Abstract :-

Chapter 1 : Introduction

Introduction of project

Methodology

Data collection

Chapter 2 : Data Exploration

Variable Identification

Numerical Features

Categorical Features

Handling Missing values

Analysis and visualization

Chapter 3 : Conclusion

Abstract : This study examines the status of loans in a diverse dataset spanning multiple demographics and loan types. Leveraging statistical analysis and machine learning techniques, we investigate factors influencing loan performance, including borrower characteristics, loan terms, and economic indicators. Our findings reveal significant predictors of loan default and repayment, shedding light on risk factors and opportunities for improved lending practices. We identify demographic disparities in loan outcomes and assess the effectiveness of credit scoring models in predicting default. Additionally, we explore the impact of macroeconomic trends on loan performance, highlighting the importance of economic conditions in credit risk assessment. Our research contributes to a deeper understanding of loan dynamics and informs strategies for mitigating credit risk and promoting financial inclusion.

Chapter 1 : Introduction

The lending industry plays a pivotal role in facilitating economic activities by providing individuals and businesses with access to capital. However, ensuring the repayment of loans remains a critical challenge for lenders, as default rates can significantly impact financial stability and profitability. In this context, understanding the factors influencing loan status is essential for effective risk management and sustainable lending practices.

Loan status exploration involves analyzing the dynamics of loan performance, including repayment, default, and delinquency, across different borrower profiles, loan types, and economic conditions. By examining historical loan data and employing advanced analytical techniques, researchers can uncover patterns, trends, and predictors of loan outcomes, thus informing decision-making processes and risk mitigation strategies for lenders.

Problem statement

Despite advancements in credit risk assessment and predictive modeling, challenges persist in accurately predicting loan status. Traditional credit scoring models often rely on limited data sources and may fail to capture the complexity of borrower behavior and external factors influencing loan performance. Moreover, demographic disparities, economic fluctuations, and regulatory changes further complicate the task of assessing credit risk and predicting loan outcomes.

Significance of the study

The significance of exploring loan status lies in its implications for lenders, policymakers, and consumers. By gaining insights into the drivers of loan repayment and default, lenders can refine their underwriting criteria, pricing strategies, and collection practices to minimize credit risk and enhance portfolio performance. Policymakers can use the findings to design targeted interventions aimed at promoting financial inclusion, consumer protection, and systemic stability. Additionally, consumers can benefit from a more transparent and equitable lending landscape, with improved access to credit and reduced vulnerability to predatory lending practices.

Methodology : A loan status exploration project typically begins with clearly defining its objectives, whether it's understanding factors affecting loan approval/denial, tracking trends over time, or something else. Data collection follows, acquiring relevant loan application data, including applicant demographics, loan details, and outcomes. Once gathered, data undergoes cleaning and preprocessing to handle missing values, outliers, and inconsistencies, and to prepare it for analysis. Exploratory data analysis (EDA) then takes place, employing descriptive statistics and visualization techniques to understand variable distributions, relationships, and correlations with loan status.

Data collection

To collect data for a loan status exploration from Kaggle's "Loan Status Prediction Dataset," start by locating the dataset on Kaggle. Once found, download the dataset files directly from the platform. Take time to review the dataset's contents and any accompanying documentation to understand its structure, variables, and any potential limitations. After that dependencies were imported through pandas dataframe using "read_csv" function.

Importing the Dependencies

In [1]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import svm
```

Loading dataset

In [19]:
```
df = pd.read_csv('read.csv')
```

Chapter 2 : Data Exploration

Variable Identification

Although we are not going to solve the original Loan Prediction problem here, it is generally helpful to keep the orginal problem in mind while performing the data exploration analysis. Thus the types of variables can be defined as following:

(Possible) Predictor Variable:

- Gender
- Gender
- Dependents
- Education
- Self_Employed
- ApplicantIncome
- CoapplicationIncome
- LoanAmount
- Loan_Amount_Term
- Credit_History
- Property_Area

Target Variable:

- Loan_Status

In [3]:
```
# First 5 rows of the Dataframe
df.head()
```

Out[3]:

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	Coapplicai
0	LP001002	Male	No	0	Graduate	No	5849	
1	LP001003	Male	Yes	1	Graduate	No	4583	
2	LP001005	Male	Yes	0	Graduate	Yes	3000	
3	LP001006	Male	Yes	0	Not Graduate	No	2583	
4	LP001008	Male	No	0	Graduate	No	6000	

Df.head() often used during data analysis to get a quick glimpse of the dataset's structure and content. By default, it shows the first five rows, providing a concise overview of the data.

```
In [5]: # The numbers of rows and columns in the dataset.
        df.shape
```

```
Out[5]: (614, 13)
```

Now, getting non NULL values of the Dataset_Number of values present in the dataset. Datatypes can also be seen as object ,Float etc.

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

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

Section Summary

The dataset can be separated into the following two variable categories:

Categorical features :

- Gender
- Gender
- Education
- Self_Employed

- Loan_Amount_Term
- Credit_History
- Property_Area
- Loan_Status

Continuous features :

- ApplicantIncome
- CoapplicantIncome
- LoanAmount

So now let's check for Missing values as making some more analysis and processing because you cannot feed a dataset into a Machine learning model that have some missing values . The function for finding the missing values would be df.isnull().sum() this will as the missing values of each column .The result is we have 13 missing values in Gender column , 15 missing values in Dependents column , 32 missing values in Self_Employed column , 22 missing values in LoanAmount column , 14 missing values in Loan_Amount_Term column and 50 missing values in Credit_History column.

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

Out[6]: Loan_ID               0
        Gender               13
        Married               3
        Dependents           15
        Education             0
        Self_Employed        32
        ApplicantIncome       0
        CoapplicantIncome     0
        LoanAmount           22
        Loan_Amount_Term     14
        Credit_History       50
        Property_Area         0
        Loan_Status           0
        dtype: int64
```

The amount of Data which is missing is pretty large in this Dataframe so we need to do some processing with this Dataframe The missing values are needed to be handled. So here we needed 2 Metrics which is Mean which gives us the average valueand Mode also which means Most Repeated Value.

Handling Missing Values

Mean -> Averagew value

Mode -> Most repeated value

```
In [20]: df['Gender'].fillna(df['Gender'].mode()[0],inplace=True)
         df.isnull().sum()

Out[20]: Loan_ID              0
         Gender               0
         Married              3
         Dependents          15
         Education            0
         Self_Employed       32
         ApplicantIncome      0
         CoapplicantIncome    0
         LoanAmount          22
         Loan_Amount_Term    14
         Credit_History      50
         Property_Area        0
         Loan_Status          0
         dtype: int64
```

We will find the Mean of the Gender Column and we will convert all the missing values to that particular mean. This method is called as "Imputation" which means impute the missing values in the Gender Column using the Mean Value of of that Column. So now we just need to find the Mean of Gender Column with function "df["Gender"].mean() and fill the missing values in Gender Column with Mean value with function "df['Gender'].fillna(df['Gender'].mode()[0],inplace=True)". This will replace all the missing values by the Mean of the Gender Column present in this df Data by giving "inplace = True" which will convert the missing values by the Mean.

<u>Bibilography</u>

```
In [22]: df['Married'].fillna(df['Married'].mode()[0],inplace=True)
         df.isnull().sum()

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

We will find the Mean of the Married Column and we will convert all the missing values to that particular mean. This method is called as "Imputation" which means impute the missing values in the Married Column using the Mean Value of of that Column. So now we just need to find the Mean of Married Column with function "df["Married"].mean() and fill the missing values in Married Column with Mean value with function "df[Married].fillna(df[Married].mode()[0],inplace=True)". This will replace all the missing values by the Mean of the Married Column present in this df Data by giving "inplace = True" which will convert the missing values by the Mean.

```
In [24]: df['Dependents'].fillna(df['Dependents'].mode()[0],inplace=True)
         df.isnull().sum()

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

We will find the Mean of the Dependents Column and we will convert all the missing values to that particular mean. This method is called as "Imputation" which means impute the missing values in the Dependents Column using the Mean Value of of that Column. So now we just need to find the Mean of Dependents Column with function "df["Dependents"].mean() and fill the missing values in Dependents Column with Mean value with function "df[Dependents].fillna(df[Dependents].mode()[0],inplace=True)". This will replace all the missing values by the Mean of the Dependents Column present in this df Data by giving "inplace = True" which will convert the missing values by the Mean.

```
In [23]: df['Self_Employed'].fillna(df['Self_Employed'].mode()[0],inplace=True)
         df.isnull().sum()
```

```
Out[23]: Loan_ID               0
         Gender                0
         Married               0
         Dependents           15
         Education             0
         Self_Employed         0
         ApplicantIncome       0
         CoapplicantIncome     0
         LoanAmount           22
         Loan_Amount_Term     14
         Credit_History       50
         Property_Area         0
         Loan_Status           0
         dtype: int64
```

We will find the Mean of the Self_Employed Column and we will convert all the missing values to that particular mean. This method is called as "Imputation" which means impute the missing values in the Self_Employed Column using the Mean Value of of that Column. So now we just need to find the Mean of Self_Employed Column with function "df["Self_Employed"].mean() and fill the missing values in Self_Employed Column with Mean value with function "df[Self_Employed].fillna(df[Self_Employed].mode()[0],inplace=True)". This will replace all the missing values by the Mean of the Self_Employed Column present in this df Data by giving "inplace = True" which will convert the missing values by the Mean.

```
In [25]: df.LoanAmount=df.LoanAmount.fillna(df.LoanAmount.mean())
         df.isnull().sum()
```

```
Out[25]: Loan_ID              0
         Gender               0
         Married              0
         Dependents           0
         Education            0
         Self_Employed        0
         ApplicantIncome      0
         CoapplicantIncome    0
         LoanAmount           0
         Loan_Amount_Term    14
         Credit_History      50
         Property_Area        0
         Loan_Status          0
         dtype: int64
```

We will find the Mean of the LoanAmount Column and we will convert all the missing values to that particular mean. This method is called as "Imputation" which means impute the missing values in the LoanAmount Column using the Mean Value of of that Column. So now we just need to find the Mean of LoanAmount Column with function "df["LoanAmount"].mean() and fill the missing values in LoanAmount with Mean value with function "df[LoanAmount].fillna(df[LoanAmount].mode()[0],inplace=True)". This will replace all the missing values by the Mean of the LoanAmount Column present in this df Data by giving "inplace = True" which will convert the missing values by the Mean.

```
In [27]: df['Loan_Amount_Term'].fillna(df['Loan_Amount_Term'].mode()[0],inplace=True)
         df.isnull().sum()

Out[27]: Loan_ID               0
         Gender                0
         Married               0
         Dependents            0
         Education             0
         Self_Employed         0
         ApplicantIncome       0
         CoapplicantIncome     0
         LoanAmount            0
         Loan_Amount_Term      0
         Credit_History       50
         Property_Area         0
         Loan_Status           0
         dtype: int64
```

We will find the Mean of the Loan_Amount_Term Column and we will convert all the missing values to that particular mean. This method is called as "Imputation" which means impute the missing values in the Loan_Amount_Term Column using the Mean Value of of that Column. So now we just need to find the Mean of Loan_Amount_Term Column with function "df["Loan_Amount_Term "].mean() and fill the missing values in Loan_Amount_Term with Mean value with function "df[Loan_Amount_Term].fillna(df[LoanAmount].mode()[0],inplace=True)". This will replace all the missing values by the Mean of the Loan_Amount_Term Column present in this df Data by giving "inplace = True" which will convert the missing values by the Mean.

In [28]:
```
df['Credit_History'].fillna(df['Credit_History'].mode()[0],inplace=True)
df.isnull().sum()
```

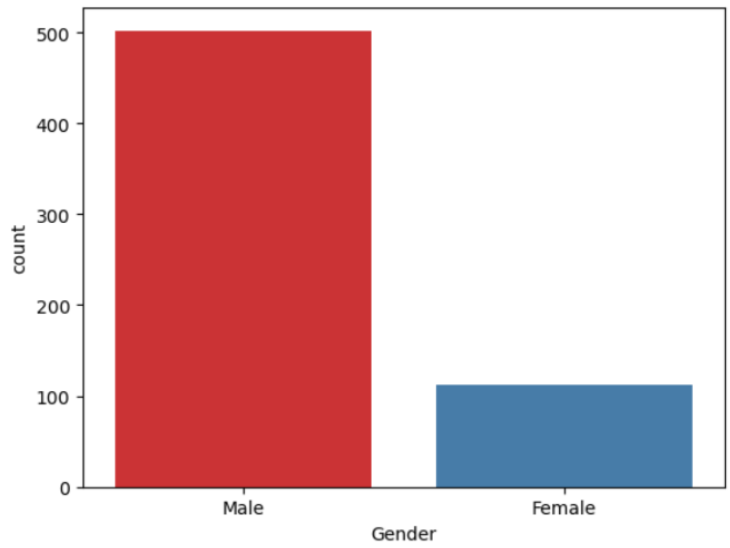
Out[28]:
```
Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
Loan_Status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

We will find the Mean of the Credit_History Column and we will convert all the missing values to that particular mean. This method is called as "Imputation" which means impute the missing values in the Credit_History Column using the Mean Value of of that Column. So now we just need to find the Mean of Credit_History Column with function "df["Credit_History "].mean() and fill the missing values in Credit_History with Mean value with function "df[Credit_History].fillna(df[Credit_History].mode()[0],inplace=True)". This will replace all the missing values by the Mean of the Credit_History Column present in this df Data by giving "inplace = True" which will convert the missing values by the Mean.

In [13]:
```
print("number of people who take loan as group by gender :")
print(df['Gender'].value_counts())
sns.countplot(x='Gender', data=df, palette = 'Set1')
```

```
number of people who take loan as group by gender :
Male      502
Female    112
Name: Gender, dtype: int64
```

Out[13]: <Axes: xlabel='Gender', ylabel='count'>

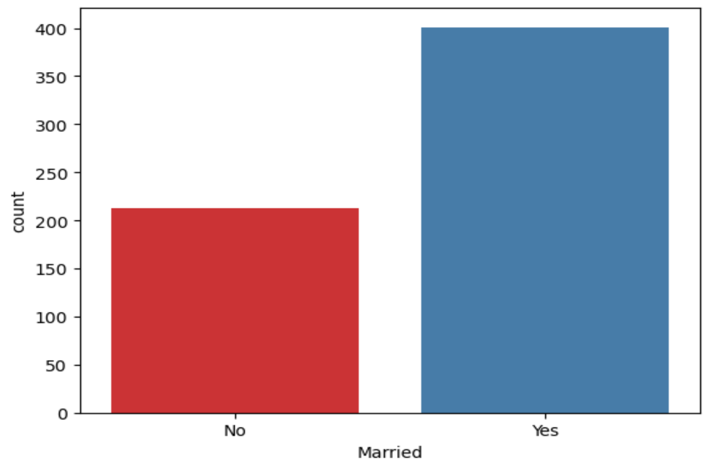


Using this code performs a simple analysis and visualization of loan applicants grouped by gender. The first line prints a descriptive message indicating the upcoming output. The second line calculates and prints the count of loan applicants for each gender category using the value_counts function in pandas, providing a numerical summary of the data. Finally, the third line generates a count plot using seaborn's value_ function, where loan applicants' gender is plotted on the x-axis. The plot visualizes the distribution of loan applicants across gender categories, offering a graphical representation of the data. The 'Set1' palette parameter sets the color scheme for the plot. Overall, this code succinctly presents both numerical and visual summaries of loan applicants by gender.

```
In [14]: print("number of people who take loan as group by marital status :")
         print(df['Married'].value_counts())
         sns.countplot(x='Married', data=df, palette = 'Set1')

number of people who take loan as group by marital status :
Yes    401
No     213
Name: Married, dtype: int64

Out[14]: <Axes: xlabel='Married', ylabel='count'>
```

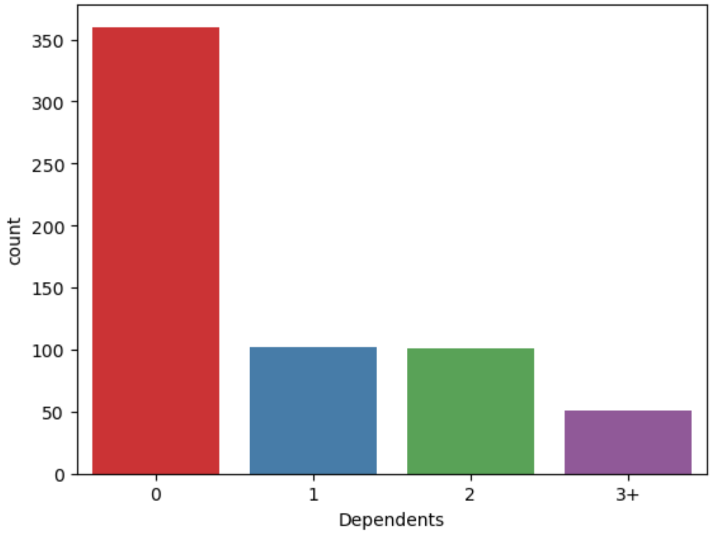


Using this code provides a concise analysis and visualization of loan applicants grouped by marital status. The first line prints a descriptive message indicating the output's context. The second line calculates and prints the count of loan applicants for each marital status category using the value_count function in pandas, offering a numerical summary of the data. Subsequently, the third line generates a count plot using seaborn's countplot() function, where loan applicants' marital status is plotted on the x-axis. This plot visually represents the distribution of loan applicants across marital status categories, allowing for a quick interpretation of the data. The 'Set1' palette parameter sets the color scheme for the plot. Overall, this code succinctly presents both numerical and visual summaries of loan applicants by marital status.

```
In [15]: print("number of people who take loan as group by dependents :")
         print(df['Dependents'].value_counts())
         sns.countplot(x='Dependents', data=df, palette = 'Set1')
```

```
number of people who take loan as group by dependents :
0     360
1     102
2     101
3+     51
Name: Dependents, dtype: int64
```

```
Out[15]: <Axes: xlabel='Dependents', ylabel='count'>
```

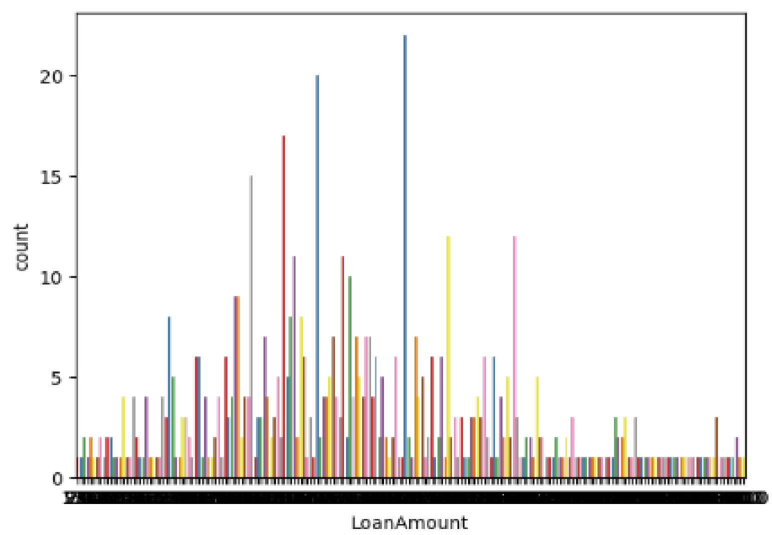


This code aims to provide a brief overview of loan applicants categorized by marital status. The first line presents a descriptive message indicating the analysis's focus. The second line calculates and displays the count of loan applicants for each marital status category using the value_count function in pandas, offering a concise numerical summary. Following this, the third line utilizes seaborn's countplot() function to generate a graphical representation of the distribution of loan applicants across marital status categories. The x-axis of the count plot depicts marital status, facilitating easy interpretation of the data's distribution. Additionally, the 'Set1' palette parameter ensures a visually appealing color scheme for the plot. Overall, this code succinctly provides both numerical insights and a visual representation of loan applicants categorized by marital status.

In [17]:
```
print("number of people who take loan as group by LoanAmount :")
print(df['LoanAmount'].value_counts())
sns.countplot(x='LoanAmount', data=df, palette = 'Set1')
```

```
number of people who take loan as group by LoanAmount :
146.412162    22
120.000000    20
110.000000    17
100.000000    15
160.000000    12
              ..
240.000000     1
214.000000     1
59.000000      1
166.000000     1
253.000000     1
Name: LoanAmount, Length: 204, dtype: int64
```

Out[17]: `<Axes: xlabel='LoanAmount', ylabel='count'>`

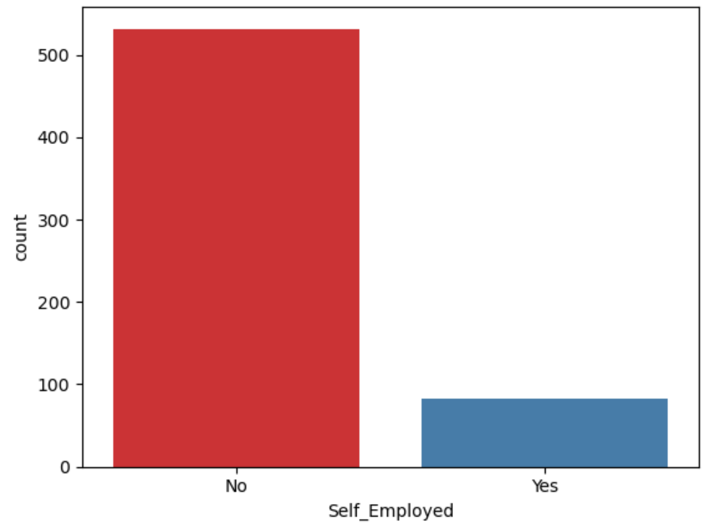


This code intends to offer a concise analysis of loan applicants grouped by loan amount. The first line presents a descriptive message highlighting the focus of the analysis. The second line computes and presents the count of loan applicants for each loan amount category using the value_count() function in pandas, providing a succinct numerical summary. Subsequently, the third line employs seaborn countplot() function to generate a graphical representation of the distribution of loan applicants across loan amount categories. The x-axis of the count plot denotes loan amount, aiding in visualizing the distribution of applicants based on their loan amounts. The 'Set1' palette parameter ensures an aesthetically pleasing color scheme for the plot. Overall, this code succinctly delivers both numerical insights and a visual depiction of loan applicants grouped by loan amount.

In [16]:
```
print("number of people who take loan as group by self_employed :")
print(df['Self_Employed'].value_counts())
sns.countplot(x='Self_Employed', data=df, palette = 'Set1')
```

```
number of people who take loan as group by self_employed :
No     532
Yes     82
Name: Self_Employed, dtype: int64
```

Out[16]: <Axes: xlabel='Self_Employed', ylabel='count'>

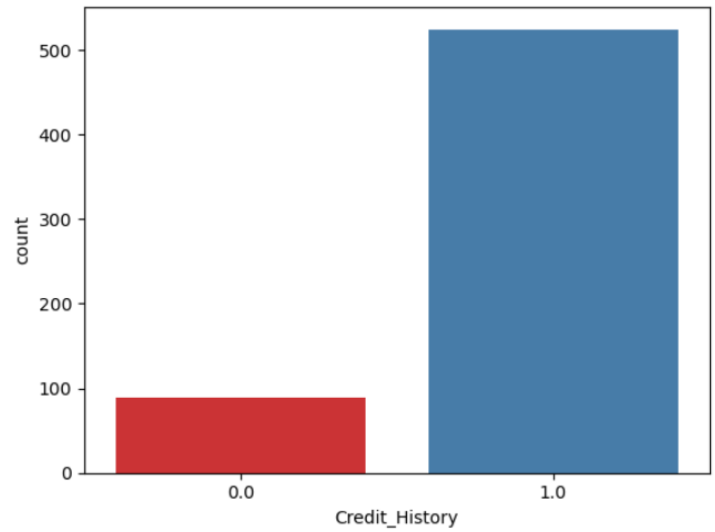


This code aims to provide a brief summary of loan applicants categorized by their self-employment status. The first line presents a descriptive message outlining the focus of the analysis. The second line calculates and displays the count of loan applicants for each self-employment category using the value_count() function in pandas, offering a concise numerical summary. Following this, the third line utilizes seaborn countplot() function to generate a graphical representation of the distribution of loan applicants across self-employment categories. The x-axis of the count plot depicts the self-employment status, facilitating easy visualization of the distribution of loan applicants based on their employment type. The 'Set1' palette parameter ensures a visually appealing color scheme for the plot. Overall, this code succinctly provides both numerical insights and a visual representation of loan applicants grouped by self-employment status.

```
In [18]: print("number of people who take loan as group by Credit history:")
         print(df['Credit_History'].value_counts())
         sns.countplot(x='Credit_History', data=df, palette = 'Set1')

number of people who take loan as group by Credit history:
1.0    525
0.0     89
Name: Credit_History, dtype: int64

Out[18]: <Axes: xlabel='Credit_History', ylabel='count'>
```



This code efficiently summarizes loan applicants grouped by their credit history status. The first line delivers a descriptive message, indicating the focus of the analysis. The second line employs the value_count() function in pandas to compute and present the count of loan applicants for each credit history category, offering a succinct numerical overview. Following this, the third line utilizes seaborn's countplot()function to generate a graphical representation of the distribution of loan applicants across credit history categories. The x-axis of the count plot represents credit history status, enabling a straightforward visualization of the distribution of loan applicants based on their credit history. The 'Set1' palette parameter ensures a visually appealing color scheme for the plot. In summary, this code succinctly provides both numerical insights and a visual representation of loan applicants categorized by credit history status.

Chapter 3 : Conclusion

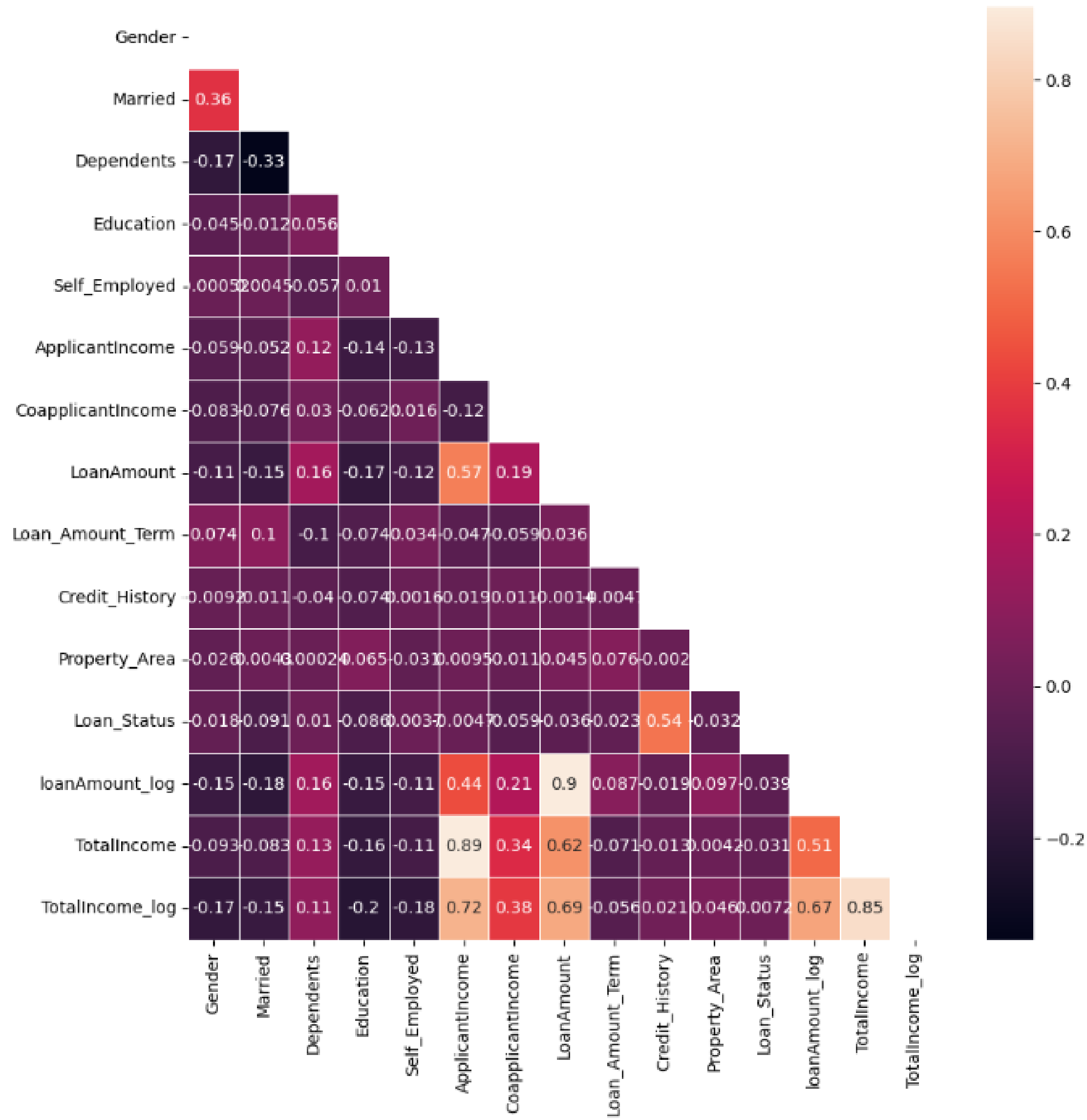
Depending on different methods of missing-value and/or outlier treatments, the level of data loss might have impact on prediction models which are used to solve the original problem. This project is only to explore the dataset and to describe dataset characterizations through data visualization and statistical techniques.

```
In [25]: #encoding to numeric data type
         code_numeric = {'Male':1, 'Female':2,
                        'Yes': 1, 'No':2,
                         'Graduate':1, 'Not Graduate':2,
                         'Urban':1, 'Semiurban':2, 'Rural':3,
                         'Y':1, 'N':0,
                         '3+':3 }
         df = df.applymap(lambda i: code_numeric.get(i) if i in code_numeric else i)
         df['Dependents'] = pd.to_numeric(df.Dependents)
         df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 16 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Loan_ID            614 non-null    object
 1   Gender             614 non-null    int64
 2   Married            614 non-null    int64
 3   Dependents         614 non-null    int64
 4   Education          614 non-null    int64
 5   Self_Employed      614 non-null    int64
 6   ApplicantIncome    614 non-null    int64
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         614 non-null    float64
 9   Loan_Amount_Term   614 non-null    float64
 10  Credit_History     614 non-null    float64
 11  Property_Area      614 non-null    int64
 12  Loan_Status        614 non-null    int64
 13  loanAmount_log     592 non-null    float64
 14  TotalIncome        614 non-null    float64
 15  TotalIncome_log    614 non-null    float64
dtypes: float64(7), int64(8), object(1)
memory usage: 76.9+ KB
```

```
In [26]: matrix = np.triu(df.corr())
         fig, ax = plt.subplots(figsize = (10,10))
         sns.heatmap(df.corr(), annot = True, mask = matrix, linewidths = .5, ax = ax)
```

Out[26]: <Axes: >




```
In [27]: m = matrix[:,11]
         m = pd.DataFrame(m)
         m1 = np.transpose(m)
```

This code segment performs three main operations on a matrix ‘matrix’ .Initially, it extracts the 12th column of the matrix, storing it in a variable m. Subsequently, it converts this extracted column into a Pandas DataFrame, effectively creating a DataFrame with a single column containing the values from the original matrix column. Lastly, it transposes this DataFrame m, swapping its rows and columns. In summary, the code selects a specific column from a matrix, converts it into a DataFrame, and then transposes it, likely as part of a data manipulation or analysis process.

```
In [29]: m1.columns = (df.columns[1:])
         m2 = np.transpose(m1)
         new_col = ['corr_to_Loan_Status']
         m2.columns =new_col
```

This code snippet involves further manipulation of the DataFram m1, obtained earlier. First, it assigns column names to m1 using the column names from another DataFrame df, starting from the second column onward. Next, it transposes m1 , to create a new DataFrame m2, which effectively swaps its rows and columns. Subsequently, it defines a new list named new_col containing a single string element, likely intended as a column name. Finally, it assigns this column name to the columns of m2. In essence, the code sets column names for a transposed DataFrame based on another DataFrame's column names and assigns a new column name to the transposed DataFrame. This likely forms part of a data transformation process, possibly for further analysis or modeling.

Below shows how much each variable correlates with the target variable.

```
In [30]: # sort predictor variables by their correlation strength to the Target variable
         m2['corr_to_Loan_Status'] = m2['corr_to_Loan_Status'].abs()
         m2.sort_values(by = new_col, ascending = False)
```

Out[30]:

	corr_to_Loan_Status
Loan_Status	1.000000
Credit_History	0.540556
Married	0.091478
Education	0.085884
CoapplicantIncome	0.059187
LoanAmount	0.036416
Property_Area	0.032112
Loan_Amount_Term	0.022549
Gender	0.017987
Dependents	0.010118
ApplicantIncome	0.004710
Self_Employed	0.003700
loanAmount_log	0.000000
TotalIncome	0.000000
TotalIncome_log	0.000000

Biboilography :

In order to complete my project , I have taken help from :

Websites:

https://www.google.com

https://www.kaggle.com