Credit Card Users Churn Prediction

Problem Statement

Business Context

The Thera bank recently saw a steep decline in the number of users of their credit card, credit cards are a good source of income for banks because of different kinds of fees charged by the banks like annual fees, balance transfer fees, and cash advance fees, late payment fees, foreign transaction fees, and others. Some fees are charged to every user irrespective of usage, while others are charged under specified circumstances.

Customers' leaving credit cards services would lead bank to loss, so the bank wants to analyze the data of customers and identify the customers who will leave their credit card services and reason for same – so that bank could improve upon those areas

You as a Data scientist at Thera bank need to come up with a classification model that will help the bank improve its services so that customers do not renounce their credit cards

Data Description

- CLIENTNUM: Client number. Unique identifier for the customer holding the account
- Attrition_Flag: Internal event (customer activity) variable - if the account is closed then "Attrited Customer" else "Existing Customer"
- Customer_Age: Age in Years
- Gender: Gender of the account holder
- Dependent_count: Number of dependents
- Education_Level: Educational Qualification of the account holder - Graduate, High School, Unknown, Uneducated, College(refers to college student), Post-Graduate, Doctorate
- Marital_Status: Marital Status of the account holder
- Income_Category: Annual Income Category of the account holder
- Card_Category: Type of Card
- Months_on_book: Period of relationship with the bank (in months)
- Total_Relationship_Count: Total no. of products held by the customer
- Months_Inactive_12_mon: No. of months inactive in the last 12 months
- Contacts_Count_12_mon: No. of Contacts in the last 12 months
- Credit_Limit: Credit Limit on the Credit Card
- Total_Revolving_Bal: Total Revolving Balance on the Credit Card
- Avg_Open_To_Buy: Open to Buy Credit Line (Average of last 12 months)
- Total_Amt_Chng_Q4_Q1: Change in Transaction Amount (Q4 over Q1)
- Total_Trans_Amt: Total Transaction Amount (Last 12 months)
- Total_Trans_Ct: Total Transaction Count (Last 12 months)
- Total_Ct_Chng_Q4_Q1: Change in Transaction Count (Q4 over Q1)

- Avg_Utilization_Ratio: Average Card Utilization Ratio

What Is a Revolving Balance?

- If we don't pay the balance of the revolving credit account in full every month, the unpaid portion carries over to the next month. That's called a revolving balance

What is the Average Open to buy?

- 'Open to Buy' means the amount left on your credit card to use. Now, this column represents the average of this value for the last 12 months.

What is the Average utilization Ratio?

- The Avg_Utilization_Ratio represents how much of the available credit the customer spent. This is useful for calculating credit scores.

Relation b/w Avg_Open_To_Buy, Credit_Limit and Avg_Utilization_Ratio:

- (Avg_Open_To_Buy / Credit_Limit) + Avg_Utilization_Ratio = 1

Please read the instructions carefully before starting the project.

This is a commented Jupyter IPython Notebook file in which all the instructions and tasks to be performed are mentioned.

- Blanks '_______' are provided in the notebook that needs to be filled with an appropriate code to get the correct result. With every '_______' blank, there is a comment that briefly describes what needs to be filled in the blank space.
- Identify the task to be performed correctly, and only then proceed to write the required code.
- Fill the code wherever asked by the commented lines like "# write your code here" or "# complete the code". Running incomplete code may throw error.
- Please run the codes in a sequential manner from the beginning to avoid any unnecessary errors.
- Add the results/observations (wherever mentioned) derived from the analysis in the presentation and submit the same.

Importing necessary libraries

Skipping (commenting out) installation of libraries as they are pre installed.

```
# Installing the libraries with the specified version.
# uncomment and run the following line if Google Colab is being used
# !pip install scikit-learn==1.2.2 seaborn==0.13.1 matplotlib==3.7.1
numpy==1.25.2 pandas==1.5.3 imbalanced-learn==0.10.1 xgboost==2.0.3 -q
--user
# Installing the libraries with the specified version.
# uncomment and run the following lines if Jupyter Notebook is being
used
# !pip install scikit-learn==1.2.2 seaborn==0.13.1 matplotlib==3.7.1
numpy==1.25.2 pandas==1.5.3 imblearn==0.12.0 xgboost==2.0.3 -q --user
# !pip install --upgrade -q threadpoolctl
```

```
import pandas as pd
import numpy as np
from sklearn import metrics
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier,
GradientBoostingClassifier
#To install xgboost library use - !pip install xgboost
from xgboost import XGBClassifier
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from sklearn.model_selection import RandomizedSearchCV
# Below libs are for scaling the financial data
from sklearn.preprocessing import StandardScaler, RobustScaler
from itertools import combinations
```

Loading the dataset

```
# Mounting google drive and initilizing path variable
from google.colab import drive
drive.mount("/content/drive")
path = '/content/drive/MyDrive/PGPAIML/Project-3/'
```

```
Mounted at /content/drive
```

```
df = pd.read_csv(path+'BankChurners.csv')
# Making a copy to keep the original data intact, may required later.
data = df.copy()
```

Data Overview

```
data.head(10)
```

{"type":"dataframe","variable_name":"data"}

```
data.shape
```

```
(10127, 21)
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10127 entries, 0 to 10126
Data columns (total 21 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   CLIENTNUM                 10127 non-null  int64  
 1   Attrition_Flag            10127 non-null  object 
 2   Customer_Age              10127 non-null  int64  
 3   Gender                    10127 non-null  object 
 4   Dependent_count           10127 non-null  int64  
 5   Education_Level           8608 non-null   object 
 6   Marital_Status            9378 non-null   object 
 7   Income_Category           10127 non-null  object 
 8   Card_Category             10127 non-null  object 
 9   Months_on_book            10127 non-null  int64  
 10  Total_Relationship_Count  10127 non-null  int64  
 11  Months_Inactive_12_mon    10127 non-null  int64  
 12  Contacts_Count_12_mon     10127 non-null  int64  
 13  Credit_Limit              10127 non-null  float64
 14  Total_Revolving_Bal       10127 non-null  int64  
 15  Avg_Open_To_Buy           10127 non-null  float64
 16  Total_Amt_Chng_Q4_Q1      10127 non-null  float64
 17  Total_Trans_Amt           10127 non-null  int64  
 18  Total_Trans_Ct            10127 non-null  int64  
 19  Total_Ct_Chng_Q4_Q1       10127 non-null  float64
 20  Avg_Utilization_Ratio     10127 non-null  float64
dtypes: float64(5), int64(10), object(6)
memory usage: 1.6+ MB
```

```
data.describe(include='all').T
```

{"summary":"{\n \"name\": \"data\",\n \"rows\": 21,\n \"fields\": [\n   {\n   \"column\": \"count\",\n   \"properties\": {\n   \"dtype\": \"date\",\n   \"min\": \"8608\",\n   \"max\": 10127.0,\n   \"num_unique_values\": 3,\n   \"samples\": [\n   10127.0,\n   \"8608\",\n   \"9378\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"unique\",\n   \"properties\": {\n   \"dtype\": \"date\",\n   \"min\": 2,\n   \"max\": 6,\n   \"num_unique_values\": 4,\n   \"samples\": [\n   6,\n   4,\n   2\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"top\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 6,\n   \"samples\": [\n   \"Existing Customer\",\n   \"F\",\n   \"Blue\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"freq\",\n   \"properties\": {\n   \"dtype\": \"date\",\n   \"min\": \"3128\",\n   \"max\": \"9436\",\n   \"num_unique_values\": 6,\n   \"samples\": [\n

```
\"8500\",\n          \"5358\",\n          \"9436\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"mean\",\n      \"properties\": {\n
\"dtype\": \"date\",\n        \"min\": 0.2748935518909845,\n
\"max\": 739177606.3336625,\n        \"num_unique_values\": 15,\n
\"samples\": [\n          7469.139636614989,\n
4404.086303939963,\n          739177606.3336625\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"std\",\n      \"properties\": {\n
\"dtype\": \"date\",\n        \"min\": 0.21920676923070248,\n
\"max\": 36903783.45023111,\n        \"num_unique_values\": 15,\n
\"samples\": [\n          9090.685323679128,\n
3397.129253557085,\n          36903783.45023111\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"min\",\n      \"properties\": {\n
\"dtype\": \"date\",\n        \"min\": 0.0,\n        \"max\":
708082083.0,\n        \"num_unique_values\": 9,\n        \"samples\":
[\n          510.0,\n          26.0,\n          1438.3\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"25%\",\n      \"properties\": {\n
\"dtype\": \"date\",\n        \"min\": 0.023,\n        \"max\":
713036770.5,\n        \"num_unique_values\": 14,\n        \"samples\":
[\n          0.631,\n          45.0,\n          713036770.5\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"50%\",\n      \"properties\": {\n        \"dtype\": \"date\",\n
\"min\": 0.176,\n        \"max\": 717926358.0,\n
\"num_unique_values\": 13,\n        \"samples\": [\n          0.702,\n
3899.0,\n          717926358.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"75%\",\n      \"properties\": {\n        \"dtype\":
\"date\",\n        \"min\": 0.503,\n        \"max\": 773143533.0,\n
\"num_unique_values\": 13,\n        \"samples\": [\n          0.818,\n
4741.0,\n          773143533.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"max\",\n      \"properties\": {\n        \"dtype\":
\"date\",\n        \"min\": 0.999,\n        \"max\": 828343083.0,\n
\"num_unique_values\": 12,\n        \"samples\": [\n          3.714,\n
139.0,\n          828343083.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe"}
```


Observations

- The data set has 10127 observations and 21 columns, 6 of which are object columns and 15 numerical columns.
- Out of 10,127 rows, 8,500 marked as “Existing Customer” and 1,627 as “Attrited Customer.” This suggests a class imbalance.

- Customer_Age: The age distribution has a mean of 46 and ranges from 26 to 73. It appears normally distributed, with the middle 50% between 41 and 52.
- Dependent_count: This variable ranges from 0 to 5, with a mean of about 2. It's an integer variable with lower variability.
- Months_on_book: This indicates customer tenure, with an average of 36 months and a range from 13 to 56. This might correlate with loyalty or retention.
- Months_Inactive_12_mon and Contacts_Count_12_mon: Both have a median of around 2 and range up to 6. Higher inactivity or frequent contact with the bank could indicate customer dissatisfaction, making these potential predictors of churn.
- Credit_Limit: This has a large range (1,438 to 34,516) with a high standard deviation.
- Total_Revolving_Bal: This represents the outstanding balance, with values between 0 and 2,517.
- Avg_Open_To_Buy: Similar to Credit_Limit, it has a large range (3 to 34,516).
- Total_Trans_Amt and Total_Trans_Ct: The total transaction amount and count also show wide ranges.
- Total_Amt_Chng_Q4_Q1 and Total_Ct_Chng_Q4_Q1: These metrics show changes in transaction amount and count between quarters.
- Avg_Utilization_Ratio This ratio varies from 0 to almost 1, with a mean of 0.27. This variable reflects how much of the available credit the customer is using

Exploratory Data Analysis (EDA)

- EDA is an important part of any project involving data.
- It is important to investigate and understand the data better before building a model with it.

Questions:

1. How is the total transaction amount distributed?
2. What is the distribution of the level of education of customers?
3. What is the distribution of the level of income of customers?
4. How does the change in transaction amount between Q4 and Q1 (`total_ct_change_Q4_Q1`) vary by the customer's account status (`Attrition_Flag`)?
5. How does the number of months a customer was inactive in the last 12 months (`Months_Inactive_12_mon`) vary by the customer's account status (`Attrition_Flag`)?
6. What are the attributes that have a strong correlation with each other?

The below functions need to be defined to carry out the Exploratory Data Analysis.

```
# function to plot a boxplot and a histogram along the same scale.


def histogram_boxplot(data, feature, figsize=(12, 7), kde=False,
bins=None):
    """
    Boxplot and histogram combined
```

```
    data: dataframe
    feature: dataframe column
    figsize: size of figure (default (12,7))
    kde: whether to the show density curve (default False)
    bins: number of bins for histogram (default None)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(
        nrows=2,  # Number of rows of the subplot grid= 2
        sharex=True,  # x-axis will be shared among all subplots
        gridspec_kw={"height_ratios": (0.25, 0.75)},
        figsize=figsize,
    )  # creating the 2 subplots
    sns.boxplot(
        data=data, x=feature, ax=ax_box2, showmeans=True,
color="violet"
    )  # boxplot will be created and a triangle will indicate the mean
value of the column
    sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2, bins=bins,
palette="winter"
    ) if bins else sns.histplot(
        data=data, x=feature, kde=kde, ax=ax_hist2
    )  # For histogram
    ax_hist2.axvline(
        data[feature].mean(), color="green", linestyle="--"
    )  # Add mean to the histogram
    ax_hist2.axvline(
        data[feature].median(), color="black", linestyle="-"
    )  # Add median to the histogram

# function to create labeled barplots


def labeled_barplot(data, feature, perc=False, n=None):
    """
    Barplot with percentage at the top

    data: dataframe
    feature: dataframe column
    perc: whether to display percentages instead of count (default is
False)
    n: displays the top n category levels (default is None, i.e.,
display all levels)
    """

    total = len(data[feature])  # length of the column
    count = data[feature].nunique()
    if n is None:
        plt.figure(figsize=(count + 1, 5))
```

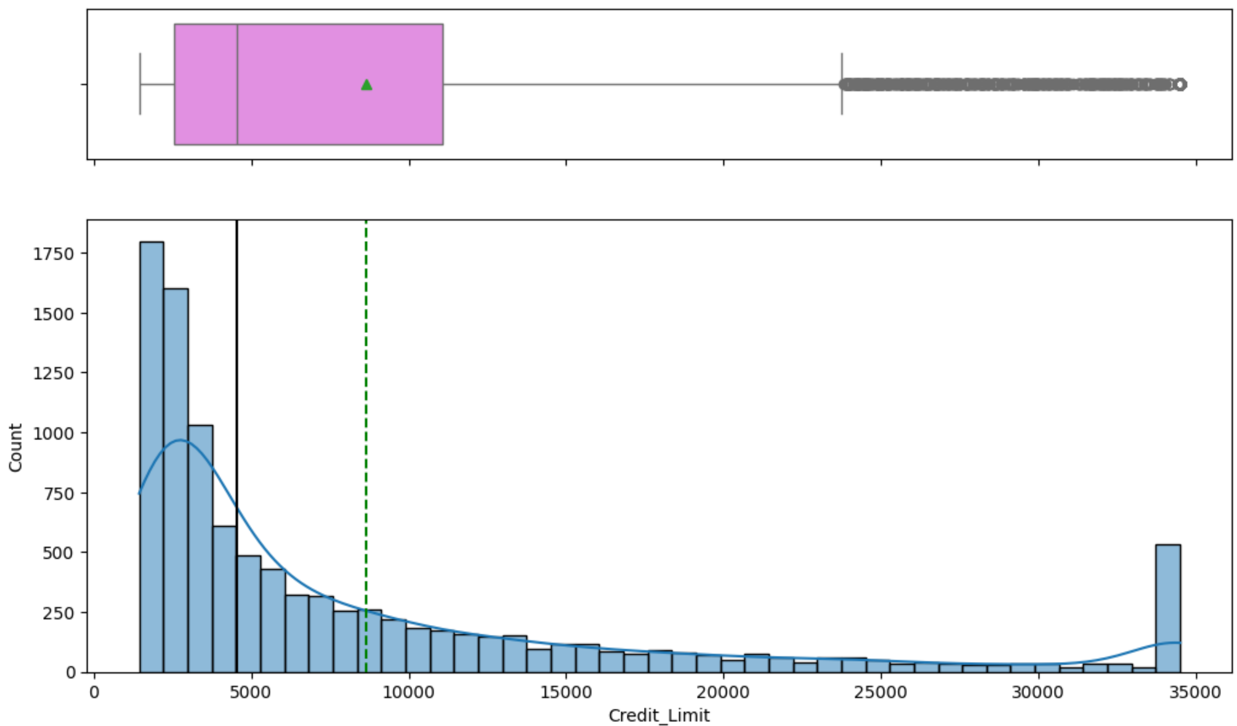
```
    else:
        plt.figure(figsize=(n + 1, 5))

    plt.xticks(rotation=90, fontsize=15)
    ax = sns.countplot(
        data=data,
        x=feature,
        palette="Paired",
        order=data[feature].value_counts().index[:n].sort_values(),
    )

    for p in ax.patches:
        if perc == True:
            label = "{:.1f}%".format(
                100 * p.get_height() / total
            )  # percentage of each class of the category
        else:
            label = p.get_height()  # count of each level of the
category

        x = p.get_x() + p.get_width() / 2  # width of the plot
        y = p.get_height()  # height of the plot

        ax.annotate(
            label,
            (x, y),
            ha="center",
            va="center",
            size=12,
            xytext=(0, 5),
            textcoords="offset points",
        )  # annotate the percentage

    plt.show()  # show the plot

# function to plot stacked bar chart

def stacked_barplot(data, predictor, target):
    """
    Print the category counts and plot a stacked bar chart

    data: dataframe
    predictor: independent variable
    target: target variable
    """
    count = data[predictor].nunique()
    sorter = data[target].value_counts().index[-1]
    tab1 = pd.crosstab(data[predictor], data[target],
margins=True).sort_values(
        by=sorter, ascending=False
```

```
    )
    print(tab1)
    print("-" * 120)
    tab = pd.crosstab(data[predictor], data[target],
normalize="index").sort_values(
        by=sorter, ascending=False
    )
    tab.plot(kind="bar", stacked=True, figsize=(count + 1, 5))
    plt.legend(
        loc="lower left", frameon=False,
    )
    plt.legend(loc="upper left", bbox_to_anchor=(1, 1))
    plt.show()

### Function to plot distributions

def distribution_plot_wrt_target(data, predictor, target):

    fig, axs = plt.subplots(2, 2, figsize=(12, 10))

    target_uniq = data[target].unique()

    axs[0, 0].set_title("Distribution of target for target=" +
str(target_uniq[0]))
    sns.histplot(
        data=data[data[target] == target_uniq[0]],
        x=predictor,
        kde=True,
        ax=axs[0, 0],
        color="teal",
    )

    axs[0, 1].set_title("Distribution of target for target=" +
str(target_uniq[1]))
    sns.histplot(
        data=data[data[target] == target_uniq[1]],
        x=predictor,
        kde=True,
        ax=axs[0, 1],
        color="orange",
    )

    axs[1, 0].set_title("Boxplot w.r.t target")
    sns.boxplot(data=data, x=target, y=predictor, ax=axs[1, 0],
palette="gist_rainbow")

    axs[1, 1].set_title("Boxplot (without outliers) w.r.t target")
    sns.boxplot(
        data=data,
        x=target,
```

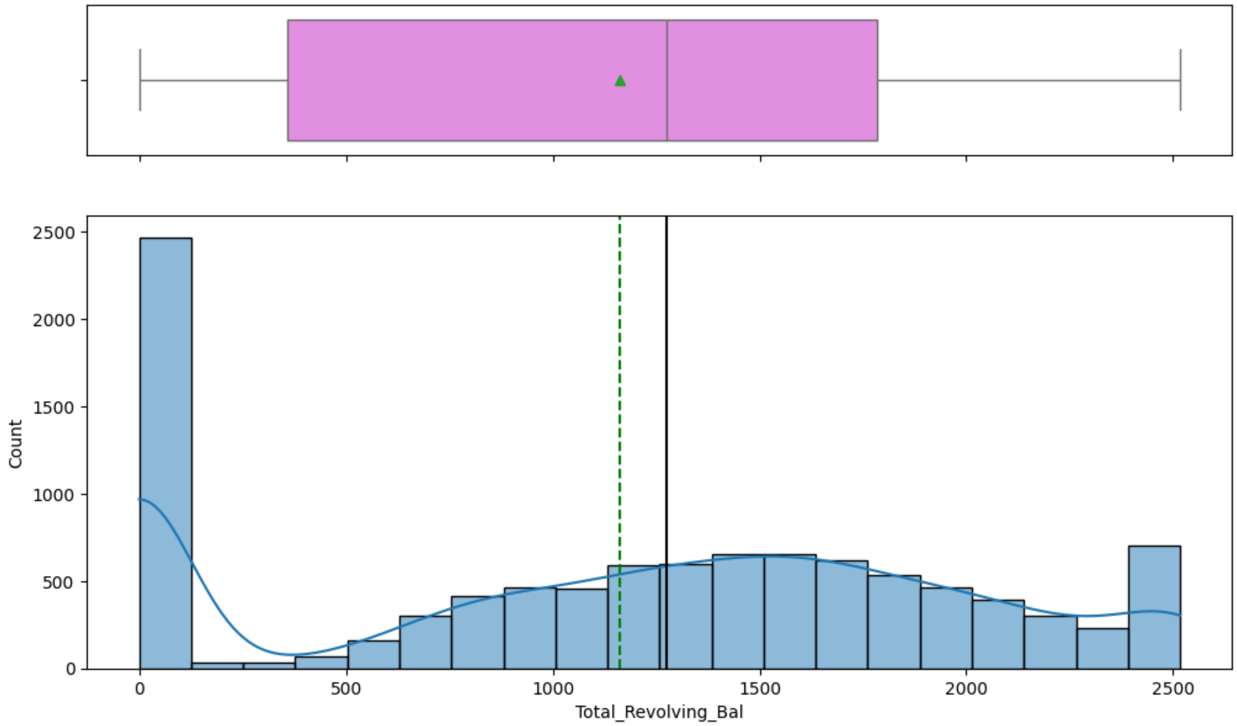
```
        y=predictor,
        ax=axs[1, 1],
        showfliers=False,
        palette="gist_rainbow",
    )

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

Univariate analysis of financial variables

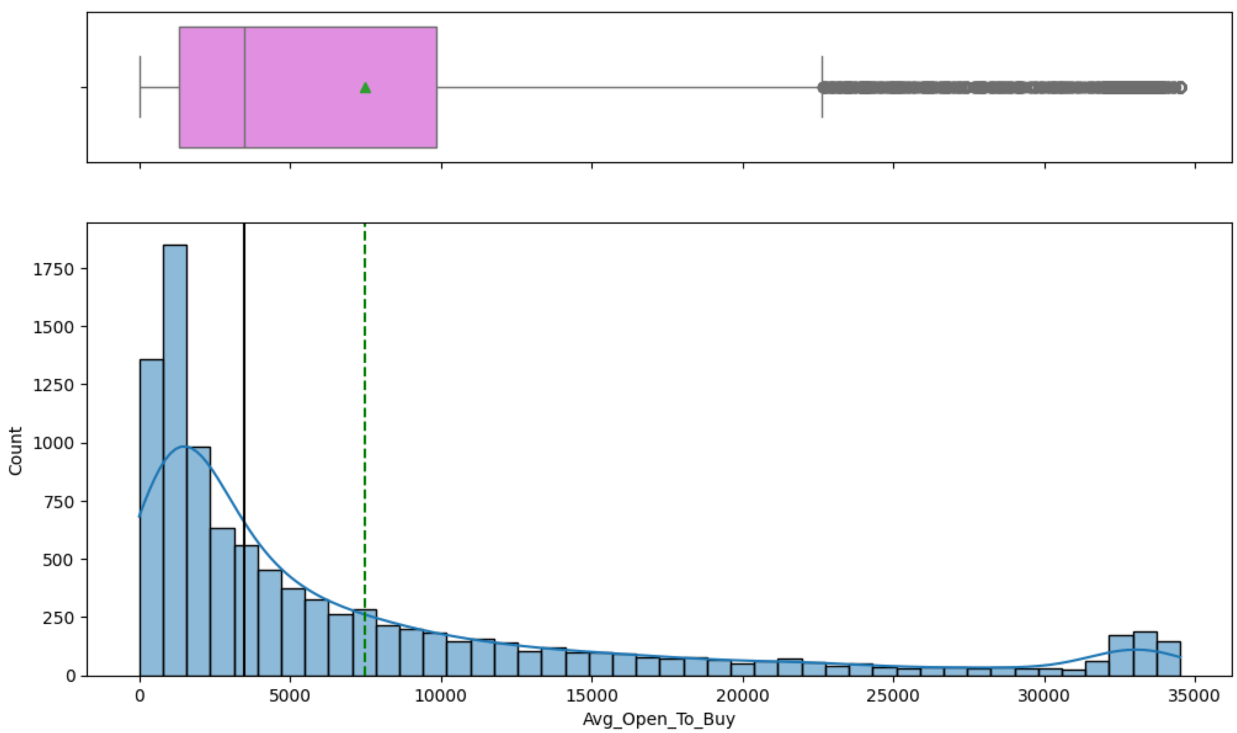
```
# Generating hist & box plot for Credit_Limit
histogram_boxplot(data=data, feature='Credit_Limit', kde=True)
```



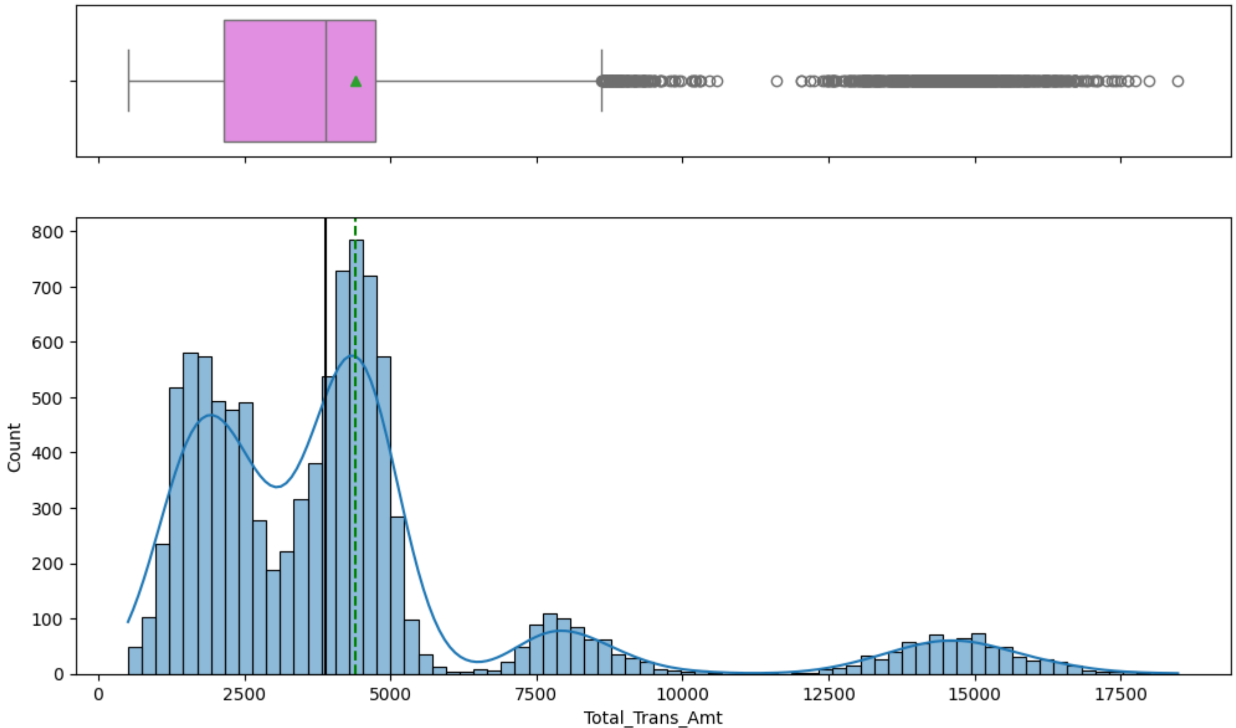

```
# Generating hist & box plot for Total_Revolving_Bal
histogram_boxplot(data=data, feature='Total_Revolving_Bal', kde=True)
```



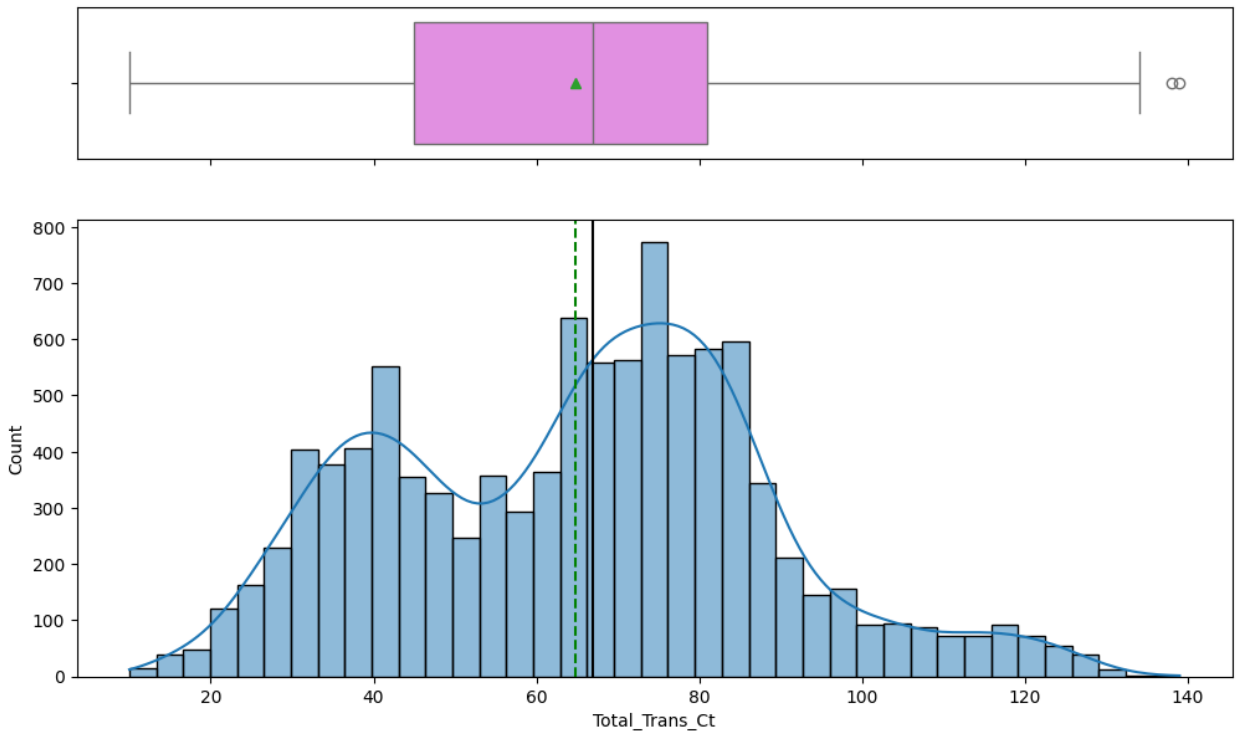

```python
# Generating hist & box plot for Avg_Open_To_Buy
histogram_boxplot(data=data, feature='Avg_Open_To_Buy', kde=True)
```



```python
# Generating hist & box plot for Total_Trans_Amt
histogram_boxplot(data=data, feature='Total_Trans_Amt', kde=True)
```




```python
# Generating hist & box plot for Total_Trans_Ct
histogram_boxplot(data=data, feature='Total_Trans_Ct', kde=True)
```

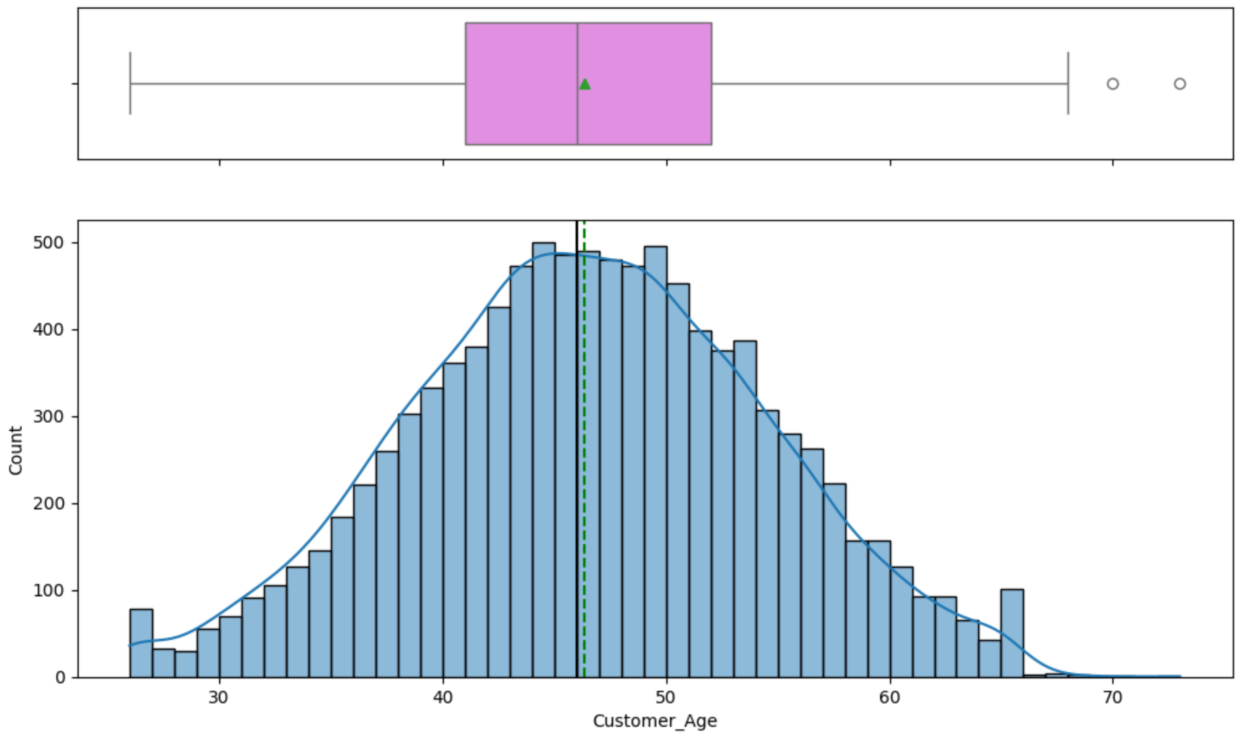


Observations

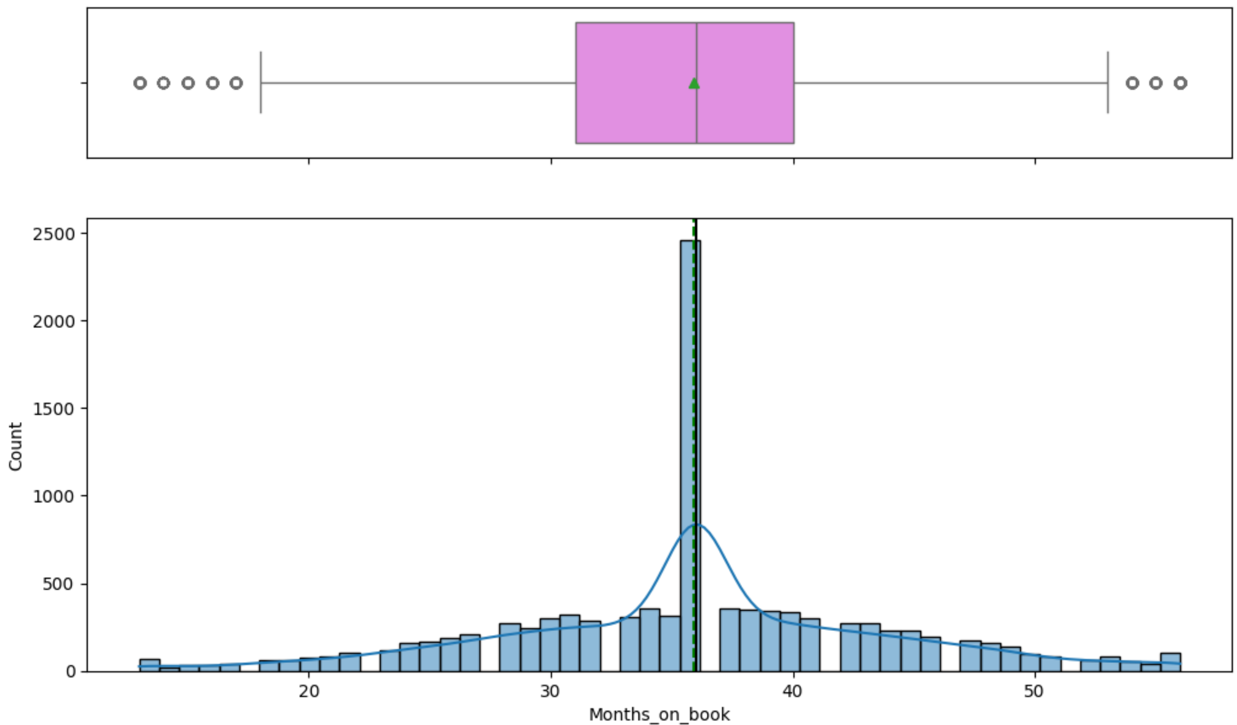
1. Credit_Limit
- Highly right-skewed with a significant number of customers having low credit limits, and a few with very high limits (up to around 35,000).
- There are several outliers on the higher end, as seen in the box plot.
1. Total_Revolving_Bal
- Appears to have a bimodal distribution, with many customers having very low or zero balances, and a spread across higher balances.
- Less extreme than Credit_Limit, but the box plot shows a wide range, but without many distinct outliers.
1. Avg_Open_To_Buy
- Similar to Credit_Limit, it's highly right-skewed with many lower values and a few high values.
- Numerous high-value outliers, indicating a small group of customers with high available credit.
1. Total_Trans_Amt
- Shows multimodal peaks, indicating different spending behavior clusters among customers.
- High outliers are present, with some customers spending significantly more than others.
1. Total_Trans_Ct
- Almost normally distributed but slightly right-skewed with a few high-value outliers.
- A few high transaction counts, as shown in the box plot.

Univariate analysis of some important engagement variables

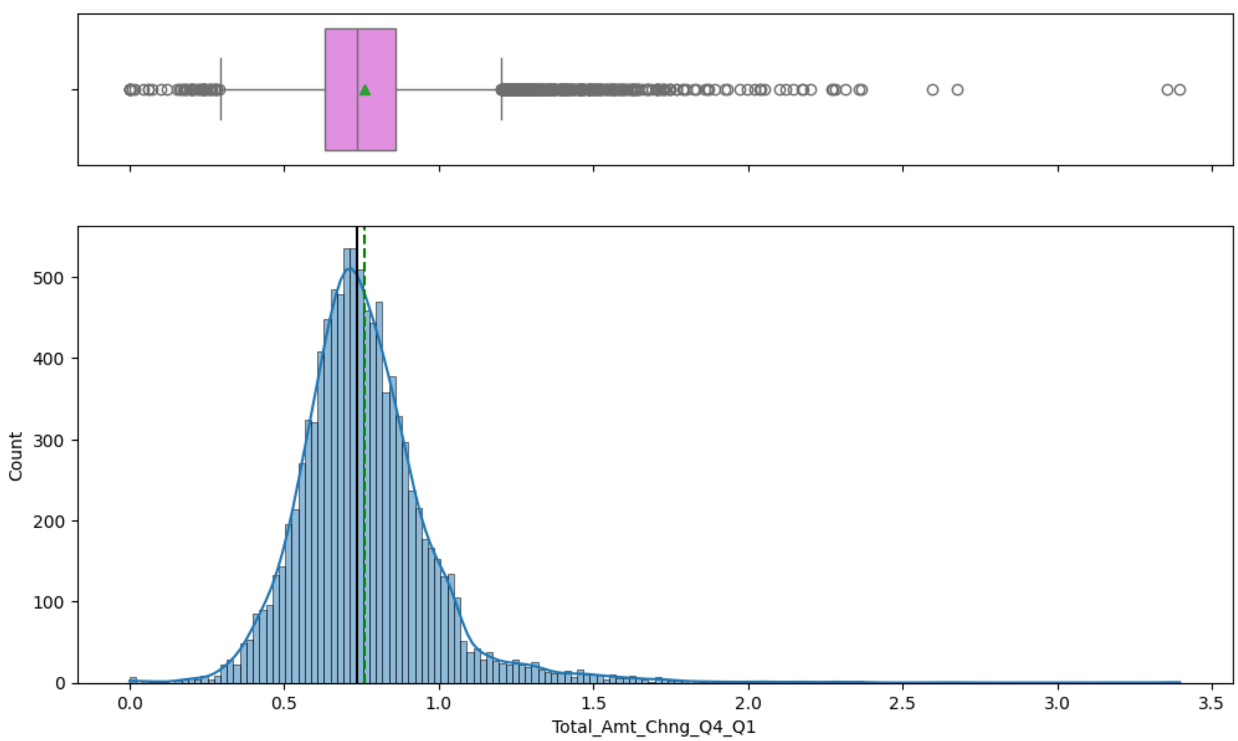
```
histogram_boxplot(data=data, feature='Customer_Age', kde=True)
```



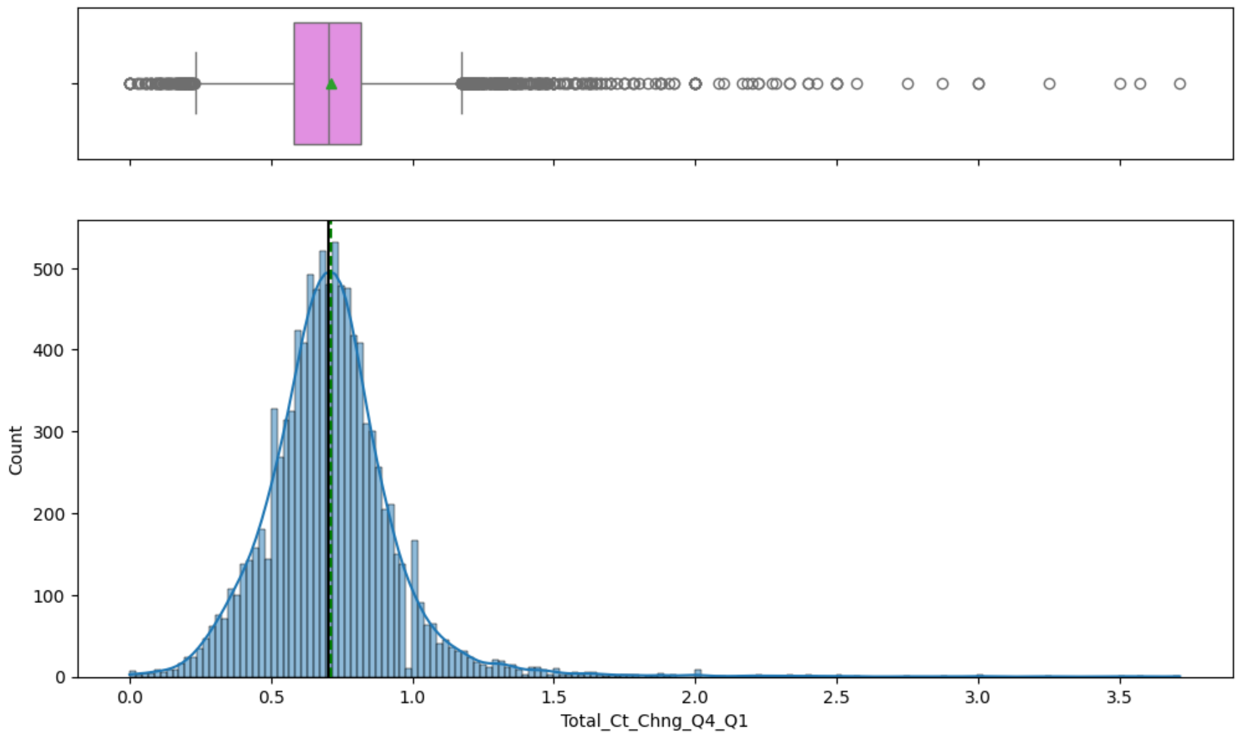

```
histogram_boxplot(data=data, feature='Months_on_book', kde=True)
```



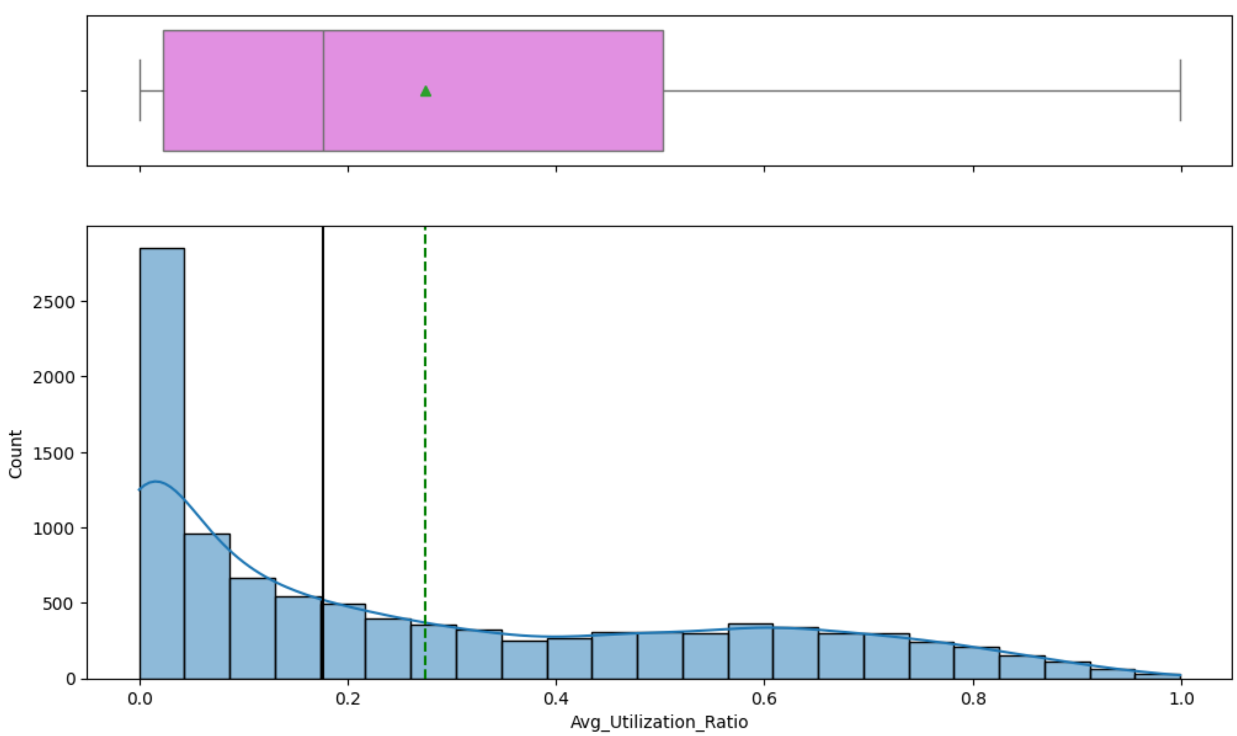

```
histogram_boxplot(data=data, feature='Total_Amt_Chng_Q4_Q1', kde=True)
```




```
histogram_boxplot(data=data, feature='Total_Ct_Chng_Q4_Q1', kde=True)
```




```
histogram_boxplot(data=data, feature='Avg_Utilization_Ratio',
kde=True)
```

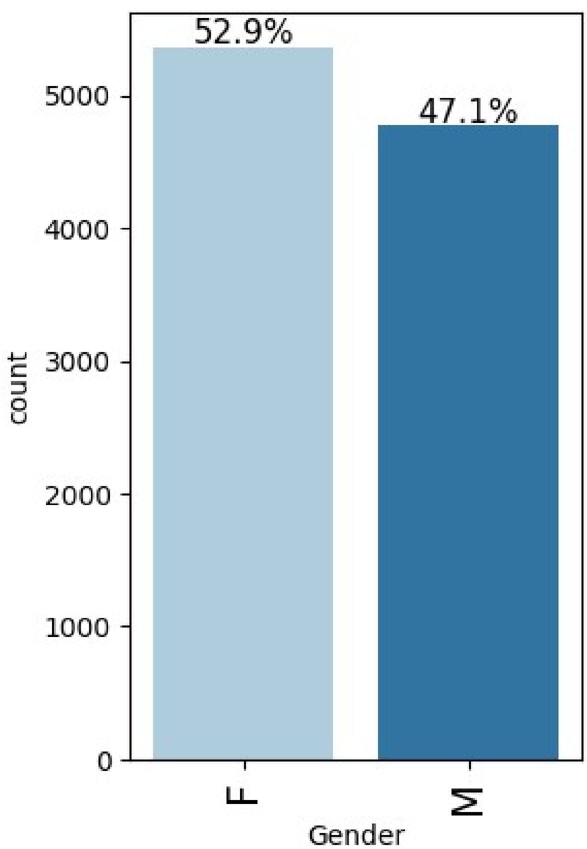


Observation

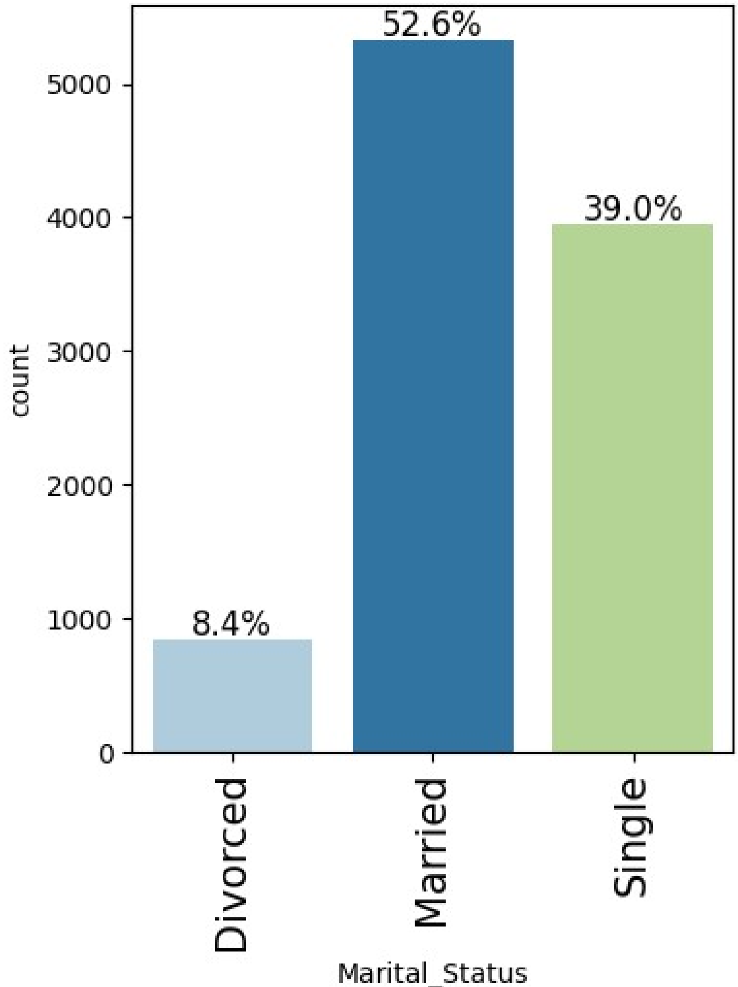
- Customer Age: The age distribution of customers is approximately normal, centered around 50 years with minimal outliers.
- Months on Book: The length of customer relationships is centered around 36 months, with some outliers at lower months.
- Total Amount Change (Q4-Q1): This plot is slightly right-skewed, with most values clustered around 0.5 to 1.0, and a few high outliers.
- Total Count Change (Q4-Q1): The distribution is also right-skewed, with values mainly around 0.5 to 1.0, and numerous high outliers.
- Average Utilization Ratio: This variable is highly right-skewed, with most values close to 0, indicating low average utilization among customers.

Univeriate Analysis - categorical variables

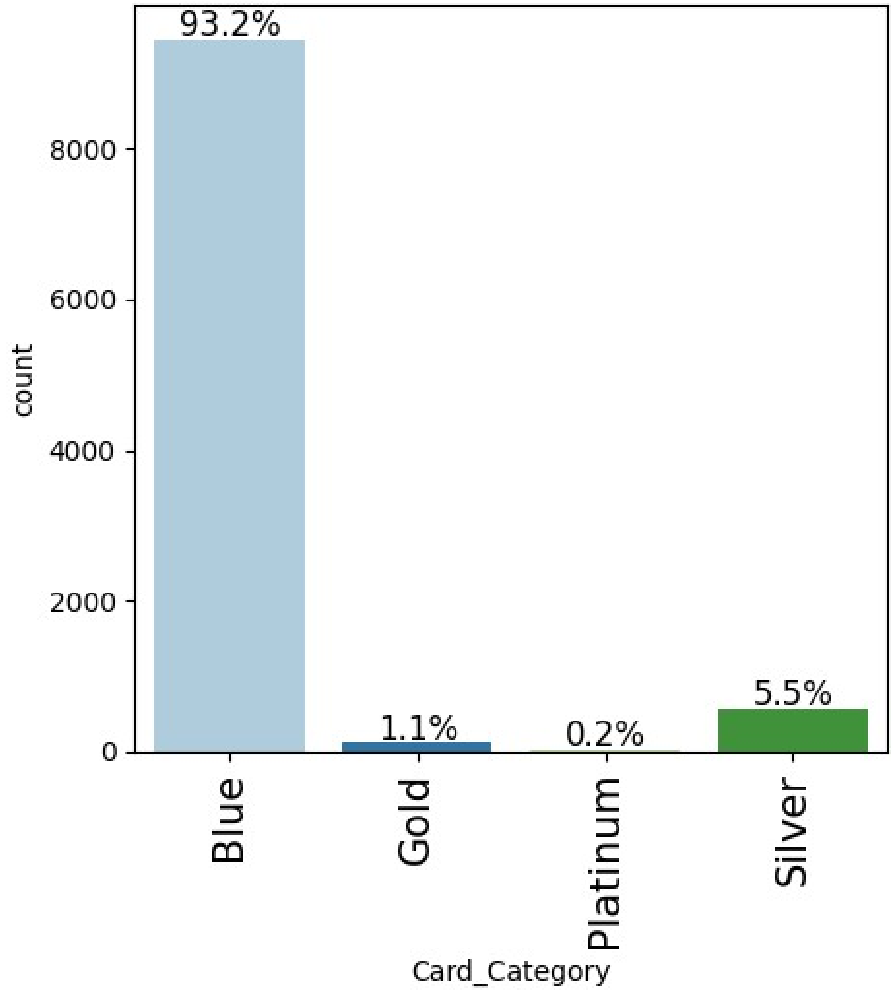
```
labeled_barplot(data=data, feature='Gender', perc=True)
```



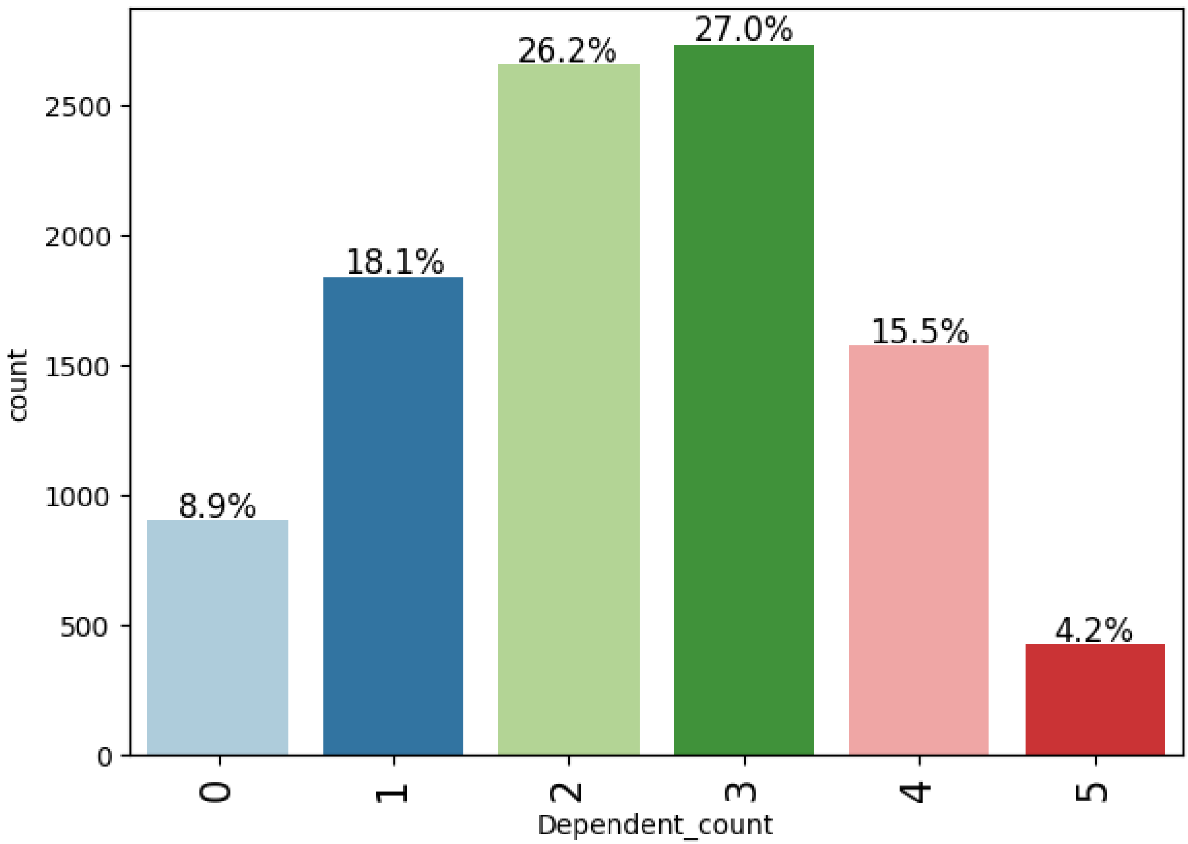

```
labeled_barplot(data=data, feature='Marital_Status', perc=True)
```



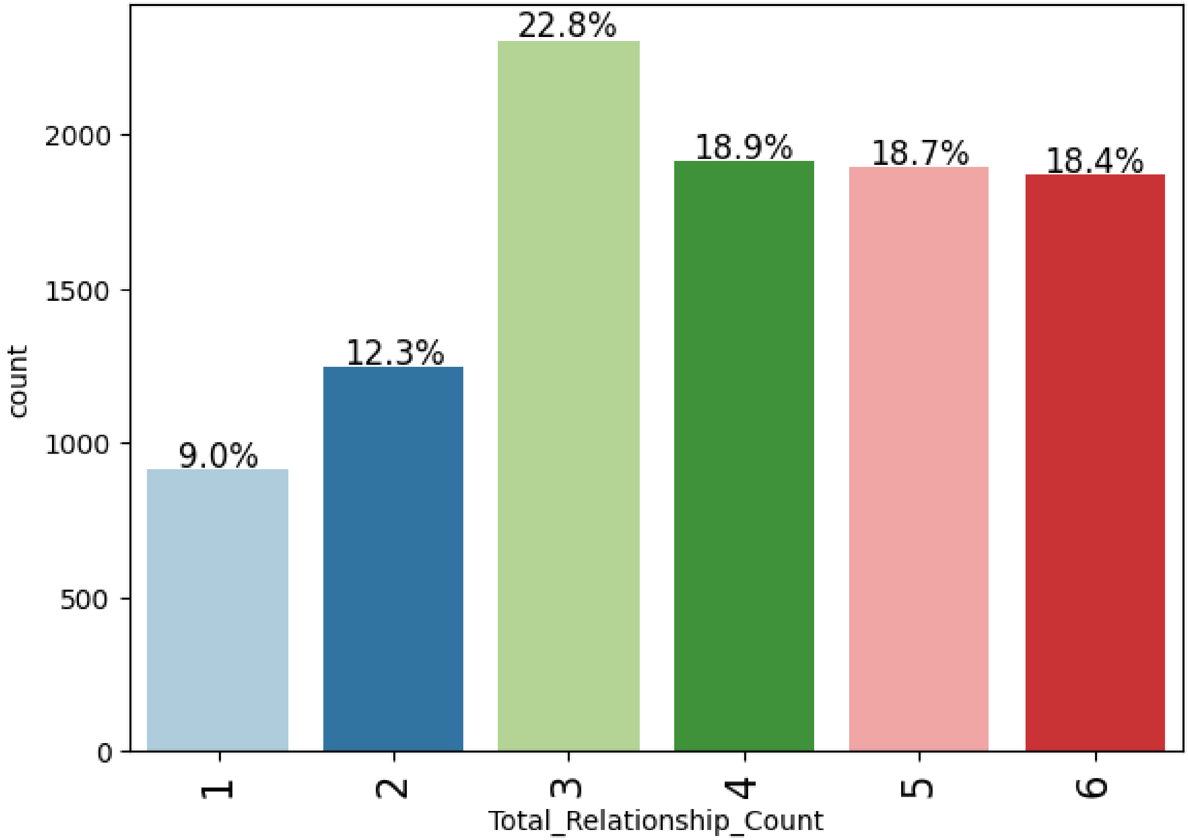

```
labeled_barplot(data=data, feature='Card_Category', perc=True)
```



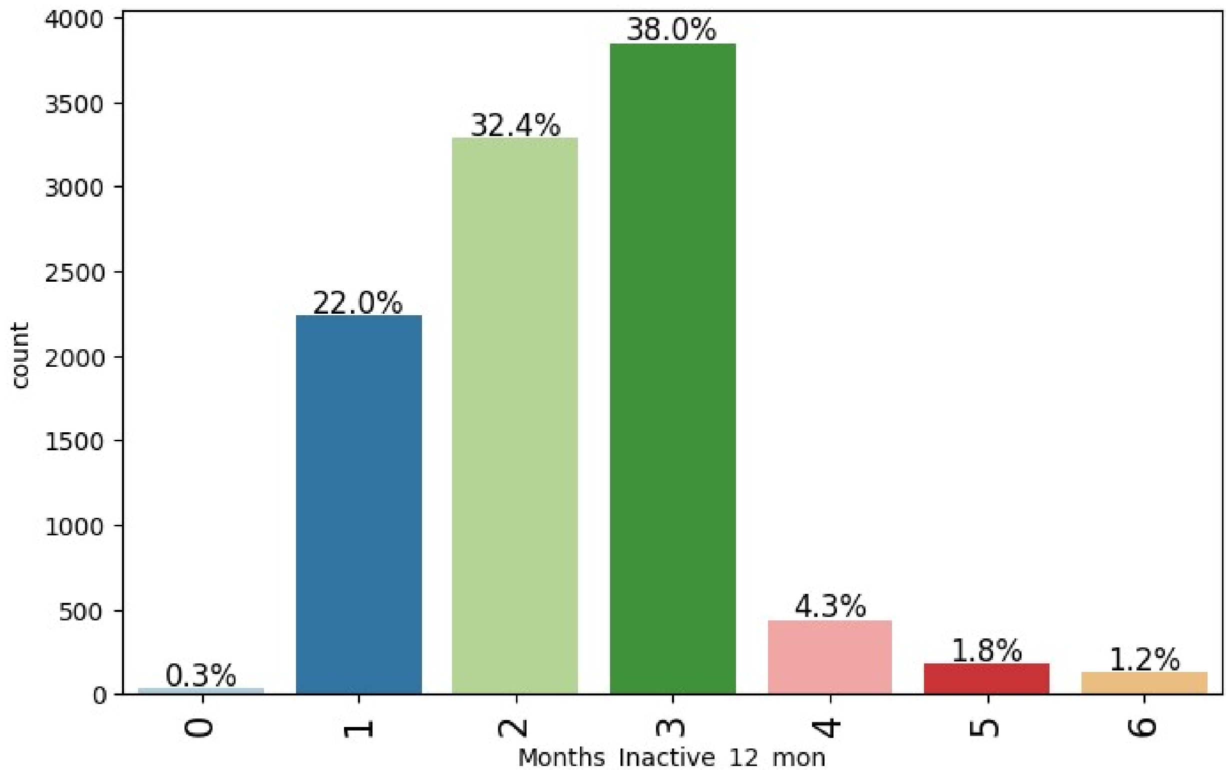

```
labeled_barplot(data=data, feature='Dependent_count', perc=True)
```



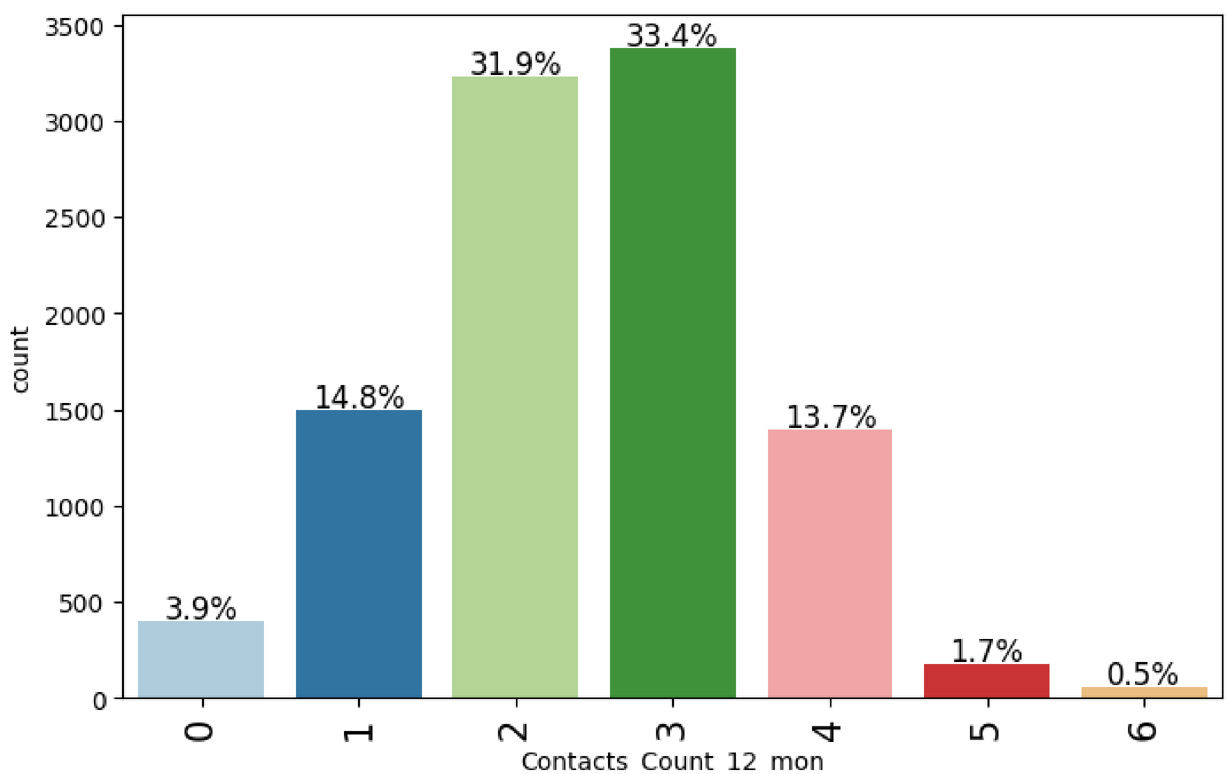

```
labeled_barplot(data=data, feature='Total_Relationship_Count',
perc=True)
```



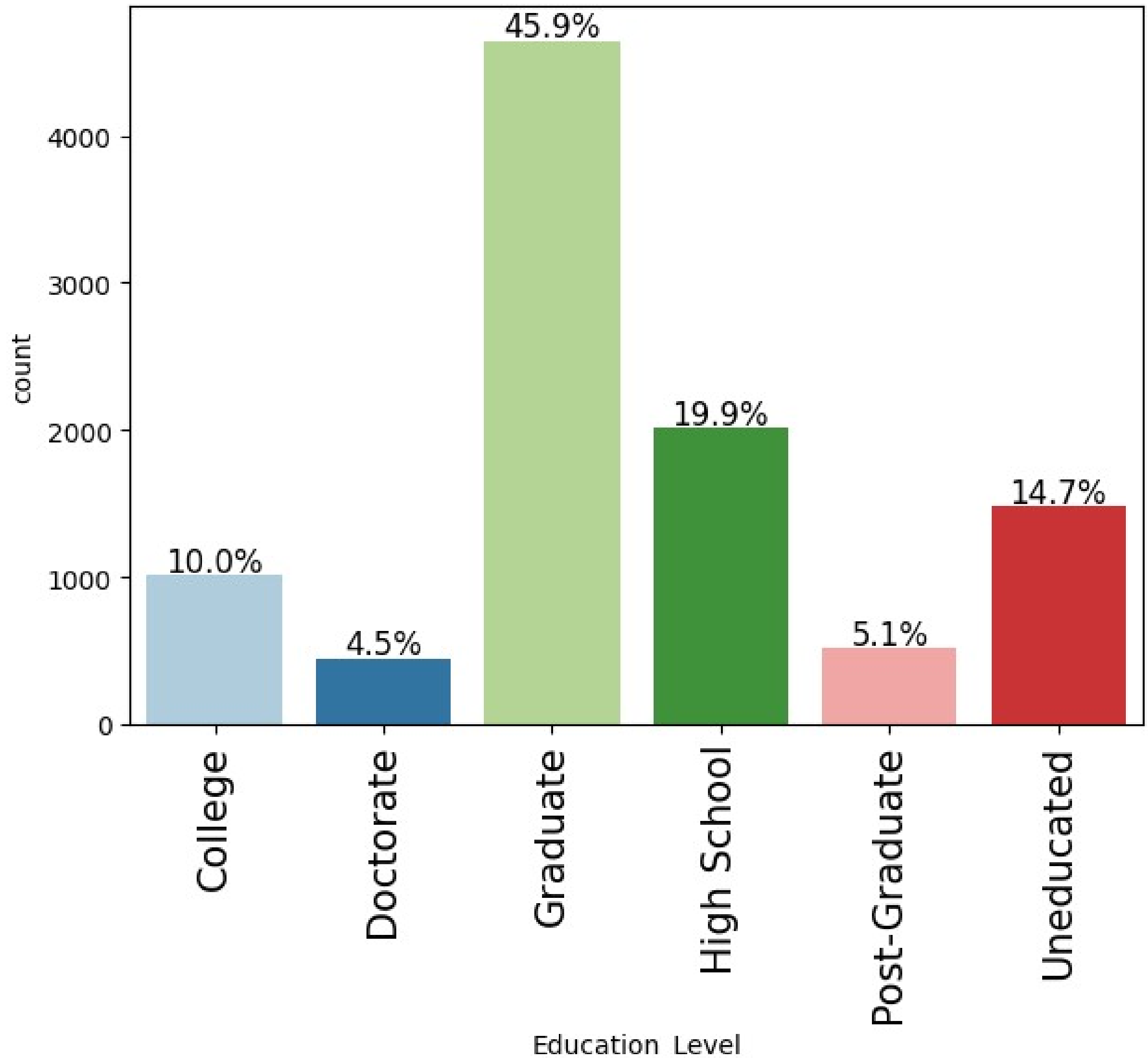

```
labeled_barplot(data=data, feature='Months_Inactive_12_mon',
perc=True)
```



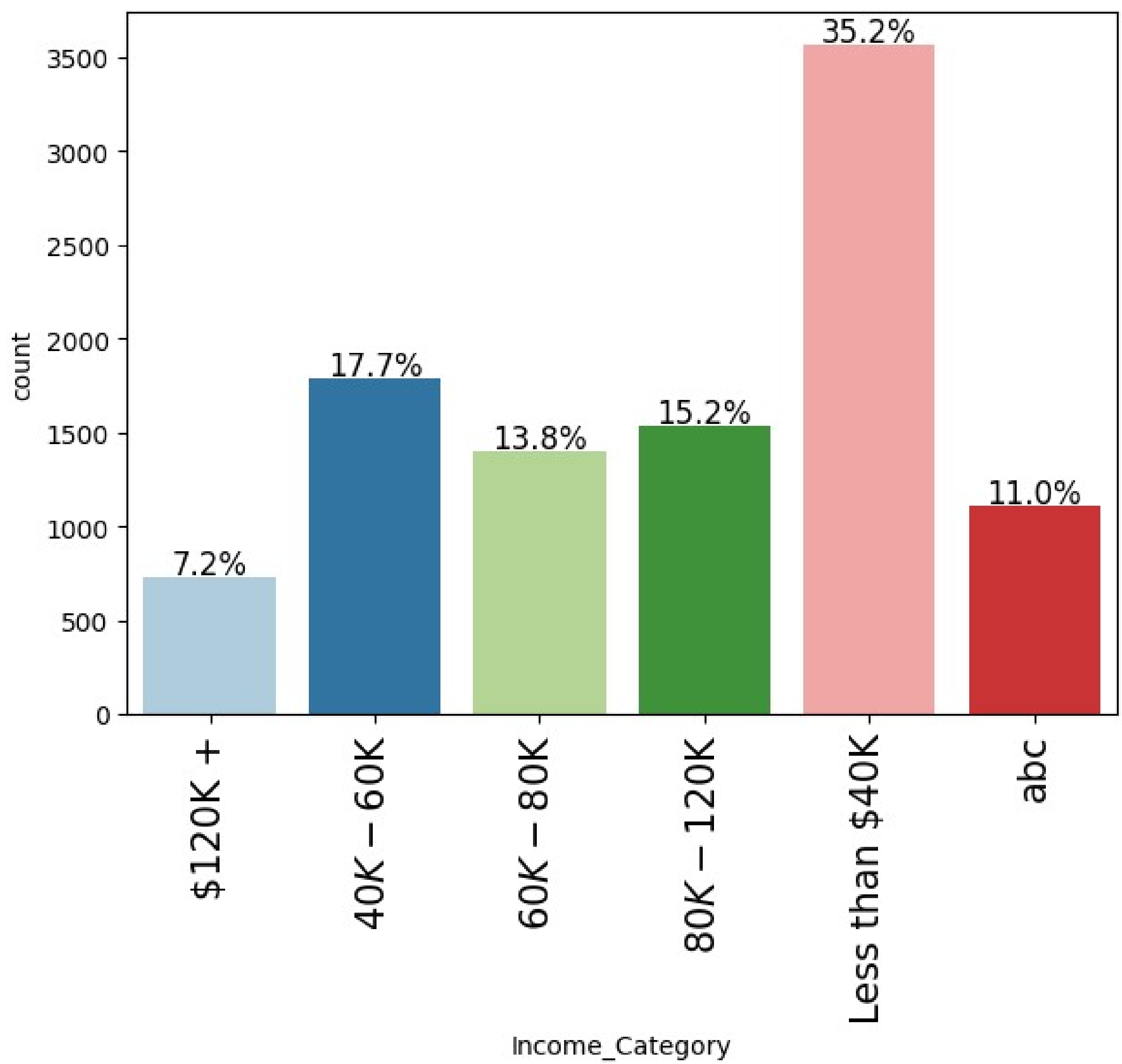

```
labeled_barplot(data=data, feature='Contacts_Count_12_mon', perc=True)
```



```
labeled_barplot(data=data, feature='Education_Level', perc=True)
```




```
labeled_barplot(data=data, feature='Income_Category', perc=True)
```



Observation

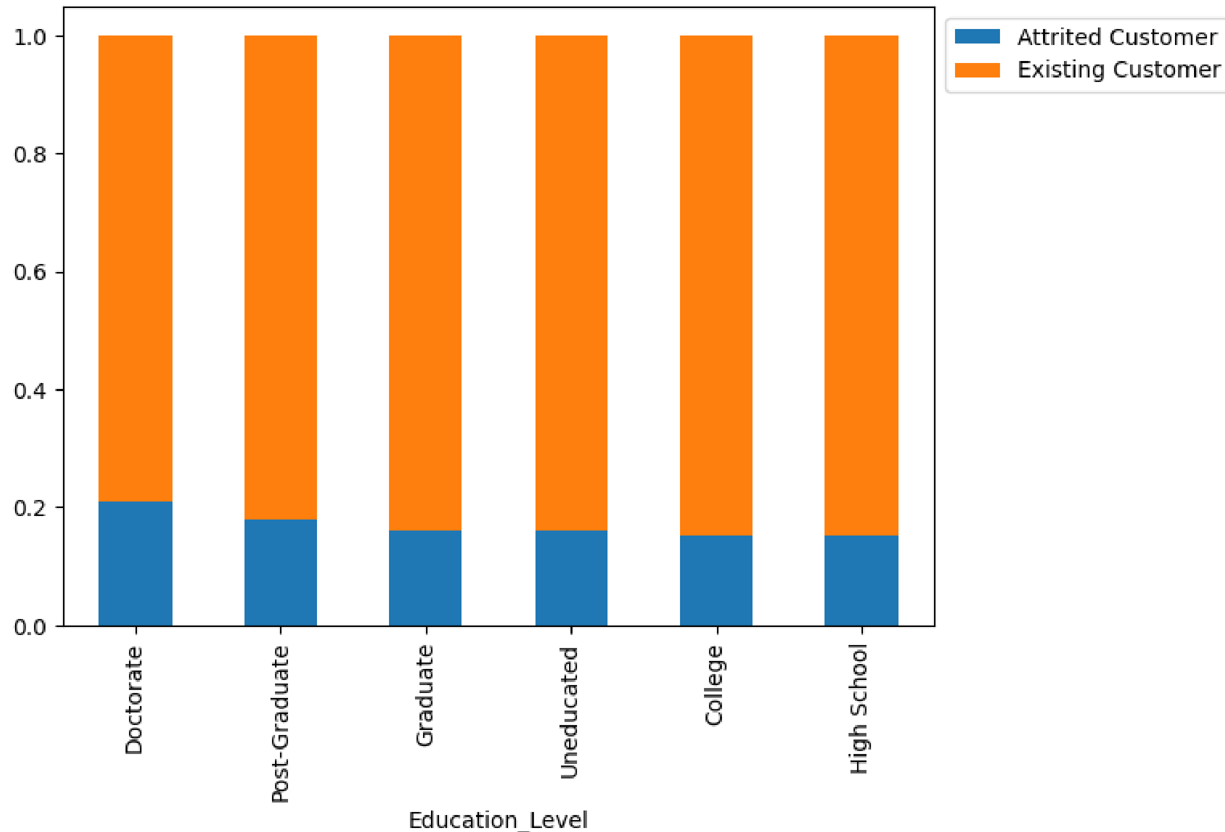
- Gender: The gender distribution is nearly balanced, with 52.9% female and 47.1% male customers.
- Card Category: 93.2% of customers have a Blue card, followed by 5.5% with a Silver card, 1.1% with Gold, and 0.2% with Platinum.
- Marital Status: 52.6% of customers are married, 39.0% are single, and 8.4% are divorced.
- Dependent Count: Most customers have 2 or 3 dependents, with smaller percentages having 0, 1, 4, or 5 dependents.
- Total Relationship Count: The majority of customers hold 3, 4, 5, or 6 products, with fewer customers having 1 or 2 products.

- Months Inactive: Most customers have been inactive for 3 months, followed by 2 and 1 month, with very few inactive for 4 months or more.
- Contacts Count: Most customers had 2 or 3 contacts with the bank in the last 12 months, with fewer customers having 0, 1, 4, or more contacts.
- Education: The majority of customers have a "Graduate" education level Education (45.9%)
- Income: The largest group is the "Less than '$40K" income bracket (35.2%), followed by the "$40K - $60K" (17.7%) bracket. This aligns with typical consumer demographics in many banks. Lower-income groups might be more susceptible to churn, to fees or balance concerns.

Bivariate Analysis with Attrition_Flag

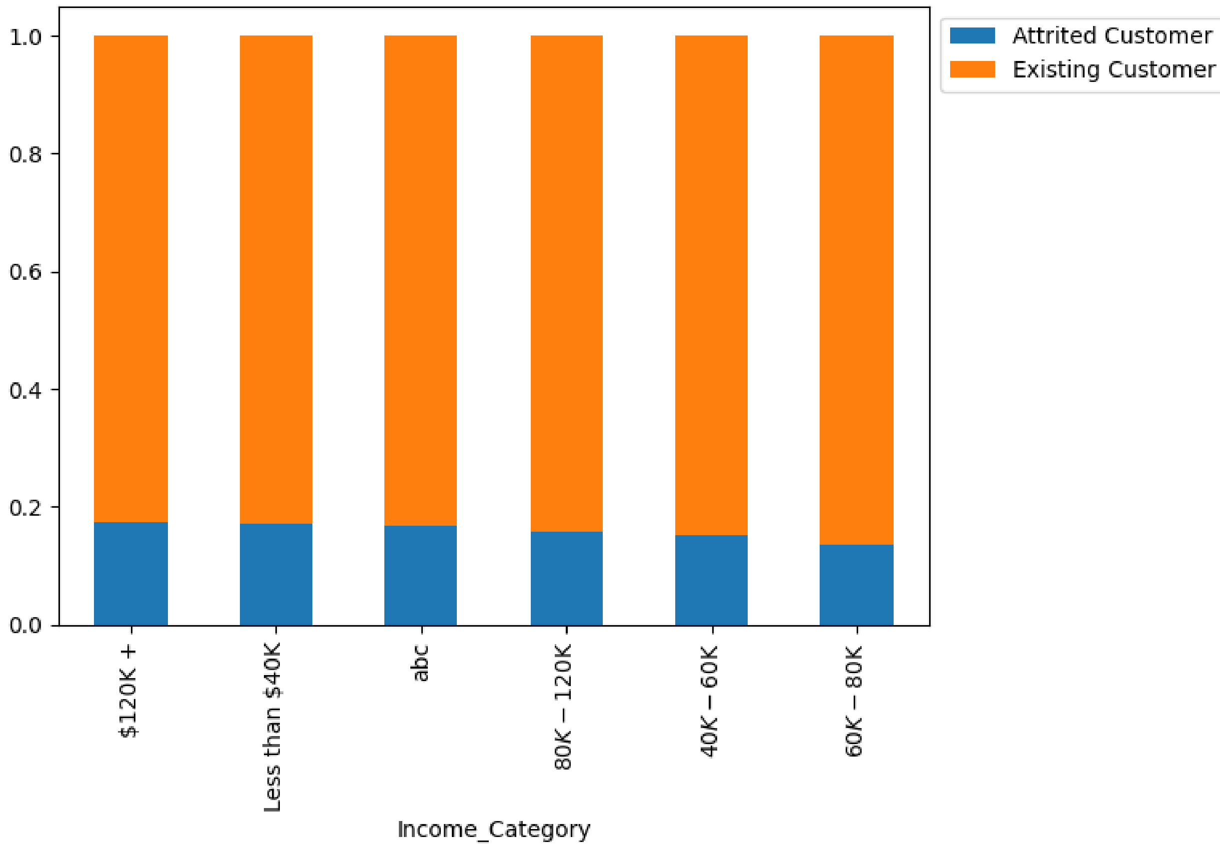
```
stacked_barplot(data=data, predictor='Education_Level',
target='Attrition_Flag')
```

```
Attrition_Flag   Attrited Customer  Existing Customer    All
Education_Level
All                           1627               8500  10127
Graduate                       743               3904   4647
High School                    306               1707   2013
Uneducated                     237               1250   1487
College                        154                859   1013
Doctorate                       95                356    451
Post-Graduate                   92                424    516
----------------------------------------------------------------------
--------------------------------------------------
```



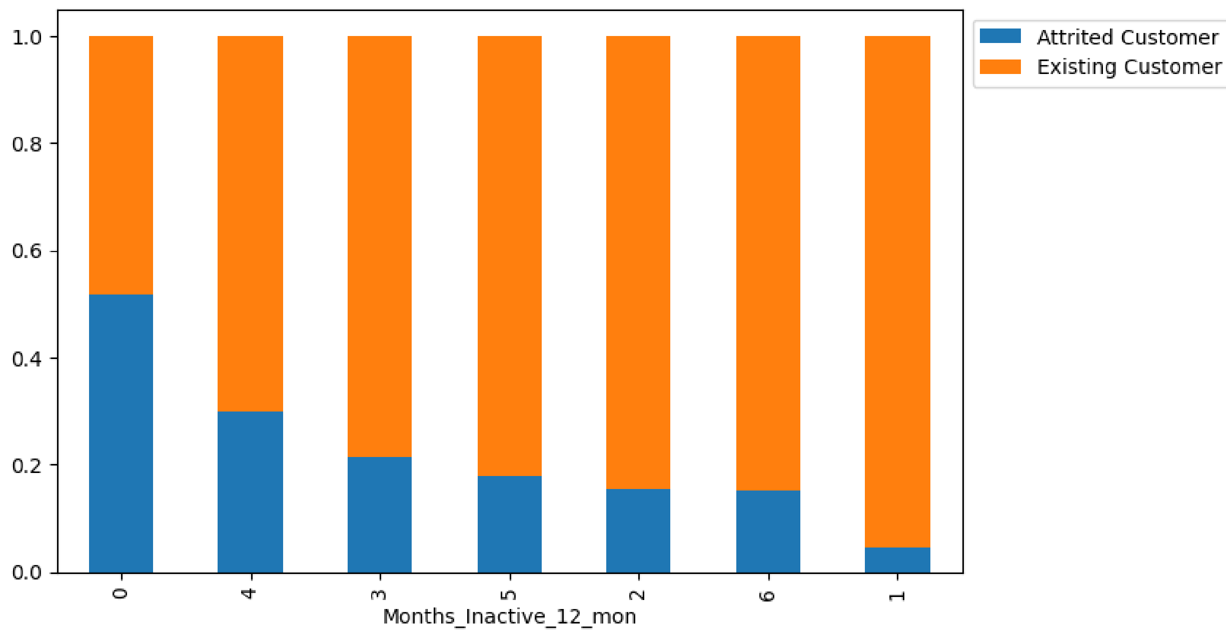

```
stacked_barplot(data=data, predictor='Income_Category',
target='Attrition_Flag')
```

```
Attrition_Flag   Attrited Customer  Existing Customer    All
Income_Category
All                           1627               8500  10127
Less than $40K                 612               2949   3561
$40K - $60K                    271               1519   1790
$80K - $120K                   242               1293   1535
$60K - $80K                    189               1213   1402
abc                            187                925   1112
$120K +                        126                601    727
----------------------------------------------------------------------
--------------------------------------------------
```

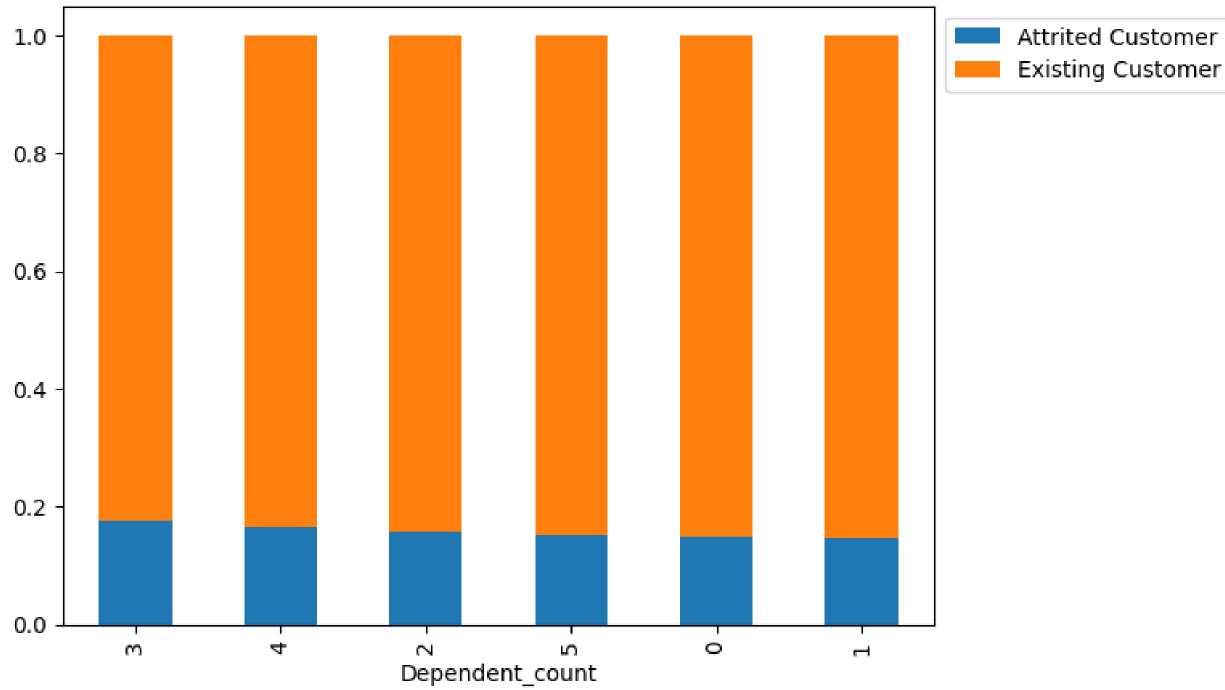



```
stacked_barplot(data=data, predictor='Months_Inactive_12_mon',
target='Attrition_Flag')
```

```
Attrition_Flag          Attrited Customer  Existing Customer    All
Months_Inactive_12_mon
All                                  1627               8500  10127
3                                     826               3020   3846
2                                     505               2777   3282
4                                     130                305    435
1                                     100               2133   2233
5                                      32                146    178
6                                      19                105    124
0                                      15                 14     29
----------------------------------------------------------------------
--------------------------------------------------
```




```
stacked_barplot(data=data, predictor='Dependent_count',
target='Attrition_Flag')

Attrition_Flag   Attrited Customer  Existing Customer    All
Dependent_count
All                           1627               8500  10127
3                              482               2250   2732
2                              417               2238   2655
1                              269               1569   1838
4                              260               1314   1574
0                              135                769    904
5                               64                360    424
----------------------------------------------------------------------
--------------------------------------------------
```

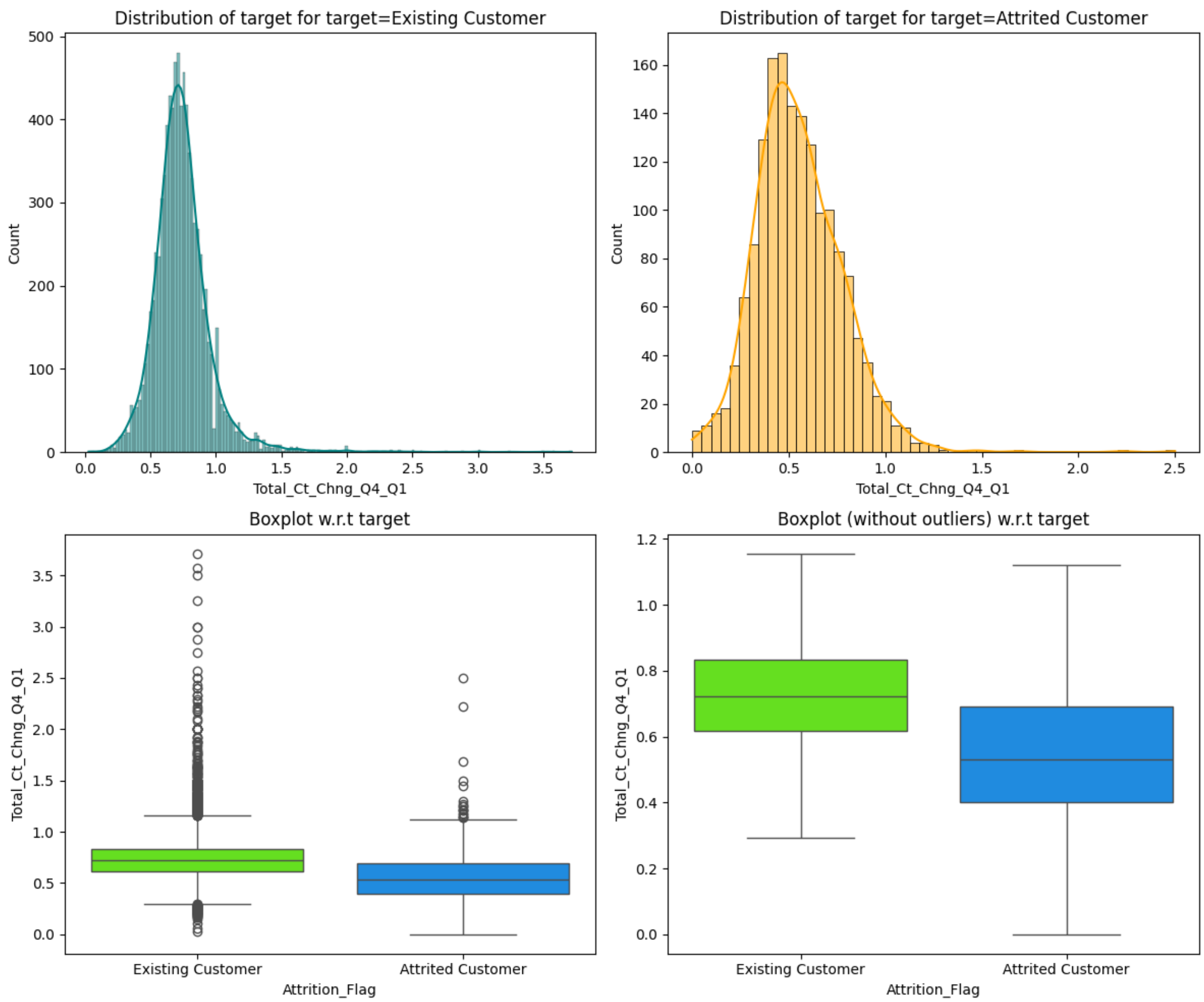


Observations

1. Dependent Count
- Attrition rates appear fairly consistent across different dependent counts, with a slight decrease as the dependent count increases. The number of dependents doesn't seem to significantly impact customer attrition.
1. Months Inactive in Last 12 Months
- Higher months of inactivity are associated with a higher attrition rate.
- This metric is a strong indicator of attrition. Customers with prolonged inactivity may be less engaged with the bank's services, signaling a risk for potential churn.
1. Income Category
- Attrition rates are relatively uniform across income categories, with slightly higher attrition rates for the lowest income bracket (Less than $40K).
- While income has some impact on attrition, it's not a dominant factor. However, customers in lower income brackets might be more price-sensitive
1. Education Level
- Attrition rates appear similar across different education levels, with only minor variations.

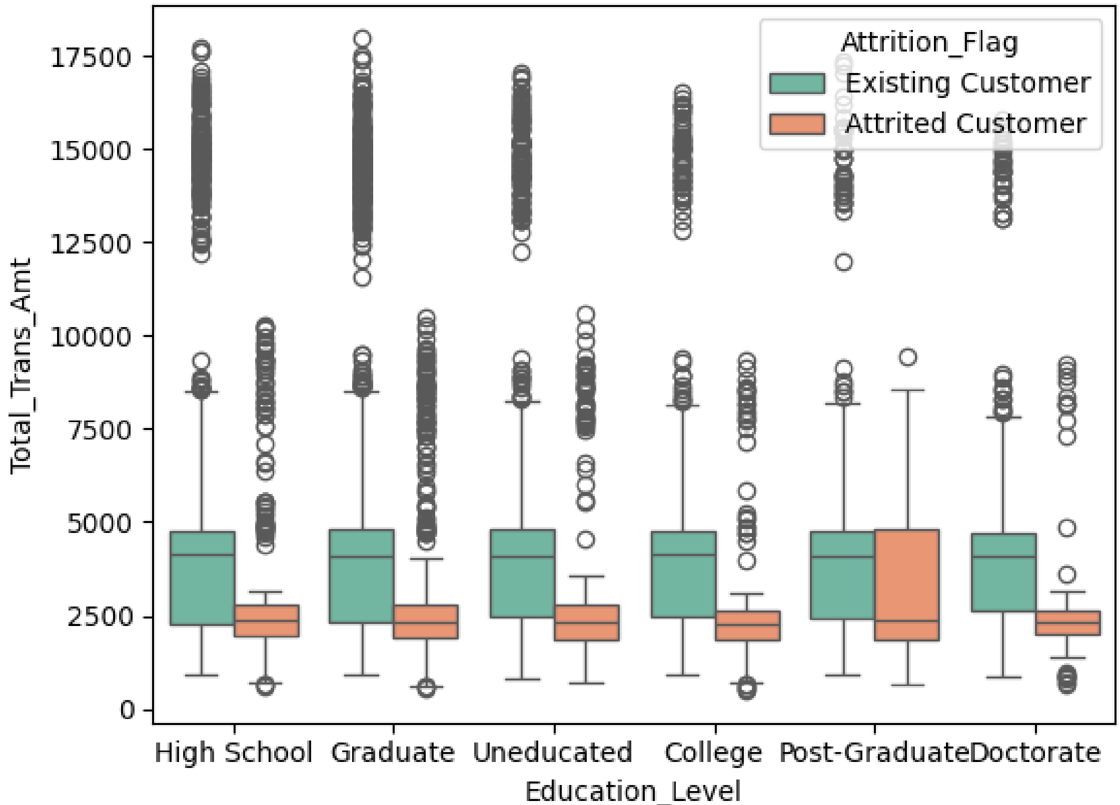
Bivariate Analysis of other variables

```
distribution_plot_wrt_target(data=data,
predictor='Total_Ct_Chng_Q4_Q1', target='Attrition_Flag')
```

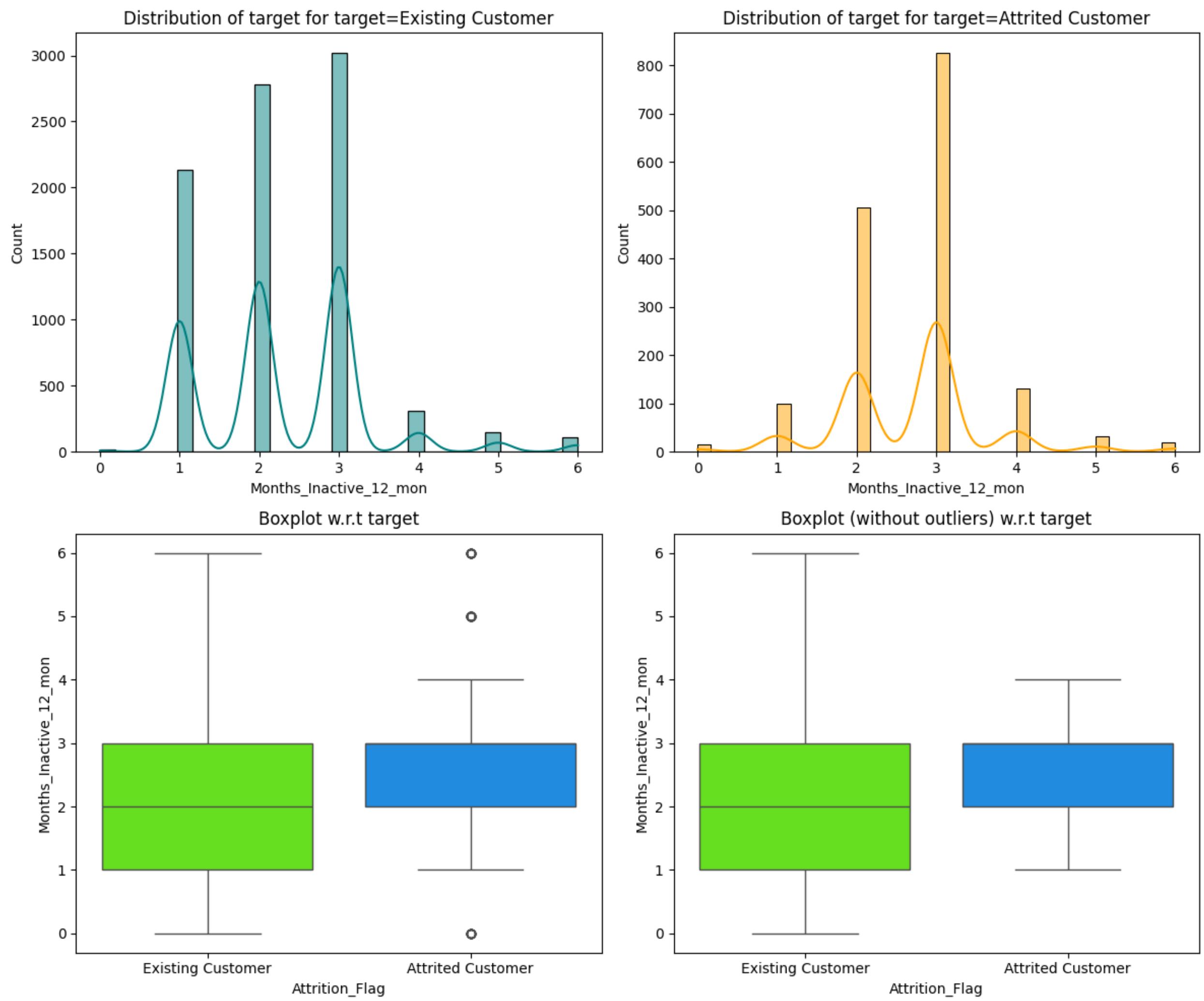



```
sns.boxplot(data=data, x='Education_Level', y='Total_Trans_Amt',
hue='Attrition_Flag', palette='Set2')
```

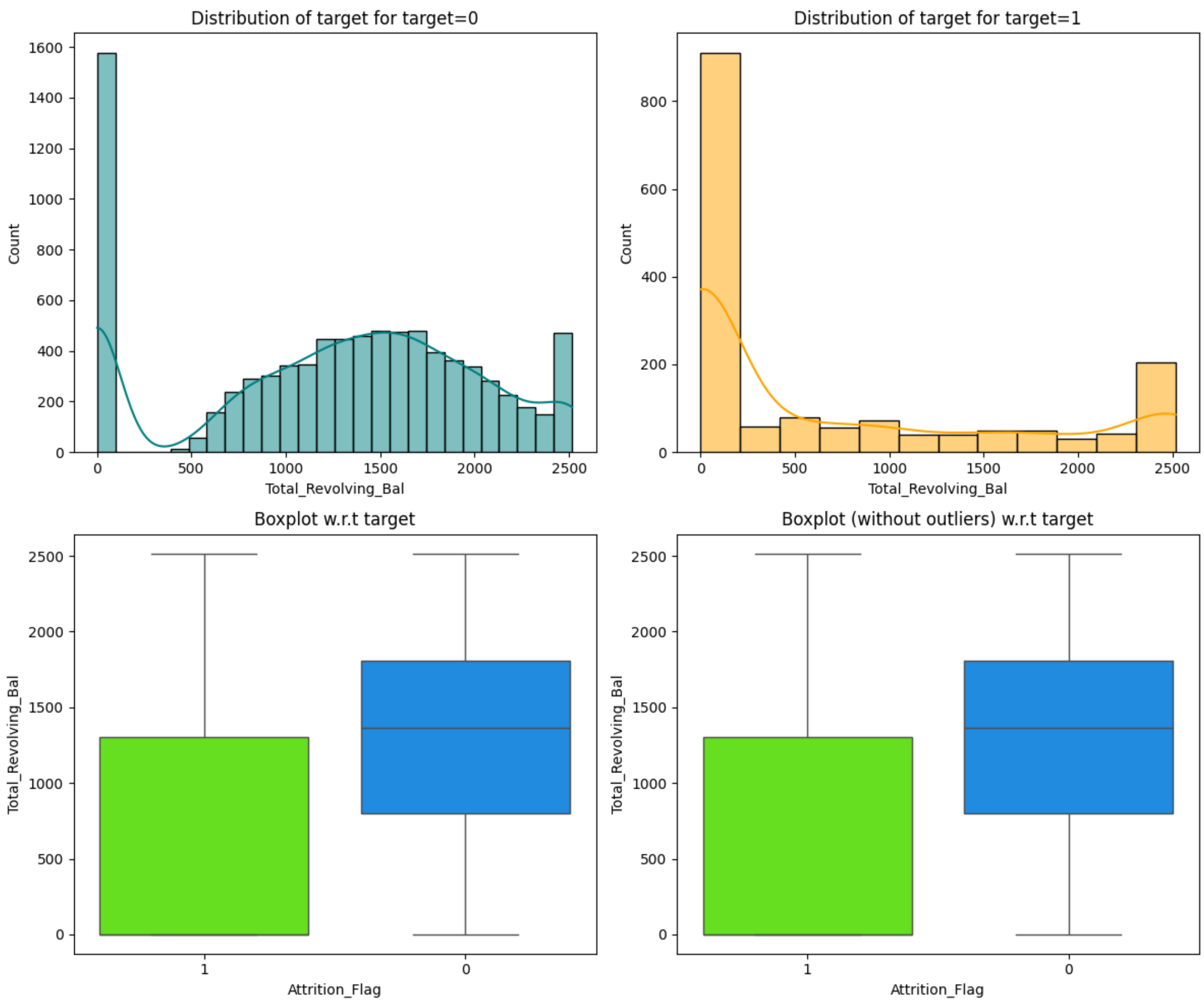
```
<Axes: xlabel='Education_Level', ylabel='Total_Trans_Amt'>
```



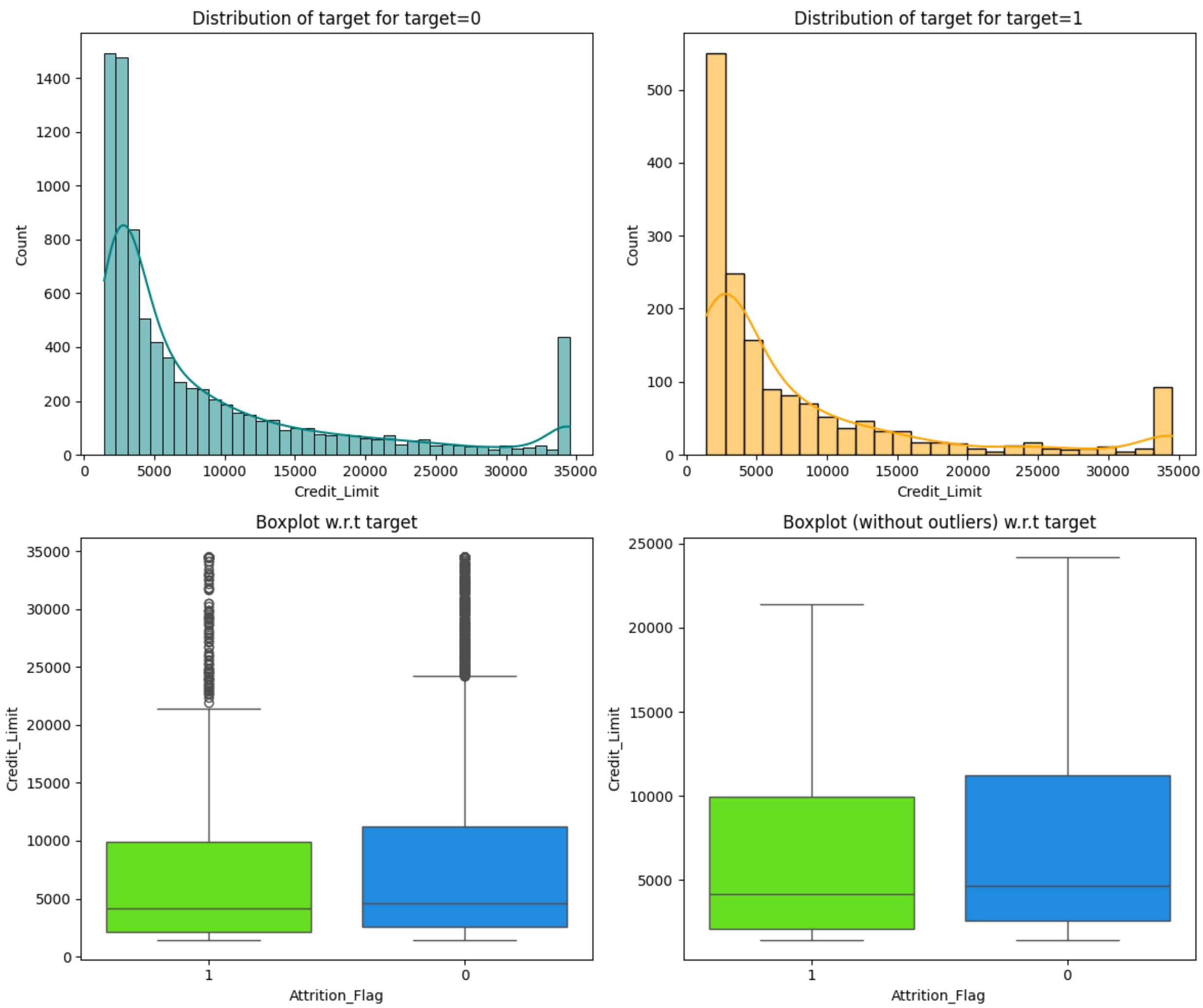

```
distribution_plot_wrt_target(data=data,
predictor='Months_Inactive_12_mon', target='Attrition_Flag')
```



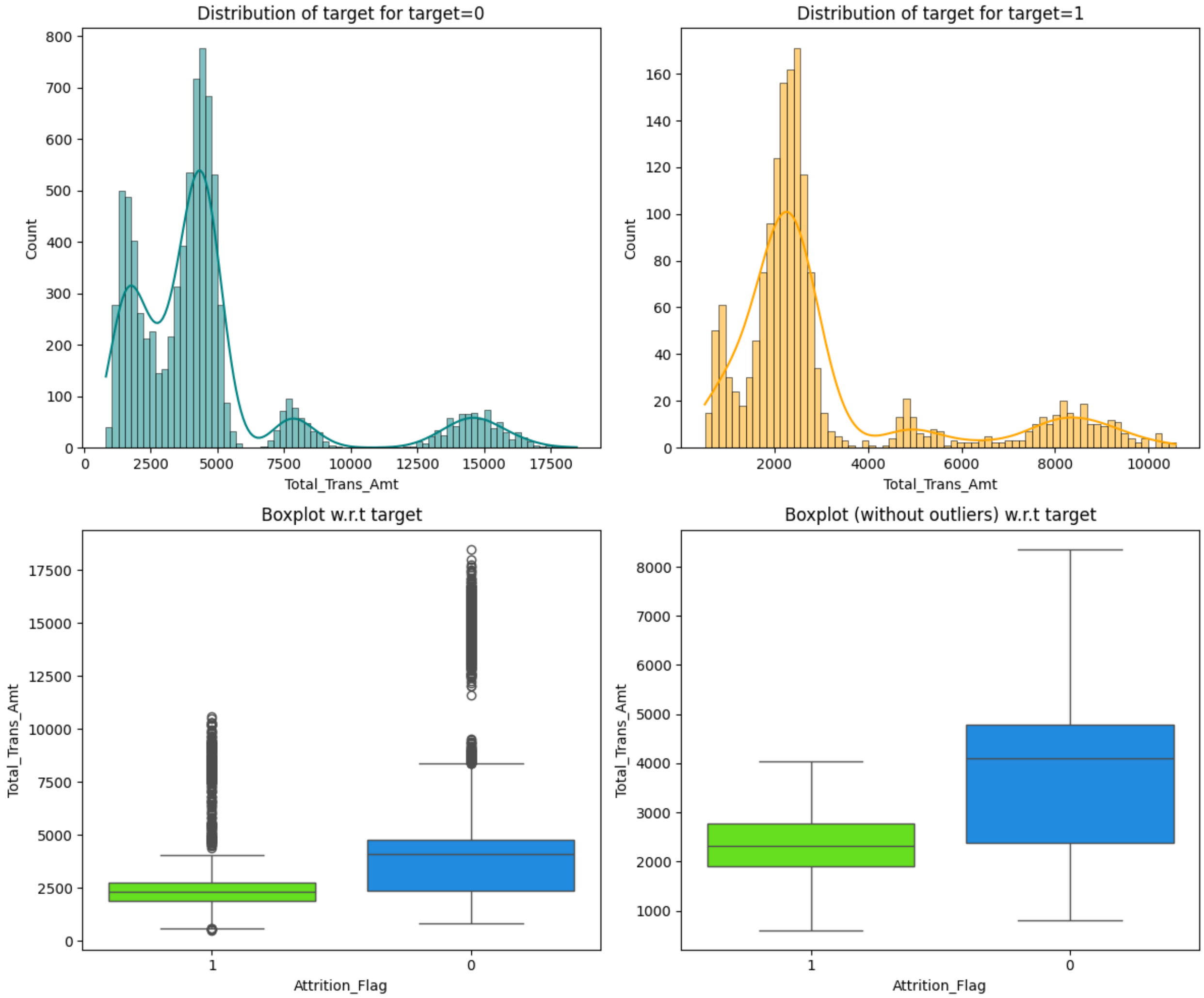

```python
distribution_plot_wrt_target(data, "Total_Revolving_Bal",
"Attrition_Flag")
```



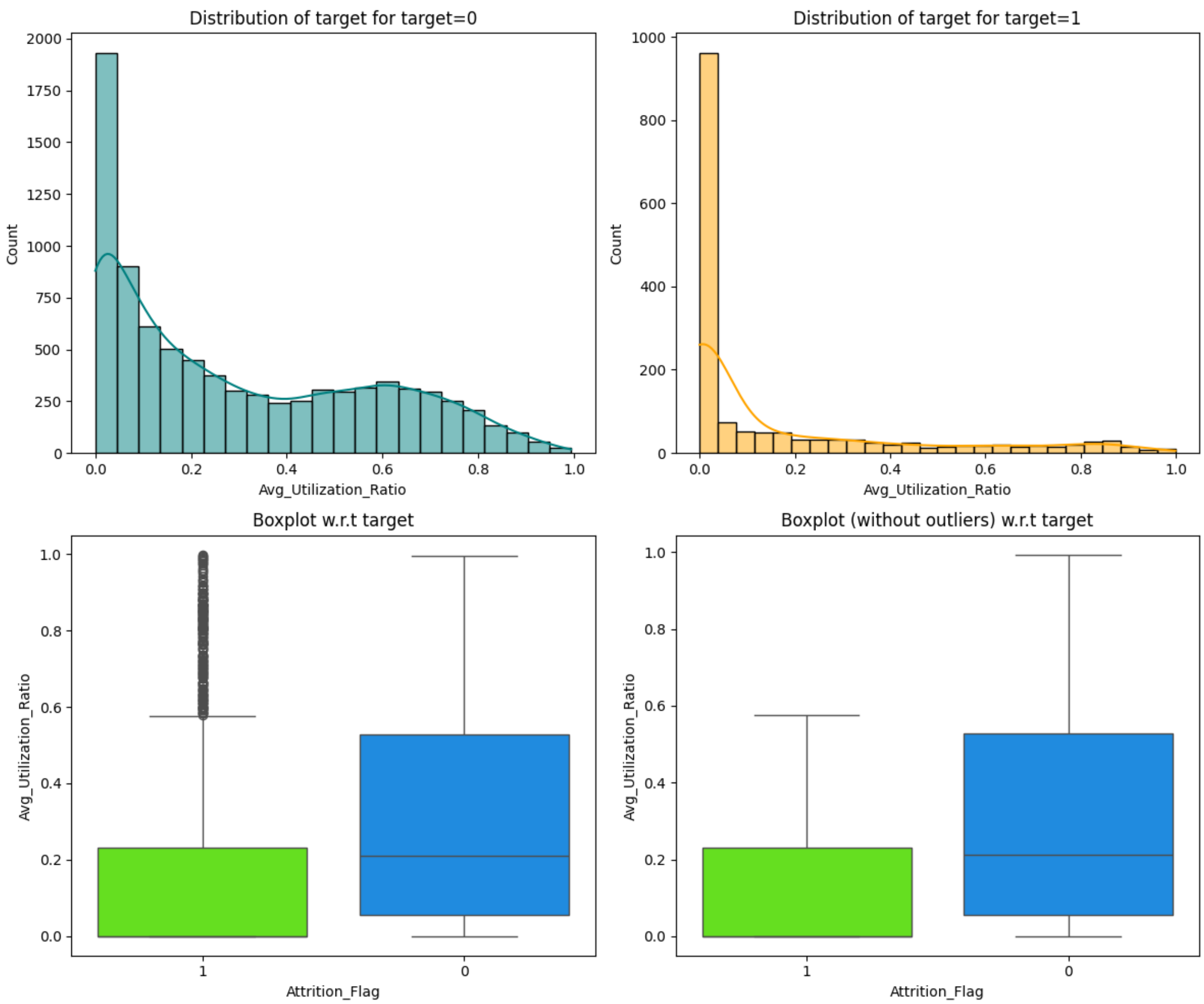

```
distribution_plot_wrt_target(data, "Credit_Limit", "Attrition_Flag")
```



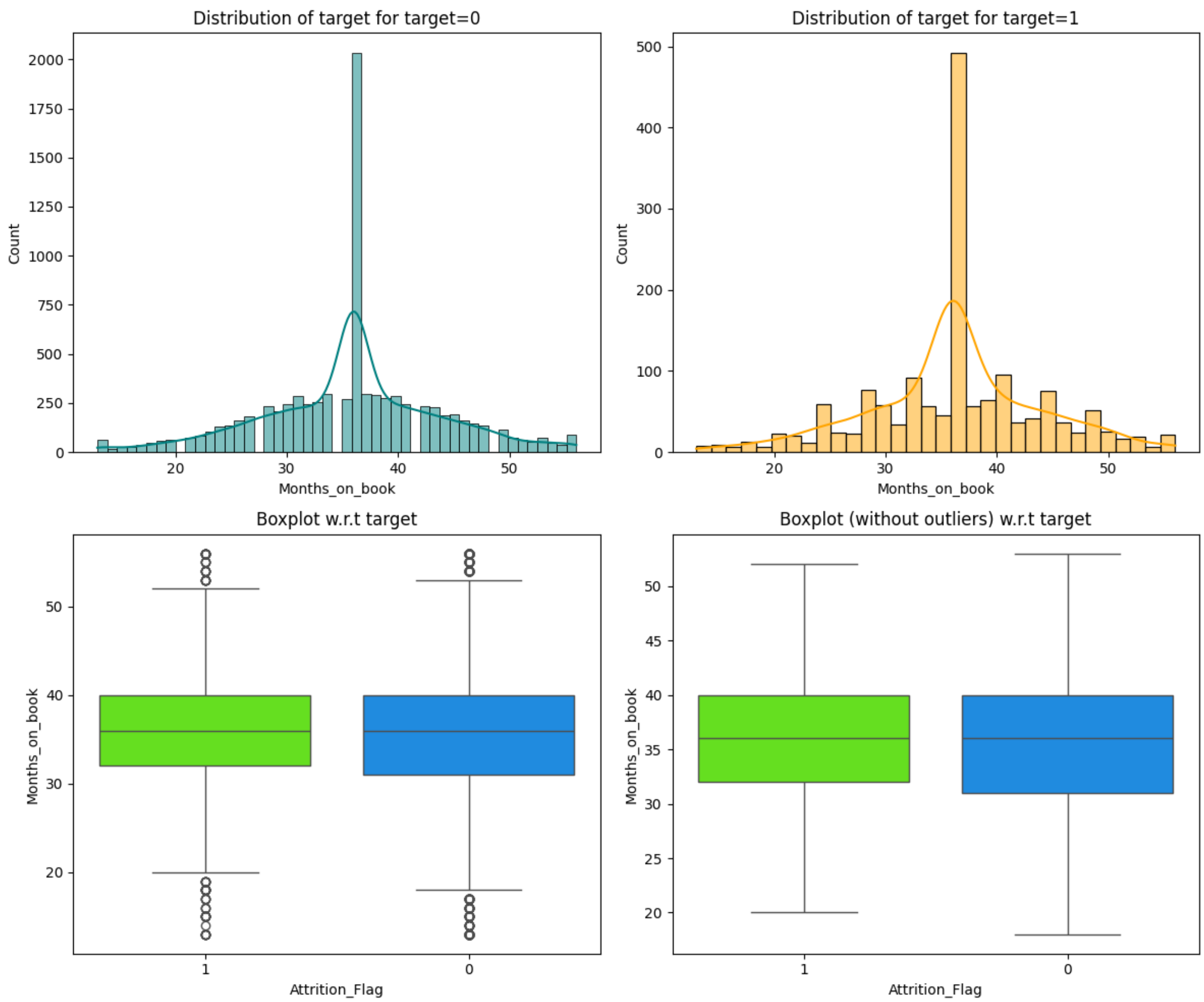

```
distribution_plot_wrt_target(data, "Total_Trans_Amt",
"Attrition_Flag")
```




```
distribution_plot_wrt_target(data, "Avg_Utilization_Ratio",
"Attrition_Flag")
```




```
distribution_plot_wrt_target(data, "Months_on_book", "Attrition_Flag")
```



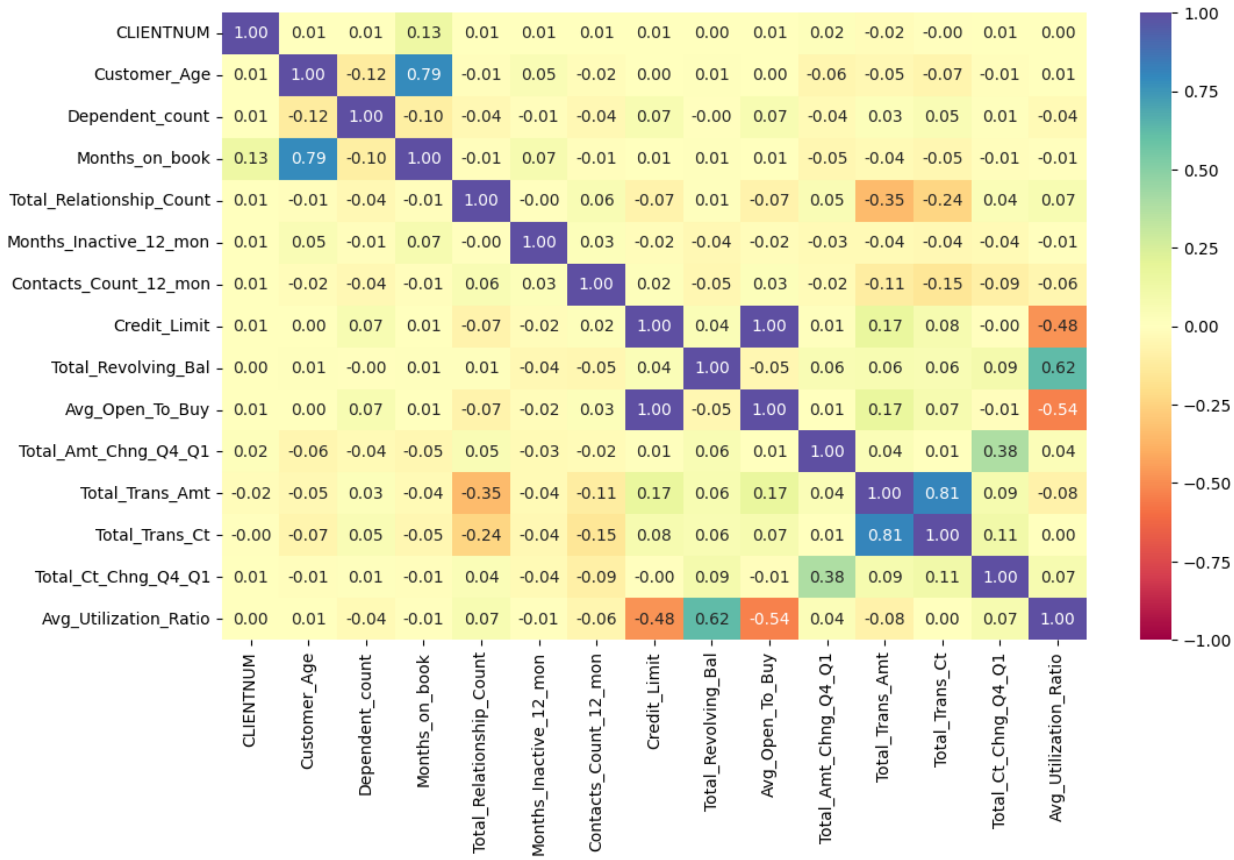
Observations

- The variable Total_Ct_Chng_Q4_Q1 seems to be a useful predictor of customer engagement. Existing customers tend to have higher changes in transaction counts, indicating active use of their credit card services. Attrited customers, on the other hand, display lower values, suggesting reduced or inconsistent usage.
- Months of Inactivity and Total Transaction Amount are strong indicators of customer engagement and potential attrition.
- Lower transaction amounts and higher inactivity both correlate with higher churn rates.
- Income and Education Levels might offer minor insights but are less clear-cut indicators of churn compared to other metrics.
- Total Revolving Balance: Existing customers have a wider distribution across Total Revolving Balance values, while attrited customers have a higher concentration at lower balances.

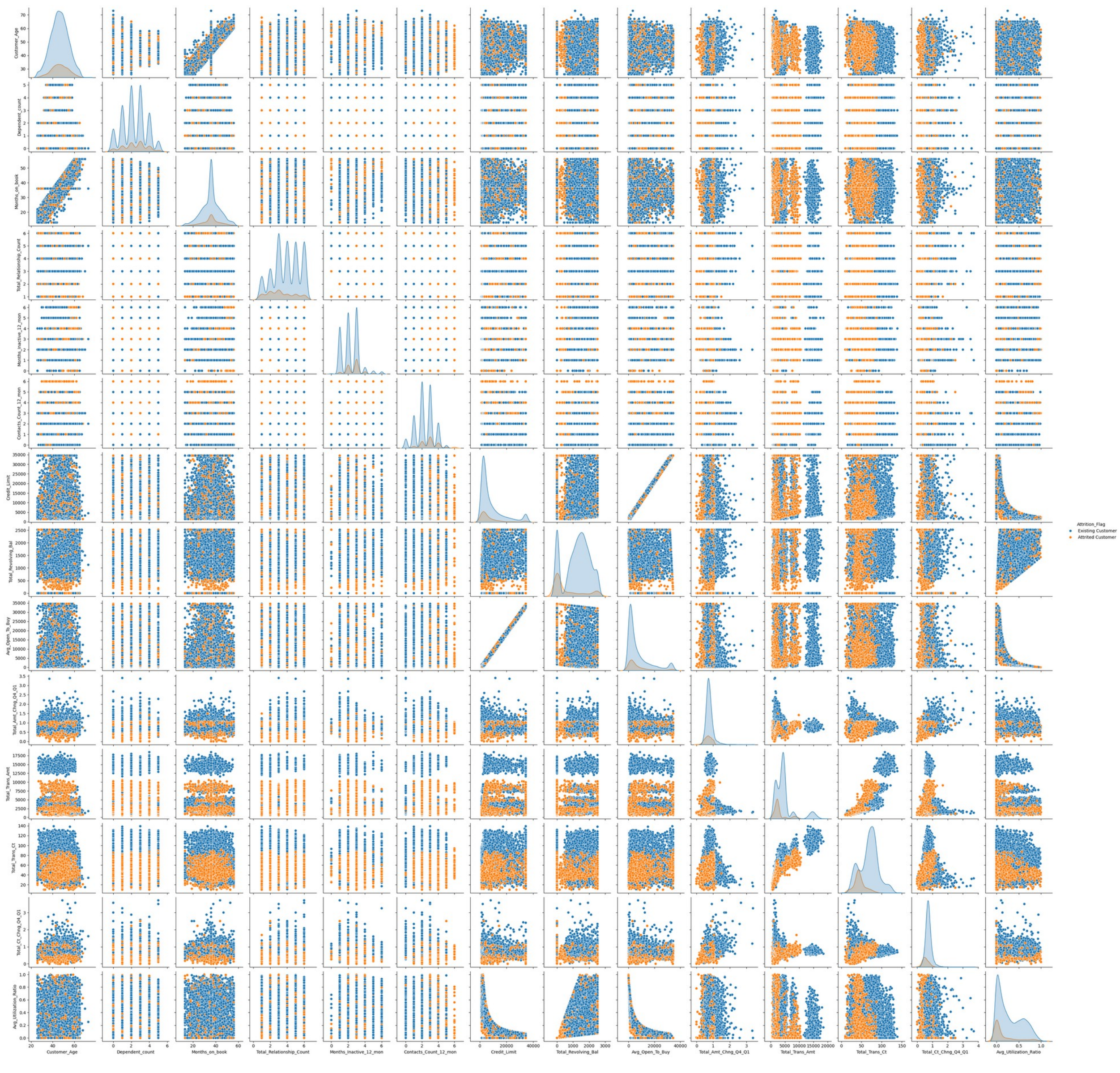
- Credit Limit: Both existing and attrited customers show a similar distribution for Credit Limit, though attrited customers have slightly lower credit limits on average.
- Total Transaction Amount: Existing customers generally have higher Total Transaction Amounts than attrited customers, with a wider range of spending behavior.
- Average Utilization Ratio: Existing customers have a higher utilization ratio on average, with a significant concentration at low utilization levels among attrited customers.
- Months on Book: The distribution of months on book is similar for both existing and attrited customers, with most customers around the 36–40 month mark.

Corelations & Pairplot

```
# Generating heatmap for all numerical veriables
# defining the size of the plot
plt.figure(figsize=(12, 7))

# plotting the heatmap for correlation
sns.heatmap(

data[data.select_dtypes(include='number').columns.tolist()].corr(),ann
ot=True, vmin=-1, vmax=1, fmt=".2f", cmap="Spectral"
);
```




```
# Generating pairplot plot for continius veriables with
hue='Attrition_Flag' <- target variable.
features_to_plot = ['Customer_Age', 'Dependent_count',
'Months_on_book',  'Total_Relationship_Count',
                    'Months_Inactive_12_mon', 'Contacts_Count_12_mon',
'Credit_Limit',
                    'Total_Revolving_Bal',  'Avg_Open_To_Buy',
'Total_Amt_Chng_Q4_Q1', 'Total_Trans_Amt',
                    'Total_Trans_Ct', 'Total_Ct_Chng_Q4_Q1',
'Avg_Utilization_Ratio']
plt.figure(figsize=(5, 6))
sns.pairplot(data, vars=features_to_plot , hue='Attrition_Flag',
diag_kind='kde');
plt.show()
```



Observations

Strong Correlations(Positive & Nagetive):

- ***Total_Trans_Amt and Total_Trans_Ct:*** Correlation of 0.81 indicates that the total transaction amount is highly correlated with the total transaction count. This makes sense, as more transactions typically lead to a higher transaction amount.
- ***Avg_Open_To_Buy and Credit_Limit:*** Correlation of 0.62 shows a moderate positive relationship. Since Avg_Open_To_Buy is the remaining available credit, a higher credit limit typically results in a higher average open-to-buy amount.
- ***Avg_Utilization_Ratio and Credit_Limit:*** Correlation of -0.48 suggests an inverse relationship, where customers with higher credit limits tend to use a smaller portion of their available credit.

- ***Total_Relationship_Count and Total_Trans_Amt:*** Correlation of -0.35 indicates that customers with more products or relationships with the bank tend to have lower total transaction amounts on the credit card.
- ***Customer_Age and Months_on_book:*** Correlation of 0.79 shows that older customers tend to have longer relationships with the bank.

Key tekeways from the pairplot

- Variables like ***Total_Trans_Amt***, ***Total_Trans_Ct***, and ***Months_Inactive_12_mon*** show clear separations between existing and attrited customers, making them valuable features for churn prediction.
- Credit Utilization (via ***Avg_Utilization_Ratio***) could be an important factor, especially for identifying customers at financial risk, as higher utilization appears more prevalent among attrited customers.
- Age and Tenure (***Customer_Age*** and ***Months_on_book***) are correlated but could be useful in identifying loyalty trends, with younger customers potentially at higher risk of churn.

Missing value imputation

```
# Checking missing values in the given data set.
data.isnull().sum()
```

```
CLIENTNUM                      0
Attrition_Flag                 0
Customer_Age                   0
Gender                         0
Dependent_count                0
Education_Level             1519
Marital_Status               749
Income_Category                0
Card_Category                  0
Months_on_book                 0
Total_Relationship_Count       0
Months_Inactive_12_mon         0
Contacts_Count_12_mon          0
Credit_Limit                   0
Total_Revolving_Bal            0
Avg_Open_To_Buy                0
Total_Amt_Chng_Q4_Q1           0
Total_Trans_Amt                0
Total_Trans_Ct                 0
Total_Ct_Chng_Q4_Q1            0
Avg_Utilization_Ratio          0
dtype: int64
```

Observation

- Two columns 'Education_Level' and 'Marital_Status' have missing values

- We need to dig deeper into these two columns and find the best strategy to impute the missing values

```
# Checking all unique values in the Marital_Status
data['Marital_Status'].unique()
```

```
array(['Married', 'Single', nan, 'Divorced'], dtype=object)
```

```
# Isolating the missing values
m_status_missing_data = data[data['Marital_Status'].isnull()]
# Considaring that Age is one of the factor to predict Marital Status, checking for details
m_status_missing_data.describe(include='all').T
```


```
{"summary":"{\n  \"name\": \"m_status_missing_data\",\n  \"rows\": 21,\n  \"fields\": [\n    {\n      \"column\": \"count\",\n      \"properties\": {\n        \"dtype\": \"date\",\n        \"min\": \"0\",\n        \"max\": 749.0,\n        \"num_unique_values\": 3,\n        \"samples\": [\n          749.0,\n          \"635\",\n          \"0\"\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"unique\",\n      \"properties\": {\n        \"dtype\": \"date\",\n        \"min\": 0,\n        \"max\": 6,\n        \"num_unique_values\": 4,\n        \"samples\": [\n          6,\n          4,\n          2\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"top\",\n      \"properties\": {\n        \"dtype\": \"category\",\n        \"num_unique_values\": 5,\n        \"samples\": [\n          \"F\",\n          \"Blue\",\n          \"Graduate\"\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"freq\",\n      \"properties\": {\n        \"dtype\": \"date\",\n        \"min\": \"227\",\n        \"max\": \"683\",\n        \"num_unique_values\": 5,\n        \"samples\": [\n          \"380\",\n          \"683\",\n          \"227\"\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"mean\",\n      \"properties\": {\n        \"dtype\": \"date\",\n        \"min\": 0.2558985313751669,\n        \"max\": 740584061.9719626,\n        \"num_unique_values\": 15,\n        \"samples\": [\n          8287.534178905207,\n          4720.0053404539385,\n          740584061.9719626\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"std\",\n      \"properties\": {\n        \"dtype\": \"date\",\n        \"min\": 0.19096337879554315,\n        \"max\": 37829864.7570304,\n        \"num_unique_values\": 15,\n        \"samples\": [\n          9585.01536355187,\n          3304.6860202574167,\n          37829864.7570304\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"min\",\n      \"properties\": {\n        \"dtype\": \"date\",\n        \"min\": 0.0,\n        \"max\": 708095133.0,\n        \"num_unique_values\": 11,\n        \"samples\": [\n          1438.3,\n          708095133.0,\n          15.0\
```

```
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"25%\",\n      \"properties\": {\n        \"dtype\": \"date\",\n
\"min\": 0.0,\n        \"max\": 713058858.0,\n
\"num_unique_values\": 11,\n        \"samples\": [\n          0.0,\n
713058858.0,\n          49.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"50%\",\n      \"properties\": {\n        \"dtype\":
\"date\",\n        \"min\": 0.15,\n        \"max\": 718388733.0,\n
\"num_unique_values\": 14,\n        \"samples\": [\n          0.734,\n
69.0,\n          718388733.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"75%\",\n      \"properties\": {\n        \"dtype\":
\"date\",\n        \"min\": 0.47,\n        \"max\": 778275783.0,\n
\"num_unique_values\": 13,\n        \"samples\": [\n          0.815,\n
4887.0,\n          778275783.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"max\",\n      \"properties\": {\n        \"dtype\":
\"date\",\n        \"min\": 0.958,\n        \"max\": 828343083.0,\n
\"num_unique_values\": 12,\n        \"samples\": [\n          2.333,\n
131.0,\n          828343083.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe"}
```


```
# Considaring that the Customer_Age min 26 and max 65 who do not have
any Marital_Status.
# Considaring 30 as average age setting all missing Marital_status to
'Single' for customers below 30.
data.loc[(data['Marital_Status'].isnull()) & (data['Customer_Age'] <
30), 'Marital_Status'] = 'Single'

# Now for the rest of the missing values of Marital_Status checking
what is the distribution of 'Married' Vs 'Divorced'.
filtered_df = data[data['Marital_Status'].isin(['Married',
'Divorced'])]
# Get the distribution
m_distribution =
filtered_df['Marital_Status'].value_counts(normalize=True)
print(m_distribution)
```

```
Marital_Status
Married     0.862374
Divorced    0.137626
Name: proportion, dtype: float64
```

```
# Using m_distribution ratio imputing Marital_Status
#randomly between 'Married' & 'Divorced' to keep
# natural distribution in Marital_Status column
data['Marital_Status'] = data['Marital_Status'].apply(
    lambda x: np.random.choice(m_distribution.index,
```

```
p=m_distribution.values) if pd.isnull(x) else x
)
data['Marital_Status'].isnull().sum()
```

```
0
```

```
# Checking all unique values in the Education_Level
data['Education_Level'].unique()
```

```
array(['High School', 'Graduate', 'Uneducated', nan, 'College',
       'Post-Graduate', 'Doctorate'], dtype=object)
```

```
# Checking what is the distribution of Education_Level in the data-
set.
# Get the distribution
e_distribution = data['Education_Level'].value_counts(normalize=True)
print(e_distribution)
```

```
Education_Level
Graduate         0.363383
High School      0.233852
Uneducated       0.172746
College          0.117681
Post-Graduate    0.059944
Doctorate        0.052393
Name: proportion, dtype: float64
```

```
# Using e_distribution ratio imputing Education_Level
#randomly between all categories of education to keep
# natural distribution.
data['Education_Level'] = data['Education_Level'].apply(
    lambda x: np.random.choice(e_distribution.index,
p=e_distribution.values) if pd.isnull(x) else x
)
data['Education_Level'].isnull().sum()
```

```
0
```

Observation

- We treated/imputed missing values for Marital_Status to 'Single' When age is < 30
- For the rest of the rows, we treated/imputed missing values for Marital_Status based on the distribution derived from the data. Ie. {Married: 0.862374, Divorced: 0.137626}
- We treated/imputed missing values for Education_Level based on the distribution derived from the data. Ie. {Graduate: 0.363383, High School: 0.233852, Uneducated: 0.172746, College: 0.117681, Post-Graduate: 0.059944, Doctorate: 0.052393}

```
data.isnull().sum()
```

```
CLIENTNUM                   0
Attrition_Flag              0
```

```
Customer_Age                  0
Gender                        0
Dependent_count               0
Education_Level               0
Marital_Status                0
Income_Category               0
Card_Category                 0
Months_on_book                0
Total_Relationship_Count      0
Months_Inactive_12_mon        0
Contacts_Count_12_mon         0
Credit_Limit                  0
Total_Revolving_Bal           0
Avg_Open_To_Buy               0
Total_Amt_Chng_Q4_Q1          0
Total_Trans_Amt               0
Total_Trans_Ct                0
Total_Ct_Chng_Q4_Q1           0
Avg_Utilization_Ratio         0
dtype: int64
```

```
data.duplicated().sum()
```

```
0
```

Observation

- We imputed all missing values for Marital_Status & Education_Level in the data-set
- We do not have any duplicate values/rows in the data-set.

Data Pre-processing

```
# Saving a copy of the data after imputation
data.to_csv(path+'data_after_imputation')
```

Replacing categorical columns with numeric codes.

```
# Seperating out and Displaying all the categorical column in the data
for feature in data.columns: # Loop through all columns in the dataframe
    if data[feature].dtype == 'object': # Only apply for columns with categorical strings
        data[feature] = pd.Categorical(data[feature])# Replace strings with an integer
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10127 entries, 0 to 10126
Data columns (total 21 columns):
 #   Column                    Non-Null Count  Dtype
```

```
---  ------                    --------------  -----
 0   CLIENTNUM                 10127 non-null  int64
 1   Attrition_Flag            10127 non-null  category
 2   Customer_Age              10127 non-null  int64
 3   Gender                    10127 non-null  category
 4   Dependent_count           10127 non-null  int64
 5   Education_Level           10127 non-null  category
 6   Marital_Status            10127 non-null  category
 7   Income_Category           10127 non-null  category
 8   Card_Category             10127 non-null  category
 9   Months_on_book            10127 non-null  int64
 10  Total_Relationship_Count  10127 non-null  int64
 11  Months_Inactive_12_mon    10127 non-null  int64
 12  Contacts_Count_12_mon     10127 non-null  int64
 13  Credit_Limit              10127 non-null  float64
 14  Total_Revolving_Bal       10127 non-null  int64
 15  Avg_Open_To_Buy           10127 non-null  float64
 16  Total_Amt_Chng_Q4_Q1      10127 non-null  float64
 17  Total_Trans_Amt           10127 non-null  int64
 18  Total_Trans_Ct            10127 non-null  int64
 19  Total_Ct_Chng_Q4_Q1       10127 non-null  float64
 20  Avg_Utilization_Ratio     10127 non-null  float64
dtypes: category(6), float64(5), int64(10)
memory usage: 1.2 MB
```

```
print(data.Attrition_Flag.value_counts())
print(data.Gender.value_counts())
print(data.Education_Level.value_counts())
print(data.Marital_Status.value_counts())
print(data.Income_Category.value_counts())
print(data.Card_Category.value_counts())
```

```
Attrition_Flag
Existing Customer    8500
Attrited Customer    1627
Name: count, dtype: int64
Gender
F    5358
M    4769
Name: count, dtype: int64
Education_Level
Graduate         3666
High School      2351
Uneducated       1774
College          1197
Post-Graduate     605
Doctorate         534
Name: count, dtype: int64
Marital_Status
Married     5334
```

```
Single      3954
Divorced     839
Name: count, dtype: int64
Income_Category
Less than $40K    3561
$40K - $60K       1790
$80K - $120K      1535
$60K - $80K       1402
abc               1112
$120K +            727
Name: count, dtype: int64
Card_Category
Blue        9436
Silver       555
Gold         116
Platinum      20
Name: count, dtype: int64
```

```
replaceStruct = {
                 "Income_Category": {"Less than $40K":1, "$40K - 
$60K":2 , "$60K - $80K": 3, "$80K - $120K": 4,"$120K +": 5, "abc": 0},
                 "Attrition_Flag": {"Existing Customer":0, "Attrited 
Customer":1}
                }

# Replacing all cetogories with replacement values

data=data.replace(replaceStruct)
```

Data Splitting & Encoding

```
# Dropping CLIENTNUM column as this column is an ID column and has no 
significance in model building
data.drop("CLIENTNUM" , axis=1, inplace=True)

# Splitting the data between indipendent variables and target variable 
and making copies - i.e. X & y
# Leaving data as intact
y = data["Attrition_Flag"].copy()
X = data.drop("Attrition_Flag" , axis=1).copy()


oneHotCols=["Gender","Education_Level","Marital_Status","Income_Catego
ry", "Card_Category"]
X = pd.get_dummies(X, columns=oneHotCols)
# Setting all column data type as float after encoding
X = X.astype(float)

# Splitting the data into train, validation and test category with 
ratio 70:15:15 with stratify=y
# Initial split: train (70%) and temp (30%, for validation + test)
```

```
X_train, X_temp, y_train, y_temp = train_test_split(X, y,
test_size=0.3, random_state=1, stratify=y)

# Split temp into validation (15%) and test (15%)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.5, random_state=1)

# Making sure that the split data sustains a balance distribution on
Attrition_Flag 0(Existing Customer) and 1(Attrited Customer) count.
print("Shape of training set:", X_train.shape)
print("Shape of validation set:", X_val.shape)
print("Shape of test set:", X_test.shape, '\n')
print("Percentage of classes in training set:")
print(100*y_train.value_counts(normalize=True), '\n')
print("Percentage of classes in validation set:")
print(100*y_val.value_counts(normalize=True), '\n')
print("Percentage of classes in test set:")
print(100*y_test.value_counts(normalize=True))

Shape of training set: (7088, 35)
Shape of validation set: (1519, 35)
Shape of test set: (1520, 35)

Percentage of classes in training set:
Attrition_Flag
0    83.930587
1    16.069413
Name: proportion, dtype: float64

Percentage of classes in validation set:
Attrition_Flag
0    85.319289
1    14.680711
Name: proportion, dtype: float64

Percentage of classes in test set:
Attrition_Flag
0    82.565789
1    17.434211
Name: proportion, dtype: float64
```

Outlier detection & treatment.

```
# Calculating - 25th & 75th perentile of all numerical columns
Q1 = data.select_dtypes(include=["float64", "int64"]).quantile(0.25)
# To find the 25th percentile
Q3 = data.select_dtypes(include=["float64", "int64"]).quantile(0.75)
# To find the 75th percentile

# Inter Quantile Range (75th perentile - 25th percentile)
```

```
IQR = Q3 - Q1

# Finding lower and upper bounds. All values outside these bounds are
outliers
lower = (Q1 - 1.5 * IQR)
upper = (Q3 + 1.5 * IQR)

# Display outliers count based on the upper & lower bounds
((data.select_dtypes(include=["float64", "int64"]) < lower) |
(data.select_dtypes(include=["float64", "int64"]) > upper)).sum()
```

```
Customer_Age                  2
Dependent_count               0
Months_on_book              386
Total_Relationship_Count      0
Months_Inactive_12_mon      331
Contacts_Count_12_mon       629
Credit_Limit                984
Total_Revolving_Bal           0
Avg_Open_To_Buy             963
Total_Amt_Chng_Q4_Q1        396
Total_Trans_Amt             896
Total_Trans_Ct                2
Total_Ct_Chng_Q4_Q1         394
Avg_Utilization_Ratio         0
dtype: int64
```

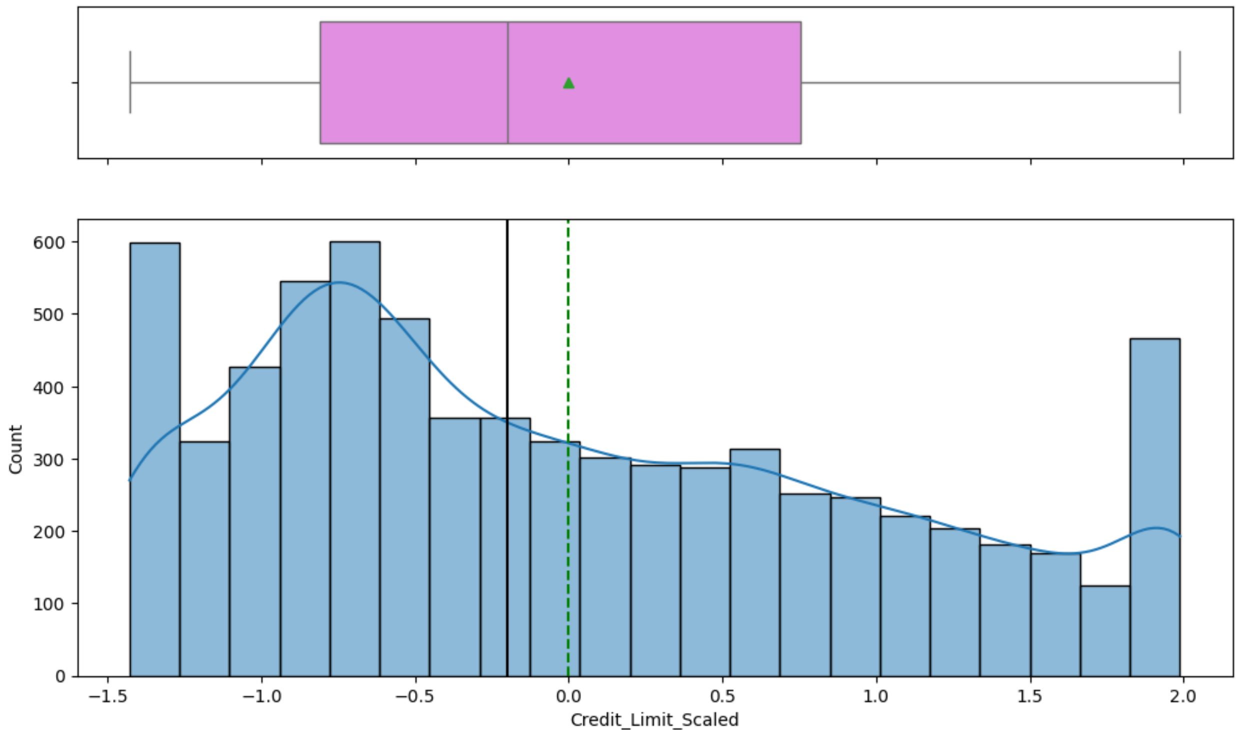
Observations

- As seen previously, we have some variables with high outliers, and the data distribution is significantly skewed or bimodal.
- After researching, found that Transformation and Scaling are the best options to handle those data set. (ref: data-processing)
- Doing further research based on Transformation-Scaling-And-Normalization, decided to do Transformation and Scaling for below columns
- Log Transformation:
  - Credit_Limit
  - Avg_Open_To_Buy
  - Total_Trans_Amt
- Scale the above-transformed variables by Standard Scaling
  - Credit_Limit
  - Avg_Open_To_Buy
  - Total_Trans_Amt
- Use Robust Scaling for the below column
  - Total_Trans_Ct

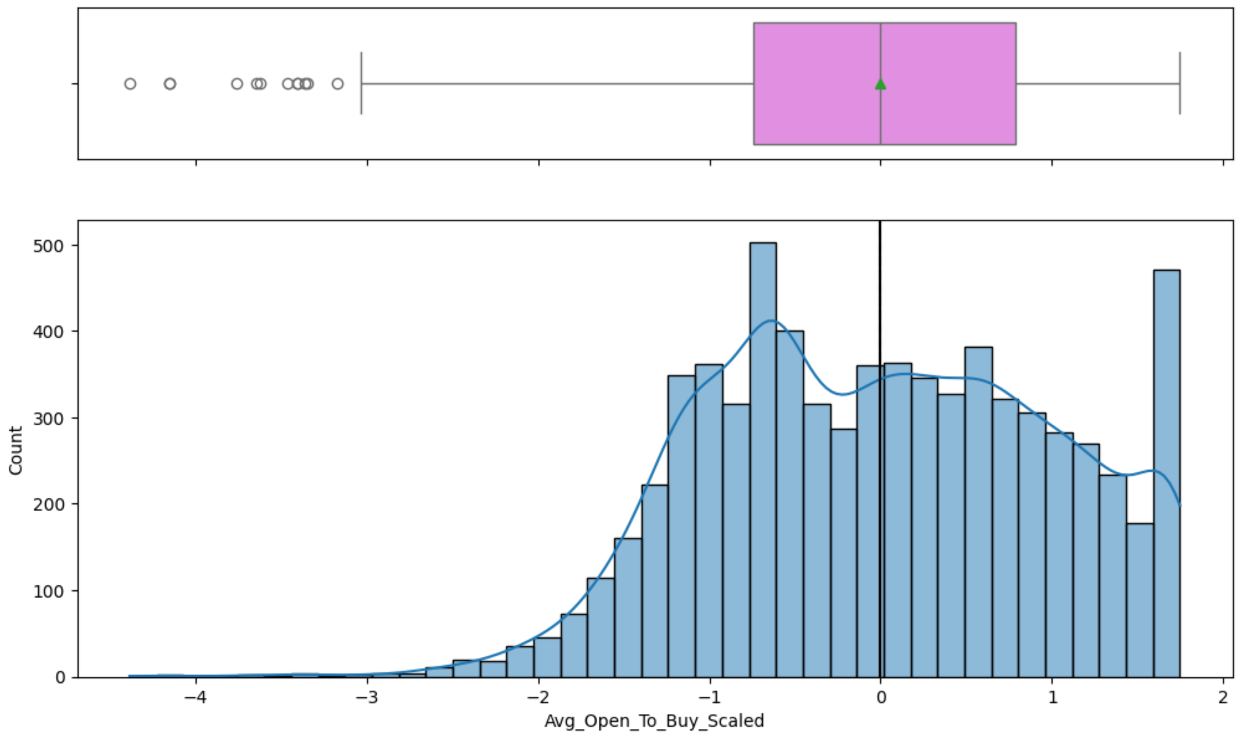
```
# Log transformation for skewed variables of training data
X_train['Credit_Limit_Log'] = np.log1p(X_train['Credit_Limit'])
```

```
X_train['Avg_Open_To_Buy_Log'] = np.log1p(X_train['Avg_Open_To_Buy'])
X_train['Total_Trans_Amt_Log'] = np.log1p(X_train['Total_Trans_Amt'])

# Initialize scalers
standard_scaler = StandardScaler()
robust_scaler = RobustScaler()

# Fit & Apply scaling to training set
X_train[['Credit_Limit_Scaled', 'Avg_Open_To_Buy_Scaled',
'Total_Trans_Amt_Scaled']] = standard_scaler.fit_transform(
        X_train[['Credit_Limit_Log', 'Avg_Open_To_Buy_Log',
'Total_Trans_Amt_Log']])

# Fit & Apply Robust Scaling to training set
X_train['Total_Trans_Ct_Scaled'] =
robust_scaler.fit_transform(X_train[['Total_Trans_Ct']])

# Log transformation for skewed variables of validation data
X_val['Credit_Limit_Log'] = np.log1p(X_val['Credit_Limit'])
X_val['Avg_Open_To_Buy_Log'] = np.log1p(X_val['Avg_Open_To_Buy'])
X_val['Total_Trans_Amt_Log'] = np.log1p(X_val['Total_Trans_Amt'])

# We only transform the data and will not fit again to avoid data
leakage
# Apply scaling to validation set
X_val[['Credit_Limit_Scaled', 'Avg_Open_To_Buy_Scaled',
'Total_Trans_Amt_Scaled']] = standard_scaler.transform(
        X_val[['Credit_Limit_Log', 'Avg_Open_To_Buy_Log',
'Total_Trans_Amt_Log']])

# Apply Robust Scaling to validation set
X_val['Total_Trans_Ct_Scaled'] =
robust_scaler.transform(X_val[['Total_Trans_Ct']])

# Log transformation for skewed variables of test data
X_test['Credit_Limit_Log'] = np.log1p(X_test['Credit_Limit'])
X_test['Avg_Open_To_Buy_Log'] = np.log1p(X_test['Avg_Open_To_Buy'])
X_test['Total_Trans_Amt_Log'] = np.log1p(X_test['Total_Trans_Amt'])

# We only transform the data and will not fit again to avoid data
leakage
# Apply scaling to test set
X_test[['Credit_Limit_Scaled', 'Avg_Open_To_Buy_Scaled',
'Total_Trans_Amt_Scaled']] = standard_scaler.transform(
        X_test[['Credit_Limit_Log', 'Avg_Open_To_Buy_Log',
'Total_Trans_Amt_Log']])

# Apply Robust Scaling to test set
X_test['Total_Trans_Ct_Scaled'] =
robust_scaler.transform(X_test[['Total_Trans_Ct']])
```

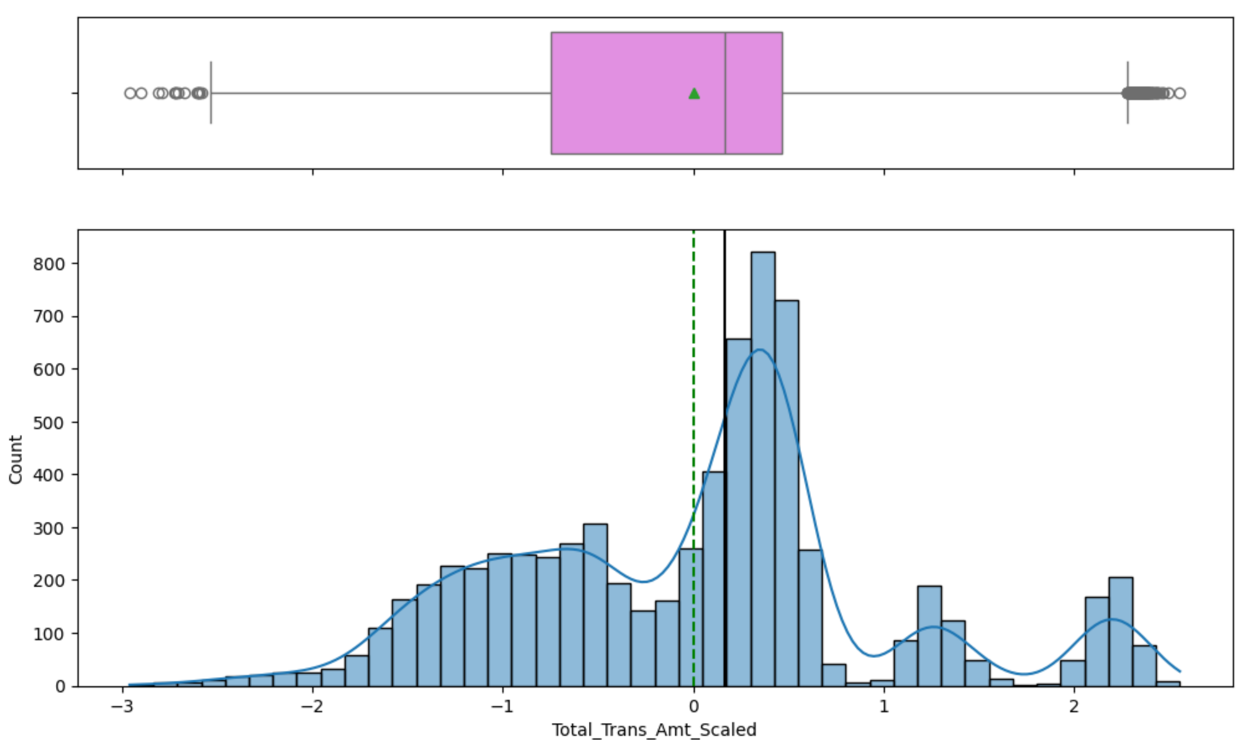
```
# Display histogram & boxplot for Credit_Limit_Scaled of training set
histogram_boxplot(data=X_train, feature='Credit_Limit_Scaled',
kde=True)
```



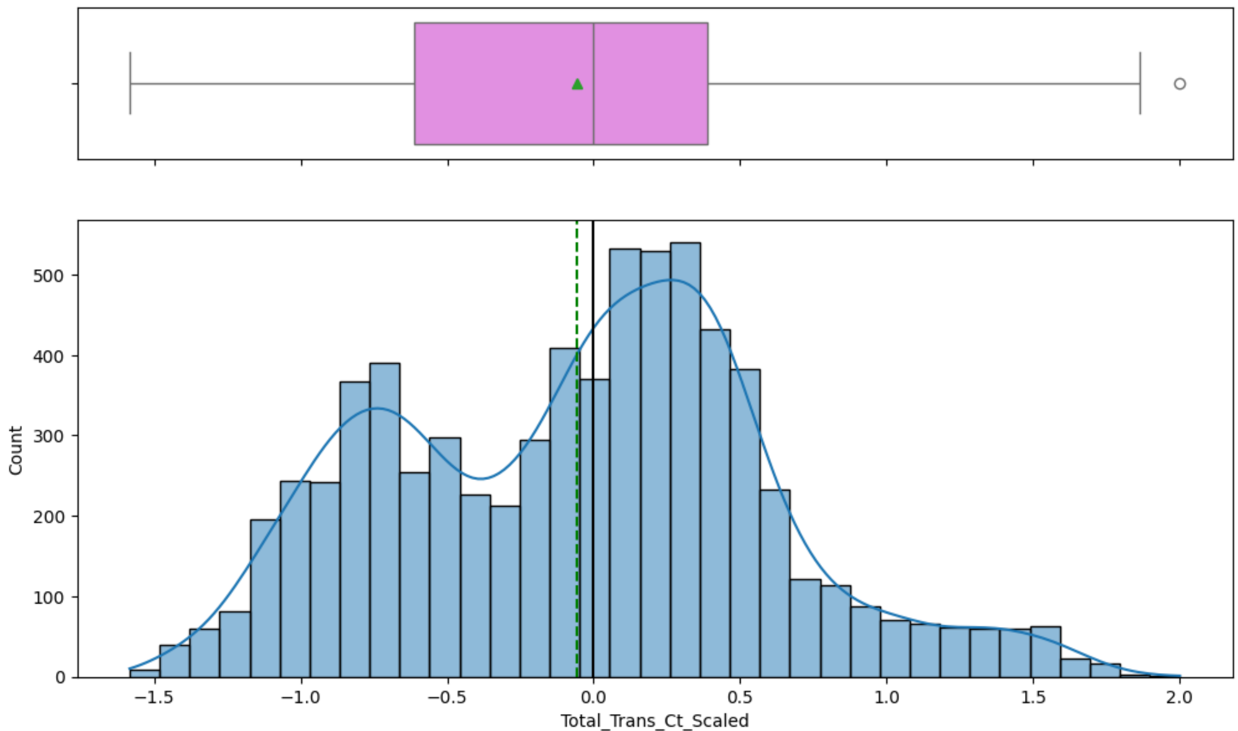

```
# Display histogram & boxplot for Avg_Open_To_Buy_Scaled of training
set
histogram_boxplot(data=X_train, feature='Avg_Open_To_Buy_Scaled',
kde=True)
```




```python
# Display histogram & boxplot for Total_Trans_Amt_Scaled of training set
histogram_boxplot(data=X_train, feature='Total_Trans_Amt_Scaled', kde=True)
```



```
# Display histogram & boxplot for Total_Trans_Ct_Scaled of training
set
histogram_boxplot(data=X_train, feature='Total_Trans_Ct_Scaled',
kde=True)
```




```
# Droping unnecessary columns from Training Dataset
X_train = X_train.drop(['Credit_Limit', 'Avg_Open_To_Buy',
'Total_Trans_Amt',
                  'Credit_Limit_Log', 'Avg_Open_To_Buy_Log',
'Total_Trans_Amt_Log', 'Total_Trans_Ct'], axis=1)

# Droping unnecessary columns from Validation Dataset
X_val = X_val.drop(['Credit_Limit', 'Avg_Open_To_Buy',
'Total_Trans_Amt',
                  'Credit_Limit_Log', 'Avg_Open_To_Buy_Log',
'Total_Trans_Amt_Log', 'Total_Trans_Ct'], axis=1)

# Droping unnecessary columns from Testing Dataset
X_test = X_test.drop(['Credit_Limit', 'Avg_Open_To_Buy',
'Total_Trans_Amt',
                  'Credit_Limit_Log', 'Avg_Open_To_Buy_Log',
'Total_Trans_Amt_Log', 'Total_Trans_Ct'], axis=1)

#Describing final training data-set to check.
X_train.describe().T
```

```
{"summary":"{\n  \"name\": \"X_train\",\n  \"rows\": 35,\n
\"fields\": [\n    {\n      \"column\": \"count\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.0,\n        \"min\": 7088.0,\n        \"max\": 7088.0,\n
\"num_unique_values\": 1,\n        \"samples\": [\n          7088.0\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"mean\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\":
195.89349085944585,\n        \"min\": -0.058236142463004764,\n
\"max\": 1160.447658013544,\n        \"num_unique_values\": 35,\n
\"samples\": [\n          0.11060948081264109\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"std\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 137.57113543888556,\n
\"min\": 0.044402096337176976,\n        \"max\": 814.7672941128471,\n
\"num_unique_values\": 34,\n        \"samples\": [\n
0.24054211297085504\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"min\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 5.010034701017514,\n        \"min\": -4.384617381391351,\n
\"max\": 26.0,\n        \"num_unique_values\": 8,\n
\"samples\": [\n          0.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"25%\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 48.120517706298195,\n        \"min\": -
0.810660137178927,\n        \"max\": 282.5,\n
\"num_unique_values\": 14,\n        \"samples\": [\n          0.0\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"50%\",\n      \"properties\": {\
n        \"dtype\": \"number\",\n        \"std\": 216.5173644227672,\n
\"min\": -0.19830465342554424,\n        \"max\": 1282.5,\n
\"num_unique_values\": 13,\n        \"samples\": [\n          -
0.006635616584503974\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"75%\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 300.82205050636253,\n        \"min\": 0.0,\n        \"max\":
1782.0,\n        \"num_unique_values\": 14,\n        \"samples\": [\n
0.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"max\",\n      \"properties\": {\n        \"dtype\": \"number\",\n
\"std\": 424.77253735601226,\n        \"min\": 0.995,\n
\"max\": 2517.0,\n        \"num_unique_values\": 13,\n
\"samples\": [\n          2.5553546436894883\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe"}
```


```
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 7088 entries, 4124 to 4752
```

```
Data columns (total 35 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Customer_Age                    7088 non-null   float64
 1   Dependent_count                 7088 non-null   float64
 2   Months_on_book                  7088 non-null   float64
 3   Total_Relationship_Count        7088 non-null   float64
 4   Months_Inactive_12_mon          7088 non-null   float64
 5   Contacts_Count_12_mon           7088 non-null   float64
 6   Total_Revolving_Bal             7088 non-null   float64
 7   Total_Amt_Chng_Q4_Q1            7088 non-null   float64
 8   Total_Ct_Chng_Q4_Q1             7088 non-null   float64
 9   Avg_Utilization_Ratio           7088 non-null   float64
 10  Gender_F                        7088 non-null   float64
 11  Gender_M                        7088 non-null   float64
 12  Education_Level_College         7088 non-null   float64
 13  Education_Level_Doctorate       7088 non-null   float64
 14  Education_Level_Graduate        7088 non-null   float64
 15  Education_Level_High School     7088 non-null   float64
 16  Education_Level_Post-Graduate   7088 non-null   float64
 17  Education_Level_Uneducated      7088 non-null   float64
 18  Marital_Status_Divorced         7088 non-null   float64
 19  Marital_Status_Married          7088 non-null   float64
 20  Marital_Status_Single           7088 non-null   float64
 21  Income_Category_5               7088 non-null   float64
 22  Income_Category_2               7088 non-null   float64
 23  Income_Category_3               7088 non-null   float64
 24  Income_Category_4               7088 non-null   float64
 25  Income_Category_1               7088 non-null   float64
 26  Income_Category_0               7088 non-null   float64
 27  Card_Category_Blue              7088 non-null   float64
 28  Card_Category_Gold              7088 non-null   float64
 29  Card_Category_Platinum          7088 non-null   float64
 30  Card_Category_Silver            7088 non-null   float64
 31  Credit_Limit_Scaled             7088 non-null   float64
 32  Avg_Open_To_Buy_Scaled          7088 non-null   float64
 33  Total_Trans_Amt_Scaled          7088 non-null   float64
 34  Total_Trans_Ct_Scaled           7088 non-null   float64
dtypes: float64(35)
memory usage: 1.9 MB
```

```
y_train.head()
```

```
4124    0
4686    0
1276    0
6119    0
2253    0
Name: Attrition_Flag, dtype: category
Categories (2, int64): [1, 0]
```

```
# Saving all the data sets so that we can read them back if required, later.
X_train.to_csv(path+'X_train')
X_val.to_csv(path+'X_val')
X_test.to_csv(path+'X_test')
y_train.to_csv(path+'y_train')
y_val.to_csv(path+'y_val')
y_test.to_csv(path+'y_test')
```

Model Building

Model evaluation criterion

Model Evaluation Considarations

Identifying All Potential Churners: Recall measures the model's ability to capture all actual churners (those who truly intend to leave). A high recall means the model successfully identifies most customers at risk of leaving, allowing the bank to take preventive action to retain them.

Using Recall as a primary metric, supplemented by Precision and F1 Score, would give a balanced view:

- Recall: To ensure we are catching the majority of potential churners.
- Precision: To gauge the accuracy of predictions when the bank reaches out to prevent churn.
- F1 Score: For an overall measure that balances recall and precision.

This approach should help the bank focus on retaining as many potential churners as possible, while still keeping the cost of false positives under control.

Functions for analysis

Let's define a function to output different metrics (including recall) on the train and test set and a function to show confusion matrix so that we do not have to use the same code repetitively while evaluating models.

```
# Defining a function to compute different metrics to check performance of a classification model built using sklearn
def train_models_and_display_metrics(models, X_train, y_train, X_test, y_test, validationType):
  print("\n" "Model Performance:" "\n")
  for name, model in models:
      model.fit(X_train, y_train)
      accurecy = metrics.accuracy_score(y_train, model.predict(X_train))
      recall = metrics.recall_score(y_train, model.predict(X_train))
      precision = metrics.precision_score(y_train, model.predict(X_train))
      f1score = metrics.f1_score(y_train, model.predict(X_train))
```

```
      taccurecy = metrics.accuracy_score(y_test,
model.predict(X_test))
      trecall = metrics.recall_score(y_test, model.predict(X_test))
      tprecision = metrics.precision_score(y_test,
model.predict(X_test))
      tf1score = metrics.f1_score(y_test, model.predict(X_test))

      print("-" * 71)
      print("{}".format(name))
      print("-" * 71)
      df = pd.DataFrame([['TRAINING', accurecy, recall, precision,
f1score],[validationType, taccurecy, trecall, tprecision, tf1score]],
columns=[' ', 'Accuracy', 'Recall', 'Precision', 'F1'])
      print(df.to_string(index=False))

# Importance of features in the tree building
def display_importance_of_features(model, feature_names):
  importances = model.feature_importances_
  df_imp = pd.DataFrame(importances, index=feature_names)
  df_imp = df_imp[df_imp[0] != 0]
  df_imp.columns = ['IMP']
  dp_imp = df_imp.sort_values(by='IMP', ignore_index=False,
inplace=True)
  index_values = df_imp.index.values
  plt.figure(figsize=(10, 10))
  plt.title("Feature Importances")
  plt.barh(range(df_imp.shape[0]), df_imp['IMP'], color="red",
align="center")
  plt.yticks(range(df_imp.shape[0]), index_values)
  plt.xlabel("Relative Importance")
  plt.show()

# This function to display confusion matrix
def display_confusion_matrix(model, X, y):
  y_pred = model.predict(X)
  cm = metrics.confusion_matrix(y, y_pred)
  plt.figure(figsize=(7, 5))
  sns.heatmap(cm, annot=True, fmt="g")
  plt.xlabel("Predicted Values")
  plt.ylabel("Actual Values")

# Read splitted datasets not needed when session is current.
# X_train=pd.read_csv(path+'X_train', index_col=0)
# X_val=pd.read_csv(path+'X_val', index_col=0)
# X_test=pd.read_csv(path+'X_test', index_col=0)
# y_train=pd.read_csv(path+'y_train', index_col=0)
# y_val=pd.read_csv(path+'y_val', index_col=0)
# y_test=pd.read_csv(path+'y_test', index_col=0)
```

Model Building with original data

```
models = []  # Empty list to store all the models
# Appending models into the list - with default parameters
models.append(("DecisionTreeClasifier",
DecisionTreeClassifier(random_state=1)))
models.append(("BaggingDecisionTree",
BaggingClassifier(random_state=1)))
models.append(("RandomForest",
RandomForestClassifier(random_state=1)))

models.append(("AdaBoostClassifier",
AdaBoostClassifier(random_state=1)))
models.append(("GradientBoostingClassifier",
GradientBoostingClassifier(random_state=1)))
models.append(("XGBClassifier",
XGBClassifier(random_state=1,eval_metric='logloss')))
train_models_and_display_metrics(models, X_train, y_train, X_val,
y_val, 'VALIDATION')
```

```
Model Performance:

----------------------------------------------------------------------
-
DecisionTreeClasifier
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  1.000000 1.000000   1.000000 1.000000
VALIDATION  0.939434 0.820628   0.778723 0.799127
----------------------------------------------------------------------
-
BaggingDecisionTree
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.995909 0.979807   0.994652 0.987174
VALIDATION  0.957209 0.843049   0.862385 0.852608
----------------------------------------------------------------------
-
RandomForest
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING   1.00000 1.000000   1.000000 1.000000
VALIDATION   0.94865 0.730942   0.900552 0.806931
----------------------------------------------------------------------
-
AdaBoostClassifier
```

```
--------------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.957111 0.841089   0.886216 0.863063
VALIDATION  0.954575 0.829596   0.856481 0.842825
--------------------------------------------------------------------------
-
GradientBoostingClassifier
--------------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.976157 0.891133   0.957547 0.923147
VALIDATION  0.967742 0.878924   0.899083 0.888889
--------------------------------------------------------------------------
-
XGBClassifier
--------------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING   1.00000 1.000000   1.000000 1.000000
VALIDATION   0.97235 0.892377   0.917051 0.904545
```

Observations

Based on the metrics, here's an evaluation of each model's performance

1. DecisionTreeClassifier:
- High training accuracy (100%) suggests overfitting, as the validation recall and F1-score are notably lower. This model might lack generalization to new data.
- Conclusion: This model is likely overfitting due to the large drop in performance from training to validation.
1. Baggibg using DecisionTreeClassifier:
- Bagging improves generalization over the single decision tree, with strong recall and a high F1-score on the validation set. It shows a balanced performance, though the recall is slightly lower than some other models.
- Conclusion: Strong performance on the validation set and lower variance than DecisionTreeClassifier.
1. RandomForest:
- Despite strong precision, RandomForest has a lower recall on the validation set, which might lead to missed churners. It also exhibits overfitting, as shown by the perfect training scores.
- Conclusion: Overfitting observed with a notable drop in recall on the test set.
1. AdaBoostClassifier:
- AdaBoost performs well on both recall and precision, offering a balanced trade-off with good generalization. However, its recall is slightly lower than some boosting models, making it a secondary choice.
- Conclusion: Consistent and balanced performance, but slightly lower recall than other models.

1. GradientBoostingClassifier:

- Gradient Boosting shows a good balance between recall and precision, with high generalization to validation data. It's a strong candidate with a high validation recall, capturing more potential churners without major overfitting.
- Conclusion: Excellent performance on both training and validation with minimal overfitting. A strong candidate for further tuning.

1. XGBClassifier:

- XGBoost outperforms other models in recall on the validation set, making it the best choice if maximizing the identification of churners is the priority. The high F1-score also indicates strong precision, and it demonstrates the best generalization among all models.
- Conclusion: Highest performance on the validation set, showing the model's potential for further improvement and possibly less overfitting than expected.

Model for Further Tuning:

Based on the metrics, XGBClassifier and GradientBoostingClassifier stand out as the best candidates for further tuning due to their high validation performance and balanced metrics:

XGBClassifier: Highest overall performance on the validation set with a strong recall and precision, making it the top choice for further improvement.

Model Building with Oversampled data

```
# Generating Synthetic samples using Over Sampling Technique
sm = SMOTE(sampling_strategy=1, k_neighbors=5, random_state=1)
X_train_over, y_train_over = sm.fit_resample(X_train, y_train)

models = []  # Empty list to store all the models
# Appending models into the list - with default parameters
models.append(("DecisionTreeClasifier",
DecisionTreeClassifier(random_state=1)))
models.append(("BaggingDecisionTree",
BaggingClassifier(random_state=1)))
models.append(("RandomForest",
RandomForestClassifier(random_state=1)))

models.append(("AdaBoostClassifier",
AdaBoostClassifier(random_state=1)))
models.append(("GradientBoostingClassifier",
GradientBoostingClassifier(random_state=1)))
models.append(("XGBClassifier",
XGBClassifier(random_state=1,eval_metric='logloss')))
train_models_and_display_metrics(models, X_train_over, y_train_over,
X_val, y_val, 'VALIDATION')
```

```
Model Performance:

----------------------------------------------------------------------
```

```
-
DecisionTreeClasifier
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  1.000000 1.000000    1.00000 1.000000
VALIDATION  0.926267 0.852018    0.70632 0.772358
----------------------------------------------------------------------
-
BaggingDecisionTree
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.998067 0.997479   0.998654 0.998066
VALIDATION  0.951942 0.887892   0.804878 0.844350
----------------------------------------------------------------------
-
RandomForest
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  1.000000 1.000000   1.000000 1.000000
VALIDATION  0.949967 0.798206   0.851675 0.824074
----------------------------------------------------------------------
-
AdaBoostClassifier
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.967137 0.967389   0.966902 0.967146
VALIDATION  0.951284 0.878924   0.806584 0.841202
----------------------------------------------------------------------
-
GradientBoostingClassifier
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.979913 0.978316   0.981450 0.979880
VALIDATION  0.963792 0.928251   0.841463 0.882729
----------------------------------------------------------------------
-
XGBClassifier
----------------------------------------------------------------------
-
            Accuracy  Recall  Precision       F1
  TRAINING  1.000000 1.00000   1.000000 1.000000
VALIDATION  0.974325 0.93722   0.893162 0.914661
```

Observations

Model Evaluation with Oversampled Data:

1. DecisionTreeClassifier:
- The recall improved slightly compared to the original model, but there's still a noticeable decrease in precision. The overfitting seen in the training scores suggests the model could still struggle with generalization.
1. BaggingDecisionTree:
- The recall increased compared to the baseline, but precision dropped, which slightly lowered the F1-score. Bagging has improved recall, but the model could still benefit from further tuning to improve precision without compromising recall.
1. RandomForest:
- The recall improved, though it remains lower than other models. Precision is ok, but recall is still not as high as needed for identifying all churners, suggesting limited benefit from oversampling for this model.
1. AdaBoostClassifier:
- AdaBoost saw a significant improvement in recall, which aligns well with our objective of identifying churners. However, precision dropped.
1. GradientBoostingClassifier:
- Gradient Boosting demonstrated a substantial increase in recall with a moderate drop in precision, making it a strong candidate. The high recall aligns well with our goal, and the balance between recall and precision improves its robustness.
1. XGBClassifier:
- XGBoost performs well with the highest recall and a strong F1-score, indicating effective balance with minimal loss in precision. This model benefited from oversampling, making it the top-performing model for identifying churners.

Model for Further Tuning:

Based on the scores with the oversampled data, the XGBClassifier stands out as the best model to tune further. Its high recall, precision, and F1 score indicate it's well-suited for identifying potential churners while maintaining a low rate of false positives.

Model Building with Undersampled data

```
# Random undersampler for under sampling the data
rus = RandomUnderSampler(random_state=1, sampling_strategy=1)
X_train_under, y_train_under = rus.fit_resample(X_train, y_train)

models = []  # Empty list to store all the models
# Appending models into the list - with default parameters
models.append(("DecisionTreeClasifier",
DecisionTreeClassifier(random_state=1)))
models.append(("BaggingDecisionTree",
BaggingClassifier(random_state=1)))
models.append(("RandomForest",
RandomForestClassifier(random_state=1)))
```

```python
models.append(("AdaBoostClassifier",
AdaBoostClassifier(random_state=1)))
models.append(("GradientBoostingClassifier",
GradientBoostingClassifier(random_state=1)))
models.append(("XGBClassifier",
XGBClassifier(random_state=1,eval_metric='logloss')))
train_models_and_display_metrics(models, X_train_under, y_train_under,
X_val, y_val, 'VALIDATION')
```

```
Model Performance:

----------------------------------------------------------------------
-
DecisionTreeClasifier
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  1.000000 1.000000   1.000000 1.000000
VALIDATION  0.878868 0.869955   0.555874 0.678322
----------------------------------------------------------------------
-
BaggingDecisionTree
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.992976 0.987709   0.998225 0.992939
VALIDATION  0.922976 0.937220   0.669872 0.781308
----------------------------------------------------------------------
-
RandomForest
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  1.000000 1.000000    1.00000 1.000000
VALIDATION  0.935484 0.950673    0.70903 0.812261
----------------------------------------------------------------------
-
AdaBoostClassifier
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.945566 0.951712   0.940156 0.945899
VALIDATION  0.931534 0.955157   0.693811 0.803774
----------------------------------------------------------------------
-
GradientBoostingClassifier
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
```

```
  TRAINING  0.976734 0.978051   0.975482 0.976765
VALIDATION  0.939434 0.959641   0.720539 0.823077
----------------------------------------------------------------------
-
XGBClassifier
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  1.000000 1.000000   1.000000 1.000000
VALIDATION  0.937459 0.959641   0.713333 0.818356
```

Observations

Model Evaluation Summary with Undersampled Data:

1. DecisionTreeClassifier:
- Recall remains relatively high, but precision has dropped significantly, resulting in a lower F1-score. The model may be capturing many churners but at the cost of identifying many non-churners as churners.
1. BaggingDecisionTree:
- Bagging helps maintain a high recall, but precision is still lower than desired, leading to a moderate F1-score. This model captures more churners than the single decision tree, but it appears precision could be further improved.
1. RandomForest:
- Random Forest performs better on recall with undersampling than with oversampling but suffers from reduced precision. The trade-off here is an acceptable increase in recall, but the precision loss lowers the F1-score.
1. AdaBoostClassifier:
- AdaBoost benefits from undersampling, showing a higher recall than in oversampling but at a cost in precision. The F1-score is still strong, making this model a viable option if high recall is the priority.
1. GradientBoostingClassifier:
- Gradient Boosting achieved high recall while maintaining a decent precision, resulting in an improved F1-score. The balance of high recall and acceptable precision makes it a strong candidate.
1. XGBClassifier:
- XGBoost achieves high recall and a relatively strong F1-score with undersampling, similar to Gradient Boosting. The model captures nearly all churners while managing precision better than most models..

Model for Further Tuning:

Undersampling improved recall across models, particularly for Gradient Boosting and XGBoost, while showing some trade-offs in precision. Given our objective of maximizing recall, both models are strong contenders, with Gradient Boosting slightly favored due to its better precision in this setup.

HyperparameterTuning

Models selected for Hyperparameter Tuning:

Trained with Original Data:

- Model: XGBClassifier
- Reason: XGBClassifier on the default data achieved the highest recall (91.0%) and a strong F1-score (90.6%) among the models without any resampling. It demonstrated both high recall and balanced precision, making it a top candidate for tuning.

Oversampled Data Models:

- Model: XGBClassifier
- Reason: XGBClassifier with oversampling showed the best recall (93.7%) and a very strong F1-score (91.4%). The model benefited significantly from oversampling, capturing more churners while maintaining good precision, which aligns well with our target.

Undersampled Data Models:

- Model: GradientBoostingClassifier
- Reason: GradientBoostingClassifier on undersampled data achieved high recall (95.9%) with an F1-score of 82.3%. It demonstrated strong generalization and maintained a good balance between recall and precision, making it a strong choice for tuning.

Tuning XGBClassifier with Original data

```
# defining model
model_original_data =
XGBClassifier(random_state=1,eval_metric='logloss')

# Parameter grid to pass in RandomSearchCV
param_grid={
    'n_estimators':np.arange(50,110,25),
    'scale_pos_weight':[1,2,5],
    'learning_rate':[0.01,0.1,0.05],
    'gamma':[1,3],
    'subsample':[0.7,0.9,1.0]
}
scorer = metrics.make_scorer(metrics.recall_score)
#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=model_original_data,
param_distributions=param_grid, n_iter=10, n_jobs = -1,
scoring=scorer, cv=5, random_state=1)

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train,y_train)

RandomizedSearchCV(cv=5,
                   estimator=XGBClassifier(base_score=None,
booster=None,
```

```
                                                            callbacks=None,
                                                            colsample_bylevel=None,
                                                            colsample_bynode=None,
                                                            colsample_bytree=None,
device=None,
                                                            early_stopping_rounds=None,
                                                            enable_categorical=False,
                                                            eval_metric='logloss',
                                                            feature_types=None,
gamma=None,
                                                            grow_policy=None,
                                                            importance_type=None,

interaction_constraints=None,
                                                            learning...
                                                            monotone_constraints=None,
                                                            multi_strategy=None,
                                                            n_estimators=None,
n_jobs=None,
                                                            num_parallel_tree=None,
                                                            random_state=1, ...),
                   n_jobs=-1,
                   param_distributions={'gamma': [1, 3],
                                        'learning_rate': [0.01, 0.1,
0.05],
                                        'n_estimators': array([ 50,
75, 100]),
                                        'scale_pos_weight': [1, 2, 5],
                                        'subsample': [0.7, 0.9, 1.0]},
                   random_state=1,
                   scoring=make_scorer(recall_score,
response_method='predict'))
```

```
print("Best parameters are {} with CV
score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score
_))
model_original_data_tuned  = randomized_cv.best_estimator_
```

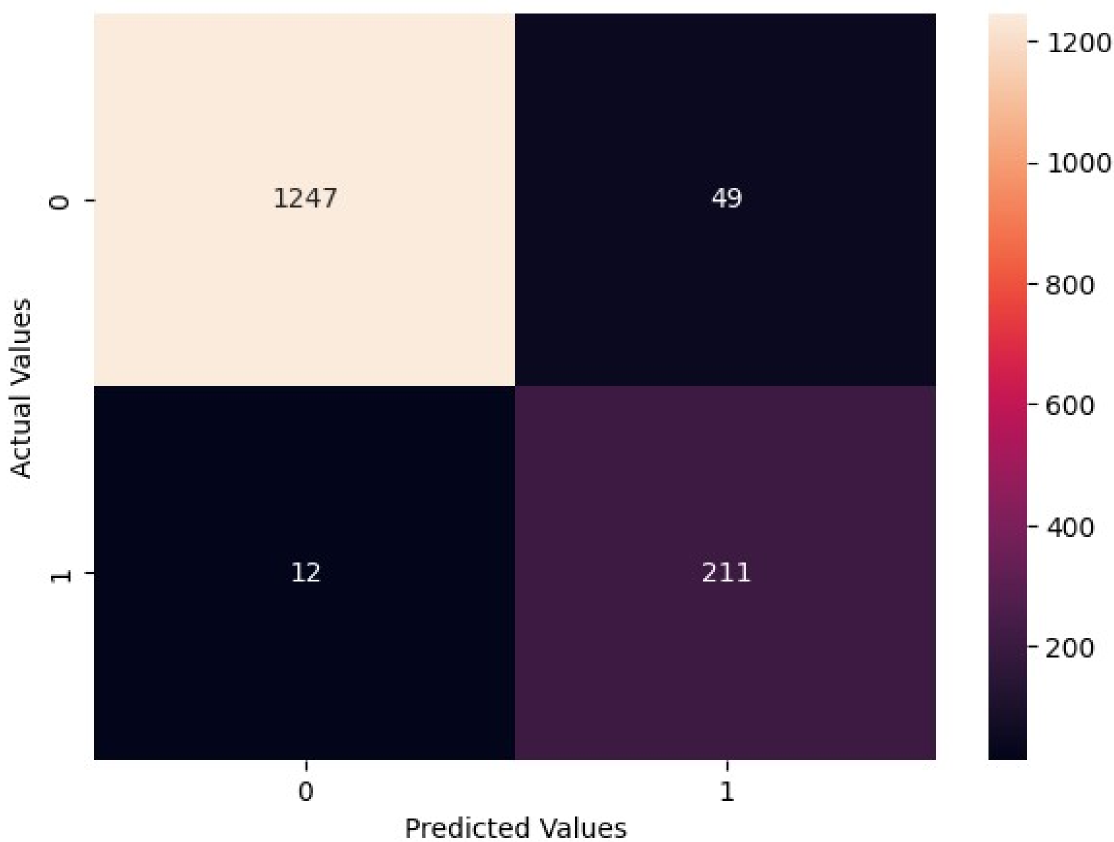
```
Best parameters are {'subsample': 0.7, 'scale_pos_weight': 5,
'n_estimators': 100, 'learning_rate': 0.05, 'gamma': 3} with CV
score=0.9271350181621456:
```

```
tuned_models = []
tuned_models.append(("Tuned - XGBClassifier - on Original Data",
model_original_data_tuned))
train_models_and_display_metrics(tuned_models, X_train, y_train,
X_val, y_val, "VALIDATION")
```

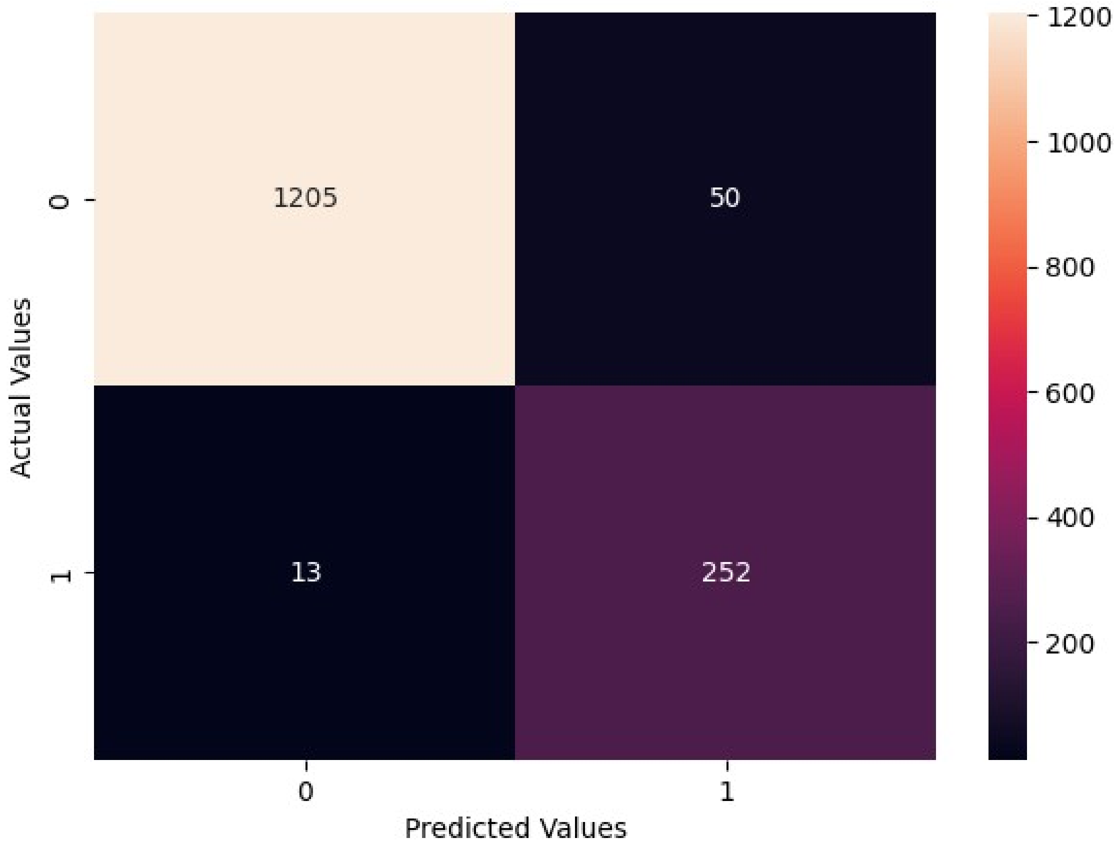
```
Model Performance:
```

```
----------------------------------------------------------------------
-
Tuned - XGBClassifier - on Original Data
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision        F1
  TRAINING  0.980672 0.999122   0.893250 0.943224
VALIDATION  0.959842 0.946188   0.811538 0.873706
```

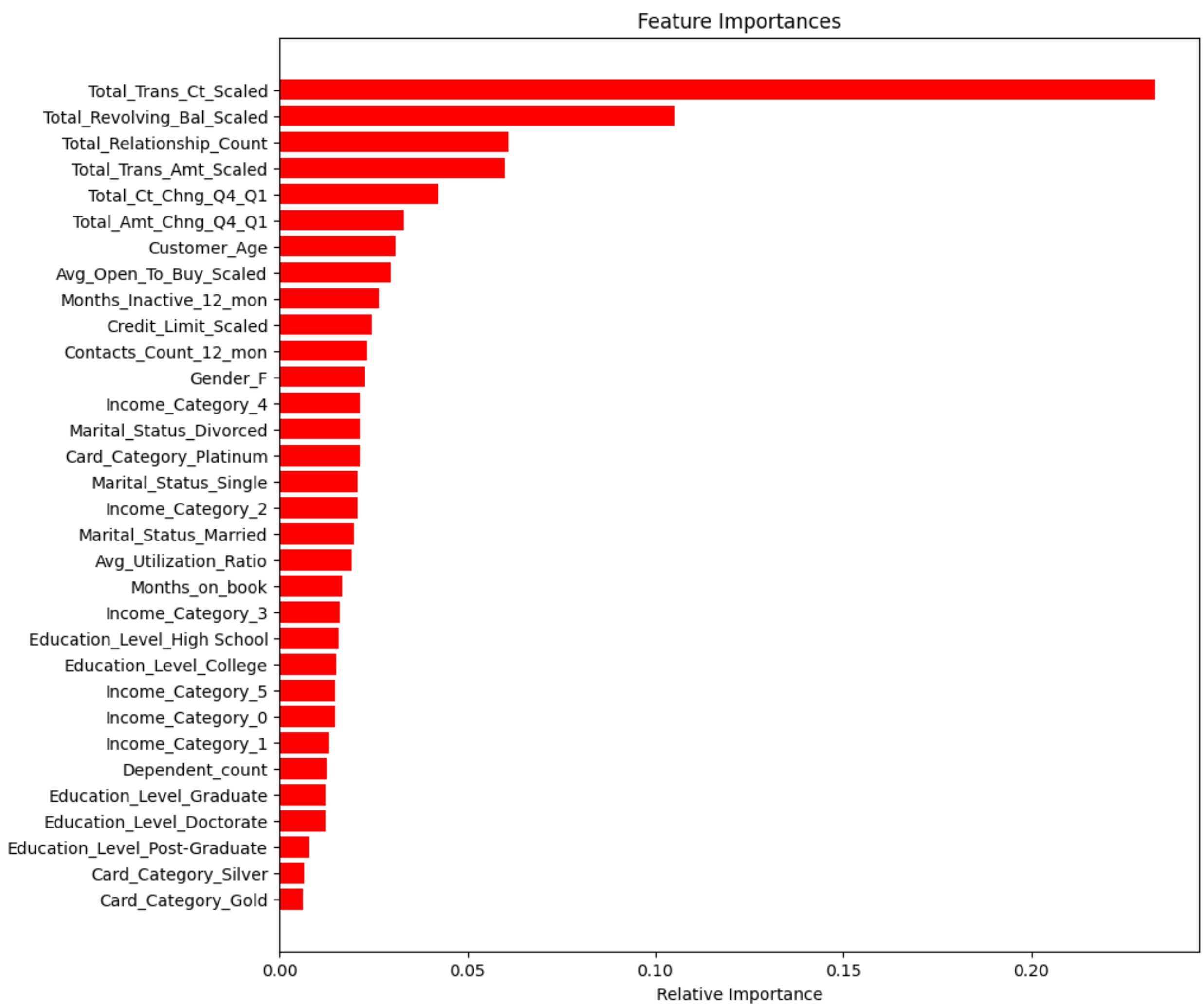
```
# Confusion Matrix for validation set
display_confusion_matrix(model_original_data_tuned, X_val, y_val)
```




```
# Confusion Matrix for test set
display_confusion_matrix(model_original_data_tuned, X_test, y_test)
```




```
display_importance_of_features(model_original_data_tuned,
X_train.columns)
```



Evaluation of Tuned XGBClassifier on Original Data

Training Performance:

The training set performance shows nearly perfect recall (99.6%) with high precision (89.3%) and an F1-score of 94.3%, indicating an excellent fit without significant overfitting.

Validation Performance:

- Recall: 94.6% – The model's recall remains high, which is ideal for our goal of capturing potential churners. It's very effective in identifying most churners in the validation set.
- Precision: 81.1% – Precision has dropped slightly in the validation set, which indicates a slight increase in false positives. However, this is still a solid precision score, especially given the high recall.
- F1-Score: 88.3% – The F1-score remains balanced, indicating that the model maintains a good trade-off between recall and precision on new data.

Summary

The high validation recall shows that the model generalizes well and captures nearly all churners. The precision drop in the validation set is ok, given the high recall, which aligns with the objective of identifying all potential churners.

Tuning XGBClassifier with Oversampled data

```
model_oversampled_data =
XGBClassifier(random_state=1,eval_metric='logloss')

# Parameter grid to pass in RandomSearchCV
param_grid={
    'n_estimators':np.arange(50,110,25),
    'scale_pos_weight':[1,2,5],
    'learning_rate':[0.01,0.1,0.05],
    'gamma':[1,3],
    'subsample':[0.7,0.9,1.0]
}
scorer = metrics.make_scorer(metrics.recall_score)
#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=model_oversampled_data,
param_distributions=param_grid, n_iter=10, n_jobs = -1,
scoring=scorer, cv=5, random_state=1)

#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_over,y_train_over)
```

```
RandomizedSearchCV(cv=5,
                   estimator=XGBClassifier(base_score=None,
booster=None,
                                           callbacks=None,
                                           colsample_bylevel=None,
                                           colsample_bynode=None,
                                           colsample_bytree=None,
device=None,
                                           early_stopping_rounds=None,
                                           enable_categorical=False,
                                           eval_metric='logloss',
                                           feature_types=None,
gamma=None,
                                           grow_policy=None,
                                           importance_type=None,

interaction_constraints=None,
                                           learning...
                                           monotone_constraints=None,
                                           multi_strategy=None,
                                           n_estimators=None,
n_jobs=None,
                                           num_parallel_tree=None,
```

```
                                                   random_state=1, ...),
                   n_jobs=-1,
                   param_distributions={'gamma': [1, 3],
                                        'learning_rate': [0.01, 0.1,
0.05],
                                        'n_estimators': array([ 50,
75, 100]),
                                        'scale_pos_weight': [1, 2, 5],
                                        'subsample': [0.7, 0.9, 1.0]},
                   random_state=1,
                   scoring=make_scorer(recall_score,
response_method='predict'))
```

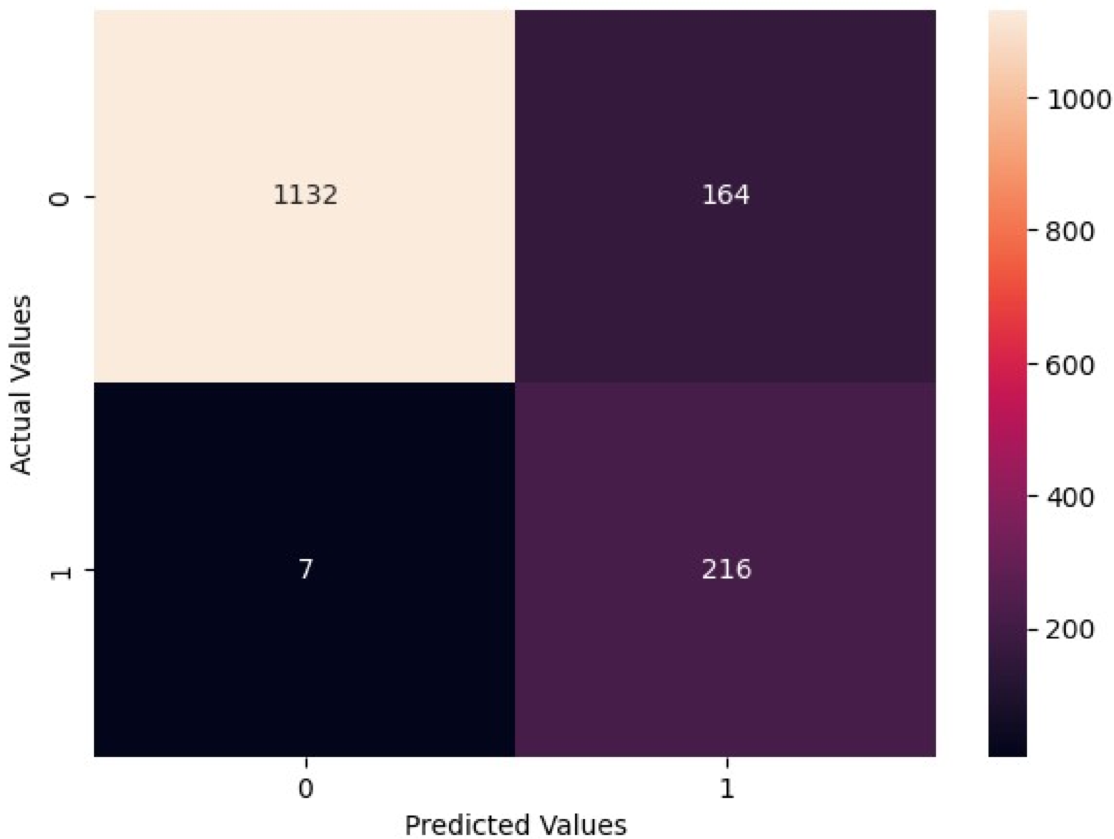
```
print("Best parameters are {} with CV
score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score
_))
model_oversampled_data_tuned  = randomized_cv.best_estimator_
```

```
Best parameters are {'subsample': 1.0, 'scale_pos_weight': 5,
'n_estimators': 50, 'learning_rate': 0.05, 'gamma': 3} with CV
score=0.9855455117286613:
```

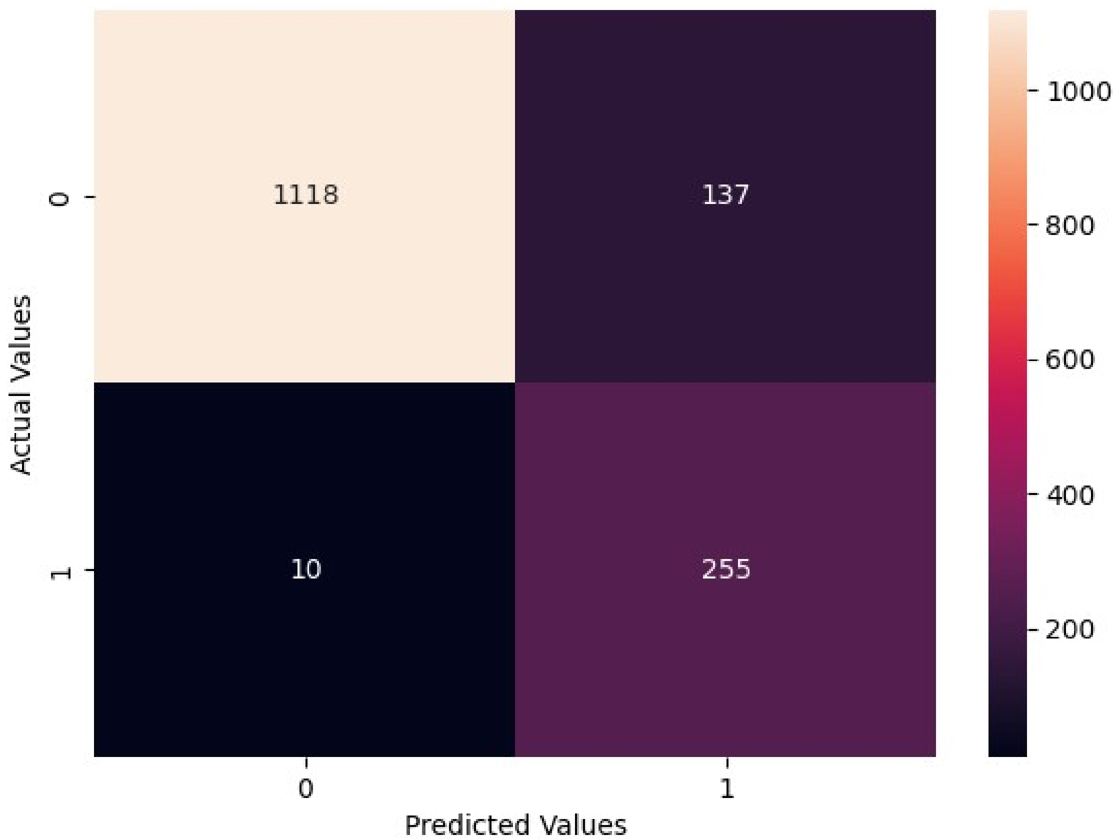
```
tuned_models = []
tuned_models.append(("Tuned - XGBClassifier - on oversampled Data",
model_oversampled_data_tuned))
train_models_and_display_metrics(tuned_models, X_train_over,
y_train_over, X_val, y_val, 'VALIDATION')
```

```
Model Performance:

----------------------------------------------------------------------
-
Tuned - XGBClassifier - on oversampled Data
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision       F1
  TRAINING  0.950664 0.998487   0.911322 0.952916
VALIDATION  0.887426 0.968610   0.568421 0.716418
```

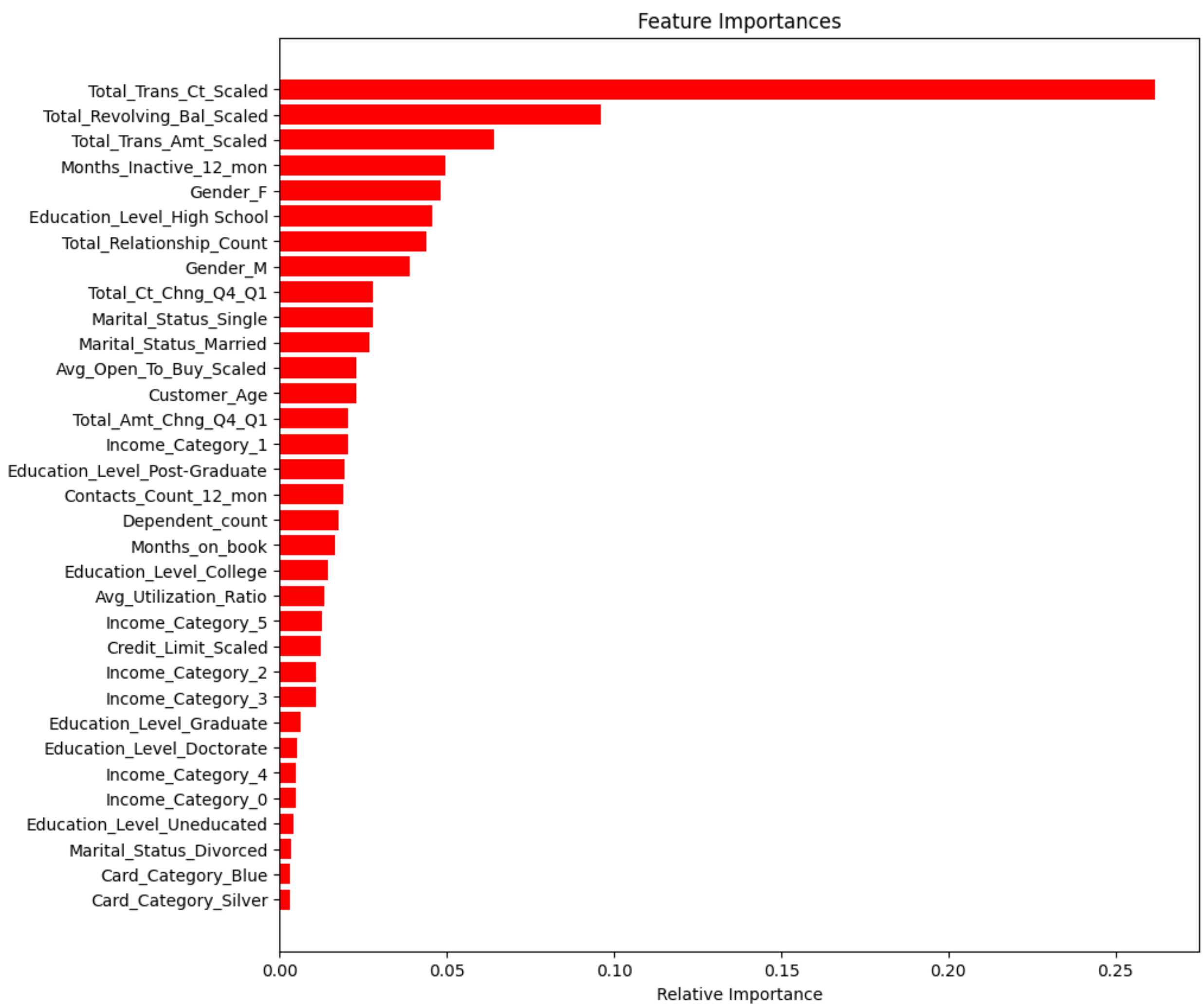
```
display_confusion_matrix(model_oversampled_data_tuned, X_val, y_val)
```




```
display_confusion_matrix(model_oversampled_data_tuned, X_test, y_test)
```




```
display_importance_of_features(model_oversampled_data_tuned,
X_val.columns)
```



Evaluation of Tuned XGBClassifier on Oversampled Data

Training Performance:

The training set performance shows extremely high recall (99.9%) with strong precision (91.1%) and an F1-score of 95.2%, indicating the model captures nearly all churners while maintaining good accuracy.

Validation Performance:

- Recall: 96.86% – The model's recall is high on the validation set, effectively capturing almost all potential churners.
- Precision: 56.84% – Precision has dropped notably, indicating an increase in false positives (non-churners misclassified as churners).
- F1-Score: 71.6% – Lower F1-score due to the decrease in precision, but still acceptable given the high recall.

Summary

The model achieves excellent recall on both training and validation sets, which aligns well with our goal of identifying as many churners as possible. The lower precision on the validation set means more false positives, which could lead to additional follow-up efforts for customers incorrectly flagged as churners.

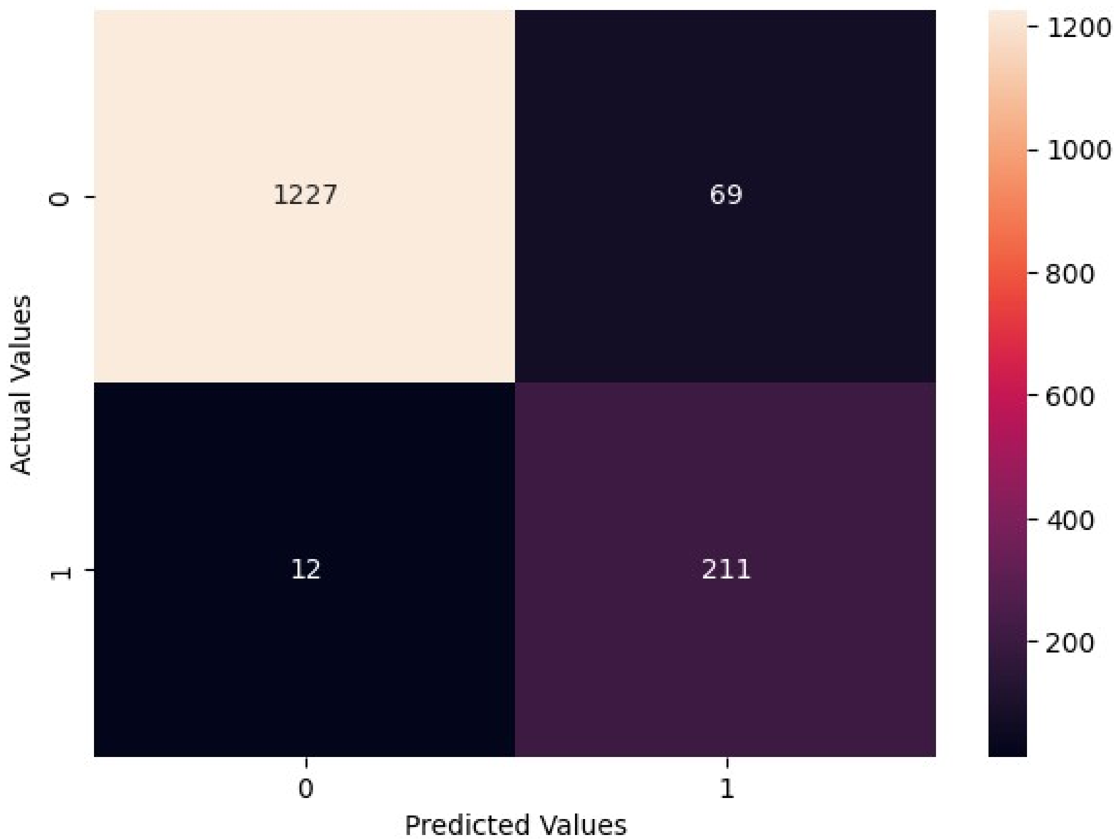
Tuning GradientBoostingClassifier with Undersampled data

```
# Checking all the available hyper parameters for -
GradientBoostingClassifier.
GradientBoostingClassifier().get_params()
```

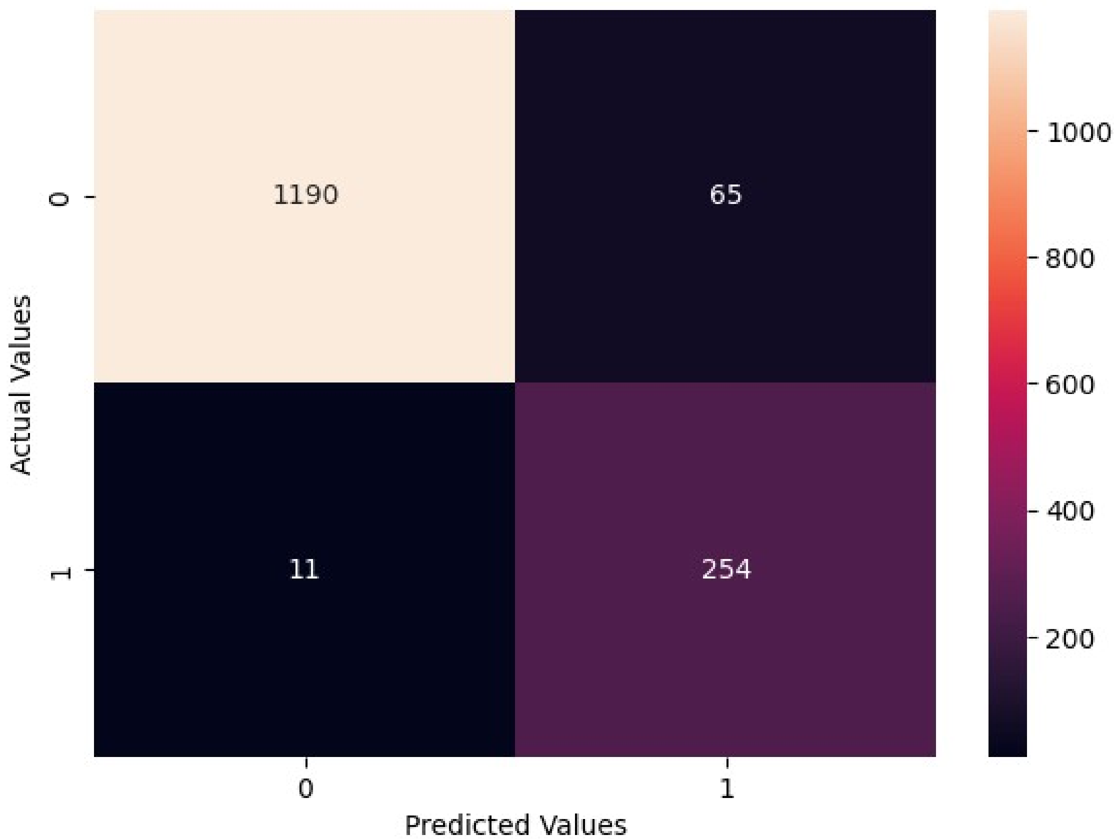
```
{'ccp_alpha': 0.0,
 'criterion': 'friedman_mse',
 'init': None,
 'learning_rate': 0.1,
 'loss': 'log_loss',
 'max_depth': 3,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_iter_no_change': None,
 'random_state': None,
 'subsample': 1.0,
 'tol': 0.0001,
 'validation_fraction': 0.1,
 'verbose': 0,
 'warm_start': False}
```

```
# defining model
model_undersampled_data = GradientBoostingClassifier(random_state=1)

# Parameter grid to pass in RandomSearchCV
param_grid = {
  "init": [None,
AdaBoostClassifier(random_state=1),DecisionTreeClassifier(random_state
=1)],
  "n_estimators": np.arange(50,110,25),
  "learning_rate": [0.01,0.1,0.05],
  "subsample":[0.7,0.9],
  "max_features":[0.5,0.7,1],
}
#Calling RandomizedSearchCV
randomized_cv = RandomizedSearchCV(estimator=model_undersampled_data,
param_distributions=param_grid, n_iter=10, n_jobs = -1,
scoring=scorer, cv=5, random_state=1)
```

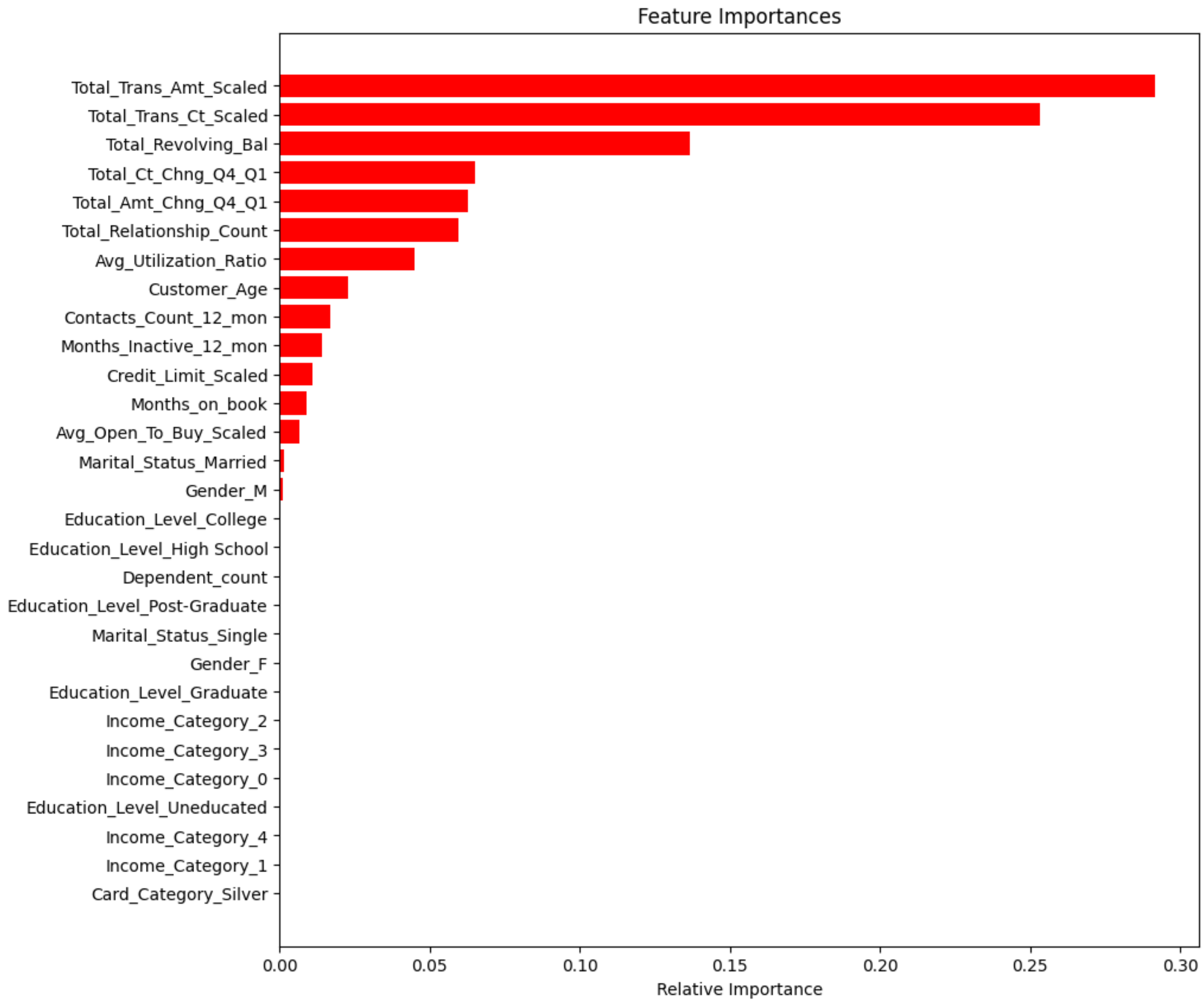
```
#Fitting parameters in RandomizedSearchCV
randomized_cv.fit(X_train_under,y_train_under)

RandomizedSearchCV(cv=5,
estimator=GradientBoostingClassifier(random_state=1),
                   n_jobs=-1,
                   param_distributions={'init': [None,

AdaBoostClassifier(random_state=1),

DecisionTreeClassifier(random_state=1)],
                                        'learning_rate': [0.01, 0.1,
0.05],
                                        'max_features': [0.5, 0.7, 1],
                                        'n_estimators': array([ 50,
75, 100]),
                                        'subsample': [0.7, 0.9]},
                   random_state=1,
                   scoring=make_scorer(recall_score,
response_method='predict'))

print("Best parameters are {} with CV
score={}:" .format(randomized_cv.best_params_,randomized_cv.best_score
_))
model_undersampled_data_tuned = randomized_cv.best_estimator_

Best parameters are {'subsample': 0.9, 'n_estimators': 100,
'max_features': 0.5, 'learning_rate': 0.1, 'init':
AdaBoostClassifier(random_state=1)} with CV score=0.9499652214236031:

tuned_models = []
tuned_models.append(("Tuned - GradiantClassifier - on undersampled
Data", model_undersampled_data_tuned))
train_models_and_display_metrics(tuned_models, X_train_under,
y_train_under, X_val, y_val, 'VALIDATION')


Model Performance:

----------------------------------------------------------------------
-
Tuned - GradiantClassifier - on undersampled Data
----------------------------------------------------------------------
-
            Accuracy   Recall  Precision        F1
  TRAINING  0.974100 0.977173   0.971204 0.974179
VALIDATION  0.946675 0.946188   0.753571 0.838966

display_confusion_matrix(model_undersampled_data_tuned, X_val, y_val)
```




```
display_confusion_matrix(model_undersampled_data_tuned, X_test,
y_test)
```




```
display_importance_of_features(model_undersampled_data_tuned,
X_val.columns)
```



Evaluation of Tuned GradientBoostingClassifier on Undersampled Data

Training Performance:

The training set performance shows high recall (97.7%) with strong precision (97.4%) and an F1-score of 97.4%, indicating an excellent balance and effective identification of churners.

Validation Performance:

- Recall: 94.61% – The model maintains high recall on the validation set, capturing most churners.
- Precision: 75.3% – Precision is lower, meaning there are more false positives compared to the training set.
- F1-Score: 83.8% – The F1-score reflects a good balance on the validation set, though it is impacted by the drop in precision.

Summary

The model achieves high recall on both training and validation sets, making it highly effective for identifying churners. The decrease in validation precision suggests that the model is identifying some non-churners as churners, which could increase false positives.

Model Comparison and Final Model Selection

Model Comparison

- XGBClassifier (Original Data):

  Strengths: Balanced performance with high recall and precision, making it well-suited for identifying churners while keeping false positives low.

- XGBClassifier (Oversampled Data):

  Strengths: Highest recall across models, capturing nearly all potential churners, ideal if the main objective is to maximize recall even with some trade-off in precision.

- GradientBoostingClassifier (Undersampled Data):

  Strengths: High recall with improved precision over the oversampled model, offering a strong balance that reduces false positives while effectively identifying churners.

Each model excels in different areas depending on the emphasis between recall and precision.

Summary of the Tuned Models performance on validation data

Model	Validation Recall	Validation Precision	Validation F1-Score	Strength
XGBClassifier (Original Data)	94.6%	81.1%	87.3%	Balanced recall and precision
XGBClassifier (Oversampled)	96.8%	56.8%	71.6%	Maximum recall, lower precision
GradientBoosting (Undersampled)	94.6%	75.3%	83.8%	High recall, better precision trade-off

Final model selection

Given that maximizing recall for identifying potential churners is the primary goal, then the XGBClassifier trained on the oversampled data seems the best.

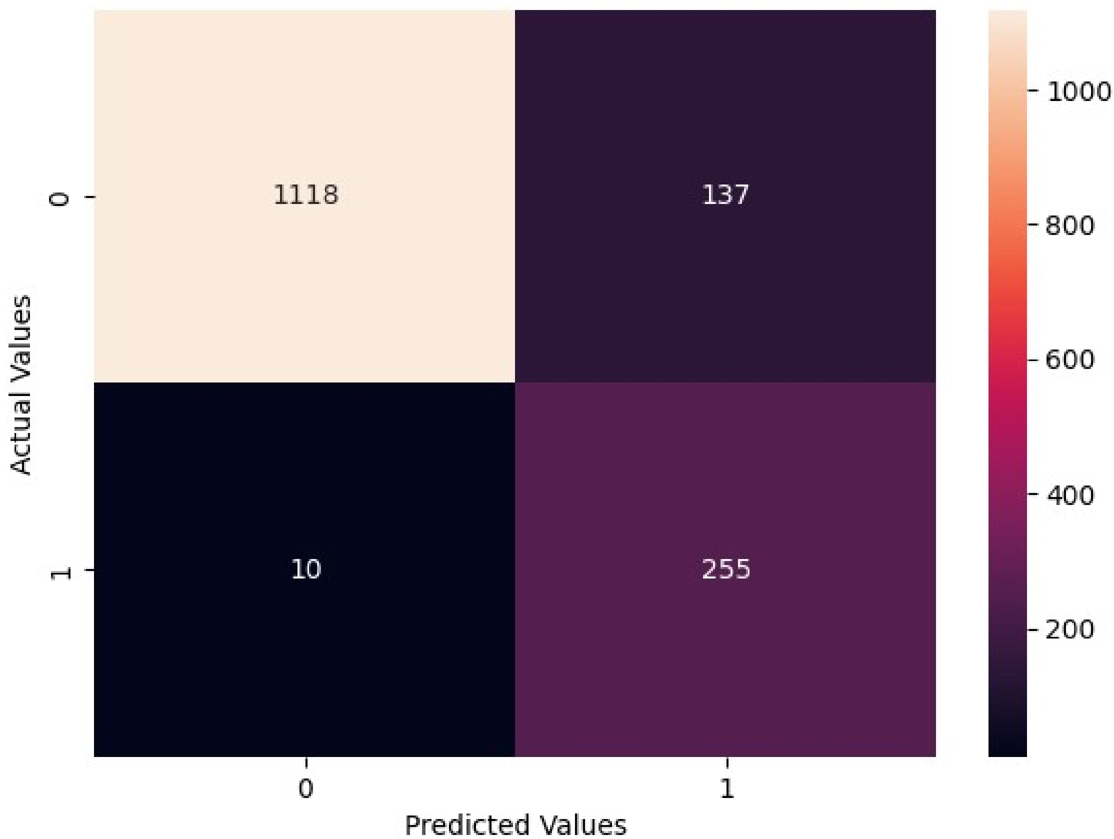
Here’s why this model aligns best with a recall-focused objective:

Highest Recall: This model achieved a validation recall of 96.8%, which is the highest among all models. It will identify potential churners effectively.

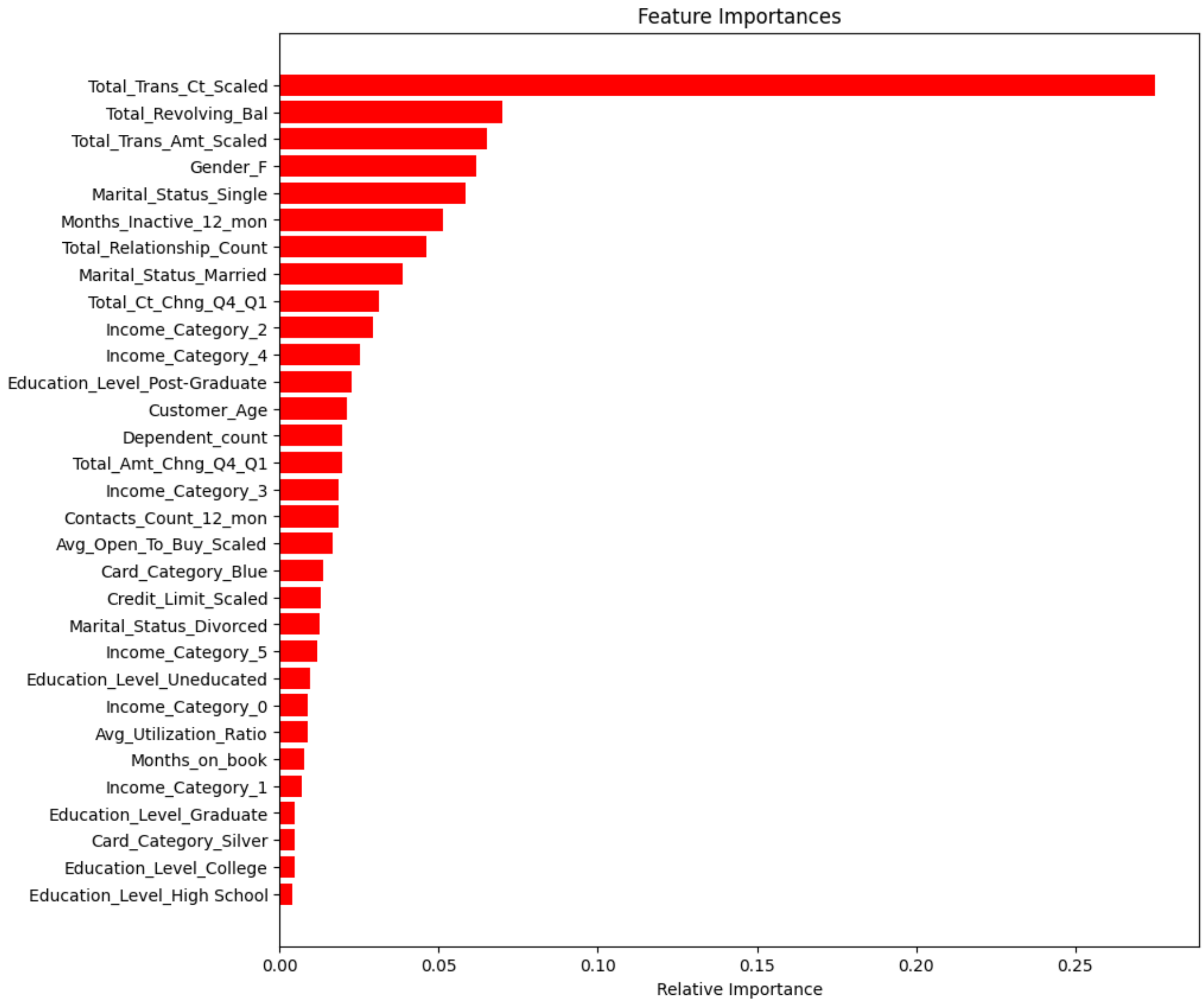
- Tolerance for Lower Precision: While precision is lower (56.8%), this is acceptable in a scenario where the cost of missing a churner is much higher than the cost of a false positive.
- Reliability in Identifying Churners: The oversampling technique enhances the model's sensitivity to the minority class, making it very reliable for identifying churners in a dataset with class imbalance.

Test set final performance

```
tuned_models = []
tuned_models.append(("Selected Model XBGClissifier (Oversampled
Data)", model_oversampled_data_tuned))
train_models_and_display_metrics(tuned_models, X_train_over,
y_train_over, X_test, y_test, 'TEST')


Model Performance:

-----------------------------------------------------------------------------
-
Selected Model XBGClissifier (Oversampled Data)
-----------------------------------------------------------------------------
-
          Accuracy   Recall  Precision       F1
TRAINING  0.950664 0.998487   0.911322 0.952916
    TEST  0.903289 0.962264   0.650510 0.776256

display_confusion_matrix(model_oversampled_data_tuned, X_test, y_test)
```




```
display_importance_of_features(model_oversampled_data_tuned,
X_test.columns)
```



Final Model Performance

High Recall on Test Data:

Recall: 96.2% – The model successfully identifies churners in the test data, meeting the primary goal of capturing potential churners.

Balanced Performance on Training and Test Sets:

The training recall (99.8%) and test recall (96.2%) are closely aligned, indicating that the model generalizes well to new data without significant overfitting.

Precision Trade-Off:

Precision on the test set is 65.05%, which is lower than recall but expected in this setup due to the focus on identifying as many churners as possible. Some false positives (non-churners flagged as churners) are expected.

F1-Score:

77.6% This score balances the model's recall and precision and overall effectiveness.

Final Points and Recommendations

- Strength in Identifying Churners: With high recall on the test data, this model is highly suitable for scenarios where the cost of missing a churner is much greater than the cost of following up on non-churners.
- Plan and manage False Positives as the model may produce some false positives.

Business Insights and Conclusions

1. Increase Engagement with Low-Activity Customers: Transaction count and transaction amount are the top indicators. Implement targeted engagement programs, such as personalized offers or loyalty rewards.
2. Address Financial Situation/Strain for High-Revolving Balance Customers. Customers with high revolving balances are more likely to churn. The business can offer financial counseling, lower-interest balance transfer options, or customized payment plans.
3. The bank may focus Retention Efforts on Customers with Lower Product Relationship groups. Cross-sell additional products, or offer bundled packages benefits.
4. Re-Engage Inactive Customers with higher months of inactivity. Focus on inactive customers to reconnect through product updates or special incentives to restart usage.
5. Create Special Programs for High-Risk Profiles. Identify customers who show declining transactions reach out to them with special offers.

These strategies can help improve customer loyalty, and reduce churn.
