SRI LANKA INSTITUTE OF INFORMATION TECHNNOLOGY

MSc in Information Technology

Credit Scoring Approach Using Machine Learning Algorithms

Module: Machine Learning – IT6080 (2023)

Supervisor: Dr. Dharshana Kasthurirathna

Fernando C. S. D.	Windana R. K. C. P.
MS21920004	MS21921476

Contents

1. Introduction 3
2. Dataset 4
   2.1. Exploration & Selection 4
   2.2. Preparation 5
   2.2.1. Feature Selection 5
   2.2.2. Data Cleaning 6
3. Methodology 29
   3.1. Model Selection 29
   3.2. Model Training 30
   3.2.1. Logistic Regression Model 30
   3.2.2. Random Forest Classifier Model 32
4. Results 36
   4.1. Model Evaluation 36
   4.1.1. Logistic Regression Model 36
   4.1.2. Random Forest Classifier Model 37
5. Discussion 37
6. Conclusion 38
7. References 39
8. Appendix 40

1. Introduction

In the contemporary landscape of financial services, the assessment of credit risk plays a pivotal role in determining lending decisions. The traditional credit scoring methods, while effective, often struggle to accommodate the dynamic and intricate patterns present in today's complex financial data. As a response to this challenge, this report presents a comprehensive exploration of credit scoring using machine learning techniques.

The primary objective of this study is to harness the power of machine learning to enhance credit risk assessment accuracy. To achieve this goal, we have employed two distinct learning algorithms, each bringing its unique strengths to the table. By leveraging the capabilities of these algorithms, we aim to develop a more robust and adaptable credit scoring model that can effectively analyze a wide array of borrower attributes and transaction histories.

In the subsequent sections, we will delve into the theoretical foundations of the selected algorithms, detailing their working principles and advantages. We will also provide insights into the feature engineering process and dataset preparation, which are crucial steps in enabling the algorithms to make informed predictions. Furthermore, the report will elaborate on the model training, validation, and evaluation procedures used to ensure the reliability and performance of our credit scoring solutions.

Ultimately, the culmination of this research will offer a comparative analysis of the two chosen learning algorithms, highlighting their respective efficacy in credit risk prediction. By juxtaposing their outcomes and performance metrics, we aim to provide financial institutions with a comprehensive understanding of the potential benefits and trade-offs associated with each algorithm. This report strives to contribute valuable insights to the ongoing discourse on leveraging machine learning to transform credit scoring practices, ushering in a new era of more accurate and adaptive lending decisions.

2. Dataset

2.1. Exploration & Selection

In our effort to address the complex issues raised by the credit scoring problem, we carefully analyzed three different datasets, each of which was rife with potentially insightful information. We discovered that one dataset stood out as the best option after carefully evaluating its characteristics, quality, and usefulness. This dataset was found on Kaggle, a well-known site for data exploration, analysis, and we ultimately chose it to power our credit scoring method. It demonstrates the qualities and depth necessary to understand the complexities of credit assessment, paving the path for a solid and trustworthy scoring model.

The 3 datasets found on Kaggle are,

I. Credit score classification (kaggle.com) [1]
II. Creditability - German Credit Data (kaggle.com) [2]
III. Default of Credit Card Clients Dataset (kaggle.com) [3]

Among the above three datasets the 1st dataset was selected for the following reasons.

1st Data Set	**2nd Data Set**	**3rd Data Set**
Has 100000 records (data sample is large)	Has only 1000 records (data sample is small)	Has 30000 records.
This dataset has a distinct edge over other public datasets because it contains many diverse data kinds and properties.	Clean data set. Seems somebody cleaned the data set or manipulated one (synthetic data).	It has multiple PAY data columns, but not given a proper explanation. Not even for the other columns as well.
Most recent data set compared to other two datasets. Updated in 2022	Last updated 3 years ago	Last update 7 years ago

2.2. Preparation

Any data-driven project must start with the preparation of a dataset to make sure the data is organized, clean, and ready for analysis. We carefully collected the data in order to create a trustworthy dataset for our project using machine learning for credit scoring. As we wanted to make sure the data was real and reliable, we started the process by obtaining the dataset from a credible and certified source. The dataset was found on Kaggle, a website renowned for hosting high-quality datasets, as was already indicated. Here is a thorough explanation of our data preparation process:

First loading the training data from a CSV file.

```
In [3]: # Specify dtype for column 'Monthly_Balance' as a string (Column 'Monthly_Balance' has mixed data types)
        dtypes = {'Monthly_Balance': str}
        train_df = pd.read_csv('train.csv', dtype = dtypes)
```

2.2.1. Feature Selection

One of the first processes in our data preparation process was feature selection, which helps reduce the amount of unnecessary or redundant characteristics.

To avoid reflecting personal bias in the analysis and to increase regulatory compliance, it is crucial to further remove personally Identifiable Data from the dataset.

Early removal of such elements enables us to deal with a smaller and more comprehensible set of variables. A smaller number of features can also make it easier to interpret and explain the model's predictions. Furthermore, reducing irrelevant or noisy features helps improve model generalization.

Following are the columns that were available in the dataset.

Display all columns

```
In [4]: train_df.columns

Out[4]: Index(['ID', 'Customer_ID', 'Month', 'Name', 'Age', 'SSN', 'Occupation',
               'Annual_Income', 'Monthly_Inhand_Salary', 'Num_Bank_Accounts',
               'Num_Credit_Card', 'Interest_Rate', 'Num_of_Loan', 'Type_of_Loan',
               'Delay_from_due_date', 'Num_of_Delayed_Payment', 'Changed_Credit_Limit',
               'Num_Credit_Inquiries', 'Credit_Mix', 'Outstanding_Debt',
               'Credit_Utilization_Ratio', 'Credit_History_Age',
               'Payment_of_Min_Amount', 'Total_EMI_per_month',
               'Amount_invested_monthly', 'Payment_Behaviour', 'Monthly_Balance',
               'Credit_Score'],
              dtype='object')
```

A specific customer can be identified using the ID, Customer_ID, Name, and SSN fields (PII data). Therefore, we will eliminate these columns from our training data set. Additionally, the 'Month' column does not give our classification any weight as it just carried the month and not mentioned the Year.

Hence, we will proceed with the removal of that specific column as well. Meaning of Num_Credit_Inquiries column is doubtful as it is generated from bank to customer or customer to bank. We will remove the Num_Credit_Inquiries as we are not clear with the data of it.

```
In [5]: #Remove sesitive data columns
        train_df1 = train_df.drop(['ID', 'Customer_ID', 'Name', 'SSN', 'Month', 'Num_Credit_Inquiries'], axis=1)
        train_df1.shape

Out[5]: (100000, 22)
```

2.2.2. Data Cleaning

There are frequently flaws in the dataset, such as missing values, duplicates, or outliers. We started by carefully cleaning the data by addressing these problems. To find and treat outliers, we employed a variety of strategies, including imputation, duplication removal, and statistical methods.

So, we will be removing such data from the dataset as shown below.

- If any column has irregular data such as '!@9#%8', we will replace them with the Null value (np.NaN).
- If any column value start or end with '_', we will replace them with the Null value (np.NaN).
- For any empty ('') or nan (a string value), we will replace them with the Null value (np.NaN).

```
In [6]: train_df1 = train_df1.applymap(
          lambda x: x if x is np.NaN or not \
            isinstance(x, str) else str(x).strip('_')).replace(['', 'nan', '!@9#%8', '#F%$D@*&8'], np.NaN)
        train_df1.shape

Out[6]: (100000, 22)
```

And also, we wanted to check the data types of the columns exist in the dataset and we listed them to get a clear idea.

```
In [7]: train_df1.dtypes

Out[7]: Age                        object
        Occupation                 object
        Annual_Income              object
        Monthly_Inhand_Salary     float64
        Num_Bank_Accounts           int64
        Num_Credit_Card             int64
        Interest_Rate               int64
        Num_of_Loan                object
        Type_of_Loan               object
        Delay_from_due_date         int64
        Num_of_Delayed_Payment     object
        Changed_Credit_Limit       object
        Credit_Mix                 object
        Outstanding_Debt           object
        Credit_Utilization_Ratio  float64
        Credit_History_Age         object
        Payment_of_Min_Amount      object
        Total_EMI_per_month       float64
        Amount_invested_monthly    object
        Payment_Behaviour          object
        Monthly_Balance            object
        Credit_Score               object
        dtype: object
```

After listing the datatypes, we notice that the Type of the Age is an object. But Age column shouldn't be an Object type as it has all numeric values. So, we converted the Age column to Integer Type. Similarly, we changed the datatype of the columns Annual_Income, Num_of_Loan, Changed_Credit_Limit, Outstanding_Debt, Amount_invested_monthly, Monthly_Balance, Num_of_Delayed_Payment to either int or float as required.

```
In [8]: train_df1['Age'] = pd.to_numeric(train_df1['Age'], errors='coerce').astype('int64')
        train_df1['Annual_Income'] = pd.to_numeric(train_df1['Annual_Income'], errors='coerce').astype('float')
        train_df1['Num_of_Loan'] = pd.to_numeric(train_df1['Num_of_Loan'], errors='coerce').astype('int32')
        train_df1['Changed_Credit_Limit'] = pd.to_numeric(train_df1['Changed_Credit_Limit'], errors='coerce').astype('float')
        train_df1['Outstanding_Debt'] = pd.to_numeric(train_df1['Outstanding_Debt'], errors='coerce').astype('float')
        train_df1['Amount_invested_monthly'] = pd.to_numeric(train_df1['Amount_invested_monthly'], errors='coerce').astype('float')
        train_df1['Monthly_Balance'] = pd.to_numeric(train_df1['Monthly_Balance'], errors='coerce').astype('float')
        train_df1['Num_of_Delayed_Payment'] = pd.to_numeric(train_df1['Num_of_Delayed_Payment'], errors='coerce').astype('float')
```

```
In [9]: train_df1.dtypes
```

```
Out[9]: Age                         int64
        Occupation                 object
        Annual_Income             float64
        Monthly_Inhand_Salary     float64
        Num_Bank_Accounts           int64
        Num_Credit_Card             int64
        Interest_Rate               int64
        Num_of_Loan                 int32
        Type_of_Loan               object
        Delay_from_due_date         int64
        Num_of_Delayed_Payment    float64
        Changed_Credit_Limit      float64
        Credit_Mix                 object
        Outstanding_Debt          float64
        Credit_Utilization_Ratio  float64
        Credit_History_Age         object
        Payment_of_Min_Amount      object
        Total_EMI_per_month       float64
        Amount_invested_monthly   float64
        Payment_Behaviour          object
        Monthly_Balance           float64
        Credit_Score               object
        dtype: object
```

I. Missing values

A key component of data preparation is the removal of missing values, which entails the removal of data points or features that lack critical information. As a percentage and as a number value we listed how many missing values are found in the dataset.

```
In [10]: # Calculate the sum of missing values in each column
missing_values_sum = train_df1.isna().sum()

# Calculate the percentage of missing values for each column
total_rows = train_df1.shape[0]
missing_values_percentage = (missing_values_sum / total_rows) * 100

# Get the data type of the column as well
column_datatypes = train_df1.dtypes

# Create a DataFrame to display the results
missing_info = pd.DataFrame({
    'Data Type': column_datatypes,
    'Missing Values': missing_values_sum,
    'Percentage Missing (%)': missing_values_percentage
})

# Display the missing values sum and percentage
print(missing_info)
```

	Data Type	Missing Values	Percentage Missing (%)
Age	int64	0	0.000
Occupation	object	7062	7.062
Annual_Income	float64	0	0.000
Monthly_Inhand_Salary	float64	15002	15.002
Num_Bank_Accounts	int64	0	0.000
Num_Credit_Card	int64	0	0.000
Interest_Rate	int64	0	0.000
Num_of_Loan	int32	0	0.000
Type_of_Loan	object	11408	11.408
Delay_from_due_date	int64	0	0.000
Num_of_Delayed_Payment	float64	7002	7.002
Changed_Credit_Limit	float64	2091	2.091
Credit_Mix	object	20195	20.195
Outstanding_Debt	float64	0	0.000
Credit_Utilization_Ratio	float64	0	0.000
Credit_History_Age	object	9030	9.030
Payment_of_Min_Amount	object	0	0.000
Total_EMI_per_month	float64	0	0.000
Amount_invested_monthly	float64	4479	4.479
Payment_Behaviour	object	7600	7.600
Monthly_Balance	float64	1200	1.200
Credit_Score	object	0	0.000

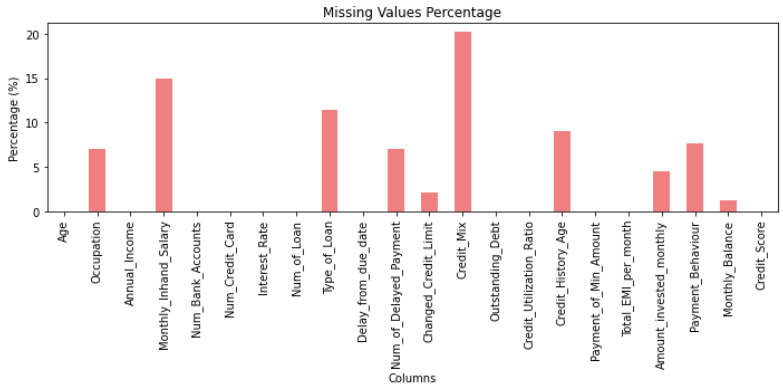
Generate a bar plot to get a visual idea of the missing value percentage against the column.

In [11]:
```
# Create a bar plot for missing values sum
plt.figure(figsize=(10, 5))

missing_values_percentage.plot(kind='bar', color='lightcoral')
plt.title('Missing Values Percentage')
plt.xlabel('Columns')
plt.ylabel('Percentage (%)')

# Adjust spacing between plots
plt.tight_layout()

# Show the plot
plt.show()
```



So as per the analysis Occupation (7%), Monthly_Inhand_Salary (15%), Type_of_Loan (11%), Num_of_Delayed_Payment (7%), Changed_Credit_Limit (2%), Credit_Mix (20%), Credit_History_Age (9%), Amount_invested_monthly (4%), Payment_Behaviour (7%) & Monthly_Balance (1%) are having missing values. We will have to correct them to bring more sense to the analysis. We followed two approaches to handle those missing cases.

- Most Frequent strategy: This strategy used in categorical columns to replace missing values with the value if has highest frequency within that column. We applied this technique for 'Occupation', 'Type_of_Loan', 'Credit_Mix', 'Payment_Behaviour' columns.
- Remove Rows strategy: here in this approach, it will not fix the missing value, instead it will remove the entire row from the data set. We applied this technique to all columns except above mentioned

```
In [12]: # Replace missing values using most_frequent strategy
imputer = SimpleImputer(strategy='most_frequent')
train_df1[['Occupation', 'Type_of_Loan', 'Credit_Mix', 'Payment_Behaviour']] = imputer.fit_transform(train_df1[['Occupation', 'Ty
```

```
In [13]: # Remove rows with missing values
train_df1 = train_df1.dropna(axis=0)
```

To verify we calculated the sum of missing values in each row to make sure that all the data with missing values are removed.

```
In [14]: # Calculate the sum of missing values in each column
missing_values_sum = train_df1.isna().sum()

# Calculate the percentage of missing values for each column
total_rows = train_df1.shape[0]
missing_values_percentage = (missing_values_sum / total_rows) * 100

# Get the data type of the column as well
column_datatypes = train_df1.dtypes

# Create a DataFrame to display the results
missing_info = pd.DataFrame({
    'Data Type': column_datatypes,
    'Missing Values': missing_values_sum,
    'Percentage Missing (%)': missing_values_percentage
})

# Display the missing values sum and percentage
print(missing_info)
```

```
                          Data Type  Missing Values  Percentage Missing (%)
Age                           int64               0                     0.0
Occupation                   object               0                     0.0
Annual_Income               float64               0                     0.0
Monthly_Inhand_Salary       float64               0                     0.0
Num_Bank_Accounts             int64               0                     0.0
Num_Credit_Card               int64               0                     0.0
Interest_Rate                 int64               0                     0.0
Num_of_Loan                   int32               0                     0.0
Type_of_Loan                 object               0                     0.0
Delay_from_due_date           int64               0                     0.0
Num_of_Delayed_Payment      float64               0                     0.0
Changed_Credit_Limit        float64               0                     0.0
Credit_Mix                   object               0                     0.0
Outstanding_Debt            float64               0                     0.0
Credit_Utilization_Ratio    float64               0                     0.0
Credit_History_Age           object               0                     0.0
Payment_of_Min_Amount        object               0                     0.0
Total_EMI_per_month         float64               0                     0.0
Amount_invested_monthly     float64               0                     0.0
Payment_Behaviour            object               0                     0.0
Monthly_Balance             float64               0                     0.0
Credit_Score                 object               0                     0.0
```

I. Outlier removal

The robustness and precision of statistical studies and machine learning models can both be greatly impacted by outliers, or data points that differ dramatically from the bulk of observations. Outliers must be found and eliminated as part of the data preparation process.

We will be investigating one column at a time. Therefore, we defined two functions to generate a box plot for a specific column, to generate a histogram for a specific column.

In [16]:
```
# Define a function to generate a box plot for a specific column
def generate_boxplot(data, column_name):
    plt.figure(figsize=(8, 6))
    sns.boxplot(y=data[column_name])

    # Customize the y-axis tick labels
    plt.gca().get_yaxis().get_major_formatter().set_scientific(False)

    plt.ylabel(column_name)
    plt.title(f'Box Plot of {column_name}')
    plt.show()
```

In [17]:
```
# Define a function to generate a histogram for a specific column
def generate_histogram(data, column_name):
    plt.figure(figsize=(8, 6))
    sns.histplot(data[column_name], bins=20, kde=True)  # Adjust the number of bins as needed

    # Customize the y-axis tick labels
    plt.gca().get_yaxis().get_major_formatter().set_scientific(False)

    plt.xlabel(column_name)
    plt.ylabel('Frequency')
    plt.title(f'Histogram of {column_name}')
    plt.show()
```

We defined another function to see the actual data distribution using a bar plot. This function needs considerable resource to render as we have huge data set. So, it will be used only if required to get any additional insights to the analysis.

In [18]:
```
# Define a function to generate a bar chart for a specific column
def generate_bar_chart(data, column_name):
    data_counts = data[column_name].value_counts().sort_index()

    # Create a bar plot
    plt.figure(figsize=(20, 12))  # Adjust the figure size as needed
    data_counts.plot(kind='bar', rot=0)  # 'rot' parameter controls x-axis label rotation

    # Customize the y-axis tick labels
    plt.gca().get_yaxis().get_major_formatter().set_scientific(False)

    plt.xlabel(column_name)
    plt.ylabel('Count')
    plt.title(f'{column_name} Distribution')
    plt.xticks(rotation=90)
    plt.show()
```

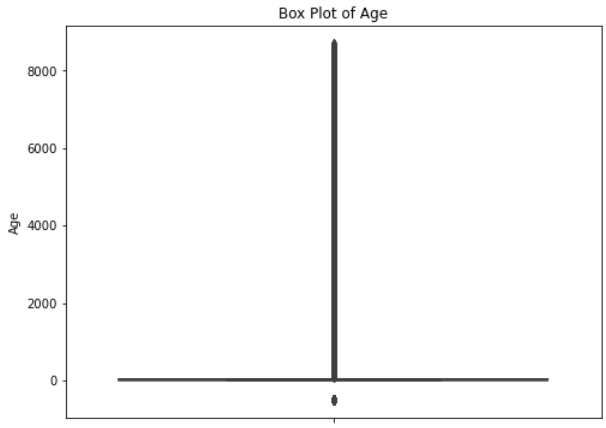
Since we have the above functions we called the above functions specifying one column at a time and then we identified outliers and we excluded those values from the dataset.

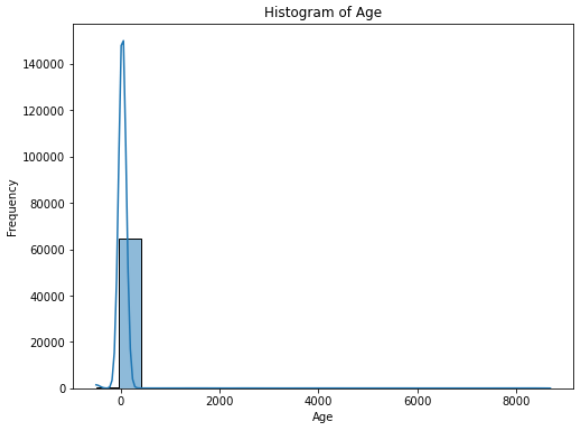
Following are the columns we specified and the actions taken to remove those outliers.

Column: Age

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [19]: # Call the function to generate a box plot for the 'Age' column
         generate_boxplot(train_df1, 'Age')
         generate_histogram(train_df1, 'Age')
         # generate_bar_chart(train_df1, 'Age')
```



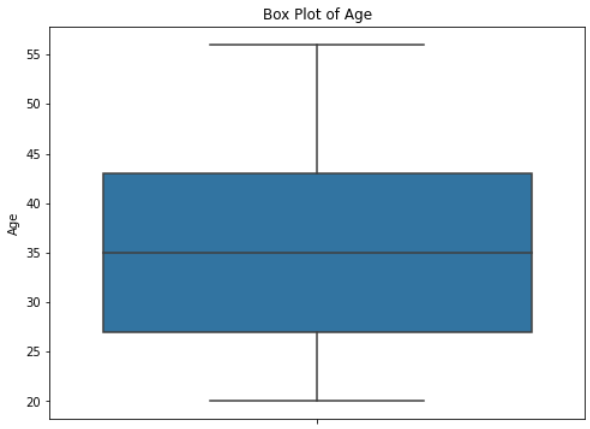


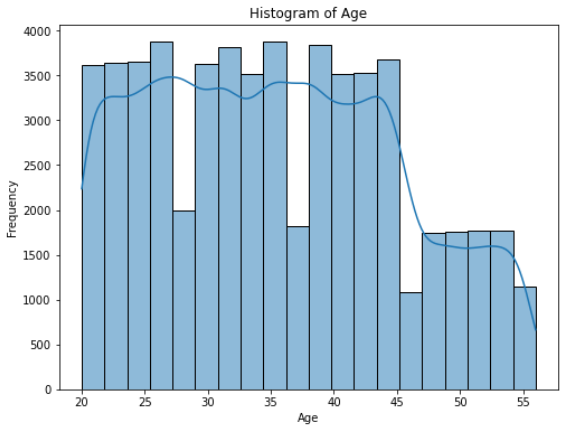
As per the above result, the Age column contains data that doesn't seems to be practical. We must exclude those data considering the optimal age range to be 20 to 60 years old.

```
In [20]: train_df1 = train_df1[(train_df1['Age'] >= 20) & (train_df1['Age'] <= 60)]

         train_df1.shape
Out[20]: (57275, 22)
```

Regenerate the box plot and the histogram after excluding the outliers.

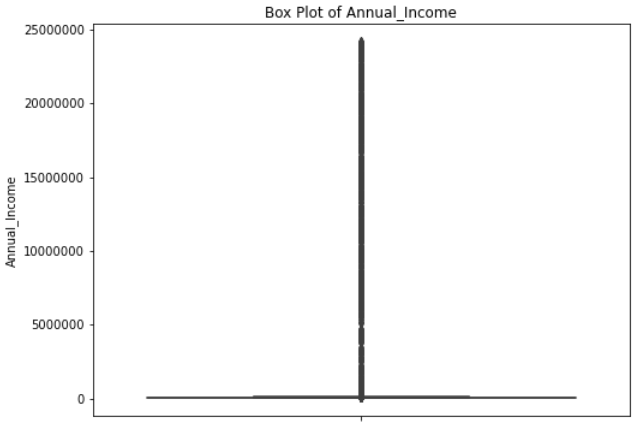


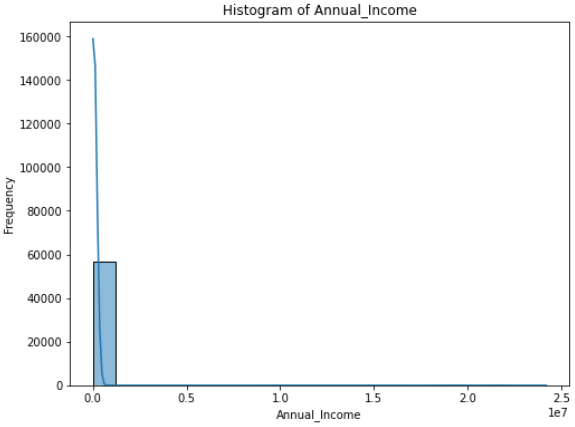


Column: Annual_Income

Generate the box plot and the histogram to get a visual idea about the data distribution.

```
In [22]: # Annual_Income analysis
         generate_boxplot(train_df1, 'Annual_Income')
         generate_histogram(train_df1, 'Annual_Income')
         # generate_bar_chart(train_df1, 'Annual_Income')
```

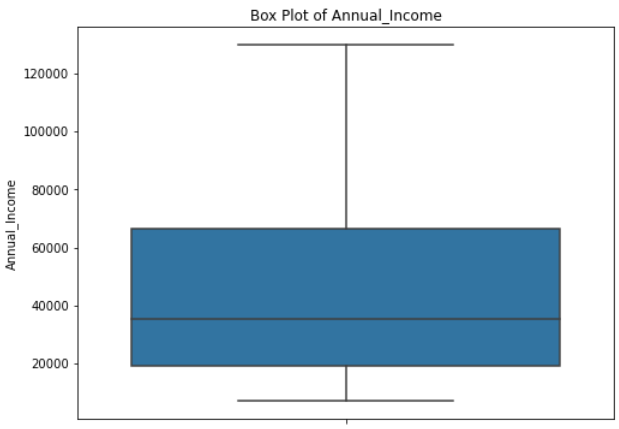


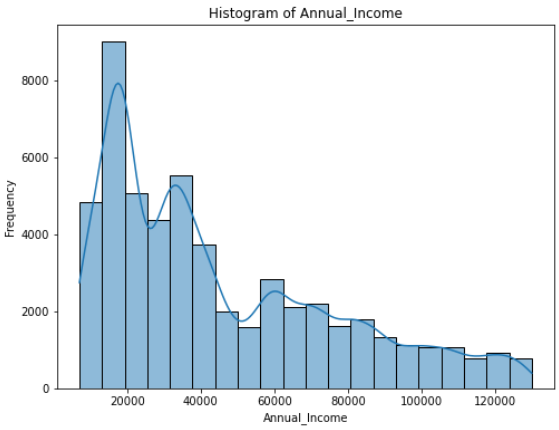


According to the above diagrams the Annual_Income is having some outliers. There are some values having more than 130000 and those data will be excluded to make it more accurate. Anyway, there are no any negatives.

```
In [23]: # Remove values greater that 130000
         train_df1 = train_df1[(train_df1['Annual_Income'] >= 0) & (train_df1['Annual_Income'] <= 130000)]
```

Regenerate the box plot and the histogram diagrams.

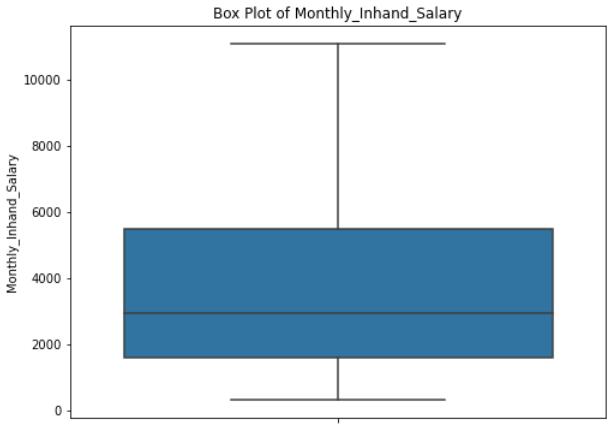


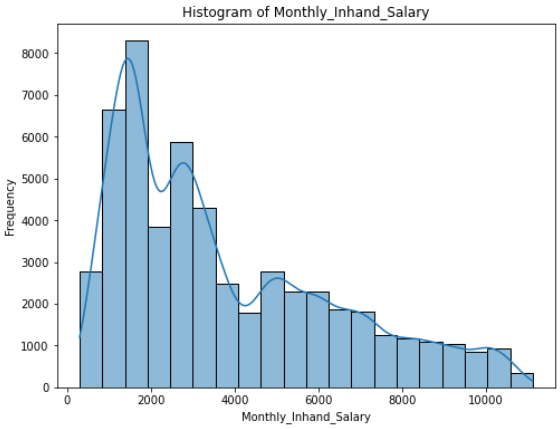


Column: Monthly_Inhand_Salary

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [26]: # Monthly_Inhand_Salary analysis
         generate_boxplot(train_df1, 'Monthly_Inhand_Salary')
         generate_histogram(train_df1, 'Monthly_Inhand_Salary')
         # generate_bar_chart(train_df1, 'Monthly_Inhand_Salary')
```



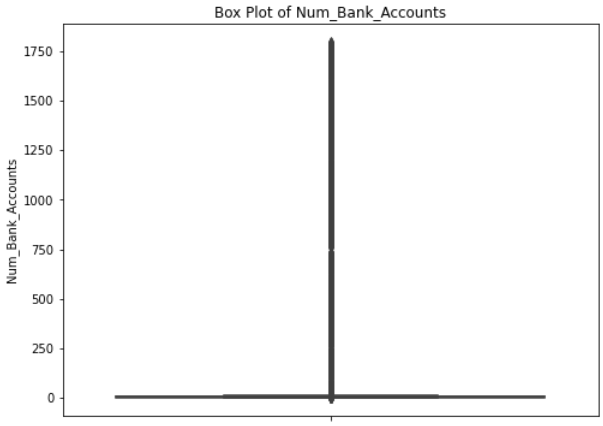


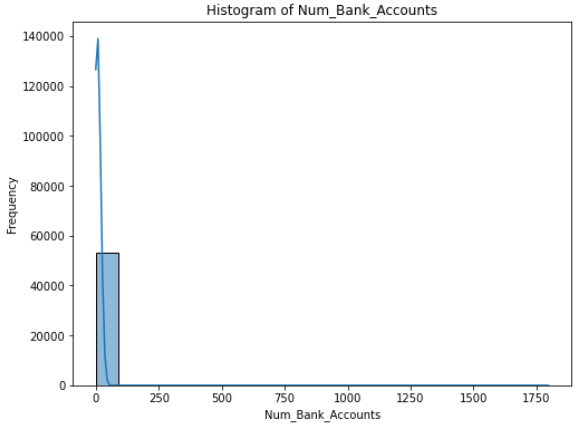
As per the diagrams the Monthly_Inhand_Salary is having a practical data distribution.

Column: Num_Bank_Accounts

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [27]: # Num_Bank_Accounts analysis
         generate_boxplot(train_df1, 'Num_Bank_Accounts')
         generate_histogram(train_df1, 'Num_Bank_Accounts')
         # generate_bar_chart(train_df1, 'Num_Bank_Accounts')
```





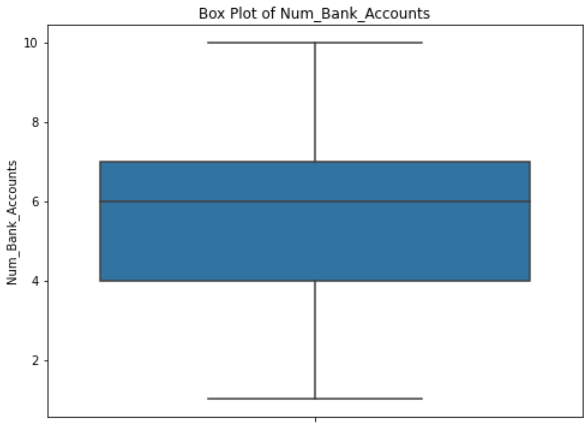
The evidence presented above doesn't seem to be accurate and the outliers should be excluded.

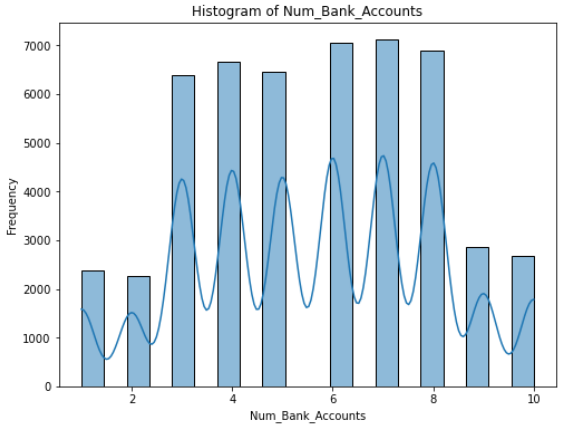
```
In [28]: train_df1 = train_df1[(train_df1['Num_Bank_Accounts'] >= 1) & (train_df1['Num_Bank_Accounts'] <= 10)]

         train_df1.shape

Out[28]: (50717, 22)
```

Regenerate the box plot and the histogram diagrams.

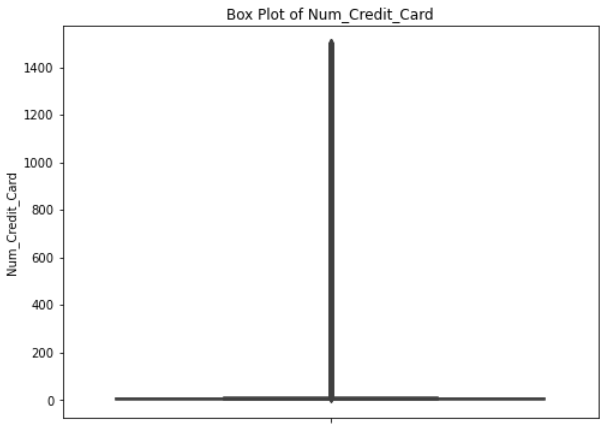


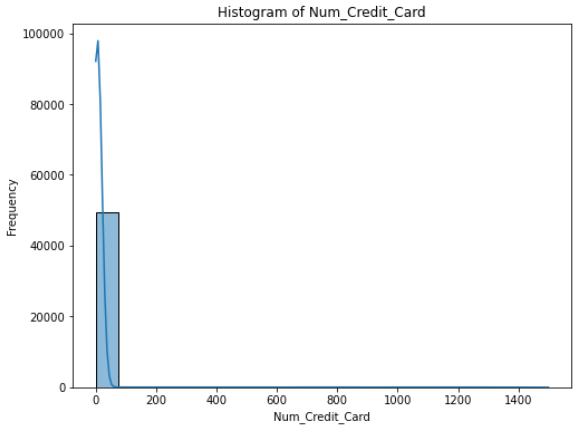


Column: Num_Credit_Card

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [30]: # Num_Credit_Card analysis
         generate_boxplot(train_df1, 'Num_Credit_Card')
         generate_histogram(train_df1, 'Num_Credit_Card')
         # generate_bar_chart(train_df1, 'Num_Credit_Card')
```

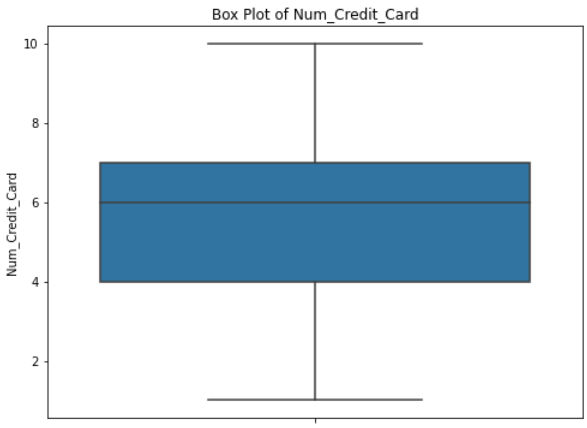


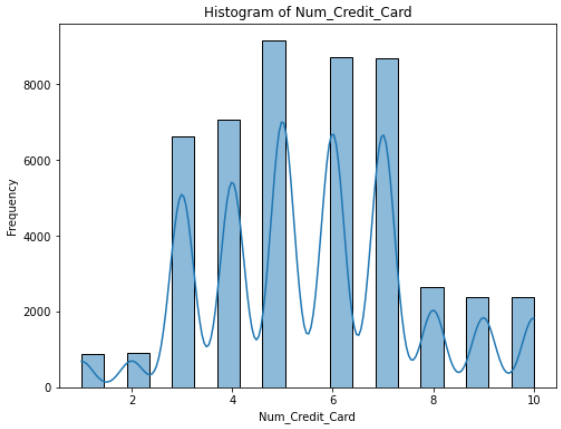


According to the above visual result, it doesn't seem to be showing accurate data. So, the outliers should be excluded.

```
In [31]: train_df1 = train_df1[(train_df1['Num_Credit_Card'] >= 1) & (train_df1['Num_Credit_Card'] <= 10)]
```

Regenerate the diagrams again to see how it reflects now.

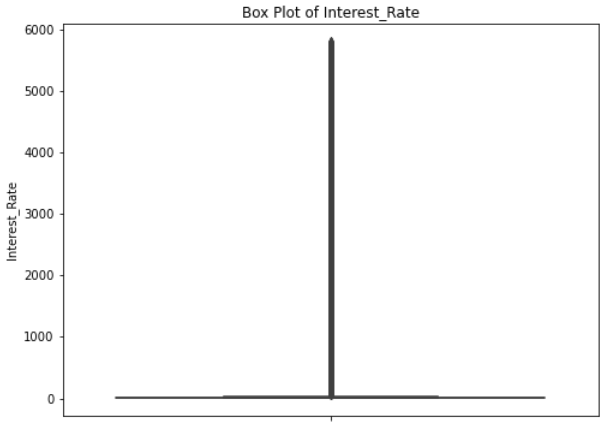


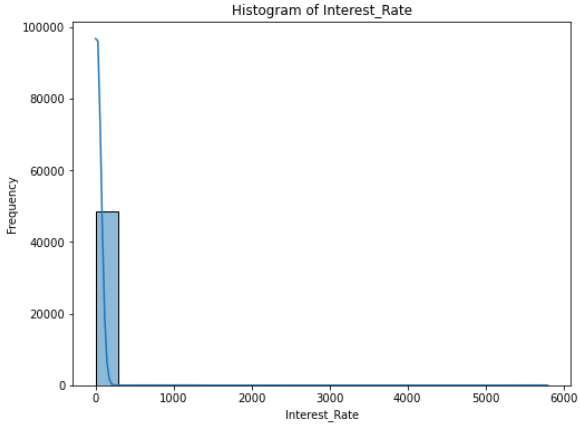


Column: Interest_Rate

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [34]: # Interest_Rate analysis
         generate_boxplot(train_df1, 'Interest_Rate')
         generate_histogram(train_df1, 'Interest_Rate')
         # generate_bar_chart(train_df1, 'Interest_Rate')
```



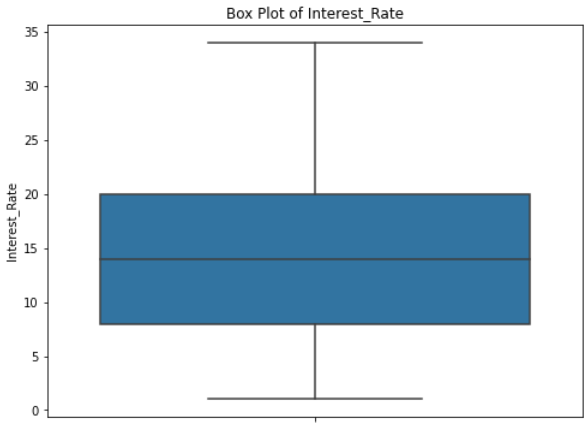


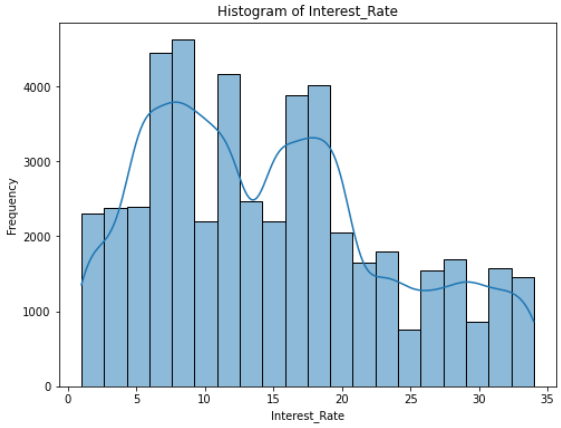
Generally, Interest_Rate should be a % and it should be always 0 to 100. But normally Interest_Rate can be very high value such as 80%. The data that appears to be impractical will be excluded.

```
In [35]: train_df1 = train_df1[(train_df1['Interest_Rate'] >= 1) & (train_df1['Interest_Rate'] < 40)]

         train_df1.shape
```

Regenerate the diagrams again to see how it reflects now.

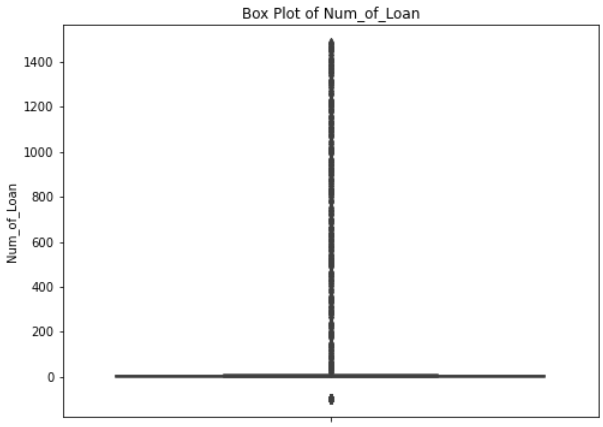


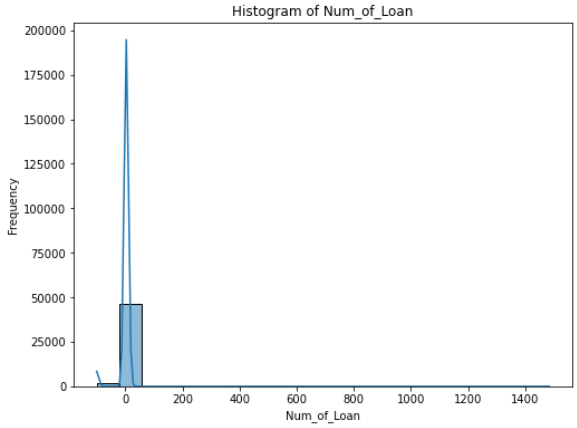


Column: Num_of_Loan

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [37]: # Num_of_Loan analysis
         generate_boxplot(train_df1, 'Num_of_Loan')
         generate_histogram(train_df1, 'Num_of_Loan')
         # generate_bar_chart(train_df1, 'Num_of_Loan')
```



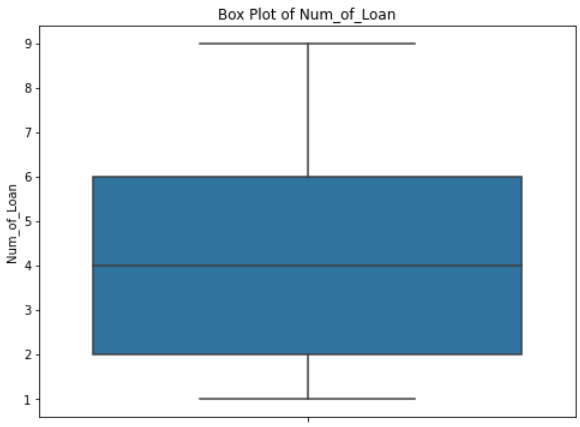


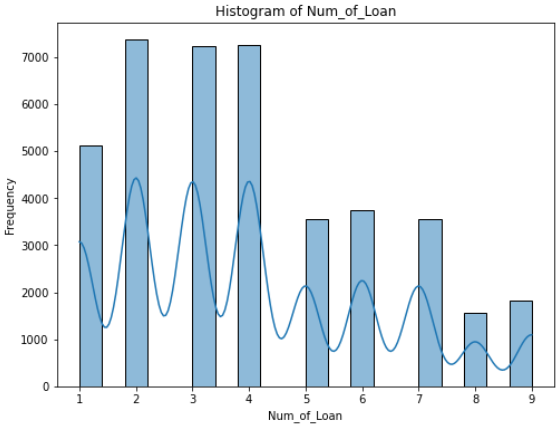
Above Num_of_Loan is not appearing to be realistic practically in real world. So, those outliers should be excluded.

```
In [38]: train_df1 = train_df1[(train_df1['Num_of_Loan'] >= 1) & (train_df1['Num_of_Loan'] < 15)]

         train_df1.shape
```

Regenerate the diagrams again to see how it reflects now.

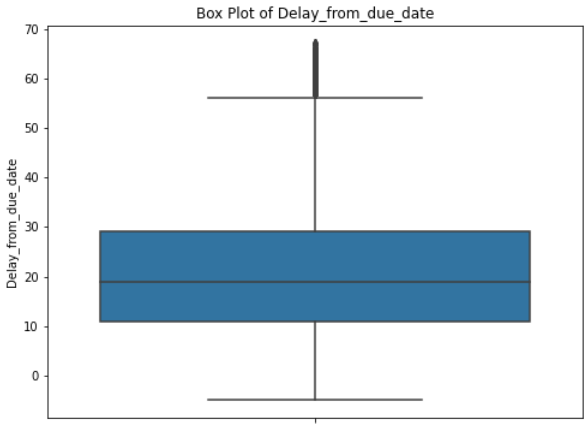


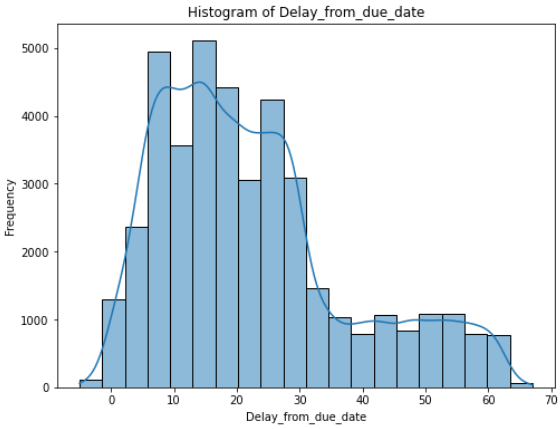


Column: Delay_from_due_date

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

In [41]:
```
# Delay_from_due_date analysis
generate_boxplot(train_df1, 'Delay_from_due_date')
generate_histogram(train_df1, 'Delay_from_due_date')
# generate_bar_chart(train_df1, 'Delay_from_due_date')
```



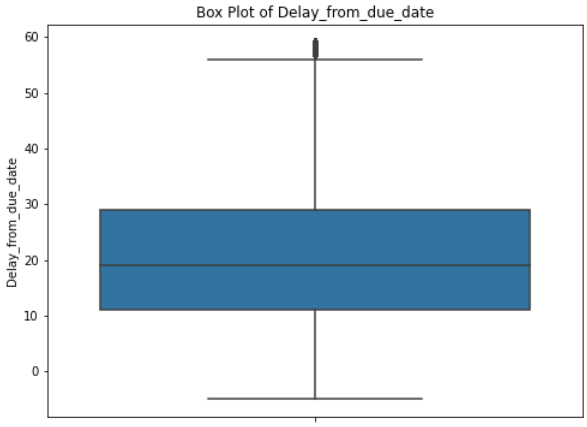


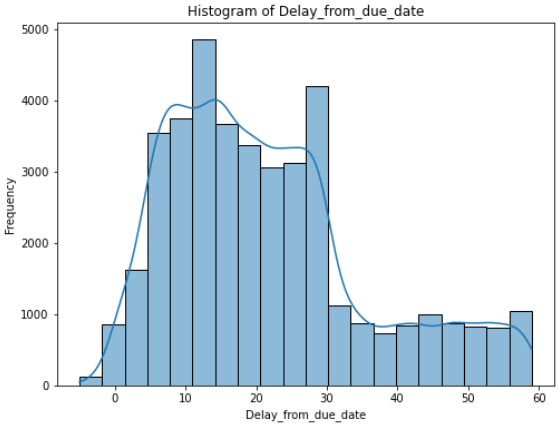
Seems there are few outliers in the Delay_from_due_date data. So, they will be excluded.

In [42]:
```
train_df1 = train_df1[(train_df1['Delay_from_due_date'] >= -10) & (train_df1['Delay_from_due_date'] < 60)]

train_df1.shape
```

Regenerate the diagrams again to see how it reflects now.

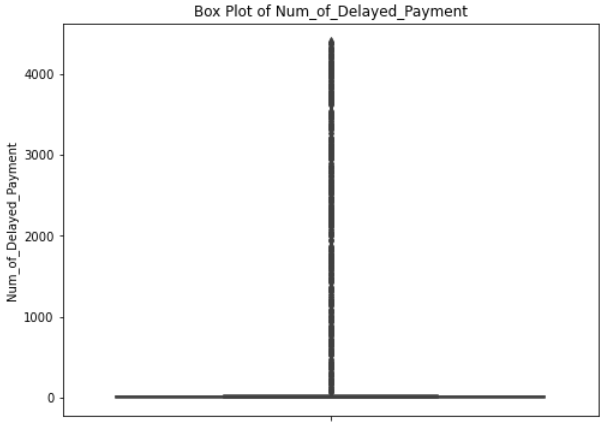


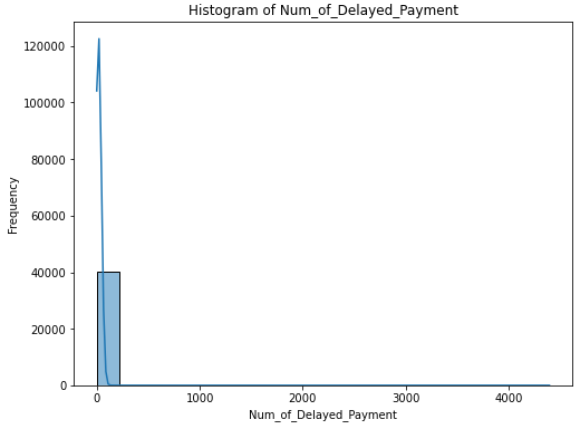


Column: Num_of_Delayed_Payment

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [44]: # Num_of_Delayed_Payment analysis
generate_boxplot(train_df1, 'Num_of_Delayed_Payment')
generate_histogram(train_df1, 'Num_of_Delayed_Payment')
# generate_bar_chart(train_df1, 'Num_of_Delayed_Payment')
```

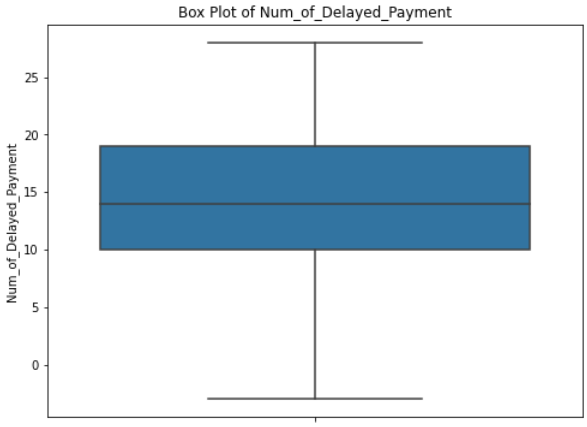


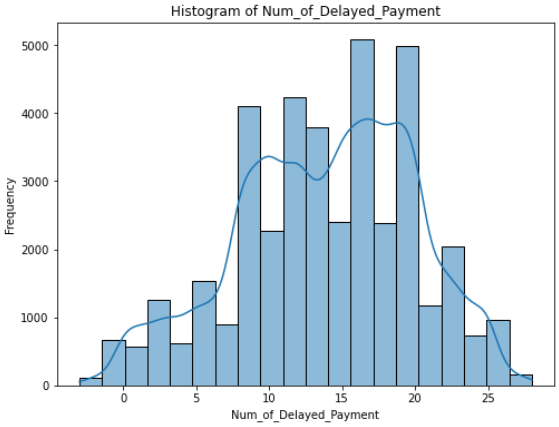


Num_of_Delayed_Payment count is having impractical data as it appears to be. So, those anomalies will be excluded.

```
In [45]: train_df1 = train_df1[(train_df1['Num_of_Delayed_Payment'] >= -4) & (train_df1['Num_of_Delayed_Payment'] < 72)]
```

Regenerate the diagrams again to see how it reflects now.

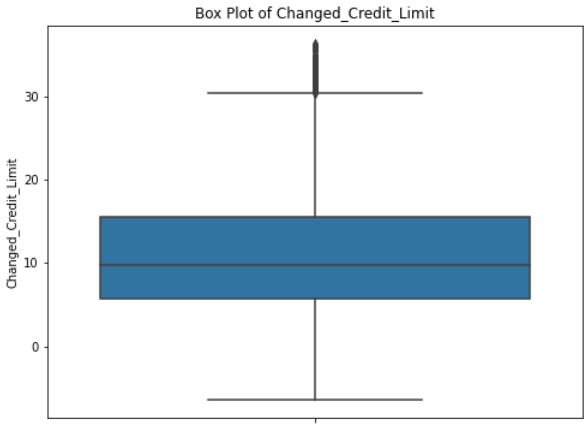


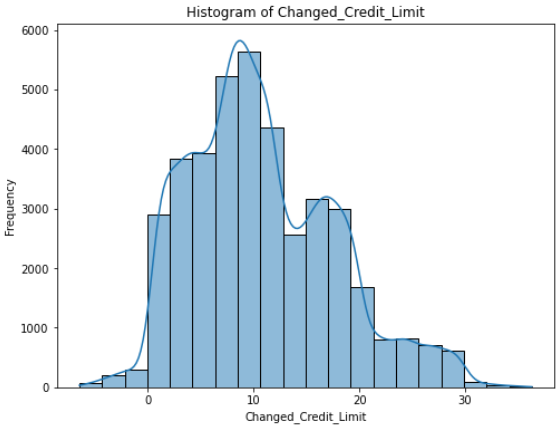


Column: Changed_Credit_Limit

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [48]: # Changed_Credit_Limit analysis
         generate_boxplot(train_df1, 'Changed_Credit_Limit')
         generate_histogram(train_df1, 'Changed_Credit_Limit')
         # generate_bar_chart(train_df1, 'Changed_Credit_Limit')
```



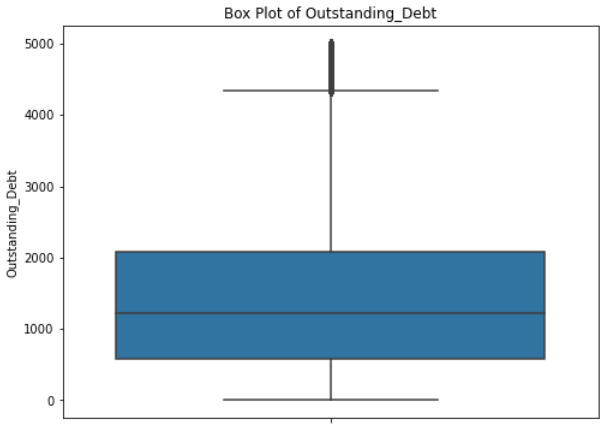


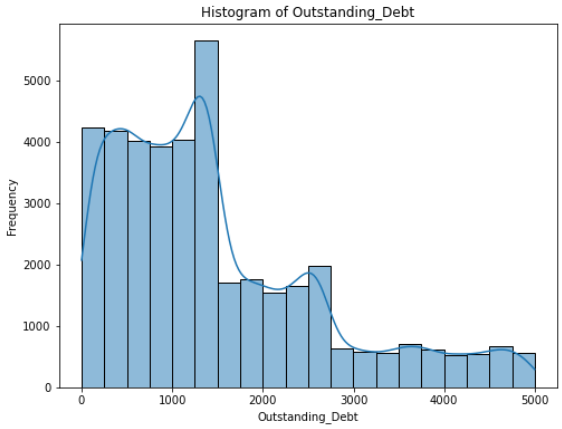
Changed_Credit_Limit having a practical data distribution.

Column: Outstanding_Debt

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [49]: # Outstanding_Debt analysis
         generate_boxplot(train_df1, 'Outstanding_Debt')
         generate_histogram(train_df1, 'Outstanding_Debt')
         # generate_bar_chart(train_df1, 'Outstanding_Debt')
```



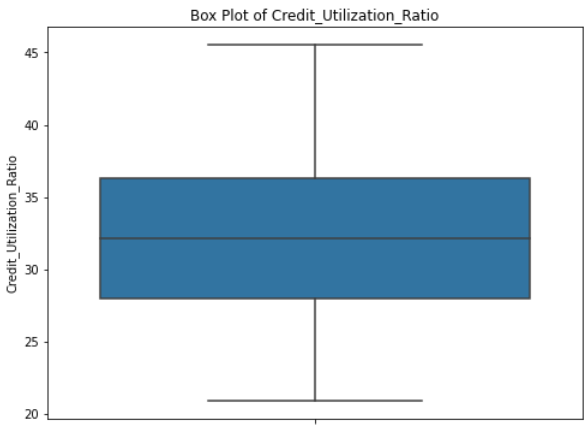


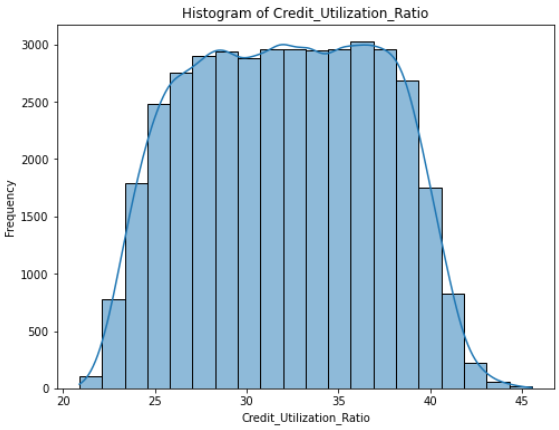
Outstanding_Debt having a practical data distribution.

Column: Credit_Utilization_Ratio

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [50]: # Credit_Utilization_Ratio analysis after cleanup
         generate_boxplot(train_df1, 'Credit_Utilization_Ratio')
         generate_histogram(train_df1, 'Credit_Utilization_Ratio')
         # generate_bar_chart(train_df1, 'Credit_Utilization_Ratio')
```



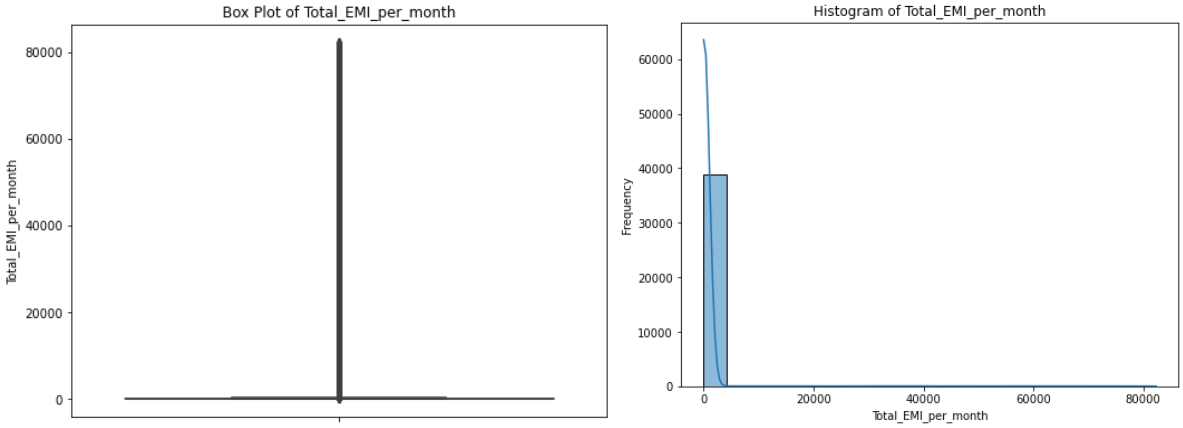


Credit_Utilization_Ratio having a practical data distribution.

Column: Total_EMI_per_month

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

```
In [51]: # Total_EMI_per_month analysis
         generate_boxplot(train_df1, 'Total_EMI_per_month')
         generate_histogram(train_df1, 'Total_EMI_per_month')
         # generate_bar_chart(train_df1, 'Total_EMI_per_month')
```

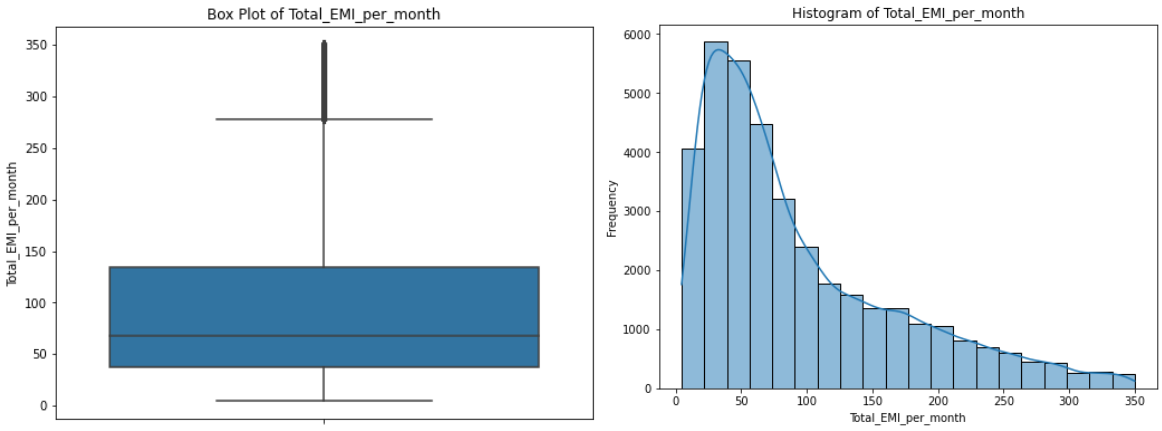


As per the box plot, seems Total_EMI_per_month is having outliers and those will be excluded.

```
In [53]: # Remove Total_EMI_per_month > 350

         train_df1 = train_df1[(train_df1['Total_EMI_per_month'] >= 0) & (train_df1['Total_EMI_per_month'] < 350)]
```

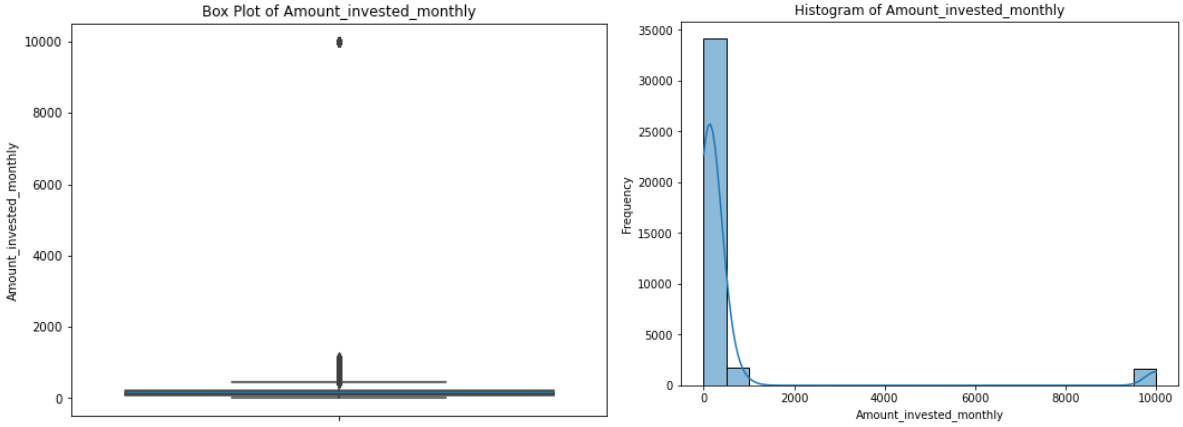
Regenerate the diagrams again to see how it reflects now.



Column: Amount_invested_monthly

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

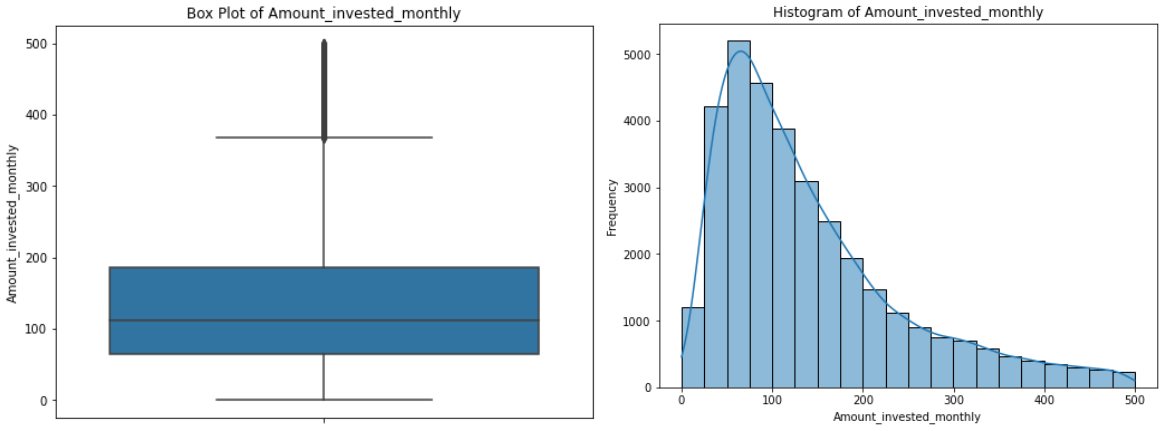
```
In [56]: # Amount_invested_monthly analysis
         generate_boxplot(train_df1, 'Amount_invested_monthly')
         generate_histogram(train_df1, 'Amount_invested_monthly')
         # generate_bar_chart(train_df1, 'Amount_invested_monthly')
```



It appears to be that there are outliers according to the boxplot, as there is no any instance from 2000 to nearly 9000. So, the outliers will be excluded and the data will be taken within the range of 0 to 500 to be more practical.

```
In [57]: train_df1 = train_df1[(train_df1['Amount_invested_monthly'] >= 0) & (train_df1['Amount_invested_monthly'] < 500)]
```

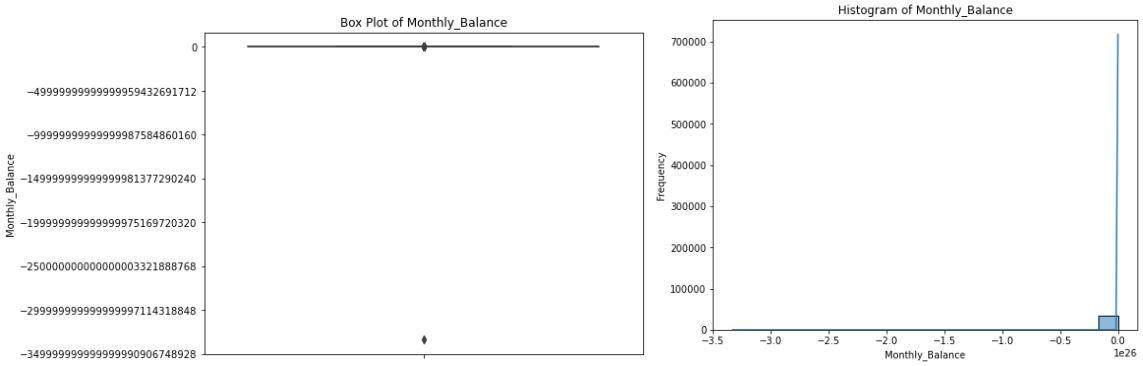
Regenerate the diagrams again to see how it reflects now.



Column: Monthly_Balance

Generate the box plot and the histogram diagrams to get a visual idea about the data distribution.

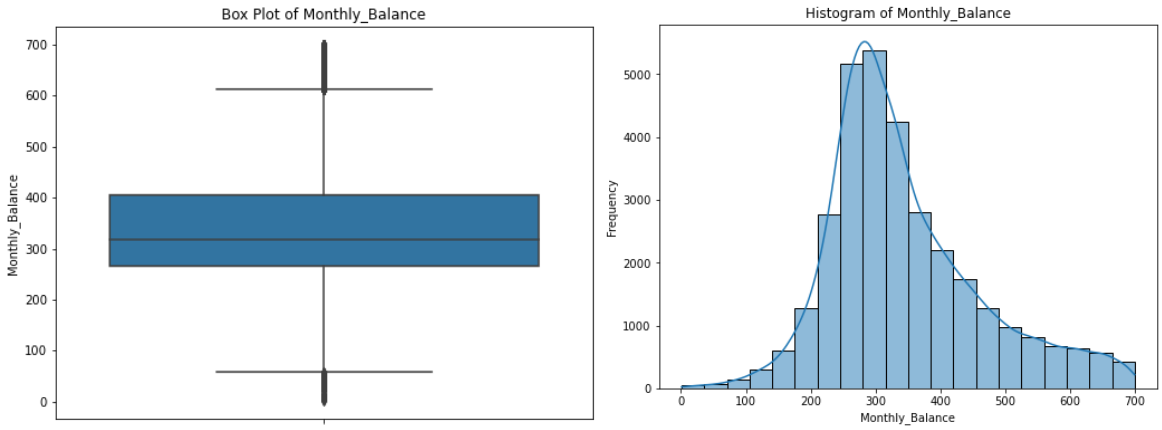
```
In [59]: # Monthly_Balance analysis
         generate_boxplot(train_df1, 'Monthly_Balance')
         generate_histogram(train_df1, 'Monthly_Balance')
         # generate_bar_chart(train_df1, 'Monthly_Balance')
```



Seems Monthly_Balance is having huge negative values as well as huge positive values so those data will be excluded.

```
In [60]: train_df1 = train_df1[(train_df1['Monthly_Balance'] >= 0) & (train_df1['Monthly_Balance'] < 700)]
```

Regenerate the diagrams again to see how it reflects now.



Now it appears that the complete data set could be used for training. The cleaned dataset will be exported to a csv so this can be used in a later stage without doing all above cleaning process.

```
In [87]: file_path = 'df_clean.csv'
         df_clean.to_csv(file_path, index=False)
```

Feature Correlations with the Target Variable

ANOVA p-values Verification

The ANOVA (Analysis of Variance) test is a statistical test that is widely used to examine whether there are significant changes in numerical feature means across multiple categories of a categorical variable [4]. It can assist you in determining whether there is a relationship or connection between a numerical feature and a categorical target variable.

This test will help to understand correlations or any relations with numerical cColumns and Credit_Score (Categorical)

```
In [65]: # List of numerical columns (excluding 'Credit_Score')
         numerical_columns = df.select_dtypes(include=['int32', 'int64', 'float64']).columns.tolist()
```

Columns that are numerical.

```
In [66]: numerical_columns
Out[66]: ['Age',
          'Annual_Income',
          'Monthly_Inhand_Salary',
          'Num_Bank_Accounts',
          'Num_Credit_Card',
          'Interest_Rate',
          'Num_of_Loan',
          'Delay_from_due_date',
          'Num_of_Delayed_Payment',
          'Changed_Credit_Limit',
          'Outstanding_Debt',
          'Credit_Utilization_Ratio',
          'Total_EMI_per_month',
          'Amount_invested_monthly',
          'Monthly_Balance']
```

Perform the ANOVA test for all the numerical features and see how they reflect.

```
In [67]: # Perform ANOVA for each numerical feature
         p_values = {}
         for col in numerical_columns:
             groups = [df[df['Credit_Score'] == val][col] for val in df['Credit_Score'].unique()]
             f_statistic, p_value = f_oneway(*groups)
             p_values[col] = p_value
```

```
In [68]: # Display p-values
         print("ANOVA p-values:")
         for col, p_value in p_values.items():
             print(f"{col}: {p_value:.4f}")

         ANOVA p-values:
         Age: 0.0000
         Annual_Income: 0.0000
         Monthly_Inhand_Salary: 0.0000
         Num_Bank_Accounts: 0.0000
         Num_Credit_Card: 0.0000
         Interest_Rate: 0.0000
         Num_of_Loan: 0.0000
         Delay_from_due_date: 0.0000
         Num_of_Delayed_Payment: 0.0000
         Changed_Credit_Limit: 0.0000
         Outstanding_Debt: 0.0000
         Credit_Utilization_Ratio: 0.0838
         Total_EMI_per_month: 0.0000
         Amount_invested_monthly: 0.0000
         Monthly_Balance: 0.0000
```

Convert the correlation and ANOVA results to Data Frames for plotting.

```
In [69]: # Convert the correlation and ANOVA results to DataFrames for plotting
         corr_df = pd.DataFrame(p_values.items(), columns=['Feature', 'Correlation'])
         anova_df = pd.DataFrame(p_values.items(), columns=['Feature', 'ANOVA_P_Value'])
```

```
In [70]: corr_df
```

Out[70]:

	Feature	Correlation
0	Age	4.903622e-149
1	Annual_Income	1.818063e-86
2	Monthly_Inhand_Salary	8.232974e-84
3	Num_Bank_Accounts	0.000000e+00
4	Num_Credit_Card	0.000000e+00
5	Interest_Rate	0.000000e+00
6	Num_of_Loan	0.000000e+00
7	Delay_from_due_date	0.000000e+00
8	Num_of_Delayed_Payment	0.000000e+00
9	Changed_Credit_Limit	1.454099e-318
10	Outstanding_Debt	0.000000e+00
11	Credit_Utilization_Ratio	8.377517e-02
12	Total_EMI_per_month	1.391171e-60
13	Amount_invested_monthly	2.333639e-31
14	Monthly_Balance	1.431242e-220

```
In [71]: anova_df
```

Out[71]:

	Feature	ANOVA_P_Value
0	Age	4.903622e-149
1	Annual_Income	1.818063e-86
2	Monthly_Inhand_Salary	8.232974e-84
3	Num_Bank_Accounts	0.000000e+00
4	Num_Credit_Card	0.000000e+00
5	Interest_Rate	0.000000e+00
6	Num_of_Loan	0.000000e+00
7	Delay_from_due_date	0.000000e+00
8	Num_of_Delayed_Payment	0.000000e+00
9	Changed_Credit_Limit	1.454099e-318
10	Outstanding_Debt	0.000000e+00
11	Credit_Utilization_Ratio	8.377517e-02
12	Total_EMI_per_month	1.391171e-60
13	Amount_invested_monthly	2.333639e-31
14	Monthly_Balance	1.431242e-220

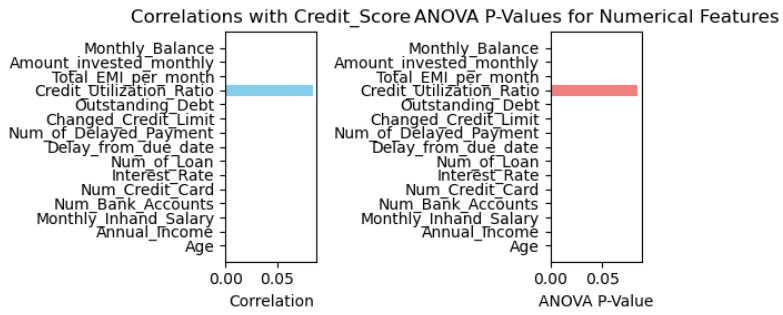
Create subplots for correlation and ANOVA p-value bar charts.

```
In [72]: # Create subplots for correlation and ANOVA p-value bar charts
         fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))

         # Bar chart for correlations
         axes[0].barh(corr_df['Feature'], corr_df['Correlation'], color='skyblue')
         axes[0].set_xlabel('Correlation')
         axes[0].set_title('Correlations with Credit_Score')

         # Bar chart for ANOVA p-values
         axes[1].barh(anova_df['Feature'], anova_df['ANOVA_P_Value'], color='lightcoral')
         axes[1].set_xlabel('ANOVA P-Value')
         axes[1].set_title('ANOVA P-Values for Numerical Features')

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



It is possible to consider that column to be a correlated column if the P-Value is less than 0.05 (5%). As a result, all columns appear to be related to the credit score, apart from Credit_Utilization_Ratio.

Chi-Square p-values Verification

The Chi-Squared test is often used to determine the correlation or independence of two categorical variables [5]. It's utilized in this case to see if there's a statistically significant link between each independent categorical column and the dependent categorical column 'Credit_Score.'

In [73]:
```
# List of categorical columns (excluding 'Credit_Score')
categorical_columns = df.select_dtypes(include=['object']).columns.tolist()
categorical_columns.remove('Credit_Score')
```

In [74]:
```
# Perform chi-squared test for each categorical feature
chi2_results = {}
for col in categorical_columns:
    # Create a contingency table
    contingency_table = pd.crosstab(df[col], df['Credit_Score'])
    chi2, p_value, _, _ = chi2_contingency(contingency_table)
    chi2_results[col] = {'Chi-Square': chi2, 'p-value': p_value}
```

In [75]:
```
# Display chi-squared test results
print("Chi-Squared Test Results:")
for col, results in chi2_results.items():
    print(f"{col}:")
    print(f"  Chi-Square: {results['Chi-Square']:.4f}")
    print(f"  p-value: {results['p-value']:.4f}")
```

```
Chi-Squared Test Results:
Occupation:
  Chi-Square: 76.6993
  p-value: 0.0000
Type_of_Loan:
  Chi-Square: 27571.7753
  p-value: 0.0000
Credit_Mix:
  Chi-Square: 10531.7895
  p-value: 0.0000
Credit_History_Age:
  Chi-Square: 6687.7563
  p-value: 0.0000
Payment_of_Min_Amount:
  Chi-Square: 6710.6645
  p-value: 0.0000
Payment_Behaviour:
  Chi-Square: 405.5424
  p-value: 0.0000
```

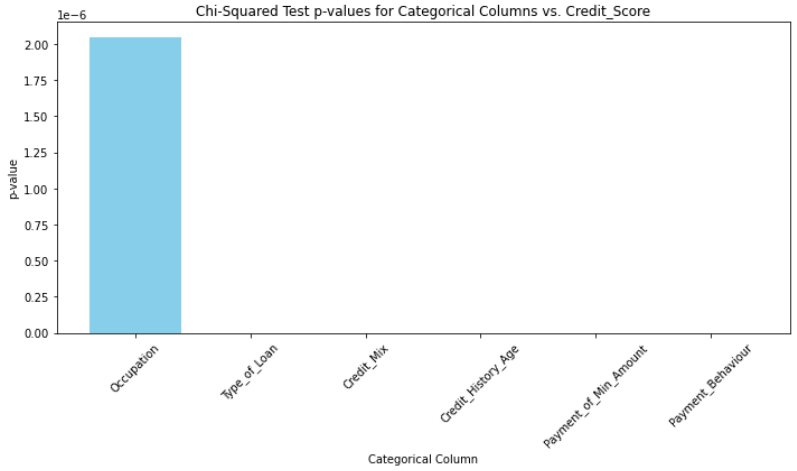
In [76]:
```
# Convert chi2_results to a DataFrame for easier plotting
chi2_df = pd.DataFrame(chi2_results).T
```

In [77]:
```
chi2_df
```

Out[77]:

	Chi-Square	p-value
Occupation	76.699306	2.048905e-06
Type_of_Loan	27571.775320	0.000000e+00
Credit_Mix	10531.789546	0.000000e+00
Credit_History_Age	6687.756257	0.000000e+00
Payment_of_Min_Amount	6710.664506	0.000000e+00
Payment_Behaviour	405.542424	6.223072e-81

In [78]:
```
# Create a bar chart for p-values
plt.figure(figsize=(10, 6))
plt.bar(chi2_df.index, chi2_df['p-value'], color='skyblue')
plt.xlabel('Categorical Column')
plt.ylabel('p-value')
plt.title('Chi-Squared Test p-values for Categorical Columns vs. Credit_Score')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



It is possible to consider a column to be a correlated column if the P-Value is less than 0.05 (5%). Each of the categorical columns (Occupation, Type of Loan, Credit Mix, Credit History Age, Payment of Minimum Amount, Payment Behavior) has a substantial correlation with the dependent categorical variable "Credit Score," as shown by the fact that all p-values are very close to 0.

II. Feature selection

Choosing the right features is essential to creating a trustworthy and efficient credit scoring model. As dimensions are reduced, model performance is improved, compliance and fairness are ensured, and the most informative and interpretable features are identified and retained. Making educated, transparent, and fair credit choices depends on this careful selection of attributes.

In [79]:
```
# Acording to the ANNOVA P-Value test, remove Credit_Utilization_Ratio column
train_df1 = train_df1.drop('Credit_Utilization_Ratio', axis=1)
```

Separate the dependent and indemnified variables into two sets before moving on.

```
In [82]: df_indi = train_df1.drop(['Credit_Score'], axis = 1)
         df_depend = train_df1['Credit_Score']

In [83]: df_depend

Out[83]: 0            Good
         6            Good
         8        Standard
         9            Good
         12           Good
                     ...
         99991    Standard
         99994        Poor
         99995        Poor
         99996        Poor
         99999        Poor
         Name: Credit_Score, Length: 32065, dtype: object
```

III. Data normalization

Another key preprocessing step in LM modeling is data normalization. By making certain that features are scaled consistently, it improves the model's performance, interpretability, and robustness while averting any potential numerical problems. The development of precise and trustworthy credit scoring algorithms depends on this normalizing procedure.

We are following 2 steps. 1st, will do the normalization for the numerical fields using MinMaxScaler [4]. and as the 2nd step will do the Categorical Encoding.

To do this, need to extract numerical and categorical columns in to two sets.

```
In [84]: # Normalize numerical columns using Min-Max scaling (scaling to the range [0, 1])
         numerical_columns = df_indi.select_dtypes(include=['number']).columns.tolist()
         categorical_columns = df_indi.select_dtypes(include=['object']).columns.tolist()

In [85]: numerical_scaler = MinMaxScaler()

In [86]: df_indi[numerical_columns] = numerical_scaler.fit_transform(df_indi[numerical_columns])

In [87]: # Encode categorical variables
         label_encoders = {}
         for col in categorical_columns:
             label_encoder = LabelEncoder()
             df_indi[col] = label_encoder.fit_transform(df_indi[col])
             label_encoders[col] = label_encoder
```

Below is the dataset after numerical normalization and the categorical encoding.

In [88]: df_indi

Out[88]:

	Age	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credit_Card	Interest_Rate	Num_of_Loan	Type_of_Loan	Dela
0	0.083333	12	0.098459	0.143959	0.222222	0.333333	0.060606	0.375	99	
6	0.083333	12	0.098459	0.143959	0.222222	0.333333	0.060606	0.375	99	
8	0.222222	7	0.226400	0.258765	0.111111	0.333333	0.151515	0.000	530	
9	0.222222	13	0.226400	0.258765	0.111111	0.333333	0.151515	0.000	530	
12	0.222222	13	0.226400	0.258765	0.111111	0.333333	0.151515	0.000	530	
...	...	...	...	...	...	...	...	...	...	
99991	0.250000	1	0.105686	0.153902	1.000000	0.777778	0.848485	0.500	3859	
99994	0.138889	9	0.265278	0.289184	0.333333	0.555556	0.181818	0.125	529	
99995	0.138889	9	0.265278	0.289184	0.333333	0.555556	0.181818	0.125	529	
99996	0.138889	9	0.265278	0.289184	0.333333	0.555556	0.181818	0.125	529	
99999	0.138889	9	0.265278	0.289184	0.333333	0.555556	0.181818	0.125	529	

32065 rows × 20 columns

3. Methodology

We have decided to use the supervised learning framework to create a reliable and predictive credit score model. Because it uses labeled historical data to create precise, understandable, and scalable models that enable informed credit choices, supervised learning is well suited for credit scoring. These models enhance the loan process' impartiality, openness, and efficiency in addition to improving risk assessment.

3.1. Model Selection

We carefully explored numerous algorithms under supervised learning for the credit scoring model to maximize predicted accuracy along with other metrics such as Precision, Recall & F1-Score [4] [7] . We have chosen the sophisticated machine learning methods of Logistic Regression and Random Forest Classifier after careful consideration.

Why logistic regression model?

The interpretability, transparency, effectiveness, and function as a baseline model of logistic regression make it a good fit for our credit rating problem. It gives us the ability to develop a credit scoring model that is not just accurate and transparent, but also understandable, in line with our goal to making well-informed credit decisions.

Why random forest classifier model?

Due to its great predictive power, robustness, feature importance analysis, and capacity to handle

skewed data, the Random Forest Classifier is an excellent option for our credit score model. We seek to attain both accuracy and reliability in credit assessments by incorporating Random Forest into our methodology, providing well-informed and impartial lending decisions.

3.2. Model Training

In this case, a dataset is used to train a model to find patterns, connections, and links in the data. Model training's main objective is to equip the model with the skills necessary to correctly forecast or categorize brand-new, untainted data.

```
In [89]: # Split data frames in to training & test (70% 30%)
         from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(df_indi,df_depend, test_size = 0.30, random_state = 100)
```

3.2.1. Logistic Regression Model

Creating a Logistic Regression model with a specified maximum number of iterations and then training it using the X_train and y_train datasets

```
In [90]: # Create and train a Logistic Regression model
         LR_model = LogisticRegression(max_iter=7000)
         LR_model.fit(X_train, y_train)

Out[90]: LogisticRegression
         LogisticRegression(max_iter=7000)
```

When the following code line is executed, the trained Logistic Regression model LR_model will use its learned parameters to make predictions based on the features in the X_test dataset

```
In [91]: # Make predictions on the test set
         LR_y_pred = LR_model.predict(X_test)
```

LR_model will use its learned parameters to make predictions based on the features in the training dataset X_train.

```
In [92]: LR_train_preds = LR_model.predict(X_train)
         LR_train_preds

Out[92]: array(['Standard', 'Poor', 'Standard', ..., 'Good', 'Standard',
                'Standard'], dtype=object)
```

LR_model will use its learned parameters to make predictions on the training dataset X_train and then calculate the training accuracy score by comparing these predictions to the actual labels in y_train.

```
In [94]: LR_train_preds_score = LR_model.score(X_train,y_train)
         LR_train_preds_score
Out[94]: 0.6453553129873023
```

LR_model will use its learned parameters to make predictions on the test dataset X_test and then calculate the test accuracy score by comparing these predictions to the actual labels in y_test.

```
In [95]: LR_test_preds_score = LR_model.score(X_test,y_test)
         LR_test_preds_score
Out[95]: 0.6477130977130977
```

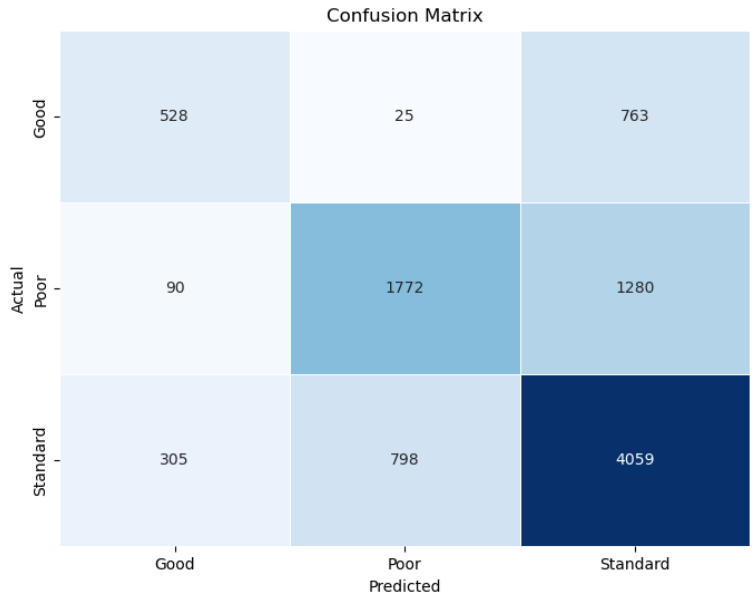
Compute a confusion matrix based on the actual target labels (y_test) and the predicted labels (y_pred)

```
In [96]: # Compute the confusion matrix
         conf_matrix = confusion_matrix(y_test, LR_y_pred)
```

Create a pandas DataFrame (confusion_df) from a confusion matrix (conf_matrix) computed based on the predictions of LR model.

```
In [97]: # Create a DataFrame for the confusion matrix (optional)
         confusion_df = pd.DataFrame(conf_matrix, index=LR_model.classes_, columns=LR_model.classes_)
```

```
In [98]: # Create a heatmap to visualize the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_df, annot=True, fmt='d', cmap='Blues', linewidths=.5, cbar=False)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



Perform accuracy, classification_rep, and confusion_mat evaluation tasks.

```
In [99]: # Evaluate the model
accuracy = accuracy_score(y_test, LR_y_pred)
classification_rep = classification_report(y_test, LR_y_pred)
confusion_mat = confusion_matrix(y_test, LR_y_pred)
```

```
In [100]: # Print the results
print(f"Accuracy: {accuracy}")
print("Classification Report:\n", classification_rep)
print("Confusion Matrix:\n", confusion_mat)
```

```
Accuracy: 0.661018711018711
Classification Report:
               precision    recall  f1-score   support

         Good       0.57      0.40      0.47      1316
         Poor       0.68      0.56      0.62      3142
     Standard       0.67      0.79      0.72      5162

     accuracy                           0.66      9620
    macro avg       0.64      0.58      0.60      9620
 weighted avg       0.66      0.66      0.65      9620

Confusion Matrix:
 [[ 528   25  763]
 [  90 1772 1280]
 [ 305  798 4059]]
```

3.2.2. Random Forest Classifier Model

Create and train Random Forest Classifier (RFC) model.

```
In [101]: # Create and train a Random Forest Classifier model
          RFC_model = RandomForestClassifier()
          RFC_model.fit(X_train, y_train)

Out[101]: ▾ RandomForestClassifier
          RandomForestClassifier()
```

```
In [102]: # Make predictions on the test set
          RFC_y_pred =RFC_model.predict(X_test)
```

RFC_model will use its learned parameters to make predictions based on the features in the training dataset X_train.

```
In [103]: RFC_train_preds = RFC_model.predict(X_train)
          RFC_train_preds

Out[103]: array(['Standard', 'Standard', 'Poor', ..., 'Good', 'Standard', 'Good'],
                dtype=object)
```

RFC_model will use its learned parameters to make predictions based on the features in the test dataset X_test.

```
In [104]: RFC_test_preds = RFC_model.predict(X_test)
          RFC_test_preds

Out[104]: array(['Standard', 'Standard', 'Good', ..., 'Poor', 'Standard', 'Good'],
                dtype=object)
```

Using the training data (X_train and y_train) to calculate the training accuracy score of the Random Forest Classifier model (RFC_model). The model will use its learned parameters to make predictions on the training dataset, and the training accuracy score will be calculated by comparing these predictions to the actual labels in y_train.

```
In [105]: RFC_train_preds_score = RFC_model.score(X_train,y_train)
          RFC_train_preds_score

Out[105]: 1.0
```

Using the test data (X_test and y_test) to calculate the test accuracy score of the Random Forest Classifier model (RFC_model). The model will use its learned parameters to make predictions on the test dataset X_test, and the test accuracy score will be calculated by comparing these predictions to the actual labels in y_test.

```
In [106]: RFC_test_preds_score = RFC_model.score(X_test,y_test)
          RFC_test_preds_score

Out[106]: 0.7903326403326403
```

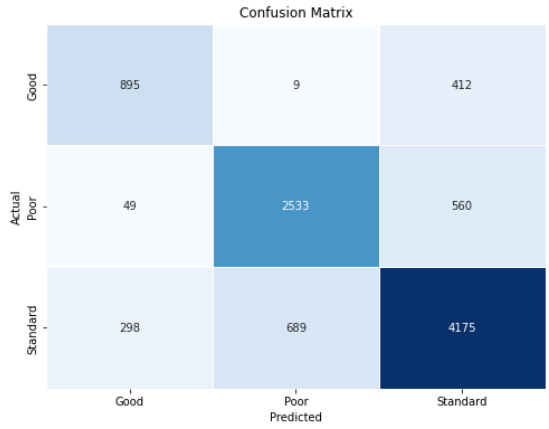
Compute a confusion matrix based on the actual target labels (y_test) and the predicted labels (rfc_test_preds).

In [107]:
```
# Compute the confusion matrix
conf_matrix_rfc = confusion_matrix(y_test, RFC_y_pred)
```

Create a pandas DataFrame (confusion_df) from a confusion matrix (conf_matrix) computed.

In [108]:
```
# Create a DataFrame for the confusion matrix (optional)
confusion_df_rfc = pd.DataFrame(conf_matrix_rfc, index=RFC_model.classes_, columns=RFC_model.classes_)
```

In [109]:
```
# Create a heatmap to visualize the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_df_rfc, annot=True, fmt='d', cmap='Blues', linewidths=.5, cbar=False)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



In [110]:
```
# Evaluate the model
accuracy_rfc = accuracy_score(y_test, RFC_y_pred)
classification_rep_rfc = classification_report(y_test, RFC_y_pred)
confusion_mat_rfc = confusion_matrix(y_test, RFC_y_pred)
```

In [111]:
```
# Print the results
print(f"Accuracy: {accuracy_rfc:.2f}")
print("Classification Report:\n", classification_rep_rfc)
print("Confusion Matrix:\n", confusion_mat_rfc)
```

```
Accuracy: 0.79
Classification Report:
              precision    recall  f1-score   support

        Good       0.72      0.68      0.70      1316
        Poor       0.78      0.81      0.79      3142
    Standard       0.81      0.81      0.81      5162

    accuracy                           0.79      9620
   macro avg       0.77      0.77      0.77      9620
weighted avg       0.79      0.79      0.79      9620

Confusion Matrix:
 [[ 895    9  412]
 [  49 2533  560]
 [ 298  689 4175]]
```

4. Results

To analysis the results, we have used the Confusion metrics which as Accuracy, Precision, Recall, F1-score values.

Logistic Regression Model

```
In [97]: # Print the results
         print(f"Accuracy: {accuracy}")
         print("Classification Report:\n", classification_rep)
         print("Confusion Matrix:\n", confusion_mat)

         Accuracy: 0.6477130977130977
         Classification Report:
                       precision    recall  f1-score   support

                Good       0.53      0.31      0.39      1316
                Poor       0.67      0.57      0.62      3142
            Standard       0.65      0.78      0.71      5162

            accuracy                           0.65      9620
           macro avg       0.62      0.55      0.57      9620
        weighted avg       0.64      0.65      0.64      9620

         Confusion Matrix:
          [[ 413   28  875]
          [  86 1795 1261]
          [ 283  856 4023]]
```

Random Forest Classifier Model

```
In [111]: # Print the results
          print(f"Accuracy: {accuracy_rfc:.2f}")
          print("Classification Report:\n", classification_rep_rfc)
          print("Confusion Matrix:\n", confusion_mat_rfc)

          Accuracy: 0.79
          Classification Report:
                        precision    recall  f1-score   support

                 Good       0.72      0.68      0.70      1316
                 Poor       0.78      0.81      0.79      3142
             Standard       0.81      0.81      0.81      5162

             accuracy                           0.79      9620
            macro avg       0.77      0.77      0.77      9620
         weighted avg       0.79      0.79      0.79      9620

          Confusion Matrix:
           [[ 895    9  412]
           [  49 2533  560]
           [ 298  689 4175]]
```

4.1. Model Evaluation

4.1.1. Logistic Regression Model

The accuracy of the Logistic Regression Model was 0.6477, meaning that it properly predicted applicants' creditworthiness in about 64.77% of cases. With the maximum precision and recall seen for "Standard" applicants, it displays modest performance. However, as evidenced by the decreased

precision and recall for the "Good" applicant group, it has difficulty correctly identifying qualified candidates. The moderate F1-scores show a trade-off between recall and precision.

4.1.2. Random Forest Classifier Model

The performance of the Random Forest Classifier Model is superior to the Logistic Regression Model. It obtains a higher accuracy of 0.79, meaning that it accurately predicts applicants' creditworthiness in about 79% of cases. It outperforms the Logistic Regression Model in terms of accuracy, precision, recall, and F1-scores for all three classes. It is a superior contender for the credit scoring task because it excels at producing predictions that are more accurate and maintaining a better balance between precision and recall.

5. Discussion

The outcomes of our investigation using the Logistic Regression and Random Forest Classifier models provide insightful information about how well each model performs when used for credit scoring. The Random Forest Classifier outperforms the Logistic Regression model, as shown by the metrics for accuracy, precision, recall, and F1-score.

The accuracy of the Random Forest Classifier is about 79%, which is much higher than the accuracy of the Logistic Regression model, which is about 64.77%. The Random Forest model routinely outperforms in accurately categorizing applicants, as seen by the greater precision, recall, and F1-scores for the three creditworthiness categories ("Good," "Poor," and "Standard") in the model.

The technique has some noteworthy drawbacks, including dataset size and data quality. First, the training and evaluation datasets may be quite small, thus restricting the model's ability to catch varied credit patterns. Inadequate sample size can lead to overfitting, a condition in which the model performs well on training data but struggles to generalize to new scenarios. Second, data quality is critical, and errors, missing values, or outliers can all have a major impact on model performance. Inaccurate estimates and potentially misleading credit decisions might result from inaccurate data. Increasing the reliability and resilience of credit scoring models requires addressing these limitations through larger, more representative datasets and comprehensive data preprocessing procedures.

To increase the accuracy of credit scoring, it is advised that future research investigate additional machine learning algorithms as well, which uses bias mitigation strategies, and take feature engineering into account. Model robustness could be increased further by adding more data and using other data sources. Further, it is suggesting to re model with newest dataset after a period time (if the current model is used to take decisions).

6. Conclusion

A crucial stage in the financial sector's efforts to determine creditworthiness and make wise lending decisions is the creation and assessment of credit scoring models. In this investigation, we investigated the accuracy of two machine learning models for predicting credit outcomes: Logistic Regression and Random Forest Classifier. Even though both models had comparable strengths and shortcomings, modest accuracy, and balanced precision-recall scores, numerous crucial issues came into focus.

Choosing a machine learning algorithm is just one step in the credit rating process, to start. The performance of the model can be strongly impacted by the quantity, quality of the dataset. Because of the dataset's short size, the models may not have been able to generalize well, and poor data quality may have caused noise in the predictions.

With the help of Logistic Regression and Random Forest Classifier, we evaluated credit scoring models and learned a lot about how well they performed. In comparison to the Logistic Regression model's accuracy of about 64.77%, the Random Forest Classifier achieved an impressively high accuracy of about 79%. Because of the increased precision, credit assessments and lending decisions are more accurate and well-informed.

In conclusion, our analysis supports the Random Forest Classifier model's dominance in credit scoring for the given data set, but it also raises crucial points and suggests areas for development. Even though the Random Forest Classifier model gave the best accuracy for the current evaluation, this algorithm could not be the best for other data samples. For the development of strong and dependable credit scoring models in real-world scenarios, considerations of data amount, quality, fairness, and continual model improvement and selecting the right algorithm are essential.

7. References

[1] R. PARIS, "Credit score classification," 2022. [Online]. Available: https://www.kaggle.com/datasets/parisrohan/credit-score-classification.

[2] M. PRATA, "Creditability - German Credit Data," 2020. [Online]. Available: https://www.kaggle.com/datasets/mpwolke/cusersmarildownloadsgermancsv.

[3] M. L. UCI , "Default of Credit Card Clients Dataset," 2016. [Online]. Available: https://www.kaggle.com/datasets/uciml/default-of-credit-card-clients-dataset.

[4] L. Chandrakantha, "Learning ANOVA concepts using simulation," in *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education*, 2014, pp. 1-5.

[5] H. M. a. K. I. Deberneh, "Prediction of type 2 diabetes based on machine learning algorithm," *International journal of environmental research and public health,* vol. 18, p. 3317, 2021.

[6] B. a. R. K. Deepa, "Epileptic seizure detection using deep learning through min max scaler normalization," *Int. J. Health Sci,* vol. 6, pp. 10981-10996, 2022.

[7] S. a. K. B. a. D. D. C. a. S. N. H. Dev, "Performance Analysis and Prediction of Diabetes using Various Machine Learning Algorithms," in *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, 2022, pp. 517-521.

[8] G. N. a. F. H. a. A. M. a. R. O. a. A. M. S. Ahmad, "Mixed machine learning approach for efficient prediction of human heart disease by identifying the numerical and categorical features," *Applied Sciences,* vol. 12, p. 7449, 2022.

8. Appendix

```python
import os
import pandas as pd
import numpy as np
import sklearn
import matplotlib.pyplot as plt
import seaborn as sns
from IPython import display
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder, MinMaxScaler

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
from sklearn.ensemble import RandomForestClassifier

from scipy.stats import f_oneway
from scipy.stats import chi2_contingency
```

```python
pd.set_option('display.max_columns', 30)
pd.set_option('display.max_rows', 10000)
```

```
# Loading training data from the CSV
```

```python

# Specify dtype for column 'Monthly_Balance' as a string (Column 'Monthly_Balance' has mixed data types)

dtypes = {'Monthly_Balance': str}

train_df = pd.read_csv('train.csv', dtype = dtypes)

```

Data Preparation - Step 1

This data set has many columns and need to remove all personally Identifiable Data

```python

train_df.columns

```

The ID, Customer_ID, Name, and SSN columns are extremely useful in identifying a specific customer (PII data). So, such columns will be removed from our trainigi data set. Furthermore, there is a column called 'Month' that does not give much weight to our classification. As a result, those columns will be excluded too.

The Num_Credit_Inquiries data is also ambiguous. It doesn't mean clearly the query count is generated from bank to customer or customer to bank. We believe it will not add value to the credit score we intend to establish. As such, that column will be excluded as well.

```python

#Remove sensitive data columns
```

train_df1 = train_df.drop(['ID', 'Customer_ID', 'Name', 'SSN', 'Month', 'Num_Credit_Inquiries'], axis=1)

train_df1.shape

```

# Data Cleanup

```python
train_df1 = train_df1.applymap(

lambda x: x if x is np.NaN or not \

isinstance(x, str) else str(x).strip('_')).replace(['', 'nan', '!@9#%8', '#F%$D@*&8'], np.NaN)

train_df1.shape
```

```python
train_df1.dtypes
```

Data Preparation - Step 2

Some columns shouldn't be an Object type as it has all numeric values. Let's convert those columns to a Numeric Type

```python
train_df1['Age'] = pd.to_numeric(train_df1['Age'], errors='coerce').astype('int64')

train_df1['Annual_Income'] = pd.to_numeric(train_df1['Annual_Income'],
errors='coerce').astype('float')
```
```

train_df1['Num_of_Loan'] = pd.to_numeric(train_df1['Num_of_Loan'], errors='coerce').astype('int32')

train_df1['Changed_Credit_Limit'] = pd.to_numeric(train_df1['Changed_Credit_Limit'], errors='coerce').astype('float')

train_df1['Outstanding_Debt'] = pd.to_numeric(train_df1['Outstanding_Debt'], errors='coerce').astype('float')

train_df1['Amount_invested_monthly'] = pd.to_numeric(train_df1['Amount_invested_monthly'], errors='coerce').astype('float')

train_df1['Monthly_Balance'] = pd.to_numeric(train_df1['Monthly_Balance'], errors='coerce').astype('float')

train_df1['Num_of_Delayed_Payment'] = pd.to_numeric(train_df1['Num_of_Delayed_Payment'], errors='coerce').astype('float')

```

```python

train_df1.dtypes

```

Missing Values

```python

# Calculate the sum of missing values in each column

missing_values_sum = train_df1.isna().sum()

# Calculate the percentage of missing values for each column

total_rows = train_df1.shape[0]

missing_values_percentage = (missing_values_sum / total_rows) * 100

```
# Get the data type of the column as well
column_datatypes = train_df1.dtypes

# Create a DataFrame to display the results
missing_info = pd.DataFrame({
'Data Type': column_datatypes,
'Missing Values': missing_values_sum,
'Percentage Missing (%)': missing_values_percentage
})

# Display the missing values sum and percentage
print(missing_info)
```

```python
# Create a bar plot for missing values percentage
plt.figure(figsize=(10, 5))

missing_values_percentage.plot(kind='bar', color='lightcoral')
plt.title('Missing Values Percentage')
plt.xlabel('Columns')
plt.ylabel('Percentage (%)')

# Adjust spacing between plots
plt.tight_layout()

# Show the plot
```

plt.show()

```

#### Occupation (7%), Monthly_Inhand_Salary (15%), Type_of_Loan (11%), Num_of_Delayed_Payment (7%), Changed_Credit_Limit (2%),  Credit_Mix (20%), Credit_History_Age (9%), Amount_invested_monthly (4%), Payment_Behaviour (7%) & Monthly_Balance (1%) are having missing values.

# Missing Value Handling Stratergy:

###  1. most_frequent strategy will use for Occupation, Monthly_Inhand_Salary, Type_of_Loan, Credit_Mix, Payment_Behaviour

### 2. Remove rows for the other colums with missing values

```python
# Replace missing values using most_frequent strategy
imputer = SimpleImputer(strategy='most_frequent')
train_df1[['Occupation', 'Type_of_Loan', 'Credit_Mix', 'Payment_Behaviour']] = imputer.fit_transform(train_df1[['Occupation', 'Type_of_Loan', 'Credit_Mix', 'Payment_Behaviour']])
```

```python
# Remove rows with missing values
train_df1 = train_df1.dropna(axis=0)
```
```

```python
# Calculate the sum of missing values in each column
missing_values_sum = train_df1.isna().sum()

# Calculate the percentage of missing values for each column
total_rows = train_df1.shape[0]
missing_values_percentage = (missing_values_sum / total_rows) * 100

# Get the data type of the column as well
column_datatypes = train_df1.dtypes

# Create a DataFrame to display the results
missing_info = pd.DataFrame({
'Data Type': column_datatypes,
'Missing Values': missing_values_sum,
'Percentage Missing (%)': missing_values_percentage
})

# Display the missing values sum and percentage
print(missing_info)
```

```python
train_df1.shape
```

Remove Outliers

```python
# Define a function to generate a box plot for a specific column
def generate_boxplot(data, column_name):
plt.figure(figsize=(8, 6))
sns.boxplot(y=data[column_name])

# Customize the y-axis tick labels
plt.gca().get_yaxis().get_major_formatter().set_scientific(False)

plt.ylabel(column_name)
plt.title(f'Box Plot of {column_name}')
plt.show()
```

```python
# Define a function to generate a histogram for a specific column
def generate_histogram(data, column_name):
plt.figure(figsize=(8, 6))
sns.histplot(data[column_name], bins=20, kde=True)  # Adjust the number of bins as needed

# Customize the y-axis tick labels
plt.gca().get_yaxis().get_major_formatter().set_scientific(False)

plt.xlabel(column_name)
```

```
plt.ylabel('Frequency')
plt.title(f'Histogram of {column_name}')
plt.show()
```

```python
# Define a function to generate a bar chart for a specific column
def generate_bar_chart(data, column_name):
data_counts = data[column_name].value_counts().sort_index()

# Create a bar plot
plt.figure(figsize=(20, 12))  # Adjust the figure size as needed
data_counts.plot(kind='bar', rot=0)  # 'rot' parameter controls x-axis label rotation

# Customize the y-axis tick labels
plt.gca().get_yaxis().get_major_formatter().set_scientific(False)

plt.xlabel(column_name)
plt.ylabel('Count')
plt.title(f'{column_name} Distribution')
plt.xticks(rotation=90)
plt.show()
```

Age Column Data Analysis

```python
# Call the function to generate a box plot for the 'Age' column
generate_boxplot(train_df1, 'Age')
generate_histogram(train_df1, 'Age')
# generate_bar_chart(train_df1, 'Age')
```

As per the above result, the Age column contains data that doesn’t seems to be practical. We must exclude those data considering the optimal age range to be 20 to 60 years old.

```python
train_df1 = train_df1[(train_df1['Age'] >= 20) & (train_df1['Age'] <= 60)]

train_df1.shape
```

```python
# Call the function to generate a box plot for the 'Age' column
generate_boxplot(train_df1, 'Age')
generate_histogram(train_df1, 'Age')
# generate_bar_chart(train_df1, 'Age')
```

Annual_Income Column Data Analysis

```python
# Annual_Income analysis
generate_boxplot(train_df1, 'Annual_Income')
generate_histogram(train_df1, 'Annual_Income')
# generate_bar_chart(train_df1, 'Annual_Income')
```

According to the above diagrams the Annual_Income is having some outliers. There are some values having more than 130000 and those data will be excluded to make it more accurate. Anyway, there are no any negatives.

```python
# Remove values greater that 130000
train_df1 = train_df1[(train_df1['Annual_Income'] >= 0) & (train_df1['Annual_Income'] <= 130000)]
```

```python
# Annual_Income analysis after outlier removals
generate_boxplot(train_df1, 'Annual_Income')
generate_histogram(train_df1, 'Annual_Income')
# generate_bar_chart(train_df1, 'Annual_Income')
```

```python
train_df1.shape
```

Monthly_Inhand_Salary Column Data Analysis

```python
# Monthly_Inhand_Salary analysis
generate_boxplot(train_df1, 'Monthly_Inhand_Salary')
generate_histogram(train_df1, 'Monthly_Inhand_Salary')
# generate_bar_chart(train_df1, 'Monthly_Inhand_Salary')
```

As per the diagrams the Monthly_Inhand_Salary is having a practical data distribution.

Num_Bank_Accounts Column Data Analysis

```python
# Num_Bank_Accounts analysis
generate_boxplot(train_df1, 'Num_Bank_Accounts')
generate_histogram(train_df1, 'Num_Bank_Accounts')
# generate_bar_chart(train_df1, 'Num_Bank_Accounts')
```

The evidence presented above doesn’t seem to be accurate and the outliers should be excluded.

```python
train_df1 = train_df1[(train_df1['Num_Bank_Accounts'] >= 1) & (train_df1['Num_Bank_Accounts'] <= 10)]
```

train_df1.shape

```

```python
# Num_Bank_Accounts analysis after cleanup
generate_boxplot(train_df1, 'Num_Bank_Accounts')
generate_histogram(train_df1, 'Num_Bank_Accounts')
# generate_bar_chart(train_df1, 'Num_Bank_Accounts')
```

Num_Credit_Card Column Data Analysis

```python
# Num_Credit_Card analysis
generate_boxplot(train_df1, 'Num_Credit_Card')
generate_histogram(train_df1, 'Num_Credit_Card')
# generate_bar_chart(train_df1, 'Num_Credit_Card')
```

According to the above visual result, it doesn’t seem to be showing accurate data. So, the outliers should be excluded.

```python
train_df1 = train_df1[(train_df1['Num_Credit_Card'] >= 1) & (train_df1['Num_Credit_Card'] <= 10)]
```
```

```python
# Num_Credit_Card analysis after cleanup
generate_boxplot(train_df1, 'Num_Credit_Card')
generate_histogram(train_df1, 'Num_Credit_Card')
# generate_bar_chart(train_df1, 'Num_Credit_Card')

```

```python
train_df1.shape
```

Interest_Rate Column Data Analysis

```python
# Interest_Rate analysis
generate_boxplot(train_df1, 'Interest_Rate')
generate_histogram(train_df1, 'Interest_Rate')
# generate_bar_chart(train_df1, 'Interest_Rate')

```

Generally, Interest_Rate should be a % and it should be always 0 to 100. But normally Interest_Rate can be very high value such as 80%. The data that appears to be impractical will be excluded.

```python
train_df1 = train_df1[(train_df1['Interest_Rate'] >= 1) & (train_df1['Interest_Rate'] < 40)]

train_df1.shape
```

```python
# Interest_Rate analysis after cleanup
generate_boxplot(train_df1, 'Interest_Rate')
generate_histogram(train_df1, 'Interest_Rate')
# generate_bar_chart(train_df1, 'Interest_Rate')

```

Num_of_Loan Column Data Analysis

```python
# Num_of_Loan analysis
generate_boxplot(train_df1, 'Num_of_Loan')
generate_histogram(train_df1, 'Num_of_Loan')
# generate_bar_chart(train_df1, 'Num_of_Loan')

```

Above Num_of_Loan is not appearing to be realistic practically in real world. So, those outliers should be excluded.

```python
train_df1 = train_df1[(train_df1['Num_of_Loan'] >= 1) & (train_df1['Num_of_Loan'] < 15)]

train_df1.shape
```

```python
# Num_of_Loan analysis after outlier removal
generate_boxplot(train_df1, 'Num_of_Loan')
generate_histogram(train_df1, 'Num_of_Loan')
# generate_bar_chart(train_df1, 'Num_of_Loan')
```

```python
train_df1.shape
```

Delay_from_due_date Column Data Analysis

```python
# Delay_from_due_date analysis
generate_boxplot(train_df1, 'Delay_from_due_date')
generate_histogram(train_df1, 'Delay_from_due_date')
```

generate_bar_chart(train_df1, 'Delay_from_due_date')

```

Seems there are few outliers in the Delay_from_due_date data. So, they will be excluded.

```python
train_df1 = train_df1[(train_df1['Delay_from_due_date'] >= -10) & (train_df1['Delay_from_due_date'] < 60)]

train_df1.shape
```

```python
# Delay_from_due_date analysis after cleanup
generate_boxplot(train_df1, 'Delay_from_due_date')
generate_histogram(train_df1, 'Delay_from_due_date')
# generate_bar_chart(train_df1, 'Delay_from_due_date')

```

Num_of_Delayed_Payment Column Data Analysis

```python
# Num_of_Delayed_Payment analysis
generate_boxplot(train_df1, 'Num_of_Delayed_Payment')

generate_histogram(train_df1, 'Num_of_Delayed_Payment')

# generate_bar_chart(train_df1, 'Num_of_Delayed_Payment')

```

Num_of_Delayed_Payment count is having impractical data as it appears to be. So, those anomalies will be excluded.

```python
train_df1 = train_df1[(train_df1['Num_of_Delayed_Payment'] >= -4) &
(train_df1['Num_of_Delayed_Payment'] < 72)]
```

```python
train_df1.shape
```

```python
# Num_of_Delayed_Payment analysis after cleanup

generate_boxplot(train_df1, 'Num_of_Delayed_Payment')

generate_histogram(train_df1, 'Num_of_Delayed_Payment')

# generate_bar_chart(train_df1, 'Num_of_Delayed_Payment')
```

Changed_Credit_Limit Column Data Analysis
```

```python
# Changed_Credit_Limit analysis
generate_boxplot(train_df1, 'Changed_Credit_Limit')
generate_histogram(train_df1, 'Changed_Credit_Limit')
# generate_bar_chart(train_df1, 'Changed_Credit_Limit')
```

Changed_Credit_Limit having a practical data distribution.

Outstanding_Debt Column Data Analysis

```python
# Outstanding_Debt analysis
generate_boxplot(train_df1, 'Outstanding_Debt')
generate_histogram(train_df1, 'Outstanding_Debt')
# generate_bar_chart(train_df1, 'Outstanding_Debt')
```

Outstanding_Debt having a practical data distribution.

Credit_Utilization_Ratio Column Data Analysis

```python
# Credit_Utilization_Ratio analysis after cleanup
generate_boxplot(train_df1, 'Credit_Utilization_Ratio')
generate_histogram(train_df1, 'Credit_Utilization_Ratio')
```

generate_bar_chart(train_df1, 'Credit_Utilization_Ratio')

```

Credit_Utilization_Ratio having a practical data distribution.

## Total_EMI_per_month Column Data Analysis

```python
# Total_EMI_per_month analysis
generate_boxplot(train_df1, 'Total_EMI_per_month')
generate_histogram(train_df1, 'Total_EMI_per_month')
# generate_bar_chart(train_df1, 'Total_EMI_per_month')
```

As per the box plot, seems Total_EMI_per_month is having outliers and those will be excluded.

```python
train_df1['Total_EMI_per_month'].value_counts().sort_index()
```

```python
# Remove Total_EMI_per_month > 350

train_df1 = train_df1[(train_df1['Total_EMI_per_month'] >= 0) & (train_df1['Total_EMI_per_month'] < 350)]
```
```

```

```python
train_df1.shape
```

```python
# Total_EMI_per_month analysis after cleanup
generate_boxplot(train_df1, 'Total_EMI_per_month')
generate_histogram(train_df1, 'Total_EMI_per_month')
# generate_bar_chart(train_df1, 'Total_EMI_per_month')
```

Amount_invested_monthly Column Data Analysis

```python
# Amount_invested_monthly analysis
generate_boxplot(train_df1, 'Amount_invested_monthly')
generate_histogram(train_df1, 'Amount_invested_monthly')
# generate_bar_chart(train_df1, 'Amount_invested_monthly')
```

It appears to be that there are outliers according to the boxplot, as there is no any instance from 2000 to nearly 9000. So, the outliers will be excluded and the data will be taken within the range of 0 to 500 to be more practical.
```

```python
train_df1 = train_df1[(train_df1['Amount_invested_monthly'] >= 0) &
(train_df1['Amount_invested_monthly'] < 500)]
```

```python
# Amount_invested_monthly analysis after cleanup
generate_boxplot(train_df1, 'Amount_invested_monthly')
generate_histogram(train_df1, 'Amount_invested_monthly')
# generate_bar_chart(train_df1, 'Amount_invested_monthly')
```

Monthly_Balance Column Data Analysis

```python
# Monthly_Balance analysis
generate_boxplot(train_df1, 'Monthly_Balance')
generate_histogram(train_df1, 'Monthly_Balance')
# generate_bar_chart(train_df1, 'Monthly_Balance')
```

Seems Monthly_Balance is having huge negative values as well as huge positive values so those data will be excluded.

```python
```

```
train_df1 = train_df1[(train_df1['Monthly_Balance'] >= 0) & (train_df1['Monthly_Balance'] < 700)]
```

```python
# Monthly_Balance analysis after cleanup
generate_boxplot(train_df1, 'Monthly_Balance')
generate_histogram(train_df1, 'Monthly_Balance')
# generate_bar_chart(train_df1, 'Monthly_Balance')
```

```python
file_path = 'df_clean.csv'
train_df1.to_csv(file_path, index=False)
```

```python
train_df1.shape
```

Feature Correlations with the Target Variable

```python
df = train_df1
```

ANOVA p-values Verification

This test will help to understand correlataions or any relations with Numerical Columns and Credit_Score (Catagorical)

```python
# List of numerical columns (excluding 'Credit_Score')
numerical_columns = df.select_dtypes(include=['int32', 'int64', 'float64']).columns.tolist()
```

```python
numerical_columns
```

```python
# Perform ANOVA for each numerical feature
p_values = {}
for col in numerical_columns:
groups = [df[df['Credit_Score'] == val][col] for val in df['Credit_Score'].unique()]
f_statistic, p_value = f_oneway(*groups)
p_values[col] = p_value
```

```python
# Display p-values
```

```
print("ANOVA p-values:")
for col, p_value in p_values.items():
print(f"{col}: {p_value:.4f}")
```

```python
# Convert the correlation and ANOVA results to DataFrames for plotting
corr_df = pd.DataFrame(p_values.items(), columns=['Feature', 'Correlation'])
anova_df = pd.DataFrame(p_values.items(), columns=['Feature', 'ANOVA_P_Value'])
```

```python
corr_df
```

```python
anova_df
```

```python
# Create subplots for correlation and ANOVA p-value bar charts
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))

# Bar chart for correlations
```

axes[0].barh(corr_df['Feature'], corr_df['Correlation'], color='skyblue')

axes[0].set_xlabel('Correlation')

axes[0].set_title('Correlations with Credit_Score')

Bar chart for ANOVA p-values

axes[1].barh(anova_df['Feature'], anova_df['ANOVA_P_Value'], color='lightcoral')

axes[1].set_xlabel('ANOVA P-Value')

axes[1].set_title('ANOVA P-Values for Numerical Features')

plt.tight_layout()

plt.show()

```

#### It is possible to consider that column to be a correlated column if the P-Value is less than 0.05 (5%). As a result, all columns appear to be related to the credit score, apart from Credit_Utilization_Ratio.

### Chi-Square p-values Verification

#### This test will help to understand correlataions or any relations with Categoricle Columns and Credit_Score (Catagorical)

```python

# List of categorical columns (excluding 'Credit_Score')

categorical_columns = df.select_dtypes(include=['object']).columns.tolist()

categorical_columns.remove('Credit_Score')

```
```

```python
# Perform chi-squared test for each categorical feature
chi2_results = {}
for col in categorical_columns:
# Create a contingency table
contingency_table = pd.crosstab(df[col], df['Credit_Score'])
chi2, p_value, _, _ = chi2_contingency(contingency_table)
chi2_results[col] = {'Chi-Square': chi2, 'p-value': p_value}
```

```python
# Display chi-squared test results
print("Chi-Squared Test Results:")
for col, results in chi2_results.items():
print(f"{col}:")
print(f"  Chi-Square: {results['Chi-Square']:.4f}")
print(f"  p-value: {results['p-value']:.4f}")
```

```python
# Convert chi2_results to a DataFrame for easier plotting
chi2_df = pd.DataFrame(chi2_results).T
```

```python
chi2_df
```

```python
# Create a bar chart for p-values
plt.figure(figsize=(10, 6))
plt.bar(chi2_df.index, chi2_df['p-value'], color='skyblue')
plt.xlabel('Categorical Column')
plt.ylabel('p-value')
plt.title('Chi-Squared Test p-values for Categorical Columns vs. Credit_Score')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

It is possible to consider a column to be a correlated column if the P-Value is less than 0.05 (5%). Each of the categorical columns (Occupation, Type of Loan, Credit Mix, Credit History Age, Payment of Minimum Amount, Payment Behavior) has a substantial correlation with the dependent categorical variable "Credit Score," as shown by the fact that all p-values are very close to 0

Feature Selection

```python
# Acording to the ANNOVA P-Value test, remove Credit_Utilization_Ratio column
```

train_df1 = train_df1.drop('Credit_Utilization_Ratio', axis=1)

```

```python
# Acording to the Chi-Squair P-Value test, remove Occupation column
# train_df1 = train_df1.drop('Occupation', axis=1)
```

```python
train_df1.dtypes
```

Decouple indemendet & dependent variables in to two sets

```python
df_indi = train_df1.drop(['Credit_Score'], axis = 1)
df_depend = train_df1['Credit_Score']
```

```python
df_depend
```

Data Normalization & Categorical Encoding
```

We are following 2 steps. 1st, will do the normalizaton for the numerical fields using MinMaxScaler(). and as the 2nd step will do the Categorical Encording.

To do this, need to extract numerical and categorical columns in to two sets.

```python
# Normalize numerical columns using Min-Max scaling (scaling to the range [0, 1])
numerical_columns = df_indi.select_dtypes(include=['number']).columns.tolist()
categorical_columns = df_indi.select_dtypes(include=['object']).columns.tolist()
```

```python
numerical_scaler = MinMaxScaler()
```

```python
df_indi[numerical_columns] = numerical_scaler.fit_transform(df_indi[numerical_columns])
```

```python
# Encode categorical variables
label_encoders = {}
for col in categorical_columns:
```

label_encoder = LabelEncoder()

df_indi[col] = label_encoder.fit_transform(df_indi[col])

label_encoders[col] = label_encoder

```

```python

df_indi

```

Modeling

```python

# Split data frames in to training & test (70% 30%)

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(df_indi,df_depend, test_size = 0.30, random_state = 100)

```

Logistic Regression Model

```python

# Create and train a Logistic Regression model

LR_model = LogisticRegression(max_iter=7000)

LR_model.fit(X_train, y_train)

```
```

```python
# Make predictions on the test set
LR_y_pred = LR_model.predict(X_test)
```

```python
LR_train_preds = LR_model.predict(X_train)
LR_train_preds
```

```python
LR_test_preds = LR_model.predict(X_test)
LR_test_preds
```

```python
LR_train_preds_score = LR_model.score(X_train,y_train)
LR_train_preds_score
```

```python
LR_test_preds_score = LR_model.score(X_test,y_test)
LR_test_preds_score
```

```

```python
# Compute the confusion matrix
conf_matrix = confusion_matrix(y_test, LR_y_pred)
```

```python
# Create a DataFrame for the confusion matrix (optional)
confusion_df = pd.DataFrame(conf_matrix, index=LR_model.classes_, columns=LR_model.classes_)

```

```python
# Create a heatmap to visualize the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_df, annot=True, fmt='d', cmap='Blues', linewidths=.5, cbar=False)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

```python

```
# Evaluate the model
accuracy = accuracy_score(y_test, LR_y_pred)
classification_rep = classification_report(y_test, LR_y_pred)
confusion_mat = confusion_matrix(y_test, LR_y_pred)
```

```python
# Print the results
print(f"Accuracy: {accuracy}")
print("Classification Report:\n", classification_rep)
print("Confusion Matrix:\n", confusion_mat)
```

Random Forest Classifier Model

```python
# Create and train a Random Forest Classifier model
RFC_model = RandomForestClassifier()
RFC_model.fit(X_train, y_train)
```

```python
# Make predictions on the test set
RFC_y_pred =RFC_model.predict(X_test)
```

```python
RFC_train_preds = RFC_model.predict(X_train)
RFC_train_preds
```

```python
RFC_test_preds = RFC_model.predict(X_test)
RFC_test_preds
```

```python
RFC_train_preds_score = RFC_model.score(X_train,y_train)
RFC_train_preds_score
```

```python
RFC_test_preds_score = RFC_model.score(X_test,y_test)
RFC_test_preds_score
```

```python
# Compute the confusion matrix
```

conf_matrix_rfc = confusion_matrix(y_test, RFC_y_pred)

```

```python

# Create a DataFrame for the confusion matrix (optional)

confusion_df_rfc = pd.DataFrame(conf_matrix_rfc, index=RFC_model.classes_, columns=RFC_model.classes_)

```

```python

# Create a heatmap to visualize the confusion matrix

plt.figure(figsize=(8, 6))

sns.heatmap(confusion_df_rfc, annot=True, fmt='d', cmap='Blues', linewidths=.5, cbar=False)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()
```

```python

# Evaluate the model

accuracy_rfc = accuracy_score(y_test, RFC_y_pred)

classification_rep_rfc = classification_report(y_test, RFC_y_pred)

confusion_mat_rfc = confusion_matrix(y_test, RFC_y_pred)

```

```python
# Print the results
print(f"Accuracy: {accuracy_rfc:.2f}")
print("Classification Report:\n", classification_rep_rfc)
print("Confusion Matrix:\n", confusion_mat_rfc)
```

```python

```
```