Loan Status Prediction

The aim of this project is to build a machine learning model that can predict whether the loan of an applicant will be approved or not on the basis of the details provided in the dataset.

The Dataset includes details of applicants who have applied for loan. The dataset includes details like credit history, loan amount, their income, dependents etc.

Dataset Description:

Independent Variables:

- Loan_ID – id of Loan Application
- Gender – Gender of the applicant
- Married – whether the applicant is married or not
- Dependents – number of dependents of the applicant
- Education - indicates the applicant’s education status - Graduate or Not graduate
- Self_Employed – whether the applicant is self employed or not
- ApplicantIncome – applicant’s income
- CoapplicantIncome – coapplicant’s income
- Loan_Amount – loan amount applied by the applicant
- Loan_Amount_Term   -- term of the loan in months
- Credit History – whether the loan applicant has any credit history in the Past
- Property_Area - location of the property – urban, rural or semi-urban

Dependent Variable (Target Variable):

- Loan_Status – loan Approval status. (approved =Y, not approved=N)

The dataset can be downloaded from

https://github.com/dsrscientist/DSData/blob/master/loan_prediction.csv

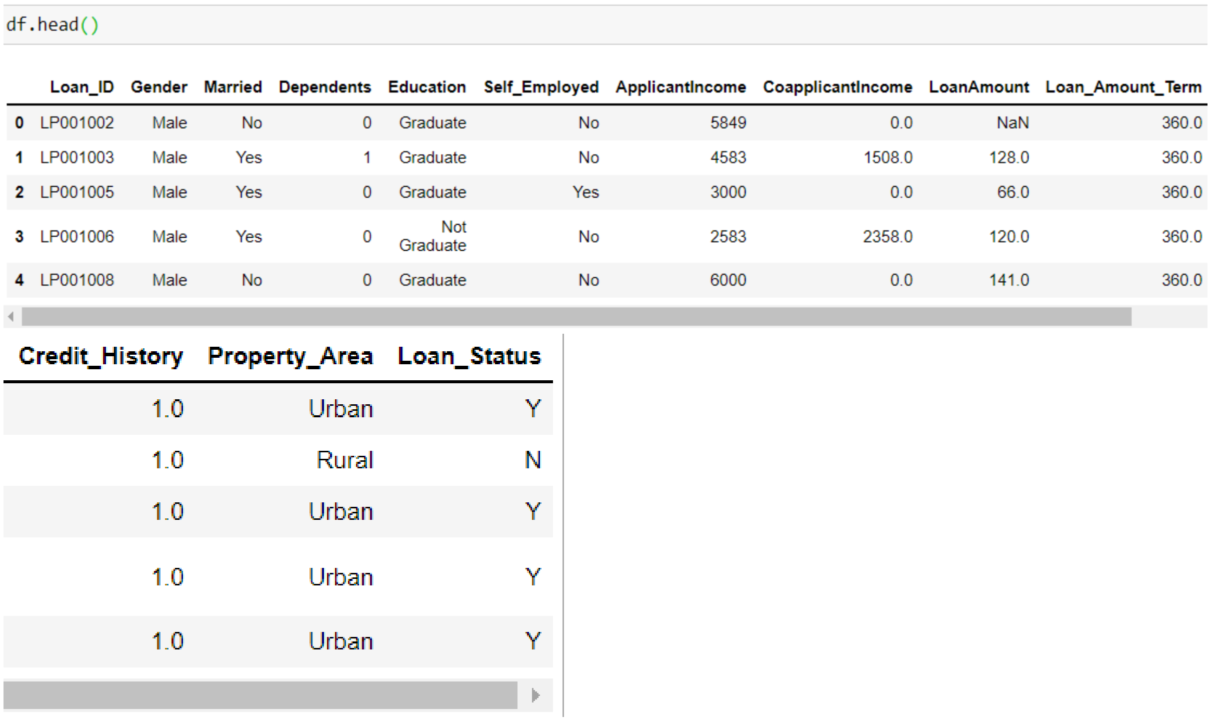
The link to the project can be found at https://github.com/sk1930/Projects/blob/main/Loan%20Application%20Status%20Prediction.ipynb

Exploratory Data Analysis

The size of the dataset is 614 rows with 13 columns.

The target feature "Loan Status" is categorical. Hence it is classification problem.

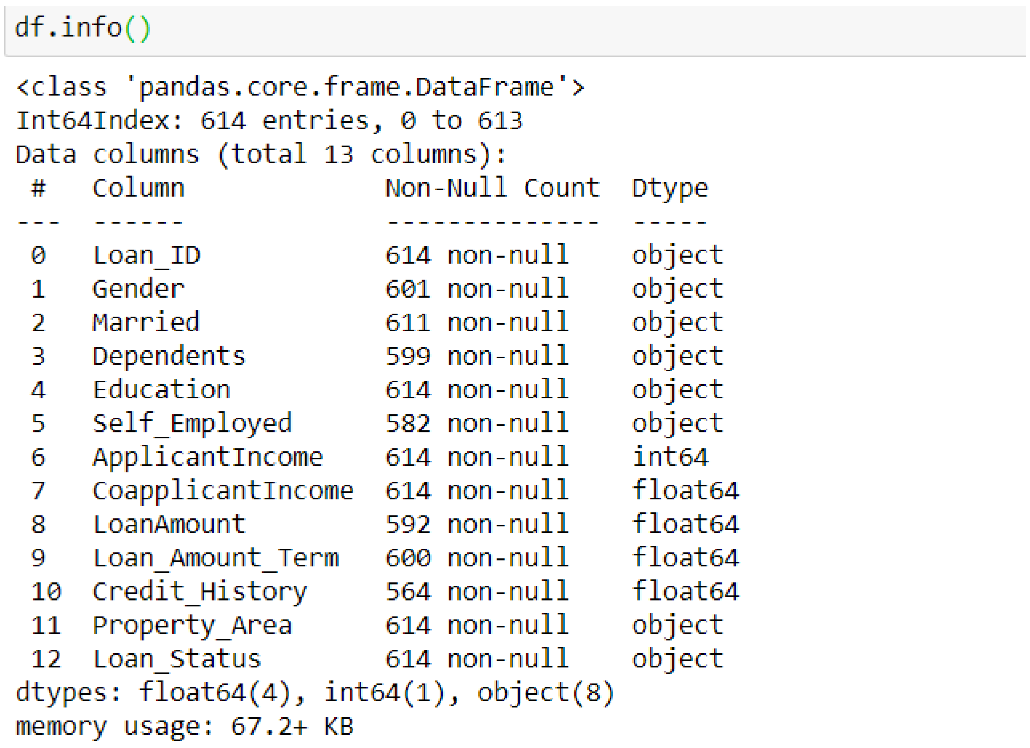
To get the overview of the data, looking at first 5 rows.



```
df.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

Checking the data types of each features using info



```
df.info()
```

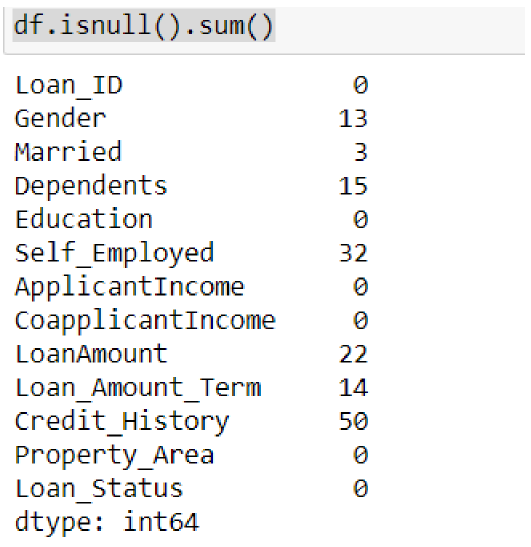
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            614 non-null    object 
 1   Gender             601 non-null    object 
 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: 67.2+ KB
```

Loan_ID, Gender, Married, Education, Self_Employed, Property_Area, Loan_Status are categorical columns

Dependents should be a numeric column, due to missing values it is being shown as object

ApplicantIncome, CoapplicantIncome, LoanAmount, Loan_Amount_Term, Credit_History are numeric columns.

Checking for missing values:



```
df.isnull().sum()
```

```
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
```

There are 13 missing values in Gender, 3 in Married , 15 for Dependents, 32 for self_employed, 22 for LoanAmount, 14 for Loan_Amount_Term, and 50 missing values in Credit_History.

```
df['Gender'].value_counts()
```

```
Male      489
Female    112
Name: Gender, dtype: int64
```

As it can be seen that most of the values in Gender are Male, filling missing values in Gender feature with Male.

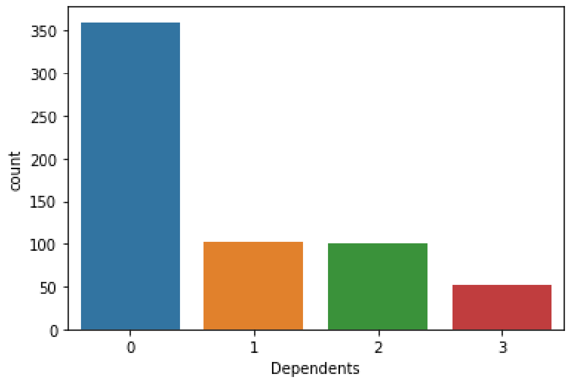
```
df.groupby('Gender')['Married'].value_counts()
```

```
Gender  Married
Female  No          80
        Yes         31
Male    Yes        367
        No         133
Name: Married, dtype: int64
```

There are 3 missing values in Married feature. Most of the females are not married and most of the males are married so filling married feature with respect to Gender, ie. For Female filling missing married values with No and for Males filling missing married values with Yes.

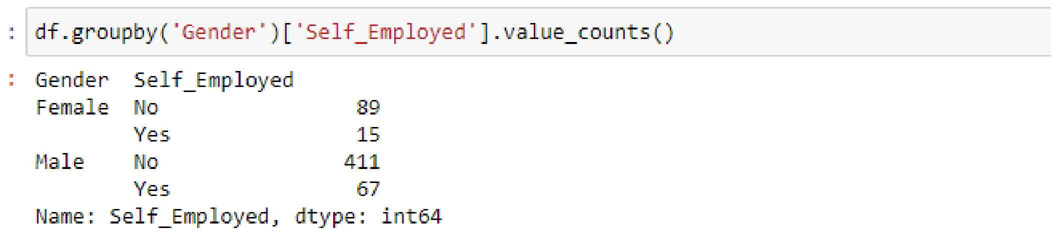
```
sns.countplot(df['Dependents'])
```

```
<AxesSubplot:xlabel='Dependents', ylabel='count'>
```



There are 15 Missing values in Dependents feature. 0 is occurring 350 times. So missing values in Dependents feature is filled with the mode (mostly repeated value). And also the value 3+ is replaced with 3 for number of dependents.

Loan applicants with 0 number of dependents is around 350, and for 1,2 number of dependents there are around 100 number of applicants each, and loan applicants with 3 number of dependents are around 50.

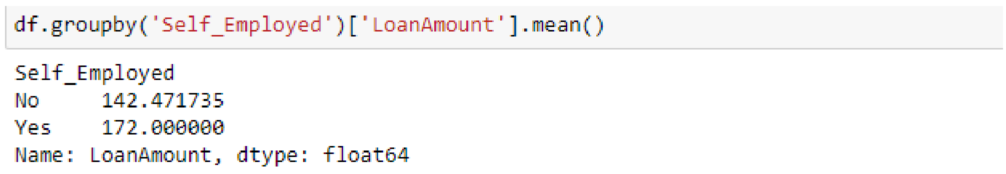


```
: df.groupby('Gender')['Self_Employed'].value_counts()
```

```
: Gender  Self_Employed
  Female  No               89
          Yes              15
  Male    No              411
          Yes              67
  Name: Self_Employed, dtype: int64
```

There are 32 missing values for self employed.

Filling missing values in Self_ Employed with respect to Gender. With respect to Gender, in females and males both majority are not Self Employed. So filling missing Self_ Employed with No.

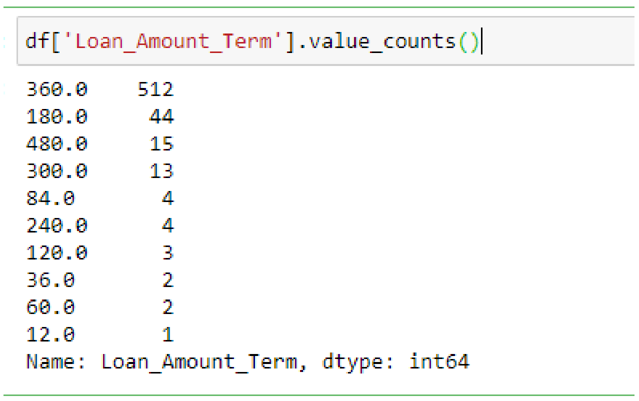


```
df.groupby('Self_Employed')['LoanAmount'].mean()
```

```
Self_Employed
No     142.471735
Yes    172.000000
Name: LoanAmount, dtype: float64
```

There are 22 missing values in Loan Amount and filling missing values based on Self employed. Grouping the data based on Self_ employed feature and taking mean of loanAmount and filling the missing values with that mean values.

df ['LoanAmount'] = df.groupby ('Self_Employed')['LoanAmount'].transform(lambda val: val.fillna(val.mean()))



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

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

There are 14 missing values in Loan Amount Term. Mostly repeated value us 360 and it is repeating 512 times. So filling missing values with the mode.

```
df['Credit_History'].value_counts()
```

```
1.0    475
0.0     89
Name: Credit_History, dtype: int64
```

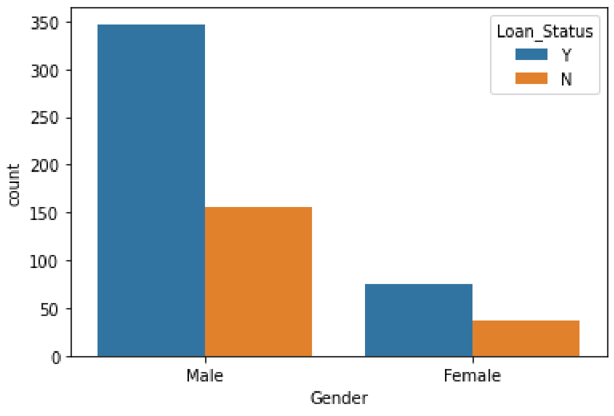
In Credit History there are 50 missing values. And credit history 1 is repeating 475 times. So filling missing values with the mode.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            614 non-null    object 
 1   Gender             614 non-null    object 
 2   Married            614 non-null    object 
 3   Dependents         614 non-null    int32  
 4   Education          614 non-null    object 
 5   Self_Employed      614 non-null    object 
 6   ApplicantIncome    614 non-null    int64  
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         614 non-null    float64
 9   Loan_Amount_Term   614 non-null    float64
 10  Credit_History     614 non-null    float64
 11  Property_Area      614 non-null    object 
 12  Loan_Status        614 non-null    object 
dtypes: float64(4), int32(1), int64(1), object(7)
memory usage: 80.9+ KB
```

After treating missing values in each feature, there are no more missing values in any feature now.

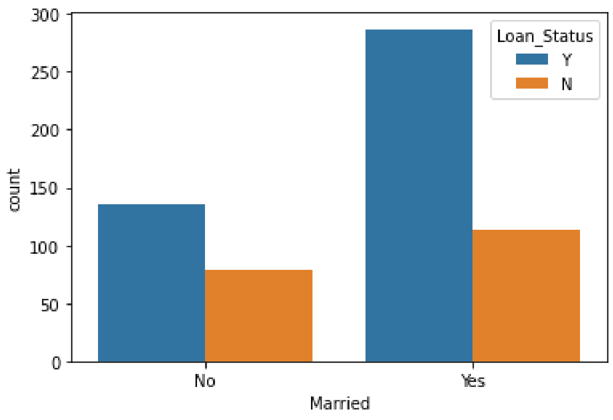
```
Gender  Loan_Status
Female  Y              0.669643
        N              0.330357
Male    Y              0.691235
        N              0.308765
Name: Loan_Status, dtype: float64
```



Gender Vs Loan Status

Gender captures two unique values Male and Female. Large number of loan applications are applied by Male compared to Females. There is high rate of loan approval in males with 69% and in females there is a bit less approval rate with 66 percent.

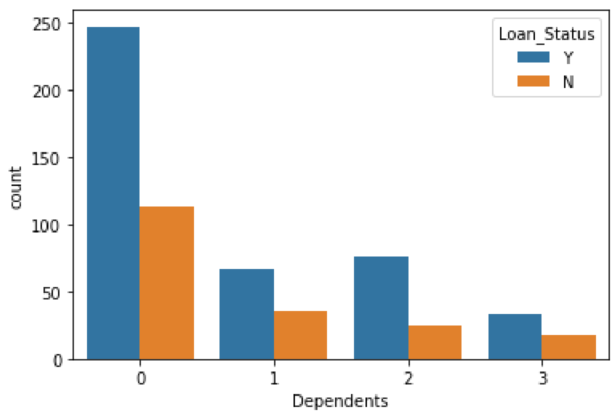
```
Married  Loan_Status
No       Y              0.630841
         N              0.369159
Yes      Y              0.717500
         N              0.282500
Name: Loan_Status, dtype: float64
```



Married vs Loan Status

Married feature has two unique values – Yes and No. More Married people apply for loans than unmarried people. There is a high approval rate for Married people with 71% and in case of unmarried people there is somewhat less approval rate with 63%.

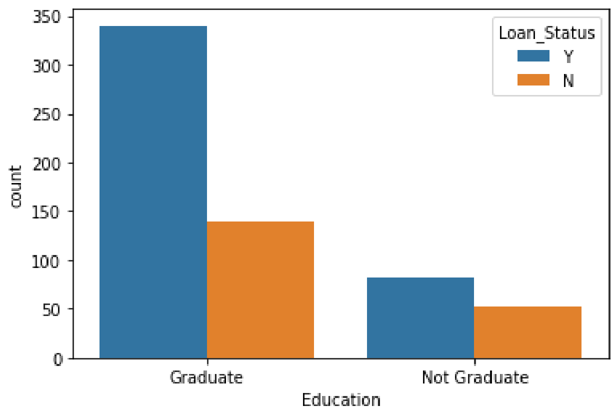
```
Dependents  Loan_Status
0           Y              0.686111
            N              0.313889
1           Y              0.647059
            N              0.352941
2           Y              0.752475
            N              0.247525
3           Y              0.647059
            N              0.352941
Name: Loan_Status, dtype: float64
```



No. Of Dependents vs Loan Status

The dependents feature indicates the number of features of the loan applicant. It has 4 unique values- 0, 1, 2 and 3. Majority of the population have zero dependents. The highest approval rate is for people having 2 dependents with 75%. The least approval rate is for people having 1 and more than or equal to 2 dependents with 64%. In people with only 0 dependents approval rate is 68 percent.

```
Education     Loan_Status
Graduate      Y              0.708333
              N              0.291667
Not Graduate  Y              0.611940
              N              0.388060
Name: Loan_Status, dtype: float64
```

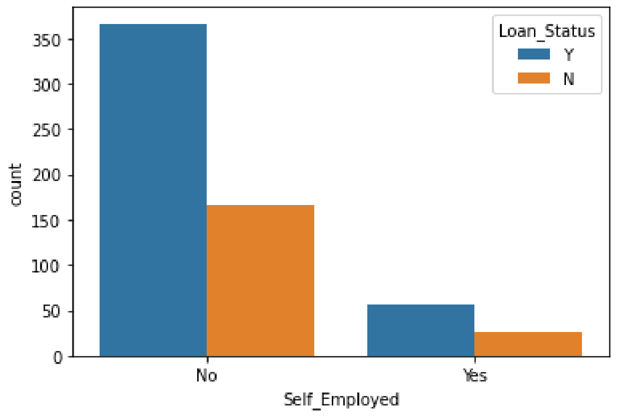


Education vs Loan Status

Education feature captures two values. One is Graduate, other is Not Graduate. More Graduates apply for loans.

As expected there is high approval rate for applicats who are graduates. The approval rate for graduates is 70% while for applicants who are not graduates the approval rate is 61 percent.

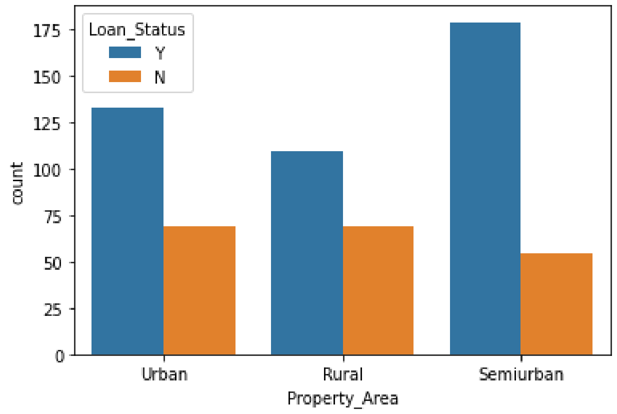
```
Self_Employed  Loan_Status
No             Y              0.687970
               N              0.312030
Yes            Y              0.682927
               N              0.317073
Name: Loan_Status, dtype: float64
```



Self Employed Vs Loan Status

Self employed feature captures whether the Loan Applicant is Self Employed or not. There are less self employed loan applicants. There is almost same approval rate whether the loan applicant is self employed or not.

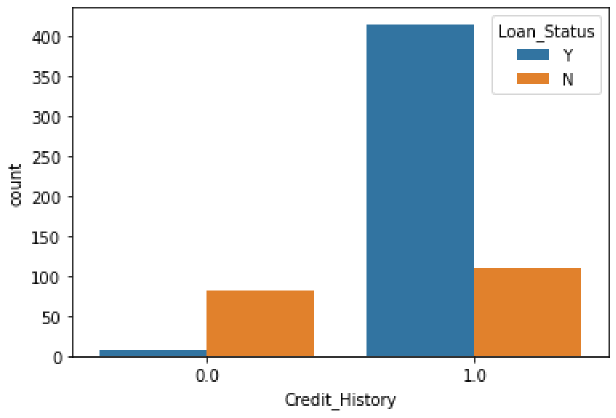
```
Property_Area  Loan_Status
Rural          Y              0.614525
               N              0.385475
Semiurban      Y              0.768240
               N              0.231760
Urban          Y              0.658416
               N              0.341584
Name: Loan_Status, dtype: float64
```



Property area vs Loan status

Property area captures the location of the property. It has 3 unique values – Urban, Rural, Semi urban. Most of the property areas are in Semi Urban area. Highest approval rate is for loan applicants with property area is Semi uban location with 76 percent approval rate, while the least approval rate is for loan applicants with property area in Rural location with 61 percent loan approval rate. And Loan applicants with property area in Urban location have approval rate of 65 percent.

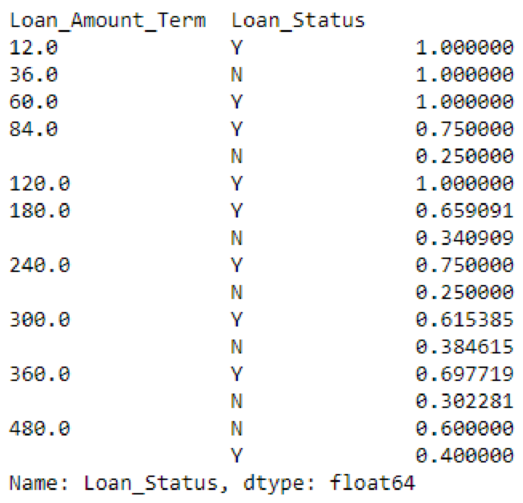
```
Credit_History  Loan_Status
0.0             N              0.921348
                Y              0.078652
1.0             Y              0.790476
                N              0.209524
Name: Loan_Status, dtype: float64
```



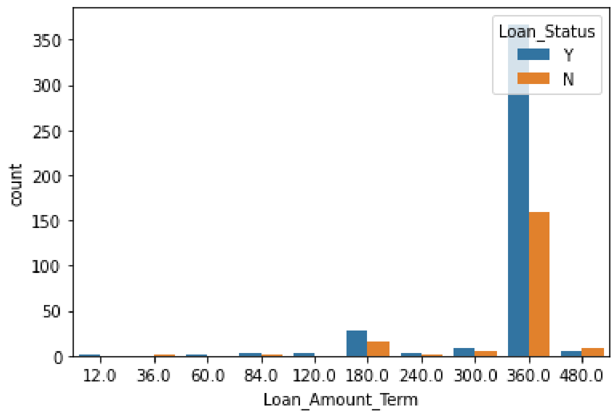
Credit History Vs Loan Status

Credit History feature indicates whether the person applying for a loan has any credit history in the past. It has 2 unique values 1 indicating that the person has a credit history and 0 indicating that the person does not have any credit history in the past.

There are large number of loan applications having credit history in the past. The highest approval rate is in applicants not having a credit history in the past with 92 percent, while in case of applicants with credit history in the past the approval rate is some what less with 79 percent.



```
Loan_Amount_Term  Loan_Status
12.0              Y              1.000000
36.0              N              1.000000
60.0              Y              1.000000
84.0              Y              0.750000
                  N              0.250000
120.0             Y              1.000000
180.0             Y              0.659091
                  N              0.340909
240.0             Y              0.750000
                  N              0.250000
300.0             Y              0.615385
                  N              0.384615
360.0             Y              0.697719
                  N              0.302281
480.0             N              0.600000
                  Y              0.400000
Name: Loan_Status, dtype: float64
```



Loan Amount Term vs Loan status.

Loan Amount term indicates the tenure during which the loan is repaid in months. There are some constant values for loan amount term like 12, 36, 60, 84, 120, 180, 240, 300, 360, 480.

Most of the loan applications are with 360 months that is 6 years of repayment term.

Highest approval rate is 100 percent for 12, 60, 120 months and least approval rate is for 36 months with 100 percent again.

```
df=pd.get_dummies(df.drop(['Loan_ID'],axis=1),drop_first=True)
```

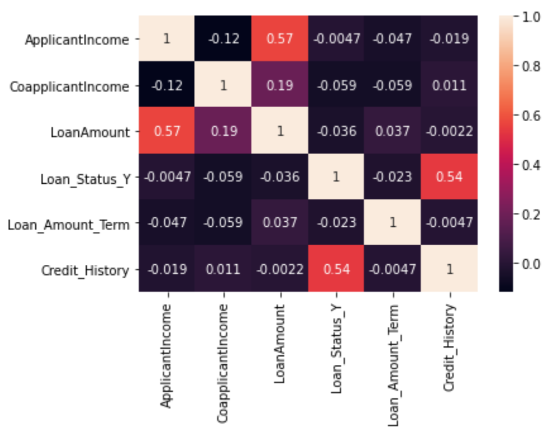
```
df.head()
```

	Dependents	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Gender_Male	Married_Yes	Education_Not Graduate	Self_Employed_Yes	Property_Area_Semiurban	Property_Area_Urban	Loan_Status_Y
0	0	5849	0.0	142.471735	360.0	1.0	1	0	0	0	0	1	1
1	1	4583	1508.0	128.000000	360.0	1.0	1	1	0	0	0	0	0
2	0	3000	0.0	66.000000	360.0	1.0	1	1	0	1	0	1	1
3	0	2583	2358.0	120.000000	360.0	1.0	1	1	1	0	0	1	1
4	0	6000	0.0	141.000000	360.0	1.0	1	0	0	0	0	1	1

Calling df = pd . get_dummies (df.drop ([‘Loan_ID'] , axis = 1) , drop_first = True) to transform all categorical features in multiple features with Yes = 1 or No = 0 values. And also passing drop_first as True so that it avoids multicollinearity problem.

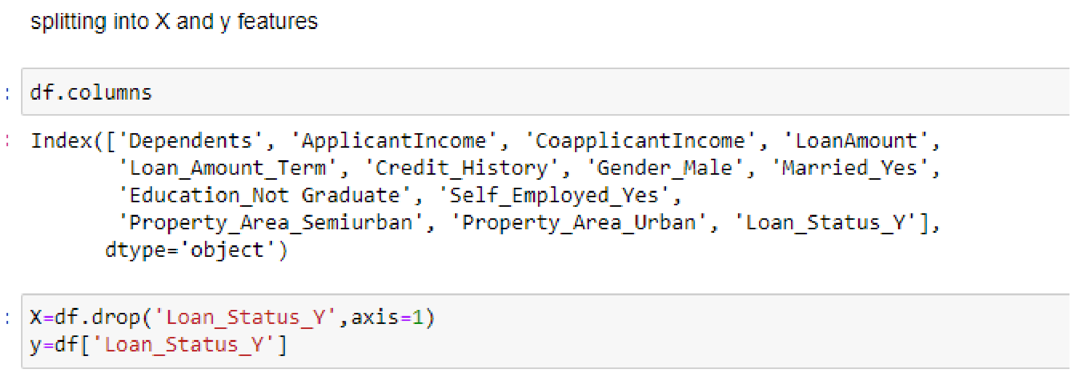
```
sns.heatmap(df[numerical_columns+['Loan_Status_Y']+['Loan_Amount_Term']+['Credit_History']].corr(),annot=True)
```

```
<AxesSubplot:>
```



Checking the corelation of the dataset.
Credit History is positively corelated with the Loan Status, while others are negatively corelated.

Splitting the data in to X and Y features:



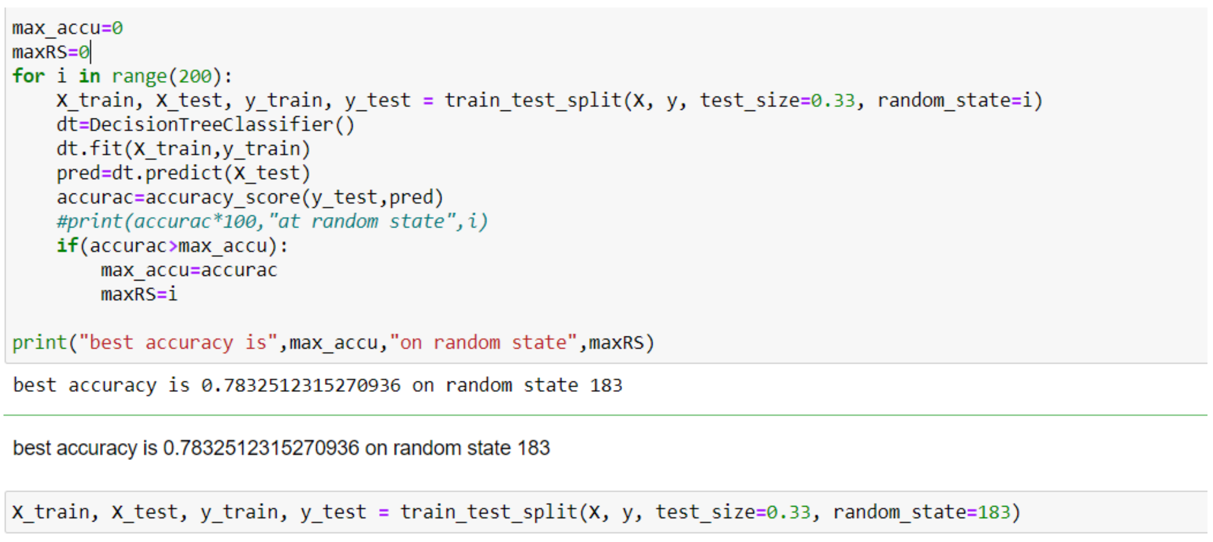
splitting into X and y features

```
df.columns
```

```
Index(['Dependents', 'ApplicantIncome', 'CoapplicantIncome', 'LoanAmount',
       'Loan_Amount_Term', 'Credit_History', 'Gender_Male', 'Married_Yes',
       'Education_Not Graduate', 'Self_Employed_Yes',
       'Property_Area_Semiurban', 'Property_Area_Urban', 'Loan_Status_Y'],
      dtype='object')
```

```
X=df.drop('Loan_Status_Y',axis=1)
y=df['Loan_Status_Y']
```

Splitting the data in to X (independent features) and y (dependent feature).



```
max_accu=0
maxRS=0
for i in range(200):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=i)
    dt=DecisionTreeClassifier()
    dt.fit(X_train,y_train)
    pred=dt.predict(X_test)
    accurac=accuracy_score(y_test,pred)
    #print(accurac*100,"at random state",i)
    if(accurac>max_accu):
        max_accu=accurac
        maxRS=i

print("best accuracy is",max_accu,"on random state",maxRS)
```

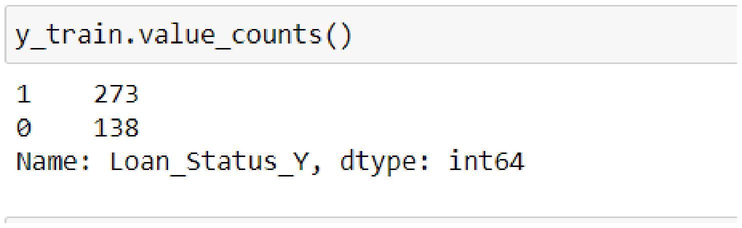
```
best accuracy is 0.7832512315270936 on random state 183
```

best accuracy is 0.7832512315270936 on random state 183

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=183)
```

Identifying the best random state by trying 200 values ranging from 0 to 199 for random state.

The best random state is 183 with an accuracy of 0.783



```
y_train.value_counts()
```

```
1    273
0    138
Name: Loan_Status_Y, dtype: int64
```

Checking the value counts of target variable, to identify any class imbalance issue.

Here, the count of 0 – loan not granted is 138 and count of 1 – loan approved is 273. So clearly there is class imbalance issue.

To overcome the class imbalance issue there are two approaches – Upsampling and Downsampling.

Upsampling:

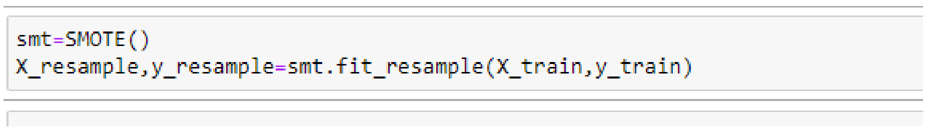
It is a procedure where synthetically generated data points (corresponding to minority class) are injected into the dataset. After this process, the counts of both labels are almost the same. This equalization procedure prevents the model from inclining towards the majority class. Furthermore, the interaction (boundary line) between the target classes remains unaltered. And also, the Upsampling mechanism introduces bias into the system because of the additional information.

Downsampling is a mechanism that reduces the count of training samples falling under the majority class. As it helps to even up the counts of target categories. By removing the collected data, we tend to lose so much valuable information.

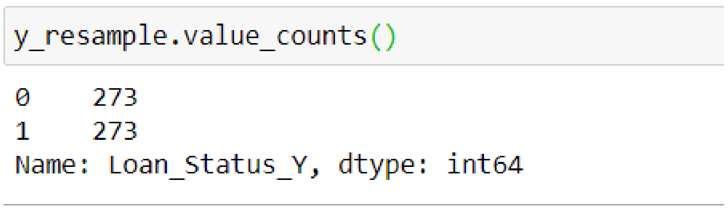
So it is better to do Upsampling and here we are applying SMOTE an implementation for Upsampling.

SMOTE (Synthetic Minority Oversampling Technique) — Upsampling :-

It works based on the K Nearest Neighbours algorithm, synthetically generating data points that fall in the proximity of the already existing outnumbered group. The input records should not contain any null values when applying this approach.



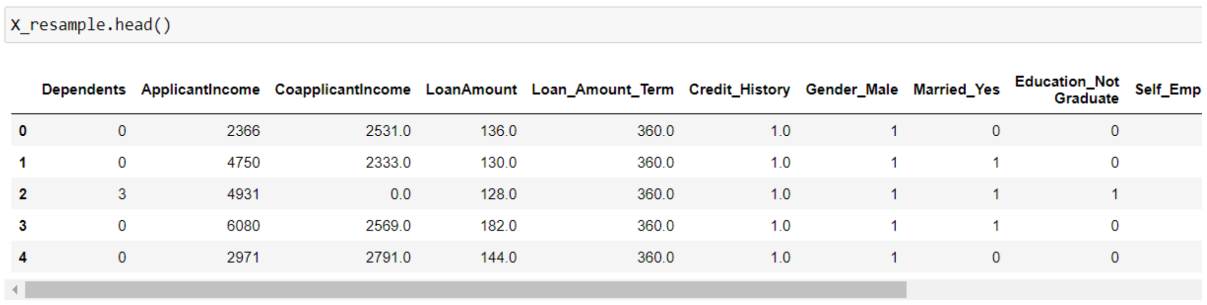
```
smt=SMOTE()
X_resample,y_resample=smt.fit_resample(X_train,y_train)
```



```
y_resample.value_counts()
```

```
0    273
1    273
Name: Loan_Status_Y, dtype: int64
```

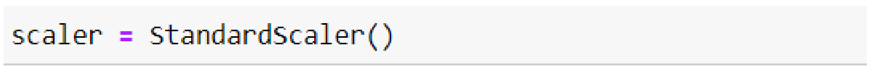
After applying SMOTE the training data is balanced with equal number of classes with each having 289 samples.



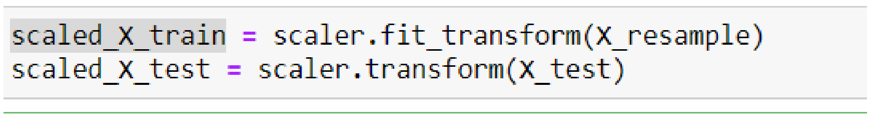
```
X_resample.head()
```

	Dependents	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Gender_Male	Married_Yes	Education_Not Graduate	Self_Emp
0	0	2366	2531.0	136.0	360.0	1.0	1	0	0	
1	0	4750	2333.0	130.0	360.0	1.0	1	1	0	
2	3	4931	0.0	128.0	360.0	1.0	1	1	1	
3	0	6080	2569.0	182.0	360.0	1.0	1	1	0	
4	0	2971	2791.0	144.0	360.0	1.0	1	0	0	

All the features' values in the training data are not in the same range. So fitting the



```
scaler = StandardScaler()
```

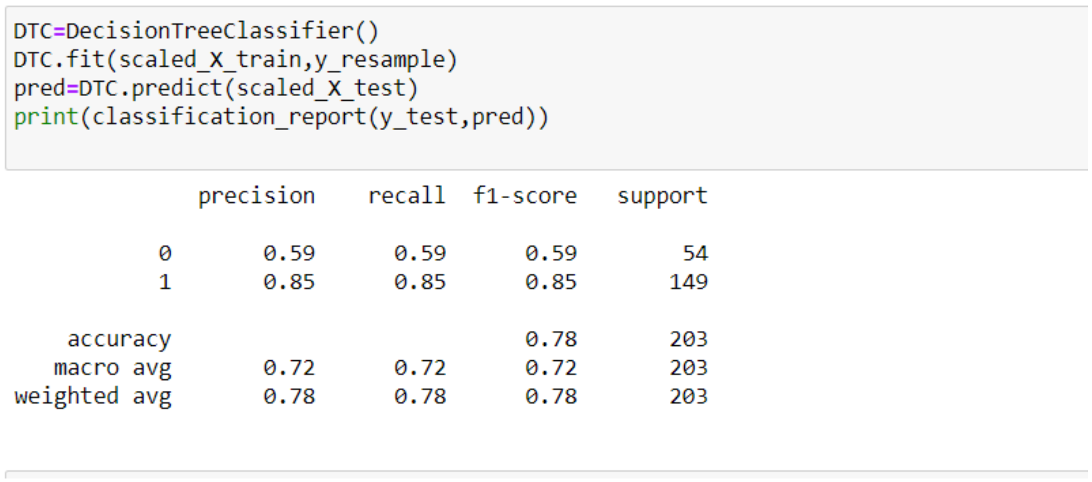


```
scaled_X_train = scaler.fit_transform(X_resample)
scaled_X_test = scaler.transform(X_test)
```

Standard Scaler on the training data and transforming the training and testing data. Once the standard scaler is applied on the training and testing set all the values will be in the same range.

Model Training:

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.



```
DTC=DecisionTreeClassifier()
DTC.fit(scaled_X_train,y_resample)
pred=DTC.predict(scaled_X_test)
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.59      0.59      0.59        54
           1       0.85      0.85      0.85       149

    accuracy                           0.78       203
   macro avg       0.72      0.72      0.72       203
weighted avg       0.78      0.78      0.78       203
```

The accuracy on test set with Decision Tree classifier is 78% and F1 scores are 59 and 85 for label 0 and label 1 respectively.

An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

```
ada=AdaBoostClassifier()
ada.fit(scaled_X_train,y_resample)
pred=ada.predict(scaled_X_test)
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.66      0.61      0.63        54
           1       0.86      0.89      0.87       149

    accuracy                           0.81       203
   macro avg       0.76      0.75      0.75       203
weighted avg       0.81      0.81      0.81       203
```

The accuracy achieved on test set with AdaBoost classifier is 81 percent and f1 scores are 63 and 87 on label 0 and 1 respectively.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the max_samples parameter if bootstrap=True (default), otherwise the whole dataset is used to build each tree.

```
RFC=RandomForestClassifier()
RFC.fit(scaled_X_train,y_resample)
pred=RFC.predict(scaled_X_test)
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.62      0.61      0.62        54
           1       0.86      0.87      0.86       149

    accuracy                           0.80       203
   macro avg       0.74      0.74      0.74       203
weighted avg       0.80      0.80      0.80       203
```

The accuracy achieved on test set with Random Forest classifier is 80 percent and f1 scores are 62 and 86 on label 0 and 1 respectively.

```
dtcAC=0.78
rfcAC=0.80
svcAC=0.78
adaAC=0.81
```

Checking the cross validation scores:

Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be ‘transforms’, that is, they must implement fit and transform methods. The final estimator only needs to implement fit. The transformers in the pipeline can be cached using memory argument.

```
imba_pipeline=make_pipeline(SMOTE(random_state=42),scaler,DTC)
DTCcv=cross_val_score(imba_pipeline,X_train,y_train,cv=5).mean()
DTCcv
```

```
0.6570672935645019
```

```
imba_pipeline=make_pipeline(SMOTE(random_state=42),scaler,RFC)
RFCcv=cross_val_score(imba_pipeline,X_train,y_train,cv=5).mean()
RFCcv
```

```
0.7178078166323832
```

```
imba_pipeline=make_pipeline(SMOTE(random_state=42),scaler,AdaBoostClassifier())
adaCV=cross_val_score(imba_pipeline,X_train,y_train,cv=5).mean()
adaCV
```

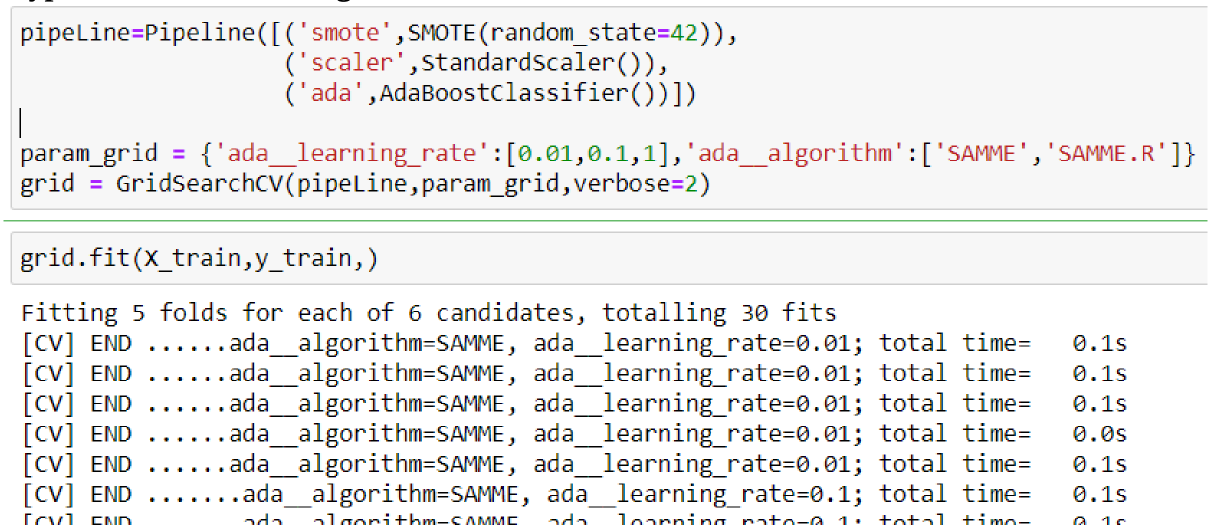
```
0.7421099030267412
```

```
print(dtcAC-DTCcv)
print(rfcAC-RFCcv)
print(adaAC-adaCV)
```

```
0.12293270643549814
0.08219218333676168
0.06789009697325887
```

AdaBoost Classifier is the best model as it has the highest precision and recall and accuracy as well. Ada Boost classifier has the less difference between the accuracy scores(on test set) and Cross Val score(on validation), so it is the best model.

Hyper Parameter Tuning - AdaBoost Classifier:



```
pipeLine=Pipeline([('smote',SMOTE(random_state=42)),
                   ('scaler',StandardScaler()),
                   ('ada',AdaBoostClassifier())])

param_grid = {'ada__learning_rate':[0.01,0.1,1],'ada__algorithm':['SAMME','SAMME.R']}
grid = GridSearchCV(pipeLine,param_grid,verbose=2)
```

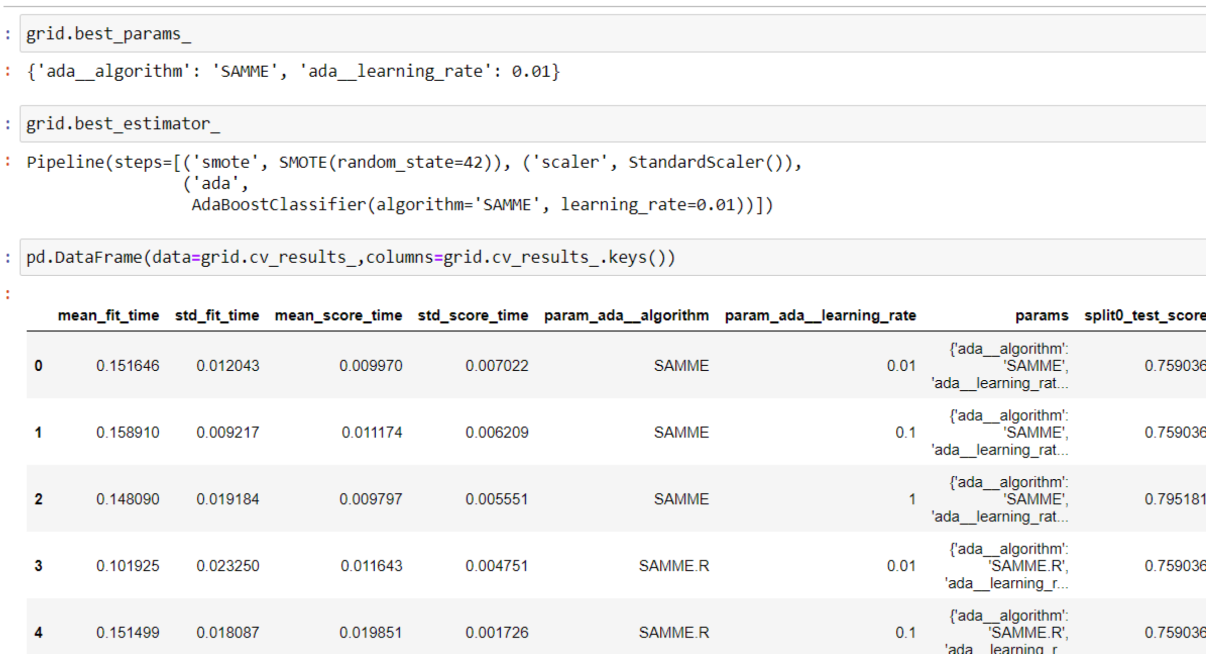
```
grid.fit(X_train,y_train,)
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
[CV] END ......ada__algorithm=SAMME, ada__learning_rate=0.01; total time=   0.1s
[CV] END ......ada__algorithm=SAMME, ada__learning_rate=0.01; total time=   0.1s
[CV] END ......ada__algorithm=SAMME, ada__learning_rate=0.01; total time=   0.1s
[CV] END ......ada__algorithm=SAMME, ada__learning_rate=0.01; total time=   0.0s
[CV] END ......ada__algorithm=SAMME, ada__learning_rate=0.01; total time=   0.1s
[CV] END .......ada__algorithm=SAMME, ada__learning_rate=0.1; total time=   0.1s
[CV] END .......ada__algorithm=SAMME, ada__learning_rate=0.1; total time=   0.1s
```

Hyperparameters are the variables that the user specify usually while building the Machine Learning model. Thus, hyperparameters are specified before specifying the parameters or we can say that hyperparameters are used to evaluate optimal parameters of the model. the best part about hyperparameters is that their values are decided by the user who is building the model. For example, max_depth in Random Forest Algorithms, k in KNN Classifier.

Grid Search uses a different combination of all the specified hyperparameters and their values and calculates the performance for each combination and selects the best value for the hyperparameters. This makes the processing time-consuming and expensive based on the number of hyperparameters involved.

In GridSearchCV, along with Grid Search, cross-validation is also performed. Cross-Validation is used while training the model. As we know that before training the model with data, we divide the data into two parts – train data and test data. In cross-validation, the process divides the train data further into two parts – the train data and the validation data.



```
: grid.best_params_
```

```
: {'ada__algorithm': 'SAMME', 'ada__learning_rate': 0.01}
```

```
: grid.best_estimator_
```

```
: Pipeline(steps=[('smote', SMOTE(random_state=42)), ('scaler', StandardScaler()),
                  ('ada',
                   AdaBoostClassifier(algorithm='SAMME', learning_rate=0.01))])
```

```
: pd.DataFrame(data=grid.cv_results_,columns=grid.cv_results_.keys())
```

:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_ada__algorithm	param_ada__learning_rate	params	split0_test_score
0	0.151646	0.012043	0.009970	0.007022	SAMME	0.01	{'ada__algorithm': 'SAMME', 'ada__learning_rat...	0.759036
1	0.158910	0.009217	0.011174	0.006209	SAMME	0.1	{'ada__algorithm': 'SAMME', 'ada__learning_rat...	0.759036
2	0.148090	0.019184	0.009797	0.005551	SAMME	1	{'ada__algorithm': 'SAMME', 'ada__learning_rat...	0.795181
3	0.101925	0.023250	0.011643	0.004751	SAMME.R	0.01	{'ada__algorithm': 'SAMME.R', 'ada__learning_r...	0.759036
4	0.151499	0.018087	0.019851	0.001726	SAMME.R	0.1	{'ada__algorithm': 'SAMME.R', 'ada__learning_r	0.759036

Training the final model with the best parameters based on GridSearchCV:

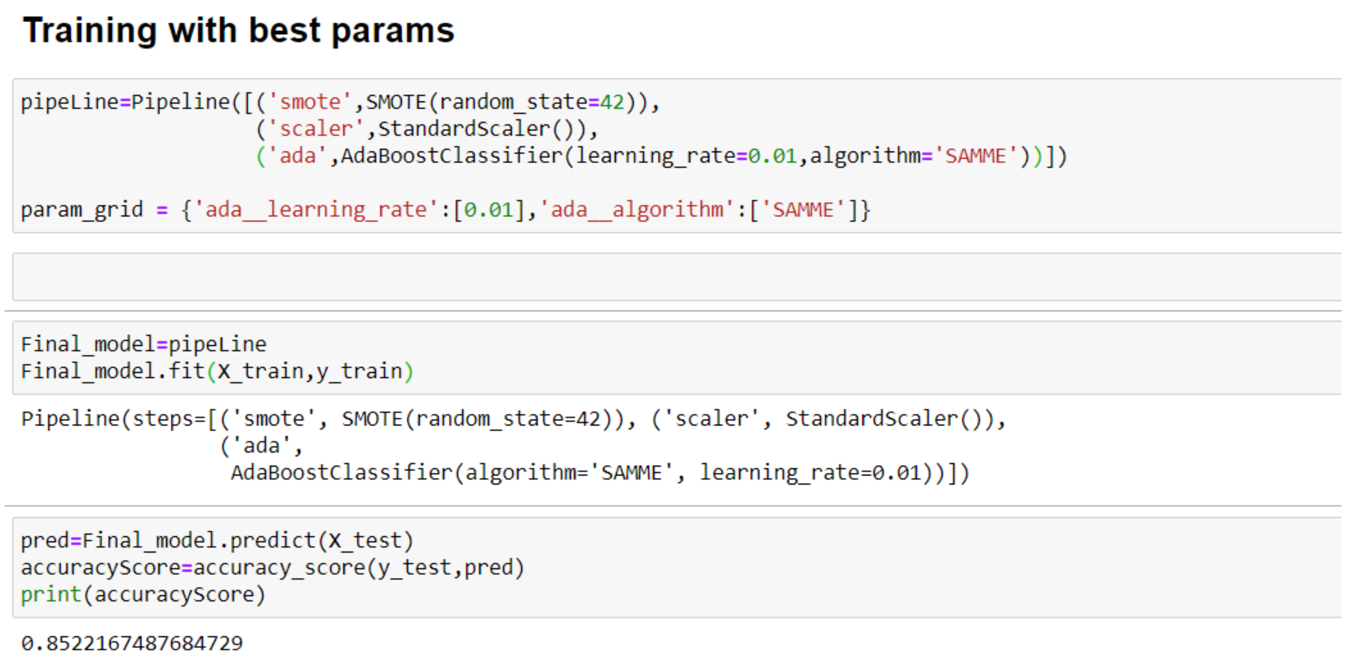
Creating a pipeline with Smote for Upsampling the training data, scaler – fitting the scaler on training data, transforming the training and testing data, AdaBoostClassifier with learning rate of 0.01 and algorithm as SAMME.

```
grid.best_params_
```

```
{'ada__algorithm': 'SAMME', 'ada__learning_rate': 0.01}
```

```
grid.best_estimator_
```

```
Pipeline(steps=[('smote', SMOTE(random_state=42)), ('scaler', StandardScaler()),
                ('ada',
                 AdaBoostClassifier(algorithm='SAMME', learning_rate=0.01))])
```



Training with best params

```
pipeLine=Pipeline([('smote',SMOTE(random_state=42)),
                   ('scaler',StandardScaler()),
                   ('ada',AdaBoostClassifier(learning_rate=0.01,algorithm='SAMME'))])

param_grid = {'ada__learning_rate':[0.01],'ada__algorithm':['SAMME']}
```

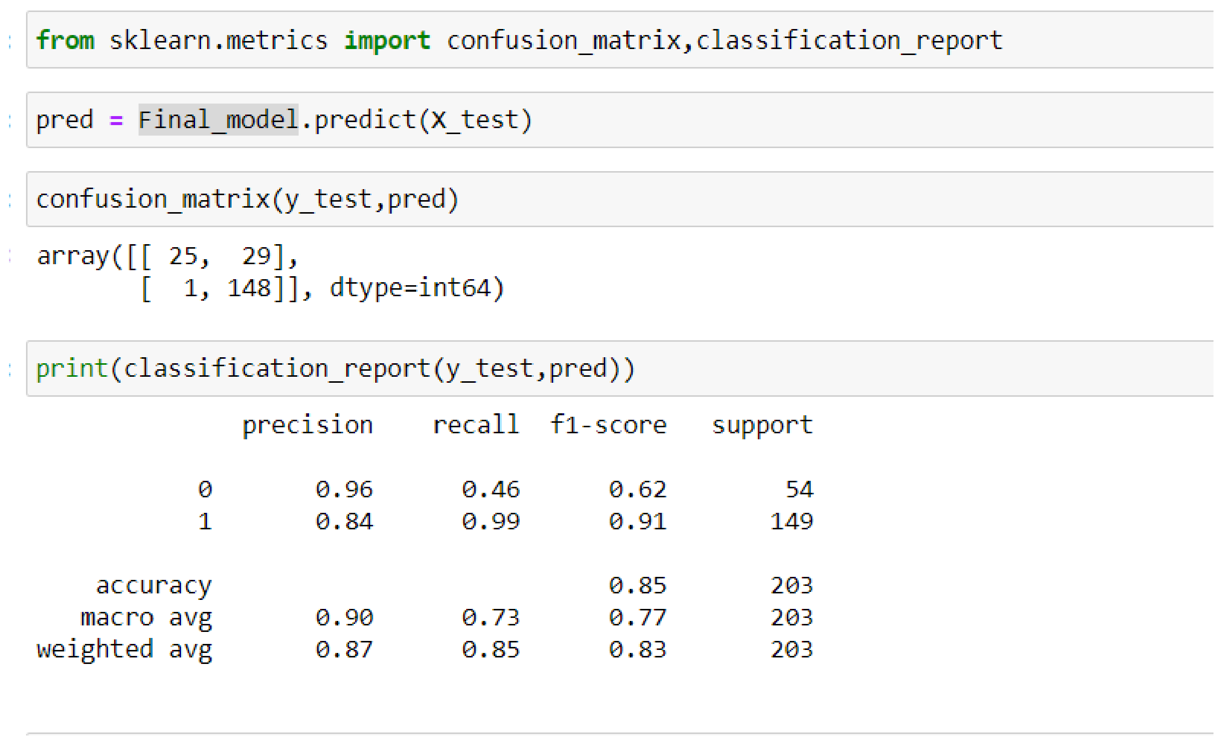
```
Final_model=pipeLine
Final_model.fit(X_train,y_train)
```

```
Pipeline(steps=[('smote', SMOTE(random_state=42)), ('scaler', StandardScaler()),
                ('ada',
                 AdaBoostClassifier(algorithm='SAMME', learning_rate=0.01))])
```

```
pred=Final_model.predict(X_test)
accuracyScore=accuracy_score(y_test,pred)
print(accuracyScore)
```

```
0.8522167487684729
```

After training the model on the training data with the best parameters , accuracy of 85% is achieved on the testing data.



```
from sklearn.metrics import confusion_matrix,classification_report
```

```
pred = Final_model.predict(X_test)
```

```
confusion_matrix(y_test,pred)
```

```
array([[ 25,  29],
       [  1, 148]], dtype=int64)
```

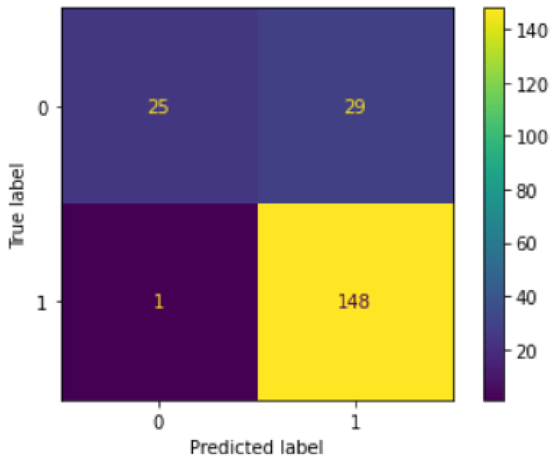
```
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.96      0.46      0.62        54
           1       0.84      0.99      0.91       149

    accuracy                           0.85       203
   macro avg       0.90      0.73      0.77       203
weighted avg       0.87      0.85      0.83       203
```

Classification report shows the f1 scores 0.62 and 0.91 on label 0 and label 1 respectively.

```
plot_confusion_matrix(Final_model,X_test,y_test)
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x24ee6a6cbe0>
```



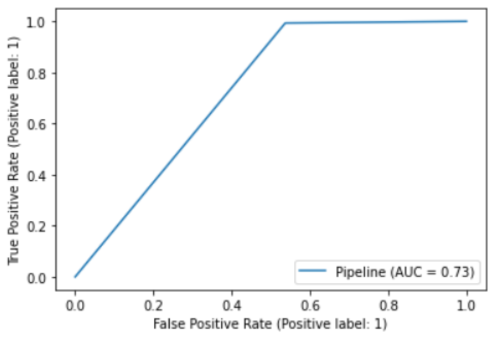
Confusion matrix reveals that out of 149 records from class1, 148 are correctly predicted as class 1 and on record is falsely predicted as class 0. Out of 54 records in class 0, 25 are correctly predicted as class 0 and 29 records are incorrectly classified as class 1.

A receiver operating characteristic (ROC), or simply ROC curve, is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting the fraction of true positives out of the positives (TPR = true positive rate) vs. the fraction of false positives out of the negatives (FPR = false positive rate), at various threshold settings. TPR is also known as sensitivity, and FPR is one minus the specificity or true negative rate.

The roc_auc_score function computes the area under the receiver operating characteristic (ROC) curve, which is also denoted by AUC or AUROC. By computing the area under the roc curve, the curve information is summarized in one number.

```
plot_roc_curve(Final_model,X_test,y_test)
```

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x24ee69ec4e0>
```



AUC-ROC Curve area is 0.73 which is good.

In Conclusion, the AdaBoost classifier is chosen as the best model and the accuracy is 85%.

The project can be found at
https://github.com/sk1930/Projects/blob/main/Loan%20Application%20Status%20Prediction
.ipynb