560366	0
470	1	0.607	28	144.000000	82.0	40.0	41.3	-0.499226	0
471	0	0.170	22	137.000000	70.0	38.0	33.2	-1.771957	0
472	0	0.259	22	119.000000	66.0	27.0	38.8	-1.350927	0
475	0	0.231	59	137.000000	84.0	27.0	27.3	-1.465338	0
476	2	0.711	29	105.000000	80.0	45.0	33.7	-0.341083	1
477	7	0.466	31	114.000000	76.0	17.0	23.8	-0.763570	0
478	8	0.162	39	126.000000	74.0	38.0	25.9	-1.820159	0
479	4	0.419	63	132.000000	86.0	31.0	28.0	-0.869884	0
480	3	0.344	35	158.000000	70.0	30.0	35.5	-1.067114	1
481	0	0.197	29	123.000000	88.0	37.0	35.2	-1.624552	0
482	4	0.306	28	85.000000	58.0	22.0	27.8	-1.184170	0
483	0	0.233	23	84.000000	82.0	31.0	38.2	-1.456717	0
485	0	0.365	24	135.000000	68.0	42.0	42.3	-1.007858	1
486	1	0.536	21	139.000000	62.0	41.0	40.7	-0.623621	0
487	0	1.159	58	173.000000	78.0	32.0	46.5	0.147558	0
488	4	0.294	28	99.000000	72.0	17.0	25.6	-1.224176	0

490	2	0.629	24	83.000000	65.0	28.0	36.8	-0.463624	0
491	2	0.292	42	89.000000	90.0	30.0	33.5	-1.231001	0
492	4	0.145	33	99.000000	68.0	38.0	32.8	-1.931022	0
493	4	1.144	45	125.000000	70.0	18.0	28.9	0.134531	1
497	2	0.547	25	81.000000	72.0	15.0	30.1	-0.603306	0
498	7	0.163	55	195.000000	70.0	33.0	25.1	-1.814005	1
499	6	0.839	39	154.000000	74.0	32.0	29.3	-0.175545	0
500	2	0.313	21	117.000000	90.0	19.0	25.2	-1.161552	0
501	3	0.267	28	84.000000	72.0	32.0	37.2	-1.320507	0
502	6	0.727	41	120.940299	68.0	41.0	39.0	-0.318829	1
503	7	0.738	41	94.000000	64.0	25.0	33.3	-0.303811	0
504	3	0.238	40	96.000000	78.0	39.0	37.3	-1.435485	0
506	0	0.314	35	180.000000	90.0	26.0	36.5	-1.158362	1
507	1	0.692	21	130.000000	60.0	23.0	28.6	-0.368169	0
508	2	0.968	21	84.000000	50.0	23.0	30.4	-0.032523	0
510	12	0.297	46	84.000000	72.0	31.0	29.7	-1.214023	1
511	0	0.207	21	139.000000	62.0	17.0	22.1	-1.575036	0
514	3	0.154	24	99.000000	54.0	19.0	25.6	-1.870803	0
515	3	0.268	28	163.000000	70.0	18.0	31.6	-1.316768	1
516	9	0.771	53	145.000000	88.0	34.0	30.3	-0.260067	1
519	6	0.582	60	129.000000	90.0	7.0	19.6	-0.541285	0
520	2	0.187	25	68.000000	70.0	32.0	25.0	-1.676647	0
521	3	0.305	26	124.000000	80.0	33.0	33.2	-1.187444	0
525	3	0.444	21	87.000000	60.0	18.0	21.8	-0.811931	0
526	1	0.299	21	97.000000	64.0	19.0	18.2	-1.207312	0
527	3	0.107	24	116.000000	74.0	15.0	26.3	-2.234926	0
528	0	0.493	22	117.000000	66.0	31.0	30.8	-0.707246	0
530	2	0.717	22	122.000000	60.0	18.0	29.8	-0.332679	0
532	1	0.917	29	86.000000	66.0	52.0	41.3	-0.086648	0
534	1	1.251	24	77.000000	56.0	30.0	33.3	0.223943	0
538	0	0.804	23	127.000000	80.0	37.0	36.3	-0.218156	0
539	3	0.968	32	129.000000	92.0	49.0	36.4	-0.032523	1
540	8	0.661	43	100.000000	74.0	40.0	39.4	-0.414001	1
541	3	0.549	27	128.000000	72.0	25.0	32.4	-0.599657	1
542	10	0.825	56	90.000000	85.0	32.0	34.9	-0.192372	1
543	4	0.159	25	84.000000	90.0	23.0	39.5	-1.838851	0
544	1	0.365	29	88.000000	78.0	29.0	32.0	-1.007858	0
545	8	0.423	37	186.000000	90.0	35.0	34.5	-0.860383	1
546	5	1.034	53	187.000000	76.0	27.0	43.6	0.033435	1
547	4	0.160	28	131.000000	68.0	21.0	33.1	-1.832581	0
548	1	0.341	50	164.000000	82.0	43.0	32.8	-1.075873	0
549	4	0.680	37	189.000000	110.0	31.0	28.5	-0.385662	0
550	1	0.204	21	116.000000	70.0	28.0	27.4	-1.589635	0
551	3	0.591	25	84.000000	68.0	30.0	31.9	-0.525939	0
553	1	0.422	23	88.000000	62.0	24.0	29.9	-0.862750	0
554	1	0.471	28	84.000000	64.0	23.0	36.9	-0.752897	0
555	7	0.161	37	124.000000	70.0	33.0	25.5	-1.826351	0

556	1	0.218	30	97.000000	70.0	40.0	38.1	-1.523260	0
558	11	0.126	42	103.000000	68.0	40.0	46.2	-2.071473	0
561	0	0.502	28	198.000000	66.0	32.0	41.3	-0.689155	1
562	1	0.401	24	87.000000	68.0	34.0	37.6	-0.913794	0
563	6	0.497	32	99.000000	60.0	19.0	26.9	-0.699165	0
565	2	0.748	22	95.000000	54.0	14.0	26.1	-0.290352	0
566	1	0.412	21	99.000000	72.0	30.0	38.6	-0.886732	0
567	6	0.085	46	92.000000	62.0	32.0	32.0	-2.465104	0
568	4	0.338	37	154.000000	72.0	29.0	31.3	-1.084709	0
569	0	0.203	33	121.000000	66.0	30.0	34.3	-1.594549	1
572	3	0.430	22	111.000000	58.0	31.0	29.5	-0.843970	0
573	2	0.198	22	98.000000	60.0	17.0	34.7	-1.619488	0
574	1	0.892	23	143.000000	86.0	30.0	30.1	-0.114289	0
575	1	0.280	25	119.000000	44.0	47.0	35.5	-1.272966	0
576	6	0.813	35	108.000000	44.0	20.0	24.0	-0.207024	0
579	2	0.575	62	197.000000	70.0	99.0	34.7	-0.553385	1
580	0	0.371	21	151.000000	90.0	46.0	42.1	-0.991553	1
581	6	0.206	27	109.000000	60.0	27.0	25.0	-1.579879	0
582	12	0.259	62	121.000000	78.0	17.0	26.5	-1.350927	0
584	8	0.687	52	124.000000	76.0	24.0	28.7	-0.375421	1
585	1	0.417	22	93.000000	56.0	11.0	22.5	-0.874669	0
588	3	1.154	52	176.000000	86.0	27.0	33.3	0.143234	1
590	11	0.925	45	111.000000	84.0	40.0	46.8	-0.077962	1
591	2	0.175	24	112.000000	78.0	50.0	39.4	-1.742969	0
593	2	1.699	25	82.000000	52.0	22.0	28.5	0.530040	0
594	6	0.733	34	123.000000	72.0	45.0	33.6	-0.310610	0
595	0	0.682	22	188.000000	82.0	14.0	32.0	-0.382726	1
597	1	0.559	21	89.000000	24.0	19.0	27.8	-0.581606	0
599	1	0.407	26	109.000000	38.0	18.0	23.1	-0.898942	0
600	1	0.400	24	108.000000	88.0	19.0	27.1	-0.916291	0
602	1	0.100	30	124.000000	74.0	36.0	27.8	-2.302585	0
603	7	0.692	54	150.000000	78.0	29.0	35.2	-0.368169	1
605	1	0.514	21	124.000000	60.0	32.0	35.8	-0.665532	0
606	1	1.258	22	181.000000	78.0	42.0	40.0	0.229523	1
607	1	0.482	25	92.000000	62.0	25.0	19.5	-0.729811	0
608	0	0.270	27	152.000000	82.0	39.0	41.5	-1.309333	0
609	1	0.138	23	111.000000	62.0	13.0	24.0	-1.980502	0
610	3	0.292	24	106.000000	54.0	21.0	30.9	-1.231001	0
611	3	0.593	36	174.000000	58.0	22.0	32.9	-0.522561	1
612	7	0.787	40	168.000000	88.0	42.0	38.2	-0.239527	1
613	6	0.878	26	105.000000	80.0	28.0	32.5	-0.130109	0
614	11	0.557	50	138.000000	74.0	26.0	36.1	-0.585190	1
617	2	0.257	23	68.000000	62.0	13.0	20.1	-1.358679	0
618	9	1.282	50	112.000000	82.0	24.0	28.2	0.248421	1
620	2	0.246	28	112.000000	86.0	42.0	38.4	-1.402424	0
621	2	1.698	28	92.000000	76.0	20.0	24.2	0.529451	0
623	0	0.347	21	94.000000	70.0	27.0	43.5	-1.058430	0

625	4	0.362	29	90.000000	88.0	47.0	37.7	-1.016111	0
629	4	0.148	21	94.000000	65.0	22.0	24.7	-1.910543	0
631	0	0.238	24	102.000000	78.0	40.0	34.5	-1.435485	0
633	1	0.115	22	128.000000	82.0	17.0	27.5	-2.162823	0
637	2	0.649	23	94.000000	76.0	18.0	31.6	-0.432323	0
638	7	0.871	32	97.000000	76.0	32.0	40.9	-0.138113	1
639	1	0.149	28	100.000000	74.0	12.0	19.5	-1.903809	0
640	0	0.695	27	102.000000	86.0	17.0	29.3	-0.363843	0
644	3	0.730	27	103.000000	72.0	30.0	27.6	-0.314711	0
645	2	0.134	30	157.000000	74.0	35.0	39.4	-2.009915	0
646	1	0.447	33	167.000000	74.0	17.0	23.4	-0.805197	1
647	0	0.455	22	179.000000	50.0	36.0	37.8	-0.787458	1
648	11	0.260	42	136.000000	84.0	35.0	28.3	-1.347074	1
649	0	0.133	23	107.000000	60.0	25.0	26.4	-2.017406	0
650	1	0.234	23	91.000000	54.0	25.0	25.2	-1.452434	0
651	1	0.466	27	117.000000	60.0	23.0	33.8	-0.763570	0
652	5	0.269	28	123.000000	74.0	40.0	34.1	-1.313044	0
654	1	0.142	22	106.000000	70.0	28.0	34.2	-1.951928	0
655	2	0.240	25	155.000000	52.0	27.0	38.7	-1.427116	1
656	2	0.155	22	101.000000	58.0	35.0	21.8	-1.864330	0
657	1	1.162	41	120.000000	80.0	48.0	38.9	0.150143	0
659	3	1.292	27	80.000000	82.0	31.0	34.2	0.256191	1
661	1	1.394	22	199.000000	76.0	43.0	42.9	0.332177	1
662	8	0.165	43	167.000000	106.0	46.0	37.6	-1.801810	1
663	9	0.637	40	145.000000	80.0	46.0	37.9	-0.450986	1
664	6	0.245	40	115.000000	60.0	39.0	33.7	-1.406497	1
665	1	0.217	24	112.000000	80.0	45.0	34.8	-1.527858	0
666	4	0.235	70	145.000000	82.0	18.0	32.5	-1.448170	1
667	10	0.141	40	111.000000	70.0	27.0	27.5	-1.958995	1
668	6	0.430	43	98.000000	58.0	33.0	34.0	-0.843970	0
669	9	0.164	45	154.000000	78.0	30.0	30.9	-1.807889	0
670	6	0.631	49	165.000000	68.0	26.0	33.6	-0.460449	0
671	1	0.551	21	99.000000	58.0	10.0	25.4	-0.596020	0
672	10	0.285	47	68.000000	106.0	23.0	35.5	-1.255266	0
673	3	0.880	22	123.000000	100.0	35.0	57.3	-0.127833	0
679	2	0.614	23	101.000000	58.0	17.0	24.2	-0.487760	0
680	2	0.332	22	56.000000	56.0	28.0	24.2	-1.102620	0
681	0	0.364	26	162.000000	76.0	36.0	49.6	-1.010601	1
682	0	0.366	22	95.000000	64.0	39.0	44.6	-1.005122	0
685	2	0.591	25	129.000000	74.0	26.0	33.2	-0.525939	0
687	1	0.181	29	107.000000	50.0	19.0	28.3	-1.709258	0
688	1	0.828	23	140.000000	74.0	26.0	24.1	-0.188742	0
689	1	0.335	46	144.000000	82.0	46.0	46.1	-1.093625	1
692	2	0.886	23	121.000000	70.0	32.0	39.1	-0.121038	0
693	7	0.439	43	129.000000	68.0	49.0	38.5	-0.823256	1
695	7	0.128	43	142.000000	90.0	24.0	30.4	-2.055725	1
696	3	0.268	31	169.000000	74.0	19.0	29.9	-1.316768	1

698	4	0.598	28	127.000000	88.0	11.0	34.5	-0.514165	0
700	2	0.483	26	122.000000	76.0	27.0	35.9	-0.727739	0
701	6	0.565	49	125.000000	78.0	31.0	27.6	-0.570930	1
702	1	0.905	52	168.000000	88.0	29.0	35.0	-0.099820	1
704	4	0.118	27	110.000000	76.0	20.0	28.4	-2.137071	0
705	6	0.177	28	80.000000	80.0	36.0	39.8	-1.731606	0
707	2	0.176	22	127.000000	46.0	21.0	34.4	-1.737271	0
709	2	0.674	23	93.000000	64.0	32.0	38.0	-0.394525	1
710	3	0.295	24	158.000000	64.0	13.0	31.2	-1.220780	0
711	5	0.439	40	126.000000	78.0	27.0	29.6	-0.823256	0
712	10	0.441	38	129.000000	62.0	36.0	41.2	-0.818710	1
713	0	0.352	21	134.000000	58.0	20.0	26.4	-1.044124	0
715	7	0.826	34	187.000000	50.0	33.0	33.9	-0.191161	1
716	3	0.970	31	173.000000	78.0	39.0	33.8	-0.030459	1
717	10	0.595	56	94.000000	72.0	18.0	23.1	-0.519194	0
718	1	0.415	24	108.000000	60.0	46.0	35.5	-0.879477	0
719	5	0.378	52	97.000000	76.0	27.0	35.6	-0.972861	1
720	4	0.317	34	83.000000	86.0	19.0	29.3	-1.148854	0
721	1	0.289	21	114.000000	66.0	36.0	38.1	-1.241329	0
722	1	0.349	42	149.000000	68.0	29.0	29.3	-1.052683	1
723	5	0.251	42	117.000000	86.0	30.0	39.1	-1.382302	0
725	4	0.236	38	112.000000	78.0	40.0	39.4	-1.443923	0
726	1	0.496	25	116.000000	78.0	29.0	36.1	-0.701179	0
727	0	0.433	22	141.000000	84.0	26.0	32.4	-0.837018	0
730	3	0.323	34	130.000000	78.0	23.0	28.4	-1.130103	1
732	2	0.646	24	174.000000	88.0	37.0	44.5	-0.436956	1
733	2	0.426	22	106.000000	56.0	27.0	29.0	-0.853316	0
735	4	0.284	28	95.000000	60.0	32.0	35.4	-1.258781	0
736	0	0.515	21	126.000000	86.0	27.0	27.4	-0.663588	0
737	8	0.600	42	65.000000	72.0	23.0	32.0	-0.510826	0
738	2	0.453	21	99.000000	60.0	17.0	36.6	-0.791863	0
740	11	0.785	48	120.000000	80.0	37.0	42.3	-0.242072	1
741	3	0.400	26	102.000000	44.0	20.0	30.8	-0.916291	0
742	1	0.219	22	109.000000	58.0	18.0	28.5	-1.518684	0
744	13	1.174	39	153.000000	88.0	37.0	40.6	0.160417	0
745	12	0.488	46	100.000000	84.0	33.0	30.0	-0.717440	0
746	1	0.358	27	147.000000	94.0	41.0	49.3	-1.027222	1
747	1	1.096	32	81.000000	74.0	41.0	46.3	0.091667	0
748	3	0.408	36	187.000000	70.0	22.0	36.4	-0.896488	1
751	1	0.261	28	121.000000	78.0	39.0	39.0	-1.343235	0
752	3	0.223	25	108.000000	62.0	24.0	26.0	-1.500584	0
753	0	0.222	26	181.000000	88.0	44.0	43.3	-1.505078	1
754	8	0.443	45	154.000000	78.0	32.0	32.4	-0.814186	1
755	1	1.057	37	128.000000	88.0	39.0	36.5	0.055435	1
756	7	0.391	39	137.000000	90.0	41.0	32.0	-0.939048	0
760	2	0.766	22	88.000000	58.0	26.0	28.4	-0.266573	0
761	9	0.403	43	170.000000	74.0	31.0	44.0	-0.908819	1

```
763      10  0.171   63  101.000000   76.0  48.0  32.9 -1.766092        0
764       2  0.340   27  122.000000   70.0  27.0  36.8 -1.078810        0
765       5  0.245   30  121.000000   72.0  23.0  26.2 -1.406497        0
767       1  0.315   23   93.000000   70.0  31.0  30.4 -1.155183        0
```

[336]:
```
from sklearn.model_selection import train_test_split
X = data_dummy.drop(['ped',"log_ped"], axis = 1)
import statsmodels
import statsmodels.api as sm
X=sm.add_constant(X)
y = pd.DataFrame(data_dummy[['ped','log_ped']])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 1)
print("The shape of y_test is:",y_test.shape)
```

```
The shape of y_test is: (109, 2)
```

[337]:
```
#Build model using sm.OLS().fit()
linreg_logmodel_full_intercept = sm.OLS(y_train['log_ped'], X_train).fit()

# print the summary output
print(linreg_logmodel_full_intercept.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                log_ped   R-squared:                       0.073
Model:                            OLS   Adj. R-squared:                  0.057
Method:                 Least Squares   F-statistic:                     4.738
Date:                Mon, 18 Dec 2023   Prob (F-statistic):           3.85e-05
Time:                        14:35:02   Log-Likelihood:                -398.11
No. Observations:                 432   AIC:                             812.2
Df Residuals:                     424   BIC:                             844.8
Df Model:                           7
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -1.0605      0.227     -4.669      0.000      -1.507      -0.614
npreg         -0.0220      0.011     -1.916      0.056      -0.045       0.001
age            0.0086      0.004      2.232      0.026       0.001       0.016
glu           -0.0005      0.001     -0.433      0.665      -0.003       0.002
bp            -0.0044      0.003     -1.619      0.106      -0.010       0.001
skin          -0.0024      0.004     -0.607      0.544      -0.010       0.005
bmi            0.0093      0.006      1.515      0.131      -0.003       0.021
type_1         0.2986      0.077      3.894      0.000       0.148       0.449
==============================================================================
Omnibus:                        3.716   Durbin-Watson:                   1.981
```

```
Prob(Omnibus):                  0.156   Jarque-Bera (JB):                2.867
Skew:                           0.062   Prob(JB):                        0.238
Kurtosis:                       2.620   Cond. No.                     1.19e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.19e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

[338]:
```
linreg_logmodel_full_intercept_predictions = linreg_logmodel_full_intercept.
 ↪predict(X_test)
predicted_ped = np.exp(linreg_logmodel_full_intercept_predictions)

# extract the 'ped' values from the test data
actual_ped= y_test['ped']
linreg_logmodel_full_intercept_rmse = rmse(actual_ped, predicted_ped)

# calculate R-squared using rsquared
linreg_logmodel_full_intercept_rsquared = linreg_logmodel_full_intercept.
 ↪rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_full_intercept_rsquared_adj = linreg_logmodel_full_intercept.
 ↪rsquared_adj
```

[339]:
```
linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg full model with log␣
 ↪of target_intercept",
                     'RMSE':linreg_logmodel_full_intercept_rmse,
                     'R-Squared': linreg_logmodel_full_intercept_rsquared,
                     'Adj. R-Squared':␣
 ↪linreg_logmodel_full_intercept_rsquared_adj
                    })
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
 ↪ignore_index = True)
result_tabulation
```

[339]:
```
                                          Model      RMSE  R-Squared  \
0       LR model with log of target variable_PCA  0.405990   0.071001
1                                 SGD_Model_pca  0.661159   0.053627
2  Linreg full model with log of target_intercept  0.405092   0.072553

   Adj. R-Squared
0        0.060097
1        0.040267
2        0.057242
```

21 Linear Regression Model without Intercept

[340]:
```
X = data_dummy.drop(['ped',"log_ped"], axis = 1)
y = pd.DataFrame(data_dummy[['ped','log_ped']])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 1)
```

[341]:
```
#Build model using sm.OLS().fit()
linreg_logmodel_full= sm.OLS(y_train['log_ped'], X_train).fit()

# print the summary output
print(linreg_logmodel_full.summary())
```

```
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                log_ped   R-squared (uncentered):
0.681
Model:                            OLS   Adj. R-squared (uncentered):
0.676
Method:                 Least Squares   F-statistic:
129.7
Date:                Mon, 18 Dec 2023   Prob (F-statistic):
2.67e-101
Time:                        14:35:25   Log-Likelihood:
-408.94
No. Observations:                 432   AIC:
831.9
Df Residuals:                     425   BIC:
860.4
Df Model:                           7
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
npreg         -0.0205      0.012     -1.749      0.081      -0.044       0.003
age            0.0060      0.004      1.520      0.129      -0.002       0.014
glu           -0.0027      0.001     -2.496      0.013      -0.005      -0.001
bp            -0.0100      0.002     -4.073      0.000      -0.015      -0.005
skin          -0.0007      0.004     -0.180      0.857      -0.009       0.007
bmi           -0.0021      0.006     -0.361      0.718      -0.013       0.009
type_1         0.4210      0.074      5.703      0.000       0.276       0.566
==============================================================================
Omnibus:                        1.869   Durbin-Watson:                   1.978
Prob(Omnibus):                  0.393   Jarque-Bera (JB):                1.793
Skew:                           0.085   Prob(JB):                        0.408
Kurtosis:                       2.734   Cond. No.                         376.
```

```
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

The model exhibits an R-squared of 0.681 and an adjusted R-squared of 0.676, indicating a performance level of 68%. However, certain features, namely age, skin, and BMI, are found to be insignificant. To address this, feature selection techniques such as Lasso regression, k-best features, and backward elimination methods should be employed to identify and retain the most relevant features.

[342]:
```python
linreg_logmodel_full_predictions = linreg_logmodel_full.predict(X_test)
predicted_ped = np.exp(linreg_logmodel_full_predictions)

# extract the 'ped' values from the test data
actual_ped= y_test['ped']
linreg_logmodel_full_rmse = rmse(actual_ped, predicted_ped)

# calculate R-squared using rsquared
linreg_logmodel_full_rsquared = linreg_logmodel_full.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_full_rsquared_adj = linreg_logmodel_full.rsquared_adj

linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg full model with log␣
 ↪of target",
                     'RMSE':linreg_logmodel_full_rmse,
                     'R-Squared': linreg_logmodel_full_rsquared,
                     'Adj. R-Squared': linreg_logmodel_full_rsquared_adj
                    })
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
 ↪ignore_index = True)
result_tabulation
```

[342]:
```
                                         Model      RMSE  R-Squared  \
0       LR model with log of target variable_PCA  0.405990   0.071001
1                                 SGD_Model_pca  0.661159   0.053627
2  Linreg full model with log of target_intercept  0.405092   0.072553
3            Linreg full model with log of target  0.414520   0.681068

   Adj. R-Squared
0        0.060097
1        0.040267
2        0.057242
3        0.675815
```

```
[ ]:
```

```
[343]: data.head(1)
```

```
[343]:    npreg    ped  age type    glu    bp  skin   bmi    log_ped
       0      6  0.627   50    1  148.0  72.0  35.0  33.6 -0.466809
```

22 Recursive Feature Elimination (RFE):

```
[344]: from sklearn.feature_selection import RFE
       X = data_dummy.drop(['ped',"log_ped"], axis = 1)
       y = pd.DataFrame(data_dummy[['ped']])

       model = LinearRegression()
       rfe = RFE(model, n_features_to_select=4)  # Specify the number of features you␣
        ↪want to keep
       fit = rfe.fit(X, y)
       selected_features = X.columns[fit.support_].tolist()
       selected_features
```

```
[344]: ['npreg', 'age', 'bmi', 'type_1']
```

The RFE technique has identified the features npreg, age, bmi, and type. Using these selected features, we need to construct a regression model and assess both its performance and the significance of the chosen features.

```
[345]: X = data_dummy.drop(['ped',"log_ped","glu","bp","skin"], axis = 1)
       y = pd.DataFrame(data_dummy[['ped','log_ped']])
       X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
        ↪random_state = 1)
       linreg_logmodel_RFE= sm.OLS(y_train['log_ped'], X_train).fit()
       print(linreg_logmodel_RFE.summary())
```

```
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                log_ped   R-squared (uncentered):
0.656
Model:                            OLS   Adj. R-squared (uncentered):
0.653
Method:                 Least Squares   F-statistic:
204.4
Date:                Mon, 18 Dec 2023   Prob (F-statistic):
7.31e-98
Time:                        14:36:50   Log-Likelihood:
-425.02
No. Observations:                 432   AIC:
```

```
858.0
Df Residuals:                     428   BIC:
874.3
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
npreg         -0.0101      0.012     -0.846      0.398      -0.033       0.013
age           -0.0066      0.003     -1.966      0.050      -0.013   -7.93e-07
bmi           -0.0229      0.003     -8.530      0.000      -0.028      -0.018
type_1         0.4208      0.071      5.928      0.000       0.281       0.560
==============================================================================
Omnibus:                        1.483   Durbin-Watson:                   1.932
Prob(Omnibus):                  0.476   Jarque-Bera (JB):                1.402
Skew:                           0.033   Prob(JB):                        0.496
Kurtosis:                       2.729   Cond. No.                         107.
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

The model demonstrates an R-squared of 0.656 and an adjusted R-squared of 0.653, representing a performance level of 66%. However, it was observed that the feature npreg is deemed insignificant in this context.

[346]:
```
linreg_logmodel_RFE_predictions = linreg_logmodel_RFE.predict(X_test)
predicted_ped = np.exp(linreg_logmodel_RFE_predictions)

# extract the 'ped' values from the test data
actual_ped= y_test['ped']
linreg_logmodel_RFE_rmse = rmse(actual_ped, predicted_ped)

# calculate R-squared using rsquared
linreg_logmodel_RFE_rsquared = linreg_logmodel_RFE.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_RFE_rsquared_adj = linreg_logmodel_RFE.rsquared_adj

linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg model with RFE",
                     'RMSE':linreg_logmodel_RFE_rmse,
                     'R-Squared': linreg_logmodel_RFE_rsquared,
                     'Adj. R-Squared': linreg_logmodel_RFE_rsquared_adj
                   })
```

```
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
 ↪ignore_index = True)
result_tabulation
```

```
[346]:                                          Model      RMSE  R-Squared  \
0       LR model with log of target variable_PCA  0.405990   0.071001
1                                  SGD_Model_pca  0.661159   0.053627
2  Linreg full model with log of target_intercept  0.405092   0.072553
3            Linreg full model with log of target  0.414520   0.681068
4                         Linreg model with RFE  0.411422   0.656409

   Adj. R-Squared
0        0.060097
1        0.040267
2        0.057242
3        0.675815
4        0.653198
```

```
[347]: from sklearn.model_selection import cross_val_score, KFold
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression


# Create a linear regression model
model = LinearRegression()

# Define the number of folds
num_folds = 4

# Create a k-fold cross-validation object
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)

# Perform k-fold cross-validation
cross_val_scores = cross_val_score(model, X, y, cv=kf,␣
 ↪scoring='neg_mean_squared_error')
# 'cv' parameter takes the k-fold cross-validation object
# 'scoring' parameter defines the metric to evaluate (here, negative mean␣
 ↪squared error)

# Display the cross-validation scores
print("Cross-Validation Scores:", cross_val_scores)

# Calculate the mean and standard deviation of cross-validation scores
print("Mean CV Score:", np.mean(cross_val_scores))
print("Std CV Score:", np.std(cross_val_scores))
```

```
Cross-Validation Scores: [-0.21820245 -0.23192636 -0.32524801 -0.24987971]
```

```
Mean CV Score: -0.2563141330277254
Std CV Score: 0.04135379399276952
```

[348]:
```
data.head(1)
```

[348]:
```
   npreg    ped  age type    glu    bp  skin   bmi   log_ped
0      6  0.627   50    1  148.0  72.0  35.0  33.6 -0.466809
```

Similarly, we applied Lasso and k-best techniques to select the optimal features, and subsequently, regression models were employed. The tables below provide the model summaries for the Lasso and k-best features models.

23 LASSO (L1 Regularization):

[349]:
```
X = data_dummy.drop(['ped',"log_ped"], axis = 1)
import statsmodels
import statsmodels.api as sm
y = pd.DataFrame(data_dummy[['log_ped']])
```

[350]:
```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.15)  # You can experiment with different alpha values
model=lasso.fit(X, y)
selected_features = X.columns[model.coef_ [0]!= 0].tolist()
selected_features
```

[350]:
```
[]
```

[351]:
```
X = data_dummy.drop(['ped',"log_ped","npreg","type_1","age"], axis = 1)
y = pd.DataFrame(data_dummy[['ped','log_ped']])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 1)
linreg_logmodel_Lasso= sm.OLS(y_train['log_ped'], X_train).fit()
print(linreg_logmodel_Lasso.summary())
```

```
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                log_ped   R-squared (uncentered):
0.654
Model:                            OLS   Adj. R-squared (uncentered):
0.651
Method:                 Least Squares   F-statistic:
202.3
Date:                Mon, 18 Dec 2023   Prob (F-statistic):
3.15e-97
Time:                        14:37:29   Log-Likelihood:
```

```
-426.50
No. Observations:                 432   AIC:
861.0
Df Residuals:                     428   BIC:
877.3
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
glu           -0.0001      0.001     -0.109      0.914      -0.002       0.002
bp            -0.0118      0.002     -5.320      0.000      -0.016      -0.007
skin           0.0002      0.004      0.045      0.964      -0.008       0.008
bmi           -0.0008      0.006     -0.133      0.894      -0.012       0.011
==============================================================================
Omnibus:                        4.541   Durbin-Watson:                   1.951
Prob(Omnibus):                  0.103   Jarque-Bera (JB):                3.267
Skew:                           0.049   Prob(JB):                        0.195
Kurtosis:                       2.586   Cond. No.                         32.2
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[352]:
```
linreg_logmodel_Lasso_predictions = linreg_logmodel_Lasso.predict(X_test)
predicted_ped = np.exp(linreg_logmodel_RFE_predictions)

# extract the 'ped' values from the test data
actual_ped= y_test['ped']
linreg_logmodel_Lasso_rmse = rmse(actual_ped, predicted_ped)

# calculate R-squared using rsquared
linreg_logmodel_Lasso_rsquared = linreg_logmodel_Lasso.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_Lasso_rsquared_adj = linreg_logmodel_Lasso.rsquared_adj

linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg model with Lasso",
                     'RMSE':linreg_logmodel_Lasso_rmse,
                     'R-Squared': linreg_logmodel_Lasso_rsquared,
                     'Adj. R-Squared': linreg_logmodel_Lasso_rsquared_adj
                   })
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
 ↪ignore_index = True)
```

```
result_tabulation
```

[352]:
```
                                          Model      RMSE  R-Squared  \
0        LR model with log of target variable_PCA  0.405990   0.071001
1                                   SGD_Model_pca  0.661159   0.053627
2  Linreg full model with log of target_intercept  0.405092   0.072553
3             Linreg full model with log of target  0.414520   0.681068
4                          Linreg model with RFE  0.411422   0.656409
5                        Linreg model with Lasso  0.411422   0.654051

   Adj. R-Squared
0        0.060097
1        0.040267
2        0.057242
3        0.675815
4        0.653198
5        0.650817
```

[353]:
```python
from sklearn.model_selection import cross_val_score, KFold
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression


# Create a linear regression model
model = LinearRegression()

# Define the number of folds
num_folds = 4

# Create a k-fold cross-validation object
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)

# Perform k-fold cross-validation
cross_val_scores = cross_val_score(model, X, y, cv=kf,␣
 ↪scoring='neg_mean_squared_error')
# 'cv' parameter takes the k-fold cross-validation object
# 'scoring' parameter defines the metric to evaluate (here, negative mean␣
 ↪squared error)

# Display the cross-validation scores
print("Cross-Validation Scores:", cross_val_scores)

# Calculate the mean and standard deviation of cross-validation scores
print("Mean CV Score:", np.mean(cross_val_scores))
print("Std CV Score:", np.std(cross_val_scores))
```

```
Cross-Validation Scores: [-0.2377052  -0.24040804 -0.33661958 -0.23284152]
```

```
Mean CV Score: -0.2618935837578776
Std CV Score: 0.043228181752897915
```

[354]:
```
X.head(1)
```

[354]:
```
     glu    bp  skin   bmi
0  148.0  72.0  35.0  33.6
```

24 SelectKBest with f_regression:

[355]:
```
from sklearn.feature_selection import SelectKBest, f_regression
X = data_dummy.drop(['ped',"log_ped"], axis = 1)
y = pd.DataFrame(data_dummy[['ped']])
k_best = SelectKBest(score_func=f_regression, k=3)  # Specify the number of␣
  ↪features you want to keep
fit = k_best.fit(X, y)
selected_features = X.columns[fit.get_support()].tolist()
selected_features
```

[355]:
```
['glu', 'bmi', 'type_1']
```

[356]:
```
X = data_dummy.drop(['ped',"log_ped","npreg","bp","age","skin"], axis = 1)
y = pd.DataFrame(data_dummy[['ped','log_ped']])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
  ↪random_state = 1)
linreg_logmodel_Kbest= sm.OLS(y_train['log_ped'], X_train).fit()
print(linreg_logmodel_Kbest.summary())
```

```
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                log_ped   R-squared (uncentered):
0.665
Model:                            OLS   Adj. R-squared (uncentered):
0.663
Method:                 Least Squares   F-statistic:
284.0
Date:                Mon, 18 Dec 2023   Prob (F-statistic):
1.63e-101
Time:                        14:37:49   Log-Likelihood:
-419.46
No. Observations:                 432   AIC:
844.9
Df Residuals:                     429   BIC:
857.1
Df Model:                           3
Covariance Type:            nonrobust
```

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
glu           -0.0045      0.001     -4.787      0.000      -0.006      -0.003
bmi           -0.0147      0.003     -4.451      0.000      -0.021      -0.008
type_1         0.4710      0.071      6.598      0.000       0.331       0.611
==============================================================================
Omnibus:                        0.448   Durbin-Watson:                   1.956
Prob(Omnibus):                  0.799   Jarque-Bera (JB):                0.518
Skew:                           0.074   Prob(JB):                        0.772
Kurtosis:                       2.916   Cond. No.                         299.
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

From the two tables above, it is evident that the K-best features selection model outperforms the Lasso features regression model. In the K-best model, all features are significant at any level of significance, whereas in the regression model, certain features do not exhibit significance.

[357]:
```python
linreg_logmodel_Kbest_predictions = linreg_logmodel_Kbest.predict(X_test)
predicted_ped = np.exp(linreg_logmodel_RFE_predictions)

# extract the 'ped' values from the test data
actual_ped= y_test['ped']
linreg_logmodel_Kbest_rmse = rmse(actual_ped, predicted_ped)

# calculate R-squared using rsquared
linreg_logmodel_Kbest_rsquared = linreg_logmodel_Kbest.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_Kbest_rsquared_adj = linreg_logmodel_Kbest.rsquared_adj

linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg model with K-Best",
                     'RMSE':linreg_logmodel_Kbest_rmse,
                     'R-Squared': linreg_logmodel_Kbest_rsquared,
                     'Adj. R-Squared': linreg_logmodel_Kbest_rsquared_adj
                   })
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
 ↪ignore_index = True)
result_tabulation
```

[357]:
```
                                          Model      RMSE  R-Squared  \
0       LR model with log of target variable_PCA  0.405990   0.071001
1                                 SGD_Model_pca  0.661159   0.053627
```

```
2  Linreg full model with log of target_intercept  0.405092   0.072553
3            Linreg full model with log of target  0.414520   0.681068
4                           Linreg model with RFE  0.411422   0.656409
5                         Linreg model with Lasso  0.411422   0.654051
6                        Linreg model with K-Best  0.411422   0.665144

   Adj. R-Squared
0        0.060097
1        0.040267
2        0.057242
3        0.675815
4        0.653198
5        0.650817
6        0.662803
```

From the above metrics, we can observe that the linear regression model for K-best features performs better compared to other models. In this model, all features are significant. However, in the standard linear regression model, some features are found to be insignificant. Therefore, we choose the linear regression model with K-best features for analysis and deployment.

```
[ ]: # Drop the target variable 'type' from X
X = data.drop(['ped','type'], axis=1)

# Standardize the features
scale = StandardScaler().fit(X)
features = scale.transform(X)
features_scaled = pd.DataFrame(features, columns=["npreg", "age", "glu", "bp",␣
 ↪"skin", "bmi"])
db= pd.DataFrame(data['type'], columns=["type"])
scaled =features_scaled.reset_index(drop=True)
data_reset = data['type'].reset_index(drop=True)

# Concatenate along the index axis
data_5 = pd.concat([scaled, data_reset], axis=1)

# Now check for missing values
print("Missing values in 'data_pca':")
print(data_5.isnull().sum())


# Create the target variable 'y'
y =np.log(data['ped'])


# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data_5, y, test_size=0.2,␣
 ↪random_state=10)
```

```
# Ensure the indices are aligned
X_train.reset_index(drop=True, inplace=True)
y_train.reset_index(drop=True, inplace=True)
data_5.head()
```

[ ]:
```
linreg_logmodel_scaled= sm.OLS(y_train, X_train).fit()
print(linreg_logmodel_scaled.summary())
```

[ ]:
```
linreg_logmodel_scaled_predictions =linreg_logmodel_scaled.predict(X_test)
predicted_ped = np.exp(linreg_logmodel_scaled_predictions)

# extract the 'ped' values from the test data
actual_ped= y_test['ped']
linreg_logmodel_scaled_rmse = rmse(actual_ped, predicted_ped)

# calculate R-squared using rsquared
linreg_logmodel_scaled_rsquared =linreg_logmodel_scaled.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_scaled_rsquared_adj =linreg_logmodel_scaled.rsquared_adj

linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg model with Scaled␣
↪Features",
                     'RMSE':linreg_logmodel_scaled_rmse,
                     'R-Squared':linreg_logmodel_scaled_rsquared,
                     'Adj. R-Squared':linreg_logmodel_scaled_rsquared_adj
                   })
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
↪ignore_index = True)
result_tabulation
```

From the above metrics, we can observe that the linear regression model for K-best features performs better compared to other models. In this model, all features are significant. However, in the standard linear regression model, some features are found to be insignificant. Therefore, we choose the linear regression model with K-best features for analysis and deployment.

Performing k-fold cross-validation for linear regression involves splitting dataset into k folds, training the model on k-1 folds, and testing on the remaining fold. This process is repeated k times, each time using a different fold as the test set, and then the performance metrics are averaged over the k folds.

[358]:
```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression
```

```
# Create a linear regression model
model = LinearRegression()

# Define the number of folds
num_folds = 4

# Create a k-fold cross-validation object
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)

# Perform k-fold cross-validation
cross_val_scores = cross_val_score(model, X, y, cv=kf,␣
 ↪scoring='neg_mean_squared_error')
# 'cv' parameter takes the k-fold cross-validation object
# 'scoring' parameter defines the metric to evaluate (here, negative mean␣
 ↪squared error)

# Display the cross-validation scores
print("Cross-Validation Scores:", cross_val_scores)

# Calculate the mean and standard deviation of cross-validation scores
print("Mean CV Score:", np.mean(cross_val_scores))
print("Std CV Score:", np.std(cross_val_scores))
```

```
Cross-Validation Scores: [-0.21852189 -0.22947818 -0.32575763 -0.25001804]
Mean CV Score: -0.2559439342203844
Std CV Score: 0.04186261537915351
```

In this case, the mean squared error values are relatively low, indicating that the model is performing well on the cross-validated data. The standard deviation is also relatively small, suggesting consistency across folds.

This is the final model selectd under multiple linear regression this model perform better than other models beacause this models contains all features(glucose,insulin,bmi,age and tpyes) significant features at 5% level of significance also R2 and adjusted R2 is almost similar to linear regression with lesso and recursive feature selection method but in these two models some of the features are insignifacant. Also my cross-validation method,the mean squared error values are relatively low, indicating that the model is performing well on the cross-validated data. The standard deviation is also relatively small, suggesting consistency across folds.

```
[188]: # Get predictions on the training set
       predictions = linreg_logmodel_Kbest.predict(X_train)
```

```
[189]: residuals = y_train['log_ped'] - predictions
```

```
[393]: data.head()
```

```
[393]:    npreg    ped  age type    glu    bp  skin   bmi   log_ped
       0      6  0.627   50    1  148.0  72.0  35.0  33.6 -0.466809
```

```
1       1  0.351   31    0    85.0   66.0   29.0   26.6 -1.046969
2       1  0.167   21    0    89.0   66.0   23.0   28.1 -1.789761
3       0  2.288   33    1   137.0   40.0   35.0   43.1  0.827678
4       3  0.248   26    1    78.0   50.0   32.0   31.0 -1.394327
```

[400]:
```python
#data frame is data_dummy
X = data_dummy.drop(['ped',"log_ped"], axis = 1)
y = pd.DataFrame(data_dummy[['log_ped']])

# Step 1: Start with the full model
model = sm.OLS(y, X).fit()

# Step 7: Stopping criterion (e.g., p-value threshold)
while model.pvalues.max() > 0.1:
    # Step 3: Evaluate predictor significance
    # Step 4: Remove the least significant predictor
    least_significant_predictor = model.pvalues.idxmax()
    X = X.drop(least_significant_predictor, axis=1)

    # Step 5: Fit the updated model
    model = sm.OLS(y,X).fit()

# Step 8: Final model
print(model.summary())
```

```
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                log_ped   R-squared (uncentered):
0.661
Model:                            OLS   Adj. R-squared (uncentered):
0.659
Method:                 Least Squares   F-statistic:
349.9
Date:                Mon, 18 Dec 2023   Prob (F-statistic):
5.60e-126
Time:                        12:32:43   Log-Likelihood:
-523.47
No. Observations:                 541   AIC:
1053.
Df Residuals:                     538   BIC:
1066.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
```

```
glu          -0.0017      0.001     -1.819      0.069      -0.003       0.000
bp           -0.0112      0.001     -7.649      0.000      -0.014      -0.008
type_1        0.4017      0.065      6.196      0.000       0.274       0.529
==============================================================================
Omnibus:                        3.206   Durbin-Watson:                   2.004
Prob(Omnibus):                  0.201   Jarque-Bera (JB):                2.581
Skew:                           0.036   Prob(JB):                        0.275
Kurtosis:                       2.669   Cond. No.                         339.
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[401]:
```
X = data_dummy.drop(['ped',"log_ped","npreg","age","skin","bmi"], axis = 1)
y = pd.DataFrame(data_dummy[['ped','log_ped']])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 1)
linreg_logmodel_Backward= sm.OLS(y_train['log_ped'], X_train).fit()
print(linreg_logmodel_Backward.summary())
```

```
                                 OLS Regression Results
=================================================================================
=======
Dep. Variable:                log_ped   R-squared (uncentered):
0.678
Model:                            OLS   Adj. R-squared (uncentered):
0.676
Method:                 Least Squares   F-statistic:
301.4
Date:                Mon, 18 Dec 2023   Prob (F-statistic):
3.22e-105
Time:                        12:42:14   Log-Likelihood:
-410.86
No. Observations:                 432   AIC:
827.7
Df Residuals:                     429   BIC:
839.9
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
glu          -0.0025      0.001     -2.443      0.015      -0.004      -0.000
bp           -0.0100      0.002     -6.168      0.000      -0.013      -0.007
type_1        0.4051      0.071      5.678      0.000       0.265       0.545
```

```
==============================================================================
Omnibus:                        1.864   Durbin-Watson:                   1.968
Prob(Omnibus):                  0.394   Jarque-Bera (JB):                1.756
Skew:                           0.074   Prob(JB):                        0.416
Kurtosis:                       2.725   Cond. No.                         340.
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[402]:
```
linreg_logmodel_Backward_predictions = linreg_logmodel_Backward.predict(X_test)
predicted_ped = np.exp(linreg_logmodel_Backward_predictions)

# extract the 'ped' values from the test data
actual_ped= y_test['ped']
linreg_logmodel_Backward_rmse = rmse(actual_ped, predicted_ped)

# calculate R-squared using rsquared
linreg_logmodel_Backward_rsquared =linreg_logmodel_Backward.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_Backward_rsquared_adj =linreg_logmodel_Backward.rsquared_adj

linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg model with Backward␣
 ↪Selection",
                     'RMSE':linreg_logmodel_Backward_rmse,
                     'R-Squared':linreg_logmodel_Backward_rsquared,
                     'Adj. R-Squared':linreg_logmodel_Backward_rsquared_adj
                   })
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
 ↪ignore_index = True)
result_tabulation
```

[402]:
```
                                            Model      RMSE  R-Squared  \
0       LR model with log of target variable_PCA  0.382453   0.089738
1                                   SGD_Model_pca  0.626279   0.078270
2  Linreg full model with log of target_intercept  0.405092   0.072553
3             Linreg full model with log of target  0.414520   0.681068
4                           Linreg model with RFE  0.411422   0.656409
5                         Linreg model with Lasso  0.411422   0.654051
6                       Linreg model with K-Best  0.411422   0.654051
7             Linreg model with Backward Selection  0.409188   0.678217

   Adj. R-Squared
```

```
0        0.079055
1        0.056835
2        0.057242
3        0.675815
4        0.653198
5        0.650817
6        0.650817
7        0.675967
```

[40]:
```
data_4.shape
```

[40]:
```
(700, 7)
```

[44]:
```

```

```
Missing values in 'data_pca':
npreg    0
age      0
glu      0
bp       0
skin     0
bmi      0
type     0
dtype: int64
```

[44]:
```
      npreg       age       glu        bp      skin       bmi  type
0  0.749013  1.718098  0.879728  0.043756  0.558557  0.102823     1
1 -0.756247 -0.052006 -1.168442 -0.446013 -0.014657 -0.918661     0
2 -0.756247 -0.983640 -1.038400 -0.446013 -0.587871 -0.699772     0
3 -1.057299  0.134321  0.522111 -2.568348  0.558557  1.489123     1
4 -0.154143 -0.517823 -1.396017 -1.752065  0.271950 -0.276586     1
```

[45]:
```
linreg_logmodel_scaled= sm.OLS(y_train, X_train).fit()
print(linreg_logmodel_scaled.summary())
```

```
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                    ped   R-squared (uncentered):
0.239
Model:                            OLS   Adj. R-squared (uncentered):
0.226
Method:                 Least Squares   F-statistic:
19.05
Date:                Thu, 21 Dec 2023   Prob (F-statistic):
3.81e-22
Time:                        12:35:58   Log-Likelihood:
-592.65
```

```
No. Observations:                 432   AIC:
1199.
Df Residuals:                     425   BIC:
1228.
Df Model:                           7
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
npreg          0.0925      0.064      1.452      0.147      -0.033       0.218
age            0.0679      0.068      0.995      0.320      -0.066       0.202
glu            0.2577      0.054      4.787      0.000       0.152       0.364
bp            -0.0472      0.051     -0.922      0.357      -0.148       0.053
skin          -0.0400      0.066     -0.609      0.543      -0.169       0.089
bmi            0.2114      0.064      3.307      0.001       0.086       0.337
type          -1.0162      0.090    -11.230      0.000      -1.194      -0.838
==============================================================================
Omnibus:                        1.167   Durbin-Watson:                   1.356
Prob(Omnibus):                  0.558   Jarque-Bera (JB):                1.264
Skew:                           0.111   Prob(JB):                        0.531
Kurtosis:                       2.855   Cond. No.                         3.16
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not
contain a constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[ ]:
```
linreg_logmodel_scaled_predictions =linreg_logmodel_scaled.predict(X_test)
predicted_ped = np.exp(linreg_logmodel_scaled_predictions)

# extract the 'ped' values from the test data
actual_ped= y_test['ped']
linreg_logmodel_scaled_rmse = rmse(actual_ped, predicted_ped)

# calculate R-squared using rsquared
linreg_logmodel_scaled_rsquared =linreg_logmodel_scaled.rsquared

# calculate Adjusted R-Squared using rsquared_adj
linreg_logmodel_scaled_rsquared_adj =linreg_logmodel_scaled.rsquared_adj

linreg_logmodel_full_metrics = pd.Series({'Model': "Linreg model with Scaled␣
 ↪Features",
                     'RMSE':linreg_logmodel_scaled_rmse,
                     'R-Squared':linreg_logmodel_scaled_rsquared,
                     'Adj. R-Squared':linreg_logmodel_scaled_rsquared_adj
```

```
                    })
result_tabulation = result_tabulation.append(linreg_logmodel_full_metrics,␣
  ↪ignore_index = True)
result_tabulation
```

[359]:
```
# Get predictions on the training set
predictions = linreg_logmodel_Kbest.predict(X_train)
residuals = y_train['log_ped'] - predictions
```

25 Residual Analysis

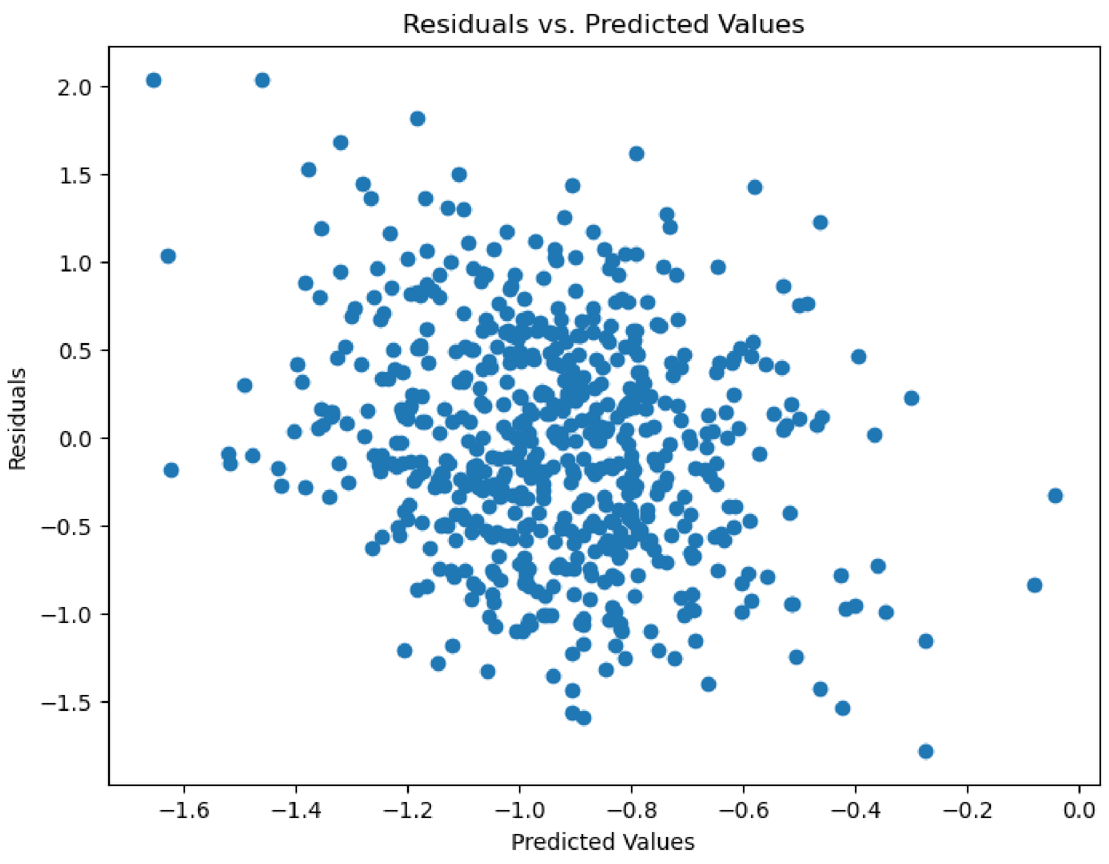
Create residual plots to visually assess the assumptions of linear regression. Common plots include the scatter plot of residuals against predicted values, a histogram of residuals, and a Q-Q plot:

26 Evaluate Homoscedasticity:

Check for homoscedasticity using a plot of residuals against predicted values. A cone-shaped pattern may indicate heteroscedasticity:

This type of plot is useful for identifying patterns or trends in the residuals. Ideally, we want to see a random scatter of points with no clear pattern, indicating that the residuals are homoscedastic.If we observe any systematic patterns (e.g., a funnel shape, curvature), it may suggest that the assumption of homoscedasticity is violated, and we might need to consider transformations or other model adjustments.

[191]:
```
plt.figure(figsize=(7, 4))
plt.scatter(predictions, residuals)
plt.title('Residuals vs. Predicted Values')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.show()
```



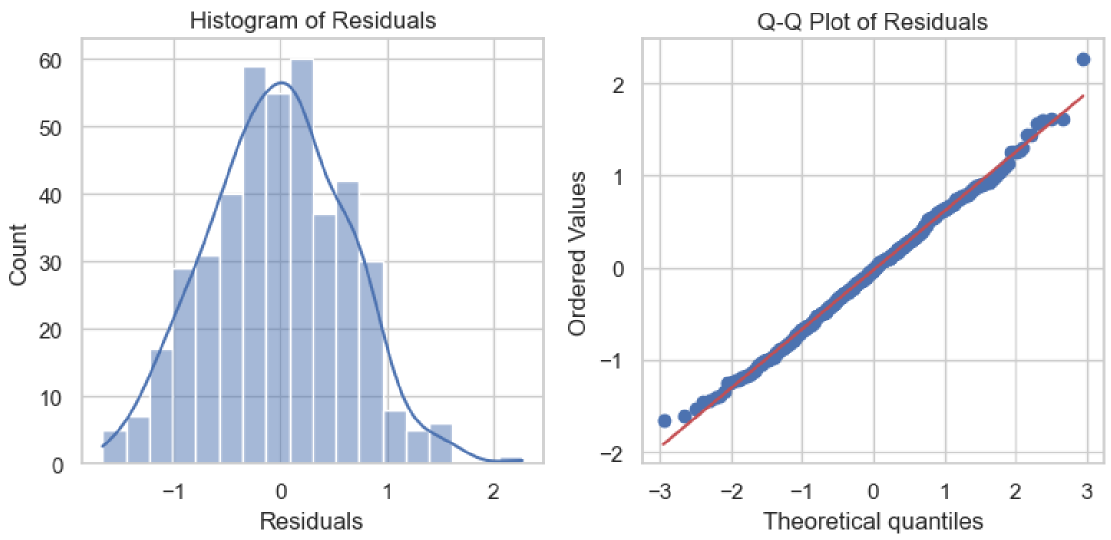
In the scatter plot above, we can observe a random scatter of points with no clear pattern, suggesting that the residuals exhibit homoscedasticity. The absence of any systematic trend in the distribution of residuals indicates that the variance of the errors remains constant across different levels of predicted values

27 Check for Normality:

Assess the normality of residuals using a histogram and a Q-Q plot:

```
[371]: plt.figure(figsize=(8, 4))

       # Histogram of residuals
       plt.subplot(1, 2, 1)
       sns.histplot(residuals, kde=True)
       plt.title('Histogram of Residuals')
       plt.xlabel('Residuals')

       # Q-Q plot
       plt.subplot(1, 2, 2)
       stats.probplot(residuals, plot=plt)
```

```
plt.title('Q-Q Plot of Residuals')

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



[369]:
```
import numpy as np
from scipy.stats import jarque_bera


# Perform Jarque-Bera test
statistic, p_value = jarque_bera(residuals)

# Display the results
print(f"Jarque-Bera statistic: {statistic}")
print(f"P-value: {p_value}")

# Check the null hypothesis
if p_value < 0.05:
    print("The Residuals are does not follows Normal distribution.")
else:
    print("The Residuals are follows Normal distribution.")
```

```
Jarque-Bera statistic: 0.5183927359564329
P-value: 0.7716714765970492
The Residuals are follows Normal distribution.
```

Based on the distribution plot, Q-Q plot, and the Jarque-Bera test, it has been confirmed that the residuals follow a normal distribution. This observation supports the assumption that the residuals are normally distributed, a key requirement in linear regression analysis.

28 Durbin-Watson Test:

The Durbin-Watson test checks for the presence of autocorrelation in the residuals. A value around 2 suggests no autocorrelation.

[195]:
```python
from statsmodels.stats.stattools import durbin_watson

# Calculate Durbin-Watson statistic
durbin_watson_stat = durbin_watson(linreg_logmodel_Kbest.resid)
print("Durbin-Watson Statistic:", durbin_watson_stat)
```

```
Durbin-Watson Statistic: 1.9485260033637735
```

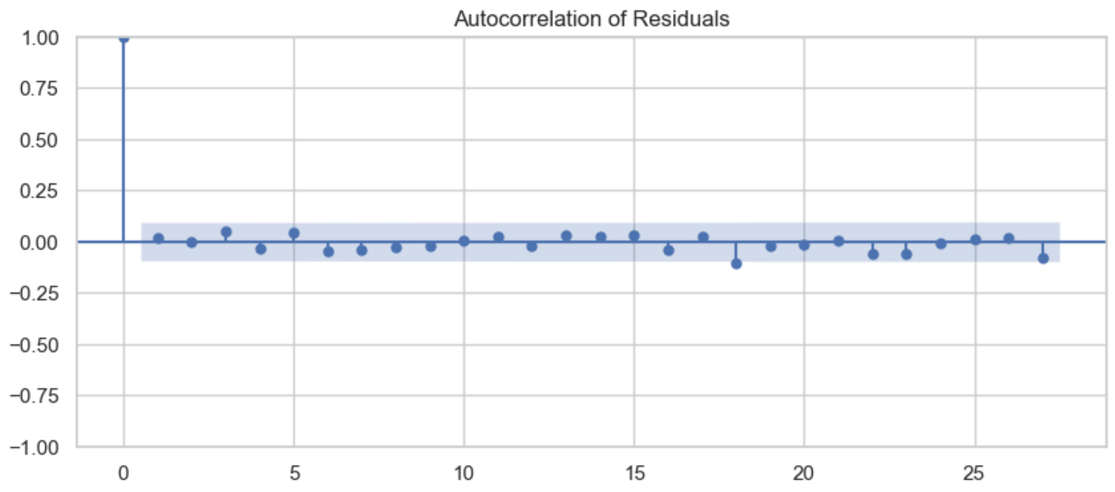
In this case, a Durbin-Watson statistic of 1.95 is very close to 2, suggesting that there is no strong evidence of significant autocorrelation in the residuals. The value being close to 2 indicates that the residuals are approximately uncorrelated with each other.

29 Autocorrelation Plot:

Visualize the autocorrelation of residuals using an autocorrelation plot:

[373]:
```python
from statsmodels.graphics.tsaplots import plot_acf

# Autocorrelation plot
plt.figure(figsize=(5, 4))
plot_acf(linreg_logmodel_Kbest.resid)
plt.title('Autocorrelation of Residuals')
plt.show()
```

```
<Figure size 500x400 with 0 Axes>
```



The Durbin-Watson test statistic is approximately equal to 2, suggesting that the autocorrelation function (ACF) of the error term is nearly zero. This indicates the absence of autocorrelation in the error term. Additionally, the Jarque-Bera (JB) test yields a p-value of 0.77. As the p-value is greater than the significance level (commonly 0.05), we accept the null hypothesis. This result implies that the residuals are normally distributed. Altogether, the Durbin-Watson test indicates no autocorrelation, and the JB test supports the normality assumption of the residuals.

30 Cluster Analysis

[84]:
```
data.head()
```

[84]:
```
   npreg    ped  age  type    glu    bp  skin   bmi
0      6  0.627   50     1  148.0  72.0  35.0  33.6
1      1  0.351   31     0   85.0  66.0  29.0  26.6
3      1  0.167   21     0   89.0  66.0  23.0  28.1
4      0  2.288   33     1  137.0  40.0  35.0  43.1
6      3  0.248   26     1   78.0  50.0  32.0  31.0
```

[85]:
```
features=data.drop(["type"],axis=1)
```

[86]:
```
scale=StandardScaler().fit(features)
features=scale.transform(features)
```

[87]:
```
features_scaled=pd.DataFrame(features,columns=data.columns[[0,1,2,4,5,6,7]])
features_scaled.head()
```

[87]:
```
      npreg       ped       age       glu        bp      skin       bmi
0  0.749013  0.352710  1.718098  0.879728  0.043756  0.558557  0.102823
1 -0.756247 -0.444246 -0.052006 -1.168442 -0.446013 -0.014657 -0.918661
2 -0.756247 -0.975549 -0.983640 -1.038400 -0.446013 -0.587871 -0.699772
3 -1.057299  5.148879  0.134321  0.522111 -2.568348  0.558557  1.489123
4 -0.154143 -0.741660 -0.517823 -1.396017 -1.752065  0.271950 -0.276586
```

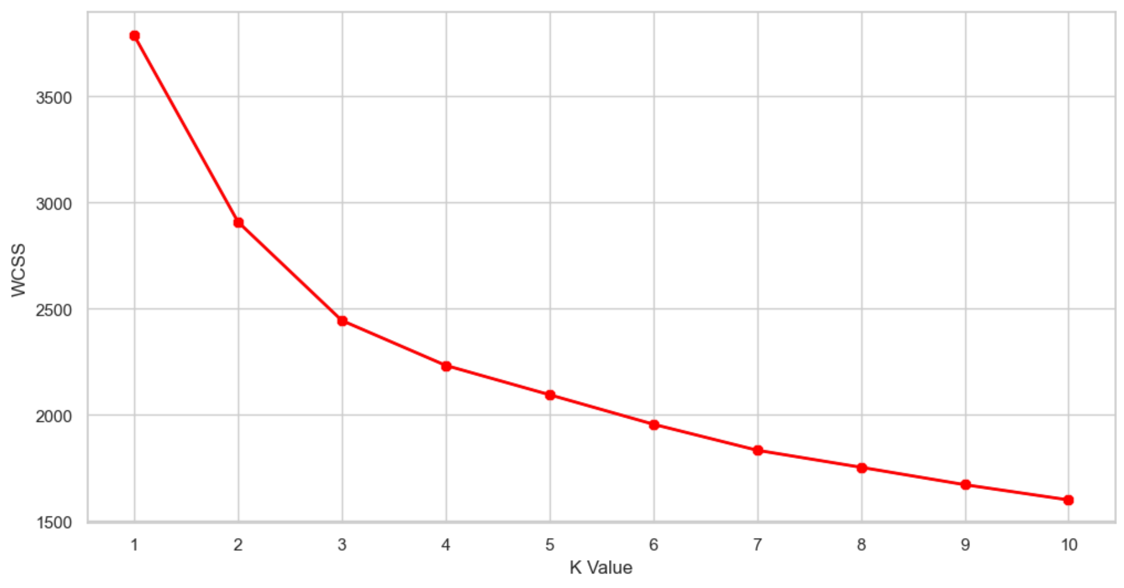
31 4.2 Build a Model with Multiple K

We constructed our models using the silhouette score method. Silhouette is a technique for interpreting and validating the consistency within clusters of data. We do not know the optimal number of clusters that would yield the most useful results. Therefore, we create clusters by varying K from 2 to 8 and subsequently determine the optimum number of clusters (K) with the assistance of the silhouette score.

[88]:
```
import warnings
warnings.filterwarnings("ignore")
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
n_clusters=[2,3,4,5,6,7,8]
for K in n_clusters:
    cluster=KMeans(n_clusters=K,random_state=10)
    predict=cluster.fit_predict(features_scaled)
    score=silhouette_score(features_scaled,predict,random_state=10)
    print("For n_clusters={}, silhoutte score is {}".format(K,score))
```

```
For n_clusters=2, silhoutte score is 0.23295225438615771
For n_clusters=3, silhoutte score is 0.21621845377939813
For n_clusters=4, silhoutte score is 0.17184136416592355
For n_clusters=5, silhoutte score is 0.17201048775549
For n_clusters=6, silhoutte score is 0.16968199563891143
For n_clusters=7, silhoutte score is 0.17378025698538754
For n_clusters=8, silhoutte score is 0.17441664982213687
```

[89]:
```
#Importing KMeans from sklearn

from sklearn.cluster import KMeans
#Now we calculate the Within Cluster Sum of Squared Errors (WSS) for different␣
 ↪values of k. Next, we
#choose the k for which WSS first starts to diminish. This value of K gives us␣
 ↪the best number of
#clusters to make from the raw data.
wcss=[]
for i in range(1,11):
    km=KMeans(n_clusters=i)
#n_clusters - The number of clusters to form as well as the number of centroids␣
 ↪to generate
    km.fit(features_scaled)
    wcss.append(km.inertia_)
#inertia_ -Sum of squared distances of samples to their closest cluster center ␣
 ↪
#The elbow curve
plt.figure(figsize=(12,6))
plt.plot(range(1,11),wcss)
plt.plot(range(1,11),wcss, linewidth=2, color="red", marker ="8")
plt.xlabel("K Value")
plt.xticks(np.arange(1,11,1))
plt.ylabel("WCSS")
plt.show()
```



The optimal value for K is identified by the highest silhouette score. From the above output, it is evident that, for K = 2, the silhouette score is the highest. Consequently, we construct the clusters with K = 2."

```
[90]: # building a K-Means model for K = 2
      model = KMeans(n_clusters= 2, random_state= 10)

      # fit the model
      model.fit(features_scaled)
```

```
[90]: KMeans(n_clusters=2, random_state=10)
```

Now, let's explore these two clusters to gain insights about them.

32 5. Retrieve the Clusters

```
[92]: data_output =features.copy()
      # add a column 'Cluster' in the data giving cluster number corresponding to␣
       ↪each observation
      data_output['Cluster'] = model.labels_
      # Reset the index, starting from 1
      data_output.index = range(1, len(data_output) + 1)

      # head() to display top five rows
      data_output.head()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
```

```
Cell In[92], line 3
      1 data_output =features.copy()
      2 # add a column 'Cluster' in the data giving cluster number correspondin␣
  ↪to each observation
----> 3 data_output['Cluster'] = model.labels_
      4 # Reset the index, starting from 1
      5 data_output.index = range(1, len(data_output) + 1)

IndexError: only integers, slices (`:`), ellipsis (`…`), numpy.newaxis (`None`)␣
  ↪and integer or boolean arrays are valid indices
```

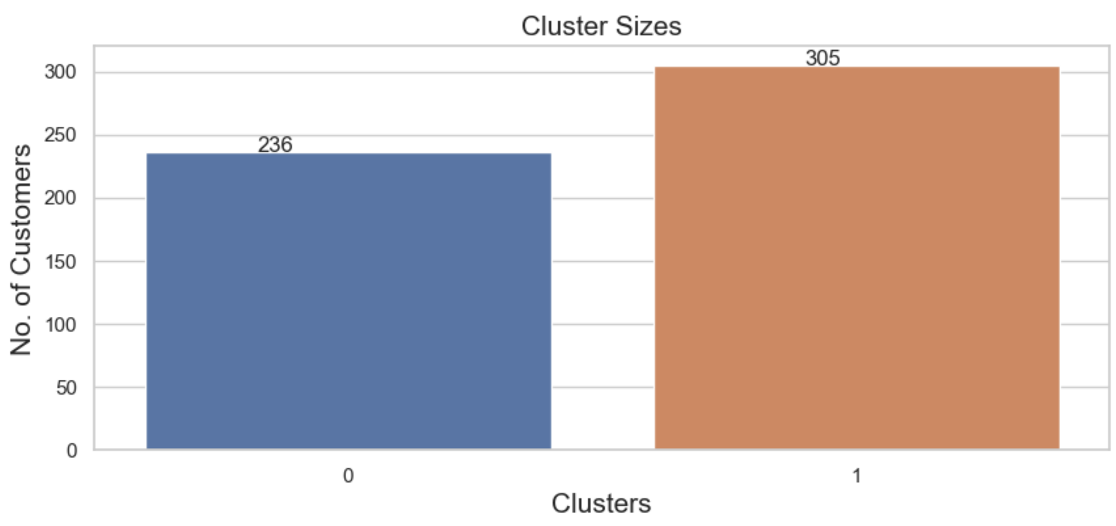
We have added a column named 'cluster' to the dataframe, indicating the cluster number for each observation.

[395]:
```
# 'return_counts = True' gives the number observation in each cluster
np.unique(model.labels_, return_counts=True)
```

[395]:
```
(array([0, 1]), array([236, 305], dtype=int64))
```

33 Plot a barplot to visualize the cluster sizes

[396]:
```
# use 'seaborn' library to plot a barplot for cluster size
sns.countplot(data= data_output, x = 'Cluster')

# set the axes and plot labels
# set the font size using 'fontsize'
plt.title('Cluster Sizes', fontsize = 15)
plt.xlabel('Clusters', fontsize = 15)
plt.ylabel('No. of Customers', fontsize = 15)

# add values in the graph
# 'x' and 'y' assigns the position to the text
# 's' represents the text on the plot
plt.text(x = -0.18, y =236, s = np.unique(model.labels_,␣
  ↪return_counts=True)[1][0])
plt.text(x = 0.9, y =305, s = np.unique(model.labels_,␣
  ↪return_counts=True)[1][1])

plt.show()
```



The second cluster is slightly larger, containing 305 customers.

34 Cluster Centers

The cluster centers can give information about the variables belonging to the clusters

[401]:
```
# form a dataframe containing cluster centers
# 'cluster_centers_' returns the co-ordinates of a cluster center
centers = pd.DataFrame(model.cluster_centers_, columns=  data_output.columns[:
 ↪7])
# head() to display top five rows
centers.head()
```

[401]:
```
      npreg       ped       age       glu        bp      skin       bmi
0  0.607417  0.157349  0.745409  0.497712  0.610658  0.526982  0.499544
1 -0.470001 -0.121752 -0.576776 -0.385115 -0.472509 -0.407763 -0.386533
```

Now, extract the variables in each of the clusters and attempt to assign a name to each cluster based on the variables

35 6 Clusters Analysis

6.1 Analysis of Cluster_1 Here, we analyze the first cluster by: Checking the size of the cluster. Sorting the variables belonging to the cluster. Computing the statistical summary for observations in the cluster.

[407]:
```
# sort the variables based on cluster centers
cluster_1 = sorted(zip(list(centers.iloc[0,:]), list(centers.columns)), reverse␣
 ↪= True)[:3]
```

[408]:
```
# size of a cluster_1
np.unique(model.labels_, return_counts=True)[1][0]
```

[408]:
```
236
```

[409]:
```
# retrieve the top 3 variables present in the cluster
cluster1_var = pd.DataFrame(cluster_1)[1]
cluster1_var
```

[409]:
```
0      age
1       bp
2    npreg
Name: 1, dtype: object
```

Here, we conduct an analysis of the first cluster, initially examining its size, followed by sorting the variables that belong to the cluster. Subsequently, we compute a statistical summary for the observations within the cluster.

Upon inspection, the first cluster comprises 236 observations. The top three variables in this cluster, ranked by importance, are age, blood pressure (bp), and number of pregnancies (npreg). This suggests that these factors play a significant role within the cluster and may warrant further investigation or attention in the context of the overall dataset.

[415]:
```
# get summary for observations in the cluster
# consider the number of orders and customer gender for cluster analysis
data_output[['age', 'bp', 'bmi', 'glu',"ped"]][data_output.Cluster == 0].
 ↪describe()
```

[415]:
```
              age          bp         bmi         glu         ped
count  236.000000  236.000000  236.000000  236.000000  236.000000
mean    39.559322   78.944915   36.318644  136.249494    0.559343
std     11.152744   10.126074    6.703330   32.529391    0.381184
min     21.000000   50.000000   19.600000   57.000000    0.085000
25%     30.000000   72.000000   32.400000  111.750000    0.263000
50%     39.000000   78.000000   35.550000  135.500000    0.447500
75%     46.000000   85.000000   39.825000  160.250000    0.728000
max     81.000000  110.000000   67.100000  199.000000    2.420000
```

36 6.2 Analysis of Cluster_2

Here, we analyze the second cluster by: Checking the size of the cluster. Sorting the variables belonging to the cluster. Computing the statistical summary for observations in the cluster.

[413]:
```
# sort the variables based on cluster centers
cluster_2 = sorted(zip(list(centers.iloc[1,:]), list(centers.columns)), reverse␣
 ↪= True)[:3]
```

```
# size of a cluster_2
np.unique(model.labels_, return_counts=True)[1][1]

# retrieve the top 10 variables present in the cluster
cluster2_var = pd.DataFrame(cluster_2)[1]
cluster2_var
```

[413]:
```
0    ped
1    glu
2    bmi
Name: 1, dtype: object
```

[414]:
```
# get summary for observations in the cluster
# consider the number of orders and customer gender for cluster analysis
data_output[['age', 'bp', 'bmi', 'glu',"ped"]][data_output.Cluster == 1].
 ↪describe()
```

[414]:
```
              age          bp         bmi         glu         ped
count  305.000000  305.000000  305.000000  305.000000  305.000000
mean    25.367213   65.675410   30.246557  109.094495    0.462685
std      4.553365   10.523918    5.718282   23.302642    0.311494
min     21.000000   24.000000   18.200000   56.000000    0.085000
25%     22.000000   60.000000   25.900000   93.000000    0.249000
50%     24.000000   66.000000   29.900000  106.000000    0.400000
75%     27.000000   72.000000   34.200000  122.000000    0.583000
max     44.000000   90.000000   55.000000  193.000000    2.288000
```

37 It can be observed that, in the second cluster, most women exhibit lower mean values for features such as ped, age, bp, bmi, and glucose compared to the first cluster. Higher values in these features are often associated with diabetes. Therefore, we may categorize the first cluster as the 'diabetes group' and the second cluster as the 'non-diabetes group,' suggesting potential differences in health characteristics between the two clusters.

[438]:
```
from sklearn.metrics import accuracy_score
import pandas as pd

# Assuming df is your DataFrame
data_1['type'] = pd.to_numeric(data_1['type'], errors='coerce')
# Replace 0 with 1 and 1 with 0
data_output.replace({0: 1, 1: 0}, inplace=True)
# Drop rows with NaN values, if any
```

```
#df = df.dropna()

# Now you can calculate accuracy
accuracy = accuracy_score(data_output['Cluster'], data_1['type'])
print("Accuracy:", accuracy)
```

```
Accuracy: 0.711645101663586
```

[439]:
```
data_output.head()
```

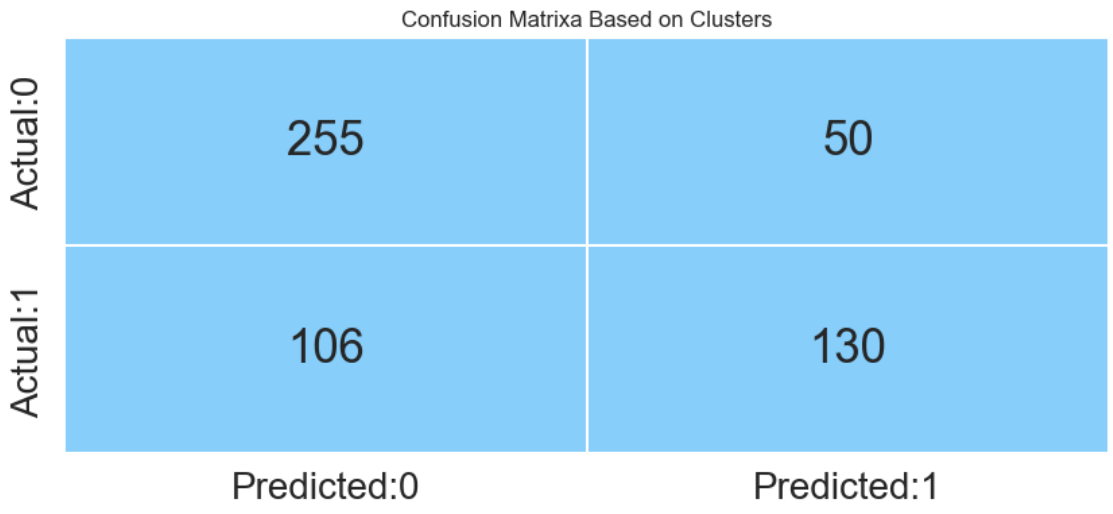
[439]:
```
   npreg    ped  age    glu    bp  skin   bmi  Cluster
1      6  0.627   50  148.0  72.0  35.0  33.6        1
2      0  0.351   31   85.0  66.0  29.0  26.6        0
3      0  0.167   21   89.0  66.0  23.0  28.1        0
4      1  2.288   33  137.0  40.0  35.0  43.1        0
5      3  0.248   26   78.0  50.0  32.0  31.0        0
```

In this data frame, '1' represents non-diabetes, and '0' represents diabetes. These labels were assigned through cluster analysis. However, we have the actual labels available, allowing us to compare them with the cluster-assigned labels and calculate the accuracy score.

[440]:
```
from sklearn.metrics import roc_auc_score
accuracy =roc_auc_score(data_output['Cluster'], data_1['type'])
print("roc_auc_score:", accuracy)
```

```
roc_auc_score: 0.6934565156988052
```

[444]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix =confusion_matrix(data_output['Cluster'], data_1['type'])
cm = confusion_matrix
conf_matrix = pd.DataFrame(data = cm,columns = ['Predicted:0','Predicted:1'],␣
 ↪index = ['Actual:0','Actual:1'])
sns.heatmap(conf_matrix, annot = True, fmt = 'd', cmap =␣
 ↪ListedColormap(['lightskyblue']),cbar = False, linewidths = 0.1, annot_kws =␣
 ↪{'size':25})
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)
plt.title("Confusion Matrixa Based on Clusters")
plt.show()
```



The accuracy score is 71.16%, suggesting that the cluster labeling is correct in approximately 71 out of 100 instances. This indicates a reasonably good performance of the cluster labeling method.

```
[ ]:
```