Project Objective

- To create a model that could predict whether the loan application should be accepted or rejected.
- To help different banks like SBI, HDFC, PNB etc. to identify suitable application for approving loan through this model.
- To save money, time and manpower required for selecting and rejecting the loan application.
- To automate the process of identifying suitable application for loan approval without any human interference.

Data Description

➢ Variable Description

Variables	Description
Loan_ID	Unique Loan ID
Gender	Male/Female
Married	Applicant married (Y/N)
Dependents	Number of dependents
Education	Applicant Education (Graduate/Under Graduate)
Self_Employed	Self employed (Y/N)
ApplicantIncome	Applicant income
CoapplicantIncome	Coapplicant income
Loan Amount	Loan amount in thousands
Loan_Amount_Term	Term of loan in months
Credit_History	Credit history meets guidelines
Property_Area	Urban/Semi Urban/Rural
Loan_Status	Loan approved (Y/N)

➢ Software And Resources Requirements

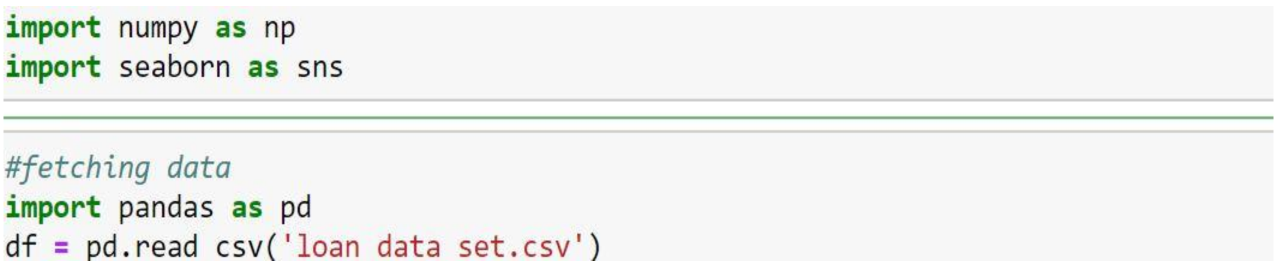
Jupyter Notebook, datasets from Anaylatics Vidyhya for training and testing the model.

Exploratory Data Analysis (EDA)

➢ Understanding the Data

In this section we will understand the data set and each column.

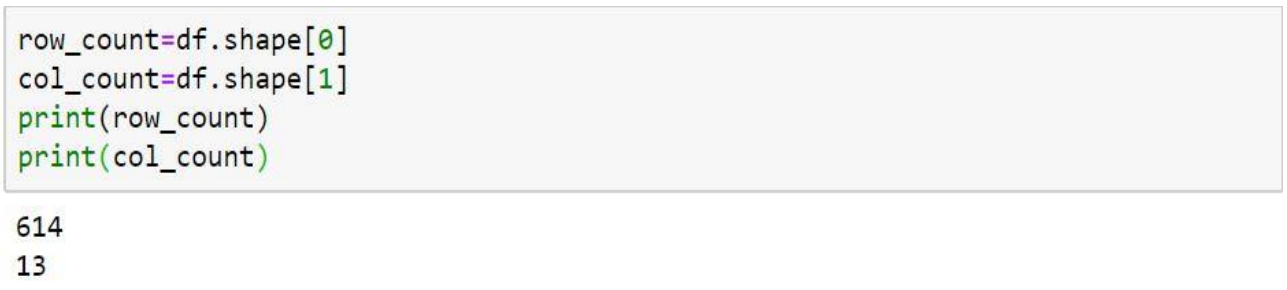
Data frames in python allows us to perform various mathematical and other useful function on the data . So, at first we will fetch the data from normal data set into the data frame.



```python
import numpy as np
import seaborn as sns
```

```python
#fetching data
import pandas as pd
df = pd.read_csv('loan_data_set.csv')
```

Then we will analyze and understand the data.
To understand the data we don't need to see each and every row of data.
A sample of data will give the idea of whole dataset.



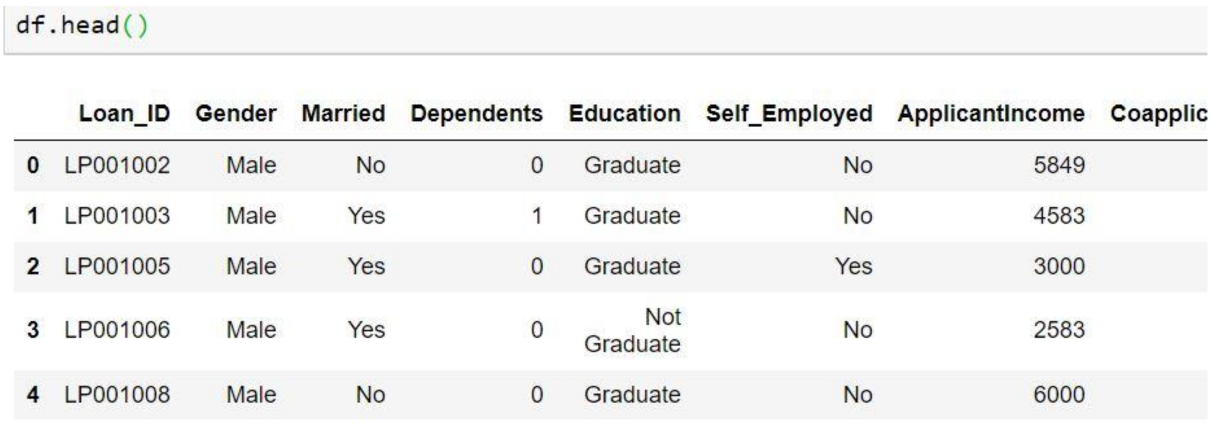
```python
row_count=df.shape[0]
col_count=df.shape[1]
print(row_count)
print(col_count)
```

```
614
13
```

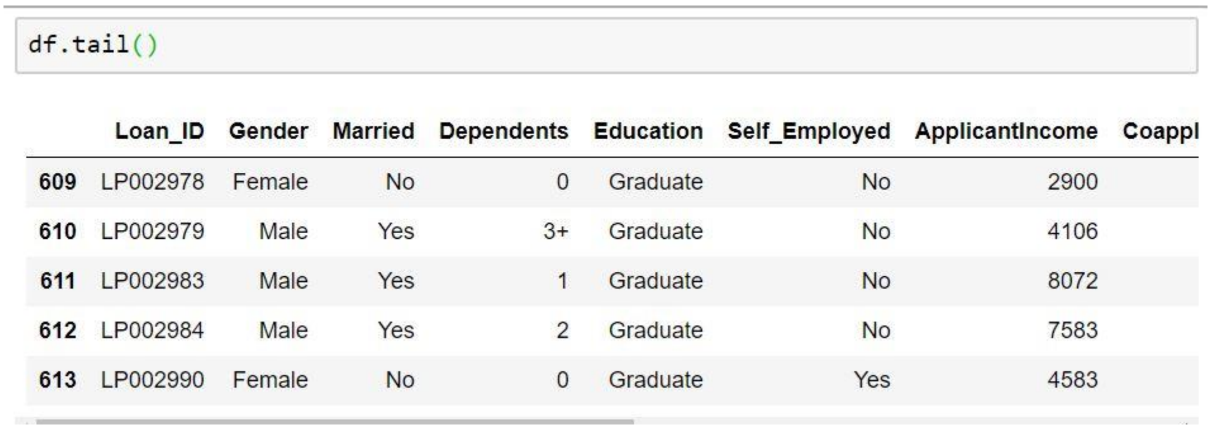
Explanation:: Shape method helps in counting the no. of rows and columns in data frame. Here shape[0] is for counting no. of rows and shape[1] represent the no. columns. The number represent the axis."0" is used for row axis and 1 is used for column axis.

Observation:: Hence we have 614 rows and 13 columns/fields in the data set.

Now, we see a sample of data using head and tail method.

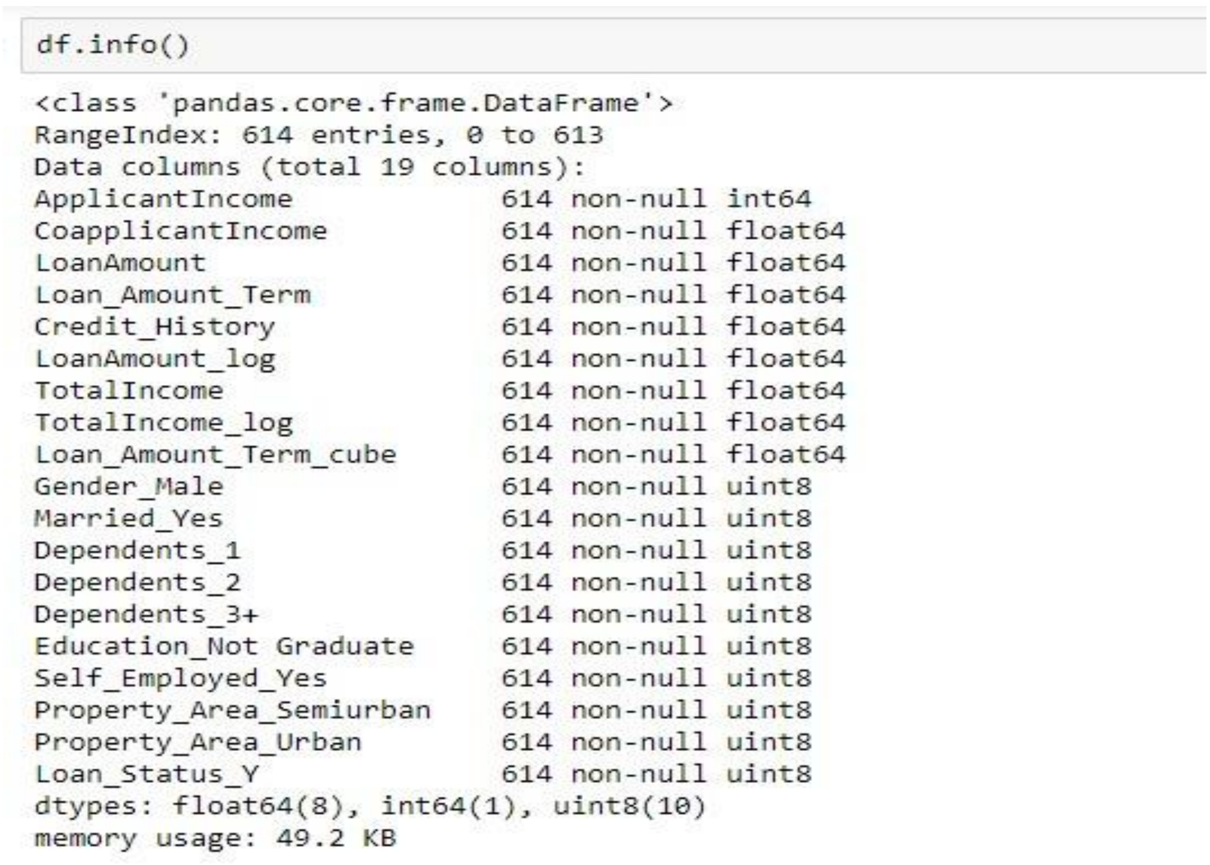

```
df.head()
```

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	Coapplic
0	LP001002	Male	No	0	Graduate	No	5849	
1	LP001003	Male	Yes	1	Graduate	No	4583	
2	LP001005	Male	Yes	0	Graduate	Yes	3000	
3	LP001006	Male	Yes	0	Not Graduate	No	2583	
4	LP001008	Male	No	0	Graduate	No	6000	


```
df.tail()
```

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	Coappl
609	LP002978	Female	No	0	Graduate	No	2900	
610	LP002979	Male	Yes	3+	Graduate	No	4106	
611	LP002983	Male	Yes	1	Graduate	No	8072	
612	LP002984	Male	Yes	2	Graduate	No	7583	
613	LP002990	Female	No	0	Graduate	Yes	4583	

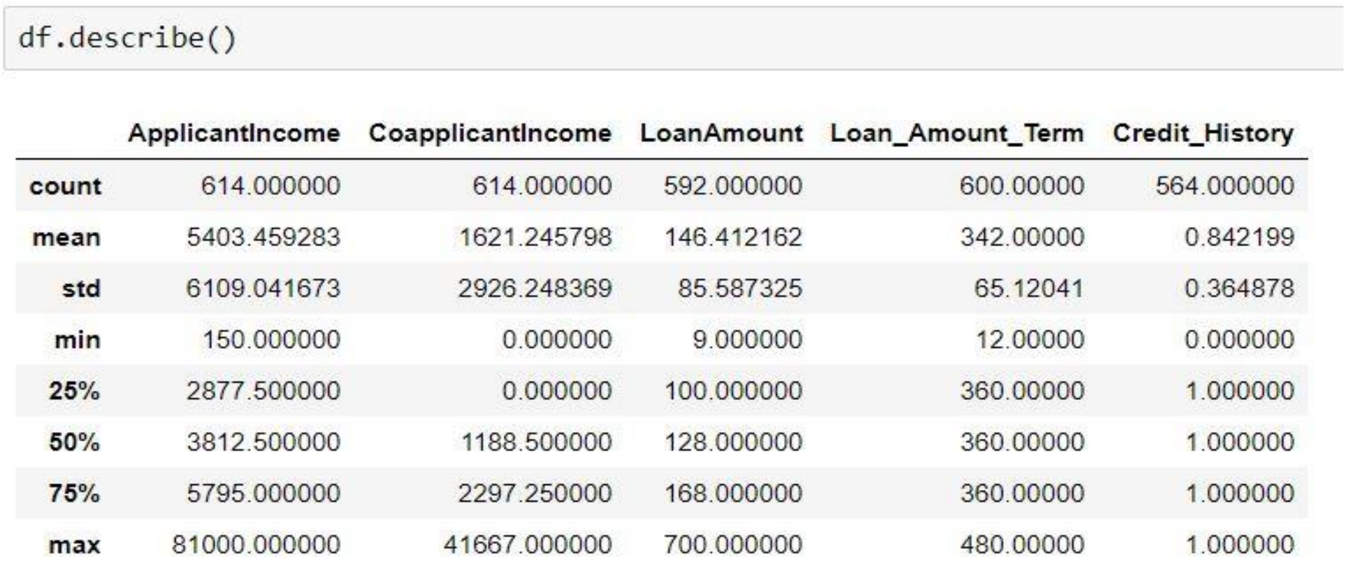
Explanation:: head method some sample data of all the columns from starting of the dataset while tail method show from ending of the data set.



```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 19 columns):
ApplicantIncome             614 non-null int64
CoapplicantIncome           614 non-null float64
LoanAmount                  614 non-null float64
Loan_Amount_Term            614 non-null float64
Credit_History              614 non-null float64
LoanAmount_log              614 non-null float64
TotalIncome                 614 non-null float64
TotalIncome_log             614 non-null float64
Loan_Amount_Term_cube       614 non-null float64
Gender_Male                 614 non-null uint8
Married_Yes                 614 non-null uint8
Dependents_1                614 non-null uint8
Dependents_2                614 non-null uint8
Dependents_3+               614 non-null uint8
Education_Not Graduate      614 non-null uint8
Self_Employed_Yes           614 non-null uint8
Property_Area_Semiurban     614 non-null uint8
Property_Area_Urban         614 non-null uint8
Loan_Status_Y               614 non-null uint8
dtypes: float64(8), int64(1), uint8(10)
memory usage: 49.2 KB
```

Explanation:: info method tell us the type of values contained in each column.



```
df.describe()
```

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
count	614.000000	614.000000	592.000000	600.00000	564.000000
mean	5403.459283	1621.245798	146.412162	342.00000	0.842199
std	6109.041673	2926.248369	85.587325	65.12041	0.364878
min	150.000000	0.000000	9.000000	12.00000	0.000000
25%	2877.500000	0.000000	100.000000	360.00000	1.000000
50%	3812.500000	1188.500000	128.000000	360.00000	1.000000
75%	5795.000000	2297.250000	168.000000	360.00000	1.000000
max	81000.000000	41667.000000	700.000000	480.00000	1.000000

Explanation:: describe method return us all the statistical information related to each column.

Observation:: Thus we have one output feature Loan_Status and the rest are input features.

Among the input features we have applicant ApplicantIncome, CoapplicantIncome, LoanAmount are continuous variable while Gender, Married, Dependents, Education, Self_Employed, Loan_Amount_Term, Credit_History, Property_Area are classification variable.

- ## Univariate Analysis:

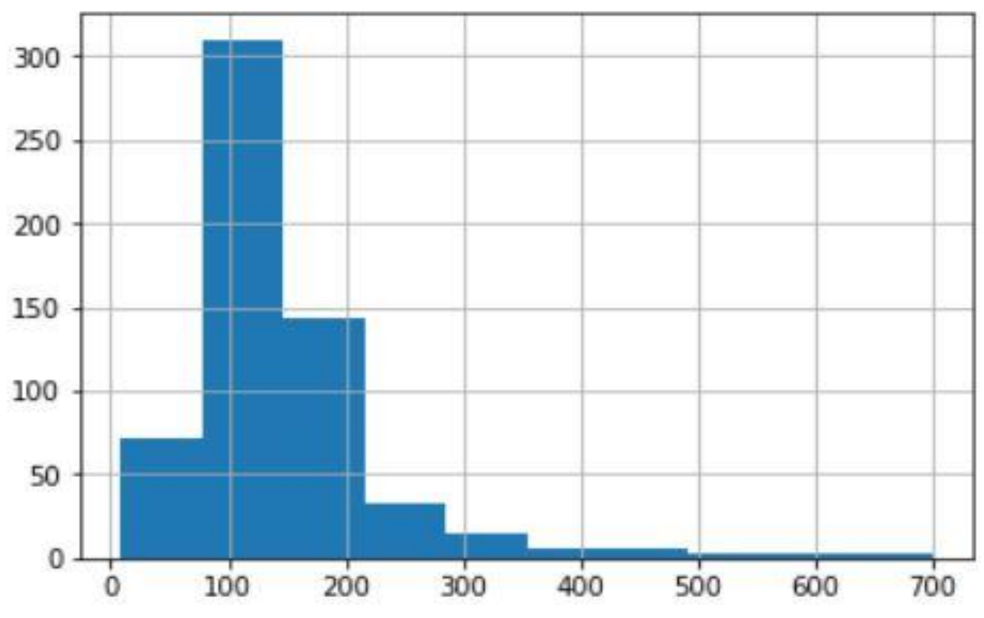
In this we analyze each columns(variable) using histogram and boxplot.

Independent Variable(Numerical):

Analysing Loan_Amount variable-

```
df['LoanAmount'].hist()
```

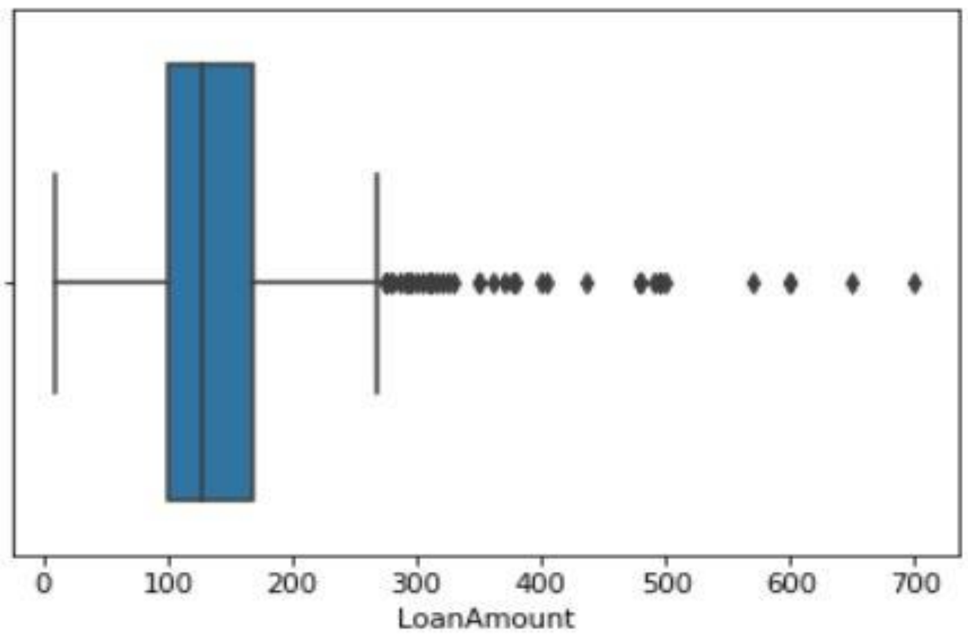
```
<matplotlib.axes._subplots.AxesSubplot at 0x274324d42e8>
```



Observation::It can be inferred that most of the data in the the distribution of Loan_amount is towards right which means it is not normally distributed.

```
sns.boxplot(x=df['LoanAmount'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x274327e1400>
```

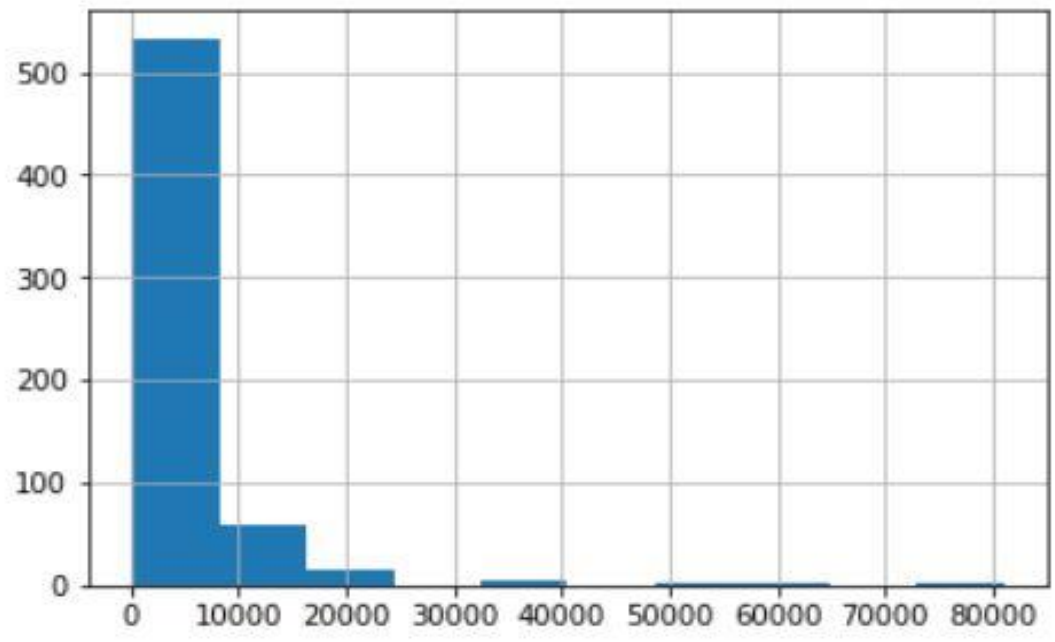


Observation:: presence of outlier/extreme values can be clearly seen.

Analysing ApplicantIncome variable-

```
df['ApplicantIncome'].hist()
```

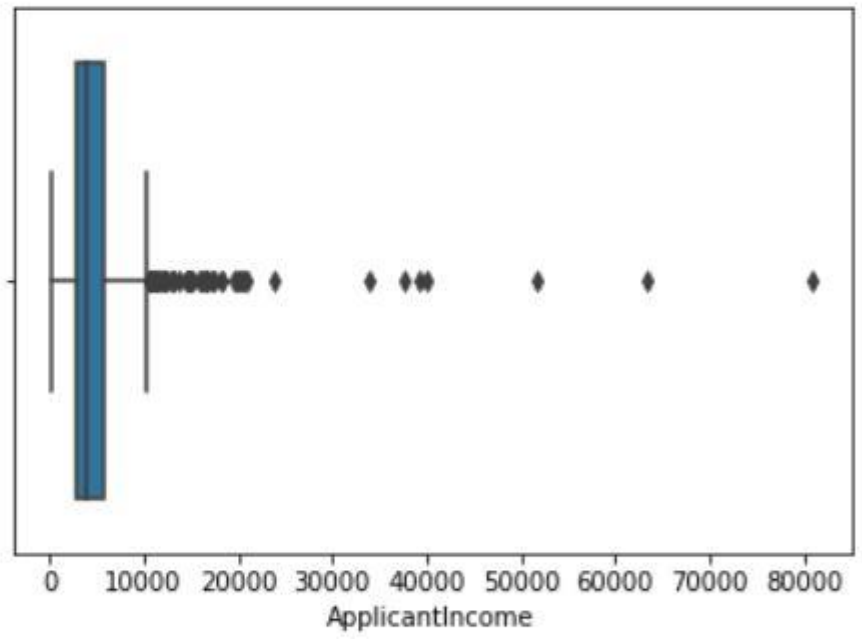
```
<matplotlib.axes._subplots.AxesSubplot at 0x27432a2004
```



Observation:: It can be said that ApplicantIncome data distribution is little more towards right.

```
sns.boxplot(x=df['ApplicantIncome'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x27432a88780>
```

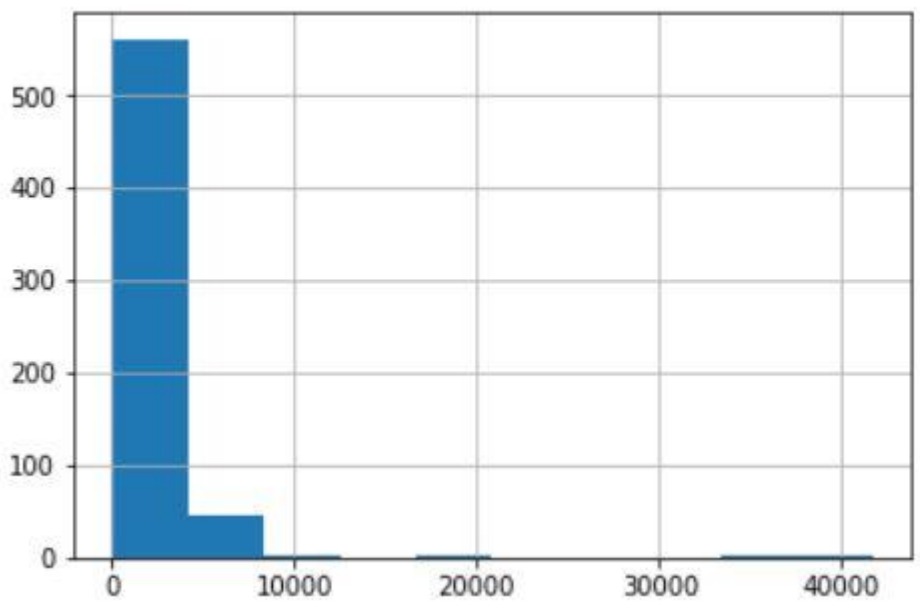


Observation:: presence of outlier/extreme values can be clearly seen.

Analysing CoapplicantIncome variable-

```
df['CoapplicantIncome'].hist()
```

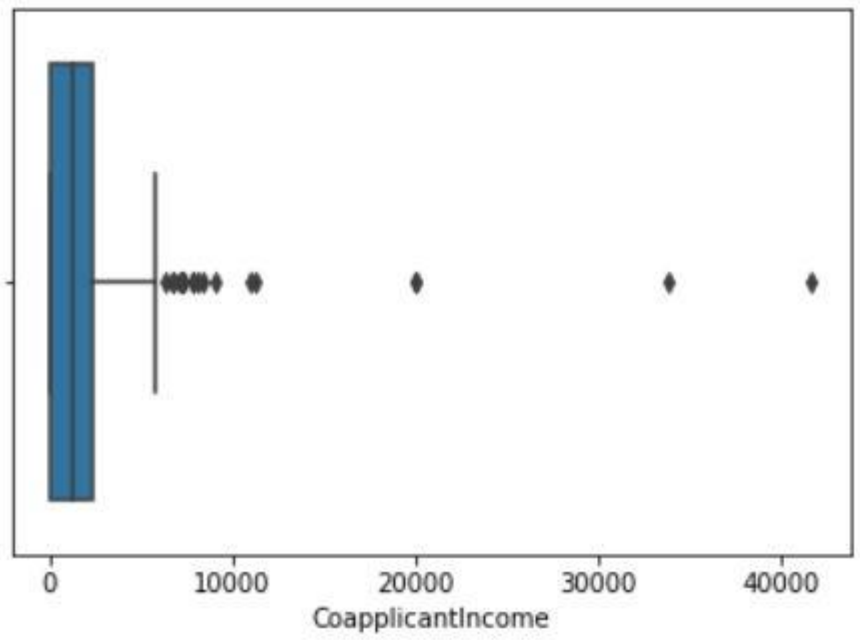
```
<matplotlib.axes._subplots.AxesSubplot at 0x27432aeb860>
```



Observation:: CoapplicantIncome data is fairly distributed.

```
sns.boxplot(x=df['CoapplicantIncome'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x27432b55ba8>
```



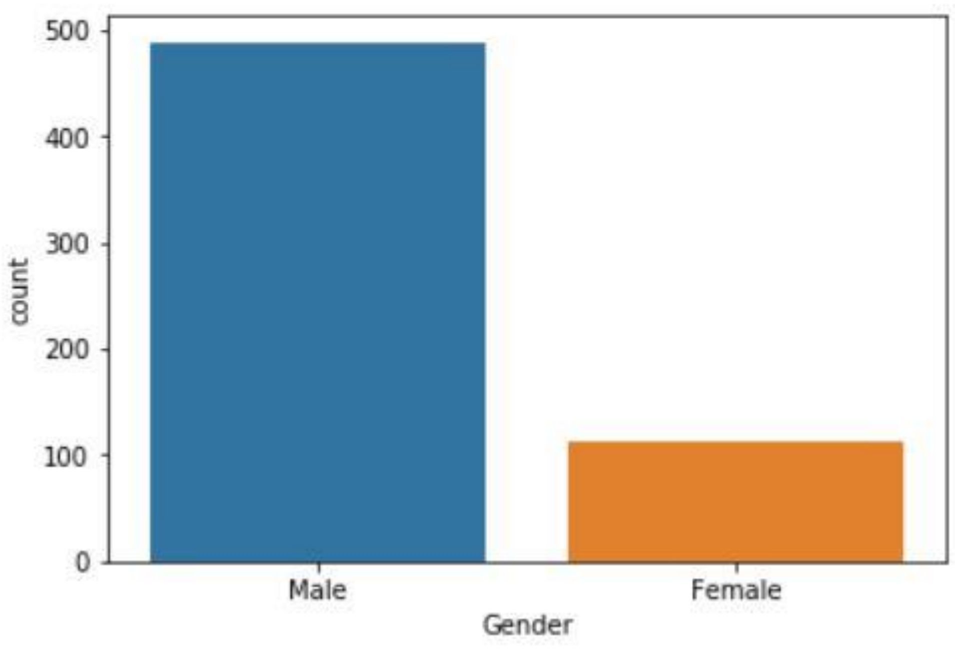
Observation:: Coapplicant Income has a high no. of outliers.

Independent Variable (Categorical):

Countplot of all Categorial columns(‘Gender’, ‘Married’, ‘Credit_history’, ‘Self_employed’, ‘dependents’, ‘Property_area’) are shown below-

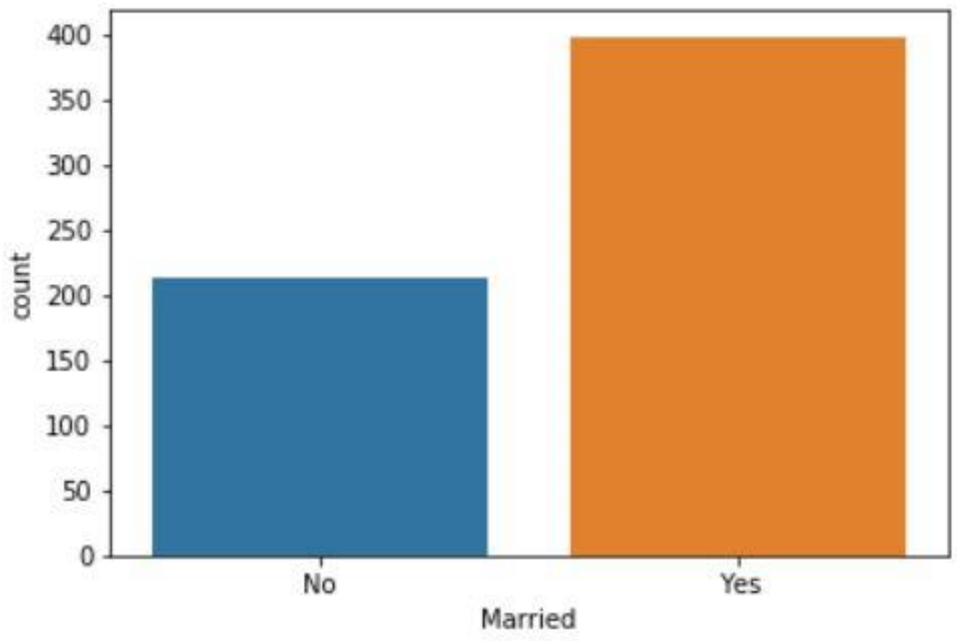
```
sns.countplot(df['Gender'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x27432dcf128>
```



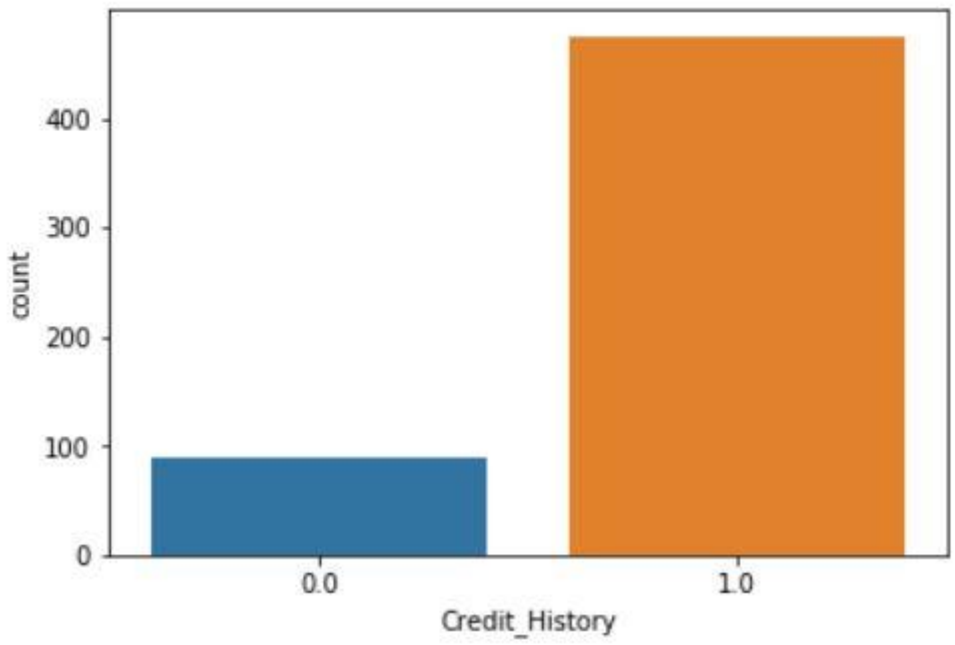
```
sns.countplot(df['Married'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x27432e53b38>
```



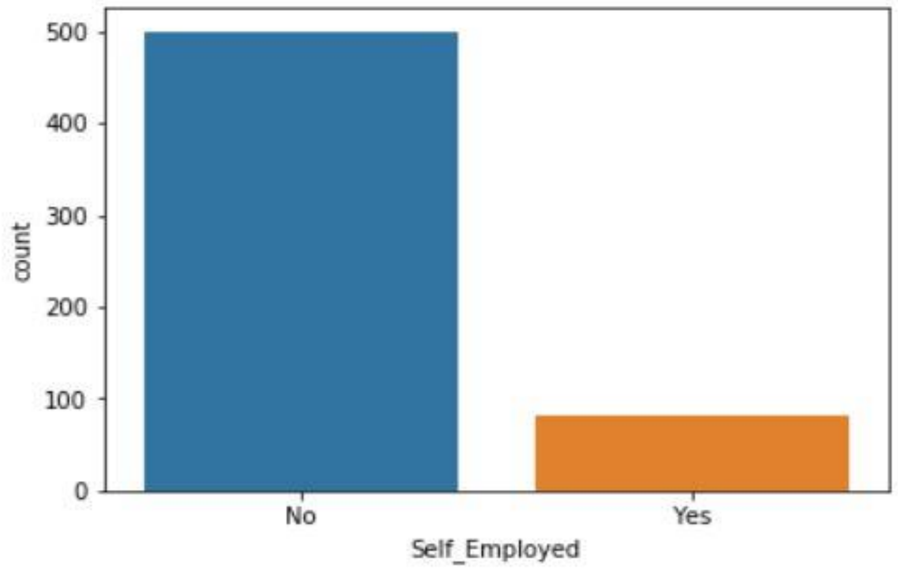
```
sns.countplot(df['Credit_History'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x27432ea72b0>
```



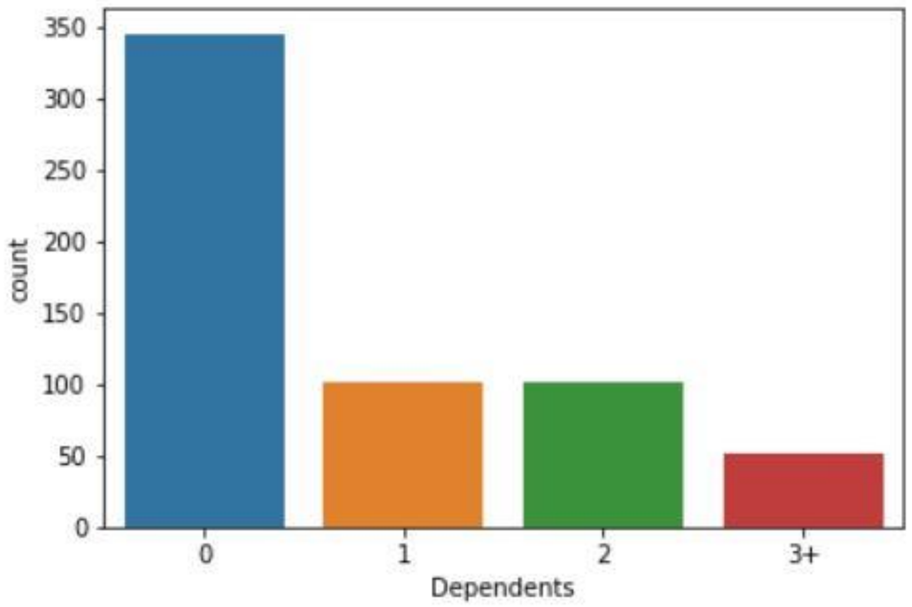
```
sns.countplot(df['Self_Employed'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x27432fa06a0>
```



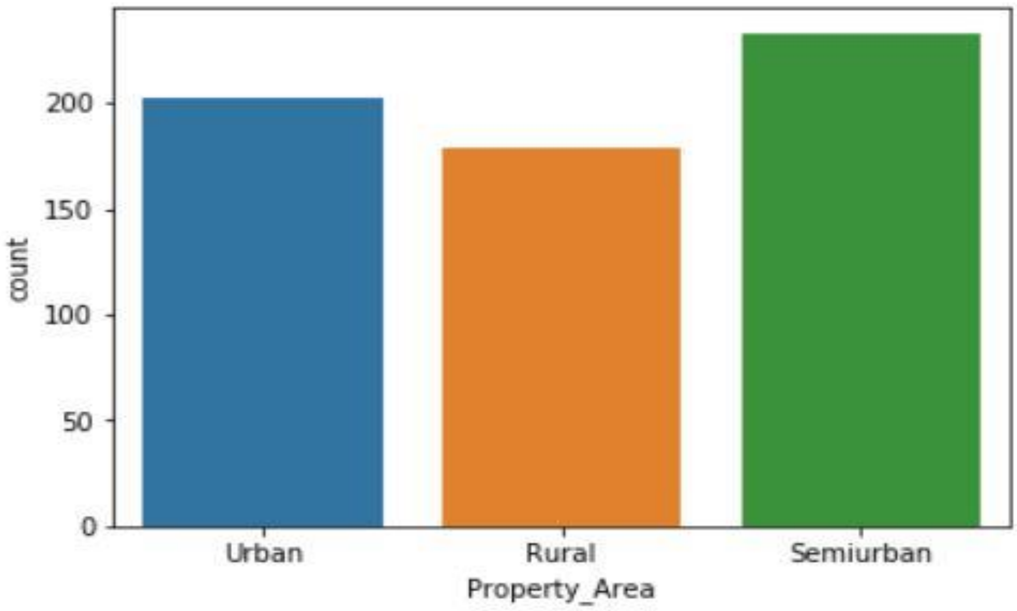
```
sns.countplot(df['Dependents'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x27432f454a8>
```



```
sns.countplot(df['Property_Area'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2743419a5f8>
```



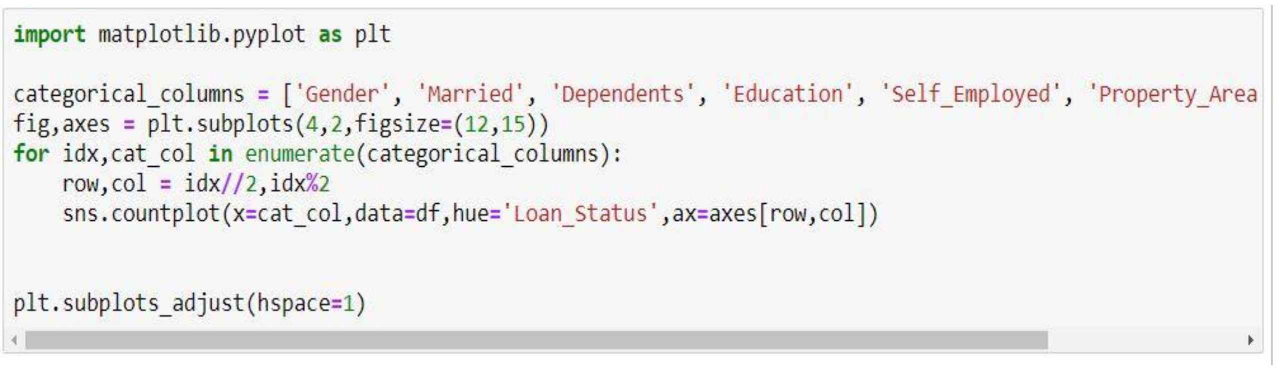
It can be inferred from the above the plots that:

- **80% of the applicants are male and rest are female.**
- **Around 65% of the applicants in the dataset are married.**
- **Around 15% of the applicants in the dataset are self employed.**
- **Around 85% applicants have repaid their debts.**
- **Around 80% of the applicants are Graduate.**
- **Most of the applicants are from Semi-urban area.**

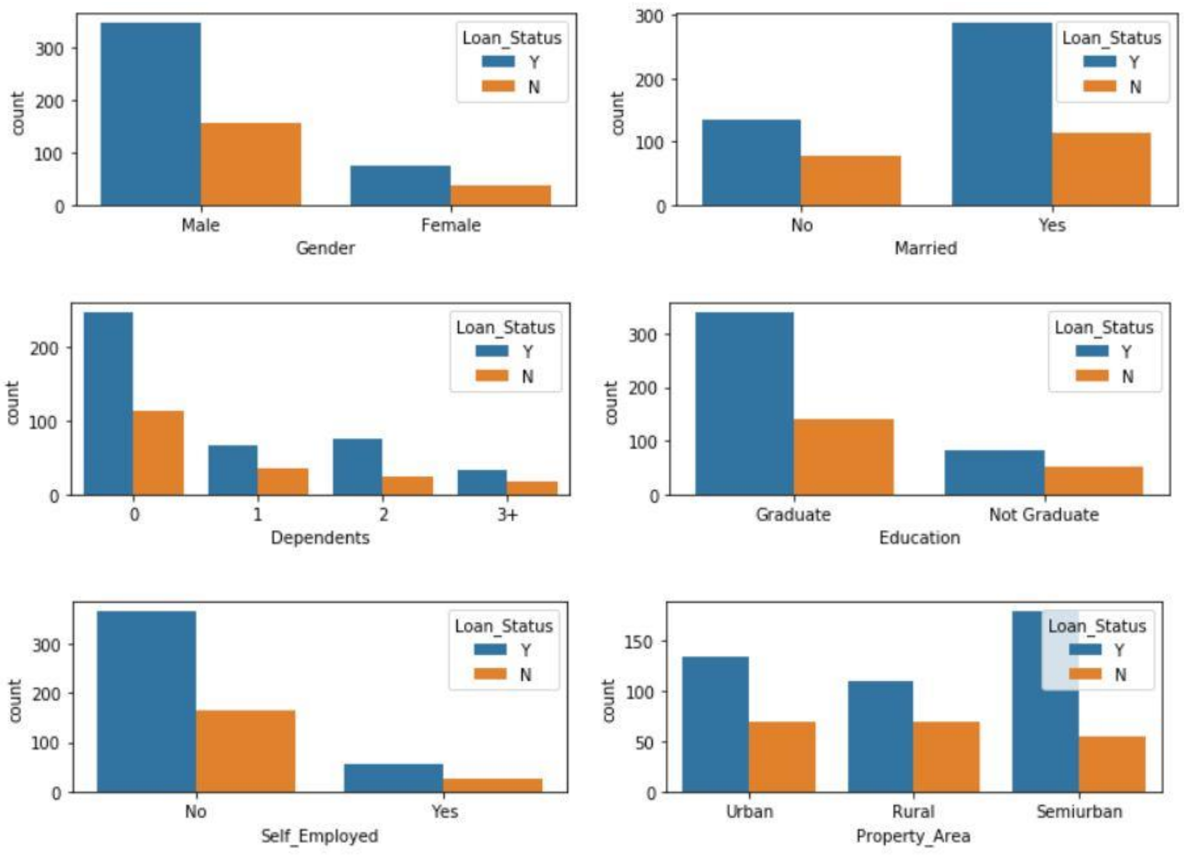
➢ Bivariate Analysis:

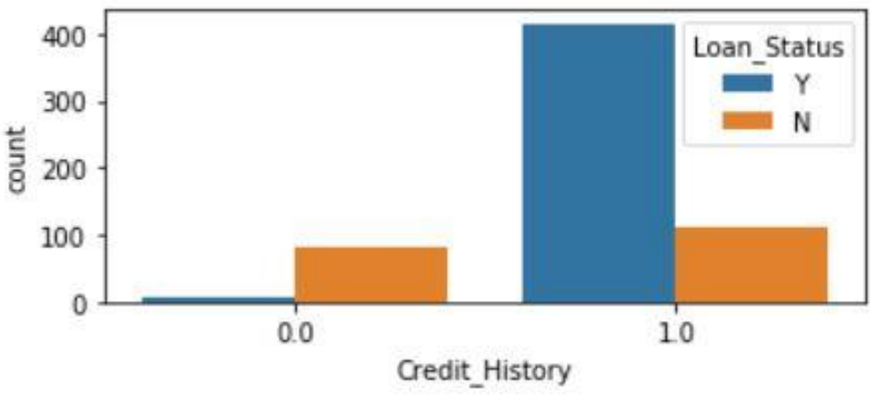
After looking at every variable individually in univariate analysis, we will now explore them again with respect to the target variable.

Categorical Independent Variable vs Target Variable



```
import matplotlib.pyplot as plt

categorical_columns = ['Gender', 'Married', 'Dependents', 'Education', 'Self_Employed', 'Property_Area
fig,axes = plt.subplots(4,2,figsize=(12,15))
for idx,cat_col in enumerate(categorical_columns):
    row,col = idx//2,idx%2
    sns.countplot(x=cat_col,data=df,hue='Loan_Status',ax=axes[row,col])


plt.subplots_adjust(hspace=1)
```





```
print("----Gender column analysis----")
count =0
for i in range(0,614):
   if df.at[i,'Gender']=='Male' and df.at[i,'Loan_Status']=='Y':
      count+=1
print("The percentage of male applicants whose loan got
passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
   if df.at[i,'Gender']=='Female' and df.at[i,'Loan_Status']=='Y':
      count+=1
print("The percentage of female applicants whose loan got
passed:",(count/614)*100,"%")

print("----Married column analysis----")
count =0
for i in range(0,614):
   if df.at[i,'Married']=='Yes' and df.at[i,'Loan_Status']=='Y':
      count+=1
print("The percentage of Married applicants whose loan got
passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
   if df.at[i,'Married']=='No' and df.at[i,'Loan_Status']=='Y':
      count+=1
print("The percentage of unmarried applicants whose loan got
passed:",(count/614)*100,"%")
```

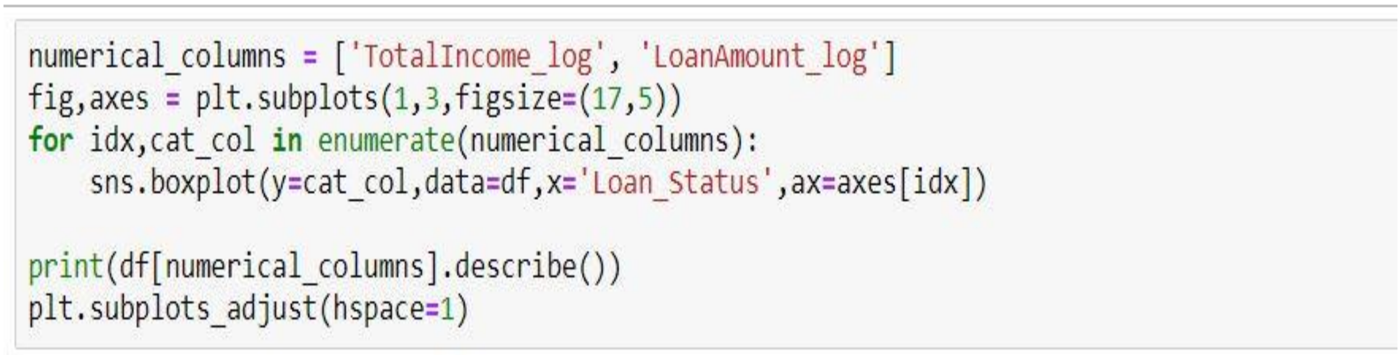
```
print("----Dependents column analysis----")
count =0
for i in range(0,614):
    if df.at[i,'Dependents']=='0' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of 0 Dependents applicants whose loan got
passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
    if df.at[i,'Dependents']=='1' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of 1 Dependents applicants whose loan got
passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
    if df.at[i,'Dependents']=='2' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of 2 Dependents applicants whose loan got
passed:",(count/614)*100,"%")

print("----Education column analysis----")
count =0
for i in range(0,614):
    if df.at[i,'Education']=='Graduate' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of Graduate applicants whose loan got
passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
    if df.at[i,'Education']=='Not Graduate' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of Not Graduate applicants whose loan got
passed:",(count/614)*100,"%")

print("----Self_Employed column analysis----")
count =0
for i in range(0,614):
    if df.at[i,'Self_Employed']=='Yes' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of Self Employed applicants whose loan got
passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
    if df.at[i,'Self_Employed']=='No' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of non Self Employed applicants whose loan got
passed:",(count/614)*100,"%")

print("----Property_Area column analysis----")
count =0
for i in range(0,614):
    if df.at[i,'Property_Area']=='Urban' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of Urban applicants whose loan got
passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
    if df.at[i,'Property_Area']=='Rural' and df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of Rural applicants whose loan got
passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
    if df.at[i,'Property_Area']=='Semiurban' and
df.at[i,'Loan_Status']=='Y':
        count+=1
print("The percentage of Semiurban applicants whose loan got
passed:",(count/614)*100,"%")

print("----Credit_History column analysis----")
count =0
for i in range(0,614):
    if df.at[i,'Credit_History']==0.0 and df.at[i,'Loan_Status']=='Y':
        count+=1
```

```
print("The percentage of Credit_History(0.0) applicants whose loan got passed:",(count/614)*100,"%")
count =0
for i in range(0,614):
   if df.at[i,'Credit_History']==1.0 and df.at[i,'Loan_Status']=='Y':
      count+=1
print("The percentage of Credit_History(1.0) applicants whose loan got passed:",(count/614)*100,"%")
```

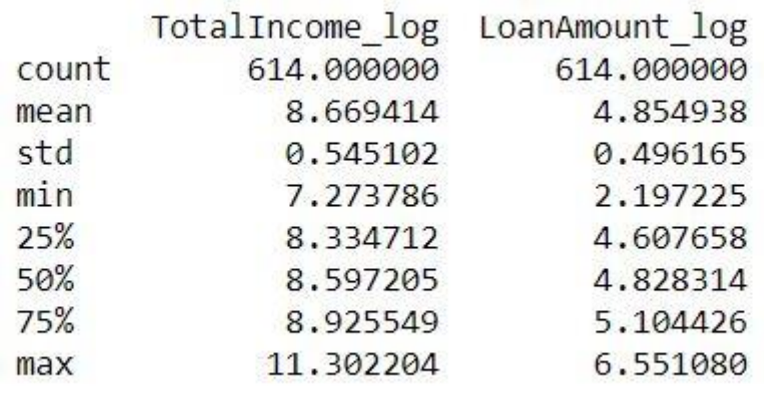
```
----Gender column analysis----
The percentage of male applicants whose loan got passed: 56.51465798045603 %
The percentage of female applicants whose loan got passed: 12.214983713355048 %
----Married column analysis----
The percentage of Married applicants whose loan got passed: 46.90553745928339 %
The percentage of unmarried applicants whose loan got passed: 21.824104234527688 %
----Dependents column analysis----
The percentage of 0 Dependents applicants whose loan got passed: 40.22801302931596 %
The percentage of 1 Dependents applicants whose loan got passed: 10.749185667752444 %
The percentage of 2 Dependents applicants whose loan got passed: 12.37785016286645 %
----Education column analysis----
The percentage of Graduate applicants whose loan got passed: 55.37459283387622 %
The percentage of Not Graduate applicants whose loan got passed: 13.355048859934854 %
----Self_Employed column analysis----
The percentage of Self Employed applicants whose loan got passed: 9.120521172638437 %
The percentage of non Self Employed applicants whose loan got passed: 59.60912052117264 %
----Property_Area column analysis----
The percentage of Urban applicants whose loan got passed: 21.661237785016286 %
The percentage of Rural applicants whose loan got passed: 17.91530944625407 %
The percentage of Semiurban applicants whose loan got passed: 29.153094462540718 %
----Credit_History column analysis----
The percentage of Credit_History(0.0) applicants whose loan got passed: 1.1400651465798046 %
The percentage of Credit_History(1.0) applicants whose loan got passed: 67.58957654723126 %
```

From the above data we could see each categorical relation with target variable. Also the percentage of each category who loan could get passed is also observed.

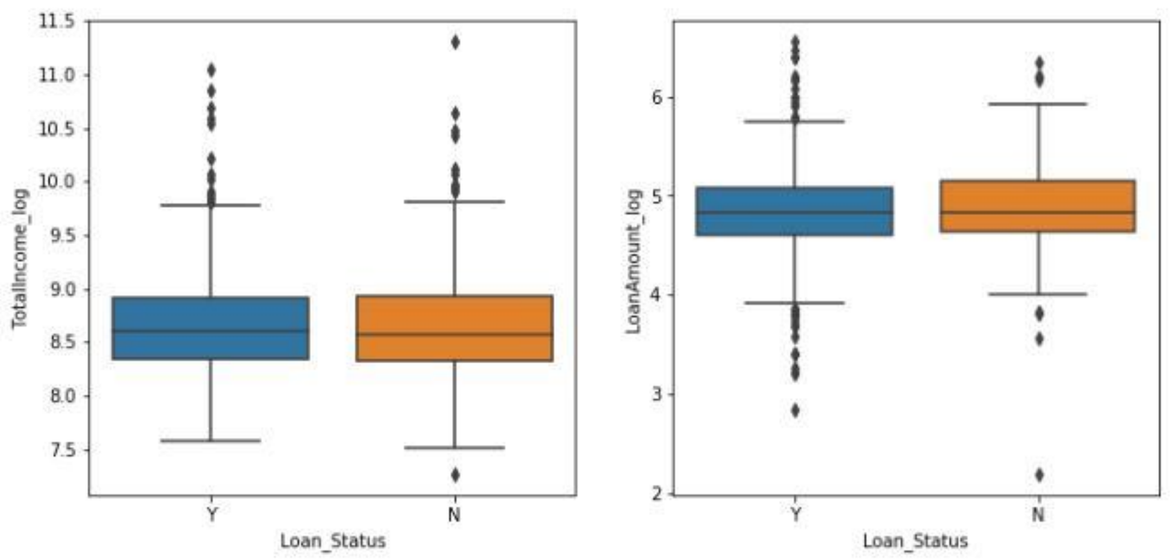
Numerical Independent Variable vs Target Variable



```python
numerical_columns = ['TotalIncome_log', 'LoanAmount_log']
fig,axes = plt.subplots(1,3,figsize=(17,5))
for idx,cat_col in enumerate(numerical_columns):
    sns.boxplot(y=cat_col,data=df,x='Loan_Status',ax=axes[idx])

print(df[numerical_columns].describe())
plt.subplots_adjust(hspace=1)
```



```
       TotalIncome_log  LoanAmount_log
count       614.000000      614.000000
mean          8.669414        4.854938
std           0.545102        0.496165
min           7.273786        2.197225
25%           8.334712        4.607658
50%           8.597205        4.828314
75%           8.925549        5.104426
max          11.302204        6.551080
```



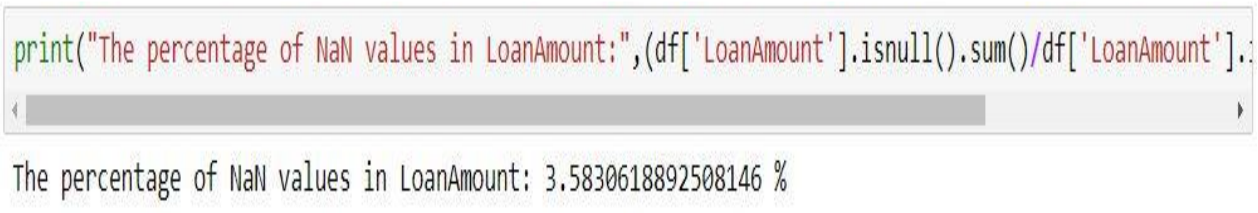
TotalIncome_log
Loan_Status
Y
N
LoanAmount_log
Loan_Status
Y
N

➢ Missing value And Outlier Treatment :

After exploring all the variables in our data, we can now impute the missing values and treat the outliers because missing data and outliers can have adverse effect on the model performance.

For Loan Amount:

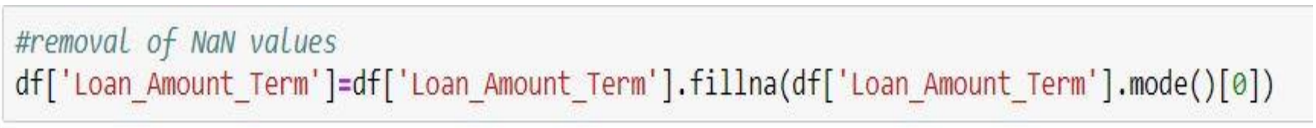
Finding the percentage of missing(NaN) values in loan amount variable :



```
print("The percentage of NaN values in LoanAmount:",(df['LoanAmount'].isnull().sum()/df['LoanAmount'].
```

```
The percentage of NaN values in LoanAmount: 3.5830618892508146 %
```

Removing(NaN) values in loan amount variable:



```
#removal of NaN values
df['Loan_Amount_Term']=df['Loan_Amount_Term'].fillna(df['Loan_Amount_Term'].mode()[0])
```

Detecting outliers:

```
dataset = df['LoanAmount']
def detect_outlier(data_1):
    count = 0
    threshold=3
    mean_1 = np.mean(data_1)
    std_1 =np.std(data_1)



    for y in data_1:
        z_score= (y - mean_1)/std_1
        if np.abs(z_score) > threshold:
            count = count+1
    return count
no_of_outliers = detect_outlier(dataset)
print("The percentage of outliers values in LoanAmount:",(no_of_outliers/df['LoanAmount'].isnull().count())*100,"%")
```

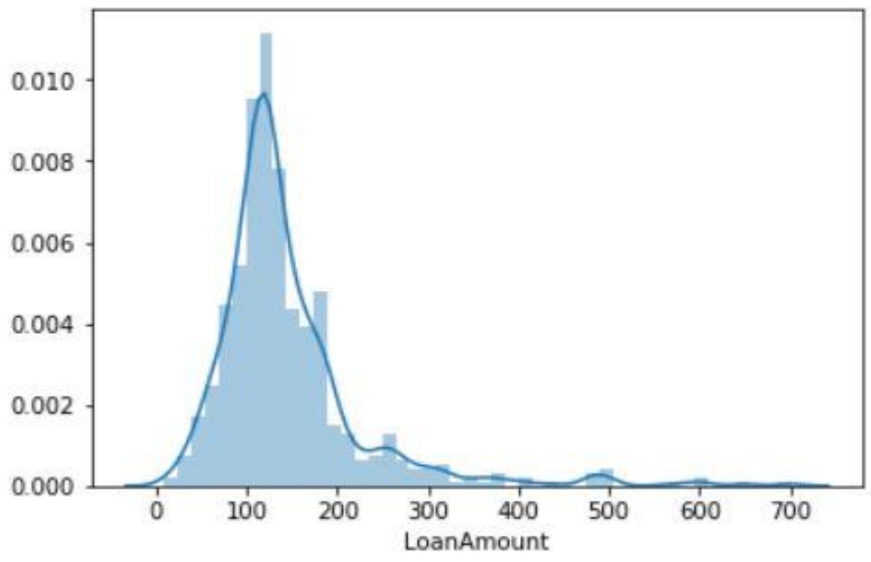
```
The percentage of outliers values in LoanAmount: 2.2801302931596092 %
```

since no. of ouliers are very less its not necessary to remove them . Just by removing the skewness we can handle the outliers.

Removing skewness in loan amount variable:

```
sns.distplot(df['LoanAmount'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x246d3143438>
```



```
df['LoanAmount'].skew()
```

```
2.7454067641709576
```

Clearly from the above the above displot we could observe the data is not distributed well. Also the skewness is high. So we will reduce the skewness.

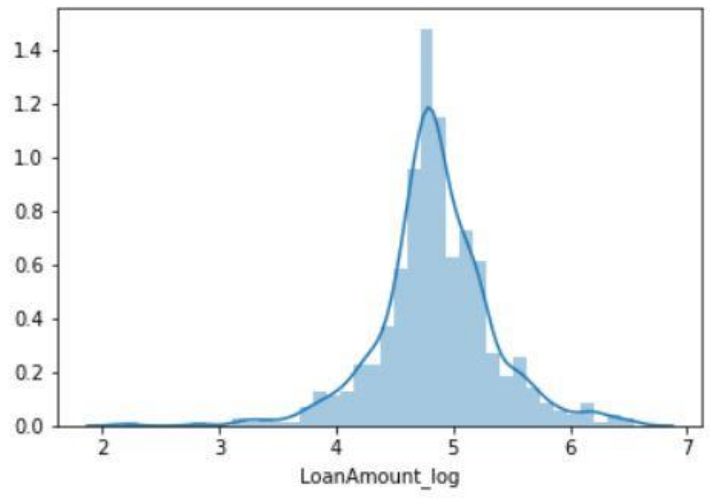
Reducing the skewness:

```
df['LoanAmount_log'] = np.log(df['LoanAmount'])
print(df['LoanAmount_log'].skew())
df.drop('LoanAmount', axis=1, inplace= True)
```

```
-0.18122724931425552
```

```
sns.distplot(df['LoanAmount_log'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x246d321a208>
```



Clearly by taking the log of the value of loan amount variable we could see the skewness is greatly reduced. Also from displot we could observe our data now become well distributed and gives a bell curve.

For ApplicantIncome:

Finding the percentage of missing(NaN) values in Applicant variable :

```
print("The percentage of NaN values in ApplicantIncome:",(df['ApplicantIncome'].isnull().sum()/df['ApplicantIncome'].isnull().count())*100,"%")
```

```
The percentage of NaN values in ApplicantIncome: 0.0 %
```

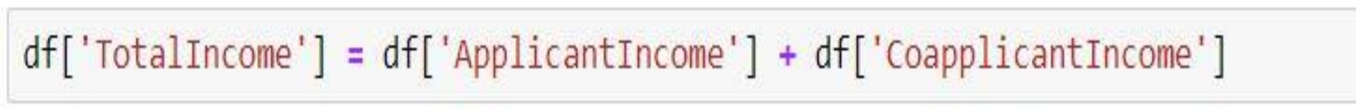
For CoapplicantIncome:

Finding the percentage of missing(NaN) values in Coapplicant variable :

```
print("The percentage of NaN values in CoapplicantIncome:",(df['CoapplicantIncome'].isnull().sum()/df['CoapplicantIncome'].isnull().count())*100,"%")
```

```
The percentage of NaN values in CoapplicantIncome: 0.0 %
```

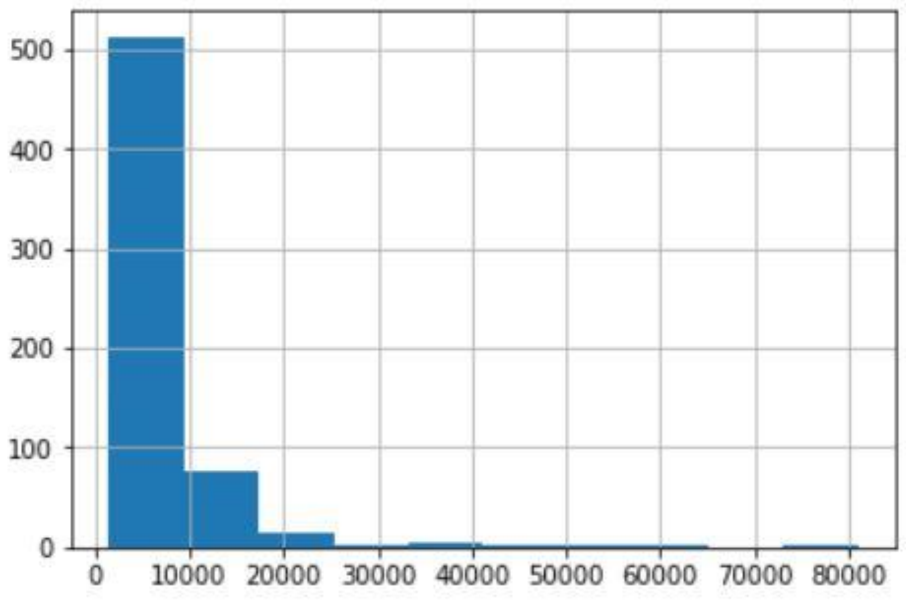
Since there are lot of 0.0 values in our CoapplicantIncome.
This value was increasing the skewness . Hence to decrease the skewness and get a well distribution of data, we join both the column ApplicantIncome and CoapplicantIncome and merge into a new column name “TotalIncome”.



```
df['TotalIncome'] = df['ApplicantIncome'] + df['CoapplicantIncome']
```

```python
df['TotalIncome'].hist()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x246d34d4978>
```



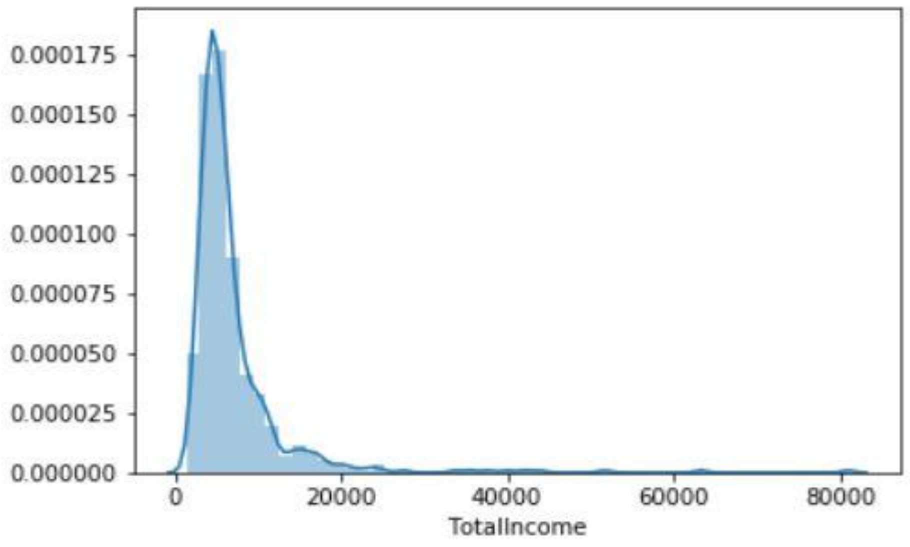
```
df['TotalIncome'].skew()
```

```
5.633448514884535
```

Still TotalIncome has a great amount of skewness. So we will be reducing the skewness below:

```
sns.distplot(df['TotalIncome'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x246d3548518>
```



```
df['TotalIncome_log'] = np.log(df['TotalIncome'])
print(df['TotalIncome_log'].skew())
df.drop('TotalIncome', axis=1, inplace= True)
df.drop('ApplicantIncome', axis=1, inplace= True)
df.drop('CoapplicantIncome', axis=1, inplace= True)
```

```
1.0767023443049968
```

Skewness of TotalIncome variable is greatly reduced by taking a log of that variable.

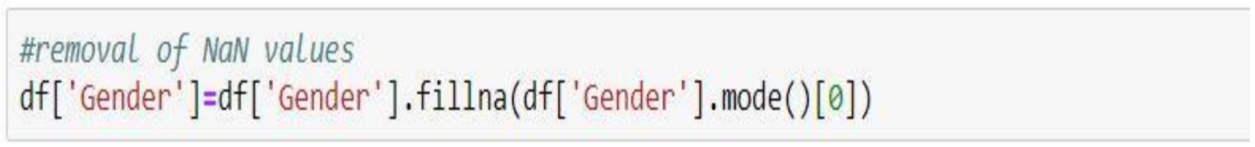
For Gender:

Finding the percentage of missing(NaN) values in Gender variable :

```
print("The percentage of NaN values in Gender:",(df['Gender'].isnull().sum()/df['Gender'].isnull().count())*100,"%")
```

```
The percentage of NaN values in Gender: 2.1172638436482085 %
```

Removing(NaN) values in Gender variable:



```
#removal of NaN values
df['Gender']=df['Gender'].fillna(df['Gender'].mode()[0])
```

For Married:

Finding the percentage of missing(NaN) values in Married variable :

```
print("The percentage of NaN values in Married:",(df['Married'].isnull().sum()/df['Married'].isnull().count())*100,"%")
```

```
The percentage of NaN values in Married: 0.4885993485342019 %
```

Removing(NaN) values in Married variable:

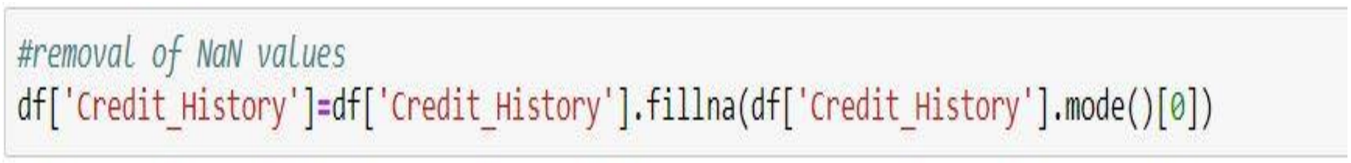
For Credit_History:

Finding the percentage of missing(NaN) values in Married variable :

```
print("The percentage of NaN values in Credit_History:",(df['Credit_History'].isnull().sum()/df['Credit_History'].isnull().count())*100,"%")
```

```
The percentage of NaN values in Credit_History: 8.143322475570033 %
```

Removing(NaN) values in Credit_History variable:



```
#removal of NaN values
df['Credit_History']=df['Credit_History'].fillna(df['Credit_History'].mode()[0])
```

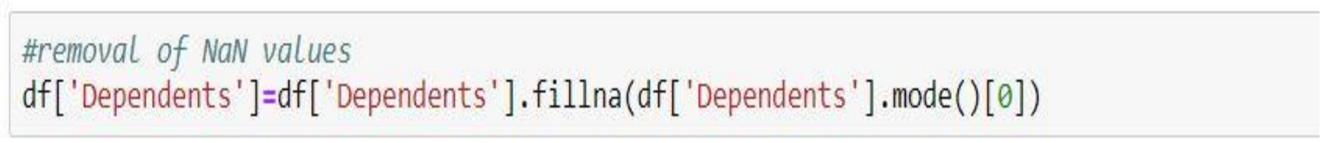
For Dependents:

Finding the percentage of missing(NaN) values in Dependents variable :

print("The percentage of NaN values in Dependents:",(df['Dependents'].isnull ().sum()/df['Dependents'].isnull().count())*100,"%")

```
The percentage of NaN values in Dependents: 2.44299674267101 %
```

Removing(NaN) values in Dependents variable:



```
#removal of NaN values
df['Dependents']=df['Dependents'].fillna(df['Dependents'].mode()[0])
```

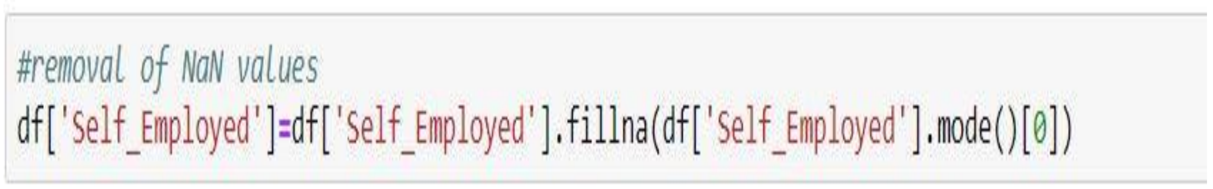
For Self_Employed:

Finding the percentage of missing(NaN) values in Self_Employed variable :

print("The percentage of NaN values in Self_Employed:",(df['Self_Employed'] .isnull().sum()/df['Self_Employed'].isnull().count())*100,"%")

```
The percentage of NaN values in Self_Employed: 5.211726384364821 %
```

Removing(NaN) values in Self_Employed variable:



```
#removal of NaN values
df['Self_Employed']=df['Self_Employed'].fillna(df['Self_Employed'].mode()[0])
```

For Property_Area:

Finding the percentage of missing(NaN) values in Self_Employed variable :

```
print("The percentage of NaN values in Property_Area:",(df['Property_Area'].isnull().sum()/df['Property_Area'].isnull().count())*100,"%")
```

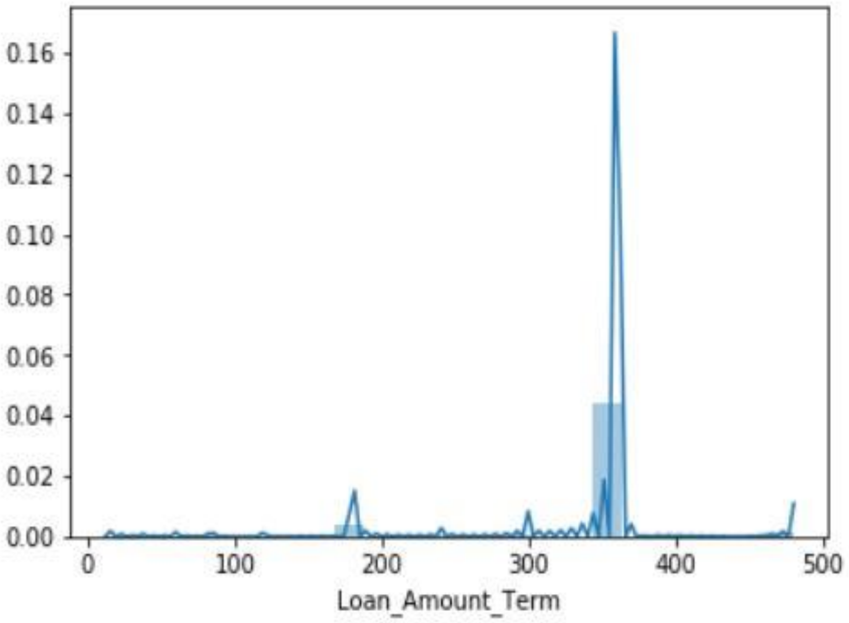
Since there is no   Nan values in property_area variable .Hence treat moment of Property_Area is not a nice thing.

For Loan_Amount_Term:

Finding the percentage of missing(NaN) values in Loan_ amount_Term variable :

```
sns.distplot(df['Loan_Amount_Term'])
```

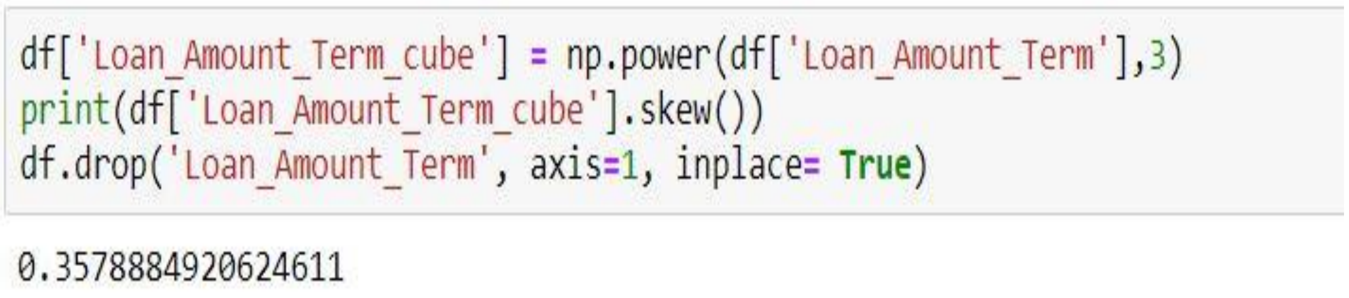
```
<matplotlib.axes._subplots.AxesSubplot at 0x246d39ad2b0>
```



Removing the Skewness of loan_Amount_Term variable:

```
df['Loan_Amount_Term'].skew()
```

```
-2.402112256389039б
```



```
df['Loan_Amount_Term_cube'] = np.power(df['Loan_Amount_Term'],3)
print(df['Loan_Amount_Term_cube'].skew())
df.drop('Loan_Amount_Term', axis=1, inplace= True)
```

```
0.3578884920624611
```

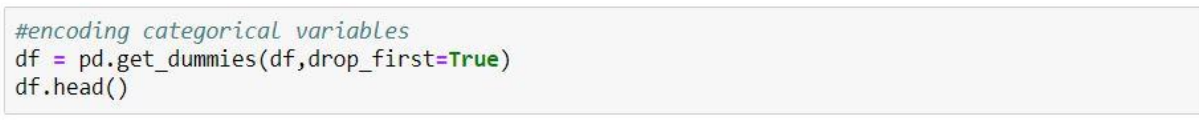
Ideally skewness of any variable should be within the range of -1 to +1

Earlier it was -2 , so to remove the skewness we take cube of the "Loan_Amount_Term_cube"

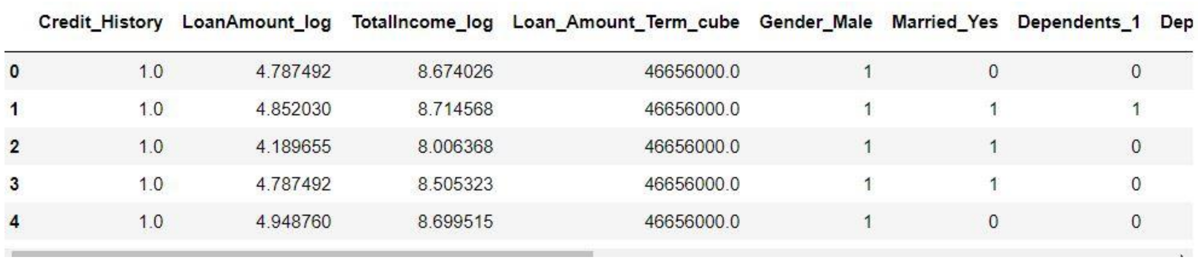
- **Variable Transformation:**

  After removing all the Skewness and perfoming data cleaning. We will now transform all our categorical values using one hot encoding method. Thus to perform one hot encoding for categorical variables dummies method is used.

  One –hot encoding categorical values-



```
#encoding categorical variables
df = pd.get_dummies(df,drop_first=True)
df.head()
```



	Credit_History	LoanAmount_log	TotalIncome_log	Loan_Amount_Term_cube	Gender_Male	Married_Yes	Dependents_1	Dep
0	1.0	4.787492	8.674026	46656000.0	1	0	0	
1	1.0	4.852030	8.714568	46656000.0	1	1	1	
2	1.0	4.189655	8.006368	46656000.0	1	1	0	
3	1.0	4.787492	8.505323	46656000.0	1	1	0	
4	1.0	4.948760	8.699515	46656000.0	1	0	0	

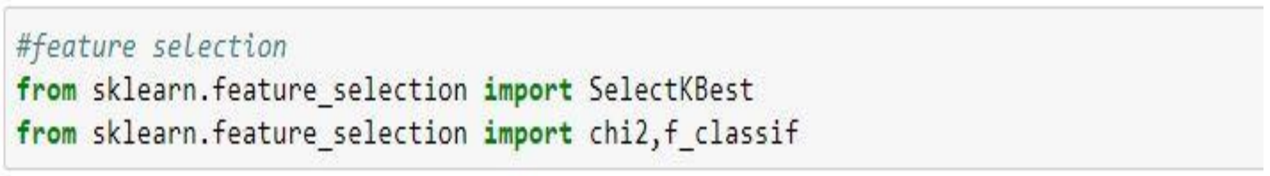
- **Feature Selection:**

Feature Selection is the process where you automatically or manually select those features which contribute most to your prediction variable or output in which you are interested in.

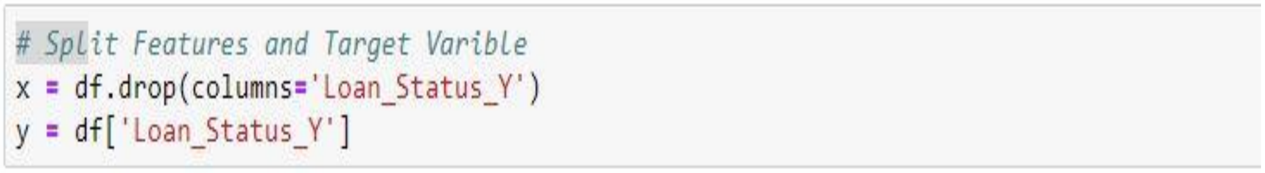
Having irrelevant features in your data can decrease the accuracy of the models and make your model learn based on irrelevant features.

Thus we will find the correlation among the various variables in the dataset. A high correlation among independent variable and target variable and low correlation among two independent variable is considered to be good.

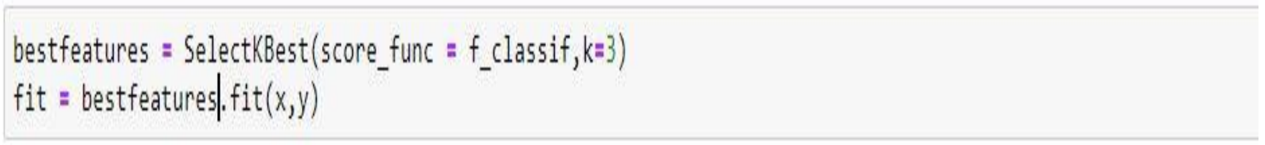
Using Select_K_best method we will find the correlation and select the best features required for our loan prediction.



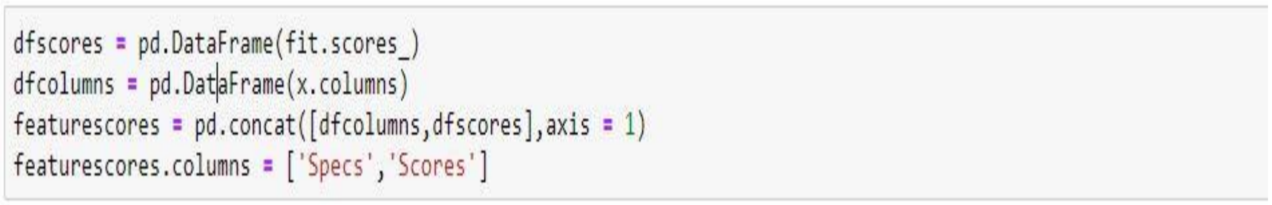
```
#feature selection
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2,f_classif
```



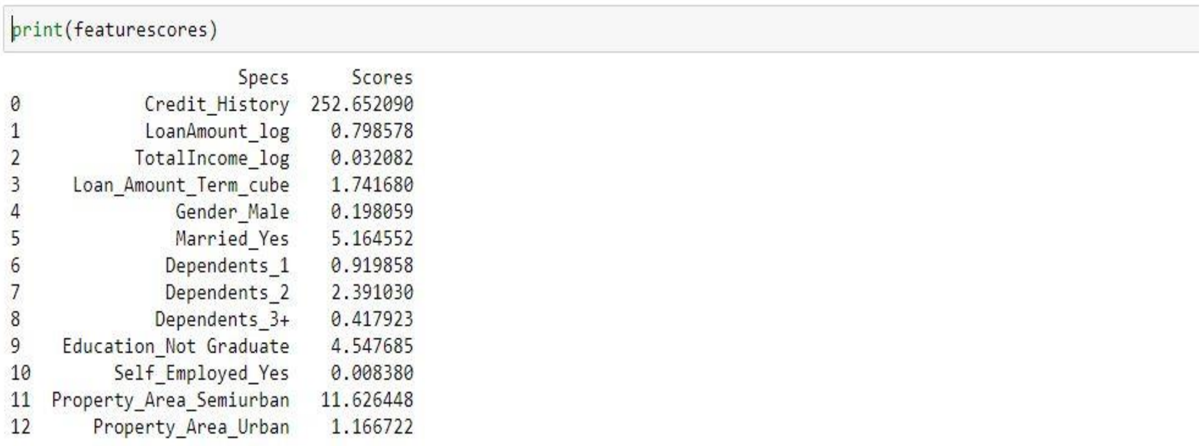
```
# Split Features and Target Varible
x = df.drop(columns='Loan_Status_Y')
y = df['Loan_Status_Y']
```



```
bestfeatures = SelectKBest(score_func = f_classif,k=3)
fit = bestfeatures.fit(x,y)
```

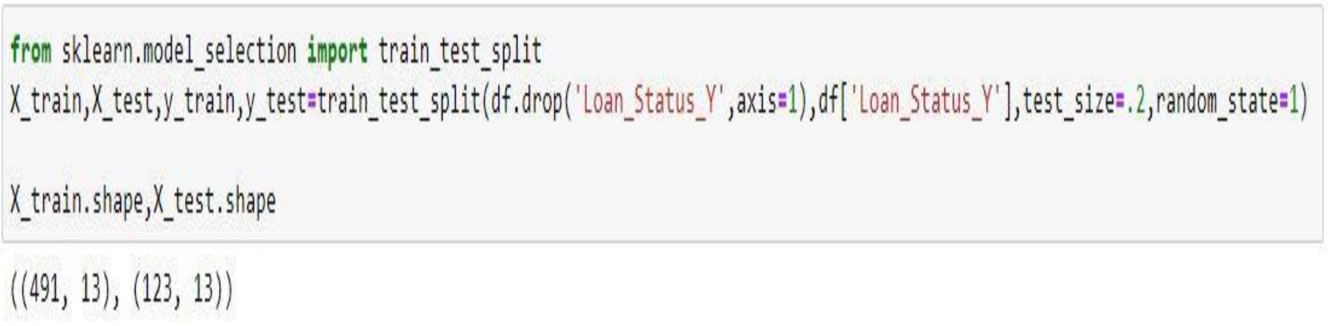


```
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(x.columns)
featurescores = pd.concat([dfcolumns,dfscores],axis = 1)
featurescores.columns = ['Specs','Scores']
```



```
print(featurescores)
```

```
                      Specs      Scores
0            Credit_History  252.652090
1            LoanAmount_log    0.798578
2            TotalIncome_log    0.032082
3       Loan_Amount_Term_cube    1.741680
4               Gender_Male    0.198059
5               Married_Yes    5.164552
6              Dependents_1    0.919858
7              Dependents_2    2.391030
8             Dependents_3+    0.417923
9      Education_Not Graduate    4.547685
10        Self_Employed_Yes    0.008380
11  Property_Area_Semiurban   11.626448
12      Property_Area_Urban    1.166722
```



```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(df.drop('Loan_Status_Y',axis=1),df['Loan_Status_Y'],test_size=.2,random_state=1)


X_train.shape,X_test.shape
```

```
((491, 13), (123, 13))
```

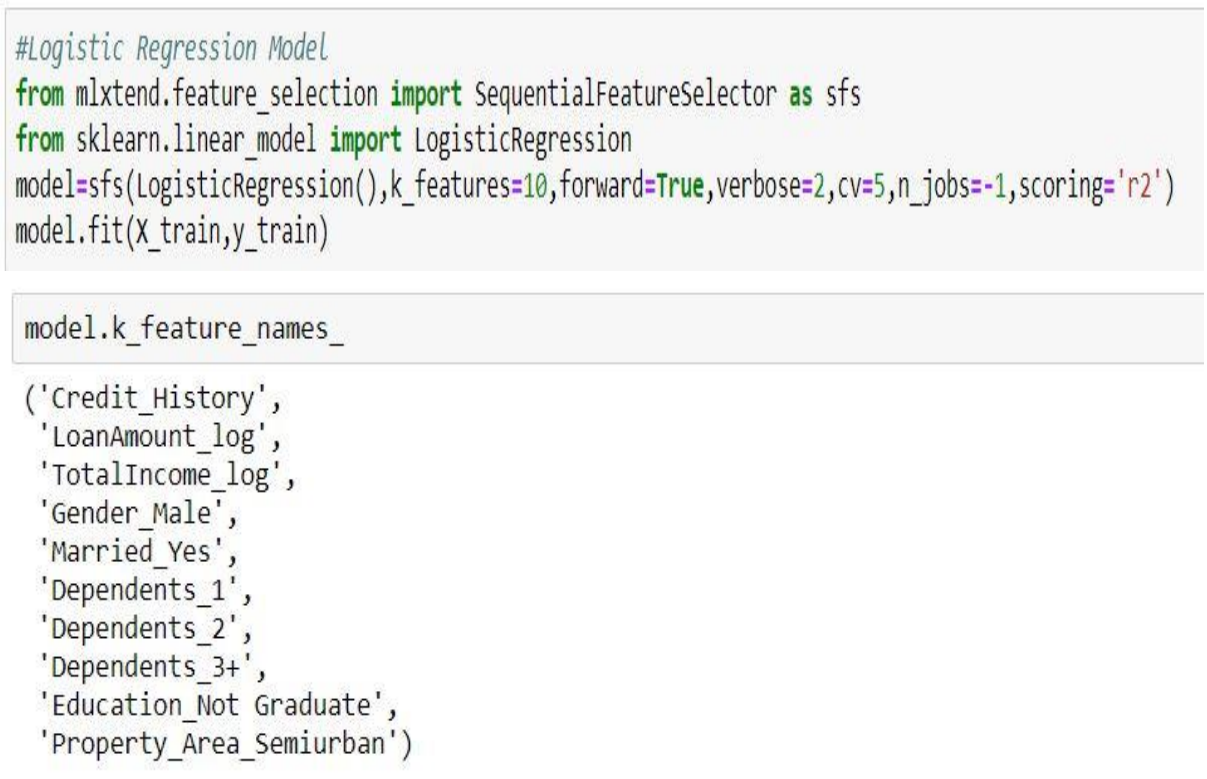
From above code we could observe that there is not very high correlation among independent variables and all independent variable are required for model building.

Model Building

After doing all the data cleaning and feature selection, we will now build different Model for predicting our loan Status.

1. Logistic Regression Model

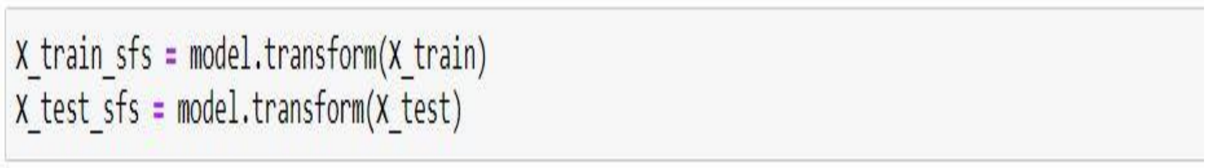
- **Logistic Regression is a classification algorithm. It is used to predict a binary outcome (1 / 0, Yes / No, True / False) given a set of independent variables.**
- **Logistic regression is an estimation of Logit function. Logit function is simply a log of odds in favor of the event.**
- **This function creates a s-shaped curve with the probability estimate, which is very similar to the required step wise function**



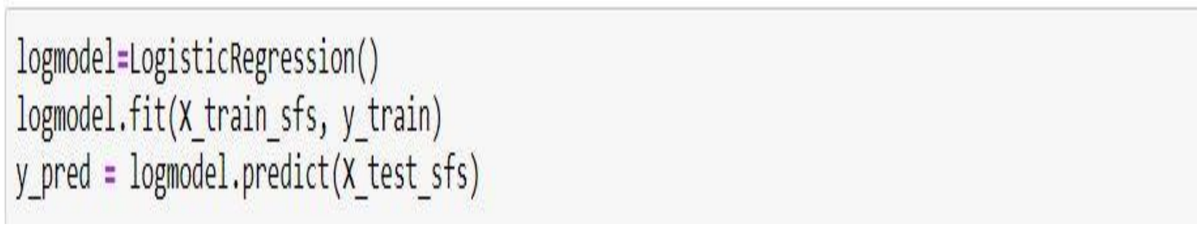
```
#Logistic Regression Model
from mlxtend.feature_selection import SequentialFeatureSelector as sfs
from sklearn.linear_model import LogisticRegression
model=sfs(LogisticRegression(),k_features=10,forward=True,verbose=2,cv=5,n_jobs=-1,scoring='r2')
model.fit(X_train,y_train)
```

```
model.k_feature_names_
```

```
('Credit_History',
 'LoanAmount_log',
 'TotalIncome_log',
 'Gender_Male',
 'Married_Yes',
 'Dependents_1',
 'Dependents_2',
 'Dependents_3+',
 'Education_Not Graduate',
 'Property_Area_Semiurban')
```



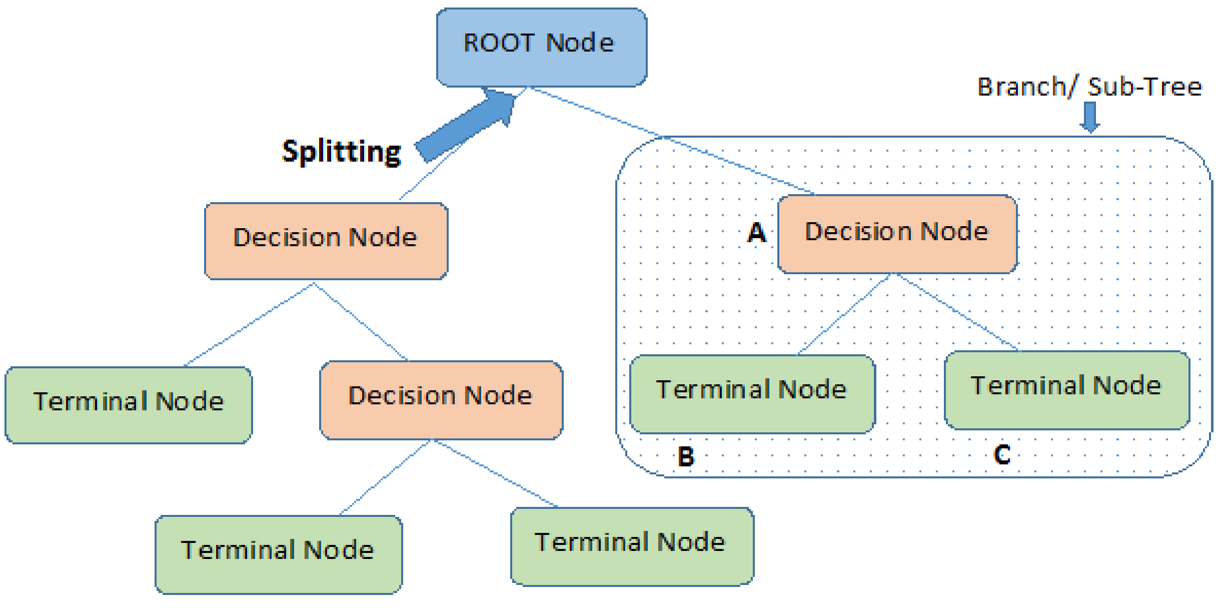
```
X_train_sfs = model.transform(X_train)
X_test_sfs = model.transform(X_test)
```



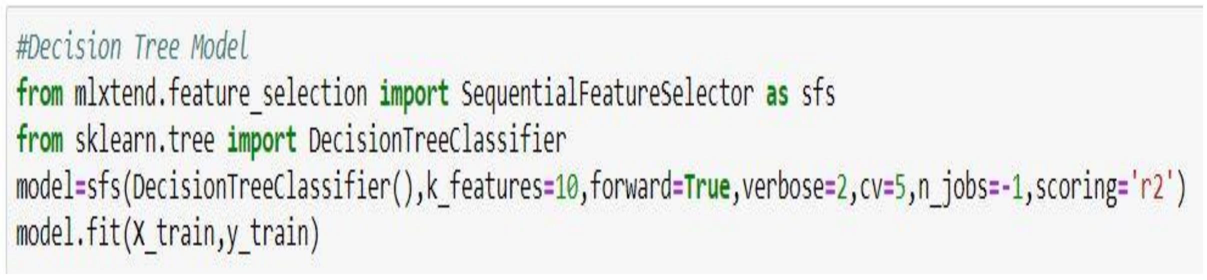
```
logmodel=LogisticRegression()
logmodel.fit(X_train_sfs, y_train)
y_pred = logmodel.predict(X_test_sfs)
```

2. Decision Tree Model

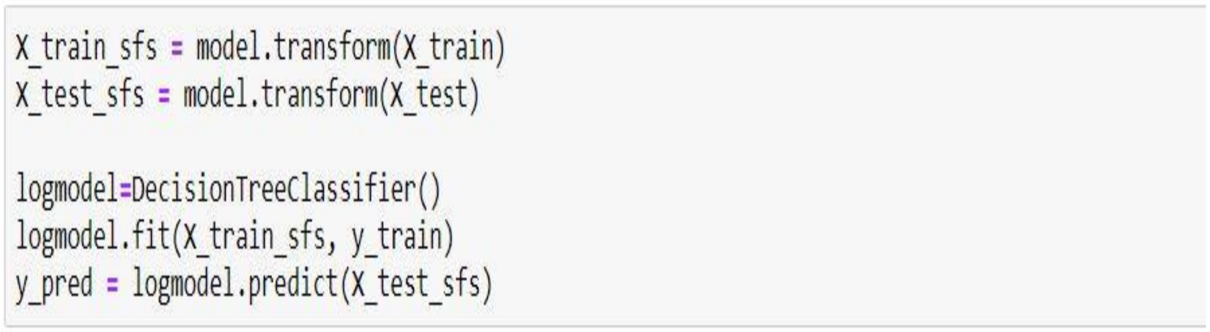
Decision tree is a type of supervised learning algorithm (having a pre-defined target variable) that is mostly used in classification problems. It works for both categorical and continuous input and output variables. In this technique, we split the population or sample into two or more homogeneous sets (or sub-populations) based on most significant splitter / differentiator in input variables.



Note:- A is parent node of B and C.



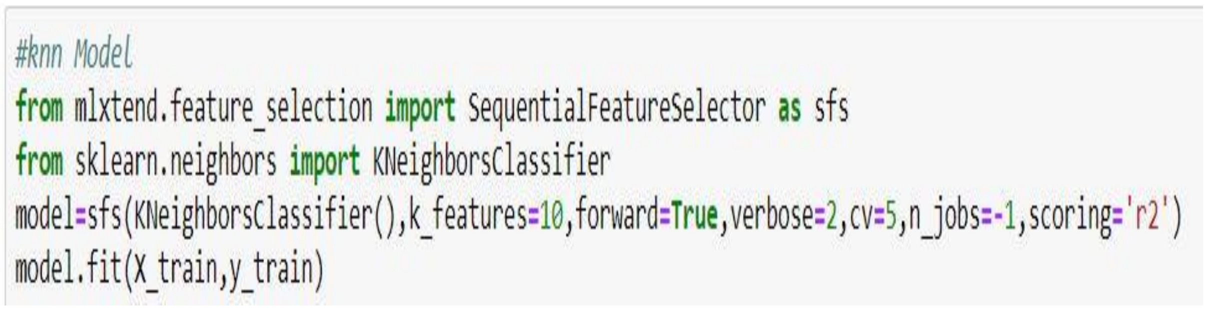
```
#Decision Tree Model
from mlxtend.feature_selection import SequentialFeatureSelector as sfs
from sklearn.tree import DecisionTreeClassifier
model=sfs(DecisionTreeClassifier(),k_features=10,forward=True,verbose=2,cv=5,n_jobs=-1,scoring='r2')
model.fit(X_train,y_train)
```



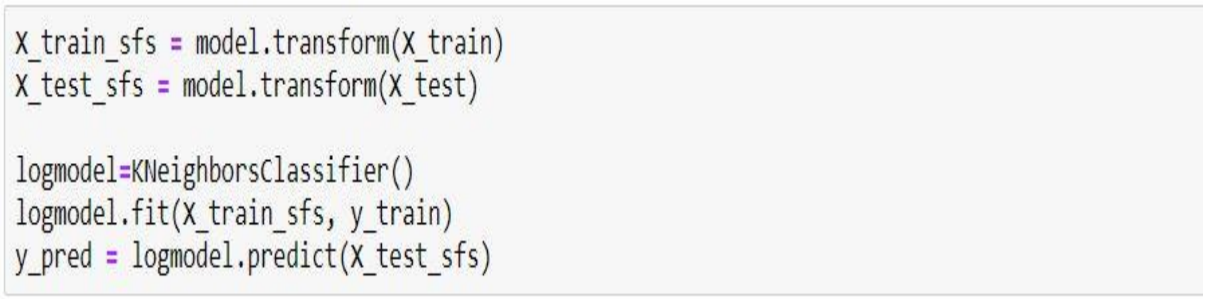
```
X_train_sfs = model.transform(X_train)
X_test_sfs = model.transform(X_test)

logmodel=DecisionTreeClassifier()
logmodel.fit(X_train_sfs, y_train)
y_pred = logmodel.predict(X_test_sfs)
```

3. K-Nearest Neighbor(KNN) Model :

K-Nearest Neighbors (KNN) is one of the simplest algorithms used in Machine Learning for regression and classification problem. KNN algorithms use data and classify new data points based on similarity measures (e.g. distance function). Classification is done by a majority vote to its neighbors. The data is assigned to the class which has the nearest neighbors. As you increase the number of nearest neighbors, the value of k, accuracy might increase.

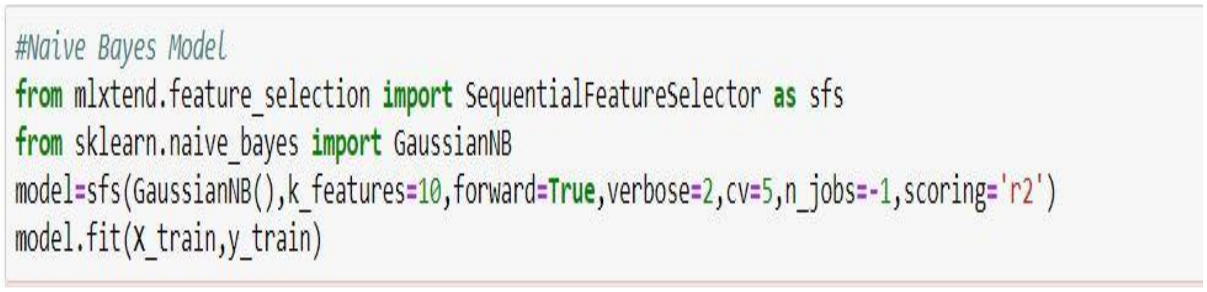


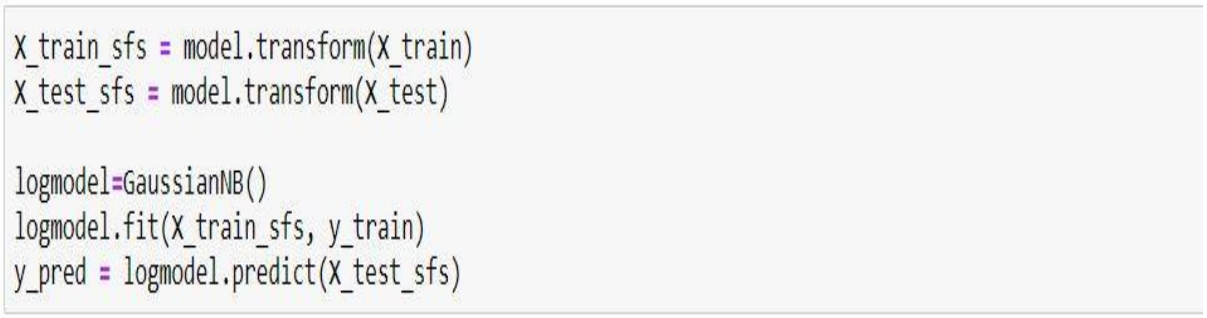
```
#knn Model
from mlxtend.feature_selection import SequentialFeatureSelector as sfs
from sklearn.neighbors import KNeighborsClassifier
model=sfs(KNeighborsClassifier(),k_features=10,forward=True,verbose=2,cv=5,n_jobs=-1,scoring='r2')
model.fit(X_train,y_train)
```


```
X_train_sfs = model.transform(X_train)
X_test_sfs = model.transform(X_test)

logmodel=KNeighborsClassifier()
logmodel.fit(X_train_sfs, y_train)
y_pred = logmodel.predict(X_test_sfs)
```

4. **Naïve Bayes Model:**

A Naive Bayes Classifier is a supervised machine-learning algorithm that uses the Bayes' Theorem, which assumes that features are statistically independent. The theorem relies on the *naive* assumption that input variables are independent of each other, i.e. there is no way to know anything about other variables when given an additional variable. Regardless of this assumption, it has proven itself to be a classifier with good results.


```
#Naive Bayes Model
from mlxtend.feature_selection import SequentialFeatureSelector as sfs
from sklearn.naive_bayes import GaussianNB
model=sfs(GaussianNB(),k_features=10,forward=True,verbose=2,cv=5,n_jobs=-1,scoring='r2')
model.fit(X_train,y_train)
```


```
X_train_sfs = model.transform(X_train)
X_test_sfs = model.transform(X_test)

logmodel=GaussianNB()
logmodel.fit(X_train_sfs, y_train)
y_pred = logmodel.predict(X_test_sfs)
```

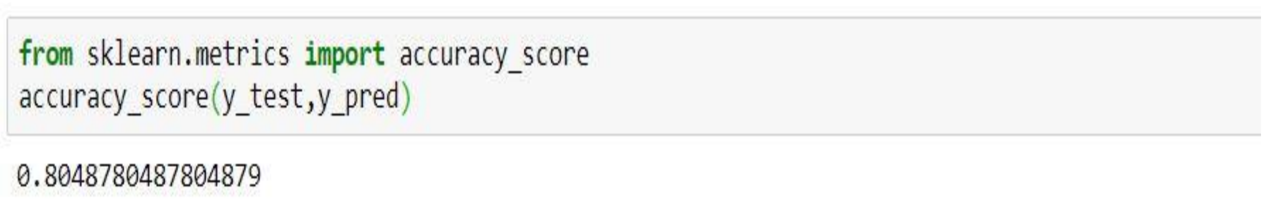
Thus we have successfully created 4 different model namely Logistic Regression Model, Decision Tree Model, KNN Model, Naïve Bayes Model.

The next step left is measuring the performance of all these model.

- **Evaluating Model**

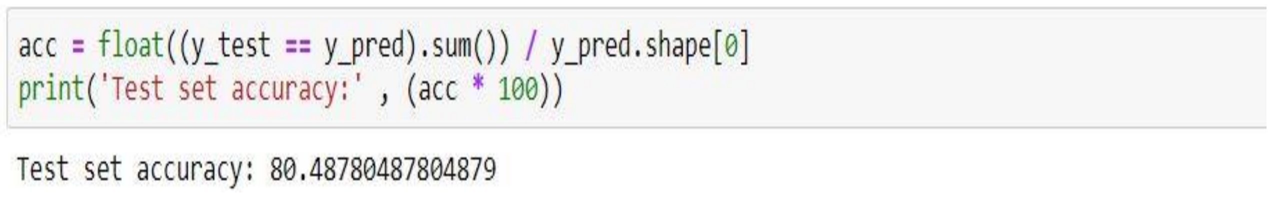
In this section we will all analyze the four model previously made and evaluate their performance. Thus at last we will find which model is more is reliable and accurate for finding the Loan application which should be accepted.

1. **Logistic Regression evaluation:**



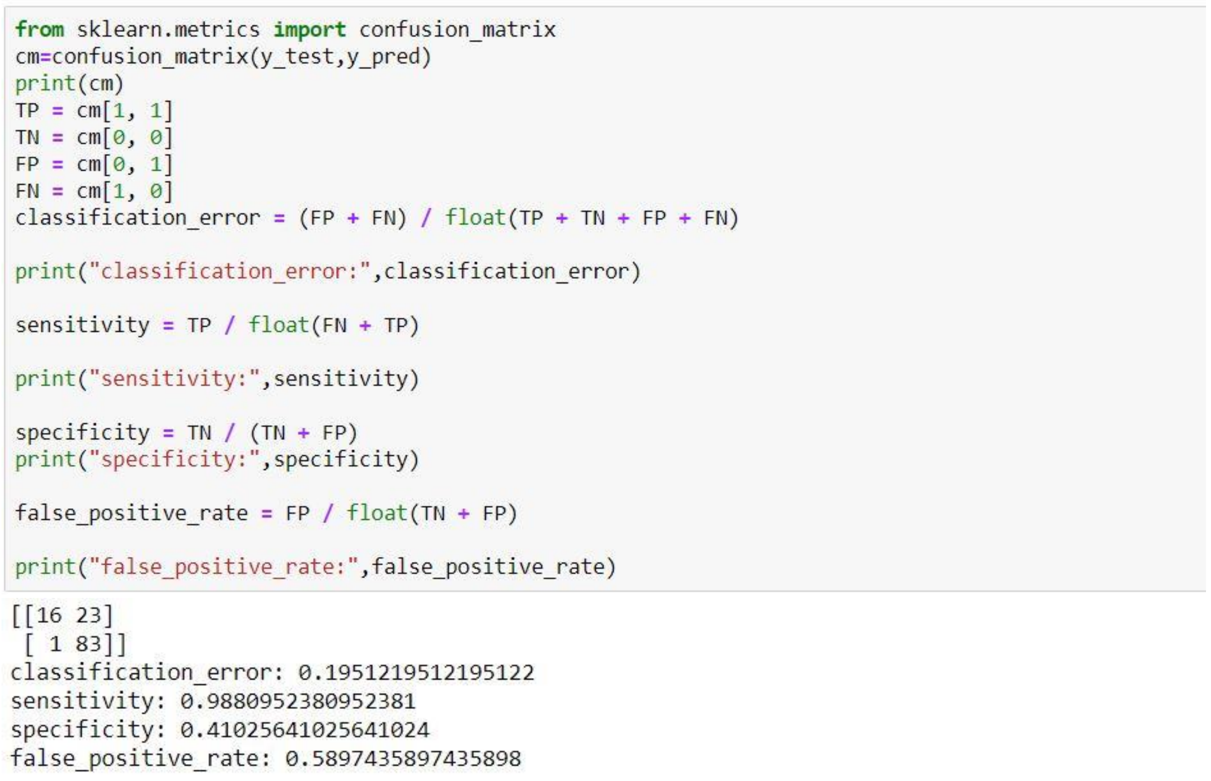
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test,y_pred)
```

```
0.8048780487804879
```



```
acc = float((y_test == y_pred).sum()) / y_pred.shape[0]
print('Test set accuracy:' , (acc * 100))
```

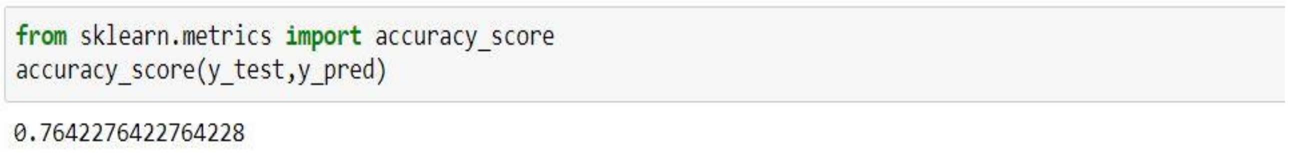
```
Test set accuracy: 80.48780487804879
```



```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
print(cm)
TP = cm[1, 1]
TN = cm[0, 0]
FP = cm[0, 1]
FN = cm[1, 0]
classification_error = (FP + FN) / float(TP + TN + FP + FN)

print("classification_error:",classification_error)

sensitivity = TP / float(FN + TP)

print("sensitivity:",sensitivity)

specificity = TN / (TN + FP)
print("specificity:",specificity)

false_positive_rate = FP / float(TN + FP)

print("false_positive_rate:",false_positive_rate)
```

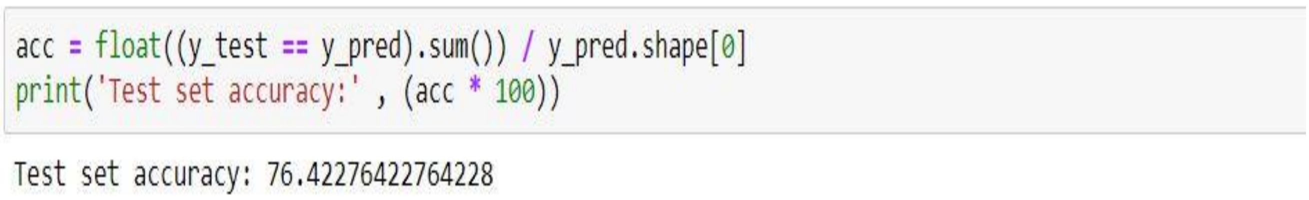
```
[[16 23]
 [ 1 83]]
classification_error: 0.1951219512195122
sensitivity: 0.9880952380952381
specificity: 0.41025641025641024
false_positive_rate: 0.5897435897435898
```

2. Decision Tree evaluation



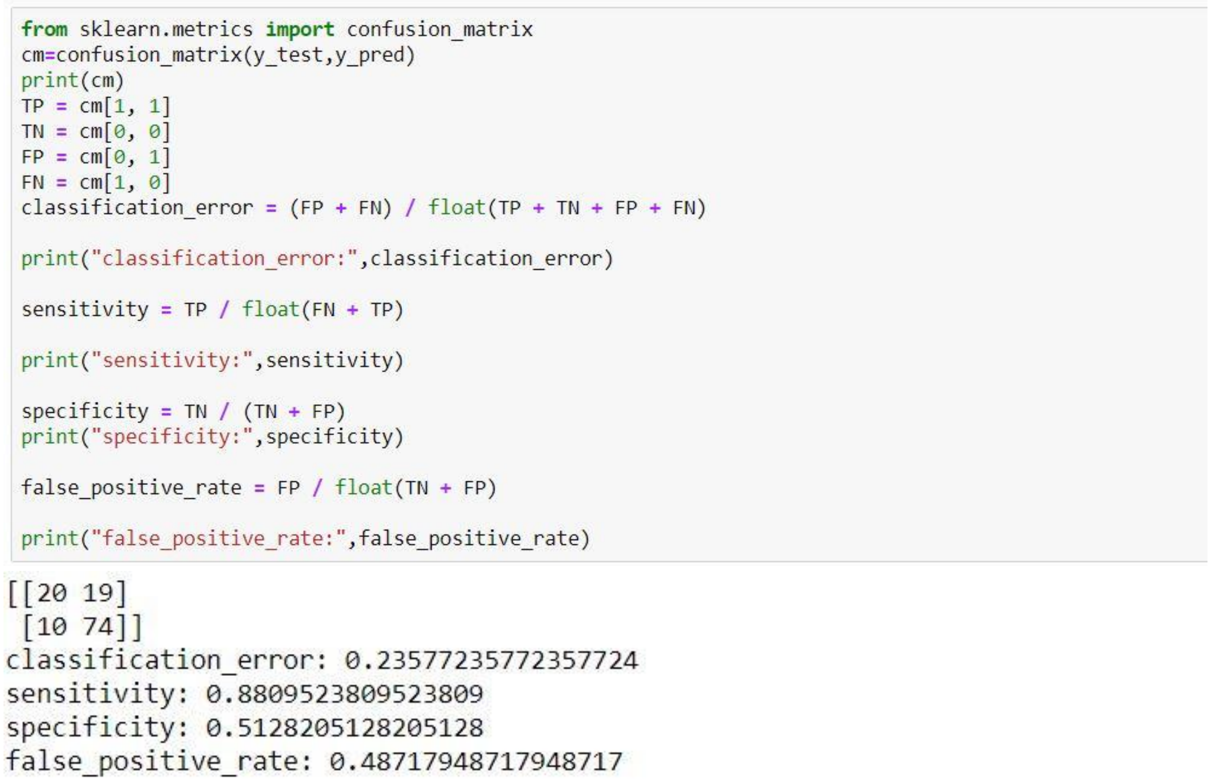
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test,y_pred)
```

```
0.7642276422764228
```



```
acc = float((y_test == y_pred).sum()) / y_pred.shape[0]
print('Test set accuracy:' , (acc * 100))
```

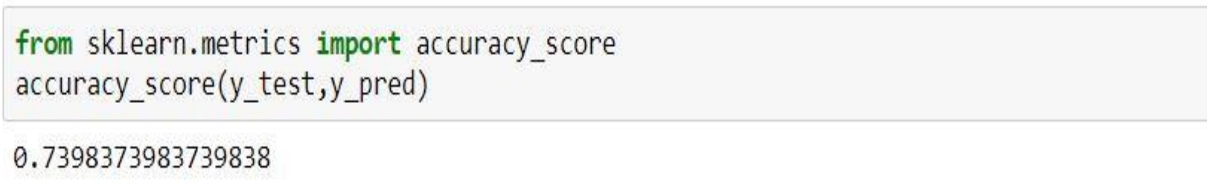
```
Test set accuracy: 76.42276422764228
```



```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
print(cm)
TP = cm[1, 1]
TN = cm[0, 0]
FP = cm[0, 1]
FN = cm[1, 0]
classification_error = (FP + FN) / float(TP + TN + FP + FN)

print("classification_error:",classification_error)

sensitivity = TP / float(FN + TP)

print("sensitivity:",sensitivity)

specificity = TN / (TN + FP)
print("specificity:",specificity)

false_positive_rate = FP / float(TN + FP)

print("false_positive_rate:",false_positive_rate)
```

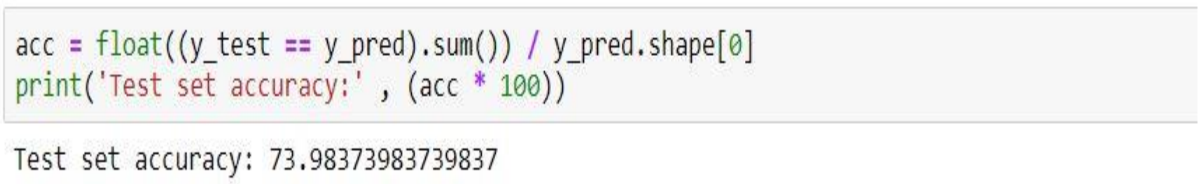
```
[[20 19]
 [10 74]]
classification_error: 0.23577235772357724
sensitivity: 0.8809523809523809
specificity: 0.5128205128205128
false_positive_rate: 0.48717948717948717
```

3. KNN Model evaluation:



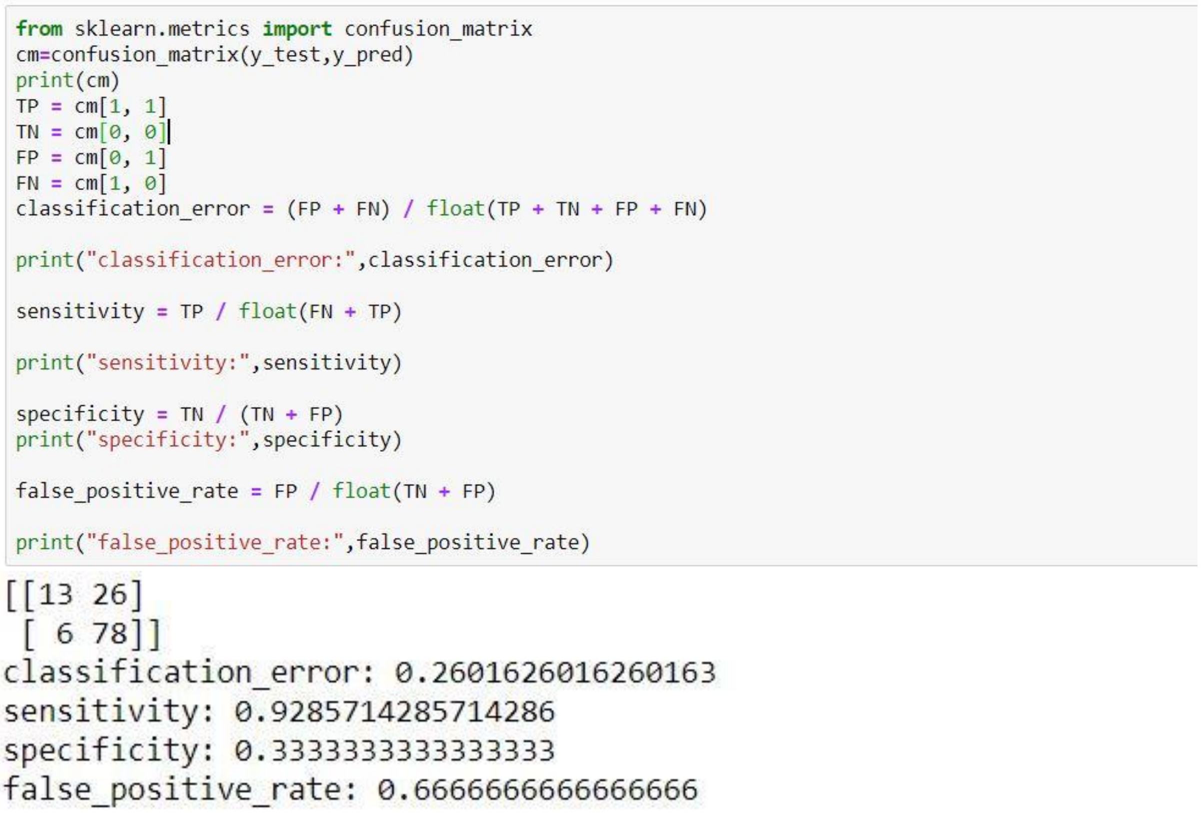
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test,y_pred)
```

```
0.7398373983739838
```



```
acc = float((y_test == y_pred).sum()) / y_pred.shape[0]
print('Test set accuracy:' , (acc * 100))
```

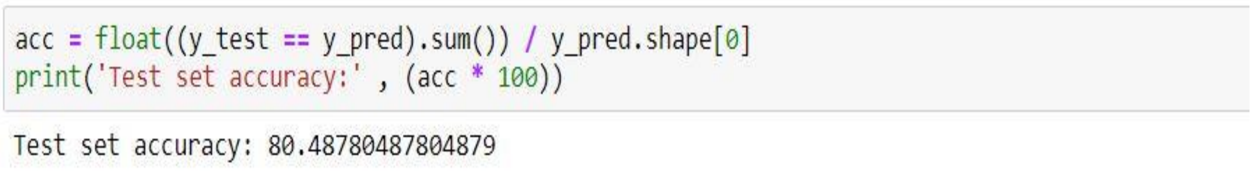
```
Test set accuracy: 73.98373983739837
```



```
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
print(cm)
TP = cm[1, 1]
TN = cm[0, 0]
FP = cm[0, 1]
FN = cm[1, 0]
classification_error = (FP + FN) / float(TP + TN + FP + FN)

print("classification_error:",classification_error)

sensitivity = TP / float(FN + TP)

print("sensitivity:",sensitivity)

specificity = TN / (TN + FP)
print("specificity:",specificity)

false_positive_rate = FP / float(TN + FP)

print("false_positive_rate:",false_positive_rate)
```

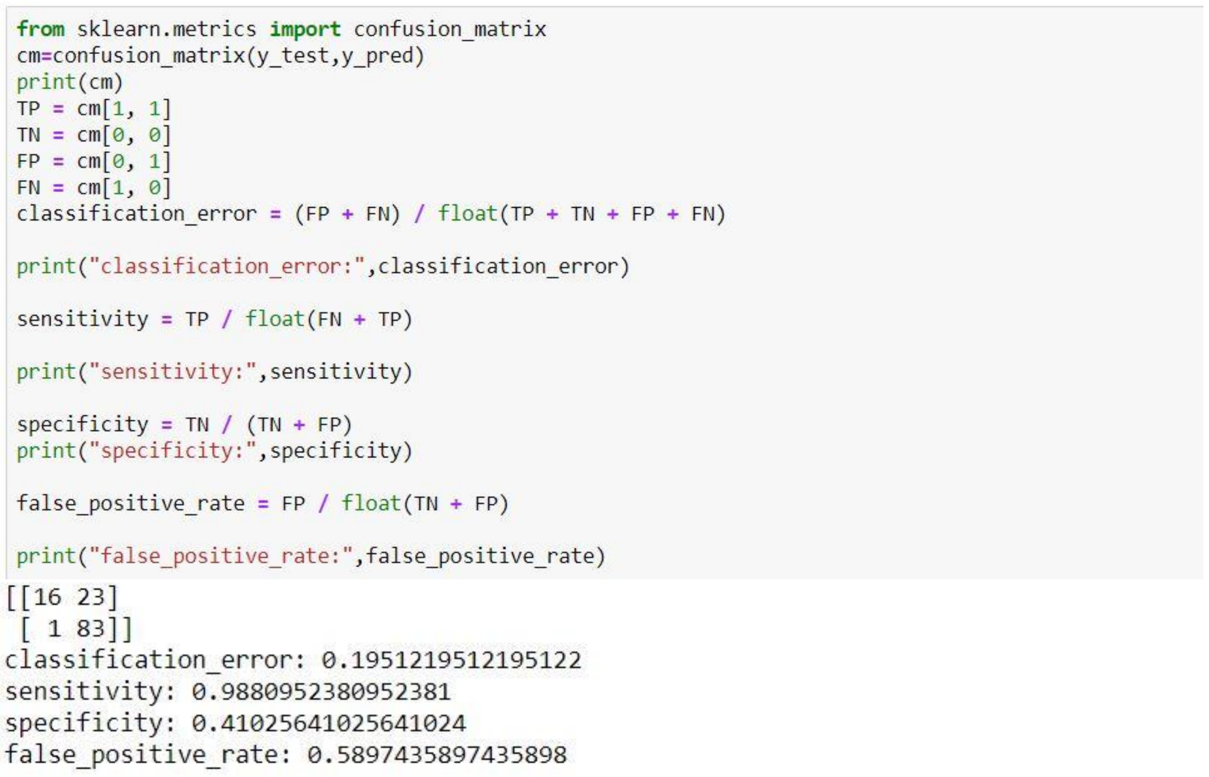
```
[[13 26]
 [ 6 78]]
classification_error: 0.2601626016260163
sensitivity: 0.9285714285714286
specificity: 0.3333333333333333
false_positive_rate: 0.6666666666666666
```

4. Naïve Bayes Model evaluation:



```
acc = float((y_test == y_pred).sum()) / y_pred.shape[0]
print('Test set accuracy:' , (acc * 100))
```

```
Test set accuracy: 80.48780487804879
```



```python
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
print(cm)
TP = cm[1, 1]
TN = cm[0, 0]
FP = cm[0, 1]
FN = cm[1, 0]
classification_error = (FP + FN) / float(TP + TN + FP + FN)

print("classification_error:",classification_error)

sensitivity = TP / float(FN + TP)

print("sensitivity:",sensitivity)

specificity = TN / (TN + FP)
print("specificity:",specificity)

false_positive_rate = FP / float(TN + FP)

print("false_positive_rate:",false_positive_rate)
```

```
[[16 23]
 [ 1 83]]
classification_error: 0.1951219512195122
sensitivity: 0.9880952380952381
specificity: 0.41025641025641024
false_positive_rate: 0.5897435897435898
```

Conclusion

So, finally, after cleaning the data and selecting the important features required for predicting the status for loan approval we build various models namely Logistic regression, Decision tree, K-tree for prediction. We evaluated all the model in regard of parameters like accuracy, sensitivity, and specificity.

Thus among all the models, we find Logistic Regression Model to be much better and reliable. It gives an accuracy of 80% which is highest among all the four models.

Finally, we can use the proposed model for predicting the Loan application status.