Question No. 1

1) Create a dataframe named df by loading dataFile.csv into Python.

```
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_style("whitegrid")
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

# Data importing
df = pd.read_csv('dataFile6_Q1.csv', sep=',')
df.head(5)
```

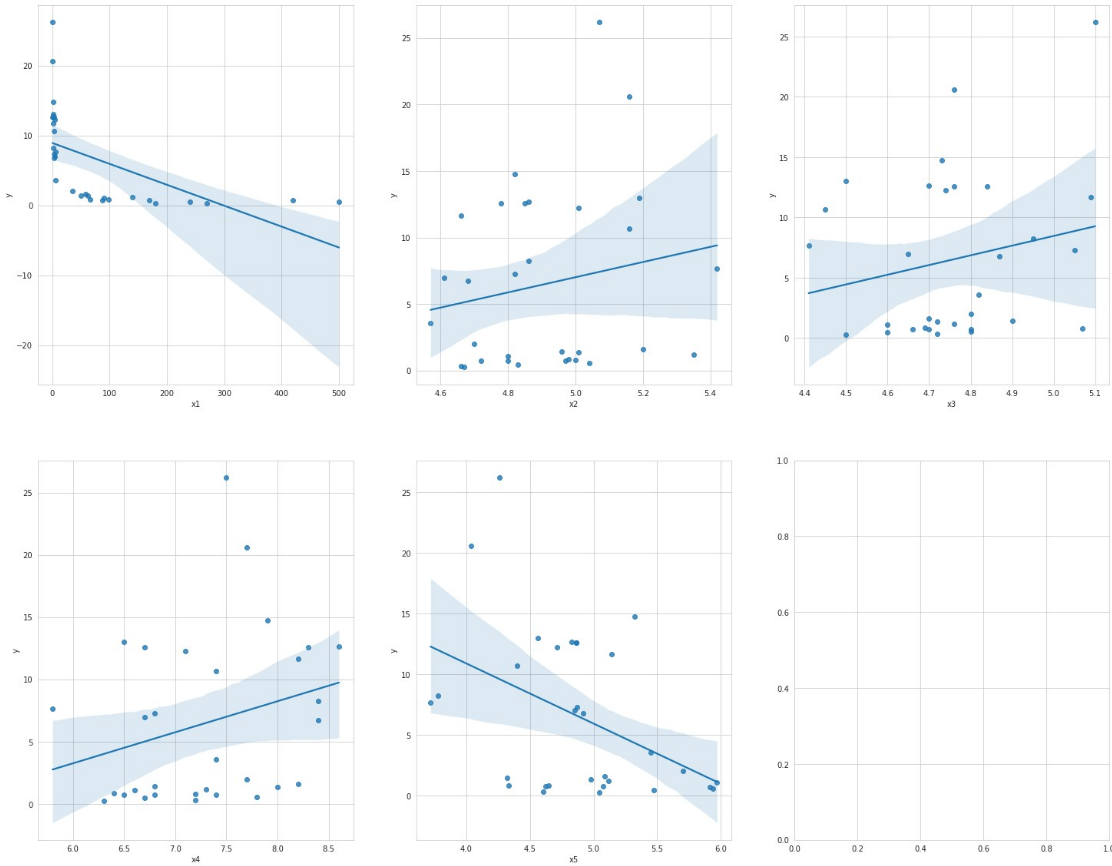
	y	x1	x2	x3	x4	x5	x6
0	6.75	2.8	4.68	4.87	8.4	4.916	-1
1	13.00	1.4	5.19	4.50	6.5	4.563	-1
2	14.75	1.4	4.82	4.73	7.9	5.321	-1
3	12.60	3.3	4.85	4.76	8.3	4.865	-1
4	8.25	1.7	4.86	4.95	8.4	3.776	-1

The above output represents first five rows of the dataset in order to have an overall observation of the dataset after importing the dataset into 'df' dataframe.

2) Examine the relationships (linear or non-linear) between five explanatory variables and the dependent variable, the change in rut depth (using appropriate graphical interpretations).

```
fig, axes = plt.subplots(2,3, figsize=(25,20))
sns.regplot(data=df, x="x1", y="y", ax=axes[0][0])
sns.regplot(data=df, x="x2", y="y", ax=axes[0][1])
sns.regplot(data=df, x="x3", y="y", ax=axes[0][2])
sns.regplot(data=df, x="x4", y="y", ax=axes[1][0])
sns.regplot(data=df, x="x5", y="y", ax=axes[1][1])
```

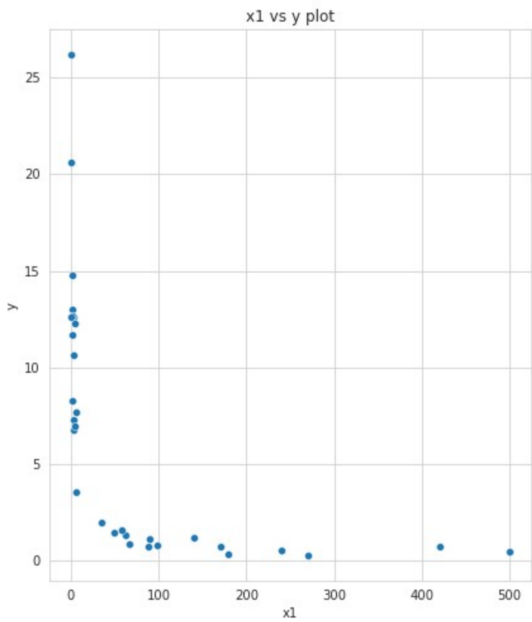
```
<AxesSubplot:xlabel='x5', ylabel='y'>
```

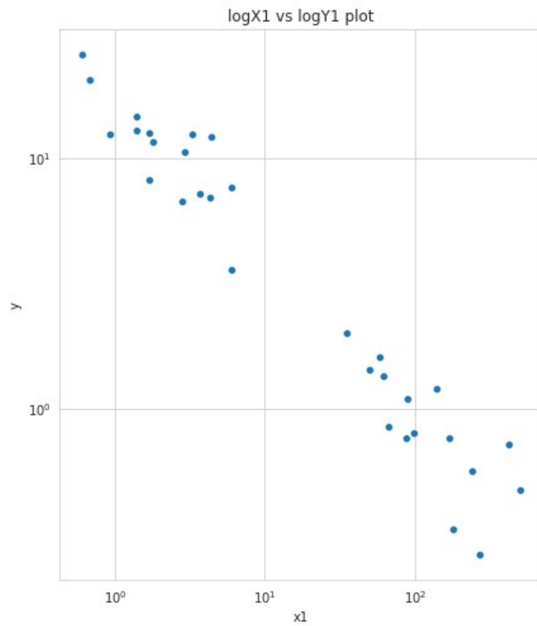


From the above plot, we can observe that the dependent variable vs the independent variables plots are not linear in nature. However, we can say that variables x2, x3, x4 have slightly linear relationship with the dependent variable y.

3) Using scatter plots, examine the relationship between the change in rut depth (y) and asphalt viscosity (x1) in their original scale (i.e. x1, y) and log scale (i.e.logx1, logy). Describe briefly the importance of transforming x1 and y into log scale.

```
f, (ax1,ax2) = plt.subplots(1,2,figsize=(15, 8))
ax2.set(xscale="log", yscale="log")
sns.scatterplot(data=df, x="x1", y="y",ax = ax1)
sns.scatterplot(data=df, x="x1", y="y",ax = ax2)

ax1.title.set_text('x1 vs y plot')
ax2.title.set_text('logX1 vs logY1 plot')
```





We can observe from the above scatter plots that by transforming the dependent and independent variables into their corresponding log scales, we are able to get strong linear relationship between the variables. This will help us with building models with higher accuracy and further perform further steps which will provide more relevant results about the dataset.

4) Create a new dataframe (df_trans) by replacing the variable x1 and y with their log transformed values. Also update the names of x and y as LogX1 and LogY, respectively.

```
df_trans = df
df_trans['logX1'] = np.log(df_trans['x1'])
df_trans['logY'] = np.log(df_trans['y'])
df_trans = df_trans.drop(columns=['x1','y'])
df_trans = df_trans.loc[:,['logY','logX1','x2', 'x3', 'x4', 'x5',
'x6']]
df_trans
```

	logY	logX1	x2	x3	x4	x5	x6
0	1.909543	1.029619	4.68	4.87	8.4	4.916	-1
1	2.564949	0.336472	5.19	4.50	6.5	4.563	-1
2	2.691243	0.336472	4.82	4.73	7.9	5.321	-1
3	2.533697	1.193922	4.85	4.76	8.3	4.865	-1
4	2.110213	0.530628	4.86	4.95	8.4	3.776	-1
5	2.367436	1.064711	5.16	4.45	7.4	4.397	-1
6	1.985131	1.308333	4.82	5.05	6.8	4.867	-1
7	2.539237	0.530628	4.86	4.70	8.6	4.828	-1
8	2.532108	-0.083382	4.78	4.84	6.7	4.865	-1
9	3.025291	-0.385662	5.16	4.76	7.7	4.034	-1
10	1.275363	1.791759	4.57	4.82	7.4	5.450	-1
11	1.945910	1.458615	4.61	4.65	6.7	4.853	-1
12	3.265759	-0.510826	5.07	5.10	7.5	4.257	-1
13	2.457021	0.587787	4.66	5.09	8.2	5.144	-1

```
14  2.037317  1.791759  5.42  4.41  5.8  3.718  -1
15  2.505526  1.481605  5.01  4.74  7.1  4.715  -1
16 -0.274437  4.477337  4.97  4.66  6.5  4.625   1
17  0.300105  4.127134  5.01  4.72  8.0  4.977   1
18  0.364643  3.912023  4.96  4.90  6.8  4.322   1
19  0.470004  4.060443  5.20  4.70  8.2  5.087   1
20  0.095310  4.499810  4.80  4.60  6.6  5.971   1
21 -0.162519  4.189655  4.98  4.69  6.4  4.647   1
22  0.182322  4.941642  5.35  4.76  7.3  5.115   1
23 -0.579818  5.480639  5.04  4.80  7.8  5.939   1
24 -0.328504  6.040255  4.80  4.80  7.4  5.916   1
25 -0.755023  6.214608  4.83  4.60  6.7  5.471   1
26 -1.108663  5.192957  4.66  4.72  7.2  4.602   1
27 -1.347074  5.598422  4.67  4.50  6.3  5.043   1
28 -0.274437  5.135798  4.72  4.70  6.8  5.075   1
29 -0.223144  4.584967  5.00  5.07  7.2  4.334   1
30  0.693147  3.555348  4.70  4.80  7.7  5.705   1
```

The above output represents the dataframe df_trans with the required updates of log scaling for varibales 'x1' and 'y'.

5) Fit the regression model given below using the statmodels package and the df_trans dataframe:

$$LogY = 0 + 1*\mathrm{LogX1} + 2*x2 + 3*x3 + 4*x4 + 5*x5 + 6*x6$$

```
import statsmodels.api as sm

#defining response variable
y = df_trans['logY']

#defining predictor variables
x = df_trans[['logX1', 'x2', 'x3', 'x4', 'x5', 'x6']]

#adding constant to predictor variables
x = sm.add_constant(x)

#fitting linear regression model
model = sm.OLS(y, x).fit()

#model parameters
print(model.params)
```

```
const   -6.090686
logX1   -0.513325
x2       1.146898
x3       0.232809
x4       0.043426
x5       0.316648
```

```
x6      -0.309446
dtype: float64
```

From the above output, we have obtained the linear regression model for the equation given in the question with the corresponding intercept and co-efficient values as displayed above.

6) Identify the explanatory variables that do have a significant impact on explaining the change in rut depth. (Use α=0.05)

Ans: A significance level of 0.05 indicates a 5% risk of concluding that an association exists when there is no actual association. If the p-value is less than or equal to the significance level (here ?=0.05), we can conclude that there is a statistically significant association between the response variable and the term.

As per the output from the above question considering the parameters having p-value < 0.05, we can identify that logX1, x2, x5, x6 are the explanatory variables that have a significant impact on explaining the change in rut depth.

7) Update your model by including the significant explanatory variables found in step 6. Examine whether the residuals follow the normal assumption.

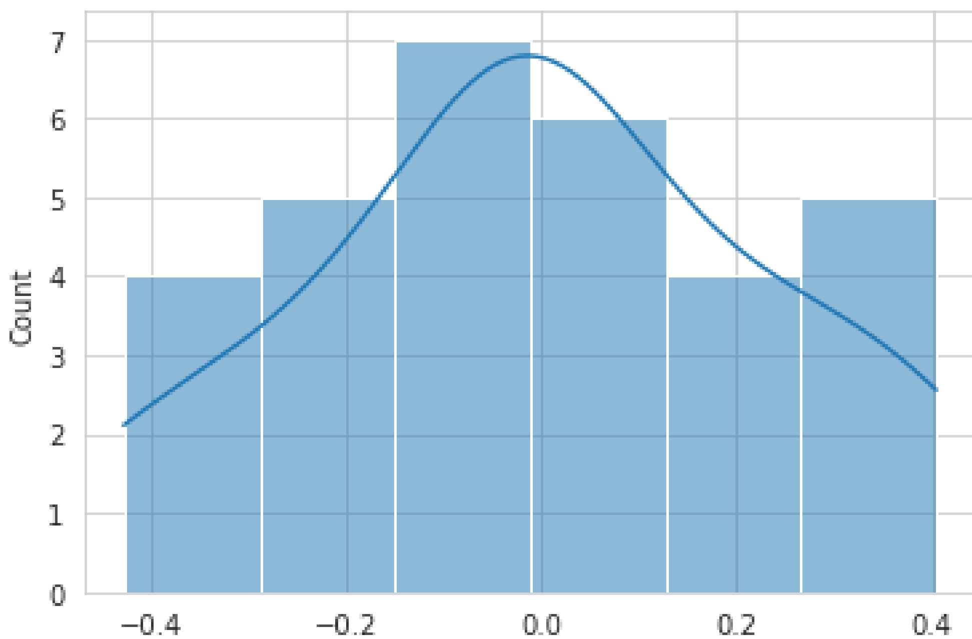
```
x = df_trans[['logX1', 'x2', 'x5', 'x6']]

#adding constant to predictor variables
x = sm.add_constant(x)

#fitting linear regression model
model = sm.OLS(y, x).fit()

#model parameters
print(model.params)
print(model.summary())
```

```
const   -4.266620
logX1   -0.547475
x2       1.070613
x5       0.330889
x6      -0.255995
dtype: float64
                            OLS Regression Results

======================================================================
========
Dep. Variable:                   logY   R-squared:
0.971
Model:                            OLS   Adj. R-squared:
0.966
Method:                 Least Squares   F-statistic:
216.0
Date:                Thu, 10 Nov 2022   Prob (F-statistic):
```

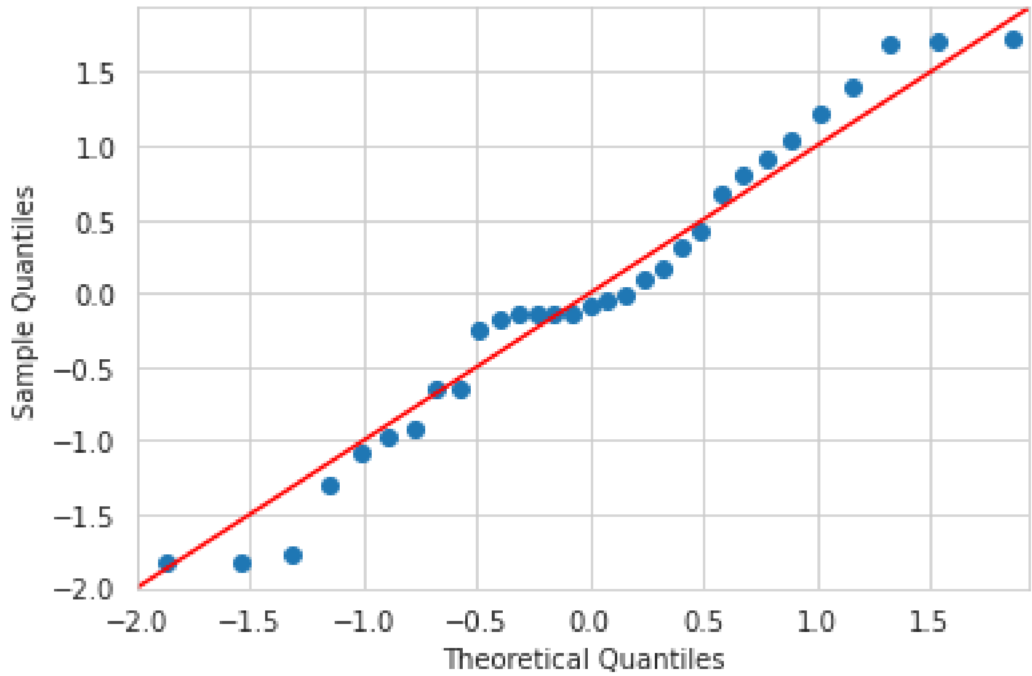
```
1.54e-19
Time:                        17:55:21   Log-Likelihood:
0.89682
No. Observations:                  31   AIC:
8.206
Df Residuals:                      26   BIC:
15.38
Df Model:                           4

Covariance Type:            nonrobust

======================================================================
========
                 coef    std err          t      P>|t|      [0.025
0.975]
----------------------------------------------------------------------
--------
const         -4.2666      1.516     -2.815      0.009      -7.382
-1.151
logX1         -0.5475      0.065     -8.363      0.000      -0.682
-0.413
x2             1.0706      0.246      4.347      0.000       0.564
1.577
x5             0.3309      0.103      3.201      0.004       0.118
0.543
x6            -0.2560      0.137     -1.872      0.073      -0.537
0.025
======================================================================
========
Omnibus:                        0.475   Durbin-Watson:
2.196
Prob(Omnibus):                  0.788   Jarque-Bera (JB):
0.584
Skew:                          -0.040   Prob(JB):
0.747
Kurtosis:                       2.333   Cond. No.
252.
======================================================================
========

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

From the above output, we have obtained the fitted updated linear regression model after dropping off the insignificant variables with the corresponding intercept and co-efficient values as displayed above. Also, we can observe from the summary statistics for the model that the p values for the independent variables is way lower than significance level and has a high R-square value. This shows the model has high correlation between variables and the independent parameters are highly contributing towards the variablity in 'y'.

```
residuals = model.resid
residMean = np.mean(residuals)
sns_plot = sns.histplot(x = residuals, kde=True)
```




```
fig = sm.qqplot(residuals, fit=True, line = '45')
```



From the above displayed histplot and QQ-plot, we can observe that the residuals follow the normal assumption as the plots show the distribution to be highly close to normal.

8) Do you think it is necessary to include an interaction term between the variables LogX1 and x6 (i.e., Logx1*x6) into the model that you created in step 4 in order to improve model performances? Please provide sufficient evidence to support your answer.

```
x7 = df.logX1*df.x6
df_trans['x7'] = x7

x = df_trans[['logX1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7']]

#adding constant to predictor variables
x = sm.add_constant(x)

#fitting linear regression model
model = sm.OLS(y, x).fit()

#model parameters
print(model.params)
print(model.pvalues)
```

```
const    -6.181829
logX1    -0.510311
x2        1.155158
x3        0.251482
x4        0.043815
x5        0.321727
x6       -0.213158
```

```
x7      -0.036952
dtype: float64
const    2.372805e-02
logX1    5.290775e-07
x2       2.791603e-04
x3       4.576294e-01
x4       5.889460e-01
x5       8.746725e-03
x6       3.557161e-01
x7       5.762777e-01
dtype: float64
```

From the above output, we have obtained the fitted updated linear regression model after including the interaction term between 'logX1' and 'x6' with the corresponding intercept and co-efficient values as displayed above. We can observe that, the p value for 'x7' (p-value=0.576) which is the new term introduced in the model as per the requirement of the question, is way above the significance level (alpha=0.05), hence not a significant variable. So, we can say that the addition of the additional term doesn't contribute significantly to improve the performance of the model obtained in step 4. Therefore, I do not think it's necessary to add this term to improve model performance.

Question No. 2

1) Load dataFile2.cvs into Python and create a dataframe named df.

```
df = pd.read_csv('dataFile6_Q2.csv', sep=',')
df.head(5)
```

```
   Unnamed: 0    x1    y1    x2    y2    x3     y3   x4    y4
0           0  10.0  8.04  10.0  9.14  10.0   7.46  8.0  6.58
1           1   8.0  6.95   8.0  8.14   8.0   6.77  8.0  5.76
2           2  13.0  7.58  13.0  8.74  13.0  12.74  8.0  7.71
3           3   9.0  8.81   9.0  8.77   9.0   7.11  8.0  8.84
4           4  11.0  8.33  11.0  9.26  11.0   7.81  8.0  8.47
```

The above output represents first five rows of the dataset in order to have an overall observation of the dataset after importing the dataset into 'df' dataframe.

2) Analyze which descriptive statistics are approximately similar across the four datasets.

```
print(df.describe())
```

```
       Unnamed: 0         x1         y1         x2         y2
x3  \
count   11.000000  11.000000  11.000000  11.000000  11.000000
11.000000
mean     5.000000   9.000000   7.500909   9.000000   7.500909
9.000000
std      3.316625   3.316625   2.031568   3.316625   2.031657
3.316625
min      0.000000   4.000000   4.260000   4.000000   3.100000
4.000000
```

```
25%       2.500000   6.500000   6.315000   6.500000   6.695000
6.500000
50%       5.000000   9.000000   7.580000   9.000000   8.140000
9.000000
75%       7.500000  11.500000   8.570000  11.500000   8.950000
11.500000
max      10.000000  14.000000  10.840000  14.000000   9.260000
14.000000

              y3         x4         y4
count  11.000000  11.000000  11.000000
mean    7.500000   9.000000   7.500909
std     2.030424   3.316625   2.030579
min     5.390000   8.000000   5.250000
25%     6.250000   8.000000   6.170000
50%     7.110000   8.000000   7.040000
75%     7.980000   8.000000   8.190000
max    12.740000  19.000000  12.500000
```

```
df.agg(
    {
          "x1": ["median", "skew"],
          "y1": ["median", "skew"],
          "x2": ["median", "skew"],
          "y2": ["median", "skew"],
          "x3": ["median", "skew"],
          "y3": ["median", "skew"],
          "x4": ["median", "skew"],
          "y4": ["median", "skew"],

    }
)
```

```
         x1        y1   x2        y2   x3        y3        x4
y4
median  9.0  7.580000  9.0  8.140000  9.0  7.110000  8.000000
7.040000
skew    0.0 -0.065036  0.0 -1.315798  0.0  1.855495  3.316625
1.506818
```

From the above output of descriptive statistics, we can conclude that the mean, median, standard deviation and range are almost same for the corresponding (x,y) values in the four dataframes.

3) Using the following Python code, fit regression models for each dataset:

a) `import statsmodels.formula.api` **as smf**

b) `smf.ols(formula` **= Y ˜1 + X, data = df).fit**

```
#Fitting model for (X1, Y1)
import statsmodels.formula.api as smf
model1=smf.ols(formula = 'y1~1 + x1', data = df).fit()
y_pred1 = model1.fittedvalues
print(model1.params)
print("R-square value for model1:", model1.rsquared)
```

```
Intercept    3.000091
x1           0.500091
dtype: float64
R-square value for model1: 0.6665424595087748
```

From the above output, we have obtained the fitted regression model with the corresponding intercept and co-efficient values as displayed above for the first dataset.

```
#Fitting model for (X2, Y2)
import statsmodels.formula.api as smf
model2=smf.ols(formula = 'y2~1 + x2', data = df).fit()
y_pred2 = model2.fittedvalues
print(model2.params)
print("R-square value for model2:", model2.rsquared)
```

```
Intercept    3.000909
x2           0.500000
dtype: float64
R-square value for model2: 0.6662420337274845
```

From the above output, we have obtained the fitted regression model with the corresponding intercept and co-efficient values as displayed above for the second dataset.

```
#Fitting model for (X3, Y3)
import statsmodels.formula.api as smf
model3=smf.ols(formula = 'y3~1 + x3', data = df).fit()
y_pred3 = model3.fittedvalues
print(model3.params)
print("R-square value for model3:", model3.rsquared)
```

```
Intercept    3.002455
x3           0.499727
dtype: float64
R-square value for model3: 0.6663240410665594
```

From the above output, we have obtained the fitted regression model with the corresponding intercept and co-efficient values as displayed above for the third dataset.

```
#Fitting model for (X4, Y4)
import statsmodels.formula.api as smf
model4=smf.ols(formula = 'y4~1 + x4', data = df).fit()
y_pred4 = model4.fittedvalues
print(model4.params)
print("R-square value for model4:", model4.rsquared)
```

```
Intercept    3.001727
x4           0.499909
dtype: float64
R-square value for model4: 0.6667072568984653
```

From the above output, we have obtained the fitted regression model with the corresponding intercept and co-efficient values as displayed above for the fourth dataset.

4) Create a dataframe named Models which includes intercept ($\beta 0$), slope ($\beta 1$), coefficient of determination ($R2$), and predicted values $(\hat{y})$ of the four models created in step 3.

```
data = {'intercept':[3.0001, 3.0009, 3.0025, 3.0017], 'slope':[
0.5001, 0.5000, 0.4997, 0.4999], 'R2':[0.667, 0.666, 0.666, 0.667],
'y_pred': [[y_pred1], [y_pred2], [y_pred3], [y_pred4]]}
models = pd.DataFrame(data)
print(models)
```

```
   intercept   slope    R2
y_pred
0     3.0001  0.5001  0.667  [[8.001, 7.000818181818181,
9.501272727272728,...
1     3.0009  0.5000  0.666  [[8.000909090909, 7.000909090909089,
9.50090...
2     3.0025  0.4997  0.666  [[7.999727272727272, 7.000272727272726,
9.4989...
3     3.0017  0.4999  0.667  [[7.0009999999999994, 7.0009999999999994,
7.00...
```

The above output for models contains the required attributes for intercept, slope, R-square and y_pred values. The values have been taken from the calculations done at each step of subquestion 3.

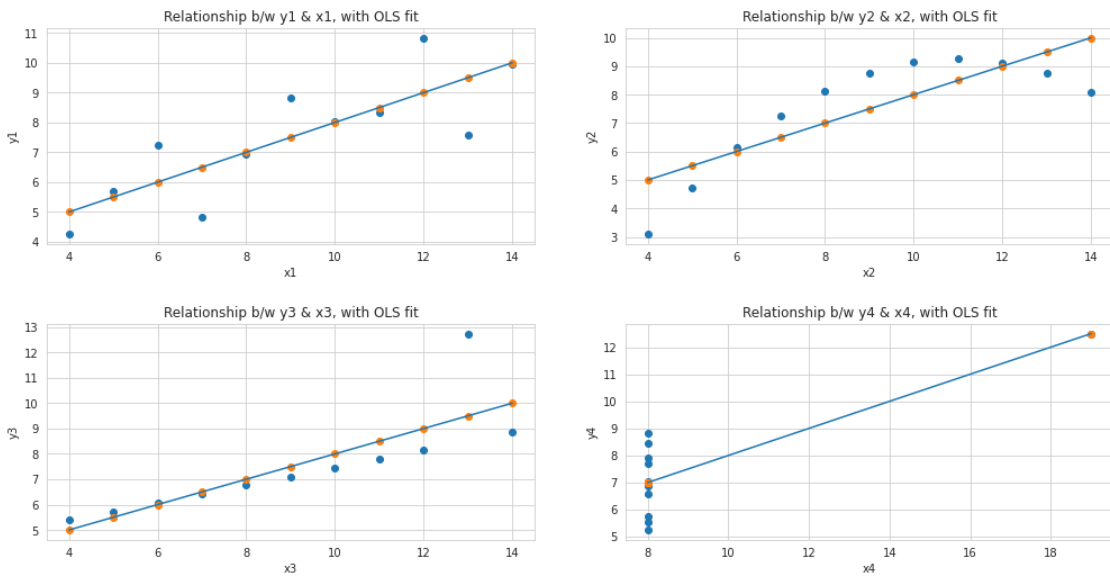
5) Show the actual data and fitted line on the same graph and display four graphs in a 2 x 2 graph.

```
import seaborn as sns
fig = plt.figure()
fig, ax = plt.subplots(2, 2, figsize=(15,8))
fig.tight_layout(pad=5)

ax[0,0].scatter(df.x1, df.y1)
x = [4, 14]
y = [model1.params.Intercept + model1.params.x1 * i for i in x]
```

```python
ax[0,0].plot(x, y)
# Plot the fitted values as "orange" dots for comparison with the
"blue" data dots
y_hat1 = model1.fittedvalues
ax[0,0].scatter(df.x1, y_hat1)
#fig.suptitle("Relationship between y1 and x1, with OLS fit")
ax[0,0].set_ylabel('y1')
ax[0,0].set_xlabel('x1')
ax[0,0].grid(True)


ax[0,1].scatter(df.x2, df.y2)
x = [4, 14]
y = [model2.params.Intercept + model2.params.x2 * i for i in x]
ax[0,1].plot(x, y)
# Plot the fitted values as "orange" dots for comparison with the
"blue" data dots
y_hat2 = model2.fittedvalues
ax[0,1].scatter(df.x2, y_hat2)
#fig.suptitle("Relationship between y2 and x2, with OLS fit")
ax[0,1].set_ylabel('y2')
ax[0,1].set_xlabel('x2')
ax[0,1].grid(True)

ax[1,0].scatter(df.x3, df.y3)
x = [4, 14]
y = [model3.params.Intercept + model3.params.x3 * i for i in x]
ax[1,0].plot(x, y)
# Plot the fitted values as "orange" dots for comparison with the
"blue" data dots
y_hat3 = model3.fittedvalues
ax[1,0].scatter(df.x3, y_hat3)
#fig.suptitle("Relationship between y3 and x3, with OLS fit")
ax[1,0].set_ylabel('y3')
ax[1,0].set_xlabel('x3')
ax[1,0].grid(True)

ax[1,1].scatter(df.x4, df.y4)
x = [8, 19]
y = [model4.params.Intercept + model4.params.x4 * i for i in x]
ax[1,1].plot(x, y)
# Plot the fitted values as "orange" dots for comparison with the
"blue" data dots
y_hat4 = model4.fittedvalues
ax[1,1].scatter(df.x4, y_hat4)
#fig.suptitle("Relationship between y4 and x4, with OLS fit")
ax[1,1].set_ylabel('y4')
ax[1,1].set_xlabel('x4')
ax[1,1].grid(True)
```

```
ax[0,0].title.set_text('Relationship b/w y1 & x1, with OLS fit')
ax[0,1].title.set_text('Relationship b/w y2 & x2, with OLS fit')
ax[1,0].title.set_text('Relationship b/w y3 & x3, with OLS fit')
ax[1,1].title.set_text('Relationship b/w y4 & x4, with OLS fit')

plt.subplots_adjust()
```

```
<Figure size 432x288 with 0 Axes>
```

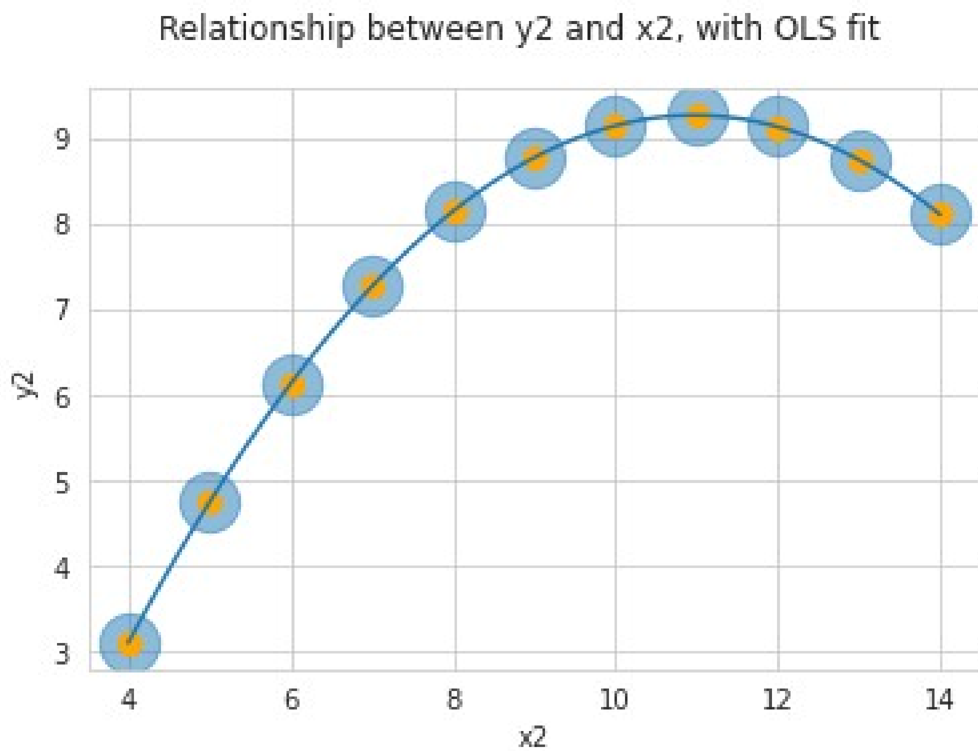


The above 2 x 2 sub-plots represent the actual data and fitted line on the same graph for the four datframes. The fitted values have been represented as "orange" dots for comparison with the "blue" data dots.

6) Consider the second dataset (i.e., {x2,y2}). Use the Python function: `np.poly1d(np.polyfit(x, y, order))` to fit a polynormial regression line that accurately describes the data. On the same graph, plot the actual data and the predicted line.

```
model_2 = np.poly1d(np.polyfit(df.x2, df.y2, 2))
fig = plt.figure()
ax_1 = fig.add_subplot(1,1,1)

ax_1.scatter(df.x2, df.y2, alpha = 0.5, s = 500)
x = np.linspace(4,14, 50)
ax_1.plot(x, model_2(x))

predict = np.poly1d(model_2)
yhat_1 = predict(df.x2)
# Plot the fitted values as "orange" dots for comparison with the
"blue" data dots
ax_1.scatter(df.x2, yhat_1, alpha = 1, s= 70, color = 'orange')
fig.suptitle("Relationship between y2 and x2, with OLS fit")
ax_1.set_ylabel('y2')
```

```
ax_1.set_xlabel('x2')
ax_1.grid(True)
```



In the above plot, the fitted values have been represented as "orange" dots for comparison with the "blue" data dots. The above plot represents the fitted polynomial regression line that accurately describes the above dataset along with the actual data and the predicted line.

7) Identify outliers in the dataset using the model fitted to the third dataset, {x3, y3}. Fit a linear model to data that is free of outliers. On the same graph, plot the actual data and the predicted line.

Outlier detection and removal using studentized residuals method

```
  - Studentized residuals is an effective way of detecting outliers
using the regression model. It removes observations one at a
time,refitting the model each time using the remaining n-1
observations. Then, using models with the ith observation deleted, we
compare the observed response values to their fitted values which
produces deleted residuals. Standardizing the deleted residuals
produces studentized residuals.

  - The command outlier_test() gives the values of the studentized
residuals for each observation

  - If an observation has a studentized residual that is larger than 3
(in absolute value) we can call it an outlier.
```

```
from statsmodels.formula.api import ols
stud_res = model3.outlier_test()
print(stud_res)
```

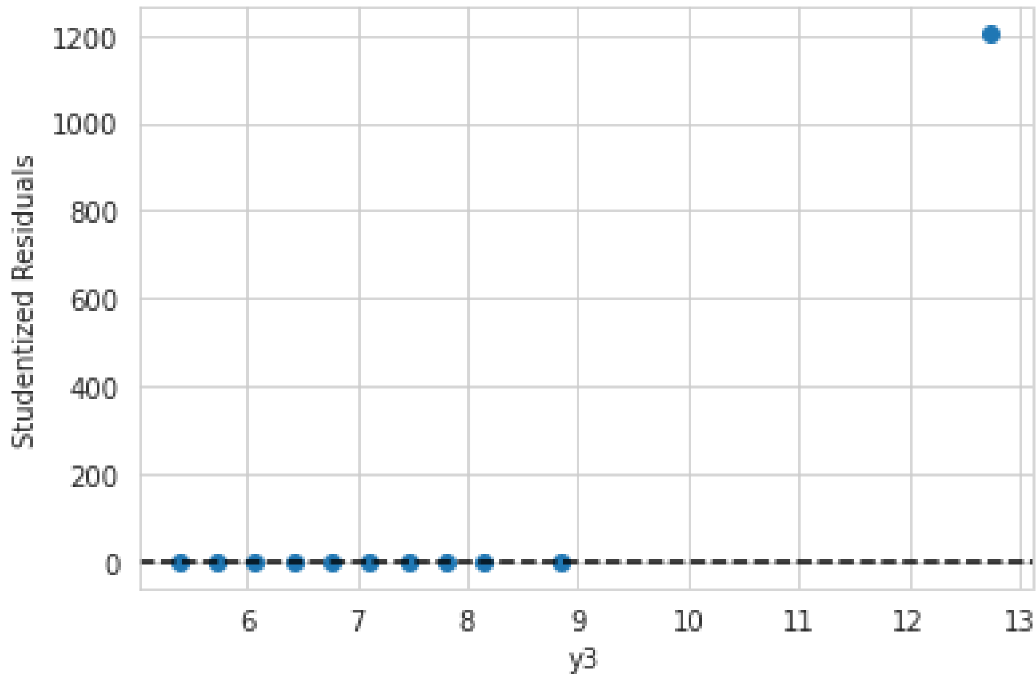
```
    student_resid       unadj_p       bonf(p)
0       -0.439055  6.722383e-01  1.000000e+00
1       -0.185502  8.574521e-01  1.000000e+00
2     1203.539464  2.544056e-22  2.798462e-21
3       -0.313844  7.616669e-01  1.000000e+00
4       -0.574295  5.815524e-01  1.000000e+00
5       -1.155982  2.810421e-01  1.000000e+00
6        0.066407  9.486831e-01  1.000000e+00
7        0.361851  7.268345e-01  1.000000e+00
8       -0.735677  4.829362e-01  1.000000e+00
9       -0.065768  9.491763e-01  1.000000e+00
10       0.200263  8.462718e-01  1.000000e+00
```

The above dataframe stud_res gives the student residual values for the model3.

Define predictor variable values and studentized residuals

```
x = df['y3']
y = stud_res['student_resid']

#create scatterplot of predictor variable vs. studentized residuals
plt.scatter(x, y)
plt.axhline(y=0, color='black', linestyle='--')
plt.xlabel('y3')
plt.ylabel('Studentized Residuals')
```

```
Text(0, 0.5, 'Studentized Residuals')
```



From the above plot of studentized residuals vs y3, we can see that we have one outlier point.

Condition for finding outlier using residuals method

```
stud_resid_3 = model3.outlier_test()
# print(df.y3, stud_resid_3)
df_resid3 = pd.concat([df.x3, df.y3, stud_resid_3.student_resid],
axis=1)
df_resid3_outlier = df_resid3.query("student_resid > 3")
print(df_resid3_outlier)
df3 = df[['x3','y3']].copy()
```

```
     x3     y3  student_resid
2  13.0  12.74    1203.539464
```

The above data point is the outlier for the above dataset in consideration.

Creating dataframe without outlier

```
df3_drop = df3.drop(df_resid3_outlier.index[0])
```

Finding outlier free model

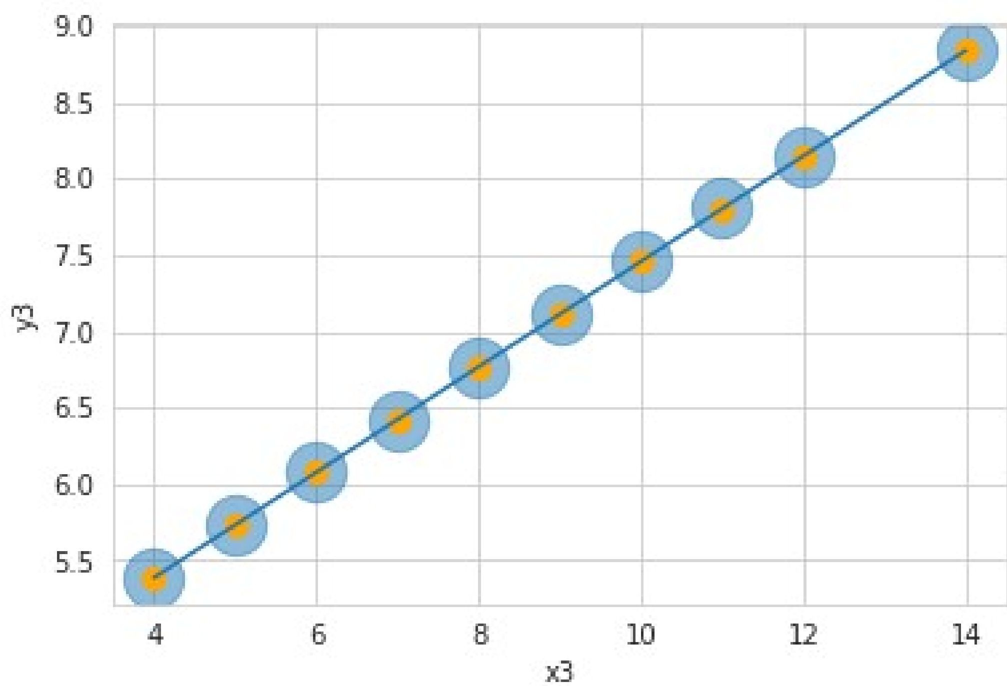
```
model_new2 = smf.ols(formula = 'y3~1 + x3', data = df3_drop).fit()
print(model_new2.params)
```

```
Intercept    4.005649
x3           0.345390
dtype: float64
```

From the above output, we have obtained the fitted regression model for the outlier-free dataframe with the corresponding intercept and co-efficient values as obtained from the regression model equation.

Plotting regression fit line with the outlier free data.

```
fig = plt.figure()
ax_3 = fig.add_subplot(1,1,1)

ax_3.scatter(df3_drop.x3, df3_drop.y3, alpha = 0.5, s = 500)
x = [4, 14]
y = [model_new2.params.Intercept + model_new2.params.x3 * i for i in
x]
ax_3.plot(x, y)
y_pred_new2 = model_new2.fittedvalues
ax_3.scatter(df3_drop.x3, y_pred_new2, alpha = 1, s= 70, color =
'orange')
#fig.suptitle("Relationship between y1 and x1, with OLS fit")
ax_3.set_ylabel('y3')
ax_3.set_xlabel('x3')
ax_3.grid(True)
```



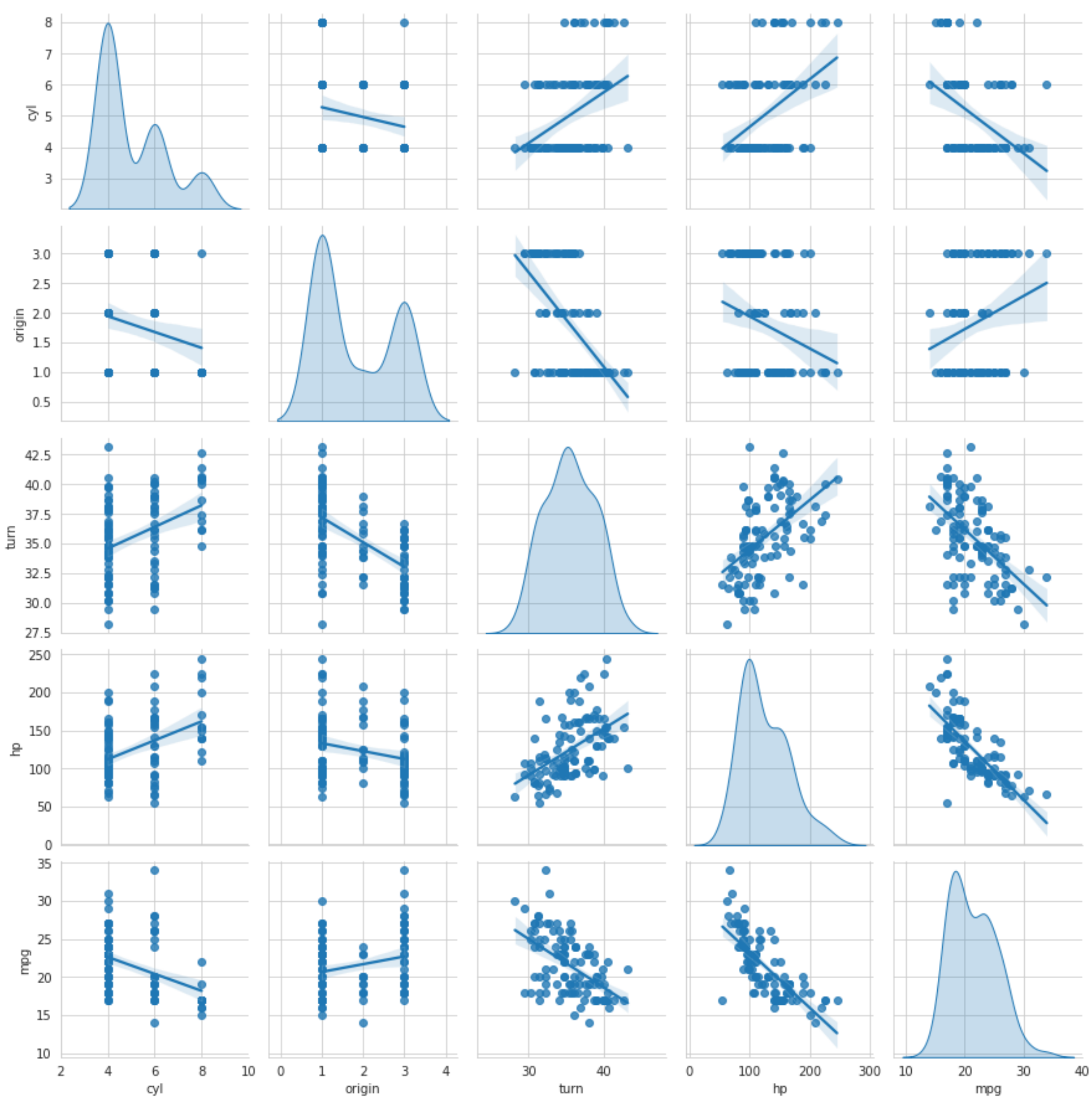
In the above plot, the fitted values have been represented as "orange" dots for comparison with the "blue" data dots. The above graph shows the linear regression model plot for outlier free data along with the actual data and the predicted line.

Question No.3

1) Explore the relationships between these five variables using the pairplot function available in the seaborn package.

```
df = pd.read_csv('dataFile6_Q3.csv', sep=',')
df = df.drop('Unnamed: 0', axis=1)
sns.pairplot(df, palette='husl',diag_kind='kde', kind='reg')
```

```
<seaborn.axisgrid.PairGrid at 0x7fdd13234460>
```



Pair plots are useful to plot multiple bivariate distributions in a dataset. The univariate plots are displayed along the diagonal and the rest of the plots give the relationships between pairwise combination of variables in the datset.

From the univariate plots, we can conclude that the variable cyl, origin, hp and mpg follow multimodal distributions since there is more than one peak whereas the variable turn has a normal distribution.

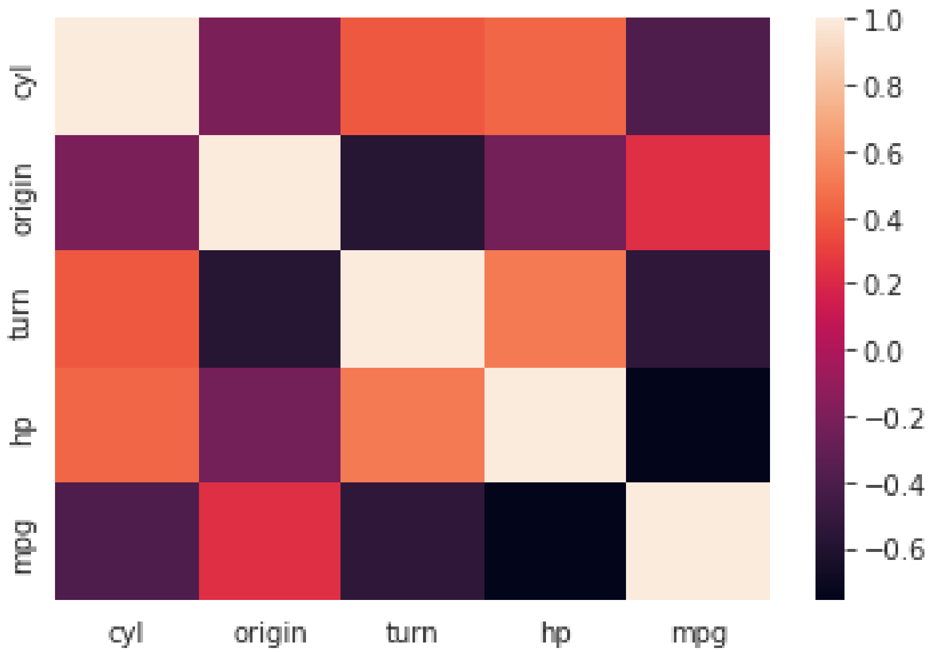
On observing the bivariate plots, it is clear that most of the relationships are non linear in nature. The target variable mpg exhibits a negative linear relationship with hp and turn. There seems to be a positive linear relationship between the predictor variables hp and turn.

2) Investigate the relationships (linear or non-linear) between five variables (you may use a heatmap to illustrate the relationships).

```
print(df.corr(method='pearson'))
sns.heatmap(df.corr())
```

```
             cyl    origin      turn        hp       mpg
cyl     1.000000 -0.204239  0.384575  0.437581 -0.398069
origin -0.204239  1.000000 -0.575194 -0.237388  0.237509
turn    0.384575 -0.575194  1.000000  0.507610 -0.541061
hp      0.437581 -0.237388  0.507610  1.000000 -0.754716
mpg    -0.398069  0.237509 -0.541061 -0.754716  1.000000
```

```
<AxesSubplot:>
```



Summary:

The Pearson correlation coefficient is the most commonly used correlation method; however, it is only sensitive to linear correlations, while several other methods tend to be more robust for non-linear correlations.

(Pearson's) Correlation coefficient, r, measures the strength of a linear relationship between two nu- merical variables. near zero means no/weak linear relationship. near ±1 zero means strong linear relationship.

The above heatmap shows the correlation between origin and cyl have a close to zero(-.20) correlation which indicates that weak linear relation or non-linear relation.

Whereas highest correlation we can find is from the turn vs hp (0.51) which specifies some degree of linear relation.

Similarly, hp vs origin -0.23 - have a weak negative linear relationship --> non-linear relation.

3) Suppose you want to explore variability in mgp in response to change in cyl, origin, turn and hp by using the linear model: $mgp = \beta 0 + \beta 1 * cyl + \beta 2 * origin + \beta 3 * turn + \beta 4 * hp$**. Use statmodels package and the df dataframe to find the model parameters.**

```
y = df['mpg']

#defining predictor variables
x = df[['cyl', 'origin', 'turn', 'hp']]

#adding constant to predictor variables
x = sm.add_constant(x)

#fitting linear regression model
model = sm.OLS(y, x).fit()

#model parameters
print(model.params)
print(model.summary())
```

```
const     40.179644
cyl       -0.120193
origin    -0.246701
turn      -0.281646
hp        -0.061290
dtype: float64
                            OLS Regression Results

======================================================================
========
Dep. Variable:                    mpg   R-squared:
0.607
Model:                            OLS   Adj. R-squared:
```

```
0.592
Method:                 Least Squares   F-statistic:
40.13
Date:                Thu, 10 Nov 2022   Prob (F-statistic):
2.69e-20
Time:                        17:55:38   Log-Likelihood:
-252.11
No. Observations:                 109   AIC:
514.2
Df Residuals:                     104   BIC:
527.7
Df Model:                           4

Covariance Type:            nonrobust

======================================================================
========
                 coef    std err          t      P>|t|      [0.025
0.975]
----------------------------------------------------------------------
--------
const         40.1796      3.621     11.095      0.000      32.998
47.361
cyl           -0.1202      0.195     -0.616      0.539      -0.507
0.267
origin        -0.2467      0.322     -0.767      0.445      -0.885
0.391
turn          -0.2816      0.102     -2.765      0.007      -0.484
-0.080
hp            -0.0613      0.007     -8.364      0.000      -0.076
-0.047
======================================================================
========
Omnibus:                       10.204   Durbin-Watson:
2.023
Prob(Omnibus):                  0.006   Jarque-Bera (JB):
22.910
Skew:                          -0.196   Prob(JB):
1.06e-05
Kurtosis:                       5.212   Cond. No.
2.05e+03
======================================================================
========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 2.05e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

From the above output, we have obtained the fitted linear regression model by using the model given in the question, with the corresponding intercept and co-efficient values as displayed above. The required parameters are printed above in the summary table.

4) What are the most significant variables that contribute to the variability in miles per gallon in city driving? Print model summary excluding variables that are not significant from your model.

```
print(model.summary())

                            OLS Regression Results

======================================================================
========
Dep. Variable:                    mpg   R-squared:
0.607
Model:                            OLS   Adj. R-squared:
0.592
Method:                 Least Squares   F-statistic:
40.13
Date:                Thu, 10 Nov 2022   Prob (F-statistic):
2.69e-20
Time:                        17:55:39   Log-Likelihood:
-252.11
No. Observations:                 109   AIC:
514.2
Df Residuals:                     104   BIC:
527.7
Df Model:                           4

Covariance Type:            nonrobust

======================================================================
========
                 coef    std err          t      P>|t|      [0.025
0.975]
----------------------------------------------------------------------
--------
const         40.1796      3.621     11.095      0.000      32.998
47.361
cyl           -0.1202      0.195     -0.616      0.539      -0.507
0.267
origin        -0.2467      0.322     -0.767      0.445      -0.885
0.391
turn          -0.2816      0.102     -2.765      0.007      -0.484
-0.080
hp            -0.0613      0.007     -8.364      0.000      -0.076
-0.047
======================================================================
========
```

```
Omnibus:                       10.204   Durbin-Watson:
2.023
Prob(Omnibus):                  0.006   Jarque-Bera (JB):
22.910
Skew:                          -0.196   Prob(JB):
1.06e-05
Kurtosis:                       5.212   Cond. No.
2.05e+03
======================================================================
========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 2.05e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

From the above summary table, considering significance level as 0.05, from the p-values we can determine that 'turn' and 'hp' contribute significantly towards the variability in miles per gallon in city driving.

```
#Printing model summary with only significant attributes
y = df['mpg']

#defining predictor variables
x = df[['turn', 'hp']]

#adding constant to predictor variables
x = sm.add_constant(x)

#fitting linear regression model
model = sm.OLS(y, x).fit()

#model parameters
print(model.params)
print("=================================================================
===============")

print("R Squared value for the model_1:",model.rsquared)
print(model)

print("=================================================================
===============")
#Display model summary
print(model.summary())
```

```
const    38.264154
turn     -0.251008
hp       -0.063076
```

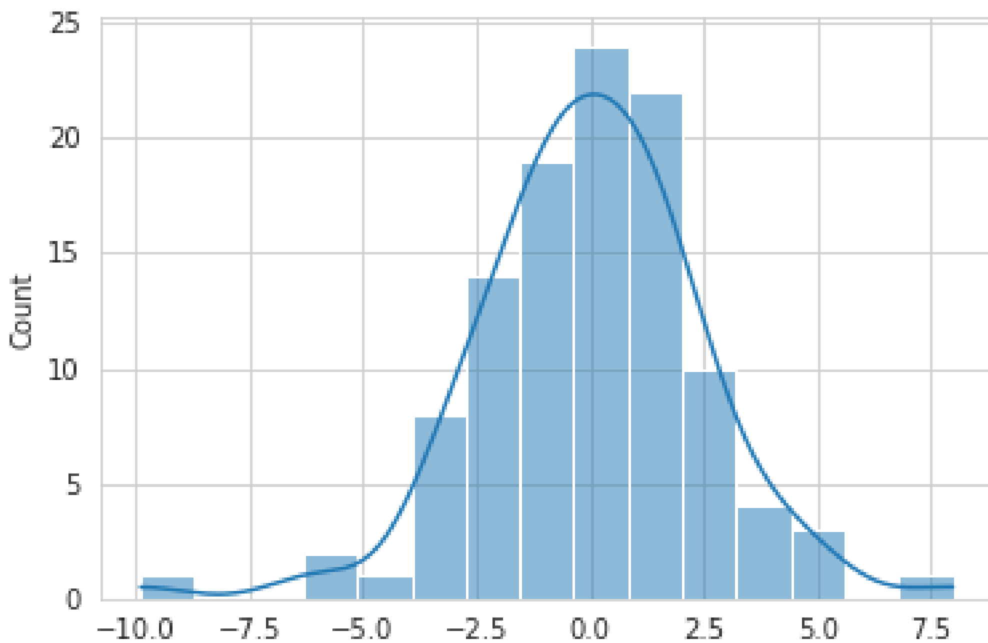
```
dtype: float64
==============================================================================
========
R Squared value for the model_1: 0.6032077228231238
<statsmodels.regression.linear_model.RegressionResultsWrapper object
at 0x7fdd13a3baf0>
==============================================================================
========
                            OLS Regression Results

==============================================================================
========
Dep. Variable:                    mpg   R-squared:
0.603
Model:                            OLS   Adj. R-squared:
0.596
Method:                 Least Squares   F-statistic:
80.57
Date:                Thu, 10 Nov 2022   Prob (F-statistic):
5.29e-22
Time:                        17:55:39   Log-Likelihood:
-252.61
No. Observations:                 109   AIC:
511.2
Df Residuals:                     106   BIC:
519.3
Df Model:                           2

Covariance Type:            nonrobust

==============================================================================
========
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
--------
const         38.2642      2.654     14.417      0.000      33.002
43.526
turn          -0.2510      0.084     -2.997      0.003      -0.417
-0.085
hp            -0.0631      0.007     -9.107      0.000      -0.077
-0.049
==============================================================================
========
Omnibus:                       12.000   Durbin-Watson:
2.021
Prob(Omnibus):                  0.002   Jarque-Bera (JB):
25.976
Skew:                          -0.335   Prob(JB):
2.29e-06
```

```
Kurtosis:                       5.296   Cond. No.
1.51e+03
======================================================================
========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 1.51e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

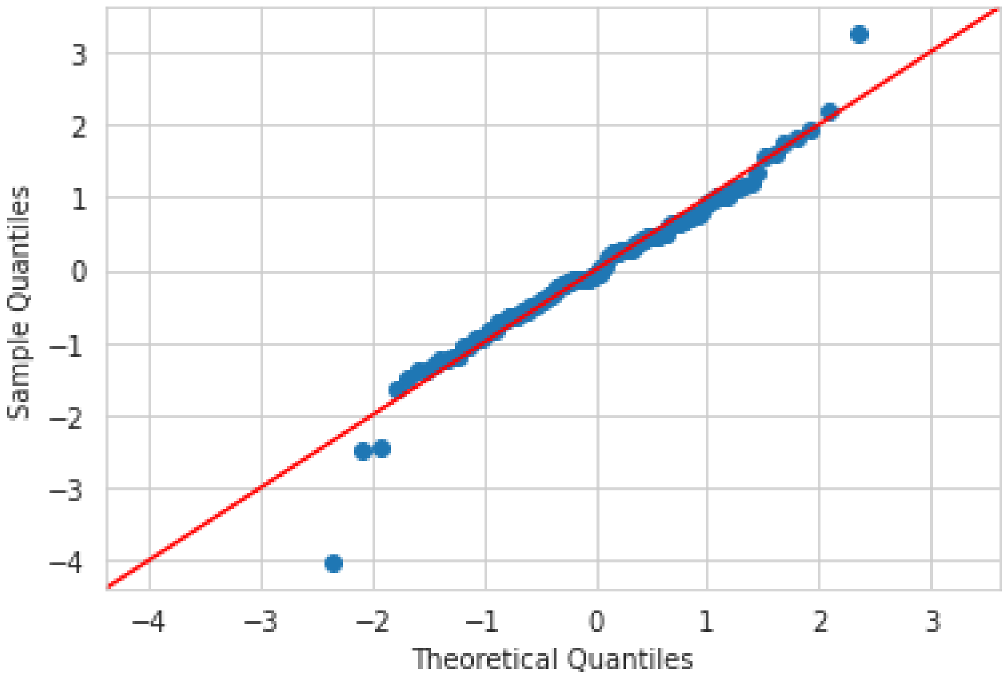
The above is the summary table for the updated model after dropping the insignificant variables.

5) Utilize appropriate graphical illustrations to examine the normality of residuals.

```
residuals = model.resid
residMean = np.mean(residuals)
sns_plot = sns.histplot(x = residuals, kde=True)
```




```
fig = sm.qqplot(residuals, fit=True, line = '45')
```



From the above displayed histplot and QQ-plot, we can observe that the residuals follow the normal assumption as the plots show the distribution to be highly close to normal.

6) After loading the mistat package, use the following codes to perform stepwise (or forward) regression.

a) outcome = 'mpg', all_vars = ['cyl', 'origin', 'turn', 'hp']

b) included, model = mistat.stepwise_regression(outcome, all_vars, df)

```
import mistat
outcome = 'mpg'; all_vars = ['cyl', 'origin', 'turn', 'hp']
included, model = mistat.stepwise_regression(outcome, all_vars, df)

Step 1 add - (F: 141.60)  hp
Step 2 add - (F: 8.98)  hp turn

formula = ' + '.join(included)
formula = f'{outcome} ~ 1 + {formula}'
print()
print('Final model')
print(formula)
print(model.summary())


Final model
mpg ~ 1 + hp + turn
                            OLS Regression Results

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

```
========
Dep. Variable:                    mpg   R-squared:
0.603
Model:                            OLS   Adj. R-squared:
0.596
Method:                 Least Squares   F-statistic:
80.57
Date:                Thu, 10 Nov 2022   Prob (F-statistic):
5.29e-22
Time:                        17:55:40   Log-Likelihood:
-252.61
No. Observations:                 109   AIC:
511.2
Df Residuals:                     106   BIC:
519.3
Df Model:                           2

Covariance Type:            nonrobust

======================================================================
========
                 coef    std err          t      P>|t|      [0.025
0.975]
----------------------------------------------------------------------
--------
Intercept     38.2642      2.654     14.417      0.000      33.002
43.526
hp            -0.0631      0.007     -9.107      0.000      -0.077
-0.049
turn          -0.2510      0.084     -2.997      0.003      -0.417
-0.085
======================================================================
========
Omnibus:                       12.000   Durbin-Watson:
2.021
Prob(Omnibus):                  0.002   Jarque-Bera (JB):
25.976
Skew:                          -0.335   Prob(JB):
2.29e-06
Kurtosis:                       5.296   Cond. No.
1.51e+03
======================================================================
========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 1.51e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

The output displayed output demonstrates the stepwise regression performed and the equation used. Also, the summary table gives description of the various performance parameters(f-statistic, R-square, p-values, t-values etc) obtained from the step-wise regression fitted model.

7) Do you see a difference between the model performance obtained in step 4 and 5?

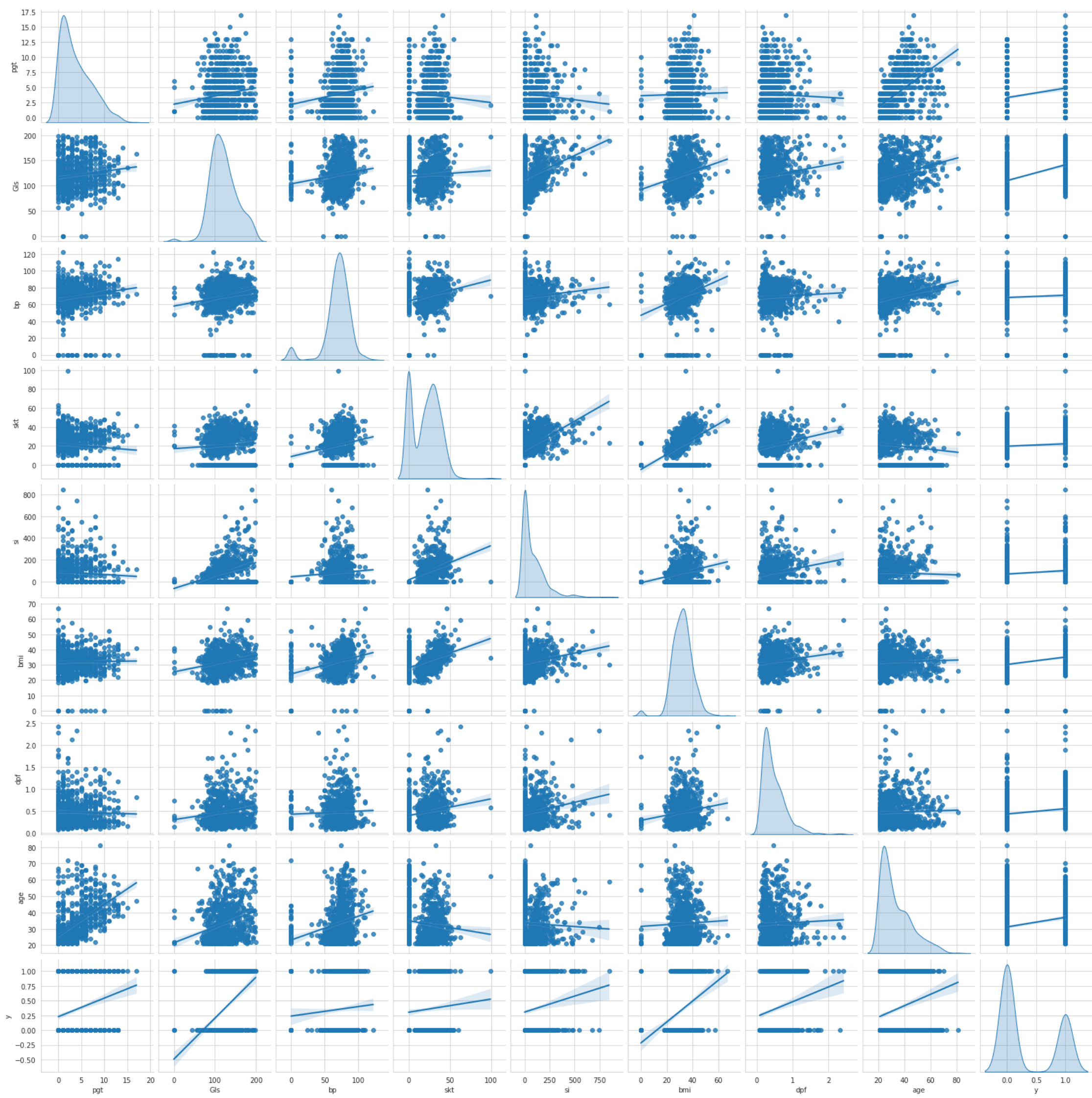
Step 4 is the model obtained after dropping the insignificant variables from the dataframe. Step 6 is using step/forward regression to give a best fit model after dropping few insignificant variables. As per the models obtained above from both the steps, we are getting the same model with 'hp' and 'turn' as the significant variables. The mistat package uses statsmodels as the underlying package and uses the same ols method to give the best fit model. So, as we are getting the same model by performing step-wise regression using the mi-stat package, therefore the performance parameters obtained in both step 4 and step 6 are same. Hence, we do not see a difference between the model parameters obtained in both the above steps.

Question No.4

1) Briefly describe relationships between the nine attributes listed above using pairplots and correlation plots (i.e., Pearson, Spearman and Phik correlation matrices as heatmaps).

```
df = pd.read_csv('dataFile6_Q4.csv', sep=',')
sns.pairplot(df, diag_kind='kde',kind='reg')
```

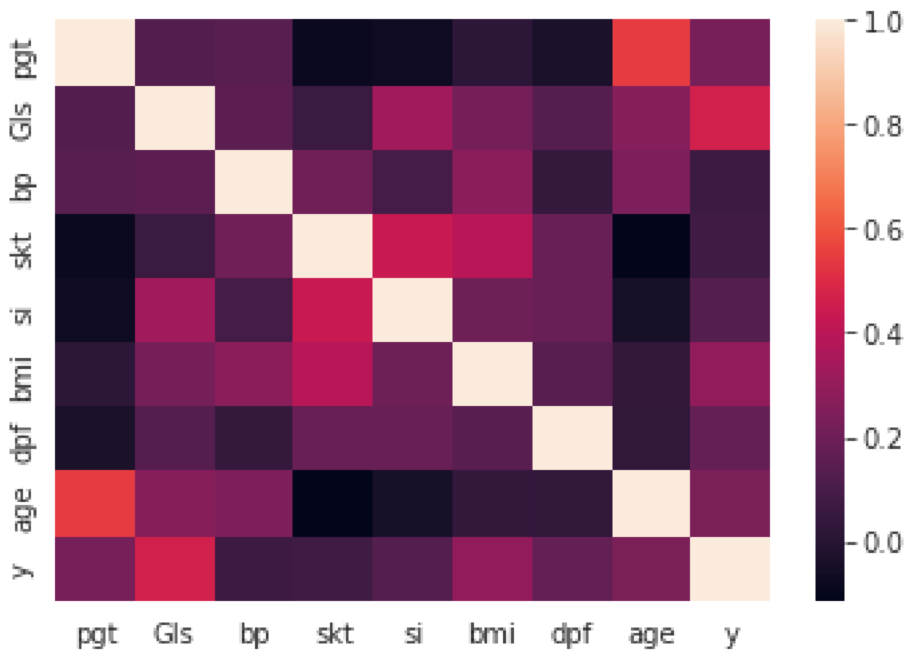
```
<seaborn.axisgrid.PairGrid at 0x7fdd16677130>
```




```
#Pearson's correlation cofficient
df_pearson = df.corr(method = "pearson")
print("Pearson Correlation")
print(df_pearson)
sns.heatmap(df_pearson)
```

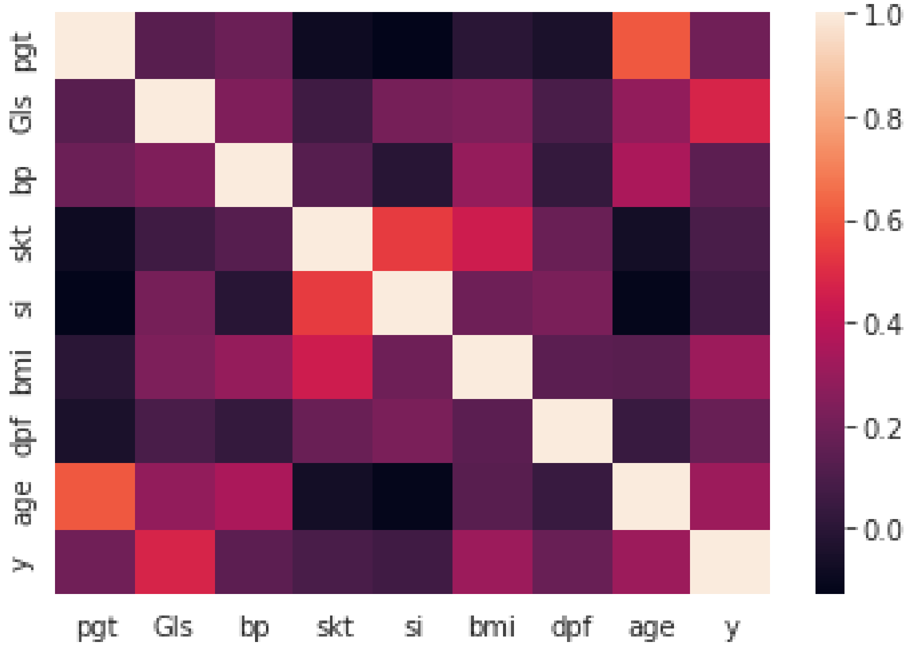
```
Pearson Correlation
          pgt       Gls        bp       skt        si       bmi
dpf  \
pgt  1.000000  0.129459  0.141282 -0.081672 -0.073535  0.017683 -
0.033523
Gls  0.129459  1.000000  0.152590  0.057328  0.331357  0.221071
0.137337
bp   0.141282  0.152590  1.000000  0.207371  0.088933  0.281805
0.041265
skt -0.081672  0.057328  0.207371  1.000000  0.436783  0.392573
0.183928
```

```
si  -0.073535  0.331357  0.088933  0.436783  1.000000  0.197859
0.185071
bmi  0.017683  0.221071  0.281805  0.392573  0.197859  1.000000
0.140647
dpf -0.033523  0.137337  0.041265  0.183928  0.185071  0.140647
1.000000
age  0.544341  0.263514  0.239528 -0.113970 -0.042163  0.036242
0.033561
y    0.221898  0.466581  0.065068  0.074752  0.130548  0.292695
0.173844

          age         y
pgt  0.544341  0.221898
Gls  0.263514  0.466581
bp   0.239528  0.065068
skt -0.113970  0.074752
si  -0.042163  0.130548
bmi  0.036242  0.292695
dpf  0.033561  0.173844
age  1.000000  0.238356
y    0.238356  1.000000

<AxesSubplot:>
```




```
df_spearman = df.corr(method = "spearman")
print("Spearman Correlation")
print(df_spearman)
sns.heatmap(df_spearman)
```

```
Spearman Correlation
          pgt       Gls        bp       skt        si       bmi
dpf  \
pgt  1.000000  0.130734  0.185127 -0.085222 -0.126723  0.000132 -
0.043242
Gls  0.130734  1.000000  0.235191  0.060022  0.213206  0.231141
0.091293
bp   0.185127  0.235191  1.000000  0.126486 -0.006771  0.292870
0.030046
skt -0.085222  0.060022  0.126486  1.000000  0.541000  0.443615
0.180390
si  -0.126723  0.213206 -0.006771  0.541000  1.000000  0.192726
0.221150
bmi  0.000132  0.231141  0.292870  0.443615  0.192726  1.000000
0.141192
dpf -0.043242  0.091293  0.030046  0.180390  0.221150  0.141192
1.000000
age  0.607216  0.285045  0.350895 -0.066795 -0.114213  0.131186
0.042909
y    0.198689  0.475776  0.142921  0.089728  0.066472  0.309707
0.175353

          age         y
pgt  0.607216  0.198689
Gls  0.285045  0.475776
bp   0.350895  0.142921
skt -0.066795  0.089728
si  -0.114213  0.066472
bmi  0.131186  0.309707
dpf  0.042909  0.175353
age  1.000000  0.309040
y    0.309040  1.000000

<AxesSubplot:>
```

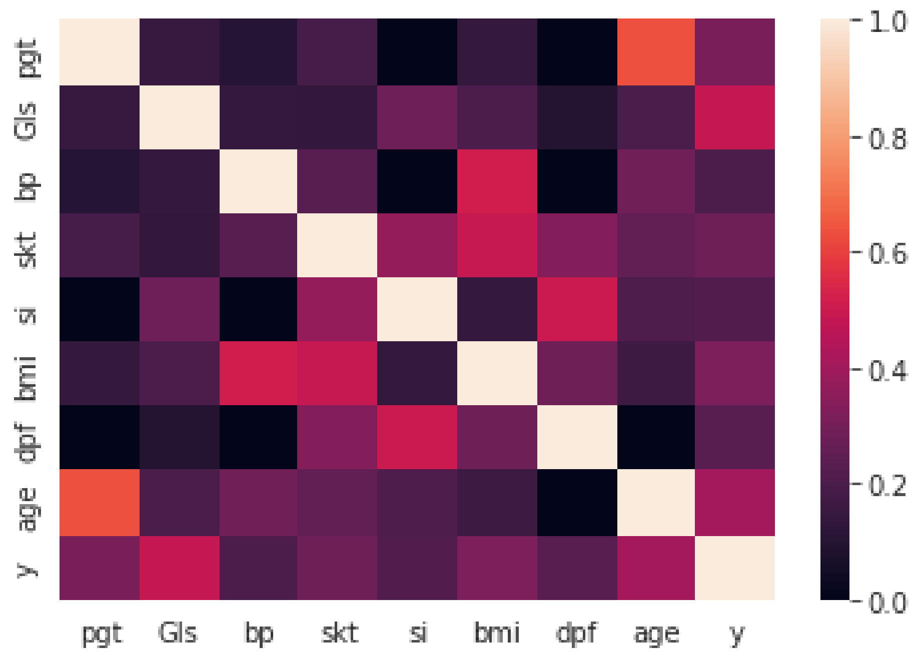



```
import phik

df_phik = df.phik_matrix()
print("Phi k Correlation")
print(df_phik)
sns.heatmap(df_phik)
```

```
interval columns not set, guessing: ['pgt', 'Gls', 'bp', 'skt', 'si',
'bmi', 'dpf', 'age', 'y']
Phi k Correlation
          pgt       Gls        bp       skt        si       bmi
dpf  \
pgt  1.000000  0.147507  0.100296  0.183777  0.000000  0.138248
0.000000
Gls  0.147507  1.000000  0.138568  0.136627  0.282687  0.202447
0.094732
bp   0.100296  0.138568  1.000000  0.232074  0.000000  0.512407
0.000000
skt  0.183777  0.136627  0.232074  1.000000  0.372447  0.491141
0.333682
si   0.000000  0.282687  0.000000  0.372447  1.000000  0.139973
0.496315
bmi  0.138248  0.202447  0.512407  0.491141  0.139973  1.000000
0.278092
dpf  0.000000  0.094732  0.000000  0.333682  0.496315  0.278092
1.000000
age  0.634490  0.198778  0.291258  0.252763  0.206671  0.156566
0.000000
y    0.307429  0.488153  0.199601  0.278824  0.208625  0.318172
0.227172
```

```
          age         y
pgt  0.634490  0.307429
Gls  0.198778  0.488153
bp   0.291258  0.199601
skt  0.252763  0.278824
si   0.206671  0.208625
bmi  0.156566  0.318172
dpf  0.000000  0.227172
age  1.000000  0.407535
y    0.407535  1.000000
```

```
<AxesSubplot:>
```



Summary

Pairplots:

- To plot mutiple bivariate distributions in a data we can use the pairplot() function. This shows that relationship for (n,2) combinations of variable in a dataframe as a matrix of plots and the diagonal plots are the univariate plots.
- It also visualizes the relation where the variables can be continuous or categorical.
- The diagonals along the pairplot shows us the univariate plots and pgt, gls ,si , dpf, age are having a normal distributions. With right skewness with features on pgt, si ,dpf and age.While the other features like bp, skt, bmi and y are having a bimodel distributions.
- In terms of Bivariate analysis, Most the data comprises of the non-linear relationship with other attributes.

- While most of the predictor variable's data is concentrated across the centers of normal distributions, there are some exceptions in case of bp and skt.

Correlation:

- The fundamental difference between pearson and spearson is that, pearson coefficient works on linear relationships(as in this data we dont have much linear relationships between the variables) whereas the Spearman Coefficient works with monotonic relationships.
- For instance,Consider the bp vs pgt has a dense set of points, pearson shows a score of 0.14 but spearman shows score of 0.19 as the pairplot shows a strict monotonic increasing function.

2) Include attributes 1-8 and 9 in two dataframes named X and y.

```
X = df.iloc[:, 0:8]
y = df.iloc[:, 8:]
print(X)
print(y)
```

```
     pgt  Gls  bp  skt   si   bmi    dpf  age
0      6  148  72   35    0  33.6  0.627   50
1      1   85  66   29    0  26.6  0.351   31
2      8  183  64    0    0  23.3  0.672   32
3      1   89  66   23   94  28.1  0.167   21
4      0  137  40   35  168  43.1  2.288   33
..   ...  ...  ..  ...  ...   ...    ...  ...
763   10  101  76   48  180  32.9  0.171   63
764    2  122  70   27    0  36.8  0.340   27
765    5  121  72   23  112  26.2  0.245   30
766    1  126  60    0    0  30.1  0.349   47
767    1   93  70   31    0  30.4  0.315   23

[768 rows x 8 columns]
     y
0    1
1    0
2    1
3    0
4    1
..  ..
763  0
764  0
765  0
766  1
767  0

[768 rows x 1 columns]
```

The above output displays the two dataframes X and Y as per the question's requirement.

3) Create training (i.e., X_train, y_train) and testing (X_test, y_test) datasets from the dataframes X and y in step 3, allocating 60% of samples to training set and 40% to testing set.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=0.6, random_state=666)
```

4) Apply logistic regression to the training dataset and evaluate classifier performance by computing the accuracy score and confusion matrix with the testing dataset (set the number of iterations to 1000).

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)

0.7727272727272727
```

The accuracy of the model is 77.27 % as per the logistic regression performed using given constraints.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test,y_pred)
cm

array([[181,  20],
       [ 50,  57]])
```

The above output represents the confusion matrix for the obtained regression model which shows that for the first attribute, 50 values (False positive) are mis-represented and for the second attribute 20 values (False negative) are mis-represented.

```
df_cm = pd.crosstab(y_test.values.flatten(),y_pred)
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'

target = df['y'].unique()
target = [str(value) for value in target]

from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, target_names=target))

              precision    recall  f1-score   support

           1       0.78      0.90      0.84       201
           0       0.74      0.53      0.62       107

    accuracy                           0.77       308
   macro avg       0.76      0.72      0.73       308
```

```
weighted avg       0.77      0.77      0.76       308
```

The above classification report shows the overall performance parameters such as accuracy, precision, recall, f1-score, support for the above model. These parameters will be used in the next sub-question.

5) By using the MinimaxScaler python function, scale the matrix created in step 2. Next, repeat steps 3 and 4 to employ the logistic regression with the training sample size set to 80% and the number of iterations set to 1000. Do data scaling and increasing training data size contribute to improved classification performance?

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, 
train_size=0.8, random_state=666)

from sklearn.linear_model import LogisticRegression
model = LogisticRegression(random_state=0,max_iter=1000)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)

0.7922077922077922
```

After scaling the X parameter and increasing the training data size, we are obtaining a higher accuracy of 79.22 % as compared to the unscaled-parameter model. So, we can say that here we are observing a performance improvement after scaling the X-parameter and increasing the training size of the data.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test,y_pred)
cm

array([[98,  9],
       [23, 24]])
```

From the above confusion matrix, we can observe that the classification has improved after scaling the X parameter and increasing the training data size. Now, only 23 values(False positives) are mis-represented in the first attribute and only 9 (False negatives) values are mis-represented in the second attriute.

```
df_cm = pd.crosstab(y_test.values.flatten(),y_pred)
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
```

```
target = df['y'].unique()
target = [str(value) for value in target]

from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, target_names=target))
```

```
              precision    recall  f1-score   support

           1       0.81      0.92      0.86       107
           0       0.73      0.51      0.60        47

    accuracy                           0.79       154
   macro avg       0.77      0.71      0.73       154
weighted avg       0.78      0.79      0.78       154
```

In the above classification report, we can see overall performance paramters have improved (precision, recall, f1-score, accuracy) as compared to the previous classification report without the added consraints in this step.

6) To perform cross-validated logistic regression, use the function LogisticRegressionCV(cv, random_state, max_iter) and the data used in step 5 (i.e., scaled). The cv value is set to 10 (i.e., 10-fold cross-validation), the random_state is set to 0, and the maximum iteration is 1000. Using the accuracy score and confusion matrix, evaluate the performance of the classifier.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
train_size=0.8, random_state=0)

from sklearn.linear_model import LogisticRegressionCV

clf_model = LogisticRegressionCV(cv=10, random_state=0,
max_iter=1000).fit(X_train, y_train)
y_pred = clf_model.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

```
0.8246753246753247
```

By using cross validation and other mentioned constrained in the question, we are obtaining a higher accuracy of 82.46 % as compared to the previous model. So, we can say that here we are observing a performance improvement by using the cross-validation and the added constraints in the given question.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test,y_pred)
cm
```

```
array([[98,  9],
       [18, 29]])
```

From the above confusion matrix, we can observe that the classification has improved further after using cross-validation and other constraints added in this step. Now, only 18 values (False positives) are mis-represented in the first attribute and no change in 9 (False negatives) values which mis-represented in the second attriute.

```
df_cm = pd.crosstab(y_test.values.flatten(),y_pred)
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'

target = df['y'].unique()
target = [str(value) for value in target]

from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, target_names=target))
```

```
              precision    recall  f1-score   support

           1       0.84      0.92      0.88       107
           0       0.76      0.62      0.68        47

    accuracy                           0.82       154
   macro avg       0.80      0.77      0.78       154
weighted avg       0.82      0.82      0.82       154
```

In the above classification report, we can see overall performance paramters have further improved (precision, recall, f1-score, accuracy) as compared to the previous classification report after using cross-validation method.

7) To answer this question, use the classification model created in step 6. Calculate the ROC curve using the steps provided below.

a. Define 20 threshold values as $\mathtt{threshold} = \mathtt{np.linspace}(0, 1, 20)$**,**

b. Use $\mathtt{y\_pred} = (\mathtt{model.predict\_proba}(\mathtt{X\_test})[:, 1] >= \mathtt{threshold}[i]).\mathtt{astype}(\mathtt{int})$ **to predict y values (y_pred) for each threshold value (i.e., threshold[i], where** $i = 0, 1, \cdots, 19$**) and model is the classifier name used in step 6),**

c. Use the confusion matrix to calculate the sensitivity and specificity for each threshold value,

d. Plot Sensitivity vs 1- spefifcity. Plot a line connecting the points (0,0) and (1,1) on the same graph.

```
threshold = np.linspace(0,1,20)
threshold
```

```
array([0.        , 0.05263158, 0.10526316, 0.15789474, 0.21052632,
       0.26315789, 0.31578947, 0.36842105, 0.42105263, 0.47368421,
       0.52631579, 0.57894737, 0.63157895, 0.68421053, 0.73684211,
       0.78947368, 0.84210526, 0.89473684, 0.94736842, 1.        ])
```

The above represents the generated threhold values as per the question.

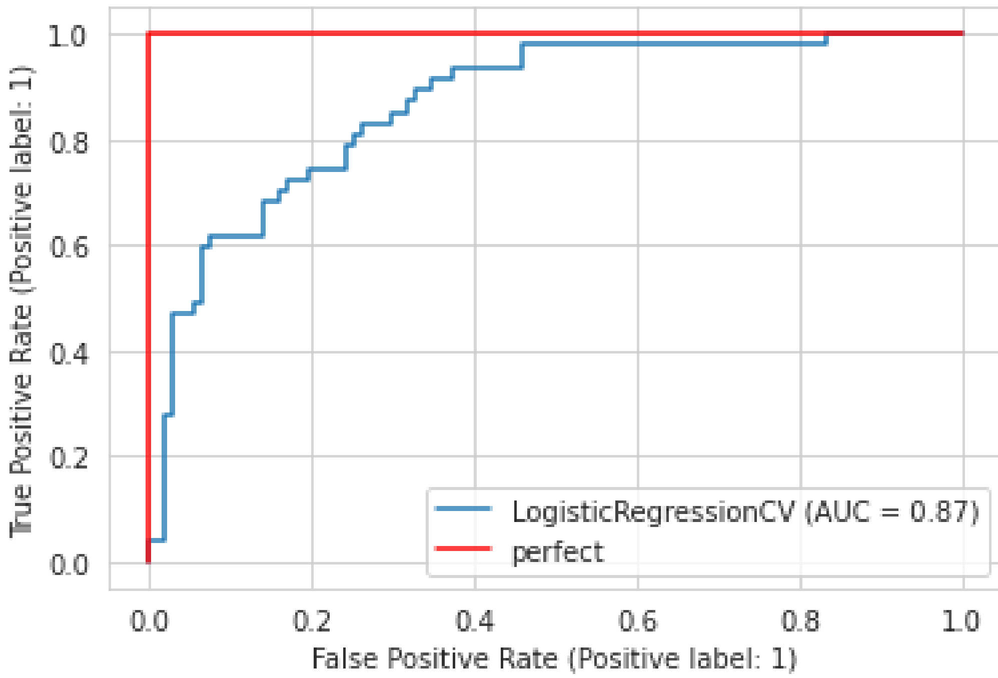
```
#y_pred = ( model.predict_proba( X_test )[ :, 1 ] >=
threshold[ i ] ).astype( int )
y_pred = list()
sensitivity_1=list()
specificity_1=list()

for i in range(0,len(threshold)):
    y_pred = (clf_model.predict_proba(X_test)[:,1] >=
threshold[i]).astype(int)
    cm = confusion_matrix(y_test, y_pred)
    print(cm)
    tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
    specificity_1.append(tn / (tn+fp))

    sensitivity_1.append(tp/(tp+fn))

print(f"Sensitivity: {sensitivity_1}")
print(f"Specificity: {specificity_1}")
```

```
[[  0 107]
 [  0  47]]
[[12 95]
 [ 0 47]]
[[32 75]
 [ 1 46]]
[[47 60]
 [ 1 46]]
[[61 46]
 [ 3 44]]
[[70 37]
 [ 4 43]]
[[75 32]
 [ 8 39]]
[[84 23]
 [12 35]]
[[91 16]
 [15 32]]
[[96 11]
 [18 29]]
[[99  8]
 [19 28]]
[[100   7]
 [ 21  26]]
[[100   7]
 [ 24  23]]
[[104   3]
 [ 28  19]]
[[104   3]
 [ 34  13]]
```

```
[[105   2]
 [ 38   9]]
[[105   2]
 [ 39   8]]
[[105   2]
 [ 44   3]]
[[106   1]
 [ 45   2]]
[[107   0]
 [ 47   0]]
Sensitivity: [1.0, 1.0, 0.9787234042553191, 0.9787234042553191,
0.9361702127659575, 0.9148936170212766, 0.8297872340425532,
0.7446808510638298, 0.6808510638297872, 0.6170212765957447,
0.5957446808510638, 0.5531914893617021, 0.48936170212765956,
0.40425531914893614, 0.2765957446808511, 0.19148936170212766,
0.1702127659574468, 0.06382978723404255, 0.0425531914893617, 0.0]
Specificity: [0.0, 0.11214953271028037, 0.29906542056074764,
0.4392523364485981, 0.5700934579439252, 0.6542056074766355,
0.7009345794392523, 0.7850467289719626, 0.8504672897196262,
0.897196261682243, 0.9252336448598131, 0.9345794392523364,
0.9345794392523364, 0.9719626168224299, 0.9719626168224299,
0.9813084112149533, 0.9813084112149533, 0.9813084112149533,
0.9906542056074766, 1.0]
```

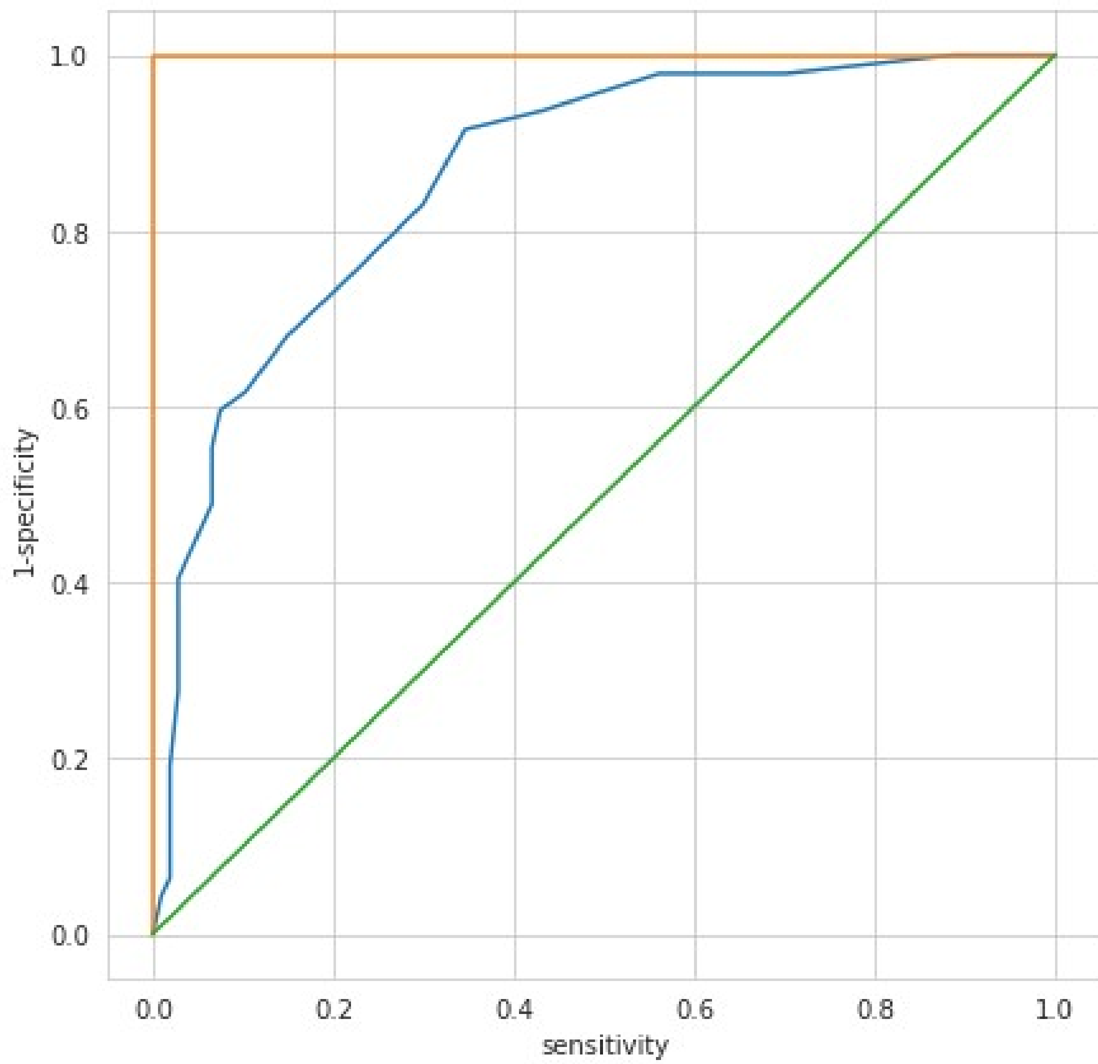
The above matrices represent the sentivity and specificity values for the y_pred input values using the threshold condition.

```
from sklearn.metrics import plot_roc_curve
plot_roc_curve(clf_model, X_test, y_test)
plt.plot([0,0,1,1],[0,1,1,1],'r-',label='perfect')
plt.legend()
plt.show()
```




```
#d plot senstivity and 1-specificity
plt.figure(figsize=(7,7))
specificity = 1-np.array(specificity_1)
plt.plot(specificity,sensitivity_1)
x=[0,1]
y=[0,1]
plt.xlabel('sensitivity')
plt.ylabel('1-specificity')
x1=[0,0,1]
y1=[0,1,1]
plt.plot(x1,y1)
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x7fdd1fff9280>]
```



From the above two graphs, we can see the ROC curve matches the plot of 'sensitivity' vs '1-specificity'.

Here, green line is data, blue line is actual test result and orange is perfect predictor.

An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters: True Positive Rate. False Positive Rate.