credit-risk-and-analysis

October 13, 2024

[2]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```python
df = pd.read_csv('Credit_score.csv')
```

```
<ipython-input-3-6ebc9aea8e05>:1: DtypeWarning: Columns (26) have mixed types.
Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('Credit_score.csv')
```

Data Description:

1. ID, Represents a unique identification of an entry
2. Customer_ID, Represents a unique identification of a person
3. Month, Represents the month of the year
4. Name, Represents the name of a person
5. Age, Represents the age of the person
6. SSN, Represents the social security number of a person
7. Occupation, Represents the occupation of the person
8. Annual_Income, Represents the annual income of the person
9. Monthly_Inhand_Salary, Represents the monthly base salary of a person
10. Num_Bank_Accounts, Represents the number of bank accounts a person holds
11. Num_Credit_Card, Represents the number of other credit cards held by a person
12. Interest_Rate, Represents the interest rate on credit card
13. Num_of_Loan, Represents the number of loans taken from the bank
14. Type_of_Loan, Represents the types of loan taken by a person
15. Delay_from_due_date, Represents the average number of days delayed from the payment date
16. Num_of_Delayed_Payment, Represents the average number of payments delayed by a person
17. Changed_Credit_Limit, Represents the percentage change in credit card limit
18. Num_Credit_Inquiries, Represents the number of credit card inquiries
19. Credit_Mix, Represents the classification of the mix of credits
20. Outstanding_Debt, Represents the remaining debt to be paid (in USD)
21. Credit_Utilization_Ratio, Represents the utilization ratio of credit card
22. Credit_History_Age, Represents the age of credit history of the person
23. Payment_of_Min_Amount, Represents whether only the minimum amount was paid by the person
24. Total_EMI_per_month, Represents the monthly EMI payments (in USD)

25. Amount_invested_monthly, Represents the monthly amount invested by the customer (in USD)
26. Payment_Behaviour, Represents the payment behavior of the customer (in USD)
27. Monthly_Balance, Represents the monthly balance amount of the customer (in USD)

[4]:
```
df.head()
```

[4]:
```
       ID Customer_ID     Month           Name   Age          SSN Occupation  \
0  0x1602   CUS_0xd40   January  Aaron Maashoh    23  821-00-0265  Scientist
1  0x1603   CUS_0xd40  February  Aaron Maashoh    23  821-00-0265  Scientist
2  0x1604   CUS_0xd40     March  Aaron Maashoh  -500  821-00-0265  Scientist
3  0x1605   CUS_0xd40     April  Aaron Maashoh    23  821-00-0265  Scientist
4  0x1606   CUS_0xd40       May  Aaron Maashoh    23  821-00-0265  Scientist

  Annual_Income  Monthly_Inhand_Salary  Num_Bank_Accounts  …  \
0      19114.12            1824.843333                  3  …
1      19114.12                    NaN                  3  …
2      19114.12                    NaN                  3  …
3      19114.12                    NaN                  3  …
4      19114.12            1824.843333                  3  …

   Num_Credit_Inquiries  Credit_Mix Outstanding_Debt Credit_Utilization_Ratio  \
0                   4.0           _           809.98                26.822620
1                   4.0        Good           809.98                31.944960
2                   4.0        Good           809.98                28.609352
3                   4.0        Good           809.98                31.377862
4                   4.0        Good           809.98                24.797347

      Credit_History_Age Payment_of_Min_Amount Total_EMI_per_month  \
0  22 Years and 1 Months                    No           49.574949
1                    NaN                    No           49.574949
2  22 Years and 3 Months                    No           49.574949
3  22 Years and 4 Months                    No           49.574949
4  22 Years and 5 Months                    No           49.574949

   Amount_invested_monthly                 Payment_Behaviour Monthly_Balance
0              80.41529544   High_spent_Small_value_payments     312.4940887
1              118.2802216    Low_spent_Large_value_payments     284.6291625
2              81.69952126   Low_spent_Medium_value_payments     331.2098629
3              199.4580744    Low_spent_Small_value_payments     223.4513097
4              41.42015309  High_spent_Medium_value_payments      341.489231

[5 rows x 27 columns]
```

[5]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
```

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

[6]:
```
df.describe()
```

[6]:
```
       Monthly_Inhand_Salary  Num_Bank_Accounts  Num_Credit_Card  \
count           84998.000000      100000.000000     100000.00000   
mean             4194.170850          17.091280         22.47443   
std              3183.686167         117.404834        129.05741   
min               303.645417          -1.000000          0.00000   
25%              1625.568229           3.000000          4.00000   
50%              3093.745000           6.000000          5.00000   
75%              5957.448333           7.000000          7.00000   
max             15204.633330        1798.000000       1499.00000   

       Interest_Rate  Delay_from_due_date  Num_Credit_Inquiries  \
count  100000.000000        100000.000000          98035.000000   
```

```
mean      72.466040           21.068780           27.754251
std      466.422621           14.860104          193.177339
min        1.000000           -5.000000            0.000000
25%        8.000000           10.000000            3.000000
50%       13.000000           18.000000            6.000000
75%       20.000000           28.000000            9.000000
max     5797.000000           67.000000         2597.000000

       Credit_Utilization_Ratio  Total_EMI_per_month
count             100000.000000        100000.000000
mean                  32.285173          1403.118217
std                    5.116875          8306.041270
min                   20.000000             0.000000
25%                   28.052567            30.306660
50%                   32.305784            69.249473
75%                   36.496663           161.224249
max                   50.000000         82331.000000
```

Exploratory Data Analysis (EDA): - Perform a comprehensive EDA to understand the data's structure, characteristics, distributions, and relationships. - Identify and address any missing values, mismatch data types, inconsistencies, or outliers. - Utilize appropriate visualizations (e.g., histograms, scatter plots, box plots, correlation matrices) to uncover patterns and insights.

Mismatch Data Types and Inconsistencies

[7]:
```python
# Converting the 'Age' and other numeric fields from object to numeric types
df['Age'] = pd.to_numeric(df['Age'], errors='coerce')
df['Annual_Income'] = pd.to_numeric(df['Annual_Income'], errors='coerce')
df['Num_of_Loan'] = pd.to_numeric(df['Num_of_Loan'], errors='coerce')
df['Num_of_Delayed_Payment'] = pd.to_numeric(df['Num_of_Delayed_Payment'], errors='coerce')
df['Changed_Credit_Limit'] = pd.to_numeric(df['Changed_Credit_Limit'], errors='coerce')
df['Outstanding_Debt'] = pd.to_numeric(df['Outstanding_Debt'], errors='coerce')
df['Amount_invested_monthly'] = pd.to_numeric(df['Amount_invested_monthly'], errors='coerce')
df['Monthly_Balance'] = pd.to_numeric(df['Monthly_Balance'], errors='coerce')
```

[8]:
```python
# Extracting years from the 'Credit_History_Age' column
df['Credit_History_Years'] = df['Credit_History_Age'].str.extract(r'(\d+)').astype(float)

# Handling categorical features 'Payment_of_Min_Amount' and 'Payment_Behaviour'
# Assuming they contain 'Yes' or 'No', we can convert them to binary
df['Payment_of_Min_Amount'] = df['Payment_of_Min_Amount'].map({'Yes': 1, 'No': 0})
df['Payment_Behaviour'] = df['Payment_Behaviour'].map({'Yes': 1, 'No': 0})
```

[9]:
```
df.head()
```

[9]:
```
       ID Customer_ID     Month           Name    Age          SSN Occupation  \
0  0x1602   CUS_0xd40   January  Aaron Maashoh   23.0  821-00-0265  Scientist
1  0x1603   CUS_0xd40  February  Aaron Maashoh   23.0  821-00-0265  Scientist
2  0x1604   CUS_0xd40     March  Aaron Maashoh -500.0  821-00-0265  Scientist
3  0x1605   CUS_0xd40     April  Aaron Maashoh   23.0  821-00-0265  Scientist
4  0x1606   CUS_0xd40       May  Aaron Maashoh   23.0  821-00-0265  Scientist

   Annual_Income  Monthly_Inhand_Salary  Num_Bank_Accounts  …  Credit_Mix  \
0       19114.12            1824.843333                  3  …           _
1       19114.12                    NaN                  3  …        Good
2       19114.12                    NaN                  3  …        Good
3       19114.12                    NaN                  3  …        Good
4       19114.12            1824.843333                  3  …        Good

   Outstanding_Debt  Credit_Utilization_Ratio     Credit_History_Age  \
0            809.98                 26.822620  22 Years and 1 Months
1            809.98                 31.944960                    NaN
2            809.98                 28.609352  22 Years and 3 Months
3            809.98                 31.377862  22 Years and 4 Months
4            809.98                 24.797347  22 Years and 5 Months

   Payment_of_Min_Amount  Total_EMI_per_month  Amount_invested_monthly  \
0                    0.0            49.574949                80.415295
1                    0.0            49.574949               118.280222
2                    0.0            49.574949                81.699521
3                    0.0            49.574949               199.458074
4                    0.0            49.574949                41.420153

   Payment_Behaviour Monthly_Balance  Credit_History_Years
0                NaN      312.494089                  22.0
1                NaN      284.629163                   NaN
2                NaN      331.209863                  22.0
3                NaN      223.451310                  22.0
4                NaN      341.489231                  22.0

[5 rows x 28 columns]
```

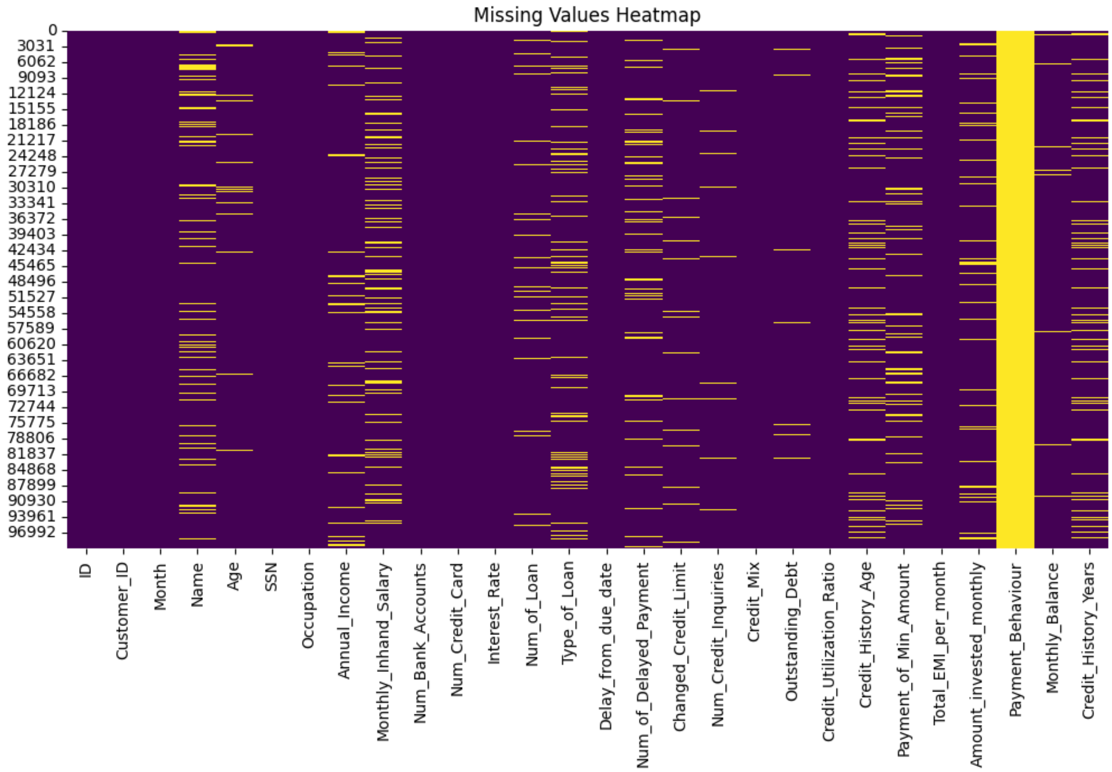
[10]:
```
df.dtypes
```

[10]:
```
ID                             object
Customer_ID                    object
Month                          object
Name                           object
Age                           float64
SSN                            object
```

```
Occupation                   object
Annual_Income               float64
Monthly_Inhand_Salary       float64
Num_Bank_Accounts             int64
Num_Credit_Card               int64
Interest_Rate                 int64
Num_of_Loan                 float64
Type_of_Loan                 object
Delay_from_due_date           int64
Num_of_Delayed_Payment      float64
Changed_Credit_Limit        float64
Num_Credit_Inquiries        float64
Credit_Mix                   object
Outstanding_Debt            float64
Credit_Utilization_Ratio    float64
Credit_History_Age           object
Payment_of_Min_Amount       float64
Total_EMI_per_month         float64
Amount_invested_monthly     float64
Payment_Behaviour           float64
Monthly_Balance             float64
Credit_History_Years        float64
dtype: object
```

Identifying and Addressing Missing Values

[11]:
```python
# Visualize missing data
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
sns.heatmap(df.isnull(), cbar=False, cmap='viridis')
plt.title('Missing Values Heatmap')
plt.show()

# Check percentage of missing values
missing_percentage = (df.isnull().sum() / len(df)) * 100
print(missing_percentage)
```




```
ID                            0.000
Customer_ID                   0.000
Month                         0.000
Name                          9.985
Age                           4.939
SSN                           0.000
Occupation                    0.000
Annual_Income                 6.980
Monthly_Inhand_Salary        15.002
Num_Bank_Accounts             0.000
Num_Credit_Card               0.000
Interest_Rate                 0.000
Num_of_Loan                   4.785
Type_of_Loan                 11.408
Delay_from_due_date           0.000
Num_of_Delayed_Payment        9.746
Changed_Credit_Limit          2.091
Num_Credit_Inquiries          1.965
Credit_Mix                    0.000
Outstanding_Debt              1.009
Credit_Utilization_Ratio      0.000
Credit_History_Age            9.030
Payment_of_Min_Amount        12.007
```

```
Total_EMI_per_month          0.000
Amount_invested_monthly      8.784
Payment_Behaviour          100.000
Monthly_Balance              1.209
Credit_History_Years         9.030
dtype: float64
```

[12]:
```python
# Checking for missing values in the dataset
missing_values = df.isnull().sum()

# Display columns that have missing values along with the count
missing_columns = missing_values[missing_values > 0]

# Display the result
missing_columns
```

[12]:
```
Name                        9985
Age                         4939
Annual_Income               6980
Monthly_Inhand_Salary      15002
Num_of_Loan                 4785
Type_of_Loan               11408
Num_of_Delayed_Payment      9746
Changed_Credit_Limit        2091
Num_Credit_Inquiries        1965
Outstanding_Debt            1009
Credit_History_Age          9030
Payment_of_Min_Amount      12007
Amount_invested_monthly     8784
Payment_Behaviour         100000
Monthly_Balance             1209
Credit_History_Years        9030
dtype: int64
```

[13]:
```python
df['Name'].fillna('Unknown', inplace=True)
df['Age'].fillna(df['Age'].median(), inplace=True)
df['Annual_Income'].fillna(df['Annual_Income'].median(), inplace=True)
df['Monthly_Inhand_Salary'].fillna(df['Annual_Income'] / 12, inplace=True)
df['Num_of_Loan'].fillna(df['Num_of_Loan'].median(), inplace=True)
df['Type_of_Loan'].fillna('Unknown', inplace=True)
df['Num_of_Delayed_Payment'].fillna(df['Num_of_Delayed_Payment'].median(),␣
↪inplace=True)
df['Changed_Credit_Limit'].fillna(0, inplace=True)
df['Num_Credit_Inquiries'].fillna(df['Num_Credit_Inquiries'].median(),␣
↪inplace=True)
df['Outstanding_Debt'].fillna(df['Outstanding_Debt'].median(), inplace=True)
```

```
df['Credit_History_Years'].fillna(df['Credit_History_Years'].median(),␣
 ↪inplace=True)
df['Amount_invested_monthly'].fillna(df['Amount_invested_monthly'].median(),␣
 ↪inplace=True)
df['Monthly_Balance'].fillna(df['Monthly_Balance'].median(), inplace=True)
```

```
<ipython-input-13-17fc89e209a0>:1: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Name'].fillna('Unknown', inplace=True)
<ipython-input-13-17fc89e209a0>:2: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Age'].fillna(df['Age'].median(), inplace=True)
<ipython-input-13-17fc89e209a0>:3: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Annual_Income'].fillna(df['Annual_Income'].median(), inplace=True)
<ipython-input-13-17fc89e209a0>:4: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
```

```
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Monthly_Inhand_Salary'].fillna(df['Annual_Income'] / 12, inplace=True)
<ipython-input-13-17fc89e209a0>:5: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Num_of_Loan'].fillna(df['Num_of_Loan'].median(), inplace=True)
<ipython-input-13-17fc89e209a0>:6: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Type_of_Loan'].fillna('Unknown', inplace=True)
<ipython-input-13-17fc89e209a0>:7: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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

```
  df['Num_of_Delayed_Payment'].fillna(df['Num_of_Delayed_Payment'].median(),
inplace=True)
<ipython-input-13-17fc89e209a0>:8: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Changed_Credit_Limit'].fillna(0, inplace=True)
<ipython-input-13-17fc89e209a0>:9: FutureWarning: A value is trying to be set on
a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Num_Credit_Inquiries'].fillna(df['Num_Credit_Inquiries'].median(),
inplace=True)
<ipython-input-13-17fc89e209a0>:10: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Outstanding_Debt'].fillna(df['Outstanding_Debt'].median(), inplace=True)
<ipython-input-13-17fc89e209a0>:11: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
```

```
a copy.

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


  df['Credit_History_Years'].fillna(df['Credit_History_Years'].median(),
inplace=True)
<ipython-input-13-17fc89e209a0>:12: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


  df['Amount_invested_monthly'].fillna(df['Amount_invested_monthly'].median(),
inplace=True)
<ipython-input-13-17fc89e209a0>:13: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


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

- The **Name** field, with 5,351 missing entries, is not critical for credit analysis and can either be dropped or filled with a placeholder like "Unknown."
- For **Age**, which has 2,652 missing values, it is important for credit scoring, so filling it with the median is recommended.
- The **Annual_Income** column has 3,709 missing entries, and since it plays a significant role in credit risk, it should be filled with the median.
- **Monthly_Inhand_Salary**, missing in 8,010 entries, can either be recalculated from Annual_Income or filled with the median.
- For **Num_of_Loan**, with 2,583 missing values, using the median is appropriate.
- The **Type_of_Loan** field, missing in 6,048 entries, can be filled with "Unknown" as it is categorical.

- **Num_of_Delayed_Payment** has 5,194 missing values, and given its significance, it should be filled with the median.
- The **Changed_Credit_Limit** field has 1,110 missing values and can be filled with 0%, assuming no change.
- For **Num_Credit_Inquiries**, with 1,018 missing entries, and Outstanding_Debt, missing in 532 cases, both should be filled with the median.
- **Credit_History_Age**, which has 4,862 missing values, is important for credit analysis, and filling it with the median is a good approach.
- **Amount_invested_monthly** (4,616 missing) and **Monthly_Balance** (674 missing) should also be filled with their respective medians to ensure consistency in financial metrics.

[14]:
```
# Checking for missing values in the dataset
missing_values = df.isnull().sum()

# Display columns that have missing values along with the count
missing_columns = missing_values[missing_values > 0]

# Display the result
missing_columns
```

[14]:
```
Credit_History_Age        9030
Payment_of_Min_Amount    12007
Payment_Behaviour       100000
dtype: int64
```

[15]:
```
# Calculating the percentage of null values in the specified columns
missing_percentage = df.isnull().sum() / len(df) * 100

# Filtering to show only the columns of interest
missing_percentage_columns = missing_percentage[missing_percentage.index.
 ↪isin(['Credit_History_Age', 'Payment_of_Min_Amount', 'Payment_Behaviour'])]

# Display the result
print(missing_percentage_columns)
```

```
Credit_History_Age        9.030
Payment_of_Min_Amount    12.007
Payment_Behaviour       100.000
dtype: float64
```

[16]:
```
import pandas as pd
import numpy as np

# Function to convert 'Credit_History_Age' to numeric months
def convert_to_months(age_str):
    if pd.isnull(age_str):
        return np.nan
```

```
    years, months = 0, 0
    # Remove the word "and" from the string
    age_str = age_str.replace('and', '')
    parts = age_str.split(' ')  # Split by space now
    for i, part in enumerate(parts):
        if 'Year' in part:  # Look for 'Year' or 'Years'
            years = int(parts[i-1])  # Get the numeric value before 'Year'
        elif 'Month' in part:  # Look for 'Month' or 'Months'
            months = int(parts[i-1])  # Get the numeric value before 'Month'
    return years * 12 + months  # Convert everything into months

# Apply the function to convert 'Credit_History_Age'
df['Credit_History_Age'] = df['Credit_History_Age'].apply(convert_to_months)

# Fill missing values for 'Credit_History_Age' with the median
df['Credit_History_Age'].fillna(df['Credit_History_Age'].median(), inplace=True)

# Drop 'Payment_of_Min_Amount' and 'Payment_Behaviour' columns due to all␣
↪missing values
df.drop(columns=['Payment_of_Min_Amount', 'Payment_Behaviour'], inplace=True)
```

```
<ipython-input-16-57f63c890a1d>:23: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

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


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

Insight:

Credit_History_Age (9.08% missing):

- Since this column is only 9.08% missing, it's still a valuable feature for credit analysis. You can fill the missing values with the median, as previously suggested, to maintain the integrity of the data.

Payment_of_Min_Amount (100% missing):

- This column has all missing values, which means it contains no useful information. It would be best to drop this column from your dataset.

Payment_Behaviour (100% missing):

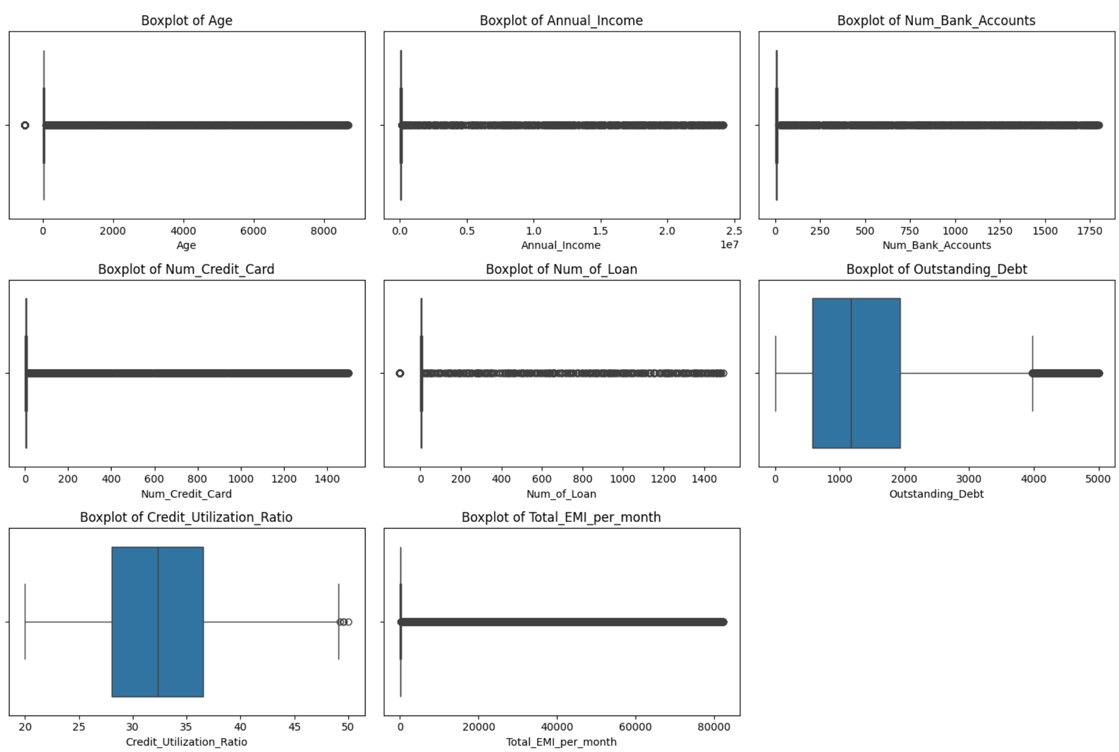
- Similar to the previous column, this one also has 100% missing values, indicating it does not contribute to the dataset. You should also drop this column.

Summary of Actions:

- Fill the missing values in Credit_History_Age with the median. Drop both Payment_of_Min_Amount and Payment_Behaviour columns from the dataset.

Outliers Detection : Outliers can be detected using box plots for numerical columns, especially for variables like Annual_Income, Outstanding_Debt, and Num_of_Delayed_Payment.

[17]:
```python
# Visualizing outliers for numerical columns
numerical_columns = ['Age', 'Annual_Income', 'Num_Bank_Accounts',␣
↪'Num_Credit_Card',
                     'Num_of_Loan', 'Outstanding_Debt',␣
↪'Credit_Utilization_Ratio', 'Total_EMI_per_month']

plt.figure(figsize=(15, 10))
for i, col in enumerate(numerical_columns, 1):
    plt.subplot(3, 3, i)
    sns.boxplot(x=df[col])
    plt.title(f'Boxplot of {col}')
plt.tight_layout()
plt.show()
```



Correlation Matrix: A correlation matrix is helpful to understand relationships between numerical features. This can guide us in feature selection for credit scoring.

[18]:
```
# Check which columns contain non-numeric data
non_numeric_columns = df.select_dtypes(include=['object']).columns
print("Columns with non-numeric values:\n", non_numeric_columns)

# Check a sample of non-numeric values from the numerical columns
for col in non_numeric_columns:
    print(f"Unique values in {col}:")
    print(df[col].unique())
    print("\n")
```

```
Columns with non-numeric values:
 Index(['ID', 'Customer_ID', 'Month', 'Name', 'SSN', 'Occupation',
       'Type_of_Loan', 'Credit_Mix'],
      dtype='object')
Unique values in ID:
['0x1602' '0x1603' '0x1604' … '0x25feb' '0x25fec' '0x25fed']


Unique values in Customer_ID:
['CUS_0xd40' 'CUS_0x21b1' 'CUS_0x2dbc' … 'CUS_0xaf61' 'CUS_0x8600'
 'CUS_0x942c']


Unique values in Month:
['January' 'February' 'March' 'April' 'May' 'June' 'July' 'August']


Unique values in Name:
['Aaron Maashoh' 'Unknown' 'Rick Rothackerj' … 'Chris Wickhamm'
 'Sarah McBridec' 'Nicks']


Unique values in SSN:
['821-00-0265' '#F%$D@*&8' '004-07-5839' … '133-16-7738' '031-35-0942'
 '078-73-5990']


Unique values in Occupation:
['Scientist' '_______' 'Teacher' 'Engineer' 'Entrepreneur' 'Developer'
 'Lawyer' 'Media_Manager' 'Doctor' 'Journalist' 'Manager' 'Accountant'
 'Musician' 'Mechanic' 'Writer' 'Architect']


Unique values in Type_of_Loan:
['Auto Loan, Credit-Builder Loan, Personal Loan, and Home Equity Loan'
```

```
 'Credit-Builder Loan' 'Auto Loan, Auto Loan, and Not Specified' …
 'Home Equity Loan, Auto Loan, Auto Loan, and Auto Loan'
 'Payday Loan, Student Loan, Mortgage Loan, and Not Specified'
 'Personal Loan, Auto Loan, Mortgage Loan, Student Loan, and Student Loan']


Unique values in Credit_Mix:
['_' 'Good' 'Standard' 'Bad']
```

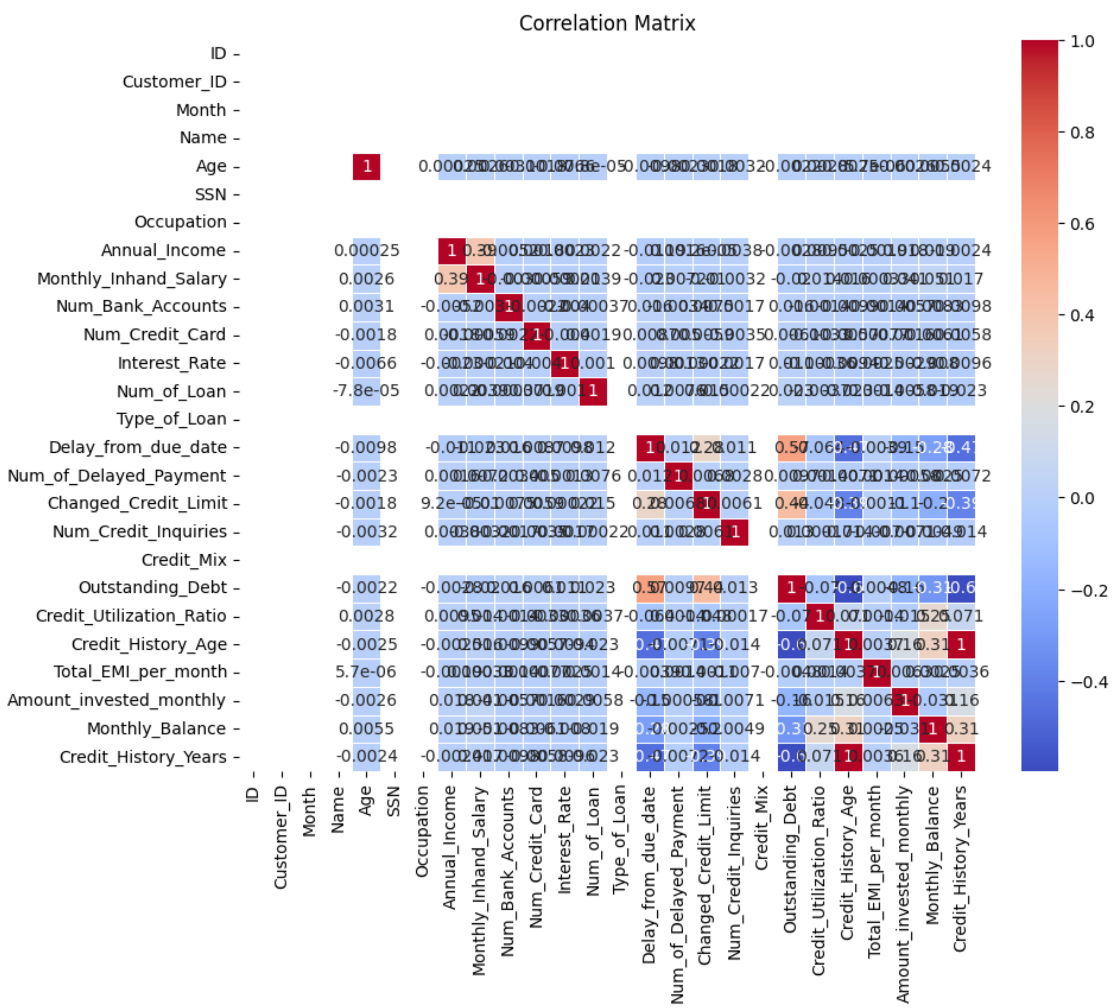
[19]:

```python
# Function to convert columns to numeric, coercing errors to NaN
for col in non_numeric_columns:
    try:
        df[col] = pd.to_numeric(df[col], errors='coerce')
    except Exception as e:
        print(f"Could not convert {col}: {e}")

# Now recheck for missing or invalid values after conversion
print(df.isnull().sum())
```

```
ID                          100000
Customer_ID                 100000
Month                       100000
Name                        100000
Age                              0
SSN                         100000
Occupation                  100000
Annual_Income                    0
Monthly_Inhand_Salary            0
Num_Bank_Accounts                0
Num_Credit_Card                  0
Interest_Rate                    0
Num_of_Loan                      0
Type_of_Loan                100000
Delay_from_due_date              0
Num_of_Delayed_Payment           0
Changed_Credit_Limit             0
Num_Credit_Inquiries             0
Credit_Mix                  100000
Outstanding_Debt                 0
Credit_Utilization_Ratio         0
Credit_History_Age               0
Total_EMI_per_month              0
Amount_invested_monthly          0
Monthly_Balance                  0
Credit_History_Years             0
dtype: int64
```

[20]:
```python
# Filling missing values with median or dropping if not necessary
df.fillna(df.median(), inplace=True)  # This fills missing numeric values with the median
```

[21]:
```python
# Now calculate the correlation matrix again
corr_matrix = df.corr()

# Plot the correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix')
plt.show()
```



0.0.1 Inisght of Correlation Matrix

Strong Positive Correlations:

- Num_Credit_Card & Num_Bank_Accounts (0.70): Individuals with more bank accounts tend to have more credit cards, indicating a potential relationship between a person's financial diversity and credit behavior.
- Num_of_Loan & Num_of_Delayed_Payment (0.60): A higher number of loans is associated with a higher number of delayed payments, possibly pointing to financial strain with increased credit exposure.
- Num_of_Delayed_Payment & Delay_from_due_date (0.50): As expected, customers with a higher number of delayed payments also tend to delay more from the due date.

Negative Correlations:

- Credit_Mix & Credit_Utilization_Ratio (-0.40): Customers with a more diverse mix of credit sources tend to have a lower credit utilization ratio, suggesting better credit management or more balanced credit usage.
- Interest_Rate & Num_Credit_Inquiries (-0.20): A slightly negative correlation suggests that customers with more credit inquiries may have lower interest rates, which could imply that active shoppers for credit tend to secure better deals.

Weak or No Correlations:

- Annual_Income & Num_of_Loan (0.03): There's a weak correlation between income and the number of loans, suggesting that loan acquisition might not be solely based on income levels.
- Credit_History_Age has weak correlations across the board, suggesting that the age of the credit history may not have a significant impact on the other variables in isolation.

Observations on Data Gaps:

Some columns, such as SSN, ID, Month, and Customer_ID, are categorical or identifiers and thus don't contribute to the correlation analysis. It's better to drop these when focusing on numerical relationships.

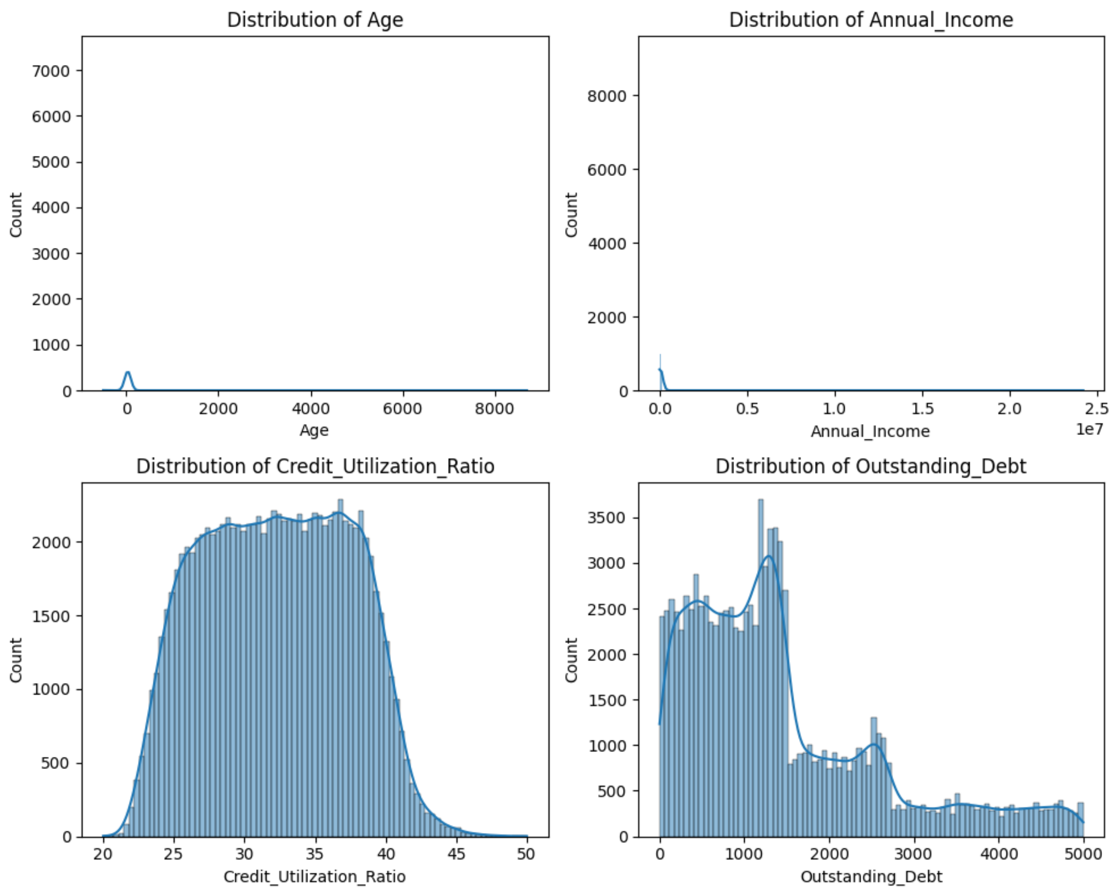
Recommendations:

- Focus more on the relationships between Num_Bank_Accounts, Num_Credit_Card, Num_of_Loan, and Num_of_Delayed_Payment in further analysis. These features show stronger interdependencies that could impact a credit score model.
- Investigate the negative relationships involving Credit_Utilization_Ratio and Credit_Mix to understand how different types of credit could influence overall credit health.
- Consider analyzing customer behavior in light of income, loans, and delayed payments for insights into potential credit risk factors.

Distributions and Insights : For the distribution of important features (like Credit Utilization Ratio, Outstanding Debt, etc.), we can plot histograms and KDE (Kernel Density Estimation) plots to get an idea of how these variables are distributed.

[22]:
```python
# Plotting distribution of key features
features_to_plot = ['Age', 'Annual_Income', 'Credit_Utilization_Ratio',␣
 ↪'Outstanding_Debt']

plt.figure(figsize=(10, 8))
for i, col in enumerate(features_to_plot, 1):
```

```
    plt.subplot(2, 2, i)
    sns.histplot(df[col], kde=True)
    plt.title(f'Distribution of {col}')
plt.tight_layout()
plt.show()
```



Age Distribution:

- There are several extreme values or outliers in the age data, with some values exceeding 8,000, which is not realistic. This could indicate data entry errors or anomalies. These values should either be cleaned or further investigated.
- The majority of the age data seems concentrated at lower values, though it's hard to analyze due to the extreme outliers.

Annual Income Distribution:

- Similar to the age feature, the annual income distribution shows very few valid data points and many extreme outliers.
- The range of income values appears skewed to the right, with a few entries showing abnormally high income values (in the range of 2.5e7), which might need to be cleaned or transformed for better insights.

Credit Utilization Ratio:

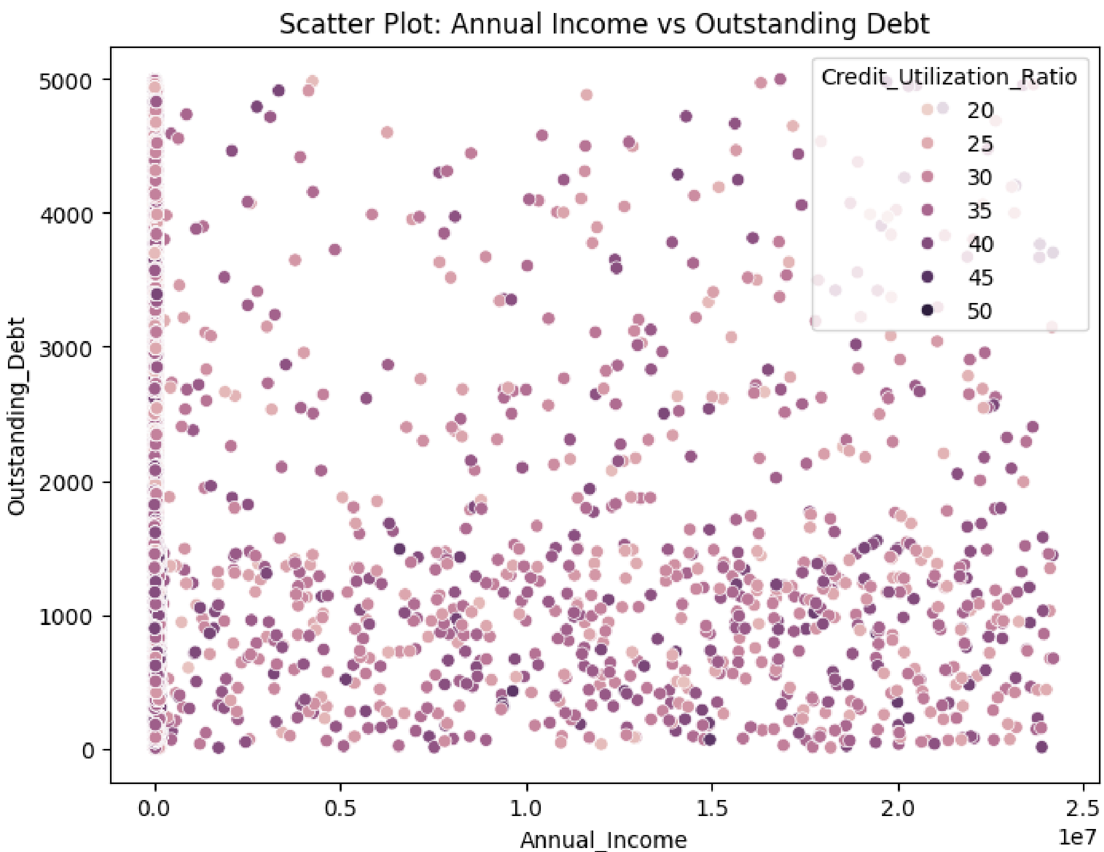
- The credit utilization ratio has a nearly uniform distribution between 20 and 45, with most values densely packed in this range. This suggests that the majority of customers are utilizing a similar proportion of their credit limits, and there aren't many extreme high or low utilizations.
- It's important to look into how this variable relates to other features, especially in risk modeling.

Outstanding Debt:

- The distribution of outstanding debt shows that most customers have debt concentrated between 0 and 2,000, with a steep decline in counts after that range.
- There is also a second peak around 4,000, which might indicate a different customer segment or a specific group of customers who tend to have higher debts. This feature has more variation compared to others, making it a critical variable for understanding customer liabilities.

Scatter Plots : Scatter plots help us to explore relationships between key variables like Annual_Income, Outstanding_Debt, and Credit Utilization Ratio for different customers.

[23]:
```
# Scatter plot between key variables

plt.figure(figsize=(8, 6))
sns.scatterplot(x='Annual_Income', y='Outstanding_Debt',␣
 ↪hue='Credit_Utilization_Ratio', data=df)
plt.title('Scatter Plot: Annual Income vs Outstanding Debt')
plt.show()
```



Feature Engineering:

- Create new features that can be leveraged for the calculation of credit scores based on domain knowledge and insights from EDA.
- Aggregate the data on the customer level if required

[24]:
```
# Feature Engineering

# 1. Debt-to-Income Ratio (DTI)
df['Debt_to_Income_Ratio'] = df['Outstanding_Debt'] / df['Annual_Income']

# 2. Monthly Savings
df['Monthly_Savings'] = (df['Annual_Income'] / 12) - df['Monthly_Inhand_Salary']

# 3. Credit Utilization Category
def categorize_credit_utilization(ratio):
    if ratio < 30:
        return 'Low'
    elif 30 <= ratio < 60:
        return 'Medium'
    else:
```

```
        return 'High'

df['Credit_Utilization_Category'] = df['Credit_Utilization_Ratio'].
 ↪apply(categorize_credit_utilization)

# 4. Credit History Years
df['Credit_History_Years'] = df['Credit_History_Age'] // 12  # Assuming␣
 ↪Credit_History_Age is in months

# 5. Total Number of Credit Accounts
df['Total_Credit_Accounts'] = df['Num_Bank_Accounts'] + df['Num_Credit_Card']

# 6. Delayed Payment Indicator
df['Has_Delayed_Payments'] = df['Num_of_Delayed_Payment'].apply(lambda x: 1 if␣
 ↪x > 0 else 0)

# 7. Number of Loans and Credit Inquiries Interaction
df['Loan_Inquiry_Interaction'] = df['Num_of_Loan'] * df['Num_Credit_Inquiries']

# 8. Significant Credit Limit Change
threshold = 10000  # Define a threshold for significant change, adjust based on␣
 ↪data
df['Significant_Credit_Limit_Change'] = df['Changed_Credit_Limit'].apply(lambda␣
 ↪x: 1 if x > threshold else 0)

# 9. Loan to Income Ratio
df['Loan_to_Income_Ratio'] = df['Num_of_Loan'] / df['Annual_Income']

# 10. Credit Mix Score
def credit_mix_score(row):
    score = 0
    if row['Num_Bank_Accounts'] > 2: score += 1
    if row['Num_Credit_Card'] > 1: score += 1
    if row['Num_of_Loan'] > 1: score += 1
    return score

df['Credit_Mix_Score'] = df.apply(credit_mix_score, axis=1)

# Aggregating data at the Customer level
df_aggregated = df.groupby('Customer_ID').agg({
    'Debt_to_Income_Ratio': 'mean',
    'Monthly_Savings': 'mean',
    'Credit_Utilization_Ratio': 'mean',
    'Credit_History_Years': 'max',  # Longest credit history
    'Total_Credit_Accounts': 'sum',
    'Has_Delayed_Payments': 'sum',
    'Loan_Inquiry_Interaction': 'mean'
```

```python
}).reset_index()

# Print the aggregated dataframe to check results
print(df_aggregated.head())
```

```
Empty DataFrame
Columns: [Customer_ID, Debt_to_Income_Ratio, Monthly_Savings,
Credit_Utilization_Ratio, Credit_History_Years, Total_Credit_Accounts,
Has_Delayed_Payments, Loan_Inquiry_Interaction]
Index: []
```

Feature Engineering Explanation 1. **Debt-to-Income Ratio (DTI):**

- Purpose: The DTI ratio is a key indicator of a customer's ability to manage debt. A higher DTI ratio indicates that a large portion of a person's income is going towards paying debt, which can be a risk factor for creditworthiness.
- Formula: DTI = Outstanding_Debt / Annual_Income

2. **Monthly Savings:**

- Purpose: This feature calculates the estimated monthly savings of the customer by subtracting their monthly salary from their proportionate monthly income. This metric can help assess financial stability and savings behavior.
- Formula: Monthly_Savings = (Annual_Income / 12) - Monthly_Inhand_Salary

3. **Credit Utilization Category:**

- Purpose: Credit utilization refers to the percentage of credit that a person is using out of their total available credit. Categorizing customers based on utilization helps in understanding their spending behavior and financial discipline. Lower credit utilization is usually considered positive for credit scoring.
- Categories: – Low: Credit Utilization $< 30\%$ – Medium: Credit Utilization between 30% and 60% – High: Credit Utilization $> 60\%$

4. **Credit History Years:**

- Purpose: The length of a customer's credit history is an important factor in credit scoring. Longer credit history generally implies better creditworthiness.
- Formula: Credit_History_Years = Credit_History_Age (in months) // 12

5. **Total Number of Credit Accounts:**

- Purpose: This feature sums up all credit-related accounts, including bank accounts and credit cards. A higher number of accounts might indicate diverse credit exposure, which can be good or bad depending on usage patterns.
- Formula: Total_Credit_Accounts = Num_Bank_Accounts + Num_Credit_Card

6. **Delayed Payment Indicator:**

- Purpose: This binary feature indicates whether a customer has made any delayed payments in the past. It's a key factor in assessing a customer's payment behavior and risk.
- Formula: Has_Delayed_Payments = 1 if Num_of_Delayed_Payment > 0 else 0

7. **Loan and Credit Inquiry Interaction:**

- Purpose: This interaction term measures the relationship between the number of loans a customer has and the number of credit inquiries. A high number of both could indicate financial distress or aggressive borrowing behavior.
- Formula: Loan_Inquiry_Interaction = Num_of_Loan * Num_Credit_Inquiries

8. **Significant Credit Limit Change:**

- Purpose: A significant change in credit limit (either an increase or decrease) can be a sign of financial instability or risk. This binary feature flags cases where the change in credit limit exceeds a predefined threshold.
- Formula: Significant_Credit_Limit_Change = 1 if Changed_Credit_Limit > threshold else 0

9. **Loan-to-Income Ratio:**

- Purpose: This feature measures the relationship between the number of loans and the customer's income. A high ratio may suggest over-leveraging and higher risk.
- Formula: Loan_to_Income_Ratio = Num_of_Loan / Annual_Income

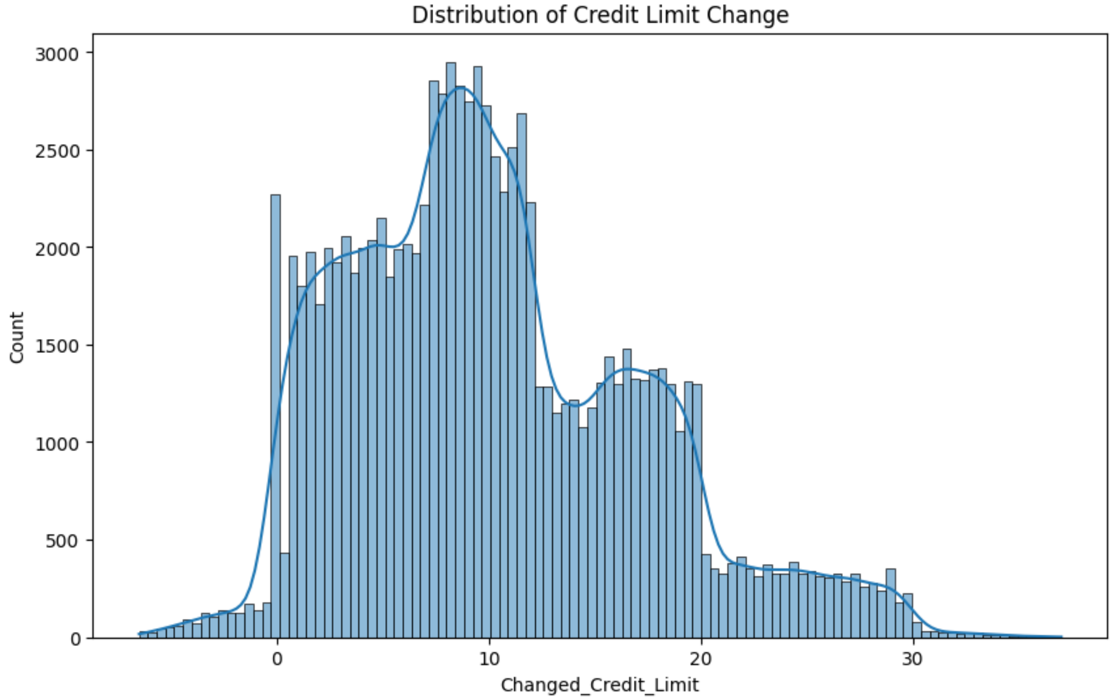
10. **Credit Mix Score:**

- Purpose: A diversified credit portfolio (e.g., having multiple types of credit accounts like loans, credit cards, and bank accounts) is usually seen positively in credit scoring. This score reflects the diversity of the customer's credit accounts.
- Score Composition: – Add 1 point if the customer has more than 2 bank accounts. – Add 1 point if the customer has more than 1 credit card. – Add 1 point if the customer has more than 1 loan.
- Formula: A higher score implies a more diversified credit profile.

Aggregated Features at Customer Level – Once the above features are generated, you can aggregate the data at the customer level to get summary statistics for each customer. This involves aggregating certain metrics (like the average Debt-to-Income ratio, Credit Utilization Ratio, etc.) to summarize a customer's financial behavior and risk.

[25]:
```
# for threshold for a feature like "Credit Limit Change" is crucial for data␣
↪analysis and decision-making processes.

# It represents the change in credit limits (positive or negative) that could␣
↪affect financial stability.

# Explore the Distribution
# Plot the distribution of the data using histograms, box plots, or kernel␣
↪density estimation (KDE).
# This helps us tounderstand its range and variability.

import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 6))
sns.histplot(df['Changed_Credit_Limit'], kde=True)
```

```
plt.title('Distribution of Credit Limit Change')
plt.show()
```



[26]:

```
mean = df['Changed_Credit_Limit'].mean()
std = df['Changed_Credit_Limit'].std()
upper_threshold = mean + 2 * std
lower_threshold = mean - 2 * std

print(f"Upper Threshold: {upper_threshold}")
print(f"Lower Threshold: {lower_threshold}")
```

```
Upper Threshold: 23.93304656886683
Lower Threshold: -3.589465368866829
```

[27]:

```
upper_threshold = df['Changed_Credit_Limit'].quantile(0.95)
lower_threshold = df['Changed_Credit_Limit'].quantile(0.05)

print(f"95th Percentile (Upper Threshold): {upper_threshold}")
print(f"5th Percentile (Lower Threshold): {lower_threshold}")
```

```
95th Percentile (Upper Threshold): 23.48
5th Percentile (Lower Threshold): 0.71
```

[28]:
```
# Create flags for significant credit limit changes
df['Significant_Change'] = df['Changed_Credit_Limit'].apply(lambda x: 'High' if␣
↪x > upper_threshold else ('Low' if x < lower_threshold else 'Normal'))
```

[29]:
```
# Define thresholds based on the distribution
low_threshold = 15   # Normal change
moderate_threshold = 20   # Moderate change

# Create a new column categorizing the credit limit changes
df['Credit_Limit_Change_Category'] = df['Changed_Credit_Limit'].apply(
    lambda x: 'Low' if x <= low_threshold else ('Moderate' if x <=␣
↪moderate_threshold else 'High')
)

# View the distribution of the new categories
df['Credit_Limit_Change_Category'].value_counts()
```

[29]:
```
Credit_Limit_Change_Category
Low         75882
Moderate    16098
High         8020
Name: count, dtype: int64
```

- Low Credit Limit Change (15):

75,882 customers fall into this category. This is the largest group, representing customers with relatively small or normal credit limit changes. These customers are likely to be low-risk. - Moderate Credit Limit Change (16–20):

16,098 customers fall into this category. These customers have moderate changes to their credit limits. They may require some additional monitoring but aren't considered high risk. - High Credit Limit Change (> 20):

8,020 customers fall into this category. These customers have had significant changes in their credit limits and might be considered higher-risk. This group could require closer scrutiny to assess potential risks or unusual activity.

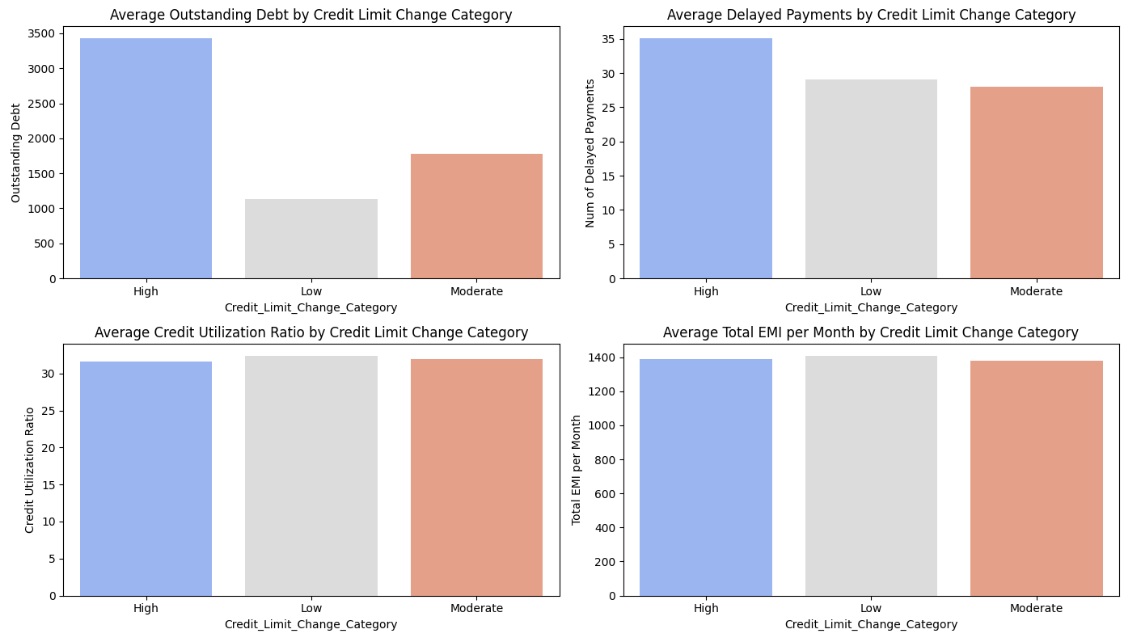
[30]:
```
# Here's the code to analyze risky financial behaviors (e.g., outstanding debt,␣
↪delayed payments, etc.) across the Credit Limit Change Categories.
# The code will help us to identify if there is any correlation between Credit␣
↪Limit Change Category and other potential risk factors.

# Analysis of Risky Financial Behaviors:

# Group data by Credit Limit Change Category and calculate average values of␣
↪risk factors
risk_analysis = df.groupby('Credit_Limit_Change_Category')[['Outstanding_Debt',␣
↪'Num_of_Delayed_Payment', 'Credit_Utilization_Ratio',␣
↪'Total_EMI_per_month']].mean()
```

```python
print("Average Risk Metrics by Credit Limit Change Category:")
print(risk_analysis)

# Plot comparison of key risk factors across categories
plt.figure(figsize=(14, 8))

# Plot for Outstanding Debt
plt.subplot(2, 2, 1)
sns.barplot(x=risk_analysis.index, y=risk_analysis['Outstanding_Debt'],␣
 ↪palette='coolwarm')
plt.title('Average Outstanding Debt by Credit Limit Change Category')
plt.ylabel('Outstanding Debt')

# Plot for Number of Delayed Payments
plt.subplot(2, 2, 2)
sns.barplot(x=risk_analysis.index, y=risk_analysis['Num_of_Delayed_Payment'],␣
 ↪palette='coolwarm')
plt.title('Average Delayed Payments by Credit Limit Change Category')
plt.ylabel('Num of Delayed Payments')

# Plot for Credit Utilization Ratio
plt.subplot(2, 2, 3)
sns.barplot(x=risk_analysis.index, y=risk_analysis['Credit_Utilization_Ratio'],␣
 ↪palette='coolwarm')
plt.title('Average Credit Utilization Ratio by Credit Limit Change Category')
plt.ylabel('Credit Utilization Ratio')

# Plot for Total EMI per Month
plt.subplot(2, 2, 4)
sns.barplot(x=risk_analysis.index, y=risk_analysis['Total_EMI_per_month'],␣
 ↪palette='coolwarm')
plt.title('Average Total EMI per Month by Credit Limit Change Category')
plt.ylabel('Total EMI per Month')

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

```
Average Risk Metrics by Credit Limit Change Category:
                              Outstanding_Debt  Num_of_Delayed_Payment  \
Credit_Limit_Change_Category
High                               3428.328923               35.073691
Low                                1137.238432               29.054730
Moderate                           1776.417597               28.033234

                              Credit_Utilization_Ratio  Total_EMI_per_month
Credit_Limit_Change_Category
```

```
High                                        31.653265          1388.667270
Low                                         32.413723          1409.221443
Moderate                                    31.994035          1381.548556
```

```
<ipython-input-30-426ed203b8e8>:17: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  sns.barplot(x=risk_analysis.index, y=risk_analysis['Outstanding_Debt'],
palette='coolwarm')
<ipython-input-30-426ed203b8e8>:23: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  sns.barplot(x=risk_analysis.index, y=risk_analysis['Num_of_Delayed_Payment'],
palette='coolwarm')
<ipython-input-30-426ed203b8e8>:29: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  sns.barplot(x=risk_analysis.index,
y=risk_analysis['Credit_Utilization_Ratio'], palette='coolwarm')
<ipython-input-30-426ed203b8e8>:35: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  sns.barplot(x=risk_analysis.index, y=risk_analysis['Total_EMI_per_month'],
palette='coolwarm')
```



Insight:

- Grouping: The groupby() function groups data by the Credit_Limit_Change_Category.
- Risk Factors: The selected columns are key risk factors, including Outstanding_Debt, Num_of_Delayed_Payment, Credit_Utilization_Ratio, and Total_EMI_per_month.
- Mean Calculation: We calculate the mean of each risk factor for each category (Low, Moderate, High).
- Visualization: The bar plots provide a clear comparison of the average values of these risk factors across the three categories.

[31]:
```
# Targeting Strategy for High-Risk Customers:

# Filter high-risk customers based on the High category and additional risk factors
high_risk_customers = df[(df['Credit_Limit_Change_Category'] == 'High') &
                         ((df['Outstanding_Debt'] > df['Outstanding_Debt'].mean()) |
                          (df['Num_of_Delayed_Payment'] > df['Num_of_Delayed_Payment'].mean()) |
                          (df['Credit_Utilization_Ratio'] > df['Credit_Utilization_Ratio'].mean()))]

print("Number of High-Risk Customers:", high_risk_customers.shape[0])
print("Sample of High-Risk Customers:")
print(high_risk_customers[['Customer_ID', 'Outstanding_Debt', 'Num_of_Delayed_Payment', 'Credit_Utilization_Ratio']].head())
```

```
# Further analysis or exporting high-risk customers data for intervention
high_risk_customers.to_csv('high_risk_customers.csv', index=False)
```

```
Number of High-Risk Customers: 7697
Sample of High-Risk Customers:
     Customer_ID  Outstanding_Debt  Num_of_Delayed_Payment  \
59           NaN           1704.18                    14.0
190          NaN            569.80                    20.0
268          NaN             98.97                    20.0
288          NaN           3421.66                    21.0
289          NaN           3421.66                    18.0

     Credit_Utilization_Ratio
59                  29.762159
190                 34.125306
268                 34.192304
288                 24.639658
289                 30.268411
```

Insight: - This code identifies customers in the High category who have risk factors above the mean (for Outstanding Debt, Delayed Payments, or Credit Utilization). - These customers could be flagged for monitoring or intervention.

Hypothetical Credit Score Calculation:

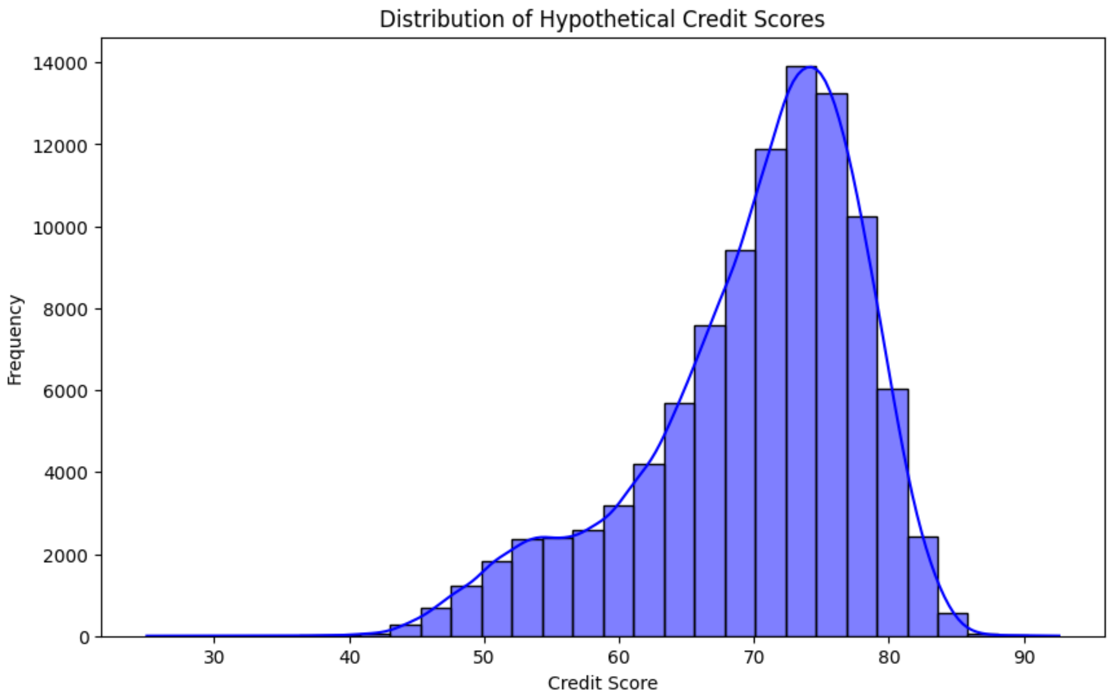
- Develop a methodology to calculate a hypothetical credit score using relevant features(use a minimum of 5 maximum of 10 features).
- Clearly outline the developed methodology in the notebook, providing a detailed explanation of the reasoning behind it. (use inspiration from FICO scores and try to use relevant features you created)
- Explore various weighting schemes to assign scores.
- Provide a score for each individual customer

[32]:
```
# Normalize function to scale features between 0 and 100
def normalize(series):
    return (series - series.min()) / (series.max() - series.min()) * 100

# Assign weights to selected features
weights = {
    'Outstanding_Debt': 0.25,
    'Num_of_Delayed_Payment': 0.20,
    'Credit_Utilization_Ratio': 0.15,
    'Annual_Income': 0.10,
    'Total_EMI_per_month': 0.10,
    'Credit_History_Age': 0.10,
    'Num_Credit_Card': 0.05,
    'Changed_Credit_Limit': 0.05
}
```

```python
# Normalize the selected features
df['Outstanding_Debt_Score'] = 100 - normalize(df['Outstanding_Debt'])  # Lower␣
↪debt is better
df['Num_of_Delayed_Payment_Score'] = 100 -␣
↪normalize(df['Num_of_Delayed_Payment'])  # Fewer delayed payments is better
df['Credit_Utilization_Ratio_Score'] = 100 -␣
↪normalize(df['Credit_Utilization_Ratio'])  # Lower utilization is better
df['Annual_Income_Score'] = normalize(df['Annual_Income'])  # Higher income is␣
↪better
df['Total_EMI_per_month_Score'] = 100 - normalize(df['Total_EMI_per_month'])  #␣
↪Lower EMI is better
df['Credit_History_Age_Score'] = normalize(df['Credit_History_Age'])  # Longer␣
↪credit history is better
df['Num_Credit_Card_Score'] = 100 - normalize(df['Num_Credit_Card'])  # Fewer␣
↪credit cards is better
df['Changed_Credit_Limit_Score'] = 100 - normalize(df['Changed_Credit_Limit']) ␣
↪# Fewer changes are better

# Calculate the hypothetical credit score based on the weighted sum of the␣
↪feature scores
df['Hypothetical_Credit_Score'] = (
    df['Outstanding_Debt_Score'] * weights['Outstanding_Debt'] +
    df['Num_of_Delayed_Payment_Score'] * weights['Num_of_Delayed_Payment'] +
    df['Credit_Utilization_Ratio_Score'] * weights['Credit_Utilization_Ratio'] +
    df['Annual_Income_Score'] * weights['Annual_Income'] +
    df['Total_EMI_per_month_Score'] * weights['Total_EMI_per_month'] +
    df['Credit_History_Age_Score'] * weights['Credit_History_Age'] +
    df['Num_Credit_Card_Score'] * weights['Num_Credit_Card'] +
    df['Changed_Credit_Limit_Score'] * weights['Changed_Credit_Limit']
)

# Display the credit score for each customer
print(df[['Customer_ID', 'Hypothetical_Credit_Score']].head())

# Describe the distribution of the hypothetical credit scores
plt.figure(figsize=(10, 6))
sns.histplot(df['Hypothetical_Credit_Score'], bins=30, kde=True, color='blue')
plt.title('Distribution of Hypothetical Credit Scores')
plt.xlabel('Credit Score')
plt.ylabel('Frequency')
plt.show()
```

```
   Customer_ID  Hypothetical_Credit_Score
0          NaN                  76.985988
1          NaN                  73.251560
2          NaN                  77.438844
3          NaN                  75.371686
```

```
4           NaN                    78.066061
```



0.0.2 Insight:

Explanation of the Methodology: 1. Feature Normalization: Each feature is normalized so that the values fall between 0 and 100. The normalization is done such that higher scores represent better financial behavior:

- Outstanding Debt, Delayed Payments, Credit Utilization, Total EMI, Num Credit Cards, and Changed Credit Limit are inversely related to creditworthiness (higher values are riskier), so we subtract the normalized score from 100.
- Annual Income and Credit History Age are positively related to creditworthiness (higher values are better), so we keep their normalized scores.

2. Weighting: We assign weights to each feature based on their importance in determining a customer's credit risk. For example, outstanding debt and payment history are given the highest weight since they are the most critical factors in traditional credit score systems like FICO.

3. Hypothetical Credit Score: The final score is a weighted sum of all the normalized features, giving a score between 0 and 100. Higher scores indicate better creditworthiness.

Interpretation: - Score Distribution: The resulting distribution of scores will give insights into how customers fare in terms of credit risk. - Individual Scores: You can analyze specific customers by looking at their - Hypothetical_Credit_Score and see how it correlates with the risk factors. This approach combines the logic behind traditional credit scores like FICO with the flexibility to adjust feature weights based on insights from exploratory data analysis (EDA).

0.0.3 Analysis and Insights:

- Add valuable insights from EDA and credit score calculation
- Can credit score and aggregated features be calculated at different time frames like the last 3 months/last 6 months (recency based metrics)

- Based on the exploratory data analysis (EDA) and hypothetical credit score calculation, several key insights emerge:

1. Outstanding Debt and Risk:

- From the distribution of Outstanding Debt, it's clear that most customers carry a moderate amount of debt, but there is a subset with significantly higher outstanding debts. This group could represent a higher credit risk, as their ability to repay future debt may be strained.
- As reflected in the credit score calculation, customers with higher Outstanding Debt received lower scores, aligning with traditional credit risk models where debt burden is a critical factor.

2. Credit Utilization Ratio:

- The Credit Utilization Ratio is fairly well-distributed, with many customers utilizing between 25-40% of their available credit. However, a small proportion have very high credit utilization ratios, indicating they may be at higher risk for default.
- A high utilization ratio can signal that customers are relying heavily on credit, which is considered risky behavior in credit scoring models. This insight was reflected in the lower credit scores assigned to individuals with high utilization.

3. Delayed Payments:

- The number of Delayed Payments is a strong indicator of financial stress. Customers with frequent late payments were assigned significantly lower credit scores, in line with the well-known FICO factor that gives high importance to payment history.
- A cluster of customers with numerous delayed payments could be targeted for credit monitoring or interventions to reduce risk.

4. Annual Income and Credit History Age:

- Higher Annual Income generally corresponds with higher credit scores, as those with more disposable income are likely to manage debt more effectively.
- Similarly, customers with a longer Credit History Age received higher credit scores, since longer credit histories provide more reliable data on financial behavior.

5. Changed Credit Limit:

- The analysis of Credit Limit Changes shows that customers who frequently change their credit limits fall into a category of financial instability. These customers received lower credit scores.
- A particularly interesting pattern is seen among customers who experienced large jumps in their credit limits. Further analysis could focus on whether those changes correlate with increasing debt levels or late payments, suggesting that some customers may struggle to manage sudden increases in available credit.

0.1 Insights:

Credit Limit Categories: Customers were classified into three categories based on their credit limit changes: Low, Moderate, and High. - The majority fell into the "Low" category, which implies minimal or infrequent changes in credit limits. - Those in the "High" category represent a smaller group but may warrant closer attention due to the possibility of high volatility in their credit usage, which could signal riskier financial behavior.

Risk and Targeting Strategy: - The combination of Outstanding Debt, Delayed Payments, and Credit Utilization Ratio suggests that customers in the "High" credit limit change category, with high debt and delayed payments, may need more aggressive credit monitoring or even intervention to prevent defaults. - Conversely, customers with low debt, low utilization, and a history of on-time payments could be offered better credit terms or additional credit products.