Exploratory Data Analysis

by Aryan Jain

Introduction

The critical process of conducting preliminary investigations on data to discover patterns, identify anomalies, test hypotheses, and validate assumptions using summary statistics and graphical representations is known as exploratory data analysis. We are using an example of a Loan Eligibility Prediction dataset to demonstrate our understanding of the Exploratory Data Analysis concept and techniques, and we are attempting to glean as many insights as possible from it.

First, we imported the necessary required libraries.

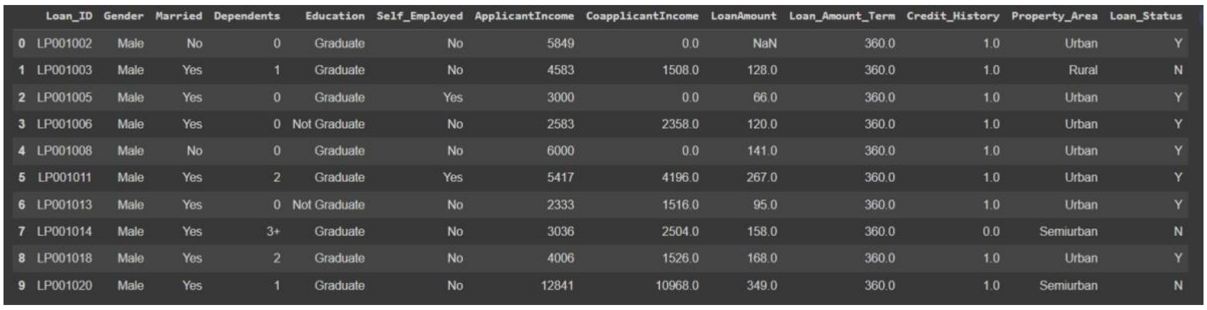
```
# The dependent libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
import pandas as pd
from sklearn.naive_bayes import MultinomialNB, ComplementNB
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno
import nltk
```

Load the dataset containing the Loan Eligibility Prediction data. The data is made up of various samples with the necessary parameters for validating loan eligibility.

```
# Mount your G-Drive

from google.colab import drive
drive.mount('/content/drive')

# read the dataset
loan_data =pd.read_csv("/content/drive/MyDrive/data/loan.csv")
loan_data.info()
```

As we can see, the 'mount()' function is being used to mount data from the Google Drive where the dataset is stored/located. Following that, the data is read from the dataset file on the drive using the'read csv()' function. In addition, to ensure that the data obtained is correct, we print a few lines of the dataset using the 'head()' function.

```
loan_data.head()
```



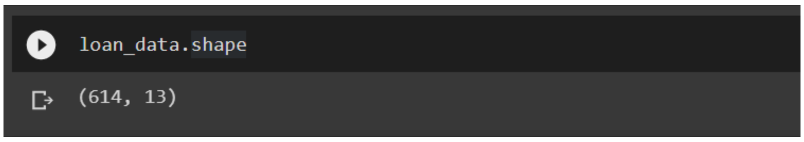
	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Area	Loan_Status
0	LP001002	Male	No	0	Graduate	No	5849	0.0	NaN	360.0	1.0	Urban	Y
1	LP001003	Male	Yes	1	Graduate	No	4583	1508.0	128.0	360.0	1.0	Rural	N
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	360.0	1.0	Urban	Y
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	360.0	1.0	Urban	Y
4	LP001008	Male	No	0	Graduate	No	6000	0.0	141.0	360.0	1.0	Urban	Y
5	LP001011	Male	Yes	2	Graduate	Yes	5417	4196.0	267.0	360.0	1.0	Urban	Y
6	LP001013	Male	Yes	0	Not Graduate	No	2333	1516.0	95.0	360.0	1.0	Urban	Y
7	LP001014	Male	Yes	3+	Graduate	No	3036	2504.0	158.0	360.0	0.0	Semiurban	N
8	LP001018	Male	Yes	2	Graduate	No	4006	1526.0	168.0	360.0	1.0	Urban	Y
9	LP001020	Male	Yes	1	Graduate	No	12841	10968.0	349.0	360.0	1.0	Semiurban	N

Observing the Data

To get a closer look at the data, I used the pandas library's ".head()" function, which returns observations from the data set. The dataset includes sample variables such as education, income, credit history, and property area, all of which indicate his/her loan eligibility.

Using ".shape", we discovered the total number of rows and columns in the dataset. One is a dependent variable, while the others are all 12 independent variables.

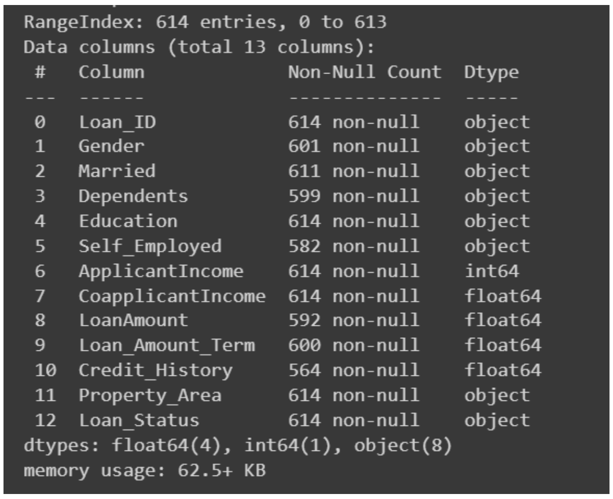
```
loan_data.shape
```



```
loan_data.shape
(614, 13)
```

There are 614 observations and 12 characteristics in the dataset. It is also a good idea to be familiar with the columns and their corresponding data types, as well as whether or not they contain null values.

```
loan_data.info()
```



```
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            614 non-null    object 
 1   Gender             601 non-null    object 
 2   Married            611 non-null    object 
 3   Dependents         599 non-null    object 
 4   Education          614 non-null    object 
 5   Self_Employed      582 non-null    object 
 6   ApplicantIncome    614 non-null    int64  
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object 
 12  Loan_Status        614 non-null    object 
dtypes: float64(4), int64(1), object(8)
memory usage: 62.5+ KB
```

- The values in the Data are integer, float, and object.
- There are no null/missing values in any variable column.

Using the '.info()' function, we can now obtain a brief summary of the dataset under consideration. This function assists us in printing information about a specific data frame, such as index dtype and columns, non-null values, and memory usage.

This accepts various parameters but does not return any value; instead, it simply prints the summary. Here, we see a display of the various columns in the Loan Eligibility Prediction dataset that we used, such as gender, married status, dependents, and so on, as well as the corresponding non-null values, count, and Dtype.

Balanced or Unbalanced Dataset

This is not a balanced dataset as it has an uneven distribution of observations, i.e one class label has a comparitively low number of observations. Here, the Loan Approvals and Loan rejections are not equal.

Identify ranges for numerical data

The pandas describe() function is extremely useful for obtaining various summary statistics.

The count, mean, standard deviation, minimum and maximum values, and quantiles of the data are returned by this function.

It appears that some data is missing. This can be investigated using the 'describe()' function. This function can be very useful in obtaining statistics such as mean, median, mode, standard deviation, minimum and maximum values, and quartiles by returning the values.

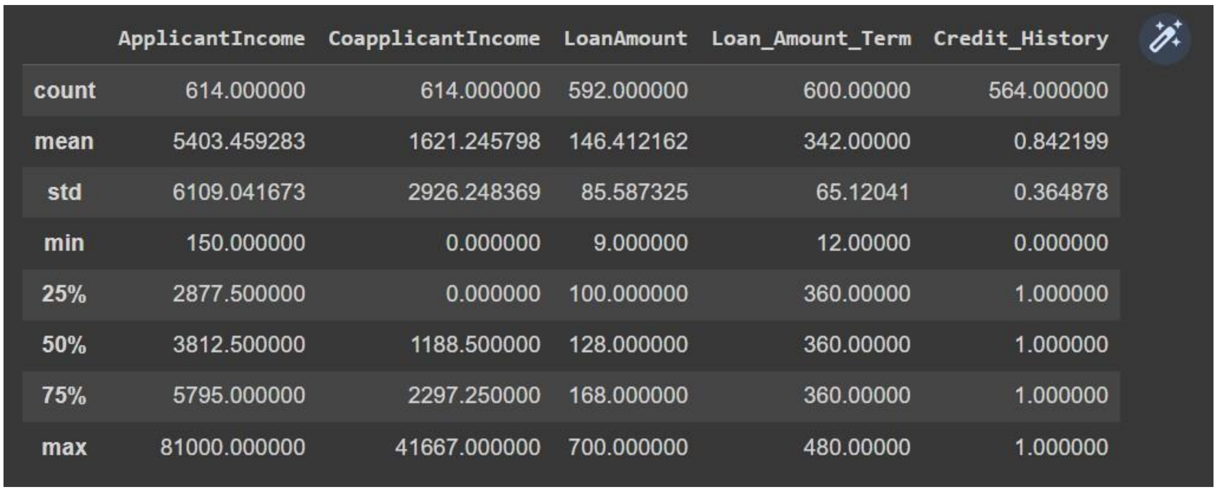
As we can see,

- The mean value of a few attributes in each column is greater, while a few attributes are less than the median represented in the index column by 50%.
- There is a significant difference between the 75 percent and maximum values in the index column for a few attributes such as 'ApplicantIncome,' 'CoapplicantIncome,' and 'LoanAmount.'

The two examples above provide evidence that there are extreme values-Outliners in the dataset.

We can also see that the mean in the Credit History column indicates that nearly 84 percent of people have a credit history. We can now use 'seaborn' to visualize the statistics of the remaining numerical features such as ApplicantIncome, CoapplicantIncome, LoanAmount, Loan Amount Term.

```
loan_data.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

Analyze the Dataset

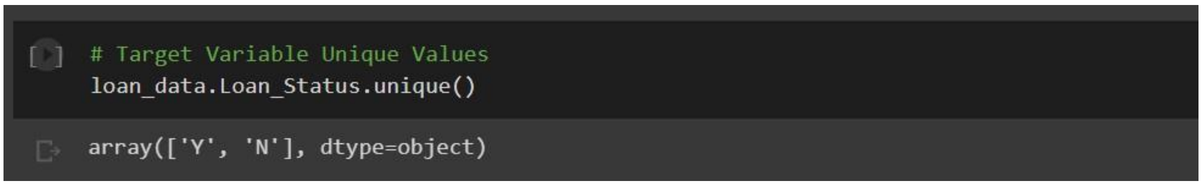
Identify the Target Variable

We'll start with the target variable, Loan Status. Let us examine its frequency table, percentage distribution, and bar plot because it is a categorical variable.

The count of each category in a variable can be found in its frequency table.

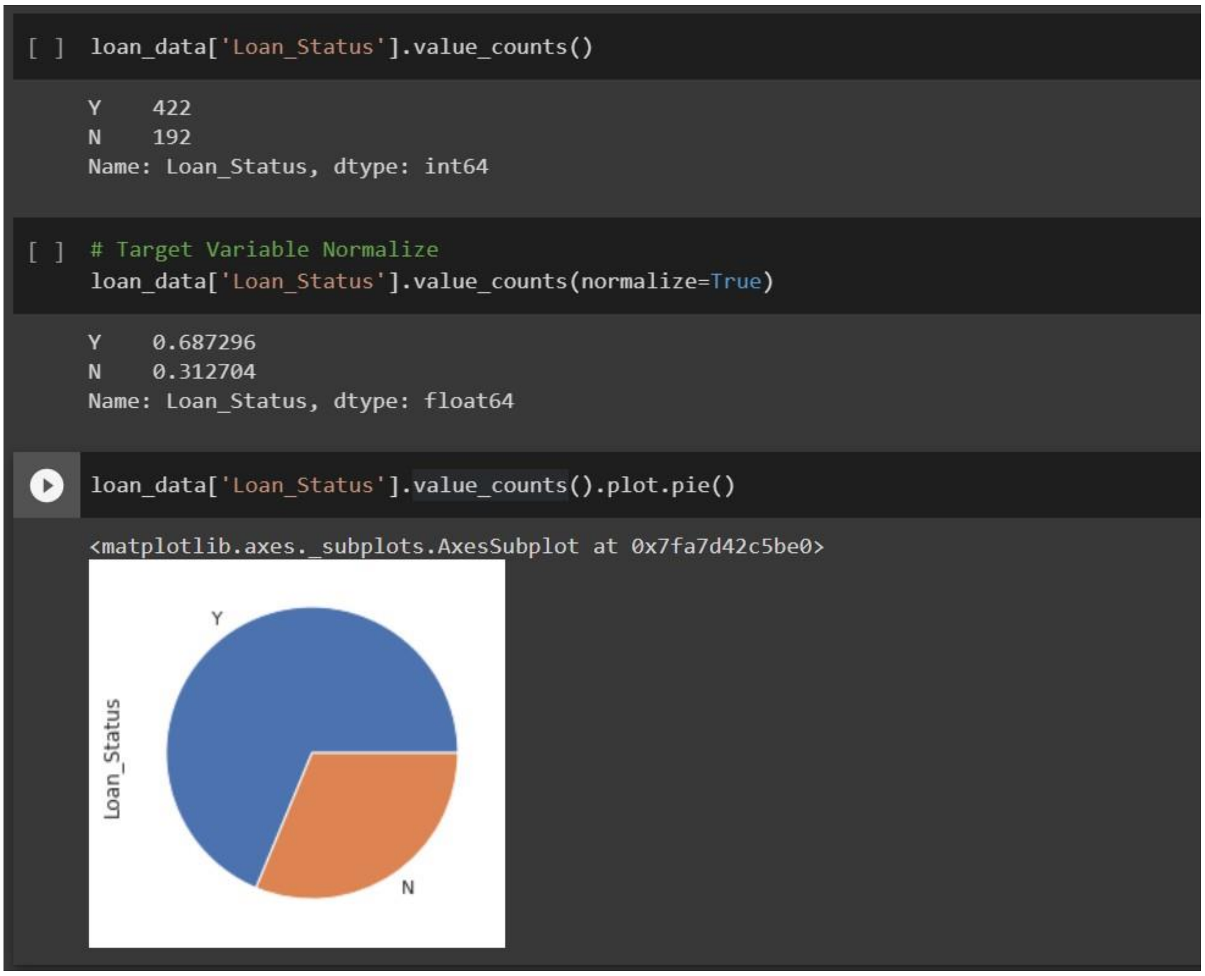
- First, we use the 'unique()' function to get the object displayed from the Loan Status.
- Second, we display the count of yes and no occurances using the 'value counts()' function.
- Third, we sort based on frequencies using the 'value counts(bool,default true)' function.
- Finally, the pie chart is drawn with the 'plot.pie()' function.

```
# Target Variable Unique Values
loan_data.Loan_Status.unique()
loan_data['Loan_Status'].value_counts()

# Target Variable Normalize
loan_data['Loan_Status'].value_counts(normalize=True)

loan_data['Loan_Status'].value_counts().plot.pie()
```



```
# Target Variable Unique Values
loan_data.Loan_Status.unique()
```

```
array(['Y', 'N'], dtype=object)
```



```
[ ]  loan_data['Loan_Status'].value_counts()

     Y    422
     N    192
     Name: Loan_Status, dtype: int64

[ ]  # Target Variable Normalize
     loan_data['Loan_Status'].value_counts(normalize=True)

     Y    0.687296
     N    0.312704
     Name: Loan_Status, dtype: float64

     loan_data['Loan_Status'].value_counts().plot.pie()

     <matplotlib.axes._subplots.AxesSubplot at 0x7fa7d42c5be0>
```

Out of 614 applicants, 422 (roughly 69 percent) were approved.

When each variable is visualized individually, the various types of variables obtained are categorical, ordinal, and numerical.

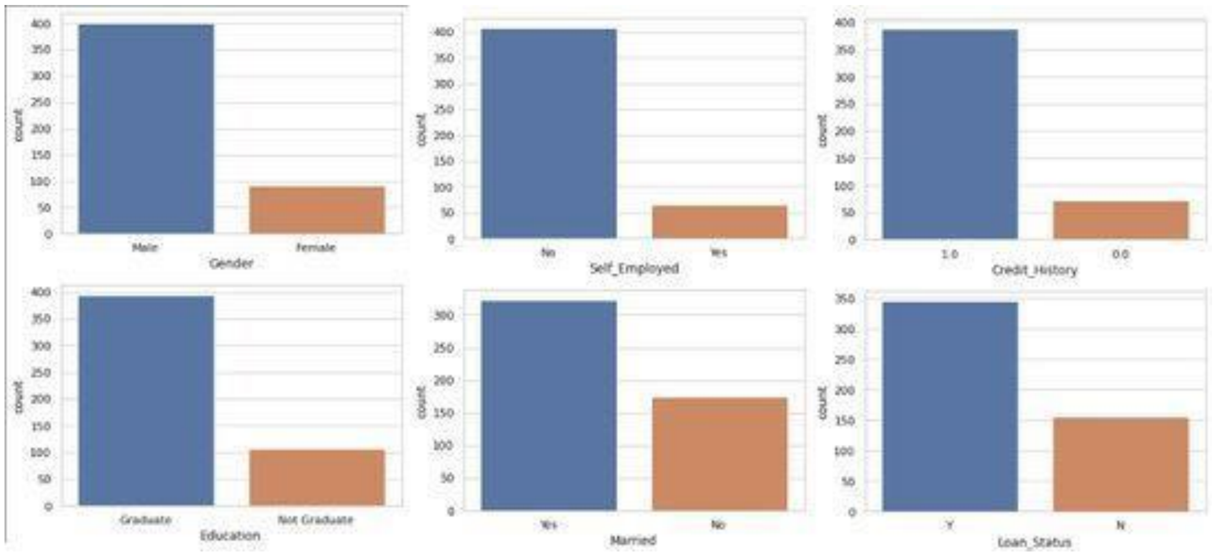
1. **Categorical features:** These include attributes such as Gender (male/female), Married (yes/no), Self Employed (yes/no), and Credit History (numeric values), Loan Status(Y/N)
2. **Ordinal features:** These features include variables with a different order than categorical features, such as Dependents (number of dependents), Education (graduate/not graduate), and so on. Property Area(urban/rural)
3. **Numerical features:** ApplicantIncome, CoapplicantIncome, LoanAmount, Loan Amount Term are examples of numerical values found in these features (numeric values)

Data Analysis using Plots

Univariate Analysis Observations

Categorical Field analysis

```
# categorical - Independent variable

sns.set_theme(style="whitegrid")

count = sns.countplot(data= loan_data, x= 'Gender', order= loan_data['Gender'].value_counts().index)
plt.show()
count = sns.countplot(data= loan_data, x= 'Education', order= loan_data['Education'].value_counts().index)
plt.show()
count = sns.countplot(data= loan_data, x= 'Self_Employed', order= loan_data['Self_Employed'].value_counts().index)
plt.show()
count = sns.countplot(data= loan_data, x= 'Married', order= loan_data['Married'].value_counts().index)
plt.show()
count = sns.countplot(data= loan_data, x= 'Credit_History', order= loan_data['Credit_History'].value_counts().index)
plt.show()
count = sns.countplot(data= loan_data, x= 'Loan_Status', order= loan_data['Loan_Status'].value_counts().index)
plt.show()
```



To begin, we can see in the first bar graph (count vs Gender) that the number of male applicants is greater than the number of female applicants.

Second, there is a significant increase in the 'no' count in comparison to the 'yes' count in the bar graph (count vs Self Employed), implying that there are few self-employed members.

Third, in the bar graph below (count vs Credit History), the count for '1.0' is greater than the count for '0.0,' indicating that there are many dependents with credit history.

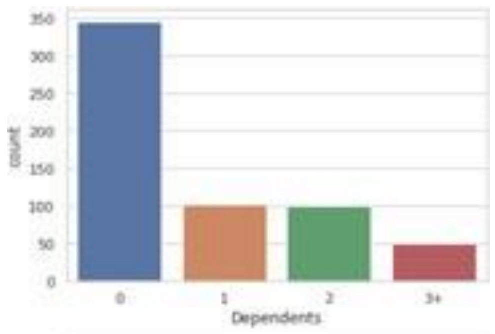
Fourth, in the following bar graph (count vs Education), the number of graduate students is greater than the number of non-graduate students.

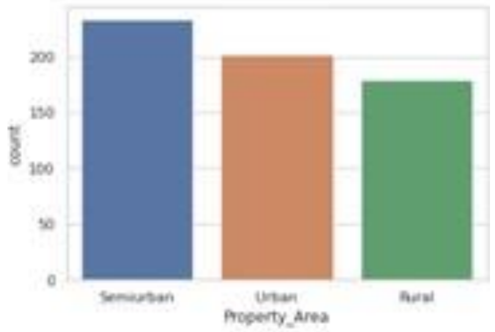
Fifth, in the next bar graph (count vs Married), the count for 'yes' is greater than the count for 'no,' indicating that more married applicants take loans than unmarried applicants.

Finally, in the last bar graph (count vs Loan Status), the count for ‘yes’ is higher compared to the count of ‘no’, indicating that more loans are being approved rather than being rejected.

Ordinal field analysis

```
# Ordinal Field analysis

sns.set_theme(style="whitegrid")

count = sns.countplot(data= loan_data, x= 'Dependents', order= loan_dat
a['Dependents'].value_counts().index)
plt.show()
count = sns.countplot(data= loan_data, x= 'Property_Area', order= loan_
data['Property_Area'].value_counts().index)
plt.show()
```





Ordinal Characteristics:

- Number of Dependents
- Property or Area Background

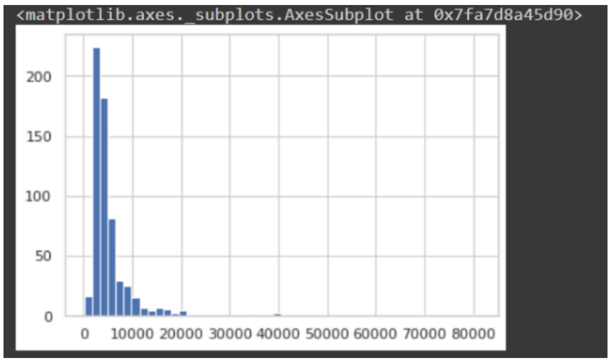
Our Visual Analysis below, indicates that:

- Nearly 350 of the 614 applicants have no dependents.
- Semi-urban areas have the most applicants, followed by urban areas.

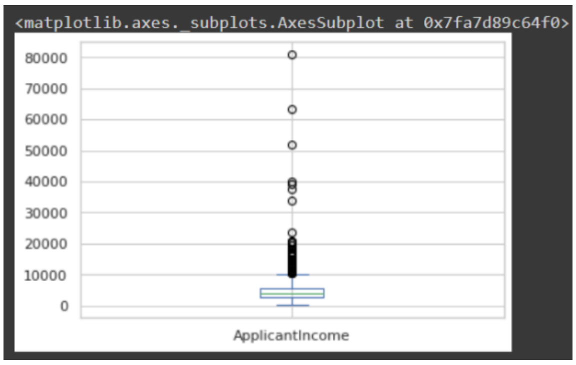
Numerical Analysis

- The Applicant's Income
- The Co-Applicant's Income

```
# Numerical Analysis
loan_data['ApplicantIncome'].hist(bins=50)

# Numerical Analysis
loan_data['ApplicantIncome'].plot.box()
```



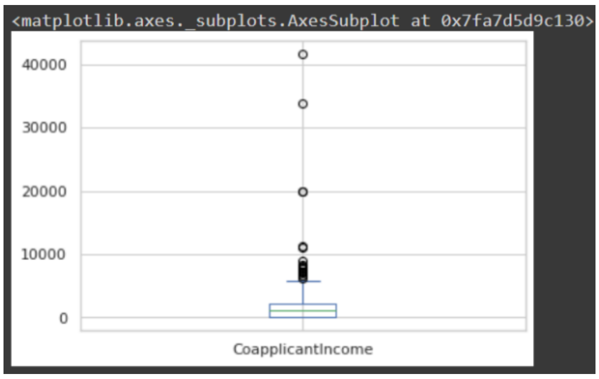
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa7d8a45d90>
```



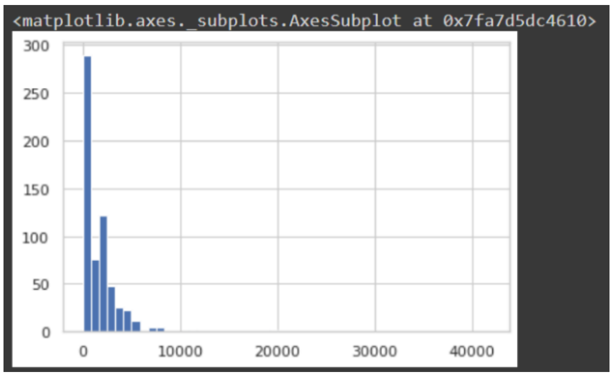
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa7d89c64f0>
```

It can be deduced that the majority of the data in Applicant income is to the left, indicating that it is not normally distributed. The presence of outliers is confirmed by the boxplot. This can be attributed to societal income disparities.

```
loan_data['CoapplicantIncome'].hist(bins=50)
loan_data['CoapplicantIncome'].plot.box()
```



```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa7d5d9c130>
```

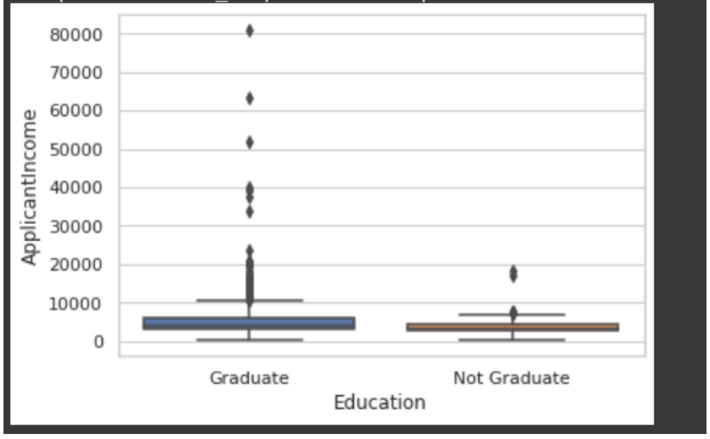


```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa7d5dc4610>
```

CoapplicantIncome is lesser than applicantIncome and is within the 5000–15000, again with some outliers.

```
sns.boxplot(x="Education", y="ApplicantIncome", data=loan_data)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa7d5ce13a0>
```



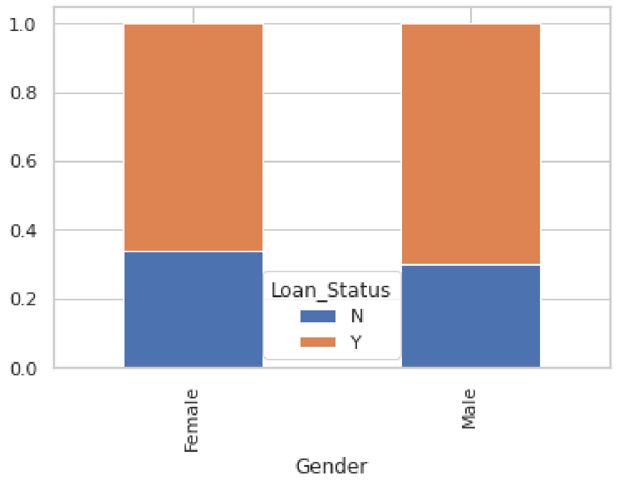
We can see that there are more graduates with very high incomes, who appear to be the outliers.

Bivariate Analysis

Categorical Independent Variable vs Target Variable

Gender vs Loan_Status

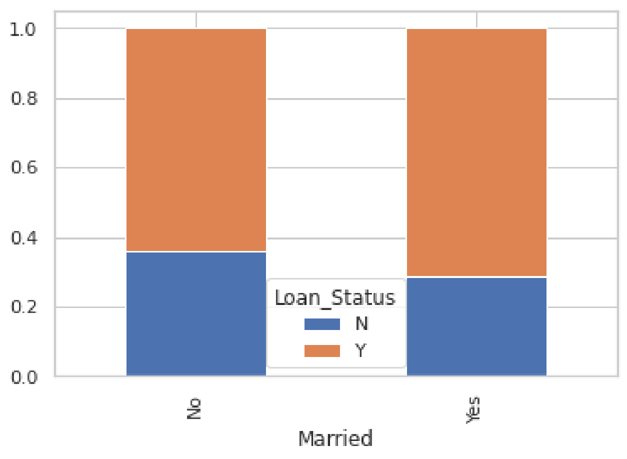
```
GenderTarget=pd.crosstab(loan_data['Gender'],loan_data['Loan_Status'])
GenderTarget.div(GenderTarget.sum(1).astype(float), axis=0).plot(kind="
bar", stacked=True)
plt.show()
```



There is not a significant difference in loan approval rates between men and women in the graph above.

Marriage Status Vs Loan_Status

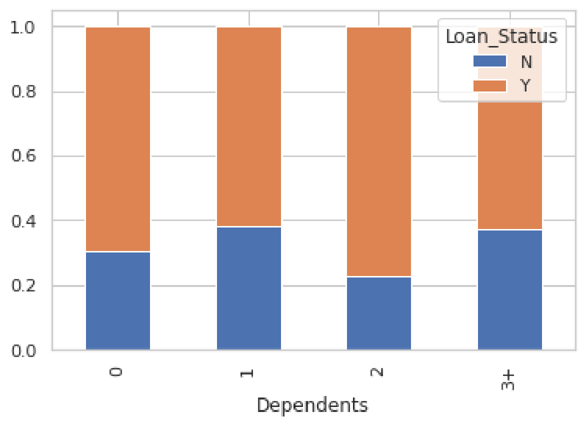
```
Married=pd.crosstab(loan_data['Married'],loan_data['Loan_Status'])
Married.div(Married.sum(1).astype(float), axis=0).plot(kind="bar", stac
ked=True)
plt.show()
```



As shown, married applicants have a higher likelihood of loan approval than unmarried applicants.

Dependency Vs Loan_Status

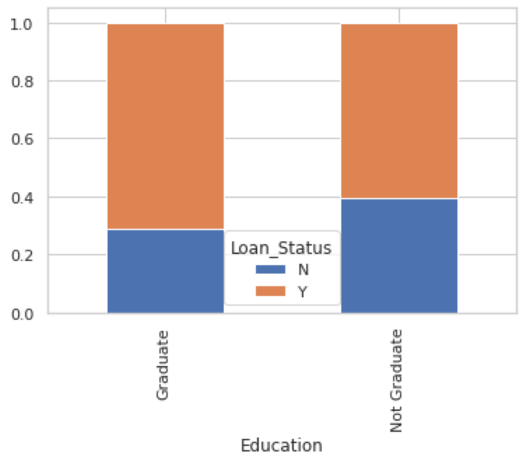
```
Dependents=pd.crosstab(loan_data['Dependents'],loan_data['Loan_Status'])
Dependents.div(Dependents.sum(1).astype(float), axis=0).plot(kind="bar", st
acked=True)
plt.show()
```



In this case, applicants with zero or two dependents have a better chance of getting their loan approved, but no proper relationship can be established.

Education Vs Loan_Status

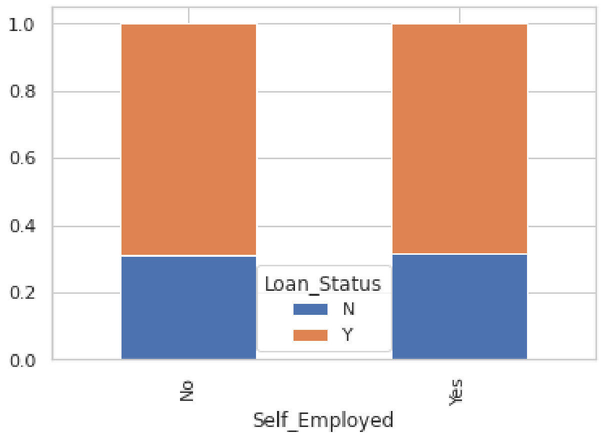
```
Education=pd.crosstab(loan_data['Education'],loan_data['Loan_Status'])
Education.div(Education.sum(1).astype(float), axis=0).plot(kind="bar",
stacked=True)
plt.show()
```



In the graph above, we can see that all graduate students have a higher chance of loan approval than non-graduate students.

Employment Type Vs Loan Status

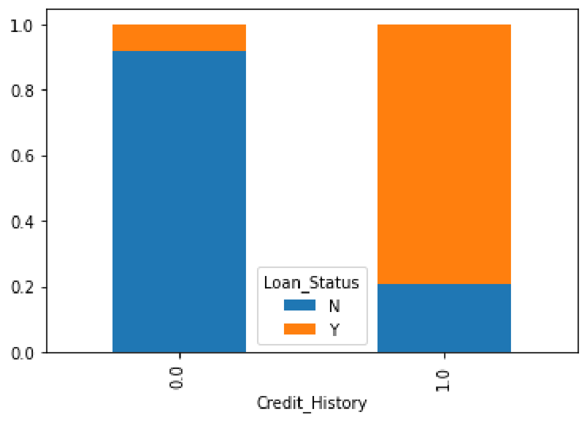
```
Self_Employed=pd.crosstab(loan_data['Self_Employed'],loan_data['Loan_Status'])
Self_Employed.div(Self_Employed.sum(1).astype(float),axis=0).plot(kind="bar",stacke
d=True)
plt.show()
```



The graph clearly shows that Self Employed employees have a lower chance of getting their loan approved, but their circumstances are not dire.

Credit_History Vs Loan_Status

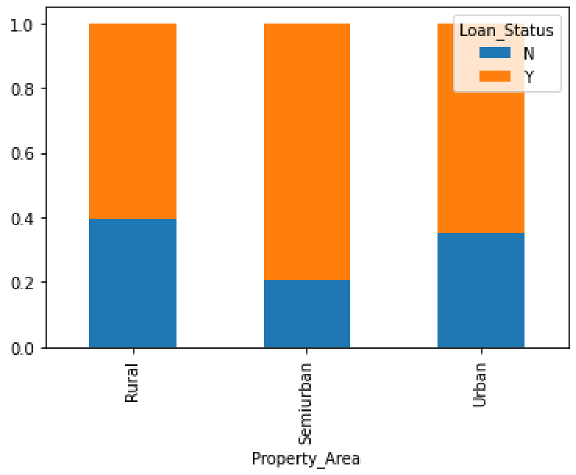
```
Credit_History=pd.crosstab(loan_data['Credit_History'],loan_data['Loan_Status'])
Credit_History.div(Credit_History.sum(1).astype(float),axis=0).plot(kind="bar",stac
ked=True)
plt.show()
```



People with a credit history of 1 are more likely to have their loans approved than those with a credit history of 0.

Property Area Vs Loan Status

```
Property_Area=pd.crosstab(loan_data['Property_Area'],loan_data['Loan_Status'])
Property_Area.div(Property_Area.sum(1).astype(float),axis=0).plot(kind="bar",stacke
d=True)
plt.show()
```

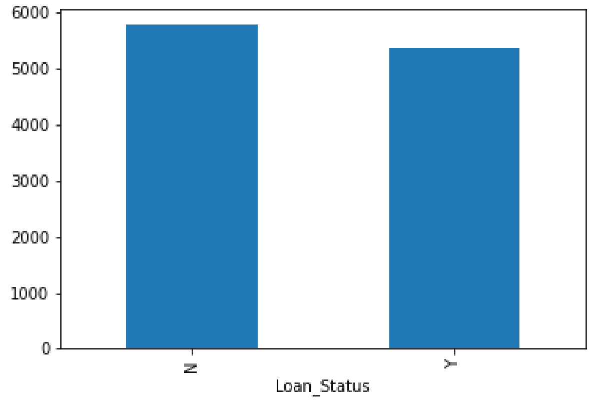


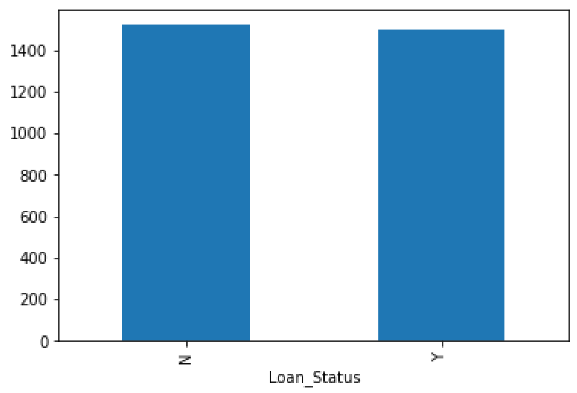
The proportion of loans approved in semiurban areas is higher than in rural or urban areas.

Numerical Independent Variable vs Target Variable

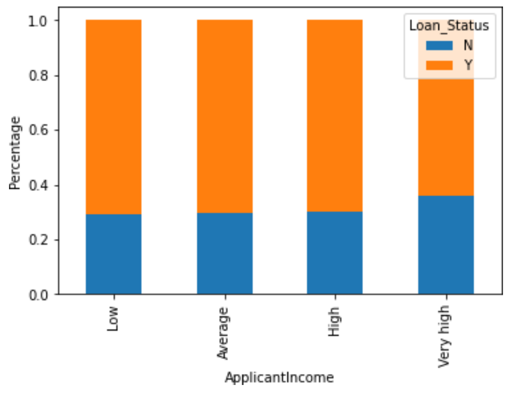
```
loan_data.groupby('Loan_Status')['ApplicantIncome'].mean().plot.bar()

loan_data.groupby('Loan_Status')['CoapplicantIncome'].mean().plot.bar()
```

We attempted to compare the mean income of people whose loans were approved versus those whose loans were denied, but there were no differences in mean income. So, based on the values in it, we create bins for the applicant income variable and analyze the corresponding loan status for each bin.





```
bins=[0,2500,4000,6000,81000]
group=['Low','Average','High', 'Very high']
loan_data['Income_bin']=pd.cut(loan_data['ApplicantIncome'],bins,labels
=group)

Income_bin=pd.crosstab(loan_data['Income_bin'],loan_data['Loan_Status']
)
Income_bin.div(Income_bin.sum(1).astype(float), axis=0).plot(kind="bar"
, stacked=True)
plt.xlabel('ApplicantIncome')
P = plt.ylabel('Percentage')
```

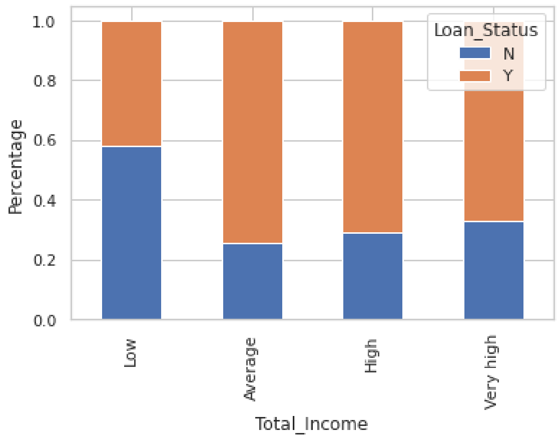


It can be inferred that applicant income has no effect on loan approval, which contradicts our hypothesis that if the applicant income is high, the chances of loan approval are also high.

```
bins=[0,1000,3000,42000]
group=['Low','Average','High']
loan_data['Coapplicant_Income_bin']=pd.cut(loan_data['CoapplicantIncome
'],bins,labels=group)

Coapplicant_Income_bin=pd.crosstab(loan_data['Coapplicant_Income_bin'],
loan_data['Loan_Status'])
Coapplicant_Income_bin.div(Coapplicant_Income_bin.sum(1).astype(float),
 axis=0).plot(kind="bar", stacked=True)
plt.xlabel('CoapplicantIncome')
P = plt.ylabel('Percentage')
```

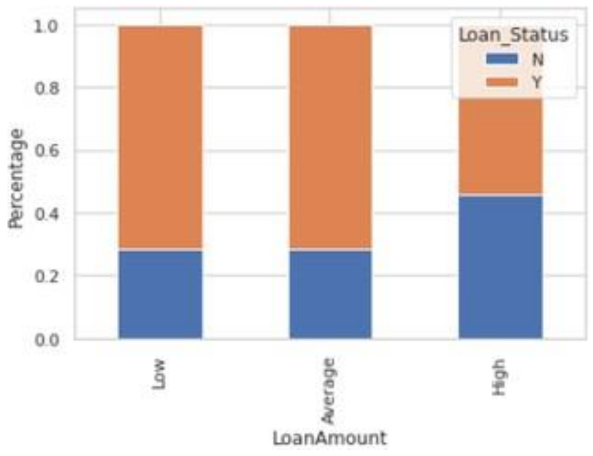
It demonstrates that if the income of the coapplicant is low, the chances of loan approval are high. However, this does not appear to be the case. The possible reason for this is that most applicants do not have a coapplicant, so the coapplicant income for such applicants is zero, and thus loan approval is not contingent on it. So we can create a new variable that includes the applicant's and coapplicant's income in order to visualize the combined effect of income on loan approval.

```
loan_data['Total_Income']=loan_data['ApplicantIncome']
+loan_data['CoapplicantIncome']

bins=[0,2500,4000,6000,81000]
group=['Low','Average','High', 'Very high']
loan_data['Total_Income_bin']=pd.cut(loan_data['Total_Income'],bins,labels=
group)

Total_Income_bin=pd.crosstab(loan_data['Total_Income_bin'],loan_data['Loan_
Status'])
Total_Income_bin.div(Total_Income_bin.sum(1).astype(float), axis=0).plot(ki
nd="bar", stacked=True)
plt.xlabel('Total_Income')
P = plt.ylabel('Percentage')
```



We can see that the proportion of loans approved for applicants with low Total Income is much lower than for applicants with Average, High, and Very High Income.

```
bins=[0,100,200,700]
group=['Low','Average','High']
loan_data['LoanAmount_bin']=pd.cut(loan_data['LoanAmount'],bins,labels=
group)

LoanAmount_bin=pd.crosstab(loan_data['LoanAmount_bin'],loan_data['Loan_
Status'])
LoanAmount_bin.div(LoanAmount_bin.sum(1).astype(float), axis=0).plot(ki
nd="bar", stacked=True)
plt.xlabel('LoanAmount')
P = plt.ylabel('Percentage')
```



It can be seen that the proportion of approved loans is higher for Low and Average Loan Amounts than for High Loan Amounts, which supports our hypothesis that the chances of loan approval are higher when the loan amount is less.

Drop unwanted columns:

```
loan_data.drop("Income_bin", axis=1,inplace=True,errors='ignore')
loan_data.drop("Coapplicant_Income_bin", axis=1,inplace=True,errors='ig
nore')

loan_data.drop("Total_Income_bin", axis=1,inplace=True,errors='ignore')
loan_data.drop("LoanAmount_bin", axis=1,inplace=True,errors='ignore')
```

Techniques for Evaluation

The Pandas `dataframe.corr()` function returns the pairwise correlation of all columns in a dataframe.

- Any na values are automatically filtered out.
- Any column in the dataframe with a non-numeric data type will be ignored.
- Parameters for dataframe.corr: dataframe.corr(method='',min periods=1)

Techniques:

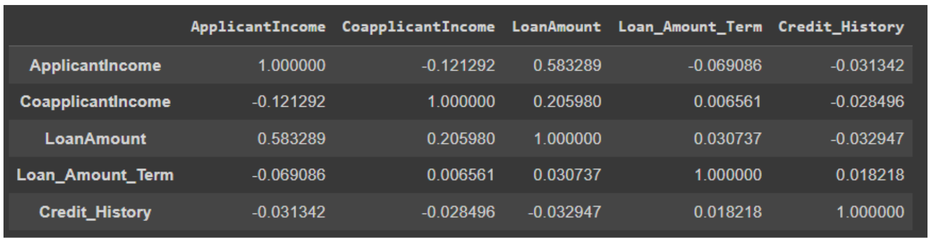
- **Pearson**: Standard correlation coefficient
- **Spearman**: Spearman rank correlation
- **Kendall**: Kendall Tau correlation coefficient

The minimum number of observations required for a valid result for each pair of columns. This is only available for pearson and spearman correlations at the moment.

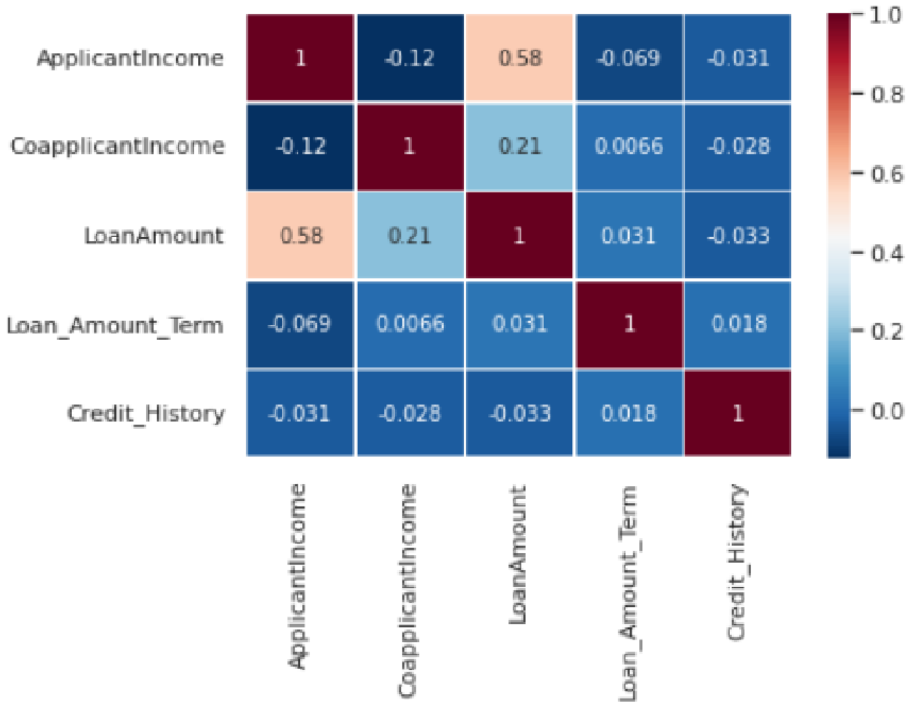
A co-efficient close to one indicates that the two variables have a very strong positive correlation. The maroon has very strong correlations in our case. The diagonal line represents the correlation of the variables to themselves, so they must be 1.

Pearson Correlation Coefficient

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb

pearsoncorr = loan_data.corr(method='pearson')
pearsoncorr
import seaborn as sb

sb.heatmap(pearsoncorr,
            xticklabels=pearsoncorr.columns,
            yticklabels=pearsoncorr.columns,
            cmap='RdBu_r',
            annot=True,
            linewidth=0.5)
```



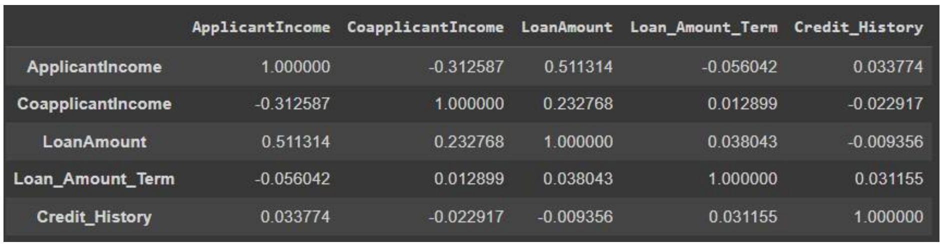
	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
ApplicantIncome	1.000000	-0.121292	0.583289	-0.069086	-0.031342
CoapplicantIncome	-0.121292	1.000000	0.205980	0.006561	-0.028496
LoanAmount	0.583289	0.205980	1.000000	0.030737	-0.032947
Loan_Amount_Term	-0.069086	0.006561	0.030737	1.000000	0.018218
Credit_History	-0.031342	-0.028496	-0.032947	0.018218	1.000000



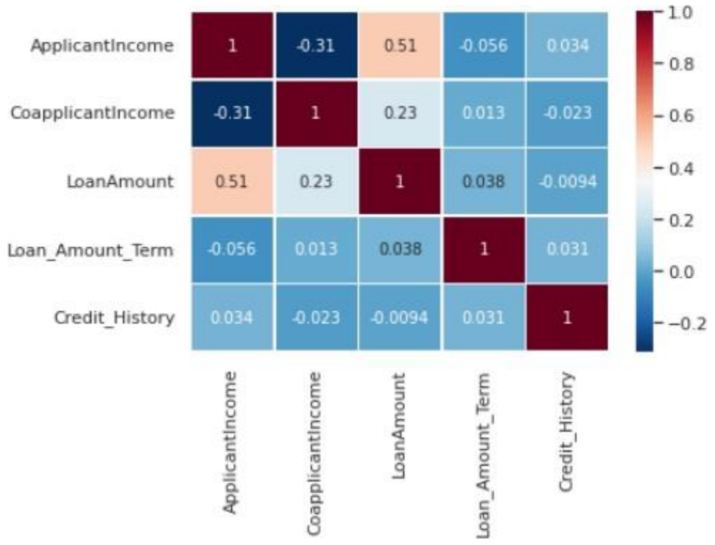
We can quickly see that the Loan Amount is strongly correlated with the Loan Status. The income of the coapplicant is strongly related to the Loan Status. A high Coapplicant income increases the likelihood of Loan Approval.

Spearman Correlation Coefficient

```
import pandas as pd
from pylab import rcParams
import seaborn as sb
from scipy.stats.stats import kendalltau

# Data Visualisation Settings
%matplotlib inline
sb.set_style('whitegrid')

kendall = loan_data.corr(method='kendall')
kendall
```



	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
ApplicantIncome	1.000000	-0.312587	0.511314	-0.056042	0.033774
CoapplicantIncome	-0.312587	1.000000	0.232768	0.012899	-0.022917
LoanAmount	0.511314	0.232768	1.000000	0.038043	-0.009356
Loan_Amount_Term	-0.056042	0.012899	0.038043	1.000000	0.031155
Credit_History	0.033774	-0.022917	-0.009356	0.031155	1.000000



The Pearson correlation coefficient is calculated using raw data values, whereas the Spearman correlation coefficient is calculated using individual value ranks. The Pearson correlation coefficient measures the linear relationship between two variables, whereas the Spearman rank correlation coefficient measures the monotonic relationship. We need a basic understanding of monotonic functions to understand the Spearman correlation.

The Spearman rank correlation coefficient assesses the monotony between two variables. Its values range from -1 to +1 and can be deduced as follows:

- **+1:** Perfectly monotonically increasing relationship

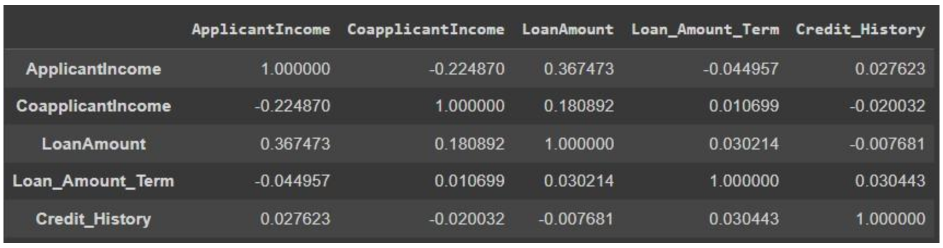
- **+0.8:** Strong monotonically increasing relationship
- **+0.2:** Weak monotonically increasing relationship
- **0:** Non-monotonic relation
- **-0.2:** Weak monotonically decreasing relationship
- **-0.8:** Strong monotonically decreasing relationship
- **-1:** Perfectly monotonically decreasing relationship

Considering the spearman correlation:

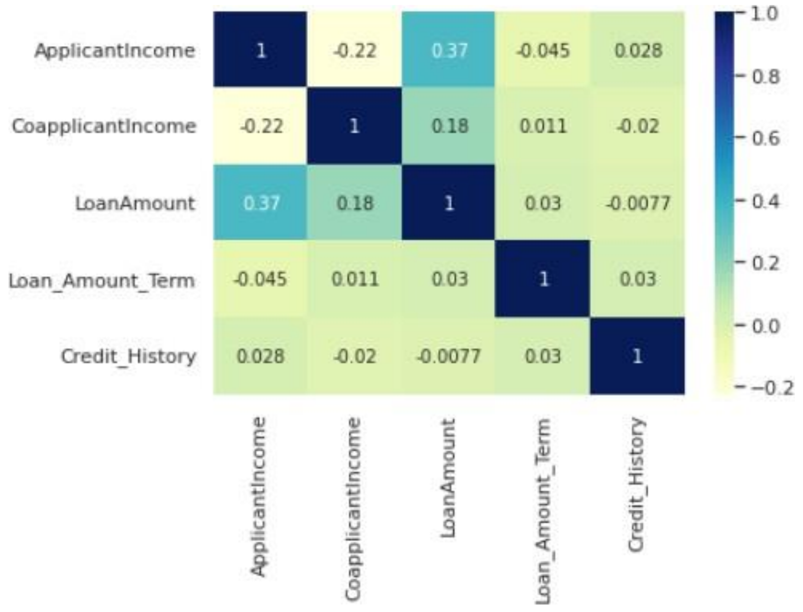
- The Loan Amount has strong monotonically increasing relationship with Loan Status.
- The co applicant income has Weak monotonically decreasing relationship with Loan Status.

Kendall Correlation Coefficient

```
sb.heatmap(kendall,
           xticklabels=kendall.columns.values,
           yticklabels=kendall.columns.values,
           cmap="YlGnBu",
          annot=True)
```



	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
ApplicantIncome	1.000000	-0.224870	0.367473	-0.044957	0.027623
CoapplicantIncome	-0.224870	1.000000	0.180892	0.010699	-0.020032
LoanAmount	0.367473	0.180892	1.000000	0.030214	-0.007681
Loan_Amount_Term	-0.044957	0.010699	0.030214	1.000000	0.030443
Credit_History	0.027623	-0.020032	-0.007681	0.030443	1.000000



Even with Kendall technique, the correlation between Coapplicant income vs Loan status or Loan Amount vs Loan Status seem be similar to above techniques.

Comparing all 3 techniques, Pearson Correlation Coefficient seem to work the best considering the raw data values.

Data Cleaning

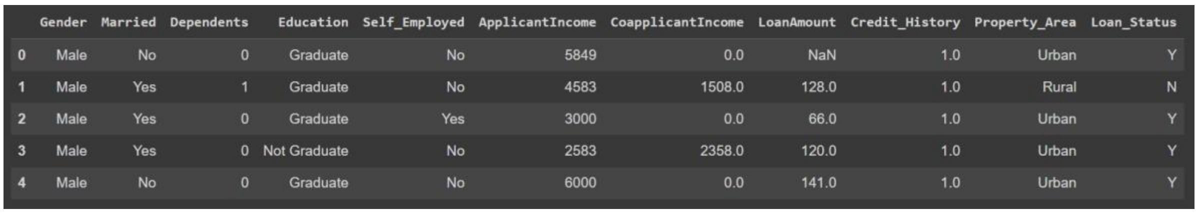
Converting Categorical to Numeric variables

We drop the bins we made for the exploration phase. Furthermore, changing the '3+' in dependent variables to 3 converts it to a numerical variable. Similarly, we convert the target variable's categories to 0 and 1 to determine its relationship with numerical variables. All the more reason to, because algorithms such as Logistic Regression only accept numerical values as input.

Missing Value Imputation

The table below displays the total amount of missing or corrupt data in our database. To correct this, we replace missing categorical variables with their modes and missing numerical variables with their means.

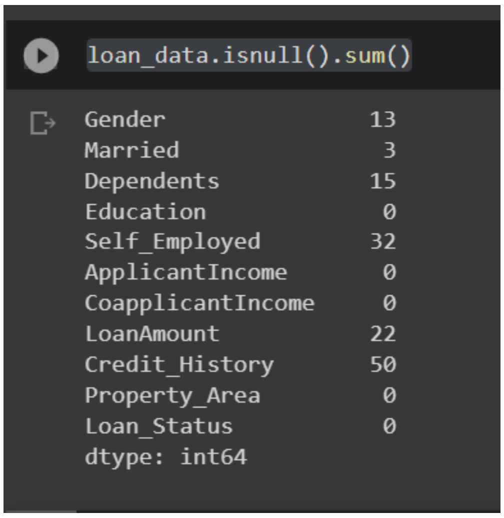
```
#Drop unwanted columns in train data

loan_data.drop("Loan_ID", axis=1,inplace=True,errors='ignore')
loan_data.drop("Loan_Amount_Term", axis=1,inplace=True,errors='ignore')
loan_data.drop("Total_Income", axis=1,inplace=True,errors='ignore')
loan_data.head()
```



	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Credit_History	Property_Area	Loan_Status
0	Male	No	0	Graduate	No	5849	0.0	NaN	1.0	Urban	Y
1	Male	Yes	1	Graduate	No	4583	1508.0	128.0	1.0	Rural	N
2	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	1.0	Urban	Y
3	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	1.0	Urban	Y
4	Male	No	0	Graduate	No	6000	0.0	141.0	1.0	Urban	Y

```
## identify Missing values
```

```
loan_data.isnull().sum()
```

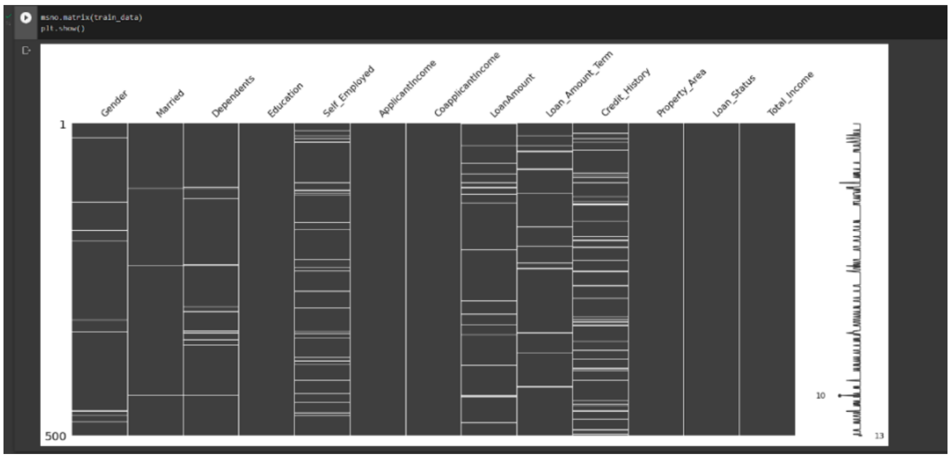


```
loan_data.isnull().sum()
```

```
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount           22
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64
```

The missing variables can also be plotted on a matrix using msno library as shown below which gives a graphical representation.

```
msno.matrix(loan_data)
plt.show()
```



```
msno.matrix(train_data)
plt.show()
```

Since, we are having missing variables across various columns of data. We are going to impute those missing ones with most repetitive variables.

- For numerical variables: imputation using mean or median
- For categorical variables: imputation using mode

Gender, Married, Dependents, Credit History, and Self Employed have very few missing values, so we can fill them using the mode of the features.

```
## Imputing the missing values
```

```
## Imputing the missing values

for i in [loan_data]:
  i["Gender"] = i["Gender"].fillna(loan_data.Gender.dropna().mode()[0])
  i["Married"] = i["Married"].fillna(loan_data.Married.dropna().mode()[
0])
  i["Dependents"] = i["Dependents"].fillna(loan_data.Dependents.dropna(
).mode()[0])
  i["Self_Employed"] = i["Self_Employed"].fillna(loan_data.Self_Employe
d.dropna().mode()[0])
  i["Credit_History"] = i["Credit_History"].fillna(loan_data.Credit_His
tory.dropna().mode()[0])
```

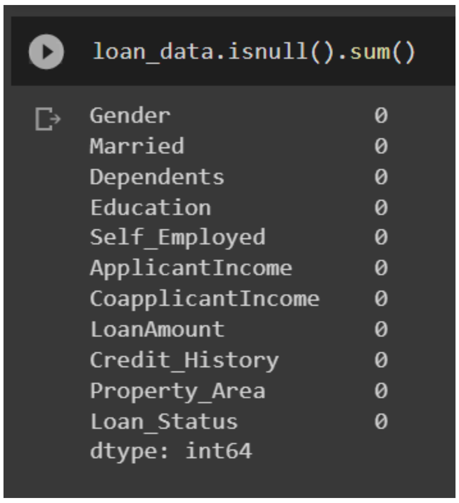
Let's now try to figure out how to fill in the blanks in Loan Amount and Loan Amount Term. Because it is a numerical variable, we can use the mean or median to fill in the blanks. We will use the median to fill in the null values because we saw earlier that "loan amount" has outliers, so the mean will not be appropriate because it is heavily influenced by the presence of outliers.

```
## imputing Missing values
```

```
loan_data['LoanAmount'].fillna(loan_data['LoanAmount'].median(),  inplace=True)
```

Now let’s check whether all the missing values are filled in the dataset.

```
## identify Missing values
train_data.isnull().sum()
```



```
loan_data.isnull().sum()
```

```
Gender              0
Married             0
Dependents          0
Education           0
Self_Employed       0
ApplicantIncome     0
CoapplicantIncome   0
LoanAmount          0
Credit_History      0
Property_Area       0
Loan_Status         0
dtype: int64
```

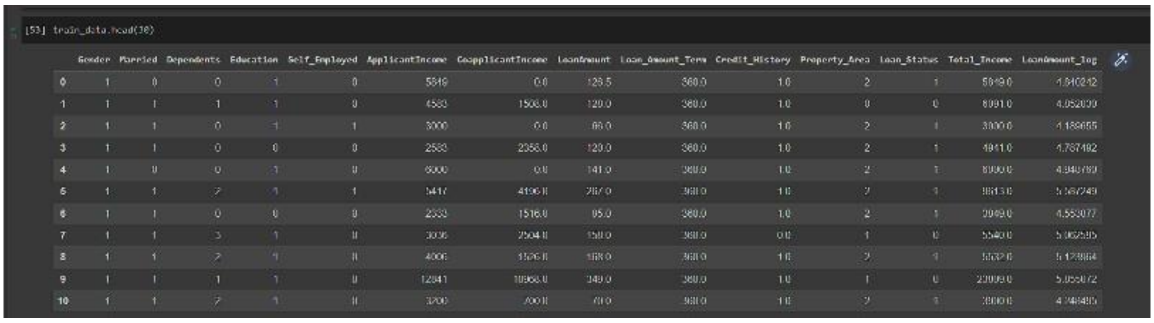
As we can see, all of the missing values in the test dataset have been filled. Let's use the same method to fill in all the missing values in the test dataset.

```
from urllib import error
loan_data["Loan_Status"] = loan_data["Loan_Status"].map({'N':0, "Y":1})
.astype(int)

# convert categorical columns to numerical values
loan_data.replace({'Married':{'No':0,'Yes':1},'Gender':{'Male':1,'Femal
e':0},'Self_Employed':{'No':0,'Yes':1},
                     'Property_Area':{'Rural':0,'Semiurban':1,'Urban':
2},'Education':{'Graduate':1,'Not Graduate':0}},inplace=True)
```

Verify if all the categorical values are converted to numerical using the .head() function.

```
loan_data["Dependents"] = loan_data["Dependents"]
.map({"3+":'3', "3":'3', "2":'2', "1":'1',"0":'0'})

loan_data['Dependents'] = loan_data['Dependents'].astype('int')
```

We can observe that the entire samples are converted into numerical variables to better determine relationships between the variables.

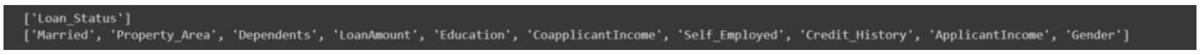


```
[53] train_data.head(10)
```

	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Area	Loan_Status	Total_Income	LoanAmount_log
0	1	0	0	1	0	5849	0.0	128.5	360.0	1.0	2	1	5849.0	4.840242
1	1	1	1	1	0	4583	1508.0	120.0	360.0	1.0	0	0	6091.0	4.852030
2	1	1	0	1	1	3000	0.0	66.0	360.0	1.0	2	1	3000.0	4.189655
3	1	1	0	0	0	2583	2358.0	120.0	360.0	1.0	2	1	4941.0	4.787492
4	1	0	0	1	0	6000	0.0	141.0	360.0	1.0	2	1	6000.0	4.948760
5	1	1	2	1	1	5417	4196.0	267.0	360.0	1.0	2	1	9613.0	5.587249
6	1	1	0	0	0	2333	1516.0	95.0	360.0	1.0	2	1	3849.0	4.553877
7	1	1	3	1	0	3036	2504.0	158.0	360.0	0.0	1	0	5540.0	5.062595
8	1	1	2	1	0	4006	1526.0	168.0	360.0	1.0	2	1	5532.0	5.123964
9	1	1	1	1	0	12841	10968.0	349.0	360.0	1.0	1	0	23809.0	5.855072
10	1	1	2	1	0	3200	700.0	70.0	360.0	1.0	2	1	3900.0	4.248495

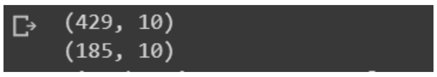
The next step is to create the train and test datasets. This is done using the code below. The last line of code prints the shape of the training set (420 observations of 7 variables) and test set (180 observations of 7 variables).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn

from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from math import sqrt

target_column = ['Loan_Status']
predictors = list(set(list(loan_data.columns))-set(target_column))

print(target_column)
print(predictors)
```



```
['Loan_Status']
['Married', 'Property_Area', 'Dependents', 'LoanAmount', 'Education', 'CoapplicantIncome', 'Self_Employed', 'Credit_History', 'ApplicantIncome', 'Gender']
```

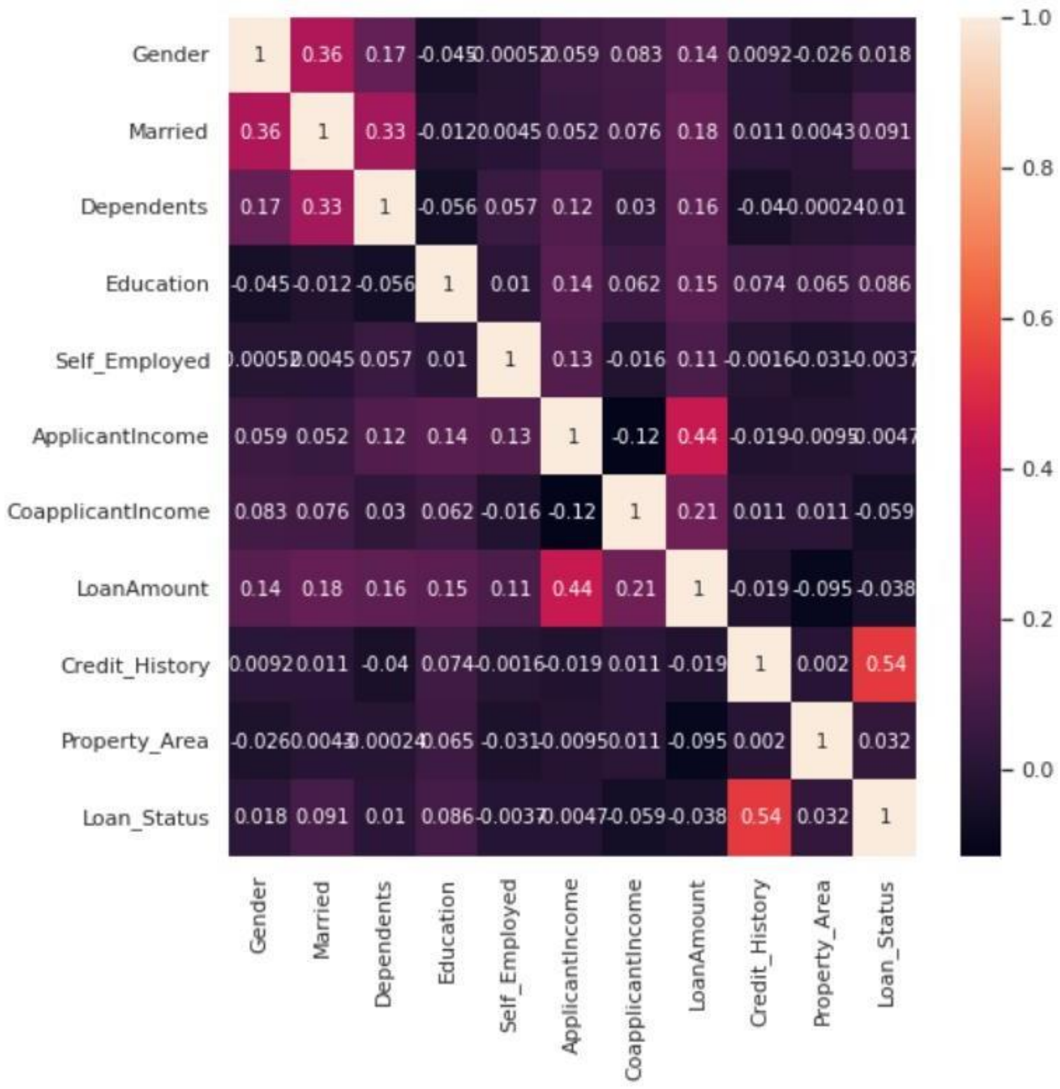
```
# X = loan_data[predictors].values
X  = loan_data.drop('Loan_Status',1)
# y = loan_data[target_column].values
y = loan_data.Loan_Status

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0
.3)
print(X_train.shape); print(X_test.shape)
```



```
(429, 10)
(185, 10)
```

Visualizing correlation via Headmap

```
plt.figure(figsize=(8,8))
correlation_matrix=loan_data.corr()
sns.heatmap(correlation_matrix,annot=True)
plt.show
```



We can see that (ApplicantIncome - LoanAmount) and (Credit History - Loan Status) are the most correlated variables. LoanAmount is also linked to CoapplicantIncome.

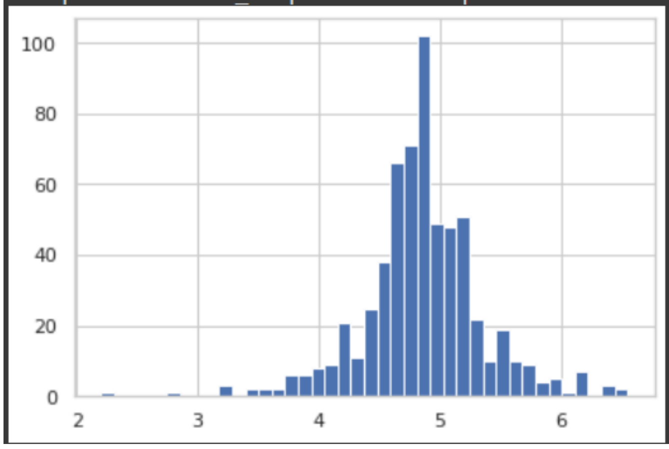
Outlier Treatment

Log Transformation

As a result of outliers in the Loan Amount. In the numerical analysis plot above, the data in the loan amount is skewed to the right, indicating that the majority of the data is to the left. A log transformation is used to remove the skewness. A log transformation has little effect on the smaller values but greatly reduces the larger values. As a result, the distribution becomes normal.

```
# Outlier treatment

loan_data['LoanAmount'] = np.log(loan_data['LoanAmount'])
loan_data['LoanAmount'].hist(bins=40)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa7d1ea31c0>
```



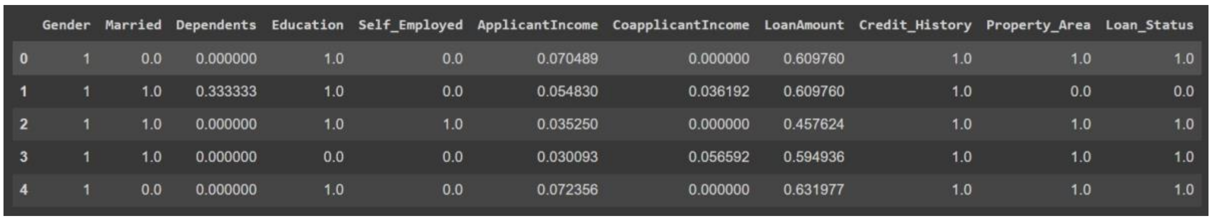
We can observe that after log transformation, the distribution is normal i.e centered.

Scaling the Values - Min Max scaler

We will be using min-max scaler which scales down the value of all columns between 0 to 1. The formula for min-max scaler is shown below.

```
for i in loan_data.columns[1:]:
    loan_data[i] = (loan_data[i] - loan_data[i].min()) / (loan_data[i].
max() - loan data[i].min())
```

$$x\,norm = \frac{x - x\,min}{x\max - x\,min}$$



	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Credit_History	Property_Area	Loan_Status
0	1	0.0	0.000000	1.0	0.0	0.070489	0.000000	0.609760	1.0	1.0	1.0
1	1	1.0	0.333333	1.0	0.0	0.054830	0.036192	0.609760	1.0	0.0	0.0
2	1	1.0	0.000000	1.0	1.0	0.035250	0.000000	0.457624	1.0	1.0	1.0
3	1	1.0	0.000000	0.0	0.0	0.030093	0.056592	0.594936	1.0	1.0	1.0
4	1	0.0	0.000000	1.0	0.0	0.072356	0.000000	0.631977	1.0	1.0	1.0

Model Evaluation

Let us start by predicting the target variable with our first model. We'll begin with Logistic Regression, which is used to forecast binary outcomes.

- A classification algorithm is Logistic Regression. Given a set of independent variables, it predicts a binary outcome (1 / 0, Yes / No, True / False).
- The Logit function is estimated using logistic regression. The logit function is simply a log of the event's odds.
- This function, which is very similar to the required stepwise function, generates an S-shaped curve with the probability estimate.

Loan ID variable is removed because it has no effect on the loan status and same are applied to test data. For creating various models, we will use "scikit-learn" (sklearn), an open-source Python library. It is one of the most efficient tools, with many built-in functions that can be used for Python modeling. Sklearn requires a separate dataset for the target variable. As a result, we will remove our target variable from the training dataset and save it in a different dataset.

We'll now create dummy variables for the categorical variables. A dummy variable converts categorical variables into a series of 0 and 1, making it much easier to quantify and compare them. But since we already converted categorical variables to numerical. We can use the same dataset for analysis.

```
X = train_data.drop('Loan_Status',1)
y = train_data.Loan_Status
X=pd.get_dummies(X)
train=pd.get_dummies(train_data)
test=pd.get_dummies(test_data)
```

We'll now train the model on the training data and make predictions for the test data. Can we, however, validate these predictions? One method is to split our train dataset into two parts: train and validation. On this training part, we can train the model and use it to make predictions for the validation part. As a result of having the true predictions for the validation part, we can validate our predictions (which we do not have for the test dataset).
To divide our train dataset, we will use the sklearn train test split function. Let us begin by importing train test split.

```
from sklearn.model_selection import train_test_split
x_train, x_cv, y_train, y_cv = train_test_split(X,y, test_size =0.3)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

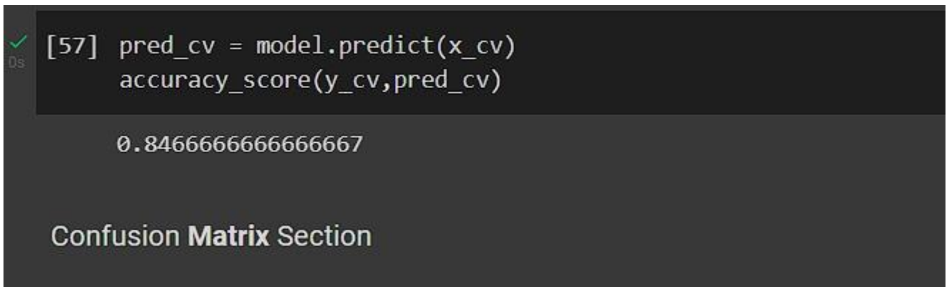
The dataset has been divided into two parts: training and validation. Let's use sklearn to import LogisticRegression and accuracy score and fit the logistic regression model.

```
model = LogisticRegression()
model.fit(x_train, y_train)
```

The C parameter represents the inverse of regularization strength in this case. In order to reduce overfitting, regularization penalizes increasing the magnitude of parameter values. Smaller C values indicate more regularization. Refer to this page to learn about other parameters:

Let us predict and calculate the Loan Status for the validation set. Let us calculate the accuracy to see how accurate our predictions are.

```
pred_cv = model.predict(x_cv)
accuracy_score(y_cv,pred_cv)
```



```
[57] pred_cv = model.predict(x_cv)
     accuracy_score(y_cv,pred_cv)

     0.8466666666666667
```

Confusion **Matrix** Section

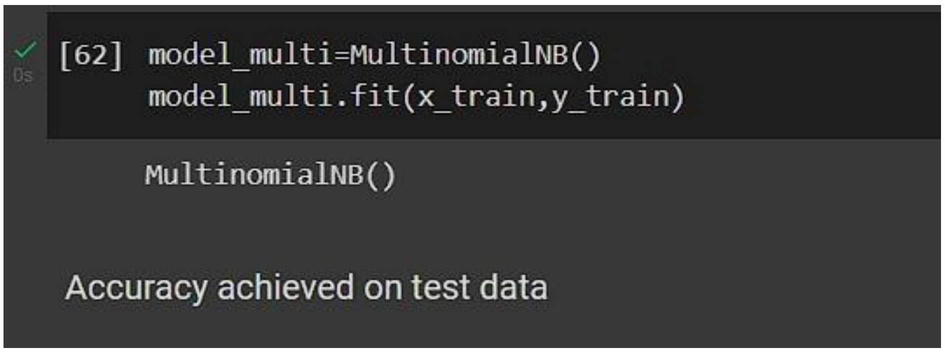
So our predictions are almost 80% accurate, i.e. we have identified 80% of the loan status correctly.

```
x=train_data.drop('Loan_Status',1)
y=train_data.Loan_Status
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size =0.3)
```

Let plot confusion matrix to get metrics will you use for your model evaluation

Accuracy, precision, recall, sensitivity, specificity, AUC, F1

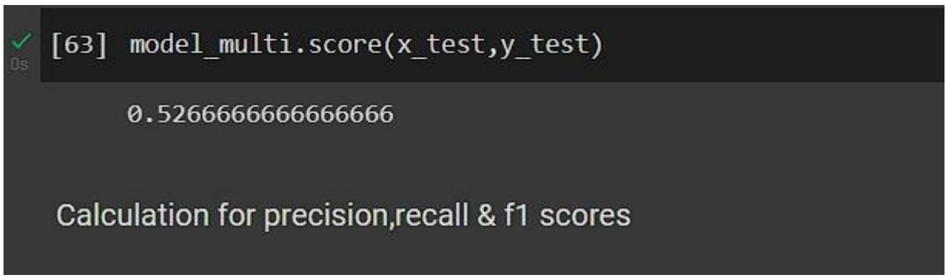
```
model_multi=MultinomialNB()
model_multi.fit(x_train,y_train)
```



```
[62] model_multi=MultinomialNB()
     model_multi.fit(x_train,y_train)

     MultinomialNB()
```

Accuracy achieved on test data

```
model_multi.score(x_test,y_test)
```

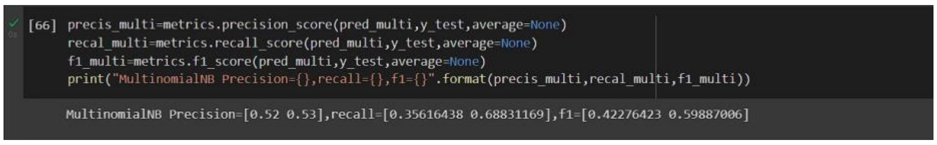


```
[63] model_multi.score(x_test,y_test)

     0.5266666666666666
```

Calculation for precision,recall & f1 scores

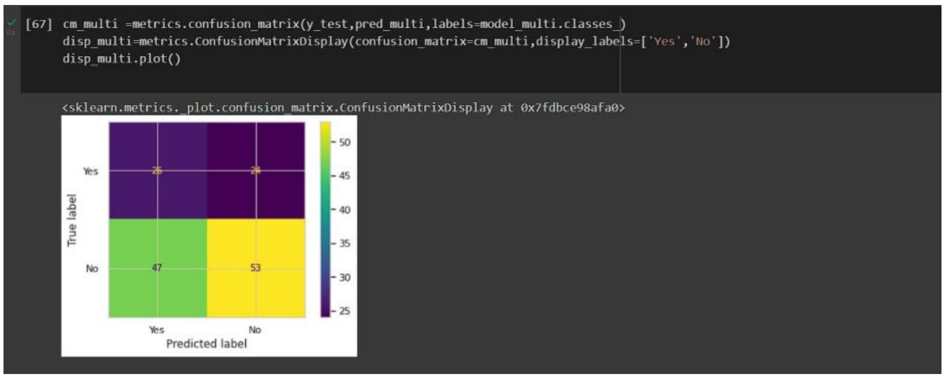
We calculated accuracy using Multinomial NB classifiers using score method by passing x_test & y_test as arguments of the method.

```
pred_multi=model_multi.predict(x_test)
precis_multi=metrics.precision_score(pred_multi,y_test,average=None)
recal_multi=metrics.recall_score(pred_multi,y_test,average=None)
f1_multi=metrics.f1_score(pred_multi,y_test,average=None)
print("MultinomialNB
Precision={},recall={},f1={}".format(precis_multi,recal_multi,f1_multi))
```



```
[66] precis_multi=metrics.precision_score(pred_multi,y_test,average=None)
     recal_multi=metrics.recall_score(pred_multi,y_test,average=None)
     f1_multi=metrics.f1_score(pred_multi,y_test,average=None)
     print("MultinomialNB Precision={},recall={},f1={}".format(precis_multi,recal_multi,f1_multi))

     MultinomialNB Precision=[0.52 0.53],recall=[0.35616438 0.68831169],f1=[0.42276423 0.59887006]
```

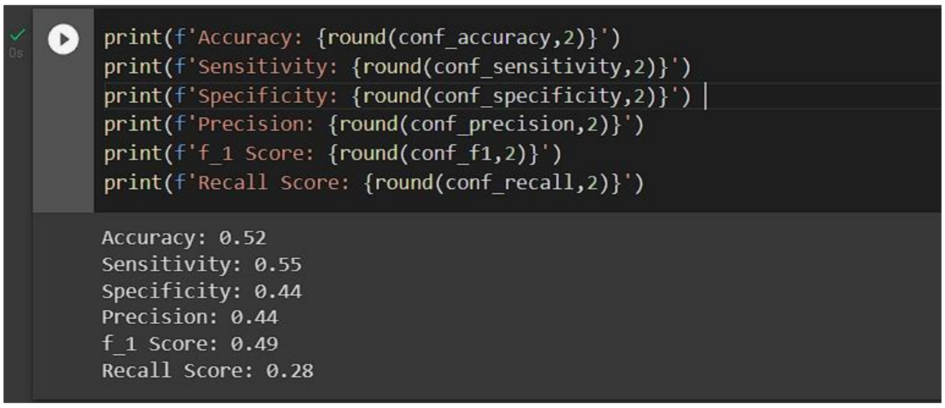
```
cm_multi =
metrics.confusion_matrix(y_test,pred_multi,labels=model_multi.classes_)
disp_multi=metrics.ConfusionMatrixDisplay(confusion_matrix=cm_multi,display
_labels=['Yes','No'])
disp_multi.plot()
```



```
[67] cm_multi =metrics.confusion_matrix(y_test,pred_multi,labels=model_multi.classes_)
     disp_multi=metrics.ConfusionMatrixDisplay(confusion_matrix=cm_multi,display_labels=['Yes','No'])
     disp_multi.plot()

     <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fdbce98afa0>
```

Confusion matrix is visualization of an algorithm. As per the matrix, each row represents the instances in predicted class and the column represents the instance in true class. It is useful for measuring the scores: Precision, recall, f1,sensitivity,specificity, and AUC. We have calculated these metrics using confusion matrix values.

```
print(f'Accuracy: {round(conf_accuracy,2)}')
print(f'Sensitivity: {round(conf_sensitivity,2)}')
print(f'Specificity: {round(conf_specificity,2)}')
print(f'Precision: {round(conf_precision,2)}')
print(f'f_1 Score: {round(conf_f1,2)}')
print(f'Recall Score: {round(conf_recall,2)}')
```



```
print(f'Accuracy: {round(conf_accuracy,2)}')
print(f'Sensitivity: {round(conf_sensitivity,2)}')
print(f'Specificity: {round(conf_specificity,2)}')
print(f'Precision: {round(conf_precision,2)}')
print(f'f_1 Score: {round(conf_f1,2)}')
print(f'Recall Score: {round(conf_recall,2)}')
```

```
Accuracy: 0.52
Sensitivity: 0.55
Specificity: 0.44
Precision: 0.44
f_1 Score: 0.49
Recall Score: 0.28
```

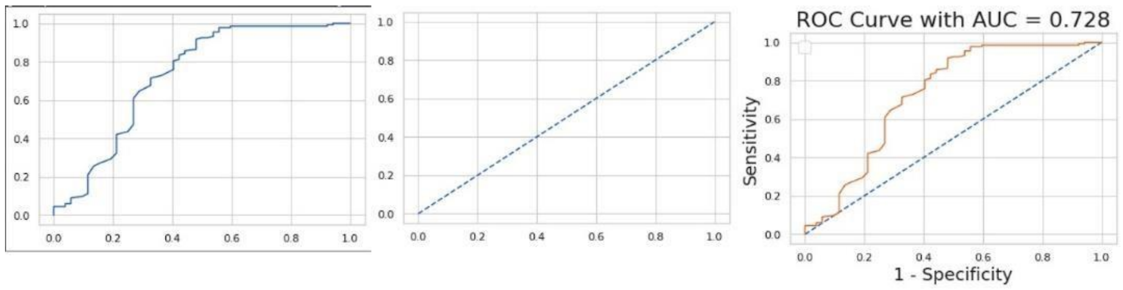
Using these values, we can calculate the accuracy of the model. The accuracy is given by:

Accuracy = True Positives + true negatives / (True Positive + True Negatives + False Positives + False Negatives)= **0.52**
Precision:-: It is a measure of correctness achieved in true prediction i.e. of observations labeled as true, how many are actually labeled true
Precision = TP / (TP + FP) =**0.44**
Recall:- It is a measure of actual observations which are predicted correctly i.e. how many observations of true class are labeled correctly.
Recall = TP / (TP + FN)=**0.28**
Specificity:- It is a measure of how many observations of false class are labeled correctly.
Specificity = TN / (TN + FP)=**0.44**
F1 Score:- It is a metric which takes into account both ***precision*** and ***recall*** and is defined as follows: f1 Score= 2*(precision*recall)/precision+recall=**0.49**

Plotting ROC curve with AOC:

```
logreg = LogisticRegression()
logreg.fit(x_train, y_train)    # model fitting
y_pred = logreg.predict(x_test)    # Predictions

pred_proba = [i[1] for i in logreg.predict_proba(x_test)]
pred_df = pd.DataFrame({'true_values': y_test,
                        'pred_probs':pred_proba})

plt.figure(figsize = (10,7))

thresholds = np.linspace(0, 1, 200)

# Define function to calculate sensitivity. (True positive rate.)
def TPR(df, true_col, pred_prob_col, threshold):
    true_positive = df[(df[true_col] == 1) & (df[pred_prob_col] >=
threshold)].shape[0]
    false_negative = df[(df[true_col] == 1) & (df[pred_prob_col] <
threshold)].shape[0]
    return true_positive / (true_positive + false_negative)

# Define function to calculate 1 - specificity. (False positive rate.)
def FPR(df, true_col, pred_prob_col, threshold):
    true_negative = df[(df[true_col] == 0) & (df[pred_prob_col] <=
threshold)].shape[0]
    false_positive = df[(df[true_col] == 0) & (df[pred_prob_col] >
threshold)].shape[0]
    return 1 - (true_negative / (true_negative + false_positive))
```

```
# Calculate sensitivity & 1-specificity for each threshold between 0 and 1.
tpr_values = [TPR(pred_df, 'true_values', 'pred_probs', prob) for prob in
thresholds]
fpr_values = [FPR(pred_df, 'true_values', 'pred_probs', prob) for prob in
thresholds]

# Plot ROC curve.
plt.plot(fpr_values, # False Positive Rate on X-axis
         tpr_values, # True Positive Rate on Y-axis
         label='ROC Curve')

# Plot baseline. (Perfect overlap between the two populations.)
plt.plot(np.linspace(0, 1, 200),
         np.linspace(0, 1, 200),
         label='baseline',
         linestyle='--')

# Label axes.
plt.title(f"ROC Curve with AUC =
{round(metrics.roc_auc_score(pred_df['true_values'],
pred_df['pred_probs']),3)}", fontsize=22)
plt.ylabel('Sensitivity', fontsize=18)
plt.xlabel('1 - Specificity', fontsize=18)
# Create legend.
plt.legend(fontsize=16);
plt.plot(np.linspace(0, 1, 200),
         np.linspace(0, 1, 200),
         label='baseline',
         linestyle='--')
plt.plot(fpr_values, # False Positive Rate on X-axis
         tpr_values, # True Positive Rate on Y-axis
         label='ROC Curve')
```



Classifiers

Machine learning models

First of all we will divide our dataset into two variables X as the features we defined earlier and y as the Loan_Status the target value we want to predict.

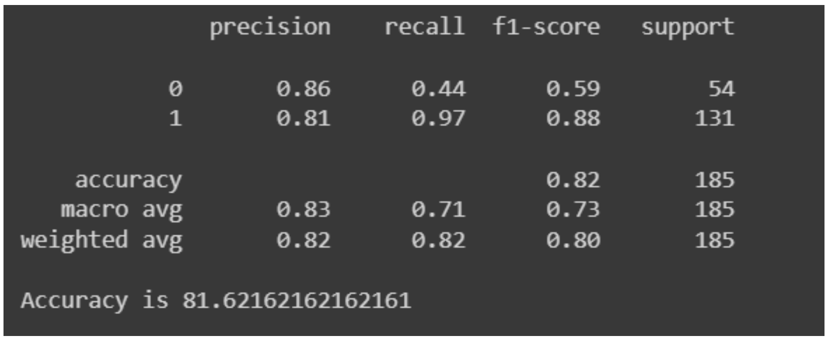
Models we will use:

- **Decision Tree**
- **Random Forest**
- **XGBoost**
- **Logistic Regression**

Logistic Regression

To check how robust our model is to unseen data, we can use Validation. It is a technique that involves reserving a particular sample of a dataset on which you do not train the model.

```
from sklearn.metrics import make_scorer, accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

LR = LogisticRegression()
LR.fit(X_train, y_train)

y_predict  =  LR.predict(X_test)

#  prediction Summary by species
print(classification_report(y_test,  y_predict))

# Accuracy score
LR_SC = accuracy_score(y_predict,y_test)
print('Accuracy is',accuracy_score(y_predict,y_test)*100)
print('')
```

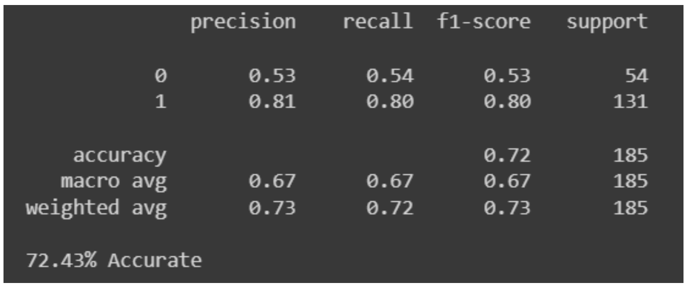


```
              precision    recall  f1-score   support

           0       0.86      0.44      0.59        54
           1       0.81      0.97      0.88       131

    accuracy                           0.82       185
   macro avg       0.83      0.71      0.73       185
weighted avg       0.82      0.82      0.80       185

Accuracy is 81.62162162162161
```

Decision Tree

Decision Trees (DTs) are a type of non-parametric supervised learning method that is used for classification and regression. The goal is to build a model that predicts the value of a target variable using simple decision rules derived from data features.

```
# Decision Tree

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

DT = DecisionTreeClassifier()
DT.fit(X_train, y_train)

y_predict  =  DT.predict(X_test)

#  prediction Summary by species
print(classification_report(y_test,  y_predict))

# Accuracy score
DT_SC = accuracy_score(y_predict,y_test)
print(f"{round(DT_SC*100,2)}% Accurate")
```



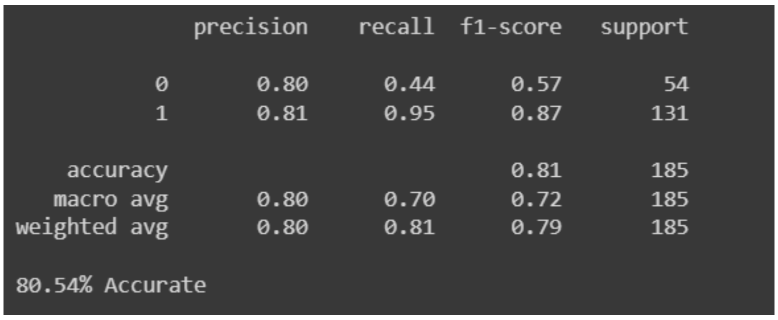
```
              precision    recall  f1-score   support

           0       0.53      0.54      0.53        54
           1       0.81      0.80      0.80       131

    accuracy                           0.72       185
   macro avg       0.67      0.67      0.67       185
weighted avg       0.73      0.72      0.73       185

72.43% Accurate
```

Random Forest

A classifier based on random forest. A random forest is a meta estimator that uses averaging to improve predictive accuracy and control over-fitting by fitting a number of decision tree classifiers on different sub-samples of the dataset.

```
# Random forest

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

RF = RandomForestClassifier()
RF.fit(X_train, y_train)

y_predict  =  RF.predict(X_test)
#  prediction Summary by species
```

```
print(classification_report(y_test,  y_predict))

# Accuracy score
RF_SC = accuracy_score(y_predict,y_test)
print(f"{round(RF_SC*100,2)}% Accurate")
```

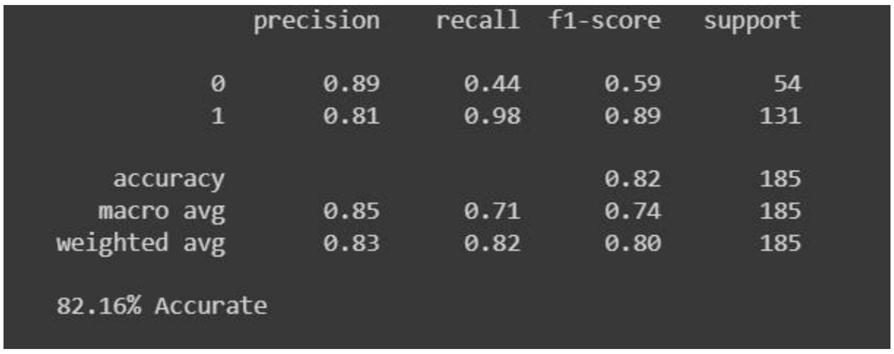


```
              precision    recall  f1-score   support

           0       0.80      0.44      0.57        54
           1       0.81      0.95      0.87       131

    accuracy                           0.81       185
   macro avg       0.80      0.70      0.72       185
weighted avg       0.80      0.81      0.79       185

80.54% Accurate
```

XGBoost

XGBoost is a popular and efficient open-source gradient boosted trees implementation.

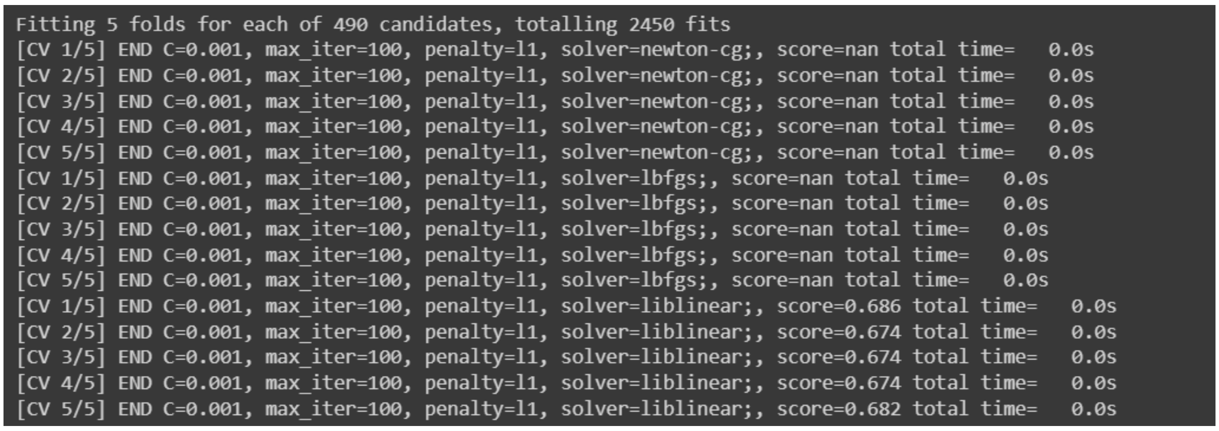
```
# XGBoost

from xgboost import XGBClassifier
from sklearn.metrics import classification_report

XGB = XGBClassifier()
XGB.fit(X_train, y_train)

y_predict = XGB.predict(X_test)

#  prediction Summary by species
print(classification_report(y_test,  y_predict))

# Accuracy score
XGB_SC = accuracy_score(y_predict,y_test)
print(f"{round(XGB_SC*100,2)}% Accurate")
print('')
```



```
              precision    recall  f1-score   support

           0       0.89      0.44      0.59        54
           1       0.81      0.98      0.89       131

    accuracy                           0.82       185
   macro avg       0.85      0.71      0.74       185
weighted avg       0.83      0.82      0.80       185

82.16% Accurate
```

Improve Accuracy for Logistic Regression Model using GridSearchCV

```
rom sklearn.model_selection import GridSearchCV

LRparam_grid = {
    'C' : [0.001, 0.01, 0.1, 1, 10, 100, 1000],
    'penalty': ['l1', 'l2'],
    'max_iter': list(range(100,800,100)),
    'solver': ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
}

LR_search = GridSearchCV(LogisticRegression(), LRparam_grid, refit = Tr
ue, verbose =3, cv=5)
LR_search.fit(X_train, y_train)
LR_search.best_params_

print('Mean Accuracy: %.3f' % LR_search.best_score_)
print('Config: %s' % LR_search.best_params_)
```



```
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
[CV 1/5] END C=0.001, max_iter=100, penalty=l1, solver=newton-cg;, score=nan total time=   0.0s
[CV 2/5] END C=0.001, max_iter=100, penalty=l1, solver=newton-cg;, score=nan total time=   0.0s
[CV 3/5] END C=0.001, max_iter=100, penalty=l1, solver=newton-cg;, score=nan total time=   0.0s
[CV 4/5] END C=0.001, max_iter=100, penalty=l1, solver=newton-cg;, score=nan total time=   0.0s
[CV 5/5] END C=0.001, max_iter=100, penalty=l1, solver=newton-cg;, score=nan total time=   0.0s
[CV 1/5] END C=0.001, max_iter=100, penalty=l1, solver=lbfgs;, score=nan total time=   0.0s
[CV 2/5] END C=0.001, max_iter=100, penalty=l1, solver=lbfgs;, score=nan total time=   0.0s
[CV 3/5] END C=0.001, max_iter=100, penalty=l1, solver=lbfgs;, score=nan total time=   0.0s
[CV 4/5] END C=0.001, max_iter=100, penalty=l1, solver=lbfgs;, score=nan total time=   0.0s
[CV 5/5] END C=0.001, max_iter=100, penalty=l1, solver=lbfgs;, score=nan total time=   0.0s
[CV 1/5] END C=0.001, max_iter=100, penalty=l1, solver=liblinear;, score=0.686 total time=   0.0s
[CV 2/5] END C=0.001, max_iter=100, penalty=l1, solver=liblinear;, score=0.674 total time=   0.0s
[CV 3/5] END C=0.001, max_iter=100, penalty=l1, solver=liblinear;, score=0.674 total time=   0.0s
[CV 4/5] END C=0.001, max_iter=100, penalty=l1, solver=liblinear;, score=0.674 total time=   0.0s
[CV 5/5] END C=0.001, max_iter=100, penalty=l1, solver=liblinear;, score=0.682 total time=   0.0s
```

The improved accuracy is 83%.

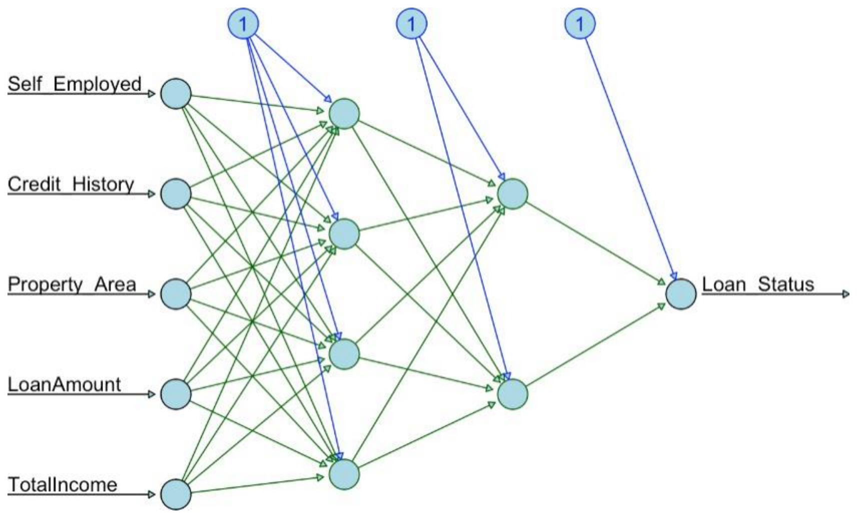
Accuracy	**Improved Accuracy**	**Precision**	**f1**	**Recall**	**ROC curve with AUC**
81 %	83 %	0.45	0.49	0.27	0.722

Potential NN Architecture

The given model is a densely connected deep neural network with multiple hidden layers and an output layer. To train a neural network following objects are to be considered:

1. Layers: These are combined into a network]
2. The input data and associated targets
3. The loss function, which defines the learning feedback signal
4. The optimizer determines how learning progresses.

We use keras packages to build the model. One prerequisite is that converting categorical variables to numerical variables. The first hidden layer has four neurons, and the second hidden layer has two neurons.



This model has 5 inputs, 2 hidden layers (i.e 4 neurons in one layer and other has 2 neurons). This is linked to the output neuron. We convert the dataset to a matrix format and divide it into two parts: train and test.

Divide the data into train and test, with train data accounting for 80% of the total and test data accounting for 20%. Create four data sets: training, testing, traintraject, and testtarget. The output variable, Loan Status, is included in the train and test targets. All variables other than the output variable are included in the training and testing datasets.

Values should be normalized. To the columns, apply the mean and standard deviation. Subtract the mean from the value and divide by the standard deviation to normalize independent variables. To normalize both the train and test sets, use the scale function.

Create the model with keras model sequential and pass the layer dense parameter (). This function creates the network's hidden layer. Using the same function, you can apply multiple hidden layers. The value for the hidden layer activation function is relu, which stands for rectified linear unit; the value for the activation function is linear by default. We need to define the input shape, and since we have 5 input variables, the shape is 5.

To compile the model, loss, optimizer, and metrics are used. There are other arguments we could pass, but we chose to pass three. The output has two values (1,0), and the input variables are all numerical variables.

loss = If the model has multiple outputs, you can apply a different loss to each one by passing a dictionary or a list of objectives. The sum of all individual losses will then be the loss value minimized by the model.

optimizer = optimizer instance; rmsprop is the default (lr stands for learning rate) metrics = A list of metrics that the model will evaluate during training and testing. We pass accuracy here.

Before fitting the training data, the model must be compiled (passing the training data to the model). The loss function, optimizer, and metrics are defined during this step.

To train model,

- training_keras = Vector, matrix, or array of training data
- trainingtarget = Vector, matrix, or array of target (label) data
- epochs = Number of epochs to train the model
- batch_size = Integer or NULL. Number of samples per gradient update. If unspecified, batch_size will default to 32.
- validation_split = Fraction of the training data to be used as validation data (between 0 and 1).

The first graph depicts loss, while the second depicts accuracy. There is initially very little difference between loss and val loss, which is very small. This indicates that the training error is low.

Lets have a look on model evaluation.

The model can be evaluated using the test feature and target sets. Using the evaluate() function, the results show the overall loss and mean absolute error. It accepts two arguments that refer to the feature and target test sets.

The evaluation provides the total loss and the accuracy of the training data. The confusion matrix and training data accuracy are shown in the table below.

Accuracy is 80%, model classify loan approve yes category well but shows poor performance for loan approve no category.

We have used 6 neurons and 2 hidden layer, there are scope for improvement. We will try with multiple neurons and layer dropout that helps to reduce over fitting. We can consider number of neurons in hidden layer to 9 after converting to numerical variables and consider the target variable as output layer. Using this approach, the accuracy can be further improved.

We apply the same code but with different neurons count and hidden layer. We would like to consider ADAM optimizer and Binary Cross Entropy Loss function. We assume number of epochs might be around 120.

Implementation 1

We have created the train and test sets and are ready to train the model and imported the required libraries .

STEPS TO CREATE A NEURAL NETWORK.

In this section, we will go over the steps required to build neural networks.

- The data is being loaded.
- Creating data for training and validation: We'll train with training data and validate with validation data.
- Defining our model's architecture: We will specify the number of input neurons, as well as the number of neurons in the hidden and output layers.
- Model compilation (loss function / optimizers): In this section, we will define our cost function, which will be used during backward propagation.
- Train the model: We'll train our model and specify the number of epochs.
- Model performance should be evaluated using training and validation data.

TRAINING & VALIDATION:

In this section, we will experiment with our training data. We will keep 10% of the training data for testing purposes and evaluate the model's accuracy on it. The model would be completely blind to this test data. We will use 20% of the remaining 90% of training data to validate our model and 70% of the data to train on. We will be defining the nos. of input, hidden and output neuron.

Defining input, hidden, and output neurons, as well as architecture. For the time being, I've chosen relu as an activation function for hidden layers. In addition, because this is a binary classification problem, I used a sigmoid activation function in the final layer.

COMPILING THE MODEL (LOSS FUNCTION / OPTIMIZER):

In this section, we define our loss function and optimizer, which will help us validate our data and perform backward propagation.

TRAINING THE MODEL:

In this section, we will train our model by providing our independent and dependent variables, as well as validation data and the number of epochs, which specifies how many times our model will go through the training set.

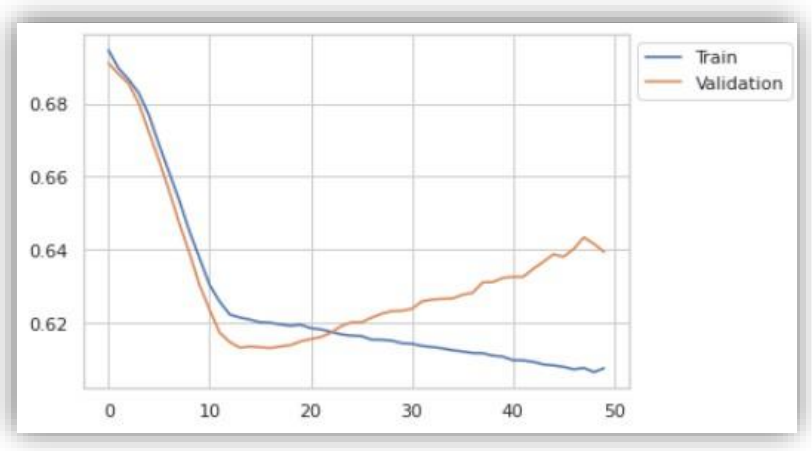
EVALUATING MODEL ON TEST DATA:

In this section, we will predict the classes for our data and use accuracy as a metric to evaluate model performance.

VISUALZING RESULTS (LOSS & ACCURACY): The variation in loss function and accuracy observed over a number of epochs is shown here.

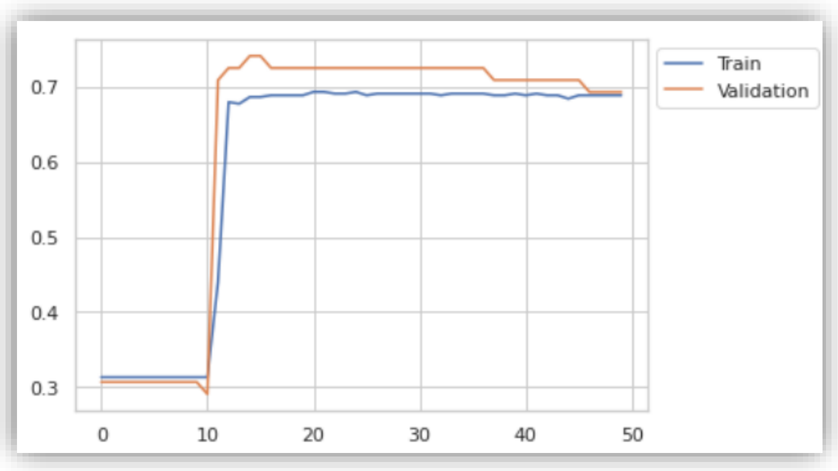
LOSS FUNCTION

The loss decreases as the number of epochs increases, as shown in the graph above. The validation loss does not appear to improve after a certain number of epochs.

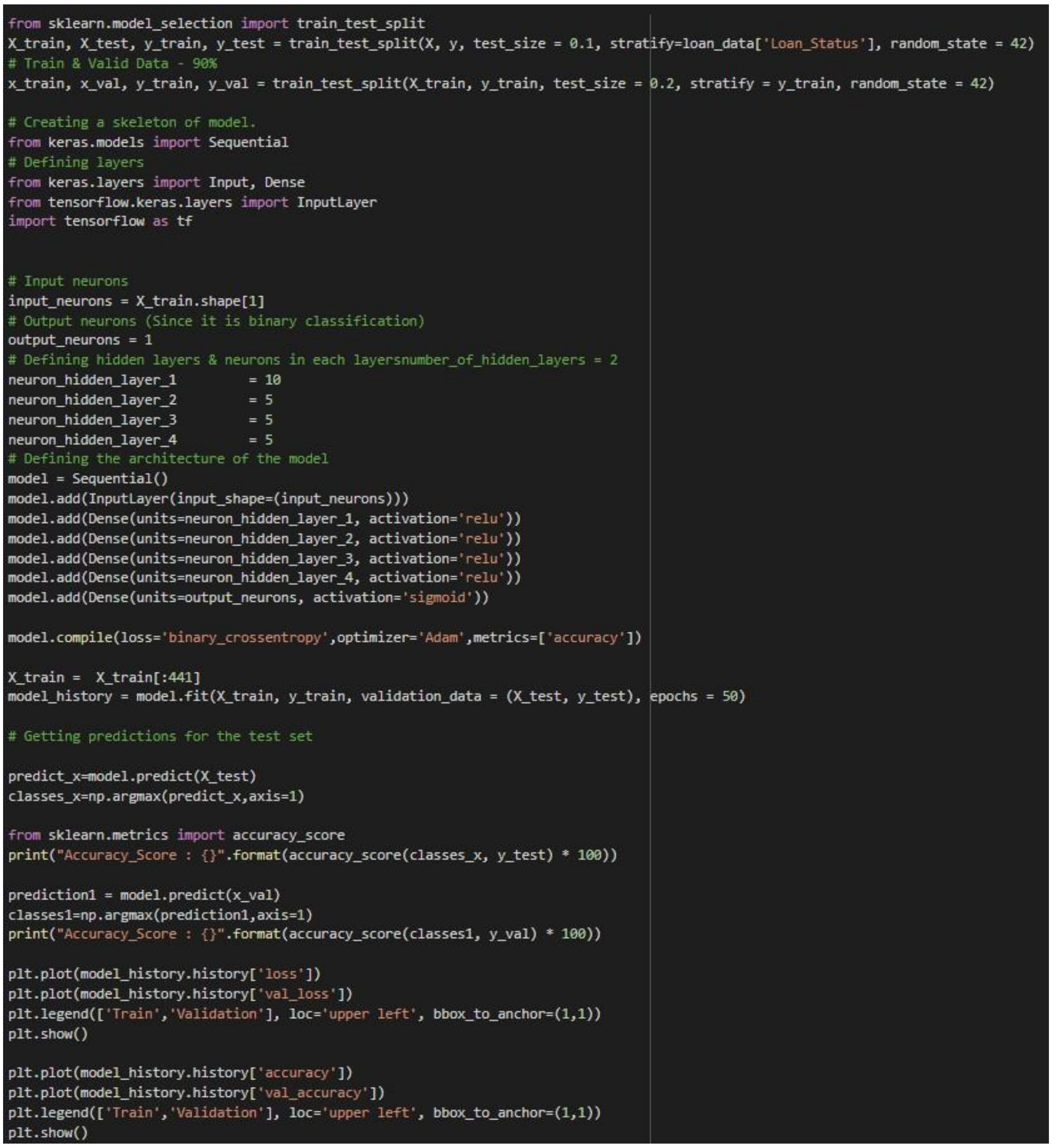


ACCURACY

As we can see, the accuracy for 5 to 10 epochs did not change significantly. Following that, the accuracy of both the training and validation sets began to improve, indicating that the model had begun to learn. There was a slight improvement from epoch 20 to 100, or the model stopped learning.

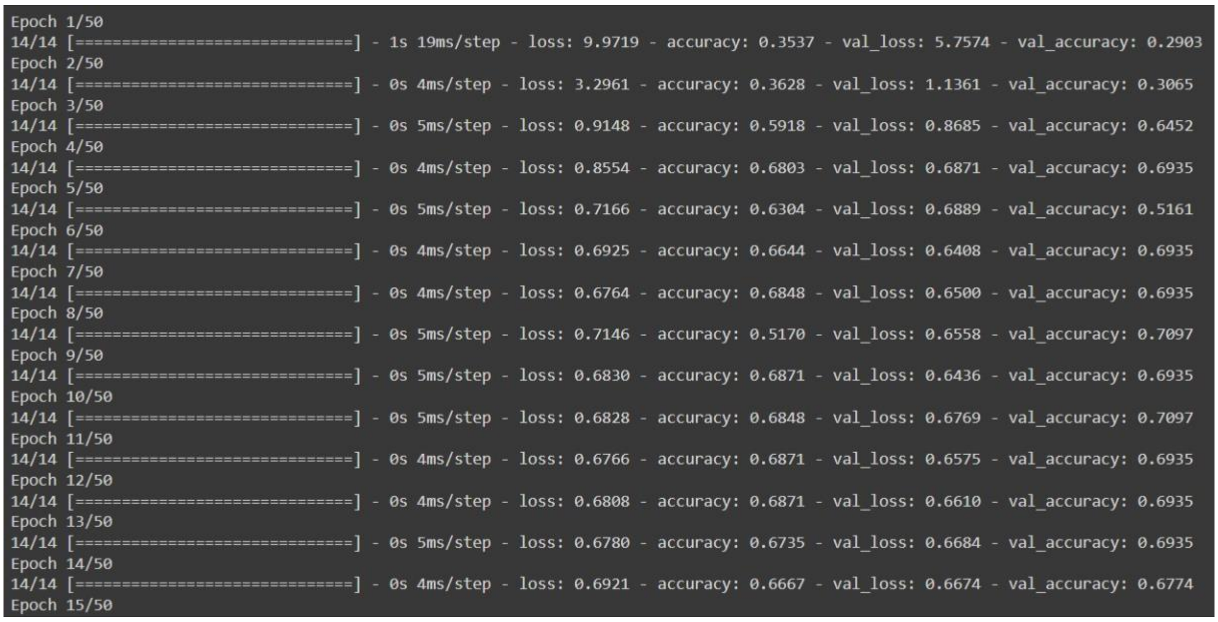


CODE:



```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1, stratify=loan_data['Loan_Status'], random_state = 42)
# Train & Valid Data - 90%
x_train, x_val, y_train, y_val = train_test_split(X_train, y_train, test_size = 0.2, stratify = y_train, random_state = 42)

# Creating a skeleton of model.
from keras.models import Sequential
# Defining layers
from keras.layers import Input, Dense
from tensorflow.keras.layers import InputLayer
import tensorflow as tf


# Input neurons
input_neurons = X_train.shape[1]
# Output neurons (Since it is binary classification)
output_neurons = 1
# Defining hidden layers & neurons in each layersnumber_of_hidden_layers = 2
neuron_hidden_layer_1         = 10
neuron_hidden_layer_2         = 5
neuron_hidden_layer_3         = 5
neuron_hidden_layer_4         = 5
# Defining the architecture of the model
model = Sequential()
model.add(InputLayer(input_shape=(input_neurons)))
model.add(Dense(units=neuron_hidden_layer_1, activation='relu'))
model.add(Dense(units=neuron_hidden_layer_2, activation='relu'))
model.add(Dense(units=neuron_hidden_layer_3, activation='relu'))
model.add(Dense(units=neuron_hidden_layer_4, activation='relu'))
model.add(Dense(units=output_neurons, activation='sigmoid'))

model.compile(loss='binary_crossentropy',optimizer='Adam',metrics=['accuracy'])

X_train =  X_train[:441]
model_history = model.fit(X_train, y_train, validation_data = (X_test, y_test), epochs = 50)

# Getting predictions for the test set

predict_x=model.predict(X_test)
classes_x=np.argmax(predict_x,axis=1)

from sklearn.metrics import accuracy_score
print("Accuracy_Score : {}".format(accuracy_score(classes_x, y_test) * 100))

prediction1 = model.predict(x_val)
classes1=np.argmax(prediction1,axis=1)
print("Accuracy_Score : {}".format(accuracy_score(classes1, y_val) * 100))

plt.plot(model_history.history['loss'])
plt.plot(model_history.history['val_loss'])
plt.legend(['Train','Validation'], loc='upper left', bbox_to_anchor=(1,1))
plt.show()

plt.plot(model_history.history['accuracy'])
plt.plot(model_history.history['val_accuracy'])
plt.legend(['Train','Validation'], loc='upper left', bbox_to_anchor=(1,1))
plt.show()
```

OUTPUT:



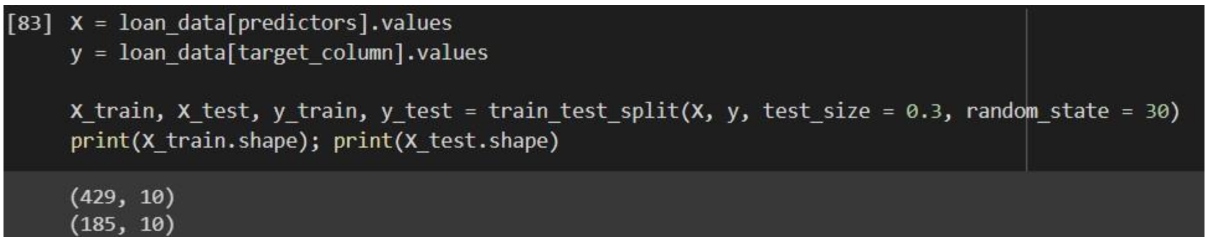
```
Epoch 1/50
14/14 [==============================] - 1s 19ms/step - loss: 9.9719 - accuracy: 0.3537 - val_loss: 5.7574 - val_accuracy: 0.2903
Epoch 2/50
14/14 [==============================] - 0s 4ms/step - loss: 3.2961 - accuracy: 0.3628 - val_loss: 1.1361 - val_accuracy: 0.3065
Epoch 3/50
14/14 [==============================] - 0s 5ms/step - loss: 0.9148 - accuracy: 0.5918 - val_loss: 0.8685 - val_accuracy: 0.6452
Epoch 4/50
14/14 [==============================] - 0s 4ms/step - loss: 0.8554 - accuracy: 0.6803 - val_loss: 0.6871 - val_accuracy: 0.6935
Epoch 5/50
14/14 [==============================] - 0s 5ms/step - loss: 0.7166 - accuracy: 0.6304 - val_loss: 0.6889 - val_accuracy: 0.5161
Epoch 6/50
14/14 [==============================] - 0s 4ms/step - loss: 0.6925 - accuracy: 0.6644 - val_loss: 0.6408 - val_accuracy: 0.6935
Epoch 7/50
14/14 [==============================] - 0s 4ms/step - loss: 0.6764 - accuracy: 0.6848 - val_loss: 0.6500 - val_accuracy: 0.6935
Epoch 8/50
14/14 [==============================] - 0s 5ms/step - loss: 0.7146 - accuracy: 0.5170 - val_loss: 0.6558 - val_accuracy: 0.7097
Epoch 9/50
14/14 [==============================] - 0s 5ms/step - loss: 0.6830 - accuracy: 0.6871 - val_loss: 0.6436 - val_accuracy: 0.6935
Epoch 10/50
14/14 [==============================] - 0s 5ms/step - loss: 0.6828 - accuracy: 0.6848 - val_loss: 0.6769 - val_accuracy: 0.7097
Epoch 11/50
14/14 [==============================] - 0s 4ms/step - loss: 0.6766 - accuracy: 0.6871 - val_loss: 0.6575 - val_accuracy: 0.6935
Epoch 12/50
14/14 [==============================] - 0s 4ms/step - loss: 0.6808 - accuracy: 0.6871 - val_loss: 0.6610 - val_accuracy: 0.6935
Epoch 13/50
14/14 [==============================] - 0s 5ms/step - loss: 0.6780 - accuracy: 0.6735 - val_loss: 0.6684 - val_accuracy: 0.6935
Epoch 14/50
14/14 [==============================] - 0s 4ms/step - loss: 0.6921 - accuracy: 0.6667 - val_loss: 0.6674 - val_accuracy: 0.6774
Epoch 15/50
```

NN Model	Loss Function	Optimizer	Hidden Layers	Output Neurons	Epochs	Activation Functions	Loss	Accuracy
Artificial Neural Network	**Binary Cross Entropy**	**Adam**	**2**	**1**	**100**	**Sigmoid, relu**	**< 0.6**	**~ 72**

Implementation 2 – SGD

We tried the NN implementation using another optimizer STOCHASTIC GRADIENT DESCENT. Even though we were not able to achieve accuracy as of Adam optimizer. This is great learning.

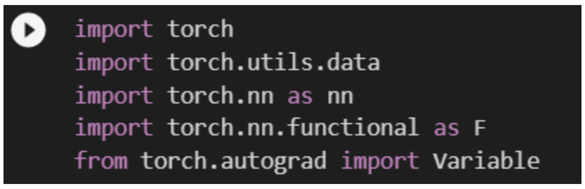
We created train and test datasets using the code below . The last line prints the training set and test set.



```
[83] X = loan_data[predictors].values
     y = loan_data[target_column].values

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 30)
     print(X_train.shape); print(X_test.shape)

     (429, 10)
     (185, 10)
```

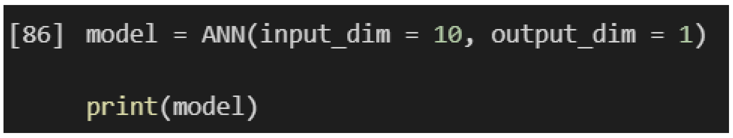
Model Building

We have created the train and test sets and are ready to train the model and imported the required libraries to work with Pytorch library.



```
import torch
import torch.utils.data
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
```

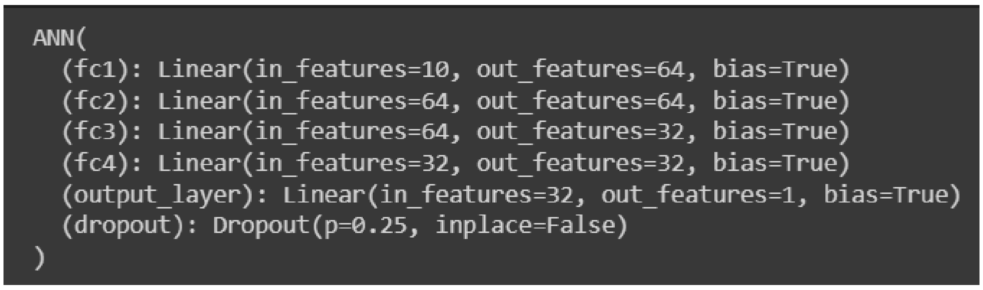
Refer the implementation 1 for ANN architecture implementation.



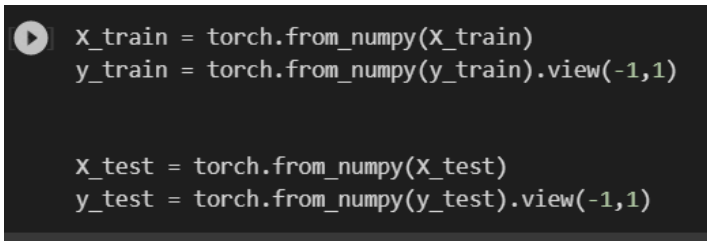
```
[86] model = ANN(input_dim = 10, output_dim = 1)

     print(model)
```

As we have defined the architecture of the model above ,instantiate the model using the code above.

Output:

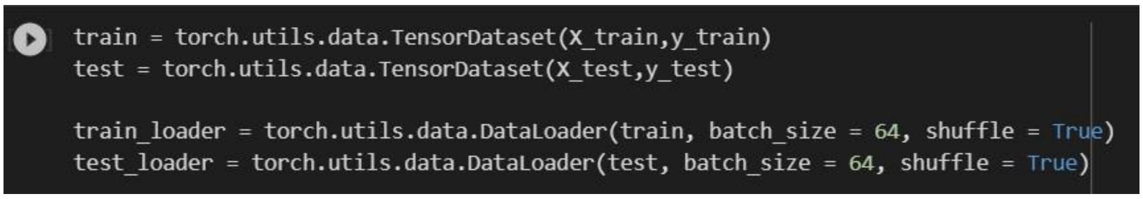


```
ANN(
  (fc1): Linear(in_features=10, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=32, bias=True)
  (fc4): Linear(in_features=32, out_features=32, bias=True)
  (output_layer): Linear(in_features=32, out_features=1, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
)
```



```
X_train = torch.from_numpy(X_train)
y_train = torch.from_numpy(y_train).view(-1,1)


X_test = torch.from_numpy(X_test)
y_test = torch.from_numpy(y_test).view(-1,1)
```

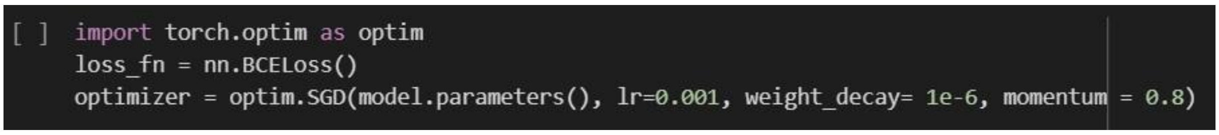
We carried out conversion on test and train arrays to make data formatted for Pytorch library.



```
train = torch.utils.data.TensorDataset(X_train,y_train)
test = torch.utils.data.TensorDataset(X_test,y_test)

train_loader = torch.utils.data.DataLoader(train, batch_size = 64, shuffle = True)
test_loader = torch.utils.data.DataLoader(test, batch_size = 64, shuffle = True)
```

We used **torch.utils** API provided by Pytorch to perform the iterations over the datasets to generate predictions and ANN is ready for predictive modelling.

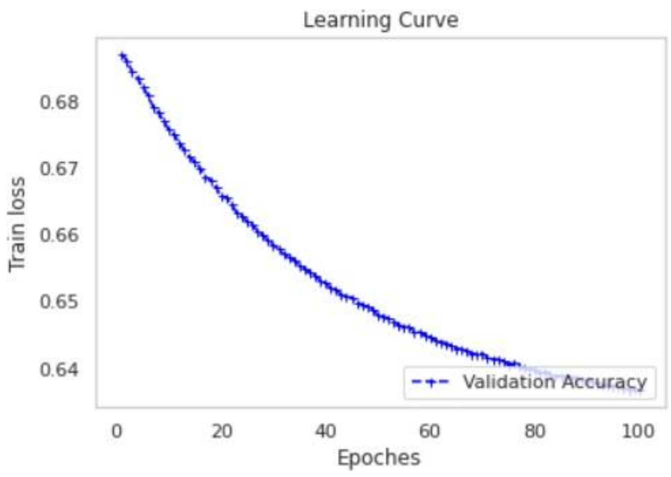
Model Evaluation



```
import torch.optim as optim
loss_fn = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, weight_decay= 1e-6, momentum = 0.8)
```

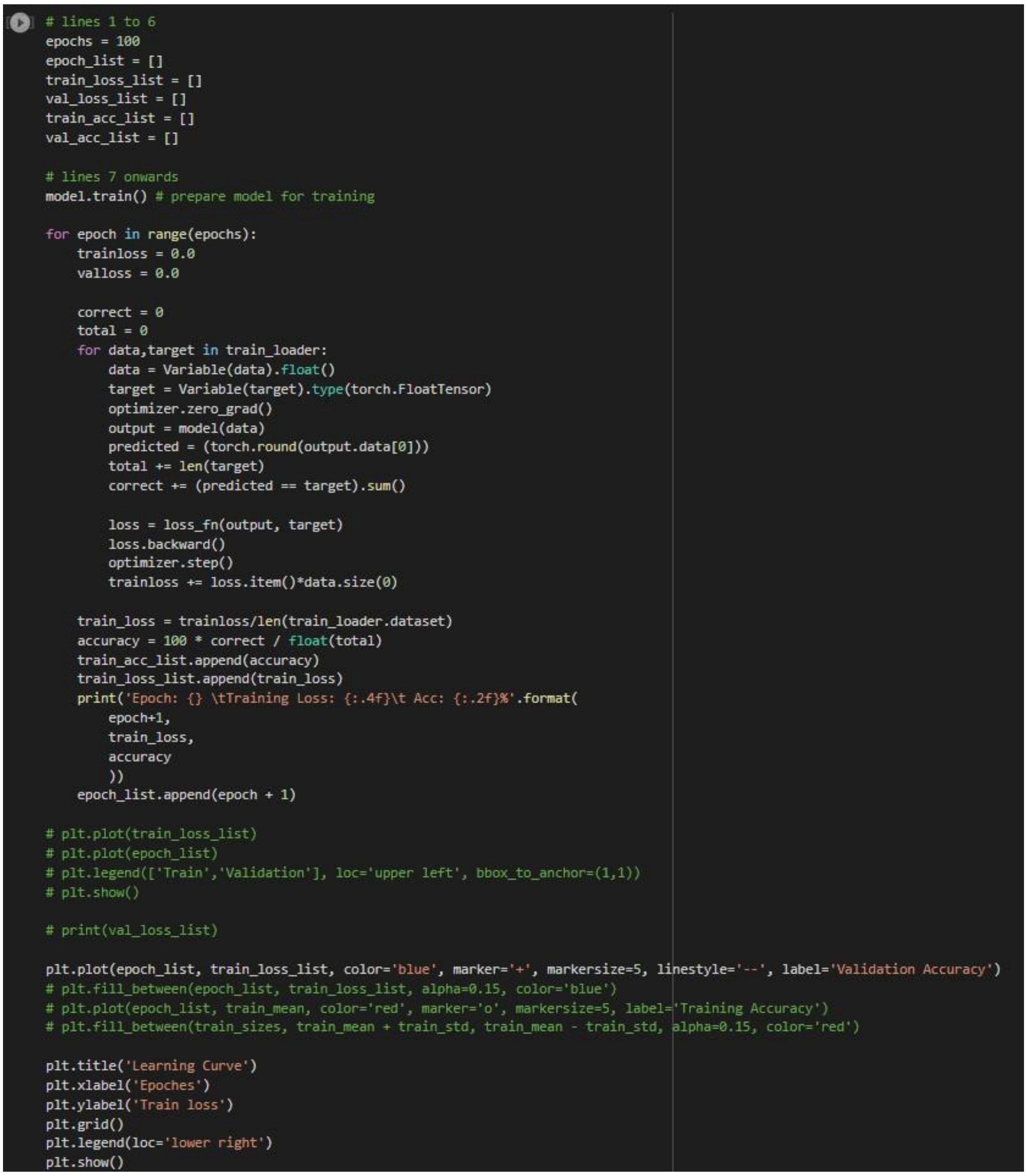
Now , we’ll start the model evaluation This is accomplished by computing loss, which measures the difference between the predicted and actual labels. In this case, use the nn.BCELoss() function to calculate Binary Cross-Entropy Loss. You must also use the stochastic gradient descent optimizer to optimize the network. The lines of code below are used to accomplish this. The optimizer function's learning rate is specified by the lr argument.

Following the definition of the loss function, the model is evaluated on the training data using the code below. In the first line of code, define the epoch, and lines two through six create lists



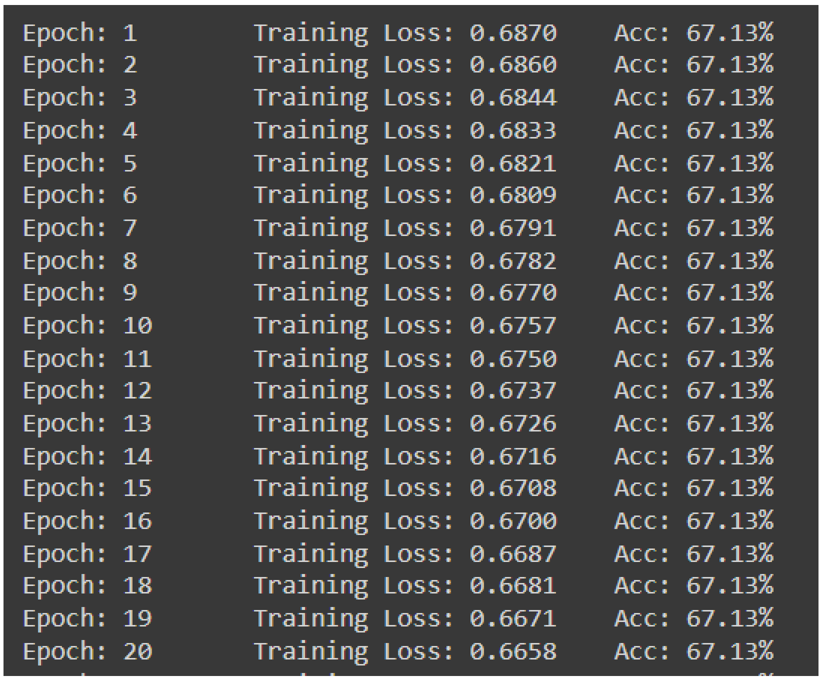
that will keep track of loss and accuracy during each epoch. Lines seven through ten are used to train the model, calculate loss and accuracy for each epoch, and finally print the output. You can observe from plot that as the epochs increases, the loss value decreases.

CODE:



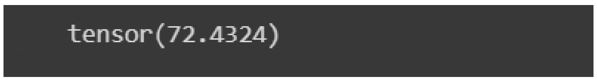
```python
# lines 1 to 6
epochs = 100
epoch_list = []
train_loss_list = []
val_loss_list = []
train_acc_list = []
val_acc_list = []

# lines 7 onwards
model.train() # prepare model for training

for epoch in range(epochs):
    trainloss = 0.0
    valloss = 0.0

    correct = 0
    total = 0
    for data,target in train_loader:
        data = Variable(data).float()
        target = Variable(target).type(torch.FloatTensor)
        optimizer.zero_grad()
        output = model(data)
        predicted = (torch.round(output.data[0]))
        total += len(target)
        correct += (predicted == target).sum()

        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
        trainloss += loss.item()*data.size(0)

    train_loss = trainloss/len(train_loader.dataset)
    accuracy = 100 * correct / float(total)
    train_acc_list.append(accuracy)
    train_loss_list.append(train_loss)
    print('Epoch: {} \tTraining Loss: {:.4f}\t Acc: {:.2f}%'.format(
        epoch+1,
        train_loss,
        accuracy
        ))
    epoch_list.append(epoch + 1)

# plt.plot(train_loss_list)
# plt.plot(epoch_list)
# plt.legend(['Train','Validation'], loc='upper left', bbox_to_anchor=(1,1))
# plt.show()

# print(val_loss_list)

plt.plot(epoch_list, train_loss_list, color='blue', marker='+', markersize=5, linestyle='--', label='Validation Accuracy')
# plt.fill_between(epoch_list, train_loss_list, alpha=0.15, color='blue')
# plt.plot(epoch_list, train_mean, color='red', marker='o', markersize=5, label='Training Accuracy')
# plt.fill_between(train_sizes, train_mean + train_std, train_mean - train_std, alpha=0.15, color='red')

plt.title('Learning Curve')
plt.xlabel('Epoches')
plt.ylabel('Train loss')
plt.grid()
plt.legend(loc='lower right')
plt.show()
```

OUTPUT:



```
Epoch: 1        Training Loss: 0.6870    Acc: 67.13%
Epoch: 2        Training Loss: 0.6860    Acc: 67.13%
Epoch: 3        Training Loss: 0.6844    Acc: 67.13%
Epoch: 4        Training Loss: 0.6833    Acc: 67.13%
Epoch: 5        Training Loss: 0.6821    Acc: 67.13%
Epoch: 6        Training Loss: 0.6809    Acc: 67.13%
Epoch: 7        Training Loss: 0.6791    Acc: 67.13%
Epoch: 8        Training Loss: 0.6782    Acc: 67.13%
Epoch: 9        Training Loss: 0.6770    Acc: 67.13%
Epoch: 10       Training Loss: 0.6757    Acc: 67.13%
Epoch: 11       Training Loss: 0.6750    Acc: 67.13%
Epoch: 12       Training Loss: 0.6737    Acc: 67.13%
Epoch: 13       Training Loss: 0.6726    Acc: 67.13%
Epoch: 14       Training Loss: 0.6716    Acc: 67.13%
Epoch: 15       Training Loss: 0.6708    Acc: 67.13%
Epoch: 16       Training Loss: 0.6700    Acc: 67.13%
Epoch: 17       Training Loss: 0.6687    Acc: 67.13%
Epoch: 18       Training Loss: 0.6681    Acc: 67.13%
Epoch: 19       Training Loss: 0.6671    Acc: 67.13%
Epoch: 20       Training Loss: 0.6658    Acc: 67.13%
```

ACCURACY:



```
tensor(72.4324)
```

NN Model	Loss Function	Optimizer	Hidden Layers	Output Neurons	Epochs	Activation Functions	Loss	Accuracy
Artificial Neural Network	Binary Cross Entropy	SGD	4	1	100	Sigmoid, relu	< 0.6	~ 67.13

CONCLUSION

We learned how to use ANN to solve a simple neural network on loan prediction problem. We can further improvise the model's performance by doing various hyper parameter tuning and observing it's accuracy on random data.