Project Title: Credit Score Prediction

Step 1: Dataset Download-

- Obtain the dataset containing relevant credit-related information.
- Highlight the features, including income, outstanding debt, credit history, etc.
- Identify the target variable: Credit_Score.

Step 1: Solution-

Import necessary libraries:

In [1]:
```
# Import the numerical algebra libs
import pandas as pd
import numpy as np

# Import visualization libs
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px
import string

#Import warnings libs
import warnings
warnings.filterwarnings('ignore')
```

Load Dataset:

In [2]:
```
data = pd.read_csv('Bank Data.csv')
data.head()
```

Out[2]:

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accoun
0	0x160a	CUS_0xd40	September	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843333	
1	0x160b	CUS_0xd40	October	Aaron Maashoh	24	821-00-0265	Scientist	19114.12	1824.843333	
2	0x160c	CUS_0xd40	November	Aaron Maashoh	24	821-00-0265	Scientist	19114.12	1824.843333	
3	0x160d	CUS_0xd40	December	Aaron Maashoh	24_	821-00-0265	Scientist	19114.12	NaN	
4	0x1616	CUS_0x21b1	September	Rick Rothackerj	28	004-07-5839	_______	34847.84	3037.986667	

5 rows × 27 columns

Dataset description:

- **ID:** Unique identifier for each record in the dataset.
- **Customer_ID:** Unique identifier for each customer.
- **Month:** Month of the data record.
- **Name:** Customer's name.
- **Age:** Age of the customer.
- **SSN:** Social Security Number or a unique identification number.
- **Occupation:** Customer's occupation or job title.
- **Annual_Income:** Annual income of the customer.
- **Monthly_Inhand_Salary:** Net monthly income available to the customer.
- **Num_Bank_Accounts:** Number of bank accounts the customer holds.
- **Num_Credit_Card:** Number of credit cards owned by the customer.
- **Interest_Rate:** Interest rate associated with financial transactions.
- **Num_of_Loan:** Number of loans the customer has.
- **Type_of_Loan:** Type or category of the loan.
- **Delay_from_due_date:** Delay in payment from the due date.
- **Num_of_Delayed_Payment:** Number of delayed payments.
- **Changed_Credit_Limit:** Any recent changes in the customer's credit limit.
- **Num_Credit_Inquiries:** Number of credit inquiries made by the customer.
- **Credit_Mix:** Variety of credit types in the customer's financial profile.
- **Outstanding_Debt:** Total outstanding debt of the cust

- **Outstanding_Debt:** Total outstanding debt of the customer.
- **Credit_Utilization_Ratio:** Ratio of credit used to credit available.
- **Credit_History_Age:** Age of the customer's credit history.
- **Payment_of_Min_Amount:** Payment behavior regarding the minimum amount due.
- **Total_EMI_per_month:** Total Equated Monthly Installments paid by the customer.
- **Amount_invested_monthly:** Amount invested by the customer monthly.
- **Payment_Behaviour:** General behavior regarding payments.
- **Monthly_Balance:** Monthly balance in the customer's financial accounts.

Key Features:

Income and Financial Status:

- Annual_Income
- Monthly_Inhand_Salary
- Outstanding_Debt
- Monthly_Balance
- Amount_invested_monthly

Credit History and Behavior:

- Credit_History_Age
- Credit_Utilization_Ratio
- Num_Credit_Inquiries
- Credit_Mix
- Payment_Behaviour
- Payment_of_Min_Amount
- Num_of_Delayed_Payment
- Delay_from_due_date

Account Information:

- Num_Bank_Accounts
- Num_Credit_Card
- Changed_Credit_Limit

Loan Information:

- Num_of_Loan
- Type_of_Loan
- Interest_Rate
- Total_EMI_per_month

Personal Information:

- Age
- Occupation

These features can be used to predict our target variable: **Credit Score**.

Observation:

- The target variable "Credit_Score" is not present in the provided list of columns. This suggests that the dataset doesn't include the credit score.
- I can infer that the "Credit_Mix" or "Payment_Behaviour" might be related to the credit score.

Identify Target Variable:

"Credit_Mix" and "Payment_Behaviour" can indeed provide valuable insights into a customer's creditworthiness, which is often reflected in their credit score. Here's how these features might relate to credit scoring:

- **`Credit Mix:`** This represents the diversity of credit accounts (e.g., credit cards, mortgages, loans). A good mix of credit types is often associated with a healthier credit score, as it shows that the customer can manage different types of credit responsibly.

- **`Payment Behaviour:`** This indicates the customer's spending and payment patterns. Regular, timely payments are crucial for maintaining a good credit score, while delayed or missed payments can negatively impact it.

Observation:

- Since we don't have an explicit "Credit_Score" column, we'll use "Credit_Mix" as our target variable, as it seems to be the closest indicator of credit worthiness in this dataset.

Step 2: Data Exploration and Preprocessing-

- Conduct exploratory data analysis (EDA) to understand the distribution of features and the target variable.
- Handle any missing values, outliers, or data inconsistencies.
- Encode categorical variables if necessary.
- Explore the distribution of the target variable.

Step 2: Solution-

Visualize Dataset:

In [3]:
```
data.columns
```

Out[3]:
```
Index(['ID', 'Customer_ID', 'Month', 'Name', 'Age', 'SSN', 'Occupation',
       'Annual_Income', 'Monthly_Inhand_Salary', 'Num_Bank_Accounts',
       'Num_Credit_Card', 'Interest_Rate', 'Num_of_Loan', 'Type_of_Loan',
       'Delay_from_due_date', 'Num_of_Delayed_Payment', 'Changed_Credit_Limit',
       'Num_Credit_Inquiries', 'Credit_Mix', 'Outstanding_Debt',
       'Credit_Utilization_Ratio', 'Credit_History_Age',
       'Payment_of_Min_Amount', 'Total_EMI_per_month',
       'Amount_invested_monthly', 'Payment_Behaviour', 'Monthly_Balance'],
      dtype='object')
```

In [4]:
```
data.shape
```

Out[4]:
```
(50000, 27)
```

In [5]:
```
data.describe().transpose()
```

Out[5]:

	count	mean	std	min	25%	50%	75%	max
Monthly_Inhand_Salary	42502.0	4182.004291	3174.109304	303.645417	1625.188333	3086.305000	5934.189094	15204.633333
Num_Bank_Accounts	50000.0	16.838260	116.396848	-1.000000	3.000000	6.000000	7.000000	1798.000000
Num_Credit_Card	50000.0	22.921480	129.314804	0.000000	4.000000	5.000000	7.000000	1499.000000
Interest_Rate	50000.0	68.772640	451.602363	1.000000	8.000000	13.000000	20.000000	5799.000000
Delay_from_due_date	50000.0	21.052640	14.860397	-5.000000	10.000000	18.000000	28.000000	67.000000
Num_Credit_Inquiries	48965.0	30.080200	196.984121	0.000000	4.000000	7.000000	10.000000	2593.000000
Credit_Utilization_Ratio	50000.0	32.279581	5.106238	20.509652	28.061040	32.280390	36.468591	48.540663
Total_EMI_per_month	50000.0	1491.304305	8595.647887	0.000000	32.222388	74.733349	176.157491	82398.000000

Create Target Variable:

In [6]:
```
data['Credit_Score'] = data['Credit_Mix']
```

In [7]:
```
data.drop(columns=['Credit_Mix'], inplace=True)
```

In [8]:
```
data.columns
```

Out[8]:
```
Index(['ID', 'Customer_ID', 'Month', 'Name', 'Age', 'SSN', 'Occupation',
       'Annual_Income', 'Monthly_Inhand_Salary', 'Num_Bank_Accounts',
       'Num_Credit_Card', 'Interest_Rate', 'Num_of_Loan', 'Type_of_Loan',
       'Delay_from_due_date', 'Num_of_Delayed_Payment', 'Changed_Credit_Limit',
       'Num_Credit_Inquiries', 'Outstanding_Debt', 'Credit_Utilization_Ratio',
       'Credit_History_Age', 'Payment_of_Min_Amount', 'Total_EMI_per_month',
       'Amount_invested_monthly', 'Payment_Behaviour', 'Monthly_Balance',
       'Credit_Score'],
      dtype='object')
```

Cleaning Data:

In [9]:
```
data.describe(include='object').T
```

Out[9]:

	count	unique	top	freq
ID	50000	50000	0x160a	1
Customer_ID	50000	12500	CUS_0xd40	4
Month	50000	4	September	12500
Name	44985	10139	Stevex	22
Age	50000	976	39	1493
SSN	50000	12501	#F%$D@*&8	2828
Occupation	50000	16	_______	3438
Annual_Income	50000	16121	109945.32	8
Num_of_Loan	50000	263	2	7173
Type_of_Loan	44296	6260	Not Specified	704
Num_of_Delayed_Payment	46502	443	19	2622
Changed_Credit_Limit	50000	3927	_	1059
Outstanding_Debt	50000	12685	1109.03	12
Credit_History_Age	45530	399	20 Years and 1 Months	254
Payment_of_Min_Amount	50000	3	Yes	26158
Amount_invested_monthly	47729	45450	__10000__	2175
Payment_Behaviour	50000	7	Low_spent_Small_value_payments	12694
Monthly_Balance	49438	49433	__-333333333333333333333333333__	6
Credit_Score	50000	4	Standard	18379

In [10]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 27 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   ID                        50000 non-null  object 
 1   Customer_ID               50000 non-null  object 
 2   Month                     50000 non-null  object 
 3   Name                      44985 non-null  object 
 4   Age                       50000 non-null  object 
 5   SSN                       50000 non-null  object 
 6   Occupation                50000 non-null  object 
 7   Annual_Income             50000 non-null  object 
 8   Monthly_Inhand_Salary     42502 non-null  float64
 9   Num_Bank_Accounts         50000 non-null  int64  
 10  Num_Credit_Card           50000 non-null  int64  
 11  Interest_Rate             50000 non-null  int64  
 12  Num_of_Loan               50000 non-null  object 
 13  Type_of_Loan              44296 non-null  object 
 14  Delay_from_due_date       50000 non-null  int64  
 15  Num_of_Delayed_Payment    46502 non-null  object 
 16  Changed_Credit_Limit      50000 non-null  object 
 17  Num_Credit_Inquiries      48965 non-null  float64
 18  Outstanding_Debt          50000 non-null  object 
 19  Credit_Utilization_Ratio  50000 non-null  float64
 20  Credit_History_Age        45530 non-null  object 
 21  Payment_of_Min_Amount     50000 non-null  object 
 22  Total_EMI_per_month       50000 non-null  float64
 23  Amount_invested_monthly   47729 non-null  object 
 24  Payment_Behaviour         50000 non-null  object 
 25  Monthly_Balance           49438 non-null  object 
 26  Credit_Score              50000 non-null  object 
dtypes: float64(4), int64(4), object(19)
memory usage: 10.3+ MB
```

In [11]:
```
data.isnull().sum()
```

Out[11]:
```
ID                             0
Customer_ID                    0
Month                          0
Name                        5015
Age                            0
SSN                            0
Occupation                     0
Annual_Income                  0
Monthly_Inhand_Salary       7498
Num_Bank_Accounts              0
Num_Credit_Card                0
Interest_Rate                  0
Num_of_Loan                    0
Type_of_Loan                5704
Delay_from_due_date            0
Num_of_Delayed_Payment      3498
Changed_Credit_Limit           0
Num_Credit_Inquiries        1035
Outstanding_Debt               0
Credit_Utilization_Ratio       0
Credit_History_Age          4470
Payment_of_Min_Amount          0
Total_EMI_per_month            0
Amount_invested_monthly     2271
Payment_Behaviour              0
Monthly_Balance              562
Credit_Score                   0
dtype: int64
```

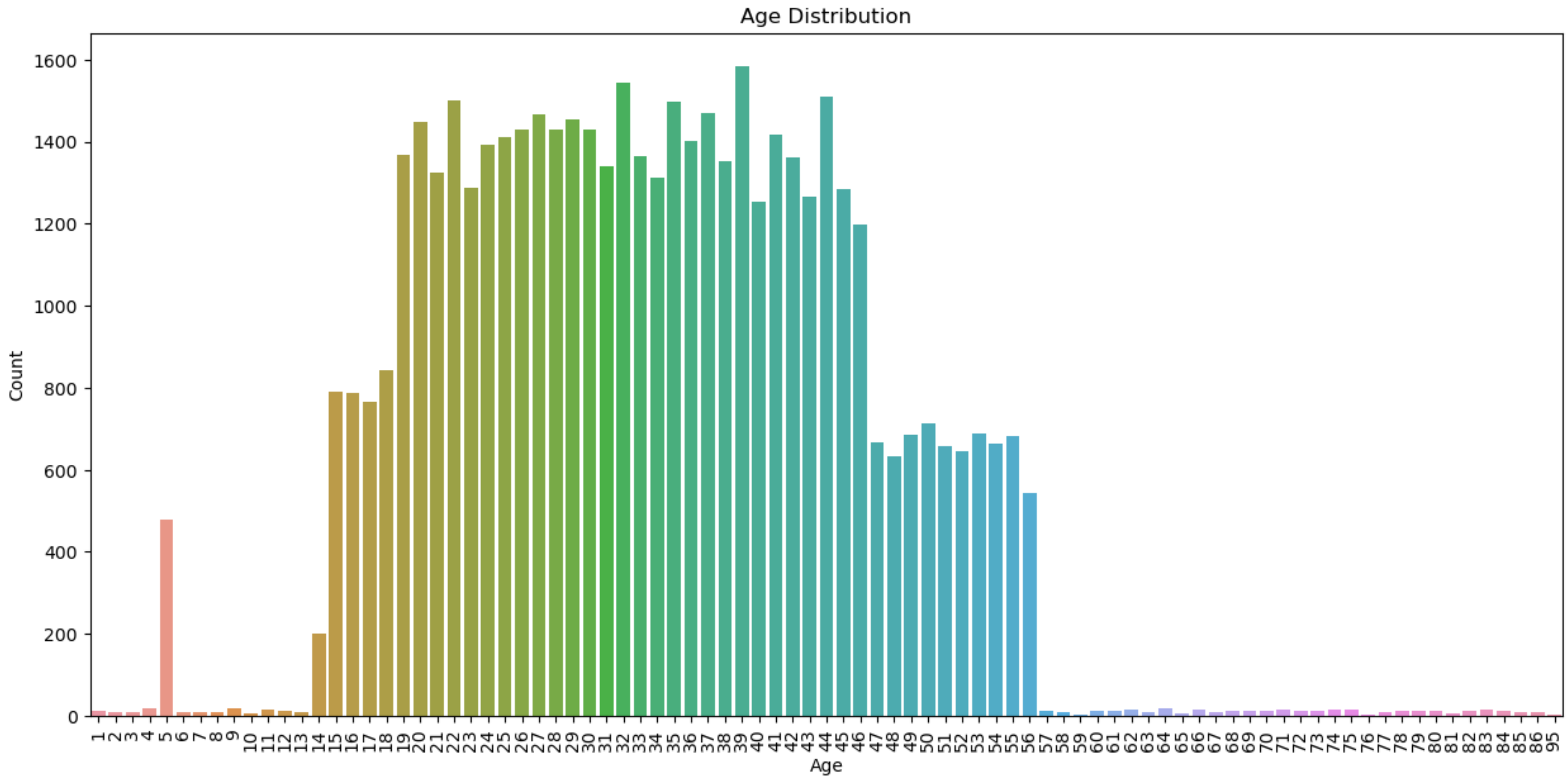
1. Age Variable:

In [12]:
```python
# 'Age'
def clean_age(age):
    try:
        return int(age)
    except ValueError:
        return None

data['Age'] = data['Age'].str.replace('_', '').str.replace('-', '')
data['Age'] = data['Age'].apply(clean_age)
```

In [13]:
```
def truncate_last_two_digits(age):
    if age > 99:
        return age // 100
    else:
        return age


data['Age'] = data['Age'].apply(truncate_last_two_digits)


data.Age
```

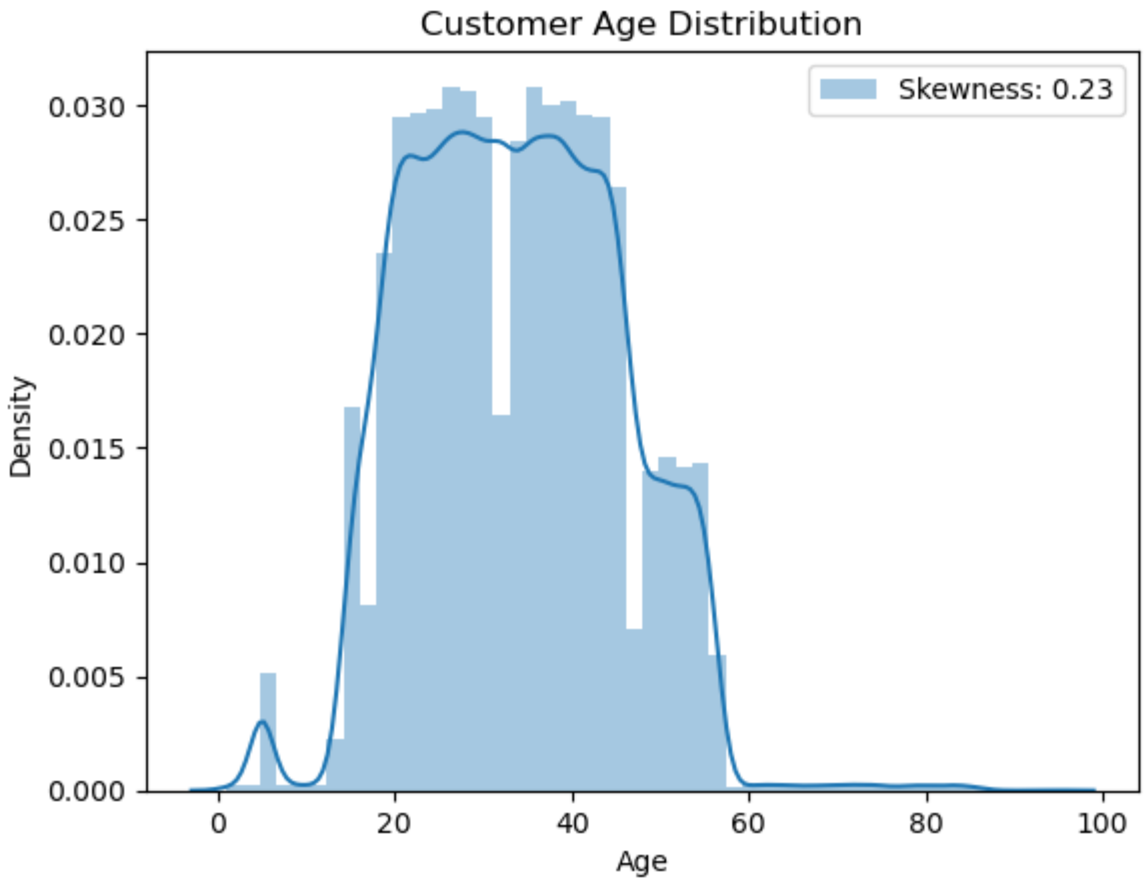
Out[13]:
```
0        23
1        24
2        24
3        24
4        28
         ..
49995    49
49996    25
49997    25
49998    25
49999    25
Name: Age, Length: 50000, dtype: int64
```

In [14]:
```
#visualize the age distribution
plt.figure(figsize=(15, 7))
sns.countplot(x='Age', data=data)
plt.title('Age Distribution')
plt.xlabel('Age')
plt.ylabel('Count')
plt.xticks(rotation=90)
plt.show()
```



In [15]:
```
sns.distplot(data['Age'], label = 'Skewness: %.2f'%(data['Age'].skew()))
plt.legend(loc = 'best')
plt.title('Customer Age Distribution')
```

Out[15]:
```
Text(0.5, 1.0, 'Customer Age Distribution')
```



2. Occupation Variable:

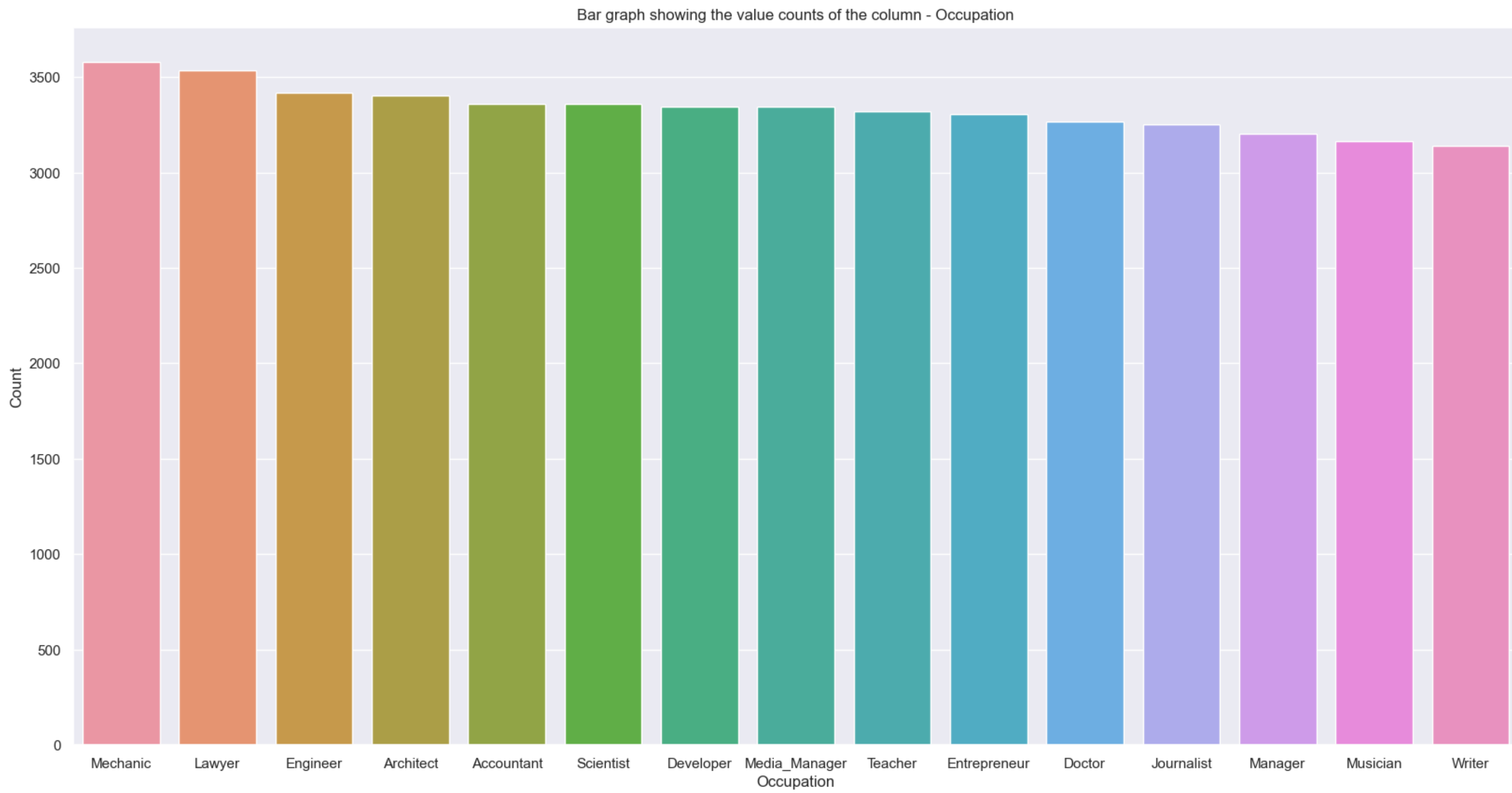
In [16]:
```
def fill_occupation_by_ssn(data):
    # Replace '_______' values in 'Occupation' column with NaN (empty) values
    data['Occupation'] = data['Occupation'].replace('_______', np.nan)

    # Find the most recurring 'Occupation' values for each SNN number
    most_common_occupation_by_ssn = data.groupby('SSN')['Occupation'].apply(lambda x: x.mode().iloc[0])

    # 'Populating '_______' values in 'Occupation' column
    for index, row in data.iterrows():
        if pd.isnull(row['Occupation']) and row['SSN'] in most_common_occupation_by_ssn:
            data.at[index, 'Occupation'] = most_common_occupation_by_ssn[row['SSN']]


fill_occupation_by_ssn(data)
```

In [17]:
```
occupation_count = data['Occupation'].value_counts()
occupation_count
```

Out[17]:
```
Occupation
Mechanic         3581
Lawyer           3536
Engineer         3417
Architect        3402
Accountant       3362
Scientist        3360
Developer        3345
Media_Manager    3344
Teacher          3320
Entrepreneur     3306
Doctor           3269
Journalist       3250
Manager          3202
Musician         3165
Writer           3141
Name: count, dtype: int64
```

In [18]:
```
# occupation_count, chart with the number of occupations in the Occupation column
sns.set(rc={'figure.figsize': (20, 10)})
sns.barplot(x=occupation_count.index, y=occupation_count.values)
plt.title('Bar graph showing the value counts of the column - Occupation')
plt.ylabel('Count', fontsize=12)
plt.xlabel('Occupation', fontsize=12)
```

Out[18]:
```
Text(0.5, 0, 'Occupation')
```



3. Annual_Incame Variable:

In [19]:
```
# To remove the tire at the end
def remove_trailing_dash(value):
    if isinstance(value, str) and value.endswith('_'):
        return value[:-1]  # Son karakteri (tireyi) kaldır
    else:
        return value


data['Annual_Income'] = data['Annual_Income'].apply(remove_trailing_dash)
```

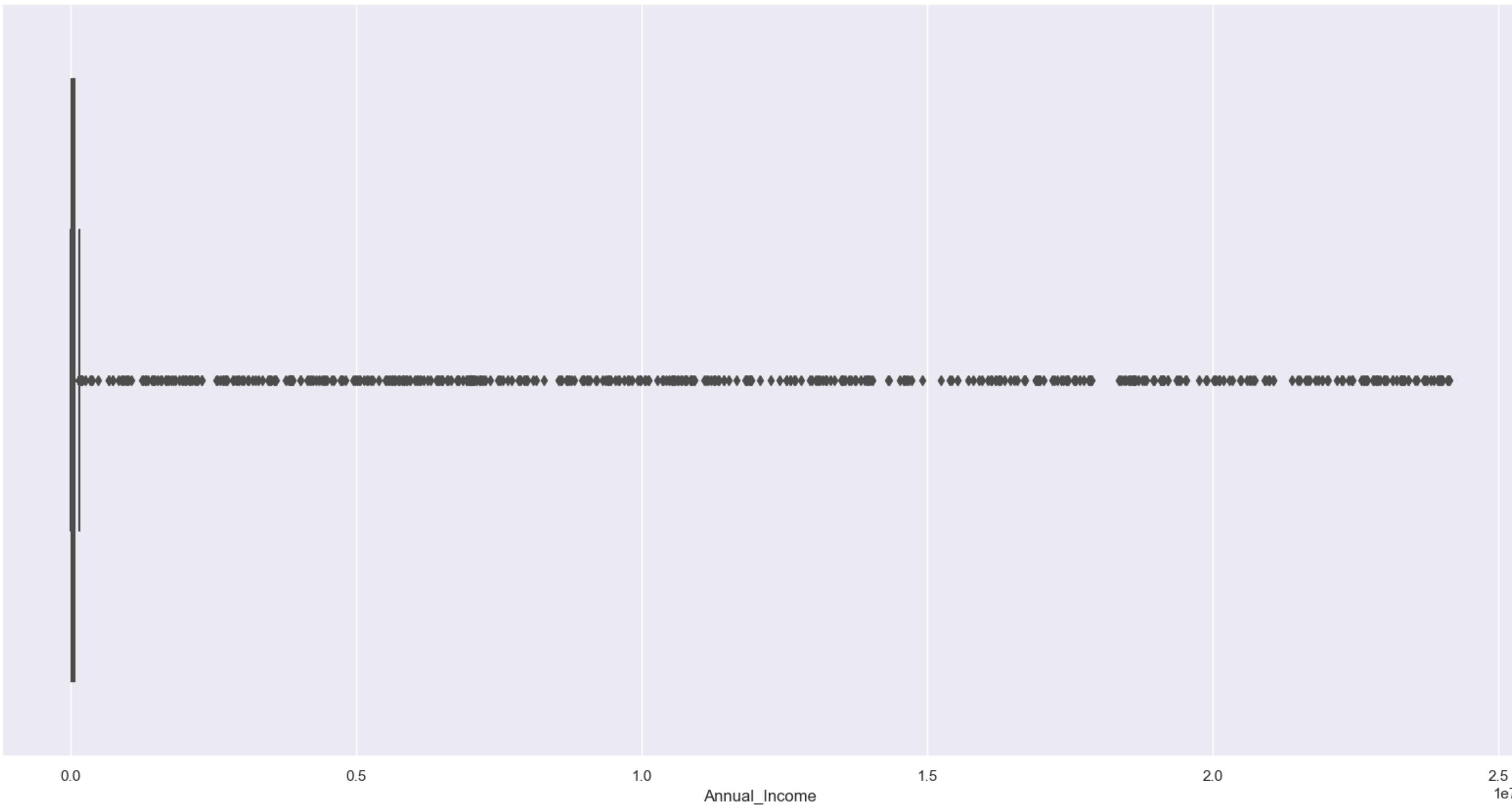
In [20]:
```
data['Annual_Income'] = data['Annual_Income'].astype(float)
```

In [21]:
```
data['Annual_Income'].unique()
```

Out[21]:
```
array([ 19114.12,  34847.84, 143162.64, ...,  37188.1 ,  20002.88,
        39628.99])
```

In [22]:
```
sns.boxplot(x = 'Annual_Income', data = data)
```

Out[22]:
```
<Axes: xlabel='Annual_Income'>
```

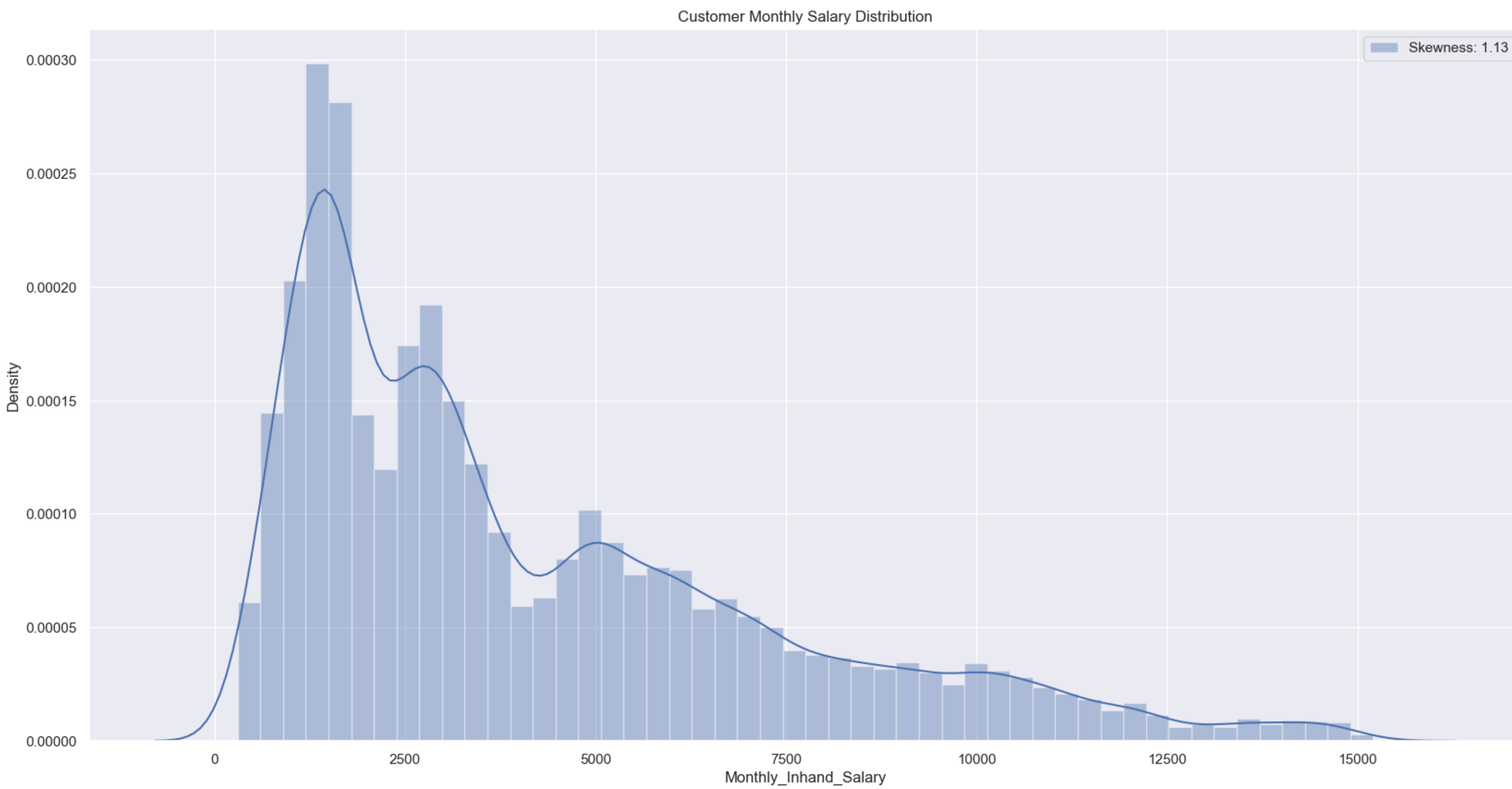


4. Monthly_Inhand_Salary Variable:

In [23]:
```
Customer_Mode_Salary = data.groupby('Customer_ID')['Monthly_Inhand_Salary'].transform(
    lambda x : x.mode().iloc[0] if not x.mode().empty else np.nan)
data['Monthly_Inhand_Salary'] = np.where(data['Monthly_Inhand_Salary'].isnull(),
                                         Customer_Mode_Salary, data['Monthly_Inhand_Salary'])

# Check for remaining NaN values and fill with the overall mean salary
if data['Monthly_Inhand_Salary'].isnull().any():
    overall_mean_salary = data['Monthly_Inhand_Salary'].mean()
    data['Monthly_Inhand_Salary'].fillna(overall_mean_salary, inplace=True)
```

In [24]:
```
sns.distplot(data['Monthly_Inhand_Salary'], label = 'Skewness: %.2f'%(data['Monthly_Inhand_Salary'].skew()))
plt.legend(loc = 'best')
plt.title('Customer Monthly Salary Distribution')
```

Out[24]:
```
Text(0.5, 1.0, 'Customer Monthly Salary Distribution')
```



In [25]:
```
data['Monthly_Inhand_Salary'].isnull().sum()
```

Out[25]:
```
0
```

5. Num_of_Loan Variable:

In [26]:
```
data['Num_of_Loan'].unique()
```

Out[26]:
```
array(['4', '1', '3', '1381', '-100', '0', '2', '7', '5', '6', '5_', '8',
       '1_', '2_', '6_', '4_', '9', '0_', '7_', '965', '3_', '428', '50',
       '8_', '256', '495', '9_', '1018', '548', '1470', '1176', '1021',
       '744', '238', '481_', '617', '1237', '602', '582', '1225', '717',
       '1316', '1146', '455', '1009', '660', '286', '505', '335', '1161',
       '765', '463', '864', '696', '95', '949', '720', '181', '1090',
       '1048', '42', '385', '814', '1019', '452', '434', '77', '639',
       '336', '249', '106', '498', '1073', '1259', '899', '769', '1292',
       '1266', '1365', '114', '810', '994', '992', '828', '1248', '543',
       '1318', '1416', '919', '1391', '58', '799', '1374', '263', '746',
       '1487', '731', '445', '1325', '1147', '808', '511', '198', '1445',
       '1140', '876', '1304', '569', '350', '1221', '608', '621', '1040',
       '1496', '570', '1063', '741', '230', '1428', '1254', '1361', '232',
       '1475', '653', '1448', '523', '1414', '426', '1308', '647', '1068',
       '954_', '1283', '909', '977', '1333', '700', '486', '1027', '170',
       '359', '1108', '1471', '628', '1284', '1489', '1446', '35', '483',
       '265', '928', '838', '1109', '1429', '1300', '547', '1385', '950',
       '1153', '539', '1240', '376', '393', '610', '725_', '1481', '1287',
       '325', '72_', '1024', '1072', '220', '478', '53', '368', '551',
       '1231', '1244', '734_', '140', '1329', '255', '1207', '257', '326',
       '273', '1187', '99', '1149', '1148', '464', '961', '519_', '142',
       '453', '641', '97', '441', '1114', '631', '295', '1444', '465',
       '960', '790', '624', '55', '370', '659', '361', '105', '1189',
       '813', '1095', '508', '879', '1373', '615', '997', '561', '161',
       '612', '488', '282', '324', '531', '339', '1057', '757', '343',
       '1196', '1041', '738', '1399', '414', '395', '1175', '663', '135',
       '910', '1447', '620', '354', '1263', '477', '544', '1134', '770',
       '204', '1369', '656', '534', '870', '418', '1292_', '1436', '1352',
       '594', '25', '1442', '939', '172', '1296_'], dtype=object)
```

In [27]:
```
# Function to remove "-" and "_" characters
def clean_num(num):
    num = num.strip("-_")
    if num == "100":
        return np.nan
    elif len(num) > 1:
        return num[0]
    else:
        return num


data["Num_of_Loan"] = data["Num_of_Loan"].apply(clean_num)


most_common_value = data["Num_of_Loan"].mode()[0]
data["Num_of_Loan"] = data["Num_of_Loan"].fillna(most_common_value)
```

In [28]:
```
data.Num_of_Loan.value_counts()
```

Out[28]:
```
Num_of_Loan
2    9507
3    7533
4    7392
0    5446
1    5400
6    3924
7    3698
5    3640
9    1855
8    1605
Name: count, dtype: int64
```

6. Type_of_Loan Variable:

In [29]:
```
data['Type_of_Loan'].fillna('Unknown', inplace=True)
```

In [30]:
```
loan_type_groups = data.groupby('Type_of_Loan').size()
print(loan_type_groups)
```

```
Type_of_Loan
Auto Loan
576
Auto Loan, Auto Loan, Auto Loan, Auto Loan, Credit-Builder Loan, Credit-Builder Loan, Mortgage Loan, and Personal Lo
an                          4
Auto Loan, Auto Loan, Auto Loan, Auto Loan, Student Loan, and Student Loan
4
Auto Loan, Auto Loan, Auto Loan, Credit-Builder Loan, Payday Loan, Not Specified, Payday Loan, Student Loan, and Deb
t Consolidation Loan        4
Auto Loan, Auto Loan, Auto Loan, Not Specified, Debt Consolidation Loan, and Credit-Builder Loan
4

...
Student Loan, and Not Specified
80
Student Loan, and Payday Loan
128
Student Loan, and Personal Loan
88
Student Loan, and Student Loan
108
Unknown
5704
Length: 6261, dtype: int64
```

In [31]:
```python
data.Type_of_Loan.value_counts()
```

Out[31]:
```
Type_of_Loan
Unknown
5704
Not Specified
704
Credit-Builder Loan
640
Personal Loan
636
Debt Consolidation Loan
632

...
Not Specified, Mortgage Loan, Auto Loan, and Payday Loan
4
Payday Loan, Mortgage Loan, Debt Consolidation Loan, and Student Loan
4
Debt Consolidation Loan, Auto Loan, Personal Loan, Debt Consolidation Loan, Student Loan, and Credit-Builder Loan
4
Student Loan, Auto Loan, Student Loan, Credit-Builder Loan, Home Equity Loan, Debt Consolidation Loan, and Debt Cons
olidation Loan      4
Personal Loan, Auto Loan, Mortgage Loan, Student Loan, and Student Loan
4
Name: count, Length: 6261, dtype: int64
```

7. Num_of_Delayed_Payment Variable:

In [32]:
```
data['Num_of_Delayed_Payment'] = data['Num_of_Delayed_Payment'].fillna('0')
```

In [33]:
```
#function to remove special character
def remove_special_characters(value):
    if isinstance(value, str):

        value = value.strip('_').strip('-')
    return value
```

In [34]:
```
data['Num_of_Delayed_Payment'] = data['Num_of_Delayed_Payment'].apply(
    remove_special_characters)
```

8. Changed_Credit_Limit Variable:

In [35]:
```
#To fills any remaining NaN values in the 'Changed_Credit_Limit'

data['Changed_Credit_Limit'] = data['Changed_Credit_Limit'].replace('-', np.nan)

data['Changed_Credit_Limit'] = pd.to_numeric(data['Changed_Credit_Limit'], errors='coerce')

mean_value = data['Changed_Credit_Limit'].mean()


data['Changed_Credit_Limit'].fillna(mean_value, inplace=True)
```

9. Num_Credit_Inquiries Variable:

In [36]:
```
data['Num_Credit_Inquiries'].unique()
```

```
Out[36]: array([2.022e+03, 4.000e+00, 5.000e+00, 3.000e+00, 7.000e+00, 1.552e+03,
       9.000e+00, 8.000e+00,       nan, 1.000e+00, 1.000e+01, 1.100e+01,
       1.700e+01, 1.905e+03, 1.300e+01, 2.000e+00, 6.000e+00, 0.000e+00,
       1.200e+01, 1.500e+01, 1.400e+01, 3.680e+02, 1.836e+03, 1.426e+03,
       2.019e+03, 1.189e+03, 1.600e+01, 1.293e+03, 1.469e+03, 1.785e+03,
       2.050e+02, 2.366e+03, 1.619e+03, 7.760e+02, 2.332e+03, 1.853e+03,
       7.960e+02, 2.210e+03, 2.173e+03, 1.094e+03, 1.868e+03, 1.813e+03,
       5.360e+02, 1.319e+03, 2.326e+03, 2.470e+03, 1.856e+03, 9.700e+01,
       2.240e+03, 3.660e+02, 1.010e+03, 1.800e+02, 1.730e+02, 7.020e+02,
       2.441e+03, 1.058e+03, 1.135e+03, 2.456e+03, 8.260e+02, 9.930e+02,
       2.592e+03, 1.292e+03, 2.371e+03, 7.680e+02, 9.340e+02, 9.840e+02,
       1.938e+03, 1.824e+03, 1.927e+03, 1.350e+03, 1.589e+03, 1.724e+03,
       4.180e+02, 5.060e+02, 9.460e+02, 1.252e+03, 1.889e+03, 8.180e+02,
       1.680e+03, 1.003e+03, 1.900e+03, 1.155e+03, 2.124e+03, 1.508e+03,
       1.571e+03, 1.175e+03, 1.784e+03, 6.370e+02, 1.512e+03, 4.610e+02,
       1.436e+03, 1.457e+03, 2.459e+03, 2.292e+03, 1.498e+03, 7.000e+01,
       6.740e+02, 2.720e+02, 9.500e+01, 8.090e+02, 1.906e+03, 1.481e+03,
       1.360e+02, 7.290e+02, 3.510e+02, 1.551e+03, 2.282e+03, 1.879e+03,
       1.984e+03, 1.452e+03, 5.450e+02, 2.340e+03, 8.240e+02, 1.063e+03,
       1.332e+03, 7.450e+02, 1.472e+03, 2.038e+03, 5.930e+02, 2.021e+03,
       1.801e+03, 2.225e+03, 1.400e+03, 4.510e+02, 1.228e+03, 4.980e+02,
       2.313e+03, 6.750e+02, 1.722e+03, 1.030e+02, 1.669e+03, 1.798e+03,
       6.870e+02, 2.830e+02, 2.450e+03, 3.550e+02, 8.700e+02, 3.410e+02,
       1.613e+03, 2.065e+03, 1.489e+03, 1.431e+03, 1.641e+03, 1.621e+03,
       3.530e+02, 1.065e+03, 1.420e+02, 1.421e+03, 8.800e+02, 2.515e+03,
       8.030e+02, 2.077e+03, 7.690e+02, 7.720e+02, 1.838e+03, 1.502e+03,
       4.630e+02, 4.080e+02, 1.993e+03, 2.557e+03, 1.996e+03, 4.590e+02,
       1.842e+03, 1.883e+03, 2.312e+03, 3.950e+02, 1.092e+03, 7.980e+02,
       2.338e+03, 6.690e+02, 1.349e+03, 2.051e+03, 1.632e+03, 4.890e+02,
       7.950e+02, 1.850e+02, 1.888e+03, 1.799e+03, 1.309e+03, 1.380e+02,
       7.150e+02, 2.593e+03, 1.737e+03, 1.999e+03, 2.093e+03, 3.270e+02,
       3.620e+02, 4.000e+02, 2.285e+03, 2.415e+03, 1.380e+03, 1.973e+03,
       3.880e+02, 6.210e+02, 9.860e+02, 2.179e+03, 8.810e+02, 2.344e+03,
       2.007e+03, 1.387e+03, 2.142e+03, 1.495e+03, 1.169e+03, 6.550e+02,
       2.575e+03, 1.232e+03, 2.041e+03, 1.510e+02, 2.034e+03, 2.558e+03,
       2.328e+03, 2.047e+03, 2.503e+03, 1.113e+03, 1.816e+03, 7.630e+02,
       1.416e+03, 2.567e+03, 1.448e+03, 7.090e+02, 5.690e+02, 8.630e+02,
       1.255e+03, 2.281e+03, 8.170e+02, 9.100e+02, 1.405e+03, 9.080e+02,
       2.360e+03, 2.497e+03, 1.061e+03, 2.190e+03, 2.243e+03, 1.499e+03,
       2.401e+03, 1.468e+03, 1.278e+03, 2.226e+03, 1.174e+03, 5.870e+02,
       1.964e+03, 1.670e+02, 1.102e+03, 1.345e+03, 1.518e+03, 6.170e+02,
       9.920e+02, 1.264e+03, 6.670e+02, 3.630e+02, 2.576e+03, 9.300e+01,
       2.880e+02, 2.060e+03, 1.266e+03, 2.387e+03, 1.108e+03, 1.219e+03,
       1.644e+03, 1.258e+03, 2.536e+03, 1.843e+03, 2.440e+03, 7.390e+02,
```

```
1.340e+03, 1.476e+03, 1.707e+03, 1.282e+03, 6.200e+02, 1.160e+02,
1.528e+03, 8.000e+01, 1.623e+03, 1.749e+03, 1.691e+03, 1.752e+03,
1.787e+03, 2.069e+03, 1.520e+02, 2.172e+03, 1.694e+03, 8.010e+02,
1.602e+03, 1.442e+03, 1.460e+03, 1.657e+03, 1.974e+03, 1.101e+03,
7.600e+01, 1.990e+03, 1.808e+03, 9.510e+02, 1.514e+03, 1.124e+03,
3.700e+02, 1.471e+03, 3.070e+02, 1.326e+03, 1.672e+03, 5.950e+02,
9.040e+02, 2.574e+03, 2.790e+02, 2.335e+03, 8.480e+02, 1.120e+02,
5.640e+02, 2.373e+03, 1.267e+03, 1.701e+03, 9.970e+02, 9.050e+02,
2.330e+03, 2.232e+03, 3.010e+02, 2.540e+03, 8.280e+02, 2.760e+02,
9.940e+02, 8.350e+02, 1.565e+03, 1.634e+03, 9.220e+02, 5.590e+02,
1.298e+03, 1.415e+03, 1.676e+03, 6.490e+02, 8.580e+02, 2.600e+01,
2.070e+03, 1.708e+03, 2.055e+03, 2.586e+03, 2.460e+02, 1.926e+03,
2.309e+03, 2.640e+02, 2.700e+02, 8.680e+02, 1.194e+03, 2.082e+03,
1.369e+03, 2.434e+03, 5.820e+02, 1.677e+03, 2.454e+03, 1.519e+03,
1.418e+03, 2.561e+03, 1.865e+03, 2.479e+03, 5.440e+02, 1.659e+03,
2.363e+03, 1.620e+02, 1.474e+03, 1.898e+03, 9.760e+02, 3.670e+02,
6.790e+02, 9.110e+02, 2.166e+03, 1.400e+02, 1.840e+02, 1.899e+03,
1.402e+03, 1.256e+03, 6.950e+02, 9.200e+01, 1.191e+03, 1.834e+03,
1.223e+03, 2.028e+03, 1.764e+03, 7.990e+02, 1.855e+03, 1.144e+03,
1.492e+03, 1.245e+03, 1.873e+03, 9.230e+02, 1.334e+03, 2.236e+03,
3.860e+02, 4.200e+01, 6.400e+02, 1.064e+03, 2.487e+03, 3.920e+02,
5.980e+02, 1.190e+03, 1.635e+03, 9.720e+02, 1.986e+03, 8.780e+02,
1.982e+03, 1.555e+03, 1.165e+03, 2.610e+02, 1.466e+03, 5.080e+02,
2.435e+03, 1.504e+03, 2.397e+03, 3.600e+02, 2.310e+02, 1.646e+03,
5.330e+02, 1.397e+03, 7.700e+01, 1.114e+03, 2.583e+03, 2.280e+03,
2.256e+03, 1.615e+03, 5.810e+02, 2.198e+03, 1.963e+03, 1.160e+03,
2.262e+03, 2.271e+03, 2.072e+03, 2.488e+03, 5.310e+02, 1.454e+03,
8.540e+02, 1.863e+03, 1.710e+03, 2.209e+03, 2.342e+03, 1.645e+03,
1.314e+03, 4.160e+02, 1.650e+03, 3.500e+02, 2.450e+02, 6.610e+02,
1.432e+03, 9.410e+02, 1.661e+03, 1.643e+03, 3.850e+02, 2.215e+03,
4.400e+02, 1.604e+03, 7.480e+02, 2.358e+03, 3.900e+01, 1.909e+03,
2.395e+03, 1.395e+03, 2.030e+02, 1.807e+03, 8.080e+02, 2.390e+03,
1.867e+03, 5.650e+02, 3.220e+02, 7.310e+02, 1.823e+03, 1.322e+03,
1.678e+03, 9.630e+02, 2.541e+03, 1.847e+03, 1.962e+03, 2.341e+03,
1.970e+02, 1.702e+03, 9.600e+01, 2.058e+03, 1.420e+03, 7.770e+02,
2.553e+03, 1.134e+03, 2.376e+03, 1.611e+03, 2.350e+03, 2.534e+03,
2.405e+03, 5.750e+02, 2.466e+03, 9.880e+02, 2.266e+03, 2.013e+03,
1.429e+03, 3.200e+01, 2.372e+03, 2.346e+03, 1.049e+03, 1.200e+02,
1.059e+03, 3.190e+02, 1.034e+03, 2.138e+03, 1.550e+03, 4.100e+01,
2.233e+03, 1.663e+03, 1.968e+03, 1.852e+03, 4.710e+02, 8.880e+02,
1.383e+03, 2.183e+03, 8.550e+02, 5.840e+02, 2.545e+03, 1.829e+03,
1.860e+02, 1.373e+03, 8.950e+02, 8.380e+02, 7.210e+02, 2.555e+03,
6.730e+02, 8.370e+02, 1.822e+03, 2.167e+03, 3.060e+02, 1.447e+03,
1.740e+03, 2.431e+03, 5.860e+02, 8.290e+02, 9.430e+02, 2.485e+03,
```

```
       5.030e+02, 2.231e+03, 9.100e+01, 1.038e+03, 3.400e+02, 6.780e+02,
       1.460e+02, 1.069e+03, 1.154e+03, 1.248e+03, 1.630e+03, 1.902e+03,
       3.960e+02, 1.760e+03, 1.138e+03, 2.345e+03, 2.390e+02, 1.051e+03,
       9.060e+02, 2.356e+03, 2.570e+03, 8.960e+02, 2.010e+02, 2.080e+02,
       2.087e+03, 1.578e+03, 1.470e+02, 1.145e+03, 9.170e+02, 2.403e+03,
       4.760e+02, 6.160e+02, 1.446e+03, 2.500e+03, 1.246e+03, 9.350e+02,
       1.097e+03, 1.437e+03, 1.176e+03, 1.218e+03, 1.166e+03, 5.700e+02,
       1.212e+03, 4.100e+02, 1.640e+02, 1.187e+03, 2.130e+02, 5.380e+02,
       7.820e+02, 1.112e+03, 2.890e+02, 1.086e+03, 8.200e+01, 9.150e+02,
       1.576e+03, 1.977e+03, 1.831e+03, 2.168e+03, 2.223e+03, 1.089e+03,
       3.590e+02, 2.030e+03, 4.520e+02, 2.322e+03, 6.030e+02, 1.810e+03,
       2.348e+03, 1.161e+03, 2.276e+03, 2.465e+03, 1.190e+02, 2.254e+03,
       1.947e+03, 1.320e+03, 2.213e+03, 5.780e+02, 1.830e+02, 1.523e+03,
       3.230e+02, 7.230e+02, 1.253e+03, 2.194e+03, 9.580e+02, 1.494e+03,
       1.670e+03, 7.930e+02, 2.157e+03, 1.269e+03, 1.357e+03, 2.448e+03,
       8.770e+02, 6.470e+02, 5.910e+02, 3.050e+02, 1.313e+03, 1.783e+03,
       2.155e+03, 1.821e+03, 2.525e+03, 1.340e+02, 1.030e+03, 1.247e+03,
       1.200e+03, 2.150e+02, 1.461e+03, 1.076e+03, 1.698e+03, 2.490e+02,
       1.132e+03, 4.220e+02, 1.023e+03, 1.071e+03, 1.496e+03, 2.325e+03,
       1.839e+03, 9.700e+02, 2.247e+03, 1.958e+03, 2.154e+03, 1.022e+03,
       1.531e+03, 1.141e+03, 1.933e+03, 8.800e+01, 2.107e+03, 8.660e+02,
       1.162e+03, 3.030e+02, 1.500e+02, 1.728e+03, 1.007e+03, 1.462e+03,
       8.690e+02, 1.530e+02, 2.780e+02, 2.343e+03, 1.195e+03, 2.200e+02,
       1.261e+03, 1.467e+03, 1.549e+03, 1.150e+03, 2.499e+03, 1.696e+03,
       2.402e+03, 2.127e+03, 1.931e+03, 2.980e+02, 6.220e+02, 2.176e+03,
       1.346e+03, 1.490e+02, 5.130e+02, 1.932e+03, 7.750e+02, 2.184e+03,
       2.477e+03, 3.840e+02, 2.494e+03, 8.160e+02, 2.393e+03, 1.090e+02,
       2.565e+03, 1.716e+03, 2.620e+02, 1.339e+03, 3.580e+02, 7.120e+02,
       4.200e+02, 7.900e+01, 9.210e+02, 3.320e+02, 1.750e+03, 2.353e+03,
       2.311e+03, 1.845e+03, 1.747e+03, 4.920e+02, 1.083e+03, 9.800e+01,
       1.511e+03, 4.330e+02, 1.419e+03, 2.135e+03, 2.446e+03, 1.991e+03,
       1.320e+02, 1.756e+03, 1.825e+03, 2.263e+03, 1.055e+03, 2.123e+03,
       9.070e+02, 1.532e+03, 2.406e+03, 6.770e+02, 4.530e+02, 1.942e+03,
       9.900e+01, 2.158e+03, 1.302e+03, 2.588e+03, 1.563e+03, 4.440e+02,
       4.830e+02, 1.626e+03, 1.180e+02, 2.251e+03, 2.391e+03, 2.442e+03,
       2.484e+03, 1.139e+03, 7.860e+02, 2.191e+03, 2.210e+02, 1.560e+03,
       6.500e+01, 1.288e+03, 1.181e+03, 6.510e+02, 3.910e+02, 8.400e+02,
       3.520e+02])
```

In [37]:
```
data['Num_Credit_Inquiries'].fillna(0, inplace=True)
```

10. Credit_Score Variable:

In [38]:
```
credit_score_count = data['Credit_Score'].value_counts()
credit_score_count
```

Out[38]:
```
Credit_Score
Standard    18379
Good        12260
_            9805
Bad          9556
Name: count, dtype: int64
```

In [39]:
```
data['Credit_Score'] = data['Credit_Score'].replace('_', np.nan)

def fill_na_cat(data, val):

    for col in data.select_dtypes(include='object').columns:
        mode_by_customer = data.groupby('Customer_ID')[col].transform(lambda x: x.mode()[0] if not x.mode().empty els
        mode_global = data[col].mode()[0]
        data[col] = data[col].fillna(mode_by_customer.fillna(mode_global))
    return data

data = fill_na_cat(data=data, val="Credit_Score")
```

In [40]:
```
data['Credit_Score'].value_counts()
```

Out[40]:
```
Credit_Score
Standard    22980
Good        15168
Bad         11852
Name: count, dtype: int64
```

11. Amount_invested_monthly Variable:

In [41]:
```
data['Amount_invested_monthly'] = data['Amount_invested_monthly'].replace('__10000__', np.nan)

data['Amount_invested_monthly'] = data.groupby('Customer_ID')['Amount_invested_monthly'].transform(
        lambda x: x.mode()[0] if not x.mode().empty else np.nan)

# Handle remaining NaN values with overall median
data['Amount_invested_monthly'].fillna(data['Amount_invested_monthly'].median(), inplace=True)
```

In [42]:
```
data['Amount_invested_monthly'].isnull().sum()
```

Out[42]:
```
0
```

12. Monthly_Balance Variable:

In [43]:
```
data['Monthly_Balance'].value_counts()
```

Out[43]:
```
Monthly_Balance
__-333333333333333333333333333__    10
164.37052922710703                   4
450.01757588255015                   4
59.42351010686326                    4
462.3772406222941                    4
                                    ..
357.4181848043565                    1
112.61218130286652                   1
202.19153281875958                   1
107.16715965280741                   1
360.37968260123847                   1
Name: count, Length: 49433, dtype: int64
```

In [44]:
```
data['Monthly_Balance'] = data['Monthly_Balance'].astype(str)

data['Monthly_Balance'] = data['Monthly_Balance'].str.replace(r'[^0-9.-]+', '').str.replace('_', '').str.replace('-',
```

In [45]:
```
data['Monthly_Balance'] = data['Monthly_Balance'].astype(float)

mean_value = data['Monthly_Balance'].mean()

data['Monthly_Balance'].fillna(mean_value, inplace=True)
```

13. Payment_Behaviour Variable:

In [46]:
```
data['Payment_Behaviour'] = data['Payment_Behaviour'].replace('!@9#%8', np.nan)
```

In [47]:
```
data['Payment_Behaviour'].isna().sum()
```

Out[47]:
```
3800
```

In [48]:
```
# Group data by 'Payment Behaviour' column
grouped_data = data.groupby('Payment_Behaviour').size()

print(grouped_data)
```

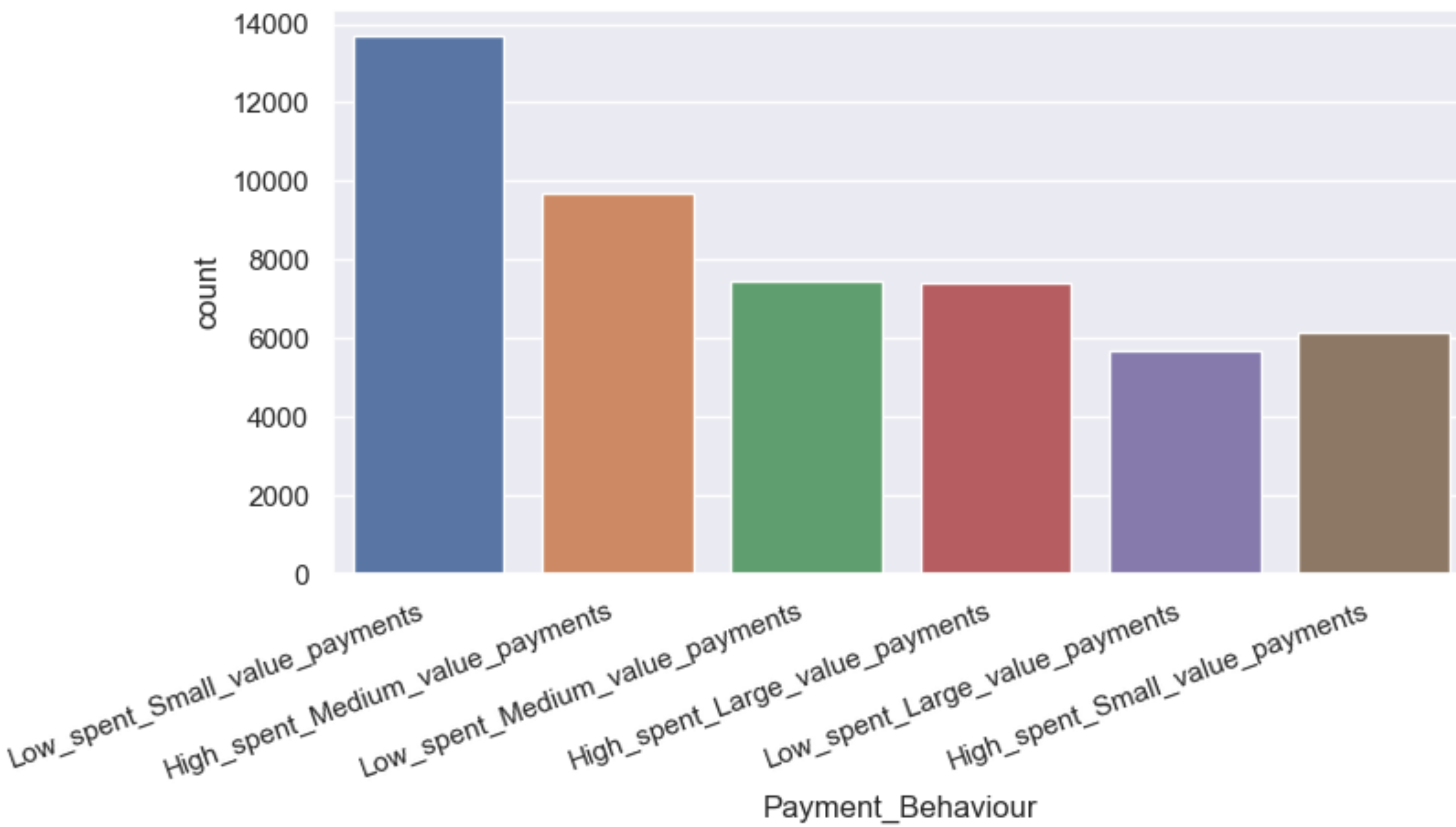
```
Payment_Behaviour
High_spent_Large_value_payments       6844
High_spent_Medium_value_payments      8922
High_spent_Small_value_payments       5651
Low_spent_Large_value_payments        5252
Low_spent_Medium_value_payments       6837
Low_spent_Small_value_payments       12694
dtype: int64
```

In [49]:
```
data['Payment_Behaviour'] = data['Payment_Behaviour'].fillna(method='ffill')
```

In [50]:
```
data['Payment_Behaviour'].isna().sum()
```

Out[50]:
```
0
```

In [51]:
```
plt.figure(figsize=(8, 4))
plot = sns.countplot(x='Payment_Behaviour', data=data)

plot.set_xticklabels(plot.get_xticklabels(), rotation=20, ha='right')

plt.show()
```

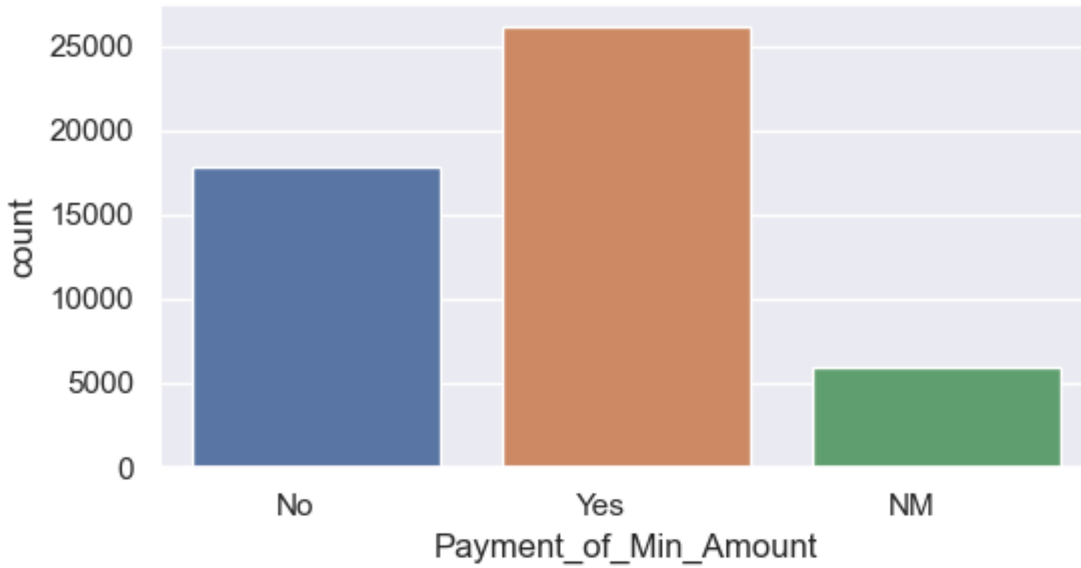


14. Payment_of_Min_Amount Variable:

In [52]:
```
min_amount_count = data['Payment_of_Min_Amount'].value_counts()
min_amount_count
```

Out[52]:
```
Payment_of_Min_Amount
Yes    26158
No     17849
NM      5993
Name: count, dtype: int64
```

In [53]:
```
plt.figure(figsize=(6, 3))
plot = sns.countplot(x='Payment_of_Min_Amount', data=data)

plot.set_xticklabels(plot.get_xticklabels(), ha='right')  # ha='right' ile etiketlerin hizalanması sağlanır

plt.show()
```



15. Interest_Rate Variable:

In [54]:
```
data['Interest_Rate'] = data['Interest_Rate'].astype(float)
```

In [55]:
```
data.Interest_Rate.value_counts()
```

Out[55]:
```
Interest_Rate
8.0       2503
5.0       2500
6.0       2368
12.0      2288
10.0      2259
          ...
1573.0       1
3279.0       1
1166.0       1
5613.0       1
4252.0       1
Name: count, Length: 945, dtype: int64
```

16. Credit_History_Age Variable:

In [56]:
```
# Covert Credit_History_Age to month

def parse_years_and_months_to_months(age):
    if isinstance(age, str):
        age_parts = age.split(' Years and ')
        years = int(age_parts[0]) if 'Years' in age else 0
        months_str = age_parts[1].split(' Months')[0] if 'Months' in age_parts[1] else '0'
        months = int(months_str)
        total_months = years * 12 + months
        return total_months
    else:
        return 0


data['Credit_History_Age_Months'] = data['Credit_History_Age'].apply(parse_years_and_months_to_months)
```

In [57]:
```
data.drop(columns=['Credit_History_Age'], inplace=True)
```

Drop unnecessary columns:

In [58]:
```
data.drop(['ID','Customer_ID', 'Month', 'Name','SSN', 'Type_of_Loan'], axis = 1, inplace = True)
```

In [59]:
```
data.head()
```

Out[59]:

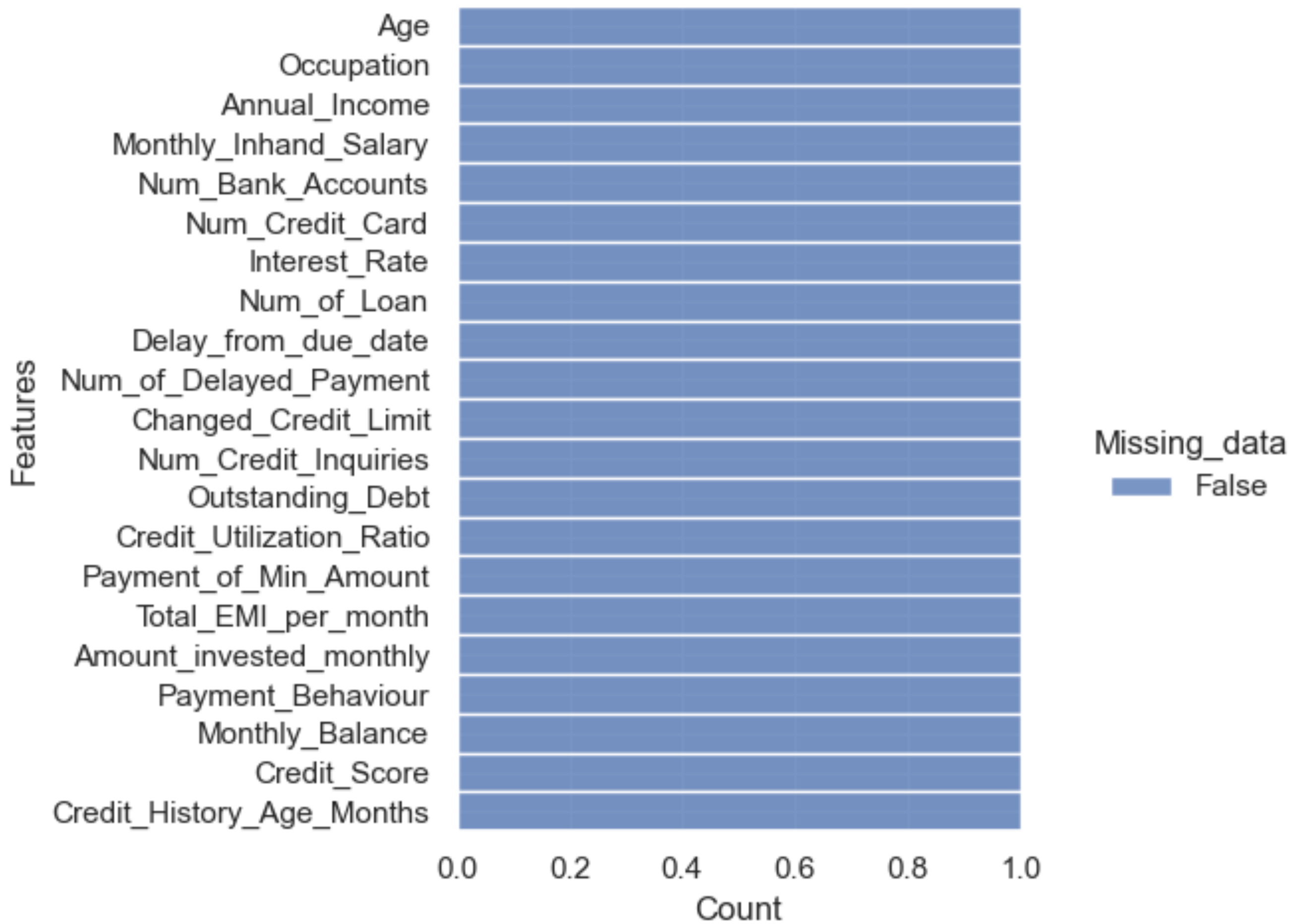
	Age	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credit_Card	Interest_Rate	Num_of_Loan	Delay_fr
0	23	Scientist	19114.12	1824.843333	3	4	3.0	4	
1	24	Scientist	19114.12	1824.843333	3	4	3.0	4	
2	24	Scientist	19114.12	1824.843333	3	4	3.0	4	
3	24	Scientist	19114.12	1824.843333	3	4	3.0	4	
4	28	Teacher	34847.84	3037.986667	2	4	6.0	1	

5 rows × 21 columns

In [60]:

```
#visualize the ratio of missing data in each feature of the dataset
def na_ratio_plot(data=data):

    sns.displot(data.isna().melt(value_name='Missing_data',var_name='Features')\
                ,y='Features',hue='Missing_data',multiple='fill',aspect=9/8)

print(data.isna().sum()[data.isna().sum()>0])
na_ratio_plot()
```

```
Series([], dtype: int64)
```



Convert numeric columns dtypes:

In [61]:
```
data.dtypes
```

Out[61]:

```
Age                           int64
Occupation                   object
Annual_Income               float64
Monthly_Inhand_Salary       float64
Num_Bank_Accounts             int64
Num_Credit_Card               int64
Interest_Rate               float64
Num_of_Loan                  object
Delay_from_due_date           int64
Num_of_Delayed_Payment       object
Changed_Credit_Limit        float64
Num_Credit_Inquiries        float64
Outstanding_Debt             object
Credit_Utilization_Ratio    float64
Payment_of_Min_Amount        object
Total_EMI_per_month         float64
Amount_invested_monthly      object
Payment_Behaviour            object
Monthly_Balance             float64
Credit_Score                 object
Credit_History_Age_Months     int64
dtype: object
```

In [62]:

```
#Modified conversion code
def safe_convert(x):
    try:
        return float(str(x).replace('_', ''))
    except ValueError:
        return np.nan

columns_to_convert = ['Num_of_Delayed_Payment', 'Outstanding_Debt', 'Amount_invested_monthly', 'Num_of_Loan']

for col in columns_to_convert:
    data[col] = data[col].apply(safe_convert)
```

In [63]:

```
data.dtypes
```

Out[63]:
```
Age                           int64
Occupation                   object
Annual_Income               float64
Monthly_Inhand_Salary       float64
Num_Bank_Accounts             int64
Num_Credit_Card               int64
Interest_Rate               float64
Num_of_Loan                 float64
Delay_from_due_date           int64
Num_of_Delayed_Payment      float64
Changed_Credit_Limit        float64
Num_Credit_Inquiries        float64
Outstanding_Debt            float64
Credit_Utilization_Ratio    float64
Payment_of_Min_Amount        object
Total_EMI_per_month         float64
Amount_invested_monthly     float64
Payment_Behaviour            object
Monthly_Balance             float64
Credit_Score                 object
Credit_History_Age_Months     int64
dtype: object
```

Handle Outliers:

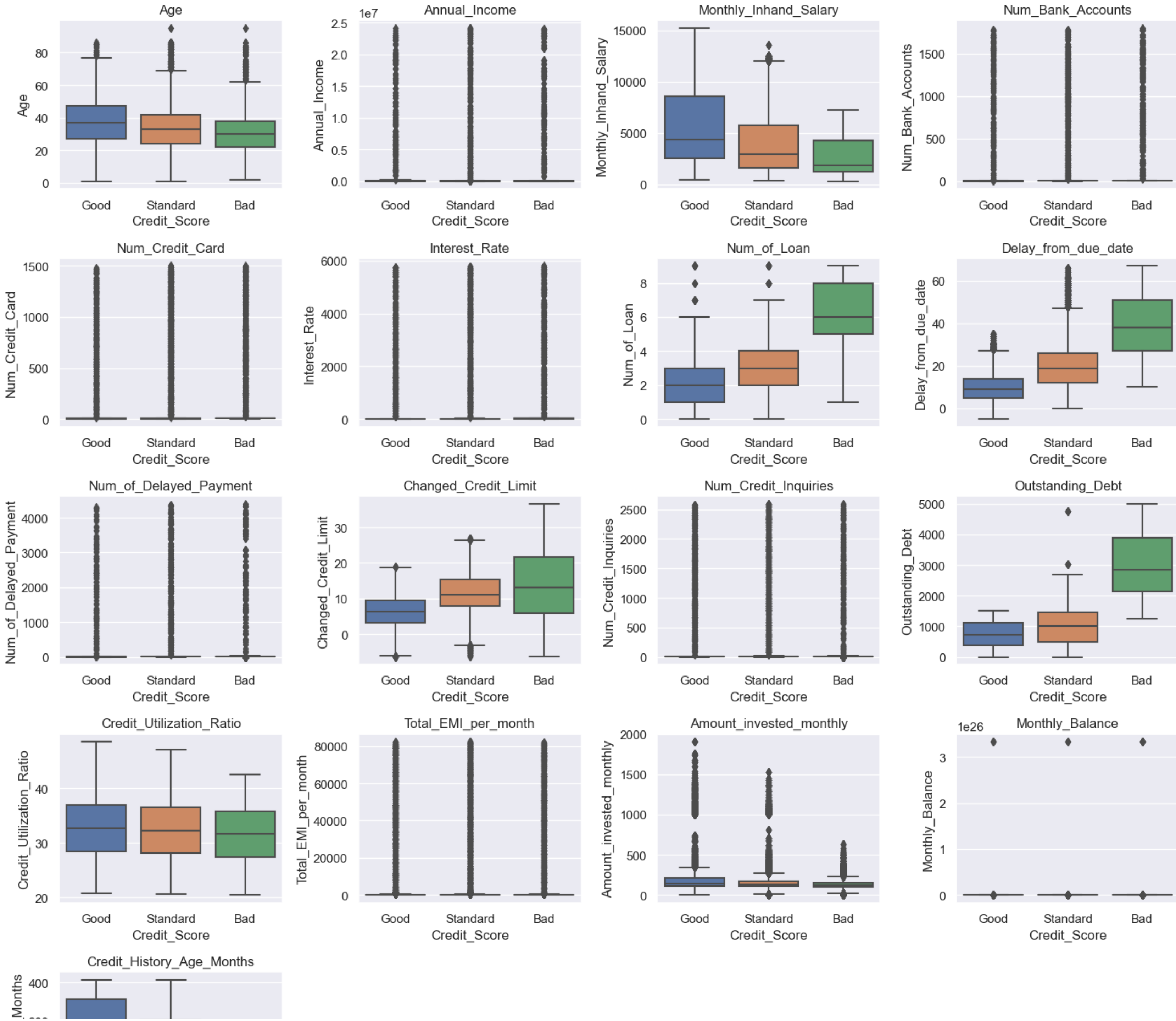
In [64]:
```
data.describe().T
```

Out[64]:

	count	mean	std	min	25%	50%	75%	max
Age	50000.0	3.373056e+01	1.156656e+01	1.000000	25.000000	33.000000	42.000000	9.500000e+01
Annual_Income	50000.0	1.663342e+05	1.351965e+06	7005.930000	19453.327500	37577.820000	72817.020000	2.413726e+07
Monthly_Inhand_Salary	50000.0	4.181292e+03	3.173165e+03	303.645417	1623.583854	3083.893333	5936.446667	1.520463e+04
Num_Bank_Accounts	50000.0	1.683826e+01	1.163968e+02	-1.000000	3.000000	6.000000	7.000000	1.798000e+03
Num_Credit_Card	50000.0	2.292148e+01	1.293148e+02	0.000000	4.000000	5.000000	7.000000	1.499000e+03
Interest_Rate	50000.0	6.877264e+01	4.516024e+02	1.000000	8.000000	13.000000	20.000000	5.799000e+03
Num_of_Loan	50000.0	3.474920e+00	2.417397e+00	0.000000	2.000000	3.000000	5.000000	9.000000e+00
Delay_from_due_date	50000.0	2.105264e+01	1.486040e+01	-5.000000	10.000000	18.000000	28.000000	6.700000e+01
Num_of_Delayed_Payment	50000.0	2.874788e+01	2.137640e+02	0.000000	8.000000	13.000000	18.000000	4.399000e+03
Changed_Credit_Limit	50000.0	1.037484e+01	6.708435e+00	-6.450000	5.440000	9.560000	14.600000	3.665000e+01
Num_Credit_Inquiries	50000.0	2.945754e+01	1.949817e+02	0.000000	4.000000	7.000000	10.000000	2.593000e+03
Outstanding_Debt	50000.0	1.426220e+03	1.155135e+03	0.230000	566.072500	1166.155000	1945.962500	4.998070e+03
Credit_Utilization_Ratio	50000.0	3.227958e+01	5.106238e+00	20.509652	28.061040	32.280390	36.468591	4.854066e+01
Total_EMI_per_month	50000.0	1.491304e+03	8.595648e+03	0.000000	32.222388	74.733349	176.157491	8.239800e+04
Amount_invested_monthly	50000.0	1.700092e+02	1.836149e+02	0.000000	107.752250	131.066905	175.799904	1.908124e+03
Monthly_Balance	50000.0	6.666667e+22	4.713621e+24	0.103402	269.850335	336.559413	470.335914	3.333333e+26
Credit_History_Age_Months	50000.0	2.271184e+02	9.966147e+01	10.000000	150.000000	225.000000	307.000000	4.080000e+02

In [65]:

```
#to see outlier in data
data_numeric_cols = [col for col in data.columns if data[col].dtype in ['int64', 'float64']]

plt.figure(figsize=(15,15))
for i, col in enumerate(data_numeric_cols):
    plt.subplot(5, 4, i+1)
    sns.boxplot(x='Credit_Score', y=col, data=data)
    plt.title(col)
plt.tight_layout()
plt.show()
```


Age
Annual_Income
1e7
Monthly_Inhand_Salary
Num_Bank_Accounts
Num_Credit_Card
Interest_Rate
Num_of_Loan
Delay_from_due_date
Num_of_Delayed_Payment
Changed_Credit_Limit
Num_Credit_Inquiries
Outstanding_Debt
Credit_Utilization_Ratio
Total_EMI_per_month
Amount_invested_monthly
Monthly_Balance
1e26
Credit_History_Age_Months
Good
Standard
Bad
Credit_Score

In [66]:

```
# outlier deletion

data_num = data.select_dtypes(include='number')

for column in data_num.columns:
    for i in data["Credit_Score"].unique():
        selected_i = data[data["Credit_Score"] == i]
        selected_column = selected_i[column]

        Q1 = selected_column.quantile(0.25)
        Q3 = selected_column.quantile(0.75)
        IQR = Q3 - Q1

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

        outliers = selected_column[(selected_column < lower_bound) | (selected_column > upper_bound)].index
        data.drop(index=outliers, inplace=True)
        print(column, i, outliers)
```

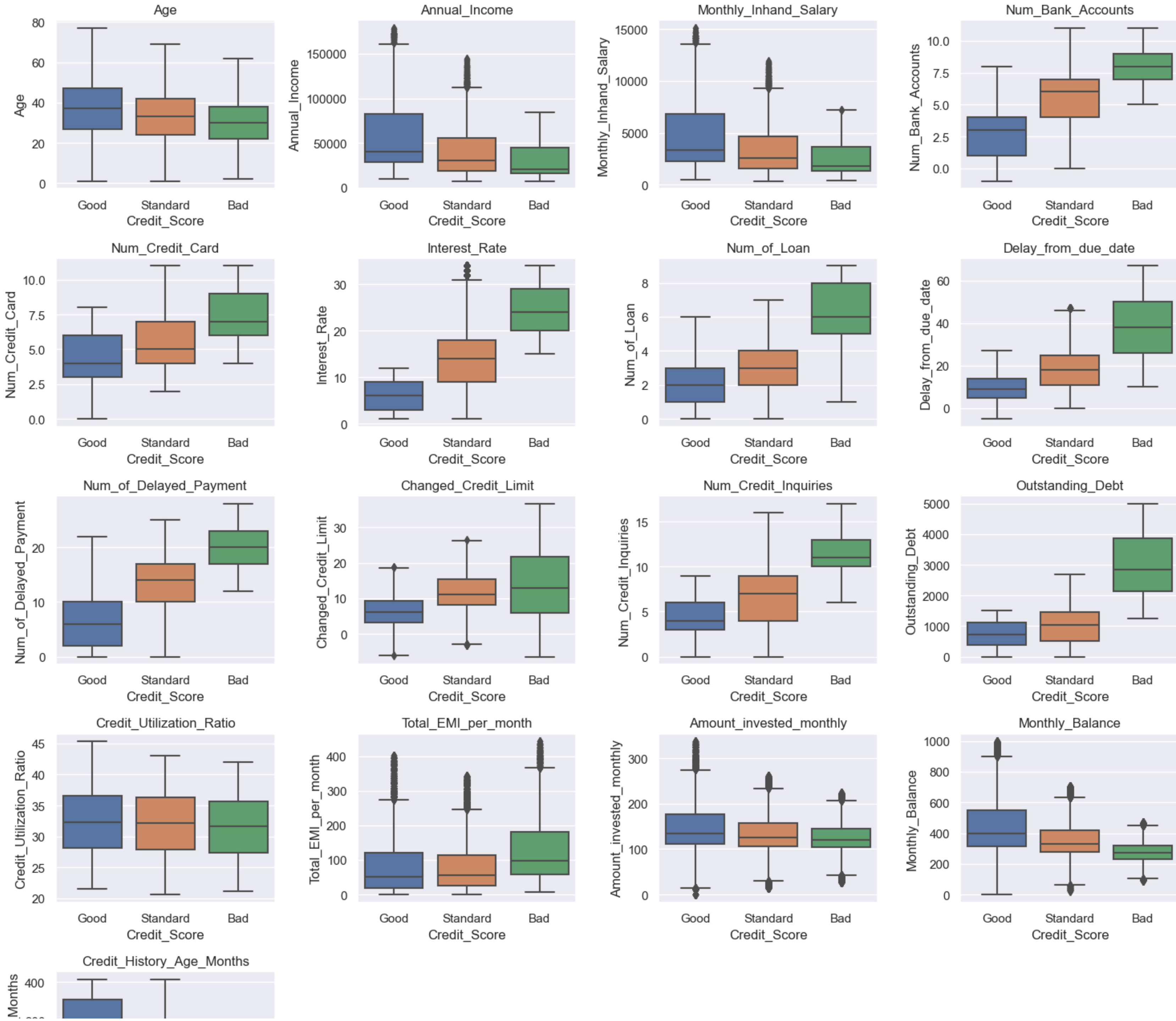
```
Age Good Index([  725,   821, 10254, 11244, 11862, 12489, 13527, 15608, 17988, 18723,
       22611, 22622, 25226, 28608, 33949, 35020, 36401, 39944, 45411, 46014],
      dtype='int64')
Age Standard Index([  334,   414,  1091,  1186,  1420,  2182,  2363,  2683,  3070,  4056,
        4957,  6147,  6167,  6210,  8763,  8864,  8884,  9352,  9756, 11550,
       11650, 13195, 13327, 14000, 14019, 14568, 14953, 15184, 17272, 17579,
       17586, 17623, 18504, 19093, 19689, 20054, 20465, 20663, 21522, 21608,
       21810, 22312, 22358, 22627, 23184, 25111, 25919, 26353, 26387, 26746,
       27570, 28337, 28513, 28892, 28955, 29006, 29060, 29188, 29667, 29764,
       30292, 30694, 31928, 31986, 32711, 32909, 33846, 33894, 34689, 35649,
       36659, 37525, 37549, 38110, 38182, 38532, 38890, 39819, 40356, 41119,
       41645, 42035, 42064, 43005, 43378, 43710, 43941, 44095, 45571, 46007,
       46092, 46464, 47238, 48331, 48930, 49269, 49431, 49935],
      dtype='int64')
Age Bad Index([ 1223,  1383,  2263,  3117,  3634,  4010,  4399,  4418,  6257,  7061,
        9668, 10592, 11944, 14121, 15321, 15328, 15383, 15539, 15718, 16004,
       17462, 17598, 19611, 20343, 21221, 23547, 23766, 27285, 28033, 28685,
       29582, 30511, 30541, 31527, 32343, 33284, 34102, 34262, 36069, 36778,
       36843, 37797, 39441, 39749, 40241, 40537, 40539, 40731, 40734, 41551,
       43319, 44149, 44777, 45206, 45739, 47047, 47278, 47625, 49329, 49575,
       49878],
      dtype='int64')
Annual_Income Good Index([   15,   326,   461,   724,   728,  1610,  1670,  2050,  2319,  2323,
       ...
       48360, 48370, 48528, 49080, 49278, 49287, 49355, 49494, 49654, 49857],
      dtype='int64', length=156)
Annual_Income Standard Index([  163,   456,  1452,  1516,  1615,  1616,  1643,  1660,  1808,  1809,
       ...
       47745, 48005, 48032, 48065, 48407, 48544, 48719, 48860, 49546, 49690],
      dtype='int64', length=269)
Annual_Income Bad Index([  295,  1261,  1852,  2015,  3182,  3425,  3458,  3818,  4009,  4076,
       ...
       44967, 45304, 45691, 46220, 46504, 47008, 48047, 48829, 49153, 49707],
      dtype='int64', length=119)
Monthly_Inhand_Salary Good Index([], dtype='int64')
Monthly_Inhand_Salary Standard Index([ 3092,  3093,  3094,  3095,  4692,  4693,  4694,  4695,  4984,  4985,
       ...
       47986, 47987, 48988, 48989, 48990, 48991, 49720, 49721, 49722, 49723],
      dtype='int64', length=129)
Monthly_Inhand_Salary Bad Index([], dtype='int64')
Num_Bank_Accounts Good Index([  308,   478,   516,   581,  1341,  1556,  2060,  2141,  2255,  2396,
       ...
       46552, 46883, 47133, 48030, 48158, 48762, 48910, 48917, 49199, 49981],
      dtype='int64', length=179)
```

```
Num_Bank_Accounts Standard Index([   61,    93,   745,   853,   943,  1133,  1207,  1431,  1510,  1580,
       ...
       48151, 48418, 48589, 48889, 48982, 49036, 49178, 49583, 49643, 49933],
      dtype='int64', length=280)
Num_Bank_Accounts Bad Index([  144,   229,   301,   563,   892,   963,  1376,  1434,  1463,  2066,
       ...
       45153, 46603, 47246, 47280, 47390, 48657, 48794, 48850, 49075, 49419],
      dtype='int64', length=164)
Num_Credit_Card Good Index([  251,   348,   447,   568,   731,   907,   914,  1108,  1110,  1201,
       ...
       48774, 48972, 49012, 49088, 49131, 49516, 49539, 49626, 49692, 49843],
      dtype='int64', length=347)
Num_Credit_Card Standard Index([  216,   390,   600,   601,   607,   840,   972,  1048,  1107,  1121,
       ...
       49120, 49179, 49204, 49268, 49344, 49432, 49522, 49782, 49855, 49885],
      dtype='int64', length=519)
Num_Credit_Card Bad Index([  116,   146,   181,   399,   468,   881,   962,  1068,  1131,  1154,
       ...
       47640, 47794, 47795, 48291, 48401, 48515, 48704, 48795, 48898, 49261],
      dtype='int64', length=265)
Interest_Rate Good Index([   46,    47,   652,  1021,  1095,  1102,  1408,  1540,  1749,  2457,
       ...
       48501, 48529, 49214, 49467, 49571, 49865, 49939, 49988, 49989, 49990],
      dtype='int64', length=266)
Interest_Rate Standard Index([   60,   260,   388,   389,   586,   734,   837,   927,  1028,  1134,
       ...
       48583, 48636, 48710, 49056, 49189, 49359, 49593, 49697, 49753, 49775],
      dtype='int64', length=450)
Interest_Rate Bad Index([  183,   378,   561,   877,  1074,  1117,  1606,  1787,  1800,  2448,
       ...
       46955, 47056, 47065, 47067, 47264, 47297, 47456, 47644, 49068, 49375],
      dtype='int64', length=194)
Num_of_Loan Good Index([ 1693,  4714,  6156,  6875,  8855, 11149, 12768, 14072, 15857, 22348,
       22432, 22804, 37107, 38342],
      dtype='int64')
Num_of_Loan Standard Index([ 5980, 12721, 12870, 17804, 18460, 24942, 26140, 26143, 26718, 29792,
       29793, 29794, 29795, 36079, 39224, 39496, 39497, 39498, 39499, 41392,
       49387],
      dtype='int64')
Num_of_Loan Bad Index([], dtype='int64')
Delay_from_due_date Good Index([  439,   566,   654,   655,   958,   959,  1008,  1009,  1011,  1284,
       ...
       49569, 49570, 49612, 49613, 49615, 49624, 49627, 49740, 49742, 49743],
      dtype='int64', length=393)
```

```
Delay_from_due_date Standard Index([  104,   105,   106,   107,   261,   262,   263,   588,   589,   590,
       ...
       48742, 48743, 48960, 48961, 48962, 48963, 49024, 49027, 49733, 49735],
      dtype='int64', length=568)
Delay_from_due_date Bad Index([], dtype='int64')
Num_of_Delayed_Payment Good Index([    8,   327,   493,   618,   773,   792,   800,   824,  2477,  2645,
       ...
       45436, 45533, 45567, 45695, 47327, 48062, 48361, 48975, 49277, 49635],
      dtype='int64', length=111)
Num_of_Delayed_Payment Standard Index([   68,   528,   740,   832,   995,  1180,  1508,  1666,  1880,  1881,
       ...
       47752, 47753, 47808, 48617, 48906, 49385, 49475, 49738, 49772, 49838],
      dtype='int64', length=170)
Num_of_Delayed_Payment Bad Index([   96,    99,   119,   187,   210,   346,   454,   475,   526,   761,
       ...
       49325, 49328, 49417, 49469, 49709, 49716, 49718, 49766, 49869, 49877],
      dtype='int64', length=824)
Changed_Credit_Limit Good Index([10665, 13427, 14354, 15497, 20845, 22833, 22834, 32741, 36855, 42248,
       42249, 43478, 46735],
      dtype='int64')
Changed_Credit_Limit Standard Index([ 4626,  5330,  5457,  5840,  9312,  9541, 11248, 12040, 13045, 13337,
       15928, 16142, 16642, 17382, 18926, 20283, 21214, 21326, 22443, 23187,
       23456, 23515, 23786, 24064, 24082, 24449, 25890, 25969, 26944, 27397,
       28878, 29723, 30600, 30752, 30998, 31935, 33997, 34253, 34342, 34362,
       35371, 35506, 36408, 38770, 39558, 41161, 42274, 42679, 43004, 43071,
       43943, 44901, 46788, 47209, 48282, 49169, 49458],
      dtype='int64')
Changed_Credit_Limit Bad Index([], dtype='int64')
Num_Credit_Inquiries Good Index([    0,    23,   359,   523,   569,   719,  1101,  2102,  2168,  2318,
       ...
       47495, 47583, 47817, 48763, 48772, 48775, 48854, 49323, 49955, 49979],
      dtype='int64', length=231)
Num_Credit_Inquiries Standard Index([  101,   323,   408,   409,   410,   411,   457,   540,   602,   888,
       ...
       48801, 48890, 48921, 48922, 49022, 49064, 49170, 49657, 49848, 49890],
      dtype='int64', length=455)
Num_Credit_Inquiries Bad Index([  207,   238,   265,   347,   417,   419,   858,   871,   879,   896,
       ...
       48692, 48744, 49041, 49073, 49331, 49442, 49534, 49574, 49602, 49868],
      dtype='int64', length=366)
Outstanding_Debt Good Index([], dtype='int64')
Outstanding_Debt Standard Index([44768, 44769, 44770, 44771], dtype='int64')
Outstanding_Debt Bad Index([], dtype='int64')
Credit_Utilization_Ratio Good Index([], dtype='int64')
```

```
Credit_Utilization_Ratio Standard Index([], dtype='int64')
Credit_Utilization_Ratio Bad Index([], dtype='int64')
Total_EMI_per_month Good Index([   24,    25,    26,    27,   280,   281,   282,   283,   609,   723,
       ...
       49616, 49617, 49618, 49619, 49756, 49757, 49758, 49759, 49799, 49991],
      dtype='int64', length=1186)
Total_EMI_per_month Standard Index([   72,    73,    74,    75,   161,   225,   332,   333,   335,   360,
       ...
       49815, 49839, 49845, 49852, 49930, 49943, 49960, 49961, 49962, 49963],
      dtype='int64', length=1671)
Total_EMI_per_month Bad Index([  117,   192,   193,   194,   195,   211,   228,   230,   231,   300,
       ...
       49033, 49034, 49035, 49071, 49125, 49443, 49712, 49713, 49714, 49715],
      dtype='int64', length=555)
Amount_invested_monthly Good Index([    9,    10,    11,   172,   173,   174,   175,   212,   213,   214,
       ...
       49693, 49694, 49695, 49832, 49833, 49834, 49835, 49936, 49937, 49938],
      dtype='int64', length=1179)
Amount_invested_monthly Standard Index([  272,   273,   274,   275,   412,   413,   415,   544,   545,   546,
       ...
       49698, 49699, 49800, 49801, 49802, 49803, 49880, 49881, 49882, 49883],
      dtype='int64', length=1594)
Amount_invested_monthly Bad Index([  164,   165,   166,   167,   264,   266,   267,   292,   293,   294,
       ...
       49324, 49326, 49327, 49416, 49418, 49572, 49573, 49764, 49765, 49767],
      dtype='int64', length=1668)
Monthly_Balance Good Index([  254,   460,   510,   767,  1188,  1191,  1269,  1272,  1273,  1274,
       ...
       48446, 48577, 48579, 48767, 48893, 49172, 49284, 49728, 49730, 49741],
      dtype='int64', length=484)
Monthly_Balance Standard Index([   65,   196,   198,   224,   227,   392,   395,   425,   556,   557,
       ...
       49642, 49658, 49659, 49810, 49811, 49830, 49836, 49853, 49944, 49946],
      dtype='int64', length=1164)
Monthly_Balance Bad Index([   97,   180,   182,   232,   234,   268,   269,   270,   271,   319,
       ...
       49087, 49309, 49310, 49330, 49470, 49556, 49565, 49566, 49589, 49925],
      dtype='int64', length=528)
Credit_History_Age_Months Good Index([], dtype='int64')
Credit_History_Age_Months Standard Index([], dtype='int64')
Credit_History_Age_Months Bad Index([], dtype='int64')
```

In [67]:
```
data_numeric_cols = [col for col in data.columns if data[col].dtype in ['int64', 'float64']]

plt.figure(figsize=(15,15))
for i, col in enumerate(data_numeric_cols):
    plt.subplot(5, 4, i+1)
    sns.boxplot(x='Credit_Score', y=col, data=data)
    plt.title(col)
plt.tight_layout()
plt.show()
```


Age
Annual_Income
Monthly_Inhand_Salary
Num_Bank_Accounts
Num_Credit_Card
Interest_Rate
Num_of_Loan
Delay_from_due_date
Num_of_Delayed_Payment
Changed_Credit_Limit
Num_Credit_Inquiries
Outstanding_Debt
Credit_Utilization_Ratio
Total_EMI_per_month
Amount_invested_monthly
Monthly_Balance
Credit_History_Age_Months
Good
Standard
Bad
Credit_Score

Visualize clean data:

In [68]:
```
data.head()
```

Out[68]:

	Age	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credit_Card	Interest_Rate	Num_of_Loan	Delay_fr
1	24	Scientist	19114.12	1824.843333	3	4	3.0	4.0	
2	24	Scientist	19114.12	1824.843333	3	4	3.0	4.0	
3	24	Scientist	19114.12	1824.843333	3	4	3.0	4.0	
4	28	Teacher	34847.84	3037.986667	2	4	6.0	1.0	
5	28	Teacher	34847.84	3037.986667	2	4	6.0	1.0	

5 rows × 21 columns

In [69]:
```
data.shape
```

Out[69]:
```
(33228, 21)
```

In [70]:
```
data.describe().T
```

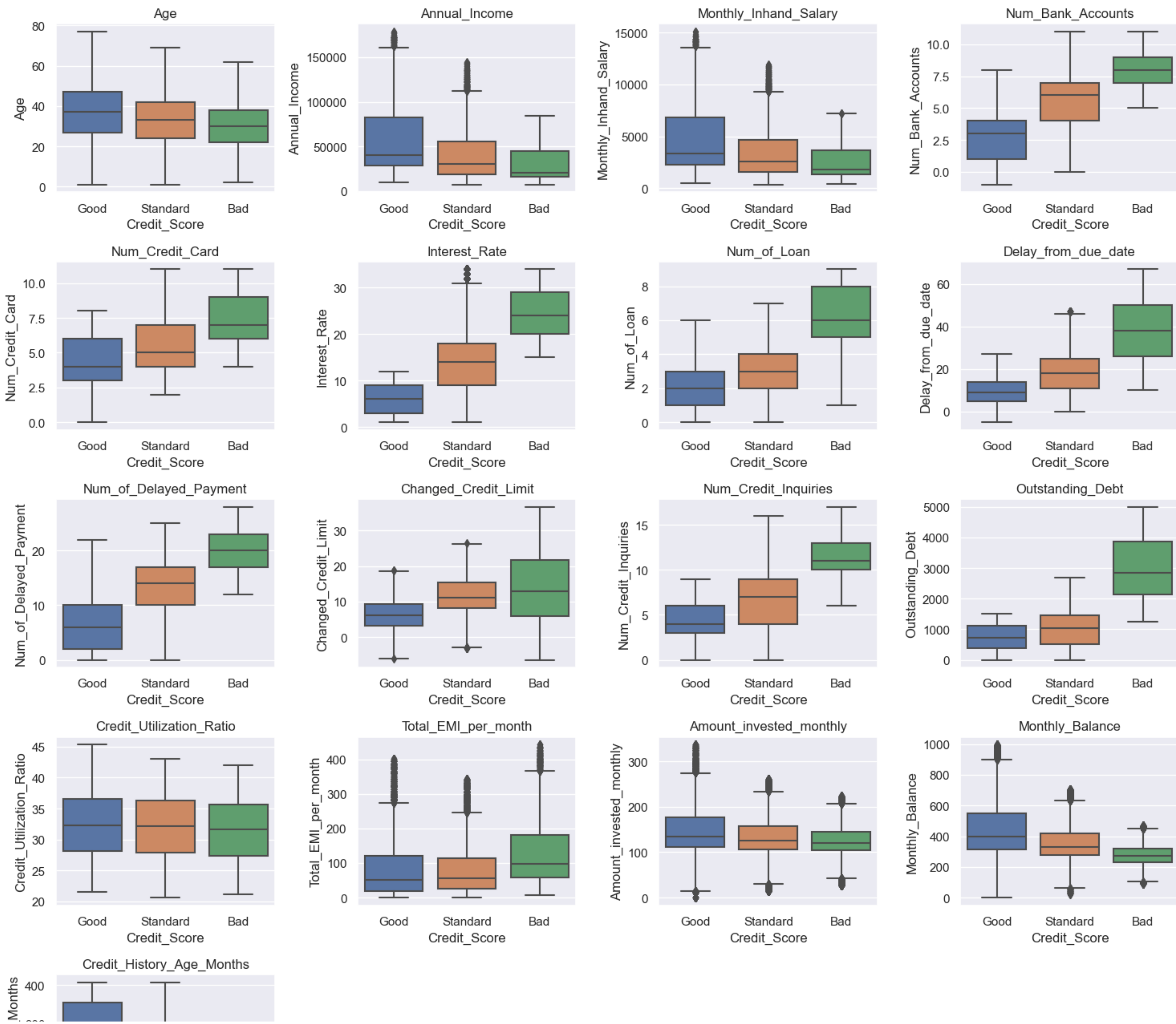
Out[70]:

	count	mean	std	min	25%	50%	75%	max
Age	33228.0	33.756681	11.315562	1.000000	25.000000	34.000000	42.000000	77.000000
Annual_Income	33228.0	42123.478364	30045.319105	7005.930000	18771.660000	33077.820000	59343.120000	178793.920000
Monthly_Inhand_Salary	33228.0	3504.179399	2504.804141	332.128333	1565.722083	2744.853333	4910.515000	15101.940000
Num_Bank_Accounts	33228.0	5.235344	2.572005	-1.000000	3.000000	5.000000	7.000000	11.000000
Num_Credit_Card	33228.0	5.445979	2.044646	0.000000	4.000000	5.000000	7.000000	11.000000
Interest_Rate	33228.0	14.128777	8.600439	1.000000	7.000000	12.000000	20.000000	34.000000
Num_of_Loan	33228.0	3.425364	2.353109	0.000000	2.000000	3.000000	5.000000	9.000000
Delay_from_due_date	33228.0	19.816751	14.048475	-5.000000	9.000000	17.000000	27.000000	67.000000
Num_of_Delayed_Payment	33228.0	12.480017	6.785080	0.000000	8.000000	13.000000	18.000000	28.000000
Changed_Credit_Limit	33228.0	10.259026	6.575846	-6.410000	5.400000	9.500000	14.280000	36.650000
Num_Credit_Inquiries	33228.0	7.030516	3.972674	0.000000	4.000000	7.000000	10.000000	17.000000
Outstanding_Debt	33228.0	1379.341590	1116.669434	0.230000	559.890000	1134.850000	1827.350000	4997.100000
Credit_Utilization_Ratio	33228.0	32.061074	5.003135	20.620017	27.873785	32.044800	36.247934	45.352930
Total_EMI_per_month	33228.0	91.233520	86.012595	0.000000	28.551433	63.535049	131.659526	442.766556
Amount_invested_monthly	33228.0	132.311827	54.087024	0.000000	107.241336	126.689165	159.751537	337.285627
Monthly_Balance	33228.0	368.580459	154.591605	1.084552	270.508368	329.405758	428.868731	996.659531
Credit_History_Age_Months	33228.0	230.353918	98.862679	10.000000	156.000000	228.000000	311.000000	408.000000

In [71]:
```
data.isnull().sum()
```

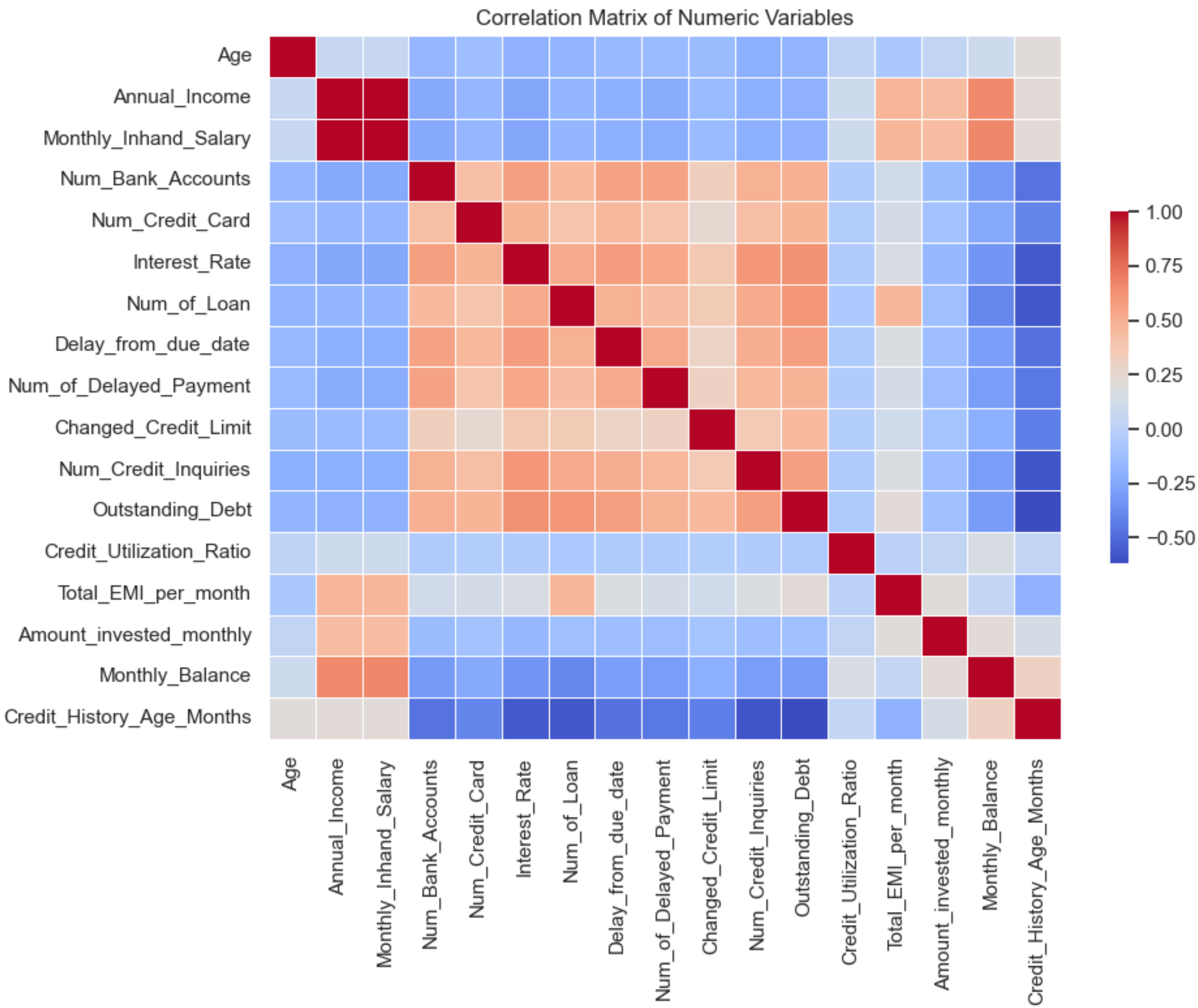
Out[71]:
```
Age                         0
Occupation                  0
Annual_Income               0
Monthly_Inhand_Salary       0
Num_Bank_Accounts           0
Num_Credit_Card             0
Interest_Rate               0
Num_of_Loan                 0
Delay_from_due_date         0
Num_of_Delayed_Payment      0
Changed_Credit_Limit        0
Num_Credit_Inquiries        0
Outstanding_Debt            0
Credit_Utilization_Ratio    0
Payment_of_Min_Amount       0
Total_EMI_per_month         0
Amount_invested_monthly     0
Payment_Behaviour           0
Monthly_Balance             0
Credit_Score                0
Credit_History_Age_Months   0
dtype: int64
```

In [72]:
```
#outliers
data_numeric_cols = [col for col in data.columns if data[col].dtype in ['int64', 'float64']]

plt.figure(figsize=(15,15))
for i, col in enumerate(data_numeric_cols):
    plt.subplot(5, 4, i+1)
    sns.boxplot(x='Credit_Score', y=col, data=data)
    plt.title(col)
plt.tight_layout()
plt.show()
```



Age
Annual_Income
Monthly_Inhand_Salary
Num_Bank_Accounts
Num_Credit_Card
Interest_Rate
Num_of_Loan
Delay_from_due_date
Num_of_Delayed_Payment
Changed_Credit_Limit
Num_Credit_Inquiries
Outstanding_Debt
Credit_Utilization_Ratio
Total_EMI_per_month
Amount_invested_monthly
Monthly_Balance
Credit_History_Age_Months
Good
Standard
Bad
Credit_Score

In [73]:
```
#Correlation Matrix
numeric_data = data.select_dtypes(include=['number'])

plt.figure(figsize=(10, 8))
sns.heatmap(numeric_data.corr(), annot=False, cmap="coolwarm", fmt=".2f", linewidths=.5, cbar_kws={"shrink": .5})
plt.xticks(rotation=90)
plt.yticks()
plt.title('Correlation Matrix of Numeric Variables')
plt.tight_layout()
plt.show()
```


Correlation Matrix of Numeric Variables
Age
Annual_Income
Monthly_Inhand_Salary
Num_Bank_Accounts
Num_Credit_Card
Interest_Rate
Num_of_Loan
Delay_from_due_date
Num_of_Delayed_Payment
Changed_Credit_Limit
Num_Credit_Inquiries
Outstanding_Debt
Credit_Utilization_Ratio
Total_EMI_per_month
Amount_invested_monthly
Monthly_Balance
Credit_History_Age_Months
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50

Encode Variable:

1. LabelEncoding for target variable:

In [74]:
```
from sklearn.preprocessing import LabelEncoder,OrdinalEncoder

data["Credit_Score"] = LabelEncoder().fit_transform(data["Credit_Score"])
data["Credit_Score"]
```

Out[74]:
```
1        1
2        1
3        1
4        1
5        1
        ..
49995    0
49996    1
49997    1
49998    1
49999    1
Name: Credit_Score, Length: 33228, dtype: int32
```

In [75]:
```
data["Credit_Score"].value_counts()
```

Out[75]:
```
Credit_Score
2    15531
1    10589
0     7108
Name: count, dtype: int64
```

In [76]:
```
data["Credit_Score"].value_counts()
```

Out[76]:
```
Credit_Score
2    15531
1    10589
0     7108
Name: count, dtype: int64
```

2. Encoding for categorical variable:

In [77]:
```
# select columns of type 'object'

data.select_dtypes(include=['object']).columns
```

Out[77]:
```
Index(['Occupation', 'Payment_of_Min_Amount', 'Payment_Behaviour'], dtype='object')
```

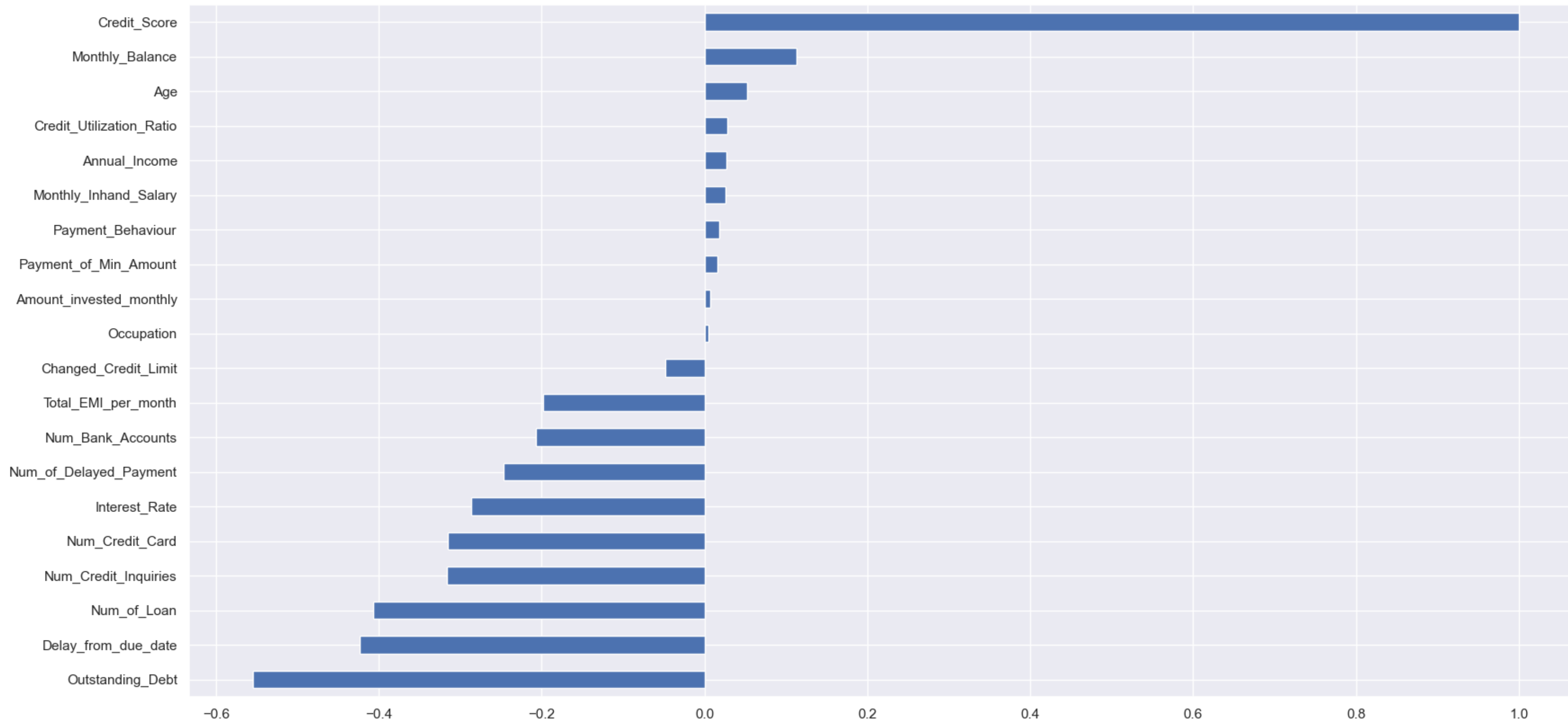
In [78]:
```
#Encode Occupation

label_encoder = LabelEncoder()
data['Occupation'] = label_encoder.fit_transform(data['Occupation'])
```

In [79]:
```
#Encode Payment_of_Min_Amount

label_encoder = LabelEncoder()
data['Payment_of_Min_Amount'] = label_encoder.fit_transform(data['Payment_of_Min_Amount'])
```

In [80]:
```
#Encode payment_behaviour

payment_behaviour_categories = ['Low_spent_Small_value_payments',
                                'Low_spent_Medium_value_payments',
                                'Low_spent_Large_value_payments',
                                'High_spent_Small_value_payments',
                                'High_spent_Medium_value_payments',
                                'High_spent_Large_value_payments']

payment_behaviour_encoder = OrdinalEncoder(categories=[payment_behaviour_categories])

data['Payment_Behaviour'] = payment_behaviour_encoder.fit_transform(data[['Payment_Behaviour']])
```

Correlation of target variable with features:

In [81]:
```
# Correlation of target variable with features after numerical transformation


numerical_data = data.select_dtypes(include=[np.number])
correlation_series = numerical_data.corr()['Credit_Score'][:-1].sort_values()
correlation_series.plot.barh();
```



In [82]:

```
data.head().T
```

Out[82]:

	1	2	3	4	5
Age	24.000000	24.000000	24.000000	28.000000	28.000000
Occupation	12.000000	12.000000	12.000000	13.000000	13.000000
Annual_Income	19114.120000	19114.120000	19114.120000	34847.840000	34847.840000
Monthly_Inhand_Salary	1824.843333	1824.843333	1824.843333	3037.986667	3037.986667
Num_Bank_Accounts	3.000000	3.000000	3.000000	2.000000	2.000000
Num_Credit_Card	4.000000	4.000000	4.000000	4.000000	4.000000
Interest_Rate	3.000000	3.000000	3.000000	6.000000	6.000000
Num_of_Loan	4.000000	4.000000	4.000000	1.000000	1.000000
Delay_from_due_date	3.000000	-1.000000	4.000000	3.000000	3.000000
Num_of_Delayed_Payment	9.000000	4.000000	5.000000	1.000000	3.000000
Changed_Credit_Limit	13.270000	12.270000	11.270000	5.420000	5.420000
Num_Credit_Inquiries	4.000000	4.000000	4.000000	5.000000	5.000000
Outstanding_Debt	809.980000	809.980000	809.980000	605.030000	605.030000
Credit_Utilization_Ratio	33.053114	33.811894	32.430559	25.926822	30.116600
Payment_of_Min_Amount	1.000000	1.000000	1.000000	1.000000	1.000000
Total_EMI_per_month	49.574949	49.574949	49.574949	18.816215	18.816215
Amount_invested_monthly	148.233938	148.233938	148.233938	153.534488	153.534488
Payment_Behaviour	4.000000	1.000000	4.000000	5.000000	2.000000
Monthly_Balance	361.444004	264.675446	343.826873	485.298434	303.355083
Credit_Score	1.000000	1.000000	1.000000	1.000000	1.000000
Credit_History_Age_Months	274.000000	274.000000	276.000000	327.000000	328.000000

Step 3. Model Selection-

Choose suitable machine learning classification models for predicting credit scores.Suggested models include:

- Logistic Regression
- Random Forest Classifier
- Support Vector Machine (SVM)
- Gradient Boosting Classifier (e.g., XGBoost)

Step 3: Solution-

Separate properties and target variable:

In [83]:
```
# Separate properties and target variable

X = data.drop("Credit_Score", axis=1)
Y = data.Credit_Score
```

Splitting the dataset into train and test:

In [84]:
```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(X, Y, random_state = 4, test_size = 0.2)
```

In [85]:
```
x_train.shape, y_train.shape
```

Out[85]:
```
((26582, 20), (26582,))
```

In [86]:
```
x_test.shape, y_test.shape
```

Out[86]:
```
((6646, 20), (6646,))
```

Normalizing the data:

In [87]:
```
from sklearn.preprocessing import StandardScaler

s = StandardScaler()
x_train = s.fit_transform(x_train)
x_test = s.fit_transform(x_test)
```

1. Apply Logistic Regression Model:

In [88]:
```
#from sklearn.linear_model import LogisticRegression

from sklearn.linear_model import LogisticRegression
```

In [89]:
```
L_reg = LogisticRegression(multi_class='multinomial', solver='lbfgs', max_iter=200).fit(x_train,y_train)
L_reg
```

Out[89]:
```
▾ LogisticRegression
LogisticRegression(max_iter=200, multi_class='multinomial')
```

In [90]:
```
L_reg.predict(x_test)
```

Out[90]:
```
array([2, 2, 0, ..., 1, 2, 2])
```

In [91]:
```
np.array(y_test)
```

Out[91]:
```
array([2, 2, 0, ..., 1, 2, 2])
```

Evaluate Logistic Regression Model:

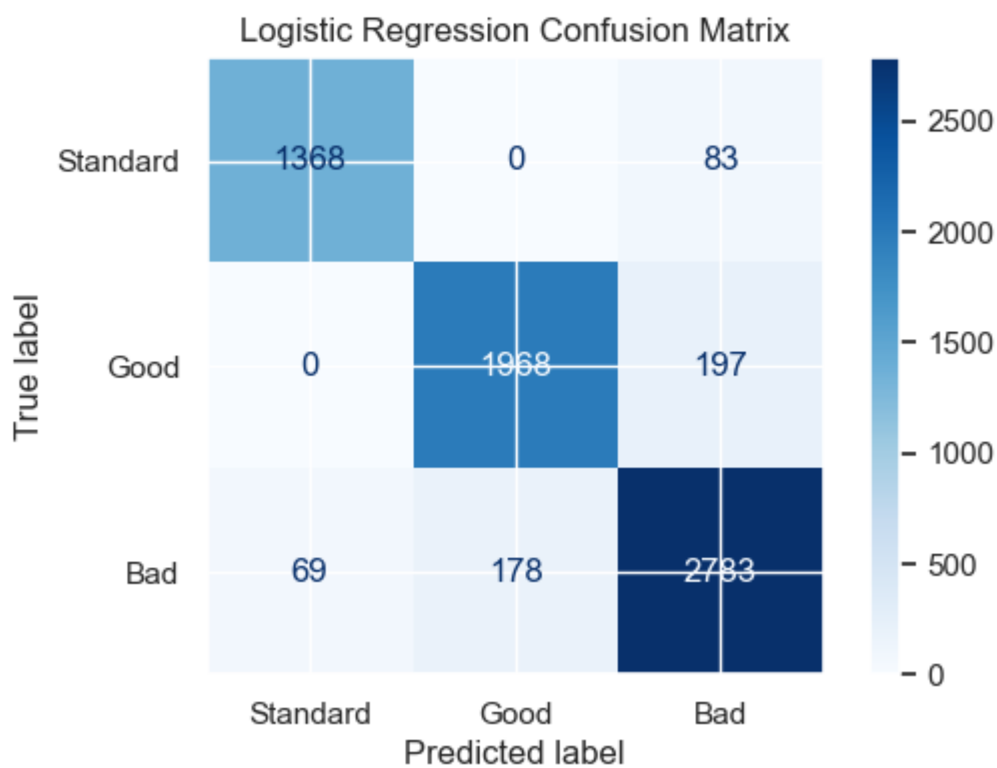
In [92]:
```
from sklearn.metrics import classification_report, accuracy_score, precision_score, recall_score, f1_score
from termcolor import colored
```

In [93]:
```
y_pred = L_reg.predict(x_test)

train_accuracy_LR = L_reg.score(x_train,y_train)
test_accuracy_LR = L_reg.score(x_test, y_test)
accuracy_score_LR = accuracy_score(y_test, y_pred)
precision_score_LR = precision_score(y_test, y_pred, average='weighted')
recall_score_LR = recall_score(y_test, y_pred, average='weighted')
f1_score_LR = f1_score(y_test, y_pred, average='weighted')

print(colored('Logistic Regression Model Evaluation:\n',color = 'blue', attrs = ['bold','dark']))
print(colored(f'train_accuracy : {round(train_accuracy_LR,2)}',color='light_magenta'))
print(colored(f'test_accuracy : {round(test_accuracy_LR,2)}',color='light_magenta'))
print(colored(f'accuracy_score : {round(accuracy_score_LR,2)}',color='light_magenta'))
print(colored(f'precision_score : {round(precision_score_LR,2)}',color='light_magenta'))
print(colored(f'recall_score : {round(recall_score_LR,2)}',color='light_magenta'))
print(colored(f'f1_score : {round(f1_score_LR,2)}',color='light_magenta'))
```

```
Logistic Regression Model Evaluation:

train_accuracy : 0.92
test_accuracy : 0.92
accuracy_score : 0.92
precision_score : 0.92
recall_score : 0.92
f1_score : 0.92
```

Confusion Matrix for Logistic Regression Model:

In [94]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# If y_test and y_pred are already 1D arrays of labels, use them directly
y_test_labels = y_test  # Assuming these are label-encoded
y_pred_labels = y_pred  # Assuming these are label-encoded

# Compute the confusion matrix
cm = confusion_matrix(y_test_labels, y_pred_labels, labels=[0, 1, 2])
fig, ax = plt.subplots(figsize=(6, 4))

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Standard', 'Good', 'Bad'])
disp.plot(cmap=plt.cm.Blues, ax=ax)
plt.title("Logistic Regression Confusion Matrix")
plt.show()
```



2. Apply Random Forest Classifier Model:

In [95]:
```
#import random forest classifier

from sklearn.ensemble import RandomForestClassifier
```

In [96]:
```
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)
```

In [97]:
```
rf_clf.fit(x_train, y_train)
```

Out[97]:
```
▾        RandomForestClassifier
RandomForestClassifier(random_state=42)
```

In [98]:
```
rf_clf.predict(x_test)
```

Out[98]:
```
array([2, 2, 0, ..., 1, 2, 2])
```

In [99]:
```
np.array(y_test)
```

Out[99]:
```
array([2, 2, 0, ..., 1, 2, 2])
```

Evaluate Random Forest Classifier Model:

In [100…

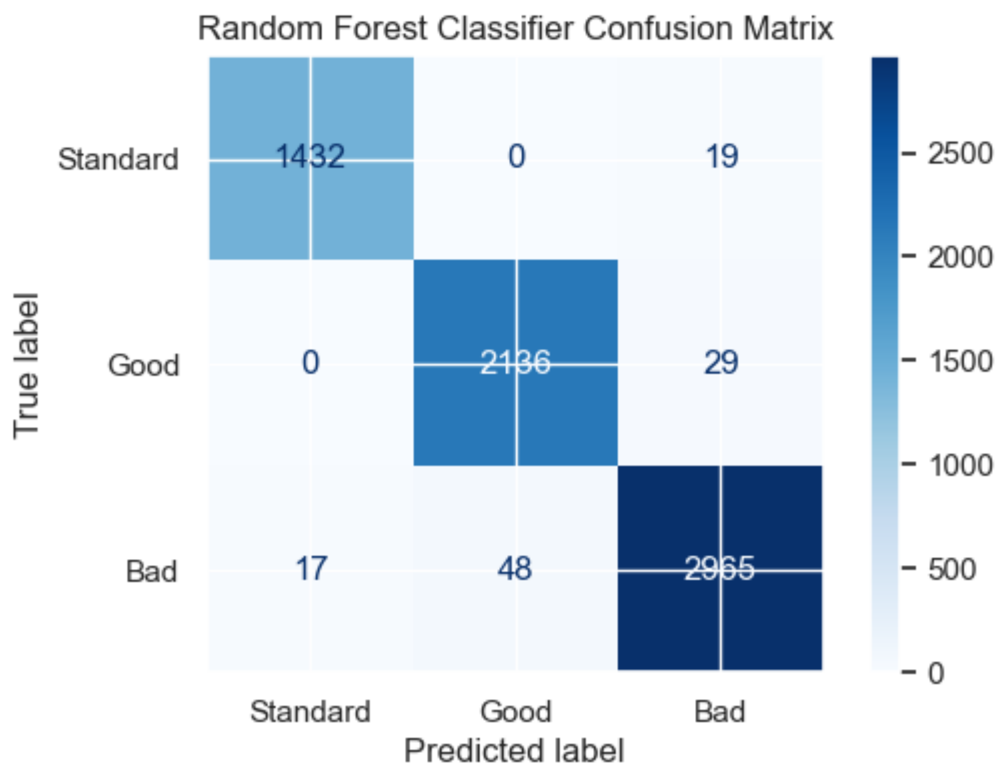
```
y_pred= rf_clf.predict(x_test)

train_accuracy_rf_clf = rf_clf.score(x_train,y_train)
test_accuracy_rf_clf= rf_clf.score(x_test, y_test)
accuracy_score_rf_clf = accuracy_score(y_test, y_pred)
precision_score_rf_clf = precision_score(y_test, y_pred, average='weighted')
recall_score_rf_clf = recall_score(y_test, y_pred, average='weighted')
f1_score_rf_clf= f1_score(y_test, y_pred, average='weighted')

print(colored('Random Forest Classifier Model Evaluation:\n',color = 'blue', attrs = ['bold','dark']))
print(colored(f'train_accuracy : {round(train_accuracy_rf_clf,2)}',color='light_magenta'))
print(colored(f'test_accuracy : {round(test_accuracy_rf_clf,2)}',color='light_magenta'))
print(colored(f'accuracy_score : {round(accuracy_score_rf_clf,2)}',color='light_magenta'))
print(colored(f'precision_score : {round(precision_score_rf_clf,2)}',color='light_magenta'))
print(colored(f'recall_score : {round(recall_score_rf_clf,2)}',color='light_magenta'))
print(colored(f'f1_score : {round(f1_score_rf_clf,2)}',color='light_magenta'))
```

```
Random Forest Classifier Model Evaluation:

train_accuracy : 1.0
test_accuracy : 0.98
accuracy_score : 0.98
precision_score : 0.98
recall_score : 0.98
f1_score : 0.98
```

Confusion Matrix for Random Forest Classifier Model:

In [101…

```
# If y_test and y_pred are already 1D arrays of labels, use them directly
y_test_labels = y_test  # Assuming these are label-encoded
y_pred_labels = y_pred  # Assuming these are label-encoded

# Compute the confusion matrix
cm = confusion_matrix(y_test_labels, y_pred_labels, labels=[0, 1, 2])
fig, ax = plt.subplots(figsize=(6, 4))

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Standard', 'Good', 'Bad'])
disp.plot(cmap=plt.cm.Blues, ax=ax)
plt.title("Random Forest Classifier Confusion Matrix")
plt.show()
```



3. Apply Support Vector Machine (SVM) Model:

In [102…

```
#import SVR classifier

from sklearn.svm import SVC
```

In [103…

```
SVC= SVC().fit(x_train, y_train)
```

In [104…

```
SVC.predict(x_test)
```

Out[104]:

```
array([2, 2, 0, ..., 1, 2, 2])
```

In [105…

```
np.array(y_test)
```

Out[105]:

```
array([2, 2, 0, ..., 1, 2, 2])
```

Evaluate Support Vector Machine (SVM) Model:

In [106…

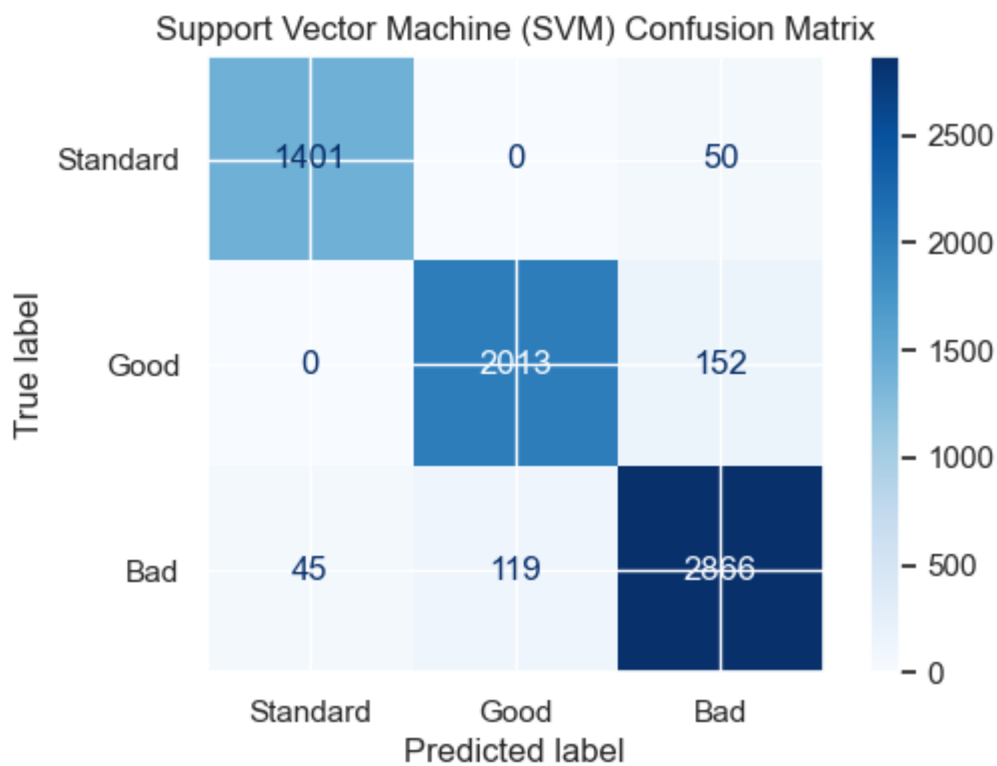
```
y_pred= SVC.predict(x_test)

train_accuracy_SVC = SVC.score(x_train,y_train)
test_accuracy_SVC = SVC.score(x_test, y_test)
accuracy_score_SVC = accuracy_score(y_test, y_pred)
precision_score_SVC = precision_score(y_test, y_pred, average='weighted')
recall_score_SVC = recall_score(y_test, y_pred, average='weighted')
f1_score_SVC = f1_score(y_test, y_pred, average='weighted')

print(colored('Support Vector Machine (SVM) Model Evaluation:\n',color = 'blue', attrs = ['bold','dark']))
print(colored(f'train_accuracy : {round(train_accuracy_SVC,2)}',color='light_magenta'))
print(colored(f'test_accuracy : {round(test_accuracy_SVC,2)}',color='light_magenta'))
print(colored(f'accuracy_score : {round(accuracy_score_SVC,2)}',color='light_magenta'))
print(colored(f'precision_score : {round(precision_score_SVC,2)}',color='light_magenta'))
print(colored(f'recall_score : {round(recall_score_SVC,2)}',color='light_magenta'))
print(colored(f'f1_score : {round(f1_score_SVC,2)}',color='light_magenta'))
```

```
Support Vector Machine (SVM) Model Evaluation:

train_accuracy : 0.95
test_accuracy : 0.94
accuracy_score : 0.94
precision_score : 0.95
recall_score : 0.94
f1_score : 0.94
```

Confusion Matrix for Support Vector Machine (SVM) Model:

In [107…

```
# If y_test and y_pred are already 1D arrays of labels, use them directly
y_test_labels = y_test  # Assuming these are label-encoded
y_pred_labels = y_pred  # Assuming these are label-encoded

# Compute the confusion matrix
cm = confusion_matrix(y_test_labels, y_pred_labels, labels=[0, 1, 2])
fig, ax = plt.subplots(figsize=(6, 4))

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Standard', 'Good', 'Bad'])
disp.plot(cmap=plt.cm.Blues, ax=ax)
plt.title("Support Vector Machine (SVM) Confusion Matrix")
plt.show()
```



4. Apply XGBoost Classifier Model:

In [108…
```
#Import XGBoost libs

import xgboost
from xgboost import XGBClassifier
```

In [109…
```
XGB = XGBClassifier().fit(x_train, y_train)
```

In [110…
```
XGB.predict(x_test)
```

Out[110]:
```
array([2, 2, 0, ..., 1, 2, 2], dtype=int64)
```

In [111…
```
np.array(y_test)
```

Out[111]:
```
array([2, 2, 0, ..., 1, 2, 2])
```

Evaluate XGBoost Classifier Model:

In [112…
```
y_pred= XGB.predict(x_test)

train_accuracy_XGB = XGB.score(x_train,y_train)
test_accuracy_XGB = XGB.score(x_test, y_test)
accuracy_score_XGB = accuracy_score(y_test, y_pred)
precision_score_XGB = precision_score(y_test, y_pred, average='weighted')
recall_score_XGB = recall_score(y_test, y_pred, average='weighted')
f1_score_XGB = f1_score(y_test, y_pred, average='weighted')

print(colored('XGBoost Classifier Model Evaluation:\n',color = 'blue', attrs = ['bold','dark']))
print(colored(f'train_accuracy : {round(train_accuracy_XGB,2)}',color='light_magenta'))
print(colored(f'test_accuracy : {round(test_accuracy_XGB,2)}',color='light_magenta'))
print(colored(f'accuracy_score : {round(accuracy_score_XGB,2)}',color='light_magenta'))
print(colored(f'precision_score : {round(precision_score_XGB,2)}',color='light_magenta'))
print(colored(f'recall_score : {round(recall_score_XGB,2)}',color='light_magenta'))
print(colored(f'f1_score : {round(f1_score_XGB,2)}',color='light_magenta'))
```

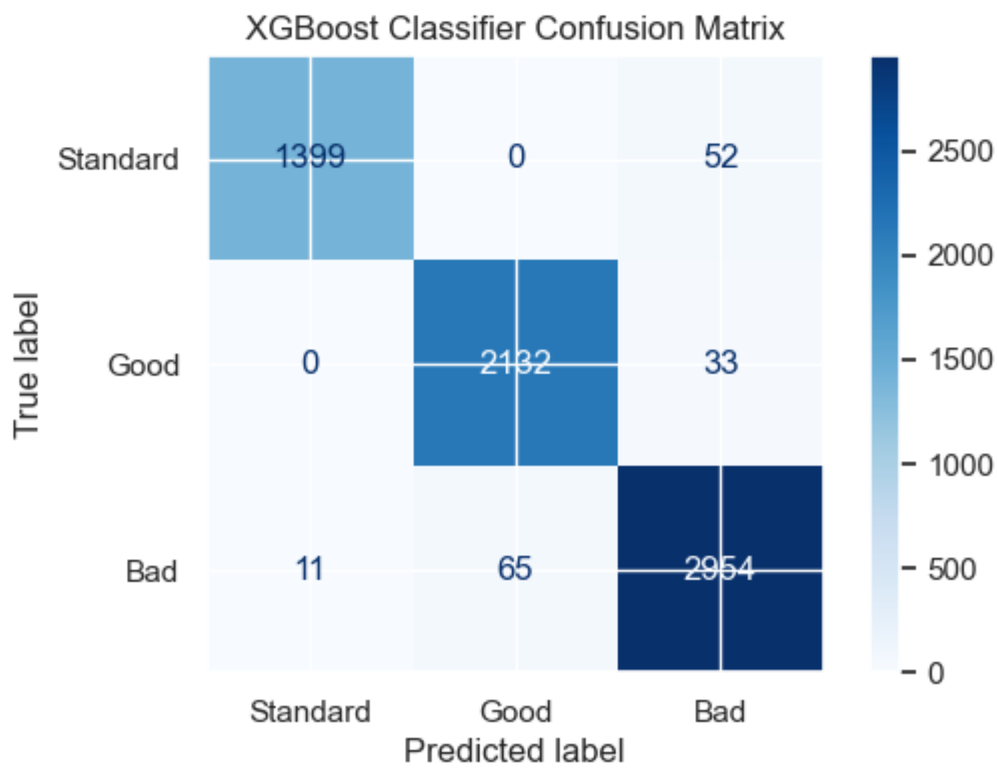
XGBoost Classifier Model Evaluation:

```
train_accuracy : 1.0
test_accuracy : 0.98
accuracy_score : 0.98
precision_score : 0.98
recall_score : 0.98
f1_score : 0.98
```

Confusion Matrix for XGBoost Classifier Model:

In [113…

```
# If y_test and y_pred are already 1D arrays of labels, use them directly
y_test_labels = y_test  # Assuming these are label-encoded
y_pred_labels = y_pred  # Assuming these are label-encoded

# Compute the confusion matrix
cm = confusion_matrix(y_test_labels, y_pred_labels, labels=[0, 1, 2])
fig, ax = plt.subplots(figsize=(6, 4))

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Standard', 'Good', 'Bad'])
disp.plot(cmap=plt.cm.Blues, ax=ax)
plt.title("XGBoost Classifier Confusion Matrix")
plt.show()
```



Step 4: Model Training-

- Train each selected model using the training dataset.
- Utilize evaluation metrics suitable for classification tasks, such as accuracy, precision, recall, F1 score, and confusion matrix.

In [114…

```
#I already done this part in Step 3.
```

Step 5: Hyperparameter Tuning-

- Conduct hyperparameter tuning for at least one model using methods like Grid Search or Random Search.
- Explain the chosen hyperparameters and the reasoning behind them.

Step 5: Solution-

- I use RandomizedSearchCV for hyperparameter optimization. It basically works with various parameters internally and finds out the best parameters.
- I apply Hyperparameter tuning technique in our Random Forest Classifier & XGBoost Classifier Model because these two model gives present higher accuracy.

Initialized Hyperparameters:

In [115…

```
#import hyperparameter
from sklearn.model_selection import RandomizedSearchCV
```

In [116…

```
#I use RandomizedSearchCV for hyperparameter optimization

from scipy.stats import randint, uniform, loguniform

#Define hyperparameters for Random Forest
rf_params = {
    'n_estimators': np.arange(10, 200, 10),
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth': np.arange(5, 50, 5),
    'min_samples_split': np.arange(2, 10, 2),
    'min_samples_leaf': np.arange(1, 10, 2),
    'bootstrap': [True, False]
}

#Define hyperparameters for  XGBoost Classifier
xgb_params = {
    'n_estimators': randint(50, 1000),
    'max_depth': randint(3, 10),
    'learning_rate': uniform(0.01, 0.3),
    'subsample': uniform(0.6, 0.4),
    'colsample_bytree': uniform(0.6, 0.4),
    'gamma': uniform(0, 0.5),
    'min_child_weight': randint(1, 10),
    'reg_alpha': uniform(0, 1),
    'reg_lambda': uniform(0, 1),
    'scale_pos_weight': uniform(1, 3)
}
```

1. Random Forest Classifier Model for DRandomizedSearchCV Hyperparameter:

In [117…

```
rf = RandomForestClassifier()

rf_random_search = RandomizedSearchCV(
    rf, param_distributions=rf_params, n_iter=100, cv=5, verbose=2, random_state=42, n_jobs=-1
)
```

In [118…

```
rf_random_search.fit(x_train, y_train)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
```

Out[118]:

```
▸ RandomizedSearchCV
▸ estimator: RandomForestClassifier
    ▸ RandomForestClassifier
```

In [119…

```
rf_random_search.best_params_
```

Out[119]:

```
{'n_estimators': 100,
 'min_samples_split': 2,
 'min_samples_leaf': 1,
 'max_features': 'sqrt',
 'max_depth': 45,
 'bootstrap': False}
```

Evaluate Random Forest Classifier Model for DRandomizedSearchCV Hyperparameter:

In [120…

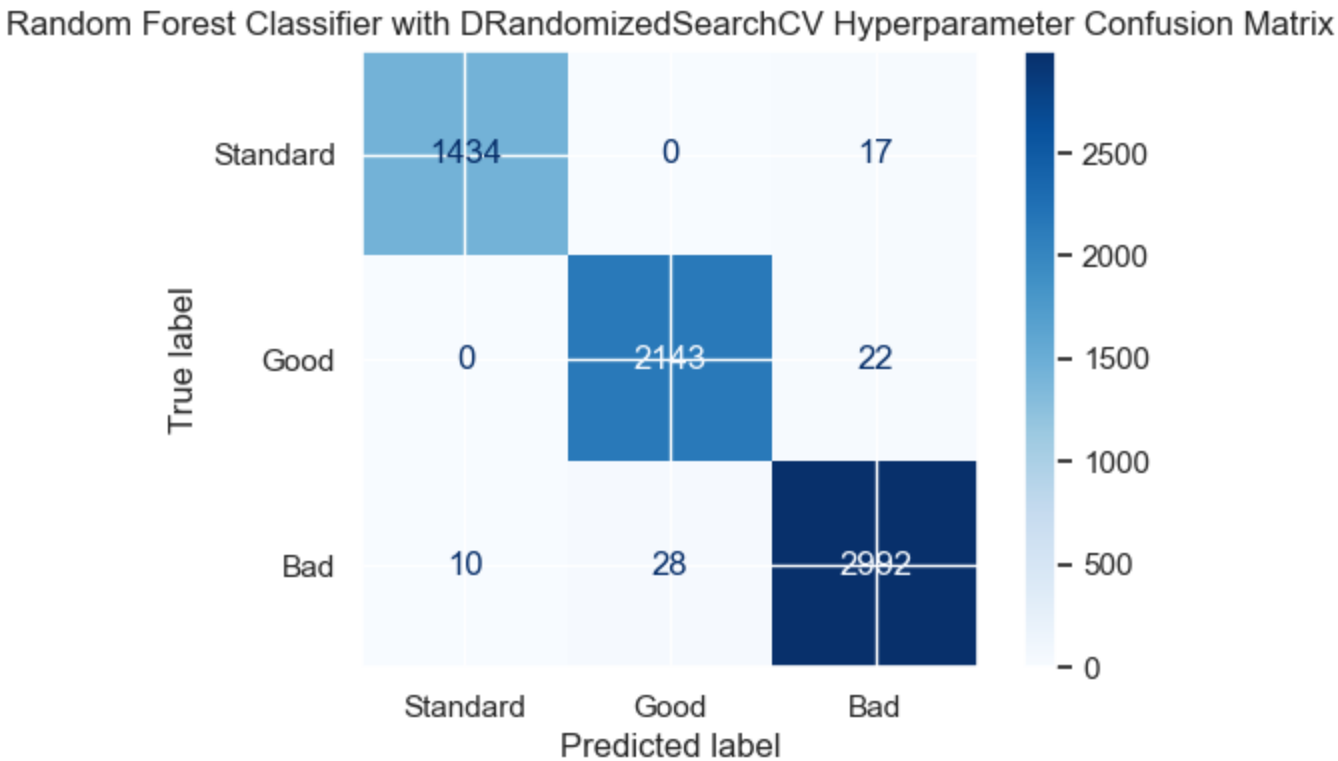
```
y_pred= rf_random_search.predict(x_test)

train_accuracy_rf_RSH = rf_random_search.score(x_train,y_train)
test_accuracy_rf_RSH= rf_random_search.score(x_test, y_test)
accuracy_score_rf_RSH = accuracy_score(y_test, y_pred)
precision_score_rf_RSH = precision_score(y_test, y_pred, average='weighted')
recall_score_rf_RSH = recall_score(y_test, y_pred, average='weighted')
f1_score_rf_RSH= f1_score(y_test, y_pred, average='weighted')

print(colored('Random Forest Classifier with DRandomizedSearchCV Hyperparameter Model Evaluation:\n',color = 'blue',
print(colored(f'train_accuracy : {round(train_accuracy_rf_RSH,2)}',color='light_magenta'))
print(colored(f'test_accuracy : {round(test_accuracy_rf_RSH,2)}',color='light_magenta'))
print(colored(f'accuracy_score : {round(accuracy_score_rf_RSH,2)}',color='light_magenta'))
print(colored(f'precision_score : {round(precision_score_rf_RSH,2)}',color='light_magenta'))
print(colored(f'recall_score : {round(recall_score_rf_RSH,2)}',color='light_magenta'))
print(colored(f'f1_score : {round(f1_score_rf_RSH,2)}',color='light_magenta'))
```

```
Random Forest Classifier with DRandomizedSearchCV Hyperparameter Model Evaluation:

train_accuracy : 1.0
test_accuracy : 0.99
accuracy_score : 0.99
precision_score : 0.99
recall_score : 0.99
f1_score : 0.99
```

Confusion Matrix for Random Forest Classifier Model with DRandomizedSearchCV Hyperparameter:

In [121…

```
# If y_test and y_pred are already 1D arrays of labels, use them directly
y_test_labels = y_test  # Assuming these are label-encoded
y_pred_labels = y_pred  # Assuming these are label-encoded

# Compute the confusion matrix
cm = confusion_matrix(y_test_labels, y_pred_labels, labels=[0, 1, 2])
fig, ax = plt.subplots(figsize=(6, 4))

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Standard', 'Good', 'Bad'])
disp.plot(cmap=plt.cm.Blues, ax=ax)
plt.title("Random Forest Classifier with DRandomizedSearchCV Hyperparameter Confusion Matrix")
plt.show()
```



2. XGBoost classifier Model for DRandomizedSearchCV Hyperparameter:

In [122…

```
xgb = XGBClassifier(use_label_encoder=False, eval_metric='mlogloss')

xgb_random_search = RandomizedSearchCV(
    xgb, param_distributions=xgb_params, n_iter=100, cv=5, verbose=2, random_state=42, n_jobs=-1
)
```

In [123…
```
xgb_random_search.fit(x_train, y_train)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
```

Out[123]:
```
▸ RandomizedSearchCV
▸ estimator: XGBClassifier
  ▸ XGBClassifier
```

In [124…
```
xgb_random_search.best_params_
```

Out[124]:
```
{'colsample_bytree': 0.6682096494749166,
 'gamma': 0.03252579649263976,
 'learning_rate': 0.29466566117599996,
 'max_depth': 6,
 'min_child_weight': 2,
 'n_estimators': 826,
 'reg_alpha': 0.015966252220214194,
 'reg_lambda': 0.230893825622149,
 'scale_pos_weight': 1.7230763980780353,
 'subsample': 0.8733054075301833}
```

Evaluate XGBoost classifier Model for DRandomizedSearchCV Hyperparameter:

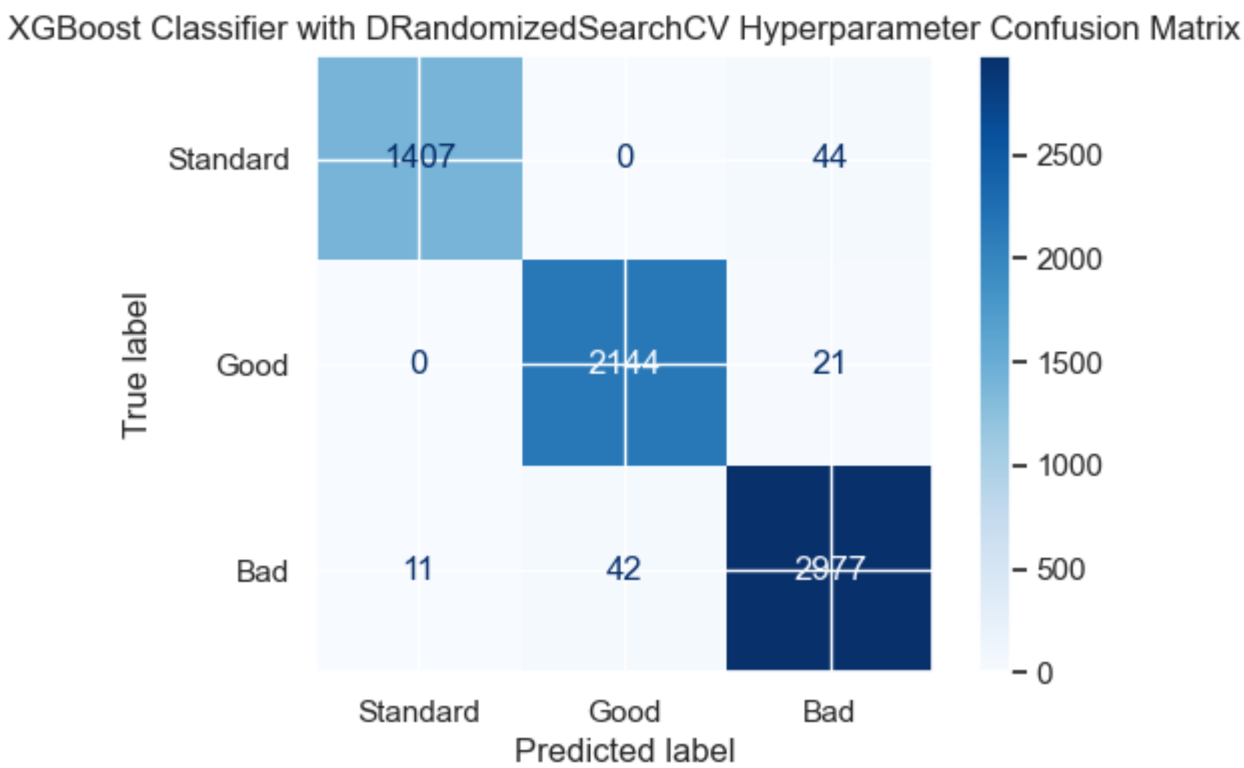
In [125…
```
y_pred= xgb_random_search.predict(x_test)

train_accuracy_XGB_RSH = xgb_random_search.score(x_train,y_train)
test_accuracy_XGB_RSH = xgb_random_search.score(x_test, y_test)
accuracy_score_XGB_RSH = accuracy_score(y_test, y_pred)
precision_score_XGB_RSH = precision_score(y_test, y_pred, average='weighted')
recall_score_XGB_RSH = recall_score(y_test, y_pred, average='weighted')
f1_score_XGB_RSH = f1_score(y_test, y_pred, average='weighted')

print(colored('XGBoost Classifier for DRandomizedSearchCV Hyperparameter Model Evaluation:\n',color = 'blue', attrs =
print(colored(f'train_accuracy : {round(train_accuracy_XGB_RSH,2)}',color='light_magenta'))
print(colored(f'test_accuracy : {round(test_accuracy_XGB_RSH,2)}',color='light_magenta'))
print(colored(f'accuracy_score : {round(accuracy_score_XGB_RSH,2)}',color='light_magenta'))
print(colored(f'precision_score : {round(precision_score_XGB_RSH,2)}',color='light_magenta'))
print(colored(f'recall_score : {round(recall_score_XGB_RSH,2)}',color='light_magenta'))
print(colored(f'f1_score : {round(f1_score_XGB_RSH,2)}',color='light_magenta'))
```

```
XGBoost Classifier for DRandomizedSearchCV Hyperparameter Model Evaluation:

train_accuracy : 1.0
test_accuracy : 0.98
accuracy_score : 0.98
precision_score : 0.98
recall_score : 0.98
f1_score : 0.98
```

Confusion Matrix for XGBoost Classifier Model with DRandomizedSearchCV Hyperparameter:

In [126…

```
# If y_test and y_pred are already 1D arrays of labels, use them directly
y_test_labels = y_test  # Assuming these are label-encoded
y_pred_labels = y_pred  # Assuming these are label-encoded

# Compute the confusion matrix
cm = confusion_matrix(y_test_labels, y_pred_labels, labels=[0, 1, 2])
fig, ax = plt.subplots(figsize=(6, 4))

# Plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Standard', 'Good', 'Bad'])
disp.plot(cmap=plt.cm.Blues, ax=ax)
plt.title("XGBoost Classifier with DRandomizedSearchCV Hyperparameter Confusion Matrix")
plt.show()
```



Observations:

- Perform DRandomizedSearchCV Hyperparameter, I see that it gives better accuracy in both Random Forest Classifier and XGBoost Classifier Model.

Explain the chosen hyperparameters:

Here, I use Random Search for Hyperparameter Tuning.

Random Search:

Random Search randomly samples the hyperparameter space. Instead of trying all combinations, it tries a fixed number of random combinations.

- Advantages:

  `1. Efficiency:` Random Search can be more efficient than Grid Search. It can cover a larger area of the hyperparameter space with fewer evaluations, especially when some hyperparameters do not significantly impact performance.

  `2. Flexibility:` It allows for more flexibility in the number of parameter combinations tested. You can control the computational cost by setting the number of iterations.

  `3. Potential to Find Better Solutions:` It has the potential to find better hyperparameter combinations, as it is not restricted to a fixed grid and can explore more diverse configurations.

  `4. Empirical Evidence:` Research has shown that Random Search can be as effective, if not more, than Grid Search for many practical problems, particularly when only a few hyperparameters significantly impact performance.

Step 6: Model Evaluation:-

- Assess the performance of each model on the testing set.
- Discuss the strengths and limitations of each model in the context of credit score classification.

Step 6: Solution-

Compare the performance of different models:

In [127…

```
#Create a table to compare different models
d = {
    'model' : ['Logistic Regression', 'Random Forest Classifier',
               'Support Vector Machine (SVM)','XGBoost Classifier',
               'Random Forest Classifier with Random Search Hyperparameter',
               'XGBoost classifier with Random Search Hyperparameter'],

    'train accuracy': [train_accuracy_LR,train_accuracy_rf_clf,train_accuracy_SVC,train_accuracy_XGB,train_accuracy_r
    'test accuracy': [test_accuracy_LR,test_accuracy_rf_clf,test_accuracy_SVC,test_accuracy_XGB,test_accuracy_rf_RSH,
    'accuracy score': [accuracy_score_LR,accuracy_score_rf_clf,accuracy_score_SVC,accuracy_score_XGB,accuracy_score_r
    'precision score': [precision_score_LR,precision_score_rf_clf,precision_score_SVC,precision_score_XGB,precision_s
    'recall score': [recall_score_LR,recall_score_rf_clf,recall_score_SVC,recall_score_XGB,recall_score_rf_RSH,recal
    'f1_score': [f1_score_LR,f1_score_rf_clf,f1_score_SVC,f1_score_XGB,f1_score_rf_RSH,f1_score_XGB_RSH],
}


d = pd.DataFrame(d)

d['train accuracy'] = d['train accuracy'].round(2)
d['test accuracy'] = d['test accuracy'].round(2)
d['accuracy score'] = d['accuracy score'].round(2)
d['precision score'] = d['precision score'].round(2)
d['recall score'] = d['recall score'].round(2)
d['f1_score'] = d['f1_score'].round(2)

d
```

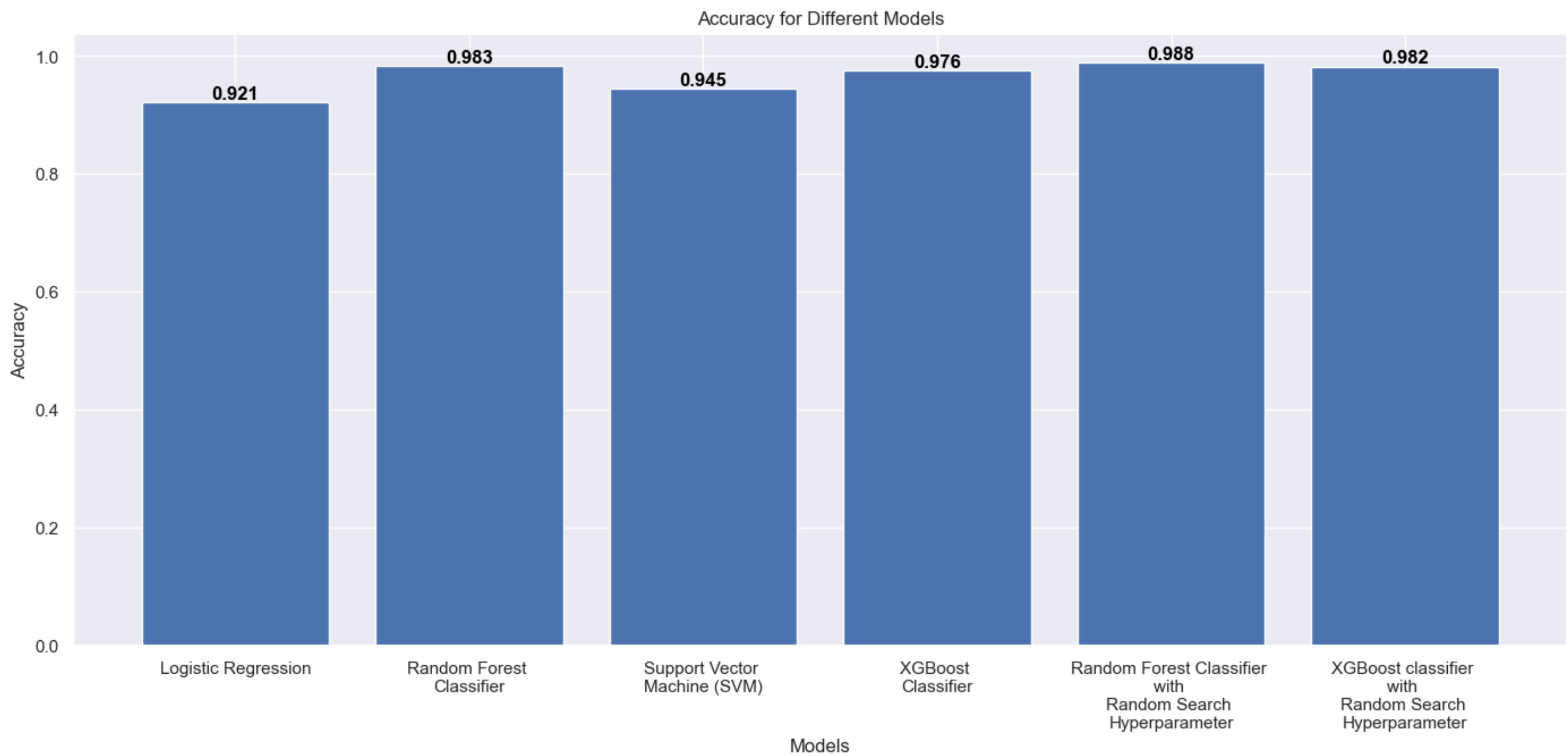
Out[127]:

	model	train accuracy	test accuracy	accuracy score	precision score	recall score	f1_score
0	Logistic Regression	0.92	0.92	0.92	0.92	0.92	0.92
1	Random Forest Classifier	1.00	0.98	0.98	0.98	0.98	0.98
2	Support Vector Machine (SVM)	0.95	0.94	0.94	0.95	0.94	0.94
3	XGBoost Classifier	1.00	0.98	0.98	0.98	0.98	0.98
4	Random Forest Classifier with Random Search Hy...	1.00	0.99	0.99	0.99	0.99	0.99
5	XGBoost classifier with Random Search Hyperpar...	1.00	0.98	0.98	0.98	0.98	0.98

Visualize performance with histogram:

In [128…

```
#visualize performance in histogram
def plot_histogram(metric_values, model_names, metric_name):
    fig, ax = plt.subplots(figsize=(17, 7))
    bars = plt.bar(model_names, metric_values)
    plt.xlabel('Models')
    plt.ylabel(metric_name)
    plt.title(f'{metric_name} for Different Models')

    for bar in bars:
        yval = bar.get_height()
        ax.text(bar.get_x() + bar.get_width()/2, yval, round(yval, 3), ha='center', va='bottom', color='black', fontw
    plt.show()

accuracy_values = [accuracy_score_LR,accuracy_score_rf_clf,accuracy_score_SVC,accuracy_score_XGB,accuracy_score_rf_RS
model_names = ['Logistic Regression', 'Random Forest \nClassifier',
               'Support Vector \nMachine (SVM)','XGBoost \nClassifier',
               'Random Forest Classifier \nwith \nRandom Search \nHyperparameter',
               'XGBoost classifier \nwith \nRandom Search \nHyperparameter' ]
plot_histogram(accuracy_values, model_names, 'Accuracy')
```



Identify the strengths and weaknesses of each model:

1. Logistic Regression-

Accuracy: 92.1%

Strengths:

- `Interpretability:` Logistic Regression provides clear insights into the contribution of each feature to the final decision, making it easy to understand and interpret.

- `Performance on Linearly Separable Data:` It performs well on linearly separable data.

- `Efficiency:` It is computationally efficient and works well on large datasets.

Weaknesses:

- `Linear Boundaries:` It assumes a linear relationship between the features and the log-odds of the outcome, which may not be suitable for complex datasets with non-linear relationships.
- `Outliers:` Sensitive to outliers, which can skew the results.

2. Random Forest Classifier-

Accuracy: 98.3%

With Hyperparameter Tuning Accuracy: 98.8%

Strengths:

- `Robustness:` Robust to overfitting due to the averaging of multiple decision trees.
- `Performance:` Generally provides good accuracy and handles non-linear relationships well.
- `Feature Importance:` Can provide insights into feature importance.
- `Control Over Model Complexity:` Hyperparameter tuning allows for better control over model complexity, helping to balance bias and variance.

Weaknesses:

- `Complexity:` The model is more complex and less interpretable compared to simpler models like Logistic Regression.
- `Computational Cost:` Training and prediction can be computationally intensive, especially with large numbers of trees.
- `Counterintuitive Results:` As seen with the 81% accuracy post-tuning, improper tuning or overfitting to the validation

data can actually degrade performance. This highlights the risk of tuning leading to suboptimal results if not done correctly.

3. Support Vector Machine (SVM)-

Accuracy: 94.5%

Strengths:

- Effective in high-dimensional spaces, making it suitable for complex datasets.
- Can be customized with different kernel functions (linear, polynomial, RBF) to find optimal decision boundaries.
- Robust to overfitting, especially with the use of a proper kernel and regularization.

Weaknesses:

- Computationally intensive, especially for large datasets.
- Performance can degrade with noisy data or when classes are not well separated.
- Difficult to interpret and visualize the model, particularly with non-linear kernels.

4. XGBoost Classifier-

Accuracy: 97.6%

With Hyperparameter Tuning Accuracy: 98.2%

Strengths:

- High predictive power and accuracy due to gradient boosting, which combines weak learners.
- Handles missing values well and can manage a variety of data types.
- Offers feature importance metrics, aiding interpretability and feature selection.

Weaknesses:

- Computationally expensive, especially during training with a large number of trees.
- Sensitive to overfitting if not properly regularized.
- Requires careful tuning of many hyperparameters to achieve optimal performance.

Step 7. Interpretability-

- If applicable, explore methods to interpret the model's decisions and understand the factors influencing credit score classifications.

Step 7. Solution -

Summary:

Logistic Regression (92.1%):

- High accuracy, simple, and interpretable, but may struggle with non-linear relationships.

Random Forest Classifier (98.3% & With Hyperparameter Tuning 98.8%):

- Good accuracy and robust, but complex and computationally intensive.
- Improper tuning or overfitting to the validation data can actually degrade performance.

Support Vector Machine (SVM) (94.5%):

- SVM models are strong with structured data and high dimensions, but may struggle with interpretability and noise.

XGBoost Classifier(97.6% & With Hyperparameter Tuning 98.2%):

- XGBoost offers strong predictive power and robustness but requires careful tuning and can be computationally intensive.

Conclusion:

Based on the accuracies provided, the models with the highest performance are:

- `Random Forest Classifier with Random Search Hyperparameter Tuning - 98.8%`
- `XGBoost Classifier with Random Search Hyperparameter Tuning - 98.2%`

To select the best model between these two, consider additional factors beyond accuracy, such as:

- **1. Precision, Recall, and F1 Score:** These metrics provide insights into the model's performance, especially in imbalanced datasets.
  - **Precision** measures the proportion of true positive predictions among all positive predictions, indicating the model's ability to avoid false positives.
  - **Recall** (or Sensitivity) measures the proportion of true positives among all actual positives, reflecting the model's ability to capture true cases.
  - **F1 Score** is the harmonic mean of precision and recall, providing a balance between the two.
- **2. Confusion Matrix:** This provides a comprehensive view of the model's performance by showing the counts of true positives, true negatives, false positives, and false negatives. It can help identify if the model is biased toward one class.

- **3. ROC Curve and AUC (Area Under the Curve):** The ROC curve plots the true positive rate against the false positive rate at various threshold settings. The AUC score summarizes the model's ability to distinguish between classes.

- **4. Training Time and Model Complexity:** Consider the time taken to train the model and the complexity of the model (e.g., the number of parameters). Simpler models are often preferred if they offer comparable performance, as they are easier to interpret and deploy.

- **5. Scalability and Resource Consumption:** Evaluate the model's resource consumption in terms of memory and computation, especially if deploying in a production environment with limited resources.

- **6. Interpretability:** Some models are more interpretable than others. For example, decision trees and linear models are generally easier to interpret than complex models like XGBoost. Depending on the application, interpretability might be an essential factor.

- **7. Robustness to Outliers and Missing Data:** Consider how sensitive the models are to outliers or missing data, as some models may handle these issues better than others.

- **8. Generalization:** Evaluate the model's performance on unseen data to ensure it generalizes well beyond the training dataset

- **8. Generalization:** Evaluate the model's performance on unseen data to ensure it generalizes well beyond the training dataset.

By considering these factors, we can make a more informed decision about which model to choose for your specific use case.

Recommendation:

In the above study, I find that in order to predict the Credit_Score-

- The best model is Random Forest Classifier.
- Using the Random Forest Classifier with Random Search Hyperparameter Tuning, I can predict the Credit_Score accurately between 98.3% to 98.8% data.

In [ ]:

```

```