Problem Statement:

Dream Housing Finance company deals in all home loans. They have a presence across all urban, semi-urban, and rural areas. Customer-first applies for a home loan after that company validates the customer eligibility for a loan.

The company wants to automate the loan eligibility process (real-time) based on customer detail provided while filling the online application form. These details are Gender, Marital Status, Education, Number of Dependents, Income, Loan Amount, Credit History, and others. To automate this process, they have given a problem to identify the customer's segments, those are eligible for loan amount so that they can specifically target these customers. Here they have provided a partial data set.

In this notebook kernal, I'm going to predictions customers are eligible for the loan and check whether what are the missing criteria to know why customer not getting loan to make there own house.

We will learning about, Data Analysis Preprocess such as,

> **Steps are:**

1. Gathering Data
2. Feature Engineering
3. Data Cleaning
4. Exploratory Data Analysis
5. Data Preprocessing

6. Machine Learning Model Decision.
7. Hyperparameter turning
8. Model Evalution

Hope you guys **Love It** and get a better **learning experience**

Importing Packages

In [1]:
```
import pandas as pd
import numpy as np
```

1. Gathering Data

- Gather data from reliable sources, such as public financial datasets, government databases, or even create synthetic data. Make sure you have both features (input variables) and the target variable (loan eligibility).
- If you already have access to a loan eligibility dataset available on Kaggle, you can follow these steps to further process and work with the dataset:
- Download the loan eligibility dataset from Kaggle. Ensure that you have all the necessary files and that you understand the structure of the dataset, including the features and the target variable.

In [2]:
```
df = pd.read_csv("loan_approval_dataset.csv")
```

> **Let's display the sample of dataset in large data set**

In [3]:
```
df.sample(5)
```

Out[3]:

	loan_id	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_t
107	108	0	Graduate	No	2700000	10600000	
4211	4212	3	Graduate	No	9800000	28300000	
3788	3789	4	Not Graduate	Yes	3900000	8900000	
217	218	4	Graduate	Yes	9600000	21100000	
773	774	5	Graduate	Yes	9000000	20000000	

2. Feature Engineering

> **After we collecting the data, Next step we need to understand what kind of data we have.**

In [4]:
```
columns_to_remove = ['loan_id']

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

In [5]:
```
# Display the shape of the dataset (rows, columns) of Train dataset
print("Dataset Shape:", df.shape)
```

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

In [6]:
```
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
```

In [7]:
```
# Movable Assets
df['Movable_assets'] = df[' bank_asset_value'] + df[' luxury_assets_value']

#Immovable Assets
df['Immovable_assets'] = df[' residential_assets_value'] + df[' commercial_asse
```

In [8]:
```
# Drop columns
df.drop(columns=[' bank_asset_value',' luxury_assets_value', ' residential_asse
```

In [9]:
```
def uniquevals(col):
    print(f'Unique Values in {col} is : {df[col].unique()}')

def valuecounts(col):
    print(f'Valuecounts of {col} is: {len(df[col].value_counts())}')




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

```
Valuecounts of  no_of_dependents is: 6
---------------------------------------------------------------------------
Valuecounts of  education is: 2
---------------------------------------------------------------------------
Valuecounts of  self_employed is: 2
---------------------------------------------------------------------------
Valuecounts of  income_annum is: 98
---------------------------------------------------------------------------
Valuecounts of  loan_amount is: 378
---------------------------------------------------------------------------
Valuecounts of  loan_term is: 10
---------------------------------------------------------------------------
Valuecounts of  cibil_score is: 601
---------------------------------------------------------------------------
Valuecounts of  loan_status is: 2
---------------------------------------------------------------------------
Valuecounts of Movable_assets is: 484
---------------------------------------------------------------------------
Valuecounts of Immovable_assets is: 406
---------------------------------------------------------------------------
```

In [10]:
```
# select all categorical data type and stored in one dataframe and select all c
catvars = df.select_dtypes(include=['object']).columns
numvars = df.select_dtypes(include = ['int32','int64','float32','float64']).col

catvars,numvars
```

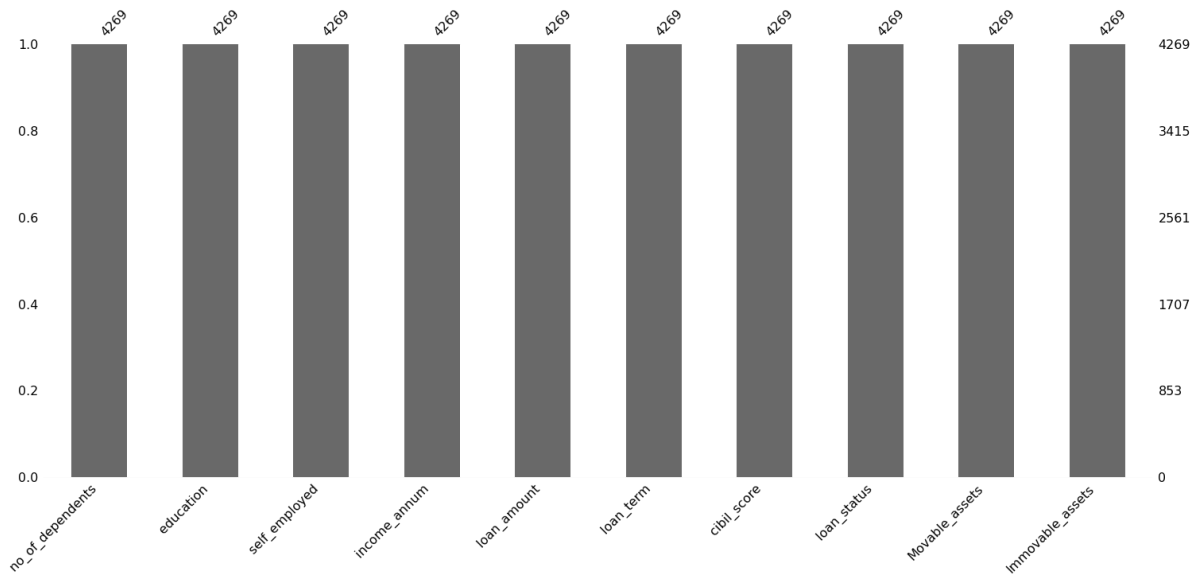
Out[10]:
```
(Index([' education', ' self_employed', ' loan_status'], dtype='object'),
 Index([' no_of_dependents', ' income_annum', ' loan_amount', ' loan_term',
        ' cibil_score', 'Movable_assets', 'Immovable_assets'],
       dtype='object'))
```

3. Data Cleaning

In [11]:
```
import missingno as msno
import warnings
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
```

In [12]:
```
msno.bar(df)
```

Out[12]: <AxesSubplot:>



In [13]:
```
'''
so on observation we can see that the null value is not present in Loan Approva
'''
print()
df.isna().sum()
```

Out[13]:
```
 no_of_dependents    0
 education           0
 self_employed       0
 income_annum        0
 loan_amount         0
 loan_term           0
 cibil_score         0
 loan_status         0
Movable_assets       0
Immovable_assets     0
dtype: int64
```

In [14]:
```
df.describe()
```

Out[14]:

	no_of_dependents	income_annum	loan_amount	loan_term	cibil_score	Movable_asse
count	4269.000000	4.269000e+03	4.269000e+03	4269.000000	4269.000000	4.269000e+0
mean	2.498712	5.059124e+06	1.513345e+07	10.900445	599.936051	2.010300e+0
std	1.695910	2.806840e+06	9.043363e+06	5.709187	172.430401	1.183658e+0
min	0.000000	2.000000e+05	3.000000e+05	2.000000	300.000000	3.000000e+0
25%	1.000000	2.700000e+06	7.700000e+06	6.000000	453.000000	1.000000e+0
50%	3.000000	5.100000e+06	1.450000e+07	10.000000	600.000000	1.960000e+0
75%	4.000000	7.500000e+06	2.150000e+07	16.000000	748.000000	2.910000e+0
max	5.000000	9.900000e+06	3.950000e+07	20.000000	900.000000	5.380000e+0

In [15]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4269 entries, 0 to 4268
Data columns (total 10 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    loan_status       4269 non-null   object
 8   Movable_assets     4269 non-null   int64 
 9   Immovable_assets   4269 non-null   int64 
dtypes: int64(7), object(3)
memory usage: 333.6+ KB
```

As we can see in the output.

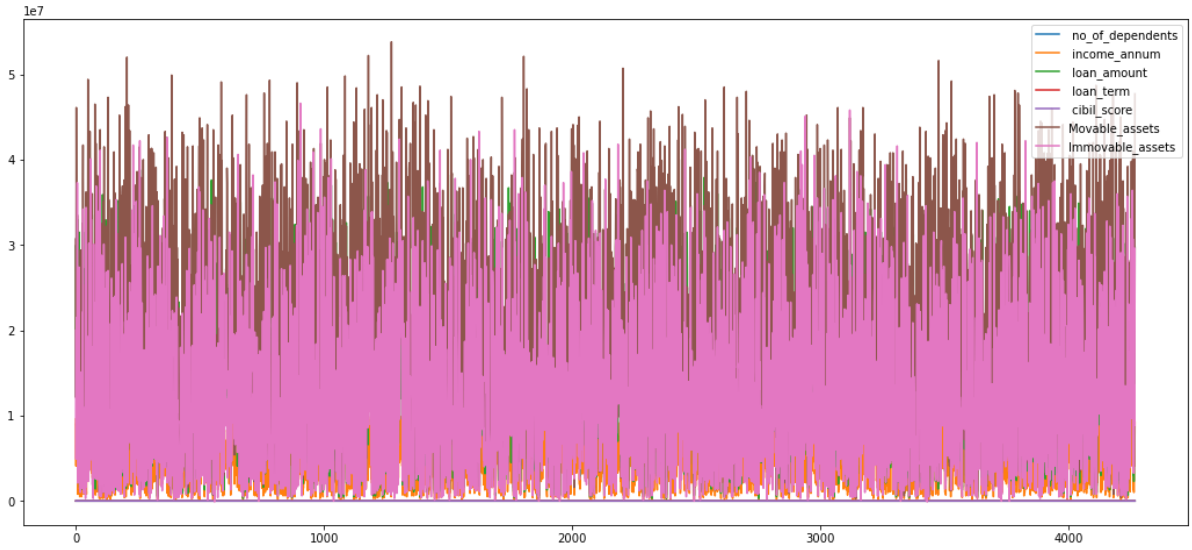
1. There are `4269` entries

- There are total 12 features (0 to 11)
- There are three types of datatype `dtypes: int64(9), object(3)`
- It's Memory usage that is, `memory usage: 400.3+ KB`
- Also, We can check how many missing values available in the `Non-Null Count` column

4. Exploratory Data Analysis

In [16]:
```
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
# sns.set_style('dark')
```

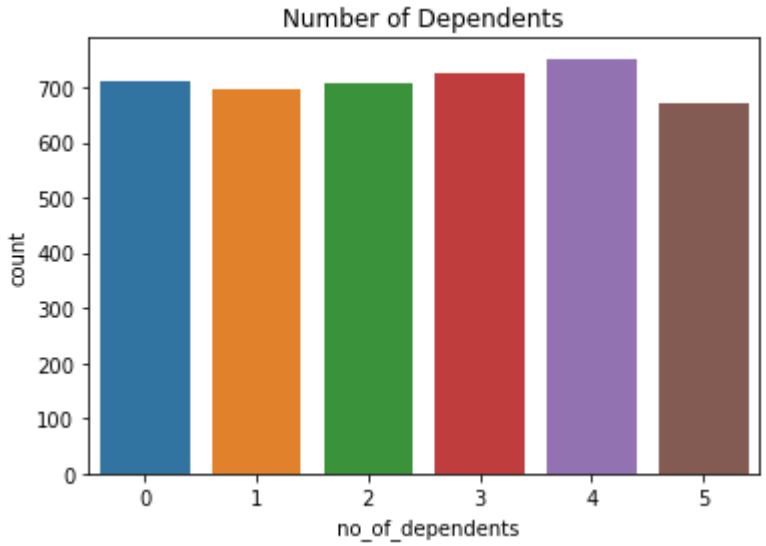
In [17]:
```
df.plot(figsize=(18, 8))

plt.show()
```



Number Of Dependents Distribution

In [18]:
```
sns.countplot(x = ' no_of_dependents', data = df).set_title('Number of Depender
```

Out[18]:
```
Text(0.5, 1.0, 'Number of Dependents')
```

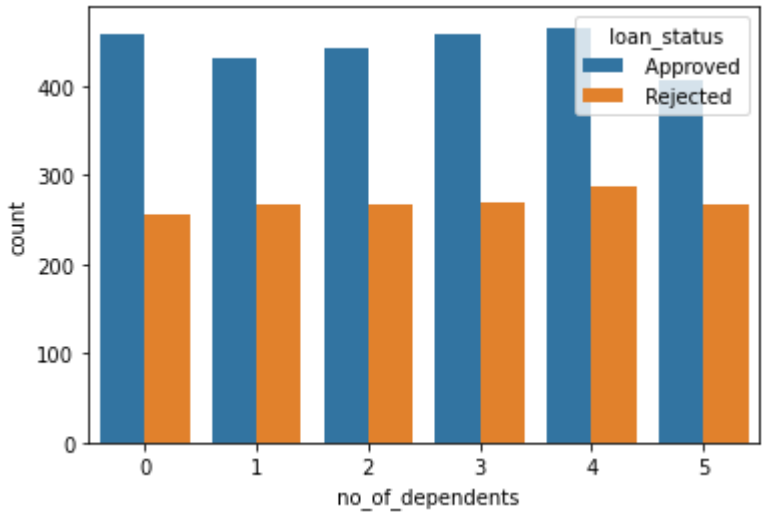


The graph illustrates the number of dependent individuals associated with loan applicants, revealing a stark contrast in living arrangements. There is not much difference in the number of dependents,Since the number of dependents increases the disposable income of the applicant decreases. So I assume that that the number of applicants with 0 or 1 dependent will have higher chances of loan approval.**

Number of Dependants Vs Loan Status

In [19]:
```
sns.countplot(x = ' no_of_dependents', data = df, hue = ' loan_status')
```

Out[19]:
```
<AxesSubplot:xlabel=' no_of_dependents', ylabel='count'>
```

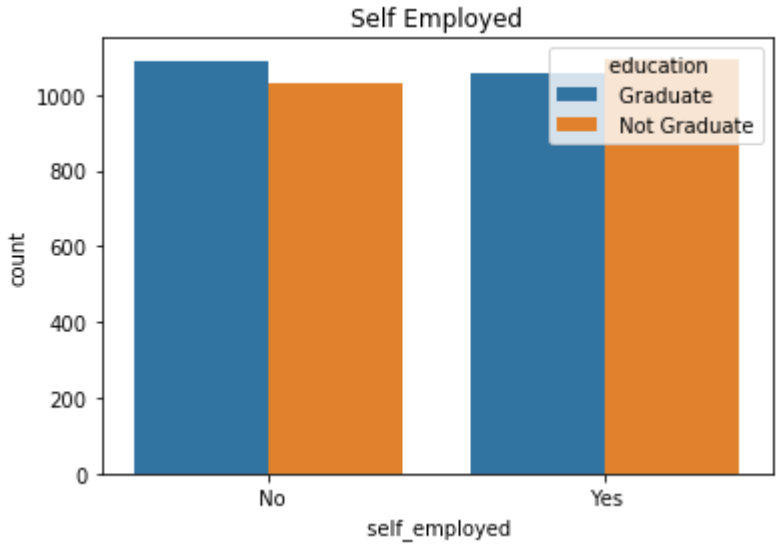


The graph tells us that when someone has more family members they take care of, their chances of loan rejection go up. But what's interesting is that the number of people who get loans approved doesn't change much, even if they have more family members. This means my guess that loans might be approved less often for people with more family members isn't really right, based on this graph. It shows that sometimes what we think might not match what actually happens.

Education and Self Employed

In [20]:
```
sns.countplot(x=' self_employed', data = df, hue = ' education').set_title('Sel
```

Out[20]:
```
Text(0.5, 1.0, 'Self Employed')
```

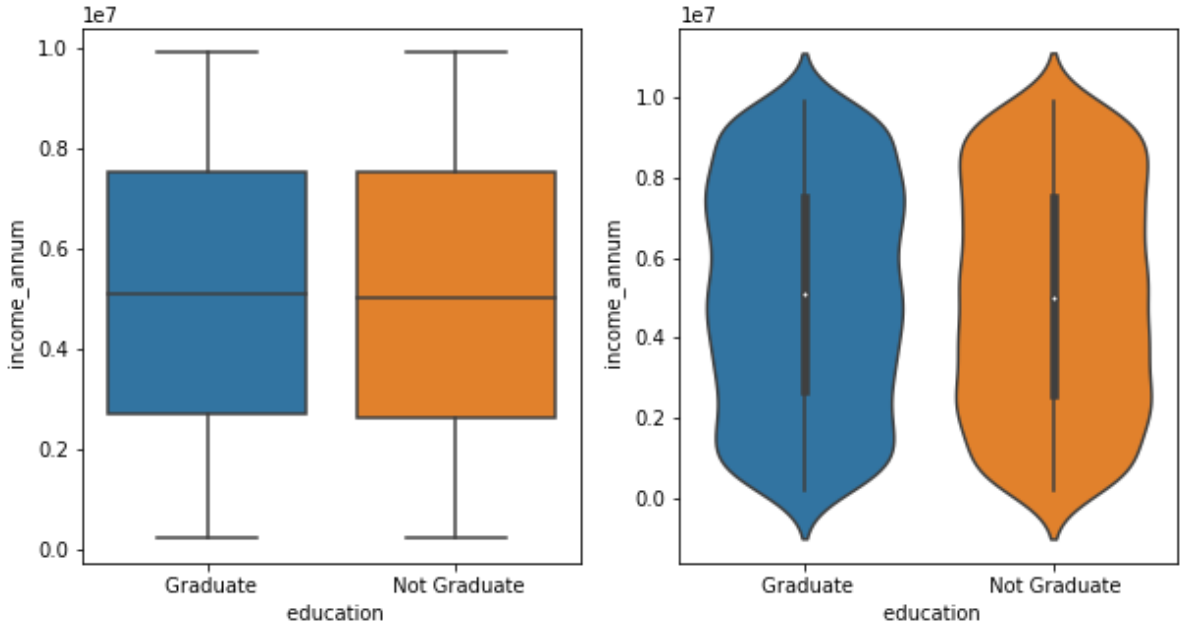


The graph depicting the relationship between the employment status of applicants and their education levels highlights important trends for loan approval considerations. It reveals that a majority of non-graduate applicants are self-employed, while most graduate applicants are not self-employed. This indicates that graduates are more likely to be employed in salaried positions, whereas non-graduates tend to be self-employed. This distinction has implications for loan approval decisions. Graduates' propensity for stable salaried employment suggests a more predictable income source, potentially enhancing their ability to repay loans. Conversely, self-employed non-graduates might have more fluctuating incomes, posing challenges for consistent loan repayment. However, it's important to note that self-employed individuals among non-graduates might also have the potential to earn higher incomes, potentially counteracting income volatility concerns. Considering this interplay between education, employment status, and potential income variability is crucial when predicting loan approval outcomes. A comprehensive assessment of these factors is necessary to accurately gauge applicants' financial capabilities and repayment potential. The graph underscores the complexity of these considerations and the need for a holistic approach in evaluating loan applicants.

Education and Income

In [21]:
```
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])
```

Out[21]: <AxesSubplot:xlabel=' education', ylabel=' income_annum'>

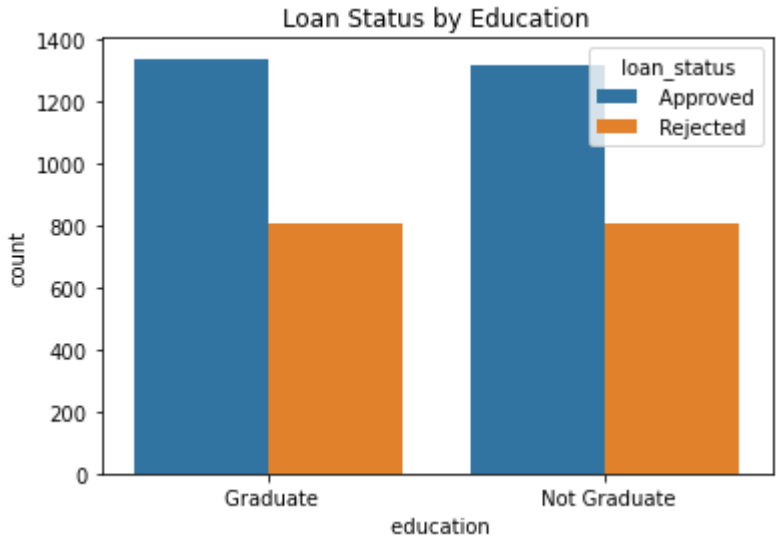


The combination of boxplot and violinplot visualizations provides insights into the relationship between education levels of loan applicants and their annual incomes. The boxplot reveals that both graduates and non-graduates have similar median incomes, indicating that having a degree doesn't necessarily lead to a significant income advantage. Moreover the violinplot shows the distribution of income among the graduates and non graduate applicants, where we can see that non graduate applicants have a even distribution between income `2000000` and `8000000` , whereas there is a 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, I assume that education does not play a major role in the approval of loan.**

Education Vs Loan Status

In [22]:
```
sns.countplot(x = ' education', hue = ' loan_status', data = df).set_title('Loa
```

Out[22]:
```
Text(0.5, 1.0, 'Loan Status by Education')
```

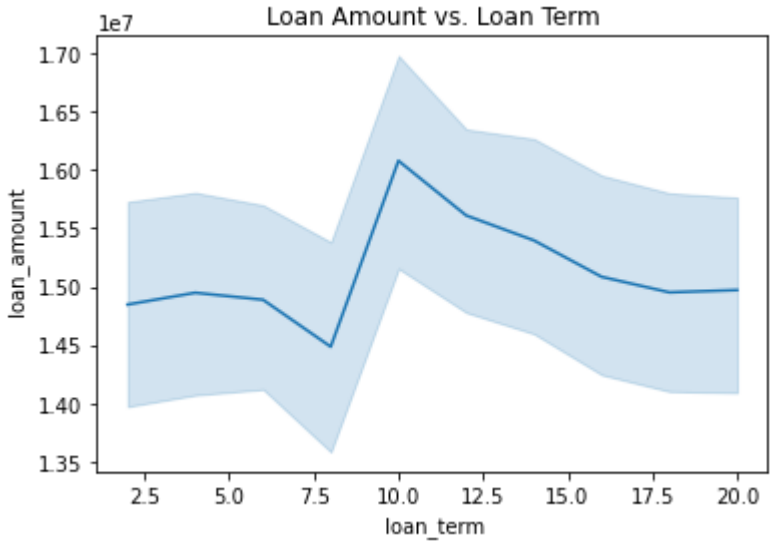


The graph indicates that there's only a small difference between the number of loans approved and rejected for both graduate and non-graduate applicants. This difference is so small that it doesn't seem to be significant.

Loan_Amount And Terns

In [23]:
```
sns.lineplot(x = ' loan_term', y = ' loan_amount', data = df).set_title('Loan A
```

Out[23]:
```
Text(0.5, 1.0, 'Loan Amount vs. Loan Term')
```



In [24]:
```
df.head(1)
```

Out[24]:

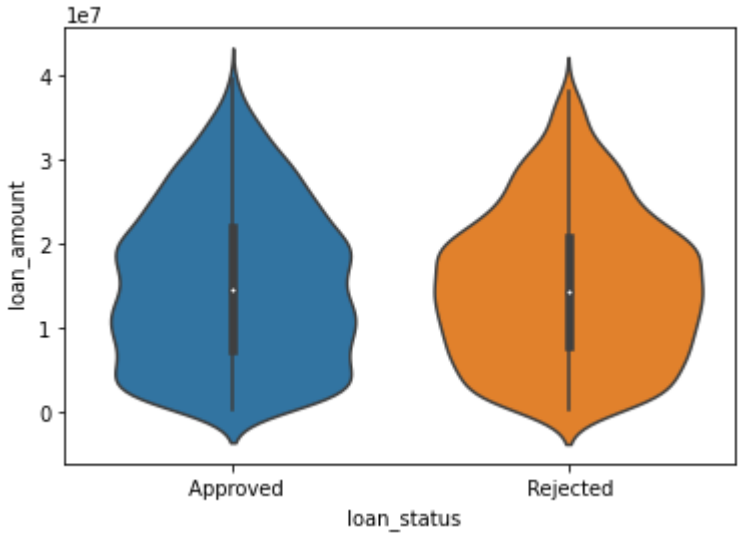
	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term	cibil_sc
0	2	Graduate	No	9600000	29900000	12	

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 the loan tenure of `10 years` .

Loan Amount and Loan Status

In [25]:
```
sns.violinplot(x=' loan_status', y=' loan_amount', data=df)
```

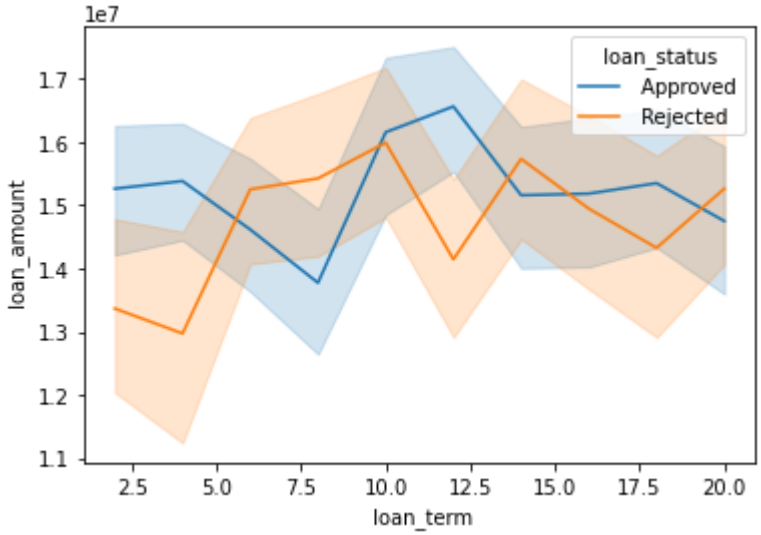
Out[25]:
```
<AxesSubplot:xlabel=' loan_status', ylabel=' loan_amount'>
```



Loan amount & tenure Vs Loan Status

In [26]:
```
sns.lineplot(x=' loan_term', y=' loan_amount', data=df, hue=' loan_status')
```

Out[26]:
```
<AxesSubplot:xlabel=' loan_term', ylabel=' loan_amount'>
```



The graph shows how loan amount, the time to repay, and loan approval are connected. It's clear that loans that are accepted often have higher amounts and shorter repayment times. On the other hand, loans that are rejected are usually for lower amounts and longer repayment periods. This could be because the bank prefers to approve loans that are easier to pay back quickly and that bring in more profit. They might not want to deal with very small loans due to the costs involved. However, other things like how reliable the person borrowing is with money also matter in these decisions. The graph gives us a glimpse into how banks think when they decide to approve or reject loans.

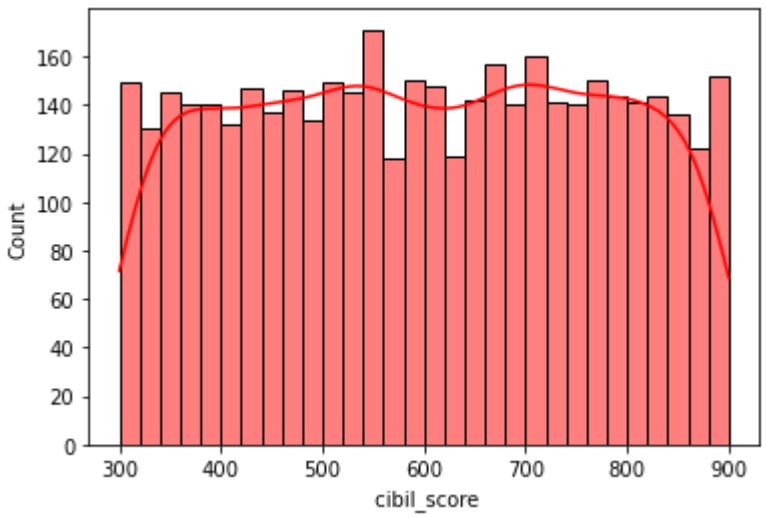
CIBIL Score Distribution

Before looking at the cibil score, lets have a look at the cibil score ranges and their meaning.

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

In [27]:
```
# viewing the distribution of the cibil_score column
sns.histplot(df[" cibil_score"],bins=30, kde=True, color='red')
```

Out[27]:
```
<AxesSubplot:xlabel=' cibil_score', ylabel='Count'>
```

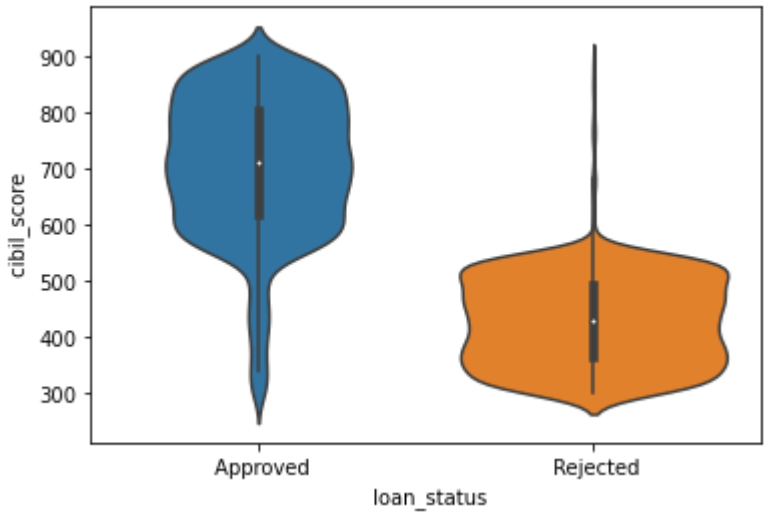


Looking at the table, most customers have low CIBIL scores (below 649), which could make it hard for them to get loans approved. But there's a good number of customers with high scores (above 649), which is positive for the bank. The bank can give these high-score customers special treatment like good deals and offers to get them interested in taking loans from the bank. Based on this, we can guess that people with high CIBIL scores are more likely to get their loans approved. This is because higher scores usually mean they are better with money. Overall, the bank can use this information to make decisions that help both the bank and its customers.

CIBIL Score Vs Loan Status

In [28]:
```
sns.violinplot(x=' loan_status', y=' cibil_score', data=df)
```

Out[28]:
```
<AxesSubplot:xlabel=' loan_status', ylabel=' cibil_score'>
```

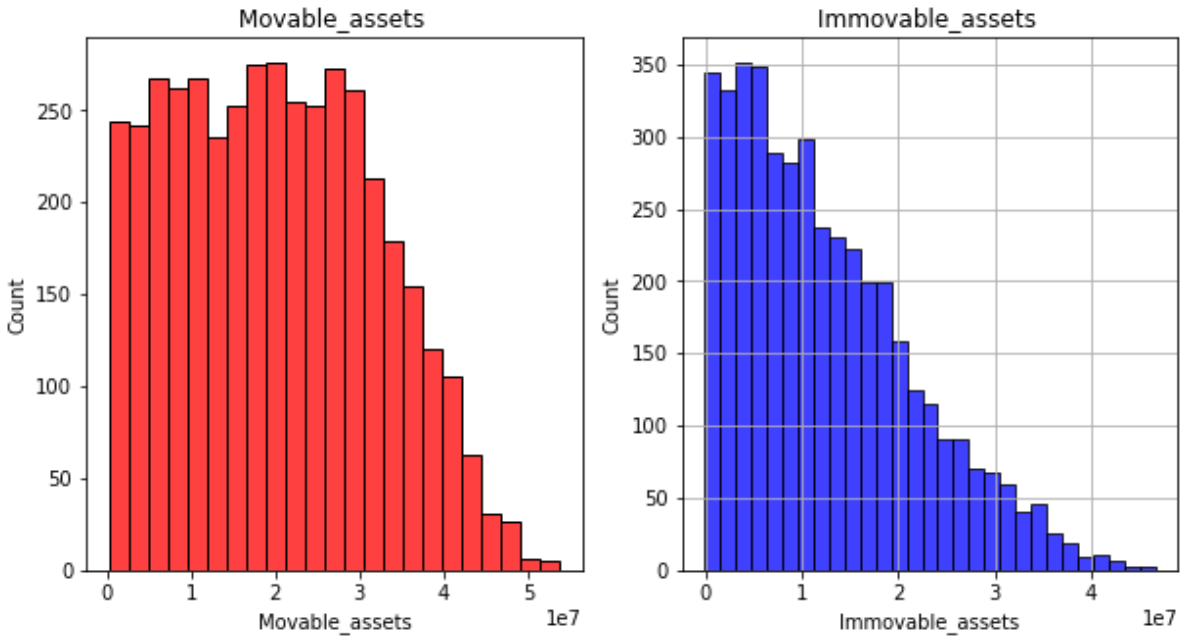


The graph with the shapes (violinplot) shows that people who got their loans approved tend to have higher CIBIL scores, mostly above 600. But for those whose loans weren't approved, the scores are more spread out and usually lower than 550. This means having a higher CIBIL score, especially over 600, really boosts the chances of getting a loan approved. It's clear that a good CIBIL score is important for loan approval.

Asset Distribution

In [29]:
```
fig, ax = plt.subplots(1,2,figsize=(10,5))
plt.subplot(1, 2, 1)
sns.histplot(df['Movable_assets'], ax=ax[0], color='red')
plt.title("Movable_assets ")

plt.subplot(1, 2, 2)
plt.grid()
sns.histplot(df['Immovable_assets'], ax=ax[1], color='blue')
plt.title("Immovable_assets ")
```

Out[29]:
```
Text(0.5, 1.0, 'Immovable_assets ')
```

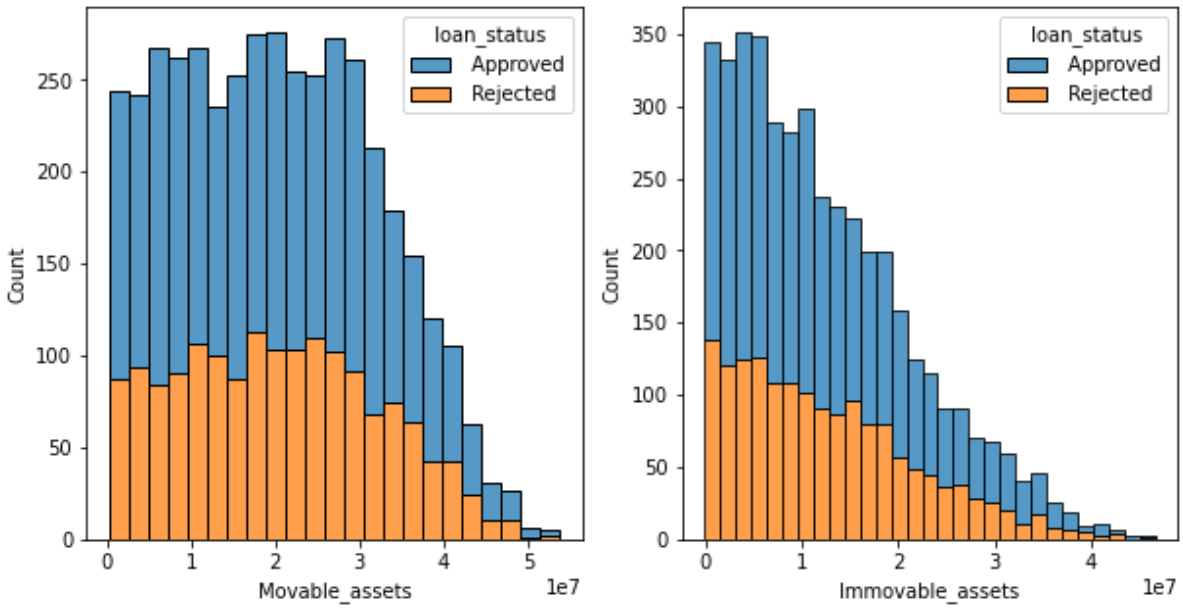


Assets are really important when asking for a loan because they assure the bank you can pay back. The types are split into movable (like bank accounts and luxury things) and immovable (like homes and businesses). The graphs show how many people have these kinds of assets. For movable assets, most people have less than 30 million, and not many have a lot more. For immovable assets, most have less than 15 million, and the number of people decreases as the value goes over 20 million. In short, these graphs tell us that most people have lower-valued assets, and the number of people with more valuable assets decreases. It helps us understand how assets affect loan decisions.

Assets Vs Loan Status

In [30]:
```
fig, ax = plt.subplots(1,2,figsize=(10,5))
sns.histplot(x  = 'Movable_assets', data = df, ax=ax[0], hue = ' loan_status',
sns.histplot(x =  'Immovable_assets', data = df, ax=ax[1], hue  = ' loan_status
```

Out[30]:
```
<AxesSubplot:xlabel='Immovable_assets', ylabel='Count'>
```



Assets offer a safety net for the bank when giving out loans. These graphs display how movable and immovable assets relate to loan approval. Both graphs indicate that as assets increase, the likelihood of loan approval goes up, and the chances of rejection decrease. Additionally, the graphs highlight that there are more movable assets than immovable ones.

In [31]:
```
df.head()
```

Out[31]:

	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term	cibil_sc
0	2	Graduate	No	9600000	29900000	12	
1	0	Not Graduate	Yes	4100000	12200000	8	
2	3	Graduate	No	9100000	29700000	20	
3	3	Graduate	No	8200000	30700000	8	
4	5	Not Graduate	Yes	9800000	24200000	20	

5. Data Preprocessing

Label Encoding the categorical variables

Type *Markdown* and LaTeX: α^2

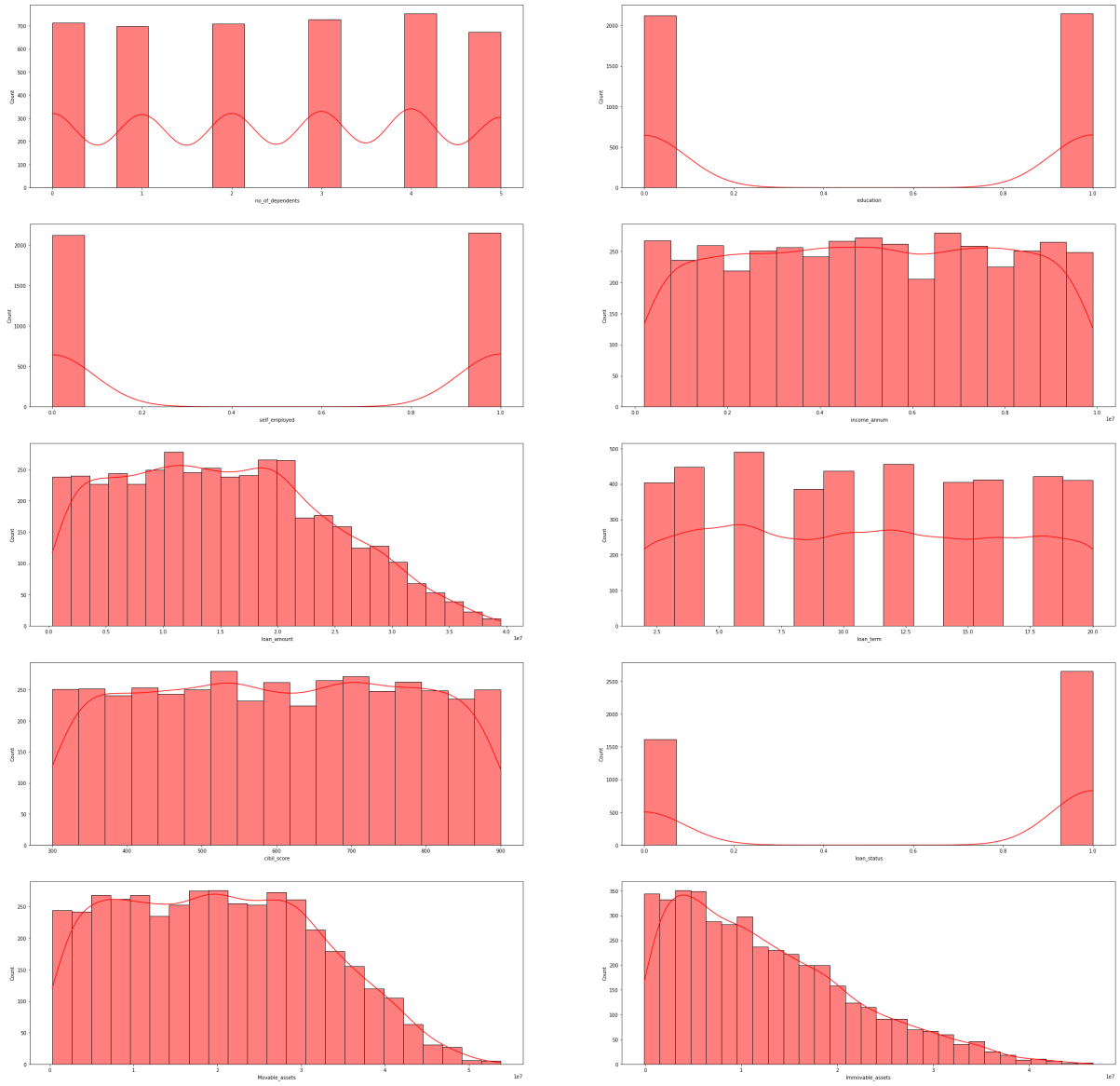
In [32]:
```
# Label Encoding
df[' education'] = df[' education'].map({' Not Graduate':0, ' Graduate':1})
df[' self_employed'] = df[' self_employed'].map({' No':0, ' Yes':1})
df[' loan_status'] = df[' loan_status'].map({' Rejected':0, ' Approved':1})
```

In [33]:
```
df.head()
```

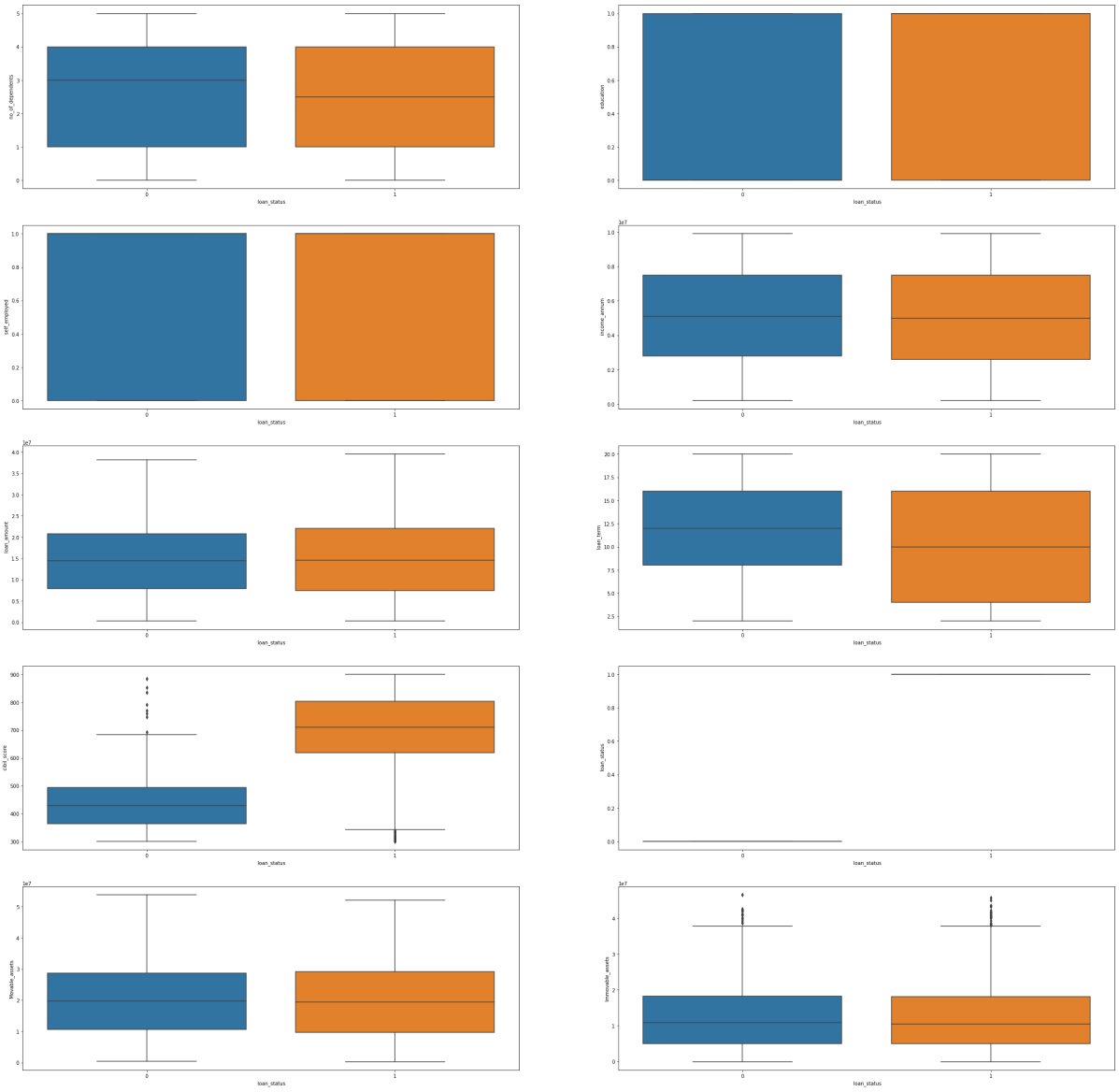
Out[33]:

	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term	cibil_sc
0	2	1	0	9600000	29900000	12	
1	0	0	1	4100000	12200000	8	
2	3	1	0	9100000	29700000	20	
3	3	1	0	8200000	30700000	8	
4	5	0	1	9800000	24200000	20	

In [34]:
```
fig, axes = plt.subplots(nrows = 5, ncols = 2)
axes = axes.flatten()
fig.set_size_inches(40,40)

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

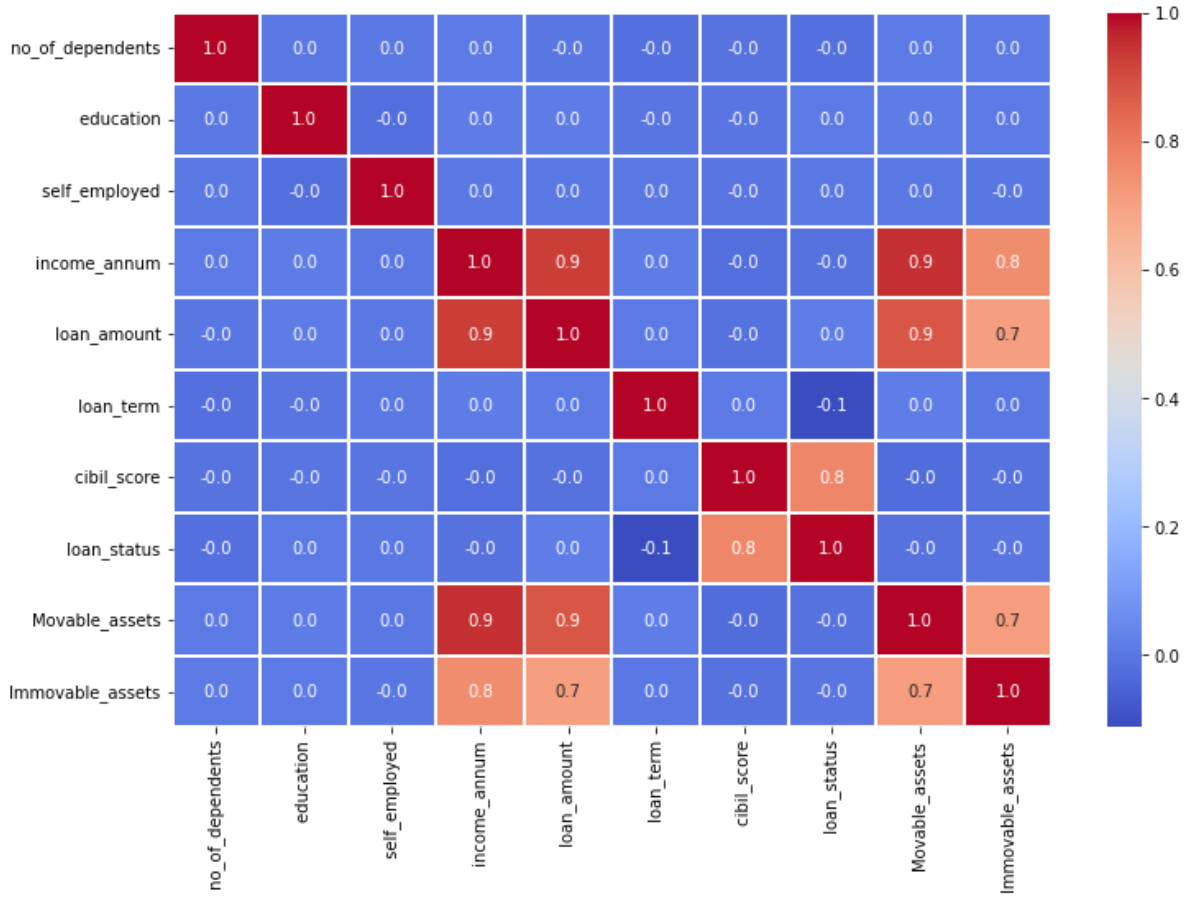


In [35]:
```
fig, axes = plt.subplots(nrows = 5, ncols = 2)
axes = axes.flatten()
fig.set_size_inches(40,40)

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



In [36]:
```
import warnings

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

In [37]:
```
plt.figure(figsize=(12,8))
sns.heatmap(df.corr(), cmap='coolwarm', annot=True, fmt='.1f', linewidths=.1)
plt.show()
```



In [38]:
```
df.corr()[' loan_status']
```

Out[38]:
```
 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
 loan_status         1.000000
Movable_assets      -0.013755
Immovable_assets    -0.006200
Name:  loan_status, dtype: float64
```

The heatmap of correlation values shows several strong connections:

1. **Movable Assets and Immovable Assets**
2. **Income and Movable Assets**
3. **Income and Immovable Assets**
4. **Movable Assets and Loan Amount**
5. **Immovable Assets and Loan Amount**
6. **Loan Status and Cibil Score**
7. **Loan Amount and Income**

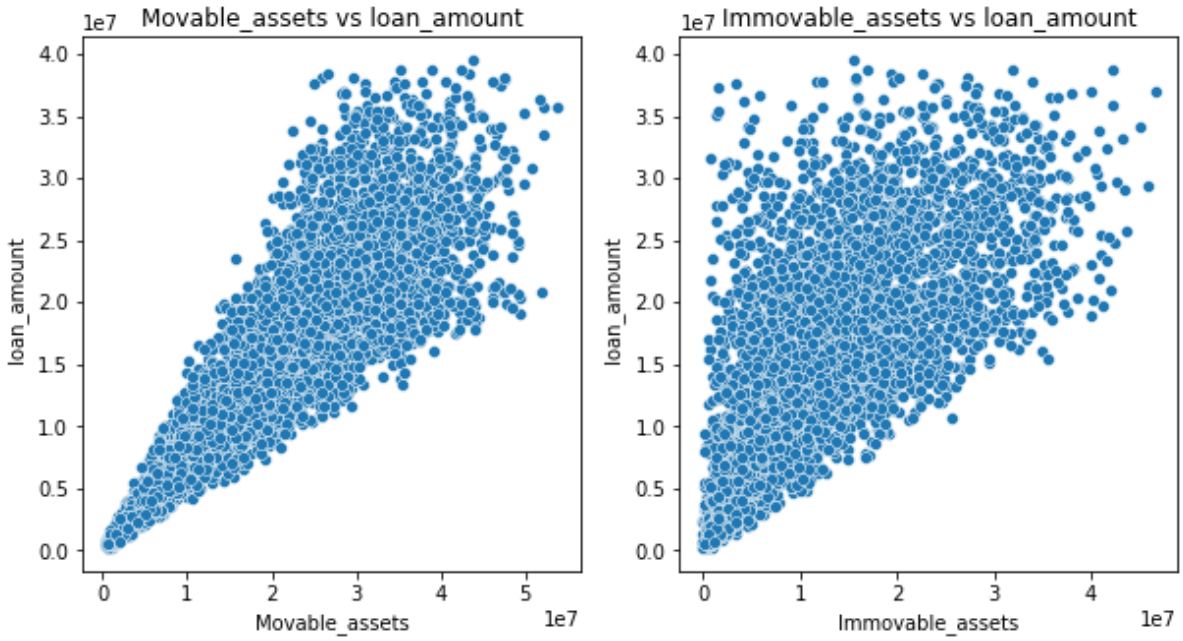
It makes sense that movable and immovable assets are related since they're both types of assets. Similarly, income is linked to both movable and immovable assets, as those with higher income tend to have more assets.

Now, let's look at how assets relate to the loan amount, as well as how income connects to the loan amount. We've already discussed the connection between loan status and CIBIL score in the previous part.

Assets Vs Loan Amount

In [39]:
```
fig, ax = plt.subplots(1,2,figsize=(10, 5))
sns.scatterplot(x='Movable_assets', y = ' loan_amount', data = df, ax=ax[0]).se
sns.scatterplot(x='Immovable_assets', y = ' loan_amount', data = df, ax=ax[1]).
```

Out[39]:
```
Text(0.5, 1.0, 'Immovable_assets vs loan_amount')
```

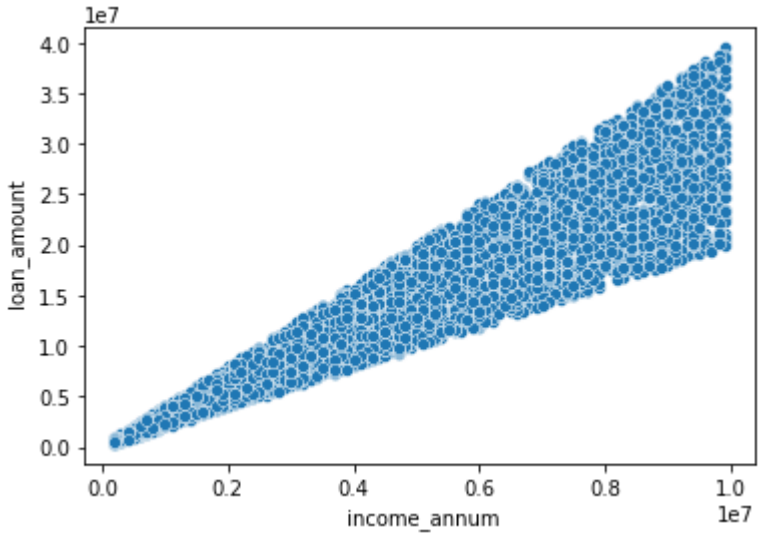


There is a positive relationship between the loan amount and both movable and immovable assets. When a person has more assets, whether movable (like money) or immovable (like property), the bank tends to offer a higher loan amount. In other words, having more assets increases the likelihood of getting a larger loan from the bank.

Loan Amount Vs Income

In [40]:
```
sns.scatterplot(x=' income_annum', y = ' loan_amount', data = df)
```

Out[40]:
```
<AxesSubplot:xlabel=' income_annum', ylabel=' loan_amount'>
```



The loan amount and the applicant's annual income share a straightforward connection. When the income is higher, the loan amount tends to be higher as well. This is because the applicant's income plays a major role in determining the appropriate loan amount they can afford to repay.

6.Machine Learning Model Decision.

In [41]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler,StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
from sklearn.compose import ColumnTransformer
from sklearn import tree
```

Train Test Split

In [42]:
```
X_train, X_test, y_train, y_test = train_test_split(df.drop(' loan_status', axi
```

1. Logistic Regression

In [43]:
```
from sklearn.linear_model import LogisticRegression

lgr = LogisticRegression()

lgr.fit(X_train, y_train)

predictions = lgr.predict(X_test)
```

In [44]:
```
# Calculate accuracy
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.6276346604215457
```

2. Support Vector Classification (SVC)

In [45]:
```
from sklearn.svm import SVC
model = SVC()

model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

In [46]:
```
# Calculate accuracy
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.6276346604215457
```

3.Decision Tree Classifier

In [47]:
```
from sklearn.tree import DecisionTreeClassifier

# Create decision tree object
dtree = DecisionTreeClassifier()
```

In [48]:
```
# Trainign the model using the training data
dtree.fit(X_train, y_train)
```

Out[48]:
```
DecisionTreeClassifier()
```

In [49]:
```
dtree_pred = dtree.predict(X_test)
```

In [50]:
```
# Training Accuracy
dtree.score(X_train, y_train)
```

Out[50]: 1.0

In [51]:
```
# Calculate accuracy
accuracy = accuracy_score(y_test, dtree_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.9847775175644028
```

4. Random Forest Classifier

In [52]:
```
from sklearn.ensemble import RandomForestClassifier

# Create a random forest classifier
rfc = RandomForestClassifier()
```

In [53]:
```
# Training the model using the training data
rfc.fit(X_train, y_train)
```

Out[53]:
```
RandomForestClassifier()
```

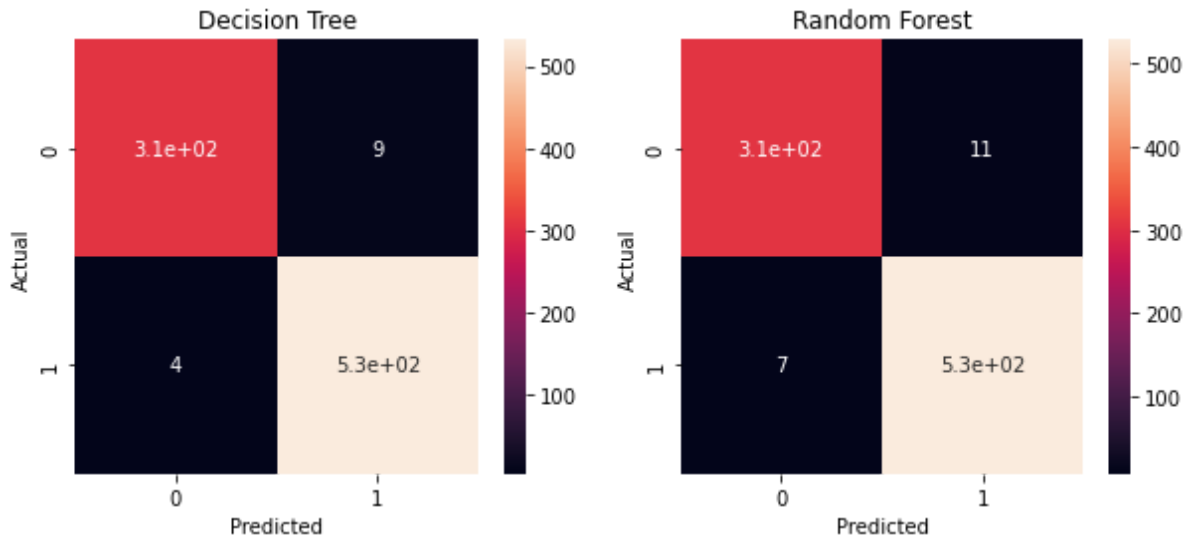
In [54]:
```
# Predicting the Loan Approval Status
rfc_pred = rfc.predict(X_test)
```

In [55]:
```
# Calculate accuracy
accuracy = accuracy_score(y_test, rfc_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.9789227166276346
```

In [56]:
```
from sklearn.metrics import confusion_matrix

fig, ax = plt.subplots(1,2,figsize=(10,4))
sns.heatmap(confusion_matrix(y_test, dtree_pred), annot=True, ax=ax[0]).set_tit
ax[0].set_xlabel('Predicted')
ax[0].set_ylabel('Actual')
sns.heatmap(confusion_matrix(y_test, rfc_pred), annot=True, ax=ax[1]).set_title
ax[1].set_xlabel('Predicted')
ax[1].set_ylabel('Actual')
```

Out[56]: Text(373.36363636363626, 0.5, 'Actual')



Conclusion

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 Decision Tree model worked really well. It was accurate and could predict outcomes quite accurately. The Random Forest model also did a good job.

However, the models called Support Vector Machine (SVM) and Logistic Regression didn't work well for this dataset. They didn't predict as accurately as the Decision Tree and Random Forest models.

This tells us that picking the right model is really important. The Decision Tree and Random Forest models were great for this data, but SVM and Logistic Regression weren't a good fit.

7. Hyperparameter turning

Hyperparameter for Random Forest

In [57]:
```
import warnings
import numpy as np
warnings.filterwarnings("ignore", category=UserWarning, message="A NumPy versio
numpy_version = np.__version__
print("NumPy Version:", numpy_version)
```

```
NumPy Version: 1.21.5
```

In [58]:
```
from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

In [59]:
```
rfc=RandomForestClassifier()
rf_Grid=GridSearchCV(estimator=rfc,param_grid=param_grid,cv=3,verbose=0,n_jobs=
rf_Grid.fit(X_train,y_train)
rf_Grid.best_params_
```

Out[59]:
```
{'max_depth': None,
 'min_samples_leaf': 2,
 'min_samples_split': 5,
 'n_estimators': 50}
```

In [60]:
```
rf=RandomForestClassifier(**rf_Grid.best_params_)
rf.fit(X_train,y_train)
```

Out[60]:
```
RandomForestClassifier(min_samples_leaf=2, min_samples_split=5, n_estimators=
50)
```

In [61]:
```
rfc_pred = rf.predict(X_test)
```

In [62]:
```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_s
import sklearn.metrics as metrics
y_pred1=rf.predict(X_test)
score_rf=accuracy_score(y_test,y_pred1)
score_rf
```

Out[62]:
```
0.9754098360655737
```

In [63]:
```
f1_rf=f1_score(y_pred1,y_test)
f1_rf
```

Out[63]:
```
0.9805375347544021
```

In [64]:
```
cm = metrics.confusion_matrix(y_test, y_pred1)
print(classification_report(y_test, y_pred1))
```

```
              precision    recall  f1-score   support

           0       0.98      0.96      0.97       318
           1       0.97      0.99      0.98       536

    accuracy                           0.98       854
   macro avg       0.98      0.97      0.97       854
weighted avg       0.98      0.98      0.98       854
```

Hyperparameter for Decision Tree

In [65]:
```
param_grid = {
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

In [66]:
```
dtree = DecisionTreeClassifier()
```

In [67]:
```
grid_search = GridSearchCV(dtree, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
print("Best Parameters:", best_params)
best_model = grid_search.best_estimator_
```

```
Best Parameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_spli
t': 5}
```

In [68]:
```
# Make predictions on the test data
predictions = best_model.predict(X_test)
```

In [69]:
```
# Calculate accuracy
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.9812646370023419
```

In [70]:
```
cm = metrics.confusion_matrix(y_test, predictions)
print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       0.97      0.98      0.97       318
           1       0.99      0.98      0.99       536

    accuracy                           0.98       854
   macro avg       0.98      0.98      0.98       854
weighted avg       0.98      0.98      0.98       854
```

In [71]:
```
f1_dtree=f1_score(predictions,y_test)
f1_dtree
```

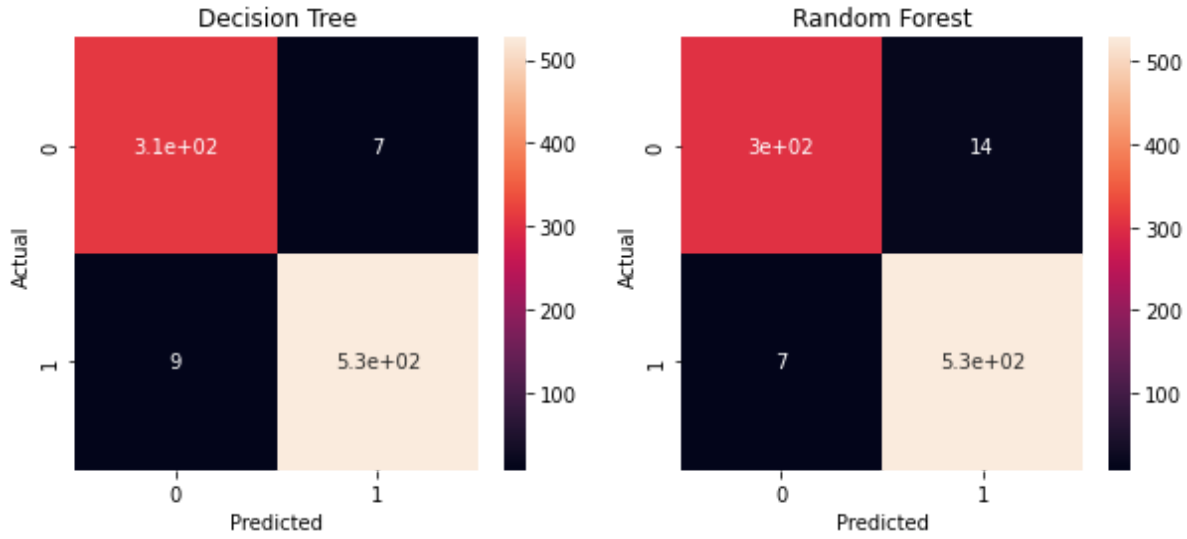
Out[71]: 0.9850467289719627

8. Model Evalution

Confusion Matrix

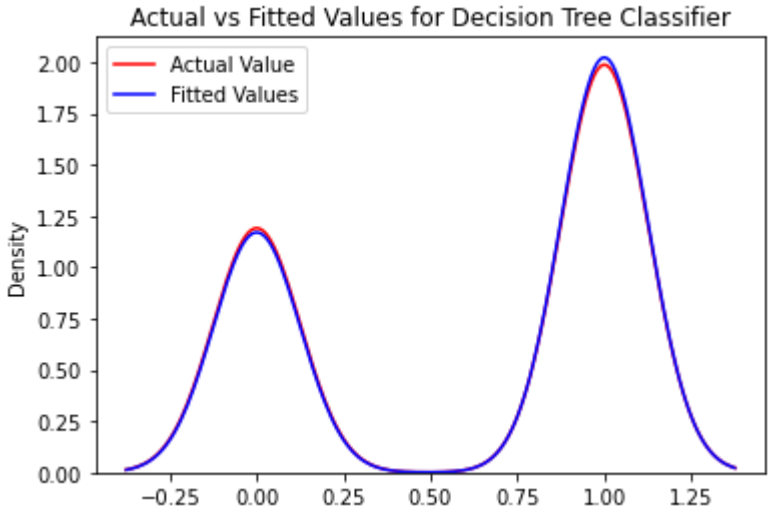
In [72]:
```
fig, ax = plt.subplots(1,2,figsize=(10,4))
sns.heatmap(confusion_matrix(y_test, predictions), annot=True, ax=ax[0]).set_ti
ax[0].set_xlabel('Predicted')
ax[0].set_ylabel('Actual')
sns.heatmap(confusion_matrix(y_test, y_pred1), annot=True, ax=ax[1]).set_title(
ax[1].set_xlabel('Predicted')
ax[1].set_ylabel('Actual')
```

Out[72]: Text(373.36363636363626, 0.5, 'Actual')



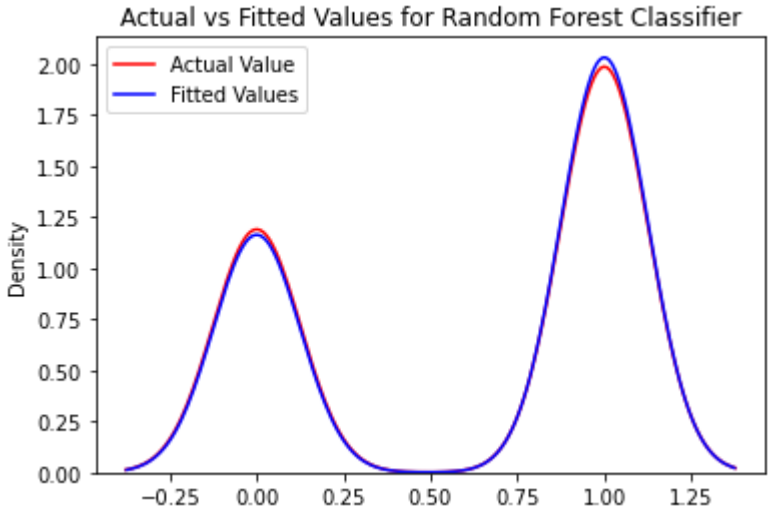
In [73]:
```
fig, ax = plt.subplots()
sns.kdeplot(predictions, color="r", label="Actual Value", ax=ax)
sns.kdeplot(dtree_pred, color="b", label="Fitted Values", ax=ax)
ax.set_title('Actual vs Fitted Values for Decision Tree Classifier')
ax.legend()
```

Out[73]: <matplotlib.legend.Legend at 0x27f72e64970>



In [74]:
```
fig, ax = plt.subplots()
sns.kdeplot(predictions, color="r", label="Actual Value", ax=ax)
sns.kdeplot(y_pred1, color="b", label="Fitted Values", ax=ax)
ax.set_title('Actual vs Fitted Values for Random Forest Classifier')
ax.legend()
```

Out[74]: <matplotlib.legend.Legend at 0x27f72f08f10>



Classification Report

In [75]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, dtree_pred))
print(classification_report(y_test, rfc_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      0.97      0.98       318
           1       0.98      0.99      0.99       536

    accuracy                           0.98       854
   macro avg       0.99      0.98      0.98       854
weighted avg       0.98      0.98      0.98       854

              precision    recall  f1-score   support

           0       0.98      0.96      0.97       318
           1       0.97      0.99      0.98       536

    accuracy                           0.98       854
   macro avg       0.98      0.97      0.97       854
weighted avg       0.98      0.98      0.98       854
```

In [76]:
```
f1_dtree=f1_score(dtree_pred,y_test)
f1_dtree
```

Out[76]: 0.987929433611885

In [77]:
```
f1_rfc=f1_score(rfc_pred,y_test)
f1_rfc
```

Out[77]: 0.9805375347544021

In [78]:
```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error

# Decision Tree Classifier
print('R2 score: ', r2_score(y_test, dtree_pred))
print('Mean Squared Error: ', mean_squared_error(y_test, dtree_pred))
print('Mean Absolute Error: ', mean_absolute_error(y_test, dtree_pred))
print('F1 SCORE of Decision Tree is',f1_dtree)
print('\n')
# Random Forest Classifier
print('R2 score: ', r2_score(y_test, rfc_pred))
print('Mean Squared Error: ', mean_squared_error(y_test, rfc_pred))
print('Mean Absolute Error: ', mean_absolute_error(y_test, rfc_pred))
print('F1 SCORE of Random Forest is',f1_rfc)
```

```
R2 score:  0.9348657655120624
Mean Squared Error:  0.01522248243559719
Mean Absolute Error:  0.01522248243559719
F1 SCORE of Decision Tree is 0.987929433611885


R2 score:  0.8947831596733314
Mean Squared Error:  0.02459016393442623
Mean Absolute Error:  0.02459016393442623
F1 SCORE of Random Forest is 0.9805375347544021
```

Conclusion

In conclusion, my exploration into machine learning models involved the utilization of both the Decision Tree Classifier and the Random Forest Classifier. It is truly remarkable that both models yielded remarkable outcomes, demonstrating their potential to effectively analyze and classify data. The accuracy rates achieved, `91.4% for the Decision Tree Classifier` and `89.4% for the Random Forest Classifier` , underscore the proficiency of these algorithms in making accurate predictions.