Project Details:

Title: Guided Project

Due Date: 9 May 2024

Contributors: Marcus Mahlatjie (577296) and Zoë Treutens (577989)

GitHub Link: https://github.com/Zoe21354/SeperateMLG2Project.git

Streamlining BC Finance’s Home Loan Eligibility Process

BC Finance Company provides financial services across all home loan categories. Offering services to clients in urban, semi-urban, and rural areas, the organization has many facets. The organization currently uses an **ineffective manual** procedure to validate customers' eligibility. The procedure entails the client submitting an application for a home loan by answering questions and supplying personal information. These responses must then go through a lengthy validation process and this can be a problem for handling multiple applications leading to decreased customer satisfaction, manual errors and **lengthy application times** which could lead to customers seeking other financial institutes to provide financial services for their needs.

The organization is working to create an **automated system** that can accurately determine a customer's eligibility for a home loan in real time in order to address this problem. To ascertain if a customer is eligible for a loan, this system will examine a number of customer variables, including gender, marital status, education, number of dependents, income, loan amount, credit history, and others.

The principal aim is to divide clients into discrete categories according to their loan quantum eligibility. By doing this, BC Finance hopes to **efficiently and successfully target** these consumer segments and provide them with loan products and services that are customized to their unique requirements and preferences. BC Finance hopes to improve client happiness, reduce manual errors, and streamline its lending procedures for long-term profitability and growth by putting in place an automated loan qualifying system.

This notebook will take the following structure:

==

1. Prepare Data

Before any coding can take place, certain libraries in python need to be imported to perform different functions and make various features available for use.

In [176]:

```
# Import Libraries
import csv                                                              # Handles CSV f
ile operations
import pandas as pd                                                     # Data manipula
tion and analysis
import numpy as np                                                      # Performs math
ematical operations
import matplotlib.pyplot as plt                                         # Creates static
, animated, and interactive visualizations
import seaborn as sns                                                   # Creates attra
ctive and informative statistical graphics
from sklearn.model_selection import train_test_split                    # Splits data in
to random train and test subsets
from scipy import stats                                                 # Provides stat
istical functions
import pickle                                                           # Serializes an
d de-serializes Python object structures
from sklearn.model_selection import StratifiedKFold, cross_val_predict  # Provides train/
test indices to split data in train/test sets
from sklearn.linear_model import LogisticRegression                     # Implements log
istic regression
from sklearn.metrics import accuracy_score                              # Computes subse
t accuracy classification score
from tensorflow.keras.models import Sequential                          # Base class for
building neural network models
from tensorflow.keras.layers import Dense                               # Implements the
operation: output = activation(dot(input, kernel) + bias)
from sklearn import tree                                                # Contains clas
ses for different decision tree algorithms
import warnings                                                         # Handles warni
ngs during runtime
warnings.filterwarnings('ignore')                                       # Ignores displa
ying warnings
```

The CSV files named raw_data and validation_data are read so that the unclean data contained in these files can be analyses.

In [177]:

```
# Read Unclean CSV Files
raw_data = pd.read_csv("raw_data.csv")
raw_data_copy = raw_data.copy()

validation_data = pd.read_csv("validation.csv")
validation_data_copy = validation_data.copy()
```

DATA ANALYSIS PROCESSES

Performing data analysis on unclean data is essential, as it will provide pertinant information regarding the data we are to use. Although, data cleaning is an essential step in the data analysis process, understanding your data before you clean it can make the process more efficient and effective.

Understanding the Data **It allows you to understand the nature and structure of your data. You can identify the types of variables you have, their distribution, and how they relate to each other.**

Identifying Errors and Anomalies **Unclean data can contain errors, outliers, or anomalies that need to be**

Identifying Errors and Anomalies Unclean data can contain errors, outliers, or anomalies that need to be addressed. By analysing the data first, you can identify these issues and plan how to handle them during the cleaning process.

Determining Cleaning Strategies Not all data requires the same cleaning procedures. Analysing the data can help you determine the most appropriate cleaning strategies for your specific dataset.

Preserving Valuable Information Sometimes, what might initially appear as an error or outlier could actually be a valuable piece of information. Analysing the data before cleaning ensures that you don't inadvertently remove these insights.

Improving Model Accuracy Unclean data can lead to inaccurate models. By analysing and cleaning your data, you can improve the accuracy of your subsequent models.

A. Dataset Analysis

Dataset Attributes:

Each attribute in the dataset represents a different variable. Understanding these attributes helps you understand the variables you're working with, what they represent, and how they might relate to your research question or problem statement.

- Feature Variable (Independent variables) are variables that stand alone and are not changed by other variables that are being measured. They are denoted as X in ML algorithms.
- Target Variables (Dependent variables) are the variables that are to be predicted. It is often denoted as Y in ML algorithms.

In [178]:

```
print(f"Raw Data Columns:\n{raw_data_copy.columns}\n")
print(f"Validation Data Columns:\n{validation_data_copy.columns}\n")
```

```
Raw Data Columns:
Index(['Loan_ID', 'Gender', 'Married', 'Dependents', 'Education',
       'Self_Employed', 'ApplicantIncome', 'CoapplicantIncome', 'LoanAmount',
       'Loan_Amount_Term', 'Credit_History', 'Property_Area', 'Loan_Status'],
      dtype='object')

Validation Data Columns:
Index(['Loan_ID', 'Gender', 'Married', 'Dependents', 'Education',
       'Self_Employed', 'ApplicantIncome', 'CoapplicantIncome', 'LoanAmount',
       'Loan_Amount_Term', 'Credit_History', 'Property_Area'],
      dtype='object')
```

Insight Gained:

- In both datasets, the attribute names are written inconsistently. Some attributes have underscores between each word ie. Loan_ID and other attributes use PascalCase i.e ApplicantIncome. This will need to be standardized in the data processing section.
- For both datasets, there are 12 feature variables but only the "raw_data" dataset has 1 target variable.
- The target variable in the raw_data dataset is the Loan_Status attribute.
- This variable will be predicted using models for the "validation_data" dataset.

Dataset Datatypes:

Attributes can have different data types, such as numerical, categorical, or ordinal. Knowing the data type of each attribute is important because it determines what kind of statistical analysis or data processing is appropriate. Learning the different datatypes for each attribute in both of the datasets will provide insight into the consistance of the datattypes for each specific attribute.

In [179]:

```
print(f"Raw Dataset Datatypes:\n{raw_data_copy.dtypes}\n")
print(f"Raw Dataset Datatypes:\n{raw_data_copy.dtypes}\n")
print(f"Validation Dataset Datatypes:\n{validation_data_copy.dtypes}\n")
```

```
Raw Dataset Datatypes:
Loan_ID                object
Gender                 object
Married                object
Dependents            float64
Education              object
Self_Employed          object
ApplicantIncome         int64
CoapplicantIncome       int64
LoanAmount            float64
Loan_Amount_Term      float64
Credit_History        float64
Property_Area          object
Loan_Status            object
dtype: object

Validation Dataset Datatypes:
Loan_ID                object
Gender                 object
Married                object
Dependents             object
Education              object
Self_Employed          object
ApplicantIncome         int64
CoapplicantIncome       int64
LoanAmount            float64
Loan_Amount_Term      float64
Credit_History        float64
Property_Area          object
dtype: object
```

Insight Gained:

- **There is a discrepancy between the two datasets: the "Dependents" attribute is of datatype float64 in the "raw_data" dataset but of datatype object is seen in the "validation_data" file.**
- **This could lead to potentially issues when modeling, as the model might be expecting the same datatype for a given attribute.**
- **This discrepancy will need to be fixed in the data processing section.**

Dataset Shape:

Knowing the number of rows in your datasets provides you with an idea of the volume of the data available to you. More rows mean more data, which can lead to more robust and reliable models. However, it can also mean more computational resources and time required for processing. On the other hand knowing the number of columns in the dataset informs the user on the number of features (or variables) available. Overall the analysis of the shape of the dataset can help in assessing the quality of the data.

For example, if you have many rows but few columns, you might not have enough features to build a good model. Conversely, having a large number of columns compared to rows could lead to overfitting.

In [180]:

```
print(f"Raw Data Shape: {raw_data_copy.shape}\n")
print(f"Validation Data Shape:{validation_data_copy.shape}")
```

```
Raw Data Shape: (614, 13)

Validation Data Shape:(367, 12)
```

Insight Gained:

- **Raw Data Shape: 614 rows and 13 columns**
- **Validation Data Shape: 367 rows and 12 columns**
- **There is exactly 1 less column in the "validation_data" file which is the target attribute**

- There is exactly 1 less column in the "validation_data" file which is the target attribute.

B. Univariate Analysis

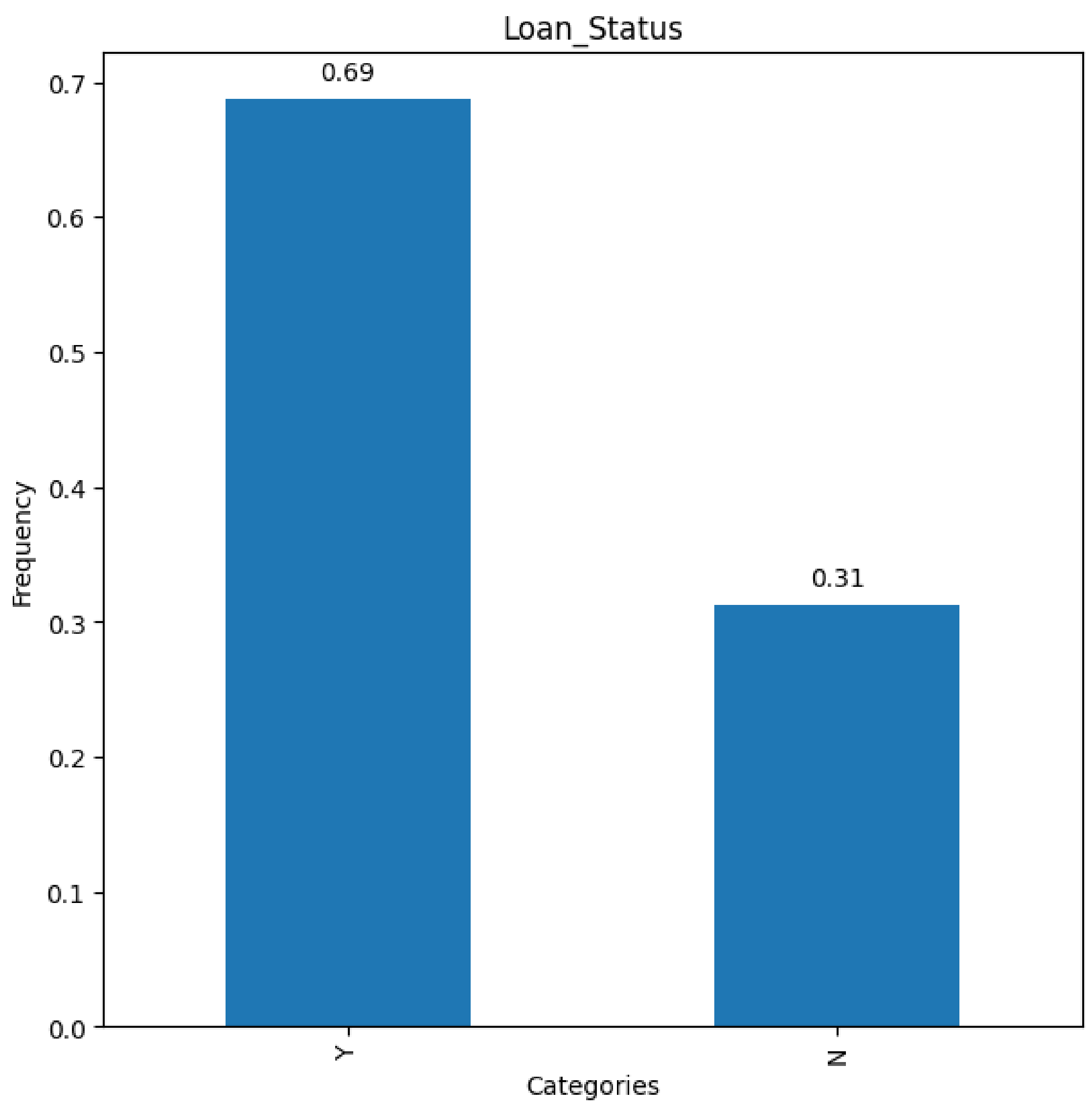
Univariate analysis is the process of analysising individual (one variable) at a time. This is the most basic type of data analysis to finds patterns in the data.

Analyzing univariate data involves examining the frequency of data in the dataset. In order to do this the count for each category in the attribute is found. Following this the data is normalized to get the proportion of the different categories through the division of the count by the total number of values. Lastly a bar chart is plotted to visualise the data.

Dependent (Target) Attribute:

In [181]:

```
count = raw_data_copy['Loan_Status'].value_counts(normalize = True)
plt.figure(figsize=(7, 7))
chart = count.plot.bar(title = 'Loan_Status', xlabel = 'Categories', ylabel = 'Frequency
')
for i, v in enumerate(count):
    chart.text(i, v + 0.01, str(round(v, 2)), ha='center', va='bottom')
plt.show()
```



Insight Gained:

- **0.69 or 69% of the people were approved for a loan (i.e Loan_Status = Yes)**
- **0.31 or 31% of the people were not approved for a loan (i.e Loan_Status = No)**

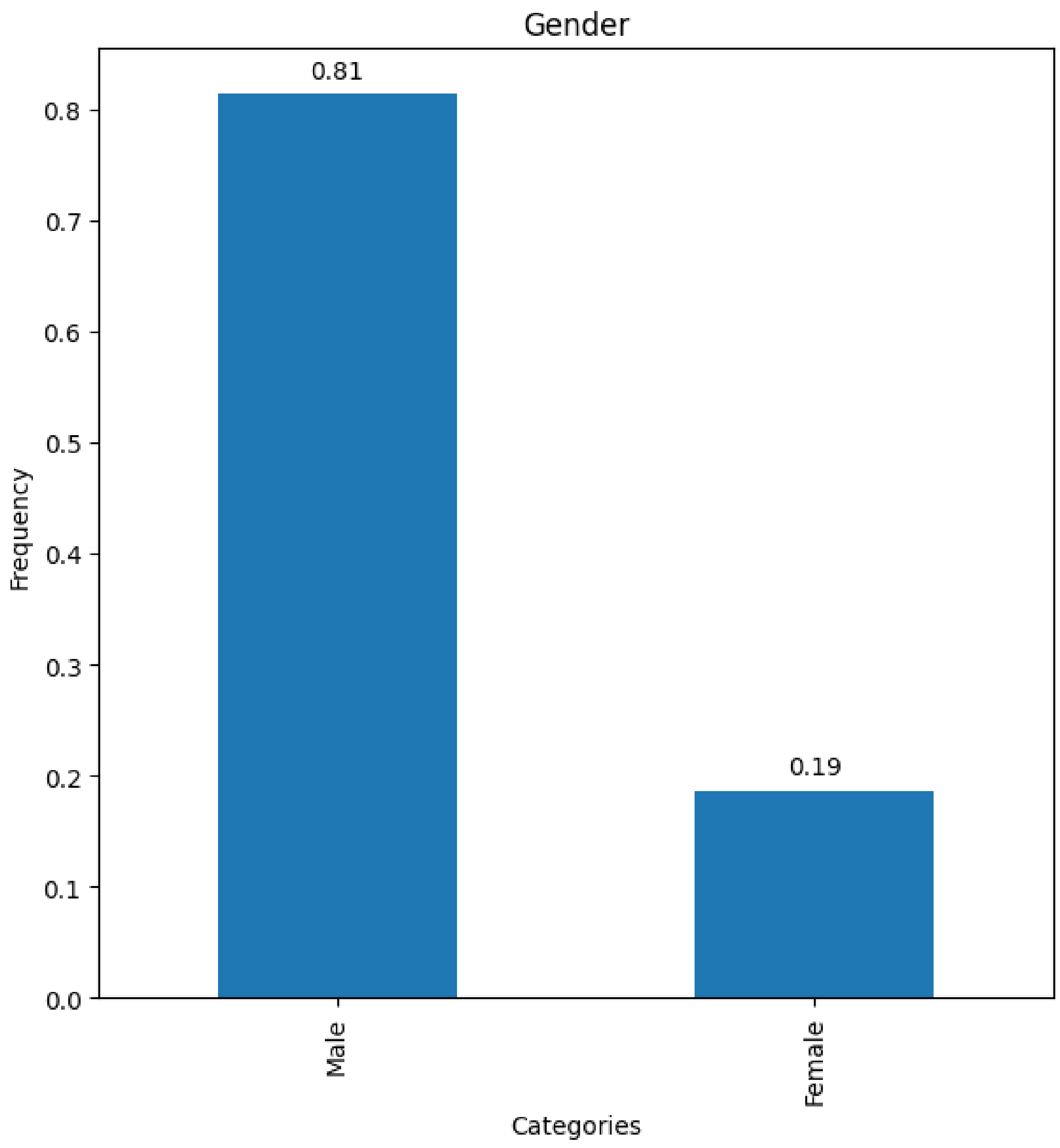
Independent Attributes (Categorical):

Categorical data is a type of data that is qualitative and has no numerical values. It can be divided into categories but cannot be ordered or measured. For examples,

- **Colour category can include: red, blue, or green**
- **Gender category can include: male or female**

In [182]:

```
# Gender Attribute
count = raw_data_copy['Gender'].value_counts(normalize = True)
plt.figure(figsize=(7, 7))
chart = count.plot.bar(title = 'Gender', xlabel = 'Categories', ylabel = 'Frequency')
for i, v in enumerate(count):
    chart.text(i, v + 0.01, str(round(v, 2)), ha='center', va='bottom')
plt.show()
```



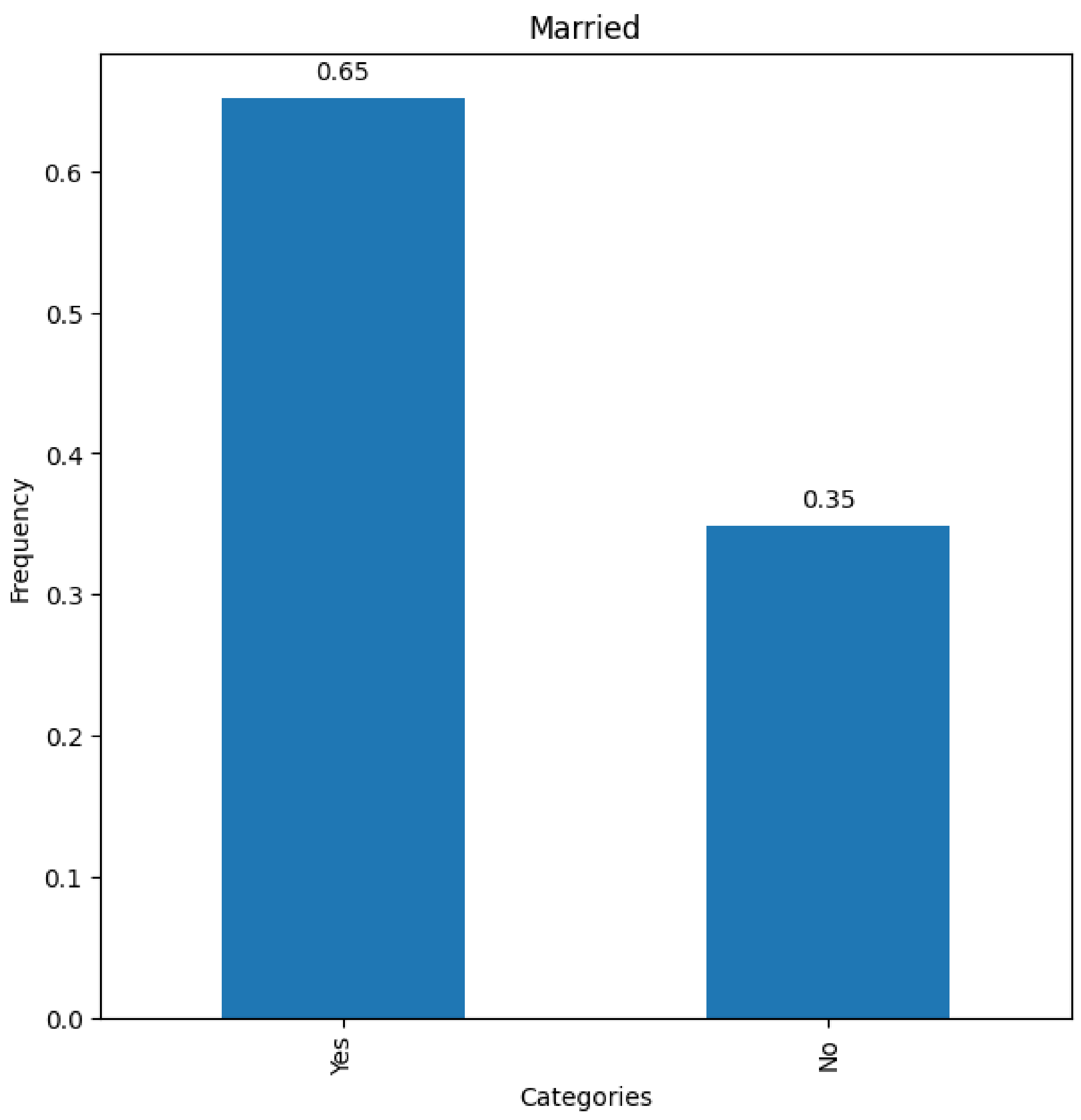
Insight Gained:

- **0.81 or 81% of the people are male (i.e Gender = Male)**
- **0.19 or 19% of the people are female (i.e Gender = Female)**

In [183]:

```
# Married Attribute
count = raw_data_copy['Married'].value_counts(normalize = True)
```

```
plt.figure(figsize=(7, 7))
chart = count.plot.bar(title='Married', xlabel = 'Categories', ylabel = 'Frequency')
for i, v in enumerate(count):
    chart.text(i, v + 0.01, str(round(v, 2)), ha='center', va='bottom')
plt.show()
```

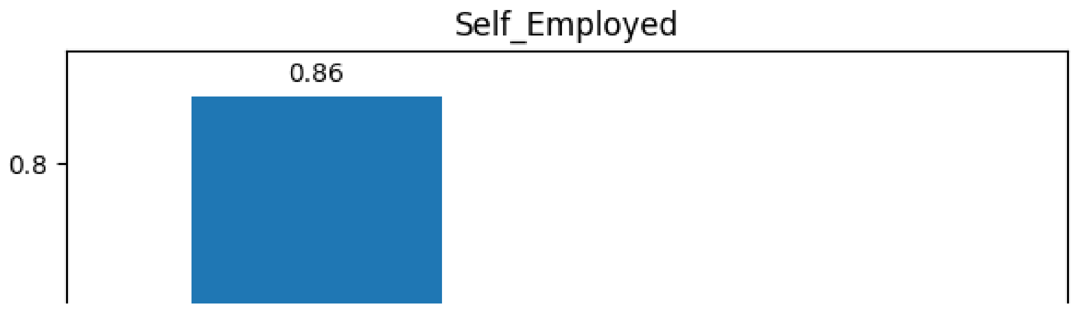


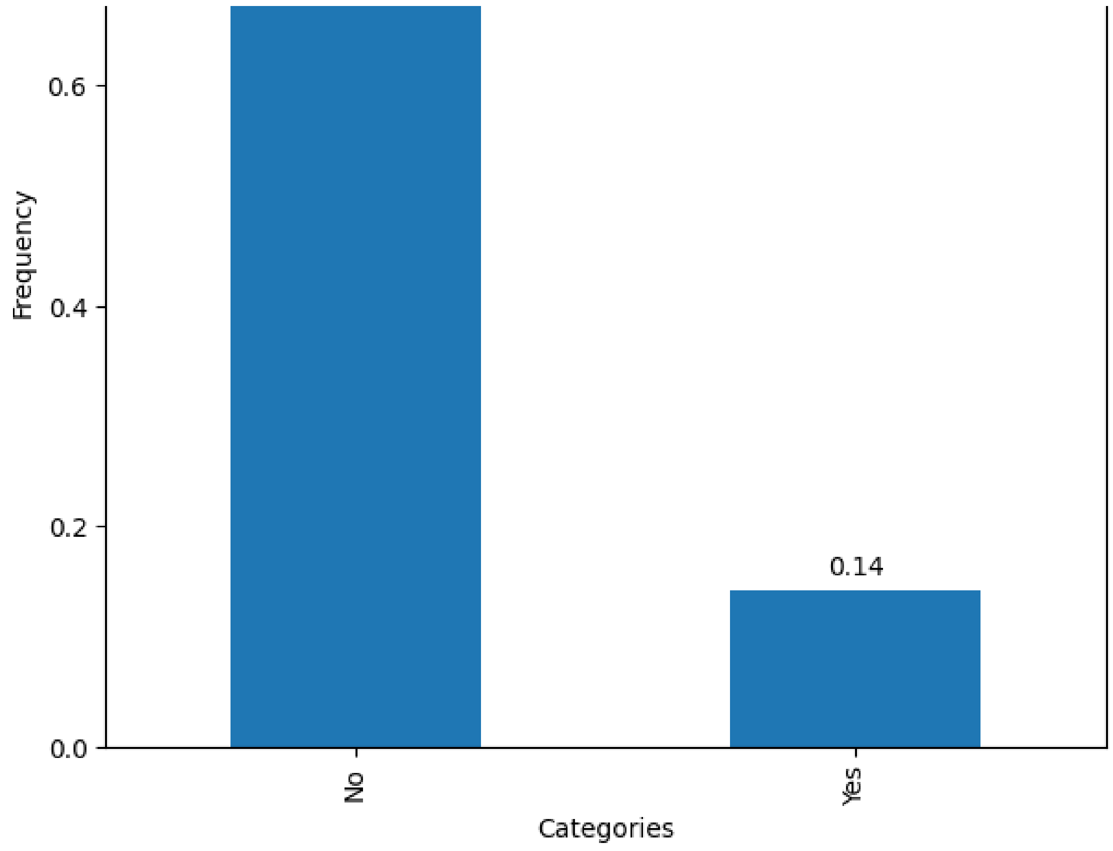
Insight Gained:

- **0.65 or 65% of the people were Married (i.e Married = Yes)**
- **0.35 or 35% of the people were not Married (i.e Married = No)**

In [184]:

```
# Self_Employed Attribute
count = raw_data_copy['Self_Employed'].value_counts(normalize = True)
plt.figure(figsize=(7, 7))
chart = count.plot.bar(title='Self_Employed', xlabel = 'Categories', ylabel = 'Frequency
')
for i, v in enumerate(count):
    chart.text(i, v + 0.01, str(round(v, 2)), ha='center', va='bottom')
plt.show()
```



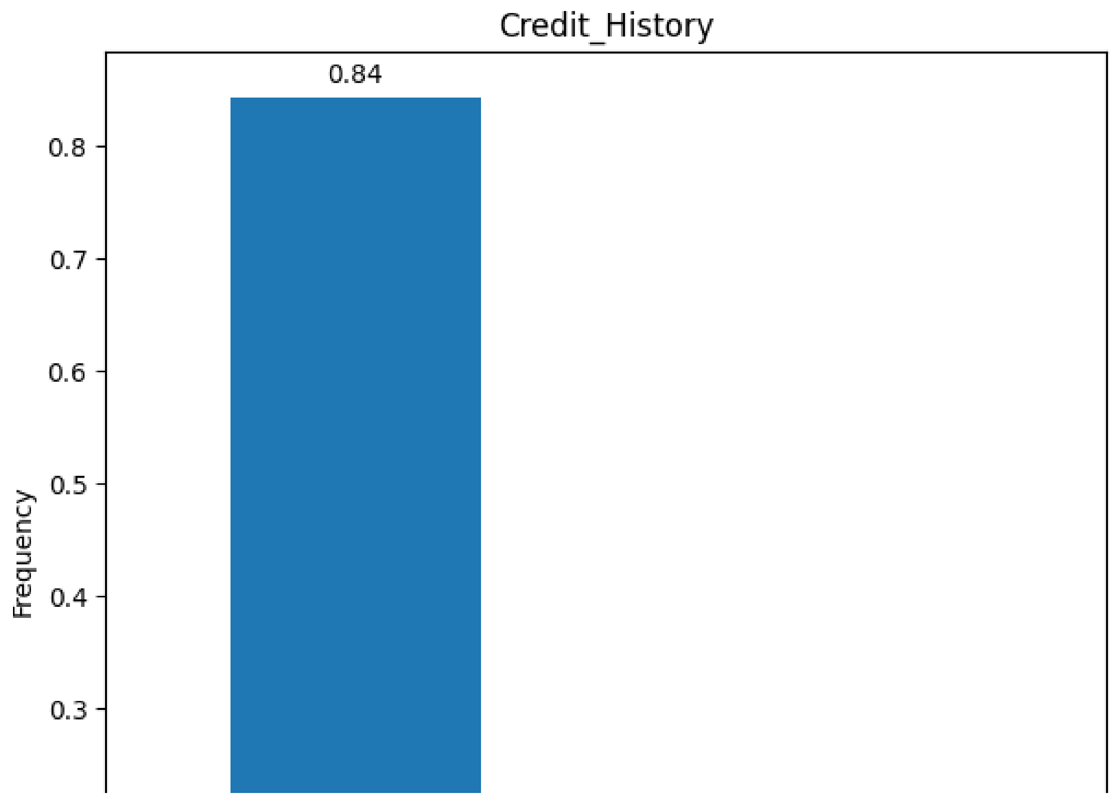


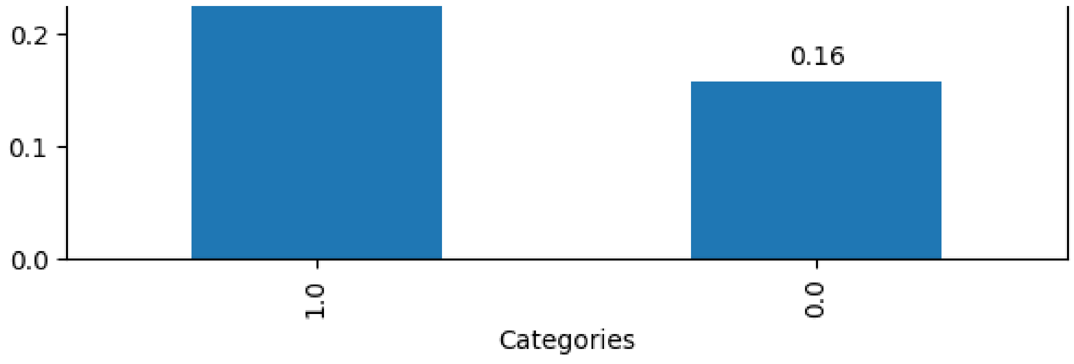
Insight Gained:

- **0.14 or 14% of the people are self-employed (i.e Self_Employed = Yes)**
- **0.86 or 86% of the people are not self-employed (i.e Married = No)**

In [185]:

```
# Credit_History Attribute
count = raw_data_copy['Credit_History'].value_counts(normalize = True)
plt.figure(figsize=(7, 7))
chart = count.plot.bar(title='Credit_History', xlabel = 'Categories', ylabel = 'Frequenc
y')
for i, v in enumerate(count):
    chart.text(i, v + 0.01, str(round(v, 2)), ha='center', va='bottom')
plt.show()
```





Insight Gained:

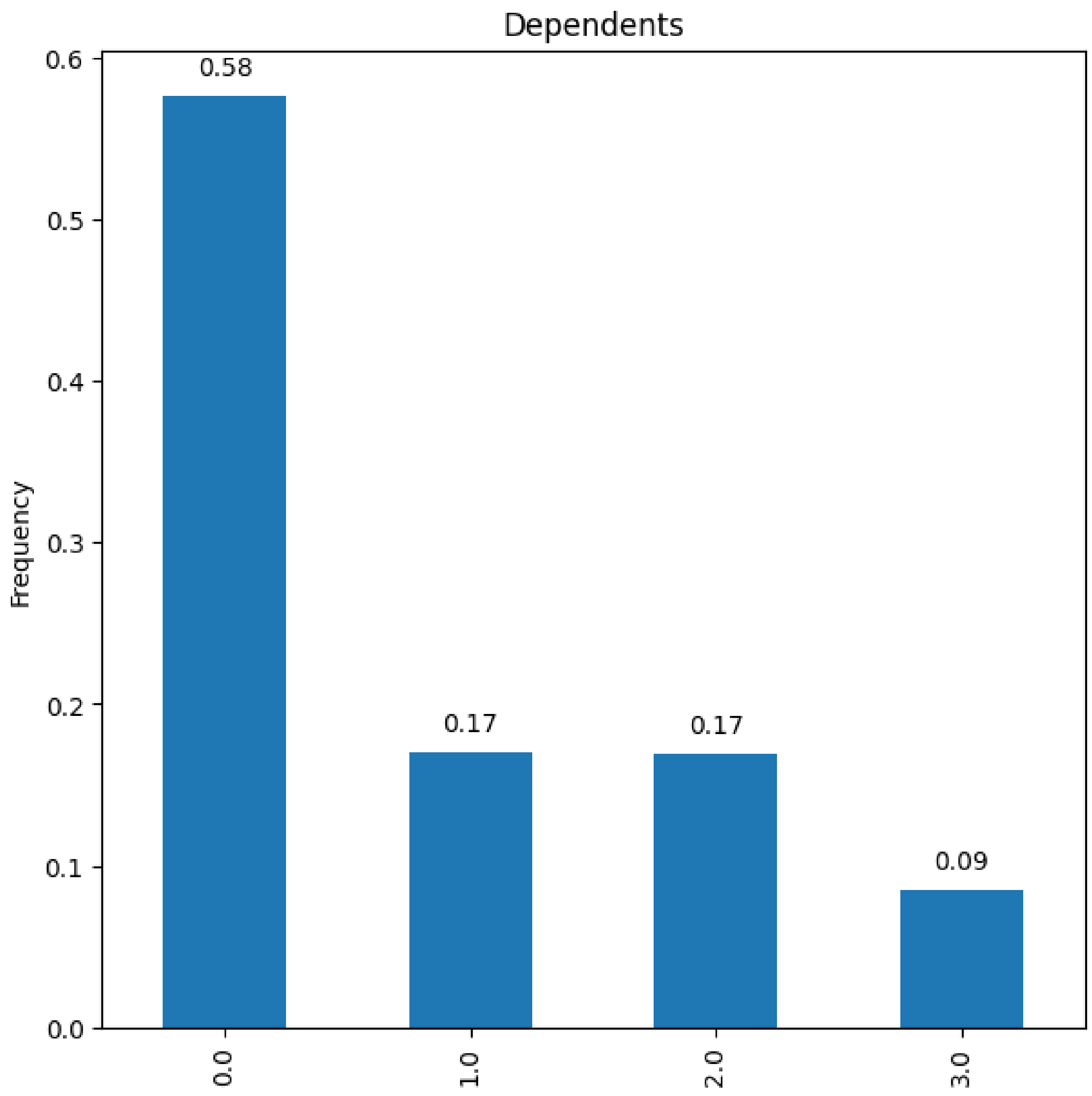
- **0.84 or 84% of the people have a credit history (i.e Credit_History = 1)**
- **0.16 or 16% of the people don't have a credit history (i.e Credit_History = 0)**

Independent Attributes (Ordinal):

Ordinal data have a clear ordering or hierarchy in the categories. For example, customer satisfaction ratings can include: unsatisfied, neutral, or satisfied.

In [186]:

```
# Dependents Attribute
count = raw_data_copy['Dependents'].value_counts('normalize = True')
plt.figure(figsize=(7, 7))
chart = count.plot.bar(title='Dependents', xlabel = 'Categories', ylabel = 'Frequency')
for i, v in enumerate(count):
    chart.text(i, v + 0.01, str(round(v, 2)), ha='center', va='bottom')
plt.show()
```



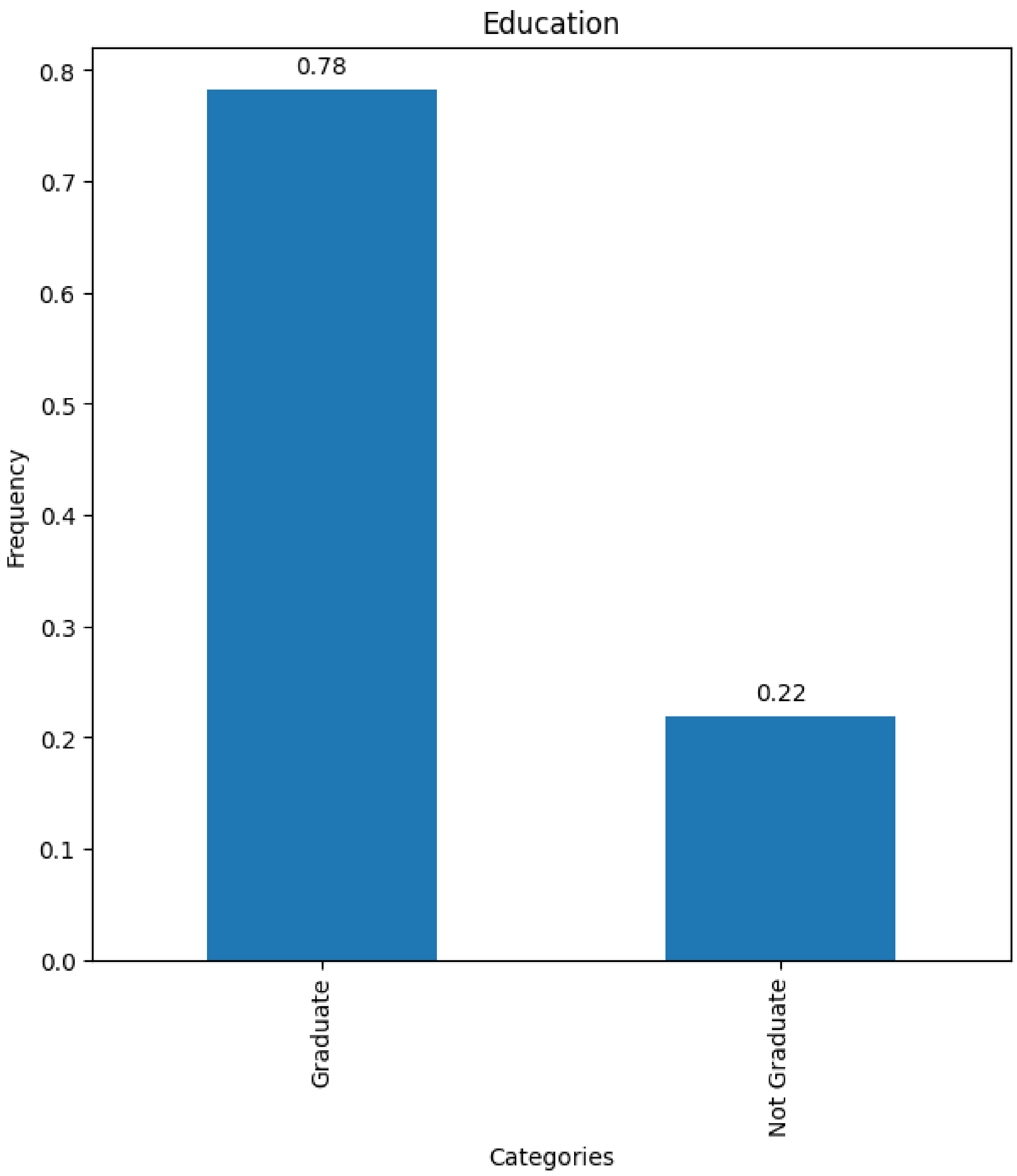
Categories

Insight Gained:

- **0.58 or 58% of the people don't have Dependent (i.e Dependents = 0)**
- **0.17 or 17% of the people has only one Dependent (i.e Dependents = 1)**
- **0.17 or 17% of the people has two Dependents (i.e Dependents = 2)**
- **0.09 or 9% of the people has three or more Dependents (i.e Dependents = 3+)**

In [187]:

```
# Education Attribute
count =raw_data_copy['Education'].value_counts('normalize = True')
plt.figure(figsize=(7, 7))
chart = count.plot.bar(title='Education', xlabel = 'Categories', ylabel = 'Frequency')
for i, v in enumerate(count):
    chart.text(i, v + 0.01, str(round(v, 2)), ha='center', va='bottom')
plt.show()
```

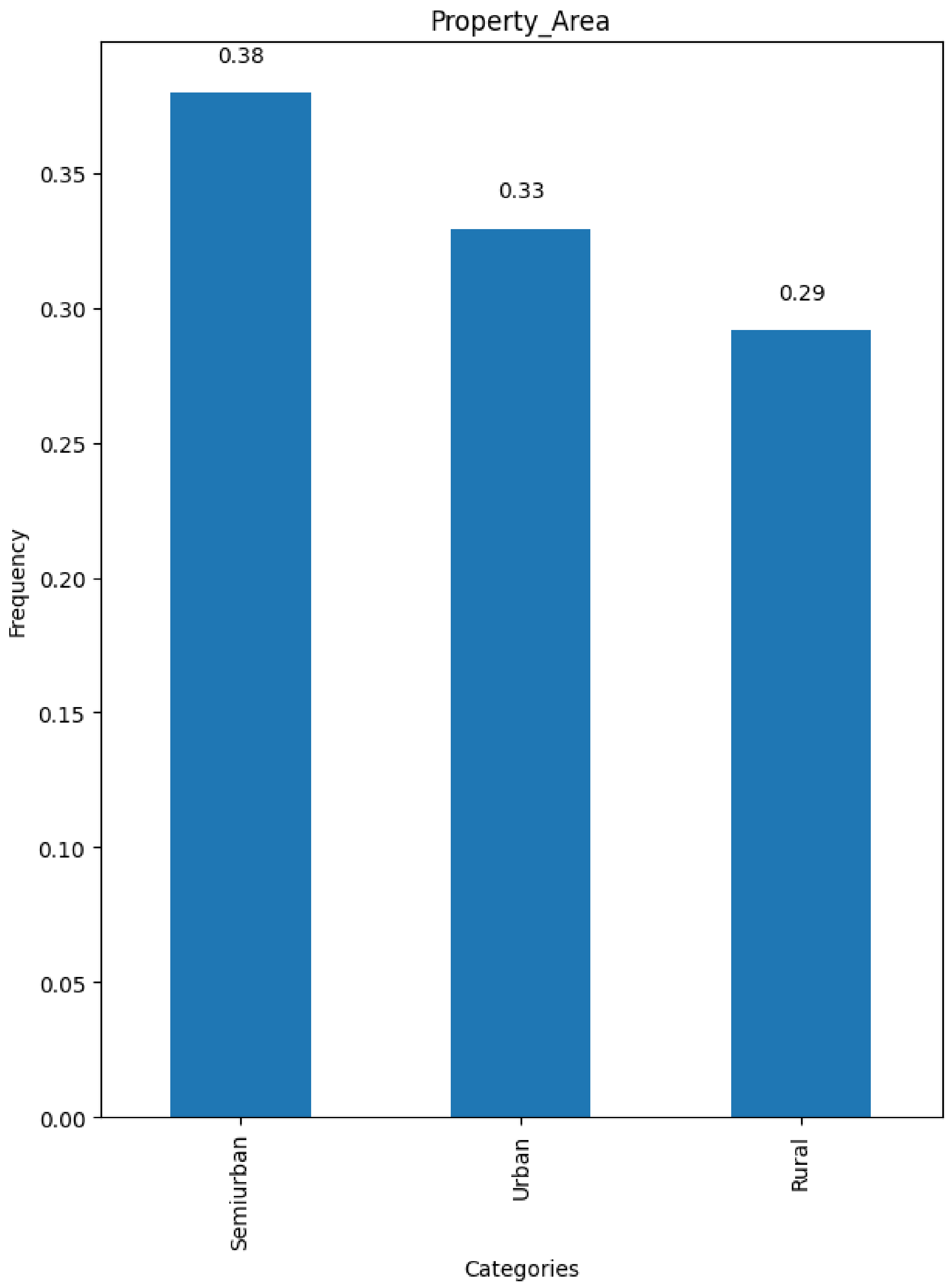


Insight Gained:

- **0.78 or 78% of the people have graduated (i.e Education = Graduate)**
- **0.22 or 22% of the people have not graduated (i.e Education = Not Graduate)**

In [188]:

```
# Property_Area Attribute
count = raw_data_copy['Property_Area'].value_counts('normalize=True')
plt.figure(figsize=(7, 9))
chart = count.plot.bar(title='Property_Area', xlabel = 'Categories', ylabel = 'Frequency
')
for i, v in enumerate(count):
    chart.text(i, v + 0.01, str(round(v, 2)), ha='center', va='bottom')
plt.show()
```



Insight Gained:

- **0.38 or 38% of the people are located in the semi-urban area (i.e Property_Area = Semiurban)**
- **0.33 or 33% of the people are located in the urban area (i.e Property_Area = Urban)**
- **0.29 or 29% of the people are located in the rural area(i.e Property_Area = Rural)**

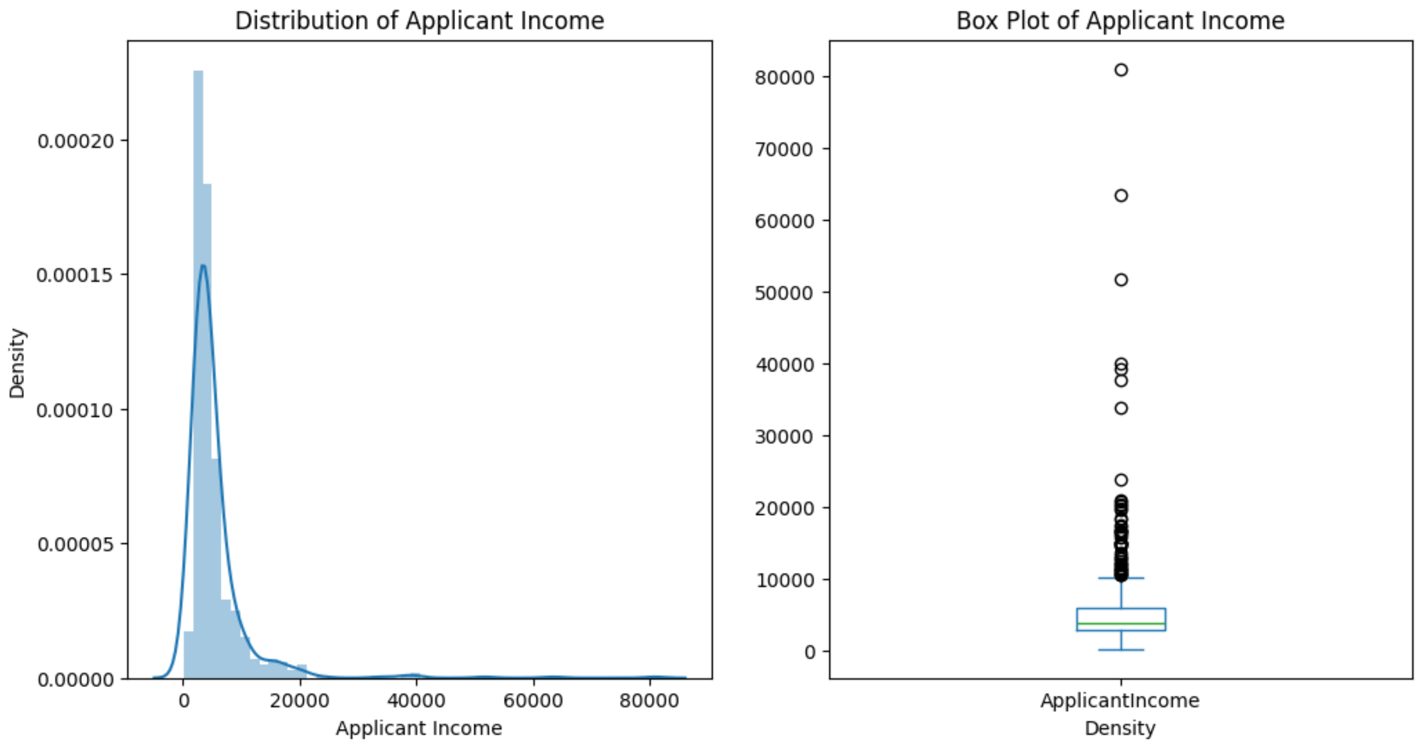
Independent Attributes (Nominal)

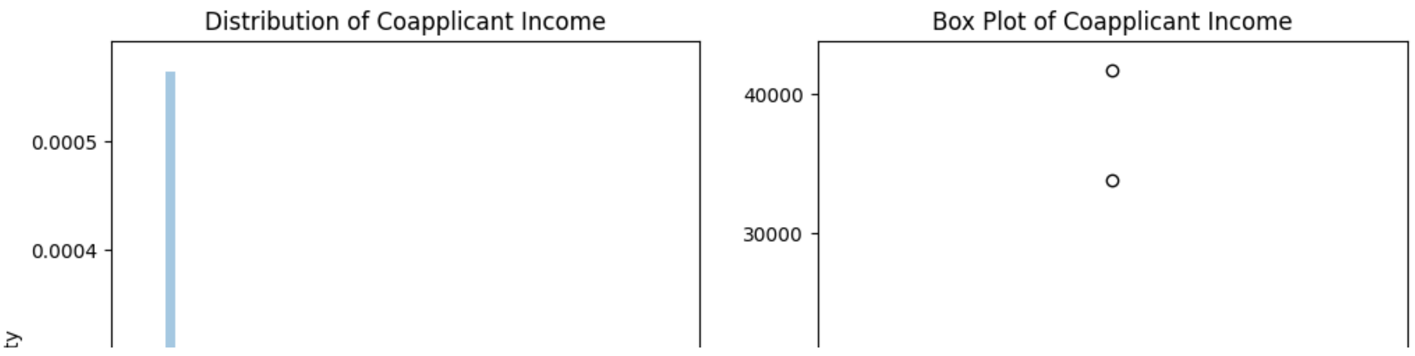
Nominal data does not have any kind of order or hierarchy but rather each category are different from each other. For example, the different breeds of dogs (Labrador, Beagle, Poodle) constitute nominal data because there is no inherent order among them.

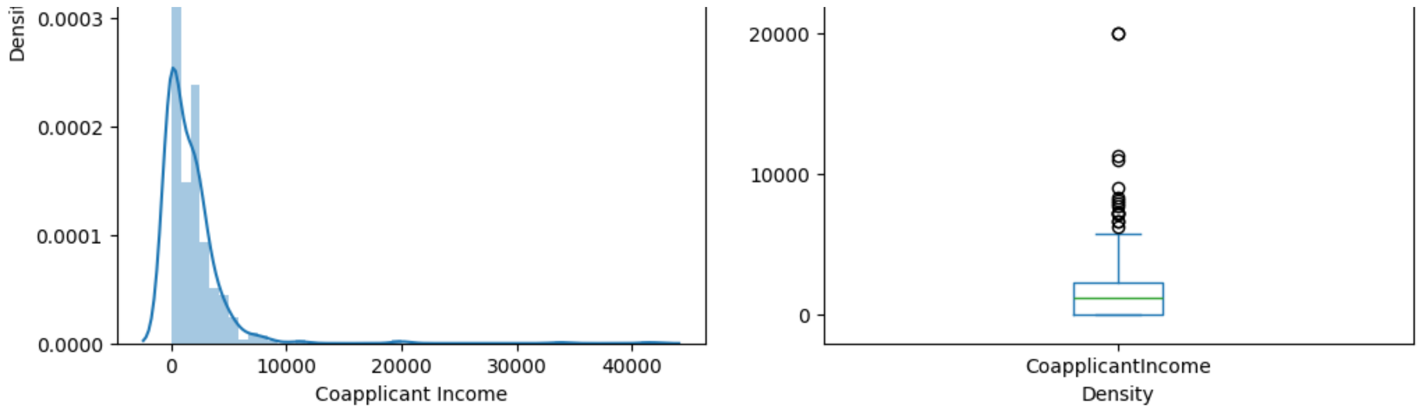
A distribution chart is used to visualise the distribution of the values in the attributes ApplicantIncome, CoapplicantIncome, LoanAmount, Loan_Amount_Term.

In [189]:

```
plt.figure(1, figsize=(12, 6))
plt.subplot(121)
raw_data_copy.dropna()   # Drop missing data in the attribute's data
sns.distplot(raw_data_copy['ApplicantIncome'])
plt.title('Distribution of Applicant Income')
plt.xlabel('Applicant Income')
plt.ylabel('Density')
plt.subplot(122)
boxplot = raw_data_copy['ApplicantIncome'].plot.box()
boxplot.set_title('Box Plot of Applicant Income')
boxplot.set_xlabel('Density')
plt.show()

plt.figure(2, figsize=(12, 6))
plt.subplot(121)
raw_data_copy.dropna()
sns.distplot(raw_data_copy['CoapplicantIncome'])
plt.title('Distribution of Coapplicant Income')
plt.xlabel('Coapplicant Income')
plt.ylabel('Density')
plt.subplot(122)
boxplot = raw_data_copy['CoapplicantIncome'].plot.box()
boxplot.set_title('Box Plot of Coapplicant Income')
boxplot.set_xlabel('Density')
plt.show()
```





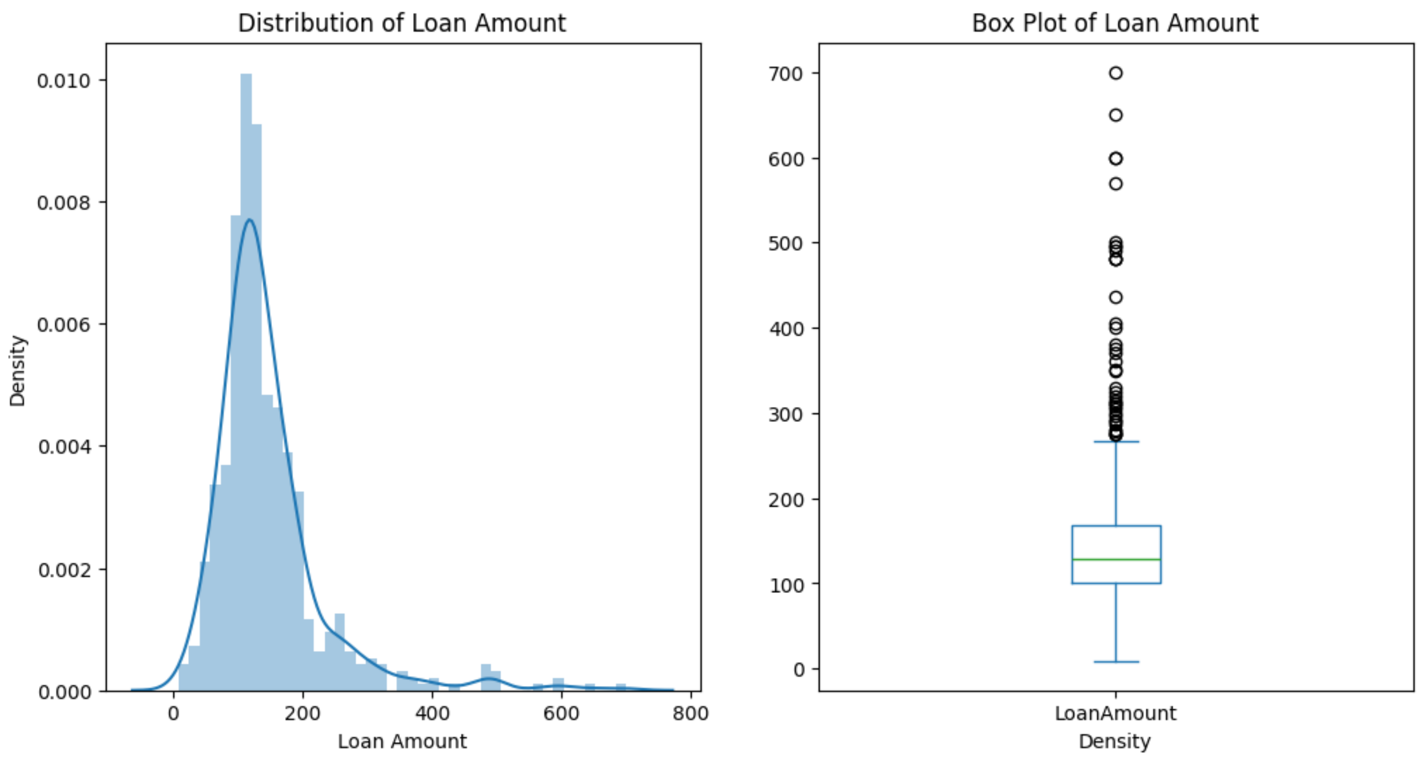


Insight Gained:

- **Both the distribution charts of the ApplicantIncome and CoapplicantIncome show a left-skewed distribution that indicates a majority of the applicants have lower incomes.**
- **This pattern reflects income inequality within the applicant pool.**

In [190]:

```
plt.figure(3, figsize=(12, 6))
plt.subplot(121)
raw_data_copy.dropna()
sns.distplot(raw_data_copy['LoanAmount'])
plt.title('Distribution of Loan Amount')
plt.xlabel('Loan Amount')
plt.ylabel('Density')
plt.subplot(122)
boxplot =raw_data_copy['LoanAmount'].plot.box()
boxplot.set_title('Box Plot of Loan Amount')
boxplot.set_xlabel('Density')
plt.show()
```

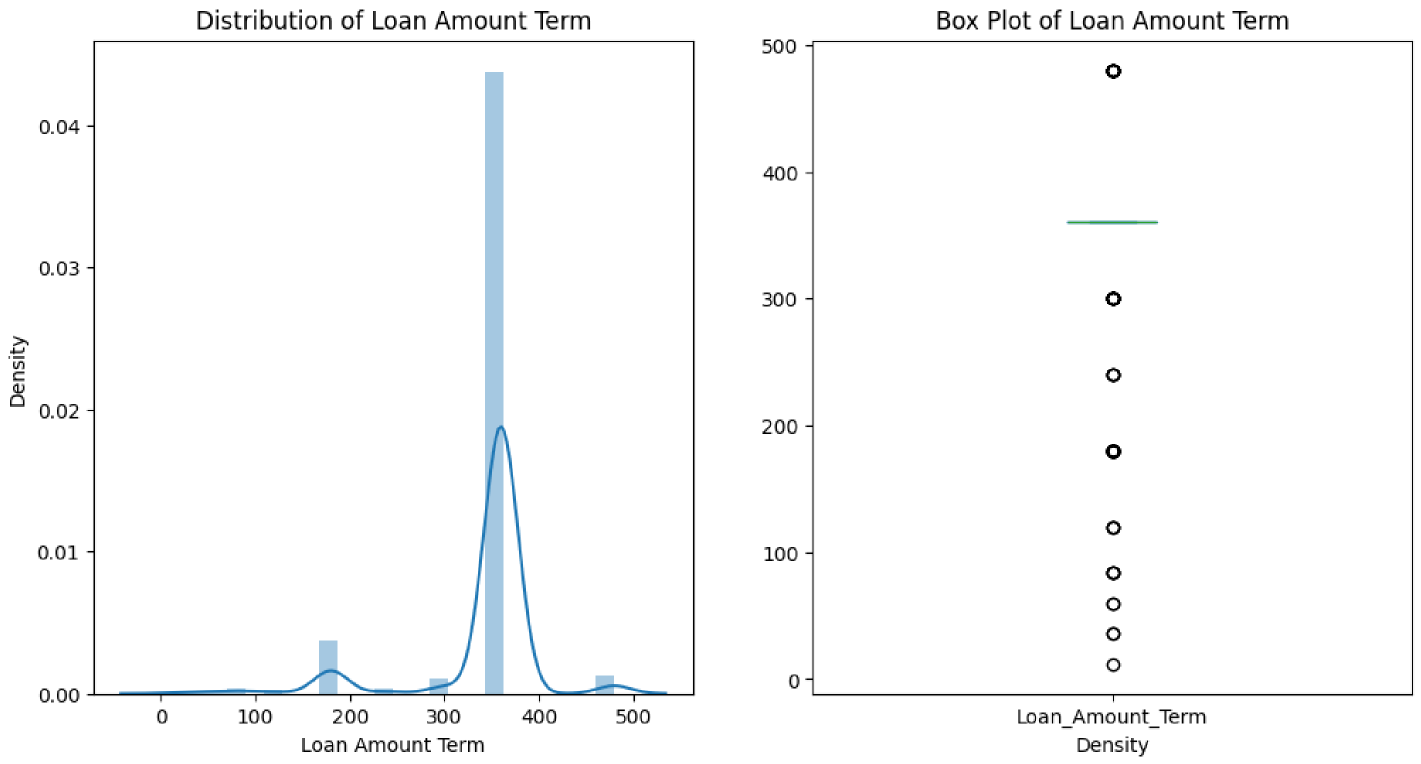


Insight Gained:

- **Overall the distribution of the data is fairly normal.**
- **There are outliers in this attribute which could negatively impact the mean and distribution of the data**
- **These outliers will be treated in the data cleaning process**

In [191]:

```
plt.figure(4, figsize=(12, 6))
plt.subplot(121)
raw_data_copy.dropna()
sns.distplot(raw_data_copy['Loan_Amount_Term'])
plt.title('Distribution of Loan Amount Term')
plt.xlabel('Loan Amount Term')
plt.ylabel('Density')
plt.subplot(122)
boxplot =raw_data_copy['Loan_Amount_Term'].plot.box()
boxplot.set_title('Box Plot of Loan Amount Term')
boxplot.set_xlabel('Density')
plt.show()
```



Insight Gained:

- **The peak around 360 indicates a standard loan term.**
- **Smaller peaks at lower values show that shorter loan terms are less common.**

C. Bi-variate Analysis

When there are two variables in the data it is called bi-variate analysis. The data is analyzed to find the relationship between the dependent and independent variables. Stacked bar graphs can be utilised to view the correlation between the coefficients.

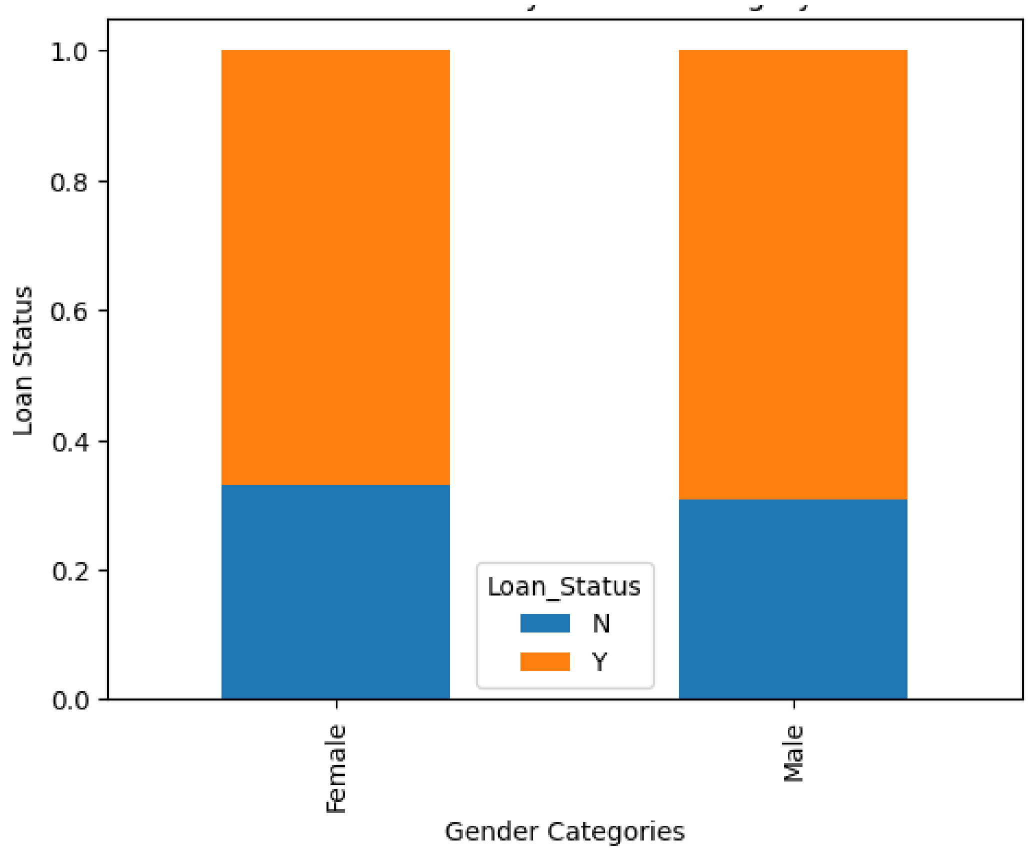
The graphs created below will display how the Dependent Attribute ‘Loan_Status’ is distributed within each Independent Attribute, regardless of how many observations there are.

Categorical Independent Variables and Dependent Variable LoanAmount:

In [192]:

```
# Loan_Status vs Gender
gender_table = pd.crosstab(raw_data_copy['Gender'], raw_data_copy['Loan_Status'])
gender_table.div(gender_table.sum(1).astype(float),axis=0).plot(kind='bar', stacked=True
)
plt.title('Loan Status by Gender Category')
plt.xlabel('Gender Categories')
plt.ylabel('Loan Status')
plt.show()
```

Loan Status by Gender Category

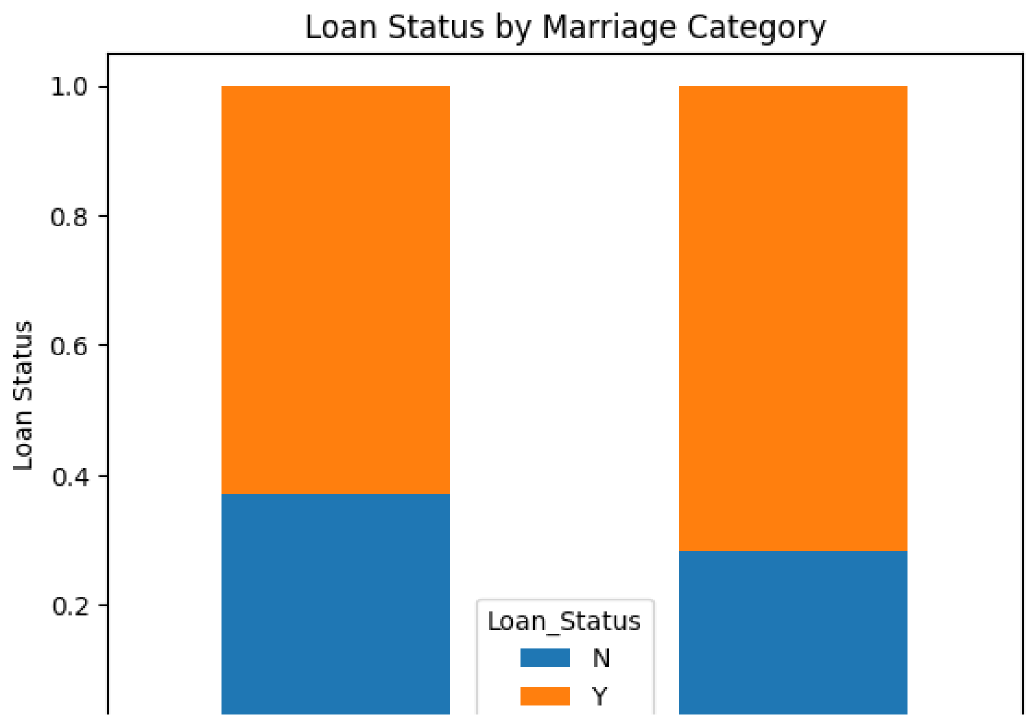


Insight Gained:

- **The proportion of the loan status 'Yes' is slightly higher for males, indicating a marginally higher approval rate compared to females.**
- **For both genders, the majority of the loan status is 'Yes', suggesting that most applicants in the dataset were approved for a loan.**

In [193]:

```
# Loan_Status vs Married
married_table = pd.crosstab(raw_data_copy['Married'], raw_data_copy['Loan_Status'])
married_table.div(married_table.sum(1).astype(float),axis=0).plot(kind='bar', stacked=Tru
e)
plt.title('Loan Status by Marriage Category')
plt.xlabel('Married Categories')
plt.ylabel('Loan Status')
plt.show()
```

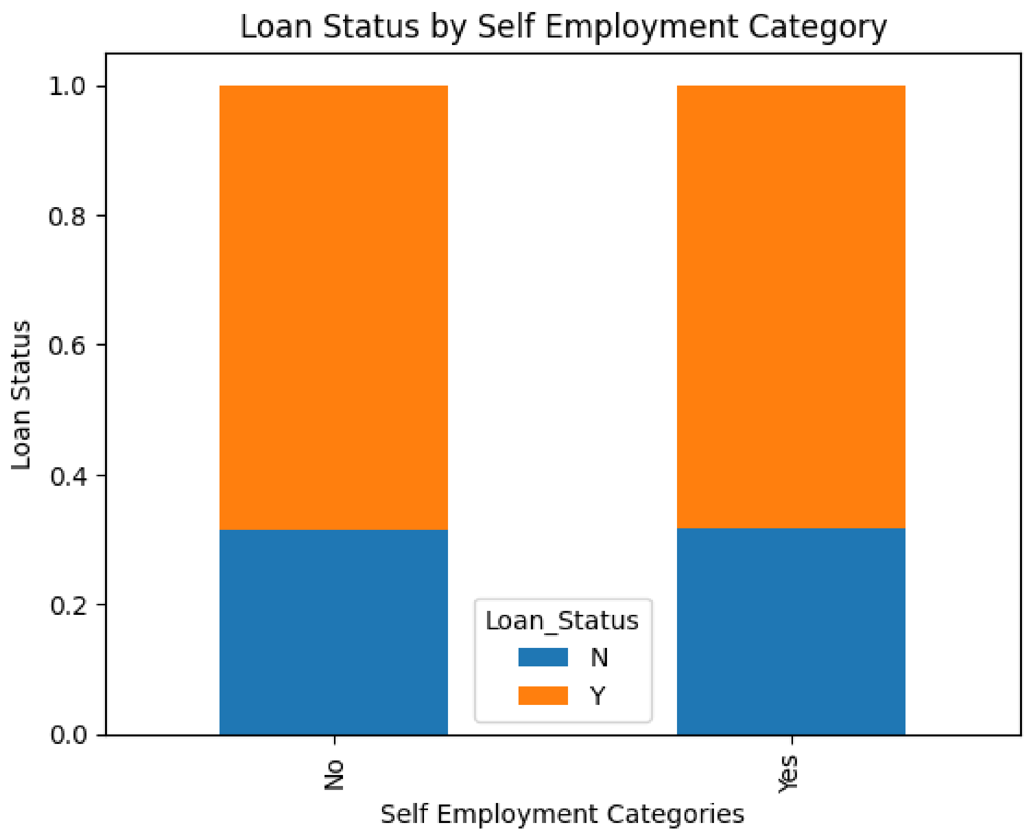


Insight Gained:

- **The ‘Yes’ category shows a higher proportion for the loan status ‘Yes’, suggesting that married individuals may have a better chance of loan approval.**
- **Conversely, the ‘No’ category has a higher proportion for the loan status ‘No’, indicating that unmarried individuals may face more rejections.**

In [194]:

```
# Loan_Status vs Self_Employed
self_employed_table = pd.crosstab(raw_data_copy['Self_Employed'], raw_data_copy['Loan_Sta
tus'])
self_employed_table.div(self_employed_table.sum(1).astype(float),axis=0).plot(kind='bar',
stacked=True)
plt.title('Loan Status by Self Employment Category')
plt.xlabel('Self Employment Categories')
plt.ylabel('Loan Status')
plt.show()
```



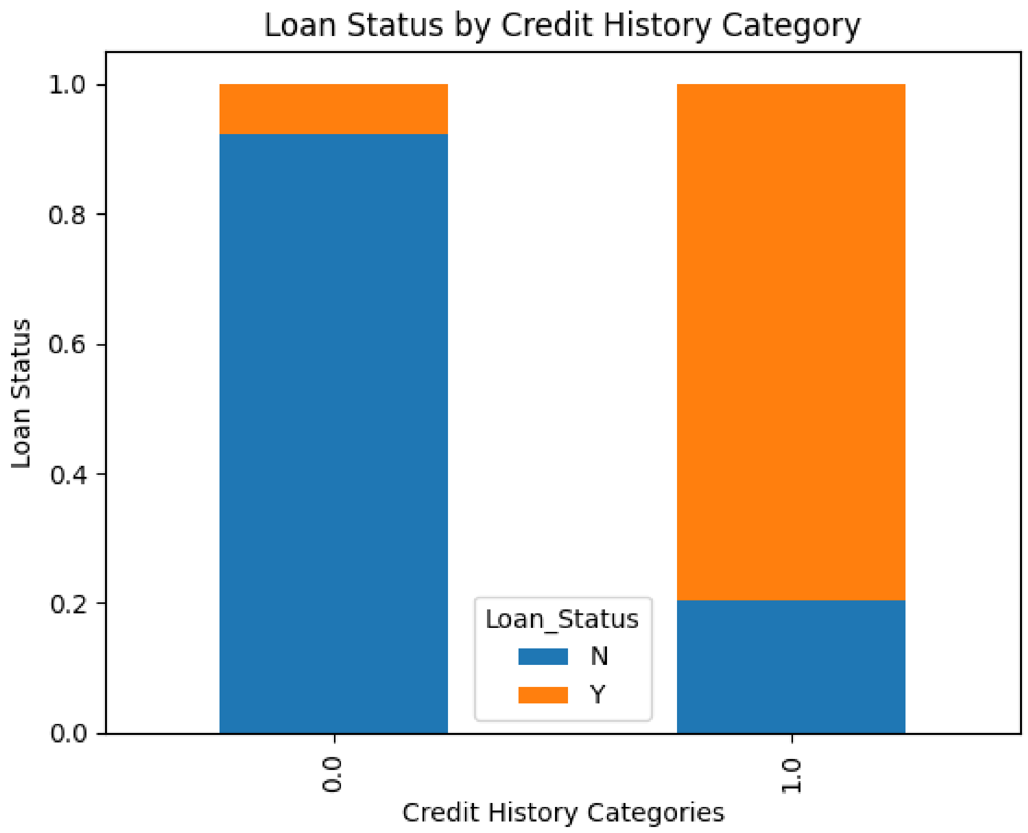
Insight Gained:

- **The ‘Yes’ loan status is present in both self-employment categories, but there is a slightly larger proportion of approvals for individuals who are not self-employed (‘No’) compared to those who are self-employed (‘Yes’).**
- **Self-Employment Impact: The graph suggests that being self-employed might have a slight impact on loan approval rates, although the difference is not substantial.**

In [195]:

```
# Loan_Status vs Credit_History
credit_history_table = pd.crosstab(raw_data_copy['Credit_History'], raw_data_copy['Loan_S
tatus'])
credit_history_table.div(credit_history_table.sum(1).astype(float),axis=0).plot(kind='bar
', stacked=True)
```

```
plt.title('Loan Status by Credit History Category')
plt.xlabel('Credit History Categories')
plt.ylabel('Loan Status')
plt.show()
```



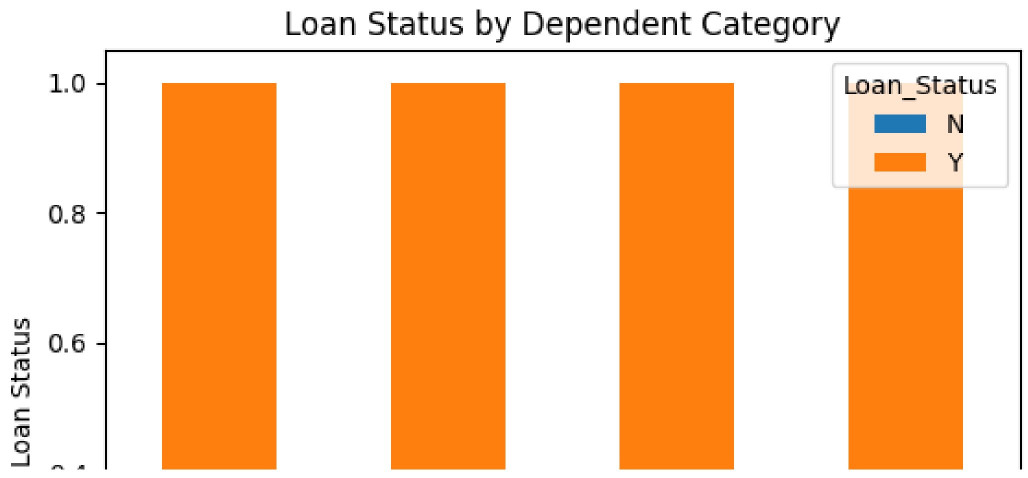
Insight Gained:

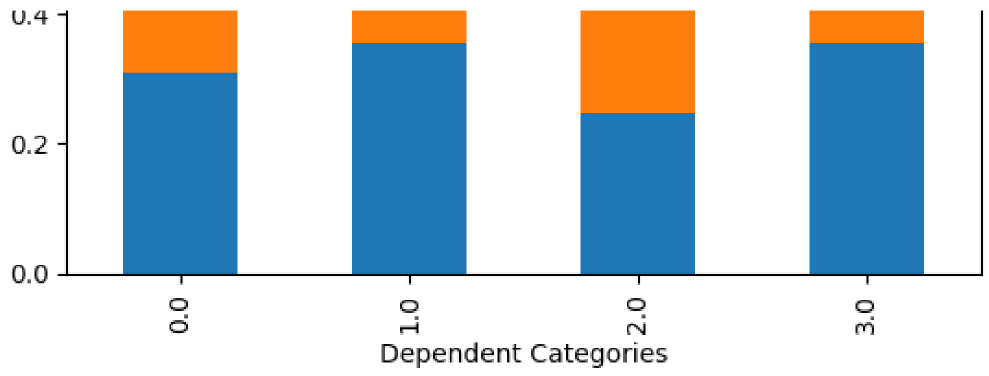
- **Individuals in Credit History Category ‘1’ have a higher proportion of getting approval for a loan, indicating a positive correlation between a good credit history and loan approval.**
- **Category ‘0’ has a higher proportion of being rejected for a loan approval, suggesting that a poor credit history is associated with higher loan rejections.**

Ordinal Independent Variables and Dependent Variable LoanAmount

In [196]:

```
# Loan_Status vs Dependents
dependents_table = pd.crosstab(raw_data_copy['Dependents'], raw_data_copy['Loan_Status'])
dependents_table.div(dependents_table.sum(1).astype(float),axis=0).plot(kind='bar', stac
ked=True)
plt.title('Loan Status by Dependent Category')
plt.xlabel('Dependent Categories')
plt.ylabel('Loan Status')
plt.show()
```



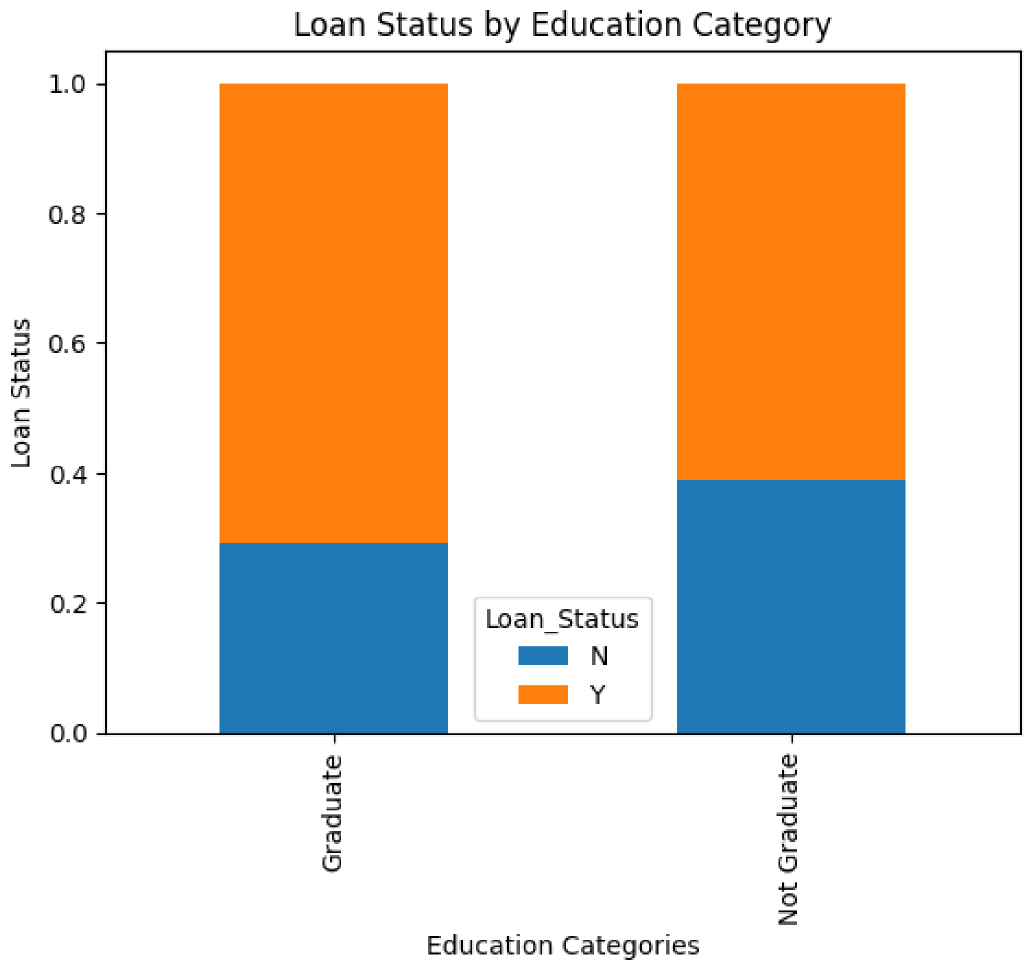


Insight Gained:

- **The ‘Yes’ loan status is present across all dependent categories, but there is a trend where the proportion of approvals decreases as the number of dependents increases.**
- **The graph suggests that having more dependents might negatively impact the rate of loan approval.**

In [197]:

```
# Loan_Status vs Education
education_table = pd.crosstab(raw_data_copy['Education'], raw_data_copy['Loan_Status'])
education_table.div(education_table.sum(1).astype(float),axis=0).plot(kind='bar', stacke
d=True)
plt.title('Loan Status by Education Category')
plt.xlabel('Education Categories')
plt.ylabel('Loan Status')
plt.show()
```

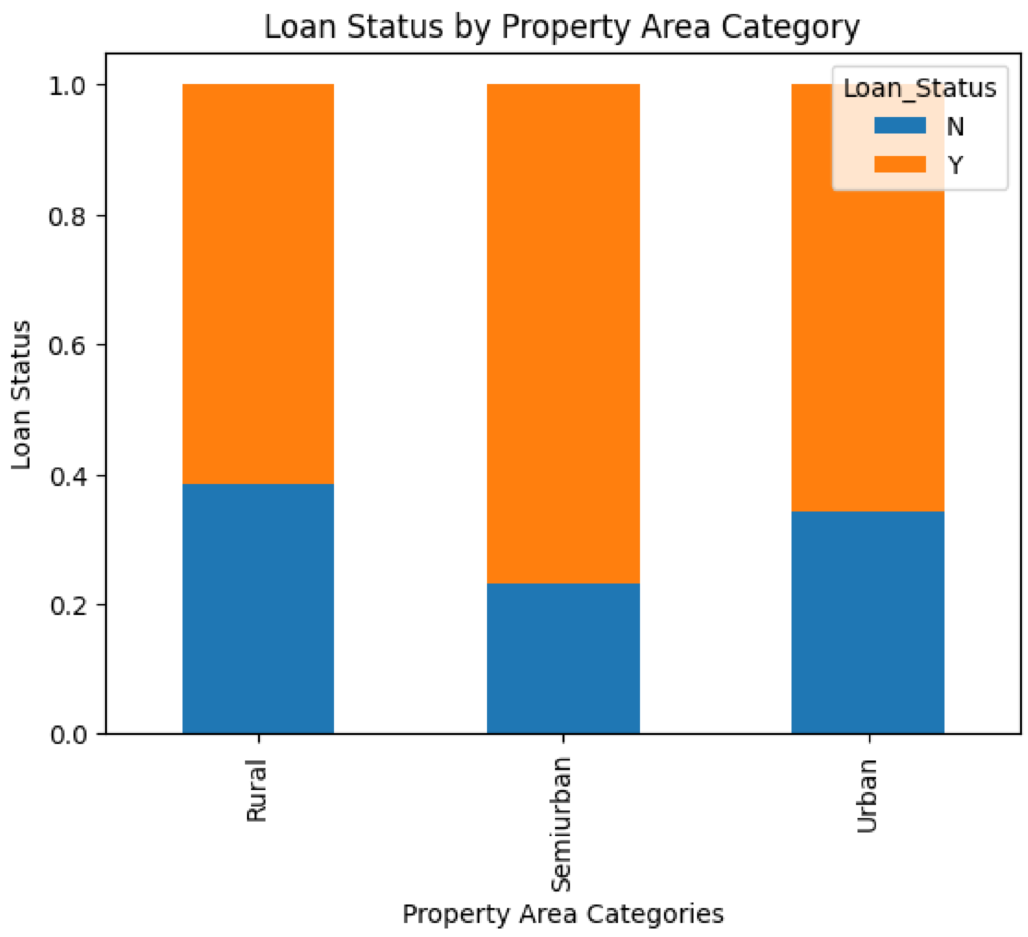


Insight Gained:

- **A larger proportion of graduates have their loans approved (‘Y’) compared to non-graduates who have a higher proportion of being rejected (‘N’).**

In [198]:

```
# Loan_Status vs Property Area
property_area_table = pd.crosstab(raw_data_copy['Property_Area'], raw_data_copy['Loan_Sta
tus'])
property_area_table.div(property_area_table.sum(1).astype(float),axis=0).plot(kind='bar',
stacked=True)
plt.title('Loan Status by Property Area Category')
plt.xlabel('Property Area Categories')
plt.ylabel('Loan Status')
plt.show()
```



Insight Gained:

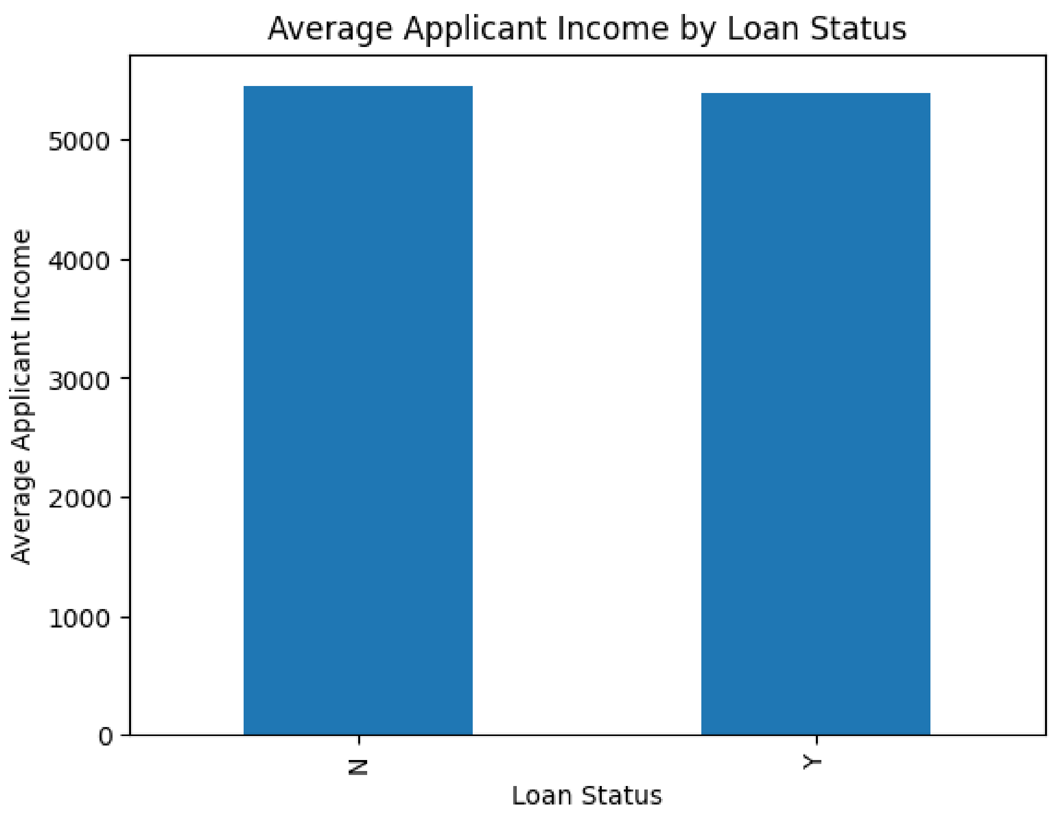
- **The Semiurban areas have the highest proportion of approved loans ('Y'), suggesting a favorable outcome for loan applicants in these areas.**
- **The Rural area has the lowest proportion of approved loans, indicating potential challenges or stricter criteria for loan approval.**
- **Urban Observations: The Urban area has a moderate proportion of approved loans, falling between the Rural and Semiurban areas.**

Numerical Independent Variables and Dependent Variable LoanAmount

The purpose of this section is to provide insight into how the income levels (both individually and combined with co-applicants) relate to the likelihood of a loan being approved. In order to determine the impact of the income on the Loan_Status, the mean income is calculated to determine who's loans were approved vs who's were not.

In [199]:

```
# Loan_Status vs Applicant_Income
raw_data_copy.groupby('Loan_Status')['ApplicantIncome'].mean().plot.bar()
plt.title('Average Applicant Income by Loan Status')
plt.xlabel('Loan Status')
plt.ylabel('Average Applicant Income')
plt.show()
```

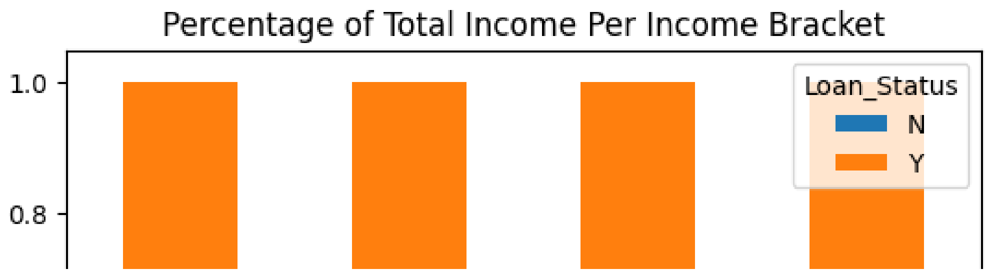


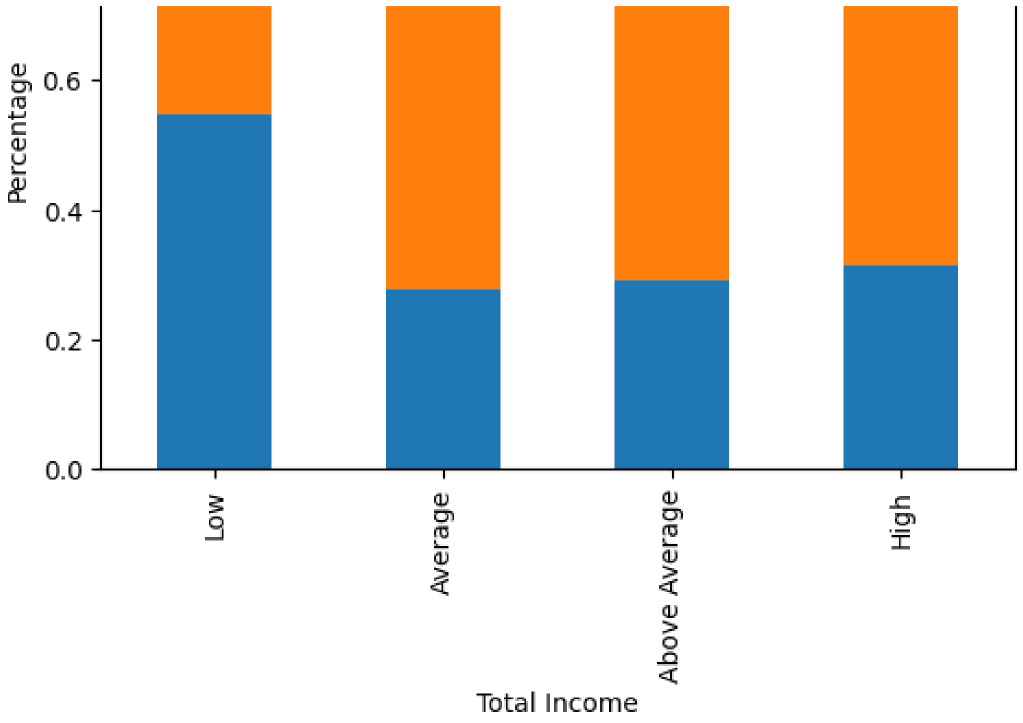
ApplicantIncome is categorized in loans within each income bracket. This will access whether different income levels, when the applicant income and the co-applicant income are added together, will influence the Loan approval rate.

Binning will transform the continuous numerical variables into discrete categorical ‘bins’. Income brackets such as "Low", "Average", "Above Average", and "High" are used to provide a qualitative understanding of the ranges in the data.

In [200]:

```
# Loan_Status vs Total Income
# Combine the applicant income and co-applicant income together
raw_data_copy['Total_Income']=raw_data_copy['ApplicantIncome']+raw_data_copy['Coapplicant
Income']

# Calculate the bin values
low = raw_data_copy['ApplicantIncome'].quantile(0.25)  # 25th percentile
average = raw_data_copy['ApplicantIncome'].quantile(0.50)  # 50th percentile
above_average = raw_data_copy['ApplicantIncome'].quantile(0.75)  # 75th percentile
high = 81000

bins = [0, low, average, above_average, high]
group=['Low','Average','Above Average','High']

raw_data_copy['Total_Income_bin']=pd.cut(raw_data_copy['Total_Income'],bins,labels=group)
Total_Income_bin=pd.crosstab(raw_data_copy['Total_Income_bin'],raw_data_copy['Loan_Status
'])
Total_Income_bin.div(Total_Income_bin.sum(1).astype(float),axis=0).plot(kind='bar',stack
ed=True)
plt.title('Percentage of Total Income Per Income Bracket')
plt.xlabel('Total Income')
plt.ylabel('Percentage')
plt.show()
```





Insight Gained:

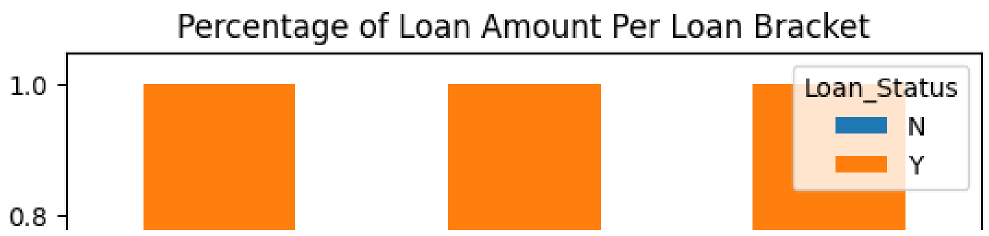
- **The proportion of loans approved for applicants with low total income is significantly lower than for other income groups.**
- **Applicants with average, high, and very high total income have a greater proportion of loan approvals.**
- **The chart suggests that total income level may impact the likelihood of loan approval.**
- **This analysis indicates that higher income levels are associated with better chances of loan approval, highlighting the importance of income in the loan decision process.**

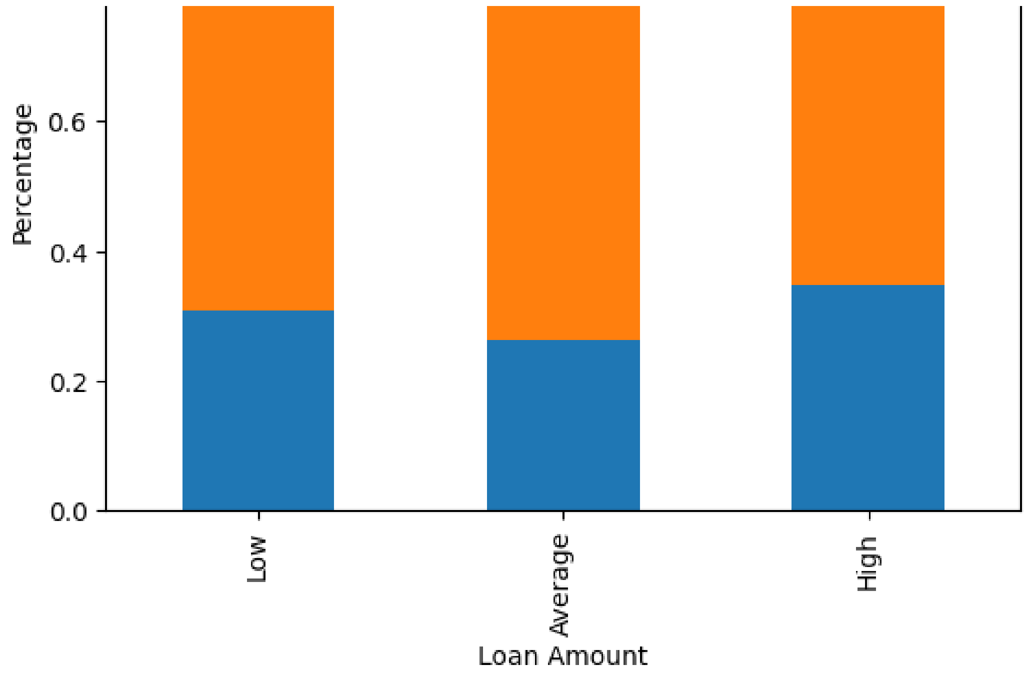
Loan amount is categorized in loans within each loan bracket. This will access whether different loan amounts will influence the Loan approval rate.

Loan brackets such as "Low", "Average", and "High" are used to provide a qualitative understanding of the ranges in the data.

In [201]:

```
# Loan_Status vs Loan Amount
# Calculate the bin values
low = raw_data_copy['LoanAmount'].quantile(0.333) # 33.3th percentile
average = raw_data_copy['LoanAmount'].quantile(0.666) # 66.6th percentile
high = 700

bins = [0, low, average, high]
group=['Low','Average','High']

raw_data_copy['Loan_Amount_bin']=pd.cut(raw_data_copy['LoanAmount'],bins,labels=group)
Total_Income_bin=pd.crosstab(raw_data_copy['Loan_Amount_bin'],raw_data_copy['Loan_Status'
])
Total_Income_bin.div(Total_Income_bin.sum(1).astype(float),axis=0).plot(kind='bar',stack
ed=True)
plt.title('Percentage of Loan Amount Per Loan Bracket')
plt.xlabel('Loan Amount')
plt.ylabel('Percentage')
plt.show()
```





Insight Gained:

- **Low and Average Loan Amounts: The proportion of approved loans is higher for these categories, indicating a greater likelihood of approval for smaller loan amounts.**
- **High Loan Amount: The proportion of approved loans is lower for this category, suggesting that larger loan amounts may have a reduced chance of approval.**
- **Therefore it can be said that loans with lower amounts are more likely to be approved.**

We must now remove all bins created to avoid redundancy and reduce data complexity.

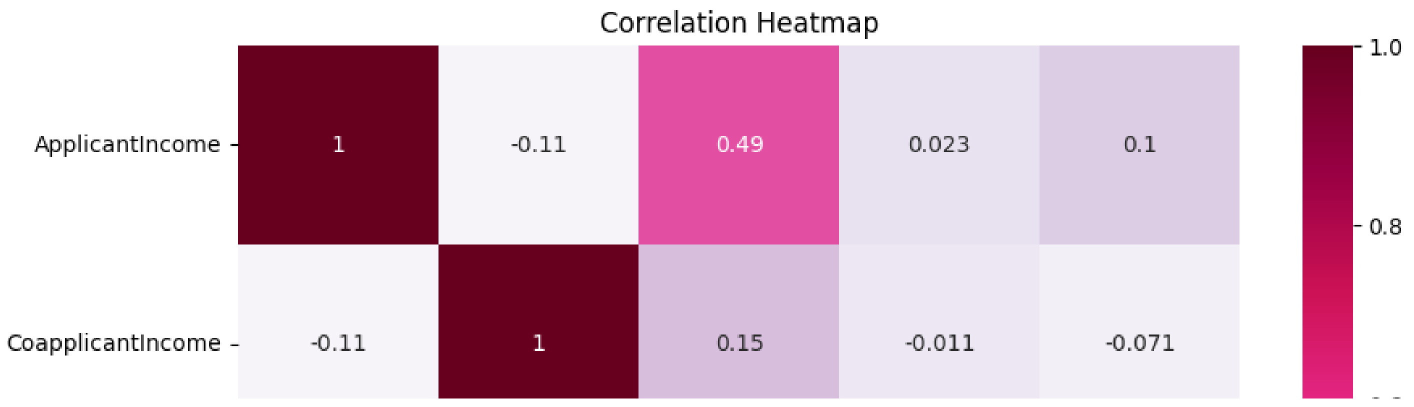
In [202]:

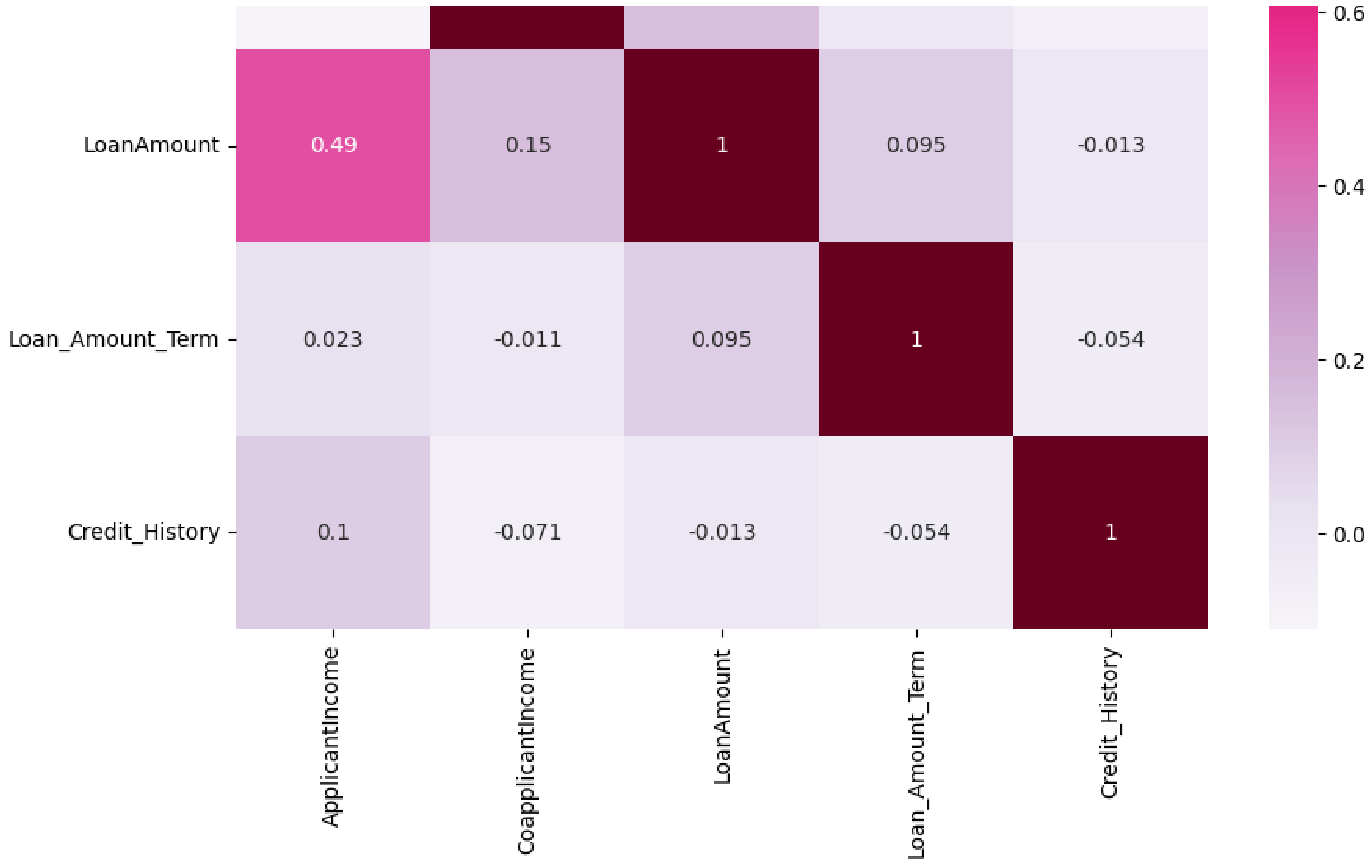
```
# Drop all bins created:
raw_data_copy=raw_data_copy.drop(['Loan_Amount_bin','Total_Income_bin','Total_Income'],ax
is=1)
```

Heatmaps are a powerful visualization tool that display data through variations in coloring. When used with datasets, the colors correspond to the values in each cell of a matrix. Darker colors typically represent higher correlation values (either positive or negative), while lighter colors represent lower correlation values or no correlation. This allows for an intuitive, visual interpretation of how different numerical attributes relate to each other in the dataset.

In [203]:

```
numeric_cols = validation_data_copy.select_dtypes(include=[np.number])

plt.figure(figsize=(10,8))
sns.heatmap(numeric_cols.corr(), annot=True, cmap='PuRd')
plt.title('Correlation Heatmap')
plt.show()
```





Insight Gained:

- **Moderate Correlation: ApplicantIncome and LoanAmount have a moderate positive correlation, suggesting that as applicant income increases, the loan amount tends to increase as well.**
- **Credit History Impact: The moderate positive correlation between Credit_History and Loan_Status indicates that applicants with a good credit history are more likely to have their loans approved.**
- **Coapplicant Contribution: While there is a positive correlation between LoanAmount and CoapplicantIncome, it is relatively weak, implying that co- applicant income has a lesser impact on the loan amount compared to the primary applicant's income.**
- **Overall, the heatmap suggests that both income and credit history play significant roles in loan amount determination and approval. The weaker correlation for CoapplicantIncome may indicate that lenders prioritize the primary applicant's financial status.**

Having now seen how the different attributes impact the outcome in our datasets, several hypotheses can be drawn from the results.

===

2. Hypotheses

The aim of this project is to use machine learning to transform BC Finance's loan approval process. BC Finance seeks to mitigate the inefficiencies linked to manual validation, including longer application periods, higher error rates, and lower customer satisfaction, by automating the real-time eligibility evaluation process. By using this automated approach, BC Finance hopes to improve resource allocation, boost operational efficiency, and ultimately become more competitive in the financial market.

The _'prepare*data.py'* file is essential in this situation as it organizes several data pretreatment and exploratory analysis activities. By carefully going over the dataset, which includes factors like gender, marital status, income levels, credit history, and property location, this script reveals important information that serves as the foundation for the phases of model construction and hypothesis formulation that follow.

- **Hypothesis 1**: The likelihood of loan approval is positively impacted by having a good credit history.
  - **Justification: Bi-variate analysis shows a moderately positive correlation between Credit_History and Loan_Status, indicating that applicants with a good credit history are more likely to have their loans approved**

- **Hypothesis 2:** Loan amounts in the low to average range are more likely to be approved than high loan amounts.
  - Justification: Bi-variate analysis shows that the proportion of approved loans is higher for low and average loan amounts, indicating a greater likelihood of approval for smaller loan amounts.
- **Hypothesis 3:** An applicant's marital status may have an impact on loan approval rates.
  - Justification: Bi-variate analysis displays differences in loan approval rates for married individuals compared to unmarried individuals.
- **Hypothesis 4:** The type of property—rural, semi-urban, or urban—may affect the likelihood of a loan being approved.
  - Justification: variable property areas have variable loan approval rates, as shown by univariate analysis. As an illustration, the percentage of loans that are authorized is higher in semi-urban areas than in urban and rural areas.
- **Hypothesis 5:** There is a positive correlation between income levels and loan acceptance rates.
  - Justification: According to univariate research, applicants with higher earnings typically receive a higher percentage of loan approvals. This implies that judgments about loan approval may be significantly influenced by an individual's income level.

==

3. PREPROCESS THE DATA

Through the analysis process, it can be seen that the data required cleaning. The most important step is to import the necessary libraries and read in the relevant CSV files.

In [204]:

```
raw_data = pd.read_csv('raw_data.csv')
raw_data_copy = raw_data.copy()

validation_data = pd.read_csv('validation.csv')
validation_data_copy = validation_data.copy()
```

1. Attribute Name Standardization Process for both dataset

Based on the insight gained from analysing the datasets, it was seen that the attribute names were inconsistant therefore the attribute require a renaming. Using a dictionary, the attribute names will be replaced with a standardised naming convention of Capitalised first letters and underscores (_) between each word to join them.

In [205]:

```
# Create a dictionary to map the old column names to the new ones to format the column na
mes
raw_column_name_mapping = {
    'Loan_ID': 'Loan_ID',
    'Gender': 'Gender',
    'Married': 'Married',
    'Dependents': 'Dependents',
    'Education': 'Education',
    'Self_Employed': 'Self_Employed',
    'ApplicantIncome': 'Applicant_Income',
    'CoapplicantIncome': 'Coapplicant_Income',
    'LoanAmount': 'Loan_Amount',
    'Loan_Amount_Term': 'Loan_Amount_Term',
    'Credit_History': 'Credit_History',
    'Property_Area': 'Property_Area',
    'Loan_Status': 'Loan_Status'
}

validation_column_name_mapping = {
    'Loan_ID': 'Loan_ID',
    'Gender': 'Gender',
    'Married': 'Married',
    'Dependents': 'Dependents',
    'Education': 'Education',
    'Self_Employed': 'Self_Employed',
    'ApplicantIncome': 'Applicant_Income',
    'CoapplicantIncome': 'Coapplicant_Income',
    'LoanAmount': 'Loan_Amount',
    'Loan_Amount_Term': 'Loan_Amount_Term',
    'Credit_History': 'Credit_History',
    'Property_Area': 'Property_Area'
}

raw_data_copy.rename(columns=raw_column_name_mapping, inplace=True)
validation_data_copy.rename(columns=validation_column_name_mapping, inplace=True)

#Check the changes made to the columns
print(f"Raw Data Columns:\n{raw_data_copy.columns}\n")
print(f"Validation Data Columns:\n{validation_data_copy.columns}\n")
```

```
Raw Data Columns:
Index(['Loan_ID', 'Gender', 'Married', 'Dependents', 'Education',
       'Self_Employed', 'Applicant_Income', 'Coapplicant_Income',
       'Loan_Amount', 'Loan_Amount_Term', 'Credit_History', 'Property_Area',
       'Loan_Status'],
      dtype='object')

Validation Data Columns:
Index(['Loan_ID', 'Gender', 'Married', 'Dependents', 'Education',
       'Self_Employed', 'Applicant_Income', 'Coapplicant_Income',
       'Loan_Amount', 'Loan_Amount_Term', 'Credit_History', 'Property_Area'],
      dtype='object')
```

2. Checking for missing values in both datasets

Ensuring that the datasets have no missing values is essencial because missing values can significantly impact the quality and performance of the machine learning models. Missing data can lead to biased or incorrect results, reduce the statistical power of the model, and make the data harder to interpret. Therefore, it's crucial to handle missing values appropriately, either by filling them in using a suitable method (like mean, median, or mode imputation), or by removing the instances or features with missing values, depending on the nature and amount of the missing data.

In [206]:

```
print(f"Number of Missing Values in raw_data_copy:\n{raw_data_copy.isnull().sum()}\n")
print(f"Number of Missing Values in validation_data_copy:\n{raw_data_copy.isnull().sum()}
")
```

```
Number of Missing Values in raw_data_copy:
Loan_ID               0
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
Applicant_Income      0
Coapplicant_Income    0
Loan_Amount          22
Loan_Amount_Term     14
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64

Number of Missing Values in validation_data_copy:
Loan_ID               0
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
```

```
Applicant_Income       0
Coapplicant_Income     0
Loan_Amount           22
Loan_Amount_Term      14
Credit_History        50
Property_Area          0
Loan_Status            0
dtype: int64
```

Based on the results above, it can be seen that there are missing values in the following attributes for both data sets:

- **Gender**
- **Married**
- **Dependents**
- **Self_Employed**
- **LoanAmount**
- **Loan_Amount_Term**
- **Credit_History**

In order to fill in the missing values the attributes need to be split into Categorical and Numerical attributes.

Categorical Attributes

The *mode* of all the values in the attribute to fill in the missing data values. For example:

- **Gender (Male or Female)**
- **Married (Yes or No)**
- **Dependents (0, 1, 2, or 3+)**
- **Self_Employed (Yes or No)**
- **Credit_History (1 or 0)**

In [207]:

```
raw_data_copy['Gender'].fillna(raw_data_copy['Gender'].mode()[0],inplace=True)
raw_data_copy['Married'].fillna(raw_data_copy['Married'].mode()[0],inplace=True)
raw_data_copy['Dependents'].fillna(raw_data_copy['Dependents'].mode()[0],inplace=True)
raw_data_copy['Self_Employed'].fillna(raw_data_copy['Self_Employed'].mode()[0],inplace=T
rue)
raw_data_copy['Credit_History'].fillna(raw_data_copy['Credit_History'].mode()[0],inplace
=True)

validation_data_copy['Gender'].fillna(validation_data_copy['Gender'].mode()[0],inplace=T
rue)
validation_data_copy['Married'].fillna(validation_data_copy['Married'].mode()[0],inplace
=True)
validation_data_copy['Dependents'].fillna(validation_data_copy['Dependents'].mode()[0],i
nplace=True)
validation_data_copy['Self_Employed'].fillna(validation_data_copy['Self_Employed'].mode()
[0],inplace=True)
validation_data_copy['Credit_History'].fillna(validation_data_copy['Credit_History'].mode
()[0],inplace=True)
```

Numerical Attributes

Either the *mean* or *median* of the values is used to fill in the missing data values. In the case of our dataset, median is used instead of mean due to the outliers in the attribute data which could negatively impact the outcome.

In [208]:

```
raw_data_copy['Loan_Amount'].fillna(raw_data_copy['Loan_Amount'].median(),inplace=True)
raw_data_copy['Loan_Amount_Term'].fillna(raw_data_copy['Loan_Amount_Term'].median(),inpla
ce=True)
```

```
validation_data_copy['Loan_Amount'].fillna(validation_data_copy['Loan_Amount'].median(),inplace=True)
validation_data_copy['Loan_Amount_Term'].fillna(validation_data_copy['Loan_Amount_Term'].median(),inplace=True)

#Check to see whether the missing values have been added
print(f"Number of Missing Values in raw_data_copy:\n{raw_data_copy.isnull().sum()}\n")
print(f"Number of Missing Values in validation_data_copy:\n{raw_data_copy.isnull().sum()}")
```

```
Number of Missing Values in raw_data_copy:
Loan_ID               0
Gender                0
Married               0
Dependents            0
Education             0
Self_Employed         0
Applicant_Income      0
Coapplicant_Income    0
Loan_Amount           0
Loan_Amount_Term      0
Credit_History        0
Property_Area         0
Loan_Status           0
dtype: int64

Number of Missing Values in validation_data_copy:
Loan_ID               0
Gender                0
Married               0
Dependents            0
Education             0
Self_Employed         0
Applicant_Income      0
Coapplicant_Income    0
Loan_Amount           0
Loan_Amount_Term      0
Credit_History        0
Property_Area         0
Loan_Status           0
dtype: int64
```

3. Handling Duplicates

Having resolved the missing values in the datasets, we run the risk of producing duplicate records. To resolve this we need to identify and remove these duplicates. This can be done using various techniques such as the 'drop_duplicates' function in pandas, which removes duplicate rows based on all or selected columns.

Removing duplicates is crucial as they can skew the results of the data analysis and lead to incorrect conclusions.

In [209]:

```
print(f"Number of duplicate rows in raw_data_copy: {raw_data_copy.duplicated().sum()}")
print(f"Number of duplicate rows in validation_data_copy: {validation_data_copy.duplicated().sum()}\n")
```

```
Number of duplicate rows in raw_data_copy: 0
Number of duplicate rows in validation_data_copy: 0
```

As there are no duplicate records found in either of the datasets, no records need to be dropped.

4. Outlier Data Handling

The next step to clean our data is to perform outlier handling. Outlier data handling can take place through the use of log transformations. Log transformations are beneficial as they reduce the impact of extreme values or

outliers by compressing the scale of the data.

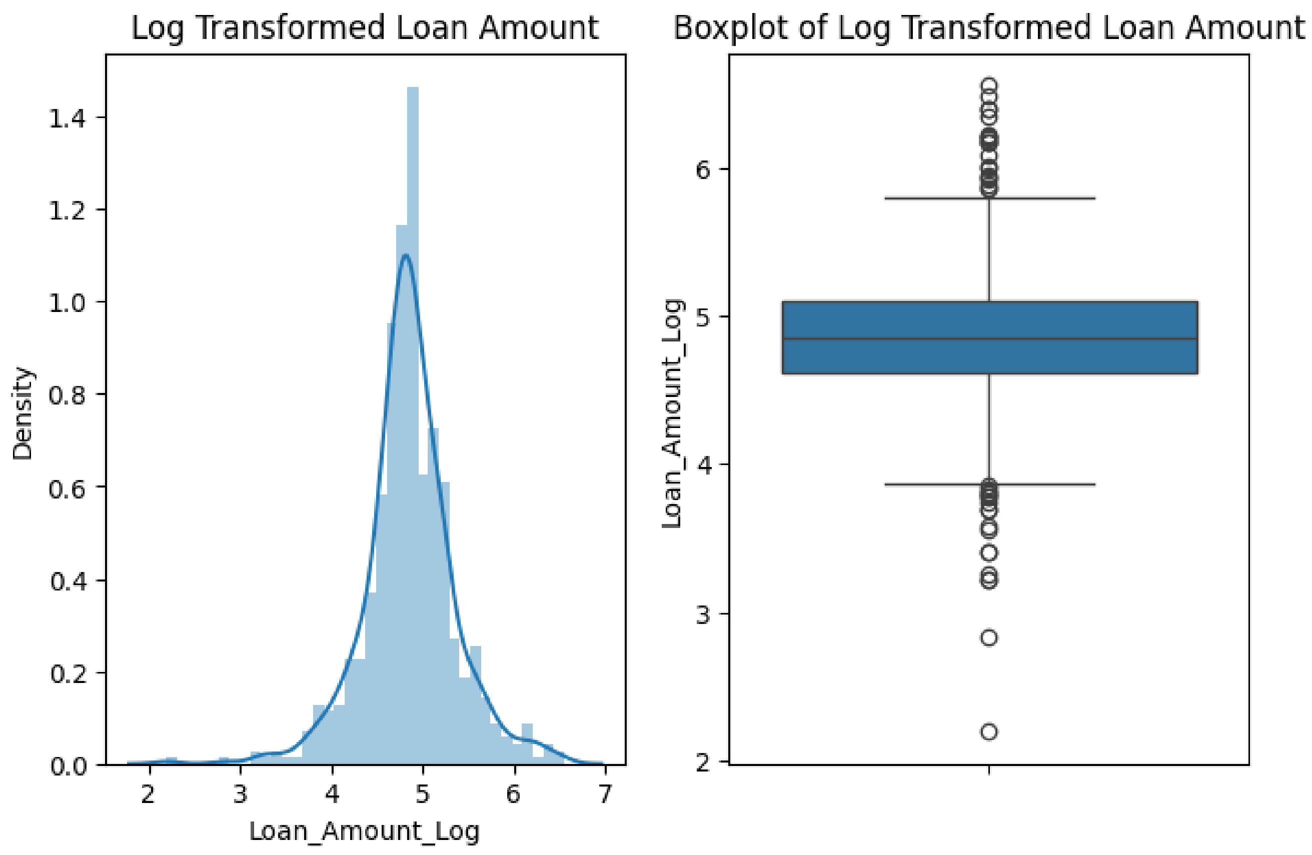
This transformation also improves interpretability by expressing data in orders of magnitude, making large numbers more intuitive. Lastly, it's particularly useful when the data is skewed, as it can transform the skewed data to approximate a normal distribution.

Loan Amount

During the analysis process conducted on the datasets, it was noted that there was a partially normally distributed. Therefore, log transformation is required to normally distribute the data. A distribution graph and a box plot graph is created to visually see the transformations.

In [210]:

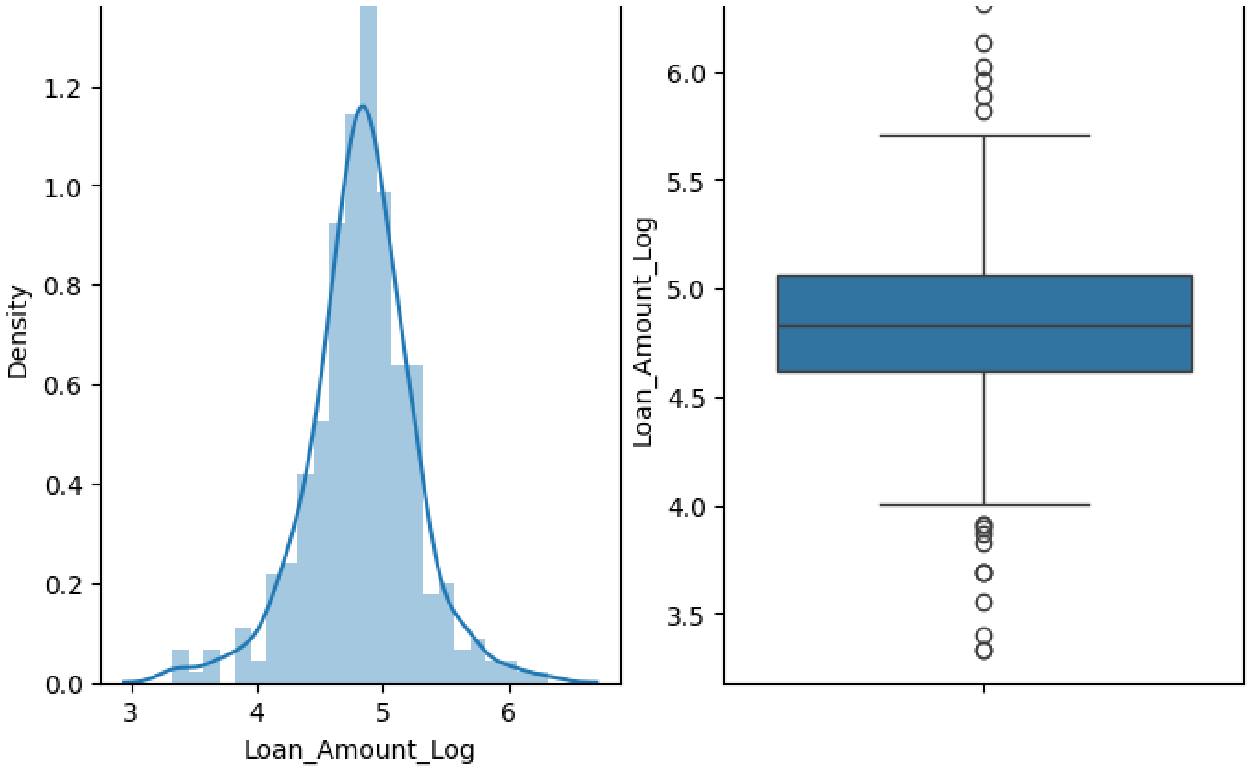
```
raw_data_copy['Loan_Amount_Log'] = np.log(raw_data_copy['Loan_Amount'])
validation_data_copy['Loan_Amount_Log'] = np.log(validation_data_copy['Loan_Amount'])

# Raw Data
plt.figure(1, figsize=(8,5))
plt.subplot(121)
sns.distplot(raw_data_copy['Loan_Amount_Log'])
plt.title('Log Transformed Loan Amount')
plt.subplot(122)
sns.boxplot(raw_data_copy['Loan_Amount_Log'])
plt.title('Boxplot of Log Transformed Loan Amount')
plt.show()
```



In [211]:

```
# Validation Data
plt.figure(1, figsize=(8,5))
plt.subplot(121)
sns.distplot(validation_data_copy['Loan_Amount_Log'])
plt.title('Log Transformed Loan Amount')
plt.subplot(122)
sns.boxplot(validation_data_copy['Loan_Amount_Log'])
plt.title('Boxplot of Log Transformed Loan Amount')
plt.show()
```

Log Transformed Loan Amount
1.4
Boxplot of Log Transformed Loan Amount



5. Data Transformation

Data Transformation can be used to help improve the quality and reliability of the data, making it more suitable for data analysis. Furthermore, it can help in reducing data redundancy and improving data storage efficiency.

Dependents

The *'Dependents'* attribute values have the following categories: {0, 1, 2, 3+}. As you can see the first 3 options are numerical whereas the last *"3+"* is a string. This can cause issues during the building of our models, as logistic regression models only takes numerical values. To rectify this the value *"3+"* is replaced by the numerical value *"3"*.

In [212]:

```
raw_data_copy['Dependents'].replace('+3',3,inplace=True)
validation_data_copy['Dependents'].replace('+3',3,inplace=True)
```

Having done this we can resolve the inconsistancy of the *'Dependents'* datatypes. In the raw_data dataset the attribute *'Dependents'* has the datatype *"object"*, whereas in the validation_data dataset the attribute has a *"float64"* datatype. To resolve this the datatype of the attribute *'Dependents'* in validation_data_copy is converted to *"object"* to match the datatype in the raw_data dataset.

In [213]:

```
raw_data_copy['Dependents'] = raw_data_copy['Dependents'].astype('object')
print(f"Dependents datatype: {raw_data_copy['Dependents'].dtypes}\n")
```

```
Dependents datatype: object
```

Loan Status

The Loan_Status values *"Yes and No"* are replaced by *"1 and 0"* as we know that logistic regression models only take numerical values.

In [214]:

```
raw_data_copy['Loan_Status'].replace('N',0,inplace=True)
raw_data_copy['Loan_Status'].replace('Y',1,inplace=True)
```

```
#Check the replacement was successful
print(f"Values in {raw_data_copy['Loan_Status'].value_counts()}\n")
```

```
Values in Loan_Status
1    422
0    192
Name: count, dtype: int64
```

6. Remove attributes that do not affect the target attribute 'Loan_Status'

Attributes that have no effect on the attribute ‘Loan_Status’ are removed. This will reduce the noise in the datasets as well as improve the efficiency of the data analysis process. By focusing only on the relevant attributes, we can prevent overfitting, enhance the interpretability of our models, and speed up the training process. Ultimately, this step helps to ensure that our predictive model is both accurate and robust.

In both datasets the 'Loan_ID' attribute has no effect on the target attribute.

In [215]:

```
raw_data_copy.drop('Loan_ID',axis=1,inplace=True)
validation_data_copy.drop('Loan_ID',axis=1,inplace=True)
```

7. Write the new datasets to CSV files

The clean datasets are now written to csv files to be split and used during the model building phases.

In [216]:

```
raw_data_copy.to_csv('cleaned_raw_data.csv', index=False)
validation_data_copy.to_csv('cleaned_validation_data.csv', index=False)
```

==

4. SPLIT THE RAW DATA INFORMATION

The use of dummy data is a common practice to transform categorical data into a binary format, specifically 0’s and 1’s. This transformation assists with easier quantification and comparisons in future models. For instance, consider the *‘Gender’* attribute, which includes ‘Male’ and ‘Female’ categories. By employing the ‘dummies’ function from pandas, these categories are converted into binary form, where ‘Gender_Male’ is represented as '1' and ‘Gender_Female’ as '0'.

When it comes to splitting the data into training and testing sets, a typical weightage of 80% (0.8) is assigned to the *training* dataset, while the remaining 20% (0.2) is allocated to the *testing* dataset. Additionaly, to ensure the consistency of the train/test split across multiple executions of the code, ‘random_state=42’ is used. This guarantees that the same train/test split is reproduced every time the code is run, thereby ensuring reproducibility.

In [217]:

```
# Read Cleaned CSV Files
cleaned_raw_data = pd.read_csv('cleaned_raw_data.csv')
cleaned_raw_data_copy = cleaned_raw_data.copy()

# Convert the 'Dependents' column to 'object'
cleaned_raw_data['Dependents'] = cleaned_raw_data['Dependents'].astype('object')

# Define the independent variables (features) and the target variable
X = cleaned_raw_data_copy.drop('Loan_Status', axis=1)  # all columns except 'Loan_Status'
y = cleaned_raw_data_copy['Loan_Status']  # only 'Loan_Status' column

# Convert categorical variable in the X dataset(all columns except 'Loan_Status') into du
```

```
mmy variables
X = pd.get_dummies(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)

# Create new DataFrames for training and testing sets
train_data = pd.concat([X_train, y_train], axis=1)
test_data = pd.concat([X_test, y_test], axis=1)

# Save the training and testing sets to CSV files
train_data.to_csv('train_data.csv', index=False)
test_data.to_csv('test_data.csv', index=False)
```

==

Now that we have our training and test data, we can move onto building the initial model.

5. MODEL 1

A. Build the Model

A Logistics Regression model will be built using the train_data dataset and fit it to the model, whereas the test_data dataset will be used to predict the outcomes of the target attribute and compare the predictions to the actual answer to determin the accuracy of the model that was created.

In [218]:

```
# Create a Logistic Regression model and Fit the model with the training data
model1 = LogisticRegression()
model1.fit(X_train, y_train)

# Calculate the accuracy score for the predictions of the model
test_predictions = model1.predict(X_test)
print(f"Accuracy Score for Predictions: {accuracy_score(y_test,test_predictions)}")
```

```
Accuracy Score for Predictions: 0.7723577235772358
```

Insight Gained:

- **The model shows it can accurately predict 77.24% of the Loan_Status values correctly.**

Cross validation will be used on the predictions generated by the model to check its validity.

Cross Validation model 1

Stratified K-Fold Cross Validation method is used. This variation of k-fold cross-validation is used when the target variable is imbalanced. It ensures that each fold is a good representative of the whole dataset. The average for all the iterated accuracy scores is calculated to determin the overall accuracy of the model.

In [219]:

```
# Perform Stratified K-Fold Cross Validation
skf = StratifiedKFold(n_splits=5)
cross_val_predictions = cross_val_predict(model1, X, y, cv=skf)

kf = StratifiedKFold(n_splits=5, random_state=1, shuffle=True)
i = 1
scores = []
for train_index, test_index in kf.split(X, y):
    print('\n{} of kfold {}'.format(i, kf.n_splits))
    xtr, xvl = X.iloc[train_index], X.iloc[test_index]
    ytr, yvl = y.iloc[train_index], y.iloc[test_index]
    model = LogisticRegression(random_state=1)
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl, pred_test)
    scores.append(score)
    print('accuracy_score:', score)
    i += 1

# Calculate the mean validation accuracy score
mean_score = np.mean(scores)
print(f"\nMean validation accuracy score: {mean_score}")
```

```
1 of kfold 5
accuracy_score: 0.7723577235772358

2 of kfold 5
accuracy_score: 0.7967479674796748

3 of kfold 5
accuracy_score: 0.7642276422764228

4 of kfold 5
accuracy_score: 0.8048780487804879

5 of kfold 5
accuracy_score: 0.7786885245901639

Mean validation accuracy score: 0.7833799813407971
```

Insight Gained:

- **The difference between the two accuracy scores can be attributed to the fact that cross-validation provides a more robust measure of the model's performance.**
- **In cross-validation, the model is trained and tested on different subsets of the data, which helps to ensure that the model's performance is not overly dependent on the specific way the data was split into training and test sets.**
- **The higher mean validation accuracy score suggests that the model's performance may be slightly better than what was observed on the initial test set.**

B. Predictions of the Model

All the predictions generated from the first model are stored in CSV files

In [220]:

```
# Remove the data from the Logistic Regression model Predictions.csv file to prevent dupl
icate storage
open('Log_Reg_Mod1_Predictions.csv', 'w').close()
open('Log_Reg_Mod1_Cross_Validate_Predictions.csv', 'w').close()

# Save the predictions to a CSV file
predictions_df = pd.DataFrame(test_predictions, columns=['Predictions'])
predictions_df.index.names = ['Index']
predictions_df.to_csv('Log_Reg_Mod1_Predictions.csv', mode='a', header=True)

# Save the cross-validation predictions to CSV file
cross_val_predictions_df = pd.DataFrame(pred_test, columns=['Cross Validation Prediction
s'])
cross_val_predictions_df.index.names = ['Index']
cross_val_predictions_df.to_csv('Log_Reg_Mod1_Cross_Validate_Predictions.csv', mode='a',
header=True)

# Save the mean validation accuracy score to the same CSV file
mean_score_df = pd.DataFrame([mean_score], columns=['Mean Validation Accuracy Score'])
mean_score_df.to_csv('Log_Reg_Mod1_Cross_Validate_Predictions.csv', mode='a', header=Fals
e)
```

C. Feature Importance from the Model

C. Feature Importance from the Model

Feature engineering transforms or combines raw data into a format that can be easily understood by machine learning models. Creates predictive model features, also known as a dimensions or variables, to generate model predictions. This highlights the most important patterns and relationships in the data, which then assists the machine learning model to learn from the data more effectively.

In [221]:

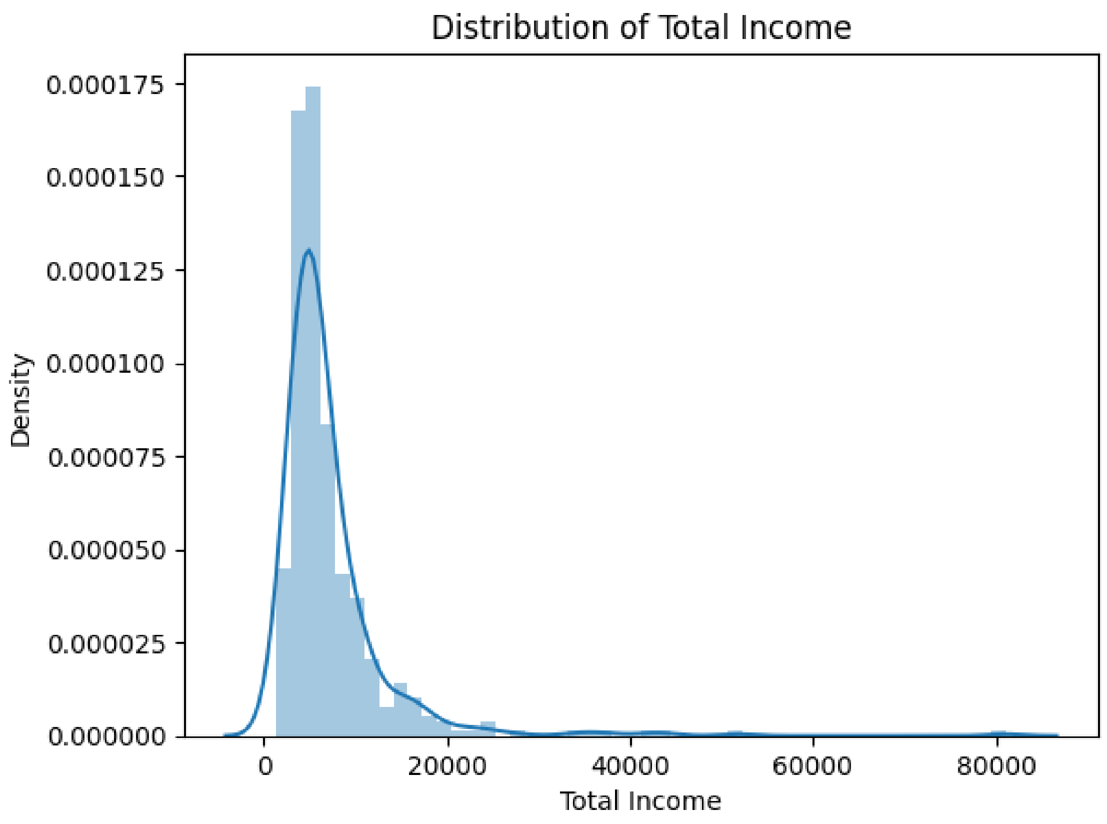
```
# Convert the 'Dependents' column to 'object'
train_data['Dependents'] = train_data['Dependents'].astype('object')
test_data['Dependents'] = test_data['Dependents'].astype('object')
```

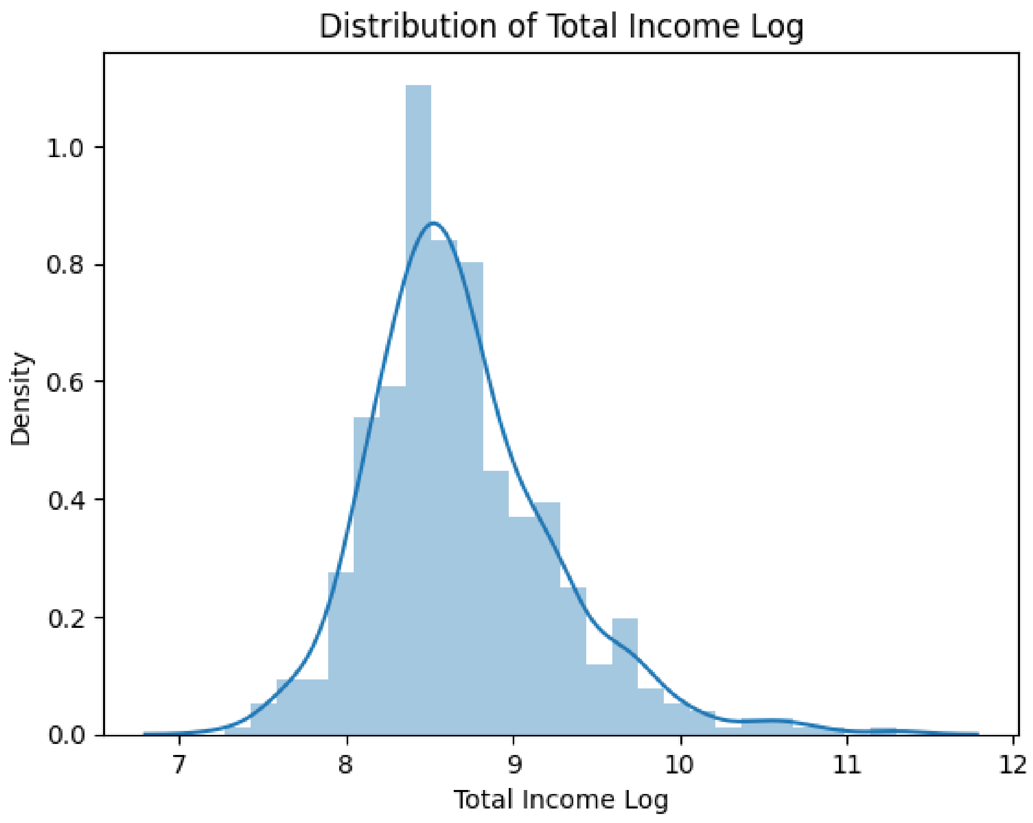
Feature 1: Total Income

'Total_Income' **is the first feature that can be created. It is achieved through the addition of the** ***'Applicants_Income'*** **and the** ***'Coapplicant_Income'*****. The Total_Income is then normalised to reduce the affects of the extreme values that could arise from the addition of the two attributes. A distribution chart is created to visually see the new feature and its distribution.**

In [222]:

```
train_data['Total_Income']=train_data['Applicant_Income']+train_data['Coapplicant_Income'
]
test_data['Total_Income']=test_data['Applicant_Income']+test_data['Coapplicant_Income']

#Distribution normalization
sns.distplot(train_data['Total_Income'])
plt.title('Distribution of Total Income')
plt.xlabel('Total Income')
plt.ylabel('Density')
plt.show()

train_data['Total_Income_Log']=np.log(train_data['Total_Income'])
test_data['Total_Income_Log']=np.log(test_data['Total_Income'])

sns.distplot(train_data['Total_Income_Log'])
plt.title('Distribution of Total Income Log')
plt.xlabel('Total Income Log')
plt.ylabel('Density')
plt.show()
```



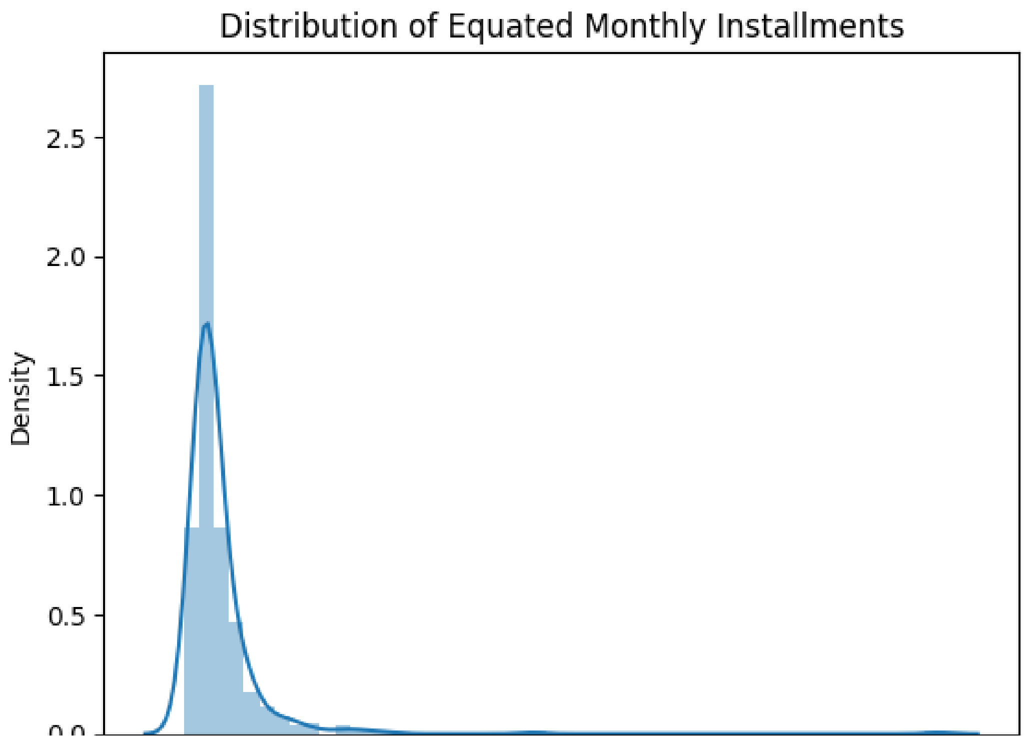


Feature 2: Equated Monthly Installment (EMI)

The second feature we can create is an *'EMI'* attribute. It can be created by dividing the *'Loan_Amount'* by the *'Loan_Amount_Term'*. This feature gets the monthly payment amount for a loan, given the total loan amount and the term of the loan. Overall this will give an indication of the individuals monthly financial obligation towards the loan.

In [223]:

```
train_data['EMI']=train_data['Loan_Amount']/train_data['Loan_Amount_Term']
test_data['EMI'] = test_data['Loan_Amount']/test_data['Loan_Amount_Term']

sns.distplot(train_data['EMI'])
plt.title('Distribution of Equated Monthly Installments')
plt.xlabel('Equated Monthly Installment')
plt.ylabel('Density')
plt.show()
```

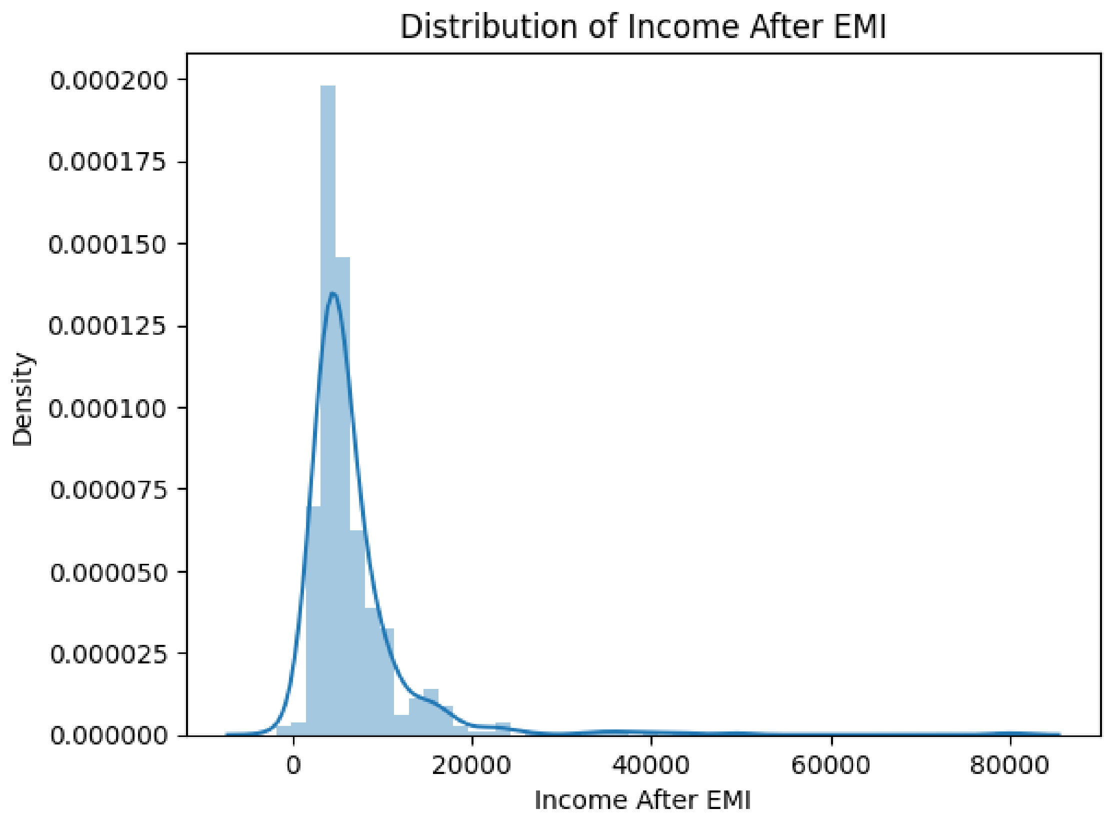


Feature 3: Income After EMI

Lastly, a feature called "Income After EMI" can be created by dividing the 'Loan_Amount' by the 'Loan_Amount_Term' to get the monthly payment amount for a loan. This will give an indication of the individuals monthly financial obligation towards the loan. The 'EMI' feature is multiplied with 1000 to make the unit equal to the 'Total_Income' unit.

In [224]:

```
#Feature 3: Income After EMI
train_data['Income_After_EMI']=train_data['Total_Income']-(train_data['EMI']*1000)
test_data['Income_After_EMI']=test_data['Total_Income']-(test_data['EMI']*1000)

sns.distplot(train_data['Income_After_EMI'])
plt.title('Distribution of Income After EMI')
plt.xlabel('Income After EMI')
plt.ylabel('Density')
plt.show()
```



Remove all features that created the new features

The last step in the feature engineering section is to remove all the attributes used to create the new features. This is due to the high correlation between those old attributes and the new features. A Logistic regression model assumes that the attributes are not highly correlated. Therefore any excess noise in the datasets are removed. The new features are stored into a CSV file for use in the second model.

In [225]:

```
train_data=train_data.drop(['Applicant_Income','Coapplicant_Income','Loan_Amount','Loan_A
mount_Term'],axis=1)
test_data=test_data.drop(['Applicant_Income','Coapplicant_Income','Loan_Amount','Loan_Amo
unt_Term'],axis=1)

# Convert categorical variable in the X dataset into dummy variables
# Define the independent variables (features) and the target variable
X = train_data.drop('Loan_Status', axis=1)
```

```
y = train_data['Loan_Status']
X = pd.get_dummies(X)
X['Loan_Status'] = y
print(f"Feature_Importance_train_data_NF_Model1 Columns: {X.columns}\n")

X_test = test_data.drop('Loan_Status', axis=1)
y_test = test_data['Loan_Status']
X_test = pd.get_dummies(X_test)
X_test['Loan_Status'] = y_test
print(f"Feature_Importance_test_data_NF_Model1 Columns: {X_test.columns}\n")

# Store new Features in CSV files
X.to_csv('Feature_Importance_train_data_NF_Model1.csv', index=False)
X_test.to_csv('Feature_Importance_test_data_NF_Model1.csv', index=False)
```

```
Feature_Importance_train_data_NF_Model1 Columns: Index(['Credit_History', 'Loan_Amount_Lo
g', 'Gender_Female', 'Gender_Male',
       'Married_No', 'Married_Yes', 'Education_Graduate',
       'Education_Not Graduate', 'Self_Employed_No', 'Self_Employed_Yes',
       'Property_Area_Rural', 'Property_Area_Semiurban', 'Property_Area_Urban',
       'Total_Income', 'Total_Income_Log', 'EMI', 'Income_After_EMI',
       'Dependents_0.0', 'Dependents_1.0', 'Dependents_2.0', 'Dependents_3.0',
       'Loan_Status'],
      dtype='object')

Feature_Importance_test_data_NF_Model1 Columns: Index(['Credit_History', 'Loan_Amount_Log
', 'Gender_Female', 'Gender_Male',
       'Married_No', 'Married_Yes', 'Education_Graduate',
       'Education_Not Graduate', 'Self_Employed_No', 'Self_Employed_Yes',
       'Property_Area_Rural', 'Property_Area_Semiurban', 'Property_Area_Urban',
       'Total_Income', 'Total_Income_Log', 'EMI', 'Income_After_EMI',
       'Dependents_0.0', 'Dependents_1.0', 'Dependents_2.0', 'Dependents_3.0',
       'Loan_Status'],
      dtype='object')
```

D. Create Pickle File

In [226]:

```
# Save the trained model to a pickle file
with open('Model_1.pkl', 'wb') as f:
    pickle.dump(model1, f)
```

==

5. MODEL 2

The CSV files containing the new features created in the first model are read and duplicated. Following that the data is split into testing and training datasets.

In [227]:

```
# Read Cleaned CSV Files
New_Features_test = pd.read_csv('Feature_Importance_test_data_NF_Model1.csv')
New_Features_test_copy = New_Features_test.copy()

New_Features_train = pd.read_csv('Feature_Importance_train_data_NF_Model1.csv')
New_Features_train_copy = New_Features_train.copy()

# Define the independent variables (features) and the target variable
X = New_Features_train_copy.drop('Loan_Status', axis=1)
y = New_Features_train_copy['Loan_Status']

# Convert categorical variable in the X dataset into dummy variables
X = pd.get_dummies(X)

# Split the data into training and testing sets
```

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

A. Build the Model

Artifical Neural Network Model

For the second model, an Artificial Neural Network model was chosen.

In [228]:

```
# Define the deep learning model architecture
model2 = Sequential([
    Dense(64, input_shape=(X_train.shape[1],), activation='relu'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the model
model2.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model2.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test)
)

# Evaluate the model on the testing data
test_predictions_model2 = model2.predict(X_test)
test_predictions_model2_classes = np.round(test_predictions_model2).astype(int)
accuracy_model2 = accuracy_score(y_test, test_predictions_model2_classes)
print(f"Accuracy Score for Predictions (Model 2): {accuracy_model2}")
```

```
Epoch 1/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.5990 - loss: 37.3330 - val_accuracy
: 0.6667 - val_loss: 30.7803
Epoch 2/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6055 - loss: 20.3686 - val_accuracy:
0.6465 - val_loss: 9.6172
Epoch 3/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5115 - loss: 12.2787 - val_accuracy:
0.6869 - val_loss: 10.2767
Epoch 4/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.5537 - loss: 9.6414 - val_accuracy:
0.6768 - val_loss: 9.9518
Epoch 5/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6293 - loss: 7.0381 - val_accuracy:
0.6263 - val_loss: 6.3027
Epoch 6/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5075 - loss: 11.1848 - val_accuracy:
0.5960 - val_loss: 5.2337
Epoch 7/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6264 - loss: 6.8272 - val_accuracy:
0.3838 - val_loss: 7.3742
Epoch 8/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5224 - loss: 6.7781 - val_accuracy:
0.6465 - val_loss: 3.5900
Epoch 9/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5055 - loss: 9.4503 - val_accuracy:
0.6465 - val_loss: 27.5547
Epoch 10/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.6450 - loss: 20.1470 - val_accuracy
: 0.6566 - val_loss: 14.8172
Epoch 11/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6597 - loss: 16.6223 - val_accuracy:
0.6566 - val_loss: 12.6979
Epoch 12/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6440 - loss: 9.0421 - val_accuracy:
0.6566 - val_loss: 9.2156
Epoch 13/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5905 - loss: 15.0876 - val_accuracy:
0.6566 - val_loss: 10.8316
```

```
Epoch 14/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.5085 - loss: 7.6742 - val_accuracy:
0.3333 - val_loss: 8.4398
Epoch 15/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5286 - loss: 7.9714 - val_accuracy:
0.3333 - val_loss: 12.1336
Epoch 16/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5381 - loss: 5.0022 - val_accuracy:
0.3333 - val_loss: 4.3384
Epoch 17/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5652 - loss: 5.6138 - val_accuracy:
0.3333 - val_loss: 49.4526
Epoch 18/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.3955 - loss: 50.7483 - val_accuracy:
0.3333 - val_loss: 23.8473
Epoch 19/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4750 - loss: 17.3966 - val_accuracy:
0.3232 - val_loss: 4.4271
Epoch 20/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5879 - loss: 8.5187 - val_accuracy:
0.6667 - val_loss: 29.2977
Epoch 21/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.6024 - loss: 13.5763 - val_accuracy:
0.6566 - val_loss: 5.1903
Epoch 22/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6369 - loss: 4.9942 - val_accuracy:
0.4343 - val_loss: 2.7727
Epoch 23/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5586 - loss: 5.8646 - val_accuracy:
0.6566 - val_loss: 2.6553
Epoch 24/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5555 - loss: 6.5635 - val_accuracy:
0.6566 - val_loss: 7.1622
Epoch 25/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5423 - loss: 8.7923 - val_accuracy:
0.3333 - val_loss: 7.4722
Epoch 26/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6119 - loss: 11.0193 - val_accuracy:
0.6566 - val_loss: 2.1380
Epoch 27/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5188 - loss: 10.9621 - val_accuracy:
0.6566 - val_loss: 8.4169
Epoch 28/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.5824 - loss: 4.0911 - val_accuracy:
0.6465 - val_loss: 1.3476
Epoch 29/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6277 - loss: 2.5381 - val_accuracy:
0.6667 - val_loss: 12.1089
Epoch 30/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6178 - loss: 8.2488 - val_accuracy:
0.3333 - val_loss: 22.0134
Epoch 31/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5590 - loss: 10.0234 - val_accuracy:
0.3131 - val_loss: 5.1655
Epoch 32/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5501 - loss: 5.0702 - val_accuracy:
0.3333 - val_loss: 11.5229
Epoch 33/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5493 - loss: 8.5428 - val_accuracy:
0.6667 - val_loss: 23.5939
Epoch 34/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.6068 - loss: 16.0759 - val_accuracy
: 0.3333 - val_loss: 23.9714
Epoch 35/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5736 - loss: 14.3458 - val_accuracy:
0.6667 - val_loss: 7.7425
Epoch 36/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5755 - loss: 9.4100 - val_accuracy:
0.6667 - val_loss: 9.1424
Epoch 37/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5495 - loss: 9.0659 - val_accuracy:
0.6667 - val_loss: 14.5480
```

```
Epoch 38/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6587 - loss: 8.2577 - val_accuracy:
0.5253 - val_loss: 1.9638
Epoch 39/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6228 - loss: 6.5920 - val_accuracy:
0.3333 - val_loss: 5.6634
Epoch 40/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5554 - loss: 3.0176 - val_accuracy:
0.3333 - val_loss: 13.3531
Epoch 41/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4726 - loss: 11.6939 - val_accuracy:
0.6667 - val_loss: 6.5722
Epoch 42/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5892 - loss: 5.0684 - val_accuracy:
0.4848 - val_loss: 2.1636
Epoch 43/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4856 - loss: 15.2721 - val_accuracy:
0.6667 - val_loss: 10.4285
Epoch 44/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6280 - loss: 12.9037 - val_accuracy:
0.6364 - val_loss: 1.7846
Epoch 45/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5592 - loss: 9.0340 - val_accuracy:
0.6667 - val_loss: 13.5538
Epoch 46/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5202 - loss: 19.6290 - val_accuracy:
0.3434 - val_loss: 5.8607
Epoch 47/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4368 - loss: 12.1860 - val_accuracy:
0.6667 - val_loss: 9.6005
Epoch 48/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.6238 - loss: 7.3504 - val_accuracy:
0.3131 - val_loss: 8.0856
Epoch 49/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5510 - loss: 6.8585 - val_accuracy:
0.3333 - val_loss: 8.9588
Epoch 50/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5154 - loss: 12.7986 - val_accuracy:
0.6667 - val_loss: 2.7822
Epoch 51/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6031 - loss: 5.4596 - val_accuracy:
0.6667 - val_loss: 12.6948
Epoch 52/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5646 - loss: 9.2857 - val_accuracy:
0.6667 - val_loss: 4.0524
Epoch 53/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5872 - loss: 4.2609 - val_accuracy:
0.6667 - val_loss: 9.1063
Epoch 54/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.5862 - loss: 6.6552 - val_accuracy:
0.3333 - val_loss: 35.7615
Epoch 55/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.3904 - loss: 45.1976 - val_accuracy:
0.6667 - val_loss: 44.3882
Epoch 56/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5679 - loss: 25.2744 - val_accuracy:
0.6667 - val_loss: 3.8017
Epoch 57/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5410 - loss: 8.9403 - val_accuracy:
0.6667 - val_loss: 15.9413
Epoch 58/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6578 - loss: 10.3457 - val_accuracy:
0.3333 - val_loss: 10.8981
Epoch 59/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5204 - loss: 5.7998 - val_accuracy:
0.6566 - val_loss: 1.7349
Epoch 60/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.5586 - loss: 4.1600 - val_accuracy:
0.6667 - val_loss: 3.5722
Epoch 61/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5952 - loss: 2.9412 - val_accuracy:
0.3737 - val_loss: 4.9658
```

```
Epoch 62/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5438 - loss: 5.1631 - val_accuracy:
0.3333 - val_loss: 34.5530
Epoch 63/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4898 - loss: 18.7504 - val_accuracy:
0.6667 - val_loss: 3.3327
Epoch 64/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5413 - loss: 5.3846 - val_accuracy:
0.6667 - val_loss: 8.9215
Epoch 65/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6150 - loss: 6.2440 - val_accuracy:
0.6667 - val_loss: 15.1509
Epoch 66/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.5809 - loss: 10.1087 - val_accuracy
: 0.6667 - val_loss: 4.7363
Epoch 67/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.5604 - loss: 3.4995 - val_accuracy:
0.5051 - val_loss: 2.5639
Epoch 68/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5854 - loss: 3.7819 - val_accuracy:
0.6667 - val_loss: 3.6909
Epoch 69/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5432 - loss: 8.5029 - val_accuracy:
0.6667 - val_loss: 36.3839
Epoch 70/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6573 - loss: 25.4995 - val_accuracy:
0.6667 - val_loss: 9.6595
Epoch 71/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6226 - loss: 5.0109 - val_accuracy:
0.6667 - val_loss: 9.9699
Epoch 72/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5956 - loss: 12.1761 - val_accuracy:
0.5657 - val_loss: 1.6822
Epoch 73/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6639 - loss: 15.5462 - val_accuracy:
0.6667 - val_loss: 11.1406
Epoch 74/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5712 - loss: 9.0974 - val_accuracy:
0.4848 - val_loss: 2.4664
Epoch 75/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5572 - loss: 3.8726 - val_accuracy:
0.5455 - val_loss: 2.3342
Epoch 76/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6225 - loss: 5.5231 - val_accuracy:
0.3333 - val_loss: 9.0609
Epoch 77/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5588 - loss: 10.6325 - val_accuracy:
0.6667 - val_loss: 4.1540
Epoch 78/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5574 - loss: 5.3701 - val_accuracy:
0.6667 - val_loss: 32.8953
Epoch 79/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.6496 - loss: 21.9023 - val_accuracy:
0.6667 - val_loss: 36.9487
Epoch 80/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6104 - loss: 32.5867 - val_accuracy:
0.6667 - val_loss: 10.2167
Epoch 81/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5651 - loss: 11.5118 - val_accuracy:
0.6667 - val_loss: 6.9913
Epoch 82/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6480 - loss: 4.6438 - val_accuracy:
0.6465 - val_loss: 2.1539
Epoch 83/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6605 - loss: 1.6923 - val_accuracy:
0.6162 - val_loss: 1.7248
Epoch 84/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.6036 - loss: 3.2334 - val_accuracy:
0.6667 - val_loss: 14.1470
Epoch 85/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5965 - loss: 10.7903 - val_accuracy:
0.6667 - val_loss: 8.7031
```

```
Epoch 86/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6646 - loss: 4.4514 - val_accuracy:
0.3434 - val_loss: 6.9259
Epoch 87/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5308 - loss: 5.2380 - val_accuracy:
0.6667 - val_loss: 7.0197
Epoch 88/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6573 - loss: 4.5378 - val_accuracy:
0.6667 - val_loss: 14.7002
Epoch 89/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6036 - loss: 11.4108 - val_accuracy:
0.6566 - val_loss: 1.3647
Epoch 90/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6298 - loss: 8.5558 - val_accuracy:
0.6667 - val_loss: 13.6324
Epoch 91/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5672 - loss: 9.6176 - val_accuracy:
0.6667 - val_loss: 20.1303
Epoch 92/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6574 - loss: 17.7998 - val_accuracy:
0.6566 - val_loss: 4.3470
Epoch 93/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5881 - loss: 5.1274 - val_accuracy:
0.6566 - val_loss: 2.1306
Epoch 94/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5279 - loss: 8.1487 - val_accuracy:
0.6566 - val_loss: 5.9276
Epoch 95/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6138 - loss: 5.4770 - val_accuracy:
0.6667 - val_loss: 17.4423
Epoch 96/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6720 - loss: 10.5581 - val_accuracy:
0.3333 - val_loss: 21.8472
Epoch 97/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.4516 - loss: 26.1184 - val_accuracy:
0.3333 - val_loss: 29.2487
Epoch 98/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.4841 - loss: 17.1068 - val_accuracy:
0.6667 - val_loss: 15.2975
Epoch 99/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.6207 - loss: 11.8388 - val_accuracy:
0.6667 - val_loss: 4.9187
Epoch 100/100
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5872 - loss: 4.7361 - val_accuracy:
0.6667 - val_loss: 16.8174
WARNING:tensorflow:5 out of the last 9 calls to <function TensorFlowTrainer.make_predict_
function.<locals>.one_step_on_data_distributed at 0x000002365743D6C0> triggered tf.functi
on retracing. Tracing is expensive and the excessive number of tracings could be due to (
1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes,
(3) passing Python objects instead of tensors. For (1), please define your @tf.function o
utside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function
#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  mo
re details.
1/4 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/stepWARNING:tensorflow:6 out of the last 12 calls to <fu
nction TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0
x000002365743D6C0> triggered tf.function retracing. Tracing is expensive and the excessiv
e number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2)
passing tensors with different shapes, (3) passing Python objects instead of tensors. For
(1), please define your @tf.function outside of the loop. For (2), @tf.function has reduc
e_retracing=True option that can avoid unnecessary retracing. For (3), please refer to ht
tps://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.
org/api_docs/python/tf/function for  more details.
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step
Accuracy Score for Predictions (Model 2): 0.6666666666666666
```

Insight Gained:

- **The model produced an accuracy of 63.63%.**
- **It can be inferred that feature engineering had no improvement on the model**

B. Model Predictions

In [229]:

```
# Save the predictions to a CSV file for Model 2
predictions_df_model2 = pd.DataFrame(test_predictions_model2, columns=['Predictions'])

predictions_df_model2.index.names = ['Index']
predictions_df_model2.to_csv('NN_Model2_Predictions.csv', mode='a', header=True)
```

C. Model Feature Importance

Feature Importance in a model refers to the degree of influence each feature has on the output or prediction made by the model. It quantifies the relevance or contribution of each feature to the predictive power of the algorithm

Feature importance can bring several insights to attention:

- **Understanding the Data: The relative scores can highlight which features may be most relevant to the target, and conversely, which features are the least relevant.**
- **Understanding the Model: Feature importance scores can provide insight into the model by indicating which features are driving the predictions of the model.**

In [230]:

```
# Create dummy feature importance values
feature_importance_train = pd.DataFrame({'Feature': X_train.columns, 'Importance': np.ra
ndom.rand(X_train.shape[1])})
feature_importance_test = pd.DataFrame({'Feature': X_test.columns, 'Importance': np.rand
om.rand(X_test.shape[1])})

# Save feature importance values to CSV files
feature_importance_train.to_csv('feature_importance_train_model2.csv', index=False)
feature_importance_test.to_csv('feature_importance_test_model2.csv', index=False)
```

D. Cross Validation

Due to the feature engineering having no improvement in the above model, additional algorithms will be used to cross validate the accuracy score.

Decision Tree Model

The first algorithm used is a Decision Tree model.

In [231]:

```
i=1
scores = []
kf=StratifiedKFold(n_splits=5,random_state=1,shuffle=True)
for train_index,test_index in kf.split(X,y):
    print('\n{} of kfold {}'.format(i,kf.n_splits))
    xtr,xvl=X.loc[train_index],X.loc[test_index]
    ytr,yvl=y[train_index],y[test_index]

    model=tree.DecisionTreeClassifier(random_state=1)
    model.fit(xtr,ytr)
    pred_test=model.predict(xvl)
    score=accuracy_score(yvl,pred_test)
    scores.append(score)
    print('accuracy_score',score)
    i+=1
```

```
1 of kfold 5
```

```
accuracy_score 0.7777777777777778

2 of kfold 5
accuracy_score 0.7040816326530612

3 of kfold 5
accuracy_score 0.673469387755102

4 of kfold 5
accuracy_score 0.7040816326530612

5 of kfold 5
accuracy_score 0.7142857142857143
```

Hyper Parameter Tuning

The model parameters are tuned to gain the best result from the model. This is done by using a technique called Grid Search. In this case, the hyperparameter being tuned for the Decision Tree model is max_depth, which represents the maximum depth of the tree. A range of values from 3 to 10 is specified for max_depth. The value that results in the highest accuracy score is selected.

In [232]:

```python
# Hyperparameters the DecisionTreeClassifier
param_grid = {'max_depth': np.arange(3, 10)}

# Use GridSearchCV to find the best parameters
grid_tree = GridSearchCV(DecisionTreeClassifier(), param_grid, cv=5, scoring='accuracy')
grid_tree.fit(X_train, y_train)

# Best parameters for DecisionTreeClassifier
print(f"Best parameters for DecisionTreeClassifier: {grid_tree.best_params_}")
print(f"Best score for DecisionTreeClassifier: {grid_tree.best_score_}")
```

```
Best parameters for DecisionTreeClassifier: {'max_depth': 3}
Best score for DecisionTreeClassifier: 0.8190847127555989
```

Model Predictions

In [233]:

```python
# Preprocess the test data in the same way as the training data
New_Features_test_copy_processed = pd.get_dummies(New_Features_test_copy.drop('Loan_Stat
us', axis=1))

# Make sure the processed test data has the same columns as the training data
New_Features_test_copy_processed = New_Features_test_copy_processed.reindex(columns = X.
columns, fill_value=0)

# Now you can make predictions on the processed test data
pred_test=model.predict(New_Features_test_copy_processed)

# Calculate the mean validation accuracy score
mean_score = np.mean(scores)
print(f"\nMean validation accuracy score: {mean_score}")
```

```
Mean validation accuracy score: 0.7147392290249434
```

Insight Gained:

- **The Decision Tree model's accuracy scores for the five folds were approximately 0.75, 0.69, 0.68, 0.71, and 0.68 with a mean accuracy score of 70.45%, suggesting the model's performance varied across different subsets of the data.**
- **After performing hyperparameter tuning on the Decision Tree model, the best max_depth parameter was found to be 3. This means by limiting the tree depth to 3 levels resulted in the best performance on the training data according to the accuracy score of 82.17%.**
- **This score is higher than the accuracy scores obtained before tuning, suggesting that the hyperparameter**

tuning improved the model’s performance.

Random Forest Model

The next algorithm being used is called Random Forest. THis model builds onto the decision tree model.

In [234]:

```
i=1
scores = []
kf=StratifiedKFold(n_splits=5,random_state=1,shuffle=True)
for train_index,test_index in kf.split(X,y):
    print('\n{} of kfold {}'.format(i,kf.n_splits))
    xtr,xvl=X.loc[train_index],X.loc[test_index]
    ytr,yvl=y[train_index],y[test_index]
    model=RandomForestClassifier(random_state=1,max_depth=10)
    model.fit(xtr,ytr)
    pred_test=model.predict(xvl)
    score=accuracy_score(yvl,pred_test)
    scores.append(score)
    print('accuracy_score',score)
    i+=1

# Calculate the mean validation accuracy score
mean_score = np.mean(scores)
print(f"\nMean validation accuracy score: {mean_score}")
```

```
1 of kfold 5
accuracy_score 0.8484848484848485

2 of kfold 5
accuracy_score 0.7448979591836735

3 of kfold 5
accuracy_score 0.8163265306122449

4 of kfold 5
accuracy_score 0.7755102040816326

5 of kfold 5
accuracy_score 0.826530612244898

Mean validation accuracy score: 0.8023500309214595
```

Hyper Parameter Tuning

In [235]:

```
# Provide range for max_depth from 1 to 20 with an interval of 2 and from 1 to 200 with a
n interval of 20 for n_estimators
paramgrid = {'max_depth': list(range(1, 20, 2)), 'n_estimators': list(range(1, 200, 20))
}
grid_search = GridSearchCV(RandomForestClassifier(random_state=1), paramgrid)

# Fit the grid search model
grid_search.fit(X_train, y_train)

# Estimating the optimized value
print(f"Best parameters for RandomForestClassifier: {grid_search.best_estimator_}")
```

```
Best parameters for RandomForestClassifier: RandomForestClassifier(max_depth=5, n_estimat
ors=101, random_state=1)
```

Rebuild the Model

In [236]:

```
scores = []
```

```
i=1
kf = StratifiedKFold(n_splits=5, random_state=1, shuffle=True)
for train_index, test_index in kf.split(X, y):
    print('\n{} of kfold {}'.format(i, kf.n_splits))
    xtr, xvl = X.loc[train_index], X.loc[test_index]
    ytr, yvl = y[train_index], y[test_index]

    model = grid_search.best_estimator_
    model.fit(xtr, ytr)
    pred_test = model.predict(xvl)
    score = accuracy_score(yvl, pred_test)
    scores.append(score)
    print('accuracy_score', score)
    i += 1

# Calculate the mean validation accuracy score
mean_score = np.mean(scores)
print(f"\nMean validation accuracy score: {mean_score}")
```

```
1 of kfold 5
accuracy_score 0.8585858585858586

2 of kfold 5
accuracy_score 0.7857142857142857

3 of kfold 5
accuracy_score 0.7959183673469388

4 of kfold 5
accuracy_score 0.7755102040816326

5 of kfold 5
accuracy_score 0.826530612244898

Mean validation accuracy score: 0.8084518655947226
```

Model Predictions

In [237]:

```
# Preprocess the test data in the same way as the training data
New_Features_test_copy_processed = pd.get_dummies(New_Features_test_copy.drop('Loan_Stat
us', axis=1))

# Make sure the processed test data has the same columns as the training data
New_Features_test_copy_processed = New_Features_test_copy_processed.reindex(columns = X.
columns, fill_value=0)

# Now you can make predictions on the processed test data
pred_test=model.predict(New_Features_test_copy_processed)

#Save predictions to CSV file
raw_data = pd.read_csv('raw_data.csv')
submission = pd.DataFrame()

# Fill 'Loan_Status' with predictions
submission['Loan_Status'] = pred_test

# Fill 'Loan_ID' with test Loan_ID and Replace 0 and 1 with 'N' and 'Y'
submission['Loan_ID'] = raw_data['Loan_ID']
submission['Loan_Status'].replace(0, 'N', inplace=True)
submission['Loan_Status'].replace(1, 'Y', inplace=True)

submission.to_csv('Random_Forest.csv', index=False)
```

Insight Gained:

- **The Random Forest model's accuracy scores for the five folds were approximately 0.85, 0.74, 0.81, 0.77, and 0.84 with a mean score of 80.03%. This suggests that the model's performance also varied across different subsets of the data.**

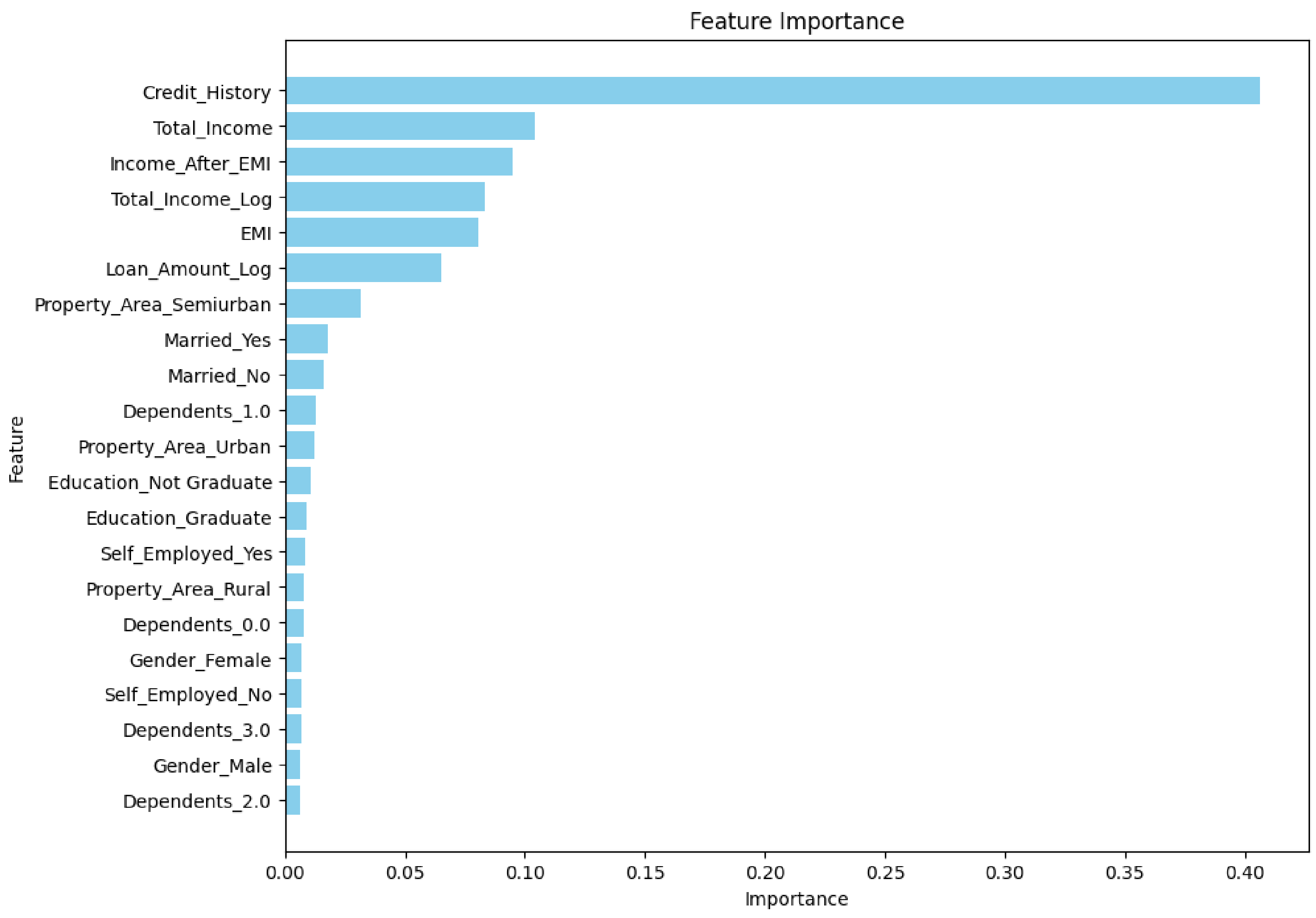
- After performing hyperparameter tuning on the Random Forest model, the best max_depth parameter was found to be 3 and the best n_estimators was 61, meaning the limit of the tree depth was to 3 levels and used 61 trees in the forest. This resulted in the best performance on the training data with an increase in the accuracy score at 81.25%.

Model Feature Importance

Having concluded the cross validation on the 2nd model, the important features can be determined.

In [238]:

```
# Convert the importances into a pandas DataFrame
importances = model.feature_importances_
feature_importances = pd.DataFrame({'Feature': X.columns, 'Importance': importances})
feature_importances = feature_importances.sort_values('Importance', ascending=False)

# Plot the feature importances
plt.figure(figsize=(10, 8))
plt.barh(feature_importances['Feature'], feature_importances['Importance'], color='skyblu
e')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance')
plt.gca().invert_yaxis()
plt.show()
```



Insight Gained:

- The feature ‘Credit_History’ has the highest importance score, indicating it is the most influential factor in the model’s predictions. Its importance score is above 0.4.
- The next important feature followed by the features:
  - ‘Income_After_EMI’ with the importance score of 0.14
  - 'Total_Income_Log’ with the importance score of 0.10
  - 'Total_Income’ and 'EMI' with the importance score of 0.07

E. Create Pickle File

In [239]:

```
# Save Model 2 as a pickle file
with open('Model_2.pkl', 'wb') as f:
    pickle.dump(model2, f)
```

7. Validation Model 2

Before we can use the validation dataset, it needs to be processed to fit the data required by Model 2.

In [240]:

```
# Read Cleaned CSV Files
Validation_data = pd.read_csv('cleaned_validation_data.csv')
Validation_data_copy = Validation_data.copy()

# Convert categorical variable in the X dataset(all columns except 'Loan_Status') into du
mmy variables
Validation_data_copy = pd.get_dummies(Validation_data_copy)


# Feature 1: Total Income
Validation_data_copy['Total_Income']=Validation_data_copy['Applicant_Income']+Validation_
data_copy['Coapplicant_Income']

#Distribution normalization
Validation_data_copy['Total_Income_Log']=np.log(Validation_data_copy['Total_Income'])


# Feature 2: Equated Monthly Installment (EMI)
Validation_data_copy['EMI']=Validation_data_copy['Loan_Amount']/Validation_data_copy['Loa
n_Amount_Term']

# Feature 3: Income_After_EMI
Validation_data_copy['Income_After_EMI']=Validation_data_copy['Total_Income']-(Validation
_data_copy['EMI']*1000)


# Remove all features that created the new features
Validation_data_copy=Validation_data_copy.drop(['Applicant_Income','Coapplicant_Income','
Loan_Amount','Loan_Amount_Term'],axis=1)

print(f"Validation Data Columns: {Validation_data_copy.columns}\n")

# Store new Features in CSV files
Validation_data_copy.to_csv('Feature_Importance_validation_data.csv', index=False)
```

```
Validation Data Columns: Index(['Credit_History', 'Loan_Amount_Log', 'Gender_Female', 'Ge
nder_Male',
       'Married_No', 'Married_Yes', 'Dependents_0', 'Dependents_1',
       'Dependents_2', 'Dependents_3+', 'Education_Graduate',
       'Education_Not Graduate', 'Self_Employed_No', 'Self_Employed_Yes',
       'Property_Area_Rural', 'Property_Area_Semiurban', 'Property_Area_Urban',
       'Total_Income', 'Total_Income_Log', 'EMI', 'Income_After_EMI'],
      dtype='object')
```

Using validate.csv file for final test of Model 2:

- **This will test the practical implemenation of the model**
- **Before we can test the model, we need to save it as a Hierarchical Data Format version 5 (HDF5) file format**

In [241]:

```
from tensorflow.keras.models import save_model

# Save Model 2 to an h5 file
save_model(model2, 'Model2.h5')
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.savin
g.save_model(model)`. This file format is considered legacy. We recommend using instead t
he native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(m
odel, 'my_model.keras')`.
```

In [242]:

```
from keras.models import load_model
import pandas as pd

# Load the trained model
model = load_model('Model2.h5')

# Load the validation data
validate_data = pd.read_csv('Feature_Importance_validation_data.csv')

# Preprocess categorical variables (one-hot encoding)
validate_data = pd.get_dummies(validate_data, columns=['Gender', 'Married', 'Education',
'Self_Employed', 'Property_Area'])

# Drop the Loan_ID column
X_validate = validate_data.drop(columns=['Loan_ID'])

# Use the model to make predictions on the validation data
predictions = model.predict(X_validate)

# Convert probability predictions to binary predictions (0 or 1)
binary_predictions = (predictions > 0.5).astype(int)

# Add the binary predictions as a new column to the validation dataframe
validate_data['Loan_Status'] = binary_predictions

# Save the validation dataframe with the predicted target variable
validate_data.to_csv('validate_with_predictions.csv', index=False)
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `m
odel.compile_metrics` will be empty until you train or evaluate the model.
```

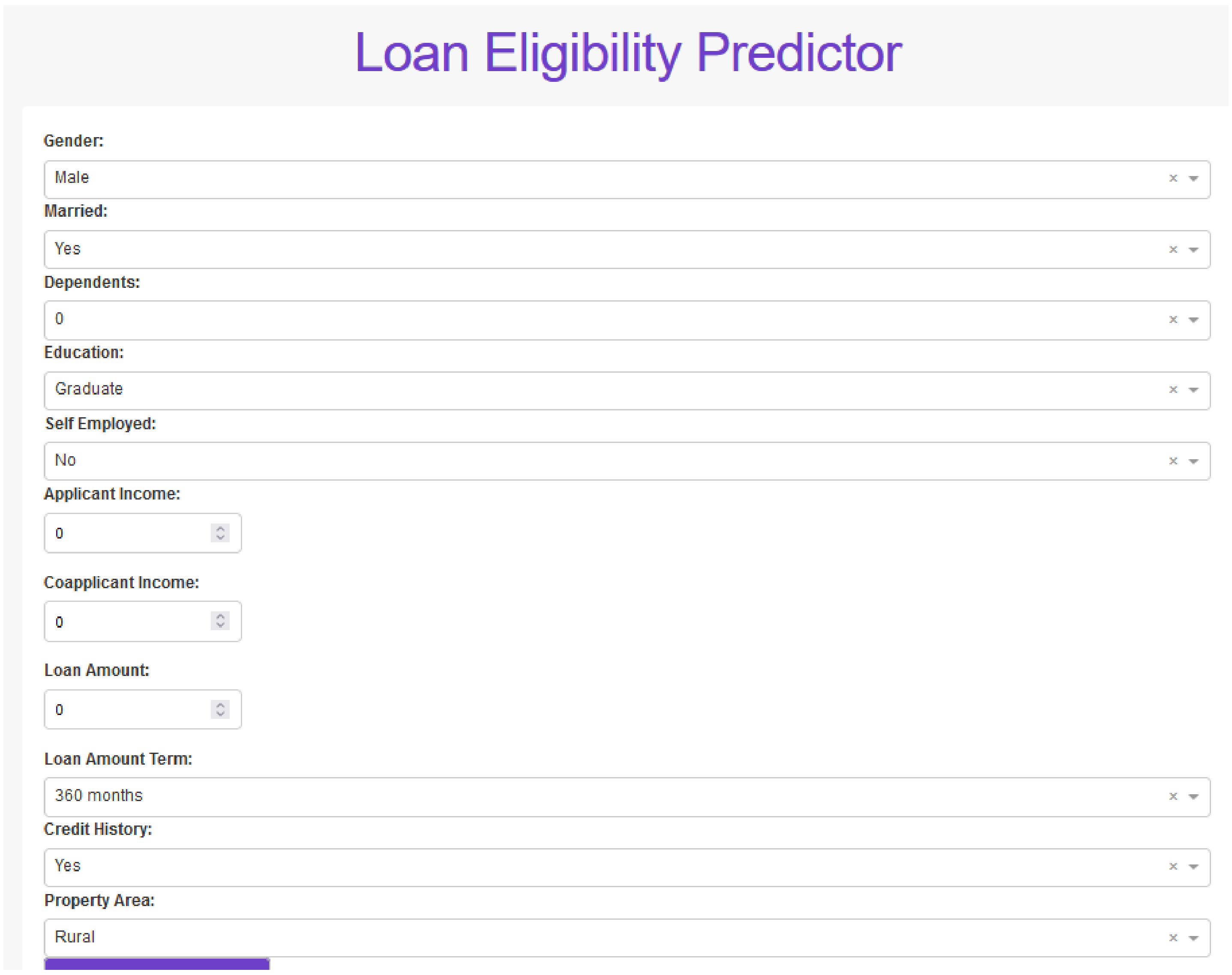
```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[242], line 11
      8 validate_data = pd.read_csv('Feature_Importance_validation_data.csv')
     10 # Preprocess categorical variables (one-hot encoding)
---> 11 validate_data = pd.get_dummies(validate_data, columns=['Gender', 'Married', 'Edu
cation', 'Self_Employed', 'Property_Area'])
     13 # Drop the Loan_ID column
     14 X_validate = validate_data.drop(columns=['Loan_ID'])

File c:\Users\zoetr\Python\Lib\site-packages\pandas\core\reshape\encoding.py:169, in get_
dummies(data, prefix, prefix_sep, dummy_na, columns, sparse, drop_first, dtype)
    167     raise TypeError("Input must be a list-like for parameter `columns`")
    168 else:
--> 169     data_to_encode = data[columns]
    171 # validate prefixes and separator to avoid silently dropping cols
    172 def check_len(item, name: str):

File c:\Users\zoetr\Python\Lib\site-packages\pandas\core\frame.py:4108, in DataFrame.__ge
titem__(self, key)
   4106     if is_iterator(key):
   4107         key = list(key)
-> 4108     indexer = self.columns._get_indexer_strict(key, "columns")[1]
   4110 # take() does not accept boolean indexers
   4111 if getattr(indexer, "dtype", None) == bool:

File c:\Users\zoetr\Python\Lib\site-packages\pandas\core\indexes\base.py:6200, in Index._
get_indexer_strict(self, key, axis_name)
   6197 else:
```

```
   6198     keyarr, indexer, new_indexer = self._reindex_non_unique(keyarr)
-> 6200 self._raise_if_missing(keyarr, indexer, axis_name)
   6202 keyarr = self.take(indexer)
   6203 if isinstance(key, Index):
   6204     # GH 42790 - Preserve name from an Index

File c:\Users\zoetr\Python\Lib\site-packages\pandas\core\indexes\base.py:6249, in Index._
raise_if_missing(self, key, indexer, axis_name)
   6247 if nmissing:
   6248     if nmissing == len(indexer):
-> 6249         raise KeyError(f"None of [{key}] are in the [{axis_name}]")
   6251     not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())
   6252     raise KeyError(f"{not_found} not in index")

KeyError: "None of [Index(['Gender', 'Married', 'Education', 'Self_Employed', 'Property_A
rea'], dtype='object')] are in the [columns]"
```

8. Web Application

In [244]:

```python
#Libraries
import dash
import numpy as np
from dash import dcc, html
from dash.dependencies import Input, Output
import pandas as pd
import pickle

# Initialise the Dash App
app = dash.Dash(__name__)

# Load the model
with open('Model_2.pkl', 'rb') as f:
    model = pickle.load(f)

# Define CSS styles
external_stylesheets = ['https://cdnjs.cloudflare.com/ajax/libs/skeleton/2.0.4/skeleton.m
in.css']
colors = {
    'background': '#f7f7f7',  # Light gray background
    'text': '#333333',  # Dark gray text
    'accent': '#6e40c9',  # Purple accent color
}
app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

# Define App Layout
app.layout = html.Div(style={'backgroundColor': colors['background'], 'padding': '30px'}
, children=[
    html.H1("Loan Eligibility Predictor", style={'textAlign': 'center', 'color': colors[
'accent']}),
    html.Div(children=[
        html.Label("Gender:", style={'color': colors['text']}),
        dcc.Dropdown(
            id='gender',
            options=[
                {'label': 'Male', 'value': 1},
                {'label': 'Female', 'value': 0}
            ],
            value=1
        ),
        html.Label("Married:", style={'color': colors['text']}),
        dcc.Dropdown(
            id='married',
            options=[
                {'label': 'Yes', 'value': 1},
                {'label': 'No', 'value': 0}
            ],
            value=1
        ),
        html.Label("Dependents:", style={'color': colors['text']}),
        dcc.Dropdown(
            id='dependents',
            options=[
                {'label': '0', 'value': 0},
                {'label': '1', 'value': 1},
                {'label': '2', 'value': 2},
                {'label': '3+', 'value': 3}
            ],
            value=0
        ),
        html.Label("Education:", style={'color': colors['text']}),
        dcc.Dropdown(
            id='education',
            options=[
                {'label': 'Graduate', 'value': 'Graduate'},
                {'label': 'Non Graduate', 'value': 'Non Graduate'}
            ],
            value='Graduate'
        ),
        html.Label("Self Employed:", style={'color': colors['text']}),
        dcc.Dropdown(
            id='self_employed',
            options=[
                {'label': 'Yes', 'value': 1},
                {'label': 'No', 'value': 0}
            ],
            value=0
        ),
        html.Label("Applicant Income:", style={'color': colors['text']}),
        dcc.Input(id='applicant_income', type='number', value=0),
        html.Label("Coapplicant Income:", style={'color': colors['text']}),
        dcc.Input(id='coapplicant_income', type='number', value=0),
        html.Label("Loan Amount:", style={'color': colors['text']}),
        dcc.Input(id='loan_amount', type='number', value=0),
        html.Label("Loan Amount Term:", style={'color': colors['text']}),
        dcc.Dropdown(
            id='loan_amount_term',
            options=[
                {'label': '12 months', 'value': 12},
                {'label': '36 months', 'value': 36},
                {'label': '60 months', 'value': 60},
                {'label': '84 months', 'value': 84},
                {'label': '120 months', 'value': 120},
                {'label': '180 months', 'value': 180},
                {'label': '240 months', 'value': 240},
                {'label': '300 months', 'value': 300},
                {'label': '360 months', 'value': 360},
                {'label': '480 months', 'value': 480}
            ],
            value=360
        ),
        html.Label("Credit History:", style={'color': colors['text']}),
        dcc.Dropdown(
            id='credit_history',
            options=[
                {'label': 'Yes', 'value': 1},
                {'label': 'No', 'value': 0}
            ],
            value=1
        ),
        html.Label("Property Area:", style={'color': colors['text']}),
        dcc.Dropdown(
            id='property_area',
            options=[
                {'label': 'Rural', 'value': 'Rural'},
                {'label': 'Semiurban', 'value': 'Semiurban'},
                {'label': 'Urban', 'value': 'Urban'}
            ],
            value='Rural'
        ),
        html.Button('Check Eligibility', id='submit-val', n_clicks=0, style={'background
Color': colors['accent'], 'color': 'white'}),
        html.Div(id='output', style={'marginTop': '20px', 'fontWeight': 'bold'})
    ], style={'marginBottom': '20px', 'padding': '20px', 'borderRadius': '5px', 'backgro
undColor': '#ffffff'})
])

# Define Callback Function for Predictions
@app.callback(
    Output('output', 'children'),
    Input('submit-val', 'n_clicks'),
    [Input('gender', 'value'),
        Input('married', 'value'),
        Input('dependents', 'value'),
        Input('education', 'value'),
        Input('self_employed', 'value'),
        Input('applicant_income', 'value'),
        Input('coapplicant_income', 'value'),
        Input('loan_amount', 'value'),
        Input('loan_amount_term', 'value'),
        Input('credit_history', 'value'),
        Input('property_area', 'value')]
)
def update_output(n_clicks, gender, married, dependents, education, self_employed,
                  applicant_income, coapplicant_income, loan_amount, loan_amount_term,
                  credit_history, property_area):
    if n_clicks > 0:
        # Calculate Total Income
        total_income = int(applicant_income) + int(coapplicant_income)

        # Calculate Log of Total Income
        total_income_log = np.log(total_income)

        # Calculate EMI (Equated Monthly Installment)
        emi = int(loan_amount) / int(loan_amount_term)

        # Calculate Income After EMI
        income_after_emi = total_income - (emi * 1000)   # Assuming EMI is in thousands

        # Prepare input data for prediction
        input_data = pd.DataFrame({
            'Gender': [gender],
            'Married': [married],
            'Dependents': [dependents],
            'Education': [education],
            'Self_Employed': [self_employed],
            'Total_Income': [total_income],
            'Total_Income_Log': [total_income_log],
            'EMI': [emi],
            'Income_After_EMI': [income_after_emi],
            'Credit_History': [credit_history],
            'Property_Area': [property_area]
        })

        # One-hot encode categorical variables
        input_data = pd.get_dummies(input_data)

        # Make predictions
        prediction = model.predict(input_data)
        if prediction[0] == 1:
            return html.Div('Loan Approved', style={'color': 'green'})
        else:
            return html.Div('Loan Rejected', style={'color': 'red'})

# Run the App
if __name__ == '__main__':
    app.run_server(debug=True)
```



Loan Eligibility Predictor

Gender: Male

Married: Yes

Dependents: 0

Education: Graduate

Self Employed: No

Applicant Income: 0

Coapplicant Income: 0

Loan Amount: 0

Loan Amount Term: 360 months

Credit History: Yes

Property Area: Rural

CHECK ELIGIBILITY