```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
credit_data = pd.read_csv('/content/Credit_score.csv')
```

```
<ipython-input-91-70ee5507a100>:1: DtypeWarning: Columns (26) have mixed types. Specify dtype option on import or set low_memory=Fal
  credit_data = pd.read_csv('/content/Credit_score.csv')
```

```
credit_data.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
```

```
credit_data.head(10)
```

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	...
0	0x1602	CUS_0xd40	January	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843333	3	...
1	0x1603	CUS_0xd40	February	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	NaN	3	...
2	0x1604	CUS_0xd40	March	Aaron Maashoh	-500	821-00-0265	Scientist	19114.12	NaN	3	...
3	0x1605	CUS_0xd40	April	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	NaN	3	...
4	0x1606	CUS_0xd40	May	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843333	3	...
5	0x1607	CUS_0xd40	June	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	NaN	3	...
6	0x1608	CUS_0xd40	July	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843333	3	...
7	0x1609	CUS_0xd40	August	NaN	23	#F%$D@*&8	Scientist	19114.12	1824.843333	3	...
8	0x160e	CUS_0x21b1	January	Rick Rothackerj	28_	004-07-5839	_______	34847.84	3037.986667	2	...
9	0x160f	CUS_0x21b1	February	Rick Rothackerj	28	004-07-5839	Teacher	34847.84	3037.986667	2	...

10 rows × 27 columns

```
# Fill missing 'Name' values based on 'Customer_ID'
# First, create a dictionary of Customer_ID to Name mapping (excluding missing values)
id_to_name = credit_data[['Customer_ID', 'Name']].dropna().drop_duplicates(subset=['Customer_ID']).set_index('Customer_ID')['Name'].to_c

# Now, use this dictionary to fill missing names
credit_data['Name'] = credit_data.apply(
    lambda row: id_to_name[row['Customer_ID']] if pd.isna(row['Name']) and row['Customer_ID'] in id_to_name else row['Name'],
    axis=1
)

# Check if the missing values in 'Name' have been filled
print(credit_data['Name'].isna().sum())
```

```
0
```

```
# Convert 'Age' column to numeric, coercing errors to NaN (if not already done)
credit_data['Age'] = pd.to_numeric(credit_data['Age'], errors='coerce')

# Calculate the median age for each Customer_ID
median_age_per_id = credit_data.groupby('Customer_ID')['Age'].median()

# Function to replace invalid ages
def clean_age(row):
    if row['Age'] < 0 or pd.isna(row['Age']):  # Check for negative age or NaN
        return median_age_per_id[row['Customer_ID']]  # Replace with median age
    return row['Age']

# Apply the cleaning function to the Age column
credit_data['Age'] = credit_data.apply(clean_age, axis=1)

# Check if any missing values remain in the 'Age' column
print(credit_data['Age'].isna().sum())
```

```
0
```

```
import re

# Define a function to validate SSN
def is_valid_ssn(ssn):
    # Check if the SSN consists only of digits (and has specific length)
    return isinstance(ssn, str) and bool(re.match(r'^\d{3}-\d{2}-\d{4}$', ssn))

# Replace invalid SSNs with NaN
credit_data['SSN'] = credit_data['SSN'].apply(lambda x: np.nan if not is_valid_ssn(x) else x)

# Calculate the mode (most frequent) SSN for each Customer_ID
mode_ssn_per_id = credit_data.groupby('Customer_ID')['SSN'].agg(lambda x: x.mode()[0] if not x.mode().empty else np.nan)

# Function to fill NaN SSNs with the mode for the respective Customer_ID
def fill_ssn(row):
    if pd.isna(row['SSN']):
        return mode_ssn_per_id[row['Customer_ID']]
    return row['SSN']

# Apply the filling function to the SSN column
credit_data['SSN'] = credit_data.apply(fill_ssn, axis=1)

# Check the unique values in the SSN column to confirm changes
print(credit_data.groupby('Customer_ID')['SSN'].unique())
```

```
Customer_ID
CUS_0x1000    [913-74-1218]
CUS_0x1009    [063-67-6938]
CUS_0x100b    [238-62-0395]
CUS_0x1011    [793-05-8223]
CUS_0x1013    [930-49-9615]
                   ...
CUS_0xff3     [726-35-5322]
CUS_0xff4     [655-05-7666]
CUS_0xff6     [541-92-8371]
CUS_0xffc     [226-86-7294]
CUS_0xffd     [832-88-8320]
Name: SSN, Length: 12500, dtype: object
```

```
# Group by Customer_ID and count the number of unique SSNs
unique_ssn_count_per_id = credit_data.groupby('Customer_ID')['SSN'].nunique()

# Check if all Customer_IDs have only one unique SSN
all_unique = (unique_ssn_count_per_id == 1).all()
```

```
# Print the result
if all_unique:
    print("All Customer_IDs have only one unique SSN.")
else:
    print("Some Customer_IDs have more than one unique SSN.")

# Optionally, print Customer_IDs with more than one unique SSN
multiple_ssn_customers = unique_ssn_count_per_id[unique_ssn_count_per_id > 1]
print("Customer_IDs with multiple SSNs:\n", multiple_ssn_customers)
```

```
All Customer_IDs have only one unique SSN.
Customer_IDs with multiple SSNs:
 Series([], Name: SSN, dtype: int64)
```

```
credit_data.head(10)
```

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	...	Num
0	0x1602	CUS_0xd40	January	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	...	
1	0x1603	CUS_0xd40	February	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	NaN	3	...	
2	0x1604	CUS_0xd40	March	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	NaN	3	...	
3	0x1605	CUS_0xd40	April	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	NaN	3	...	
4	0x1606	CUS_0xd40	May	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	...	
5	0x1607	CUS_0xd40	June	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	NaN	3	...	
6	0x1608	CUS_0xd40	July	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	...	
7	0x1609	CUS_0xd40	August	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	...	
8	0x160e	CUS_0x21b1	January	Rick Rothackerj	28.0	004-07-5839	_______	34847.84	3037.986667	2	...	
9	0x160f	CUS_0x21b1	February	Rick Rothackerj	28.0	004-07-5839	Teacher	34847.84	3037.986667	2	...	

10 rows × 27 columns

```
# Replace any sequence of underscores (regardless of length) with NaN
credit_data['Occupation'] = credit_data['Occupation'].replace(r'^_+$', np.nan, regex=True)

# Calculate the mode (most frequent value) of 'Occupation' for each Customer_ID
mode_occupation_per_id = credit_data.groupby('Customer_ID')['Occupation'].agg(lambda x: x.mode()[0] if not x.mode().empty else np.nan)

# Function to fill missing Occupation values with the mode for the respective Customer_ID
def fill_occupation(row):
    if pd.isna(row['Occupation']):  # If Occupation is NaN (was previously underscores)
        return mode_occupation_per_id[row['Customer_ID']]  # Replace with mode occupation
    return row['Occupation']

# Apply the filling function to the Occupation column
credit_data['Occupation'] = credit_data.apply(fill_occupation, axis=1)

# Check for remaining missing values in the Occupation column
print(credit_data['Occupation'].isna().sum())
```

```
0
```

```
credit_data['Occupation'][credit_data['Occupation']=='_______']
```

Occupation

dtype: object

```
# Remove underscores from 'Annual_Income' and convert to numeric
credit_data['Annual_Income'] = credit_data['Annual_Income'].replace(r'[_]', '', regex=True)

# Convert 'Annual_Income' to numeric, coercing errors to NaN
credit_data['Annual_Income'] = pd.to_numeric(credit_data['Annual_Income'], errors='coerce')

# Check for negative values or missing (NaN) entries
median_income_per_id = credit_data.groupby('Customer_ID')['Annual_Income'].median()

# Function to clean invalid income values
def clean_annual_income(row):
    if pd.isna(row['Annual_Income']) or row['Annual_Income'] < 0:  # Handle NaN or negative values
        return median_income_per_id[row['Customer_ID']]  # Replace with median income
    return row['Annual_Income']

# Apply the cleaning function to the 'Annual_Income' column
credit_data['Annual_Income'] = credit_data.apply(clean_annual_income, axis=1)

# Check for remaining missing values
print(credit_data['Annual_Income'].isna().sum())
```

```
0
```

```
# Calculate the median of Monthly_Inhand_Salary
median_inhand_salary = credit_data['Monthly_Inhand_Salary'].median()

# Replace NaN values in Monthly_Inhand_Salary with the median
credit_data['Monthly_Inhand_Salary'] = credit_data['Monthly_Inhand_Salary'].fillna(median_inhand_salary)

# Check for remaining missing values in Monthly_Inhand_Salary
print(credit_data['Monthly_Inhand_Salary'].isna().sum())
```

```
0
```

```
# Check for missing values
missing_count = credit_data['Num_Bank_Accounts'].isna().sum()
print(f"Missing values in Num_Bank_Accounts: {missing_count}")

# Check the data type
data_type = credit_data['Num_Bank_Accounts'].dtype
print(f"Data type of Num_Bank_Accounts: {data_type}")

# Check for negative values
negative_values = credit_data[credit_data['Num_Bank_Accounts'] < 0]
print(f"Negative values in Num_Bank_Accounts:\n{negative_values}")

# Check for outliers using a simple threshold
# You can adjust the threshold based on domain knowledge
outlier_threshold = 10  # Assuming more than 10 accounts might be an outlier
outliers = credit_data[credit_data['Num_Bank_Accounts'] > outlier_threshold]
print(f"Potential outliers in Num_Bank_Accounts:\n{outliers}")

# Check for any non-integer or unexpected characters
non_integer_values = credit_data[~credit_data['Num_Bank_Accounts'].apply(lambda x: isinstance(x, int))]
print(f"Non-integer values in Num_Bank_Accounts:\n{non_integer_values}")
```

```
339    ...              11.0   Standard          749.95
356    ...               3.0          _         1095.73
...    ...               ...        ...             ...
99591  ...               3.0          _         1452.79
99638  ...               7.0       Good          827.56
99666  ...               1.0       Good          928.28
99722  ...               2.0       Good         1019.46
99916  ...               3.0       Good          909.01

      Credit_Utilization_Ratio      Credit_History_Age Payment_of_Min_Amount  \
267                  29.766107                     NaN                   Yes
288                  24.639658                     NaN                   Yes
310                  29.706454    8 Years and 4 Months                   Yes
339                  36.559538   11 Years and 2 Months                   Yes
356                  41.661802  19 Years and 11 Months                    No
...                        ...                     ...                   ...
99591                28.051684   32 Years and 6 Months                    NM
99638                33.201730   25 Years and 8 Months                    NM
99666                43.274889   22 Years and 3 Months                    No
99722                26.578799   16 Years and 9 Months                    No
99916                29.808796                     NaN                    NM

      Total_EMI_per_month  Amount_invested_monthly  \
267            149.897199               158.648276
288             69.685459              59.82559612
310             43.070520               80.4844201
339             49.348666              25.16140443
356              0.000000              70.82263262
...                   ...                      ...
99591           13.109663              55.72695329
99638          241.065885              180.5600146
99666           72.250125               121.284825
99722           86.809918              123.9155591
99916           45.076827              49.71299351

                       Payment_Behaviour Monthly_Balance
267     High_spent_Medium_value_payments     407.9295246
288                             !@9#%8     363.272112
```

```
negative_values = credit_data[credit_data['Num_Bank_Accounts'] < 0]
print(f"Negative values in Num_Bank_Accounts:\n{negative_values[['Customer_ID', 'Num_Bank_Accounts']]}")
```

```
Negative values in Num_Bank_Accounts:
      Customer_ID  Num_Bank_Accounts
30330   CUS_0x4f2a                 -1
30331   CUS_0x4f2a                 -1
30332   CUS_0x4f2a                 -1
30333   CUS_0x4f2a                 -1
30334   CUS_0x4f2a                 -1
30335   CUS_0x4f2a                 -1
43689   CUS_0xa878                 -1
43690   CUS_0xa878                 -1
43691   CUS_0xa878                 -1
43692   CUS_0xa878                 -1
43693   CUS_0xa878                 -1
43694   CUS_0xa878                 -1
43695   CUS_0xa878                 -1
47212   CUS_0x43bc                 -1
47213   CUS_0x43bc                 -1
47214   CUS_0x43bc                 -1
47215   CUS_0x43bc                 -1
55636   CUS_0x5993                 -1
55637   CUS_0x5993                 -1
55638   CUS_0x5993                 -1
55639   CUS_0x5993                 -1
```

```
credit_data[['Customer_ID', 'Num_Bank_Accounts']][credit_data['Customer_ID']=='CUS_0x5993']
```

	Customer_ID	Num_Bank_Accounts
55632	CUS_0x5993	0
55633	CUS_0x5993	0
55634	CUS_0x5993	0
55635	CUS_0x5993	0
55636	CUS_0x5993	-1
55637	CUS_0x5993	-1
55638	CUS_0x5993	-1
55639	CUS_0x5993	-1

```
outlier_threshold = 10  # Assuming more than 10 accounts might be an outlier
outliers = credit_data[credit_data['Num_Bank_Accounts'] > outlier_threshold]
```

```
print(f"Potential outliers in Num_Bank_Accounts:\n{outliers[['Customer_ID', 'Num_Bank_Accounts']]}")
```

```
Potential outliers in Num_Bank_Accounts:
      Customer_ID  Num_Bank_Accounts
267    CUS_0x4004               1414
288    CUS_0x4080               1231
310    CUS_0x42ac                 67
339    CUS_0x9bc1                572
356    CUS_0xaedb               1488
...           ...                ...
99591   CUS_0x544                813
99638  CUS_0x296f               1481
99666   CUS_0xb09                474
99722  CUS_0x11c7                697
99916  CUS_0x1619                182

[1324 rows x 2 columns]
```

```
credit_data[['Customer_ID', 'Num_Bank_Accounts']][credit_data['Customer_ID']=='CUS_0x296f']
```

	Customer_ID	Num_Bank_Accounts
99632	CUS_0x296f	2
99633	CUS_0x296f	2
99634	CUS_0x296f	2
99635	CUS_0x296f	2
99636	CUS_0x296f	2
99637	CUS_0x296f	2
99638	CUS_0x296f	1481
99639	CUS_0x296f	2

```
# Replace -1 with 0 in Num_Bank_Accounts
credit_data['Num_Bank_Accounts'] = credit_data['Num_Bank_Accounts'].replace(-1, 0)

# Calculate the mode (most frequent value) of Num_Bank_Accounts for each Customer_ID
mode_accounts_per_id = credit_data.groupby('Customer_ID')['Num_Bank_Accounts'].agg(lambda x: x.mode()[0] if not x.mode().empty else 0)

# Replace outliers with the mode for the respective Customer_ID
# Assuming outliers are defined as greater than a certain threshold, e.g., 10
outlier_threshold = 10
credit_data.loc[credit_data['Num_Bank_Accounts'] > outlier_threshold, 'Num_Bank_Accounts'] = credit_data['Customer_ID'].map(mode_account

# Verify the changes
print(credit_data['Num_Bank_Accounts'].value_counts())
```

```
Num_Bank_Accounts
6     13179
7     12992
8     12944
4     12344
5     12299
3     12105
9      5502
10     5338
1      4541
0      4416
2      4340
Name: count, dtype: int64
```

Focusing on cleaning only the necessary columns for feature engineering is a practical approach.

Key Columns for Feature Engineering

Based on the features we want to create, we'll likely need to clean the following columns:

- **Monthly Inhand Salary**: For calculating the Debt-to-Income Ratio.
- **Annual Income**: Also for the Debt-to-Income Ratio and to ensure it is in a usable format.
- **Num_Bank_Accounts**: To create features related to banking behavior, if necessary.
- **Debt Amount**: For calculating the Debt-to-Income Ratio.
- **Credit Limit**: For calculating Credit Utilization Rate.
- **Credit History**: For calculating the Length of Credit History.

```
# Check for missing values
missing_count = credit_data['Monthly_Inhand_Salary'].isna().sum()
print(f"Missing values in Monthly_Inhand_Salary: {missing_count}")

# Check the data type
data_type = credit_data['Monthly_Inhand_Salary'].dtype
print(f"Data type of Monthly_Inhand_Salary: {data_type}")

# Check for negative values
negative_values = credit_data[credit_data['Monthly_Inhand_Salary'] < 0]
print(f"Negative values in Monthly_Inhand_Salary:\n{negative_values}")
```

```
Missing values in Monthly_Inhand_Salary: 0
Data type of Monthly_Inhand_Salary: float64
Negative values in Monthly_Inhand_Salary:
Empty DataFrame
Columns: [ID, Customer_ID, Month, Name, Age, SSN, Occupation, Annual_Income, Monthly_Inhand_Salary, Num_Bank_Accounts, Num_Credit_Ca
Index: []

[0 rows x 27 columns]
```

```
# Calculate the median Monthly_Inhand_Salary for each Customer_ID
median_salary_per_id = credit_data.groupby('Customer_ID')['Monthly_Inhand_Salary'].median()

# Fill NaN values in Monthly_Inhand_Salary with the median for the respective Customer_ID
credit_data['Monthly_Inhand_Salary'] = credit_data.apply(
    lambda row: median_salary_per_id[row['Customer_ID']] if pd.isna(row['Monthly_Inhand_Salary']) else row['Monthly_Inhand_Salary'],
    axis=1
)

# Check for remaining NaN values in Monthly_Inhand_Salary
remaining_na = credit_data['Monthly_Inhand_Salary'].isna().sum()
print(f"Remaining NaN values in Monthly_Inhand_Salary: {remaining_na}")
```

```
Remaining NaN values in Monthly_Inhand_Salary: 0
```

```
# Check for missing values
missing_count = credit_data['Outstanding_Debt'].isna().sum()
print(f"Missing values in Outstanding_Debt: {missing_count}")

# Check the data type
data_type = credit_data['Outstanding_Debt'].dtype
print(f"Data type of Outstanding_Debt: {data_type}")

# # Check for negative values
# negative_values = credit_data[credit_data['Outstanding_Debt'] < 0]
# print(f"Negative values in Outstanding_Debt:\n{negative_values}")
```

```
Missing values in Outstanding_Debt: 0
Data type of Outstanding_Debt: object
```

```
# Remove underscores and convert to numeric
credit_data['Outstanding_Debt'] = credit_data['Outstanding_Debt'].replace(r'[_]', '', regex=True)

# Convert to numeric, forcing errors to NaN
credit_data['Outstanding_Debt'] = pd.to_numeric(credit_data['Outstanding_Debt'], errors='coerce')

# Calculate the median Outstanding_Debt
median_outstanding_debt = credit_data['Outstanding_Debt'].median()

# Fill NaN values with the median Outstanding_Debt
credit_data['Outstanding_Debt'] = credit_data['Outstanding_Debt'].fillna(median_outstanding_debt)

# Check for remaining NaN values in Outstanding_Debt
remaining_na_outstanding_debt = credit_data['Outstanding_Debt'].isna().sum()
print(f"Remaining NaN values in Outstanding_Debt: {remaining_na_outstanding_debt}")
```

```
Remaining NaN values in Outstanding_Debt: 0
```

```
# Check for missing values
missing_count = credit_data['Outstanding_Debt'].isna().sum()
print(f"Missing values in Outstanding_Debt: {missing_count}")

# Check the data type
data_type = credit_data['Outstanding_Debt'].dtype
print(f"Data type of Outstanding_Debt: {data_type}")

# Check for negative values
```

```
negative_values = credit_data[credit_data['Outstanding_Debt'] < 0]
print(f"Negative values in Outstanding_Debt:\n{negative_values}")
```

```
Missing values in Outstanding_Debt: 0
Data type of Outstanding_Debt: float64
Negative values in Outstanding_Debt:
Empty DataFrame
Columns: [ID, Customer_ID, Month, Name, Age, SSN, Occupation, Annual_Income, Monthly_Inhand_Salary, Num_Bank_Accounts, Num_Credit_Ca
Index: []

[0 rows x 27 columns]
```

```
# Function to convert Credit_History_Age to total months
def convert_credit_history_age(age):
    # Check if age is a string
    if isinstance(age, str):
        if age == 'NA':
            return np.nan  # Replace 'NA' with NaN
        match = re.match(r'(\d+)\s*Years?\s+and\s+(\d+)\s*Months?', age)
        if match:
            years = int(match.group(1))
            months = int(match.group(2))
            return years * 12 + months  # Convert to total months
    return np.nan  # Return NaN for any non-string or unmatched format

# Apply the function to the Credit_History_Age column
credit_data['Credit_History_Age'] = credit_data['Credit_History_Age'].apply(convert_credit_history_age)

# Calculate the median Credit_History_Age for each Customer_ID
median_age_per_id = credit_data.groupby('Customer_ID')['Credit_History_Age'].median()

# Fill NaN values in Credit_History_Age with the median for the respective Customer_ID
credit_data['Credit_History_Age'] = credit_data.apply(
    lambda row: median_age_per_id[row['Customer_ID']] if pd.isna(row['Credit_History_Age']) else row['Credit_History_Age'],
    axis=1
)

# Check for remaining NaN values in Credit_History_Age
remaining_na_credit_history_age = credit_data['Credit_History_Age'].isna().sum()
print(f"Remaining NaN values in Credit_History_Age: {remaining_na_credit_history_age}")
```

```
Remaining NaN values in Credit_History_Age: 0
```

```
credit_data.head(10)
```

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	...	Num
0	0x1602	CUS_0xd40	January	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	...	
1	0x1603	CUS_0xd40	February	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	3093.745000	3	...	
2	0x1604	CUS_0xd40	March	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	3093.745000	3	...	
3	0x1605	CUS_0xd40	April	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	3093.745000	3	...	
4	0x1606	CUS_0xd40	May	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	...	
5	0x1607	CUS_0xd40	June	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	3093.745000	3	...	
6	0x1608	CUS_0xd40	July	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	...	
7	0x1609	CUS_0xd40	August	Aaron Maashoh	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	...	
8	0x160e	CUS_0x21b1	January	Rick Rothackerj	28.0	004-07-5839	Teacher	34847.84	3037.986667	2	...	
9	0x160f	CUS_0x21b1	February	Rick Rothackerj	28.0	004-07-5839	Teacher	34847.84	3037.986667	2	...	

10 rows × 27 columns

```
# Check for missing values
missing_count = credit_data['Total_EMI_per_month'].isna().sum()
print(f"Missing values in Total_EMI_per_month: {missing_count}")

# Check the data type
data_type = credit_data['Total_EMI_per_month'].dtype
print(f"Data type of Total_EMI_per_month: {data_type}")

# Check for negative values
negative_values = credit_data[credit_data['Total_EMI_per_month'] < 0]
print(f"Negative values in Total_EMI_per_month:\n{negative_values}")
```

```
Missing values in Total_EMI_per_month: 0
Data type of Total_EMI_per_month: float64
Negative values in Total_EMI_per_month:
Empty DataFrame
Columns: [ID, Customer_ID, Month, Name, Age, SSN, Occupation, Annual_Income, Monthly_Inhand_Salary, Num_Bank_Accounts, Num_Credit_Ca
Index: []

[0 rows x 27 columns]
```

```
# Ensure we have cleaned data before feature engineering

# 1. Calculate Debt-to-Income Ratio using Total_EMI_per_month
# Assuming Monthly_Inhand_Salary is the monthly salary
credit_data['Debt_to_Income_Ratio'] = credit_data['Total_EMI_per_month'] / credit_data['Monthly_Inhand_Salary']

# To ensure there are no infinite or NaN values due to divisions, we can fill those cases
credit_data['Debt_to_Income_Ratio'].replace([np.inf, -np.inf], np.nan, inplace=True)

# Check the new features
print(credit_data[['Debt_to_Income_Ratio', 'Credit_Utilization_Ratio', 'Credit_History_Age']].head())
```

```
   Debt_to_Income_Ratio  Credit_Utilization_Ratio  Credit_History_Age
0              0.027167                 26.822620               265.0
1              0.016024                 31.944960               268.5
2              0.016024                 28.609352               267.0
3              0.016024                 31.377862               268.0
4              0.027167                 24.797347               269.0
<ipython-input-117-99edcd42fe05>:8: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained as
```

```
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col

  credit_data['Debt_to_Income_Ratio'].replace([np.inf, -np.inf], np.nan, inplace=True)
```

```
credit_data['Credit_History_Age_Months'] = credit_data['Credit_History_Age']
```

Hypothetical Credit Score Calculation Method:

We will assign weights to each feature and then combine them to create a score. Example weights:

- Debt-to-Income Ratio (DTI): 40%
- Credit Utilization Rate: 35%
- Length of Credit History: 25%

We'll normalize or invert the ratios where lower values are better (like DTI).

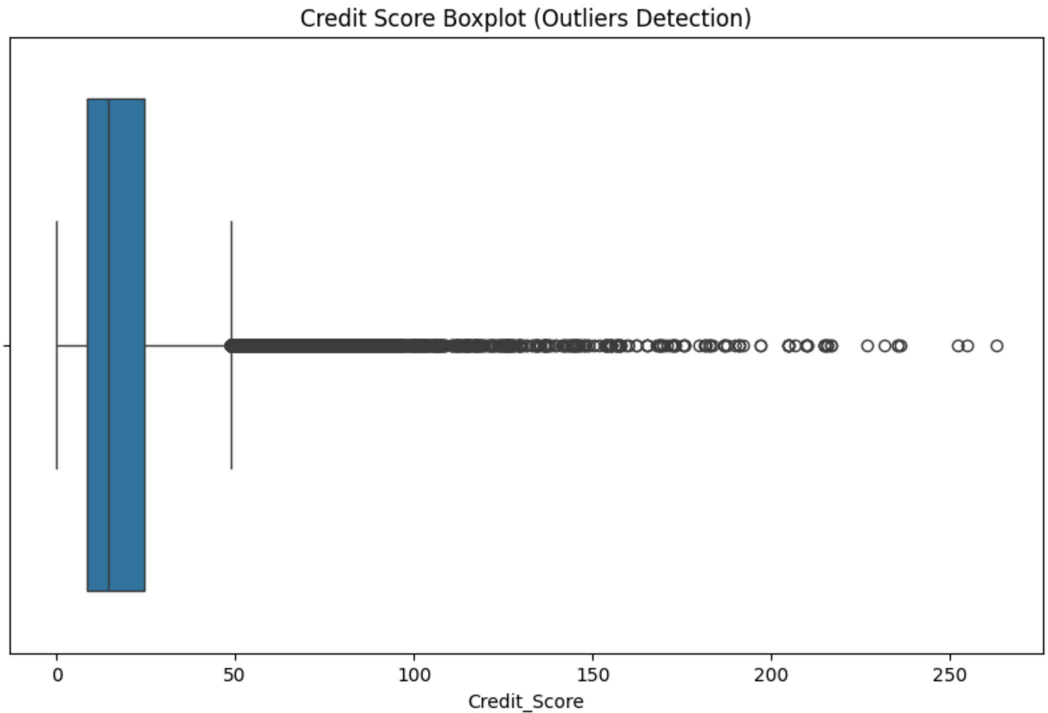
```
# Normalize DTI and Credit Utilization (lower values are better)
credit_data['Normalized_DTI'] = 1 / credit_data['Debt_to_Income_Ratio']
credit_data['Normalized_Credit_Utilization'] = 1 / credit_data['Credit_Utilization_Ratio']

# Normalize Credit History Length (longer is better, so we divide by the maximum length)
credit_data['Normalized_Credit_History'] = credit_data['Credit_History_Age'] / credit_data['Credit_History_Age'].max()

# Define weights for each feature
dti_weight = 0.40
credit_utilization_weight = 0.35
credit_history_weight = 0.25

# Calculate the hypothetical credit score
credit_data['Credit_Score'] = (
    credit_data['Normalized_DTI'] * dti_weight +
    credit_data['Normalized_Credit_Utilization'] * credit_utilization_weight +
    credit_data['Normalized_Credit_History'] * credit_history_weight
)

# Display the calculated credit scores
print(credit_data[['Customer_ID', 'Credit_Score']].head())
```

```
  Customer_ID  Credit_Score
0   CUS_0xd40     14.900949
1   CUS_0xd40     25.139271
2   CUS_0xd40     25.139620
3   CUS_0xd40     25.139160
4   CUS_0xd40     14.904490
```

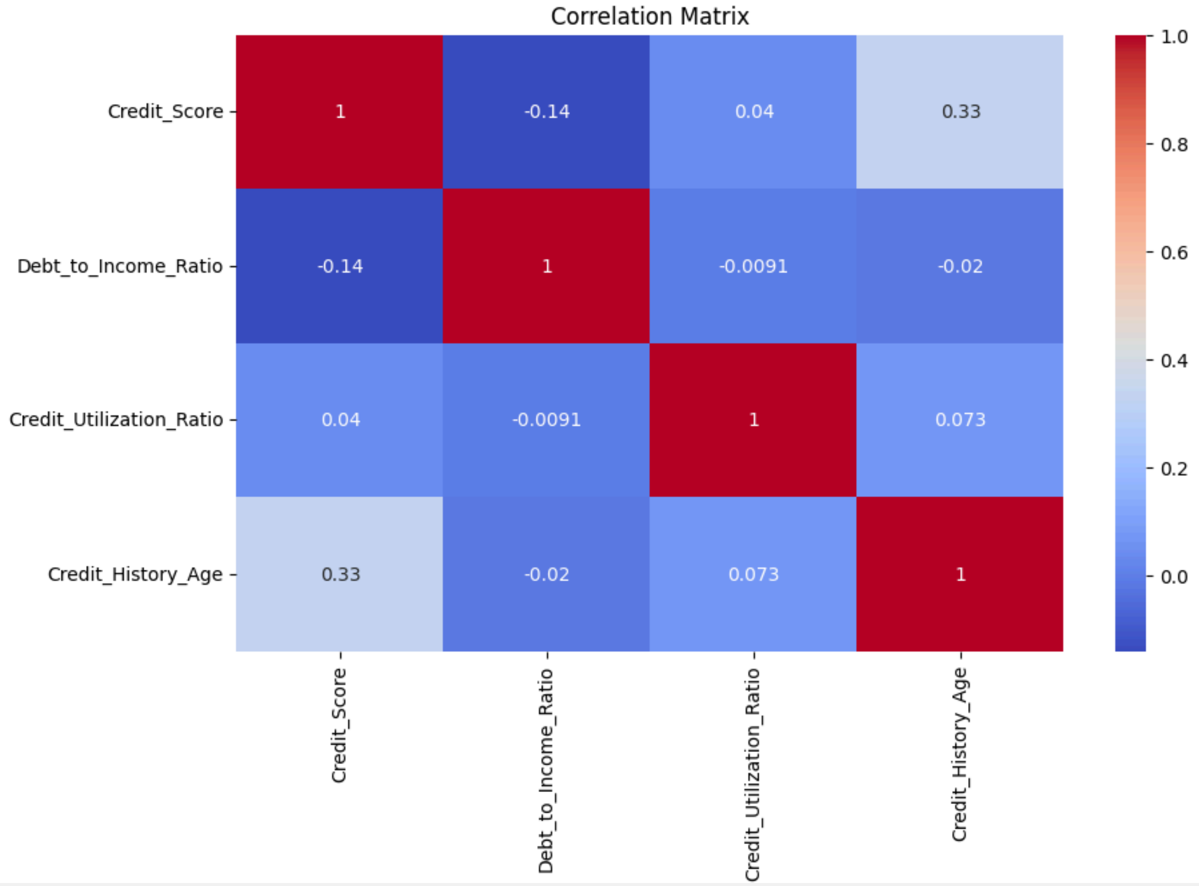
```
# 1. Plot the distribution of credit scores
plt.figure(figsize=(10, 6))
sns.histplot(credit_data['Credit_Score'], bins=30, kde=True)
plt.title('Distribution of Credit Scores')
plt.xlabel('Credit Score')
plt.ylabel('Frequency')
plt.show()
```


```
plt.figure(figsize=(10, 6))
sns.boxplot(x=credit_data['Credit_Score'])
plt.title('Credit Score Boxplot (Outliers Detection)')
plt.show()
```

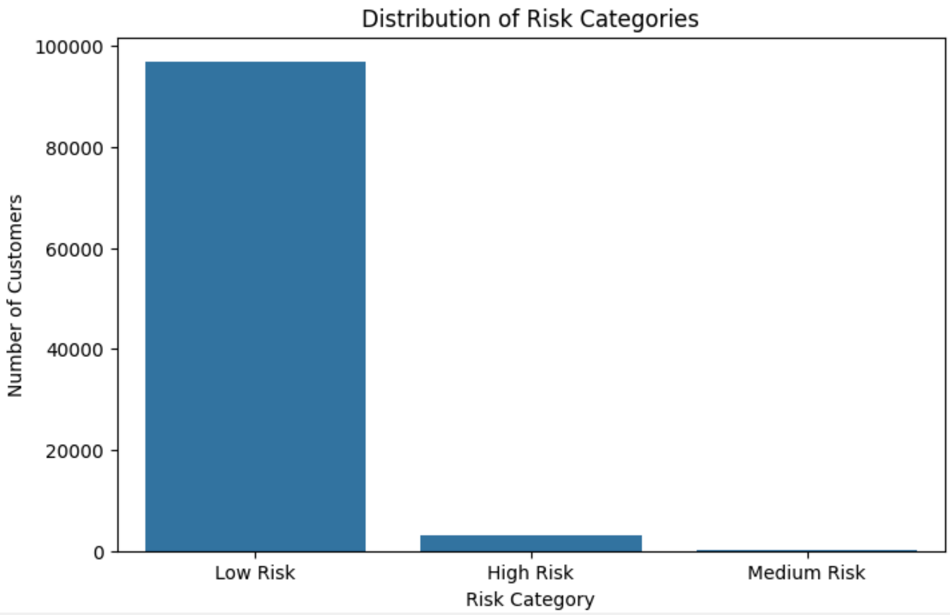
```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```




```
# 3. Correlation between Credit Score and features
plt.figure(figsize=(10, 6))
sns.heatmap(credit_data[['Credit_Score', 'Debt_to_Income_Ratio', 'Credit_Utilization_Ratio', 'Credit_History_Age']].corr(), annot=True,
plt.title('Correlation Matrix')
plt.show()
```




```
# 4. Segment customers based on Credit Score
def categorize_risk(score):
    if score >= 0.75:
        return 'Low Risk'
    elif score >= 0.50:
        return 'Medium Risk'
    else:
        return 'High Risk'

credit_data['Risk_Category'] = credit_data['Credit_Score'].apply(categorize_risk)


# Check distribution of customers by risk category
plt.figure(figsize=(8, 5))
sns.countplot(x=credit_data['Risk_Category'])
plt.title('Distribution of Risk Categories')
plt.xlabel('Risk Category')
plt.ylabel('Number of Customers')
plt.show()
```



```
# Print the number of customers in each risk category
print(credit_data['Risk_Category'].value_counts())
```

```
Risk_Category
Low Risk       96907
High Risk       3021
Medium Risk       72
Name: count, dtype: int64
```

Separate Monthly EMIs from One-Time Payments:

```
# Function to identify large payments and cap them by Customer_ID
def process_emis_by_customer(group):
    # Calculate the 95th percentile threshold for Total_EMI_per_month for this customer
    large_payment_threshold = group['Total_EMI_per_month'].quantile(0.95)

    # Create a new feature for identifying one-time payments
    group['Is_One_Time_Payment'] = group['Total_EMI_per_month'].apply(lambda x: 1 if x > large_payment_threshold else 0)

    # Cap large payments
    group['Capped_Total_EMI'] = np.where(group['Total_EMI_per_month'] > large_payment_threshold, large_payment_threshold, group['Total_E

    # Replace zeros with the median of Total_EMI_per_month for this customer
    median_emi = group['Total_EMI_per_month'].median()
    group['Capped_Total_EMI'] = group['Capped_Total_EMI'].replace(0, median_emi)

    return group

# Apply the processing function to each customer group
credit_data = credit_data.groupby('Customer_ID').apply(process_emis_by_customer)

# Check the updated columns
print(credit_data[['Customer_ID', 'Total_EMI_per_month', 'Capped_Total_EMI', 'Is_One_Time_Payment']].head())
```

```
                  Customer_ID  Total_EMI_per_month  Capped_Total_EMI  \
Customer_ID
CUS_0x1000  56752  CUS_0x1000             42.94109          42.94109
            56753  CUS_0x1000             42.94109          42.94109
            56754  CUS_0x1000             42.94109          42.94109
            56755  CUS_0x1000             42.94109          42.94109
            56756  CUS_0x1000             42.94109          42.94109

                   Is_One_Time_Payment
Customer_ID
CUS_0x1000  56752                    0
            56753                    0
            56754                    0
            56755                    0
            56756                    0
<ipython-input-126-5a524447be3e>:19: DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns. This behavior is d
  credit_data = credit_data.groupby('Customer_ID').apply(process_emis_by_customer)
```

```
credit_data[['Customer_ID', 'Total_EMI_per_month', 'Capped_Total_EMI', 'Is_One_Time_Payment']].sample(10)
```

Customer_ID		Customer_ID	Total_EMI_per_month	Capped_Total_EMI	Is_One_Time_Payment
CUS_0xa833	**28273**	CUS_0xa833	61008.000000	39669.542823	1
CUS_0xcf0	**58996**	CUS_0xcf0	390.588799	390.588799	0
CUS_0x89e3	**65757**	CUS_0x89e3	208.810585	208.810585	0
CUS_0x776d	**93878**	CUS_0x776d	42.350618	42.350618	0
CUS_0x3b11	**64079**	CUS_0x3b11	125.870950	125.870950	0
CUS_0x5386	**77306**	CUS_0x5386	45.243874	45.243874	0
CUS_0x5148	**84824**	CUS_0x5148	187.518642	187.518642	0
CUS_0x7d05	**58703**	CUS_0x7d05	109.496774	109.496774	0
CUS_0x6673	**15182**	CUS_0x6673	339.495329	339.495329	0
CUS_0xb967	**63476**	CUS_0xb967	0.000000	0.000000	0

- Debt-to-Income Ratio: 30%

- Credit Utilization Ratio: 25%
- Capped Total EMI: 20%
- Length of Credit History: 15%
- Monthly Inhand Salary: 5%
- Number of Bank Accounts: 5%

```
from sklearn.preprocessing import MinMaxScaler

# Initialize the scaler
scaler = MinMaxScaler()

# Define the features to normalize
features_to_normalize = ['Debt_to_Income_Ratio', 'Credit_Utilization_Ratio',
                         'Total_EMI_per_month', 'Credit_History_Age',
                         'Monthly_Inhand_Salary', 'Num_Bank_Accounts']

# Normalize the selected features
credit_data[features_to_normalize] = scaler.fit_transform(credit_data[features_to_normalize])
```

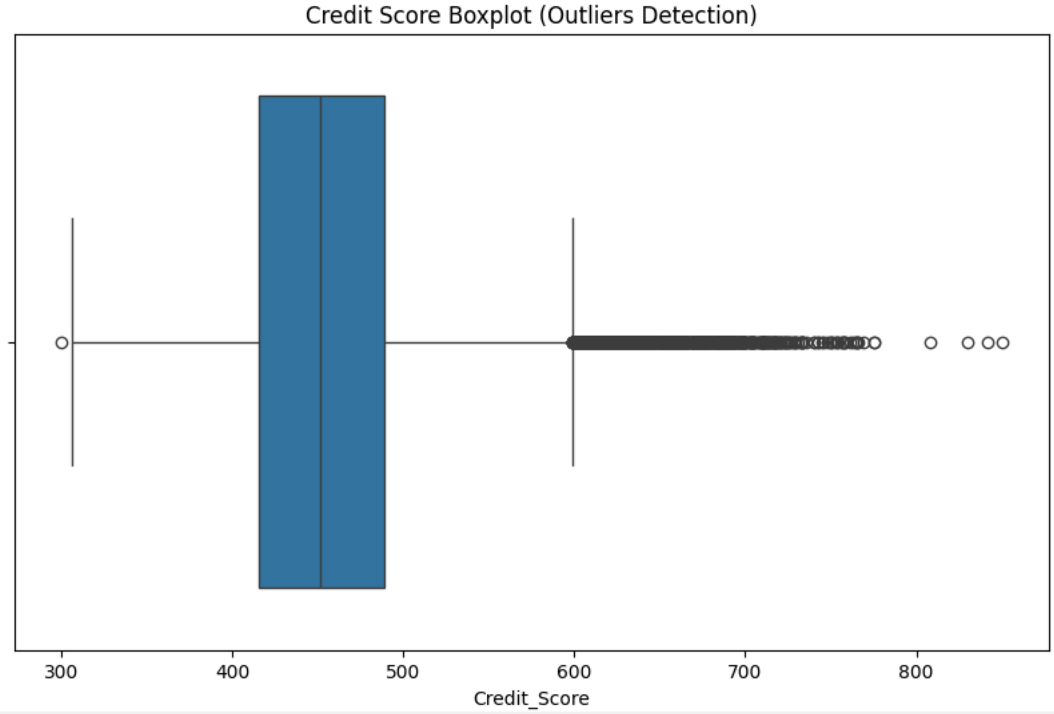
```
# Define weights for each feature
weights = {
    'Debt_to_Income_Ratio': 0.3,
    'Credit_Utilization_Ratio': 0.25,
    'Total_EMI_per_month': 0.2,
    'Credit_History_Age': 0.15,
    'Monthly_Inhand_Salary': 0.05,
    'Num_Bank_Accounts': 0.05
}

# Calculate Credit Score using weighted sum of normalized features
credit_data['Credit_Score'] = (
    credit_data['Debt_to_Income_Ratio'] * weights['Debt_to_Income_Ratio'] +
    credit_data['Credit_Utilization_Ratio'] * weights['Credit_Utilization_Ratio'] +
    credit_data['Total_EMI_per_month'] * weights['Total_EMI_per_month'] +
    credit_data['Credit_History_Age'] * weights['Credit_History_Age'] +
    credit_data['Monthly_Inhand_Salary'] * weights['Monthly_Inhand_Salary'] +
    credit_data['Num_Bank_Accounts'] * weights['Num_Bank_Accounts']
)
```

```
# Scale the Credit Score to a range of 300 to 850
min_score = credit_data['Credit_Score'].min()
max_score = credit_data['Credit_Score'].max()

credit_data['Credit_Score'] = (credit_data['Credit_Score'] - min_score) / (max_score - min_score) * 550 + 300
```

```
plt.figure(figsize=(12, 6))
sns.histplot(credit_data['Credit_Score'], bins=30, kde=True)
plt.title('Distribution of Credit Scores')
plt.xlabel('Credit Score')
plt.ylabel('Frequency')
plt.grid()
plt.show()
```

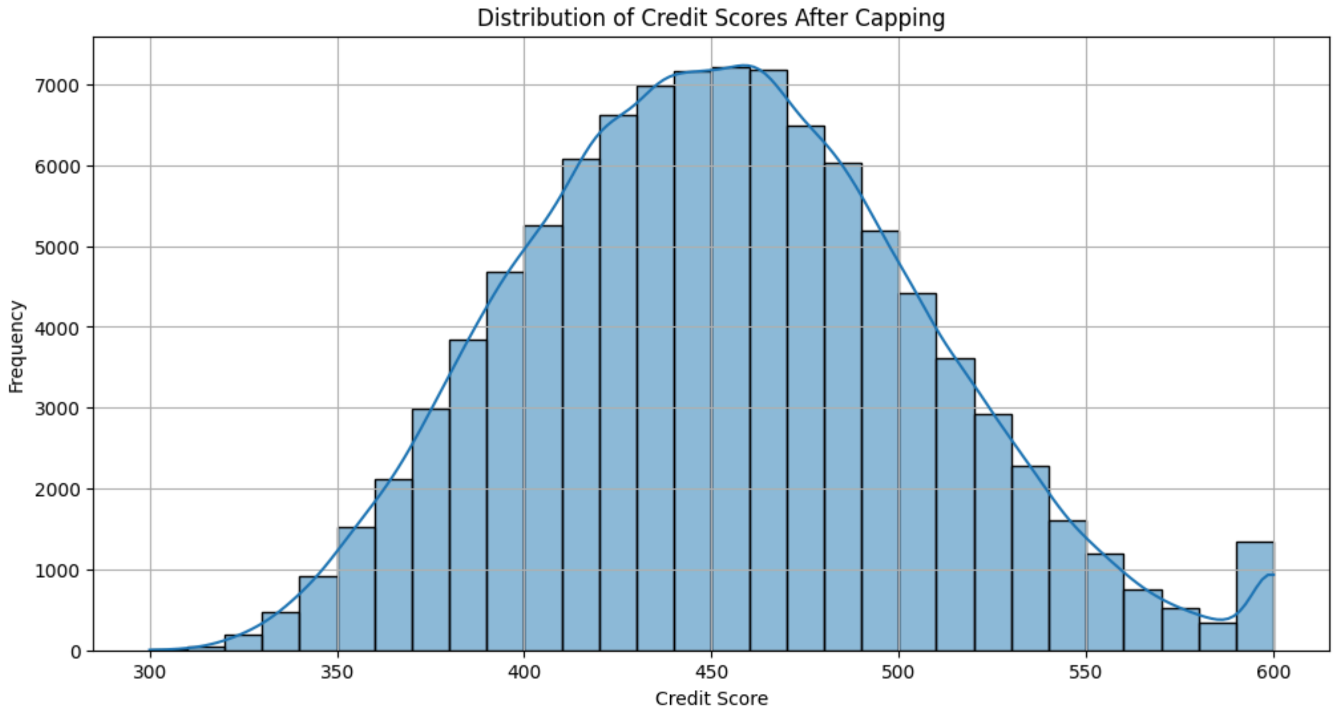

```
plt.figure(figsize=(10, 6))
sns.boxplot(x=credit_data['Credit_Score'])
plt.title('Credit Score Boxplot (Outliers Detection)')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```



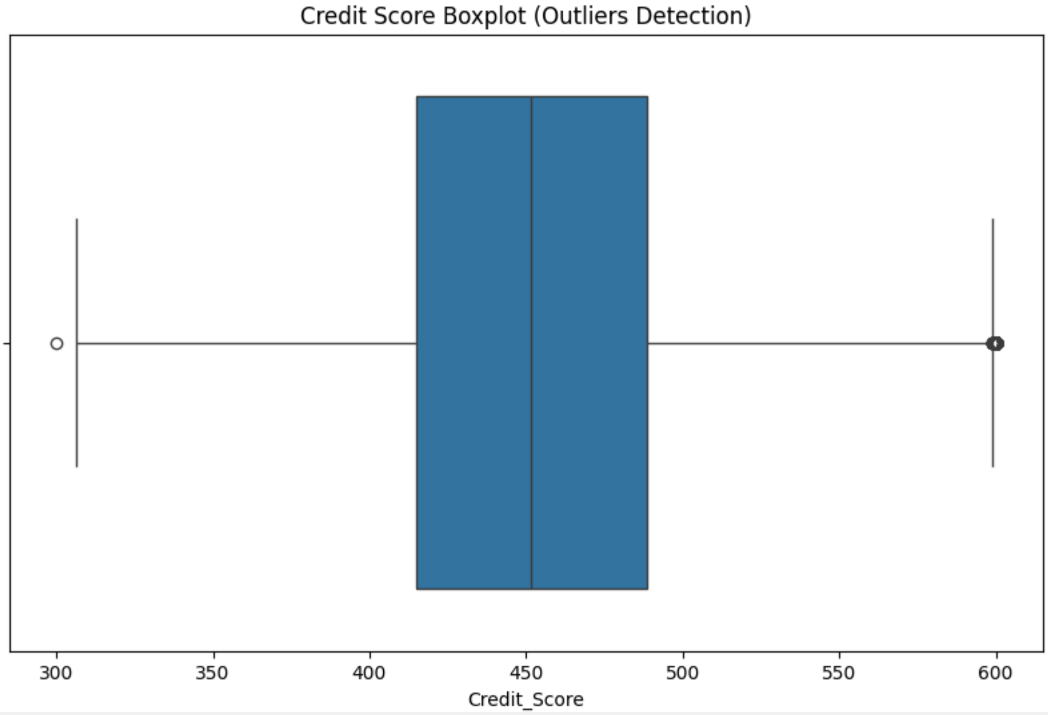

```
# Define the score limits
lower_limit = 300
upper_limit = 600

# Cap the scores to the defined limits
credit_data['Credit_Score'] = credit_data['Credit_Score'].clip(lower=lower_limit, upper=upper_limit)

# Optionally, remove rows where Credit_Score is below 300 or above 600
# Uncomment the line below to drop outliers entirely
# credit_data = credit_data[(credit_data['Credit_Score'] >= lower_limit) & (credit_data['Credit_Score'] <= upper_limit)]

# Verify the distribution of credit scores after capping
plt.figure(figsize=(12, 6))
sns.histplot(credit_data['Credit_Score'], bins=30, kde=True)
```

```
plt.title('Distribution of Credit Scores After Capping')
plt.xlabel('Credit Score')
plt.ylabel('Frequency')
plt.grid()
plt.show()
```




```
plt.figure(figsize=(10, 6))
sns.boxplot(x=credit_data['Credit_Score'])
plt.title('Credit Score Boxplot (Outliers Detection)')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```




```
# Check skewness
skewness = credit_data['Credit_Score'].skew()
print(f'Skewness of Credit Score distribution: {skewness}')
```

```
Skewness of Credit Score distribution: 0.24703527281039994
```

Defining the Risk Scale

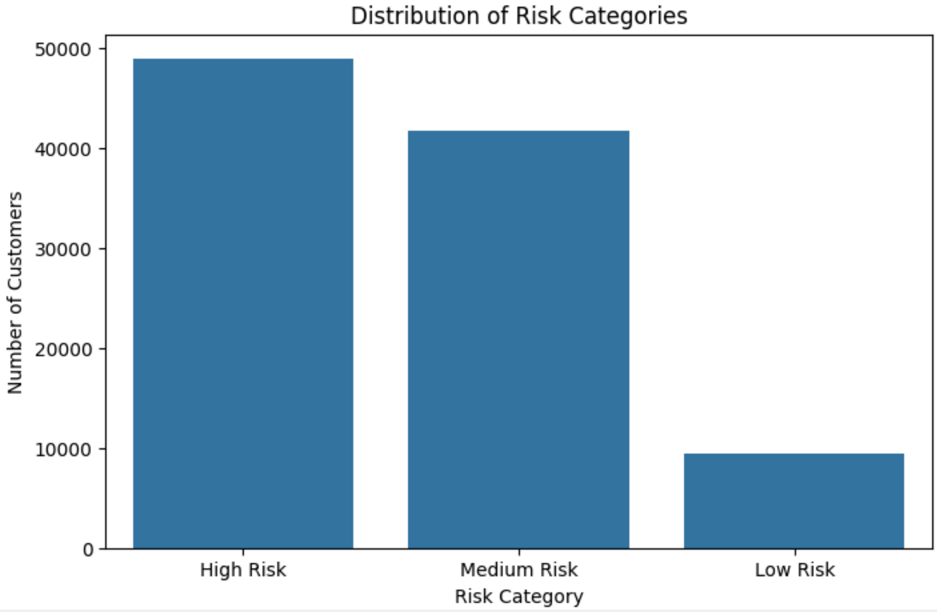
Let's assume your credit scores are now normalized between 300 and 600 after addressing outliers. Here's a proposed risk scale based on the adjusted scores:

- Low Risk: Credit Score ≥ 525 (approximately 87.5% of the maximum score)
- Medium Risk: Credit Score 450 to 524 (75% to 87.5% of the maximum score)
- High Risk: Credit Score < 450 (below 75% of the maximum score)

```
# Define the function to categorize risk
def categorize_risk(score):
    if score >= 525:  # Low Risk threshold
        return 'Low Risk'
    elif score >= 450:  # Medium Risk threshold
        return 'Medium Risk'
    else:  # High Risk threshold
        return 'High Risk'

# Apply the function to the Credit_Score column
credit_data['Risk_Category'] = credit_data['Credit_Score'].apply(categorize_risk)

# Display the first few rows to check the new Risk_Category
print(credit_data[['Credit_Score', 'Risk_Category']].head())
```

```
                         Credit_Score Risk_Category
Customer_ID
CUS_0x1000  56752    375.806002     High Risk
            56753    397.775055     High Risk
            56754    461.149051   Medium Risk
            56755    421.984652     High Risk
            56756    418.589927     High Risk
```

```
# Check distribution of customers by risk category
plt.figure(figsize=(8, 5))
sns.countplot(x=credit_data['Risk_Category'])
plt.title('Distribution of Risk Categories')
plt.xlabel('Risk Category')
plt.ylabel('Number of Customers')
plt.show()
```



Fair Isaac Corporation (FICO)

Payment History (35%)

```
credit_data = credit_data.reset_index(drop=True)
```

```
credit_data.head()
```

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	...	Month
0	0x1628a	CUS_0x1000	January	Alistair Barrf	17.0	913-74-1218	Lawyer	30625.94	0.161232	0.6	...	2
1	0x1628b	CUS_0x1000	February	Alistair Barrf	17.0	913-74-1218	Lawyer	30625.94	0.187243	0.6	...	3
2	0x1628c	CUS_0x1000	March	Alistair Barrf	17.0	913-74-1218	Lawyer	30625.94	0.161232	0.6	...	3
3	0x1628d	CUS_0x1000	April	Alistair Barrf	17.0	913-74-1218	Lawyer	30625.94	0.161232	0.6	...	4
4	0x1628e	CUS_0x1000	May	Alistair Barrf	17.0	913-74-1218	Lawyer	30625.94	0.161232	0.6	...	3

5 rows × 36 columns

```
credit_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 36 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  float64
 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  float64
 20  Credit_Utilization_Ratio      100000 non-null  float64
 21  Credit_History_Age            100000 non-null  float64
 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
 27  Debt_to_Income_Ratio          100000 non-null  float64
 28  Credit_History_Age_Months     100000 non-null  float64
 29  Normalized_DTI                100000 non-null  float64
 30  Normalized_Credit_Utilization 100000 non-null  float64
 31  Normalized_Credit_History     100000 non-null  float64
 32  Credit_Score                  100000 non-null  float64
 33  Risk_Category                 100000 non-null  object
 34  Is_One_Time_Payment           100000 non-null  int64
 35  Capped_Total_EMI              100000 non-null  float64
dtypes: float64(16), int64(4), object(16)
memory usage: 27.5+ MB
```

```
credit_data['Num_of_Delayed_Payment'].isna().sum()
```

```
7002
```

```
# Step 1: Replace negative values (-1) with 0 (indicating no delay)
credit_data['Num_of_Delayed_Payment'] = credit_data['Num_of_Delayed_Payment'].replace(-1, 0)

# Step 2: Remove underscores from string values and convert to numeric
credit_data['Num_of_Delayed_Payment'] = credit_data['Num_of_Delayed_Payment'].astype(str).str.replace('_', '').astype(float)

# Step 3: Calculate the median of 'Num_of_Delayed_Payment' for each Customer_ID
median_delayed_payment_per_id = credit_data.groupby('Customer_ID')['Num_of_Delayed_Payment'].transform('median')

# Step 4: Replace missing or NaN values with the median of the respective Customer_ID
credit_data['Num_of_Delayed_Payment'] = credit_data['Num_of_Delayed_Payment'].fillna(median_delayed_payment_per_id)
```

```
credit_data['Num_of_Delayed_Payment'].isna().sum()
```

```
0
```

```
credit_data['Delay_from_due_date'].isna().sum()
```

```
0
```

```
# Step 1: Replace negative values (-1) with 0 (indicating no delay)
credit_data['Delay_from_due_date'] = credit_data['Delay_from_due_date'].replace(-1, 0)
```

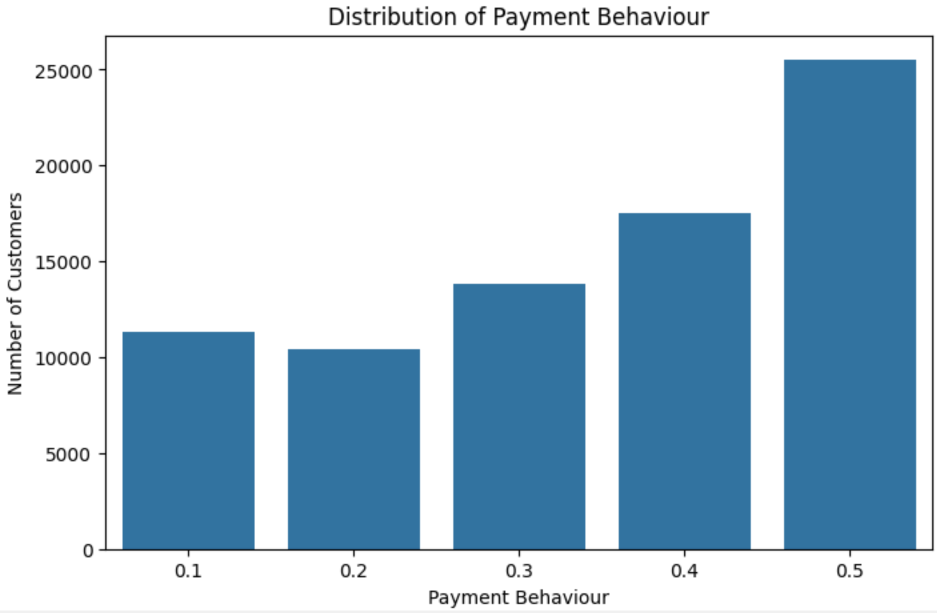
```
credit_data['Num_of_Delayed_Payment_norm'] = credit_data['Num_of_Delayed_Payment'] / credit_data['Num_of_Delayed_Payment'].max()
credit_data['Delay_from_due_date_norm'] = credit_data['Delay_from_due_date'] / credit_data['Delay_from_due_date'].max()
```

```
credit_data['Payment_Behaviour'].isna().sum()
```

```
0
```

```
payment_behaviour_mapping = {
    'High_spent_Small_value_payments': 0.1,
    'Low_spent_Medium_value_payments': 0.3,
    'Low_spent_Large_value_payments': 0.2,
    'High_spent_Medium_value_payments': 0.4,
    'Low_spent_Small_value_payments': 0.5
}
credit_data['Payment_Behaviour_numeric'] = credit_data['Payment_Behaviour'].map(payment_behaviour_mapping)
```

```
# Check distribution of customers by risk category
plt.figure(figsize=(8, 5))
sns.countplot(x=credit_data['Payment_Behaviour_numeric'])
plt.title('Distribution of Payment Behaviour')
plt.xlabel('Payment Behaviour')
plt.ylabel('Number of Customers')
plt.show()
```




```
# Calculate mode of Payment_of_Min_Amount for each Customer_ID
mode_payment_per_id = credit_data.groupby('Customer_ID')['Payment_of_Min_Amount'].agg(lambda x: x.mode().iloc[0] if not x.mode().empty
```

```
# Define a function to replace "NM" with the mode for each Customer_ID
def replace_nm_with_mode(row):
    if row['Payment_of_Min_Amount'] == 'NM':
        # Return the mode value for the corresponding Customer_ID
        return mode_payment_per_id.get(row['Customer_ID'])  # Use .get() to safely access
    return row['Payment_of_Min_Amount']

# Apply the function to the Payment_of_Min_Amount column
credit_data['Payment_of_Min_Amount'] = credit_data.apply(replace_nm_with_mode, axis=1)
```

```
credit_data['Payment_of_Min_Amount_numeric'] = credit_data['Payment_of_Min_Amount'].apply(lambda x: 1 if x == 'Yes' else 0)
```

```
credit_data['Payment_History_Score'] = (
    1
    - 0.35 * credit_data['Num_of_Delayed_Payment_norm']
    - 0.35 * credit_data['Delay_from_due_date_norm']
    - 0.2 * credit_data['Payment_Behaviour_numeric']
    - 0.1 * credit_data['Payment_of_Min_Amount_numeric']
)
```

```
credit_data[['Customer_ID', 'Payment_History_Score']].head()
```

	Customer_ID	Payment_History_Score
0	CUS_0x1000	0.534129
1	CUS_0x1000	0.554289
2	CUS_0x1000	0.493891
3	CUS_0x1000	NaN
4	CUS_0x1000	0.508010

```
credit_data['Payment_History_Score'].isna().sum()
```

21321

```
credit_data[['Customer_ID', 'Payment_History_Score']].sample(10)
```

	Customer_ID	Payment_History_Score
49690	CUS_0x6cf5	NaN
31443	CUS_0x4c43	NaN
14015	CUS_0x2be8	NaN
10011	CUS_0x247b	0.932905
92067	CUS_0xbb75	0.863194
62317	CUS_0x8506	0.648398
35865	CUS_0x5407	NaN
16643	CUS_0x30b9	NaN
38480	CUS_0x58e3	0.662408
37639	CUS_0x5746	NaN

```
# Step 1: Calculate the median Payment_History_Score for each Customer_ID
median_payment_history_score = credit_data.groupby('Customer_ID')['Payment_History_Score'].median()

# Step 2: Create a mapping for Customer_ID to their median score
median_mapping = median_payment_history_score.to_dict()

# Step 3: Replace NaN values in Payment_History_Score using the mapping
credit_data['Payment_History_Score'] = credit_data['Payment_History_Score'].fillna(credit_data['Customer_ID'].map(median_mapping))

# Now ensure that every entry for the same Customer_ID has the same Payment_History_Score
credit_data['Payment_History_Score'] = credit_data.groupby('Customer_ID')['Payment_History_Score'].transform(lambda x: x.fillna(x.mediar


# Replace Payment_History_Score with the median for each Customer_ID
credit_data['Payment_History_Score'] = credit_data['Customer_ID'].map(median_mapping)

# Check if the values are updated correctly
```

```
print(credit_data[['Customer_ID', 'Payment_History_Score']].head(10))
```

```
  Customer_ID  Payment_History_Score
0  CUS_0x1000               0.520169
1  CUS_0x1000               0.520169
2  CUS_0x1000               0.520169
3  CUS_0x1000               0.520169
4  CUS_0x1000               0.520169
5  CUS_0x1000               0.520169
6  CUS_0x1000               0.520169
7  CUS_0x1000               0.520169
8  CUS_0x1009               0.776776
9  CUS_0x1009               0.776776
```

Amounts Owed (30%)

```
# Assuming credit_data is your DataFrame
def calculate_amounts_owed_score(row):
    # Normalize Outstanding Debt
    max_outstanding_debt = credit_data['Outstanding_Debt'].max()
    norm_outstanding_debt = row['Outstanding_Debt'] / max_outstanding_debt

    # Normalize Credit Utilization Ratio
    norm_credit_utilization = 1 - row['Credit_Utilization_Ratio']

    # Normalize Total EMI per Month
    max_total_emi = credit_data['Total_EMI_per_month'].max()
    norm_total_emi = row['Total_EMI_per_month'] / max_total_emi

    # Combine into a final score (weights can be adjusted as needed)
    score = (0.4 * (1 - norm_outstanding_debt)) + \
            (0.4 * norm_credit_utilization) + \
            (0.2 * (1 - norm_total_emi))  # lower EMI is better

    return score

# Apply the function to calculate the Amounts Owed Score
credit_data['Amounts_Owed_Score'] = credit_data.apply(calculate_amounts_owed_score, axis=1)
```

```
print(credit_data[['Customer_ID', 'Amounts_Owed_Score']].head(10))
```

```
  Customer_ID  Amounts_Owed_Score
0  CUS_0x1000            0.786653
1  CUS_0x1000            0.748951
2  CUS_0x1000            0.631002
3  CUS_0x1000            0.703574
4  CUS_0x1000            0.710408
5  CUS_0x1000            0.607051
6  CUS_0x1000            0.632821
7  CUS_0x1000            0.740452
8  CUS_0x1009            0.934456
9  CUS_0x1009            0.870833
```

Length of Credit History (15%)

```
print(credit_data[['Customer_ID', 'Credit_History_Age']].head(10))
```

```
  Customer_ID  Credit_History_Age
0  CUS_0x1000            0.300248
1  CUS_0x1000            0.302730
2  CUS_0x1000            0.305211
3  CUS_0x1000            0.307692
4  CUS_0x1000            0.310174
5  CUS_0x1000            0.312655
6  CUS_0x1000            0.315136
7  CUS_0x1000            0.317618
8  CUS_0x1009            0.903226
9  CUS_0x1009            0.905707
```

```
# Normalize the Credit History Age
max_age_months = credit_data['Credit_History_Age_Months'].max()
credit_data['Length_of_Credit_History_Score'] = credit_data['Credit_History_Age_Months'] / max_age_months

# Scale to fit into the final score (15% of the total score)
credit_data['Length_of_Credit_History_Score'] *= 0.15
```

```
# Display the results
print(credit_data[['Customer_ID', 'Credit_History_Age', 'Length_of_Credit_History_Score']])
```

```
      Customer_ID  Credit_History_Age  Length_of_Credit_History_Score
0       CUS_0x1000            0.300248                        0.045297
1       CUS_0x1000            0.302730                        0.045668
2       CUS_0x1000            0.305211                        0.046040
3       CUS_0x1000            0.307692                        0.046411
4       CUS_0x1000            0.310174                        0.046782
...            ...                 ...                             ...
99995   CUS_0xffd             0.545906                        0.082054
99996   CUS_0xffd             0.548387                        0.082426
99997   CUS_0xffd             0.550868                        0.082797
99998   CUS_0xffd             0.553350                        0.083168
99999   CUS_0xffd             0.555831                        0.083540

[100000 rows x 3 columns]
```

New Credit Accounts (10%)

```
credit_data['Num_Credit_Inquiries'].sample(10)
```

	Num_Credit_Inquiries
43463	12.0
2116	6.0
65294	3.0
87181	3.0
22864	9.0
97858	7.0
7586	0.0
16057	3.0
82688	1.0
42084	7.0

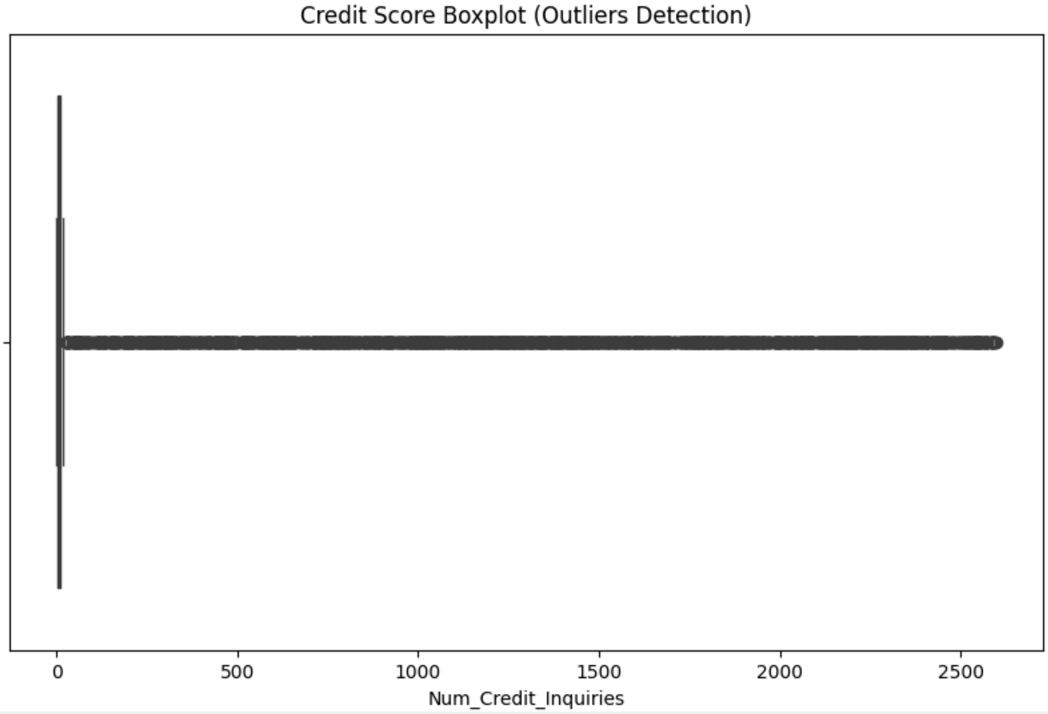
dtype: float64

```
credit_data['Num_Credit_Inquiries'].isna().sum()
```

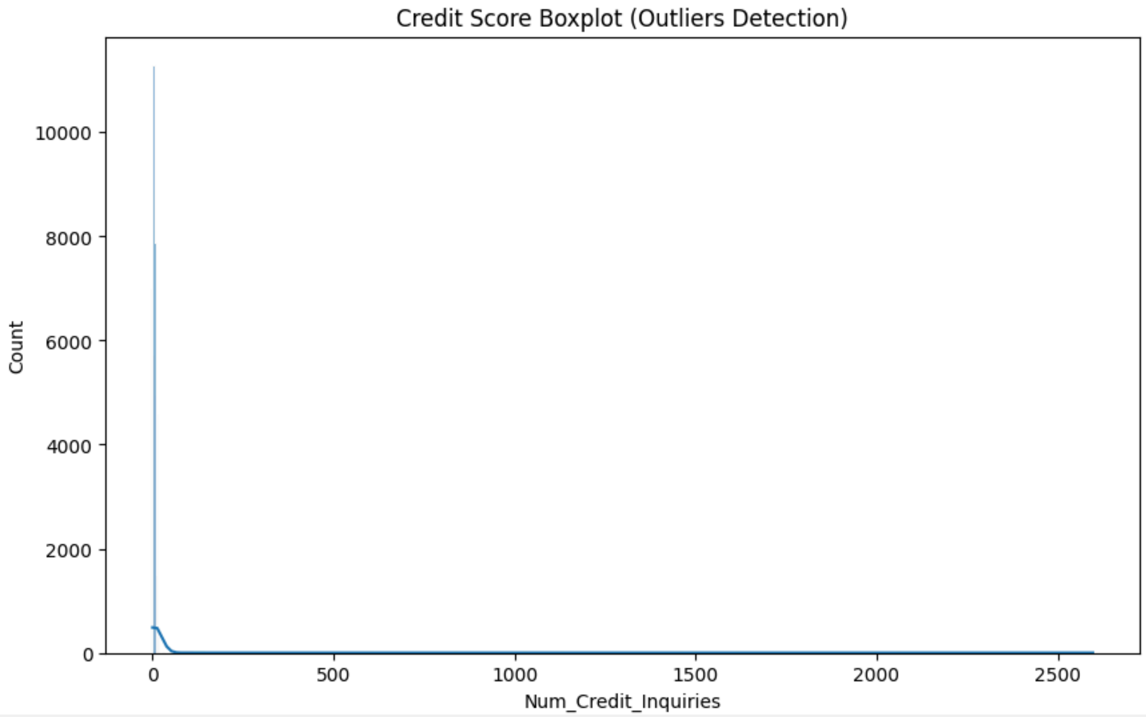
```
1965
```

```
plt.figure(figsize=(10, 6))
sns.boxplot(x=credit_data['Num_Credit_Inquiries'])
plt.title('Credit Score Boxplot (Outliers Detection)')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```




```
plt.figure(figsize=(10, 6))
sns.histplot(x=credit_data['Num_Credit_Inquiries'], kde=True)
plt.title('Credit Score Boxplot (Outliers Detection)')
plt.show()
```



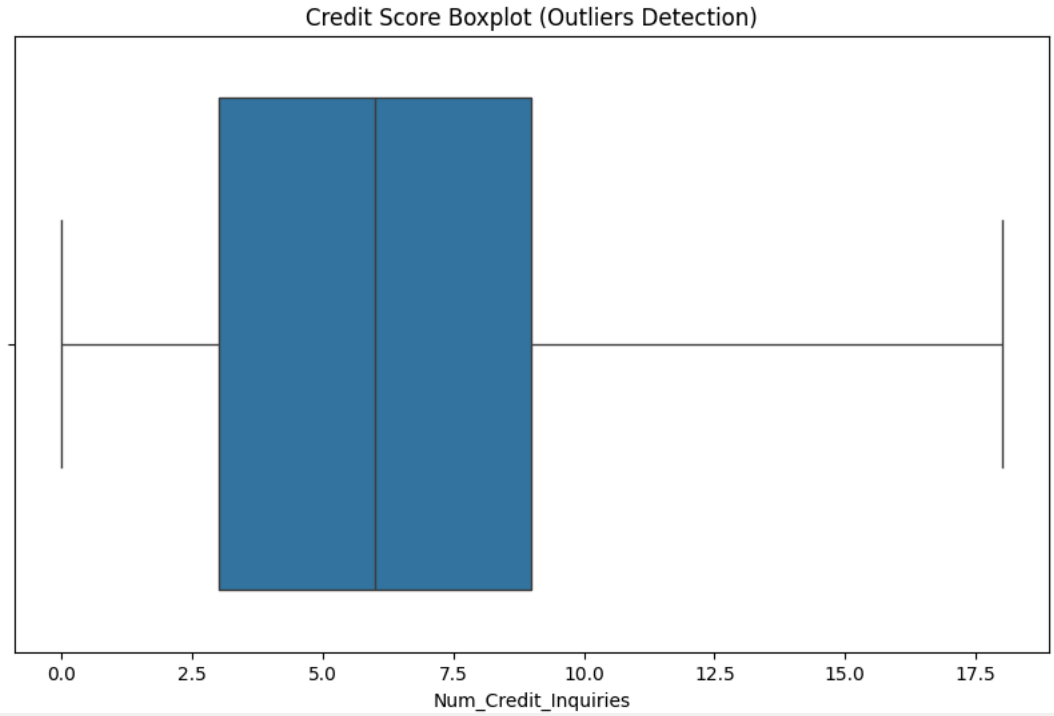

```
# Step 1: Calculate the median for each Customer_ID
median_inquiries_per_id = credit_data.groupby('Customer_ID')['Num_Credit_Inquiries'].median()

# Step 2: Map the median values to the original DataFrame to fill NaN
credit_data['Num_Credit_Inquiries'] = credit_data['Num_Credit_Inquiries'].fillna(
    credit_data['Customer_ID'].map(median_inquiries_per_id)
)
```

```
credit_data['Num_Credit_Inquiries'].isna().sum()
```

```
0
```

```
# 2. Identify and replace outliers using IQR method
Q1 = credit_data['Num_Credit_Inquiries'].quantile(0.25)
Q3 = credit_data['Num_Credit_Inquiries'].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Clip outliers to upper and lower bounds
credit_data['Num_Credit_Inquiries'] = credit_data['Num_Credit_Inquiries'].clip(lower=lower_bound, upper=upper_bound)


plt.figure(figsize=(10, 6))
sns.boxplot(x=credit_data['Num_Credit_Inquiries'])
plt.title('Credit Score Boxplot (Outliers Detection)')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```




```
# Normalize the Num_Credit_Inquiries
max_inquiries = credit_data['Num_Credit_Inquiries'].max()
credit_data['New_Credit_Accounts_Score'] = 1 - (credit_data['Num_Credit_Inquiries'] / max_inquiries)

# Scale to fit into the final score (10% of the total score)
credit_data['New_Credit_Accounts_Score'] *= 0.10

# Display the results
print(credit_data[['Customer_ID', 'Num_Credit_Inquiries', 'New_Credit_Accounts_Score']])
```

```
      Customer_ID  Num_Credit_Inquiries  New_Credit_Accounts_Score
0      CUS_0x1000                  10.0                   0.044444
1      CUS_0x1000                  11.0                   0.038889
2      CUS_0x1000                  11.0                   0.038889
3      CUS_0x1000                  11.0                   0.038889
4      CUS_0x1000                  11.0                   0.038889
...           ...                   ...                        ...
99995   CUS_0xffd                   7.0                   0.061111
99996   CUS_0xffd                   7.0                   0.061111
99997   CUS_0xffd                   7.0                   0.061111
99998   CUS_0xffd                   7.0                   0.061111
99999   CUS_0xffd                   7.0                   0.061111

[100000 rows x 3 columns]
```

Types of Credit Used (10%)

```
credit_data[['Customer_ID', 'Credit_Mix', 'Num_Bank_Accounts', 'Num_Credit_Card', 'Num_of_Loan']]
```

	Customer_ID	Credit_Mix	Num_Bank_Accounts	Num_Credit_Card	Num_of_Loan
0	CUS_0x1000	Bad	0.6	5	2
1	CUS_0x1000	Bad	0.6	5	2
2	CUS_0x1000	Bad	0.6	5	2
3	CUS_0x1000	Bad	0.6	5	2
4	CUS_0x1000	Bad	0.6	5	2
...	...	...	...	...	...
99995	CUS_0xffd	_	0.8	7	-100
99996	CUS_0xffd	Standard	0.8	7	6_
99997	CUS_0xffd	Standard	0.8	7	6
99998	CUS_0xffd	Standard	0.8	7	6
99999	CUS_0xffd	Standard	0.8	7	6

100000 rows × 5 columns

```
credit_data[['Customer_ID', 'Credit_Mix', 'Num_Bank_Accounts', 'Num_Credit_Card', 'Num_of_Loan']].info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 5 columns):
 #   Column             Non-Null Count   Dtype  
---  ------             --------------   -----  
 0   Customer_ID        100000 non-null  object 
 1   Credit_Mix         100000 non-null  object 
 2   Num_Bank_Accounts  100000 non-null  float64
 3   Num_Credit_Card    100000 non-null  int64  
 4   Num_of_Loan        100000 non-null  object 
dtypes: float64(1), int64(1), object(3)
memory usage: 3.8+ MB
```

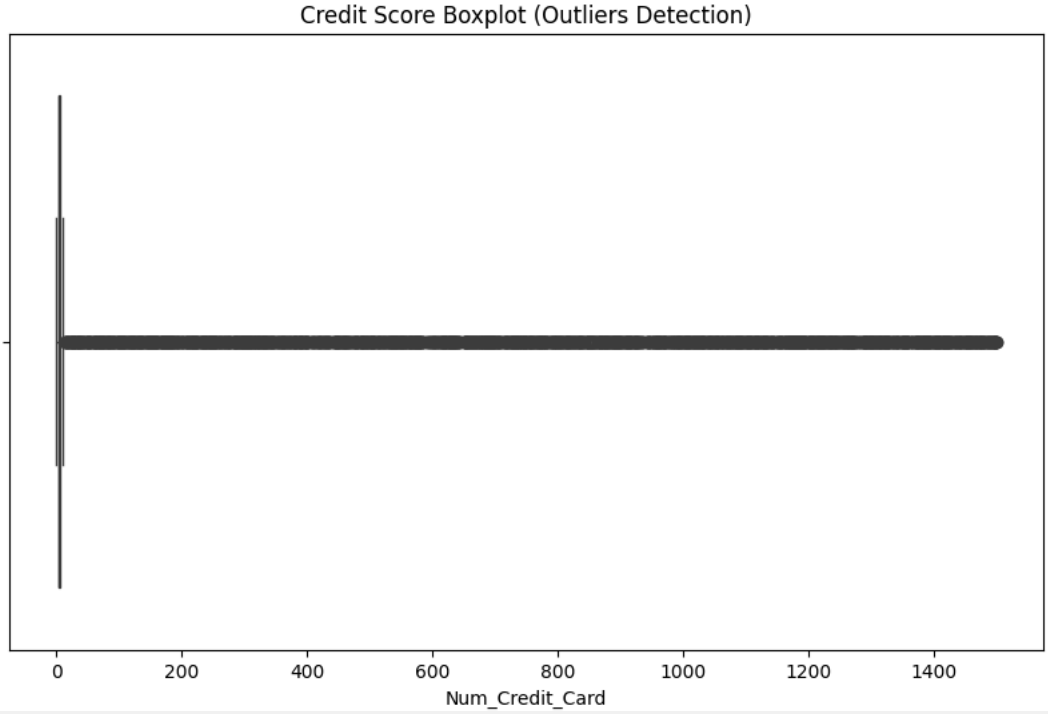
```
# Step 1: Define a function to replace "_" with the mode of Credit_Mix for each Customer_ID
def replace_credit_mix_with_mode(df):
    mode_mapping = df.groupby('Customer_ID')['Credit_Mix'].agg(lambda x: x.mode()[0] if not x.mode().empty else None)

    # Replace "_" with the mode value
    df['Credit_Mix'] = df.apply(
        lambda row: mode_mapping[row['Customer_ID']] if row['Credit_Mix'] == '_' else row['Credit_Mix'], axis=1
    )
    return df

# Step 2: Apply the function to clean the Credit_Mix column
credit_data = replace_credit_mix_with_mode(credit_data)
```

```
credit_data[['Customer_ID', 'Credit_Mix', 'Num_Bank_Accounts', 'Num_Credit_Card', 'Num_of_Loan']].sample(10)
```

	Customer_ID	Credit_Mix	Num_Bank_Accounts	Num_Credit_Card	Num_of_Loan
92705	CUS_0xbca4	Good	0.0	5	2_
24719	CUS_0x406d	Bad	1.0	8	6_
91603	CUS_0xbab0	Standard	0.5	5	4
98176	CUS_0xc730	Standard	0.7	4	6
94706	CUS_0xc063	Standard	0.5	4	3
72608	CUS_0x97d	Bad	0.7	9	6
21855	CUS_0x3b15	Standard	0.3	5	1
59132	CUS_0x7eb6	Bad	0.9	96	8
47374	CUS_0x68d1	Bad	0.7	6	8
90415	CUS_0xb87a	Good	0.4	5	0

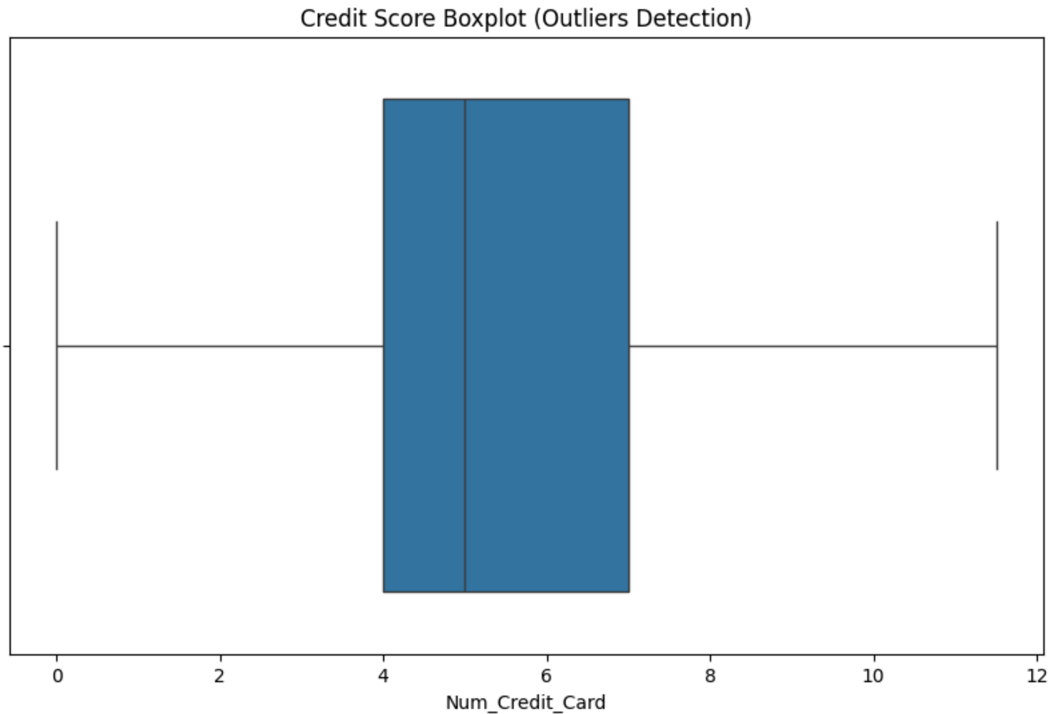
```
plt.figure(figsize=(10, 6))
sns.boxplot(x=credit_data['Num_Credit_Card'])
plt.title('Credit Score Boxplot (Outliers Detection)')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```




```
# Step 1: Calculate Q1 (25th percentile) and Q3 (75th percentile)
Q1 = credit_data['Num_Credit_Card'].quantile(0.25)
Q3 = credit_data['Num_Credit_Card'].quantile(0.75)
IQR = Q3 - Q1

# Step 2: Define the lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Step 3: Clip the outliers
credit_data['Num_Credit_Card'] = credit_data['Num_Credit_Card'].clip(lower=lower_bound, upper=upper_bound)



plt.figure(figsize=(10, 6))
sns.boxplot(x=credit_data['Num_Credit_Card'])
plt.title('Credit Score Boxplot (Outliers Detection)')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```



```
# Step 1: Replace -100 with 0
credit_data['Num_of_Loan'] = credit_data['Num_of_Loan'].replace(-100, 0)

# Step 2: Remove trailing underscores and convert to integers
credit_data['Num_of_Loan'] = credit_data['Num_of_Loan'].replace(r'[^\d]', '', regex=True)  # Remove non-numeric characters
credit_data['Num_of_Loan'] = credit_data['Num_of_Loan'].str.rstrip('_')  # Remove trailing underscores
credit_data['Num_of_Loan'] = pd.to_numeric(credit_data['Num_of_Loan'], errors='coerce').fillna(0).astype(int)  # Convert to integers, f:

# Display the cleaned DataFrame
print(credit_data[['Num_of_Loan']].isna().sum())
```

```
Num_of_Loan    0
dtype: int64
```

```
credit_data[['Customer_ID', 'Credit_Mix', 'Num_Bank_Accounts', 'Num_Credit_Card', 'Num_of_Loan']].sample(10)
```

	Customer_ID	Credit_Mix	Num_Bank_Accounts	Num_Credit_Card	Num_of_Loan
85820	CUS_0xb01a	Bad	1.0	7.0	9
67929	CUS_0x8f0c	Standard	0.8	3.0	3
95137	CUS_0xc143	Standard	0.3	3.0	2
55248	CUS_0x77a2	Standard	0.6	6.0	1
55089	CUS_0x7749	Bad	0.6	7.0	6
98305	CUS_0xcb9	Standard	0.5	4.0	1
97756	CUS_0xc650	Standard	0.5	3.0	6
25427	CUS_0x41a8	Standard	0.3	7.0	3
77750	CUS_0xa130	Standard	0.3	4.0	0
77144	CUS_0xa015	Bad	0.7	5.0	8

```
# Step 1: Assign weights for credit mix
credit_mix_weights = {
    'Good': 1.0,
    'Average': 0.5,
    'Bad': 0.0
}

# Step 2: Create a function to calculate the score based on the weights
def calculate_credit_mix_score(row):
    credit_mix_score = credit_mix_weights.get(row['Credit_Mix'], 0)  # Default to 0 if not found
    num_accounts = row['Num_Bank_Accounts']
    num_credit_cards = row['Num_Credit_Card']
    num_loans = row['Num_of_Loan']

    # Combining the features with weights (you can adjust these weights as needed)
    score = (credit_mix_score * 0.4) + (num_accounts * 0.1) + (num_credit_cards * 0.1) + (num_loans * 0.2)
    return score

# Step 3: Apply the function to create a new feature
credit_data['Credit_Used_Score'] = credit_data.apply(calculate_credit_mix_score, axis=1)

# Display the DataFrame with the new feature
print(credit_data[['Customer_ID', 'Credit_Mix', 'Num_Bank_Accounts', 'Num_Credit_Card', 'Num_of_Loan', 'Credit_Used_Score']].sample(10)]
```

```
      Customer_ID Credit_Mix  Num_Bank_Accounts  Num_Credit_Card  Num_of_Loan  \
12475  CUS_0x291b       Good                0.8              3.0            3
99241   CUS_0xe7e        Bad                0.8             10.0            7
89163  CUS_0xb647   Standard                0.3              5.0            3
70728  CUS_0x9444        Bad                0.7              7.0            5
41246  CUS_0x5e16       Good                0.1              5.0            1
96381  CUS_0xc3ae       Good                0.3              1.0            4
16230  CUS_0x2fe0        Bad                0.6             10.0            6
39782  CUS_0x5b6c        Bad                0.8              4.0            7
109    CUS_0x103e       Good                0.4              6.0            1
29434  CUS_0x488b   Standard                0.6             10.0            7

       Credit_Used_Score
12475               1.38
99241               2.48
89163               1.13
70728               1.77
41246               1.11
96381               1.33
16230               2.26
39782               1.88
109                 1.24
29434               2.46
```

```
print(credit_data[['Customer_ID', 'Credit_Mix', 'Num_Bank_Accounts', 'Num_Credit_Card', 'Num_of_Loan', 'Credit_Used_Score']].sample(10))
```

```
       Customer_ID Credit_Mix  Num_Bank_Accounts  Num_Credit_Card  Num_of_Loan  \
68015  CUS_0x8f34   Standard                0.3              4.0            3
61568  CUS_0x8380        Bad                0.9              6.0            6
45225  CUS_0x6503   Standard                0.3              3.0            7
95393  CUS_0xc1bb       Good                0.0              7.0            0
99942   CUS_0xfe4   Standard                0.7              3.0            7
44954  CUS_0x6484   Standard                0.6              3.0          100
84518  CUS_0xade0       Good                0.1              2.0            1
57387  CUS_0x7b7a        Bad                0.8              8.0            7
8792   CUS_0x2226   Standard                0.3              5.0            4
87609  CUS_0xb38b   Standard                0.3              4.0            6

       Credit_Used_Score
68015               1.03
61568               1.89
45225               1.73
95393               1.10
99942               1.77
44954              20.36
84518               0.81
57387               2.28
8792                1.33
87609               1.63
```

Hypothetical FICO Score

```
print(credit_data[['Customer_ID','Payment_History_Score','Amounts_Owed_Score','Length_of_Credit_History_Score','New_Credit_Accounts_Scor
```

```
       Customer_ID  Payment_History_Score  Amounts_Owed_Score  \
31301  CUS_0x4bf4               0.903164            0.876151
16390  CUS_0x302f               0.939761            0.801328
70379   CUS_0x939               0.699304            0.669501
8946   CUS_0x2288               0.887244            0.629857
30884  CUS_0x4b19               0.692219            0.734409
36538  CUS_0x553e               0.806995            0.680466
55836  CUS_0x78ce               0.857015            0.841301
20022  CUS_0x379f               0.902219            0.638598
33884  CUS_0x5053               0.914537            0.820714
13344  CUS_0x2abc               0.757095            0.678594

       Length_of_Credit_History_Score  New_Credit_Accounts_Score  \
31301                        0.093564                   0.061111
16390                        0.070916                   0.083333
70379                        0.062748                   0.038889
8946                         0.145173                   0.077778
30884                        0.105446                   0.055556
36538                        0.070916                   0.066667
55836                        0.105074                   0.088889
20022                        0.094678                   0.072222
33884                        0.142203                   0.088889
13344                        0.131807                   0.094444

       Credit_Used_Score
31301               0.91
16390               1.04
70379               2.37
8946               21.59
30884               0.33
36538               0.48
55836               1.57
20022               1.22
33884               1.83
13344               0.53
```

```
# Define a function to calculate the hypothetical FICO score
def calculate_fico_score(row):
    # Payment History Score (scaled to 35% of total score)
    payment_history_score = row['Payment_History_Score'] * 0.35

    # Amounts Owed Score (assuming lower is better, scaled to 30%)
    amounts_owed_score = (1 - row['Amounts_Owed_Score']) * 0.30  # Inverse scaling for amounts owed

    # Length of Credit History Score (already scaled to 15%)
    length_of_credit_history_score = row['Length_of_Credit_History_Score']  # No additional scaling

    # New Credit Accounts Score (scaled to 10%)
    new_credit_accounts_score = row['New_Credit_Accounts_Score'] # No additional scaling

    # Credit Used Score (scaled to 10%)
    credit_used_score = row['Credit_Used_Score'] * 0.10
```

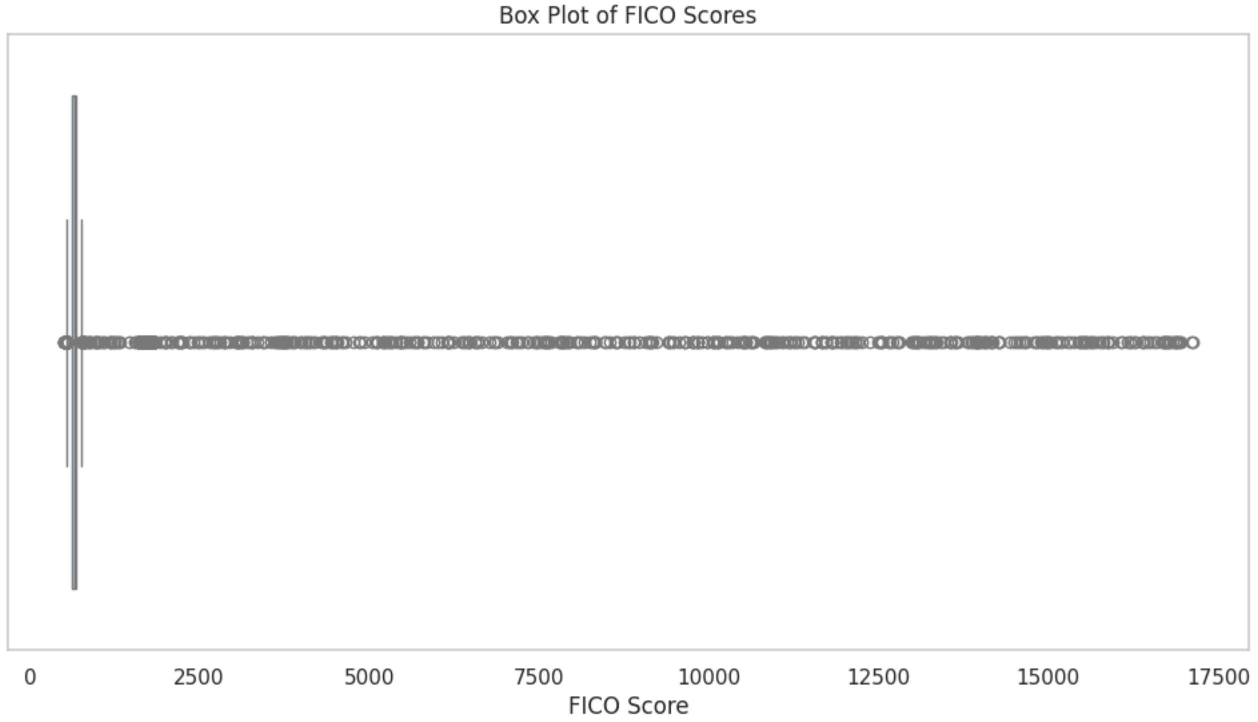
```
    # Calculate total score
    total_score = (payment_history_score + amounts_owed_score +
                   length_of_credit_history_score + new_credit_accounts_score +
                   credit_used_score)

    # Scale the total score to fit into the typical FICO score range (300 to 850)
    return total_score * (850 - 300) + 300  # Adjusting to 300-850 scale

# Calculate the hypothetical FICO score for each customer
credit_data['FICO_Score'] = credit_data.apply(calculate_fico_score, axis=1)

# Display the first few rows to check the new hypothetical FICO score
print(credit_data[['Customer_ID', 'FICO_Score']].head())
```

```
  Customer_ID  FICO_Score
0  CUS_0x1000  509.567638
1  CUS_0x1000  512.937156
2  CUS_0x1000  532.602935
3  CUS_0x1000  520.832881
4  CUS_0x1000  519.909428
```

```
# Set the aesthetic style of the plots
sns.set(style="whitegrid")

# Plot histogram of FICO Scores
plt.figure(figsize=(12, 6))
sns.histplot(credit_data['FICO_Score'], bins=30, kde=True, color='blue')
plt.title('Distribution of FICO Scores')
plt.xlabel('FICO Score')
plt.ylabel('Frequency')
plt.xlim(300, 850)
plt.grid(axis='y')
plt.show()
```


```
# Plot box plot of FICO Scores
plt.figure(figsize=(12, 6))
sns.boxplot(x=credit_data['FICO_Score'], color='lightblue')
plt.title('Box Plot of FICO Scores')
plt.xlabel('FICO Score')
plt.grid(axis='x')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```




```
# Calculate the upper whisker value
Q1 = credit_data['FICO_Score'].quantile(0.25)
Q3 = credit_data['FICO_Score'].quantile(0.75)
IQR = Q3 - Q1
upper_whisker = Q3 + 1.5 * IQR

# Identify outliers
outliers = credit_data[credit_data['FICO_Score'] > upper_whisker]
print(outliers[['Customer_ID', 'FICO_Score']])
```

```
      Customer_ID    FICO_Score
11     CUS_0x1009  12652.282593
54     CUS_0x1018   1688.972948
101    CUS_0x1038   1674.679515
128    CUS_0x1048  14045.786870
150    CUS_0x104e   1748.879433
...           ...           ...
99852  CUS_0xfb6    1766.594128
99881  CUS_0xfcb    1699.308954
99898  CUS_0xfd1    1744.747128
99927  CUS_0xfdf    1783.916727
99995  CUS_0xffd    1714.771223

[4371 rows x 2 columns]
```

```
# Check the features of the identified outliers
outlier_customer_ids = outliers['Customer_ID']
outlier_features = credit_data[credit_data['Customer_ID'].isin(outlier_customer_ids)]
print(outlier_features)
```

```
       Annual_Income  Monthly_Inhand_Salary  Num_Bank_Accounts  ...  \
8           52312.68               0.264865                0.6  ...
9           52312.68               0.264865                0.6  ...
10          52312.68               0.264865                0.6  ...
11          52312.68               0.264865                0.6  ...
12          52312.68               0.264865                0.6  ...
...              ...                    ...                ...  ...
```

```
99996                 0.002729                 0.343284
99997                 0.002729                 0.343284
99998                 0.002729                 0.343284
99999                 0.002502                 0.343284

       Payment_Behaviour_numeric Payment_of_Min_Amount_numeric  \
8                            0.3                              1
9                            0.4                              1
10                           0.4                              1
11                           0.1                              1
12                           0.4                              1
...                          ...                            ...
99995                        NaN                              1
99996                        0.4                              1
99997                        0.4                              1
99998                        0.1                              1
99999                        NaN                              1

       Payment_History_Score  Amounts_Owed_Score  \
8                   0.776776            0.934456
9                   0.776776            0.870833
10                  0.776776            0.898556
11                  0.776776            0.916900
12                  0.776776            0.845092
...                      ...                 ...
99995               0.698896            0.644892
99996               0.698896            0.781132
99997               0.698896            0.637085
99998               0.698896            0.677801
99999               0.698896            0.597797

      Length_of_Credit_History_Score  New_Credit_Accounts_Score  \
8                           0.135520                   0.088889
9                           0.135891                   0.088889
10                          0.136262                   0.088889
11                          0.137005                   0.088889
12                          0.137005                   0.077778
```

```
# Identify and print outliers
high_fico_threshold = credit_data['FICO_Score'].quantile(0.95)  # You can adjust this threshold
outliers = credit_data[credit_data['FICO_Score'] > high_fico_threshold]
print(outliers[['Customer_ID', 'FICO_Score', 'Credit_Used_Score', 'Payment_History_Score', 'Amounts_Owed_Score']])
```

```
      Customer_ID    FICO_Score  Credit_Used_Score  Payment_History_Score  \
11     CUS_0x1009  12652.282593             219.36               0.776776
54     CUS_0x1018   1688.972948              20.77               0.692876
101    CUS_0x1038   1674.679515              20.43               0.692697
128    CUS_0x1048  14045.786870             245.13               0.787572
137    CUS_0x104a    753.049271               1.92               0.924299
...           ...           ...                ...                    ...
99852   CUS_0xfb6   1766.594128              20.74               0.888597
99881   CUS_0xfcb   1699.308954              20.65               0.773512
99898   CUS_0xfd1   1744.747128              20.54               0.787313
99927   CUS_0xfdf   1783.916727              20.80               0.914736
99995   CUS_0xffd   1714.771223              20.78               0.698896

       Amounts_Owed_Score
11               0.916900
54               0.709252
101              0.729458
128              0.664590
137              0.703389
...                   ...
99852            0.735536
99881            0.861302
99898            0.669222
99927            0.653681
99995            0.644892

[5000 rows x 5 columns]
```

Improve Data Quality

```
# Calculate the upper bound for outliers (e.g., using the 95th percentile)
upper_bound_credit_used = credit_data['Credit_Used_Score'].quantile(0.95)
credit_data.loc[credit_data['Credit_Used_Score'] > upper_bound_credit_used, 'Credit_Used_Score'] = upper_bound_credit_used
```

```
# Apply log transformation to Credit_Used_Score (ensure all values are positive)
credit_data['Credit_Used_Score'] = np.log1p(credit_data['Credit_Used_Score'])
```

```
# Check for records with high Credit_Used_Score
high_credit_used_records = credit_data[credit_data['Credit_Used_Score'] > 50]  # Adjust threshold as needed
print(high_credit_used_records)
```

```
Empty DataFrame
Columns: [ID, Customer_ID, Month, Name, Age, SSN, Occupation, Annual_Income, Monthly_Inhand_Salary, Num_Bank_Accounts, Num_Credit_Ca
Index: []

[0 rows x 46 columns]
```

```
# Cap the Credit_Used_Score at a reasonable max value
credit_data['Credit_Used_Score'] = credit_data['Credit_Used_Score'].clip(upper=10)  # Example max value
```

```
# Normalize Credit_Used_Score to be between 0 and 1
credit_data['Credit_Used_Score'] = (credit_data['Credit_Used_Score'] - credit_data['Credit_Used_Score'].min()) / (credit_data['Credit_Us
```

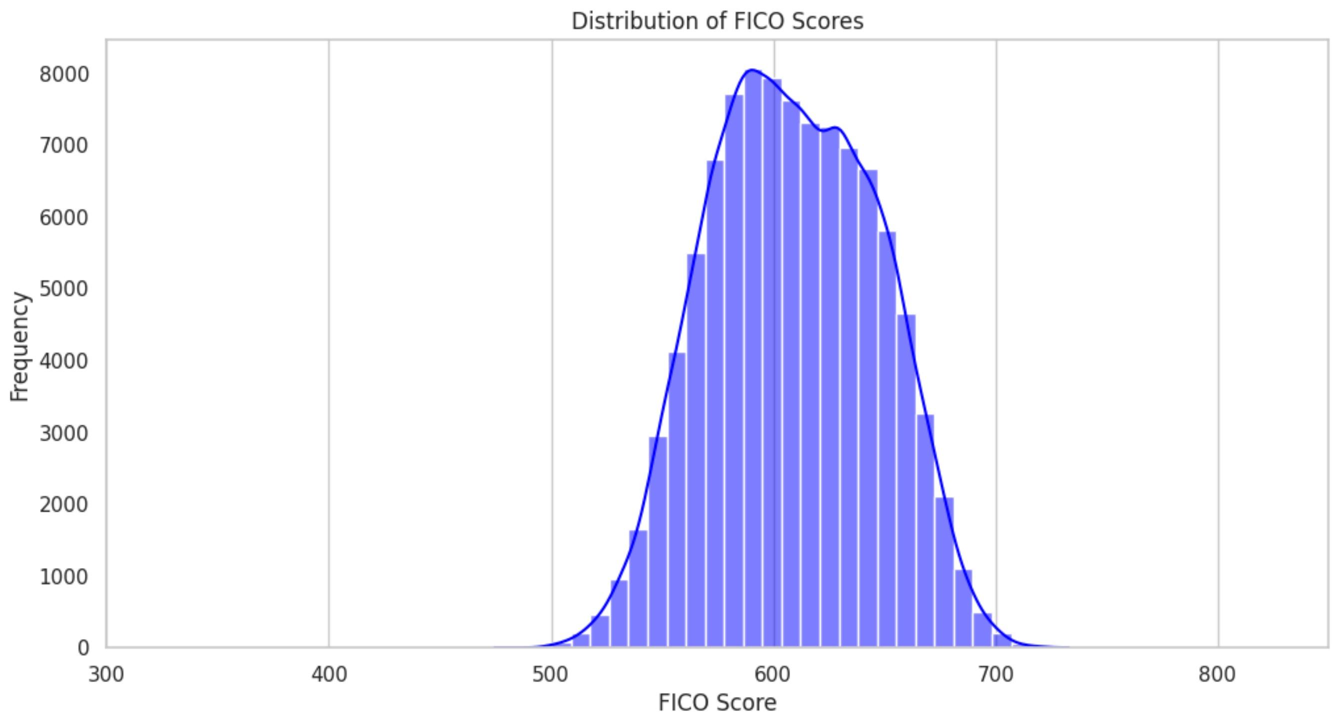
```
# Calculate correlation matrix
correlation_matrix = credit_data[['FICO_Score', 'Credit_Used_Score']].corr()
print(correlation_matrix[['FICO_Score', 'Credit_Used_Score']])
```

```
                   FICO_Score  Credit_Used_Score
FICO_Score           1.000000           0.235413
Credit_Used_Score    0.235413           1.000000
```

```
# Define a function to calculate the hypothetical FICO score
def calculate_fico_score(row):
    # Payment History Score (scaled to 35% of total score)
    payment_history_score = row['Payment_History_Score'] * 0.35

    # Amounts Owed Score (assuming lower is better, scaled to 30%)
    amounts_owed_score = (1 - row['Amounts_Owed_Score']) * 0.30  # Inverse scaling for amounts owed

    # Length of Credit History Score (already scaled to 15%)
    length_of_credit_history_score = row['Length_of_Credit_History_Score']  # No additional scaling

    # New Credit Accounts Score (scaled to 10%)
    new_credit_accounts_score = row['New_Credit_Accounts_Score'] # No additional scaling

    # Credit Used Score (scaled to 10%)
    credit_used_score = row['Credit_Used_Score'] * 0.10

    # Calculate total score
    total_score = (payment_history_score + amounts_owed_score +
                   length_of_credit_history_score + new_credit_accounts_score +
                   credit_used_score)

    # Scale the total score to fit into the typical FICO score range (300 to 850)
    return total_score * (850 - 300) + 300  # Adjusting to 300-850 scale

# Calculate the hypothetical FICO score for each customer
credit_data['FICO_Score'] = credit_data.apply(calculate_fico_score, axis=1)

# Display the first few rows to check the new hypothetical FICO score
print(credit_data[['Customer_ID', 'FICO_Score']].head())
```

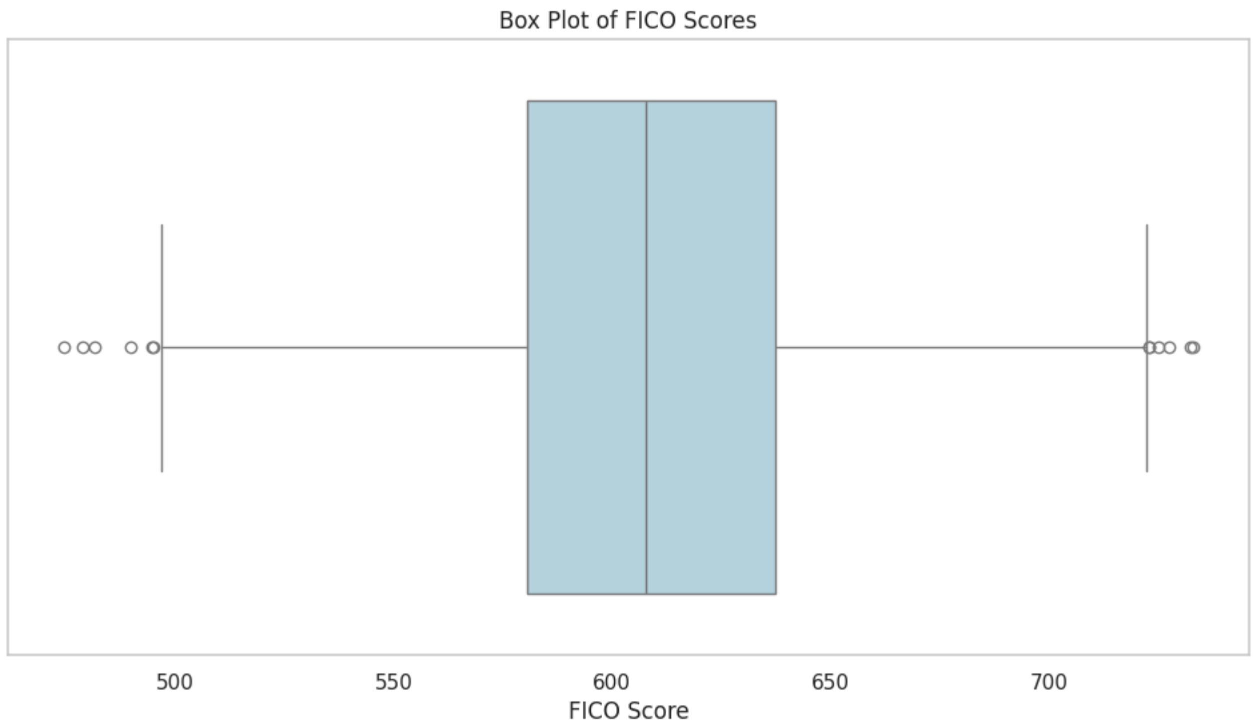
```
  Customer_ID  FICO_Score
0  CUS_0x1000  509.567638
1  CUS_0x1000  512.937156
2  CUS_0x1000  532.602935
3  CUS_0x1000  520.832881
4  CUS_0x1000  519.909428
```

```
# Set the aesthetic style of the plots
sns.set(style="whitegrid")

# Plot histogram of FICO Scores
plt.figure(figsize=(12, 6))
sns.histplot(credit_data['FICO_Score'], bins=30, kde=True, color='blue')
plt.title('Distribution of FICO Scores')
plt.xlabel('FICO Score')
plt.ylabel('Frequency')
plt.xlim(300, 850)
plt.grid(axis='y')
plt.show()
```




```
# Plot box plot of FICO Scores
plt.figure(figsize=(12, 6))
sns.boxplot(x=credit_data['FICO_Score'], color='lightblue')
plt.title('Box Plot of FICO Scores')
plt.xlabel('FICO Score')
plt.grid(axis='x')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/seaborn/categorical.py:640: FutureWarning: SeriesGroupBy.grouper is deprecated and will be r
  positions = grouped.grouper.result_index.to_numpy(dtype=float)
```




```
# Define risk categories based on FICO score ranges
def categorize_fico(score):
    if score < 580:
        return 'Poor'
    elif score < 670:
        return 'Fair'
    elif score < 740:
        return 'Good'
    elif score < 800:
        return 'Very Good'
    else:
```

```
        else:
            return 'Excellent'

    # Apply the categorization function
```