Project Report

Project title	**Classification Model Using Logistic Regression**
Module Name	**NICF Statistical Thinking for Data Science and Analytics (SF)**
Qualification Name	**NICF Diploma in Infocomm Technology (Data)**

Student name	**Assessor name**
TAN DAI JUN	SHANTI SEKHAR

Date issued	**Completion date**	**Submitted on**
18 MARCH, 2022	20 MARCH, 2022	20 MARCH, 2022

Project title	**Classification Model Using Logistic Regression**

Learner declaration
I certify that the work submitted for this assignment is my own and research sources are fully acknowledged.
Student signature: DJ Date: 13 MARCH, 2022

Table of Contents

1. Introduction

In supervised machine learning, classification is one of the techniques that has been widely implemented to predict the qualitative response. Logistic regression is a classification technique used in machine learning that uses one or more independent variables to predict the dichotomous dependant variables.

In this project, we will be using logistic regression to model “Pima Indians Diabetes” dataset and diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Prima Indian heritage.

2. Methodology

We first execute this project by extract the “Pima Indians Diabetes” dataset which is in csv format. Since we do not have prior knowledge regarding on this dataset, it is best to implement exploratory data analysis to understand each variable and their respective characteristics with respect to the context of the project. This will greatly help us in deciding the kind of imputations and assumptions during the data pre-processing stage.

Once we have done with the exploratory analysis, we will then impute missing or null values in the dataset to avoid bias in the model fitting stage. Lastly, we will fir the processed dataset into logistic regression model and examine its accuracy.

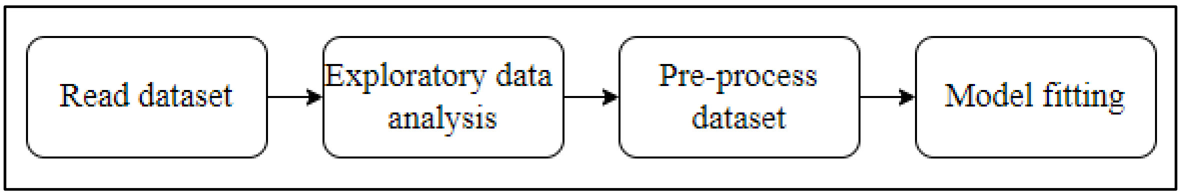
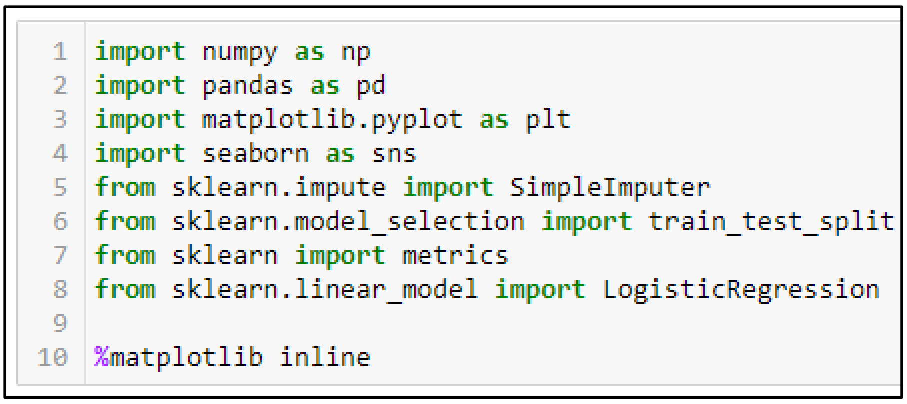


Figure 2.0: Conceptual diagram of methodology of this project

The entire project will be done using Python in Jupyter Notebook. Table below showcases some of the powerful libraries used in this project.

Library	Descriptions
numpy	House with large collection of mathematical functions to operate arrays
pandas	Built on top of numpy, offers data structure and operations for manipulating numerical table and time series
matplotlib	A cross-platform, data visualization and graphical plotting library
seaborn	Built on top of matplotlib, provides high level interface for drawing attractive and informative statistical graphics
sklearn	House with large collection of machine learning computation and operation functions
scipy	House with large collection of scientific and technical computing functions



```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.linear_model import LogisticRegression

%matplotlib inline
```

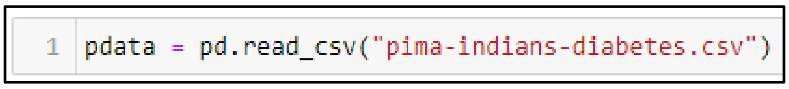
Figure 2.1: Libraries used in this project

3. Project Execution

With all the necessary libraries imported, we are ready to execute the project. Noted that for every important execution, there will be a dedicated explanation and summary throughout this section.

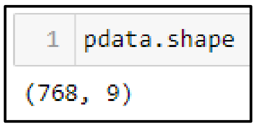
3.1 Dataset overview

In this section, we will read the "pima-indians-diabetes.csv" dataset using pandas. We can notice that the dataset has **768 observations** and **9 variables**.



```
pdata = pd.read_csv("pima-indians-diabetes.csv")
```

Figure 3.1.0: Read the dataset

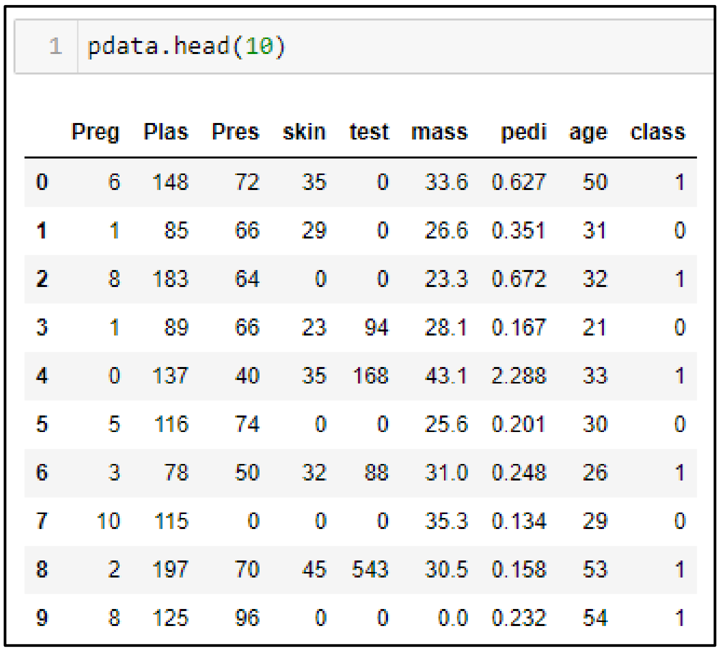


```
pdata.shape
```

```
(768, 9)
```

Figure 3.1.1: Dimension of dataset

We can eyeball the first 10 rows of observations of the dataset. We can understand the dataset using the summary table of data dictionary below.



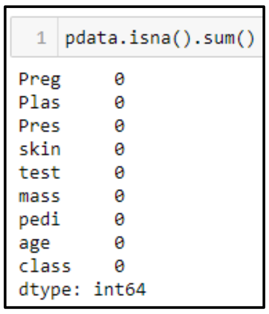
```
pdata.head(10)
```

	Preg	Plas	Pres	skin	test	mass	pedi	age	class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1

Figure 3.1.2: First 10 observations of the dataset

Column name	**Description**
Preg	Number of times pregnant
Plas	Glucose concentration after 2 hours in an oral glucose tolerance test
Pres	Blood pressure (*mm Hg*)
Skin	Skin fold thickness (*mm*)
Test	2-hour serum insulin (*mm U/ml*)
Mass	BMI (*kg/m*^{2})
Pedi	Pedigree function
Age	Age (*year*)
Class	Dependant variable, whether has diabetes. Represented by value of 1 for has diabetes and 0 for not having diabetes.

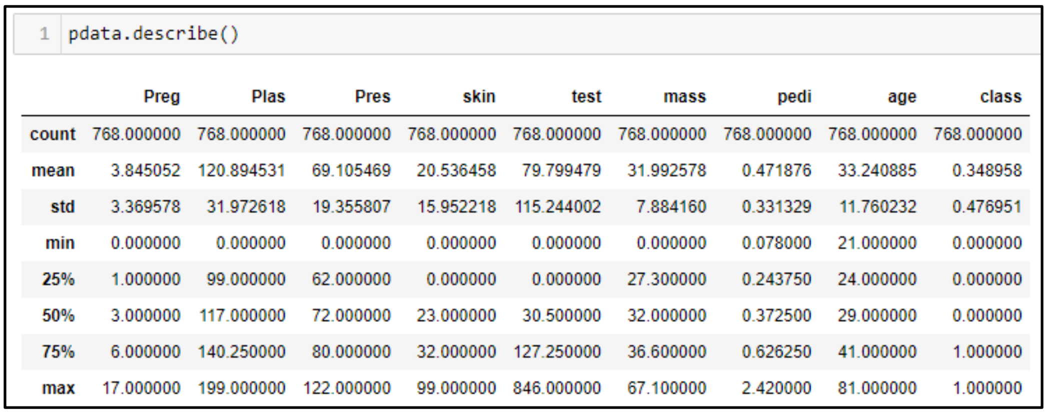
Besides, we check if there are any null values in the dataset. We can observe that there are no missing values across the entire dataset.



```
pdata.isna().sum()
```

```
Preg     0
Plas     0
Pres     0
skin     0
test     0
mass     0
pedi     0
age      0
class    0
dtype: int64
```

Figure 3.1.3: Check the existence of missing values in dataset



```
pdata.describe()
```

	Preg	Plas	Pres	skin	test	mass	pedi	age	class
count	768.000000	768.000000	768.000000	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	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Figure 3.1.4: Summary of 5 value for all independent variables

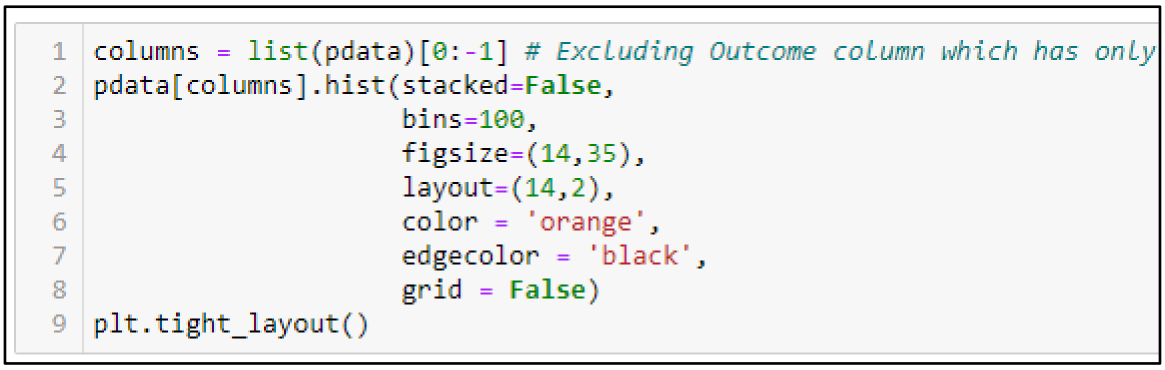
Based on summary of 5 value (*min, max, lower quartile, median and upper quartile*), we can observe some abnormalities such that value of 0 should not be exist inside the dataset. For instance, one should not be having value of 0 in the following body measurements:

Plas, Pres, Skin, Test, Mass, Pedi

In order to further investigate and impute missing values with suitable value, we will conduct exploratory data analysis.

3.2 Univariate exploratory data analysis

Univariate exploratory data analysis can be described as the analysis of one variable. We first try to understand the distribution of each independent variables by plotting histogram of their respective distribution.



```python
columns = list(pdata)[0:-1] # Excluding Outcome column which has only
pdata[columns].hist(stacked=False,
                    bins=100,
                    figsize=(14,35),
                    layout=(14,2),
                    color = 'orange',
                    edgecolor = 'black',
                    grid = False)
plt.tight_layout()
```

Figure 3.2.0: Source code for distribution plots for all independent variables

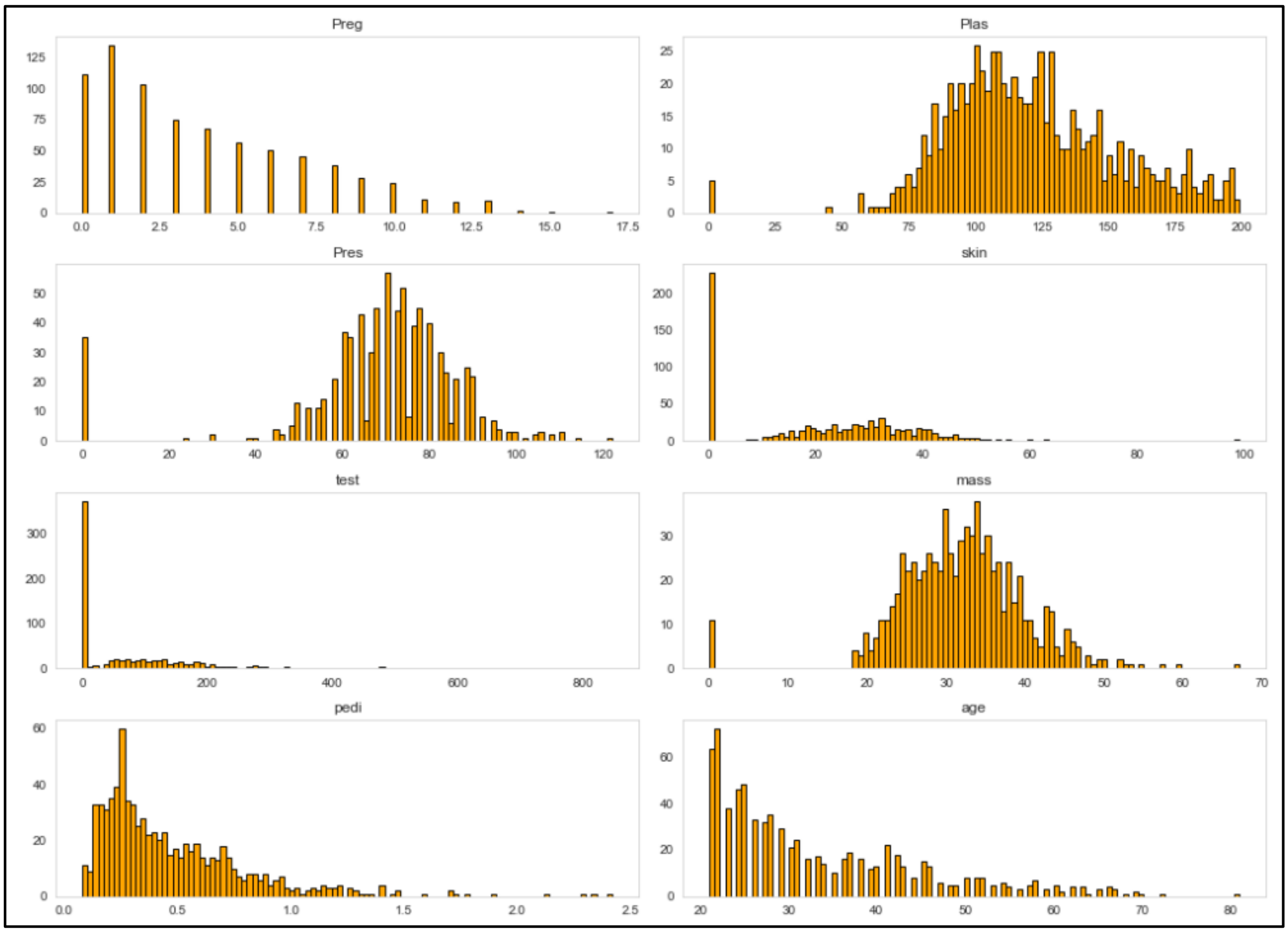


Figure 3.2.1: Distribution plots for all independent variables

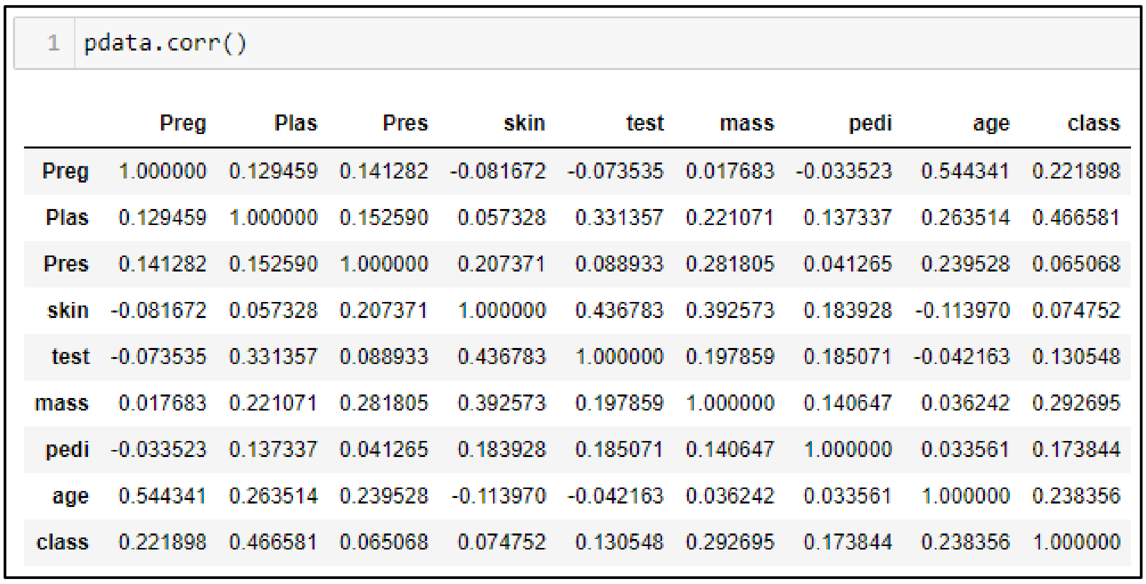
Based on figure 3.2.1, we can summarize the distribution of each independent variable as table below:

Independent variable	Summary
Preg	Right skewed distribution
Plas	Right skewed
Pres	Close to normal distribution, outlier exist
Skin	Right skewed distribution, outliers exist
Test	Right skewed distribution, outliers exist
Mass	Close to normal distribution, outliers exist
Pedi	Right skewed distribution, outlies exist
Age	Right skewed distribution, outliers exist

Based on table above, we can presume that excludes "Age" column, the other columns that are having outliers are most probably due to the missing values. One extra thing to highlight is that by understanding the distribution of each independent variables, we can then choose the right scaling method.

3.3 Bivariate exploratory data analysis

Bivariate analysis can be understood as exploring the relationship between 2 independent variables from the dataset. Since logistic regression assumes no multicollinearity among the independent variables, we can benefit from bivariate exploratory analysis to check if the dataset is suitable to fit in logistic regression model.

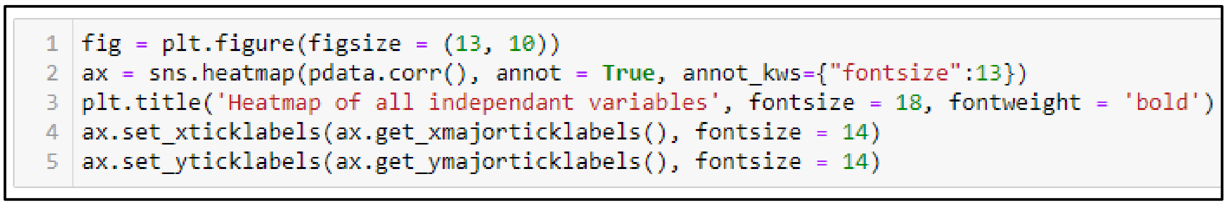


```
pdata.corr()
```

	Preg	Plas	Pres	skin	test	mass	pedi	age	class
Preg	1.000000	0.129459	0.141282	-0.081672	-0.073535	0.017683	-0.033523	0.544341	0.221898
Plas	0.129459	1.000000	0.152590	0.057328	0.331357	0.221071	0.137337	0.263514	0.466581
Pres	0.141282	0.152590	1.000000	0.207371	0.088933	0.281805	0.041265	0.239528	0.065068
skin	-0.081672	0.057328	0.207371	1.000000	0.436783	0.392573	0.183928	-0.113970	0.074752
test	-0.073535	0.331357	0.088933	0.436783	1.000000	0.197859	0.185071	-0.042163	0.130548
mass	0.017683	0.221071	0.281805	0.392573	0.197859	1.000000	0.140647	0.036242	0.292695
pedi	-0.033523	0.137337	0.041265	0.183928	0.185071	0.140647	1.000000	0.033561	0.173844
age	0.544341	0.263514	0.239528	-0.113970	-0.042163	0.036242	0.033561	1.000000	0.238356
class	0.221898	0.466581	0.065068	0.074752	0.130548	0.292695	0.173844	0.238356	1.000000

Figure 3.3.0: Correlation matrix across all independent variables

In order to examine if multicollinearity exist in our dataset more easily, we can visualize the correlation using heatmap.



```python
fig = plt.figure(figsize = (13, 10))
ax = sns.heatmap(pdata.corr(), annot = True, annot_kws={"fontsize":13})
plt.title('Heatmap of all independant variables', fontsize = 18, fontweight = 'bold')
ax.set_xticklabels(ax.get_xmajorticklabels(), fontsize = 14)
ax.set_yticklabels(ax.get_ymajorticklabels(), fontsize = 14)
```

Figure 3.3.1: Source code for heatmap of all independent variables

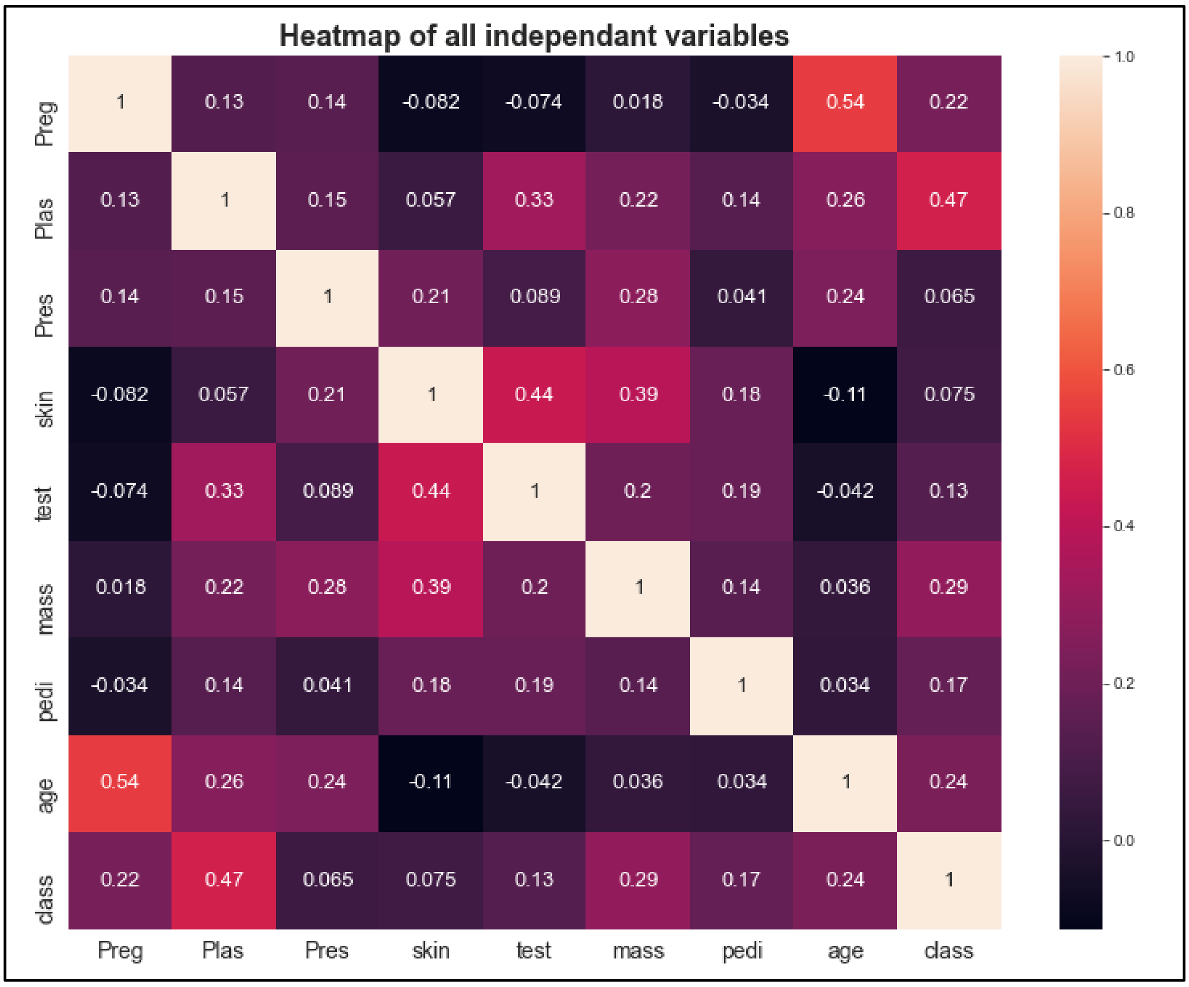
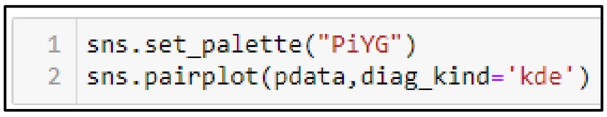


Figure 3.3.2: Heatmap of all independent variables

Based on figure 3.3.2, we can notice that independent variables of the dataset are not highly correlated. Thus, it is safe to conclude that assumption of multicollinearity of logistic regression is not violated. Furthermore, we can use pair plot to visually observe the trend of relationship between each variable with each other as well as shown in figure 3.3.4.



```python
sns.set_palette("PiYG")
sns.pairplot(pdata,diag_kind='kde')
```

Figure 3.3.3: Source code of pair plot for all variables in dataset

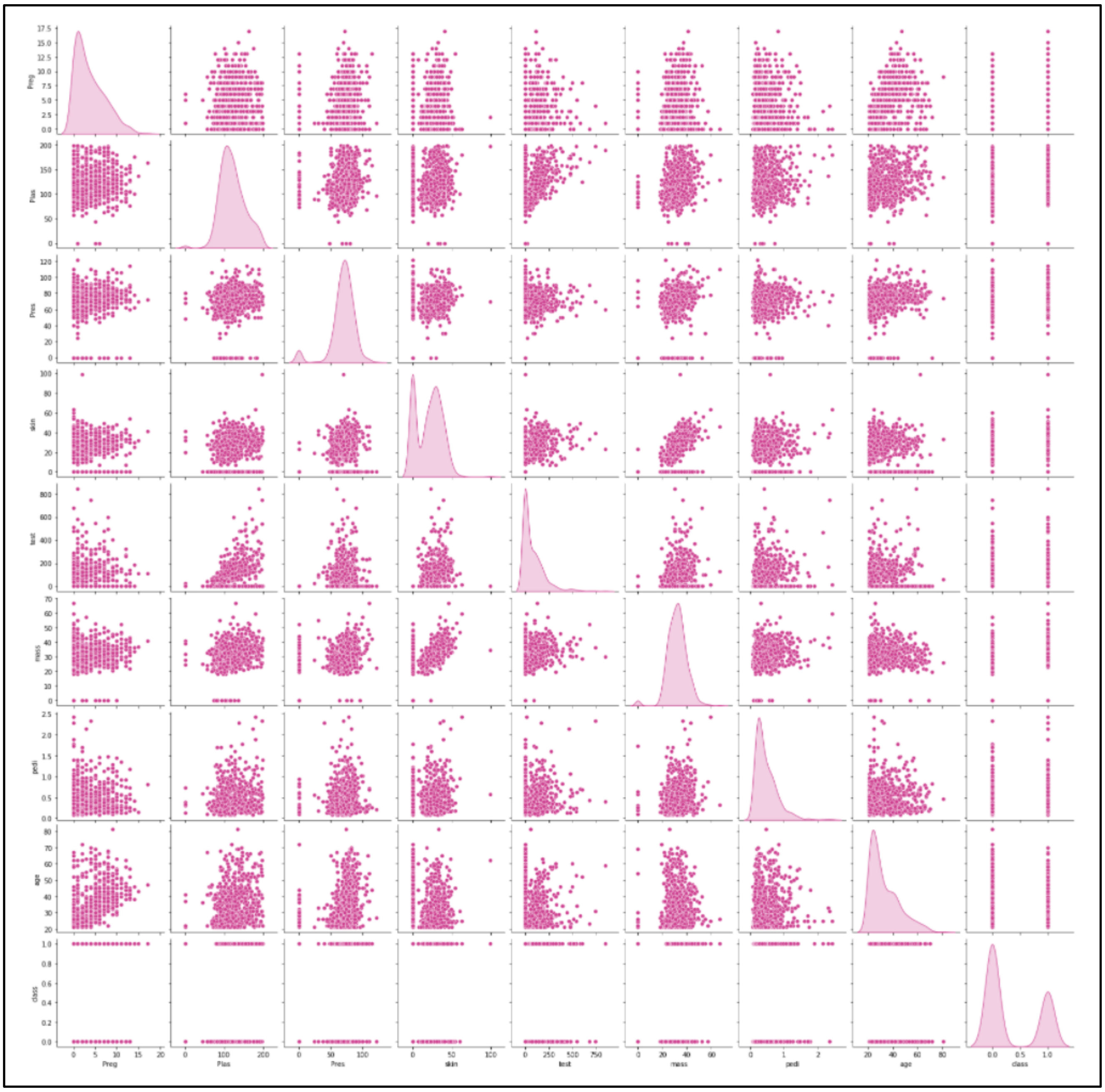
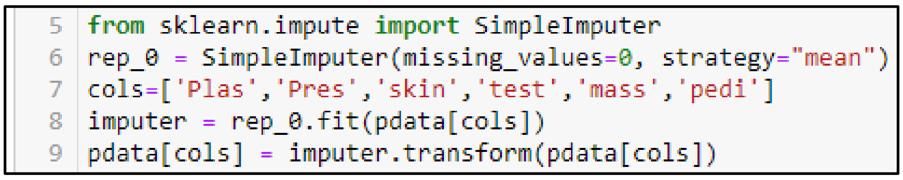


Figure 3.3.4: Pair plots of all variables in the dataset

Based on the pair plot, we can once again conclude that there is no obvious correlated trend between independent variables.

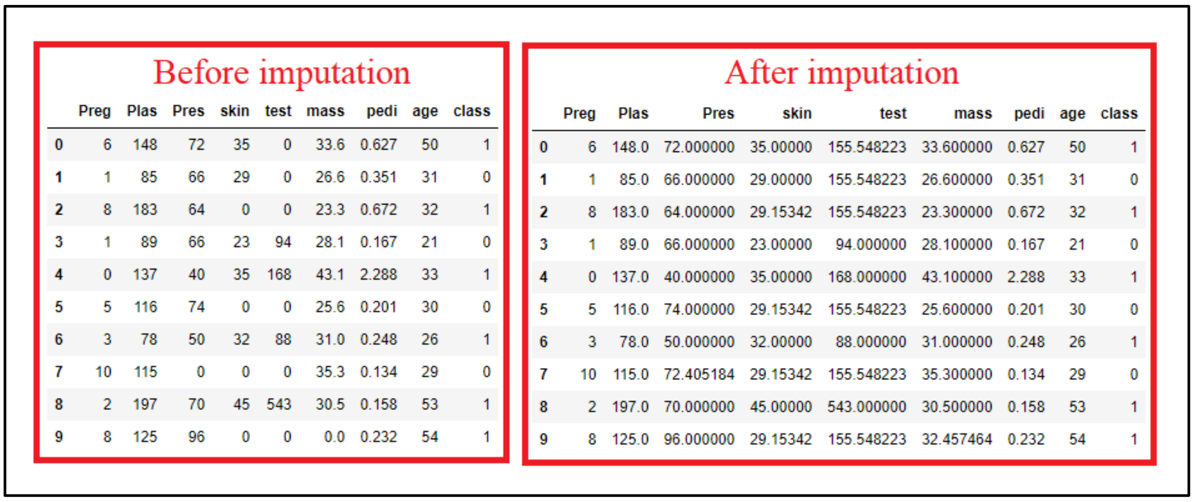
3.4 Data pre-processing

After the exploratory data analysis from previous section, we have understood each variable from the dataset and it is time to impute the missing values (*value of 0*). In this section, we will impute columns that are mentioned previously with their respective mean value.



```python
from sklearn.impute import SimpleImputer
rep_0 = SimpleImputer(missing_values=0, strategy="mean")
cols=['Plas','Pres','skin','test','mass','pedi']
imputer = rep_0.fit(pdata[cols])
pdata[cols] = imputer.transform(pdata[cols])
```

Figure 3.4.0: Source code of imputation



Before imputation

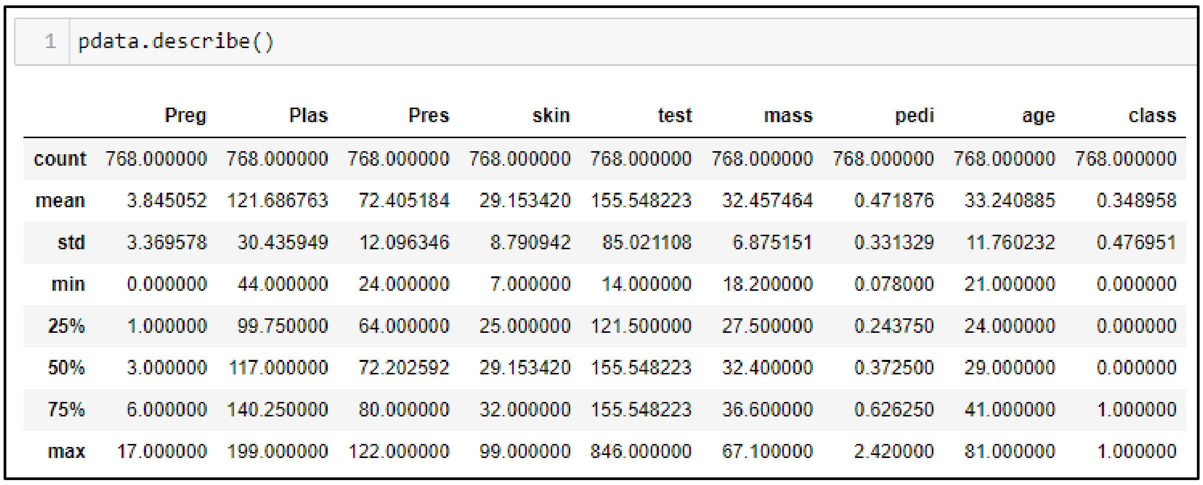
	Preg	Plas	Pres	skin	test	mass	pedi	age	class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1

After imputation

	Preg	Plas	Pres	skin	test	mass	pedi	age	class
0	6	148.0	72.000000	35.00000	155.548223	33.600000	0.627	50	1
1	1	85.0	66.000000	29.00000	155.548223	26.600000	0.351	31	0
2	8	183.0	64.000000	29.15342	155.548223	23.300000	0.672	32	1
3	1	89.0	66.000000	23.00000	94.000000	28.100000	0.167	21	0
4	0	137.0	40.000000	35.00000	168.000000	43.100000	2.288	33	1
5	5	116.0	74.000000	29.15342	155.548223	25.600000	0.201	30	0
6	3	78.0	50.000000	32.00000	88.000000	31.000000	0.248	26	1
7	10	115.0	72.405184	29.15342	155.548223	35.300000	0.134	29	0
8	2	197.0	70.000000	45.00000	543.000000	30.500000	0.158	53	1
9	8	125.0	96.000000	29.15342	155.548223	32.457464	0.232	54	1

Figure 3.4.1: Dataset before and after imputation

Furthermore, the summary of 5 values of each independent variable has been updated as shown below.



```python
pdata.describe()
```

	Preg	Plas	Pres	skin	test	mass	pedi	age	class
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	121.686763	72.405184	29.153420	155.548223	32.457464	0.471876	33.240885	0.348958
std	3.369578	30.435949	12.096346	8.790942	85.021108	6.875151	0.331329	11.760232	0.476951
min	0.000000	44.000000	24.000000	7.000000	14.000000	18.200000	0.078000	21.000000	0.000000
25%	1.000000	99.750000	64.000000	25.000000	121.500000	27.500000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.202592	29.153420	155.548223	32.400000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	155.548223	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

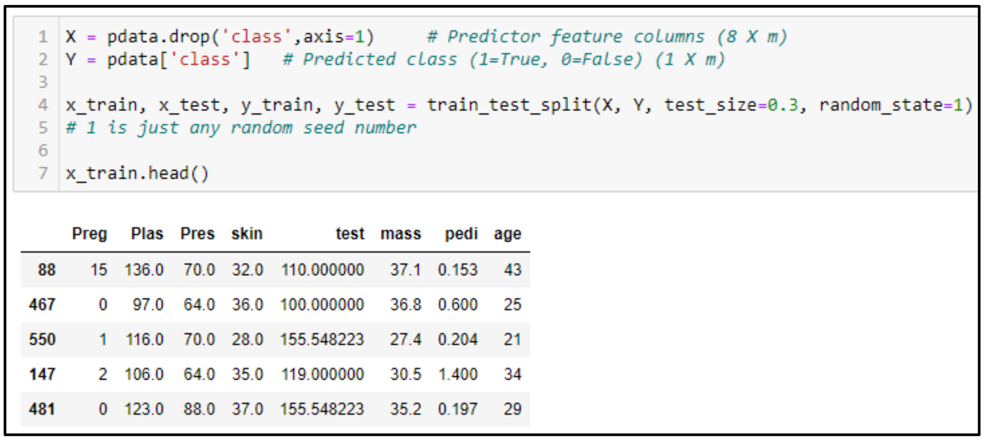
Figure 3.4.2: Summary of 5 values of each independent variable

3.5 Split data

Supervised machine learning requires labelled datasets to train the algorithm in order to classify or predict accurately. The contemporary way to train machine learning model is splitting the dataset into train-test, which become two sets of data.

In this section, the train dataset is used to fit logistic regression model and contrary the remaining dataset, test dataset is used to evaluate the fir of our logistic regression model. In short, the main objective of data splitting is to estimate the performance of our machine learning model.

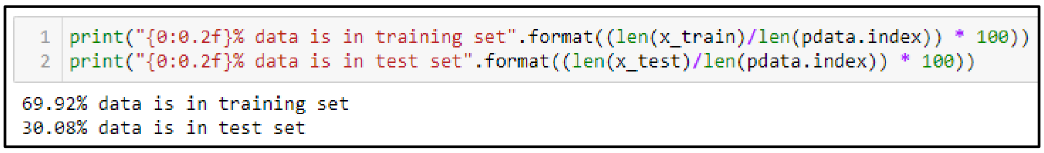
Noted that dataset will be split randomly into the ratio of 70:30, which means 70% of the dataset are fed into model training while the remaining 30% are meant for model testing.



```python
X = pdata.drop('class',axis=1)     # Predictor feature columns (8 X m)
Y = pdata['class']   # Predicted class (1=True, 0=False) (1 X m)

x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=1)
# 1 is just any random seed number

x_train.head()
```

	Preg	Plas	Pres	skin	test	mass	pedi	age
88	15	136.0	70.0	32.0	110.000000	37.1	0.153	43
467	0	97.0	64.0	36.0	100.000000	36.8	0.600	25
550	1	116.0	70.0	28.0	155.548223	27.4	0.204	21
147	2	106.0	64.0	35.0	119.000000	30.5	1.400	34
481	0	123.0	88.0	37.0	155.548223	35.2	0.197	29

Figure 3.5.0: Source code of data splitting with the ratio of 70:30

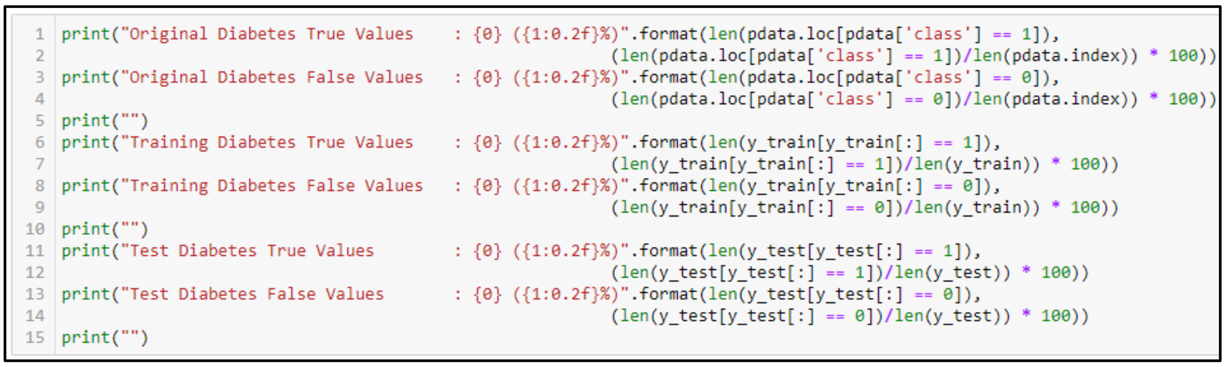


```python
print("{0:0.2f}% data is in training set".format((len(x_train)/len(pdata.index)) * 100))
print("{0:0.2f}% data is in test set".format((len(x_test)/len(pdata.index)) * 100))
```

```
69.92% data is in training set
30.08% data is in test set
```

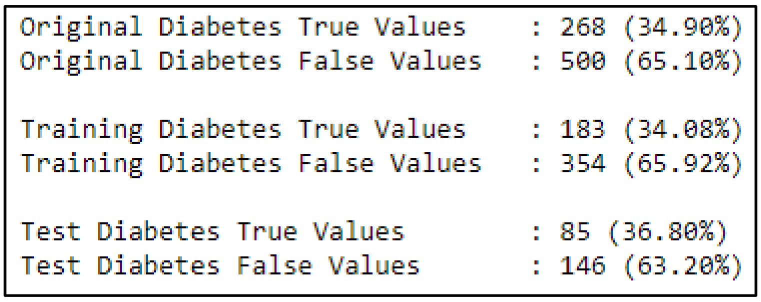
Figure 3.5.1: Dataset after splitting

We can notice that the dataset is indeed spit into 69.92% for training set and 30.08% for testing set. Lastly, we take a look at the distribution of diabetes before data split and after data split.



```python
print("Original Diabetes True Values    : {0} ({1:0.2f}%)".format(len(pdata.loc[pdata['class'] == 1]),
                                                        (len(pdata.loc[pdata['class'] == 1])/len(pdata.index)) * 100))
print("Original Diabetes False Values   : {0} ({1:0.2f}%)".format(len(pdata.loc[pdata['class'] == 0]),
                                                        (len(pdata.loc[pdata['class'] == 0])/len(pdata.index)) * 100))
print("")
print("Training Diabetes True Values    : {0} ({1:0.2f}%)".format(len(y_train[y_train[:] == 1]),
                                                        (len(y_train[y_train[:] == 1])/len(y_train)) * 100))
print("Training Diabetes False Values   : {0} ({1:0.2f}%)".format(len(y_train[y_train[:] == 0]),
                                                        (len(y_train[y_train[:] == 0])/len(y_train)) * 100))
print("")
print("Test Diabetes True Values        : {0} ({1:0.2f}%)".format(len(y_test[y_test[:] == 1]),
                                                        (len(y_test[y_test[:] == 1])/len(y_test)) * 100))
print("Test Diabetes False Values       : {0} ({1:0.2f}%)".format(len(y_test[y_test[:] == 0]),
                                                        (len(y_test[y_test[:] == 0])/len(y_test)) * 100))
print("")
```

Figure 3.5.2: Source code of checking the distribution of dependent variable



```
Original Diabetes True Values    : 268 (34.90%)
Original Diabetes False Values   : 500 (65.10%)

Training Diabetes True Values    : 183 (34.08%)
Training Diabetes False Values   : 354 (65.92%)

Test Diabetes True Values        : 85 (36.80%)
Test Diabetes False Values       : 146 (63.20%)
```

Figure 3.5.3: Distribution of dependent variables for original, train and test dataset

Based on figure 3.5.3, we can notice that training dataset, the dataset with 70% of the sample size has the distribution proportion of dependent variable approximately same as the original dataset while the test distribution proportion from test dataset is slightly vary a bit.

This can be explained by the larger the sample size, the more it approximates the population parameter.

3.6 Model fitting

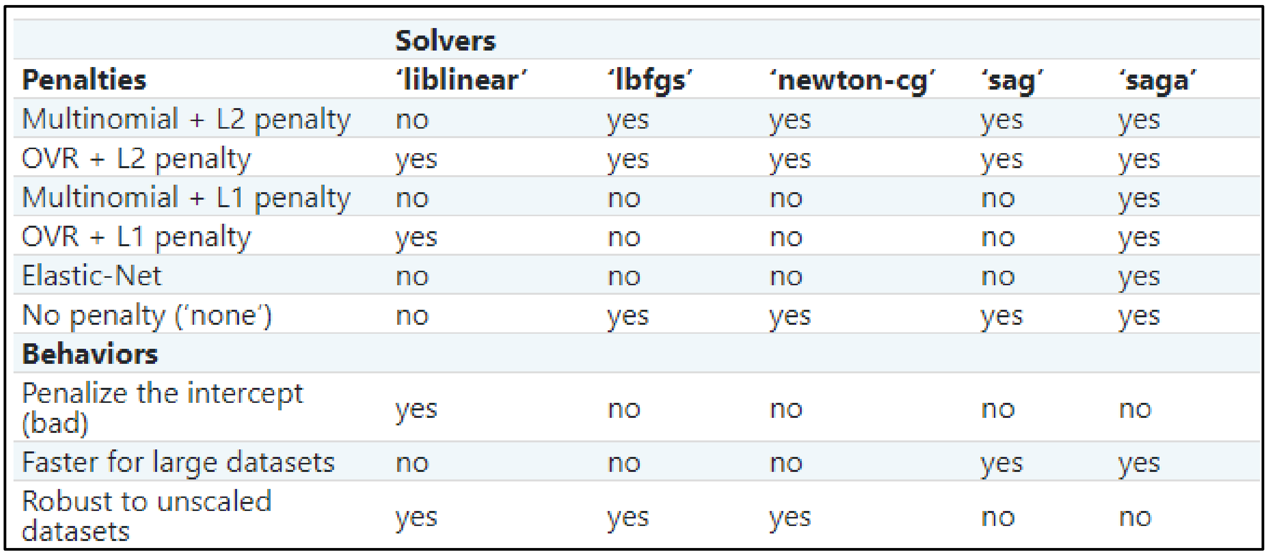
During machine learning modelling, one of the important components is regularization. Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or underfitting. By using regularization, we can fit our machine learning model appropriately on a given test set and hence reduce the errors in it.

Table below describes 3 most frequently used regularization approaches.

Regularization approach	**Formula**	**Short description**
L1 regularization	$\min_{w,c} \|w\|_1 + C\sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$	Shrink less important features or removing them
L2 regularization	$\min_{w,c} \frac{1}{2} w^T w + C\sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$	Remove the percentage of weight of features, do not remove them
Elastic-Net regularization	$\min_{w,c} \frac{1-\rho}{2} w^T w + \rho\|w\|_1 + C\sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$	Combination of l1 and l2 by using ρ to control the strength of l1 against l2

By understanding the fundamentals of regularization approaches, it can help us for solver selection during model fitting. Solver in regression model can be understood as the tool to find the parameter/coefficient weights that minimize a cost function, which related to regularization approaches mentioned earlier.

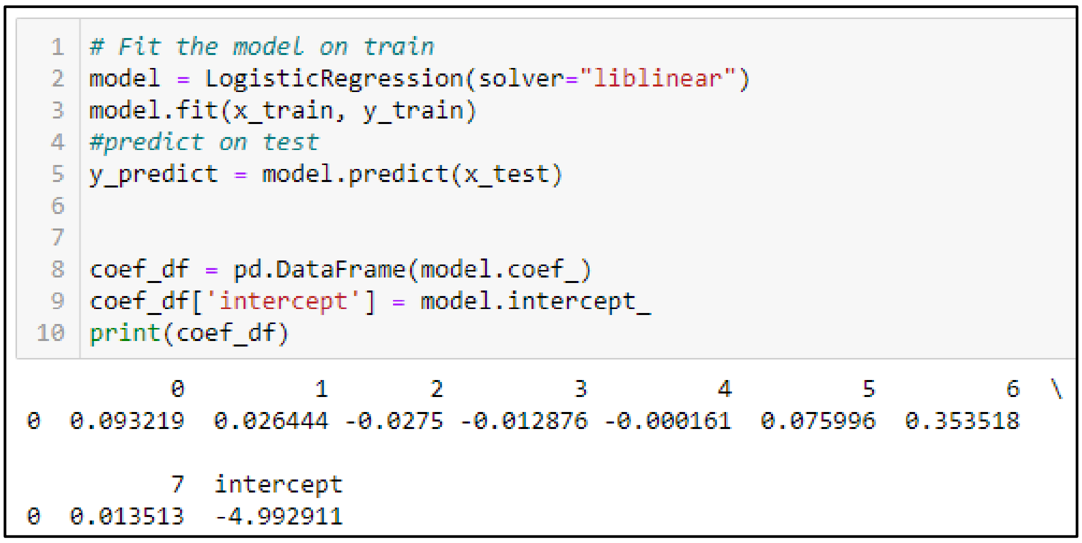
The library we are using in model fitting, Scikit-Learn comes with 5 different solvers, which are *newton-cg*, *lbfgs*, *liblinear*, *sag*, *saga*. Figure below from the Scikit-Learn documentation lists characteristics of the solvers, including the regularization penalties available.



	Solvers				
Penalties	**‘liblinear’**	**‘lbfgs’**	**‘newton-cg’**	**‘sag’**	**‘saga’**
Multinomial + L2 penalty	no	yes	yes	yes	yes
OVR + L2 penalty	yes	yes	yes	yes	yes
Multinomial + L1 penalty	no	no	no	no	yes
OVR + L1 penalty	yes	no	no	no	yes
Elastic-Net	no	no	no	no	yes
No penalty (‘none’)	no	yes	yes	yes	yes
Behaviors					
Penalize the intercept (bad)	yes	no	no	no	no
Faster for large datasets	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	no	no

Figure 3.6.0: Characteristics of 5 different solvers in Scikit-Learn

Since *newton-cg*, *sag*, *sage*, *lbfgs* are generally more suitable in solving multiclass problems (*A classification task with more than two classes*) while *liblinear* solver is more suitable for smaller dataset, we will be using *liblinear* as the solver to regularize our logistic regression model.



```
# Fit the model on train
model = LogisticRegression(solver="liblinear")
model.fit(x_train, y_train)
#predict on test
y_predict = model.predict(x_test)


coef_df = pd.DataFrame(model.coef_)
coef_df['intercept'] = model.intercept_
print(coef_df)
```

```
          0         1       2         3         4         5         6  \
0  0.093219  0.026444 -0.0275 -0.012876 -0.000161  0.075996  0.353518

          7  intercept
0  0.013513  -4.992911
```

Figure 3.6.1: Source code and computed coefficients of logistic regression model with *liblinear* as solver

Based on the coefficients printed in our output, we can express our logistic regression as

$$\log(odds) = -4.993 + 0.0932x_{Preg} + 0.0264x_{Plas} - 0.0275x_{Pres} - 0.0129x_{skin} - 0.0002x_{test} + 0.0760x_{mass} + 0.3535x_{pedi} + 0.0135x_{age}$$

3.7 Model Evaluation

Once our logistic regression model has been trained with 70% of the dataset, we will use the remaining 30% of the test dataset to examine the performance of our model. Confusion matrix is a table that used to define the performance of a classification algorithm as it visualizes and summarizes the performance of a classification model.

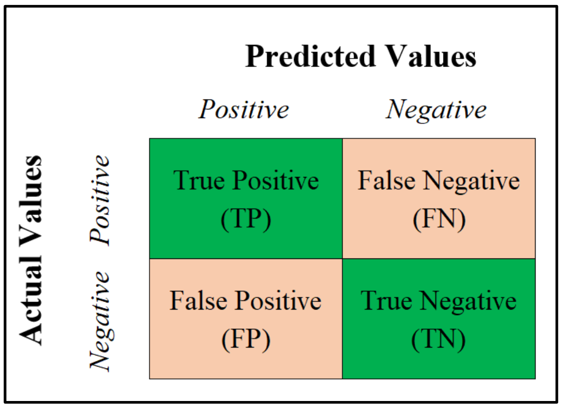
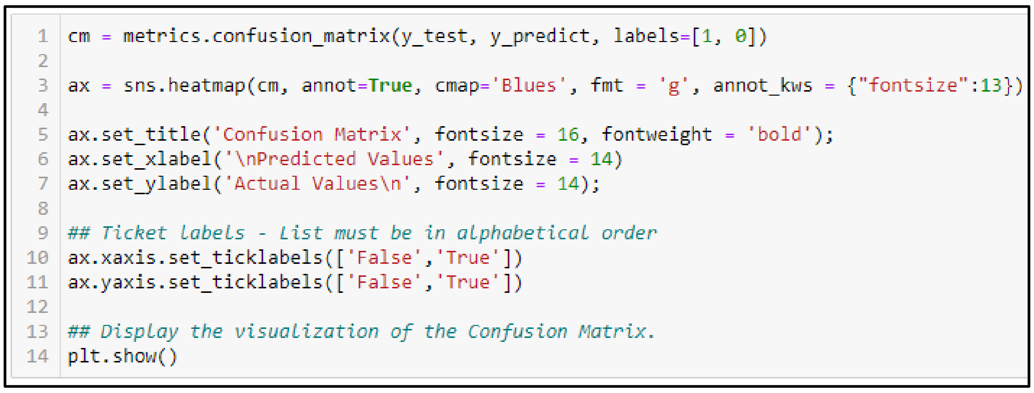


Figure 3.7.0: Conceptual diagram of confusion matrix



```
cm = metrics.confusion_matrix(y_test, y_predict, labels=[1, 0])

ax = sns.heatmap(cm, annot=True, cmap='Blues', fmt = 'g', annot_kws = {"fontsize":13})

ax.set_title('Confusion Matrix', fontsize = 16, fontweight = 'bold');
ax.set_xlabel('\nPredicted Values', fontsize = 14)
ax.set_ylabel('Actual Values\n', fontsize = 14);

## Ticket labels - List must be in alphabetical order
ax.xaxis.set_ticklabels(['False','True'])
ax.yaxis.set_ticklabels(['False','True'])

## Display the visualization of the Confusion Matrix.
plt.show()
```

Figure 3.7.1: Source code for confusion matrix of our logistic regression model

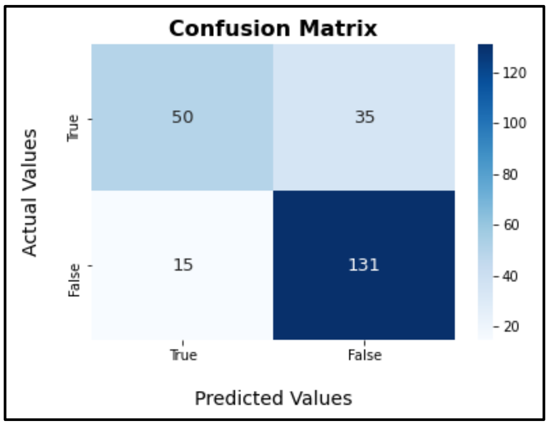


Figure 3.7.2: Confusion matrix of our model

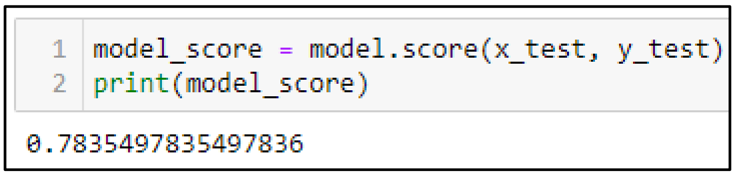
Based on figure 3.7.2, we can summarize our model as

- We have **correctly** predicted **50 diabetes positive** cases
- We have **correctly** predicted **131 diabetes negative** cases
- We have **incorrectly** predicted **35 diabetes positive** cases
- We have **incorrectly** predicted **15 diabetes negative** cases

Having to know all the distribution of true and false predictions made, we can use them to evaluate the performance of our model. Table below showcases some of the useful measures/metrics for model performance evaluation.

Measures	**Descriptions**	**Formula**
Accuracy	The ratio of correct predictions to total predictions made. Best accuracy is 1 and the worst is 0.	$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$
Recall	Indicates how many actual positive values the model was able to predict correctly out of total true positive cases.	$Recall = \frac{TP}{TP + FN}$
Precision	Indicates how many correctly predicted cases actually turned out to be positive out of total predicted positive cases.	$Precision = \frac{TP}{TP + FP}$
F1 Score	The harmonic mean of precision and recall. It is maximum when precision is equal to recall.	$F1\ Score = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}}$

One thing to highlight is that we can use python to calculate the accuracy of our logistic regression model.



```
model_score = model.score(x_test, y_test)
print(model_score)
```

```
0.7835497835497836
```

Figure 3.7.3: Model sore after evaluation

Based on figure 3.7.3, we can notice that our regression has the accuracy of around 78.35%.

3.8 Exploration in effect of scaling

In this section, we will be trying to explore if features scaling will help to improve the accuracy of our logistic regression model. The scaling technique we will be using in this section is z-score normalization. Z-score normalization is a variation of scaling that represents the number of standard deviations away from the mean and it is particularly useful when there are outliers (*however not extreme outliers*).

The reason we use z-score normalization is because after imputation, we can observe that the distribution of the independent variables in our dataset still show the sign of outliers.

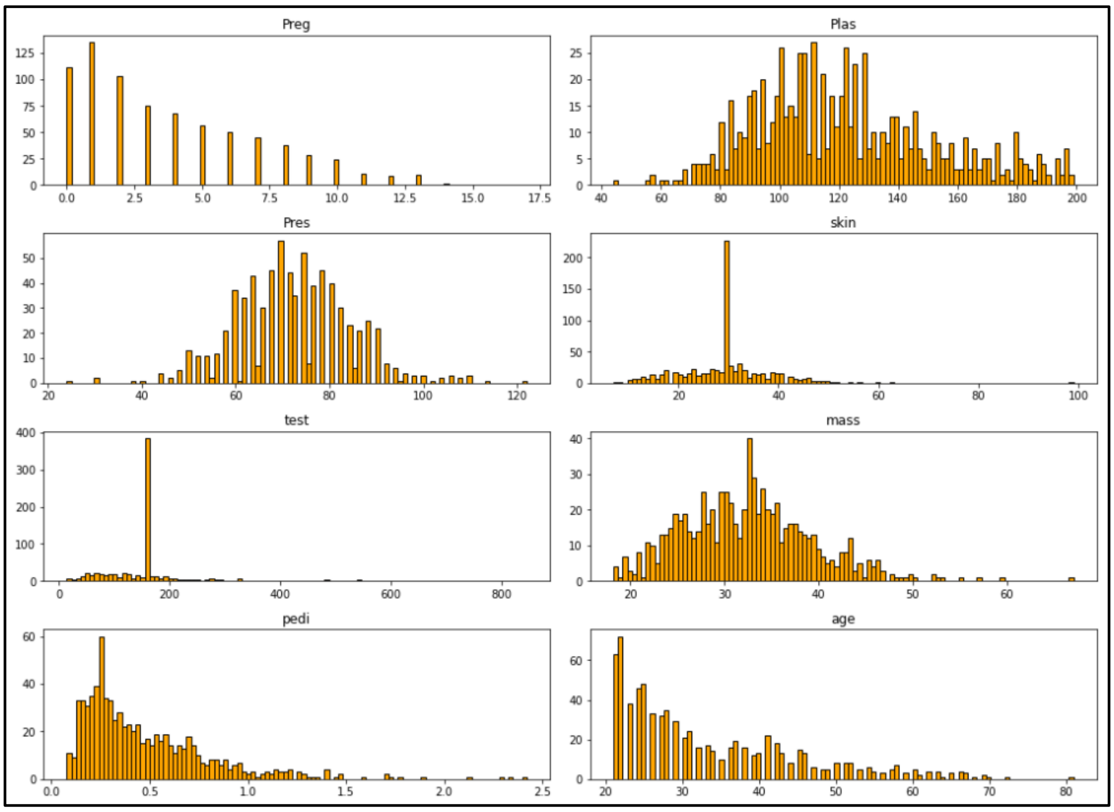
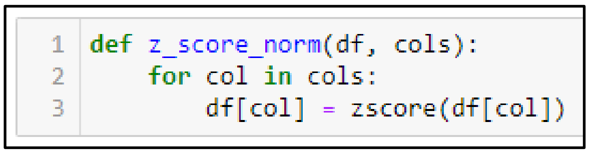
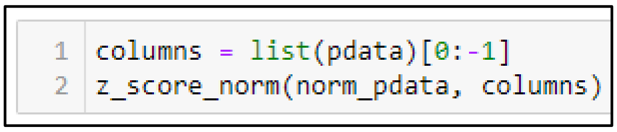


Figure 3.8.0: Distribution of independent variables after imputation



```
def z_score_norm(df, cols):
    for col in cols:
        df[col] = zscore(df[col])
```

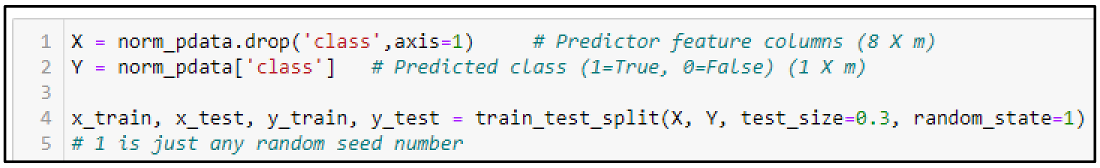
Figure 3.8.1: Source code of function to conduct z-score normalization



```
columns = list(pdata)[0:-1]
z_score_norm(norm_pdata, columns)
```

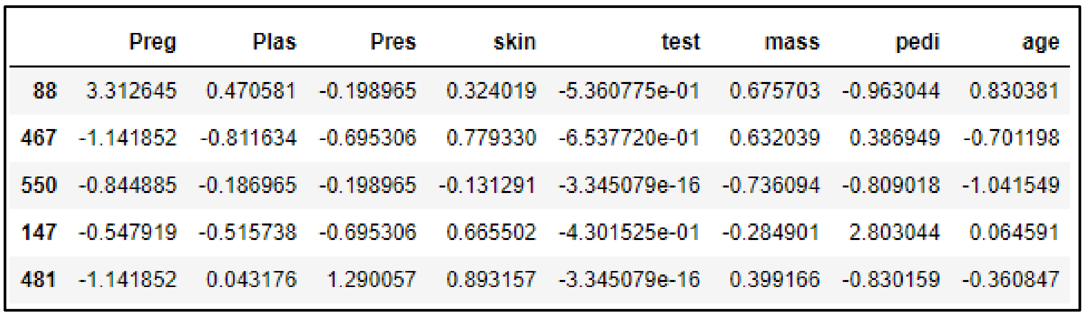
Figure 3.8.2: Source code of implementation of z-score normalization function

Once the independent variables have been normalized, we use the same approach as previous section and split data into ratio of 70:30.



```python
X = norm_pdata.drop('class',axis=1)     # Predictor feature columns (8 X m)
Y = norm_pdata['class']   # Predicted class (1=True, 0=False) (1 X m)

x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=1)
# 1 is just any random seed number
```

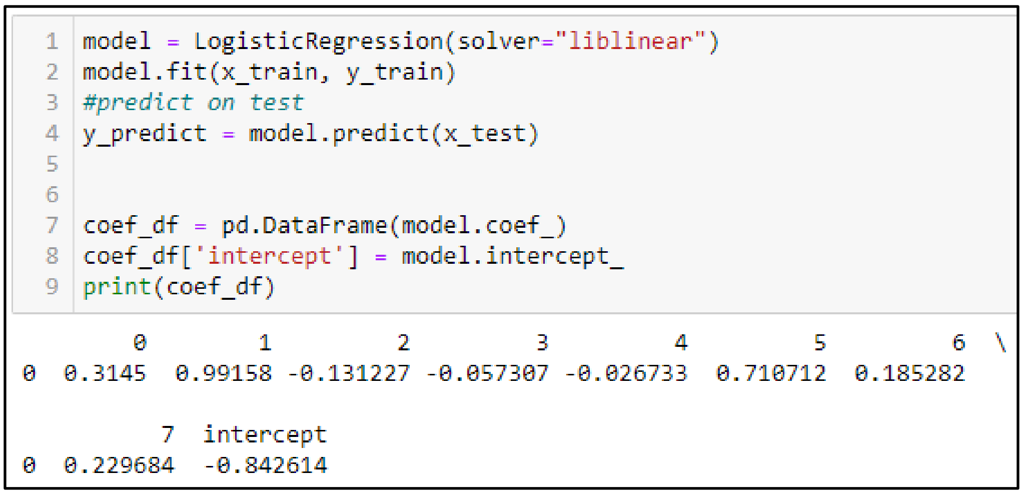
Figure 3.8.3: Normalized dataset data splitting



	Preg	Plas	Pres	skin	test	mass	pedi	age
88	3.312645	0.470581	-0.198965	0.324019	-5.360775e-01	0.675703	-0.963044	0.830381
467	-1.141852	-0.811634	-0.695306	0.779330	-6.537720e-01	0.632039	0.386949	-0.701198
550	-0.844885	-0.186965	-0.198965	-0.131291	-3.345079e-16	-0.736094	-0.809018	-1.041549
147	-0.547919	-0.515738	-0.695306	0.665502	-4.301525e-01	-0.284901	2.803044	0.064591
481	-1.141852	0.043176	1.290057	0.893157	-3.345079e-16	0.399166	-0.830159	-0.360847

Figure 3.8.4: Normalized test dataset

We will now fit 70% of the normalized dataset into the logistic regression model,



```python
model = LogisticRegression(solver="liblinear")
model.fit(x_train, y_train)
#predict on test
y_predict = model.predict(x_test)


coef_df = pd.DataFrame(model.coef_)
coef_df['intercept'] = model.intercept_
print(coef_df)
```

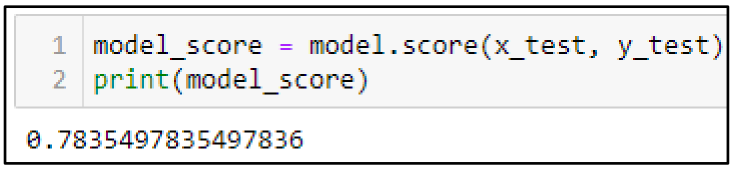
```
        0        1         2         3         4         5         6  \
0  0.3145  0.99158 -0.131227 -0.057307 -0.026733  0.710712  0.185282

          7  intercept
0  0.229684  -0.842614
```

Figure 3.8.5: Model fitting with normalized train dataset

We can now express our new logistic regression model as:

$$\log(odds) = -0.8426 + 0.3145x_{Preg} + 0.9916x_{Plas} - 0.1312x_{Pres} - 0.0573x_{skin} - 0.0267x_{test} + 0.7107x_{mass} + 0.1853x_{pedi} + 0.2297x_{age}$$

Lastly, we compute the accuracy of our new model. We can notice that the accuracy of our new model is exactly the same as our old model (*without normalized dataset*).



```
model_score = model.score(x_test, y_test)
print(model_score)
```

```
0.7835497835497836
```

Figure 3.8.6: Accuracy of the model with normalized dataset

This probably can be explained by the dataset although is normalized, but the weightage of each independent variables, their respective coefficient inside the new model is still the same. Therefore, our new model will still be having the same accuracy as our previous one.

4. Conclusion

As a conclusion, we have considered experience one life cycle of the data modelling process, which is from data extraction, data exploration, data munging and lastly model fitting. On the other hand, we have experienced the importance of the exploratory data analysis throughout this project as it can help us in understanding the dataset as well as decide the best approach for imputation during data munging.

Lastly, we have learnt that there are variety of approaches can be used during model fitting. It is crucial to have a solid mathematical and statistical background in order to create a well perform model.