PROBLEM STATEMENT

This data has credit transaction information of customers over time and can be used to rate a custome furhter credit card usage puposes. Using data, we can formuate a credit score in order for future credit and identify risky customers.

In [209]:
```
import pandas as pd


# Load the CSV file
file_path = 'Credit_score.csv'
df = pd.read_csv(file_path)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\2545253867.py:5: DtypeWarning: Columns (26) have mixed types. Specify dtype o
ort or set low_memory=False.
  df = pd.read_csv(file_path)
```

In [210]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 27 columns):
 #   Column                    Non-Null Count   Dtype  
---  ------                    --------------   -----  
 0   ID                        100000 non-null  object 
 1   Customer_ID               100000 non-null  object 
 2   Month                     100000 non-null  object 
 3   Name                      90015 non-null   object 
 4   Age                       100000 non-null  object 
 5   SSN                       100000 non-null  object 
 6   Occupation                100000 non-null  object 
 7   Annual_Income             100000 non-null  object 
 8   Monthly_Inhand_Salary     84998 non-null   float64
 9   Num_Bank_Accounts         100000 non-null  int64  
 10  Num_Credit_Card           100000 non-null  int64  
 11  Interest_Rate             100000 non-null  int64  
 12  Num_of_Loan               100000 non-null  object 
 13  Type_of_Loan              88592 non-null   object 
 14  Delay_from_due_date       100000 non-null  int64  
 15  Num_of_Delayed_Payment    92998 non-null   object 
 16  Changed_Credit_Limit      100000 non-null  object 
 17  Num_Credit_Inquiries      98035 non-null   float64
 18  Credit_Mix                100000 non-null  object 
 19  Outstanding_Debt          100000 non-null  object 
 20  Credit_Utilization_Ratio  100000 non-null  float64
 21  Credit_History_Age        90970 non-null   object 
 22  Payment_of_Min_Amount     100000 non-null  object 
 23  Total_EMI_per_month       100000 non-null  float64
 24  Amount_invested_monthly   95521 non-null   object 
 25  Payment_Behaviour         100000 non-null  object 
 26  Monthly_Balance           98800 non-null   object 
dtypes: float64(4), int64(4), object(19)
memory usage: 20.6+ MB
```

This data will require missing data handling as a lot of columns have missing data.

NAME-COLOUMN

In [211]:
```
# Fill missing names using forward fill and backward fill within each 'Customer_ID'
df['Name'] = df.groupby('Customer_ID', group_keys=False)['Name'].apply(lambda x: x.ffill().bfill())
```

AGE-COLUMN

In [212]:
```
# Step 1: Replace non-numeric and negative values with NaN
df['Age'] = pd.to_numeric(df['Age'], errors='coerce')  # Convert to numeric
df['Age'] = df['Age'].where(df['Age'] >= 0)  # Keep only non-negative ages; others become NaN

# Step 2: Define a function to fill the age based on mode
def fill_age_with_mode(group):
    # Calculate mode of valid ages (ignoring NaN)
    mode_age = group.mode()
    if not mode_age.empty:
        mode_age_value = mode_age[0]  # Get the first mode value
        return group.fillna(mode_age_value)  # Fill NaN with the mode
    return group  # If there's no mode, return the group unchanged

# Step 3: Apply the mode filling within each Customer_ID group
df['Age'] = df.groupby('Customer_ID', group_keys=False)['Age'].apply(fill_age_with_mode)
```

ANNUAL_INCOME

In [213]:
```
# Step 1: Replace non-numeric and negative values with NaN
df['Annual_Income'] = pd.to_numeric(df['Annual_Income'], errors='coerce')  # Convert to numeric
# df['Annual_Income'] = df['Age'].where(df['Age'] >= 0)  # Keep only non-negative ages; others becc

# Step 2: Define a function to fill the age based on mode
def fill_annual_income_with_mode(group):
    # Calculate mode of valid ages (ignoring NaN)
    mode_income = group.mode()
    if not mode_income.empty:
        mode_income_value = mode_income[0]  # Get the first mode value
        return group.fillna(mode_income_value)  # Fill NaN with the mode
    return group  # If there's no mode, return the group unchanged

# Step 3: Apply the mode filling within each Customer_ID group
df['Annual_Income'] = df.groupby('Customer_ID', group_keys=False)['Annual_Income'].apply(fill_annua
```

In [214]:
```
# Step 1: Replace non-numeric and negative values with NaN
df['Monthly_Inhand_Salary'] = pd.to_numeric(df['Monthly_Inhand_Salary'], errors='coerce')  # Conver
# df['Annual_Income'] = df['Age'].where(df['Age'] >= 0)  # Keep only non-negative ages; others becc

# Step 2: Define a function to fill the age based on mode
def fill_monthly_income_with_mode(group):
    # Calculate mode of valid ages (ignoring NaN)
    mode_income = group.mode()
    if not mode_income.empty:
        mode_income_value = mode_income[0]  # Get the first mode value
        return group.fillna(mode_income_value)  # Fill NaN with the mode
    return group  # If there's no mode, return the group unchanged

# Step 3: Apply the mode filling within each Customer_ID group
df['Monthly_Inhand_Salary'] = df.groupby('Customer_ID', group_keys=False)['Monthly_Inhand_Salary'].
```

In [215]:
```
# Step 1: Remove underscores and convert to integers
df['Num_of_Delayed_Payment'] = df['Num_of_Delayed_Payment'].str.replace('_', '', regex=False)  # Re

# Step 2: Convert to numeric and handle errors
df['Num_of_Delayed_Payment'] = pd.to_numeric(df['Num_of_Delayed_Payment'], errors='coerce')  # Conv
# Step 3: Fill NaN values with 0
df['Num_of_Delayed_Payment'].fillna(0, inplace=True)
# Step 3: Replace negative values with positive values
df['Num_of_Delayed_Payment'] = df['Num_of_Delayed_Payment'].abs()  # Take absolute values
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\890857431.py:7: FutureWarning: A value is trying to be set on a copy of a Dat
ries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Num_of_Delayed_Payment'].fillna(0, inplace=True)
```

In [216]:
```
import pandas as pd



# Step 1: Replace '_' with 'Unspecified'
df['Credit_Mix'] = df['Credit_Mix'].replace('_', 'Unspecified')

# Step 2: Fill NaN values with 'Unspecified'
df['Credit_Mix'].fillna('Unspecified', inplace=True)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\926344742.py:8: FutureWarning: A value is trying to be set on a copy of a Dat
ries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Credit_Mix'].fillna('Unspecified', inplace=True)
```

In [217]:
```
df['Occupation'] = df.groupby('Customer_ID', group_keys=False)['Occupation'].apply(lambda x: x.ffil
```

In [218]:
```
# Step 1: Remove underscores and convert to integers
df['Outstanding_Debt'] = df['Outstanding_Debt'].str.replace('_', '', regex=False)  # Remove undersc

# Step 2: Convert to numeric and handle errors
df['Outstanding_Debt'] = pd.to_numeric(df['Outstanding_Debt'], errors='coerce')  # Convert to numer
# Step 3: Fill NaN values with 0
df['Outstanding_Debt'].fillna(0, inplace=True)
# Step 3: Replace negative values with positive values
df['Outstanding_Debt'] = df['Outstanding_Debt'].abs()  # Take absolute values
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\714176131.py:7: FutureWarning: A value is trying to be set on a copy of a Dat
ries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Outstanding_Debt'].fillna(0, inplace=True)
```

In [219]:
```
# Step 1: Replace unwanted values with 'Unspecified_spent_Unspecified_value_payments'
df['Payment_Behaviour'] = df['Payment_Behaviour'].replace({'': 'Unspecified_spent_Unspecified_value

# Step 2: Fill NaN values with 'Unspecified_spent_Unspecified_value_payments'
df['Payment_Behaviour'].fillna('Unspecified_spent_Unspecified_value_payments', inplace=True)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\176375327.py:5: FutureWarning: A value is trying to be set on a copy of a Dat
ries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Payment_Behaviour'].fillna('Unspecified_spent_Unspecified_value_payments', inplace=True)
```

In [220]:
```
df['Amount_invested_monthly'] = pd.to_numeric(df['Amount_invested_monthly'], errors='coerce')
df['Amount_invested_monthly'].fillna('Unspecified', inplace=True)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\3139593598.py:2: FutureWarning: A value is trying to be set on a copy of a Da
eries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Amount_invested_monthly'].fillna('Unspecified', inplace=True)
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\3139593598.py:2: FutureWarning: Setting an item of incompatible dtype is depr
ill raise an error in a future version of pandas. Value 'Unspecified' has dtype incompatible with float64, please explicitly c
patible dtype first.
  df['Amount_invested_monthly'].fillna('Unspecified', inplace=True)
```

In [221]:
```
df['Num_of_Delayed_Payment'] = pd.to_numeric(df['Num_of_Delayed_Payment'], errors='coerce')
df['Num_of_Delayed_Payment'].fillna(0, inplace=True)
df['Num_of_Delayed_Payment']=df['Num_of_Delayed_Payment'].abs()
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\3624863341.py:2: FutureWarning: A value is trying to be set on a copy of a Da
eries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Num_of_Delayed_Payment'].fillna(0, inplace=True)
```

In [222]:
```
df['Num_Credit_Inquiries'].fillna(0,inplace=True)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\3137116375.py:1: FutureWarning: A value is trying to be set on a copy of a Da
eries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Num_Credit_Inquiries'].fillna(0,inplace=True)
```

In [223]:
```
# Step 1: Remove underscores and convert to integers
df['Num_of_Loan'] = df['Num_of_Loan'].str.replace('_', '', regex=False)  # Remove underscores

# Step 2: Convert to numeric and handle errors
df['Num_of_Loan'] = pd.to_numeric(df['Num_of_Loan'], errors='coerce')  # Convert to numeric
# Step 3: Fill NaN values with 0
df['Num_of_Loan'].fillna(0, inplace=True)
# Step 3: Replace negative values with positive values
df['Num_of_Loan'] = df['Num_of_Loan'].abs()  # Take absolute values
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\1340985954.py:7: FutureWarning: A value is trying to be set on a copy of a Da
eries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Num_of_Loan'].fillna(0, inplace=True)
```

In [224]:
```
df['Monthly_Balance'] = pd.to_numeric(df['Monthly_Balance'], errors='coerce')
df['Monthly_Balance'].fillna(0,inplace=True)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\1958528288.py:2: FutureWarning: A value is trying to be set on a copy of a Da
eries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Monthly_Balance'].fillna(0,inplace=True)
```

In [225]:
```
# Step 1: Remove underscores and convert to integers
df['Changed_Credit_Limit'] = df['Changed_Credit_Limit'].str.replace('_', '', regex=False)  # Remove

# Step 2: Convert to numeric and handle errors
df['Changed_Credit_Limit'] = pd.to_numeric(df['Changed_Credit_Limit'], errors='coerce')  # Convert
# Step 3: Fill NaN values with 0
df['Changed_Credit_Limit'].fillna(0, inplace=True)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\13115354.py:7: FutureWarning: A value is trying to be set on a copy of a Data
ies through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Changed_Credit_Limit'].fillna(0, inplace=True)
```

In [226]:
```
df['Type_of_Loan'].fillna('Not Specified',inplace = True)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\4038097645.py:1: FutureWarning: A value is trying to be set on a copy of a Da
eries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Type_of_Loan'].fillna('Not Specified',inplace = True)
```

In [227]:
```
df['Credit_History_Age'].fillna('NA',inplace =True)
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\482903698.py:1: FutureWarning: A value is trying to be set on a copy of a Dat
ries through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are se
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] =
od(value) instead, to perform the operation inplace on the original object.


  df['Credit_History_Age'].fillna('NA',inplace =True)
```

FEATURE CREATION

TYPE-OF-LOAN can be visualized as different columsn for better analysis.

In [228]:
```
# Step 1: Replace 'and' with ',' and split 'Type_of_Loan' into individual loans
df['Type_of_Loan'] = df['Type_of_Loan'].str.replace('and', ',')
df['Type_of_Loan_Split'] = df['Type_of_Loan'].str.split(',')

# Step 2: Get unique loan types
all_loan_types = set(df['Type_of_Loan_Split'].explode().str.strip())

# Step 3: Create columns for each loan type and mark with 1 or 0
for loan_type in all_loan_types:
    df[loan_type] = df['Type_of_Loan_Split'].apply(lambda x: 1 if loan_type in [i.strip() for i in

# Drop the temporary split column
df.drop(columns=['Type_of_Loan_Split'], inplace=True)

# Step 4: Apply the rule that if any loan type has a 1, 'Not Specified' must be 0
loan_type_columns = [col for col in df.columns if col != 'Not Specified']

# Set 'Not Specified' to 0 if any other loan type column has a 1
df['Not Specified'] = df.apply(lambda row: 0 if any(row[loan_type_columns] == 1) else row['Not Spec
```

In [229]:
```
df.drop(columns=[''], inplace=True)
```

CONVERTING CREDIT HISTORY AGE TO MONTHS FOR NUMERICAL ANALYSIS

In [230]:
```
import numpy as np
# Step 1: Define a function to convert the "Years and Months" format to total months
def convert_to_months(value):
    if pd.isna(value) or value == 'NA':
        return np.nan  # Return NA for 'NA' values

    # Try to split the value and extract years and months
    try:
        years = int(value.split(' Years and ')[0].strip())
        months = int(value.split(' Years and ')[1].replace(' Months', '').strip())
        return years * 12 + months  # Convert to total months
    except:
        return np.nan  # In case of any error, return NaN

# Step 2: Apply the conversion function to the 'Credit_History_Age' column
df['Credit_History_Months'] = df['Credit_History_Age'].apply(convert_to_months)
```

In [231]:
```
# df['Credit_History_Months'].fillna('NA', inplace=True)
df['Credit_History_Months'] = df.groupby('Customer_ID', group_keys=False)['Credit_History_Months'].
```

SPILTTING PAYMENT BEHAVIOUR INTO SPENT AND PAYMENTS

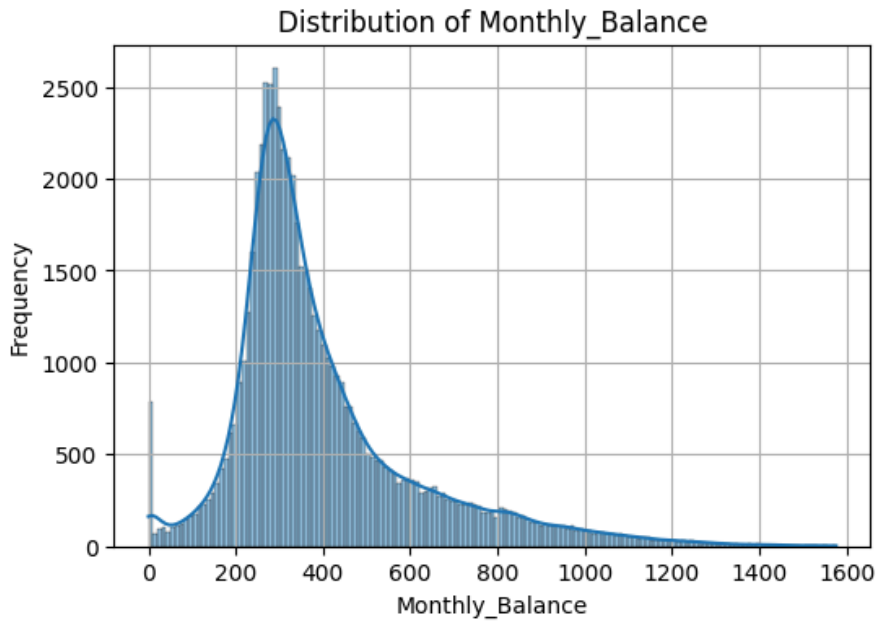
In [232]:
```
# Step 1: Split the 'Payment_Behaviour' column into three parts
df[['Expenditure', 'Value_of_Payments']] = df['Payment_Behaviour'].str.split('_spent_', expand=True

# Step 2: Further split 'Value_of_Payments' column to extract the actual payment size
df['Value_of_Payments'] = df['Value_of_Payments'].str.split('_value_payments').str[0]
```

In [233]:
```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 39 columns):
 #   Column                    Non-Null Count   Dtype  
---  ------                    --------------   -----  
 0   ID                        100000 non-null  object 
 1   Customer_ID               100000 non-null  object 
 2   Month                     100000 non-null  object 
 3   Name                      100000 non-null  object 
 4   Age                       100000 non-null  float64
 5   SSN                       100000 non-null  object 
 6   Occupation                100000 non-null  object 
 7   Annual_Income             100000 non-null  float64
 8   Monthly_Inhand_Salary     100000 non-null  float64
 9   Num_Bank_Accounts         100000 non-null  int64  
 10  Num_Credit_Card           100000 non-null  int64  
 11  Interest_Rate             100000 non-null  int64  
 12  Num_of_Loan               100000 non-null  int64  
 13  Type_of_Loan              100000 non-null  object 
 14  Delay_from_due_date       100000 non-null  int64  
 15  Num_of_Delayed_Payment    100000 non-null  float64
 16  Changed_Credit_Limit      100000 non-null  float64
 17  Num_Credit_Inquiries      100000 non-null  float64
 18  Credit_Mix                100000 non-null  object 
 19  Outstanding_Debt          100000 non-null  float64
 20  Credit_Utilization_Ratio  100000 non-null  float64
 21  Credit_History_Age        100000 non-null  object 
 22  Payment_of_Min_Amount     100000 non-null  object 
 23  Total_EMI_per_month       100000 non-null  float64
 24  Amount_invested_monthly   100000 non-null  object 
 25  Payment_Behaviour         100000 non-null  object 
 26  Monthly_Balance           100000 non-null  float64
 27  Personal Loan             100000 non-null  int64  
 28  Student Loan              100000 non-null  int64  
 29  Credit-Builder Loan       100000 non-null  int64  
 30  Home Equity Loan          100000 non-null  int64  
 31  Not Specified             100000 non-null  int64  
 32  Payday Loan               100000 non-null  int64  
 33  Mortgage Loan             100000 non-null  int64  
 34  Auto Loan                 100000 non-null  int64  
 35  Debt Consolidation Loan   100000 non-null  int64  
 36  Credit_History_Months     100000 non-null  float64
 37  Expenditure               100000 non-null  object 
 38  Value_of_Payments         100000 non-null  object 
dtypes: float64(11), int64(14), object(14)
memory usage: 29.8+ MB
```

In [234]:

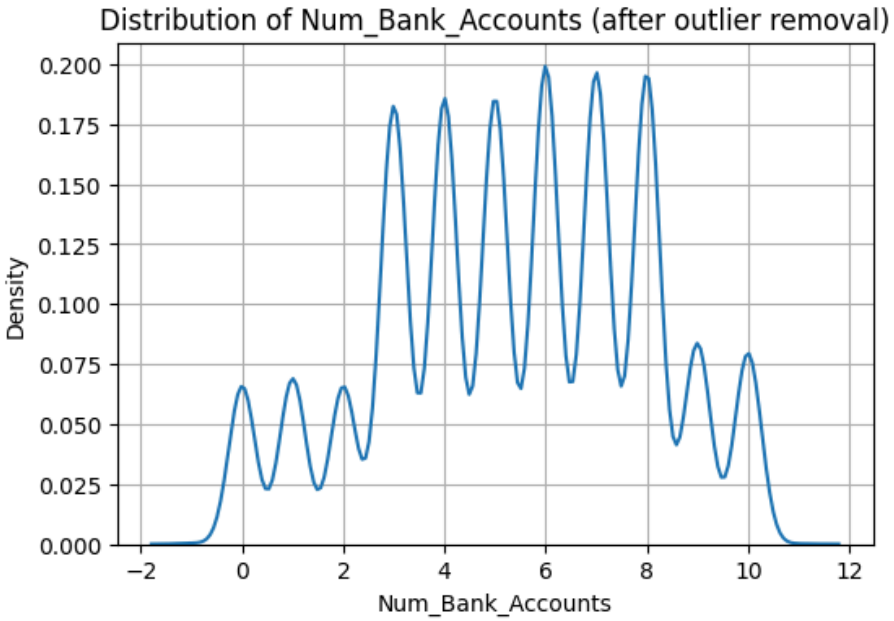
```
import matplotlib.pyplot as plt
import seaborn as sns


# Step 1: Make a copy of the DataFrame to avoid modifying the original
df_copy = df.copy()

# Step 2: Replace 'NA' and 'Unspecified' with NaN
df_copy.replace(['NA', 'Unspecified'], pd.NA, inplace=True)

grouped_df = df_copy.groupby('Customer_ID').apply(lambda group: group.dropna())

# Step 3: Iterate through each column and plot for each customer_id
for col in [ 'Num_Bank_Accounts',
'Monthly_Inhand_Salary',

'Interest_Rate',
'Num_of_Loan',

'Delay_from_due_date',
'Num_of_Delayed_Payment',
'Changed_Credit_Limit',
'Num_Credit_Inquiries',

'Outstanding_Debt',
'Credit_Utilization_Ratio',
'Credit_History_Age',


'Amount_invested_monthly',

'Monthly_Balance']:
    if col != 'Customer_ID':  # Skip the Customer_ID column
        # Try to convert column to numeric, errors='coerce' will turn non-numeric values to Na
        numeric_col = pd.to_numeric(grouped_df[col], errors='coerce')

        # Check if the column has any numeric data
        if numeric_col.notna().any():
            plt.figure(figsize=(6, 4))
            sns.histplot(numeric_col.dropna(), kde=True)  # Plot histogram with KDE
            plt.title(f"Distribution of {col}")
            plt.xlabel(col)
            plt.ylabel('Frequency')
            plt.grid(True)
            plt.show()
```

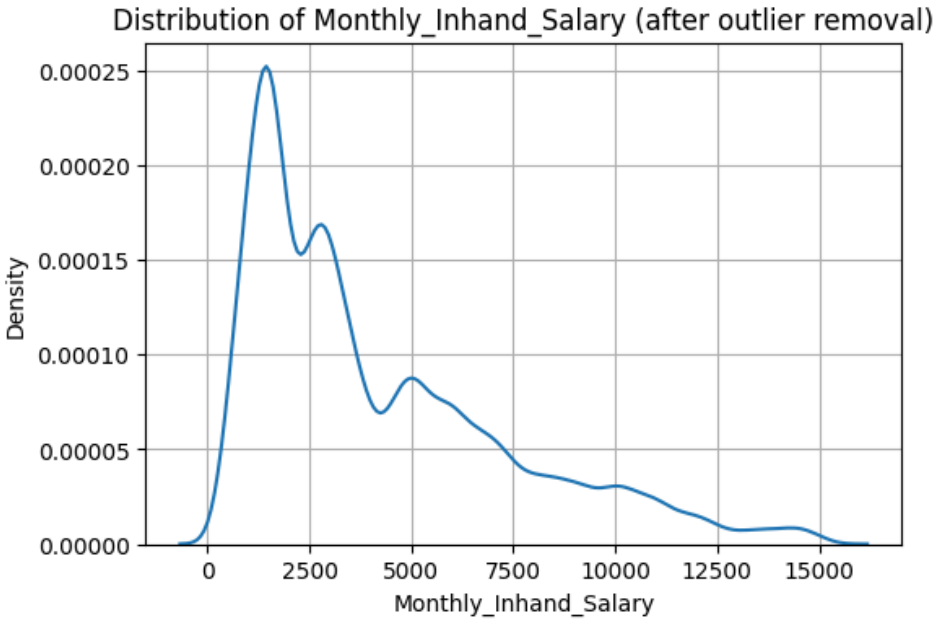

Distribution of Monthly_Balance
Frequency
2500
2000
1500
1000
500
0
0
200
400
600
800
1000
1200
1400
1600
Monthly_Balance

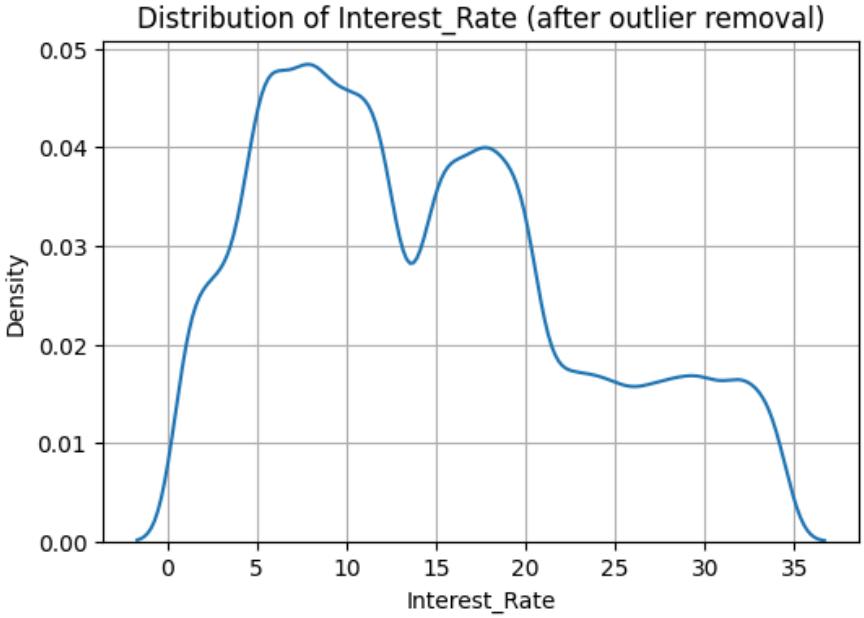
In [235]:

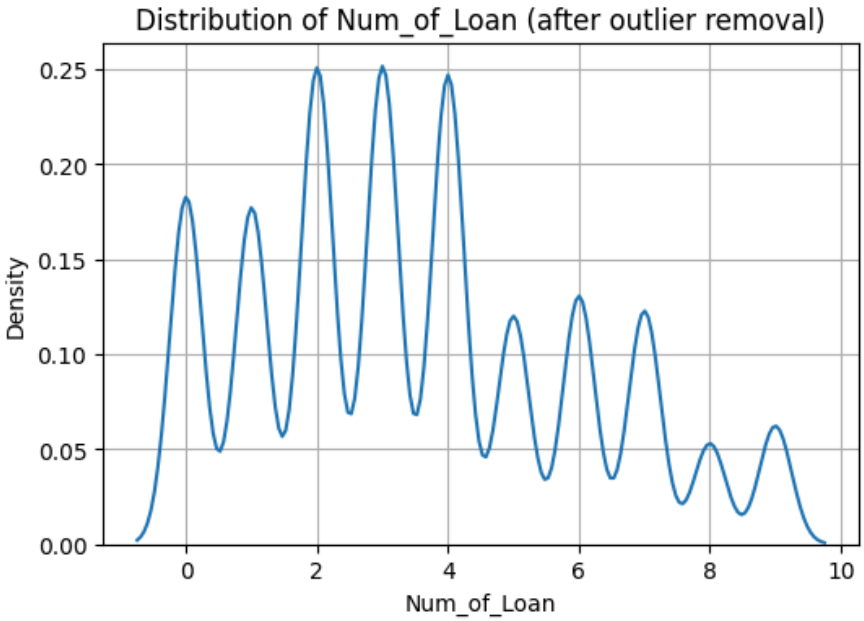
```
# Step 2: Define columns that are skewed and need outlier removal
columns_to_clean = [ 'Num_Bank_Accounts',
'Interest_Rate',
'Num_of_Loan',
'Num_of_Delayed_Payment',
'Num_Credit_Inquiries',]

# Step 3: Remove outliers using the IQR method for each of the identified columns
def remove_outliers(df, columns):
    for col in columns:
        # Calculate Q1 (25th percentile) and Q3 (75th percentile)
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1  # Interquartile range

        # Define the acceptable range
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        # Remove outliers (those outside the lower and upper bounds)
        df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]

    return df

# Remove outliers from the specified columns
df_cleaned = remove_outliers(df, columns_to_clean)

# Step 4: Group by Customer_ID and drop rows with NaN before plotting
grouped_df = df_cleaned.groupby('Customer_ID').apply(lambda group: group.dropna())

# Step 5: Iterate through each column and plot KDE for each customer_id
for col in ['Num_Bank_Accounts', 'Monthly_Inhand_Salary','Interest_Rate','Num_of_Loan','Delay_
'Num_of_Delayed_Payment','Changed_Credit_Limit','Num_Credit_Inquiries','Outstanding_Debt',
'Credit_Utilization_Ratio','Credit_History_Age','Amount_invested_monthly','Monthly_Balance']:
    if col != 'Customer_ID':  # Skip the Customer_ID column
        # Try to convert column to numeric, errors='coerce' will turn non-numeric values to Na
        numeric_col = pd.to_numeric(grouped_df[col], errors='coerce')

        # Check if the column has any numeric data
        if numeric_col.notna().any():
            plt.figure(figsize=(6, 4))
            sns.kdeplot(numeric_col.dropna())  # Plot KDE plot
            plt.title(f"Distribution of {col} (after outlier removal)")
            plt.xlabel(col)
            plt.ylabel('Density')
            plt.grid(True)
            plt.show()
```

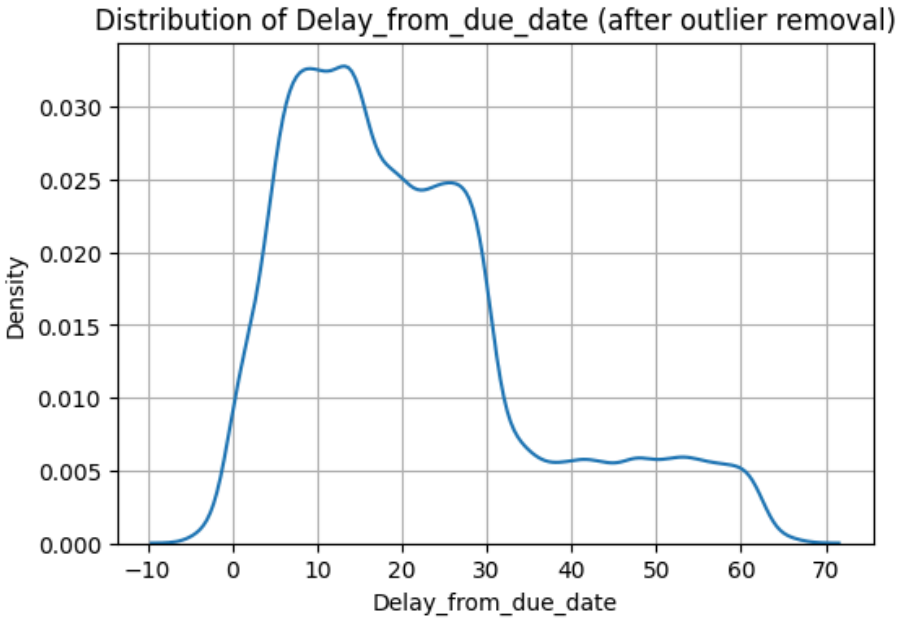
```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\1656327990.py:29: DeprecationWarning: DataFrameGroupBy.apply operated on the
umns. This behavior is deprecated, and in a future version of pandas the grouping columns will be excluded from the operation.
`include_groups=False` to exclude the groupings or explicitly select the grouping columns after groupby to silence this warnin
  grouped_df = df_cleaned.groupby('Customer_ID').apply(lambda group: group.dropna())
```

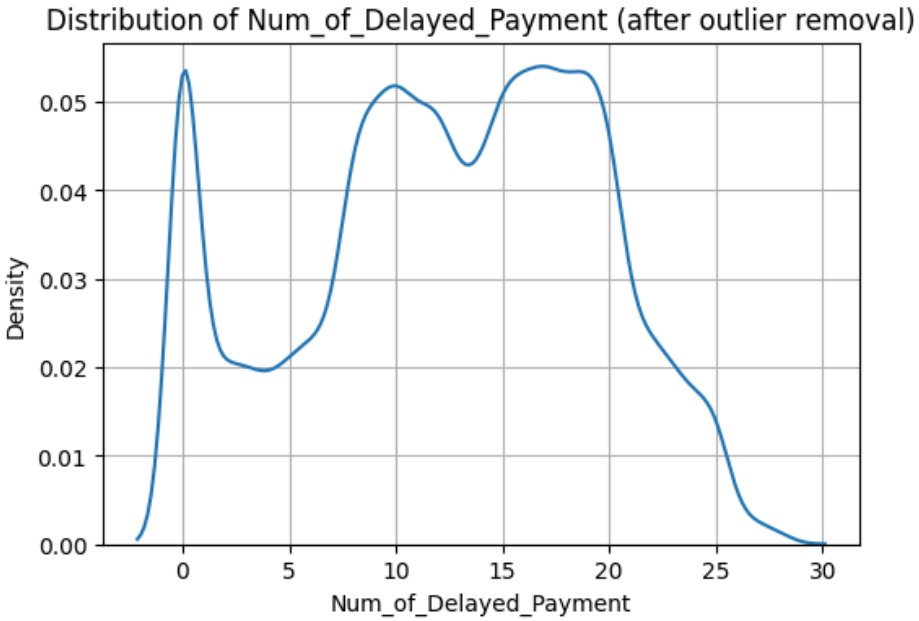


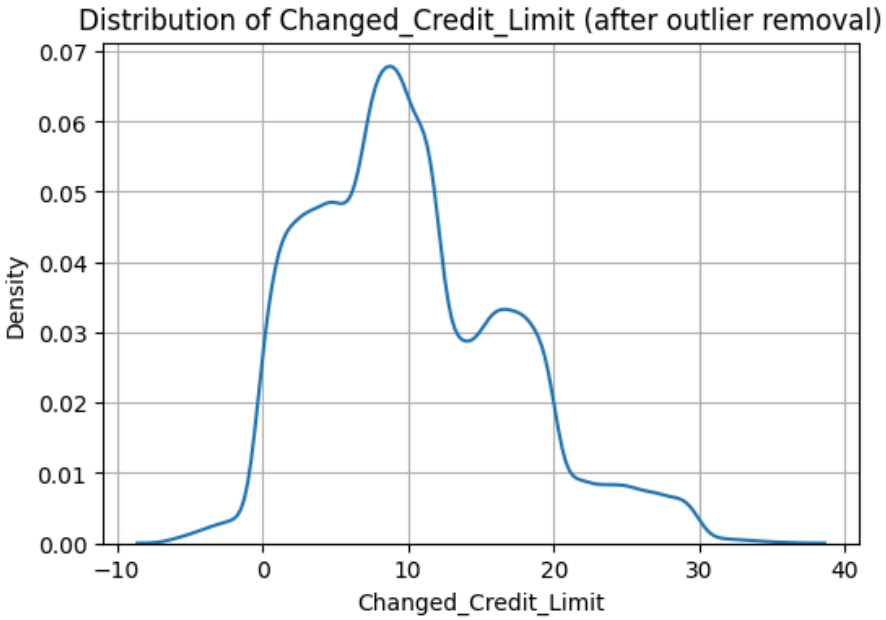


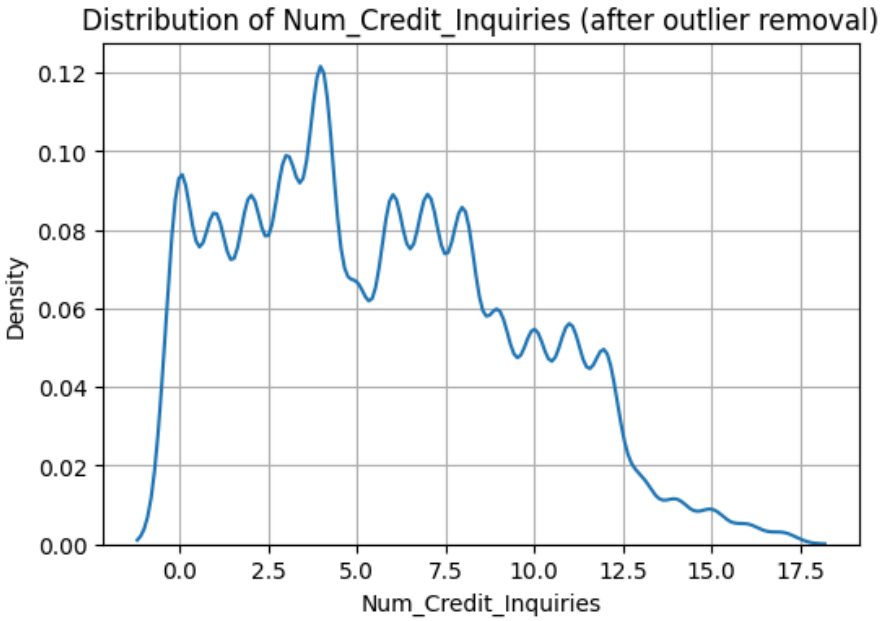

Distribution of Interest_Rate (after outlier removal)
Density
Interest_Rate
0.05
0.04
0.03
0.02
0.01
0.00
0
5
10
15
20
25
30
35

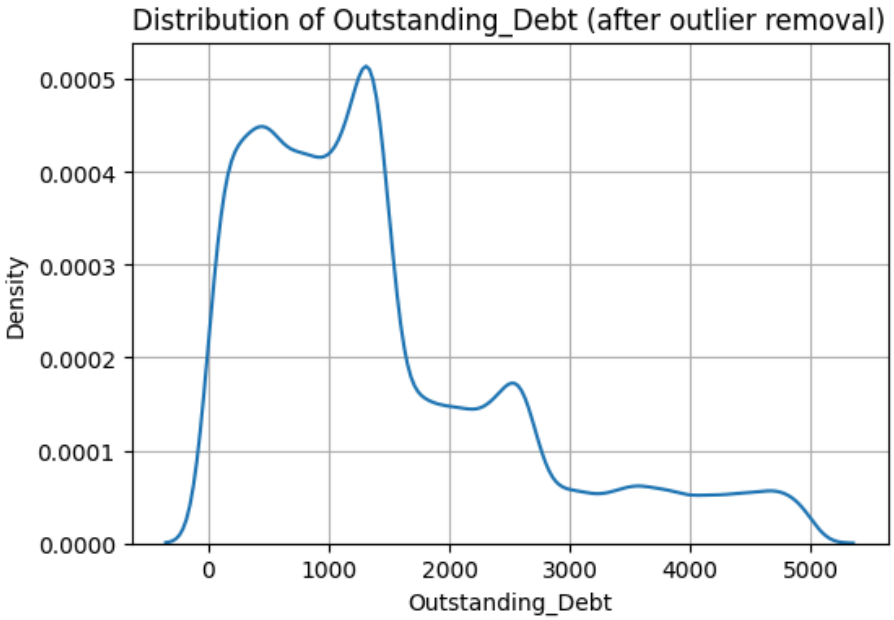

Distribution of Num_of_Loan (after outlier removal)
Density
Num_of_Loan
0.25
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10

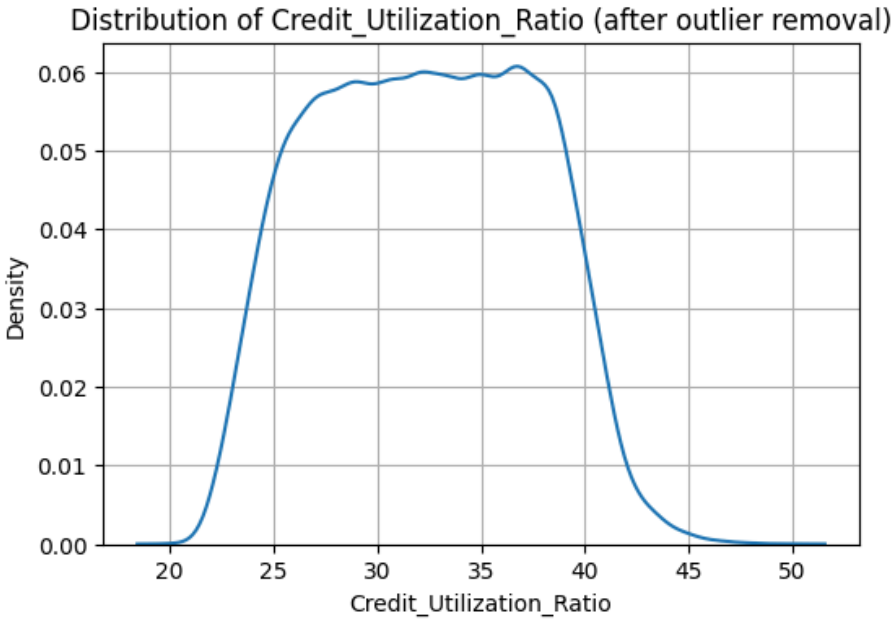

Distribution of Delay_from_due_date (after outlier removal)
Density
0.030
0.025
0.020
0.015
0.010
0.005
0.000
−10
0
10
20
30
40
50
60
70
Delay_from_due_date

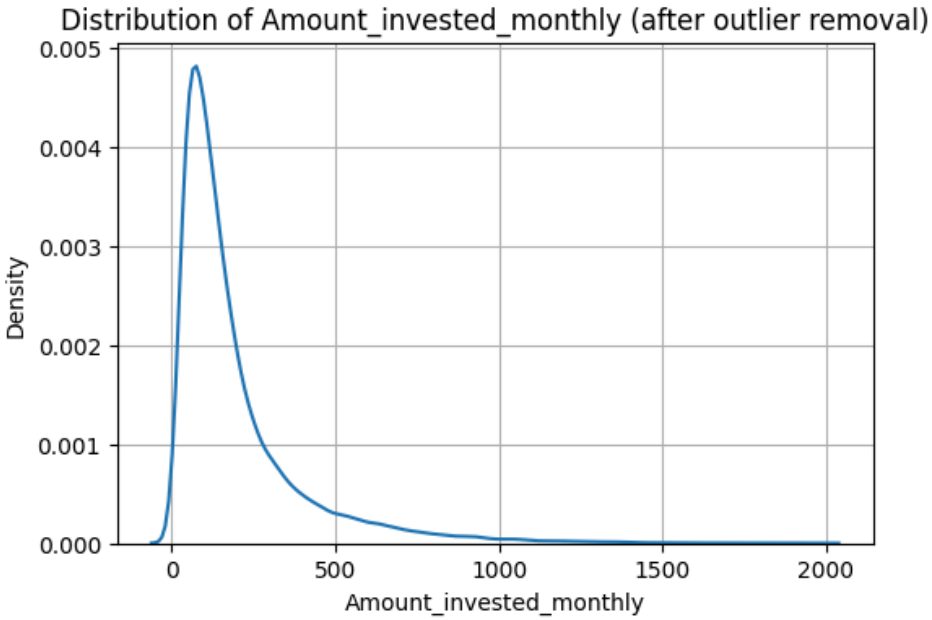

Distribution of Num_of_Delayed_Payment (after outlier removal)
Density
0.05
0.04
0.03
0.02
0.01
0.00
0
5
10
15
20
25
30
Num_of_Delayed_Payment

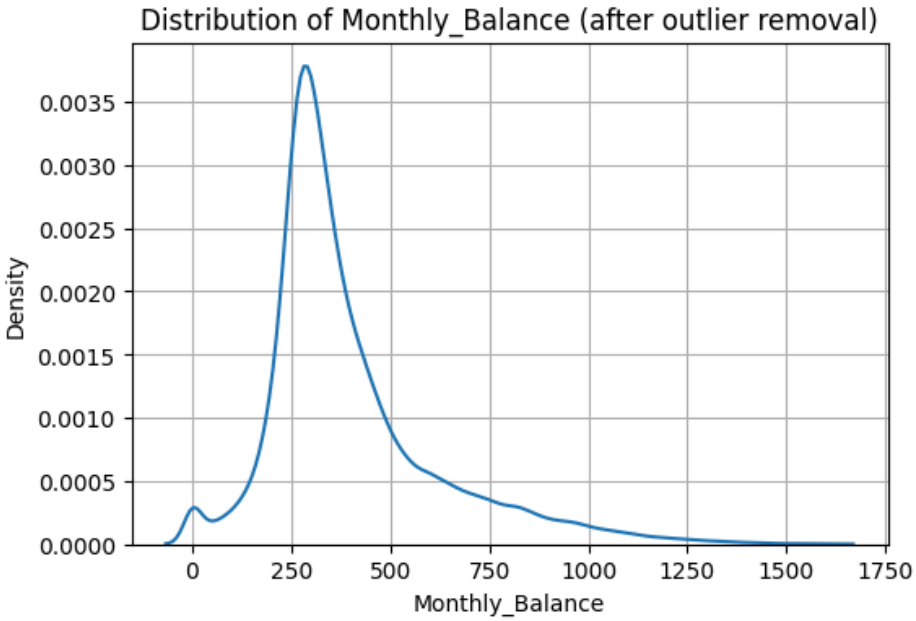

Distribution of Changed_Credit_Limit (after outlier removal)
Density
Changed_Credit_Limit
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−10
0
10
20
30
40


Distribution of Num_Credit_Inquiries (after outlier removal)
Density
Num_Credit_Inquiries
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Distribution of Outstanding_Debt (after outlier removal)
Density
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
0
1000
2000
3000
4000
5000
Outstanding_Debt

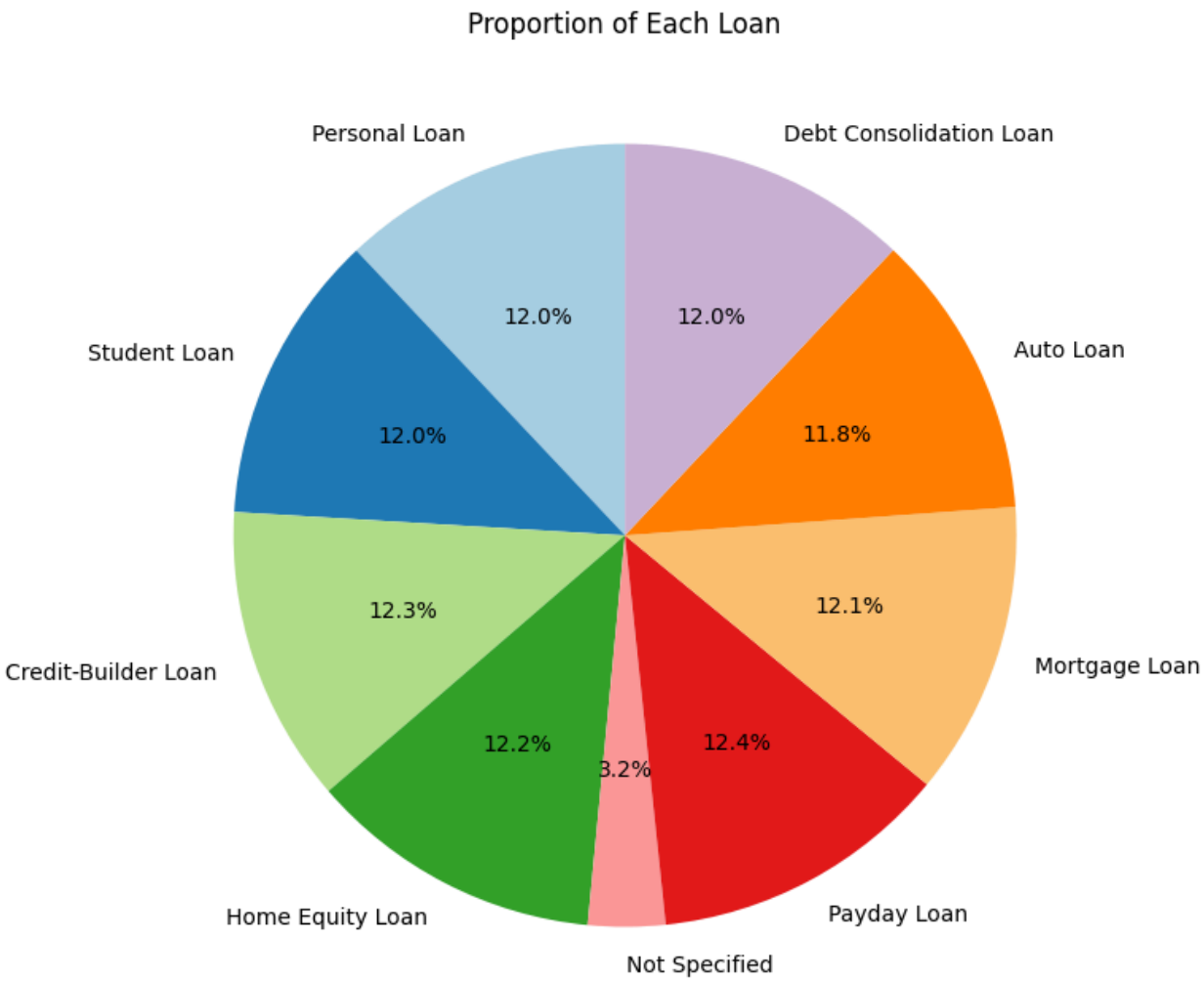

Distribution of Credit_Utilization_Ratio (after outlier removal)
Density
0.06
0.05
0.04
0.03
0.02
0.01
0.00
20
25
30
35
40
45
50
Credit_Utilization_Ratio


Distribution of Amount_invested_monthly (after outlier removal)
Density
0.005
0.004
0.003
0.002
0.001
0.000
0
500
1000
1500
2000
Amount_invested_monthly


Distribution of Monthly_Balance (after outlier removal)
Density
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
250
500
750
1000
1250
1500
1750
Monthly_Balance

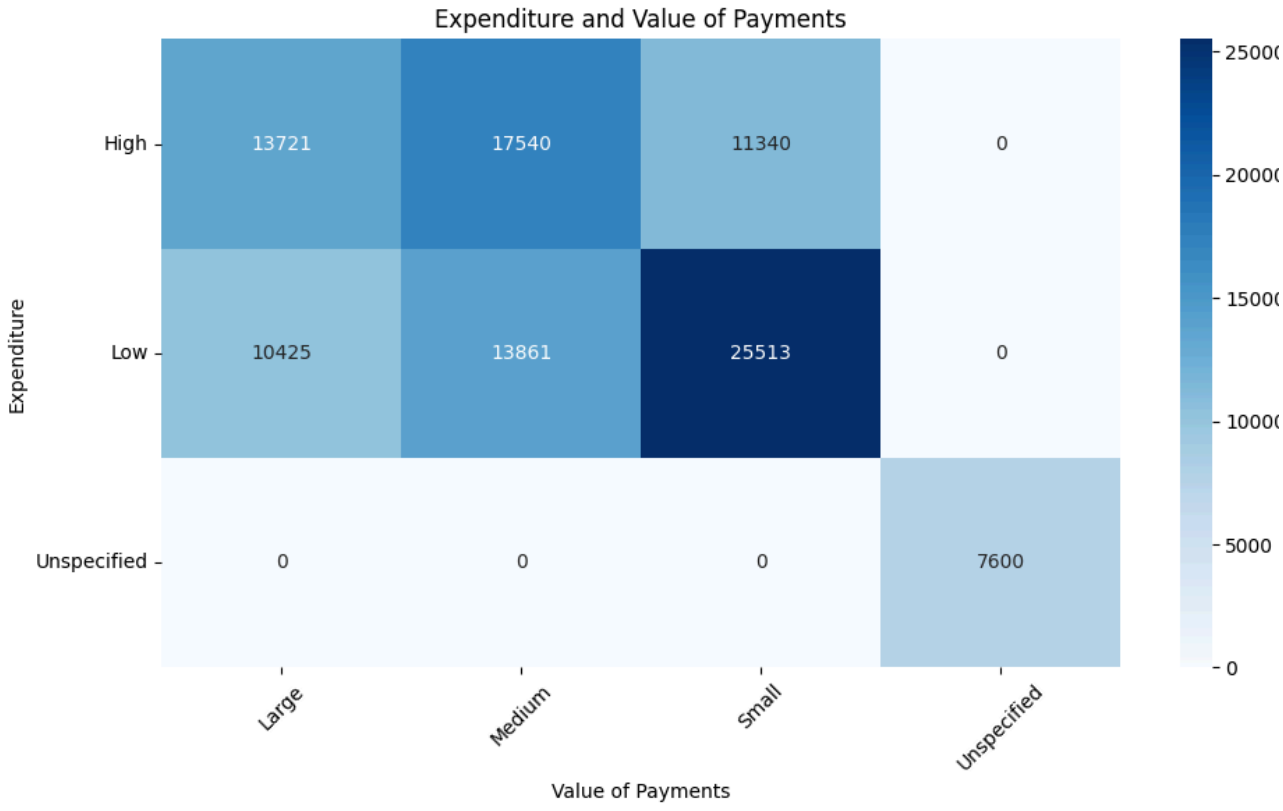
In [236]:

```
# Assuming df is your dataframe, and columns_with_1s is the list of columns with only 1 or 0 v
type_of_loans = ['Personal Loan', 'Student Loan', 'Credit-Builder Loan', 'Home Equity Loan',
 'Not Specified', 'Payday Loan', 'Mortgage Loan', 'Auto Loan', 'Debt Consolidation Loan']  # R

# Step 1: Calculate the count of 1s for each column
count_1s = df[type_of_loans].apply(lambda col: (col == 1).sum())

# Step 3: Plot Pie Chart (Optional)
plt.figure(figsize=(8, 8))
count_1s.plot(kind='pie', autopct='%1.1f%%', startangle=90, colors=plt.cm.Paired.colors)
plt.title('Proportion of Each Loan')
plt.ylabel('')  # Remove default y-axis label for pie chart
plt.show()
```



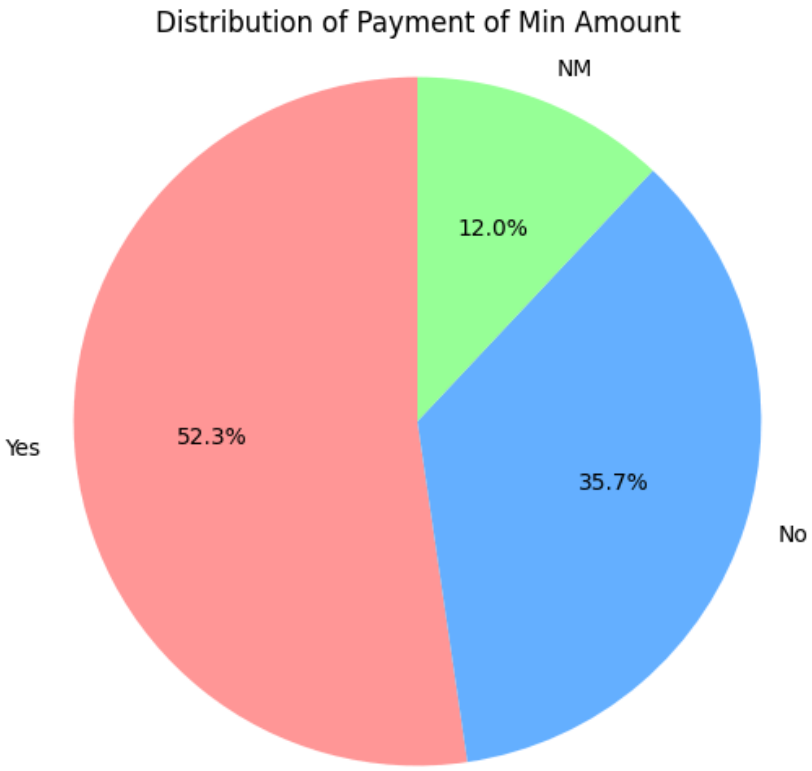
In [237]:

```
# Step 1: Convert 'Expenditure' and 'Value_of_Payments' to categorical columns
df['Expenditure'] = df['Expenditure'].astype('category')
df['Value_of_Payments'] = df['Value_of_Payments'].astype('category')

# Step 2: Create a crosstab between 'Expenditure' and 'Value_of_Payments'
crosstab_result = pd.crosstab(df['Expenditure'], df['Value_of_Payments'])

# Step 3: Visualize the crosstab result using a heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(crosstab_result, annot=True, fmt="d", cmap='Blues', cbar=True)

# Customizing the plot
plt.title('Expenditure and Value of Payments')
plt.xlabel('Value of Payments')
plt.ylabel('Expenditure')
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.tight_layout()

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



In [238]:

```
df['Payment_of_Min_Amount'] = df['Payment_of_Min_Amount'].astype('category')

# Step 1: Get the value counts of the 'Payment_of_Min_Amount' column
payment_counts = df['Payment_of_Min_Amount'].value_counts()

# Step 2: Plot a pie chart
plt.figure(figsize=(6, 6))
plt.pie(payment_counts, labels=payment_counts.index, autopct='%1.1f%%', startangle=90, colors=

# Step 3: Customizing the pie chart
plt.title('Distribution of Payment of Min Amount')
plt.axis('equal')  # Equal aspect ratio ensures that the pie chart is drawn as a circle.

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



EDA-INSIGHTS

1. Most credit transaction have resulted in Minimum amount being piad.
2. Of available data, Most credit card transactiona are high expenditure and repaid in small payment
3. Loans are qually divinde around 12% for most categories with Payday Loan being highest at 12.4
4. Most of Customer acheive 25-40% of credit utilization.
5. Number of Bank Accounts largely vary between 3 to 8.

CREDIT SCORING

I have considered annual income , credit utilixation ratio, credit history age as paositive factor for credi
calculation. 'Numbe of loans, Number of delayed payments, Outstanding debts and payment of minimu
as negative factors. The weightage

In [239]:

```
# Convert necessary columns to numeric, forcing errors to NaN
numeric_columns = ['Annual_Income', 'Num_of_Loan', 'Num_of_Delayed_Payment',
                   'Changed_Credit_Limit', 'Outstanding_Debt',
                   'Credit_Utilization_Ratio', 'Credit_History_Age']

df[numeric_columns] = df[numeric_columns].apply(pd.to_numeric, errors='coerce')

# Function to calculate the hypothetical credit score
def calculate_credit_score(group):
    score = 0

    # Scoring logic
    score += group['Annual_Income'].mean()   # Example: Adjust income (weights can be changed)
    score += (100 - group['Num_of_Loan'].mean() * 10)  # Subtract loans, scale appropriately
    score += (100 - group['Num_of_Delayed_Payment'].mean() * 20)  # Subtract delays, scale app
    score += group['Changed_Credit_Limit'].mean()  # Add positive limit changes
    score += (100 - (group['Outstanding_Debt'].mean()))  # Less debt is better, scale appropri
    score += (100 - group['Credit_Utilization_Ratio'].mean() * 100)  # Lower utilization is be
    score += (group['Credit_History_Months'].mean() )  # More history is better

    # Adjust score for Payment_of_Min_Amount
    if group['Payment_of_Min_Amount'].str.contains('Yes').any():
        score -= 15  # Increase penalty if minimum payment was made

    return score  # Return raw score

# Group by Customer_ID and calculate the raw score
df['Raw_Score'] = df.groupby('Customer_ID').apply(calculate_credit_score).reset_index(drop=Tru

# Determine min and max scores for normalization
min_score = df['Raw_Score'].min()+300
max_score = df['Raw_Score'].max()

# Normalize the scores to a specific range [300, 900]
def normalize_score(raw_score, min_score, max_score):
    # Ensure score is within bounds
    if raw_score < min_score:
        raw_score = min_score
    if raw_score > max_score:
        raw_score = max_score

    # Normalize to range [300, 900]
    normalized = 300 + ((raw_score - min_score) / (max_score - min_score)) * (900 - 300)
    return normalized

# Apply normalization
df['Credit_Score'] = df['Raw_Score'].apply(lambda x: normalize_score(x, min_score, max_score))

# Display the DataFrame with scores
```

```
[52] print(df[['Customer_ID', 'Credit_Score']].drop_duplicates())
```

```
C:\Users\31602\AppData\Local\Temp\ipykernel_8904\819692661.py:28: DeprecationWarning: DataFrameGroupBy.apply operated on the g
mns. This behavior is deprecated, and in a future version of pandas the grouping columns will be excluded from the operation.
`include_groups=False` to exclude the groupings or explicitly select the grouping columns after groupby to silence this warnin
  df['Raw_Score'] = df.groupby('Customer_ID').apply(calculate_credit_score).reset_index(drop=True)
```

	Customer_ID	Credit_Score
0	CUS_0xd40	302.656834
1	CUS_0xd40	304.114862
2	CUS_0xd40	309.426264
3	CUS_0xd40	305.099264
4	CUS_0xd40	308.210555
...	...	...
99960	CUS_0x372c	NaN
99968	CUS_0xf16	NaN
99976	CUS_0xaf61	NaN
99984	CUS_0x8600	NaN
99992	CUS_0x942c	NaN

[23438 rows x 2 columns]

```
[1] # RECOMMENDATIONS
```

```
[1] 1. Diversify the loans segements and focus low expenditure credit transaction.
[2] Improve large value payment benifit. This wil increase credit pay-back rate as well.
[3] Focus on Incresing credit utilixzation ration. A large part of customer is not even using 50%
```