Final Project Update

Date: Dec- 11 -2024

Project Name: Loan Approval Prediction model utilizing Machine Learning

Student Name: Navitha Nelluri

Student Id: 008148524

Project Overview

The Loan Approval Prediction project focuses on developing a machine learning model that predicts loan approval outcomes based on historical data. The goal is to assist financial instructions in automating the loan approval process, ensuring a more efficient and data-driven decision-making system.

Data Collec1on

- The dataset for this project has been provided by Analytics Vidhya on their Hackathon page. It is divided into two sets: a training set and a test set.

  • The training set will be used to train the model, enabling it to learn patterns and relationships within the data. This set includes both the independent variables (features) and the dependent variable (target variable: Loan_Status).

  • The test set contains only the independent variables, without the target variable. The trained model will be applied to predict the target variable for this data.

Dataset Overview:

- **Training Set**: 13 columns (features) and 614 rows (records).
- **Test Set**: 12 columns (features, excluding Loan_Status) and 367 rows (records).

Exploratory Data Analysis (EDA):

We will utilize Python to explore the dataset, aiming to gain a comprehensive understanding of both the features and the target variable. Additionally, we will analyze the data to summarize its key characteristics, employing various visualization techniques to enhance insights.

Univariate analysis

Univariate analysis involves examining each variable individually. For categorical features, we can use frequency tables or bar plots to visualize the count of each category within a variable. For numerical features, histograms and box plots are useful tools to understand their distribution.

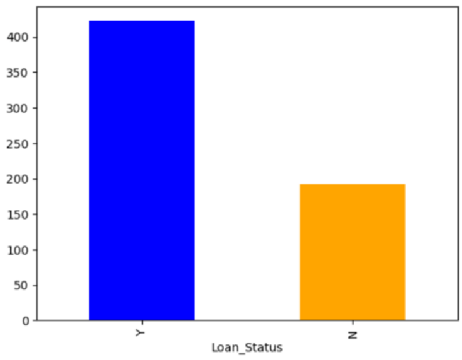
- Histograms allow us to observe characteristics such as central tendency, variability, modality, and kurtosis of a distribution.

- However, histograms are not effective for identifying outliers. To address this, box plots are also used, as they provide a clear visualization of potential outliers and the spread of the data.

Target Variable (Categorical)

We will begin by examining the target variable, **Loan_Status**, which is categorical. To analyze this variable, we will review its frequency table, calculate the percentage distribution, and visualize the data using a bar plot.

```
[23]: Loan_Status
      Y    422
      N    192
      Name: count, dtype: int64

[25]: # percentage distribution can be calculated by setting the normalize=True to show proportions instead of number
      train['Loan_Status'].value_counts(normalize=True)

[25]: Loan_Status
      Y    0.687296
      N    0.312704
      Name: proportion, dtype: float64

[27]: # bar plot to visualize the frequency
      train['Loan_Status'].value_counts().plot.bar(color=['blue', 'orange'])
      plt.show()
```

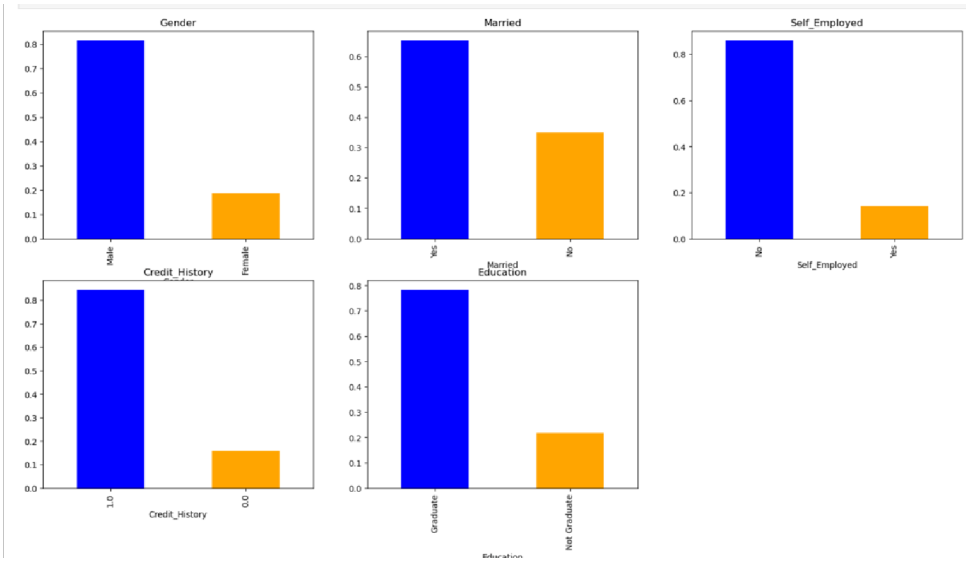


Note: Out of 614 applicants, 422 loans (approximately 69%) were approved. This indicates that there is no significant class imbalance in the dataset, making accuracy a suitable evaluation metric. However, in cases where the classes are imbalanced or skewed, evaluation metrics such as precision and recall would be more appropriate.

Independent Variable (Categorical):

The dataset contains five features that are either categorical or binary: Gender, Married, Self_Employed, Credit_History, and Education.


```
[29]: # Visualizing categorical features with custom colors
      plt.subplot(231)
      train['Gender'].value_counts(normalize=True).plot.bar(figsize=(20,10), title='Gender', color=['blue', 'orange'])

      plt.subplot(232)
      train['Married'].value_counts(normalize=True).plot.bar(title='Married', color=['blue', 'orange'])

      plt.subplot(233)
      train['Self_Employed'].value_counts(normalize=True).plot.bar(title='Self_Employed', color=['blue', 'orange'])

      plt.subplot(234)
      train['Credit_History'].value_counts(normalize=True).plot.bar(title='Credit_History', color=['blue', 'orange'])

      plt.subplot(235)
      train['Education'].value_counts(normalize=True).plot.bar(title='Education', color=['blue', 'orange'])

      plt.show()
```



The following insights can be drawn from the bar plots:

- Approximately **80%** of the applicants in the dataset are male.
- Around **65%** of the applicants are married.
- About **15%** of the applicants are self-employed.
- Nearly **85%** of the applicants have a credit history, indicating they have repaid their debts.
- Around **80%** of the applicants are graduates.

Independent Variable (Ordinal):

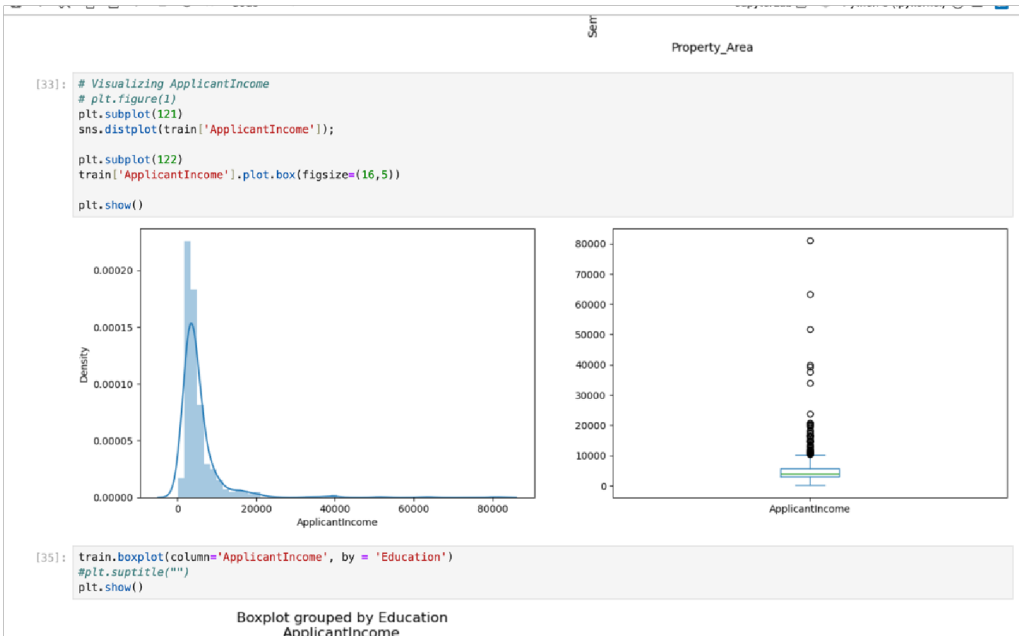
here are 2 features that are Ordinal: Variables in categorical features having some order involved (Dependents, Property_Area)

Following inferences can be made from the above bar plots:

- More than half of the applicants don’t have any dependents.
- Most of the applicants are from Semiurban area.

Independent Variable (Numerical)

There are 4 features that are Numerical: These features have numerical values (ApplicantIncome, CoapplicantIncome, LoanAmount, Loan_Amount_Term)
Firstly, let’s look at the Applicant income distribution:

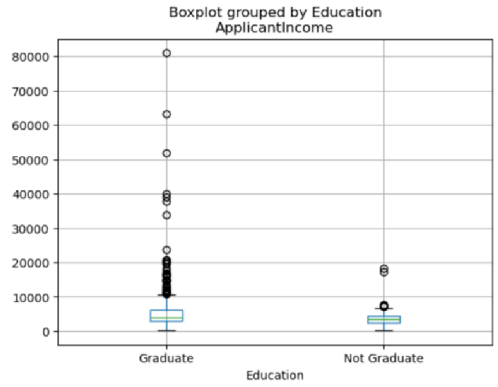


It can be observed that the distribution of applicant income is left-skewed, indicating that it is not normally distributed but rather right-skewed (positive skewness). We will attempt to normalize it in later sections, as algorithms generally perform better with normally distributed data.

The boxplot further reveals the presence of numerous outliers and extreme values, likely due to income disparity in society. This could also be influenced by the varying education levels of the applicants. To explore this further, let's segregate the data based on education level.

ApplicantIncome

[35]:
```
train.boxplot(column='ApplicantIncome', by = 'Education')
#plt.suptitle("")
plt.show()
```

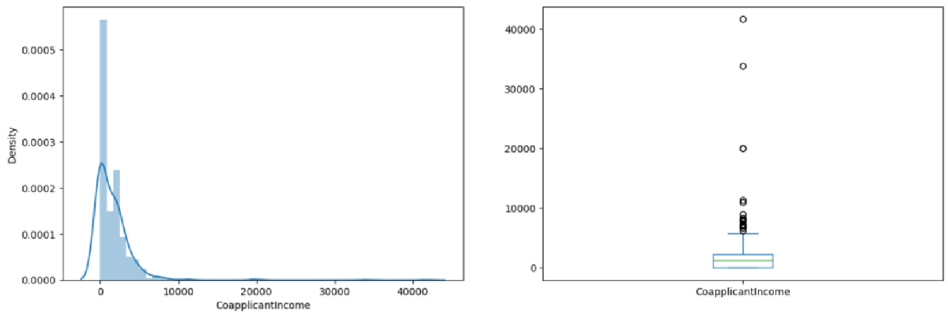


[37]:
```
# plt.figure(1)
plt.subplot(121)
sns.distplot(train['CoapplicantIncome']);

plt.subplot(122)
train['CoapplicantIncome'].plot.box(figsize=(16,5))

plt.show()
```

- We can see that there are a higher number of graduates with very high incomes, which are appearing to be the outliers.

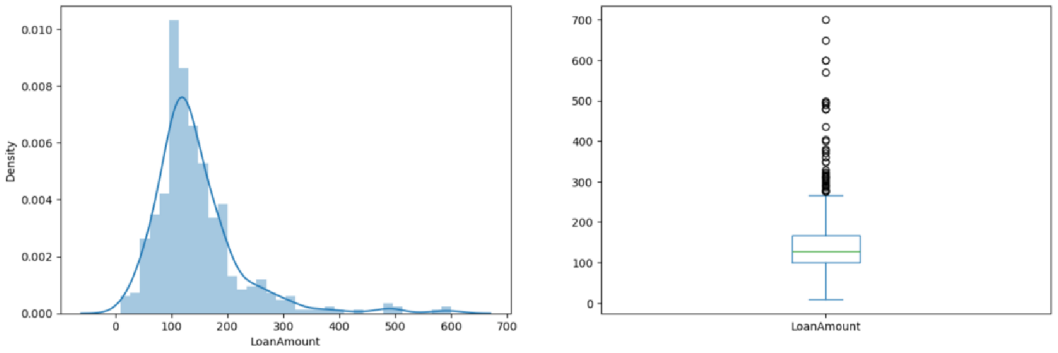
- Secondly, Let's look at the Coapplicant income distribution.

[37]:
```
# plt.figure(1)
plt.subplot(121)
sns.distplot(train['CoapplicantIncome']);

plt.subplot(122)
train['CoapplicantIncome'].plot.box(figsize=(16,5))

plt.show()
```



[39]:
```
# plt.figure(1)
plt.subplot(121)
df=train.dropna()
sns.distplot(df['LoanAmount']);
```


```
[39]: # plt.figure(1)
plt.subplot(121)
df=train.dropna()
sns.distplot(df['LoanAmount']);

plt.subplot(122)
train['LoanAmount'].plot.box(figsize=(16,5))

plt.show()
```




```
[41]: # frequency table of a variable will give us the count of each category in that variable
train['Loan_Amount_Term'].value_counts()
```

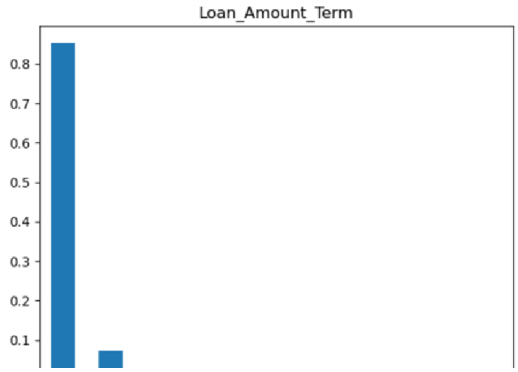
Bivariate Analysis:

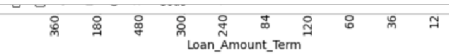
- After conducting the univariate analysis for each variable, we will now explore the relationships between the variables and the target variable through bivariate analysis. This will help us test the hypotheses generated earlier.
- Categorical Independent Variable vs Target Variable
- First, we will examine the relationship between the target variable and categorical independent variables. A stacked bar plot will be used to visualize the proportion of approved and unapproved loans for each category. For instance, we will investigate whether an applicant's gender influences the chances of loan approval.

[41]:
```
# frequency table of a variable will give us the count of each category in that variable
train['Loan_Amount_Term'].value_counts()
```

[41]:
```
Loan_Amount_Term
360.0    512
180.0     44
480.0     15
300.0     13
240.0      4
84.0       4
120.0      3
60.0       2
36.0       2
12.0       1
Name: count, dtype: int64
```

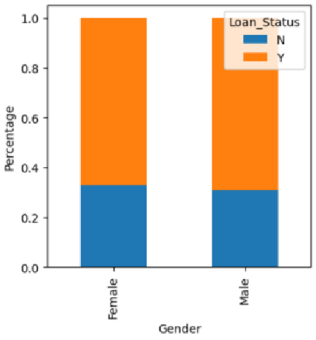
[43]:
```
# plot bar chart
train['Loan_Amount_Term'].value_counts(normalize=True).plot.bar(title= 'Loan_Amount_Term')
plt.show()
```





[45]:
```
print(pd.crosstab(train['Gender'],train['Loan_Status']))

Gender = pd.crosstab(train['Gender'],train['Loan_Status'])
Gender.div(Gender.sum(1).astype(float), axis = 0).plot(kind="bar", stacked=True, figsize=(4,4))
plt.xlabel('Gender')
p = plt.ylabel('Percentage')
plt.show()
```

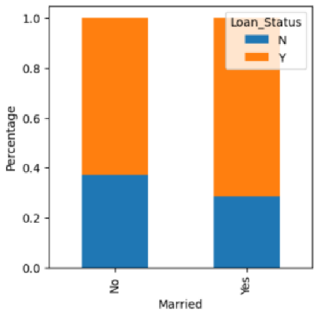
```
Loan_Status    N    Y
Gender
Female        37   75
Male         150  339
```



[47]:
```
print(pd.crosstab(train['Married'],train['Loan_Status']))
```

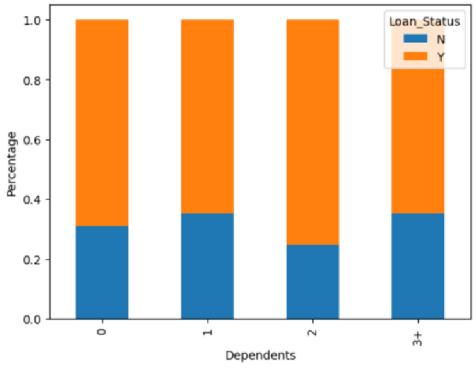
[47]:
```
print(pd.crosstab(train['Married'],train['Loan_Status']))

Married = pd.crosstab(train['Married'],train['Loan_Status'])
Married.div(Married.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))
plt.xlabel('Married')
plt.ylabel('Percentage')
plt.show()
```

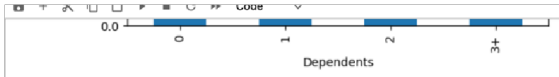
```
Loan_Status    N    Y
Married
No            79  134
Yes          113  285
```



[49]:
```
print(pd.crosstab(train['Dependents'],train['Loan_Status']))

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

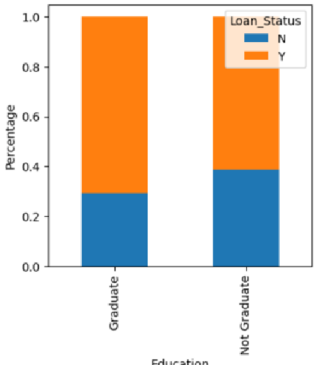
```
Loan_Status    N    Y
Dependents
0            107  238
1             36   66
2             25   76
3+            18   33
```





[51]:
```
print(pd.crosstab(train['Education'],train['Loan_Status']))

Education=pd.crosstab(train['Education'],train['Loan_Status'])
Education.div(Education.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))
plt.xlabel('Education')
plt.ylabel('Percentage')
plt.show()
```

```
Loan_Status     N    Y
Education
Graduate      140  340
Not Graduate   52   82
```



Code JupyterLab Python 3 (ipykernel)

[55]:
```
print(pd.crosstab(train['Credit_History'],train['Loan_Status']))

Credit_History=pd.crosstab(train['Credit_History'],train['Loan_Status'])
Credit_History.div(Credit_History.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))
plt.xlabel('Credit_History')
p = plt.ylabel('Percentage')
plt.show()
```

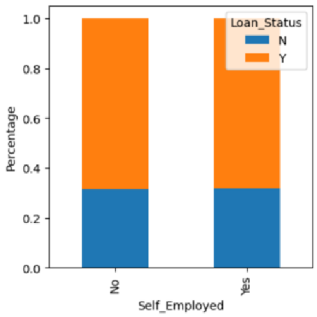
```
Loan_Status      N    Y
Credit_History
0.0             82    7
1.0             97  378
```

[57]:
```
print(pd.crosstab(train['Property_Area'],train['Loan_Status']))
```

Education

[53]:
```
print(pd.crosstab(train['Self_Employed'],train['Loan_Status']))

Self_Employed=pd.crosstab(train['Self_Employed'],train['Loan_Status'])
Self_Employed.div(Self_Employed.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))
plt.xlabel('Self_Employed')
plt.ylabel('Percentage')
plt.show()
```

```
Loan_Status      N    Y
Self_Employed
No             157  343
Yes             26   56
```

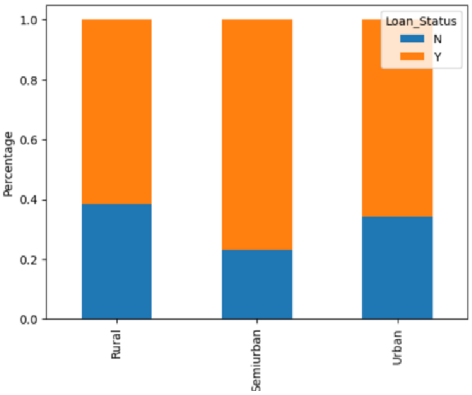


[55]:
```
print(pd.crosstab(train['Credit_History'],train['Loan_Status']))

Credit_History=pd.crosstab(train['Credit_History'],train['Loan_Status'])
Credit_History.div(Credit_History.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True, figsize=(4,4))
plt.xlabel('Credit_History')
```

Credit_History

[57]:
```
print(pd.crosstab(train['Property_Area'],train['Loan_Status']))

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

```
Loan_Status     N    Y
Property_Area
Rural          69  110
Semiurban      54  179
Urban          69  133
```



From the bar charts above, the following insights can be inferred:

- The proportion of male and female applicants is nearly the same for both approved and unapproved loans.
- Married applicants have a higher proportion of loan approvals.
- The distribution of applicants with 1 or 3+ dependents is similar across both loan approval categories.
- No significant insights can be drawn from the Self_Employed vs Loan_Status plot.
- Graduates have a higher proportion of loan approvals compared to non-graduates.
- Applicants with a credit history of 1 are more likely to have their loans approved.

- Loans in semi-urban areas are more likely to be approved compared to those in rural or urban areas.

Numerical Independent Variable vs Target Variable

Next, we will compare the mean income of applicants whose loans were approved versus those whose loans were not approved.

- Here, the y-axis represents the mean applicant income. There is no significant difference in the mean income between applicants with approved loans (5384) and those with unapproved loans (5446).
- To gain more insight, we will create bins for the **ApplicantIncome** variable based on its values and analyze the corresponding loan status for each income bin.

Loan_Status

[61]:
```
# making bins for applicant income variable
bins = [0,2500,4000,6000,81000]
group = ['Low','Average','High', 'Very high']
train['Income_bin'] = pd.cut(df['ApplicantIncome'],bins,labels=group)
```

[63]:
```
# take a look at the train set
train.head(8)
```

[63]:

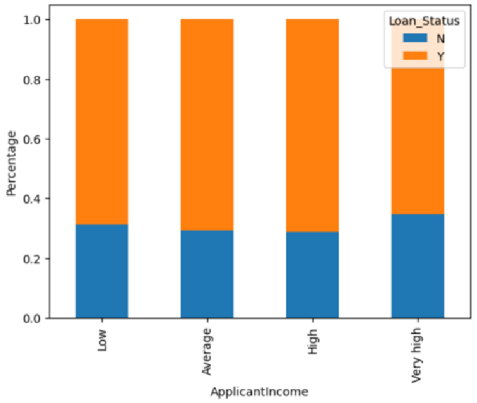
	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	F
0	LP001002	Male	No	0	Graduate	No	5849	0.0	NaN	360.0	1.0	
1	LP001003	Male	Yes	1	Graduate	No	4583	1508.0	128.0	360.0	1.0	
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	360.0	1.0	
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	360.0	1.0	
4	LP001008	Male	No	0	Graduate	No	6000	0.0	141.0	360.0	1.0	
5	LP001011	Male	Yes	2	Graduate	Yes	5417	4196.0	267.0	360.0	1.0	
6	LP001013	Male	Yes	0	Not Graduate	No	2333	1516.0	95.0	360.0	1.0	
7	LP001014	Male	Yes	3+	Graduate	No	3036	2504.0	158.0	360.0	0.0	

[65]:
```
print(pd.crosstab(train['Income_bin'],train['Loan_Status']))

Income_bin = pd.crosstab(train['Income_bin'],train['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')
plt.show()
```

```
Loan_Status   N    Y
Income_bin
Low          26   57
Average      51  123
High         32   79
Very high    39   73
```

[65]:
```
print(pd.crosstab(train['Income_bin'],train['Loan_Status']))

Income_bin = pd.crosstab(train['Income_bin'],train['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')
plt.show()
```

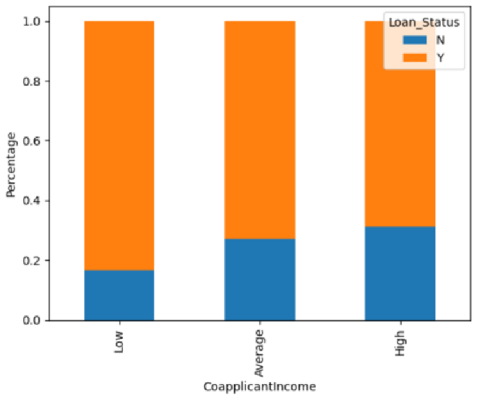
```
Loan_Status   N    Y
Income_bin
Low          26   57
Average      51  123
High         32   79
Very high    39   73
```



ApplicantIncome

```
[67]: # making bins for Coapplicant income variable
bins = [0,1000,3000,42000]
group = ['Low','Average','High']
train['Coapplicant_Income_bin'] = pd.cut(df['CoapplicantIncome'],bins,labels=group)
```

```
[69]: # plot the chart
Coapplicant_Income_bin = pd.crosstab(train['Coapplicant_Income_bin'],train['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')
plt.show()
```



Code JupyterLab Python 3 (ipykernel)

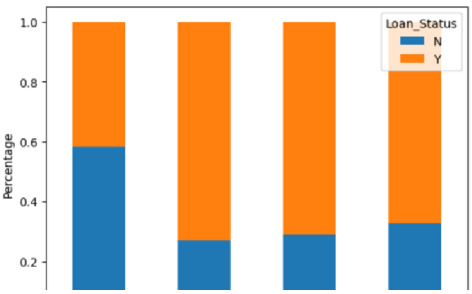
```
[71]: print(len(train[train["CoapplicantIncome"] == 0]))
"Percentage of CoapplicantIncome = 0 is:",  len(train[train["CoapplicantIncome"] == 0])/len(train["CoapplicantIncome"])
```

```
273
[71]: ('Percentage of CoapplicantIncome = 0 is:', 0.4446540716612375)
```

```
[73]: # create a new variable
train['Total_Income'] = train['ApplicantIncome'] + train['CoapplicantIncome']
```

```
[75]: # making bins for Total Income variable
bins = [0,2500,4000,6000,81000]
group = ['Low','Average','High', 'Very high']
train['Total_Income_bin'] = pd.cut(train['Total_Income'],bins,labels=group)
```

```
[77]: # plot the chart
Total_Income_bin = pd.crosstab(train['Total_Income_bin'],train['Loan_Status'])
Total_Income_bin.div(Total_Income_bin.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True)
plt.xlabel('Total_Income')
P = plt.ylabel('Percentage')
plt.show()
```

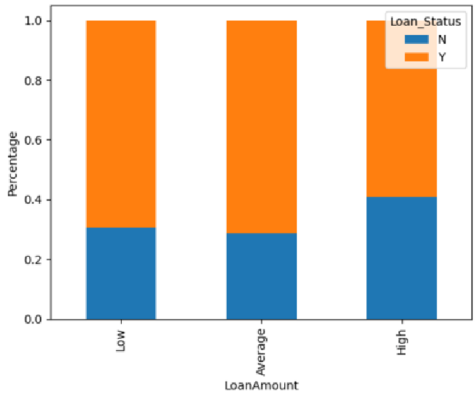


- We observe that the proportion of loans approved for applicants with low **Total_Income** is significantly lower compared to those with average, high, or very high income. This aligns more closely with our hypothesis that applicants with higher income are more likely to have their loans approved.
- Next, let's visualize the **LoanAmount** variable.

Code JupyterLab Python 3 (ipykernel)

[79]:
```
# making bins for LoanAmount variable
bins = [0,100,200,700]
group = ['Low','Average','High']
train['LoanAmount_bin'] = pd.cut(df['LoanAmount'],bins,labels=group)
```

[81]:
```
# plot the chart
LoanAmount_bin = pd.crosstab(train['LoanAmount_bin'],train['Loan_Status'])
LoanAmount_bin.div(LoanAmount_bin.sum(1).astype(float), axis=0).plot(kind="bar", stacked=True)
plt.xlabel('LoanAmount')
P = plt.ylabel('Percentage')
plt.show()
```



[83]:
```
# before dropping
train.head()
```

[83]:

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
0	LP001002	Male	No	0	Graduate	No	5849	0.0	NaN	360.0	1.0
1	LP001003	Male	Yes	1	Graduate	No	4583	1508.0	128.0	360.0	1.0
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	360.0	1.0
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	360.0	1.0
4	LP001008	Male	No	0	Graduate	No	6000	0.0	141.0	360.0	1.0

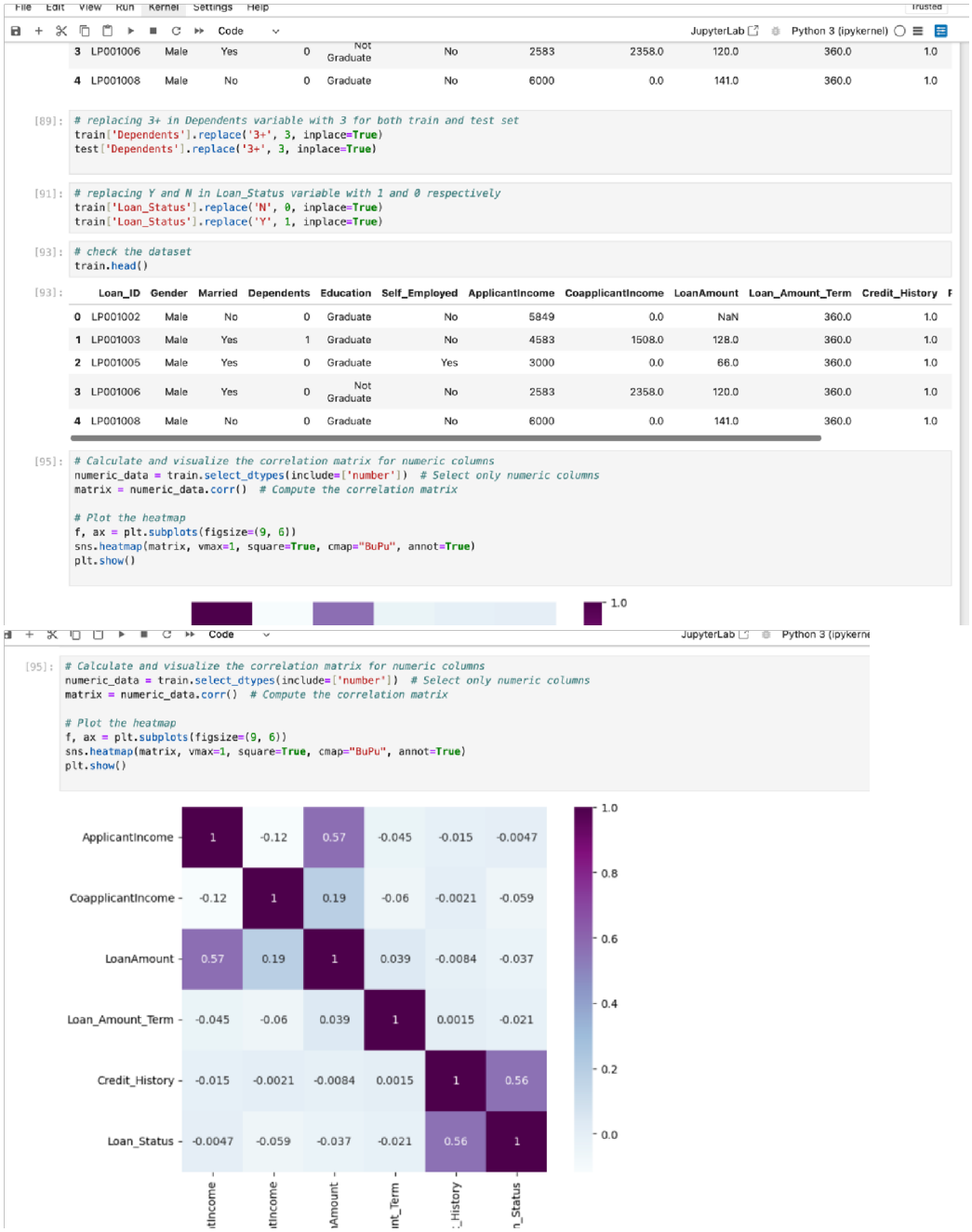
[85]:
```
# drop the new variable of bins
train = train.drop(['Income_bin', 'Coapplicant_Income_bin', 'LoanAmount_bin', 'Total_Income_bin', 'Total_Income'], axis=1)
```

[87]:
```
# after dropping
train.head()
```

[87]:

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
0	LP001002	Male	No	0	Graduate	No	5849	0.0	NaN	360.0	1.0
1	LP001003	Male	Yes	1	Graduate	No	4583	1508.0	128.0	360.0	1.0
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	360.0	1.0
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	360.0	
4	LP001008	Male	No	0	Graduate	No	6000	0.0	141.0	360.0	

[89]:
```
# replacing 3+ in Dependents variable with 3 for both train and test set
train['Dependents'].replace('3+', 3, inplace=True)
test['Dependents'].replace('3+', 3, inplace=True)
```



Note: The most correlated variables are:

- ApplicantIncome and LoanAmount, with a correlation coefficient of 0.57.
- Credit_History and Loan_Status, with a correlation coefficient of 0.56.
- LoanAmount and CoapplicantIncome, with a correlation coefficient of 0.19.

Data Pre-processing:

Data pre-processing is a crucial step in data mining that involves transforming raw data into a more understandable and usable format. Real-world data often suffers from issues such as incompleteness,

inconsistency, and errors, which can negatively impact model performance. Data pre-processing helps address these issues.

Missing Value and Outlier Treatment:

Acer thoroughly exploring the variables in our dataset, we can now address the missing values and outliers, as both can adversely affect model performance.

Missing Value Imputa1on:

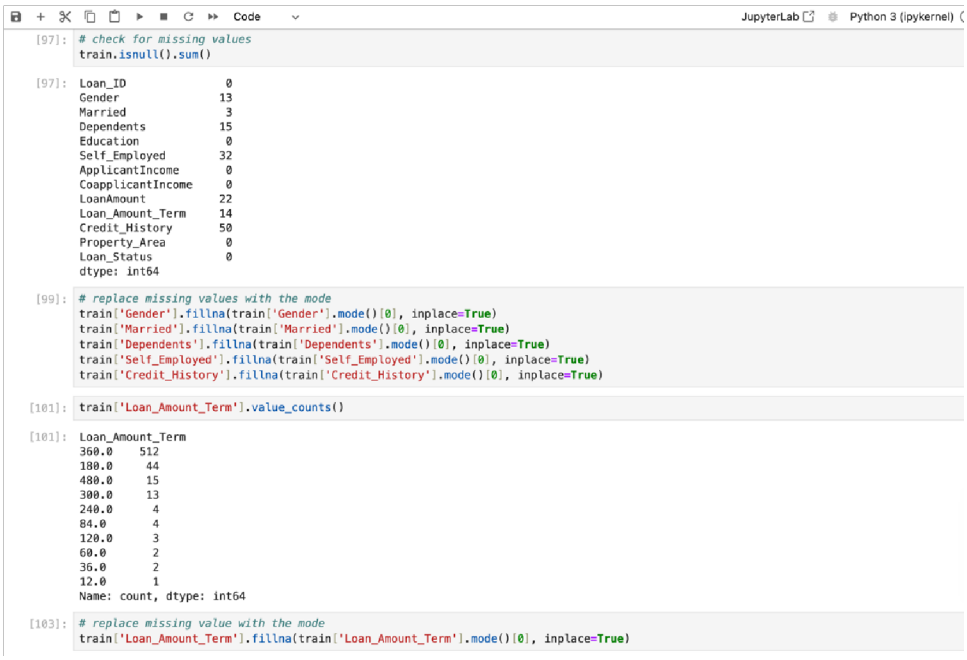
Let's begin by listing the feature-wise count of missing values.

There are missing values in Gender, Married, Dependents, Self_Employed, LoanAmount, Loan_Amount_Term and Credit_History features. We will treat the missing values in all the features one by one.

We can consider these methods to fill the missing values:

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

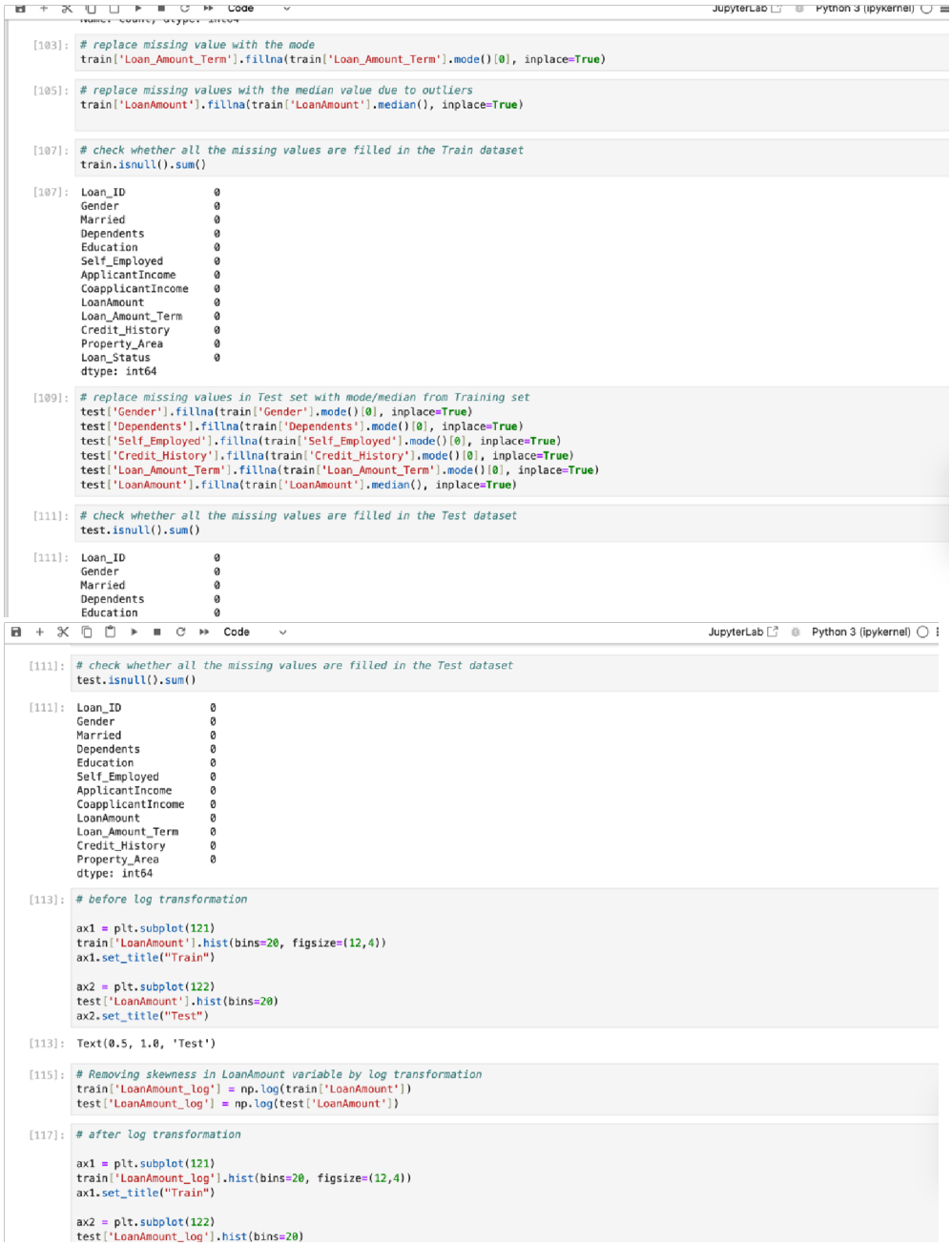
There are very less missing values in Gender, Married, Dependents, Credit_History and Self_Employed features so we can fill them using the mode of the features. If an independent variable in our dataset has huge amount of missing data e.g. 80% missing values in it, then we would drop the variable from the dataset.



```
[97]: # check for missing values
      train.isnull().sum()

[97]: Loan_ID              0
      Gender              13
      Married              3
      Dependents          15
      Education            0
      Self_Employed       32
      ApplicantIncome      0
      CoapplicantIncome    0
      LoanAmount          22
      Loan_Amount_Term    14
      Credit_History      50
      Property_Area        0
      Loan_Status          0
      dtype: int64

[99]: # replace missing values with the mode
      train['Gender'].fillna(train['Gender'].mode()[0], inplace=True)
      train['Married'].fillna(train['Married'].mode()[0], inplace=True)
      train['Dependents'].fillna(train['Dependents'].mode()[0], inplace=True)
      train['Self_Employed'].fillna(train['Self_Employed'].mode()[0], inplace=True)
      train['Credit_History'].fillna(train['Credit_History'].mode()[0], inplace=True)

[101]: train['Loan_Amount_Term'].value_counts()

[101]: Loan_Amount_Term
       360.0    512
       180.0     44
       480.0     15
       300.0     13
       240.0      4
       84.0       4
       120.0      3
       60.0       2
       36.0       2
       12.0       1
       Name: count, dtype: int64

[103]: # replace missing value with the mode
       train['Loan_Amount_Term'].fillna(train['Loan_Amount_Term'].mode()[0], inplace=True)
```

As we can see that all the missing values have been filled in the Train dataset. Let’s fill all the missing values in the test dataset too with the same approach.

**Note: ** We need to replace the missing values in Test set using the mode/median/mean of the Training set, not from the Test set. Likewise, if you remove values above some threshold in the test case, make sure that the threshold is derived from the training and not test set. Make sure to calculate the mean (or any other metrics) only on the train data to avoid data leakage to your test set.



```
[103]: # replace missing value with the mode
train['Loan_Amount_Term'].fillna(train['Loan_Amount_Term'].mode()[0], inplace=True)

[105]: # replace missing values with the median value due to outliers
train['LoanAmount'].fillna(train['LoanAmount'].median(), inplace=True)

[107]: # check whether all the missing values are filled in the Train dataset
train.isnull().sum()

[107]: Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
Loan_Status          0
dtype: int64

[109]: # replace missing values in Test set with mode/median from Training set
test['Gender'].fillna(train['Gender'].mode()[0], inplace=True)
test['Dependents'].fillna(train['Dependents'].mode()[0], inplace=True)
test['Self_Employed'].fillna(train['Self_Employed'].mode()[0], inplace=True)
test['Credit_History'].fillna(train['Credit_History'].mode()[0], inplace=True)
test['Loan_Amount_Term'].fillna(train['Loan_Amount_Term'].mode()[0], inplace=True)
test['LoanAmount'].fillna(train['LoanAmount'].median(), inplace=True)

[111]: # check whether all the missing values are filled in the Test dataset
test.isnull().sum()

[111]: Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
```



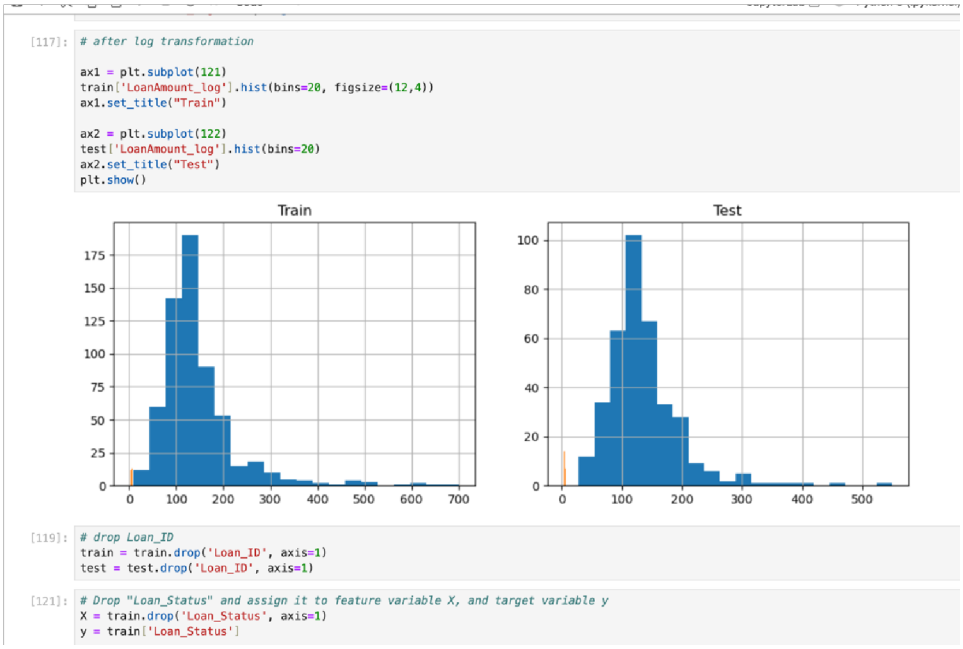
```
[111]: # check whether all the missing values are filled in the Test dataset
test.isnull().sum()

[111]: Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
dtype: int64

[113]: # before log transformation

ax1 = plt.subplot(121)
train['LoanAmount'].hist(bins=20, figsize=(12,4))
ax1.set_title("Train")

ax2 = plt.subplot(122)
test['LoanAmount'].hist(bins=20)
ax2.set_title("Test")

[113]: Text(0.5, 1.0, 'Test')

[115]: # Removing skewness in LoanAmount variable by log transformation
train['LoanAmount_log'] = np.log(train['LoanAmount'])
test['LoanAmount_log'] = np.log(test['LoanAmount'])

[117]: # after log transformation

ax1 = plt.subplot(121)
train['LoanAmount_log'].hist(bins=20, figsize=(12,4))
ax1.set_title("Train")

ax2 = plt.subplot(122)
test['LoanAmount_log'].hist(bins=20)
```

Outlier Treatment:

- As observed during the univariate analysis, LoanAmount contains outliers that need to be addressed, as their presence affects the distribution of the data. Outliers can significantly impact

the mean and standard deviation, thus distorting the data distribution. Therefore, it is crucial to remove or mitigate outliers in the dataset.

- The outliers in the LoanAmount feature cause most of the data to be concentrated on the left, with a longer right tail. This is referred to as right skewness (or positive skewness). One effective method to reduce skewness is log transformation. Log transformation has a minimal effect on smaller values but reduces the influence of larger values, thereby bringing the distribution closer to a normal distribution.
- Let's visualize the effect of log transformation. The same transformations will be applied to the test data simultaneously.



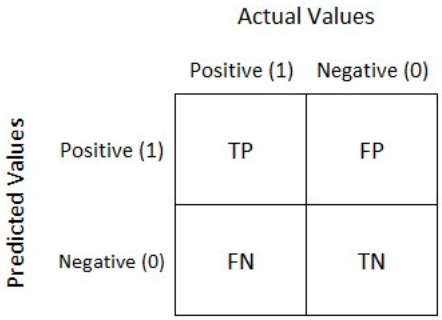
Evaluation Metrics for Classification Problems:

Model evaluation is an essential part of the model-building process. Once we have predictions from the model, it's important to assess how accurate these predictions are. We can compare the predicted values to the actual values, calculating the distance between them. The smaller this distance, the more accurate the predictions will be.

Since this is a **classification problem**, we can evaluate our models using various metrics. One of the most common metrics is:

Accuracy

Accuracy can be understood through the **confusion matrix**, which is a tabular representation that compares the actual values with the predicted values. Here's what a typical confusion matrix looks like:



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

Accuracy = (TP+TN) / (TP+TN+FP+FN)

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)

Recall (Sensitivity) - It is a measure of actual observations which are predicted correctly i.e. how many observations of true class are labeled correctly. It is also known as 'Sensitivity'. E.g. Proportion of patients with a disease who test positive.

Recall = TP / (TP + FN)

Specificity - It is a measure of how many observations of false class are labeled correctly. E.g. Proportion of patients without the disease who test negative.

Specificity = TN / (TN + FP)

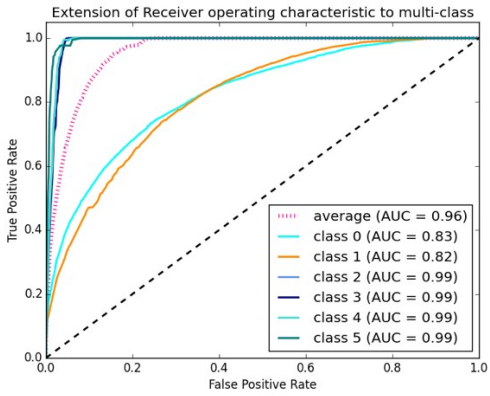
Specificity and Sensitivity plays a crucial role in deriving ROC curve. **ROC**

curve

- Receiver Operating Characteristic(ROC) summarizes the model's performance by evaluating the tradeoffs between true positive rate (Sensitivity) and false positive rate (1- Specificity).
- The area under curve (AUC), referred to as index of accuracy(A) or concordance index, is a perfect performance metric for ROC curve. Higher the area under curve, better the prediction power of the model.
- The area of this curve measures the ability of the model to correctly classify true positives and true negatives. We want our model to predict the true classes as true and false classes as false.
- So it can be said that we want the true positive rate to be 1. But we are not concerned with the true positive rate only but the false positive rate too. For example in our problem, we are not only concerned about predicting the Y classes as Y but we also want N classes to be predicted as N.

- We want to increase the area of the curve which will be maximum for class 2,3,4 and 5 in the above example.
- For class 1 when the false positive rate is 0.2, the true positive rate is around 0.6. But for class 2 the true positive rate is 1 at the same false positive rate. So, the AUC for class 2 will be much more as compared to the AUC for class 1. So, the model for class 2 will be better.
- The class 2,3,4 and 5 model will predict more accurately as compared to the class 0 and 1 model as the AUC is more for those classes.

This is how a ROC curve looks like:



Model Building: Part I

Now, let's build our first model to predict the target variable using **Logistic Regression**. Logistic Regression is a classification algorithm commonly used for predicting binary outcomes, such as 1/0, Yes/No, True/False, given a set of independent variables.

In Logistic Regression, the **Logit function** is used to estimate the probability of a certain event happening. The logit function is the logarithm of the odds in favor of the event, creating an S-shaped curve that maps inputs to probability values. This probability estimate lies between 0 and 1, which makes it a perfect fit for binary classification problems.

To start, we'll drop the **Loan_ID** variable from our dataset, as it doesn't have any effect on the loan status. We will apply the same preprocessing steps to the **test dataset** as we did for the **training dataset**.

We will use scikit-learn (sklearn) for making different models which is an open-source library for Python. It is one of the most efficient tools which contains many inbuilt functions that can be used for modeling in Python.

Sklearn requires the target variable in a separate dataset. So, we will drop our target variable from the train dataset and save it in another dataset.

[119]:
```
# drop Loan_ID
train = train.drop('Loan_ID', axis=1)
test = test.drop('Loan_ID', axis=1)
```

[121]:
```
# Drop "Loan_Status" and assign it to feature variable X, and target variable y
X = train.drop('Loan_Status', axis=1)
y = train['Loan_Status']
```

[123]:
```
# adding dummies to the dataset
X = pd.get_dummies(X)
train = pd.get_dummies(train)
test = pd.get_dummies(test)
```

[125]:
```
X.shape, train.shape, test.shape
```

[125]:
```
((614, 21), (614, 22), (367, 21))
```

[127]:
```
X.head()
```

[127]:

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	LoanAmount_log	Gender_Female	Gender_Male	Married_No	Married_
0	5849	0.0	128.0	360.0	1.0	4.852030	False	True	True	F
1	4583	1508.0	128.0	360.0	1.0	4.852030	False	True	False	1
2	3000	0.0	66.0	360.0	1.0	4.189655	False	True	False	1
3	2583	2358.0	120.0	360.0	1.0	4.787492	False	True	False	1
4	6000	0.0	141.0	360.0	1.0	4.948760	False	True	True	

5 rows × 21 columns

[129]:
```
# import library
from sklearn.model_selection import train_test_split
```

5 rows × 21 columns

[129]:
```
# import library
from sklearn.model_selection import train_test_split
```

[131]:
```
# split the data into train and cross validation set
x_train, x_cv, y_train, y_cv = train_test_split(X, y, test_size=0.3, random_state=0)
```

[133]:
```
# take a look at the dimension of the data
x_train.shape, x_cv.shape, y_train.shape, y_cv.shape
```

[133]:
```
((429, 21), (185, 21), (429,), (185,))
```

[135]:
```
# import libraries
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

[137]:
```
# fit the model
model = LogisticRegression()
model.fit(x_train, y_train)
```

[137]:
```
LogisticRegression()
```

[139]:
```
# make prediction
pred_cv = model.predict(x_cv)
```

[141]:
```
# calculate accuracy score
accuracy_score(y_cv, pred_cv)
```

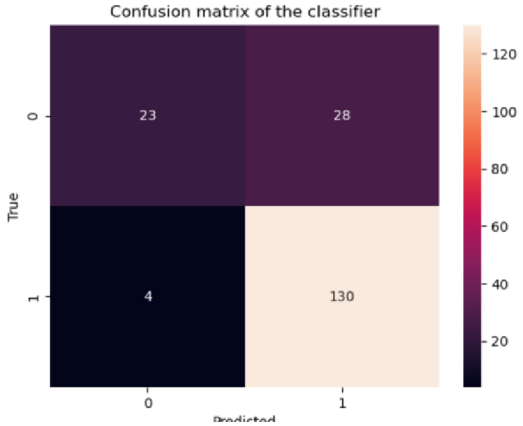
[141]:
```
0.827027027027027
```

[143]:
```
# import confusion_matrix
```

Code JupyterLab Python 3 (ipykernel)

[143]:
```
# import confusion_matrix
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_cv, pred_cv)
print(cm)

# f, ax = plt.subplots(figsize=(9, 6))
sns.heatmap(cm, annot=True, fmt="d")
plt.title('Confusion matrix of the classifier')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

```
[[ 23  28]
 [  4 130]]
```



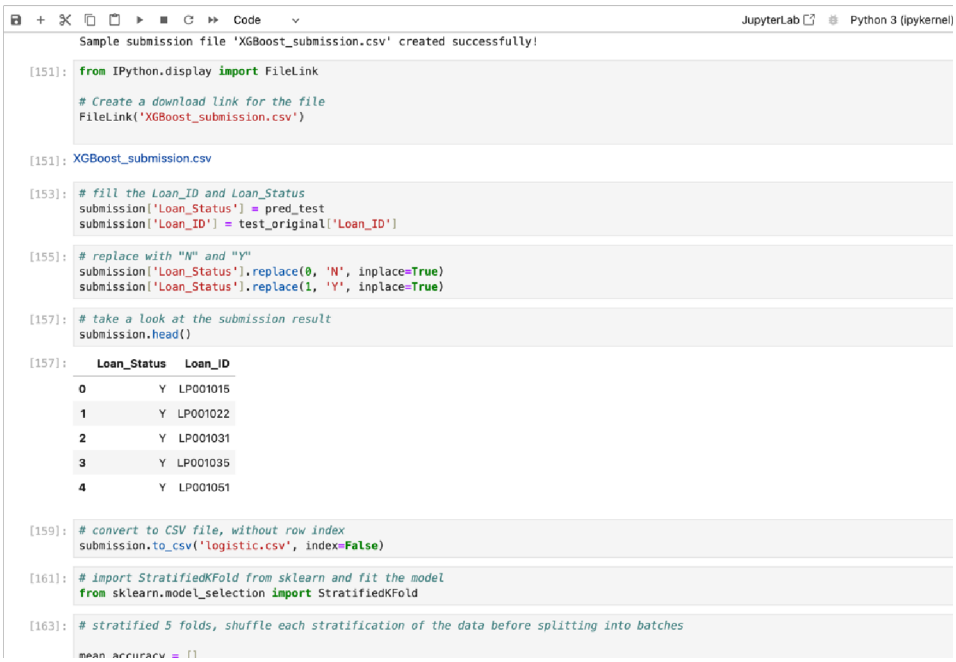
Code JupyterLab Python 3 (ipykernel)

[145]:
```
# import classification_report
from sklearn.metrics import classification_report
print(classification_report(y_cv, pred_cv))
```

```
              precision    recall  f1-score   support

           0       0.85      0.45      0.59        51
           1       0.82      0.97      0.89       134

    accuracy                           0.83       185
   macro avg       0.84      0.71      0.74       185
weighted avg       0.83      0.83      0.81       185
```

[147]:
```
# make prediction on test set
pred_test = model.predict(test)
```

[149]:
```
import pandas as pd

# Assuming you have the following:
# 'pred_test' contains your predicted loan statuses (0 or 1)
# 'test_original' contains your test dataset and its 'Loan_ID' column

# Create the submission DataFrame
submission = pd.DataFrame()

# Fill in the predicted loan status
submission['Loan_Status'] = pred_test

# Fill in the Loan_ID from your original test data
submission['Loan_ID'] = test_original['Loan_ID']

# Replace 0 and 1 with 'N' and 'Y'
submission['Loan_Status'].replace(0, 'N', inplace=True)
submission['Loan_Status'].replace(1, 'Y', inplace=True)

# Convert the DataFrame to CSV format and save it
submission.to_csv('XGBoost_submission.csv', index=False)
```



```
Sample submission file 'XGBoost_submission.csv' created successfully!

[151]: from IPython.display import FileLink

       # Create a download link for the file
       FileLink('XGBoost_submission.csv')

[151]: XGBoost_submission.csv

[153]: # fill the Loan_ID and Loan_Status
       submission['Loan_Status'] = pred_test
       submission['Loan_ID'] = test_original['Loan_ID']

[155]: # replace with "N" and "Y"
       submission['Loan_Status'].replace(0, 'N', inplace=True)
       submission['Loan_Status'].replace(1, 'Y', inplace=True)

[157]: # take a look at the submission result
       submission.head()

[157]:
```

	Loan_Status	Loan_ID
0	Y	LP001015
1	Y	LP001022
2	Y	LP001031
3	Y	LP001035
4	Y	LP001051

```
[159]: # convert to CSV file, without row index
       submission.to_csv('logistic.csv', index=False)

[161]: # import StratifiedKFold from sklearn and fit the model
       from sklearn.model_selection import StratifiedKFold

[163]: # stratified 5 folds, shuffle each stratification of the data before splitting into batches

       mean accuracy = []
```

Logistic Regression with Stratified K-Fold Cross Valida1on

In our recent submission, we achieved an accuracy of **0.7847** on the leaderboard. To improve our model's robustness and validate its performance on unseen data, we can use **cross-validation** instead of creating a separate validation set.

Cross-Validation Techniques:

- **Validation Set Approach**
- **K-Fold Cross Validation**
- **Leave-One-Out Cross Validation (LOOCV)**
- **Stratified K-Fold Cross Validation**

In this section, we focus on **Stratified K-Fold Cross Validation**, which is a robust validation technique. Here's how it works:

- **Stratification** involves rearranging the data to ensure that each fold is a good representation of the whole dataset. For example, in a binary classification problem with a balanced distribution of classes (50% each), each fold will contain approximately equal proportions of both classes.
- This technique is especially beneficial when dealing with **class imbalance** as it ensures each fold adequately represents the minority class, preventing biased performance estimates.

If the value of **K** equals the number of observations **N** in the dataset, it is referred to as **Leave-One-Out Cross Validation (LOOCV)**.

Below is a visualization of Stratified K-Fold Cross Validation with **k=5** folds.

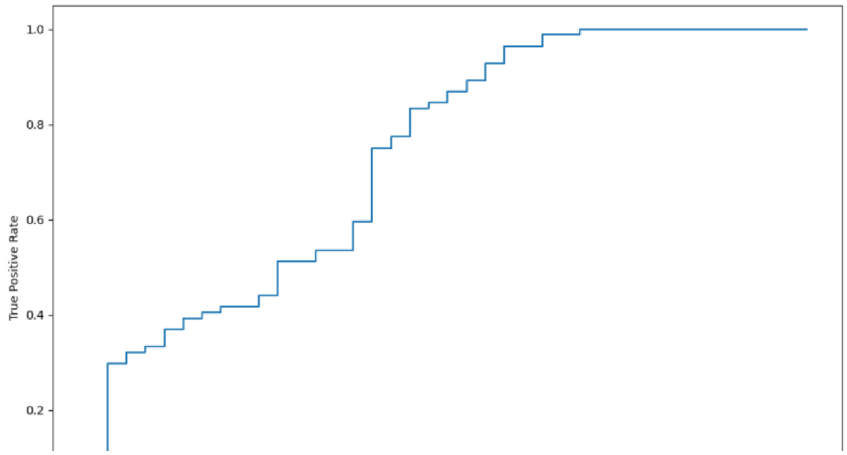
```
from sklearn.model_selection import StratifiedKFold
```

```
[163]: # stratified 5 folds, shuffle each stratification of the data before splitting into batches

mean_accuracy = []
i = 1
kf = StratifiedKFold(n_splits=5, random_state=1, shuffle=True)

for train_index, test_index in kf.split(X, y):
    print('\n{} of kfold {}'.format(i, kf.n_splits))
    xtr, xvl = X.loc[train_index], X.loc[test_index]
    ytr, yvl = y[train_index], y[test_index]

    model = LogisticRegression(random_state=1)
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl, pred_test)
    mean_accuracy.append(score)
    print('accuracy_score', score)
    i+=1

print("\nMean validation accuracy: ", sum(mean_accuracy)/len(mean_accuracy))


# make prediction on test set
pred_test = model.predict(test)


# calculate probability estimates of loan approval
# column 0 is the probability for class 0 and column 1 is the probability for class 1
# probability of loan default = 1 - model.predict_proba(test)[:,1]
pred = model.predict_proba(xvl)[:,1]
```

```
1 of kfold 5
accuracy_score 0.8048780487804879

2 of kfold 5
accuracy_score 0.8373983739837398

3 of kfold 5
```

Code JupyterLab Python 3 (ipykernel)

```
[167]: # visualize ROC curve
from sklearn import metrics
fpr, tpr, _ = metrics.roc_curve(yvl,  pred)
auc = metrics.roc_auc_score(yvl, pred)
plt.figure(figsize=(12,8))
plt.plot(fpr,tpr,label="validation, auc="+str(auc))
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc=4)
plt.show()
```



Feature Engineering:

Feature engineering involves creating new features based on domain knowledge, which may help improve the model's predictive power. Based on the analysis, we will create the following three new features:

1. **Total Income**: This feature is the sum of the **Applicant Income** and **Coapplicant Income**. A higher total income may indicate a higher likelihood of loan approval, as it reflects a greater financial capacity to repay the loan.

2. **Equated Monthly Installment (EMI)**: EMI represents the monthly repayment amount for the loan. We calculate it as the ratio of **Loan Amount** to the **Loan Amount Term** (in months). Individuals with higher EMI may face more difficulty in repaying the loan, which could reduce their chances of loan approval.

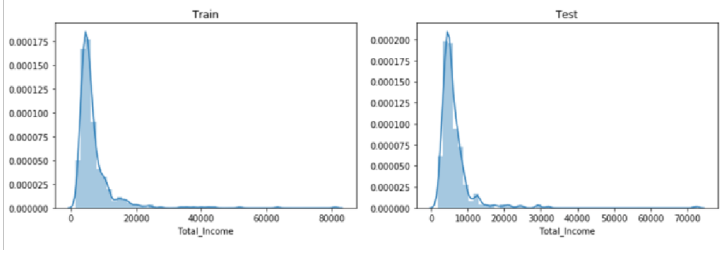
3. **Balance Income**: This feature is the income left after paying the EMI. It represents the remaining disposable income after the monthly loan repayment. A higher balance income suggests that the applicant has sufficient funds left after paying the EMI, increasing the likelihood of loan repayment and therefore, loan approval.

These new features will help provide additional insights into the financial profile of applicants, enhancing the model's predictive accuracy.

```
# combine Applicant Income and Coapplicant Income into a new variable
train['Total_Income'] = train['ApplicantIncome'] + train['CoapplicantIncome']
test['Total_Income'] = test['ApplicantIncome'] + test['CoapplicantIncome']
```

```
# check the distribution of Total Income
# before log transformation

fig = plt.figure(figsize=(14, 4))
ax1 = plt.subplot(121)
sns.distplot(train['Total_Income'])

ax1.set_title("Train")

ax1 = plt.subplot(122)
sns.distplot(test['Total_Income'])
ax1.set_title("Test")
```

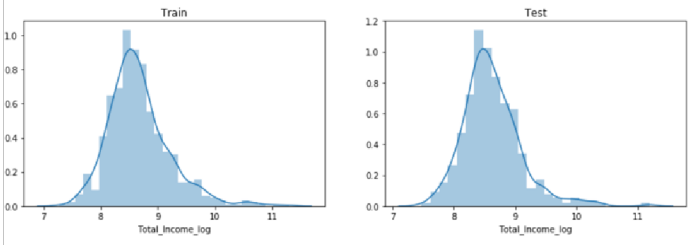
```
Text(0.5,1,'Test')
```




```
# log transformation
train['Total_Income_log'] = np.log(train['Total_Income'])
test['Total_Income_log'] = np.log(test['Total_Income'])
```

```
# after log transformation
fig = plt.figure(figsize=(14, 4))
ax1 = plt.subplot(121)
sns.distplot(train['Total_Income_log'])
ax1.set_title("Train")

ax1 = plt.subplot(122)
sns.distplot(test['Total_Income_log'])
ax1.set_title("Test")
```

```
Text(0.5,1,'Test')
```



False Positive Rate

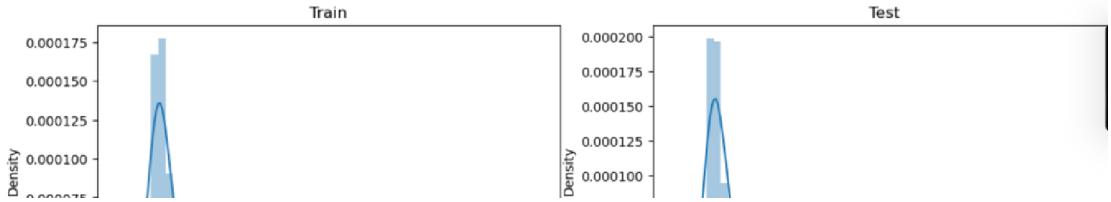
[169]:
```
submission['Loan_Status'] = pred_test
submission['Loan_ID'] = test_original['Loan_ID']
```

[171]:
```
submission['Loan_Status'].replace(0, 'N', inplace=True)
submission['Loan_Status'].replace(1, 'Y', inplace=True)
```

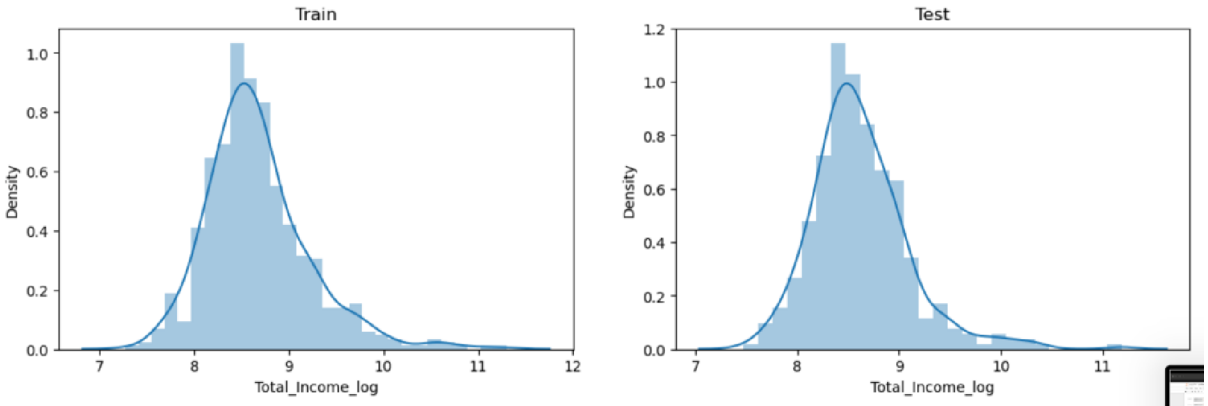
[173]:
```
# convert to CSV file, without row index
submission.to_csv('logistic.csv', index=False)
```

[175]:
```
# combine Applicant Income and Coapplicant Income into a new variable
train['Total_Income'] = train['ApplicantIncome'] + train['CoapplicantIncome']
test['Total_Income'] = test['ApplicantIncome'] + test['CoapplicantIncome']
```

[177]:
```
# check the distribution of Total Income
# before log transformation

fig = plt.figure(figsize=(14, 4))
ax1 = plt.subplot(121)
sns.distplot(train['Total_Income'])

ax1.set_title("Train")

ax1 = plt.subplot(122)
sns.distplot(test['Total_Income'])
ax1.set_title("Test")
plt.show()
```



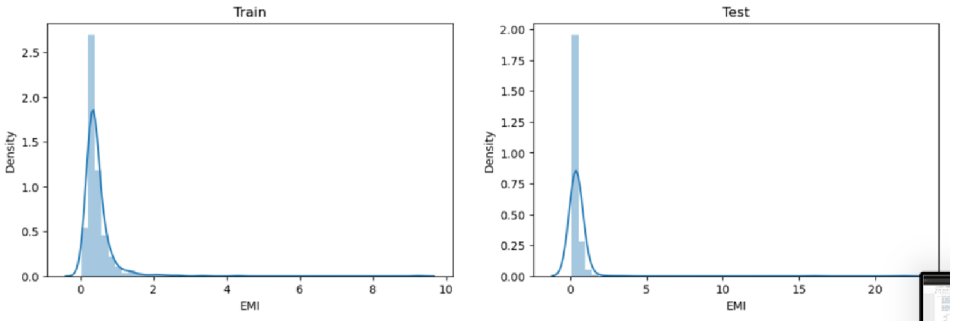
[181]:
```
# after log transformation
fig = plt.figure(figsize=(14, 4))
ax1 = plt.subplot(121)
sns.distplot(train['Total_Income_log'])
ax1.set_title("Train")

ax1 = plt.subplot(122)
sns.distplot(test['Total_Income_log'])
ax1.set_title("Test")
plt.show()
```



[183]:
```
# create EMI feature
train['EMI'] = train['LoanAmount'] / train['Loan_Amount_Term']
test['EMI'] = test['LoanAmount'] / test['Loan_Amount_Term']
```

[185]:
```
# check the distribution of EMI

fig = plt.figure(figsize=(14, 4))
```

[185]:
```
# check the distribution of EMI

fig = plt.figure(figsize=(14, 4))
ax1 = plt.subplot(121)
sns.distplot(train['EMI'])
ax1.set_title("Train")

ax1 = plt.subplot(122)
sns.distplot(test['EMI'])
ax1.set_title("Test")
plt.show()
```



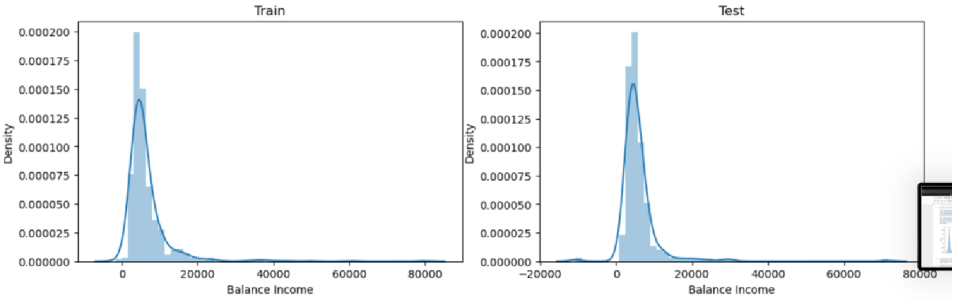
[187]:
```
# create new "Balance Income" variable
train['Balance Income'] = train['Total_Income'] - (train['EMI']*1000) # Multiply with 1000 to make the units equal
test['Balance Income'] = test['Total_Income'] - (test['EMI']*1000)
```

[189]:
```
# check the distribution of EMI
# before log transformation
```

Code JupyterLab Python 3 (ipykernel)

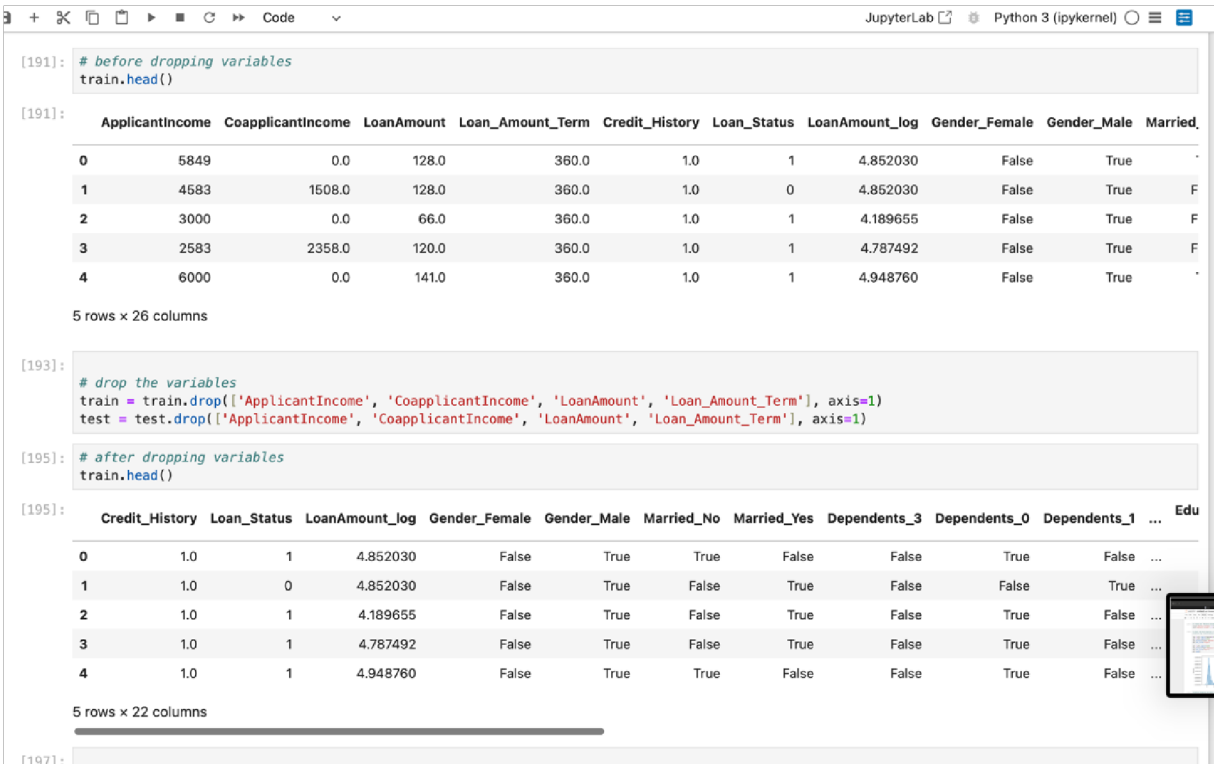
[187]:
```
# create new "Balance Income" variable
train['Balance Income'] = train['Total_Income'] - (train['EMI']*1000) # Multiply with 1000 to make the units equal
test['Balance Income'] = test['Total_Income'] - (test['EMI']*1000)
```

[189]:
```
# check the distribution of EMI
# before log transformation

fig = plt.figure(figsize=(14, 4))
ax1 = plt.subplot(121)
sns.distplot(train['Balance Income'])
ax1.set_title("Train")

ax1 = plt.subplot(122)
sns.distplot(test['Balance Income'])
ax1.set_title("Test")
plt.show()
```



[191]:
```
# before dropping variables
```

Now, the distribution appears much closer to normal, and the impact of extreme values has been significantly reduced. Next, let's create the **EMI** feature. We can calculate the **EMI** by taking the ratio of the **Loan Amount** to the **Loan Amount Term**. This calculation provides an approximation of the actual EMI amount, giving us an estimate of the monthly repayment



```
[191]: # before dropping variables
       train.head()
```

[191]:

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Loan_Status	LoanAmount_log	Gender_Female	Gender_Male	Married
0	5849	0.0	128.0	360.0	1.0	1	4.852030	False	True	
1	4583	1508.0	128.0	360.0	1.0	0	4.852030	False	True	F
2	3000	0.0	66.0	360.0	1.0	1	4.189655	False	True	F
3	2583	2358.0	120.0	360.0	1.0	1	4.787492	False	True	F
4	6000	0.0	141.0	360.0	1.0	1	4.948760	False	True	

5 rows × 26 columns

```
[193]: # drop the variables
       train = train.drop(['ApplicantIncome', 'CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term'], axis=1)
       test = test.drop(['ApplicantIncome', 'CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term'], axis=1)

[195]: # after dropping variables
       train.head()
```

[195]:

	Credit_History	Loan_Status	LoanAmount_log	Gender_Female	Gender_Male	Married_No	Married_Yes	Dependents_3	Dependents_0	Dependents_1	...	Edu
0	1.0	1	4.852030	False	True	True	False	False	True	False	...	
1	1.0	0	4.852030	False	True	False	True	False	False	True	...	
2	1.0	1	4.189655	False	True	False	True	False	True	False	...	
3	1.0	1	4.787492	False	True	False	True	False	True	False	...	
4	1.0	1	4.948760	False	True	True	False	False	True	False	...	

5 rows × 22 columns

Model Building: Part II

After creating the new features, we can proceed with the model building process. We will begin with a Logistic Regression model and then explore more complex models such as Decision Tree, Random Forest, and XGBoost.

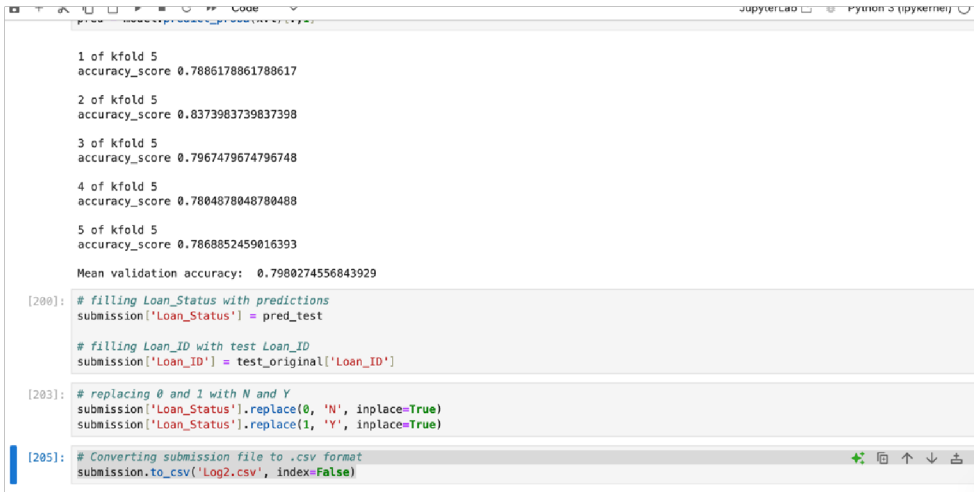
The models we will build in this section are:

1. Logistic Regression
2. Decision Tree
3. Random Forest
4. XGBoost

Logistic Regression:

[199]:
```
# stratified 5 folds, shuffle each stratification of the data before splitting into batches

mean_accuracy = []
i = 1
kf = StratifiedKFold(n_splits=5, random_state=1, shuffle=True)

for train_index, test_index in kf.split(X, y):
    print('\n{} of kfold {}'.format(i, kf.n_splits))
    xtr, xvl = X.loc[train_index], X.loc[test_index]
    ytr, yvl = y[train_index], y[test_index]

    model = LogisticRegression(random_state=1)
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl, pred_test)
    mean_accuracy.append(score)
    print('accuracy_score', score)
    i+=1

print("\nMean validation accuracy: ", sum(mean_accuracy)/len(mean_accuracy))


# make prediction on test set
pred_test = model.predict(test)


# calculate probability estimates of loan approval
# column 0 is the probability for class 0 and column 1 is the probability for class 1
# probability of loan default = 1 - model.predict_proba(test)[:,1]
pred = model.predict_proba(xvl)[:,1]
```

```
1 of kfold 5
accuracy_score 0.7886178861788617

2 of kfold 5
accuracy_score 0.8373983739837398

3 of kfold 5
accuracy_score 0.7967479674796748
```



```
1 of kfold 5
accuracy_score 0.7886178861788617

2 of kfold 5
accuracy_score 0.8373983739837398

3 of kfold 5
accuracy_score 0.7967479674796748

4 of kfold 5
accuracy_score 0.7804878048780488

5 of kfold 5
accuracy_score 0.7868852459016393

Mean validation accuracy:  0.7980274556843929
```

[200]:
```
# filling Loan_Status with predictions
submission['Loan_Status'] = pred_test

# filling Loan_ID with test Loan_ID
submission['Loan_ID'] = test_original['Loan_ID']
```

[203]:
```
# replacing 0 and 1 with N and Y
submission['Loan_Status'].replace(0, 'N', inplace=True)
submission['Loan_Status'].replace(1, 'Y', inplace=True)
```

[205]:
```
# Converting submission file to .csv format
submission.to_csv('Log2.csv', index=False)
```

•

```
accuracy_score 0.7983870967741935

2 of kfold 5
accuracy_score 0.8225806451612904

3 of kfold 5
accuracy_score 0.7786885245901639

4 of kfold 5
accuracy_score 0.7868852459016393

5 of kfold 5
accuracy_score 0.8278688524590164

Mean validation accuracy:  0.8028820729772608
```

The mean validation accuracy for this model is 0.803

```
# filling Loan_Status with predictions
submission['Loan_Status'] = pred_test

# filling Loan_ID with test Loan_ID
submission['Loan_ID'] = test_original['Loan_ID']
```

```
# replacing 0 and 1 with N and Y
submission['Loan_Status'].replace(0, 'N', inplace=True)
submission['Loan_Status'].replace(1, 'Y', inplace=True)
```

```
# Converting submission file to .csv format
submission.to_csv('Log2.csv', index=False)
```

Decision Tree:

Decision tree is a type of supervised learning algorithm (having a pre-defined target variable) that is mostly used in classification problems. 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.

Decision trees use multiple algorithms to decide to split a node in two or more sub-nodes. The creation of sub-nodes increases the homogeneity of resultant sub-nodes. In other words, we can say that purity of the node increases with respect to the target variable.

```
# import library
from sklearn import tree
```

Let's fit the decision tree model with 5 folds of cross validation.

```
mean_accuracy = []
i=1
kf = StratifiedKFold(n_splits=5,random_state=1,shuffle=True)
for train_index,test_index in kf.split(X,y):
    print('\n{} of kfold {}'.format(i,kf.n_splits))
    xtr,xvl = X.loc[train_index],X.loc[test_index]
    ytr,yvl = y[train_index],y[test_index]

    model = tree.DecisionTreeClassifier(random_state=1)
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl,pred_test)
    mean_accuracy.append(score)
    print('accuracy_score',score)
    i+=1

print("\nMean validation accuracy: ", sum(mean_accuracy)/len(mean_accuracy))
pred_test = model.predict(test)
```

```
1 of kfold 5
accuracy_score 0.7258064516129032

2 of kfold 5
accuracy_score 0.7419354838709677

3 of kfold 5
accuracy_score 0.7049180327868853

4 of kfold 5
accuracy_score 0.680327868852459

5 of kfold 5
accuracy_score 0.7049180327868853

Mean validation accuracy:  0.7115811739820201
```

```
# filling Loan_Status with predictions
submission['Loan_Status'] = pred_test

# filling Loan_ID with test Loan_ID
submission['Loan_ID'] = test_original['Loan_ID']
```

```
# replacing 0 and 1 with N and Y
submission['Loan_Status'].replace(0, 'N', inplace=True)
submission['Loan_Status'].replace(1, 'Y', inplace=True)
```

```
# Converting submission file to .csv format
submission.to_csv('Decision Tree.csv', index=False)
```

Random Forest:

- RandomForest is a tree based bootstrapping algorithm wherein a certain no. of weak learners (decision trees) are combined to make a powerful prediction model.
- For every individual learner, a random sample of rows and a few randomly chosen variables are used to build a decision tree model.
- Final prediction can be a function of all the predictions made by the individual learners.
- In case of regression problem, the final prediction can be mean of all the predictions.

There are some parameters worth exploring with the sklearn RandomForestClassifier:

- n_estimators

- max_features
- n_estimators = usually bigger the forest the better, there is small chance of overfitting here. The more estimators you give it, the better it will do. We will use the default value of 10.

max depth of each tree (default none, leading to full tree) - reduction of the maximum depth helps fighting with overfishing. We will limit at 10

```
# import library
from sklearn.ensemble import RandomForestClassifier
```

```
mean_accuracy = []
i=1
kf = StratifiedKFold(n_splits=5,random_state=1,shuffle=True)
for train_index,test_index in kf.split(X, y):
    print('\n{} of kfold {}'.format(i,kf.n_splits))
    xtr,xvl = X.loc[train_index],X.loc[test_index]
    ytr,yvl = y[train_index],y[test_index]

    model = RandomForestClassifier(random_state=1, max_depth=10, n_estimators=10)
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl,pred_test)
    mean_accuracy.append(score)
    print('accuracy_score',score)
    i+=1

print("\nMean validation accuracy: ", sum(mean_accuracy)/len(mean_accuracy))
pred_test = model.predict(test)
```

```
1 of kfold 5
accuracy_score 0.8225806451612904

2 of kfold 5
accuracy_score 0.8145161290322581

3 of kfold 5
accuracy_score 0.7377049180327869

4 of kfold 5
accuracy_score 0.7295081967213115

5 of kfold 5
accuracy_score 0.8114754098360656

Mean validation accuracy:  0.7831570597567425
```

GridSearchCV:

We will try to improve the accuracy by tuning the hyperparameters for this model. We will use grid search to get the optimized values of hyper parameters. GridSearch is a way to select the best of a family of hyper parameters, parametrized by a grid of parameters.

We will use GridSearchCV in sklearn.model_selection for an exhaustive search over specified parameter values for an estimator. GridSearchCV implements a “fit” and a “score” method. It also implements “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform” if they are implemented in the estimator used.

```
# import library
from sklearn.model_selection import GridSearchCV
```

```
# Provide range for max_depth from 1 to 20 with an interval of 2 and from 1 to 200 with an interval of 20 for n_e
paramgrid = {'max_depth': list(range(1, 20, 2)), 'n_estimators': list(range(1, 200, 20))}
```

```
# default 3-fold cross validation, cv=3
grid_search = GridSearchCV(RandomForestClassifier(random_state=1), paramgrid)
```

```
# split the data
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, random_state=1)
```

```
# fit the grid search model
grid_search.fit(x_train, y_train)
```

```
GridSearchCV(cv=None, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
            oob_score=False, random_state=1, verbose=0, warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'max_depth': [1, 3, 5, 7, 9, 11, 13, 15, 17, 19], 'n_estimators': [1, 21, 41, 61, 81, 101, 12
1, 141, 161, 181]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

After the grid search model has been fit, we can use use `best_estimator_` to obtain the estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data.

```
# estimate the optimized value
```

```
# estimate the optimized value
grid_search.best_estimator_
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=3, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=141, n_jobs=1,
            oob_score=False, random_state=1, verbose=0, warm_start=False)
```

So, the optimized value for the max_depth variable is 3 and for n_estimator is 141. Now let's build the model using these optimized values.

```
mean_accuracy = []
i=1
kf = StratifiedKFold(n_splits=5,random_state=1,shuffle=True)
for train_index,test_index in kf.split(X,y):
    print('\n{} of kfold {}'.format(i,kf.n_splits))
    xtr,xvl = X.loc[train_index],X.loc[test_index]
    ytr,yvl = y[train_index],y[test_index]

    model = RandomForestClassifier(random_state=1, max_depth=3, n_estimators=141)
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl,pred_test)
    mean_accuracy.append(score)
    print('accuracy_score',score)
    i+=1

print("\nMean validation accuracy: ", sum(mean_accuracy)/len(mean_accuracy))
pred_test = model.predict(test)
pred2=model.predict_proba(test)[:,1]
```

```
 of kfold 5
ccuracy_score 0.8064516129032258

 of kfold 5
ccuracy_score 0.8306451612903226

 of kfold 5
```

```
print("\nMean validation accuracy: ", sum(mean_accuracy)/len(mean_accuracy))
pred_test = model.predict(test)
pred2=model.predict_proba(test)[:,1]
```

```
1 of kfold 5
accuracy_score 0.8064516129032258

2 of kfold 5
accuracy_score 0.8306451612903226

3 of kfold 5
accuracy_score 0.8032786885245902

4 of kfold 5
accuracy_score 0.7950819672131147

5 of kfold 5
accuracy_score 0.8278688524590164

Mean validation accuracy:  0.8126652564780541
```

The mean validation accuracy has improved from 0.783 to 0.813

```
# filling Loan_Status with predictions
submission['Loan_Status']=pred_test

# filling Loan_ID with test Loan_ID
submission['Loan_ID']=test_original['Loan_ID']
```

```
# replacing 0 and 1 with N and Y
submission['Loan_Status'].replace(0, 'N', inplace=True)
submission['Loan_Status'].replace(1, 'Y', inplace=True)
```

```
# Converting submission file to .csv format
submission.to_csv('Random Forest.csv', index=False)
```

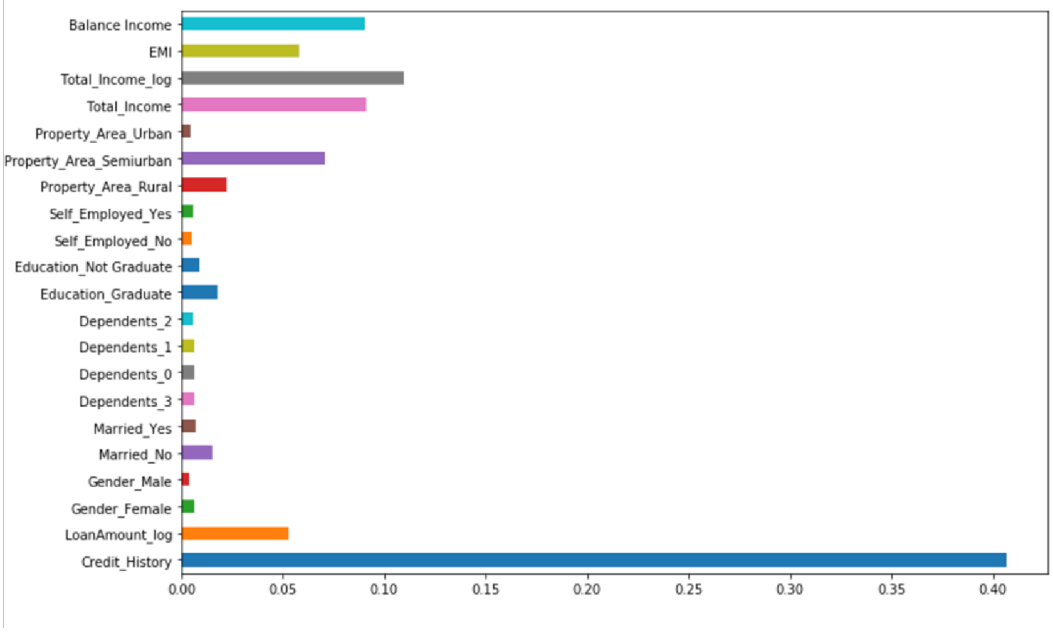
We got an accuracy of 0.7638 from the random forest model on leaderboard.

Feature Importance:

Let us find the feature importance now, i.e. which features are most important for this problem. We will use feature_importances_ attribute of sklearn to do so. It will return the feature importances (the higher, the more important the feature).

```
# extract feature importances, convert into a Series
importances = pd.Series(model.feature_importances_, index=X.columns)

# plot the horizontal bar chart
importances.plot(kind='barh', figsize=(12,8))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x2954721ef28>
```



XGBoost:

```
# import library
from xgboost import XGBClassifier
```

```
mean_accuracy = []
i=1
kf = StratifiedKFold(n_splits=5,random_state=1,shuffle=True)
for train_index,test_index in kf.split(X,y):
    print('\n{} of kfold {}'.format(i,kf.n_splits))
    xtr,xvl = X.loc[train_index],X.loc[test_index]
    ytr,yvl = y[train_index],y[test_index]

    model = XGBClassifier(random_state=1, n_estimators=50, max_depth=4)
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl,pred_test)
    mean_accuracy.append(score)
    print('accuracy_score',score)
    i+=1

print("\nMean validation accuracy: ", sum(mean_accuracy)/len(mean_accuracy))
pred_test = model.predict(test)
pred3=model.predict_proba(test)[:,1]

# warnings.filterwarnings(action='ignore', category=DeprecationWarning)
```

```
1 of kfold 5
accuracy_score 0.782258064516129

2 of kfold 5
accuracy_score 0.8225806451612904

3 of kfold 5
accuracy_score 0.7622950819672131

4 of kfold 5
accuracy_score 0.7459016393442623

5 of kfold 5
accuracy_score 0.7868852459016393
```

```
# import library
from sklearn.model_selection import GridSearchCV
```

```
# Provide range for max_depth from 1 to 20 with an interval of 2 and from 1 to 200 with an interval of 20 for n_e
paramgrid = {'max_depth': list(range(1, 20, 2)), 'n_estimators': list(range(1, 200, 20))}
```

```
# default 3-fold cross validation, cv=3
grid_search = GridSearchCV(XGBClassifier(random_state=1), paramgrid)
```

```
# split the data
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, random_state=1)
```

```
# fit the grid search model
grid_search.fit(x_train, y_train)
```

```
GridSearchCV(cv=None, error_score='raise',
       estimator=XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
       n_jobs=1, nthread=None, objective='binary:logistic', random_state=1,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'max_depth': [1, 3, 5, 7, 9, 11, 13, 15, 17, 19], 'n_estimators': [1, 21, 41, 61, 81, 101, 12
1, 141, 161, 181]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

```
# estimate the optimized value
grid_search.best_estimator_
```

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
```

```
mean_accuracy = []
i=1
kf = StratifiedKFold(n_splits=5,random_state=1,shuffle=True)
for train_index,test_index in kf.split(X,y):
    print('\n{} of kfold {}'.format(i,kf.n_splits))
    xtr,xvl = X.loc[train_index],X.loc[test_index]
    ytr,yvl = y[train_index],y[test_index]

    model = XGBClassifier(random_state=1, n_estimators=81, max_depth=1)
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl,pred_test)
    mean_accuracy.append(score)
    print('accuracy_score',score)
    i+=1

print("\nMean validation accuracy: ", sum(mean_accuracy)/len(mean_accuracy))
pred_test = model.predict(test)
pred3 = model.predict_proba(test)[:,1]
```

```
1 of kfold 5
accuracy_score 0.8064516129032258

2 of kfold 5
accuracy_score 0.8306451612903226

3 of kfold 5
accuracy_score 0.8032786885245902

4 of kfold 5
accuracy_score 0.7868852459016393

5 of kfold 5
accuracy_score 0.8360655737704918

Mean validation accuracy:  0.8126652564780541
```

The mean validation accuracy has improved from 0.78 to 0.813


```
accuracy_score 0.8360655737704918

Mean validation accuracy:  0.8126652564780541
```

The mean validation accuracy has improved from 0.78 to 0.813


```
# filling Loan_Status with predictions
submission['Loan_Status'] = pred_test

# filling Loan_ID with test Loan_ID
submission['Loan_ID']=test_original['Loan_ID']
```

```
# replacing 0 and 1 with N and Y
submission['Loan_Status'].replace(0, 'N', inplace=True)
submission['Loan_Status'].replace(1, 'Y', inplace=True)
```

```
# Converting submission file to .csv format
submission.to_csv('XGBoost.csv', index=False)
```

We got an accuracy of 0.7778 with this model.

Conclusion:

- Best Accuracy: Logistic Regression achieved the highest accuracy on the public leaderboard (0.7847), followed by Random Forest (0.7778) and XGBoost (0.7778), with Decision Tree performing the worst (0.6458).
- Feature Engineering: While new features created via feature engineering helped in prediction, they did not significantly improve the model's overall accuracy.
- Hyperparameter Tuning: GridSearchCV improved model performance by optimizing the hyperparameters, resulting in better mean validation accuracy.
- Logistic Regression: The Logistic Regression model outperformed more complex algorithms (Random Forest, XGBoost), proving effective with no feature engineering needed.
- Baseline Model: Logistic Regression, due to its simplicity and quick implementation, serves as a good baseline to compare against more complex algorithms.