Sri Lanka Institute of Information Technology

Fundamentals Of Data Mining (IT3051)

Continuous Assignment – 2023, Semester 2

Final Report – Mini Project

Group No - G10

Arawwala D.J.S.S.	IT21053528
Weerasinghe C.C.	IT20229016
Kasthurirathne K.K.I.	IT21077692
Samaraweera G.P.M.D.	IT21016066

Repository: https://github.com/KavinduKasthurirathne/Loan-Approval-Prediction-Model.git

Interface Link: https://loan-status.streamlit.app/

Video Presentation: Loan Status predictor - Made with Clipchamp.mp4

Table of Contents

Introduction

In the world of finance, loan approval is a pivotal process for both borrowers seeking financial assistance and financial institutions aiming to manage risk and grow their business. Traditionally, this process has been largely manual, involving a comprehensive evaluation of an applicant's financial history, credit score, income, employment status, and many other factors. However, with the advent of advanced technologies and the abundance of data available, financial institutions are increasingly turning to predictive modeling to transform and optimize this critical procedure.

Predictive modeling for loan approval offers a host of advantages, foremost among them being the potential for increased efficiency. By leveraging data-driven techniques and algorithms, financial institutions can streamline their loan approval processes, reducing the time and effort required for manual assessments. This not only expedites the decision-making process but also enhances the overall customer experience.

Furthermore, predictive models have the potential to improve fairness in loan approval. Traditional methods may inadvertently introduce bias or subjectivity into the decision-making process, which can result in disparities and discrimination. Predictive models, when developed and deployed responsibly, can help mitigate these issues by providing consistent, data-driven decisions, thereby promoting fairness and equality in lending.

Another compelling aspect of predictive modeling for loan approval is the potential to enhance the accuracy of credit risk assessments. By analyzing a broader range of data and considering various factors, predictive models can provide a more comprehensive view of an applicant's creditworthiness. This improved accuracy not only helps financial institutions make better lending decisions but also reduces the risk of defaults, ultimately benefiting both lenders and borrowers.

This introduction sets the stage for a comprehensive exploration of predictive modeling for loan approval. In the subsequent discussion, we will delve into the key components of building such models, including data collection, feature engineering, model selection, and validation. We will also examine the critical issue of fairness in lending and the responsible use of predictive models to ensure equitable access to financial services. Finally, we will explore the potential challenges and ethical considerations that must be addressed in the development and deployment of these models in the financial industry.

Dataset Features

loan_approval_dataset. – https://www.kaggle.com/datasets/architsharma01/loan-approval-prediction-dataset

Features	Description
loan_id	Unique loan id
no_of_dependents	Number of dependents of the applicant
education	The education level of the applicant
self_employed	If the applicant is self-employed or not
income_annum	Annual income of the applicant
loan_amount	Loan amount requested by the applicant
loan_tenure	Tenure of the loan requested by the applicant (in years)
cibil_score	CIBIL score of the applicant
residential_asset_value	Value of the residential asset of the applicant
commercial_asset_value	Value of the commercial asset of the applicant
luxury_asset_value	Value of the luxury asset of the applicant
bank_asset_value	Value of the bank asset of the applicant
loan_status	Status of the loan (Approved/Rejected)

Data Collection

Importing Packages

[75]:
```
import pandas as pd
import numpy as np
import pickle
```

1. Data Collection

[76]:
```
df = pd.read_csv("loan_approval_dataset.csv")
```

Get 5 rows of samples of the dataframe

[77]:
```
df.sample(5)
```

[77]:

	loan_id	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term	cibil_score	residential_assets_value	commercial_assets_value	luxu
3722	3723	3	Graduate	No	7000000	21700000	14	781	5400000	8500000	
26	27	4	Graduate	No	8200000	28100000	12	696	11500000	10600000	
3365	3366	3	Not Graduate	Yes	7800000	24400000	20	490	6700000	5100000	
2296	2297	4	Graduate	No	7000000	25700000	20	597	19900000	13500000	
3280	3281	1	Graduate	No	7400000	26600000	18	505	2200000	2400000	

Feature Engineering

2. Feature Engineering

Drop irrelavant columns

[78]:
```
columns_to_remove = ['loan_id']

# Remove the specified columns
df.drop(columns=columns_to_remove, inplace=True)
```

[79]:
```
# No. of rows & Columns in the dataframe (shape)
print("Dataset Shape:", df.shape)
```

```
Dataset Shape: (4269, 12)
```

[80]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4269 entries, 0 to 4268
Data columns (total 12 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0    no_of_dependents          4269 non-null   int64
 1    education                 4269 non-null   object
 2    self_employed             4269 non-null   object
 3    income_annum              4269 non-null   int64
 4    loan_amount               4269 non-null   int64
 5    loan_term                 4269 non-null   int64
 6    cibil_score               4269 non-null   int64
 7    residential_assets_value  4269 non-null   int64
 8    commercial_assets_value   4269 non-null   int64
 9    luxury_assets_value       4269 non-null   int64
 10   bank_asset_value          4269 non-null   int64
 11   loan_status               4269 non-null   object
dtypes: int64(9), object(3)
memory usage: 400.3+ KB
```

[81]:
```
# Descriptive statistics of the DataFrame
df.describe()
```

[81]:

	no_of_dependents	income_annum	loan_amount	loan_term	cibil_score	residential_assets_value	commercial_assets_value	luxury_assets_value	bank_asset_valu
count	4269.000000	4.269000e+03	4.269000e+03	4269.000000	4269.000000	4.269000e+03	4.269000e+03	4.269000e+03	4.269000e+0
mean	2.498712	5.059124e+06	1.513345e+07	10.900445	599.936051	7.472617e+06	4.973155e+06	1.512631e+07	4.976692e+0
std	1.695910	2.806840e+06	9.043363e+06	5.709187	172.430401	6.503637e+06	4.388966e+06	9.103754e+06	3.250185e+0
min	0.000000	2.000000e+05	3.000000e+05	2.000000	300.000000	-1.000000e+05	0.000000e+00	3.000000e+05	0.000000e+0
25%	1.000000	2.700000e+06	7.700000e+06	6.000000	453.000000	2.200000e+06	1.300000e+06	7.500000e+06	2.300000e+0
50%	3.000000	5.100000e+06	1.450000e+07	10.000000	600.000000	5.600000e+06	3.700000e+06	1.460000e+07	4.600000e+0
75%	4.000000	7.500000e+06	2.150000e+07	16.000000	748.000000	1.130000e+07	7.600000e+06	2.170000e+07	7.100000e+0
max	5.000000	9.900000e+06	3.950000e+07	20.000000	900.000000	2.910000e+07	1.940000e+07	3.920000e+07	1.470000e+0

Finding the unique values

[82]:
```
def uniquevals(col):
    print(f'Unique Values in {col} is : {df[col].unique()}')

for col in df.columns:
    uniquevals(col)
    print("-"*75)
```

```
Unique Values in  no_of_dependents is : [2 0 3 5 4 1]
---------------------------------------------------------------------------
Unique Values in  education is : [' Graduate' ' Not Graduate']
---------------------------------------------------------------------------
Unique Values in  self_employed is : [' No' ' Yes']
---------------------------------------------------------------------------
Unique Values in  income_annum is : [9600000 4100000 9100000 8200000 9800000 4800000 8700000 5700000  800000
 1100000 2900000 6700000 5000000 1900000 4700000  500000 2700000 6300000
 5800000 6500000 4900000 3100000 2400000 7000000 9000000 8400000 1700000
 1600000 8000000 3600000 1500000 7800000 1400000 4200000 5500000 9500000
 7300000 3800000 5100000 4300000 9300000 7400000 8500000 8800000 3300000
 3900000 8300000 5600000 5300000 2600000  700000 3500000 9900000 3000000
 6800000 2000000 1000000  300000 6600000 9400000 4400000  400000 6200000
 9700000 7100000  600000 7200000  900000  200000 1800000 4600000 2200000
 2500000 8600000 4000000 5200000 8900000 1300000 4500000 8100000 9200000
 2800000 7500000 6400000 6900000 7700000 3200000 7900000 5900000 3400000
 2100000 3700000 5400000 2300000 7600000 6000000 6100000 1200000]
---------------------------------------------------------------------------
```

Checking categorical and numerical features

[83]:
```
# Select all categorical data type and stored in one dataframe and select all other categorical and stored in one data frame
cat_var = df.select_dtypes(include=['object']).columns
num_var = df.select_dtypes(include = ['int32','int64','float32','float64']).columns

cat_var ,num_var
```

[83]:
```
(Index([' education', ' self_employed', ' loan_status'], dtype='object'),
 Index([' no_of_dependents', ' income_annum', ' loan_amount', ' loan_term',
        ' cibil_score', ' residential_assets_value', ' commercial_assets_value',
        ' luxury_assets_value', ' bank_asset_value'],
       dtype='object'))
```

Data Cleaning

3. Data Cleaning

Checking for duplicate rows

[84]:
```
# Check for duplicates based on all columns
duplicates_all = df[df.duplicated()]

# Print the results
print("Duplicates based on all columns:")
print(duplicates_all)
```

```
Duplicates based on all columns:
Empty DataFrame
Columns: [ no_of_dependents,  education,  self_employed,  income_annum,  loan_amount,  loan_term,  cibil_score,  residential_assets_value,  commercial_as
sets_value,  luxury_assets_value,  bank_asset_value,  loan_status]
Index: []
```

> It is clear that there are no duplicates

Handling null values

[85]:
```
df.isna().sum()
```

[85]:
```
 no_of_dependents            0
 education                   0
 self_employed               0
 income_annum                0
 loan_amount                 0
 loan_term                   0
 cibil_score                 0
 residential_assets_value    0
 commercial_assets_value     0
 luxury_assets_value         0
 bank_asset_value            0
 loan_status                 0
dtype: int64
```

Counting zeros of each column

[86]:
```
# Count zeros in each column
zero_counts = (df == 0).sum()

print(zero_counts)
```

```
 no_of_dependents            712
 education                     0
 self_employed                 0
 income_annum                  0
 loan_amount                   0
 loan_term                     0
 cibil_score                   0
 residential_assets_value     45
 commercial_assets_value     107
 luxury_assets_value           0
 bank_asset_value              8
 loan_status                   0
dtype: int64
```

Assuming that `residential_assets_value`, `commercial_assets_value`, and `bank_asset_value` cannot be zero, we treat zero values as null values and replace them with the respective column mean. However, it's noteworthy that this process does not take into account the variable `no_of_dependents`.

[87]:
```
# Calculate means for the columns
mean_residential_assets = df[' residential_assets_value'].mean()
mean_commercial_assets = df[' commercial_assets_value'].mean()
mean_bank_assets = df[' bank_asset_value'].mean()

# Replace zeros with means in the specified columns
df[' residential_assets_value'].replace(0, mean_residential_assets, inplace=True)
df[' commercial_assets_value'].replace(0, mean_commercial_assets, inplace=True)
df[' bank_asset_value'].replace(0, mean_bank_assets, inplace=True)
```

The DataFrame 'df' now reflects the replacement of zeros in the specified columns with their respective means

[88]:
```
(df == 0).sum()
```

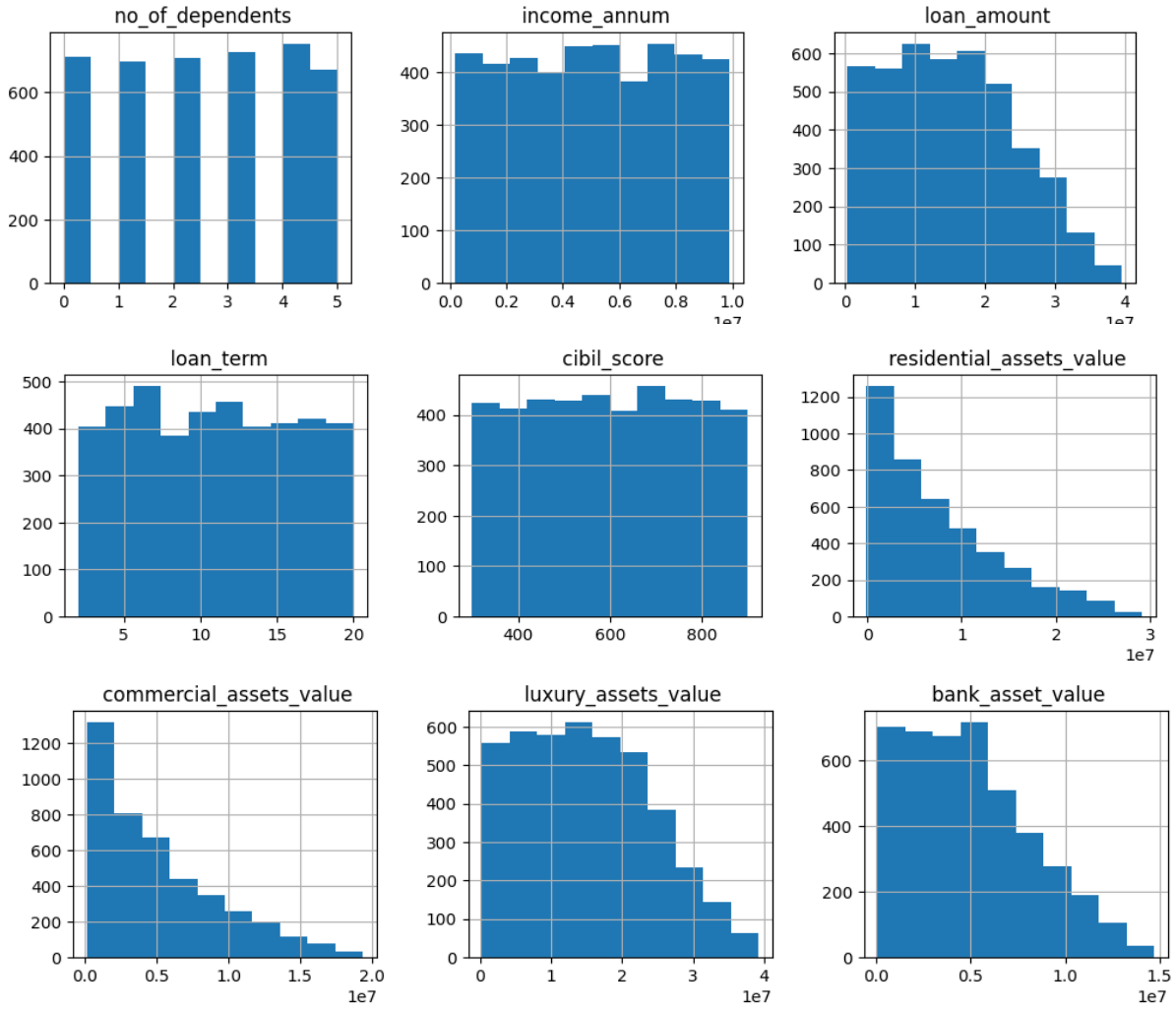
[88]:
```
no_of_dependents            712
education                     0
self_employed                 0
income_annum                  0
loan_amount                   0
loan_term                     0
cibil_score                   0
residential_assets_value      0
commercial_assets_value       0
luxury_assets_value           0
bank_asset_value              0
loan_status                   0
dtype: int64
```

Exploratory Data Analysis

4. Exploratory Data Analysis

[89]:
```
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns

import warnings
# Ignore FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

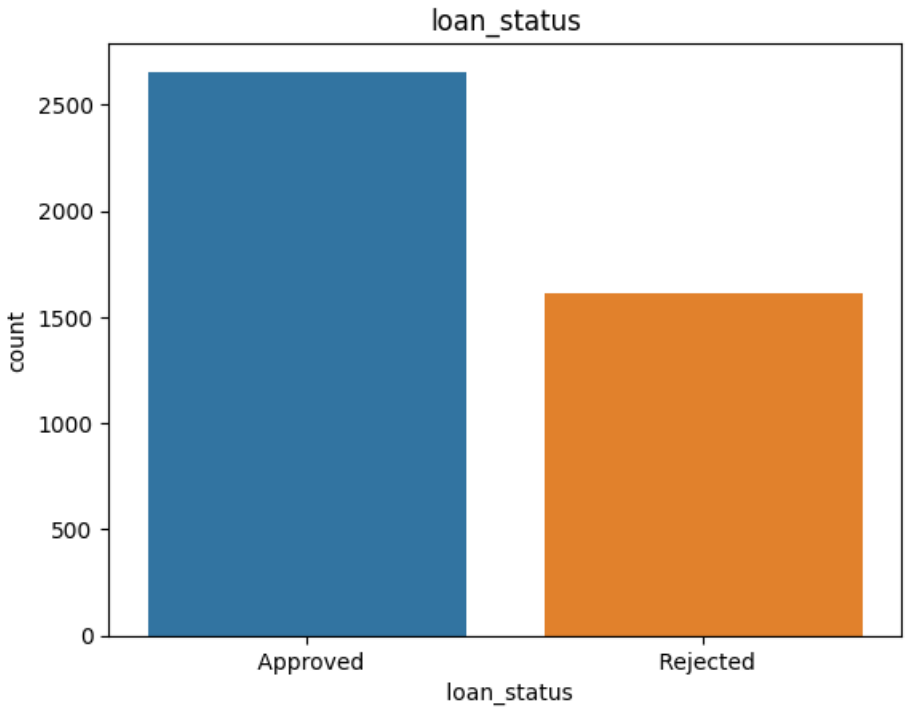
[90]:
```
plot = df.hist(figsize=(12,10))
```



Loan Status Distribution

```
[91]: sns.countplot(x = ' loan_status', data = df).set_title('loan_status')
```

```
[91]: Text(0.5, 1.0, 'loan_status')
```

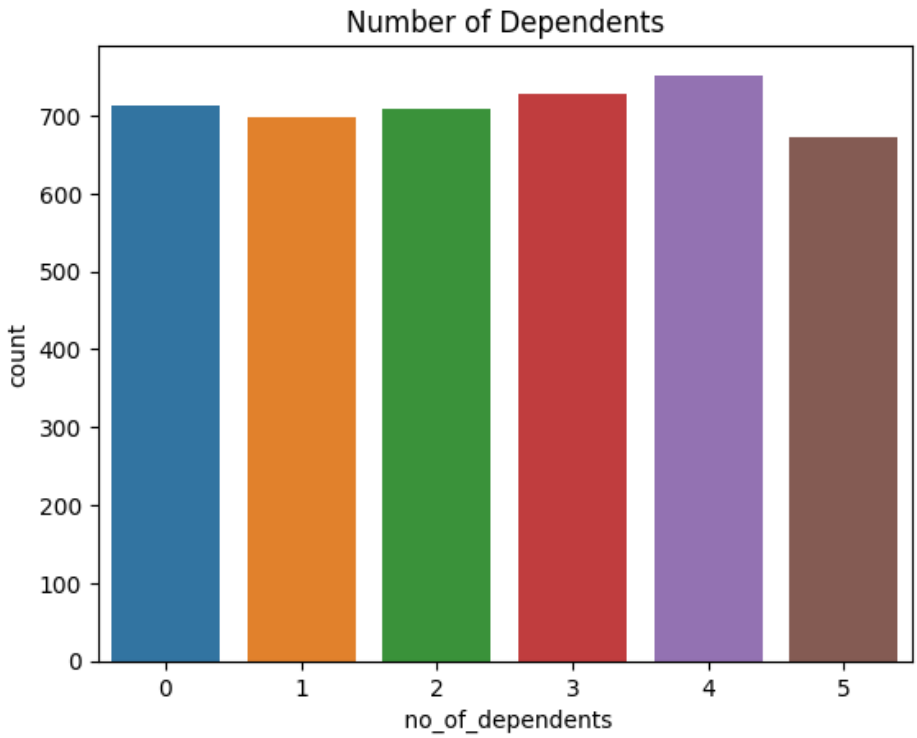


The dataset shows that there are more 'Approved' instances than 'Rejected', indicating an imbalance.

Number Of Dependents Distribution

```
[92]: sns.countplot(x = ' no_of_dependents', data = df).set_title('Number of Dependents')
```

```
[92]: Text(0.5, 1.0, 'Number of Dependents')
```

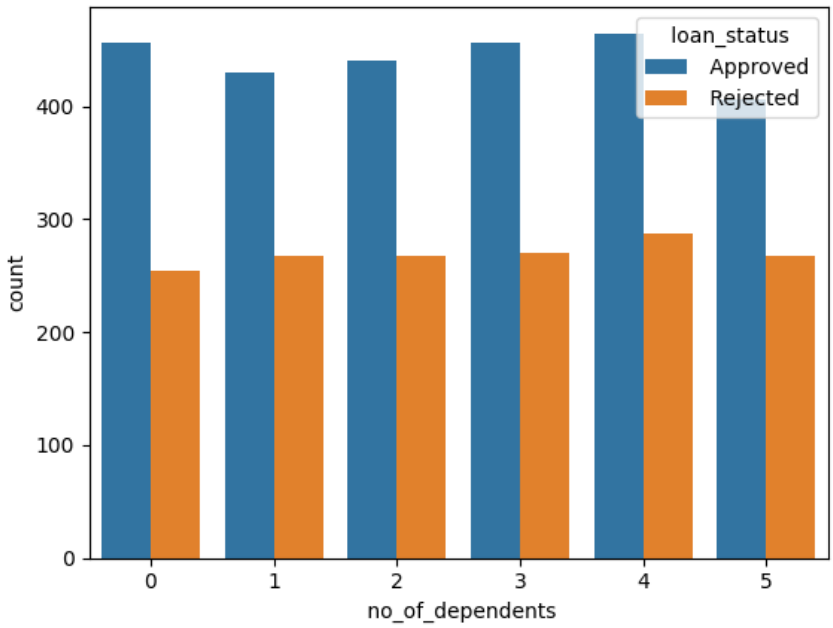


The chart shows the number of dependents for loan applicants, highlighting a clear difference in living arrangements. Notably, there isn't a substantial variance in the count of dependents. However, it's worth noting that as the number of dependents increases, the disposable income of the applicant tends to decrease. Consequently, I hypothesize that applicants with 0 or 1 dependent might have higher chances of loan approval.

Number of Dependants Vs Loan Status

```
[93]: sns.countplot(x = ' no_of_dependents', data = df, hue = ' loan_status')
```

```
[93]: <Axes: xlabel=' no_of_dependents', ylabel='count'>
```

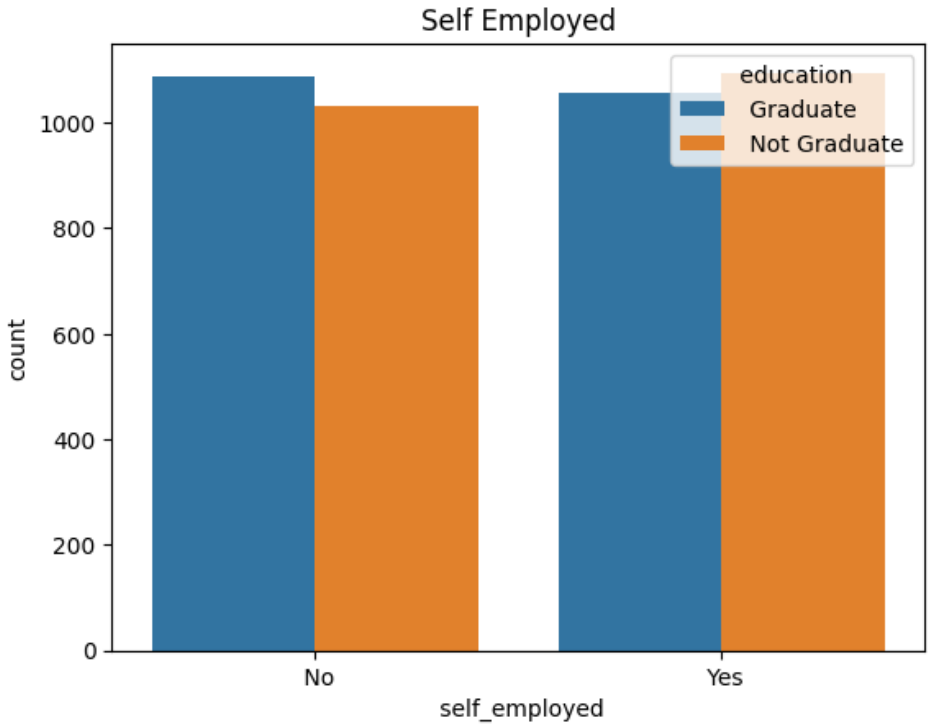


The graph implies that having more dependents is associated with a higher chance of loan rejection. However, it's noteworthy that the number of approved loans doesn't significantly change, even for individuals with more family members. This challenges the idea that loans are less likely to be approved for those with more dependents. The key takeaway is the importance of relying on observed data rather than assumptions, as the real outcomes may differ from what was initially expected.

Education and Self Employed

```
[94]: sns.countplot(x=' self_employed', data = df, hue = ' education').set_title('Self Employed')
```

```
[94]: Text(0.5, 1.0, 'Self Employed')
```

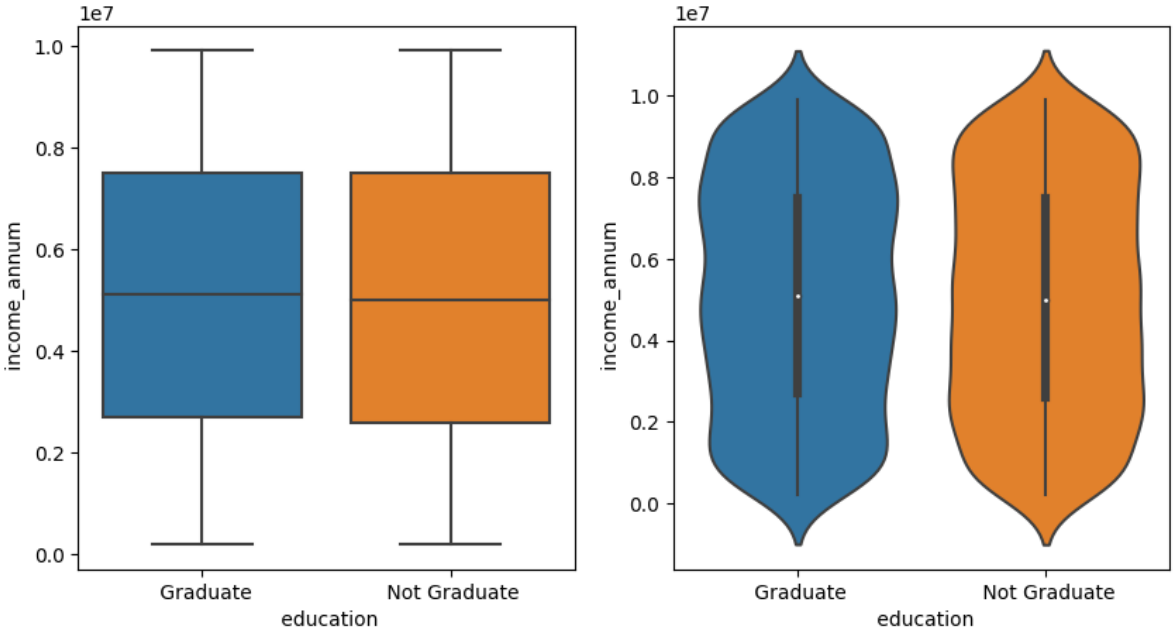


The graph depicting the relationship between the employment status of applicants and their education levels highlights important trends for loan approval considerations. It reveals that a majority of non-graduate applicants are self-employed, while most graduate applicants are not self-employed. This indicates that graduates are more likely to be employed in salaried positions, whereas non-graduates tend to be self-employed.

Education and Income

```
[95]: fig, ax = plt.subplots(1,2,figsize=(10, 5))
      sns.boxplot(x = ' education', y = ' income_annum', data = df, ax=ax[0])
      sns.violinplot(x = ' education', y = ' income_annum', data = df, ax=ax[1])
```

```
[95]: <Axes: xlabel=' education', ylabel=' income_annum'>
```



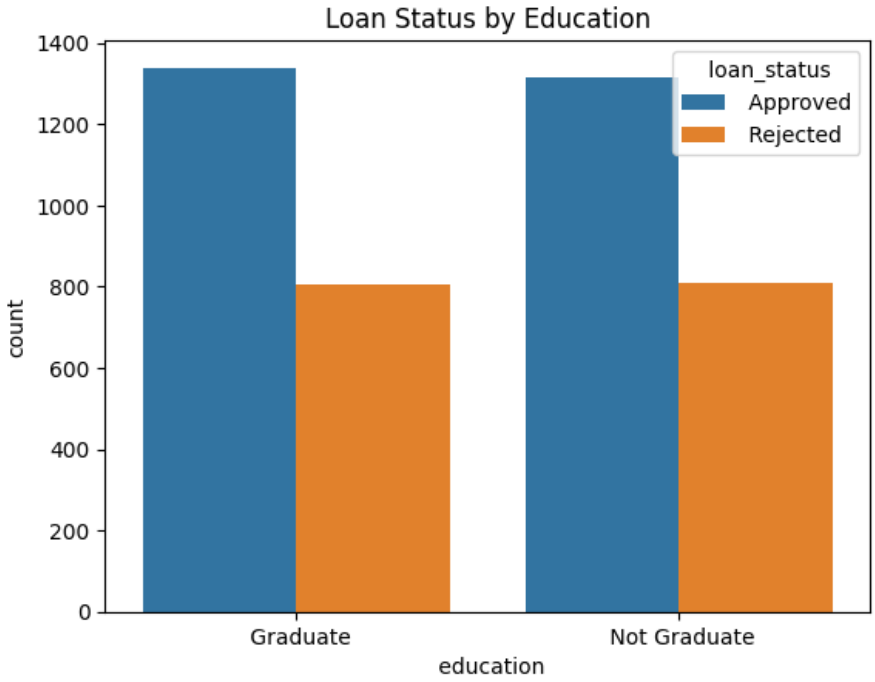
The combination of boxplot and violin plot visualizations provides insights into the relationship between the education levels of loan applicants and their annual incomes. The boxplot reveals that both graduates and non-graduates have similar median incomes, indicating that having a degree doesn't necessarily lead to a significant income advantage.

Moreover, the violin plot shows the distribution of income among the graduates and non-graduate applicants, where we can see that nongraduate applicants have an even distribution between income 2000000 and 8000000, whereas there is an uneven distribution among the graduates with more applicants having income between 6000000 and 8000000 Since there is not much change in annual income of graduates and non-graduates, the assumption is that education does not play a major role in the approval of the loan.

Education Vs Loan Status

```
[96]: sns.countplot(x = ' education', hue = ' loan_status', data = df).set_title('Loan Status by Education')
```

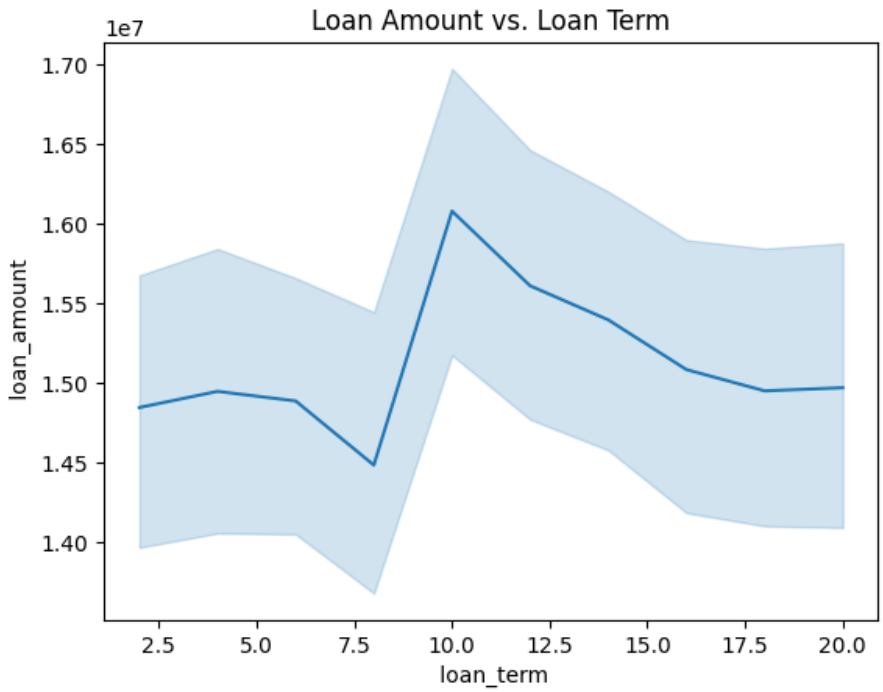
```
[96]: Text(0.5, 1.0, 'Loan Status by Education')
```



The graph indicates that there's only a small difference between the number of loans approved and rejected for both graduate and non-graduate applicants. This difference is so small that it doesn't seem to be significant.

Loan Amount vs Terms

```
[97]: sns.lineplot(x = ' loan_term', y = ' loan_amount', data = df).set_title('Loan Amount vs. Loan Term')

[97]: Text(0.5, 1.0, 'Loan Amount vs. Loan Term')
```



This line plot shows the trend between the loan amount and the loan tenure. Between the loan tenure of 2.5 - 7.5 years, the loan amount is between 1400000 - 15500000.

However, the loan amount is significantly higher for a loan tenure of 10 years. There is a huge difference.

Loan Amount vs Loan Status

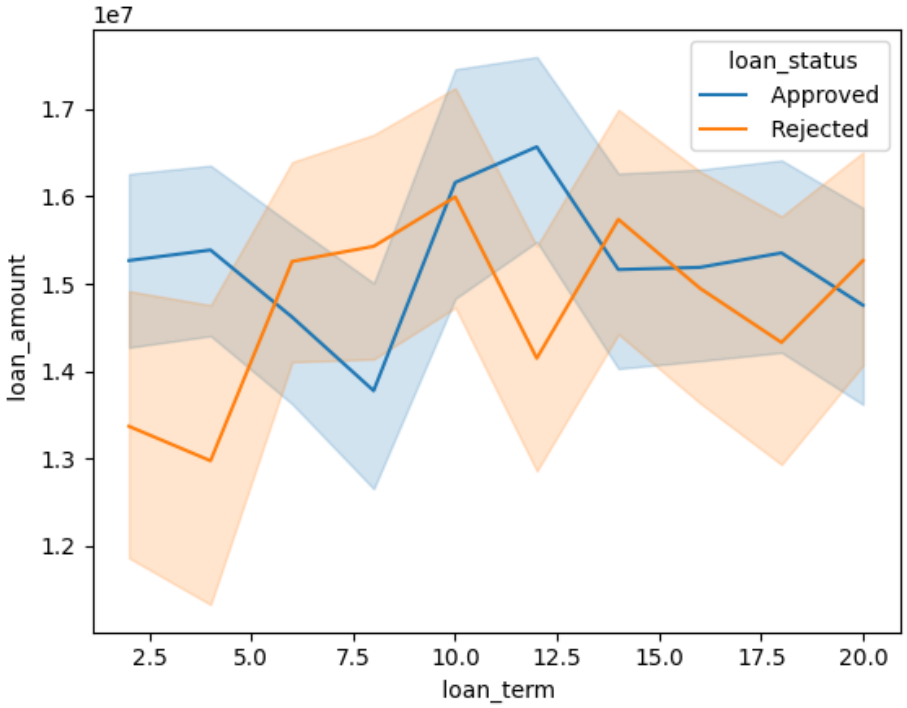
```
[99]: sns.violinplot(x=' loan_status', y=' loan_amount', data=df)
```

```
[99]: <Axes: xlabel=' loan_status', ylabel=' loan_amount'>
```

Loan Amount & Tenure vs Loan Status

```
[100]: sns.lineplot(x=' loan_term', y=' loan_amount', data=df, hue=' loan_status')
```

```
[100]: <Axes: xlabel=' loan_term', ylabel=' loan_amount'>
```



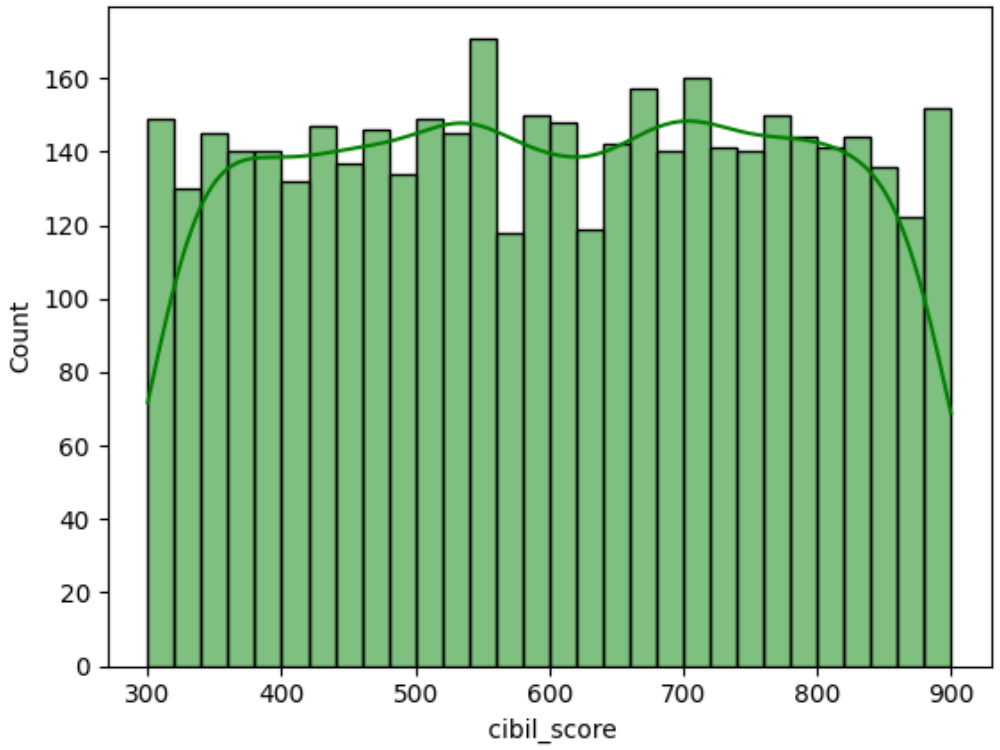
The graph shows how loans are connected to the amount you borrow, how long you take to pay them back, and whether the bank approves or rejects them. If a loan is approved, it usually means it's for a higher amount and you must pay it back faster. On the flip side, rejected loans are usually for less money and a longer time to pay back. This happens because banks like to approve loans that can be paid back quickly and make more money. Dealing with small loans might cost the bank too much. But there are other things like how good you are with money that also matter in these decisions. So, the graph gives us a peek into how banks decide to say 'yes' or 'no' to loans.

CIBIL Score Distribution

> CIBIL Score ranges and their meaning.

CIBIL	*Meaning*
$300 - 549$	*Poor*
$550 - 649$	*Fair*
$650 - 749$	*Good*
$750 - 799$	*VeryGood*
$800 - 900$	*Excellent*

```
[101]: # Viewing the distribution of the cibil_score column
       sns.histplot(df[" cibil_score"], bins=30, kde=True, color='green')

[101]: <Axes: xlabel=' cibil_score', ylabel='Count'>
```



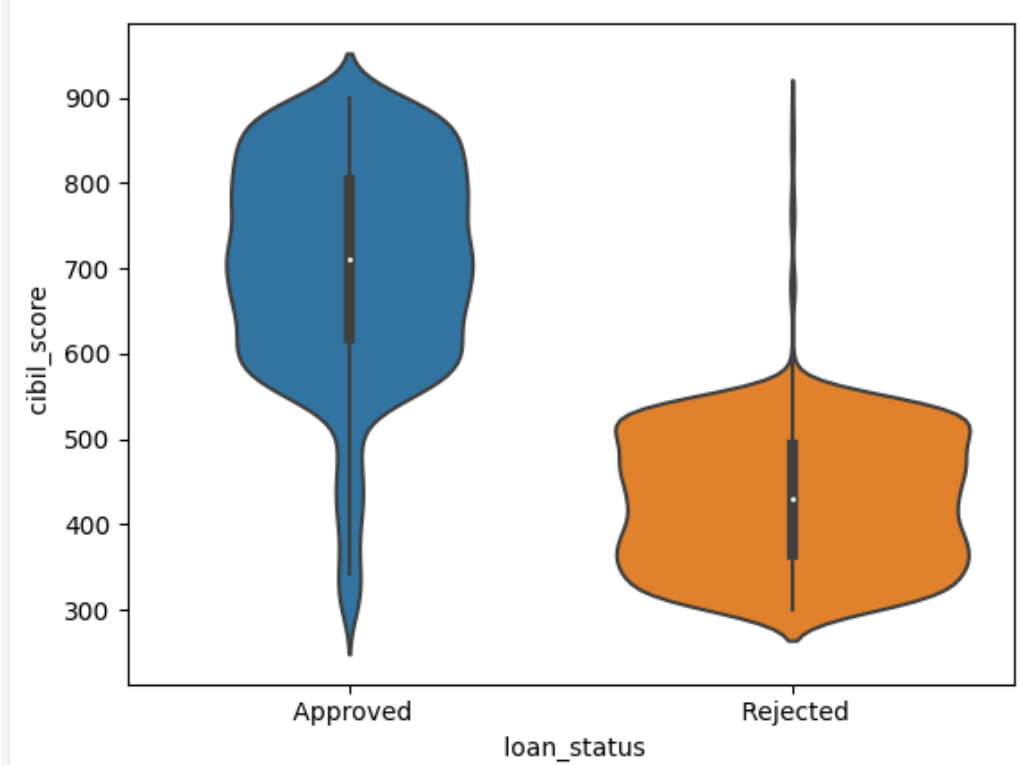
Analyzing the table reveals that a majority of customers have low CIBIL scores, specifically below 649. This might pose challenges for them in getting loan approvals. However, there's a notable portion of customers with high scores, exceeding 649, which is advantageous for the bank. It opens the opportunity for the bank to provide special treatment, such as attractive deals and offers, to attract these high-score customers to take loans from the bank.

From this, we can infer that individuals with higher CIBIL scores are likely to have a higher chance of getting their loans approved. Typically, higher scores indicate better financial management.

CIBIL Score Vs Loan Status

```
[102]: sns.violinplot(x=' loan_status', y=' cibil_score', data=df)
```

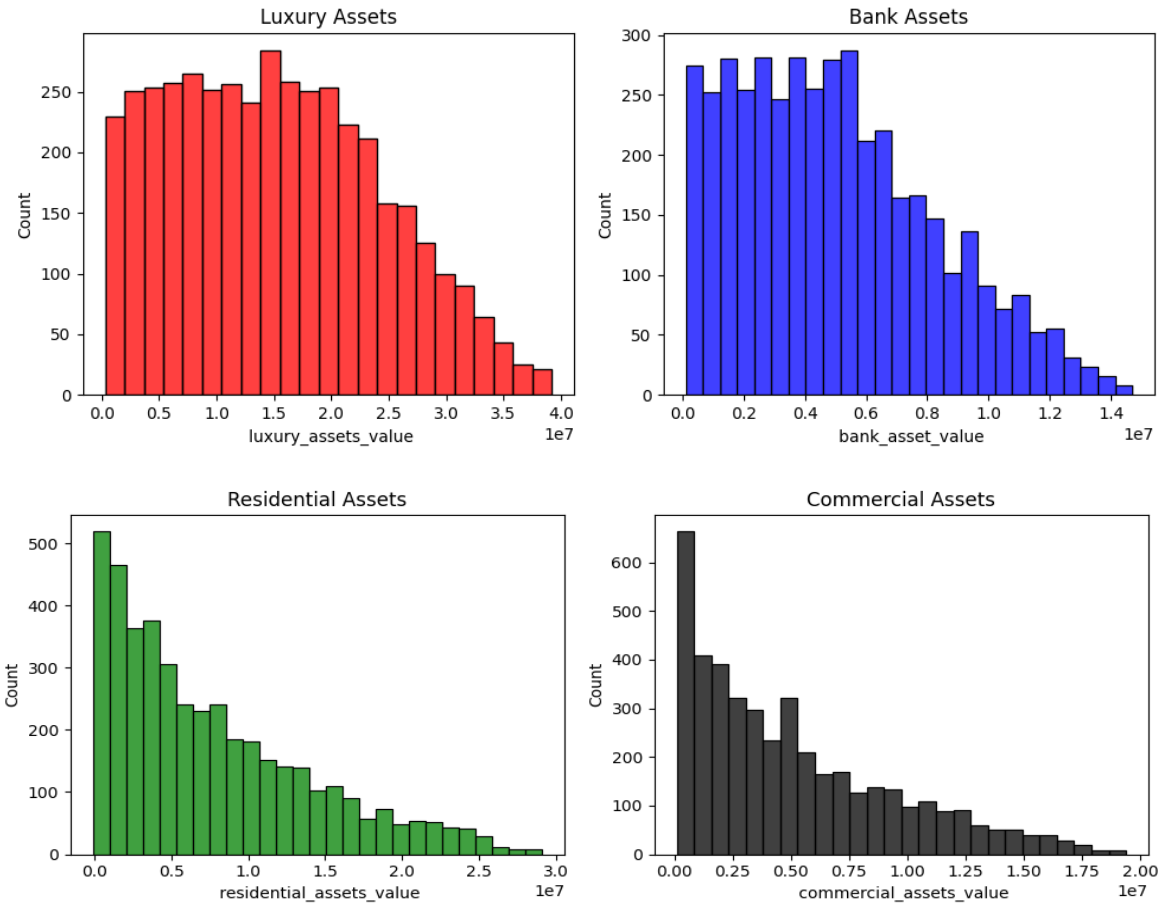
```
[102]: <Axes: xlabel=' loan_status', ylabel=' cibil_score'>
```



The violin plot distinctly illustrates that individuals with approved loans generally possess higher CIBIL scores, predominantly exceeding 600. In contrast, for those with rejected loans, scores exhibit greater variability and tend to be lower, often below 550. This underscores the significance of having a higher CIBIL score, particularly surpassing 600, in significantly boosting the chances of loan approval. The graph unmistakably highlights the pivotal role of a good CIBIL score in the loan approval process.

Asset Distribution ¶

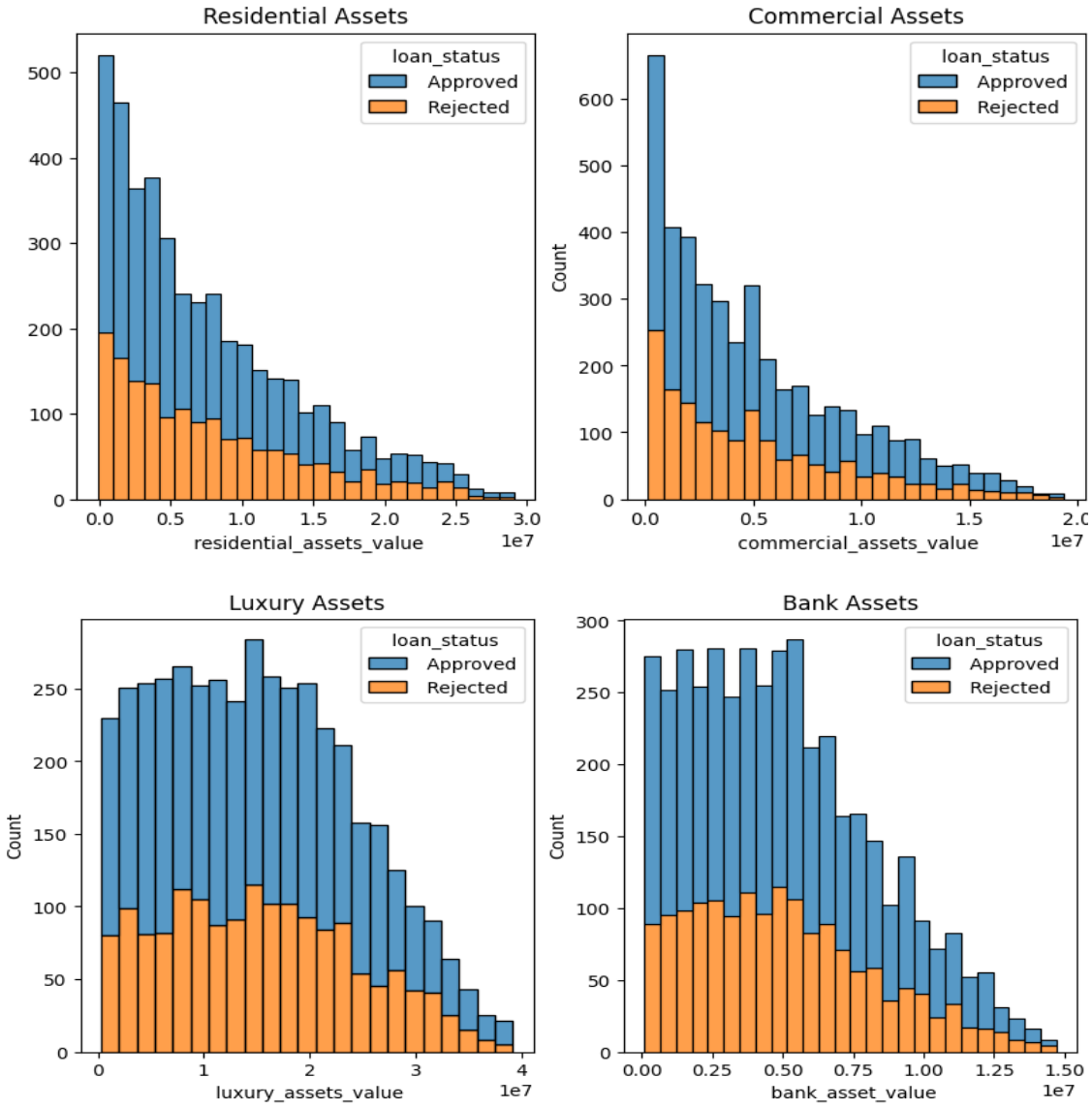
```
[103]: fig, ax = plt.subplots(2, 2, figsize=(10, 8))

plt.subplot(2, 2, 1)
sns.histplot(df[' luxury_assets_value'], color='red')
plt.title("Luxury Assets")

plt.subplot(2, 2, 2)
sns.histplot(df[' bank_asset_value'], color='blue')
plt.title("Bank Assets")

plt.subplot(2, 2, 3)
sns.histplot(df[' residential_assets_value'], color='green')
plt.title("Residential Assets")

plt.subplot(2, 2, 4)
sns.histplot(df[' commercial_assets_value'], color='black')
plt.title("Commercial Assets")

plt.tight_layout()
plt.show()
```



These graphs reveal a trend where the majority of individuals possess lower-valued assets, and the count of people with more valuable assets gradually decreases. This insight enhances our understanding of how assets play a role in influencing loan decisions.

Assets vs Loan Status

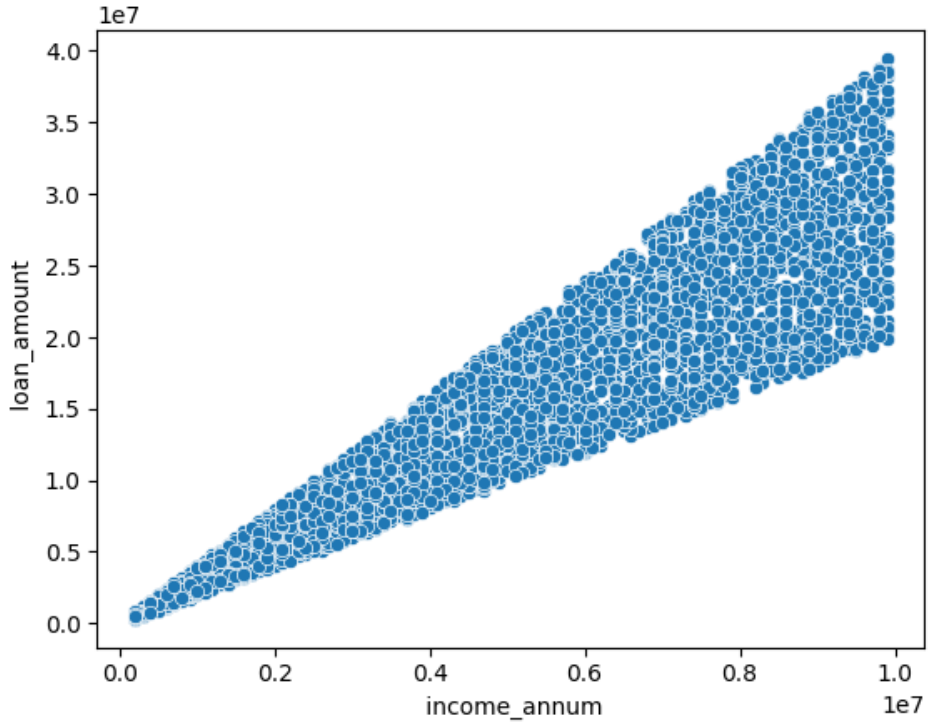
[104]:
```
fig, ax = plt.subplots(1, 4, figsize=(20, 5))

sns.histplot(x=' luxury_assets_value', data=df, ax=ax[0], hue=' loan_status', multiple='stack')
ax[0].set_title("Luxury Assets")

sns.histplot(x=' bank_asset_value', data=df, ax=ax[1], hue=' loan_status', multiple='stack')
ax[1].set_title("Bank Assets")

sns.histplot(x=' residential_assets_value', data=df, ax=ax[2], hue=' loan_status', multiple='stack')
ax[2].set_title("Residential Assets")

sns.histplot(x=' commercial_assets_value', data=df, ax=ax[3], hue=' loan_status', multiple='stack')
ax[3].set_title("Commercial Assets")

plt.show()
```



Loan Amount vs Income

```
[105]: sns.scatterplot(x=' income_annum', y = ' loan_amount', data = df)
```

```
[105]: <Axes: xlabel=' income_annum', ylabel=' loan_amount'>
```

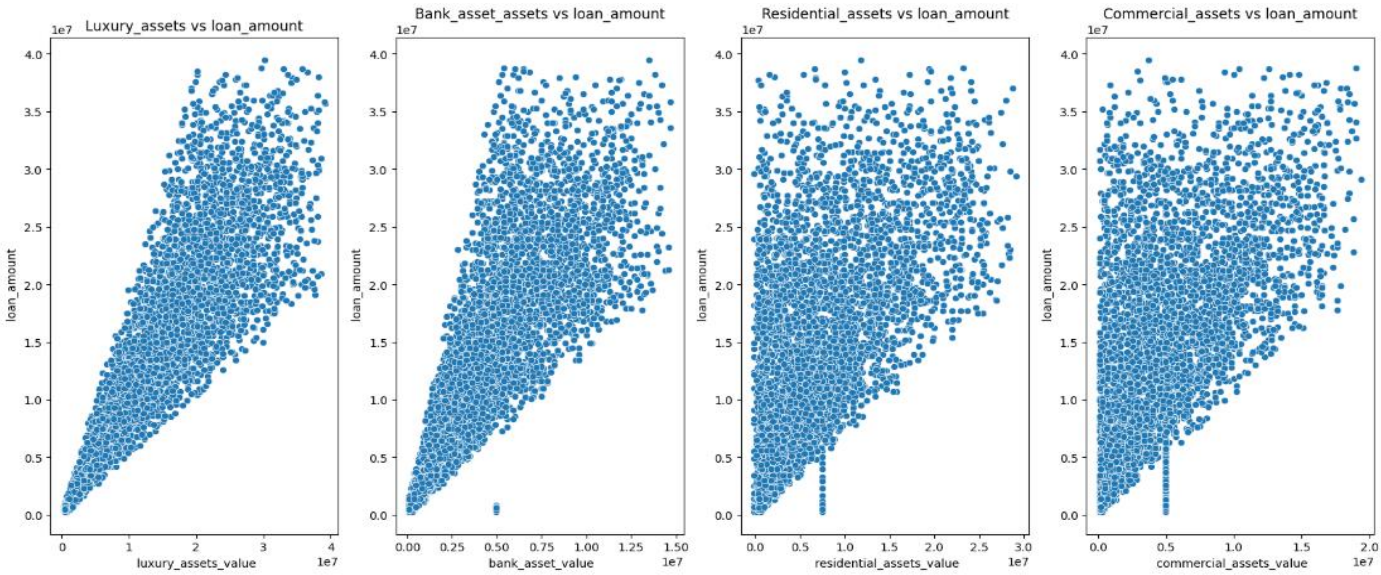


The relationship between the loan amount and the applicant's annual income is straightforward. When the income is higher, the loan amount tends to be higher as well. This correlation is rooted in the fact that the applicant's income significantly influences the determination of a suitable loan amount they can comfortably repay.

Assets vs Loan Amount ¶

[106]:
```
fig, ax = plt.subplots(1,4,figsize=(20, 8))
sns.scatterplot(x=' luxury_assets_value', y = ' loan_amount', data = df, ax=ax[0]).set_title('Luxury_assets vs loan_amount')
sns.scatterplot(x=' bank_asset_value', y = ' loan_amount', data = df, ax=ax[1]).set_title('Bank_asset_assets vs loan_amount')
sns.scatterplot(x=' residential_assets_value', y = ' loan_amount', data = df, ax=ax[2]).set_title('Residential_assets vs loan_amount')
sns.scatterplot(x=' commercial_assets_value', y = ' loan_amount', data = df, ax=ax[3]).set_title('Commercial_assets vs loan_amount')
```

[106]:
```
Text(0.5, 1.0, 'Commercial_assets vs loan_amount')
```



The observation indicates that possessing more assets enhances the probability of securing a larger loan from the bank. However, it's worth noting the presence of outliers, signifying instances where individuals with comparatively fewer assets may still obtain larger loans.

Label Encoding the categorical variables

[107]:
```
# Label Encoding
df[' education'] = df[' education'].map({' Not Graduate':0, ' Graduate':1})
df[' self_employed'] = df[' self_employed'].map({' No':0, ' Yes':1})
df[' loan_status'] = df[' loan_status'].map({' Rejected':0, ' Approved':1})
```

> Now all features are numerical.

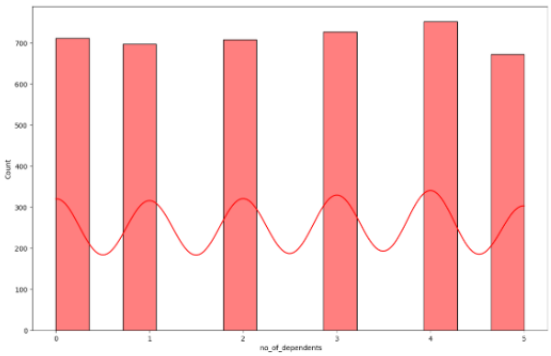
[108]:
```
df.head()
```

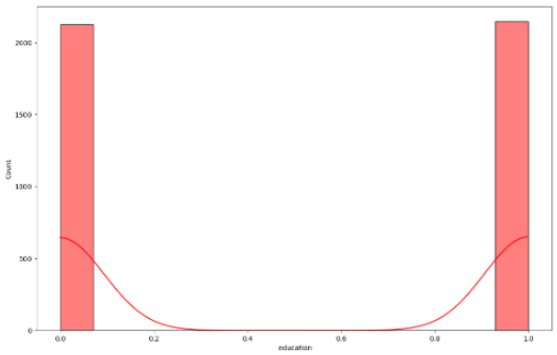
[108]:

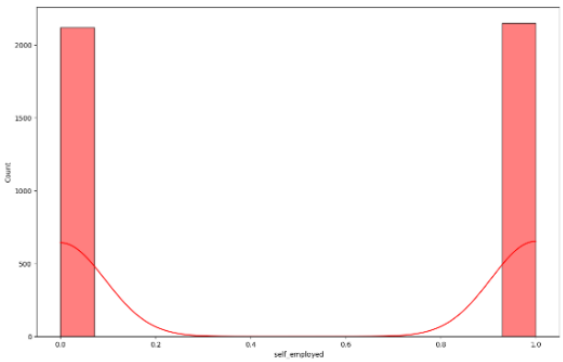
	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term	cibil_score	residential_assets_value	commercial_assets_value	luxury_assets_valu
0	2	1	0	9600000	29900000	12	778	2400000.0	17600000.0	227000(
1	0	0	1	4100000	12200000	8	417	2700000.0	2200000.0	880000
2	3	1	0	9100000	29700000	20	506	7100000.0	4500000.0	333000(
3	3	1	0	8200000	30700000	8	467	18200000.0	3300000.0	233000(
4	5	0	1	9800000	24200000	20	382	12400000.0	8200000.0	294000(

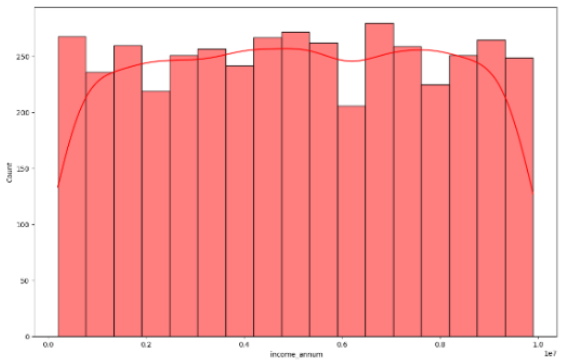
Histograms for each feature

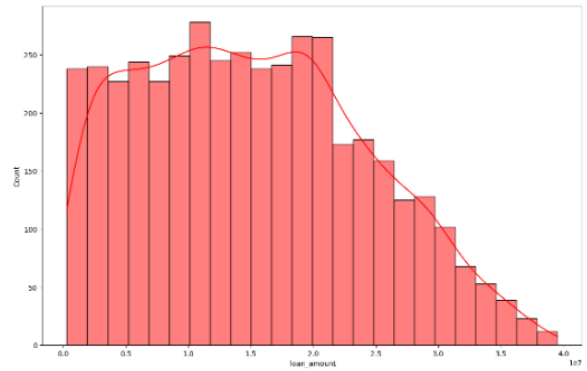
[109]:
```
fig, axes = plt.subplots(nrows = 5, ncols = 2)
axes = axes.flatten()
fig.set_size_inches(30,50)

for ax, col in zip(axes, df.columns):
    sns.histplot(df[col],kde=True, color='red', ax = ax)
```

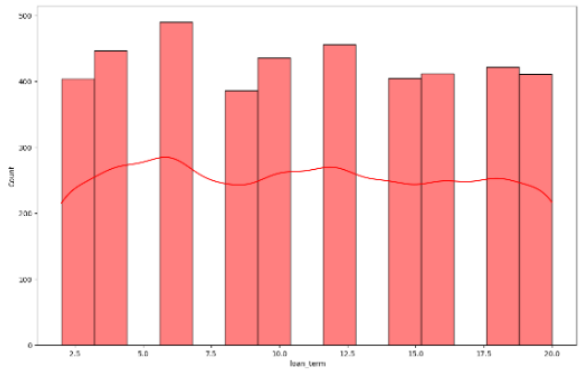


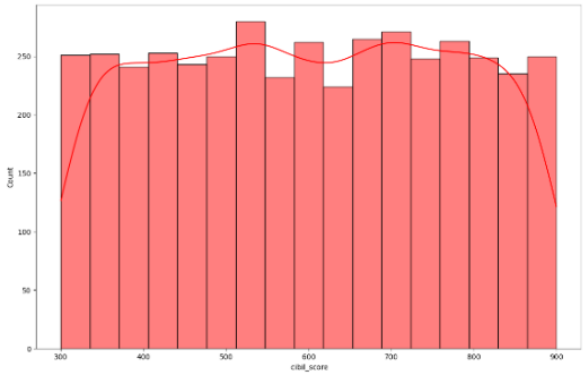


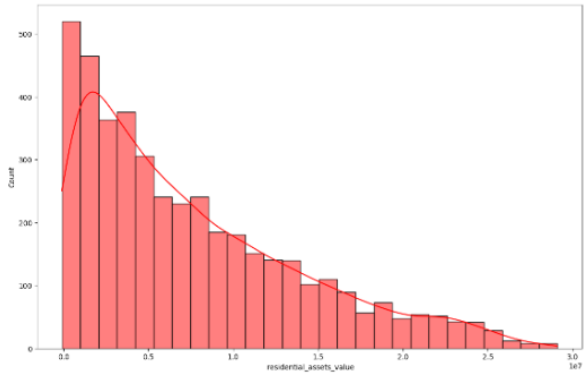


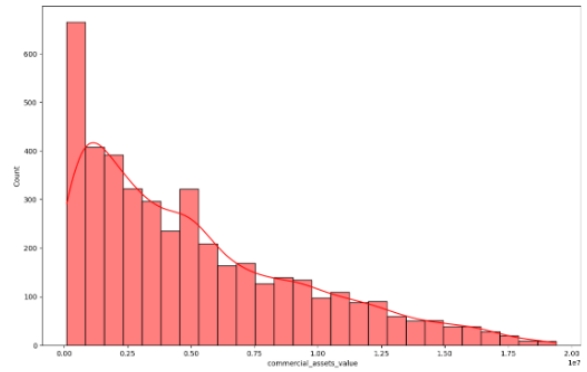


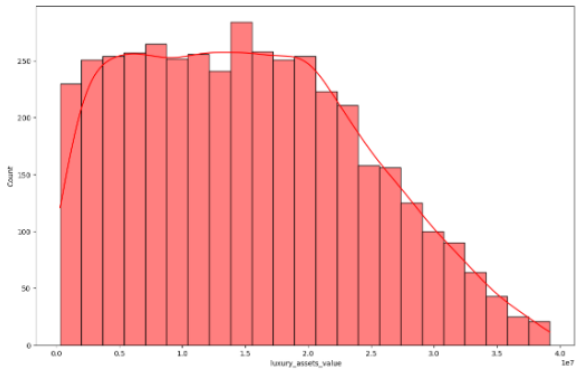

Count
loan_amount
1e7


Count
loan_term


Count
cibil_score

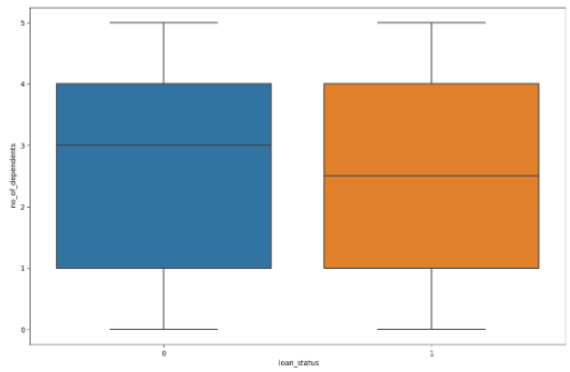

Count
residential_assets_value
1e7

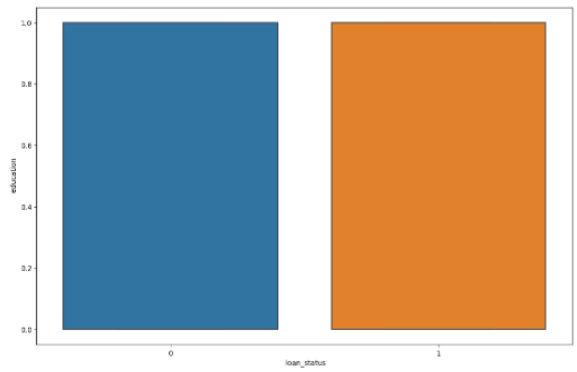

Count
commercial_assets_value
1e7

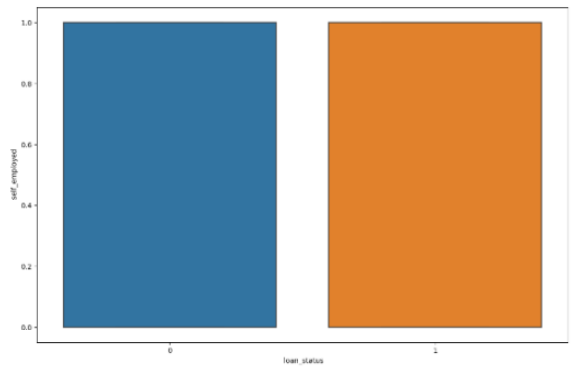

Count
luxury_assets_value
1e7

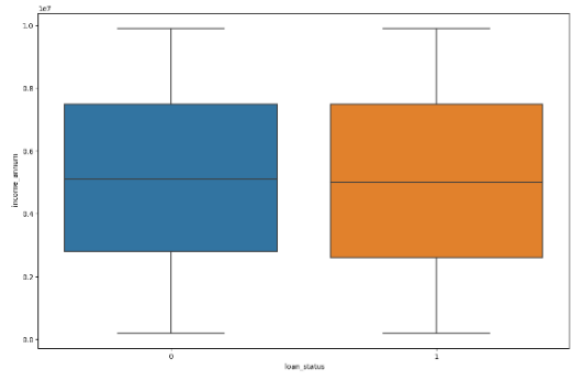
Boxplot for each feature

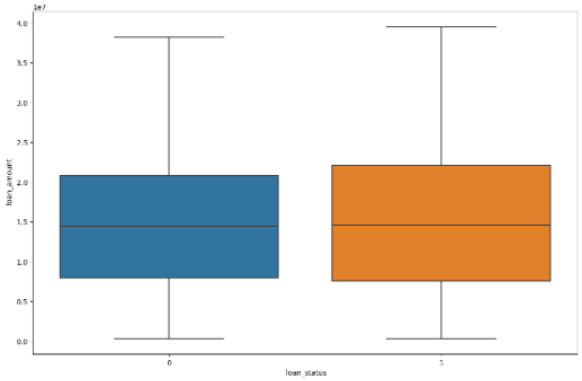
[110]:
```
fig, axes = plt.subplots(nrows = 5, ncols = 2)
axes = axes.flatten()
fig.set_size_inches(30,50)

for ax, col in zip(axes, df.columns):
  sns.boxplot(x=' loan_status',y=df[col], ax = ax , data=df)
```

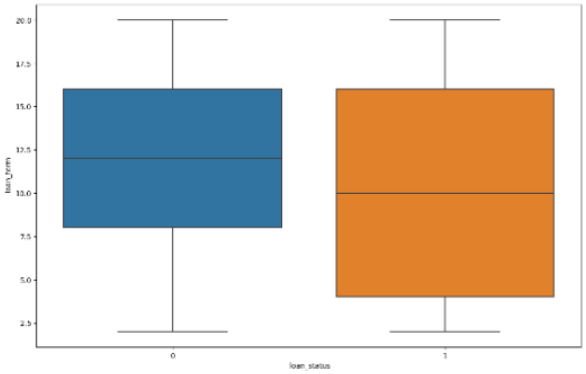


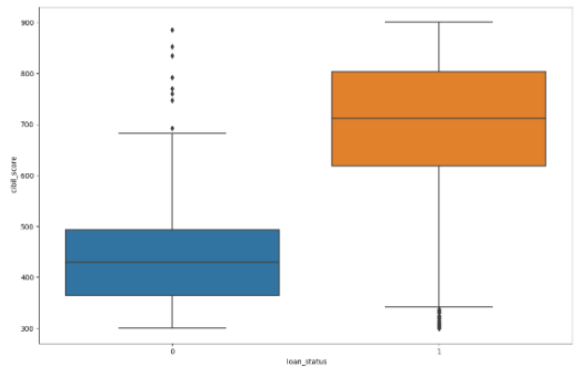


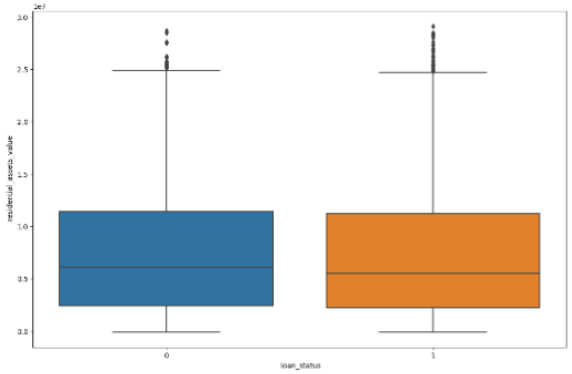


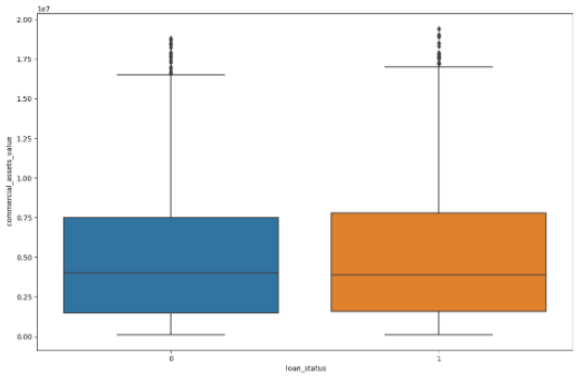


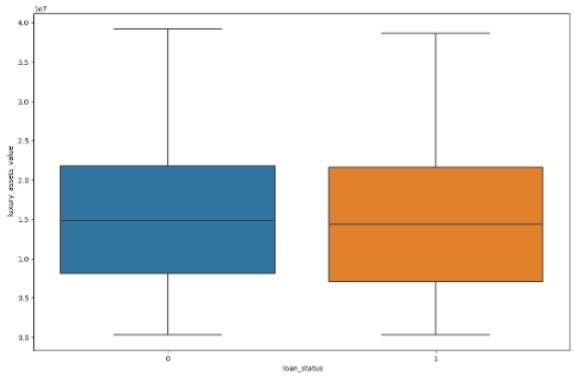





cibil_score
loan_status
0
1

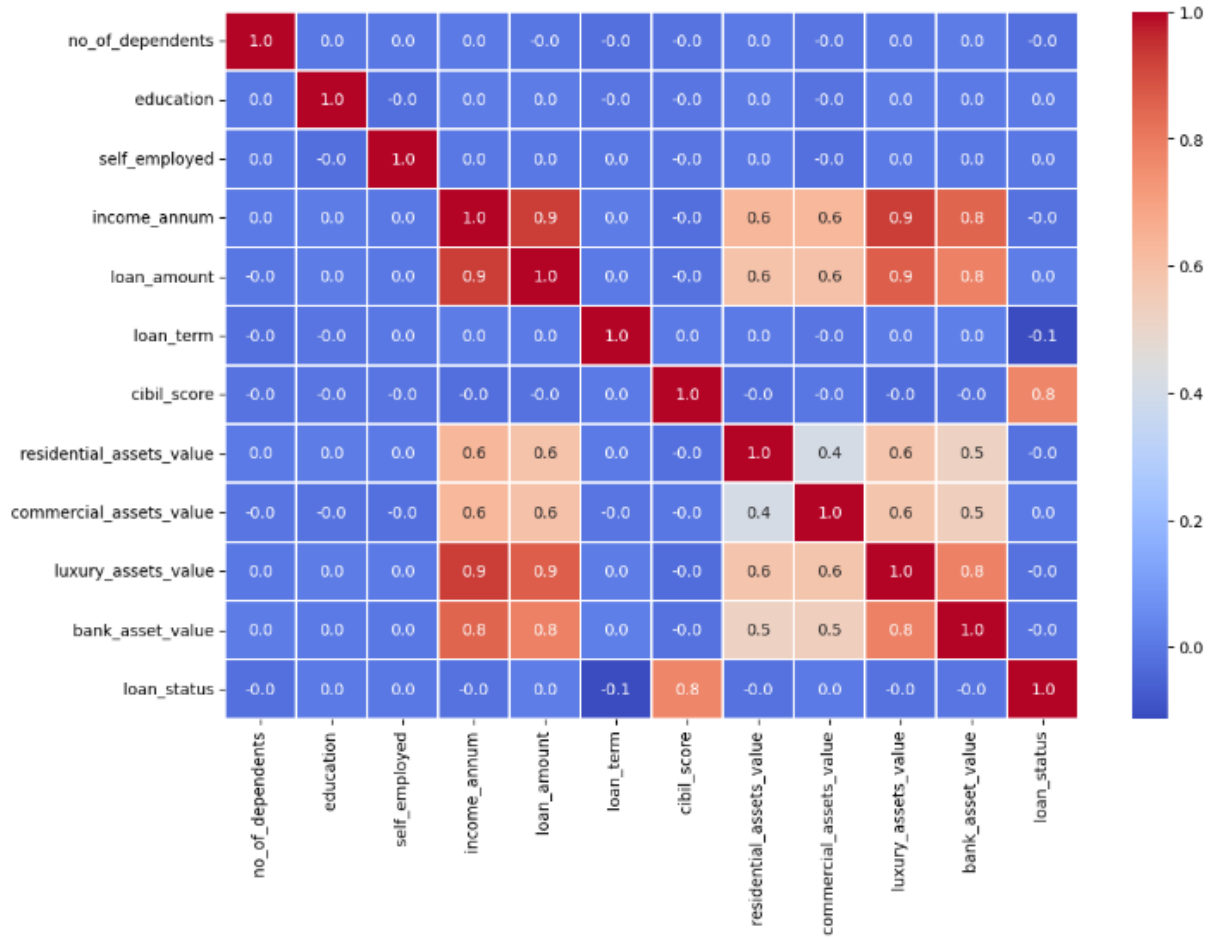

1e7
residential_assets_value
loan_status
0
1


1e7
commercial_assets_value
loan_status
0
1


1e7
luxury_assets_value
loan_status
0
1

Correlation Matrix

[111]:
```
plt.figure(figsize=(12,8))
sns.heatmap(df.corr(), cmap='coolwarm', annot=True, fmt='.1f', linewidths=.1)
plt.show()
```



[112]:
```
df.corr()[' loan_status']
```

[112]:
```
no_of_dependents           -0.018114
education                   0.004918
self_employed               0.000345
income_annum               -0.015189
loan_amount                 0.016150
loan_term                  -0.113036
cibil_score                 0.770518
residential_assets_value   -0.014467
commercial_assets_value     0.007511
luxury_assets_value        -0.015465
bank_asset_value           -0.006777
loan_status                 1.000000
Name:  loan_status, dtype: float64
```

Data Preprocessing

5. Data Preprocessing

[113]:
```
df
```

[113]:

	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term	cibil_score	residential_assets_value	commercial_assets_value	luxury_assets_
0	2	1	0	9600000	29900000	12	778	2400000.0	17600000.0	227
1	0	0	1	4100000	12200000	8	417	2700000.0	2200000.0	88
2	3	1	0	9100000	29700000	20	506	7100000.0	4500000.0	333
3	3	1	0	8200000	30700000	8	467	18200000.0	3300000.0	233
4	5	0	1	9800000	24200000	20	382	12400000.0	8200000.0	294
...	...	...	...	...	...	...	...	...	...	
4264	5	1	1	1000000	2300000	12	317	2800000.0	500000.0	33
4265	0	0	1	3300000	11300000	20	559	4200000.0	2900000.0	110
4266	2	0	0	6500000	23900000	18	457	1200000.0	12400000.0	181
4267	1	0	0	4100000	12800000	8	780	8200000.0	700000.0	141
4268	1	1	0	9200000	29700000	10	607	17800000.0	11800000.0	357

4269 rows × 12 columns

Outlier Detection

Using zscore

[114]:
```
from scipy.stats import zscore

# Create a copy of the DataFrame to avoid modifying the original
df_copy = df.copy()

# Calculate Z-scores for each numeric column
numeric_columns = df_copy.select_dtypes(include=[np.number]).columns
df_copy[numeric_columns] = df_copy[numeric_columns].apply(zscore)

# Set a threshold for Z-score (e.g., 3)
threshold = 3

# Identify outliers based on Z-score
outliers = df_copy[(np.abs(df_copy[numeric_columns]) > threshold).any(axis=1)]

print(outliers.count())
```

```
no_of_dependents            33
education                   33
self_employed               33
income_annum                33
loan_amount                 33
loan_term                   33
cibil_score                 33
residential_assets_value    33
commercial_assets_value     33
luxury_assets_value         33
bank_asset_value            33
loan_status                 33
dtype: int64
```

Using IsolationForest

[115]:
```
from sklearn.ensemble import IsolationForest

# Create an Isolation Forest model
clf = IsolationForest(contamination='auto', random_state=42)  # Adjust contamination based on your data

# Fit the model and predict outliers
df_copy['outlier'] = clf.fit_predict(df_copy)

# Count the number of outlier rows
outlier_count = df_copy[df_copy['outlier'] == -1].shape[0]

# Display the count of outliers
print("Number of outlier rows:", outlier_count)
```

```
Number of outlier rows: 2706
```

[116]:
```
X = df.drop(' loan_status', axis = 1)
y = df[' loan_status']
```

Although outlier detection is tested it is not implemented because accuracy slightly decreases. Also, the accuracy is significantly higher without outlier detection.

[117]:
```
y.value_counts()
```

[117]:
```
 loan_status
1    2656
0    1613
Name: count, dtype: int64
```

> It is clearly unbalanced data, so we need to oversample the minority class

Oversampling the minority class

[118]:
```
from imblearn.over_sampling import RandomOverSampler

rs = RandomOverSampler()

X, y = rs.fit_resample(X,y)
```

[119]:
```
y.value_counts()
```

[119]:
```
 loan_status
1    2656
0    2656
Name: count, dtype: int64
```

[120]:
```
import warnings

# Suppress warnings within this code block
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
```

Machine Learning Modeling with Hyperparameter Tuning

6. ML Modelling with Hyperparameter Tuning

[121]:
```
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
from sklearn.compose import ColumnTransformer
from sklearn import tree
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
```

Train Test Split

[137]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Feature Scaling

[138]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

[139]:
```
# Save the fitted scaler to a file using pickle
scaler_filename = 'standard_scaler.pkl'
with open(scaler_filename, 'wb') as scaler_file:
    pickle.dump(sc, scaler_file)
```

1. Random Forest Classifier ¶

[125]:
```
from sklearn.ensemble import RandomForestClassifier

# Define the model
rf_model = RandomForestClassifier()

# Define hyperparameters for tuning
param_grid = {
    'n_estimators': [10, 50, 100],
    'max_depth': [None, 10, 15, 20],
    'min_samples_split': [2, 5, 10]
}

# Perform GridSearchCV
grid_search_rf = GridSearchCV(rf_model, param_grid, cv=5)
grid_search_rf.fit(X_train, y_train)

# Print the best hyperparameters
best_hyperparameters = grid_search_rf.best_params_
print("Best Hyperparameters (Random Forest):", best_hyperparameters)
```

```
# Evaluate the model
train_accuracy = accuracy_score(y_train, grid_search_rf.predict(X_train))
test_accuracy = accuracy_score(y_test, grid_search_rf.predict(X_test))

print("Random Forest Classifier:")
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

```
Best Hyperparameters (Random Forest): {'max_depth': 20, 'min_samples_split': 2, 'n_estimators': 50}
Random Forest Classifier:
Training Accuracy: 1.0
Testing Accuracy: 0.9821260583254939
```

2. Support Vector Classification (SVC) ¶

[140]:
```
from sklearn.svm import SVC

# Define the model
svm_model = SVC()

# Define hyperparameters for tuning
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf'],
}

# Perform GridSearchCV
grid_search_svm = GridSearchCV(svm_model, param_grid, cv=5)
grid_search_svm.fit(X_train, y_train)

# Print the best hyperparameters
best_hyperparameters = grid_search_svm.best_params_
print("Best Hyperparameters (SVM):", best_hyperparameters)
```

```
# Evaluate the model
train_accuracy = accuracy_score(y_train, grid_search_svm.predict(X_train))
test_accuracy = accuracy_score(y_test, grid_search_svm.predict(X_test))

print("SVM Classifier:")
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

```
Best Hyperparameters (SVM): {'C': 10, 'kernel': 'rbf'}
SVM Classifier:
Training Accuracy: 0.9781124970581313
Testing Accuracy: 0.955785512699906
```

3. Naive Bayes

[127]:
```
from sklearn.naive_bayes import GaussianNB

# Define the model
nb_model = GaussianNB()

# No hyperparameters to tune for Gaussian Naive Bayes

# Fit the model
nb_model.fit(X_train, y_train)

# Evaluate the model
train_accuracy = accuracy_score(y_train, nb_model.predict(X_train))
test_accuracy = accuracy_score(y_test, nb_model.predict(X_test))

print("Naive Bayes Classifier:")
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

```
Naive Bayes Classifier:
Training Accuracy: 0.9515180042362908
Testing Accuracy: 0.9501411100658513
```

4. Gradient Boosting Classifier

[128]:
```
from sklearn.ensemble import GradientBoostingClassifier

# Define the model
gb_model = GradientBoostingClassifier()

# Define hyperparameters for tuning
param_grid = {
    'n_estimators': [10, 50, 100],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5]
}

# Perform GridSearchCV
grid_search_gb = GridSearchCV(gb_model, param_grid, cv=5)
grid_search_gb.fit(X_train, y_train)

# Print the best hyperparameters
best_hyperparameters = grid_search_gb.best_params_
print("Best Hyperparameters (GradientBoostingClassifier):", best_hyperparameters)
```

```
# Evaluate the model
train_accuracy = accuracy_score(y_train, grid_search_gb.predict(X_train))
test_accuracy = accuracy_score(y_test, grid_search_gb.predict(X_test))

print("Gradient Boosting Classifier:")
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

```
Best Hyperparameters (GradientBoostingClassifier): {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 50}
Gradient Boosting Classifier:
Training Accuracy: 0.9990586020240056
Testing Accuracy: 0.9858889934148636
```

5. Decision Tree Classifier

[129]:
```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import GridSearchCV

# Define the model
dt_model = DecisionTreeClassifier()

# Define hyperparameters for tuning
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Perform GridSearchCV
grid_search_dc = GridSearchCV(dt_model, param_grid, cv=5)
grid_search_dc.fit(X_train, y_train)

# Print the best hyperparameters
best_hyperparameters = grid_search_dc.best_params_
print("Best Hyperparameters (Decision Tree):", best_hyperparameters)
```

```
# Get the best model
best_dt_model = grid_search_dc.best_estimator_

# Evaluate the model
train_accuracy = accuracy_score(y_train, best_dt_model.predict(X_train))
test_accuracy = accuracy_score(y_test, best_dt_model.predict(X_test))

print("Decision Tree Classifier:")
print("Training Accuracy:", train_accuracy)
print("Testing Accuracy:", test_accuracy)
```

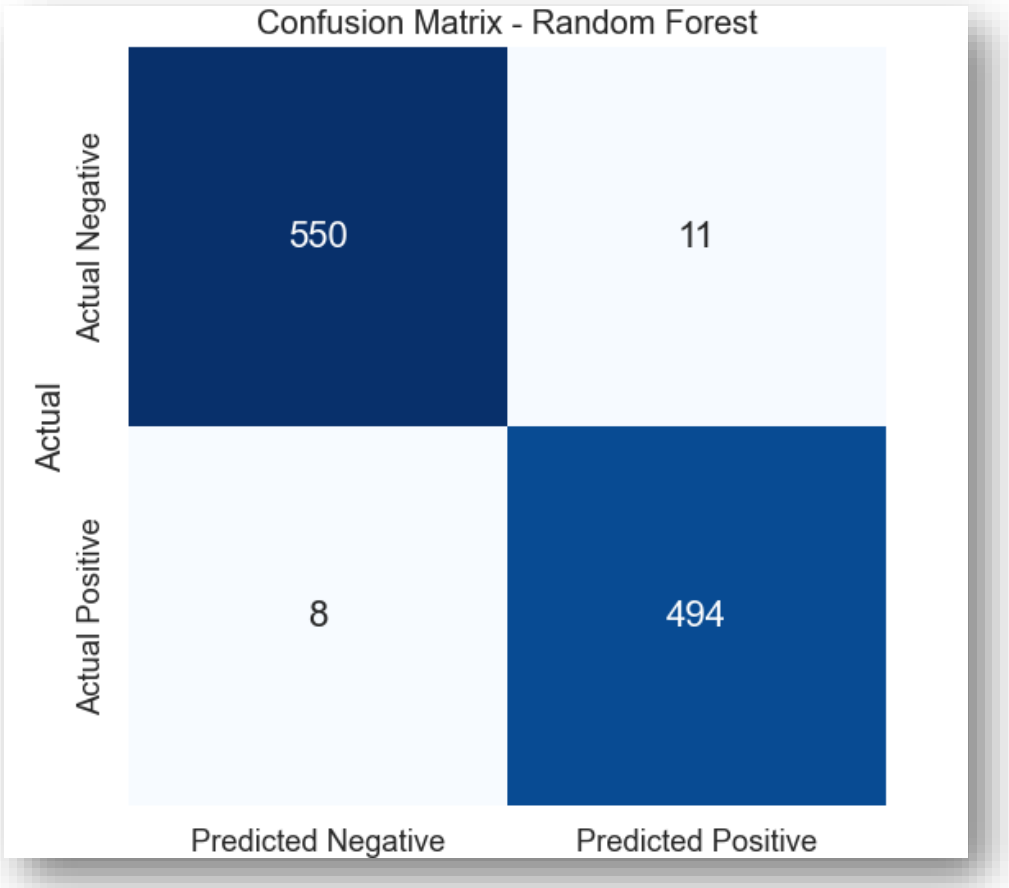
```
Best Hyperparameters (Decision Tree): {'criterion': 'entropy', 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
Decision Tree Classifier:
Training Accuracy: 1.0
Testing Accuracy: 0.9840075258701787
```

Model Evaluation

8. Model Evalution (Confusion Matrix and Classification Report) ¶

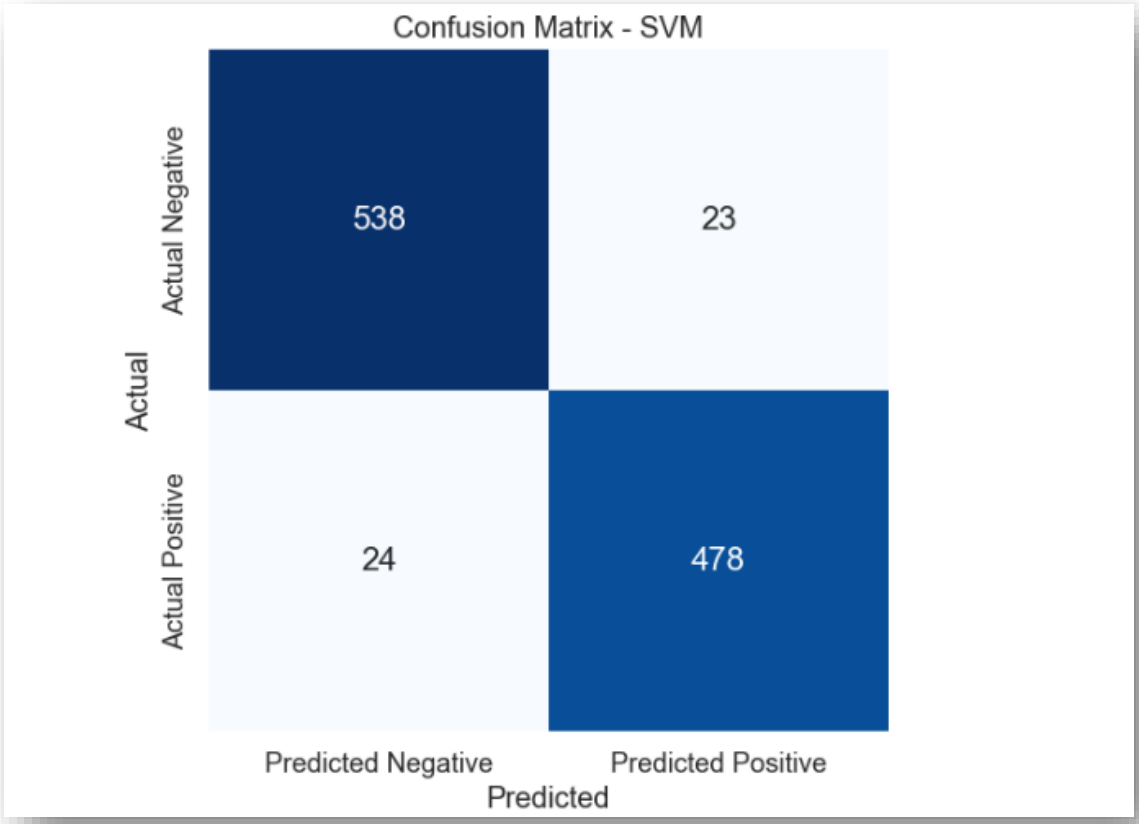
[130]:
```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.preprocessing import StandardScaler

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Define a list of models
models = [
    ("Random Forest", grid_search_rf),
    ("SVM", grid_search_svm),
    ("Naive Bayes", nb_model),
    ("Gradient Boosting", grid_search_gb),
    ("Decision Tree", grid_search_dc),
]

# Loop through each model
for model_name, model in models:
    # Get model predictions
    predictions = model.predict(X_test)
```

```
    # Calculate confusion matrix
    cm = confusion_matrix(y_test, predictions)

    # Create a confusion matrix heatmap
    plt.figure(figsize=(8, 6))
    sns.set(font_scale=1.2)  # Adjust font size
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                annot_kws={"size": 16}, square=True,
                xticklabels=['Predicted Negative', 'Predicted Positive'],
                yticklabels=['Actual Negative', 'Actual Positive'])
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(f"Confusion Matrix - {model_name}")
    plt.show()

    # Display classification report
    print(f"Classification Report - {model_name}:\n")
    print(classification_report(y_test, predictions))
```



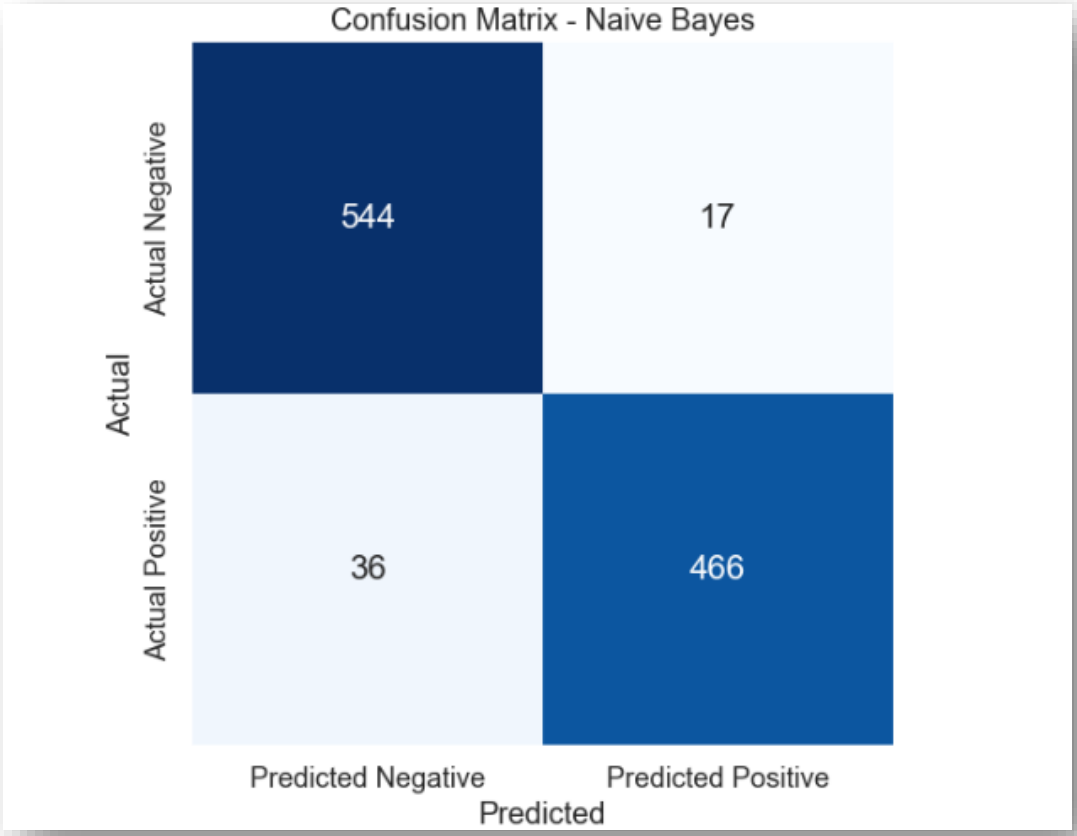
Classification Report - Random Forest:

	precision	recall	f1-score	support
0	0.99	0.98	0.98	561
1	0.98	0.98	0.98	502
accuracy			0.98	1063
macro avg	0.98	0.98	0.98	1063
weighted avg	0.98	0.98	0.98	1063



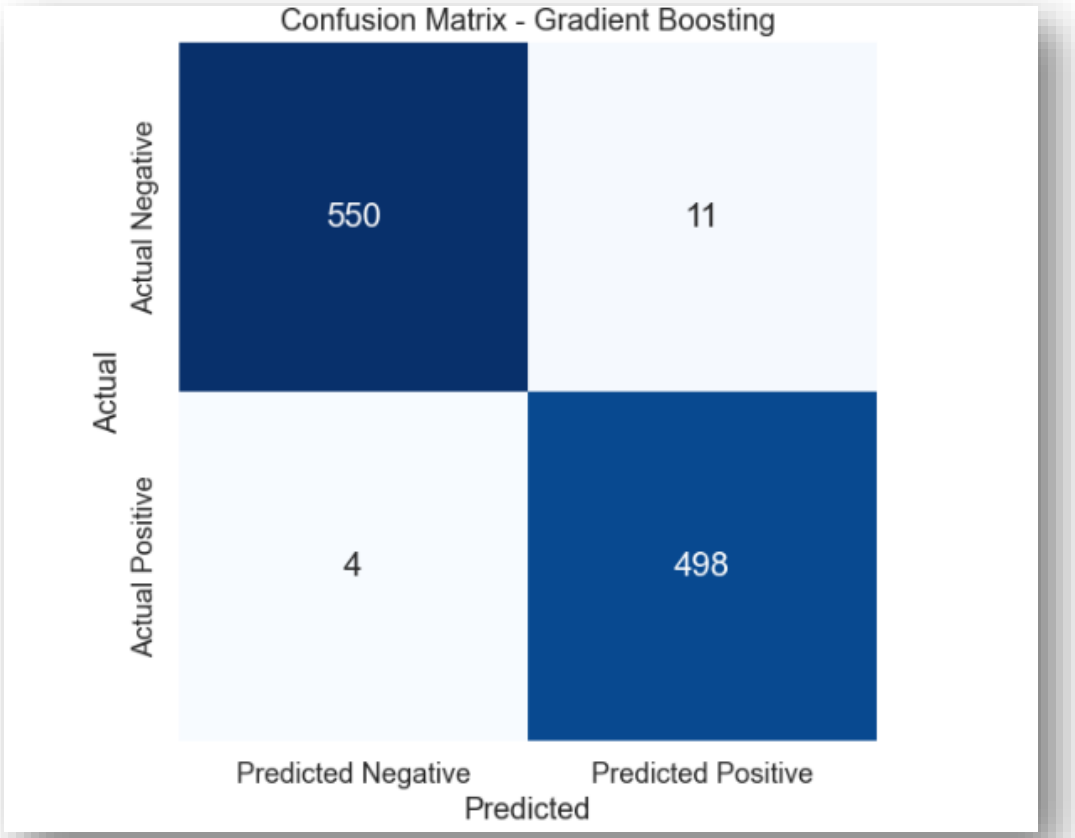
Classification Report - SVM:

	precision	recall	f1-score	support
0	0.96	0.96	0.96	561
1	0.95	0.95	0.95	502
accuracy			0.96	1063
macro avg	0.96	0.96	0.96	1063
weighted avg	0.96	0.96	0.96	1063



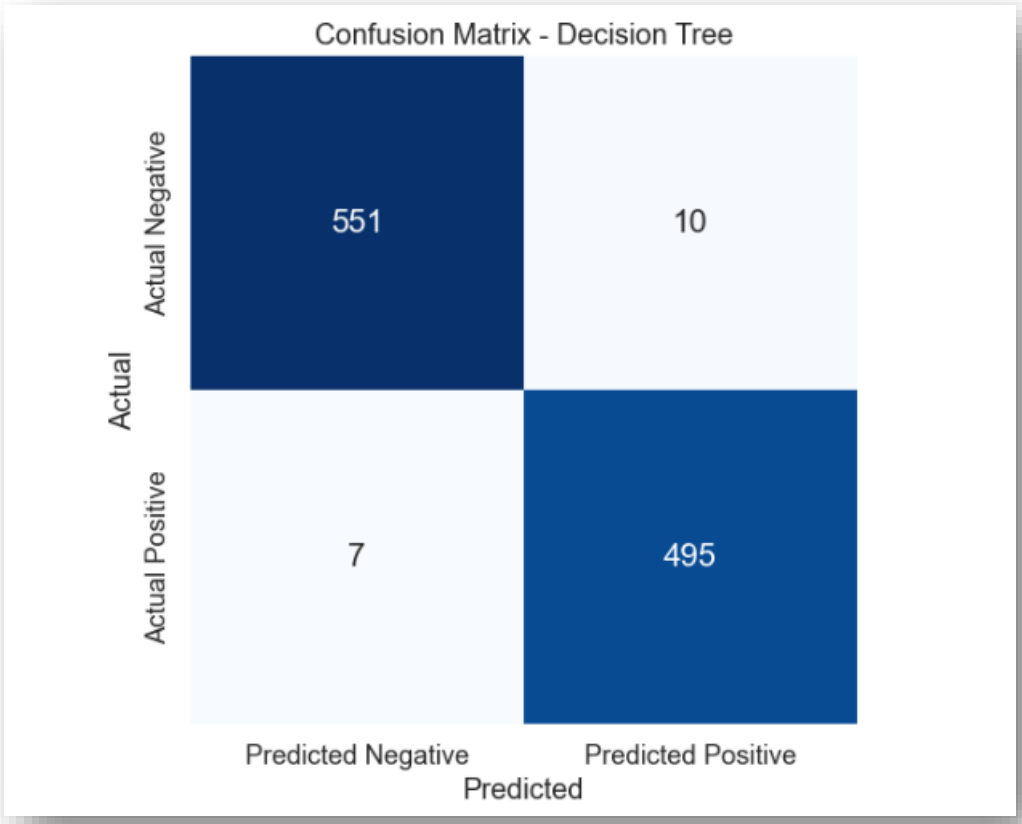
Classification Report - Naive Bayes:

	precision	recall	f1-score	support
0	0.94	0.97	0.95	561
1	0.96	0.93	0.95	502
accuracy			0.95	1063
macro avg	0.95	0.95	0.95	1063
weighted avg	0.95	0.95	0.95	1063



Classification Report - Gradient Boosting:

	precision	recall	f1-score	support
0	0.99	0.98	0.99	561
1	0.98	0.99	0.99	502
accuracy			0.99	1063
macro avg	0.99	0.99	0.99	1063
weighted avg	0.99	0.99	0.99	1063



Classification Report - Decision Tree:

	precision	recall	f1-score	support
0	0.99	0.98	0.98	561
1	0.98	0.99	0.98	502
accuracy			0.98	1063
macro avg	0.98	0.98	0.98	1063
weighted avg	0.98	0.98	0.98	1063

Summary of Model Performance for Loan Approval Prediction

> When looking at different ways to predict if loans will be approved or not, we found that the Random Forest and Gradient Boosting performedworked really well. It was accurate and could predict outcomes quite accurately. Decision tree model also did a good job.

> However, Support Vector Machine (SVM) and Naive Bayes didn't work well like the above models. They didn't predict as accurately as the Decision Tree and Random Forest models.

Saving the Gradient Boosting model

[131]:
```
import pickle

# Save the grid_search_rf model to a file using pickle
model_filename = 'grid_search_gb_model.pkl'
with open(model_filename, 'wb') as model_file:
    pickle.dump(grid_search_gb, model_file)
```

Loading the saved model

[132]:
```
# Load the saved model from the file
model_filename = 'grid_search_gb_model.pkl'
with open(model_filename, 'rb') as model_file:
    loaded_model = pickle.load(model_file)
```

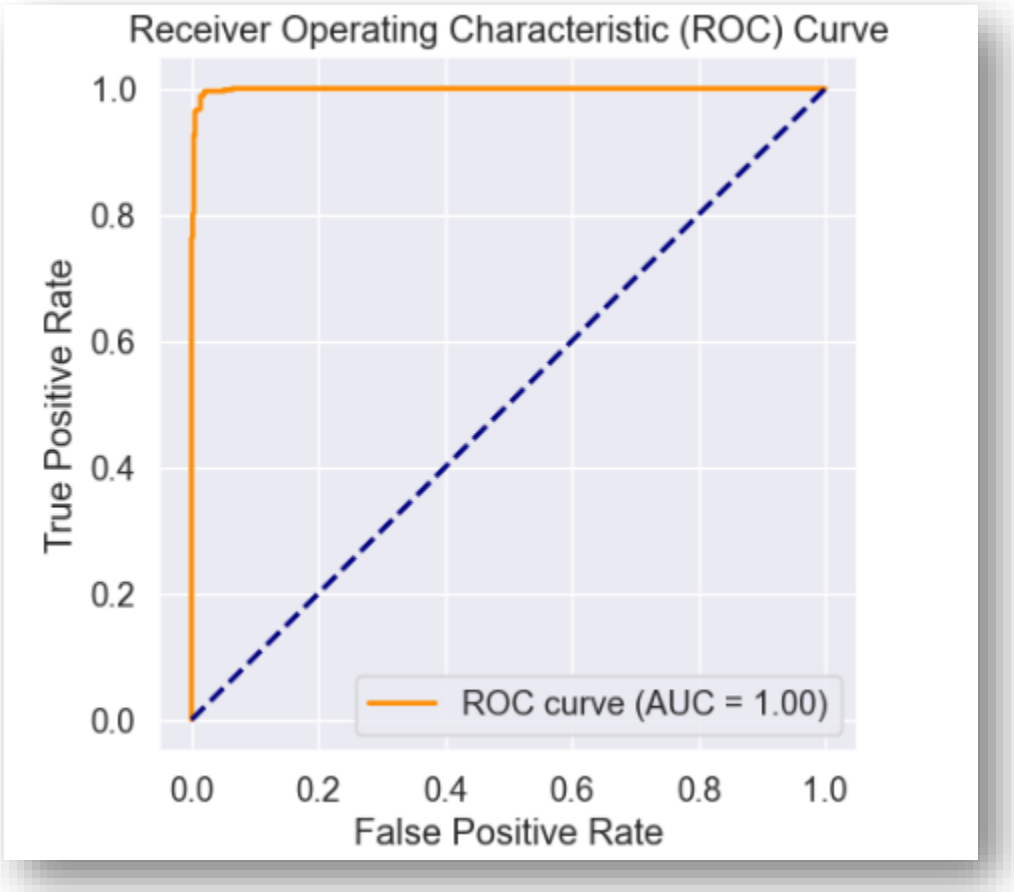
[133]:
```
loaded_model
```

[133]:
```
GridSearchCV(cv=5, estimator=GradientBoostingClassifier(),
             param_grid={'learning_rate': [0.01, 0.1, 0.2],
                         'max_depth': [3, 4, 5],
                         'n_estimators': [10, 50, 100]})
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

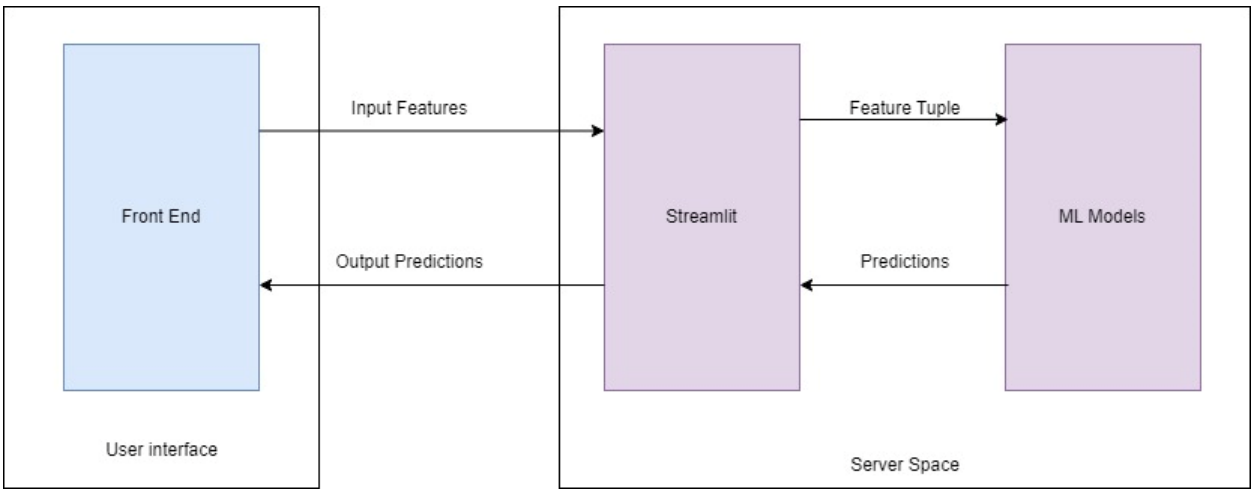
ROC curve for Gradient Boosting model

[134]:
```
from sklearn.metrics import roc_curve, auc

# Get predicted probabilities for the positive class
y_probabilities = loaded_model.predict_proba(X_test)[:, 1]

# Compute ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_test, y_probabilities)
roc_auc = auc(fpr, tpr)

# Plot the ROC curve
plt.figure(figsize=(5, 5))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

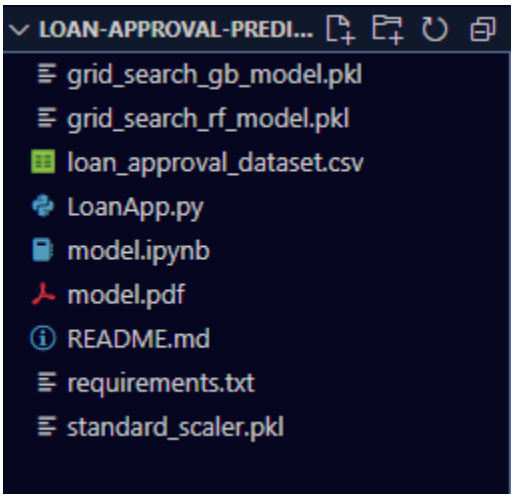


Deployment Implementation

The model was deployed using Python and Streamlit.



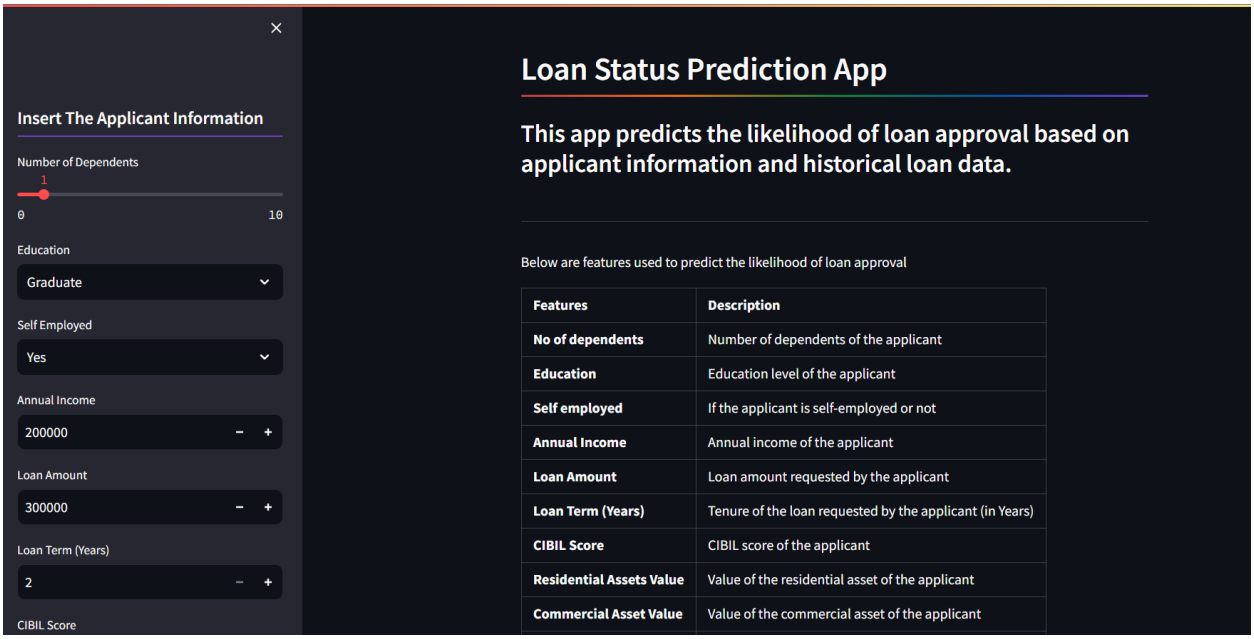
The Project Structure



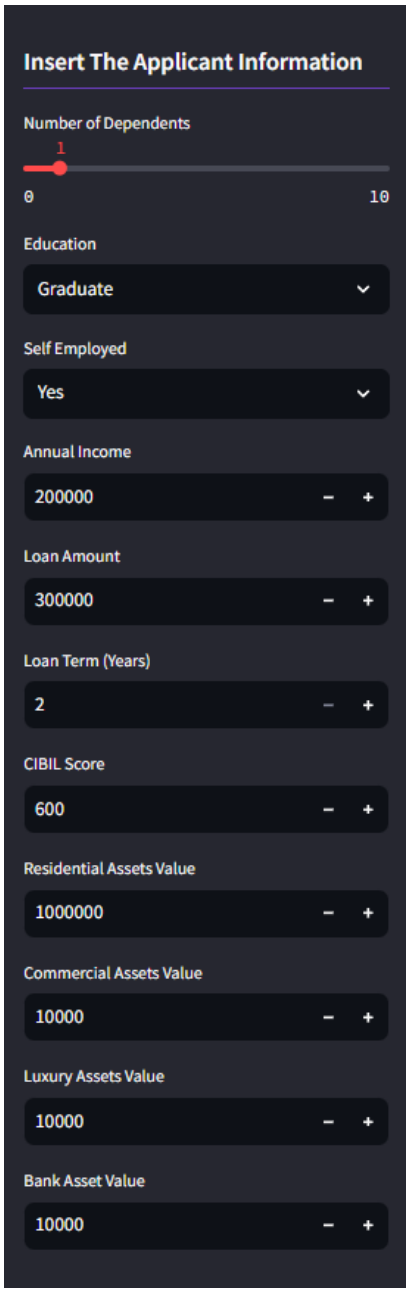
User Interface

The interface is created using Streamlit.

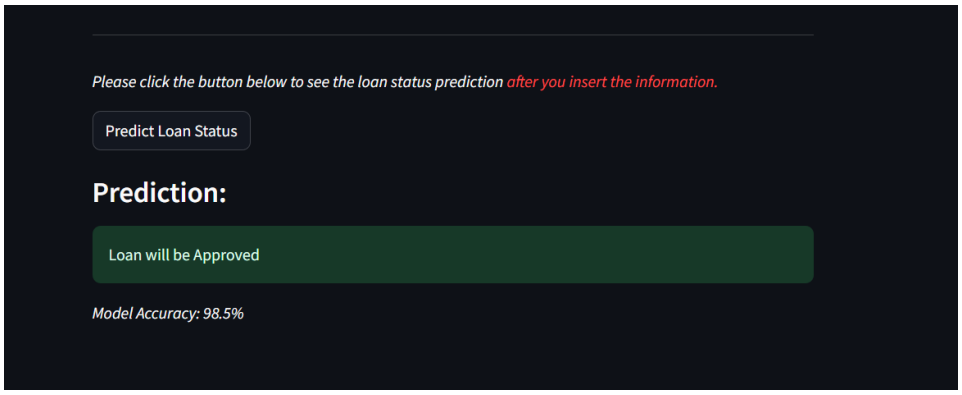
Link - https://loan-status.streamlit.app/



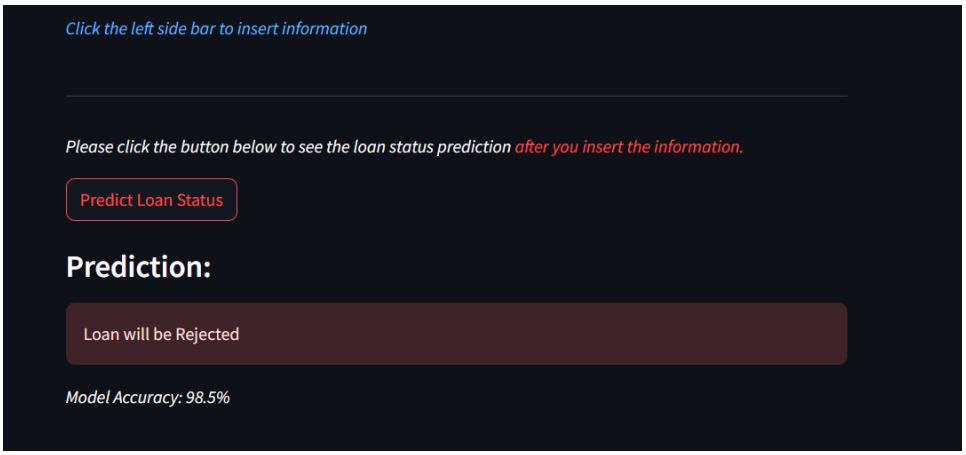
Input Tab



Loan Approval



Loan Rejection



Benefits of the Proposed Solution

Building a predictive model for loan approval offers a range of benefits for financial institutions and their customers. Here are some key advantages of such a solution:

1. Efficiency and Speed - Automated loan approval models can process applications much faster than manual underwriting. This reduces the waiting time for customers and can lead to a more responsive and competitive lending service.

2. Consistency - Predictive models provide consistent and standardized evaluation criteria, ensuring that every application is assessed using the same rules and factors. This minimizes the potential for bias or errors in the approval process.

3. Reduced Operational Costs - Automation of loan approval reduces the need for extensive manual labor, resulting in cost savings for financial institutions. This can also lead to a reduction in human errors, which may result in costly mistakes.

4. Risk Management - By using data-driven models, financial institutions can better assess the creditworthiness and risk associated with each applicant. This helps in making more informed lending decisions and reduces the likelihood of granting loans to high-risk borrowers.

5. Improved Accuracy - Predictive models can analyze a wide array of data, including credit history, income, employment, and other relevant factors, leading to a more accurate assessment of an applicant's creditworthiness. This reduces the likelihood of approving loans to individuals who may not be able to repay them.

6. Fairness and Bias Mitigation - Building fairness into predictive models is crucial. Institutions can carefully design and test models to minimize the potential for bias in lending decisions. This ensures that loans are approved or denied based on objective, non-discriminatory criteria, promoting social and regulatory compliance.

7. Better Customer Experience - Faster, more accurate loan approval processes result in improved customer experience. Applicants receive quicker responses, and those who are eligible for loans are likely to have a smoother and more positive interaction with the financial institution.

8. Data-Driven Decision-Making - Predictive models can provide insights into the most influential factors for loan approval, helping institutions fine-tune their lending criteria and strategies based on actual data rather than gut feeling or intuition.

9. Real-time Decision-Making - With automated loan approval models, decisions can be made in real-time, allowing for quick responses to changing market conditions and customer needs.

Conclusion

In conclusion, the implementation of a predictive model for loan approval offers multifaceted benefits that significantly enhance the efficiency, fairness, and overall performance of financial institutions. The advantages range from expediting the approval process and ensuring consistency to reducing operational costs and mitigating risks associated with lending. The utilization of data-driven models not only improves the accuracy of creditworthiness assessments but also facilitates better decision-making, allowing financial institutions to adapt and refine their lending strategies based on empirical insights.

Moreover, the incorporation of fairness measures into predictive models addresses concerns related to bias, ensuring that lending decisions are objective and compliant with social and regulatory standards. This not only promotes ethical lending practices but also establishes trust with customers and regulators. The improved customer experience resulting from faster, more accurate loan approvals contributes to enhanced satisfaction and loyalty among the clientele.

The scalability of predictive models accommodates the growing volume of loan applications without proportional increases in manpower, offering a sustainable solution for institutions experiencing expansion. Additionally, the real-time decision-making capability of automated models enables financial institutions to respond promptly to changing market conditions and customer demands, fostering agility and competitiveness.

Furthermore, adherence to regulatory compliance standards ensures legal conformity and reduces the risk of penalties or legal complications. In essence, the adoption of predictive models for loan approval aligns with the broader trend of digitization in the financial sector, providing institutions with a powerful tool to navigate the evolving landscape of lending while delivering optimal outcomes for both lenders and borrowers.

Project Team, Roles, and Responsibilities

IT Number	Name	Responsibilities
IT21053528	Arawwala D.J.S. S	Model Building Optimizing and tuning the Models. Deploying Model Designing the UI Testing the model
IT21053528	Weerasinghe C.C	Exploratory Data Analysis Data Collection Feature Engineering Data Preprocessing Testing the model
IT21077692	Kasthurirathne K.K. I	Feature Selection Feature Engineering Presentation Design Testing the model
IT21016066	Samaraweera G.P.M. D	Model Evaluations Data Visualization Documentation Designing the UI Testing the model