Data Preprocessing

About dataset:

- The loan approval dataset is a compilation of financial statements and related data utilized to assess the qualifications of individuals or businesses seeking loans from a financial institution.
- It consists of multiple factors of the applicant such as number of dependents, education, employment status, annual income, amount of loan required, termn of the loan in years, credit score, value of residential, commercial, luxury, bank assets.
- It consists of 4269 rows and 13 columns.

- Importing required libraries like pandas, numpy, seaborn, matplotlib.pyplot.
- warnings.filterwarnings("ignore") ignores any warnings that might be generated whie executing th code.
- df =pd.read_csv("loan_approval_dataset.csv") reads the csv file.

In [2]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

import warnings

warnings.filterwarnings("ignore")

df =pd.read_csv("loan_approval_dataset.csv")
```

In [2]:
```
df.head()
```

Out[2]:

	loan_id	no_of_dependents	education	self_employed	income_annum	loan_amount	loa
0	1	2	Graduate	No	9600000	29900000	
1	2	0	Not Graduate	Yes	4100000	12200000	
2	3	3	Graduate	No	9100000	29700000	
3	4	3	Graduate	No	8200000	30700000	
4	5	5	Not Graduate	Yes	9800000	24200000	

- df.describe(include='all') is used to get both numerical and non-numerical summary of the dataset

- we can see the number of non null, unique values and the most frequent value along with the frequency of the most frequent value.
- we can also see the mean, standard deviation, minimum and maximun value, 25% 50% and 75% quartile.

In [3]:
```
df.describe(include='all')
```

Out[3]:

	loan_id	no_of_dependents	education	self_employed	income_annum	loan_an
count	4269.000000	4269.000000	4269	4269	4.269000e+03	4.26900
unique	NaN	NaN	2	2	NaN	
top	NaN	NaN	Graduate	Yes	NaN	
freq	NaN	NaN	2144	2150	NaN	
mean	2135.000000	2.498712	NaN	NaN	5.059124e+06	1.51334
std	1232.498479	1.695910	NaN	NaN	2.806840e+06	9.04336
min	1.000000	0.000000	NaN	NaN	2.000000e+05	3.00000
25%	1068.000000	1.000000	NaN	NaN	2.700000e+06	7.70000
50%	2135.000000	3.000000	NaN	NaN	5.100000e+06	1.45000
75%	3202.000000	4.000000	NaN	NaN	7.500000e+06	2.15000
max	4269.000000	5.000000	NaN	NaN	9.900000e+06	3.95000

- df.info() is used to get the structure of the dataset.
- There are NO non null values in the dataset.
- out of the 13 columns, 10 columns have int64 datatype and 3 columns have object datatype.
- the total memory used by the dataset is 433.7+ KB.

In [4]:
```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4269 entries, 0 to 4268
Data columns (total 13 columns):
 #   Column                     Non-Null Count  Dtype 
---  ------                     --------------  ----- 
 0   loan_id                    4269 non-null   int64 
 1    no_of_dependents          4269 non-null   int64 
 2    education                 4269 non-null   object
 3    self_employed             4269 non-null   object
 4    income_annum              4269 non-null   int64 
 5    loan_amount               4269 non-null   int64 
 6    loan_term                 4269 non-null   int64 
 7    cibil_score               4269 non-null   int64 
 8    residential_assets_value  4269 non-null   int64 
 9    commercial_assets_value   4269 non-null   int64 
 10   luxury_assets_value       4269 non-null   int64 
 11   bank_asset_value          4269 non-null   int64 
 12   loan_status               4269 non-null   object
dtypes: int64(10), object(3)
memory usage: 433.7+ KB
None
```

- Creating a new dataframe and dropping the column 'loan_id.
- Loan id is just used to identity the entity so it can be omitted whilen analyzing the data and training the model.
- Removing the unwanted spaces in the names of the columns.

Loan id is just used to identity the entity so it can be omitted whilen analyzing the data and training the model

In [4]:
```
df1 = df.drop(['loan_id'],axis=1)

df1.columns = df1.columns.str.replace(' ','')

df1.head()
```

Out[4]:

	no_of_dependents	education	self_employed	income_annum	loan_amount	loan_term
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

- Using df1.isnull().sum() we can see there are no null values in dataframe df1

- df1.dtypes gives us the datatypes of the columns. We have 9 int64 and 3 object datatypes.
- df1.shape shows us that 4269 rows and 12 columns.

In [6]:
```
df1.isnull().sum()
```

Out[6]:
```
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
```

In [7]:
```
df1.dtypes
```

Out[7]:
```
no_of_dependents             int64
education                   object
self_employed               object
income_annum                 int64
loan_amount                  int64
loan_term                    int64
cibil_score                  int64
residential_assets_value     int64
commercial_assets_value      int64
luxury_assets_value          int64
bank_asset_value             int64
loan_status                 object
dtype: object
```

In [8]:
```
df1.shape
```

Out[8]:
```
(4269, 12)
```

Exploratory Data Analysis

Data set overview

Dataset comntains 12 columns and 4269 rows. Most of the columns are integers except education, self_employed and loan_status are object which can be considered as categorical variables.

Description of each column:

- loan_id : identifier for each loan application
- no_of_dependents : number of dependents on the applicants

- education : Whether the applicant is graduate or not
- self_employed : Whether the applicant is self-employed or not
- income_annum : Annual income of the applicant
- loan_amount : amount of loan applied
- loan_term : duration of the loan in months
- cibil_score : CIBIL score of the applicant
- residential_assets_value : Value of residential assets owned
- commercial_assets_value : Value of commercial assets owned
- luxury_assets_value : Value of luxury assets owned by the applicant
- bank_asset_value : Value of bank assets owned
- loan_status : whether the loan application is approved or not

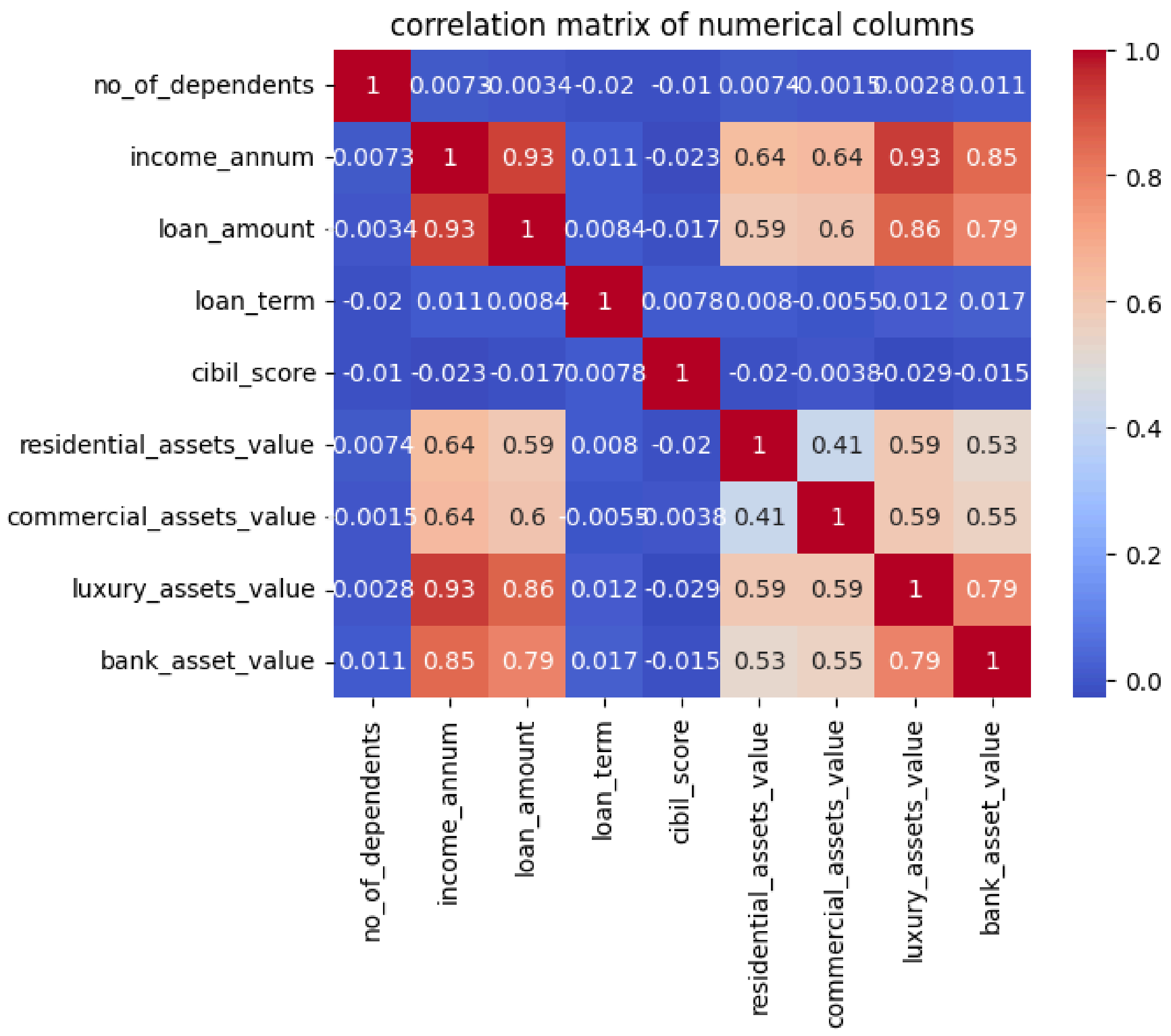
Analysis of Data

correlation matrix between numerical features

- Generates a heatmap to display the correlation matrix of numeric columns in the dataframe.
- df1.select_dtypes(include=['number'] is used to select only numerical columns in the dataframe.

In [9]:
```
plt.figure()
sns.heatmap(df1.select_dtypes(include=['number']).corr(),annot=True,cmap='coolwarm'
plt.title('correlation matrix of numerical columns')
```

Out[9]:
```
Text(0.5, 1.0, 'correlation matrix of numerical columns')
```



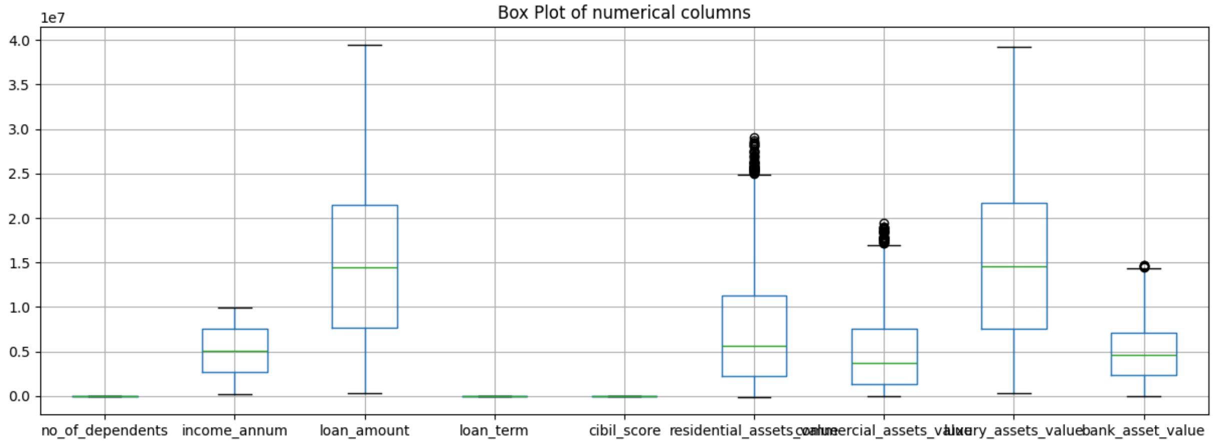
	no_of_dependents	income_annum	loan_amount	loan_term	cibil_score	residential_assets_value	commercial_assets_value	luxury_assets_value	bank_asset_value
no_of_dependents	1	0.0073	0.0034	-0.02	-0.01	0.0074	0.0015	0.0028	0.011
income_annum	0.0073	1	0.93	0.011	-0.023	0.64	0.64	0.93	0.85
loan_amount	0.0034	0.93	1	0.0084	-0.017	0.59	0.6	0.86	0.79
loan_term	-0.02	0.011	0.0084	1	0.0078	0.008	-0.0055	0.012	0.017
cibil_score	-0.01	-0.023	-0.017	0.0078	1	-0.02	-0.0038	-0.029	-0.015
residential_assets_value	0.0074	0.64	0.59	0.008	-0.02	1	0.41	0.59	0.53
commercial_assets_value	0.0015	0.64	0.6	-0.0055	0.0038	0.41	1	0.59	0.55
luxury_assets_value	0.0028	0.93	0.86	0.012	-0.029	0.59	0.59	1	0.79
bank_asset_value	0.011	0.85	0.79	0.017	-0.015	0.53	0.55	0.79	1

- `loan_amount` is positively correlated with `income_annum`, suggesting that higher income applicants tend to request larger loans.
- Asset values show moderate correlations with each other,indicating that applicants who own one type of asset often owns others as well.
- `cibil_score` does not show a strong correlation with most variables, which might suggest that other factors are more significant in determining the loan approval decision.
- Asset values have moderate to strong relationship with annual income. People with high income may be able to afford luxury assets.

Boxplots to find the outliers in the Data

- Generates a boxplot to find the outliers of numeric columns in the dataframe.
- df1.select_dtypes(include=['number'] is used to select only numerical columns in the dataframe.

In [10]:
```python
plt.figure(figsize=(15,5))
df1.select_dtypes(include=['number']).boxplot()
plt.title('Box Plot of numerical columns')
plt.show()
```

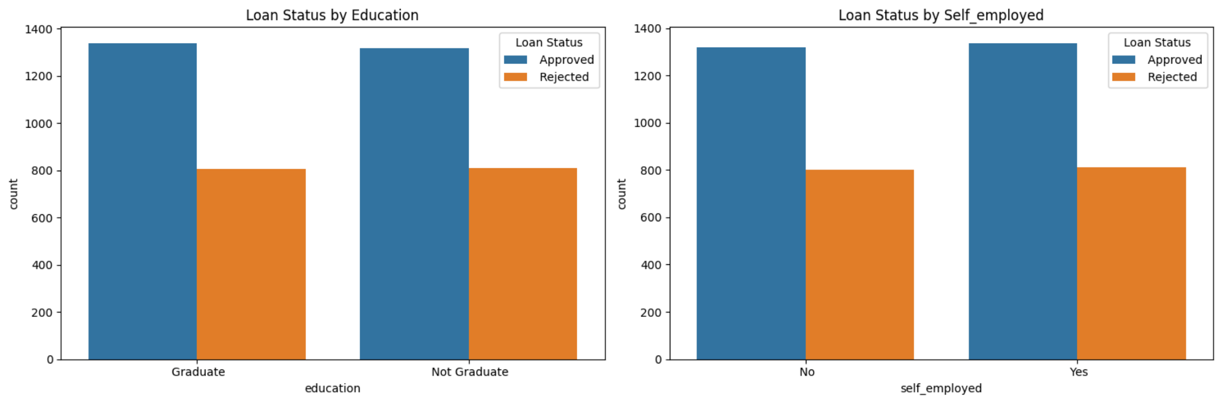


From the box plots residential assets, commercial assets and bank asset values has outlier. But they don't have significant impact on loan status which can be ignored.

Countplots to find relation between categorical features

- Creating a countplot to visualize how the categorial columns are linked to the loan status.
- Subplots are created for the categorial columns education and self-employed.

In [11]:
```python
categorical_columns = ['education', 'self_employed', 'loan_status']

plt.figure(figsize=(15, 5))

for i, column in enumerate(categorical_columns[:-1], 1):
    plt.subplot(1, 2, i)
    sns.countplot(x=column, hue='loan_status', data=df1)
    plt.title(f'Loan Status by {column.capitalize()}')
    plt.legend(title='Loan Status', loc='upper right')

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



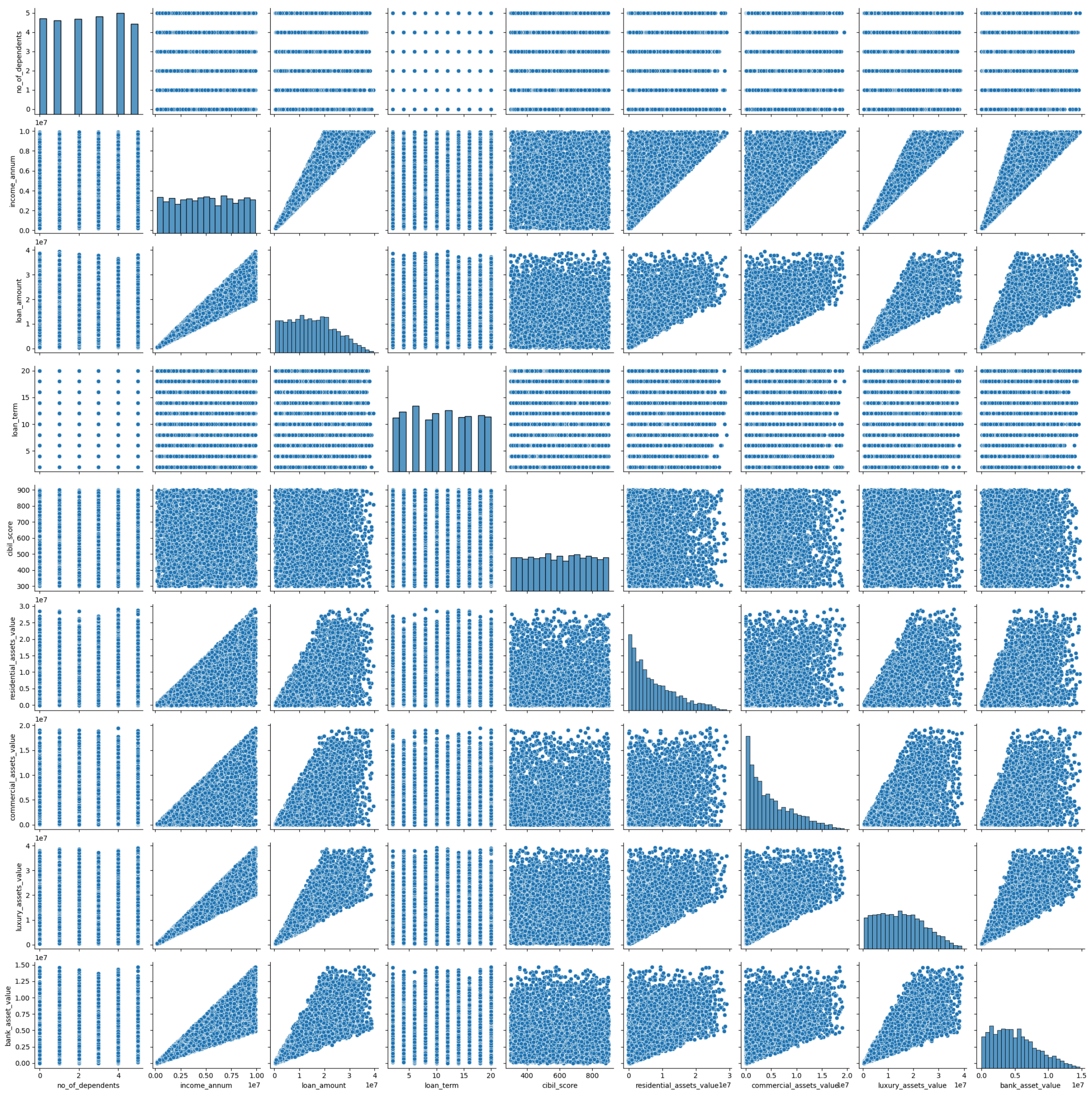
There is no strong relationship between the education and employement for loan approvals. There might be other factors such as assets, annual income and cibil score which may play important role in loan approvals

Pairplot to find relation between each numerical feature

- sns.pairplot(df1) is used to represent connections among numerous numerical variables in df1.

In [12]:
```
sns.pairplot(df1)
```

Out[12]:
```
<seaborn.axisgrid.PairGrid at 0x2609acee180>
```

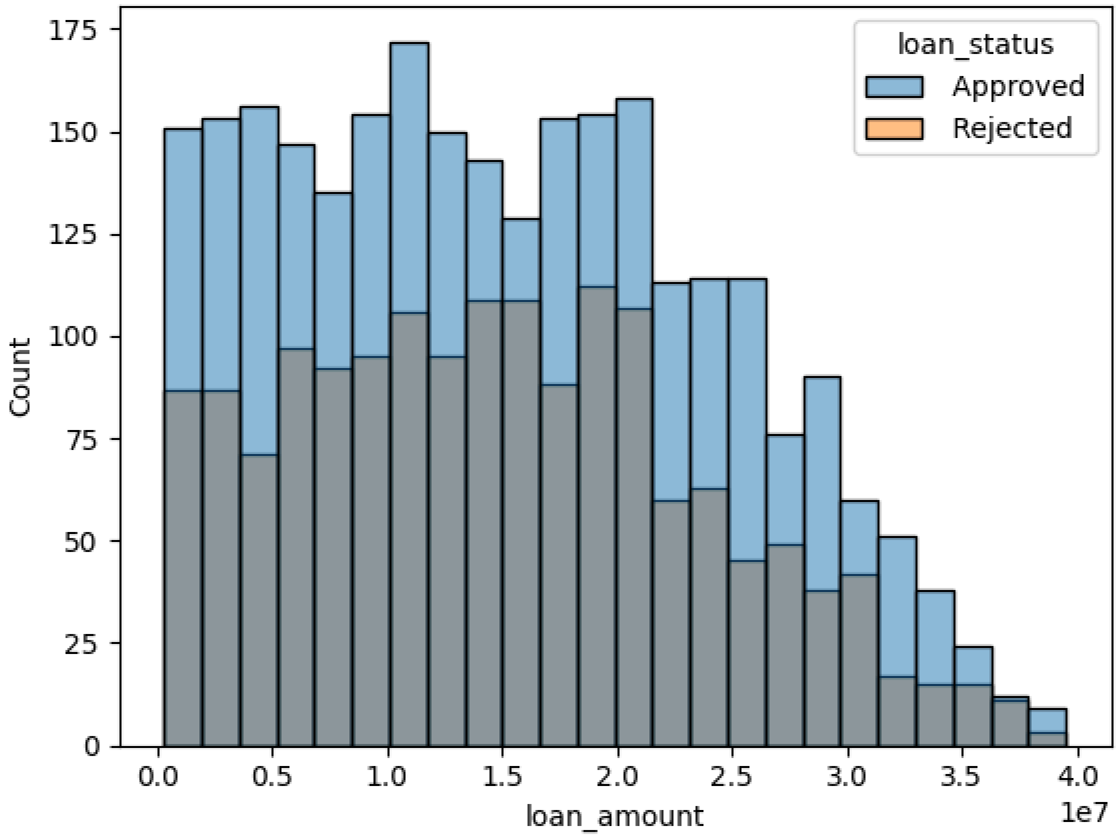


`loan_amount`, `income_annum`, `luxury_asset_value`, `bank_asset_value`, `income_annum` has positive correlation with other variables.

- Generating a histogram to display how loan amounts are spread out in df1, separating the information based on loan status.
- This shows how the loan amounts and status of the loan are related.

In [13]:
```
sns.histplot(df1,x='loan_amount',hue='loan_status')
```

Out[13]:
```
<Axes: xlabel='loan_amount', ylabel='Count'>
```

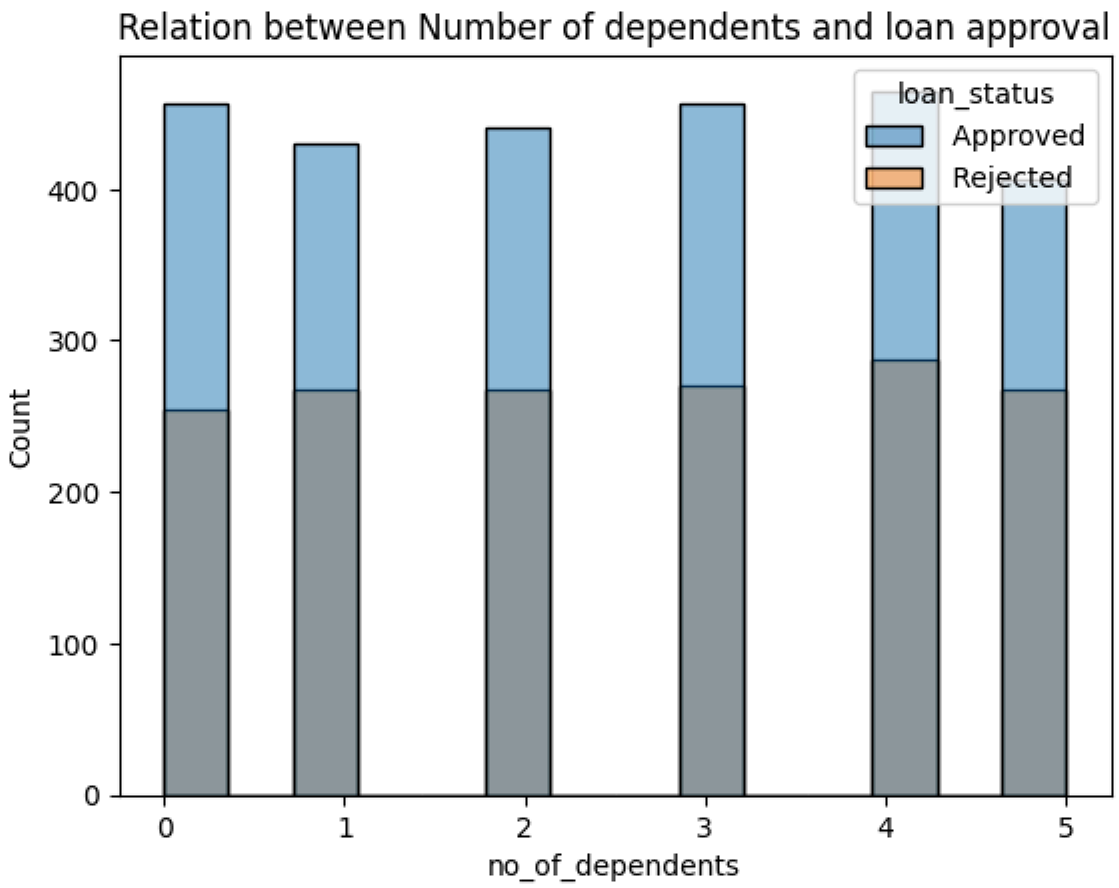


Same trend is going on between Approved loans and rejected loans. Seems like there is no strong relationship between the loan status and loan amount from the graph.

Relation between loan approval and number of dependents

- Generating a histogram to display how the number of dependents impacts the approval status of loans.
- This shows how the number of dependents and status of the loan are related.

In [14]:
```
sns.histplot(data=df1,x='no_of_dependents',hue='loan_status')
plt.title('Relation between Number of dependents and loan approval')
plt.show()
```

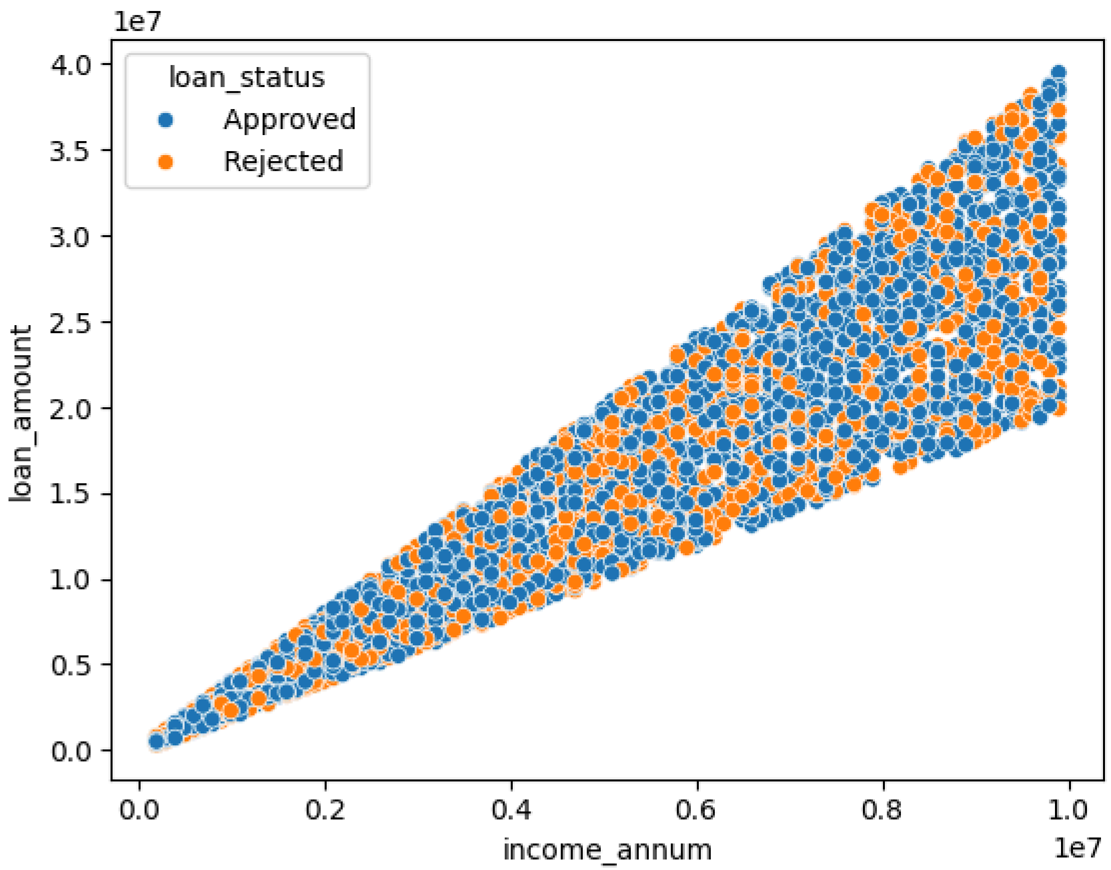


Number of dependents does not have any signficance on the loan approval rate.

- Generating a scatter plot to display the correlation between annual income and loan amount, categorizing by loan approval.

In [15]:
```
sns.scatterplot(data = df1, x='income_annum',y='loan_amount',hue='loan_status')
```

Out[15]:
```
<Axes: xlabel='income_annum', ylabel='loan_amount'>
```



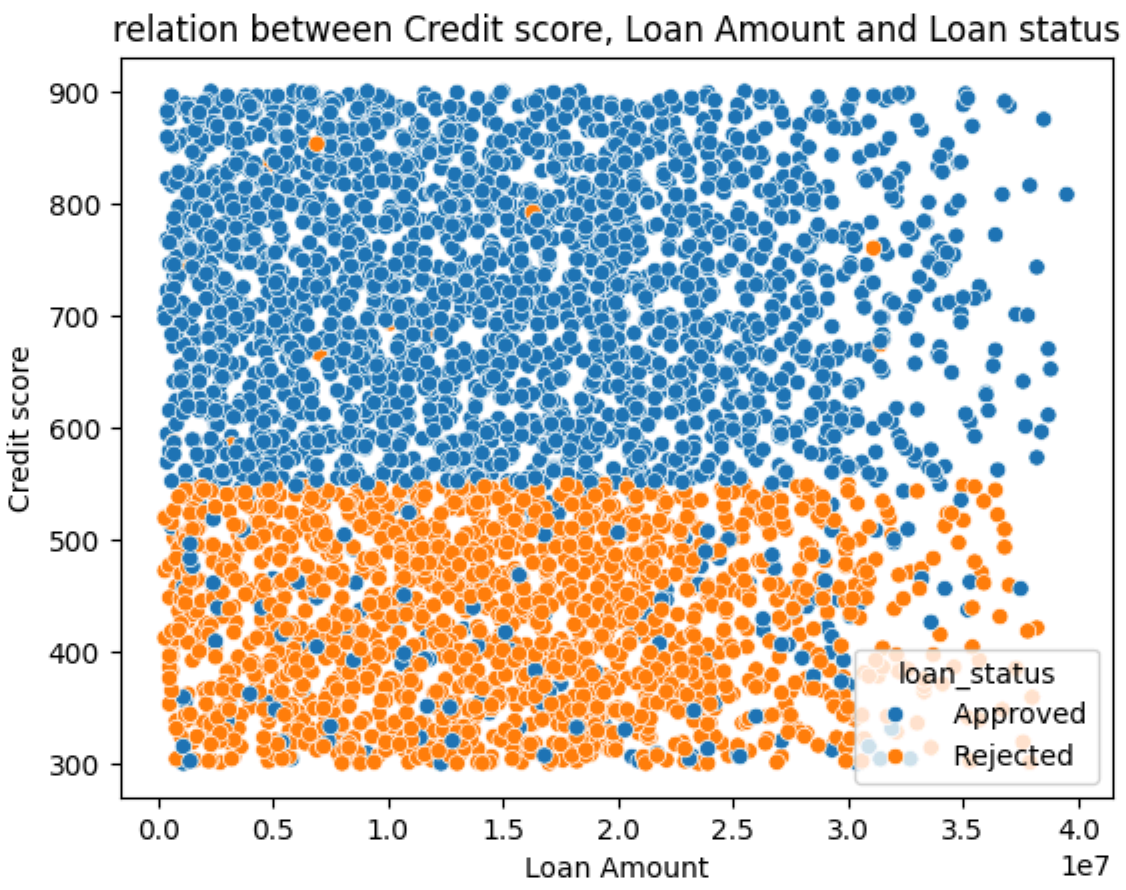
`loan_amount` increses with the `annual_income`, still loan was rejected for applicants with higher annual income. Other factors such as CIBIL score also plays a role in `loan_status`. Applicants with lower income will have a small range of loan amounts and applicants with higher income will have wider range of loan amount. This shows that ability to pay back loan which is dependent on annual income.

Relation between `credit_score`, `loan_amount` and `loan_status`

- Generating scatterplot to display the relationship between loan amount, credit score, and loan status.

In [16]:
```
sns.scatterplot(data=df1,x='loan_amount',y='cibil_score',hue='loan_status')
plt.title('relation between Credit score, Loan Amount and Loan status')
plt.ylabel('Credit score')
plt.xlabel('Loan Amount')
```

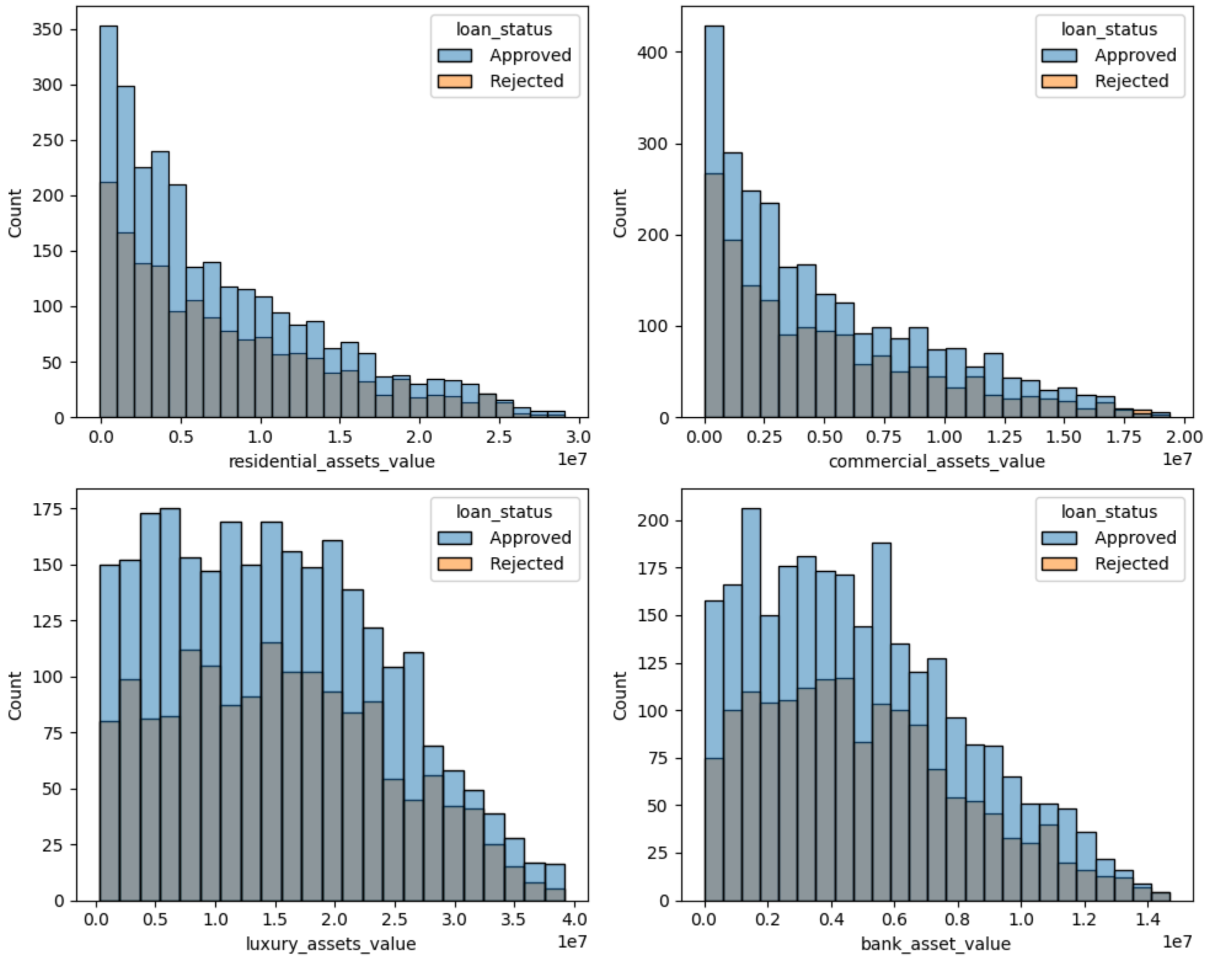
Out[16]:
```
Text(0.5, 0, 'Loan Amount')
```



credit score from 550 seperated loan status into two parts. Loan status is highly correlated with credit score. With credit scores above 550 has a good chance loan approval.

In some cases credit score below 550 got approved for loan and credit score greater than 550 also got rejected.

In [17]:
```
fig,ax=plt.subplots(2,2,figsize=(10,8))
sns.histplot(data=df1, x = 'residential_assets_value', hue = 'loan_status',ax=ax[0,
sns.histplot(data=df1, x = 'commercial_assets_value', hue = 'loan_status',ax=ax[0,1
sns.histplot(data=df1, x = 'luxury_assets_value', hue = 'loan_status',ax=ax[1,0])
sns.histplot(data=df1, x = 'bank_asset_value', hue = 'loan_status',ax=ax[1,1])

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

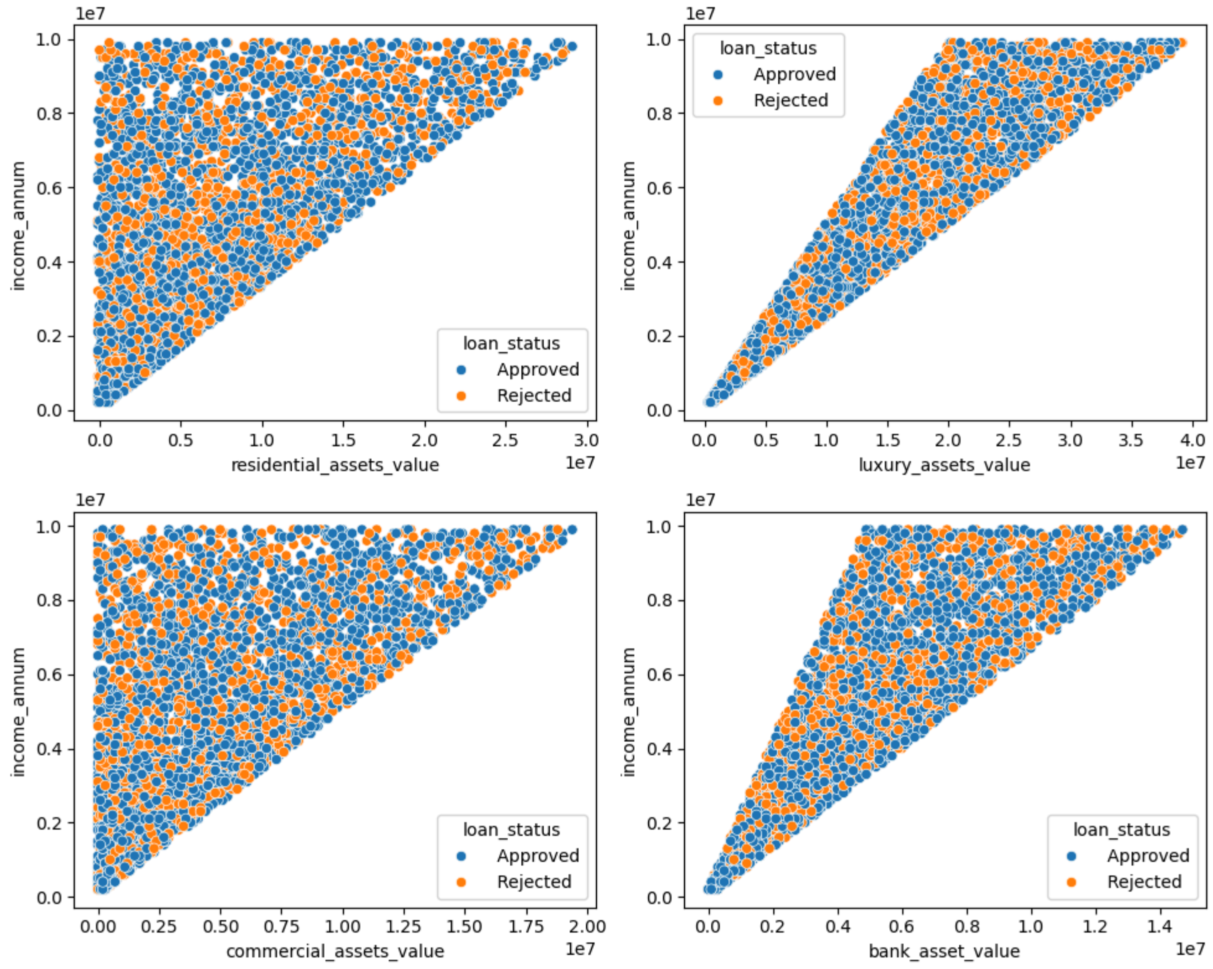


From the plots we can see there are no clear trends between the assets and loan approval.

Relationship between assets and income

- Generating a figure with multiple histograms to display how the different assets (residential, luxury, commercial, bank) are related to the loan approval status.

```
In [18]: fig,ax=plt.subplots(2,2,figsize=(10,8))
sns.scatterplot(data=df1, x = 'residential_assets_value', y= 'income_annum', hue = 
sns.scatterplot(data=df1, x = 'commercial_assets_value', y= 'income_annum', hue = '
sns.scatterplot(data=df1, x = 'luxury_assets_value', y= 'income_annum', hue = 'loan
sns.scatterplot(data=df1, x = 'bank_asset_value', y= 'income_annum', hue = 'loan_st

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



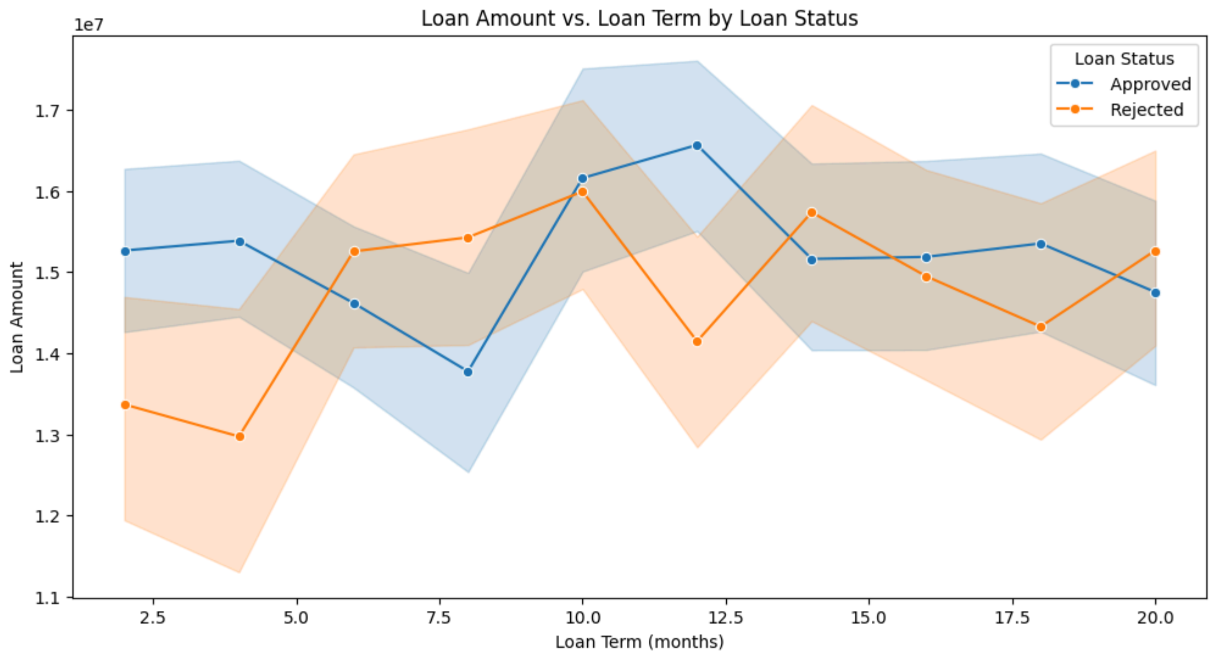
- commercial assets divided the plot into two sections which indicates relation between annual income. When asset reaches a certain values then annucal income increased significantly
- The shape of other assets show different kind of relation between the annual income. luxury assets shows a good indicator for higher income.

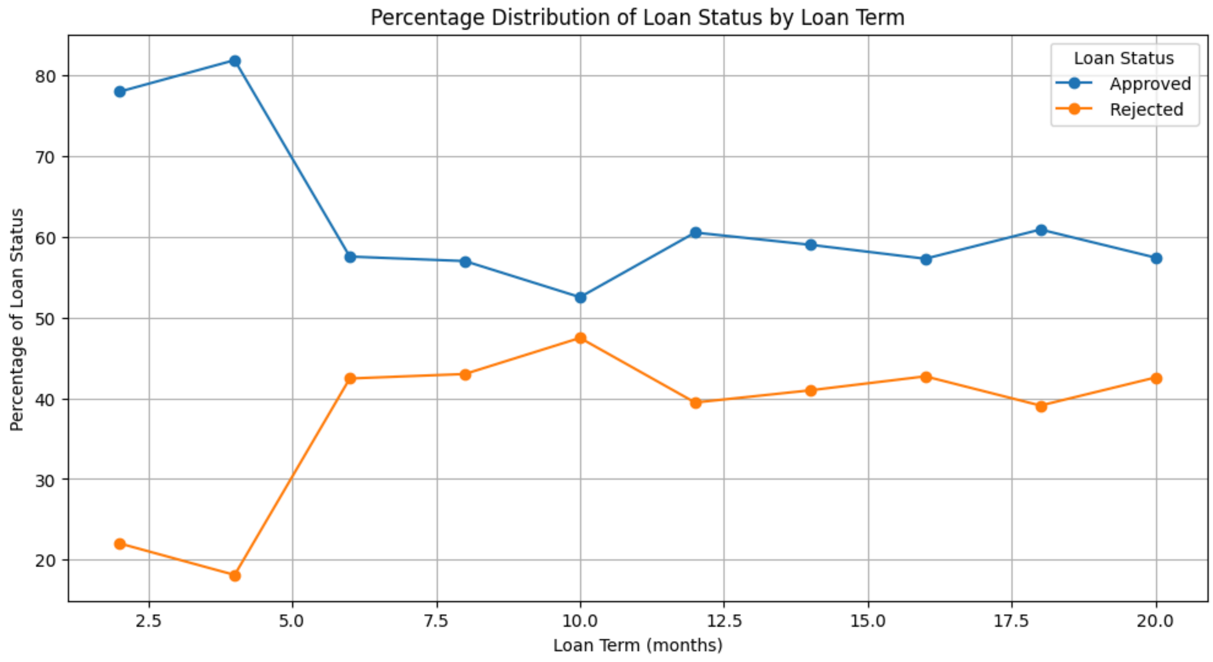
Loan Term and Loan status

- Generating a line plot to display the relationship between loan amount and loan term catogorized by the status of the loan.
- Also generating a percentage of loan statuses against loan term.

In [19]:
```
plt.figure(figsize=(12, 6))
sns.lineplot(data=df1, x='loan_term', y='loan_amount', hue='loan_status', marker='o

plt.title('Loan Amount vs. Loan Term by Loan Status')
plt.xlabel('Loan Term (months)')
plt.ylabel('Loan Amount')
plt.legend(title='Loan Status')

plt.show()
```

```
loan_term_status = df1.groupby(['loan_term', 'loan_status']).size().unstack().filln
loan_term_status_percentage = loan_term_status.div(loan_term_status.sum(axis=1), ax

# Plotting the percentage distribution of loan statuses against loan term
plt.figure(figsize=(12, 6))
loan_term_status_percentage.plot(kind='line', marker='o', ax=plt.gca())

# Adding labels and title to the plot
plt.title('Percentage Distribution of Loan Status by Loan Term')
plt.xlabel('Loan Term (months)')
plt.ylabel('Percentage of Loan Status')
plt.legend(title='Loan Status')
plt.grid(True)
```





Model Training

- LabelEncoder is imported to convert categorial values to numerical values.
- Columns like 'education', 'self_employed' and 'loan_status' is mapped to an unique integer.

In [20]:
```
from sklearn.preprocessing import LabelEncoder

df1['education'] = LabelEncoder().fit_transform(df1['education'])
df1['self_employed']= LabelEncoder().fit_transform(df1['self_employed'])
df1['loan_status']=LabelEncoder().fit_transform(df1['loan_status'])
```

In [21]:
```
print(df1[['education','self_employed','loan_status']])
```

```
      education  self_employed  loan_status
0             0              0            0
1             1              1            1
2             0              0            1
3             0              0            1
4             1              1            1
...         ...            ...          ...
4264          0              1            1
4265          1              1            0
4266          1              0            1
4267          1              0            0
4268          0              0            0

[4269 rows x 3 columns]
```

In [22]:
```
df1.dtypes
```

Out[22]:
```
no_of_dependents            int64
education                   int32
self_employed               int32
income_annum                int64
loan_amount                 int64
loan_term                   int64
cibil_score                 int64
residential_assets_value    int64
commercial_assets_value     int64
luxury_assets_value         int64
bank_asset_value            int64
loan_status                 int32
dtype: object
```

- Feature scaling is performed to standardize numerical columns which can be helpful in enhancing model performance and robustness to outliers.

In [63]:
```
from sklearn.preprocessing import StandardScaler

x = df1.drop(columns=['loan_status'])
y = df1['loan_status']
```

```
numerical_columns = ['no_of_dependents', 'income_annum', 'loan_amount', 'loan_term'
                     'residential_assets_value', 'commercial_assets_value', 'luxur
                     'bank_asset_value']

x[numerical_columns] = StandardScaler().fit_transform(x[numerical_columns])
print("Scaled Feature : \n",x.head())

print("Target Feature : \n",y.head())
```

```
Scaled Feature :
    no_of_dependents  education  self_employed  income_annum  loan_amount  \
0         -0.294102          0              0      1.617979     1.633052
1         -1.473548          1              1     -0.341750    -0.324414
2          0.295621          0              0      1.439822     1.610933
3          0.295621          0              0      1.119139     1.721525
4          1.475067          1              1      1.689242     1.002681

   loan_term  cibil_score  residential_assets_value  commercial_assets_value  \
0   0.192617     1.032792                 -0.780058                 2.877289
1  -0.508091    -1.061051                 -0.733924                -0.631921
2   1.594031    -0.544840                 -0.057300                -0.107818
3  -0.508091    -0.771045                  1.649637                -0.381263
4   1.594031    -1.264055                  0.757724                 0.735304

   luxury_assets_value  bank_asset_value
0             0.832028          0.930304
1            -0.694993         -0.515936
2             1.996520          2.407316
3             0.897943          0.899533
4             1.568075          0.007172
Target Feature :
 0    0
1    1
2    1
3    1
4    1
Name: loan_status, dtype: int32
```

- Dividing the dataset into training datase

In [65]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,classification_report
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size = 0.3 ,random_state
```

K-Nearest Neighbors (KNN)

A supervised learning classifier uses proximity to make classifications or predictions the grouping of a specific data point.

- KNeighborsClassifier is imported and the classifier is fitted on the training data.
- KNN classifier finds KNN for each test data and makes predictions.
- The accuracy of the model is calculated (0.8930523028883685).

In [48]:
```
from sklearn.neighbors import KNeighborsClassifier

knn_classifier = KNeighborsClassifier()
knn_classifier.fit(x_train,y_train)

y_pred = knn_classifier.predict(x_test)

accuracy = accuracy_score(y_test,y_pred)

print("Accuracy: ", accuracy)
```

```
Accuracy:  0.8930523028883685
```

- cross_val_score is imported.
- Performing 10-fold cross validation technique to improve model performance.
- Summmarizing the validation by calculating its mean.
- We expect the model to be around 91.10 % accurate on average.

In [49]:
```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(knn_classifier, x_train, y_train, cv = 10)

print('Cross-validation scores:{}'.format(scores))

print('Average cross-validation score: {:.4f}'.format(scores.mean()))
```

```
Cross-validation scores:[0.89632107 0.86956522 0.91638796 0.94314381 0.909699   0.88
628763
 0.909699   0.90635452 0.94295302 0.9295302 ]
Average cross-validation score: 0.9110
```

- KNeighborsRegressor is imported an fitted to training data.
- KNN Regressor predicts the mean square errors based on the k values, weights, and metrics.
- The used metrics are euclidean, manhattan, and minkowski.

In [69]:
```
from sklearn.preprocessing import StandardScaler

x = df1.drop(columns=['loan_status'])
y = df1['loan_status']

numerical_columns = ['no_of_dependents', 'income_annum', 'loan_amount', 'loan_term'
                     'residential_assets_value', 'commercial_assets_value', 'luxur
                     'bank_asset_value']

x[numerical_columns] = StandardScaler().fit_transform(x[numerical_columns])
print("Scaled Feature : \n",x.head())

print("Target Feature : \n",y.head())

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,classification_report
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size = 0.3 ,random_state
```

```
Scaled Feature :
    no_of_dependents  education  self_employed  income_annum  loan_amount  \
0          -0.294102          0              0      1.617979     1.633052
1          -1.473548          1              1     -0.341750    -0.324414
2           0.295621          0              0      1.439822     1.610933
3           0.295621          0              0      1.119139     1.721525
4           1.475067          1              1      1.689242     1.002681

   loan_term  cibil_score  residential_assets_value  commercial_assets_value  \
0   0.192617     1.032792                 -0.780058                 2.877289
1  -0.508091    -1.061051                 -0.733924                -0.631921
2   1.594031    -0.544840                 -0.057300                -0.107818
3  -0.508091    -0.771045                  1.649637                -0.381263
4   1.594031    -1.264055                  0.757724                 0.735304

   luxury_assets_value  bank_asset_value
0             0.832028          0.930304
1            -0.694993         -0.515936
2             1.996520          2.407316
3             0.897943          0.899533
4             1.568075          0.007172
Target Feature :
 0    0
1    1
2    1
3    1
4    1
Name: loan_status, dtype: int32
```

In [70]:
```
import sklearn.metrics
import sklearn.neighbors
from sklearn.neighbors import KNeighborsRegressor

k_values = [1, 3, 5]
weights = ['uniform', 'distance']
metrics = ['euclidean', 'manhattan', 'minkowski']
for k in k_values:
    for weight in weights:
        for metric in metrics:
            regressor = KNeighborsRegressor(n_neighbors=k, weights=weight, metric=m
            regressor.fit(x_train, y_train)
            yhats = regressor.predict(x_test)
            error = sklearn.metrics.mean_squared_error(yhats, y_test, squared=False
            print(f"k={k}, weight={weight}, metric={metric} MSE :  {error:.2f}")
```

```
k=1, weight=uniform, metric=euclidean MSE :  0.37
k=1, weight=uniform, metric=manhattan MSE :  0.39
k=1, weight=uniform, metric=minkowski MSE :  0.37
k=1, weight=distance, metric=euclidean MSE :  0.37
k=1, weight=distance, metric=manhattan MSE :  0.39
k=1, weight=distance, metric=minkowski MSE :  0.37
k=3, weight=uniform, metric=euclidean MSE :  0.30
k=3, weight=uniform, metric=manhattan MSE :  0.29
k=3, weight=uniform, metric=minkowski MSE :  0.30
k=3, weight=distance, metric=euclidean MSE :  0.30
k=3, weight=distance, metric=manhattan MSE :  0.29
k=3, weight=distance, metric=minkowski MSE :  0.30
k=5, weight=uniform, metric=euclidean MSE :  0.28
k=5, weight=uniform, metric=manhattan MSE :  0.28
k=5, weight=uniform, metric=minkowski MSE :  0.28
k=5, weight=distance, metric=euclidean MSE :  0.28
k=5, weight=distance, metric=manhattan MSE :  0.28
k=5, weight=distance, metric=minkowski MSE :  0.28
```

- Hyperparameter Tuning with different values of k with weights and metrics.
- The accuracy of model is 1 (which represents clean and accurate data).

some of the numerical columns such as no_of_dependents, commercial_assets_value and luxury_asset_values are removed and dataset is normalized using standardscaler() function

In [83]:
```python
from sklearn.preprocessing import StandardScaler

x = df1.drop(columns=['loan_status','no_of_dependents','commercial_assets_value','l
y = df1['loan_status']

numerical_columns = ['income_annum', 'loan_amount', 'loan_term', 'cibil_score','res

x[numerical_columns] = StandardScaler().fit_transform(x[numerical_columns])
print("Scaled Feature : \n",x.head())

print("Target Feature : \n",y.head())

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,classification_report
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size = 0.3 ,random_state
Scaled Feature :
    education  self_employed  income_annum  loan_amount  loan_term  \
0          0              0      1.617979     1.633052   0.192617
1          1              1     -0.341750    -0.324414  -0.508091
2          0              0      1.439822     1.610933   1.594031
3          0              0      1.119139     1.721525  -0.508091
4          1              1      1.689242     1.002681   1.594031

   cibil_score  residential_assets_value  bank_asset_value
0     1.032792                 -0.780058          0.930304
1    -1.061051                 -0.733924         -0.515936
2    -0.544840                 -0.057300          2.407316
3    -0.771045                  1.649637          0.899533
4    -1.264055                  0.757724          0.007172
Target Feature :
 0    0
1    1
2    1
3    1
4    1
Name: loan_status, dtype: int32
```

In [89]:

```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier

# Hyperparameter tuning for KNN
k_values = [1, 3, 5]
weights = ['uniform', 'distance']
metrics = ['euclidean', 'manhattan', 'minkowski']

for k in k_values:
    for weight in weights:
        for metric in metrics:
            classifier = KNeighborsClassifier(n_neighbors=k, weights=weight, metric
            classifier.fit(x_train, y_train)

            y_pred = classifier.predict(x_test)

            accuracy = accuracy_score(y_test, y_pred)
            print(f"k : {k}, weight : {weight}, metric : {metric}  Accuracy : {accu
k : 1, weight : uniform, metric : euclidean  Accuracy : 0.89
k : 1, weight : uniform, metric : manhattan  Accuracy : 0.89
k : 1, weight : uniform, metric : minkowski  Accuracy : 0.89
k : 1, weight : distance, metric : euclidean  Accuracy : 0.89
k : 1, weight : distance, metric : manhattan  Accuracy : 0.89
k : 1, weight : distance, metric : minkowski  Accuracy : 0.89
k : 3, weight : uniform, metric : euclidean  Accuracy : 0.90
k : 3, weight : uniform, metric : manhattan  Accuracy : 0.90
k : 3, weight : uniform, metric : minkowski  Accuracy : 0.90
k : 3, weight : distance, metric : euclidean  Accuracy : 0.90
k : 3, weight : distance, metric : manhattan  Accuracy : 0.90
k : 3, weight : distance, metric : minkowski  Accuracy : 0.90
k : 5, weight : uniform, metric : euclidean  Accuracy : 0.91
k : 5, weight : uniform, metric : manhattan  Accuracy : 0.92
k : 5, weight : uniform, metric : minkowski  Accuracy : 0.91
k : 5, weight : distance, metric : euclidean  Accuracy : 0.91
k : 5, weight : distance, metric : manhattan  Accuracy : 0.92
k : 5, weight : distance, metric : minkowski  Accuracy : 0.91
```

Logistic Regression

Logistic regression is a type of supervised machine learning method utilized for classification purposes in which the objective is to forecast the likelihood of whether an observation pertains to a specific category or not.

- LogisticRegression is imported and fitted on the training data.
- It learns coefficients to make predictions on the test data.
- The accuracy is calculated(0.9032006245120999).
- The classification report(precision, recall, and F1-score) is obtained.

In [86]:
```python
from sklearn.linear_model import LogisticRegression

reg = LogisticRegression(random_state=42)
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size = 0.3 ,random_state
reg.fit(x_train,y_train)

y_pred = reg.predict(x_test)
accuracy = accuracy_score(y_test,y_pred)

report = classification_report(y_test, y_pred)

print(report)
print("Accuracy : ", accuracy)
```

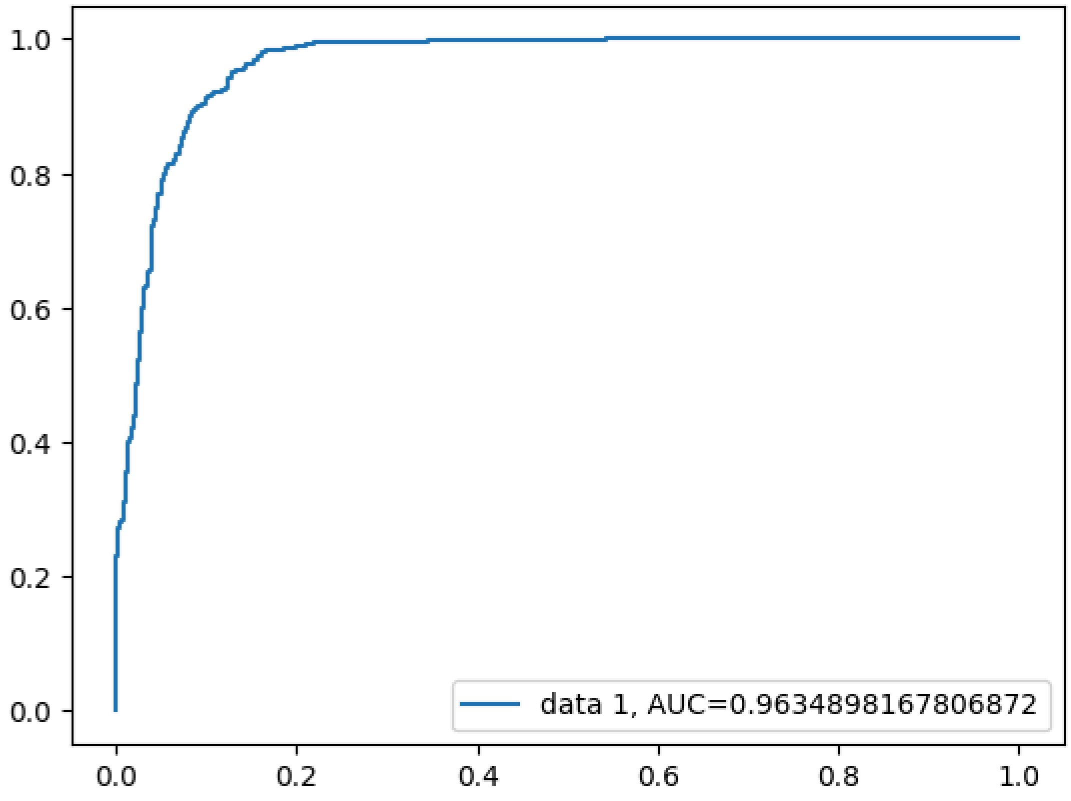
```
              precision    recall  f1-score   support

           0       0.92      0.93      0.92       810
           1       0.87      0.86      0.87       471

    accuracy                           0.90      1281
   macro avg       0.90      0.89      0.89      1281
weighted avg       0.90      0.90      0.90      1281

Accuracy :  0.9024199843871975
```

- imported classification_report, roc_auc_score, roc_curve
- Maximum iterations are set to perform cross validation scores.
- The mean accurate for 5 fold logistic regression cross validation is about 91.73%

In [79]:
```
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report, roc_auc_score, roc_curve

# Logistic Regression model
logreg = LogisticRegression(max_iter=10000)
# Logistic Regression Cross-validation
logreg_cv = cross_val_score(logreg, x, y, cv=5)
print("Logistic Regression Cross-Validation Scores:", logreg_cv)

print('Average cross-validation score: {:.4f}'.format(logreg_cv.mean()))
```

```
Logistic Regression Cross-Validation Scores: [0.92388759 0.91920375 0.91451991 0.923
88759 0.90504103]
Average cross-validation score: 0.9173
```

In [80]:
```
from sklearn import metrics
import matplotlib.pyplot as plt

y_pred_proba = reg.predict_proba(x_test)[:, 1]
fpr, tpr, _ = metrics.roc_curve(y_test, y_pred_proba)
auc = metrics.roc_auc_score(y_test, y_pred_proba)

plt.plot(fpr, tpr, label="data 1, AUC="+str(auc))
plt.legend(loc=4)
plt.show()
```



Support Vector Machine (SVM)

It is a supervised machine learning technique used to discover the best hyperplane in an N-dimensional space that can effectively divide the data points within various classes within the feature space.

- SVC is imported and trained on the training data.
- It makes predictions on the testing data by determining on which side of the hyperplane the test data point lies.
- The accuracy of the model is calculated(0.94).

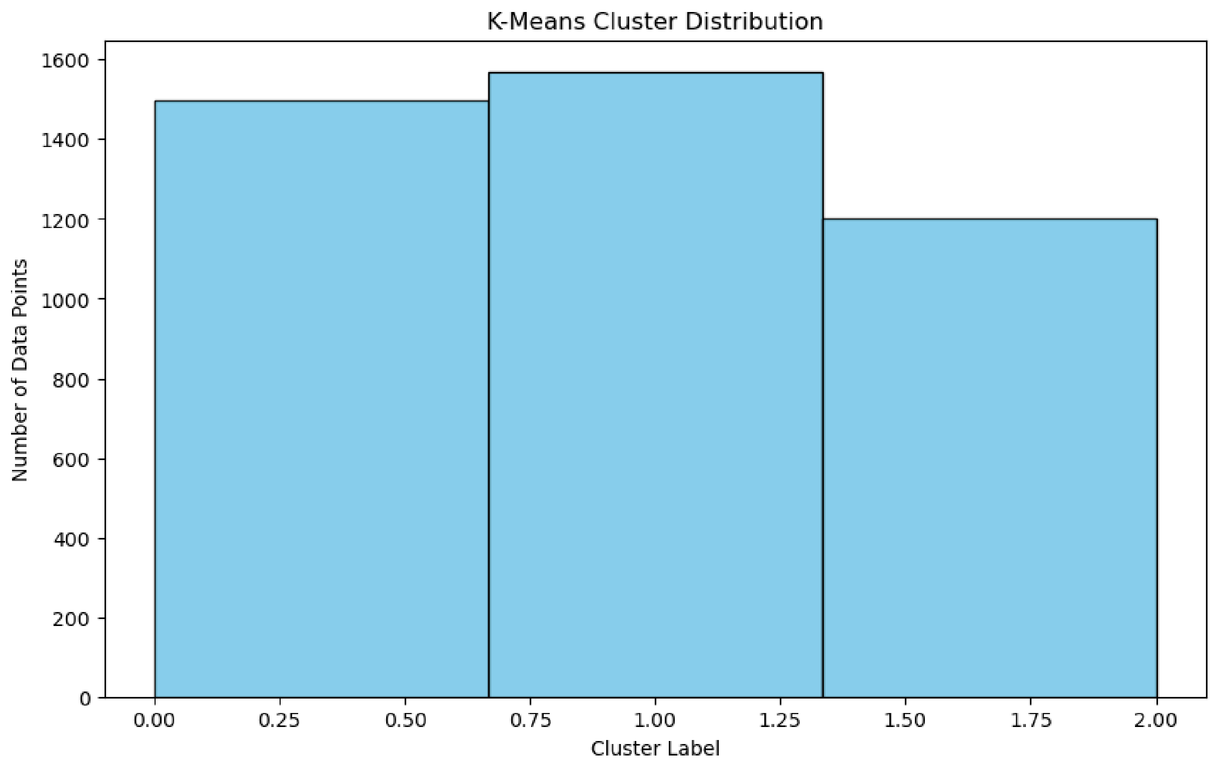
In [87]:
```
from sklearn.svm import SVC

svm_classifier = SVC(random_state = 42)
svm_classifier.fit(x_train,y_train)

y_pred = svm_classifier.predict(x_test)

accuracy = accuracy_score(y_test,y_pred)

print(accuracy)
```

```
0.942232630757221
```

Unsupervised Learning

K-means Clustering

In [11]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

# Clean the column names
df1.columns = df1.columns.str.strip()

# Preprocess the data
numeric_data = df1.drop(columns=["education", "self_employed", "loan_status"])
numeric_data = numeric_data.fillna(numeric_data.median())

# Standardize the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(numeric_data)

# Apply K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
df1["KMeans_Cluster"] = kmeans.fit_predict(scaled_data)

# Visualize the cluster distribution
plt.figure(figsize=(10, 6))
plt.hist(df1["KMeans_Cluster"], bins=3, color='skyblue', edgecolor='black')
plt.title("K-Means Cluster Distribution")
plt.xlabel("Cluster Label")
plt.ylabel("Number of Data Points")
plt.show()

# Display the resulting dataset with K-Means clusters
print(df1.head())
```



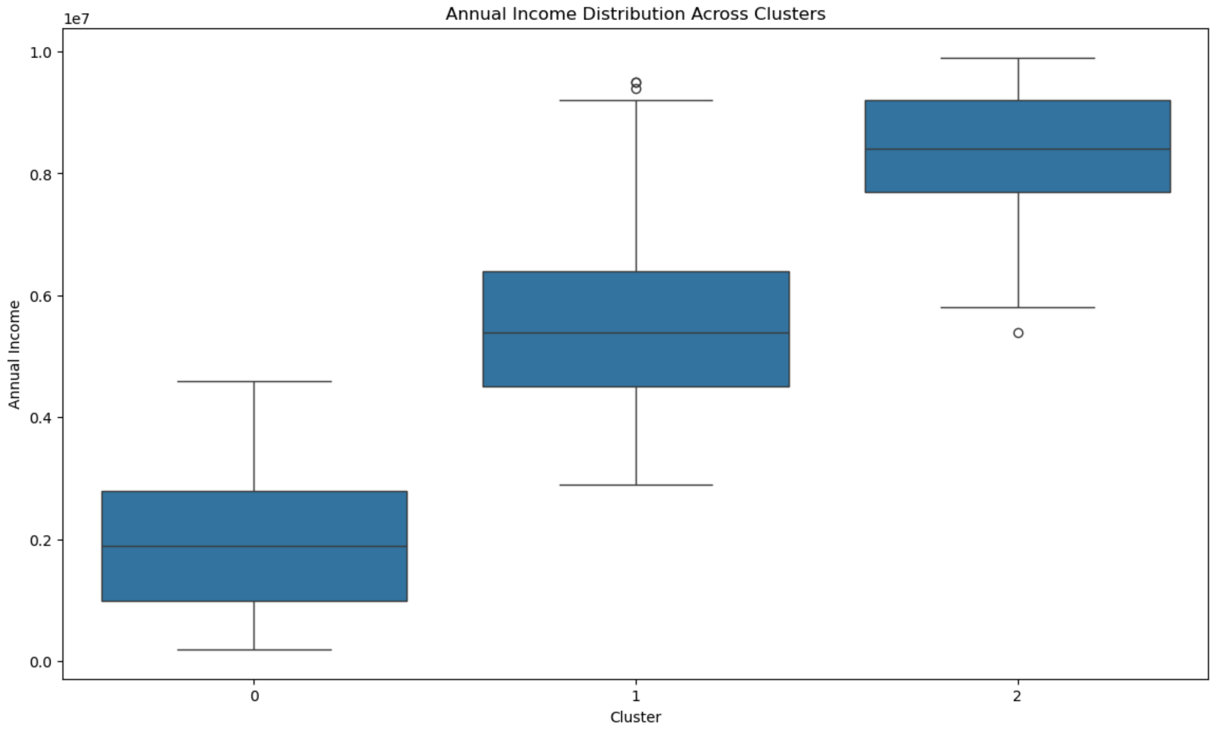
```
   no_of_dependents      education self_employed  income_annum  loan_amount  \
0                 2       Graduate            No       9600000     29900000   
1                 0   Not Graduate           Yes       4100000     12200000   
2                 3       Graduate            No       9100000     29700000   
3                 3       Graduate            No       8200000     30700000   
4                 5   Not Graduate           Yes       9800000     24200000   

   loan_term  cibil_score  residential_assets_value  commercial_assets_value  \
0         12          778                   2400000                 17600000   
1          8          417                   2700000                  2200000   
2         20          506                   7100000                  4500000   
3          8          467                  18200000                  3300000   
4         20          382                  12400000                  8200000   

   luxury_assets_value  bank_asset_value loan_status  KMeans_Cluster  
0             22700000           8000000    Approved               2  
1              8800000           3300000    Rejected               0  
2             33300000          12800000    Rejected               2  
3             23300000           7900000    Rejected               2  
4             29400000           5000000    Rejected               2  
```

Summary of Insights:

- Cluster 0 represents a lower-income group that tends to request smaller loans but enjoys higher approval rates, possibly due to more conservative loan requests.
- Cluster 1 includes a mix of lower-income applicants who may be over-leveraged (as indicated by the higher loan requests), leading to more frequent rejections.
- Cluster 2 consists of higher-income applicants who request larger loans, with a relatively balanced approval rate but also higher risk associated with the larger loan amounts

In [23]:
```python
numeric_columns = numeric_data.columns
cluster_summary = df1.groupby("KMeans_Cluster")[numeric_columns].mean()
print("Cluster Summary Statistics:\n", cluster_summary)

# Visualization: Boxplot for Annual Income Distribution
plt.figure(figsize=(14, 8))
sns.boxplot(x="KMeans_Cluster", y="income_annum", data=df1)
plt.title("Annual Income Distribution Across Clusters")
plt.xlabel("Cluster")
plt.ylabel("Annual Income")
plt.show()
```

```
Cluster Summary Statistics:
                 no_of_dependents  income_annum   loan_amount  loan_term  \
KMeans_Cluster
0                        2.400534  1.927637e+06  5.674299e+06  10.962617
1                        2.645860  5.498471e+06  1.617662e+07  10.701911
2                        2.428809  8.390674e+06  2.556811e+07  11.082431

                cibil_score  residential_assets_value  \
KMeans_Cluster
0                610.461282              2.708211e+06
1                591.408280              7.217962e+06
2                597.955870              1.374813e+07

                commercial_assets_value  luxury_assets_value  bank_asset_value
KMeans_Cluster
0                          1.708211e+06         5.588518e+06      1.796328e+06
1                          4.926943e+06         1.622127e+07      5.307452e+06
2                          9.105912e+06         2.559134e+07      8.511157e+06
```



Annual Income Distribution Across Clusters :

The boxplot reveals distinct differences in income levels among the clusters:

Cluster 0:

- Represents the group with the lowest annual income, with a median income significantly below 4 million.
- The income range is narrower, indicating less variability within this group.

Cluster 1:

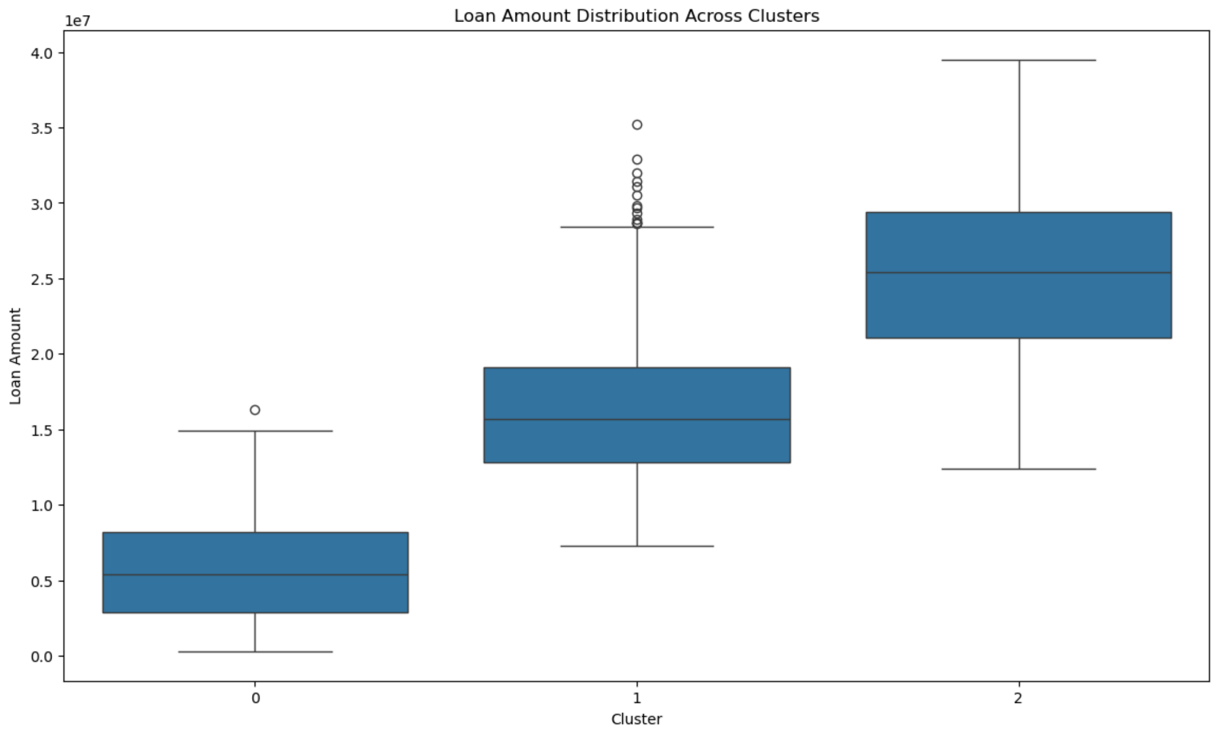
- Shows a higher median income than Cluster 0, but still lower than Cluster 2.

- The income distribution is more variable, with some outliers indicating higher income applicants.

Cluster 2:

- Has the highest median income among all clusters, with values reaching close to 10 million.
- This cluster has the widest income range, suggesting a mix of both high and very high-income applicants.

In [25]:
```
# Visualization: Boxplot for Loan Amount Distribution
plt.figure(figsize=(14, 8))
sns.boxplot(x="KMeans_Cluster", y="loan_amount", data=df1)
plt.title("Loan Amount Distribution Across Clusters")
plt.xlabel("Cluster")
plt.ylabel("Loan Amount")
plt.show()
```



Loan Amount Distribution Across Clusters:

- The boxplot for loan amounts aligns well with the income levels observed:

Cluster 0:

- This group generally applies for smaller loan amounts, with the median below 1.5 million.
- The lower loan amounts are consistent with the lower income levels seen in this cluster.

Cluster 1:

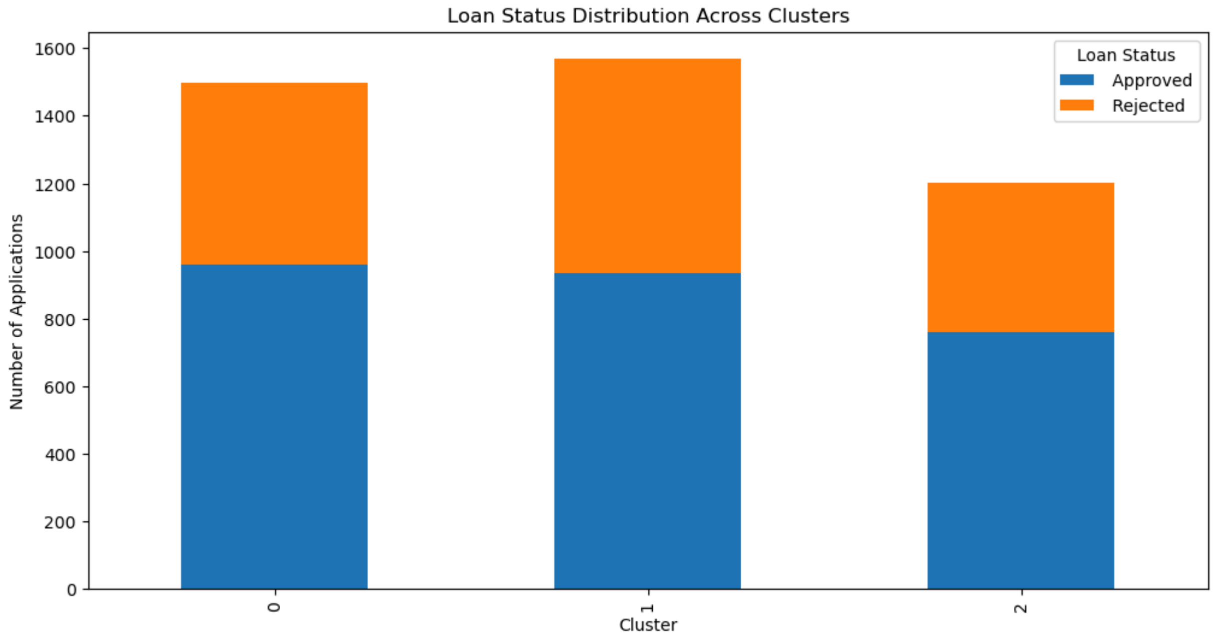
- Applicants in this cluster tend to apply for slightly higher loan amounts compared to Cluster 0.
- The presence of several outliers suggests that some applicants request larger loans despite lower income, which might impact approval rates.

Cluster 2:

- Has the highest loan amounts, with a median above 2 million and values extending up to 4 million.
- The higher loan requests correlate with the higher income levels, indicating that this cluster consists of wealthier applicants seeking larger loans.

In [27]:
```
# Analyze the relationship between clusters and loan status
cluster_loan_status = df1.groupby(["KMeans_Cluster", "loan_status"]).size().unstack
print("Cluster Loan Status Relationship:\n", cluster_loan_status)

# Visualization: Stacked Bar Plot for Loan Status Distribution
cluster_loan_status.plot(kind="bar", stacked=True, figsize=(12, 6))
plt.title("Loan Status Distribution Across Clusters")
plt.xlabel("Cluster")
plt.ylabel("Number of Applications")
plt.legend(title="Loan Status")
plt.show()
```

```
Cluster Loan Status Relationship:
 loan_status     Approved  Rejected
KMeans_Cluster                     
0                    961       537
1                    936       634
2                    759       442
```



Loan Status Distribution Across Clusters:

- The stacked bar plot shows the distribution of loan approvals and rejections across the clusters:

Cluster 0:

- A higher number of approved loans compared to rejected ones.
- This cluster, despite having lower income, might consist of applicants with stable financial profiles leading to higher approval rates.

Cluster 1:

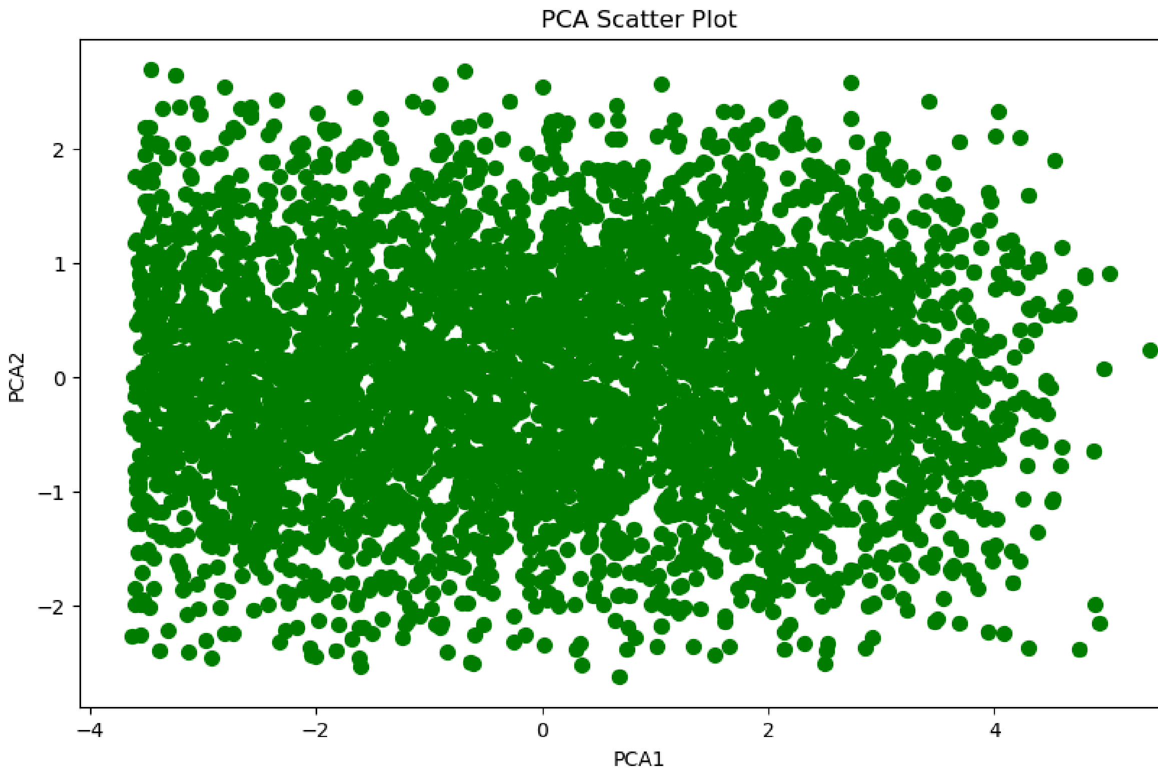
- Has a relatively balanced distribution of approvals and rejections, but slightly more rejections.
- The presence of lower-income applicants requesting higher loans (as seen from the outliers) may contribute to the higher rejection rate.

Cluster 2:

- Shows a better approval rate compared to Cluster 1, but still has a significant number of rejections.
- This is likely due to the high loan amounts requested, which might be riskier for approval despite higher income levels.

Principal Component Analysis (PCA)

In [13]:
```python
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Apply PCA
pca = PCA(n_components=2)
pca_components = pca.fit_transform(scaled_data)

# Add PCA components to the dataset
df1["PCA1"] = pca_components[:, 0]
df1["PCA2"] = pca_components[:, 1]

# Visualize the PCA components
plt.figure(figsize=(10, 6))
plt.scatter(df1["PCA1"], df1["PCA2"], color='green', s=50)
plt.title("PCA Scatter Plot")
plt.xlabel("PCA1")
plt.ylabel("PCA2")
plt.show()

# Display the resulting dataset with PCA components
print(df1.head())
```



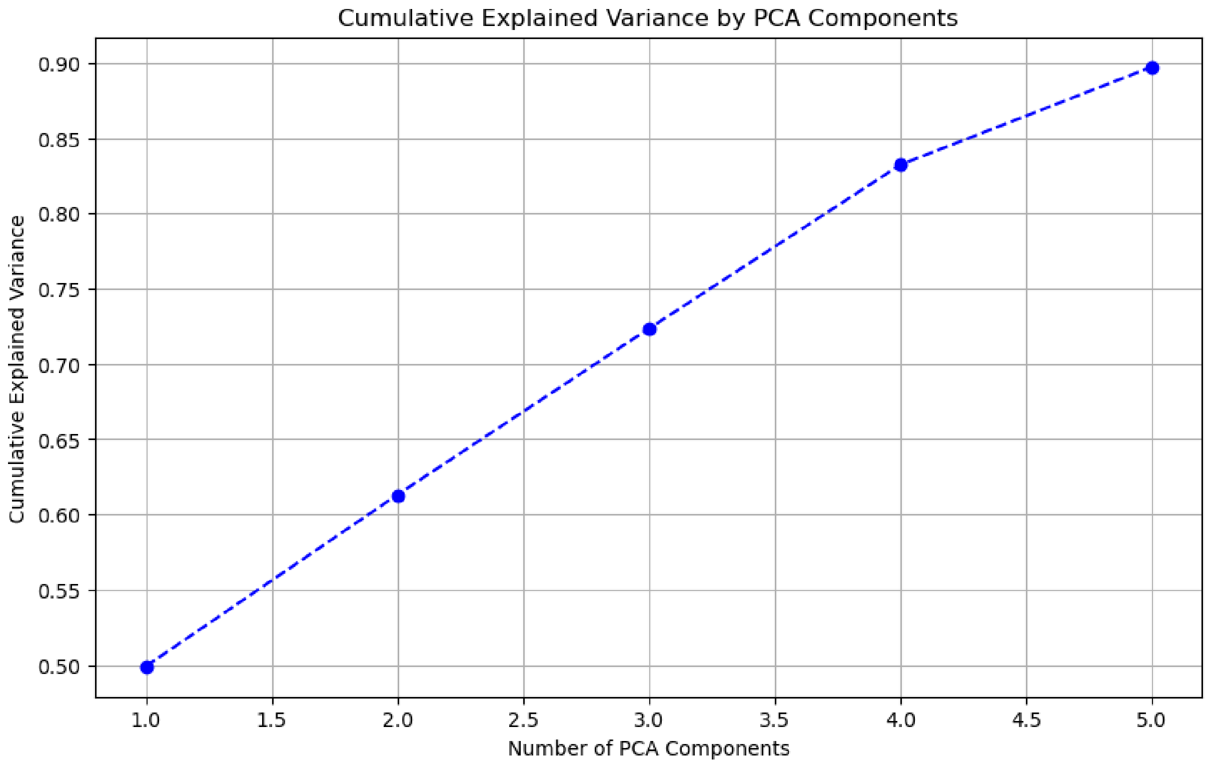
```
   no_of_dependents      education self_employed  income_annum  loan_amount  \
0                 2       Graduate            No       9600000     29900000   
1                 0   Not Graduate           Yes       4100000     12200000   
2                 3       Graduate            No       9100000     29700000   
3                 3       Graduate            No       8200000     30700000   
4                 5   Not Graduate           Yes       9800000     24200000   

   loan_term  cibil_score  residential_assets_value  commercial_assets_value  \
0         12          778                   2400000                 17600000   
1          8          417                   2700000                  2200000   
2         20          506                   7100000                  4500000   
3          8          467                  18200000                  3300000   
4         20          382                  12400000                  8200000   

   luxury_assets_value  bank_asset_value loan_status  KMeans_Cluster  \
0             22700000           8000000    Approved               2   
1              8800000           3300000    Rejected               0   
2             33300000          12800000    Rejected               2   
3             23300000           7900000    Rejected               2   
4             29400000           5000000    Rejected               2   

       PCA1      PCA2  
0  2.918010 -0.786042  
1 -1.275304 -0.177065  
2  3.210459 -0.599451  
3  2.473858  0.839491  
4  2.445725  0.495872  
```

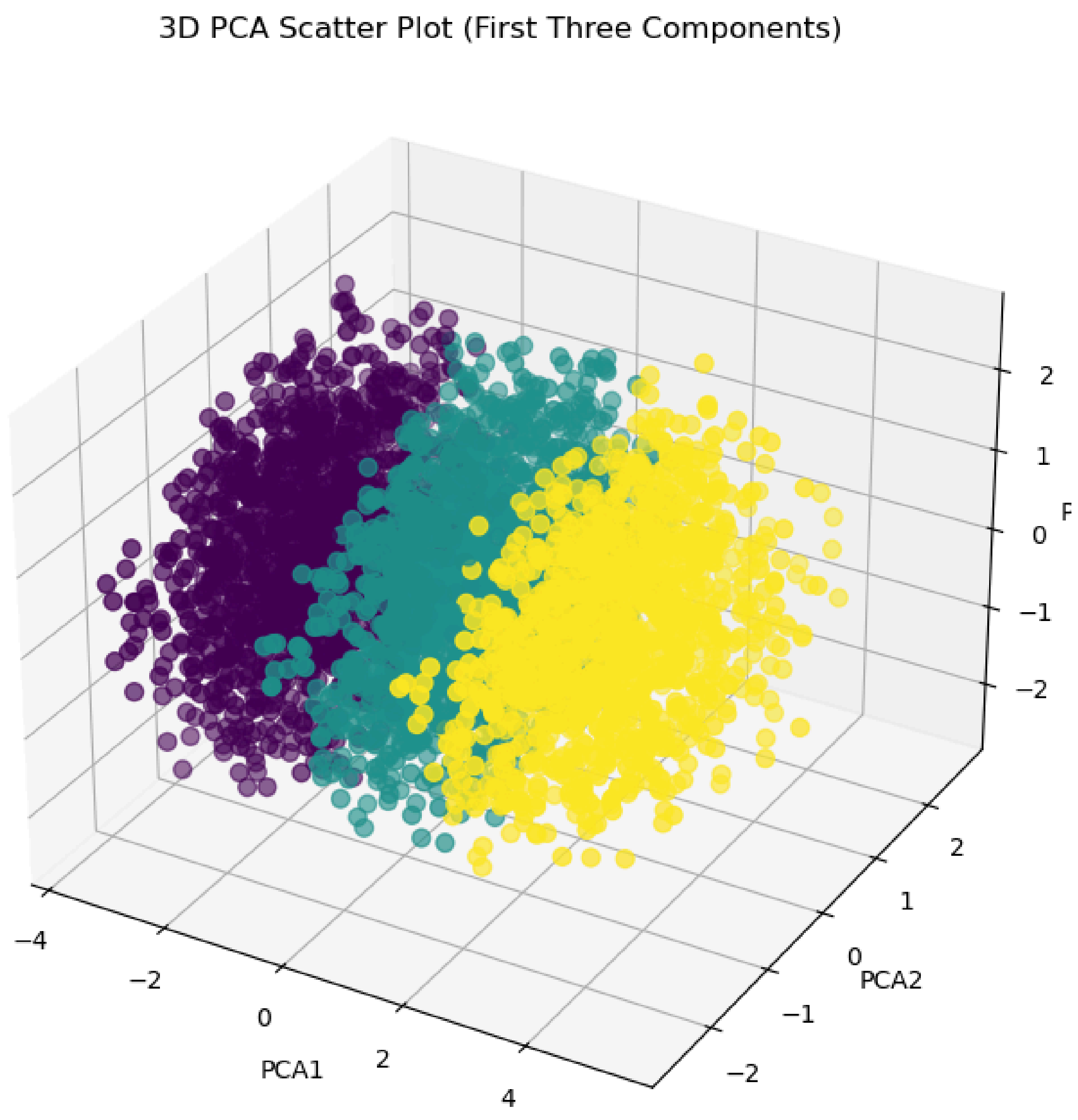
- The provided table includes the transformed values for PCA1 and PCA2 along with other features.

- Each row shows the new coordinates of the data points after being projected onto the PCA components.
- Example:
  - For the first row (Applicant 0):
    - The values for PCA1 and PCA2 are 2.918010 and -0.786042, respectively.
    - This point lies towards the positive side of PCA1, indicating a higher score along the direction that captures the most variance in the dataset.
  - For the second row (Applicant 1):
    - The values for PCA1 and PCA2 are -1.275304 and -0.177065, respectively.
    - This point lies towards the negative side of PCA1, suggesting it has characteristics that are less aligned with the dominant variance direction.

In [38]:
```
from mpl_toolkits.mplot3d import Axes3D

# Apply PCA with more components
pca_full = PCA(n_components=5)
pca_components = pca_full.fit_transform(scaled_data)

# Explained variance ratio
explained_variance = pca_full.explained_variance_ratio_
cumulative_variance = explained_variance.cumsum()

# Create a DataFrame for explained variance
variance_df = pd.DataFrame({
    "PCA Component": [f"PCA{i+1}" for i in range(len(explained_variance))],
    "Explained Variance": explained_variance,
    "Cumulative Variance": cumulative_variance
})

# Display the explained variance data
print("Explained Variance of PCA Components:\n", variance_df)

# Plot cumulative explained variance
plt.figure(figsize=(10, 6))
plt.plot(range(1, 6), cumulative_variance, marker='o', linestyle='--', color='b')
plt.title("Cumulative Explained Variance by PCA Components")
plt.xlabel("Number of PCA Components")
plt.ylabel("Cumulative Explained Variance")
plt.grid()
plt.show()

# 3D Scatter Plot using the first three PCA components
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(pca_components[:, 0], pca_components[:, 1], pca_components[:, 2],
           c=df1["KMeans_Cluster"], cmap="viridis", s=50)
ax.set_title("3D PCA Scatter Plot (First Three Components)")
ax.set_xlabel("PCA1")
ax.set_ylabel("PCA2")
ax.set_zlabel("PCA3")
plt.show()
```

```
Explained Variance of PCA Components:
   PCA Component  Explained Variance  Cumulative Variance
0           PCA1            0.498863             0.498863
1           PCA2            0.114025             0.612888
2           PCA3            0.110453             0.723341
3           PCA4            0.108912             0.832253
4           PCA5            0.065061             0.897314
```





- Explained Variance of PCA Components:
- The table and cumulative variance plot provide insights into how much of the dataset's variance is explained by each PCA component:
    - PCA1: Explains 49.9% of the variance. This indicates that the first principal component captures almost half of the information in the data.
    - PCA2: Adds another 11.4%, bringing the cumulative explained variance to 61.3%.
    - PCA3: Contributes an additional 11.0%, increasing the cumulative variance to 72.3%.
    - PCA4: Explains 10.9% more, raising the cumulative variance to 83.2%.
    - PCA5: Adds the final 6.5%, resulting in a total cumulative explained variance of 89.7%.
- The cumulative explained variance curve shows a diminishing return after the first three components.

- This suggests that the first three components are sufficient to capture the majority of the information in the dataset, while additional components contribute less.

Random Forest

- RandomForestClassifier and Label Encoder is imported to fit and transform the categorical columns.
- The Random Forest Classifier combines all decision trees to improve overall accuracy.
- The accuracy (98.36%) and classification report is calculated.

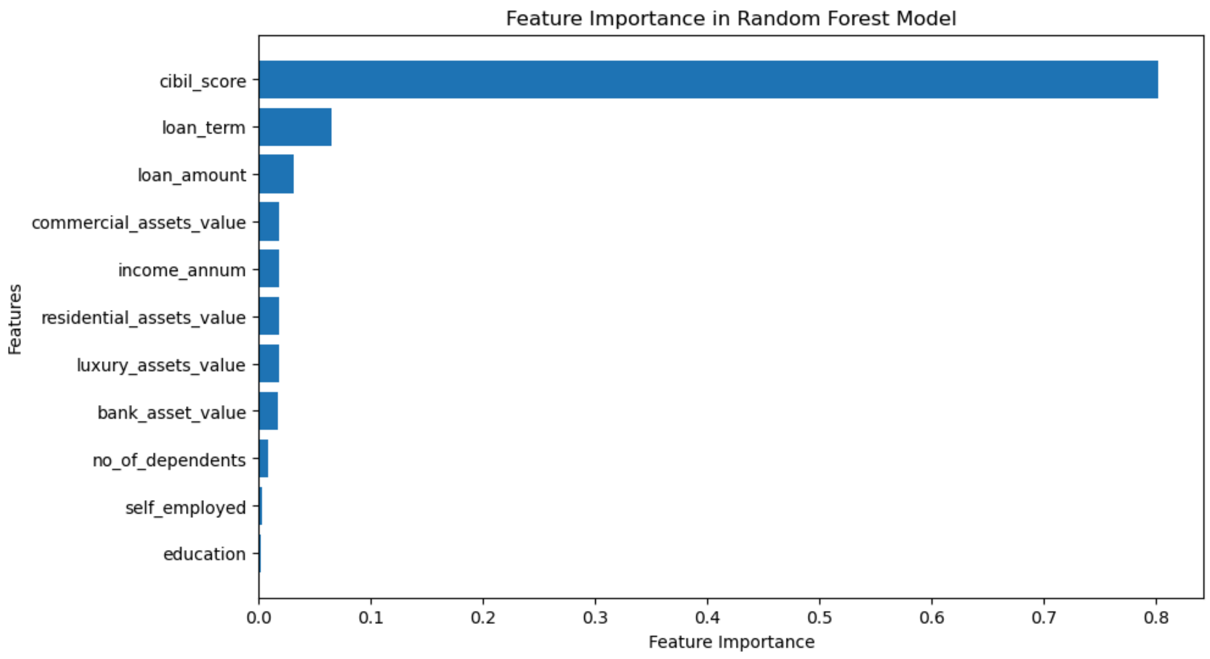
In [11]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score
from sklearn.preprocessing import LabelEncoder

# Clean column names
df.columns = df.columns.str.strip()

# Data Preprocessing
label_encoders = {}
categorical_columns = ['education', 'self_employed', 'loan_status']

# Encoding categorical variables
for col in categorical_columns:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])
    label_encoders[col] = le

# Splitting data into features and target
X = df.drop(columns=['loan_id', 'loan_status'])  # Exclude 'loan_id' and target var
y = df['loan_status']

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_sta

# Train the Random Forest model
rf_model = RandomForestClassifier(random_state=42, n_estimators=100)
rf_model.fit(X_train, y_train)

# Predictions
y_pred = rf_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Print results
print(f"Accuracy: {accuracy * 100:.2f}%")
print("\nClassification Report:")
print(classification_rep)
```

```
Accuracy: 98.36%

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       797
           1       0.99      0.97      0.98       484

    accuracy                           0.98      1281
   macro avg       0.98      0.98      0.98      1281
weighted avg       0.98      0.98      0.98      1281
```

Feature Importance

- The rf_model.feature_importances_ contains all the important scores for each and every feature.
- By sorting, re-ordering the features, plotting them using matplot.
- Here can be seen that, **cibil_score** plays the main role in loan approvals and rejections.

In [13]:
```python
# Calculate the feature importance
feature_importance = rf_model.feature_importances_
features = X.columns

# Sorting feature importance for better visualization
sorted_idx = feature_importance.argsort()
sorted_features = features[sorted_idx]
sorted_importance = feature_importance[sorted_idx]

# Plot the feature importance
plt.figure(figsize=(10, 6))
plt.barh(sorted_features, sorted_importance, align='center')
plt.xlabel('Feature Importance')
plt.ylabel('Features')
plt.title('Feature Importance in Random Forest Model')
plt.show()
```


Feature Importance in Random Forest Model
cibil_score
loan_term
loan_amount
commercial_assets_value
income_annum
residential_assets_value
luxury_assets_value
bank_asset_value
no_of_dependents
self_employed
education
Features
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Feature Importance