Open in Colab

Loading and Exploring the Data

This section explores the train and test datasets as shown below.

- **The train dataset** contains 614 observations and 13 features represents 12 independent variables and 1 target variable.
- **The test dataset** contains the same features except the target variable.
- The data type of each variable is also provided below whether its categorical or numerical.

In [1]:
```
# Importing the libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.model_selection import cross_val_score, train_test_split, GridSearchCV
from sklearn import metrics
from sklearn.discriminant_analysis import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

%matplotlib inline
```

In [2]:
```
sns.set_theme()
```

In [3]:
```
# Importing the datasets
train = pd.read_csv("train.csv")
test = pd.read_csv("test.csv")

train_original=train.copy()
test_original=test.copy()
```

In [4]:
```
t_copy = train.copy()
```

The dataset consists of the following columns:

- **Loan_ID :** Unique Loan ID
- **Gender :** Male/ Female
- **Married :** Applicant married (Y/N)
- **Dependents :** Number of dependents
- **Education :** Applicant Education (Graduate/ Under Graduate)
- **Self_Employed :** Self employed (Y/N)
- **ApplicantIncome :** Applicant income
- **CoapplicantIncome :** Coapplicant income
- **LoanAmount :** Loan amount in thousands of dollars
- **Loan_Amount_Term :** Term of loan in months
- **Credit_History :** credit history meets guidelines yes or no
- **Property_Area :** Urban/ Semi Urban/ Rural
- **Loan_Status :** Loan approved (Y/N) this is the target variable

In [5]:
```
# Understanding the dataset
train.head()
```

Out[5]:

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

In [6]:
```
# Checking the columns and the shape of the train dataset
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            614 non-null    object 
 1   Gender             601 non-null    object 
 2   Married            611 non-null    object 
 3   Dependents         599 non-null    object 
 4   Education          614 non-null    object 
 5   Self_Employed      582 non-null    object 
 6   ApplicantIncome    614 non-null    int64  
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object 
 12  Loan_Status        614 non-null    object 
dtypes: float64(4), int64(1), object(8)
memory usage: 62.5+ KB
```

In [7]:

```
# Checking the columns and the shape of the test dataset
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 367 entries, 0 to 366
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            367 non-null    object 
 1   Gender             356 non-null    object 
 2   Married            367 non-null    object 
 3   Dependents         357 non-null    object 
 4   Education          367 non-null    object 
 5   Self_Employed      344 non-null    object 
 6   ApplicantIncome    367 non-null    int64  
 7   CoapplicantIncome  367 non-null    int64  
 8   LoanAmount         362 non-null    float64
 9   Loan_Amount_Term   361 non-null    float64
 10  Credit_History     338 non-null    float64
 11  Property_Area      367 non-null    object 
dtypes: float64(3), int64(2), object(7)
memory usage: 34.5+ KB
```

Exploratory Data Analysis (EDA)

Univariate Analysis

Univariate analysis is used in this section to analyze each variable individually.

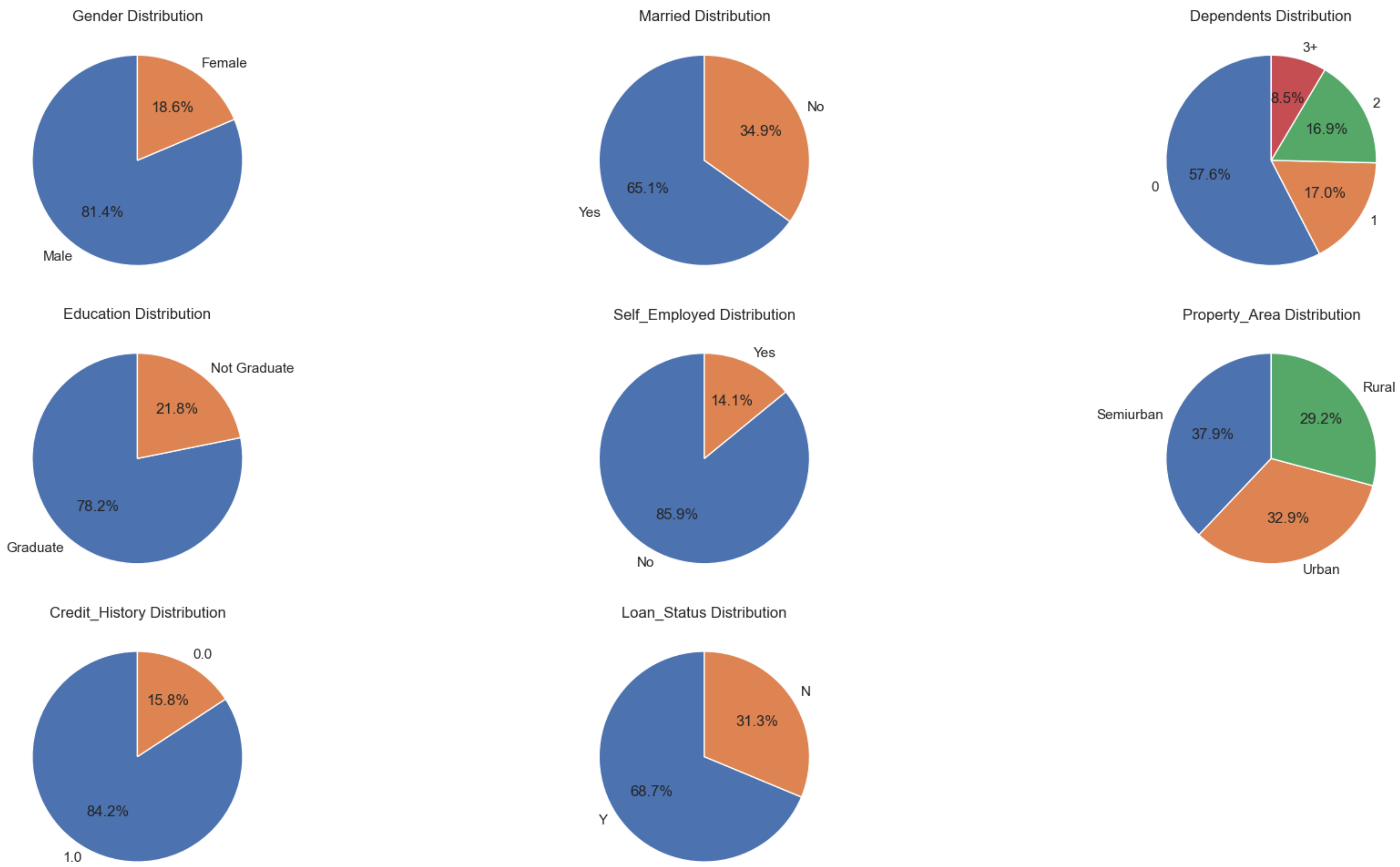
For numerical features ,we can use Probability Density Functions(PDF) to look at the distribution of the numerical variables.

For categorical features ,frequency tables or bar plots can be used to calculate the number of each category in a particular variable.

Categorical Features

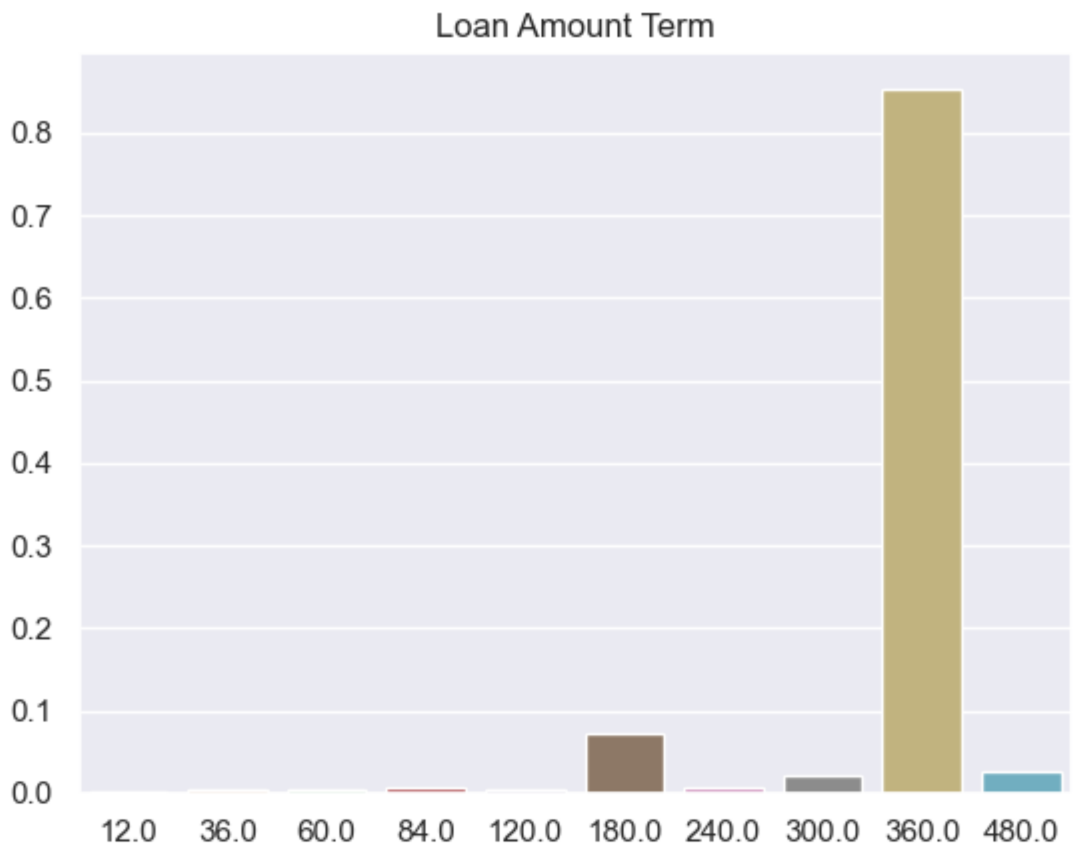
In [8]:
```
categorical_var = ['Gender', 'Married', 'Dependents', 'Education', 'Self_Employed', 'Property_Area', 'Credit_History', 'Loa

def visualize_categorical_data(df, columns, nrows, ncols, figsize):
    """
    Creates a grid of pie charts to visualize the distribution of categorical features.

    Parameters:
    -----------
    df : pandas.DataFrame
        The input dataframe containing the categorical features to be plotted.
    columns : list
        A list of column names corresponding to the categorical features to be plotted.
    nrows : int
        The number of rows in the subplot grid.
    ncols : int
        The number of columns in the subplot grid.
    figsize : tuple
        The size of the plot figure in inches, specified as a tuple (width, height).

    Returns:
    --------
    None
        Displays the plot figure.
    """

    fig, axs = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
    for i, column in enumerate(columns):

        # Each category value count
        val_count = df[column].value_counts()

        # Create a pie chart
        axs.flat[i].pie(val_count, labels=val_count.index, autopct='%1.1f%%', startangle=90)

        # Set a title for each subplot
        axs.flat[i].set_title(f'{column} Distribution')

    # Remove empty subplots
    if len(columns) < nrows * ncols:
        for i in range(len(columns), nrows * ncols):
            fig.delaxes(axs.flat[i])

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

In [9]:
```
visualize_categorical_data(train, categorical_var, 3, 3, (20, 10))
```



In [10]:
```
Loan_Amount_Term_values = train['Loan_Amount_Term'].value_counts(sort=True, normalize=True)
sns.barplot(x=Loan_Amount_Term_values.index, y=Loan_Amount_Term_values.values, )
plt.title('Loan Amount Term')
```

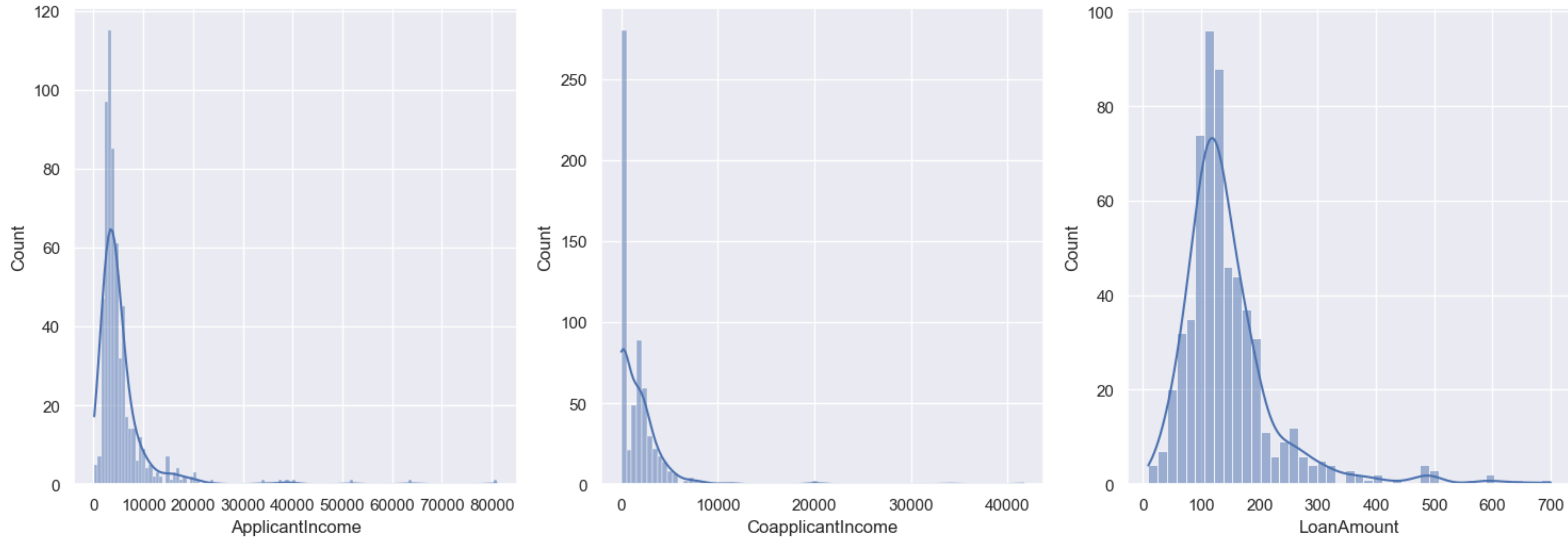
Out[10]: Text(0.5, 1.0, 'Loan Amount Term')



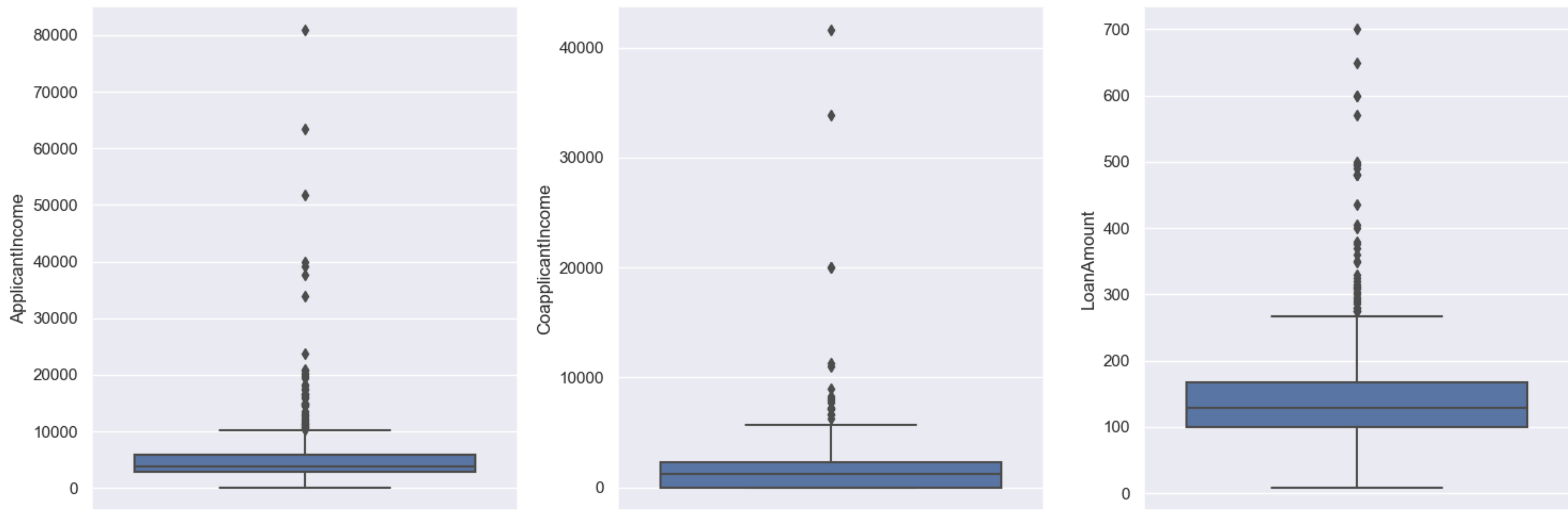
Numerical Features

In [11]:
```
numerical_var = ['ApplicantIncome', 'CoapplicantIncome', 'LoanAmount']

def visualize_numerical_data(df, columns, nrows, ncols, figsize, plot_type='hist'):

    """
        Creates a grid of plots to visualize the distribution of numerical features.

    Parameters:
    -----------
    df : pandas.DataFrame
        The input dataframe containing the numerical features to be plotted.
    columns : list
        A list of column names corresponding to the numerical features to be plotted.
    nrows : int
        The number of rows in the subplot grid.
    ncols : int
        The number of columns in the subplot grid.
    figsize : tuple
        The size of the plot figure in inches, specified as a tuple (width, height).
    plot_type : str, optional
        The type of plot to create for each feature. Valid options are 'hist' (default),
        'box', and 'violin'.

    Returns:
    --------
    None
        Displays the plot figure.
    """


    fig, axs = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)

    for i, column in enumerate(columns):
        if plot_type == 'box':
            sns.boxplot(y=column, data=df, ax=axs.flat[i])
        elif plot_type == 'violin':
                sns.violinplot(y=column, data=df, ax=axs.flat[i])
        elif plot_type == 'hist':
            sns.histplot(data=df, x=column, ax=axs.flat[i], kde=True, stat='count')

    # Remove empty subplots
    if len(columns) < nrows * ncols:
        for i in range(len(columns), nrows * ncols):
            fig.delaxes(axs.flat[i])

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

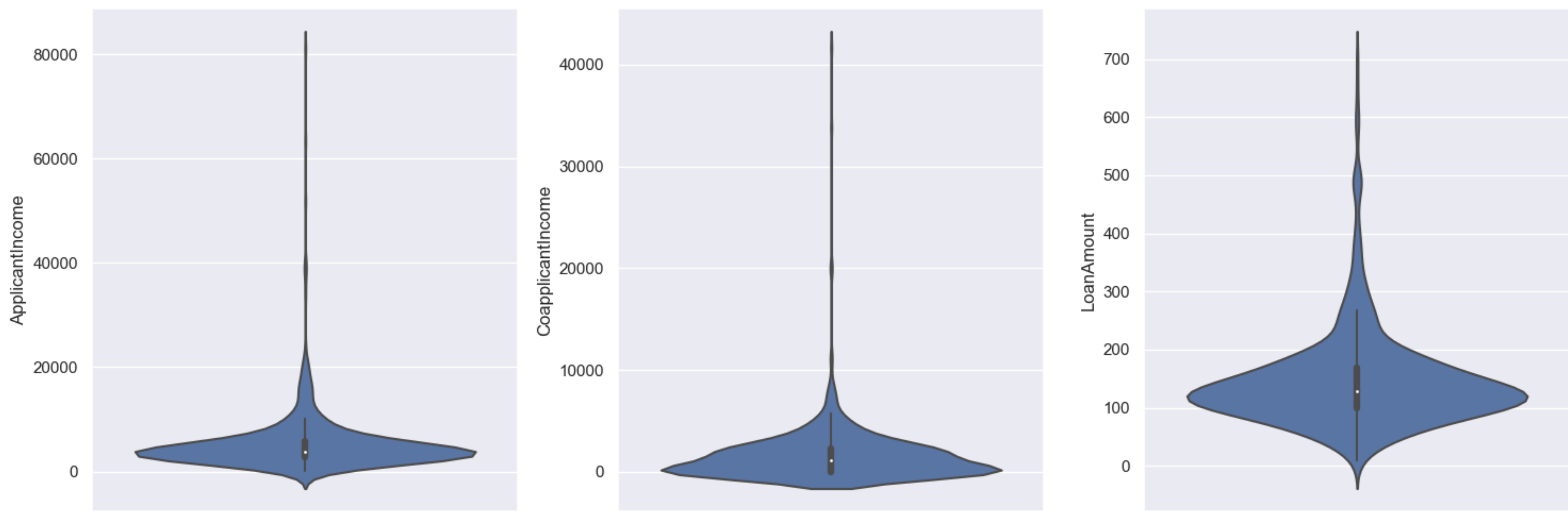
In [12]:
```
visualize_numerical_data(train, numerical_var, 2, 3, (15, 10))
```



In [13]:
```
visualize_numerical_data(train, numerical_var, 2, 3, (15, 10), 'box')
```



In [14]:
```
visualize_numerical_data(train, numerical_var, 2, 3, (15, 10), 'violin')
```



Insights from the univariate analysis.

- 81.4% of applicants in the dataset are male.
- Around 65% of the applicants in the dataset are married.
- Most of the applicants don't have dependents.
- 78.2% of the applicants are graduates.
- About 15% of applicants in the dataset are self-employed.
- About 85% of the applicants chosed the loan on 360 months.
- 84.2% of applicants have repaid their debts.
- Most of the applicants are from semi-urban areas.
- 68.7% of the applicants got the approval.
- The applicant income and coapplicant income has a similiar extremely left-skewed distribution.
- The loan amount is fairly normal but contains outliers.

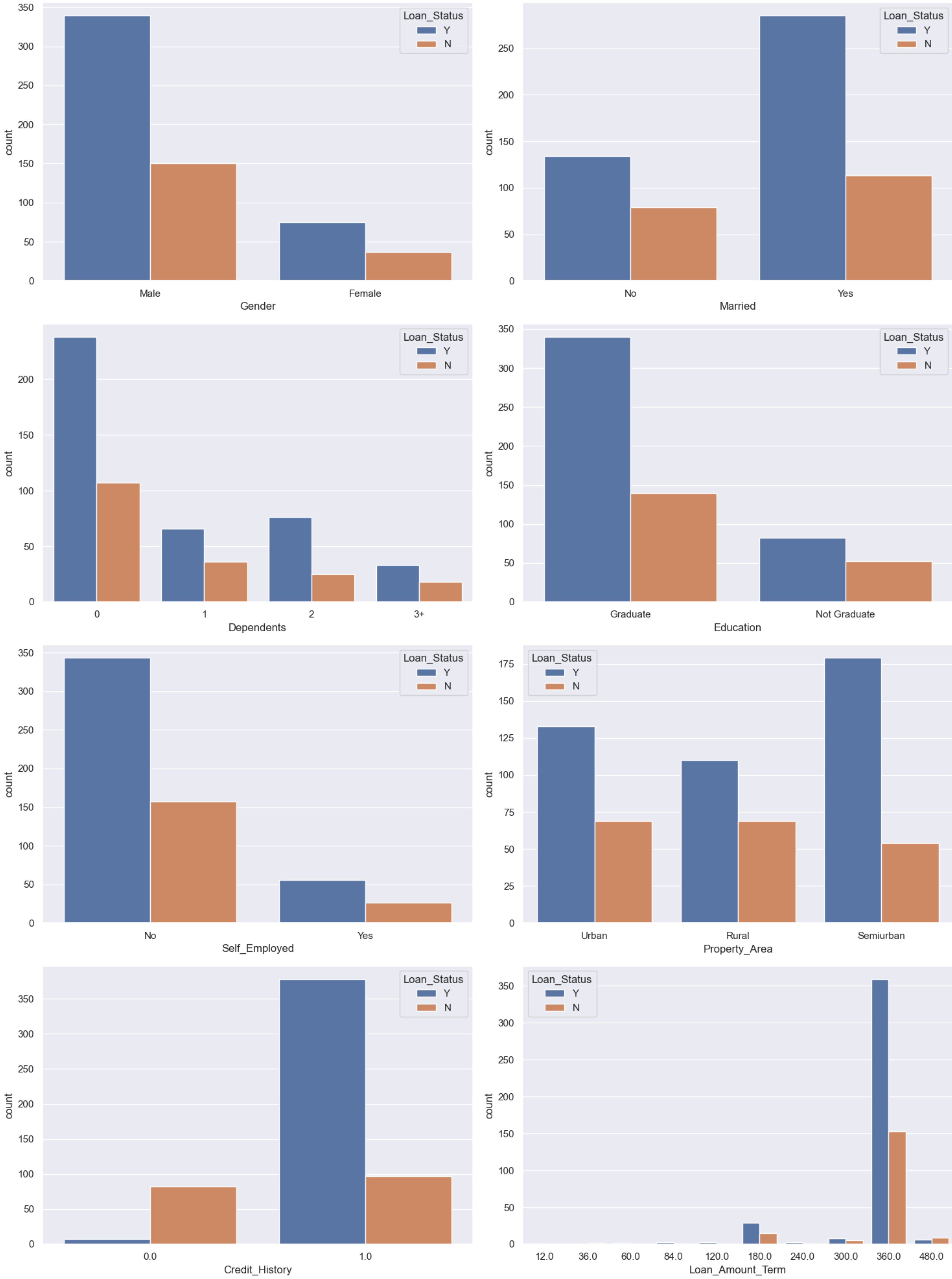
Bivariate Analysis

Bivariate Analysis is used in this section to know how well each feature correlates with Loan Status.

Categorical Features vs Target Variable

In [15]:
```
def visualize_categorical_data_with_target(df, columns, target, nrows, ncols, figsize):
    """
        Creates a grid of count plots to visualize the relationship between categorical
        features and a target variable.

    Parameters:
    -----------
    df : pandas.DataFrame
        The input dataframe containing the categorical features and target variable.
    columns : list
        A list of column names corresponding to the categorical features to be plotted.
    target : str
        The name of the target variable column in the dataframe.
    nrows : int
        The number of rows in the subplot grid.
    ncols : int
        The number of columns in the subplot grid.
    figsize : tuple
        The size of the plot figure in inches, specified as a tuple (width, height).

    Returns:
    --------
    None
        Displays the plot figure.
    """

    fig, axs = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
    for i, column in enumerate(columns):
        sns.countplot(x=column, hue=target, data=df, ax=axs.flat[i])

    # Remove empty subplots
    if len(columns) < nrows * ncols:
        for i in range(len(columns), nrows * ncols):
            fig.delaxes(axs.flat[i])

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

In [16]:
```
visualize_categorical_data_with_target(train, categorical_var[:-1] + ['Loan_Amount_Term'], 'Loan_Status', 4, 2, (15, 20))
```



Numerical Features vs Target Variable

In [17]:
```
train[["ApplicantIncome", "CoapplicantIncome", "LoanAmount"]].describe()
```

Out[17]:

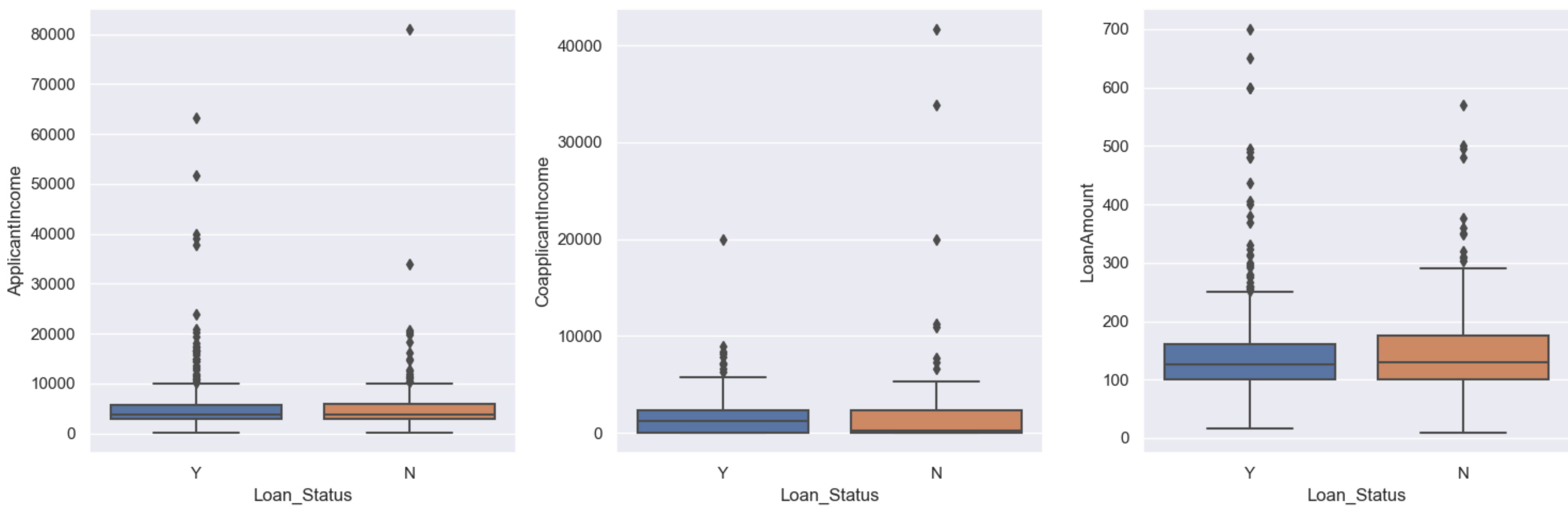
	ApplicantIncome	CoapplicantIncome	LoanAmount
count	614.000000	614.000000	592.000000
mean	5403.459283	1621.245798	146.412162
std	6109.041673	2926.248369	85.587325
min	150.000000	0.000000	9.000000
25%	2877.500000	0.000000	100.000000
50%	3812.500000	1188.500000	128.000000
75%	5795.000000	2297.250000	168.000000
max	81000.000000	41667.000000	700.000000

In [18]:

```python
def visualize_numerical_data_with_target(df, columns, target, nrows, ncols, figsize, plot_type='box'):
    """
    Visualize the relationship between numerical features and a target variable.

        Parameters:
    df : pandas.DataFrame
        The input dataframe containing the numerical features and target variable.
    columns : list
        A list of column names corresponding to the numerical features to be plotted.
    target : str
        The name of the target variable column in the dataframe.
    nrows : int
        The number of rows in the subplot grid.
    ncols : int
        The number of columns in the subplot grid.
    figsize : tuple
        The size of the plot figure in inches, specified as a tuple (width, height).
    plot_type : str, optional
        The type of plot to create for each feature. Valid options are 'box' (default)
        and 'violin'.

    Returns:
    None
        Displays the plot figure.
    """

    fig, axs = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
    for i, column in enumerate(columns):
        if plot_type == 'box':
            sns.boxplot(x=target, y=column, data=df, ax=axs.flat[i])
        elif plot_type == 'violin':
            sns.violinplot(x=target, y=column, data=df, ax=axs.flat[i])

    # Remove empty subplots
    if len(columns) < nrows * ncols:
        for i in range(len(columns), nrows * ncols):
            fig.delaxes(axs.flat[i])

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

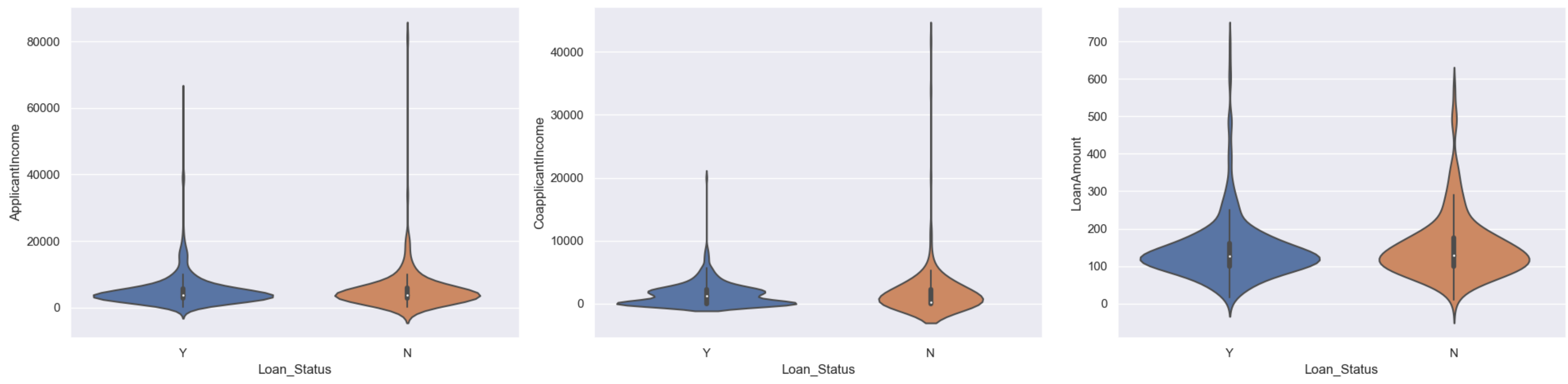
In [19]:

```python
visualize_numerical_data_with_target(train, numerical_var, 'Loan_Status', 1, 3, (15, 5))
```



In [20]:

```python
visualize_numerical_data_with_target(train, numerical_var, 'Loan_Status', 1, 3, (20, 5), 'violin')
```



Insights from the bivariate analysis

- Gender and Self_Employed features don't seem to have any impact on the loan status.
- Married applicants are more likely to be approved for loans.
- Applicants with 1 or 3+ dependents are less likely to be approved for loans.
- Applicants with credit history as 1 are more likely to be approved.
- Applicants who are not graduates are less likely to be approved.
- Applicants from Semiurban areas are more likely to be approved for loans.
- Applicant income and coapplicante income do not affect the chances of loan approval.

Data Preprocessing

Missing Value Treatment

The `check_missing(df)` function below takes a dataframe as an input and outputs the count of null values for each variable.

In [21]:
```
def check_missing(df):
    return df.isnull().sum().sort_values(ascending=False)
```

There are missing values in Gender, Married, Dependents, Self_Employed, LoanAmount, Loan_Amount_Term, and Credit_History features.

In [22]:
```
check_missing(train)
```

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

Imputing Categorical Missing Values

In [23]:
```
# Imputing categorical missing values
def impute_categorical(df):
    for column in df.columns:
        if df[column].dtype == 'object':
            df[column].fillna(df[column].mode()[0], inplace=True)
    return df
```

In [24]:
```
impute_categorical(train)
check_missing(train)
```

Out[24]:
```
Credit_History       50
LoanAmount           22
Loan_Amount_Term     14
Loan_ID               0
Gender                0
Married               0
Dependents            0
Education             0
Self_Employed         0
ApplicantIncome       0
CoapplicantIncome     0
Property_Area         0
Loan_Status           0
dtype: int64
```

In [25]:
```
impute_categorical(test)
check_missing(test)
```

Out[25]:
```
Credit_History       29
Loan_Amount_Term      6
LoanAmount            5
Loan_ID               0
Gender                0
Married               0
Dependents            0
Education             0
Self_Employed         0
ApplicantIncome       0
CoapplicantIncome     0
Property_Area         0
dtype: int64
```

Imputing Numerical Missing Values

Note: I choosed to impute the missing values with the median of the column because there are outliers in the numerical features.

In [26]:
```
# Treating numerical missing values
def impute_numerical(df, method):
    for column in df.columns:
        if df[column].dtype != 'object':
            if method == 'mean':
                df[column].fillna(df[column].mean(), inplace=True)
            elif method == 'median':
                df[column].fillna(df[column].median(), inplace=True)
    return df
```

In [27]:
```
impute_numerical(train, 'median')
check_missing(train)
```

Out[27]:
```
Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
Loan_Status          0
dtype: int64
```

In [28]:
```
impute_numerical(test, 'median')
check_missing(test)
```

Out[28]:
```
Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
dtype: int64
```

In [29]:
```
train.drop(['Loan_ID'], axis=1, inplace=True)
test.drop(['Loan_ID'], axis=1, inplace=True)
```

Categorical Features Encoding

In [30]:
```
for column in train.columns:
    if train[column].dtype == 'object' or train[column].name == 'Loan_Amount_Term':
        print(f'{column} : {train[column].unique()}')
```

```
Gender : ['Male' 'Female']
Married : ['No' 'Yes']
Dependents : ['0' '1' '2' '3+']
Education : ['Graduate' 'Not Graduate']
Self_Employed : ['No' 'Yes']
Loan_Amount_Term : [360. 120. 240. 180.  60. 300. 480.  36.  84.  12.]
Property_Area : ['Urban' 'Rural' 'Semiurban']
Loan_Status : ['Y' 'N']
```

```
In [31]: # Encoding categorical variables
def encode_categorical(df):
    for column in df.columns:
        if df[column].dtype == 'object' or df[column].name == 'Loan_Amount_Term':
            le = LabelEncoder()
            le.fit(df[column].unique())
            df[column] = le.transform(df[column])
            print (f"{column}: {df[column].unique()}")
```

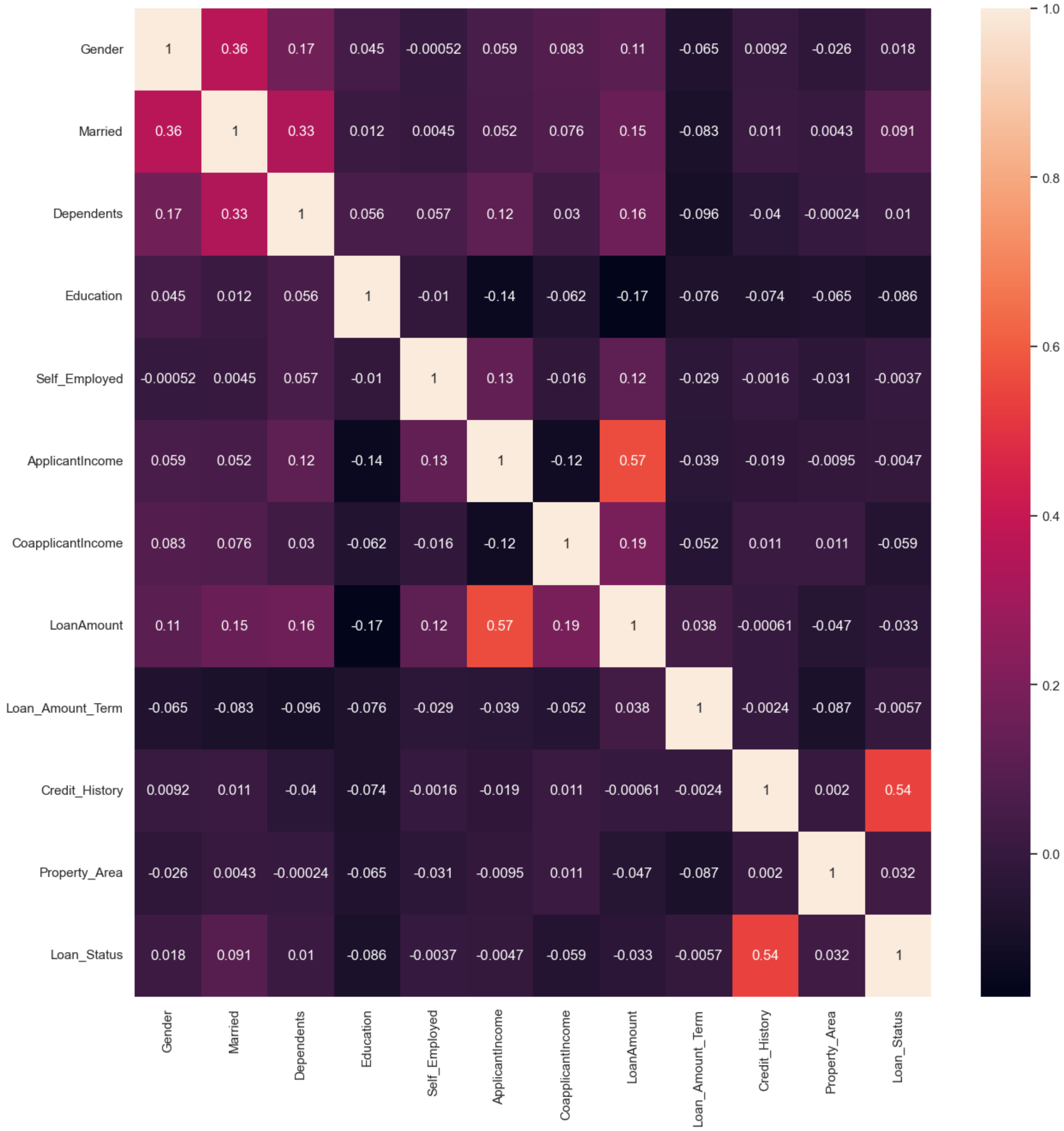
```
In [32]: encode_categorical(train)
```

```
Gender: [1 0]
Married: [0 1]
Dependents: [0 1 2 3]
Education: [0 1]
Self_Employed: [0 1]
Loan_Amount_Term: [8 4 6 5 2 7 9 1 3 0]
Property_Area: [2 0 1]
Loan_Status: [1 0]
```

Correlation Matrix

```
In [33]: plt.figure(figsize=(15, 15))
sns.heatmap(train.corr(), fmt='.2g', annot=True)
```

```
Out[33]: <AxesSubplot:>
```

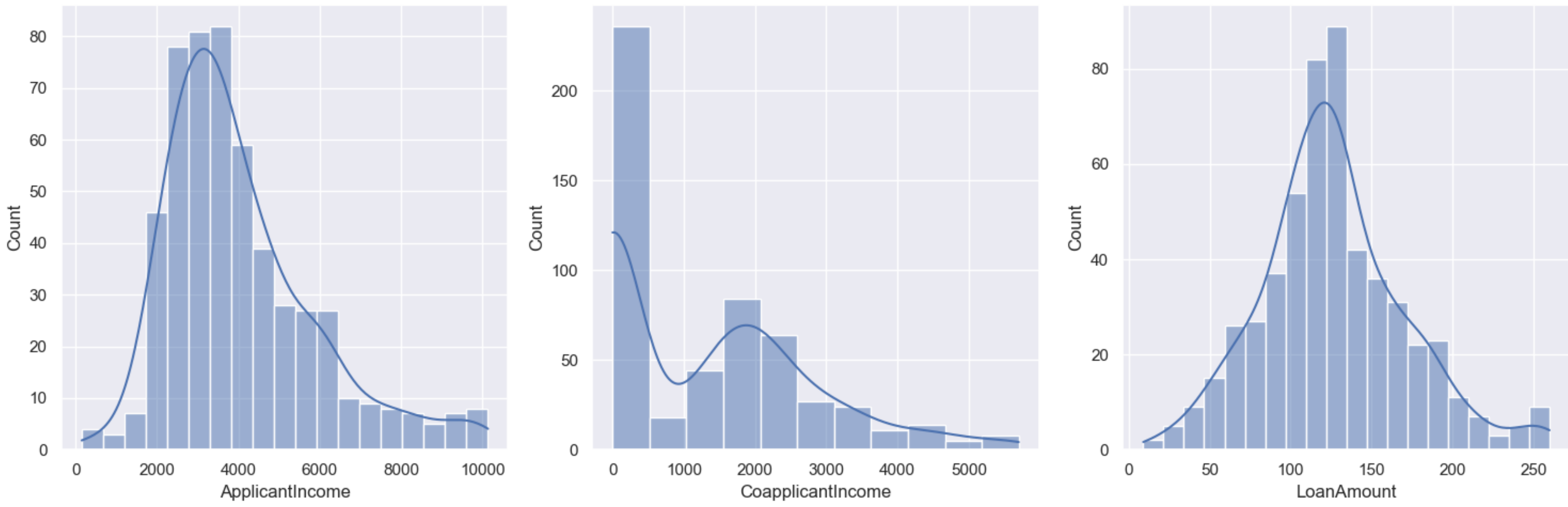


Outliers Treatment

In [34]:
```
q1 = train[numerical_var].quantile(0.25)
q3 = train[numerical_var].quantile(0.75)

iqr = q3 - q1

Lower_tail = q1 - 1.5 * iqr
Upper_tail = q3 + 1.5 * iqr
Lower_tail, Upper_tail

train = train[~((train[numerical_var] < Lower_tail) | (train[numerical_var] > Upper_tail)).any(axis=1)]
train.shape
```

Out[34]:
```
(535, 12)
```

In [35]:
```
visualize_numerical_data(train, numerical_var, 1, 3, (15, 5), 'hist')
```



Developing the Model

Splitting the dataset

In [36]:
```
X = train.drop('Loan_Status', axis=1)
y = train.Loan_Status
```

Normalize

In [37]:
```
X = StandardScaler().fit_transform(X)
```

In [38]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[38]:
```
((428, 11), (107, 11), (428,), (107,))
```

Logistic Regression

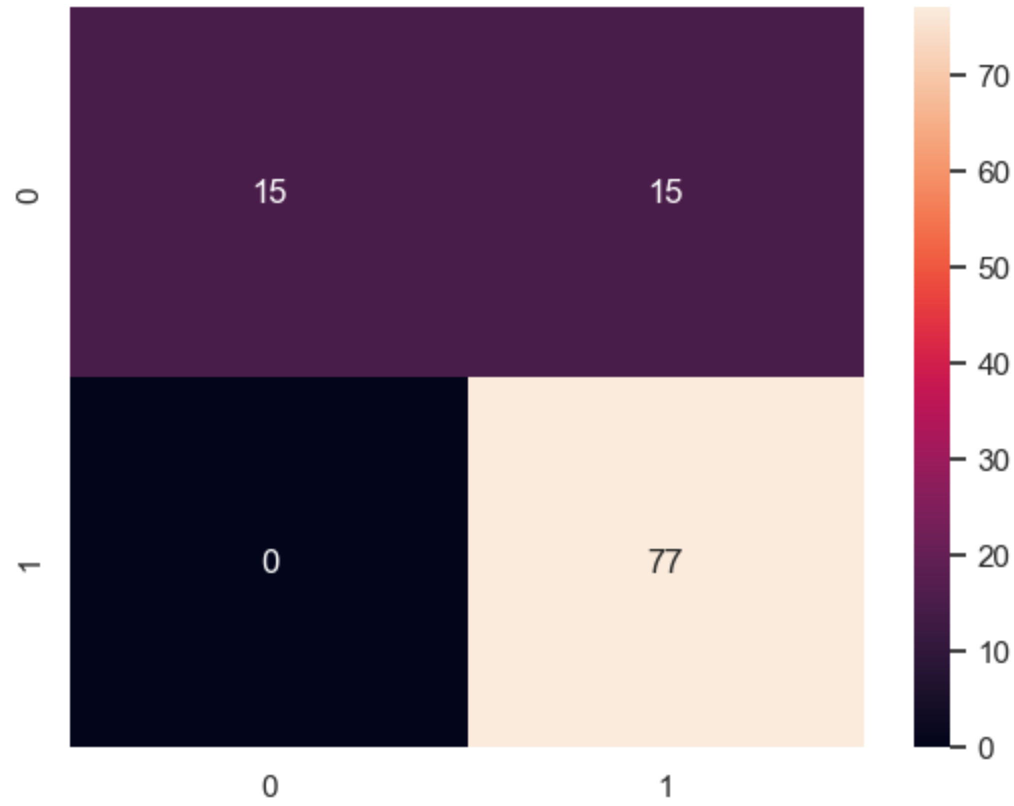
In [39]:
```
LR_model = LogisticRegression()
LR_model.fit(X_train, y_train)
```

Out[39]:
```
▾ LogisticRegression
LogisticRegression()
```

In [40]:
```
predict = LR_model.predict(X_test)
```

In [41]:
```
cm = confusion_matrix(y_test, predict)
sns.heatmap(cm, annot=True)
```

Out[41]:
```
<AxesSubplot:>
```



In [42]:
```
print(classification_report(y_test, predict))
```

```
              precision    recall  f1-score   support

           0       1.00      0.50      0.67        30
           1       0.84      1.00      0.91        77

    accuracy                           0.86       107
   macro avg       0.92      0.75      0.79       107
weighted avg       0.88      0.86      0.84       107
```

In [43]:
```
cross_val_score(LR_model, X_train, y_train, cv=5, scoring='f1').mean()
```

Out[43]: 0.8791454488461581

In [44]:
```
cross_val_score(LR_model, X_test, y_test , cv=5, scoring='f1').mean()
```

Out[44]: 0.8990355233002292

Decision Tree

In [45]:
```
DT_model = DecisionTreeClassifier()
```

In [46]:
```
param_grid = {'max_features': ['sqrt'],
              'max_depth' : [3, 4, 5, 6, 7, 8, 9],
              'min_samples_split': [2, 3, 4, 5],
              'criterion' :['gini', 'entropy'],
              'random_state': [0, 42]
             }
```

In [47]:
```
grid_search = GridSearchCV(estimator=DT_model, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

Out[47]:

```
▸ GridSearchCV
▸ estimator: DecisionTreeClassifier
  ▸ DecisionTreeClassifier
```

In [48]:
```
print(grid_search.best_params_)
print(grid_search.best_score_)
```

```
{'criterion': 'gini', 'max_depth': 3, 'max_features': 'sqrt', 'min_samples_split': 2, 'random_state': 42}
0.8084541723666211
```

In [49]:
```
print(grid_search.best_estimator_)
```

```
DecisionTreeClassifier(max_depth=3, max_features='sqrt', random_state=42)
```

In [60]:
```
DT_model = DecisionTreeClassifier(criterion='gini', max_depth=3, max_features='sqrt', min_samples_split=2, random_state=42)
DT_model.fit(X_train, y_train)
```

Out[60]:

```
▾ DecisionTreeClassifier
DecisionTreeClassifier(max_depth=3, max_features='sqrt', random_state=42)
```

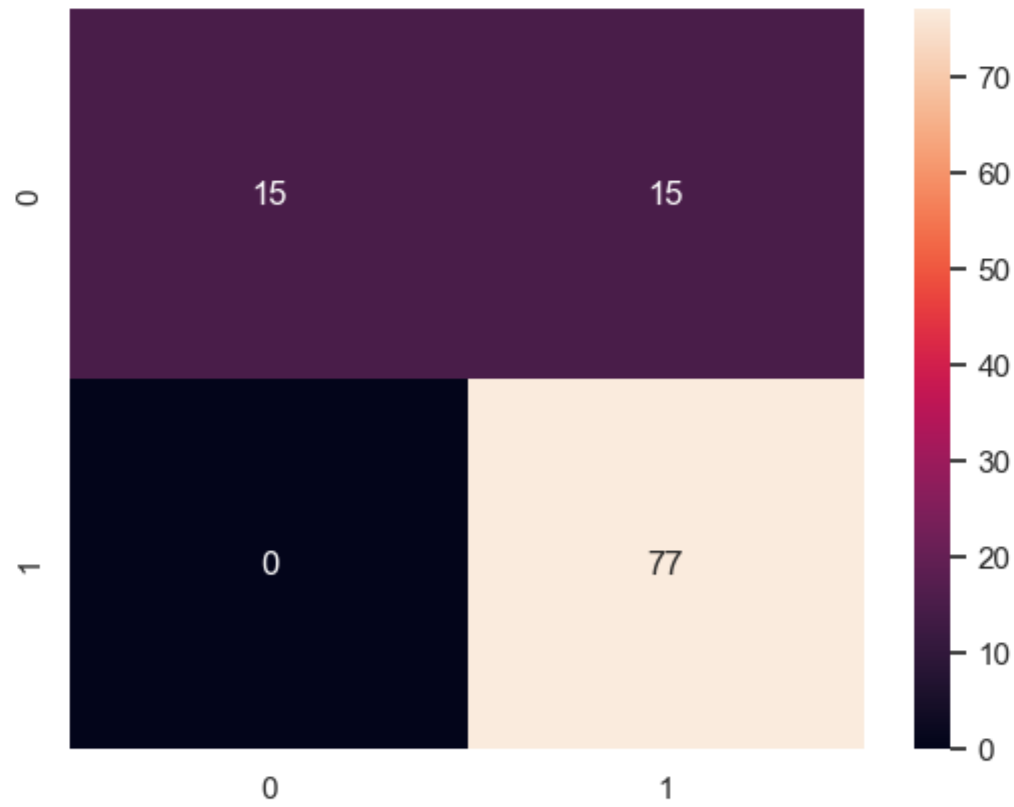
In [51]:
```
DT_predict = DT_model.predict(X_test)
```

In [52]:
```
print(round(DT_model.score(X_test, y_test) * 100, 2), "%")
```

```
85.98 %
```

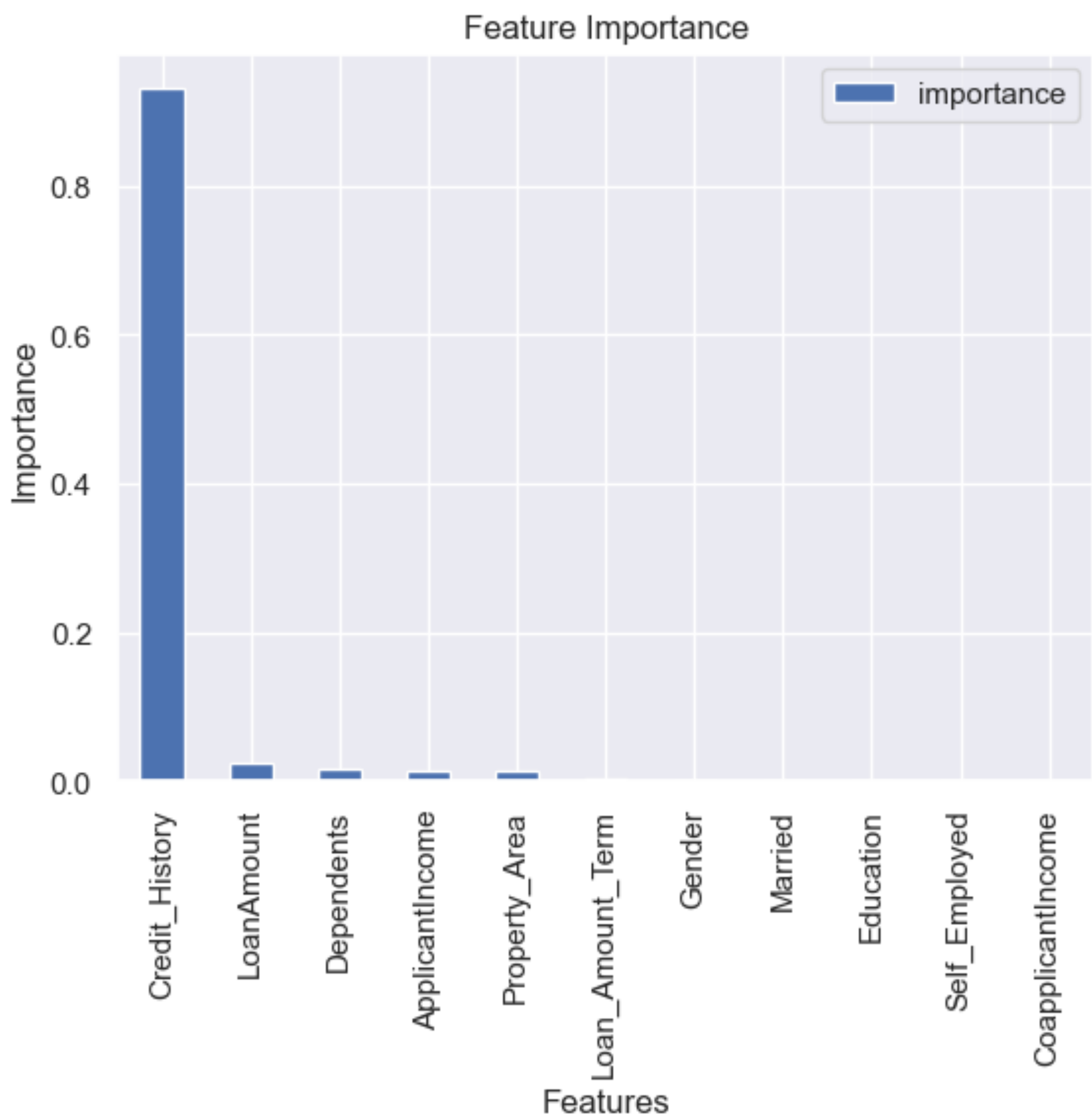
In [53]:
```
cm_DT = confusion_matrix(y_test, DT_predict)
sns.heatmap(cm_DT, annot=True)
```

Out[53]:
```
<AxesSubplot:>
```



In [54]:
```
# Feature importance
feature_importance = pd.DataFrame({'feature': train.drop('Loan_Status', axis=1).columns,
'importance': DT_model.feature_importances_})

feature_importance.sort_values(by='importance', ascending=False, inplace=True)
feature_importance.plot.bar(x='feature', y='importance')

plt.title('Feature Importance')
plt.xlabel('Features')
plt.ylabel('Importance')
```

Out[54]:
```
Text(0, 0.5, 'Importance')
```



In [55]:
```
feature_importance.head()
```

Out[55]:

	feature	importance
9	Credit_History	0.930481
7	LoanAmount	0.023880
2	Dependents	0.015436
5	ApplicantIncome	0.013169
10	Property_Area	0.013002

Random Forest

In [56]:
```
RF_model = RandomForestClassifier()

rf_param_grid = { 'n_estimators': [10, 500, 1000 ],
              'max_depth' : [2, 5, 8, 10],
              'min_samples_split': [2, 3, 5],
              'random_state': [0, 42]
             }
```

In [57]:
```
rf_grid_search = GridSearchCV(estimator=RF_model, param_grid=rf_param_grid, cv=5)
rf_grid_search.fit(X_train, y_train)

print(rf_grid_search.best_params_)
print(rf_grid_search.best_score_)
```

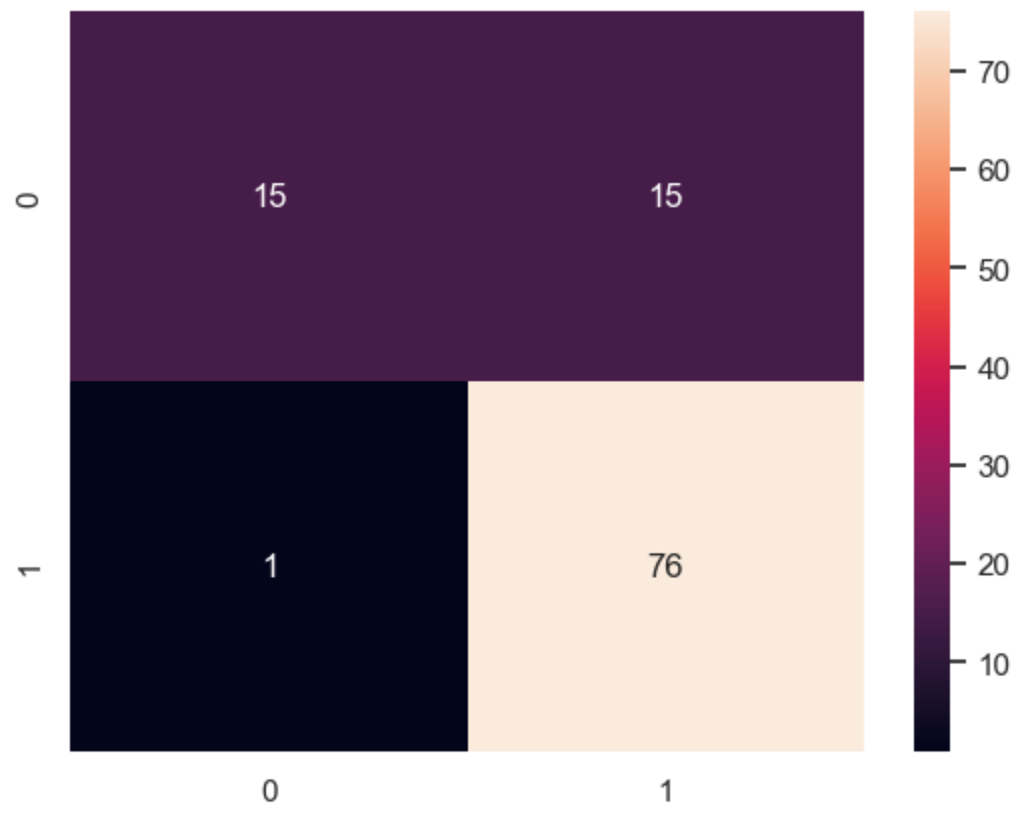
```
{'max_depth': 5, 'min_samples_split': 5, 'n_estimators': 10, 'random_state': 42}
0.8154856361149111
```

In [58]:
```
RF_model = RandomForestClassifier(max_depth=5, min_samples_split=5, n_estimators=10, random_state=42)
RF_model.fit(X_train, y_train)

RF_predict = RF_model.predict(X_test)

print(round(RF_model.score(X_test, y_test) * 100, 2), "%")
```

```
85.05 %
```

In [59]:
```
cm_RF = confusion_matrix(y_test, RF_predict)
sns.heatmap(cm_RF, annot=True)
```

Out[59]: <AxesSubplot:>



In [ ]: