Clustering Task

April 8, 2024

1 Clustering Task

Name: ***R.M. Nipuna Upeksha***

IIT ID: ***20230106***

RGU ID: ***2322823***

1.1 Introduction

1. Clearly define the data mining problem and objectives.
2. Formulates the problem in a way suitable for data mining techniques.

For the clustering task, the following data set was chosen from Kaggle. https://www.kaggle.com/datasets/arjunbhasin2013/ccdata. The dataset is comprised of 9000 credit card holders and their credit card usage behaviors. Following are the columns in the data set.

- `CUST ID` → Credit Card holder identification.
- `BALANCE` → The credit card balance amount.
- `BALANCE_FREQUENCY` → The frequency of updating the balance (0 = Not frequently updated, 1 = Frequently updated).
- `PURCHASES` → Number of purchases made from the credit card account.
- `ONEOFF_PURCHASES` → Maximum purchase price done for a purchase.
- `INSTALLMENTS PURCHASES` → Purchase amount done in installments.
- `CASH ADVANCE` → Advance amount provided by the user.
- `PURCHASES FREQUENCY` → The frequency of the purchases(0 = Not frequently purchased, 1 = Frequently purchased).
- `ONEOFF_PURCHASES_FREQUENCY` → Frequency of purchases happening in one-go(0 = Not frequently purchased, 1 = Frequently purchased).
- `PURCHASES_INSTALLMENTS_FREQUENCY` → Frequency of purchases done in installments(0 = Not frequently purchased, 1 = Frequently purchased).
- `CASH_ADVANCE_FREQUENCY` → The frequency of using cash advance.
- `CASH_ADVANCE_TRX` → Number of cash advance transactions.
- `PURCHASES_TRX` → Number of purchase transactions.
- `CREDIT_LIMIT` → Credit card limit for the user.
- `PAYMENTS` → Payment amount.
- `MINIMUM_PAYMENTS` → Minimum payment amounts done by the user.
- `PRC_FULL_PAYMENT` → Full payment percentage paid by the user.

- `TENURE` → Tenure period of the user.

1.1.1 Data Mining Problem and Objectives

The data mining problem is to categorize credit card users into distinct groups, aiming to gain valuable insights into their credit card behaviors. Typically, the credit card holders fall into one of the following categories.

- `Transactors` → Customers who pay the minimum amount of interest charges and use the credit cards carefully without making large impacts.
- `Revolvers` → The customers who use their credit card as a loan, and pay around 20% of interest due to that.
- `New Customers` → New customers who get low-tenure credit cards only for specific purposes(e.g. traveling).
- `VIP/PRIME` → The customers who have high credit limits and credit percentages. By applying this conceptual framework, we intend to analyze customer data and cluster them into optimal segments. This segmentation will allow using to discern trends and gain valuable insights.

The objectives of the customer market segmentation are listed below. - Can get important insights on segmentation and maximize the marketing conversion rates. - Can find more spending customers and provide them more benefits to spend more. - Can identify potential increases and decreases in segments and take necessary steps to resolve them.

1.1.2 Problem Formulation with Data Mining Techniques

We can divide the problem formulation and the necessary steps that can be taken to solve it into the following steps.

Description

The goal of this data mining problem is to categorize credit card holders into different segments to gain insights into their spending habits and usage patterns. By leveraging the aforementioned dataset columns/features we intend to identify different customer groups and identify their behaviors.

Methodology

We need to follow the data mining techniques mentioned below to achieve our goal. 1. Data Preprocessing
(a) Handling NA/missing values if any.
(b) Handling outliers.
(c) Feature scaling for uniformity.
(d) Encoding categorical variables, if present. 3. Exploratory Data Analysis(EDA)
(a) Understanding the data and feature distribution. 4. Feature Selection
(a) Identifying relevant features for segmentation.
(b) Removing redundant or less impactful features. 6. Clustering Algorithm
(a) Utilizing clustering algorithms like K-Means clustering.
(b) Grouping the customers based on similar credit card usage patterns.
(c) Determining the optimal number of clusters using the Elbow Method or Silhouette Score.

1.2 Pre-process the Dataset

1. Handle missing values and outliers if any.
2. Produce Q-Q plots and histograms of the features and apply the transformations if required.
3. If it is required, apply suitable feature coding techniques.
4. Scale and/or standardize the features and produce relevant graphs to show the scaling/standardizing effect.
5. If required, apply suitable feature discretization techniques.

Before applying the algorithms to our dataset, we need to pre-process the dataset to remove NA/missing values, duplications, and outliers.

Let's import the necessary libraries and our dataset first.

The following code is used to ignore the warnings that may pop up in the libraries.

```
import warnings
warnings . filterwarnings ("ignore" , category=FutureWarning)
warnings.filterwarnings("ignore", category=UserWarning)
```

[1]:
```
# Import libraries
# Data Structures
import pandas as pd
import numpy as np

# Data processing
import scipy.stats as stats
from sklearn.preprocessing import FunctionTransformer, KBinsDiscretizer
from scipy.sparse import csr_matrix
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import StandardScaler, normalize
from sklearn.cluster import KMeans, MiniBatchKMeans, DBSCAN
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score, silhouette_samples,␣
 ↪davies_bouldin_score, calinski_harabasz_score
from timeit import timeit

# Visualization and analysis
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
from matplotlib.ticker import FixedLocator, FixedFormatter
from yellowbrick.cluster import KElbowVisualizer
from prettytable import PrettyTable

# Warnings
import warnings
```

```
warnings.filterwarnings("ignore", category=FutureWarning)
warnings.filterwarnings("ignore", category=UserWarning)
```

Next let's import the dataset as a `Pandas` data frame.

[2]:
```
# Constants
# The dataset
MARKETING_DATASET = './datasets/marketing_data.csv'
```

[3]:
```
# Read the dataset
df = pd.read_csv(MARKETING_DATASET)
```

Let's check the number of records and features in the dataset. Also, let's print the first 10 records to check the data in our dataset.

[4]:
```
# Count the number of records in the data frame
number_of_records = len(df.index)
print(f'Number of records in the dataset: {number_of_records}')

# Count the number of features in the data frame
number_of_features = len(df.columns)
print(f'Number of features in the dataset: {number_of_features}')

# Show first 10 records
df.head(10)
```

```
Number of records in the dataset: 8950
Number of features in the dataset: 18
```

[4]:

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	\
0	C10001	40.900749	0.818182	95.40	0.00	
1	C10002	3202.467416	0.909091	0.00	0.00	
2	C10003	2495.148862	1.000000	773.17	773.17	
3	C10004	1666.670542	0.636364	1499.00	1499.00	
4	C10005	817.714335	1.000000	16.00	16.00	
5	C10006	1809.828751	1.000000	1333.28	0.00	
6	C10007	627.260806	1.000000	7091.01	6402.63	
7	C10008	1823.652743	1.000000	436.20	0.00	
8	C10009	1014.926473	1.000000	861.49	661.49	
9	C10010	152.225975	0.545455	1281.60	1281.60	

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	\
0	95.40	0.000000	0.166667	
1	0.00	6442.945483	0.000000	
2	0.00	0.000000	1.000000	
3	0.00	205.788017	0.083333	
4	0.00	0.000000	0.083333	
5	1333.28	0.000000	0.666667	
6	688.38	0.000000	1.000000	

```
7                  436.20      0.000000             1.000000
8                  200.00      0.000000             0.333333
9                    0.00      0.000000             0.166667

   ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
0                    0.000000                          0.083333
1                    0.000000                          0.000000
2                    1.000000                          0.000000
3                    0.083333                          0.000000
4                    0.083333                          0.000000
5                    0.000000                          0.583333
6                    1.000000                          1.000000
7                    0.000000                          1.000000
8                    0.083333                          0.250000
9                    0.166667                          0.000000

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PURCHASES_TRX  CREDIT_LIMIT  \
0                0.000000                 0              2        1000.0
1                0.250000                 4              0        7000.0
2                0.000000                 0             12        7500.0
3                0.083333                 1              1        7500.0
4                0.000000                 0              1        1200.0
5                0.000000                 0              8        1800.0
6                0.000000                 0             64       13500.0
7                0.000000                 0             12        2300.0
8                0.000000                 0              5        7000.0
9                0.000000                 0              3       11000.0

      PAYMENTS  MINIMUM_PAYMENTS  PRC_FULL_PAYMENT  TENURE
0   201.802084        139.509787          0.000000      12
1  4103.032597       1072.340217          0.222222      12
2   622.066742        627.284787          0.000000      12
3     0.000000               NaN          0.000000      12
4   678.334763        244.791237          0.000000      12
5  1400.057770       2407.246035          0.000000      12
6  6354.314328        198.065894          1.000000      12
7   679.065082        532.033990          0.000000      12
8   688.278568        311.963409          0.000000      12
9  1164.770591        100.302262          0.000000      12
```

To get a general idea about the dataset, we can use the `df.describe()` function, which will provide the metrics like `mean`, `count`, and `standard deviation` of the dataset.

[5]:
```
# Get a general idea about the dataset
df.describe()
```

[5]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES
count	8950.000000	8950.000000	8950.000000	8950.000000
mean	1564.474828	0.877271	1003.204834	592.437371
std	2081.531879	0.236904	2136.634782	1659.887917
min	0.000000	0.000000	0.000000	0.000000
25%	128.281915	0.888889	39.635000	0.000000
50%	873.385231	1.000000	361.280000	38.000000
75%	2054.140036	1.000000	1110.130000	577.405000
max	19043.138560	1.000000	49039.570000	40761.250000

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY
count	8950.000000	8950.000000	8950.000000
mean	411.067645	978.871112	0.490351
std	904.338115	2097.163877	0.401371
min	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.083333
50%	89.000000	0.000000	0.500000
75%	468.637500	1113.821139	0.916667
max	22500.000000	47137.211760	1.000000

	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY
count	8950.000000	8950.000000
mean	0.202458	0.364437
std	0.298336	0.397448
min	0.000000	0.000000
25%	0.000000	0.000000
50%	0.083333	0.166667
75%	0.300000	0.750000
max	1.000000	1.000000

	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX	PURCHASES_TRX	CREDIT_LIMIT
count	8950.000000	8950.000000	8950.000000	8949.000000
mean	0.135144	3.248827	14.709832	4494.449450
std	0.200121	6.824647	24.857649	3638.815725
min	0.000000	0.000000	0.000000	50.000000
25%	0.000000	0.000000	1.000000	1600.000000
50%	0.000000	0.000000	7.000000	3000.000000
75%	0.222222	4.000000	17.000000	6500.000000
max	1.500000	123.000000	358.000000	30000.000000

	PAYMENTS	MINIMUM_PAYMENTS	PRC_FULL_PAYMENT	TENURE
count	8950.000000	8637.000000	8950.000000	8950.000000
mean	1733.143852	864.206542	0.153715	11.517318
std	2895.063757	2372.446607	0.292499	1.338331
min	0.000000	0.019163	0.000000	6.000000
25%	383.276166	169.123707	0.000000	12.000000
50%	856.901546	312.343947	0.000000	12.000000

```
75%      1901.134317        825.485459          0.142857     12.000000
max     50721.483360      76406.207520          1.000000     12.000000
```

The following code also provides a general idea about the dataset. It provides information on the data types in the dataset and the non-null data count.

[6]:
```
# Get a general idea about the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8950 entries, 0 to 8949
Data columns (total 18 columns):
 #   Column                            Non-Null Count  Dtype
---  ------                            --------------  -----
 0   CUST_ID                           8950 non-null   object
 1   BALANCE                           8950 non-null   float64
 2   BALANCE_FREQUENCY                 8950 non-null   float64
 3   PURCHASES                         8950 non-null   float64
 4   ONEOFF_PURCHASES                  8950 non-null   float64
 5   INSTALLMENTS_PURCHASES            8950 non-null   float64
 6   CASH_ADVANCE                      8950 non-null   float64
 7   PURCHASES_FREQUENCY               8950 non-null   float64
 8   ONEOFF_PURCHASES_FREQUENCY        8950 non-null   float64
 9   PURCHASES_INSTALLMENTS_FREQUENCY  8950 non-null   float64
 10  CASH_ADVANCE_FREQUENCY            8950 non-null   float64
 11  CASH_ADVANCE_TRX                  8950 non-null   int64
 12  PURCHASES_TRX                     8950 non-null   int64
 13  CREDIT_LIMIT                      8949 non-null   float64
 14  PAYMENTS                          8950 non-null   float64
 15  MINIMUM_PAYMENTS                  8637 non-null   float64
 16  PRC_FULL_PAYMENT                  8950 non-null   float64
 17  TENURE                            8950 non-null   int64
dtypes: float64(14), int64(3), object(1)
memory usage: 1.2+ MB
```

1.2.1 Removing Duplicated Records, Missing Values, and Outliers from the Dataset

The following code snippets show how you can remove the duplicate records, and process the missing values and outliers from the dataset.

Step 1 - Removing Duplicated Records Let's first look at the data frame and remove the duplicate records if present. To check whether we have duplicate records we can use `df.duplicated().sum()` function.

[7]:
```
# Get a copy of the data frame
df1 = df.copy()

# First check whether there are any duplicates.
```

```
# Since we need to check duplicate elements from all the data in the data frame␣
  ↪we can use the following to check.
print(f'The number of duplicated values in the dataset is: {df1.duplicated().
  ↪sum()}')
```

```
The number of duplicated values in the dataset is: 0
```

Step 2 - Handling Missing Values Since there are no duplications in the dataset, next we need to check for the missing values(NA/null values). To get an idea about whether we have missing values, we can iterate the data rows and check for the unique values in each column.

[8]:
```
# Checking the unique data in the dataset
for col_name in df1:
    print(f'==========={col_name}===========')
    print(df1[col_name].unique())
    print('\n')
```

```
===========CUST_ID===========
['C10001' 'C10002' 'C10003' … 'C19188' 'C19189' 'C19190']


===========BALANCE===========
[  40.900749 3202.467416 2495.148862 …   23.398673   13.457564
  372.708075]


===========BALANCE_FREQUENCY===========
[0.818182 0.909091 1.       0.636364 0.545455 0.875    0.454545 0.727273
 0.5      0.888889 0.090909 0.272727 0.363636 0.       0.666667 0.75
 0.857143 0.181818 0.333333 0.6      0.3      0.125    0.9      0.833333
 0.8      0.2      0.777778 0.555556 0.25     0.142857 0.571429 0.4
 0.444444 0.714286 0.222222 0.1      0.625    0.428571 0.111111 0.285714
 0.7      0.375    0.166667]


===========PURCHASES===========
[  95.4     0.    773.17 …  291.12  144.4  1093.25]


===========ONEOFF_PURCHASES===========
[   0.    773.17 1499.   …  734.4  1012.73 1093.25]


===========INSTALLMENTS_PURCHASES===========
[  95.4     0.   1333.28 …  113.28  291.12  144.4 ]


===========CASH_ADVANCE===========
```

```
[   0.       6442.945483  205.788017 … 8555.409326   36.558778
  127.040008]


==========PURCHASES_FREQUENCY==========
[0.166667 0.       1.       0.083333 0.666667 0.333333 0.25     0.75
 0.5      0.416667 0.916667 0.583333 0.375    0.625    0.272727 0.833333
 0.909091 0.111111 0.142857 0.090909 0.363636 0.1      0.875    0.125
 0.818182 0.636364 0.2      0.8      0.3      0.9      0.285714 0.727273
 0.181818 0.7      0.545455 0.888889 0.714286 0.454545 0.857143 0.555556
 0.428571 0.4      0.571429 0.6      0.222222 0.777778 0.444444]


==========ONEOFF_PURCHASES_FREQUENCY==========
[0.       1.       0.083333 0.166667 0.25     0.916667 0.5      0.416667
 0.333333 0.666667 0.375    0.583333 0.1      0.090909 0.833333 0.75
 0.111111 0.142857 0.125    0.875    0.363636 0.2      0.818182 0.8
 0.3      0.636364 0.181818 0.909091 0.285714 0.222222 0.727273 0.571429
 0.6      0.272727 0.714286 0.545455 0.428571 0.444444 0.454545 0.625
 0.777778 0.555556 0.7      0.9      0.4      0.857143 0.888889]


==========PURCHASES_INSTALLMENTS_FREQUENCY==========
[0.083333 0.       0.583333 1.       0.25     0.916667 0.75     0.5
 0.333333 0.666667 0.416667 0.166667 0.833333 0.4      0.181818 0.818182
 0.272727 0.375    0.125    0.636364 0.545455 0.909091 0.888889 0.2
 0.8      0.1      0.142857 0.857143 0.444444 0.454545 0.111111 0.6
 0.555556 0.777778 0.3      0.9      0.363636 0.714286 0.875    0.222222
 0.285714 0.7      0.727273 0.571429 0.090909 0.428571 0.625   ]


==========CASH_ADVANCE_FREQUENCY==========
[0.       0.25     0.083333 0.166667 0.333333 0.363636 0.833333 0.5
 0.727273 0.125    0.875    0.111111 0.416667 0.181818 0.545455 0.75
 0.142857 0.583333 0.666667 0.222222 0.285714 0.909091 0.2      0.625
 0.090909 0.4      1.       0.8      0.636364 0.3      0.916667 0.444444
 1.25     0.1      0.428571 0.272727 0.555556 0.6      0.454545 1.166667
 0.375    0.777778 0.714286 0.571429 0.857143 1.125    1.1      1.5
 0.7      0.818182 0.9      0.888889 1.090909 1.142857]


==========CASH_ADVANCE_TRX==========
[  0   4   1   3   6  13   5  16  10   2  11   7  12  37  27  23  21  14
  40   8   9  26  15  18  28  24  20  17  22  31 123  52  51  62  19  25
  30  29  53  45  43  42 107  38  56  39  32  33  50  34  63  36 110  47
  48  71  35  93  80  44  61  46  49  69  41]
```

```
==========PURCHASES_TRX==========
[  2   0  12   1   8  64   5   3   6  26  11   9  92  17  13  45  14  41
  27  20  87  18   4  42  61  33   7  50  22  23  60  46  75  31  10  34
  81  25  85 217  19  52 216  97  24  77 130  90  44  39  15  30  36 123
 151  21 101  49  98  28  84  93  72  38  99  62  48  16  32  51  74  29
  59  76  47 126 229  40 103 121 157 114  53  83  43  54 222  66 141  37
  79  70  80 194 117 100 111  67 219  55 152 104 182  88  82  71  78  35
 122 105 108  69 175 135  91  65  68 119  63 140 113 358  58 248 129  56
  89  57 139 176 136 195  73 109 208 115 110 147 273 102 185 171 168 232
  95  86 148 112 128 254 198 298 154 116 142 131 347 204 200  94 118 344
 162 308 199 309  96 274 224 143 133 127 186]


==========CREDIT_LIMIT==========
[ 1000.        7000.        7500.        1200.        1800.
 13500.        2300.       11000.        2000.        3000.
  8000.        2500.       13000.        4000.       11250.
  9000.        6000.        1700.       10500.        6900.
  5000.        4500.        1500.        8500.        2400.
  4200.        3300.       12000.        3500.        6500.
  1600.        4150.        1850.        6250.        2250.
  9500.       16500.        5700.        5500.       17000.
  3200.       19000.        2800.       18000.       21500.
 10000.       20000.        7900.       15000.       12500.
 14000.        5300.         900.        6150.       11500.
 23000.        2700.       14500.       19500.        1400.
  2900.        1950.        7200.       10950.        2100.
   500.       21000.        4800.        7300.       18500.
  6550.        6200.        3150.       11800.       11300.
  2750.       10300.        2850.       17500.       22500.
  6750.       16000.        1750.       15500.        3800.
  6600.        3650.        2600.        4600.        3350.
  4300.        5250.       28000.        3600.       10200.
   750.        7600.        2720.        2200.        7350.
   300.        3750.        1900.       10400.        5200.
  4350.        3050.        6700.       20500.        5750.
  8300.        7950.        3100.        4050.        6300.
  3666.666667   200.        9100.        3700.        7100.
  4750.        3900.       10100.        1300.       19600.
 10450.       11150.         600.        4700.         150.
  3400.        1350.        5100.        7227.272727 13550.
  9200.        6100.        2050.        6400.        4250.
 14600.        8600.        3511.111111  5600.        6727.272727
 13600.        8800.       11100.       12200.        8100.
  8050.        4100.        5550.        4450.        5650.
  8200.        5400.       25000.        9950.        1150.
  6800.        4650.        7800.        2150.       22000.
   400.        7700.        4400.        1050.         700.
```

```
 5181.818182  4900.        1100.        2550.        2283.333333
30000.        9700.        9400.              nan  2725.
  450.        8700.       12300.        7050.        1550.
 9800.        5450.         800.        8954.545455  2950.
 2450.        6850.        4950.        5900.        1250.
17150.        9600.          50.       10750.        1120.
13450.        3777.777778   650.        1450.        2350.
 1833.333333]
```

```
==========PAYMENTS==========
[ 201.802084 4103.032597  622.066742 …   81.270775   52.549959
   63.165404]
```

```
==========MINIMUM_PAYMENTS==========
[ 139.509787 1072.340217  627.284787 …   82.418369   55.755628
   88.288956]
```

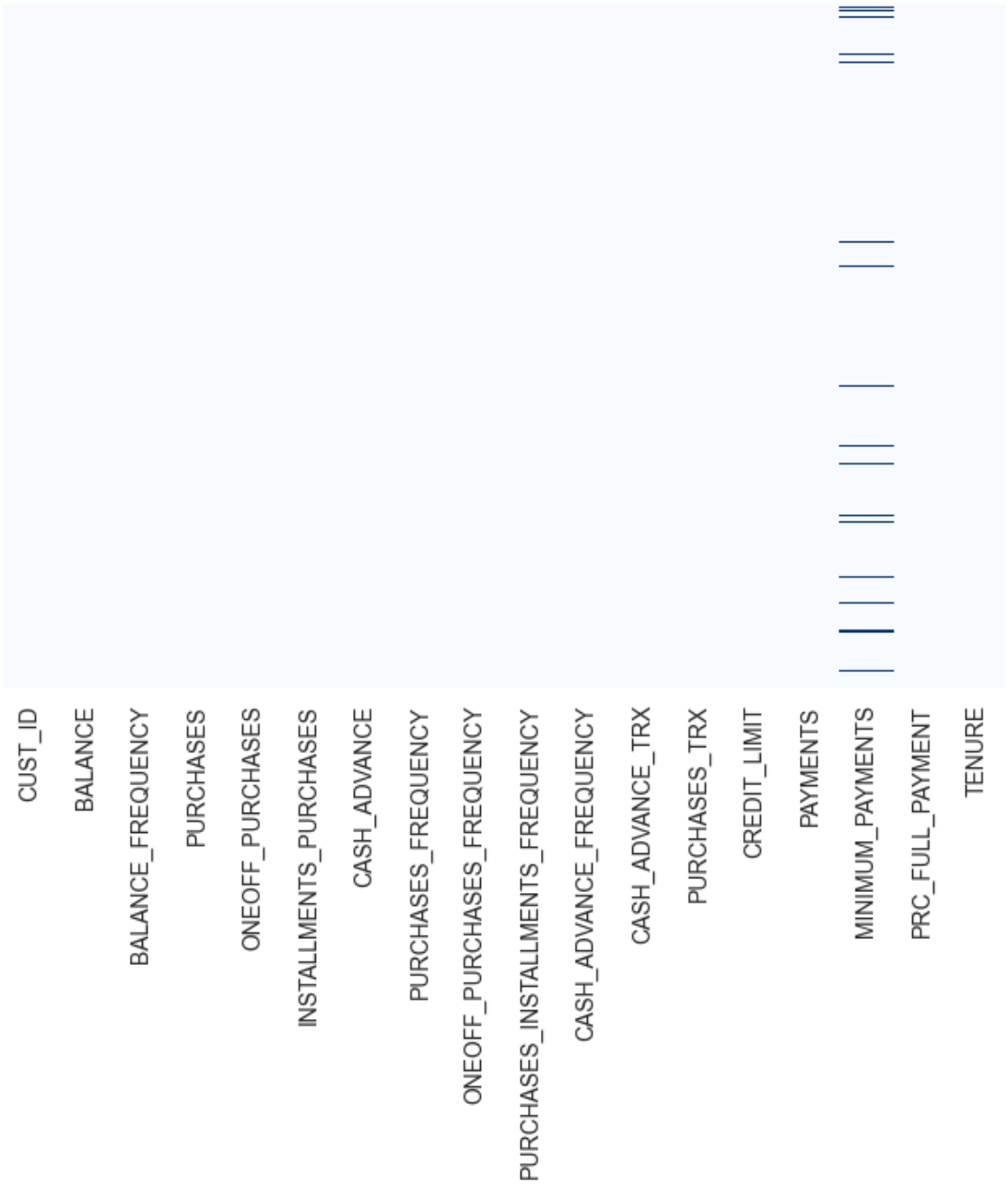
```
==========PRC_FULL_PAYMENT==========
[0.       0.222222 1.       0.25     0.083333 0.3      0.333333 0.166667
 0.111111 0.916667 0.2      0.090909 0.454545 0.181818 0.444444 0.636364
 0.5      0.75     0.142857 0.888889 0.545455 0.818182 0.363636 0.833333
 0.666667 0.909091 0.1      0.583333 0.8      0.416667 0.4      0.125
 0.714286 0.6      0.571429 0.375    0.9      0.285714 0.7      0.272727
 0.777778 0.875    0.727273 0.428571 0.625    0.857143 0.555556]
```

```
==========TENURE==========
[12  8 11  9 10  7  6]
```

As you can see, to check for missing values in each column, we have to check the data manually, and it is a laborious task. Therefore, what we can do first is not check for the unique values, but plot the missing values using the `Seaborn` library's `heatmap()` function. This will give us an idea about whether we have missing values or not.

[9]:
```
# View the heatmap to check null/NaN values
sns.heatmap(df1.isnull(), yticklabels=False, cbar=False, cmap="Blues")
```

[9]: <Axes: >



Since we can see that there are missing values present in the dataset(because of the graph), next we need to check how many missing values are present in each column. To get that information you can use the below given options. - `df.isnull().sum()` - `df.info()`

[10]:
```
# Similarly check with df.isnull().sum() function to get the count of missing␣
↪values
df1.isnull().sum()
```

[10]:
```
CUST_ID                             0
BALANCE                             0
BALANCE_FREQUENCY                   0
PURCHASES                           0
ONEOFF_PURCHASES                    0
INSTALLMENTS_PURCHASES              0
CASH_ADVANCE                        0
PURCHASES_FREQUENCY                 0
ONEOFF_PURCHASES_FREQUENCY          0
PURCHASES_INSTALLMENTS_FREQUENCY    0
CASH_ADVANCE_FREQUENCY              0
CASH_ADVANCE_TRX                    0
PURCHASES_TRX                       0
CREDIT_LIMIT                        1
PAYMENTS                            0
MINIMUM_PAYMENTS                  313
PRC_FULL_PAYMENT                    0
TENURE                              0
dtype: int64
```

There are two options to handle missing data in a data set.

1. If the missing value count of a column is greater than 50% - 70%, then we need to remove that column from the dataset.
2. If the missing value count of a column is less than 50% - 70%, then we need to impute the missing values.

Since we only have 1 missing value from `CREDIT_LIMIT` and 313 missing values from `MINIMUM_PAYMENTS`, we can simply replace(data imputation) the missing values with the mean values of their respective columns.

[11]:
```
# Fill up the missing elements in the 'MINIMUM_PAYMENTS' with the mean of the␣
 ↪'MINIMUM_PAYMENTS'
df1.loc[(df1['MINIMUM_PAYMENTS'].isnull() == True), 'MINIMUM_PAYMENTS'] =␣
 ↪df1['MINIMUM_PAYMENTS'].mean()
```

[12]:
```
# Check again to verify whether the missing values have been filled or not␣
 ↪after data imputation
df1.isnull().sum()
```

[12]:
```
CUST_ID                           0
BALANCE                           0
BALANCE_FREQUENCY                 0
PURCHASES                         0
ONEOFF_PURCHASES                  0
INSTALLMENTS_PURCHASES            0
CASH_ADVANCE                      0
PURCHASES_FREQUENCY               0
ONEOFF_PURCHASES_FREQUENCY        0
```

```
PURCHASES_INSTALLMENTS_FREQUENCY    0
CASH_ADVANCE_FREQUENCY              0
CASH_ADVANCE_TRX                    0
PURCHASES_TRX                       0
CREDIT_LIMIT                        1
PAYMENTS                            0
MINIMUM_PAYMENTS                    0
PRC_FULL_PAYMENT                    0
TENURE                              0
dtype: int64
```

[13]:
```
# Fill up the missing elements in the 'CREDIT_LIMIT' with the mean of the␣
↪'CREDIT_LIMIT'
df1.loc[(df1['CREDIT_LIMIT'].isnull() == True), 'CREDIT_LIMIT'] =␣
↪df1['CREDIT_LIMIT'].mean()
```

[14]:
```
# Check again to verify whether the missing values have been filled or not
df1.isnull().sum()
```

[14]:
```
CUST_ID                             0
BALANCE                             0
BALANCE_FREQUENCY                   0
PURCHASES                           0
ONEOFF_PURCHASES                    0
INSTALLMENTS_PURCHASES              0
CASH_ADVANCE                        0
PURCHASES_FREQUENCY                 0
ONEOFF_PURCHASES_FREQUENCY          0
PURCHASES_INSTALLMENTS_FREQUENCY    0
CASH_ADVANCE_FREQUENCY              0
CASH_ADVANCE_TRX                    0
PURCHASES_TRX                       0
CREDIT_LIMIT                        0
PAYMENTS                            0
MINIMUM_PAYMENTS                    0
PRC_FULL_PAYMENT                    0
TENURE                              0
dtype: int64
```

Furthermore, we don't need the `CUST_ID` column since it only contains a unique identifier for the user and cannot used to provide a machine-learning solution to predict the customer ID. Therefore, we can drop it from our dataset.

[15]:
```
# We don't need the column 'CUST_ID'. Therefore, let's remove that column.
df1.drop('CUST_ID', axis=1, inplace=True)
```

[16]:
```
# Check the data to verify whether the 'CUST_ID' column is removed or not
df1.head(10)
```

[16]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES
0	40.900749	0.818182	95.40	0.00
1	3202.467416	0.909091	0.00	0.00
2	2495.148862	1.000000	773.17	773.17
3	1666.670542	0.636364	1499.00	1499.00
4	817.714335	1.000000	16.00	16.00
5	1809.828751	1.000000	1333.28	0.00
6	627.260806	1.000000	7091.01	6402.63
7	1823.652743	1.000000	436.20	0.00
8	1014.926473	1.000000	861.49	661.49
9	152.225975	0.545455	1281.60	1281.60

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY
0	95.40	0.000000	0.166667
1	0.00	6442.945483	0.000000
2	0.00	0.000000	1.000000
3	0.00	205.788017	0.083333
4	0.00	0.000000	0.083333
5	1333.28	0.000000	0.666667
6	688.38	0.000000	1.000000
7	436.20	0.000000	1.000000
8	200.00	0.000000	0.333333
9	0.00	0.000000	0.166667

	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY
0	0.000000	0.083333
1	0.000000	0.000000
2	1.000000	0.000000
3	0.083333	0.000000
4	0.083333	0.000000
5	0.000000	0.583333
6	1.000000	1.000000
7	0.000000	1.000000
8	0.083333	0.250000
9	0.166667	0.000000

	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX	PURCHASES_TRX	CREDIT_LIMIT
0	0.000000	0	2	1000.0
1	0.250000	4	0	7000.0
2	0.000000	0	12	7500.0
3	0.083333	1	1	7500.0
4	0.000000	0	1	1200.0
5	0.000000	0	8	1800.0
6	0.000000	0	64	13500.0
7	0.000000	0	12	2300.0
8	0.000000	0	5	7000.0
9	0.000000	0	3	11000.0

	PAYMENTS	MINIMUM_PAYMENTS	PRC_FULL_PAYMENT	TENURE
0	201.802084	139.509787	0.000000	12
1	4103.032597	1072.340217	0.222222	12
2	622.066742	627.284787	0.000000	12
3	0.000000	864.206542	0.000000	12
4	678.334763	244.791237	0.000000	12
5	1400.057770	2407.246035	0.000000	12
6	6354.314328	198.065894	1.000000	12
7	679.065082	532.033990	0.000000	12
8	688.278568	311.963409	0.000000	12
9	1164.770591	100.302262	0.000000	12

Step 3 - Removing/Capping Outliers Now, what we need to do is check for the outliers in the dataset and remove them. There are a few ways that you can do to remove the outliers in a dataset.

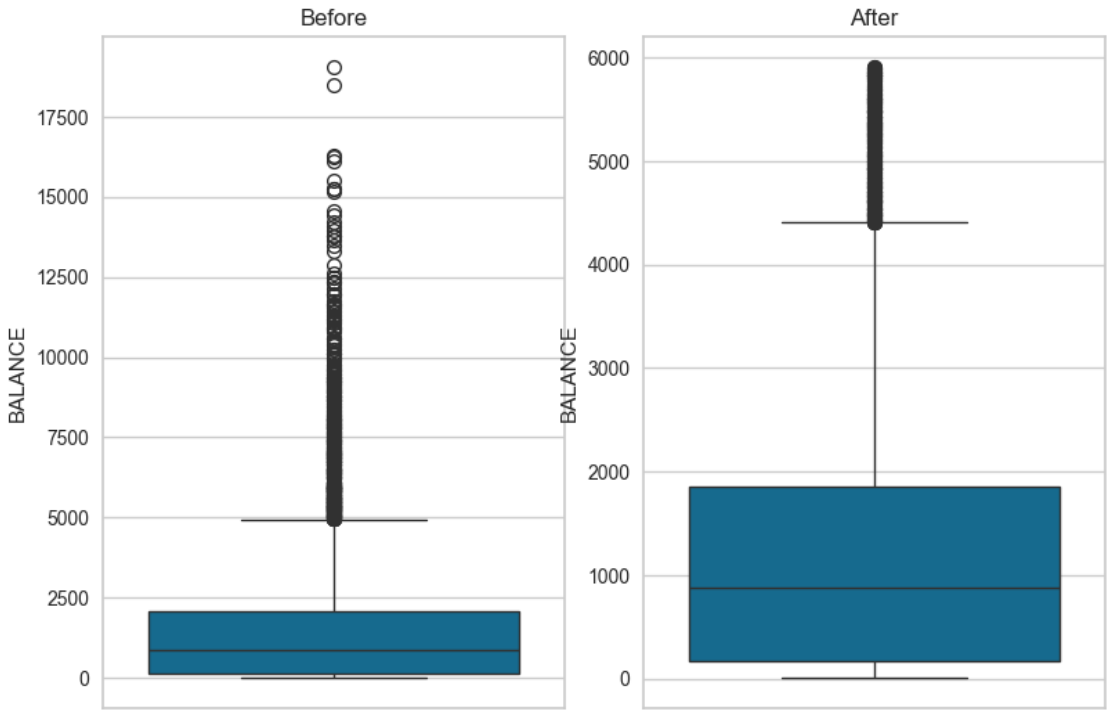
1. **Z-Score** → Calculate the Z-score for each observation and remove the data points with a Z-Score beyond the threshold.
2. **Inter-quartile Range(IQR) Method** → Compute the IQR for the data and remove data points that fall below $Q_1 - 1.5 \times IQR$ or above $Q_3 + 1.5 \times IQR$ where is the Q_1 25th percentile and Q_3 is the 75th percentile.
3. **Percentile Method** → Rather than using the 25th and 75th percentile, we can define custom ranges and use them.

Using Percentile Method We want to check the outliers with numerical values. Therefore, we first need to remove the categorical values and other binary values present in the data frame. To do that, we can copy the data frame using `df.copy()` and select a slice from that where we have the necessary numerical columns. The following code snippet shows how to get the results using the percentile method.

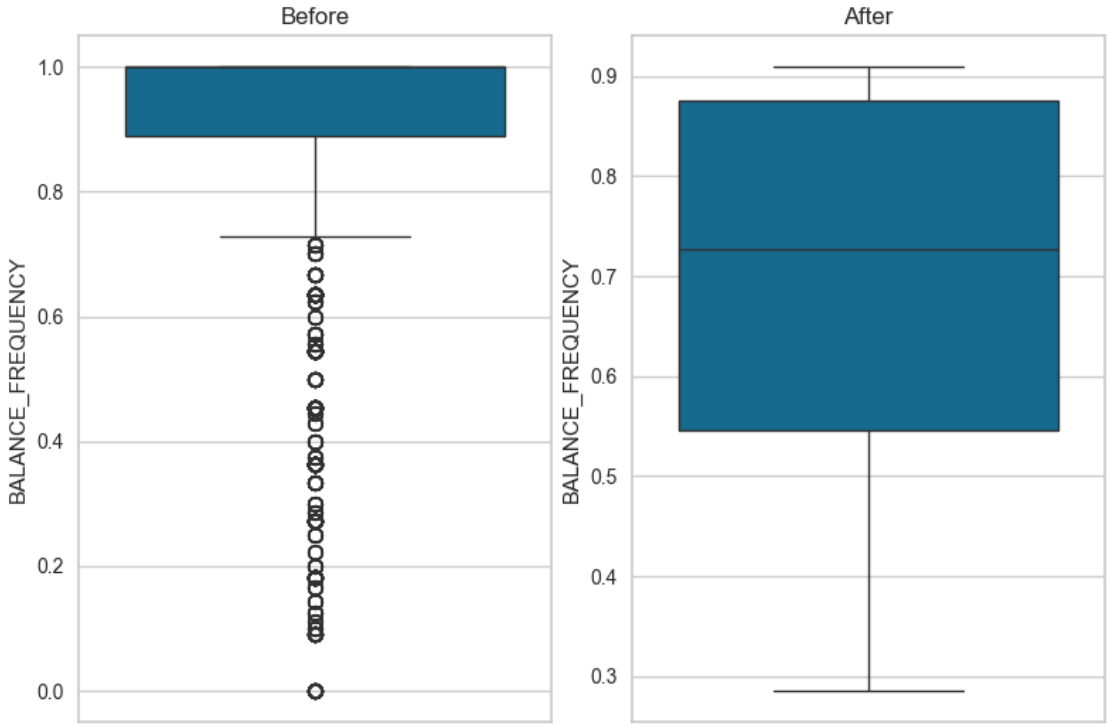
[17]:
```
# Handling the outliers using the percentile method
# Copy the data frame
df2 = df1.copy()

# Get the numerical features
numerical_features = df2.dtypes!=object
numerical_fields = df2.columns[numerical_features].tolist()

# Remove unwanted fields if there are any
UNWANTED_FIELDS = ['TENURE']
numerical_fields = [field for field in numerical_fields if field not in␣
 ↪UNWANTED_FIELDS]

# Removing outliers using the percentile method
def remove_outliers_using_percentile_method(field):
    fig, axes = plt.subplots(1,2)
```

```
    plt.tight_layout()
    print(f'Before removing outliers the shape of {field}:{df1.shape}')
    sns.boxplot(df1[field], orient='v', ax=axes[0])
    axes[0].title.set_text('Before')

    # Max and min percentile
    max_percentile_val = df1[field].quantile(0.95)
    min_percentile_val = df1[field].quantile(0.05)

    # Remove the outliers
    df2 = df1[(df1[field] > min_percentile_val) & (df1[field] <␣
 ↪max_percentile_val)]

    # Visualize
    print(f'After removing outliers the shape of {field}:{df2.shape}')
    sns.boxplot(df2[field], orient='v', ax=axes[1])
    axes[1].title.set_text('After')

    # Show plots
    plt.show()

for field in numerical_fields:
    remove_outliers_using_percentile_method(field)
```

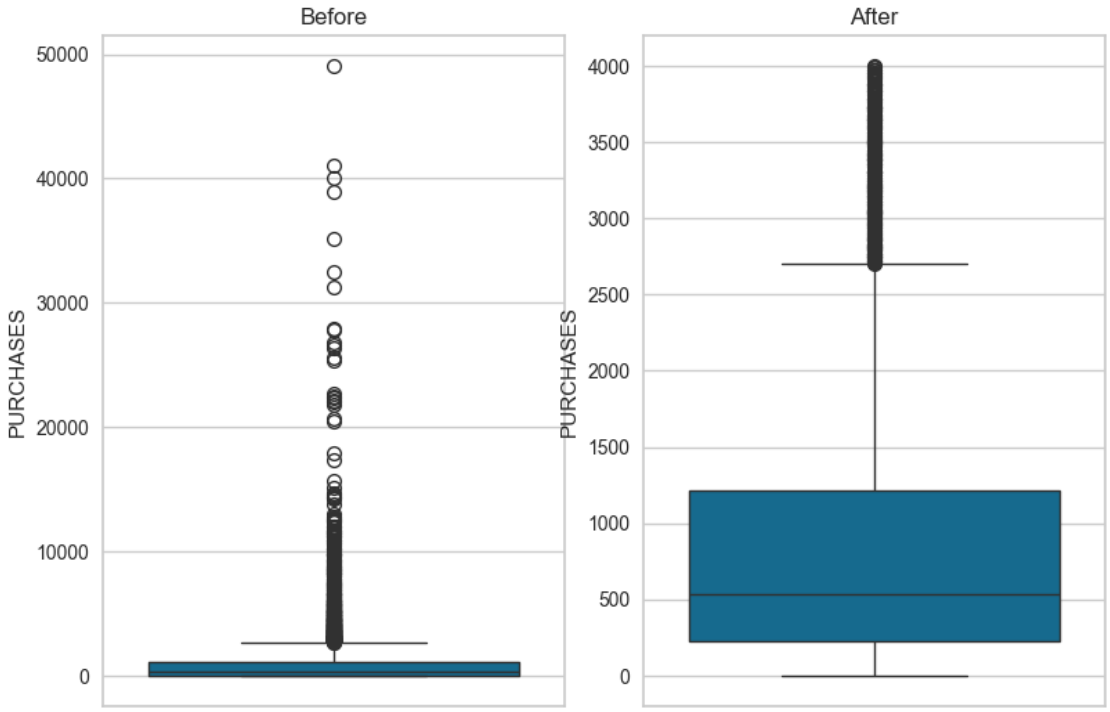
```
Before removing outliers the shape of BALANCE:(8950, 17)
After removing outliers the shape of BALANCE:(8054, 17)
```



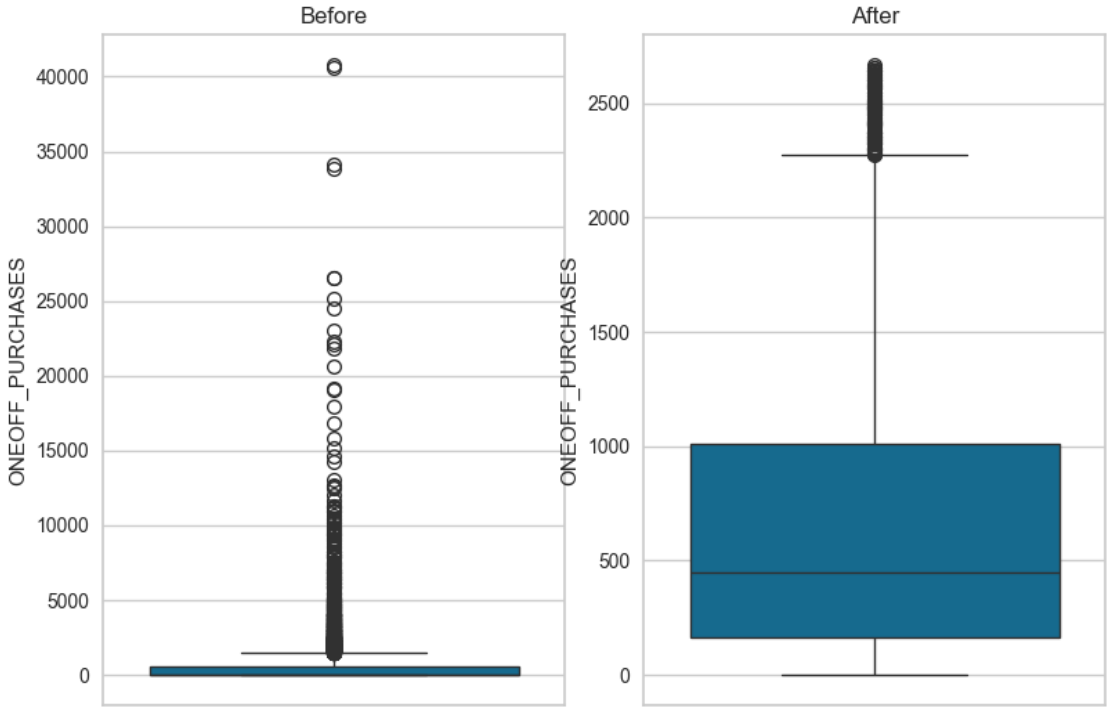
```
Before removing outliers the shape of BALANCE_FREQUENCY:(8950, 17)
After removing outliers the shape of BALANCE_FREQUENCY:(2237, 17)
```



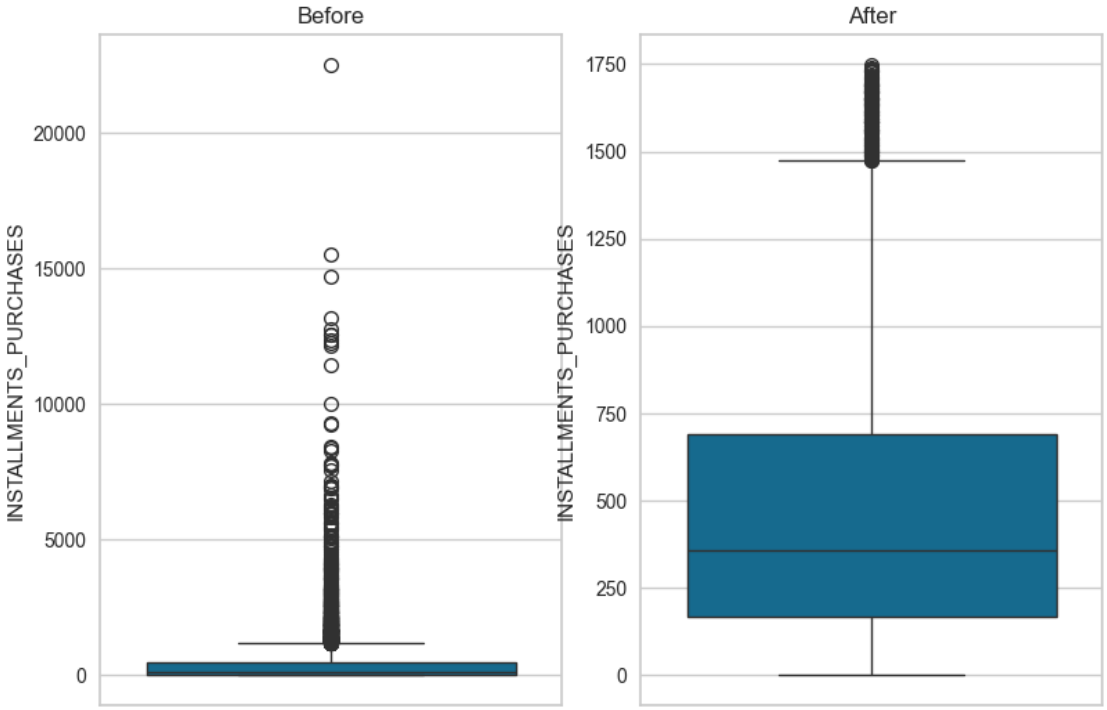

```
Before removing outliers the shape of PURCHASES:(8950, 17)
After removing outliers the shape of PURCHASES:(6458, 17)
```



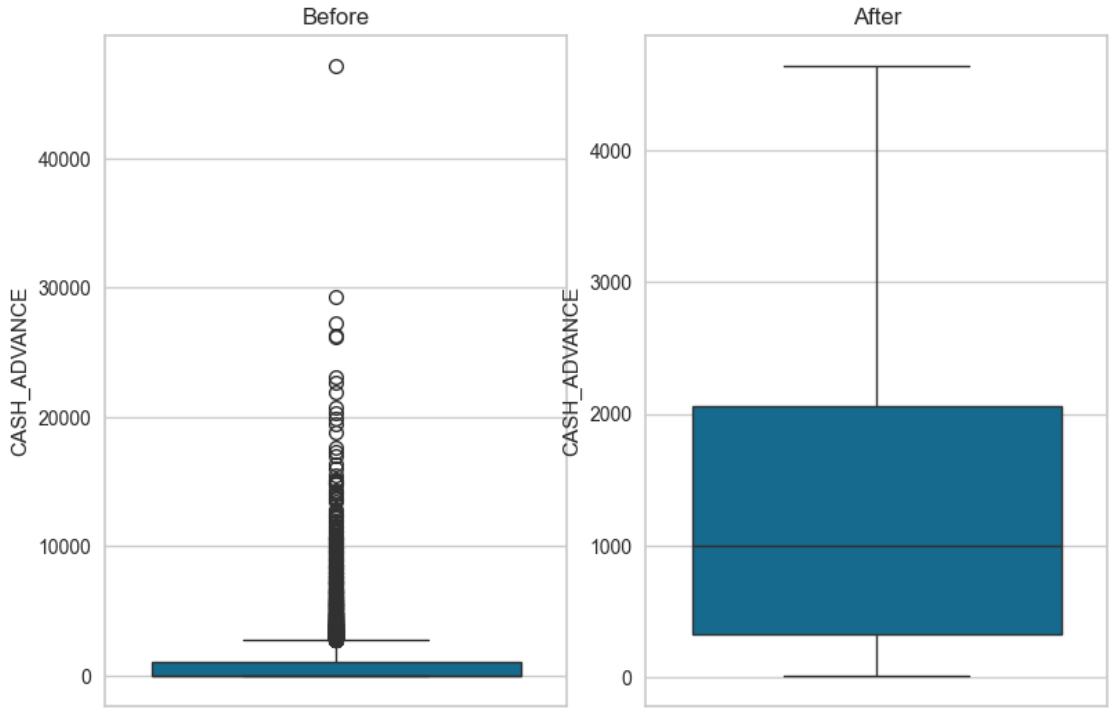

```
Before removing outliers the shape of ONEOFF_PURCHASES:(8950, 17)
After removing outliers the shape of ONEOFF_PURCHASES:(4200, 17)
```



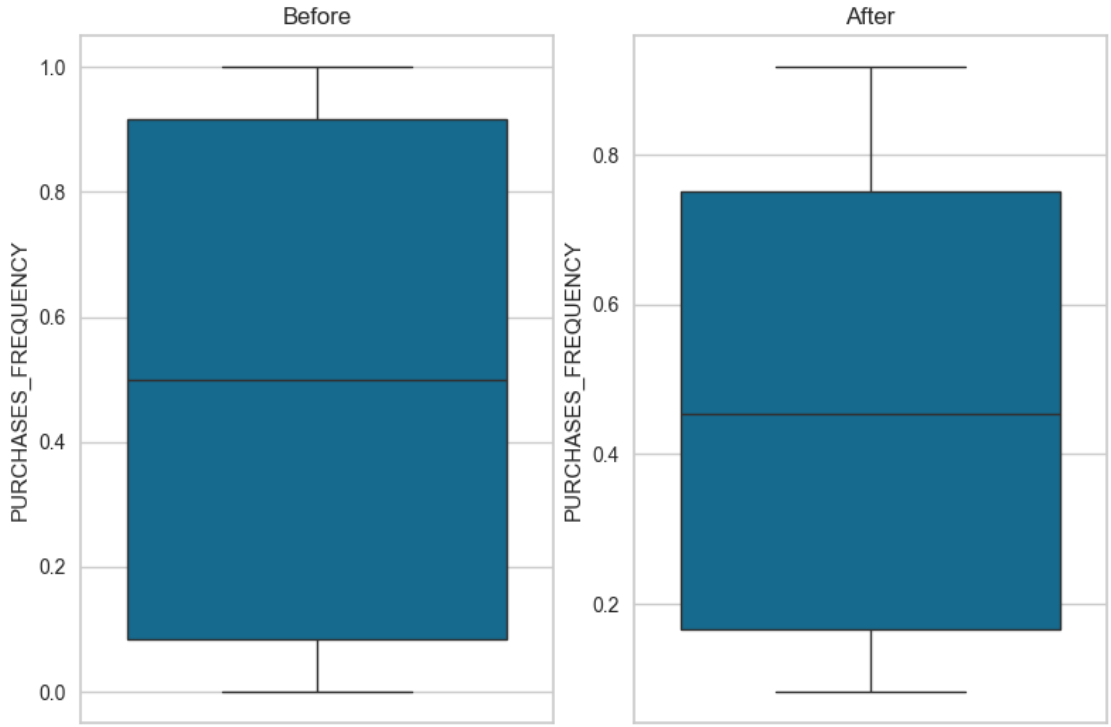
```
Before removing outliers the shape of INSTALLMENTS_PURCHASES:(8950, 17)
After removing outliers the shape of INSTALLMENTS_PURCHASES:(4586, 17)
```



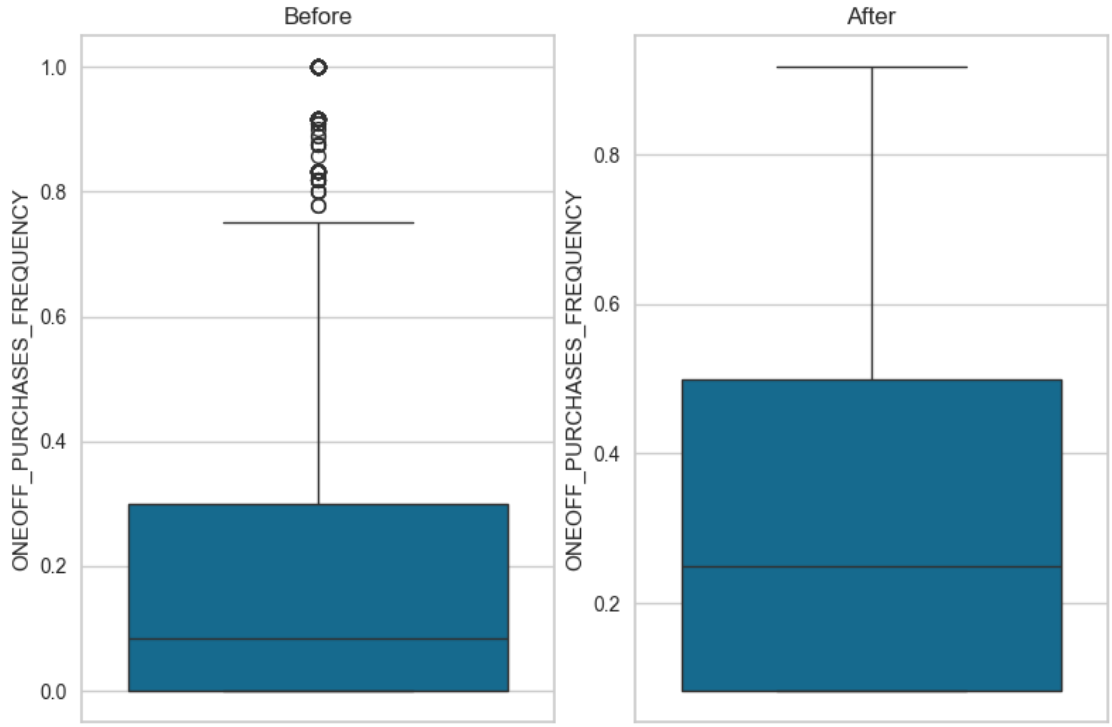

```
Before removing outliers the shape of CASH_ADVANCE:(8950, 17)
After removing outliers the shape of CASH_ADVANCE:(3874, 17)
```



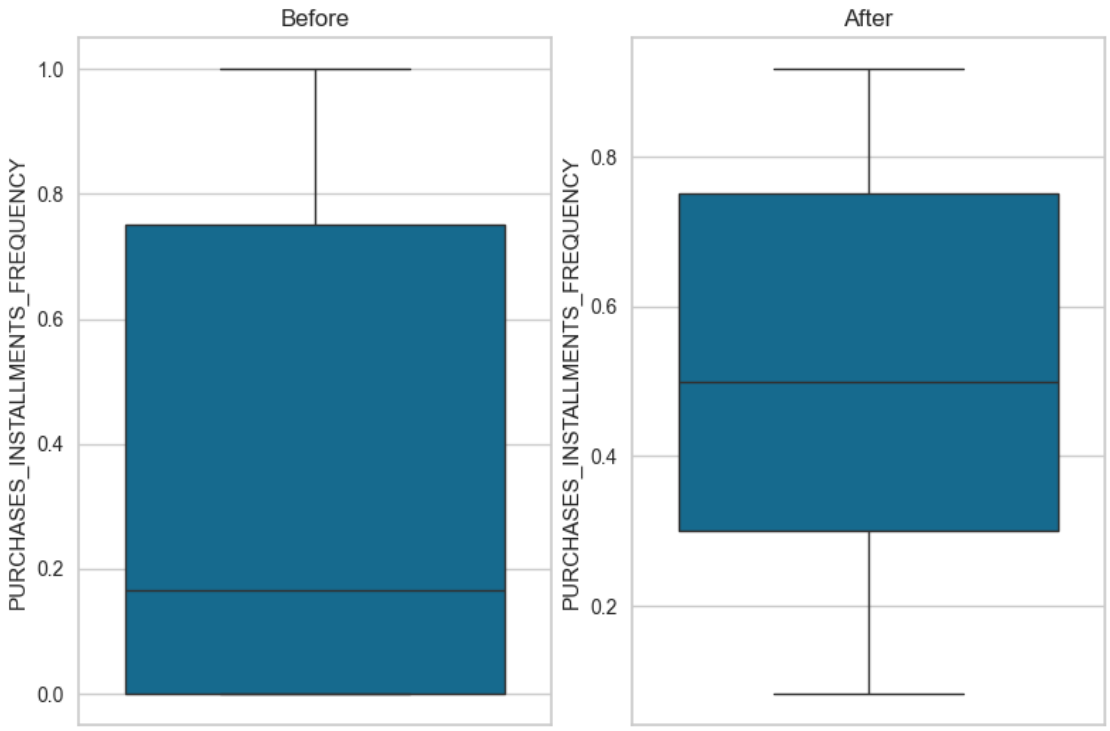

```
Before removing outliers the shape of PURCHASES_FREQUENCY:(8950, 17)
After removing outliers the shape of PURCHASES_FREQUENCY:(4729, 17)
```



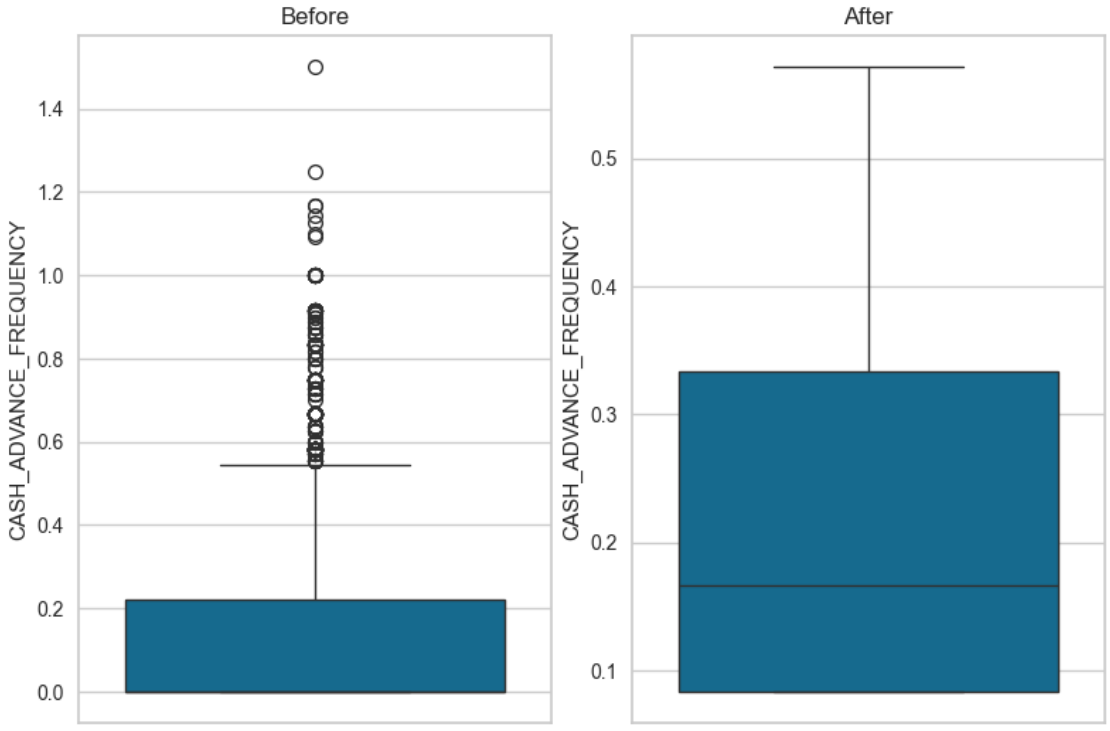
```
Before removing outliers the shape of ONEOFF_PURCHASES_FREQUENCY:(8950, 17)
After removing outliers the shape of ONEOFF_PURCHASES_FREQUENCY:(4167, 17)
```



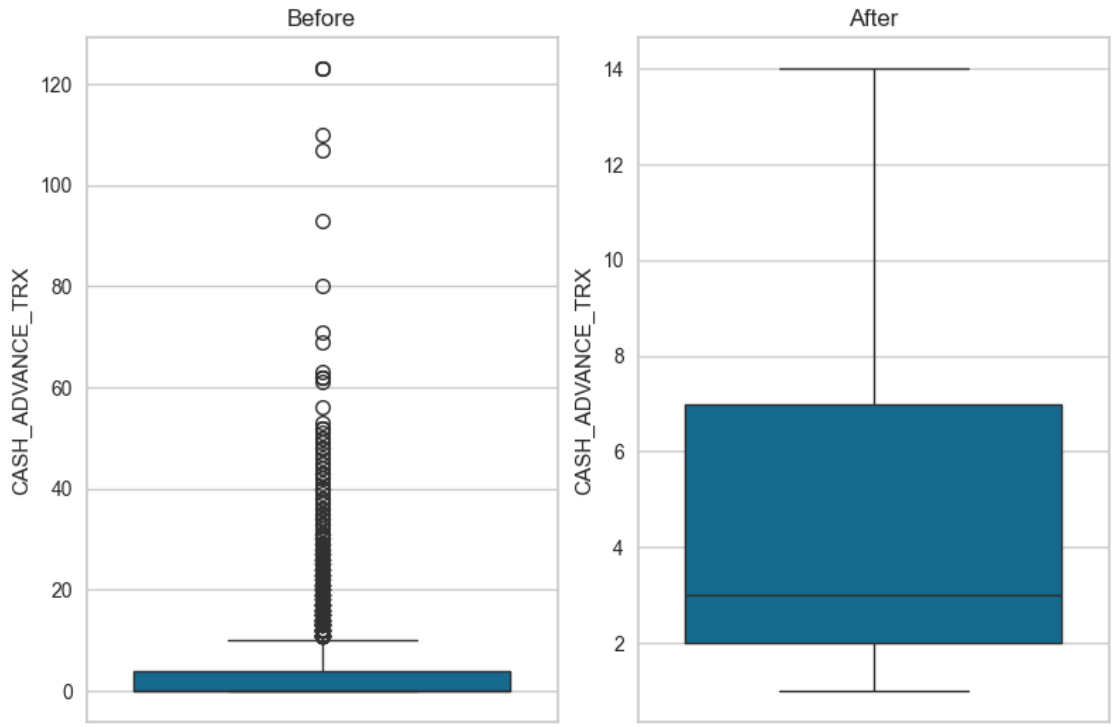

```
Before removing outliers the shape of PURCHASES_INSTALLMENTS_FREQUENCY:(8950,
17)
After removing outliers the shape of PURCHASES_INSTALLMENTS_FREQUENCY:(3704, 17)
```



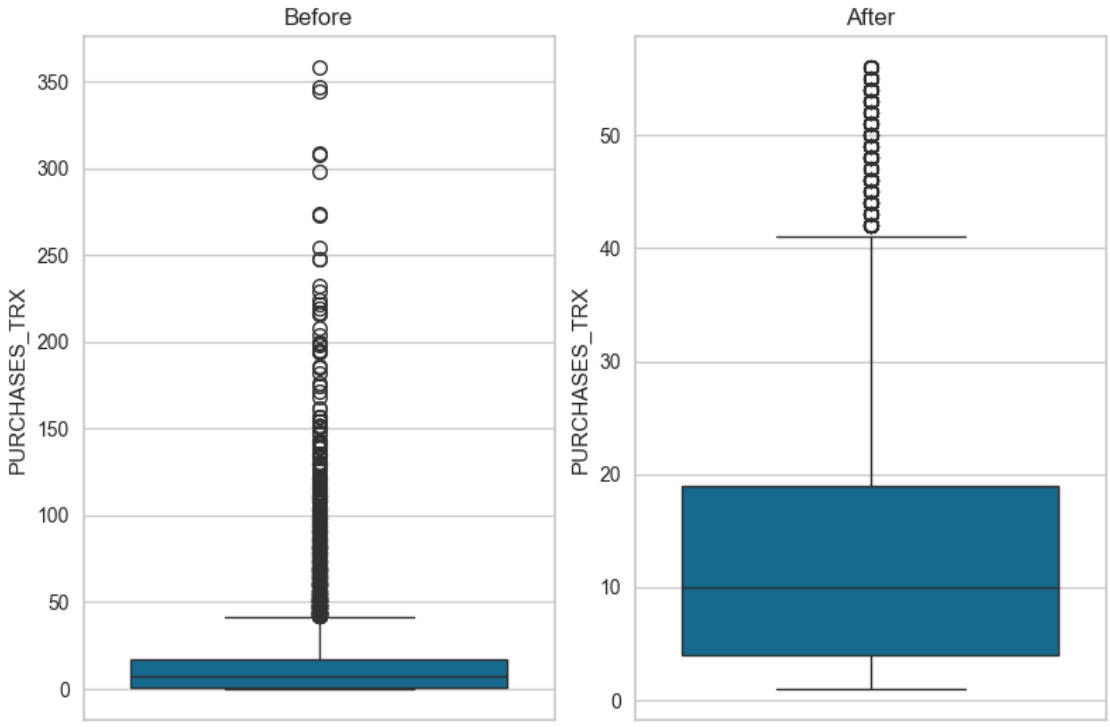

```
Before removing outliers the shape of CASH_ADVANCE_FREQUENCY:(8950, 17)
After removing outliers the shape of CASH_ADVANCE_FREQUENCY:(3821, 17)
```



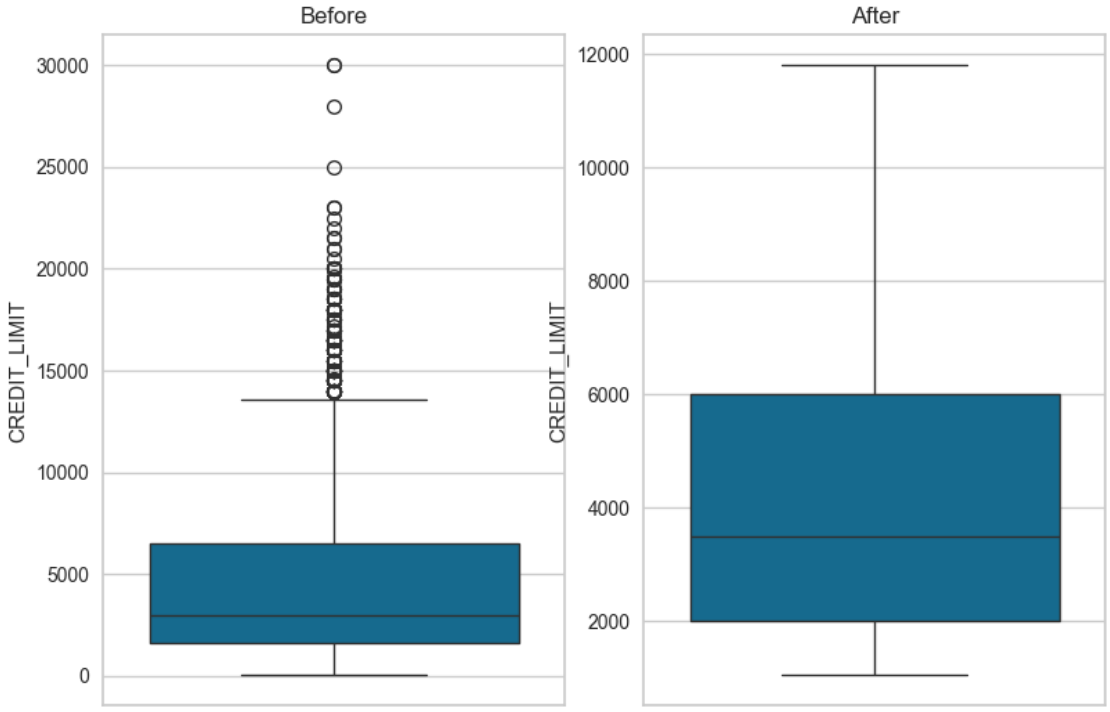
```
Before removing outliers the shape of CASH_ADVANCE_TRX:(8950, 17)
After removing outliers the shape of CASH_ADVANCE_TRX:(3852, 17)
```



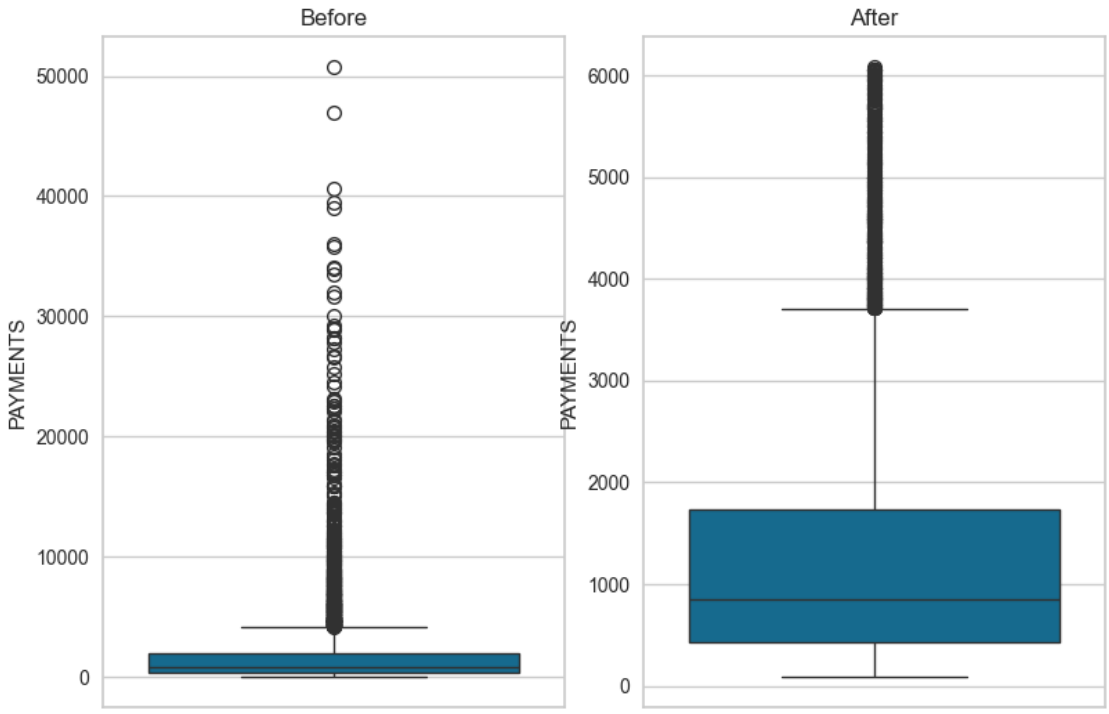

```
Before removing outliers the shape of PURCHASES_TRX:(8950, 17)
After removing outliers the shape of PURCHASES_TRX:(6454, 17)
```



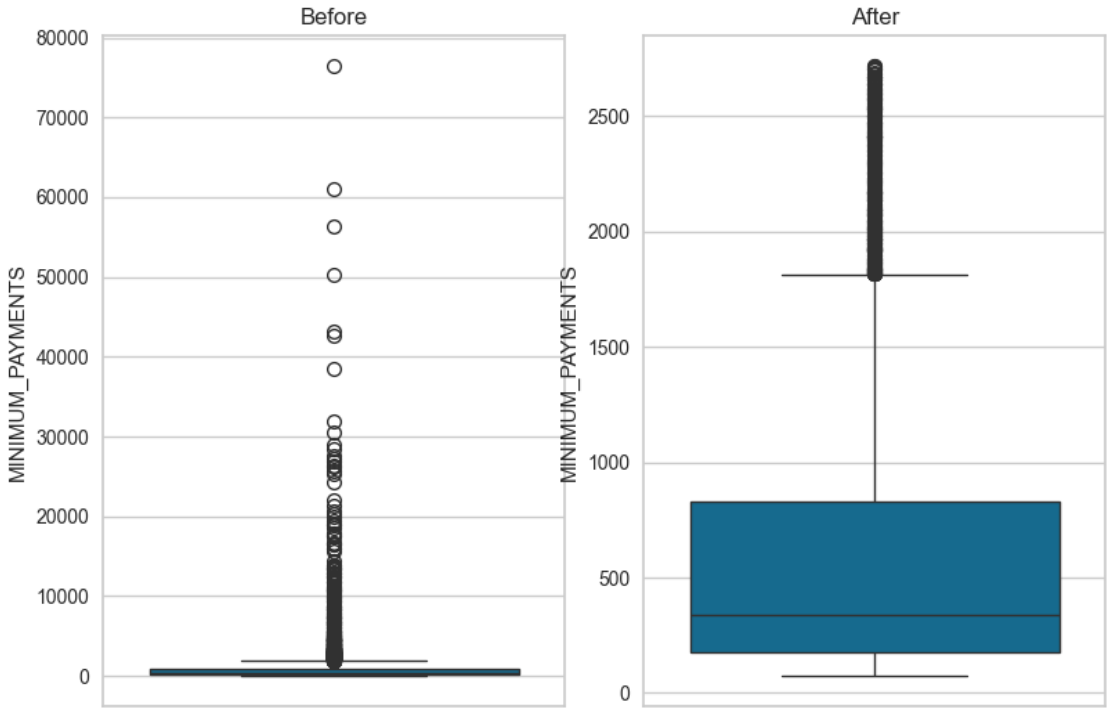

```
Before removing outliers the shape of CREDIT_LIMIT:(8950, 17)
After removing outliers the shape of CREDIT_LIMIT:(7659, 17)
```



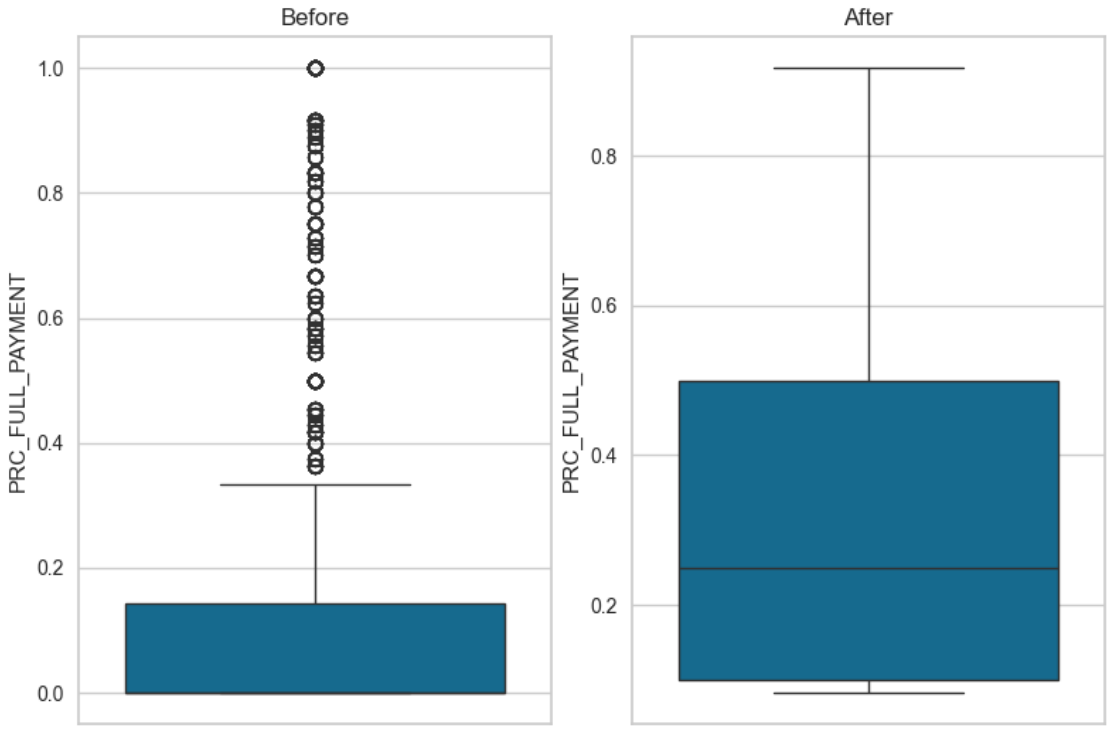
```
Before removing outliers the shape of PAYMENTS:(8950, 17)
After removing outliers the shape of PAYMENTS:(8054, 17)
```




```
Before removing outliers the shape of MINIMUM_PAYMENTS:(8950, 17)
After removing outliers the shape of MINIMUM_PAYMENTS:(8054, 17)
```




```
Before removing outliers the shape of PRC_FULL_PAYMENT:(8950, 17)
After removing outliers the shape of PRC_FULL_PAYMENT:(2559, 17)
```

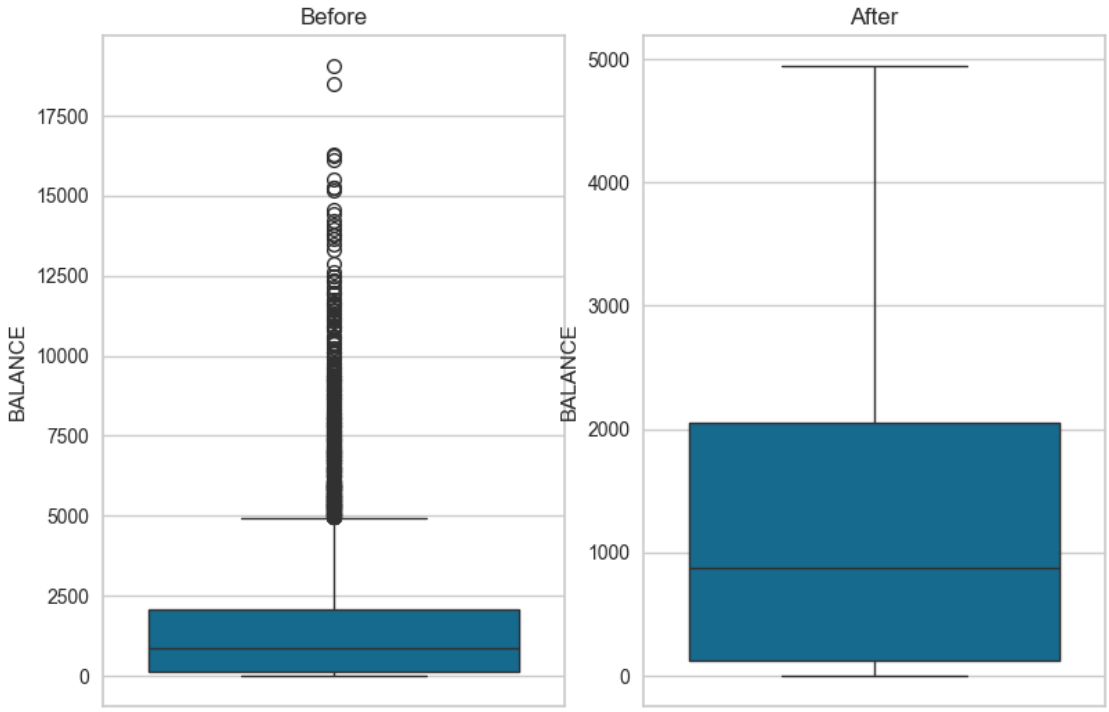


It can be seen that if we use the `Percentile Method` there will be a huge loss in data. Therefore, we need to find an alternative option to handle the outliers. Now, let's check what kind of results we can get from using the `Inter-quartile Range` method.

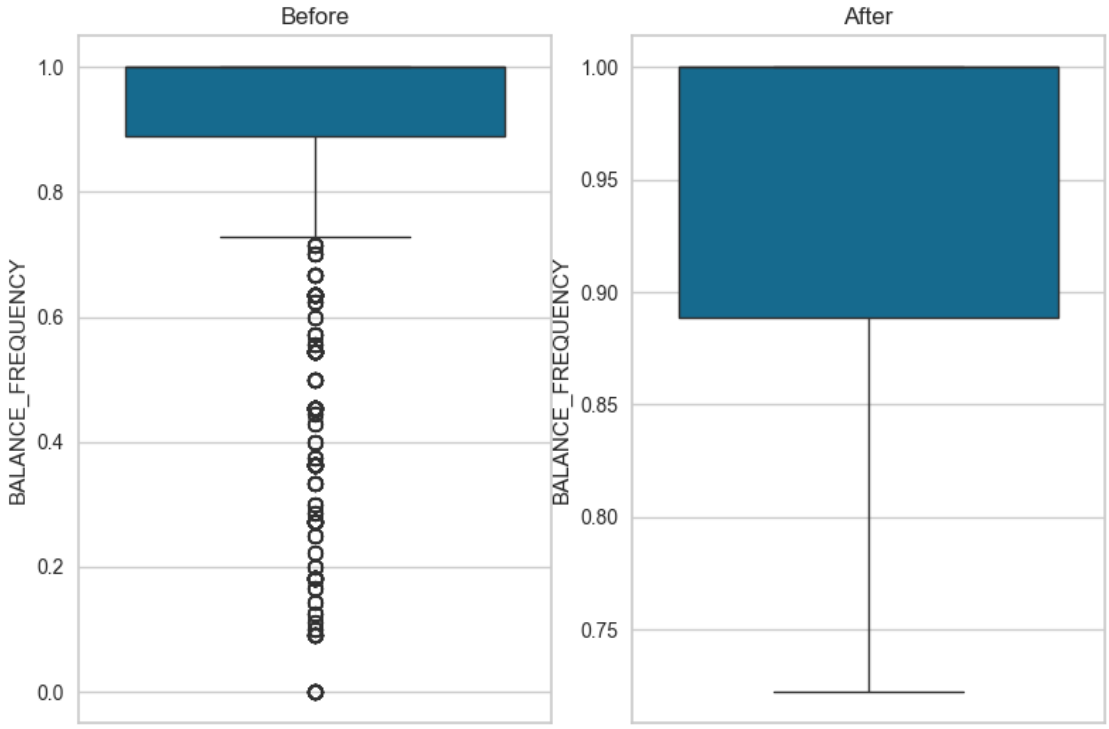
Inter-quartile Range Method The following code snippet is used to get the calculated `Inter-quartile Range` and get the necessary results. Next, let's check the results we have obtained from the above code snippet.

[18]:
```
# Handling outliers using the IQR method
# Copy data frame
df3= df1.copy()

# Removing outliers using the IQR method
def remove_outliers_using_iqr_method(field):
    fig, axes = plt.subplots(1,2)
    plt.tight_layout()
    print(f'Before removing outliers the shape of {field}:{df1.shape}')
    sns.boxplot(df1[field], orient='v', ax=axes[0])
    axes[0].title.set_text('Before')

    # Calculating the IQR
    q1 = df2[field].quantile(0.25)
    q3 = df2[field].quantile(0.75)
    print(f'Q1: {q1}, Q3: {q3}')
    iqr = q3 - q1
    print(f'IQR: {iqr}')
    lower_limit = q1 - 1.5 * iqr
    upper_limit = q3 + 1.5 * iqr
    print(f'Lower limit: {lower_limit}, Upper limit: {upper_limit}')
    df3[field] = np.where(df3[field] > upper_limit, upper_limit, df3[field])
    df3[field] = np.where(df3[field] < lower_limit, lower_limit, df3[field])
    print(f'After removing outliers the shape of {field}:{df3.shape}')
    sns.boxplot(df3[field], orient='v', ax=axes[1])
    axes[1].title.set_text('After')

    # Show plots
    plt.show()

for field in numerical_fields:
    remove_outliers_using_iqr_method(field)
```

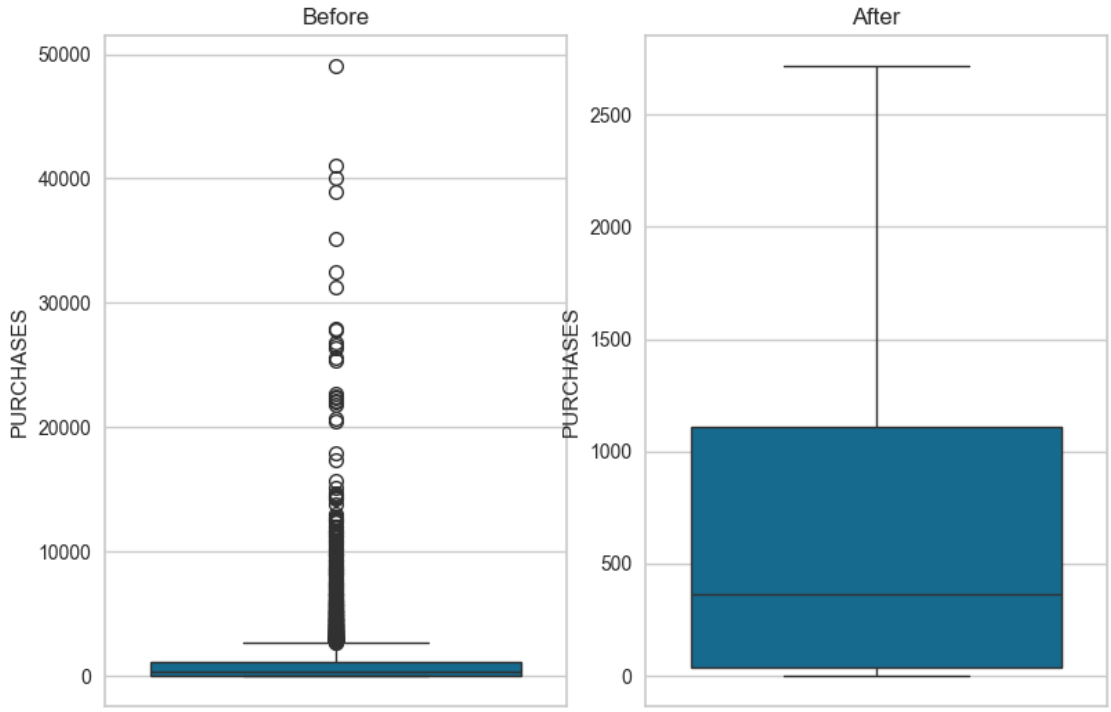
```
Before removing outliers the shape of BALANCE:(8950, 17)
Q1: 128.2819155, Q3: 2054.1400355
IQR: 1925.85812
Lower limit: -2760.5052645, Upper limit: 4942.9272155
After removing outliers the shape of BALANCE:(8950, 17)
```



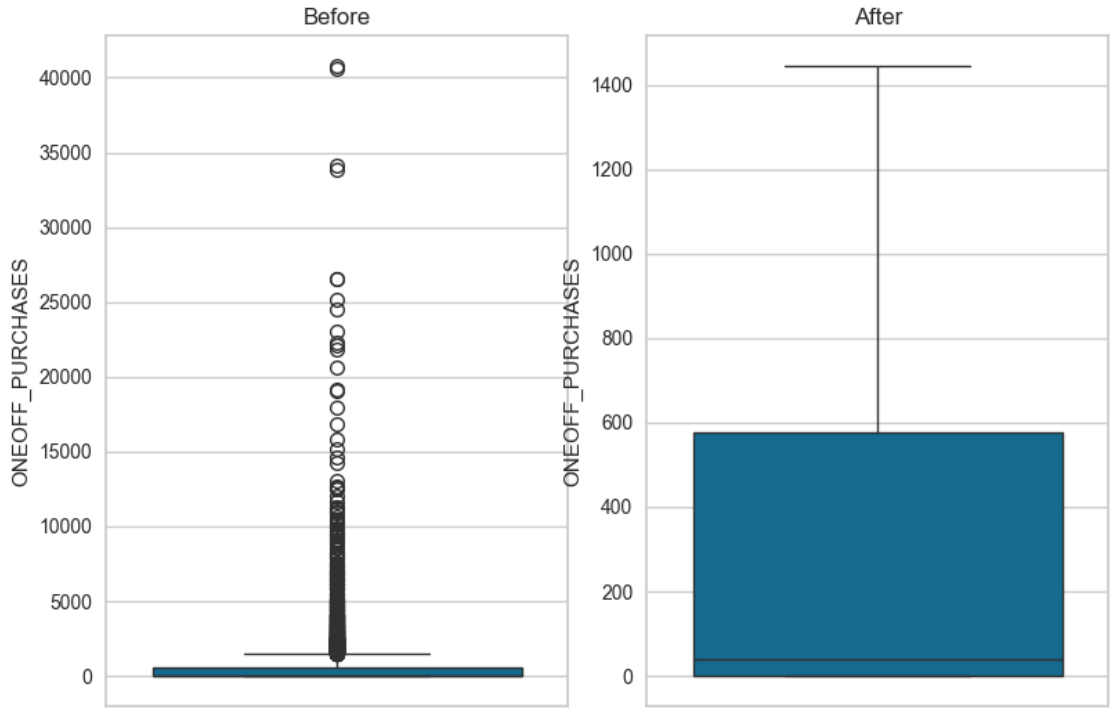

```
Before removing outliers the shape of BALANCE_FREQUENCY:(8950, 17)
Q1: 0.888889, Q3: 1.0
IQR: 0.11111099999999996
Lower limit: 0.7222225000000001, Upper limit: 1.1666664999999998
After removing outliers the shape of BALANCE_FREQUENCY:(8950, 17)
```



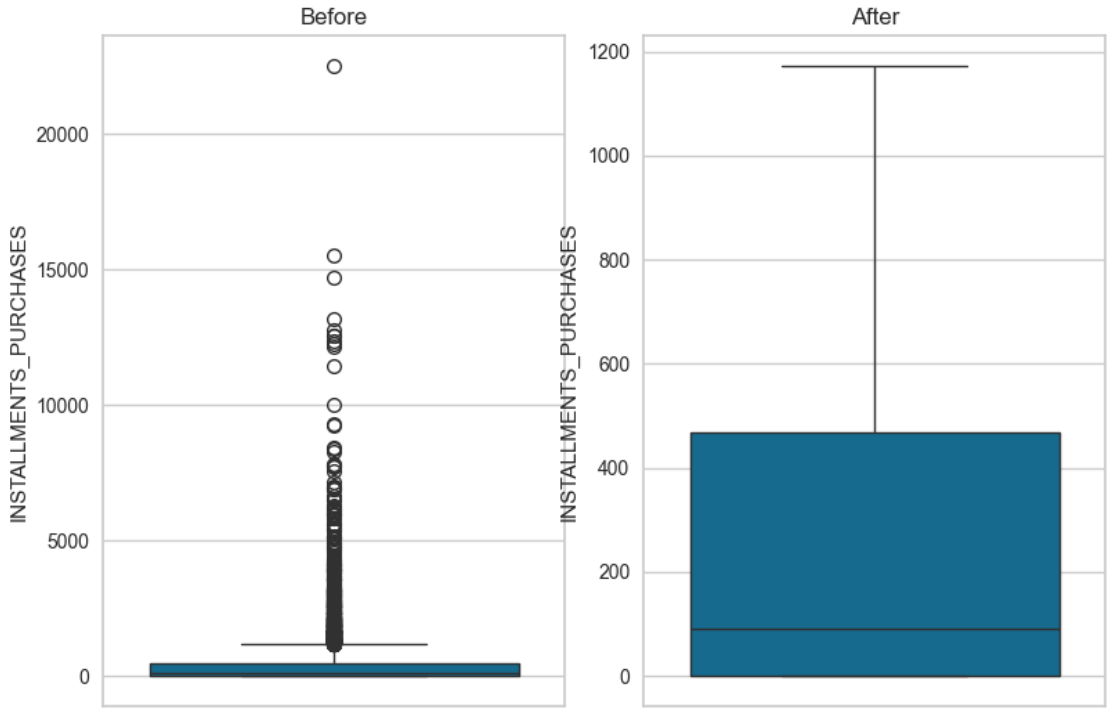

```
Before removing outliers the shape of PURCHASES:(8950, 17)
Q1: 39.635, Q3: 1110.13
IQR: 1070.4950000000001
Lower limit: -1566.1075000000003, Upper limit: 2715.8725000000004
After removing outliers the shape of PURCHASES:(8950, 17)
```



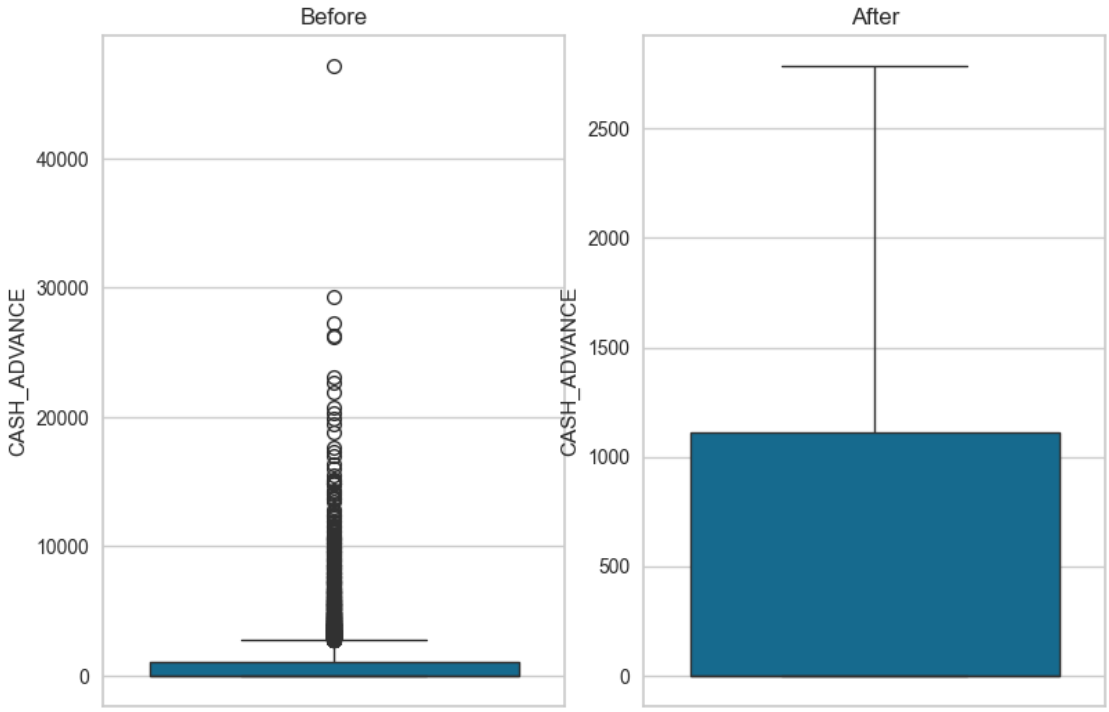

```
Before removing outliers the shape of ONEOFF_PURCHASES:(8950, 17)
Q1: 0.0, Q3: 577.405
IQR: 577.405
Lower limit: -866.1075, Upper limit: 1443.5124999999998
After removing outliers the shape of ONEOFF_PURCHASES:(8950, 17)
```



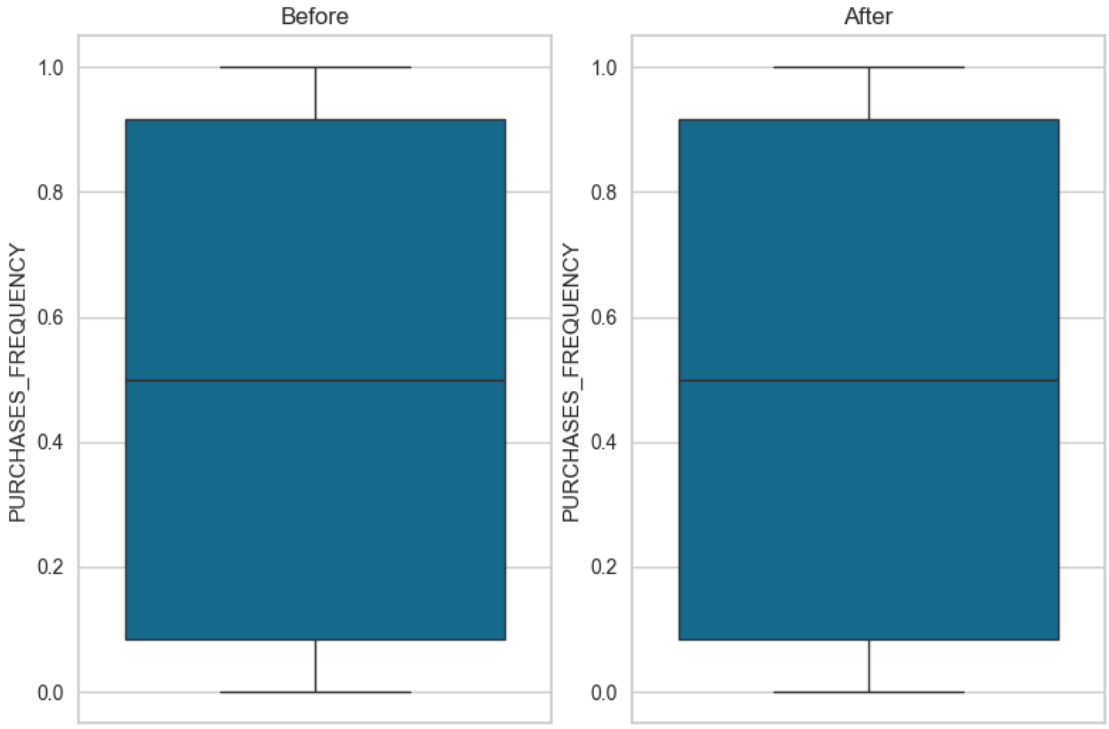

```
Before removing outliers the shape of INSTALLMENTS_PURCHASES:(8950, 17)
Q1: 0.0, Q3: 468.6375
IQR: 468.6375
Lower limit: -702.95625, Upper limit: 1171.59375
After removing outliers the shape of INSTALLMENTS_PURCHASES:(8950, 17)
```



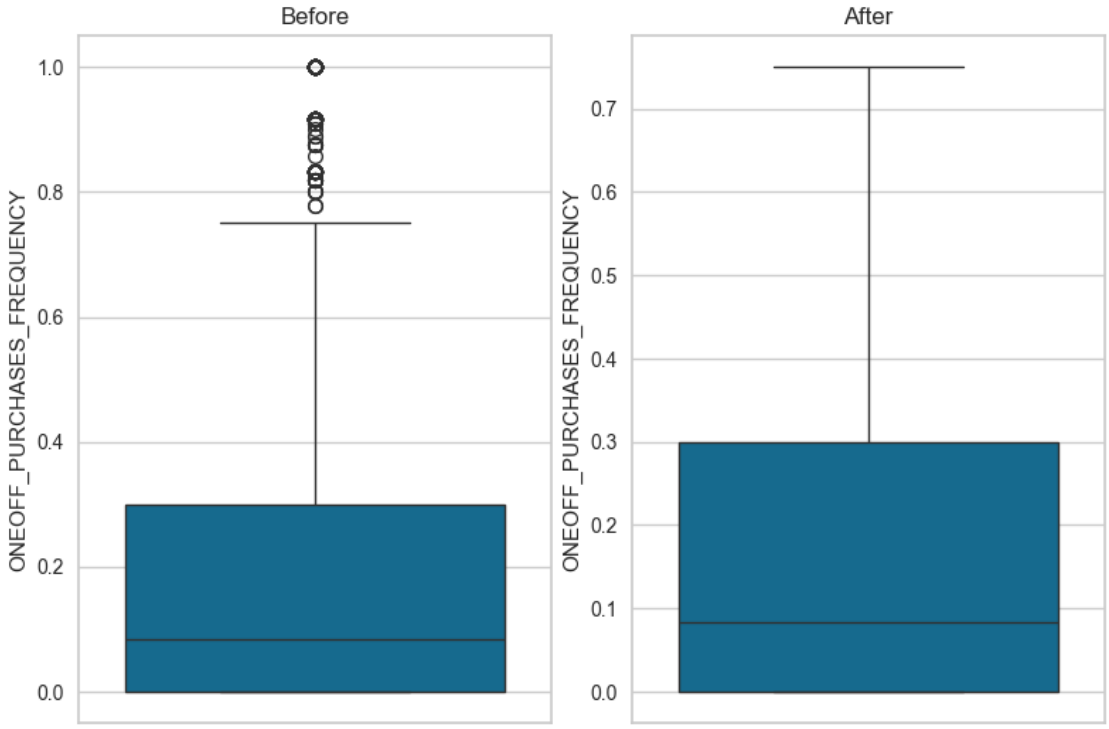

```
Before removing outliers the shape of CASH_ADVANCE:(8950, 17)
Q1: 0.0, Q3: 1113.8211392500002
IQR: 1113.8211392500002
Lower limit: -1670.7317088750003, Upper limit: 2784.5528481250003
After removing outliers the shape of CASH_ADVANCE:(8950, 17)
```



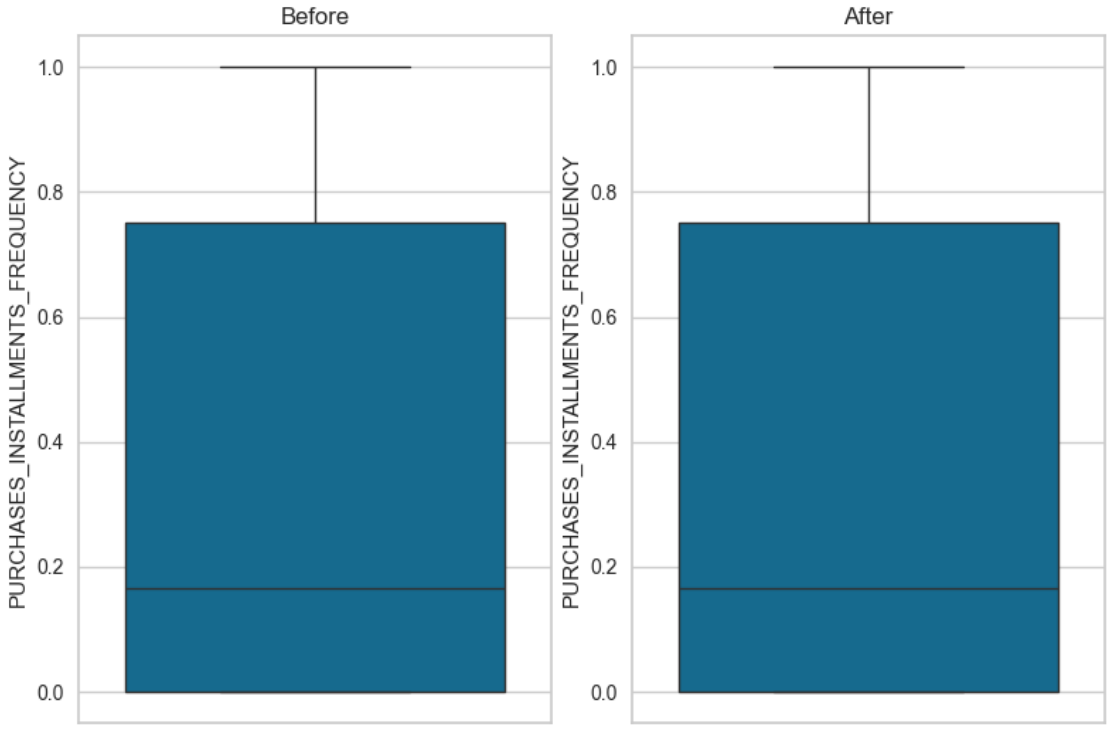

```
Before removing outliers the shape of PURCHASES_FREQUENCY:(8950, 17)
Q1: 0.083333, Q3: 0.916667
IQR: 0.833334
Lower limit: -1.166668, Upper limit: 2.166668
After removing outliers the shape of PURCHASES_FREQUENCY:(8950, 17)
```



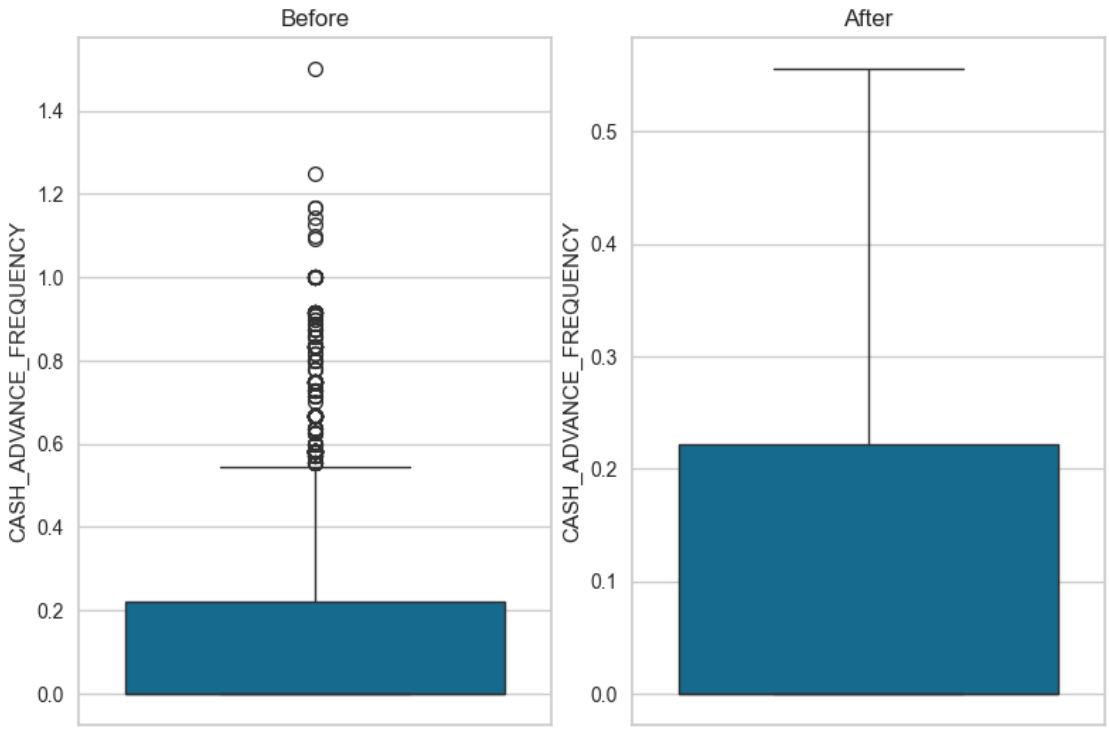

```
Before removing outliers the shape of ONEOFF_PURCHASES_FREQUENCY:(8950, 17)
Q1: 0.0, Q3: 0.3
IQR: 0.3
Lower limit: -0.44999999999999996, Upper limit: 0.75
After removing outliers the shape of ONEOFF_PURCHASES_FREQUENCY:(8950, 17)
```



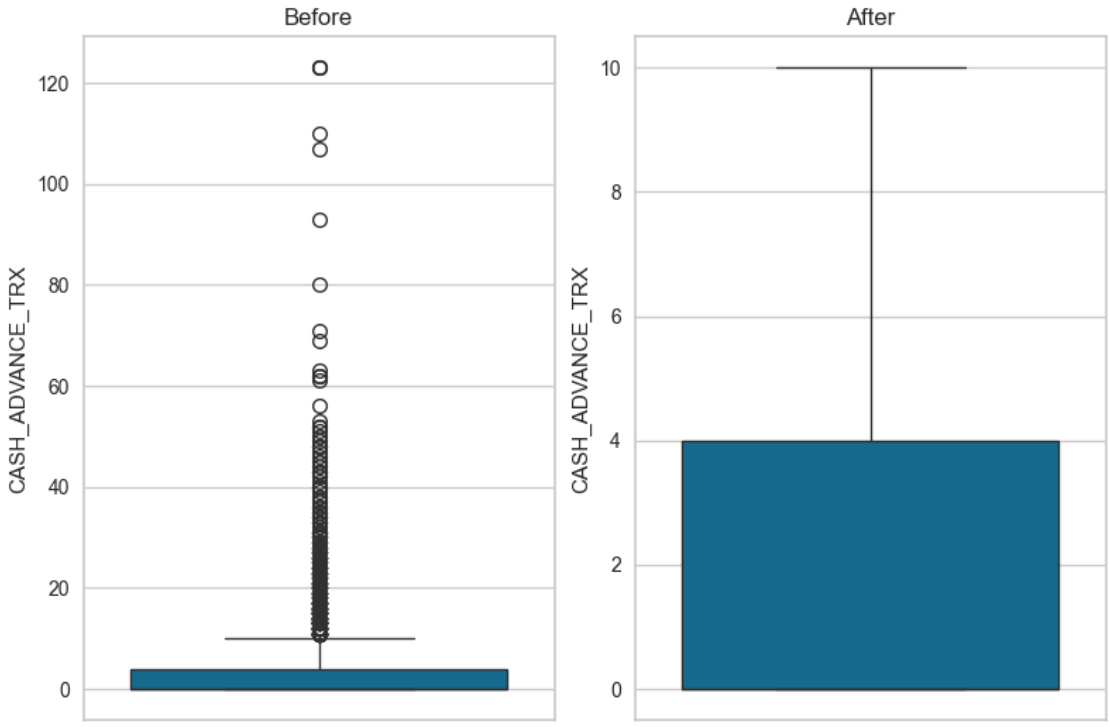

```
Before removing outliers the shape of PURCHASES_INSTALLMENTS_FREQUENCY:(8950,
17)
Q1: 0.0, Q3: 0.75
IQR: 0.75
Lower limit: -1.125, Upper limit: 1.875
After removing outliers the shape of PURCHASES_INSTALLMENTS_FREQUENCY:(8950, 17)
```



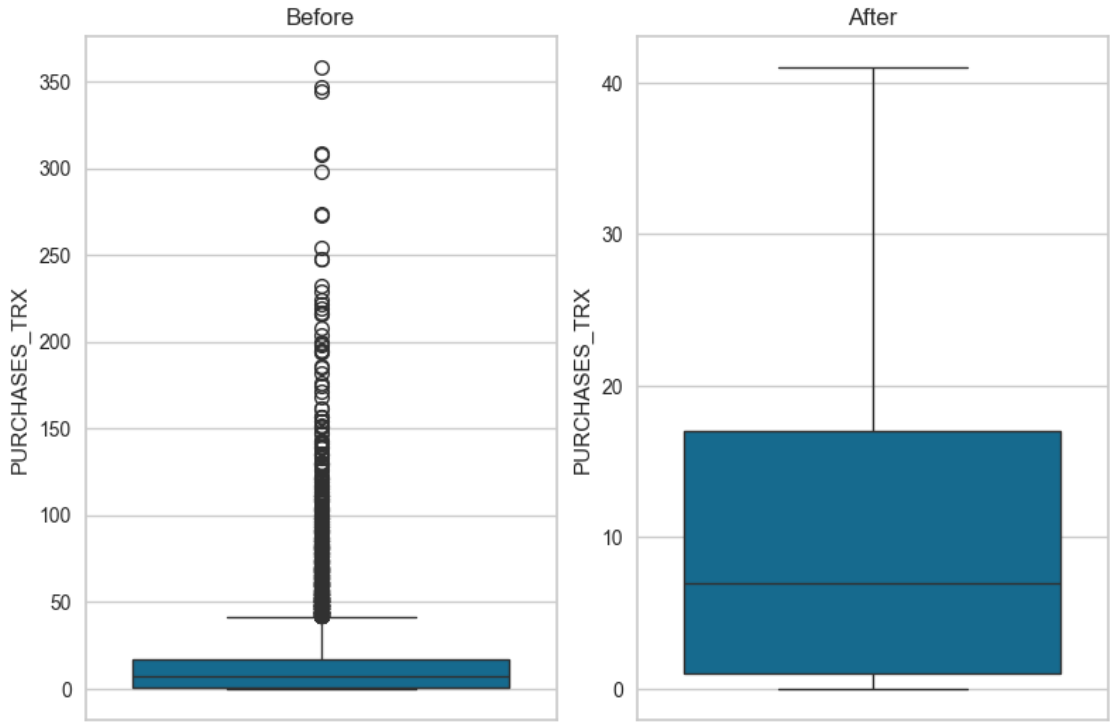

```
Before removing outliers the shape of CASH_ADVANCE_FREQUENCY:(8950, 17)
Q1: 0.0, Q3: 0.222222
IQR: 0.222222
Lower limit: -0.333333, Upper limit: 0.555555
After removing outliers the shape of CASH_ADVANCE_FREQUENCY:(8950, 17)
```



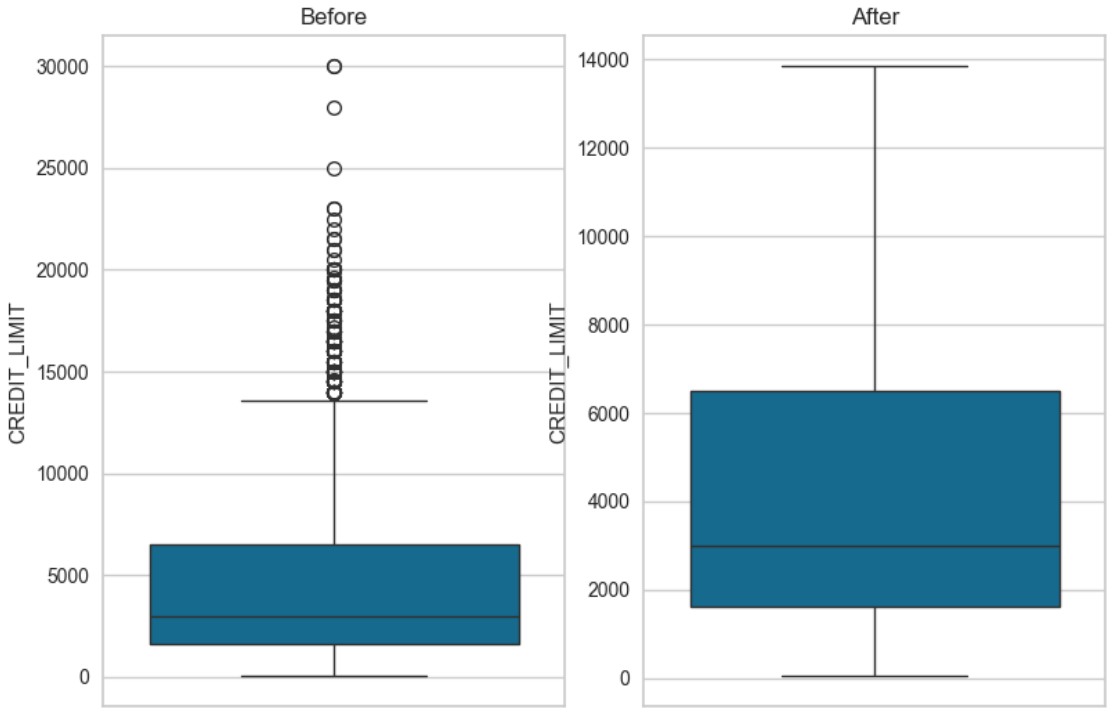

```
Before removing outliers the shape of CASH_ADVANCE_TRX:(8950, 17)
Q1: 0.0, Q3: 4.0
IQR: 4.0
Lower limit: -6.0, Upper limit: 10.0
After removing outliers the shape of CASH_ADVANCE_TRX:(8950, 17)
```



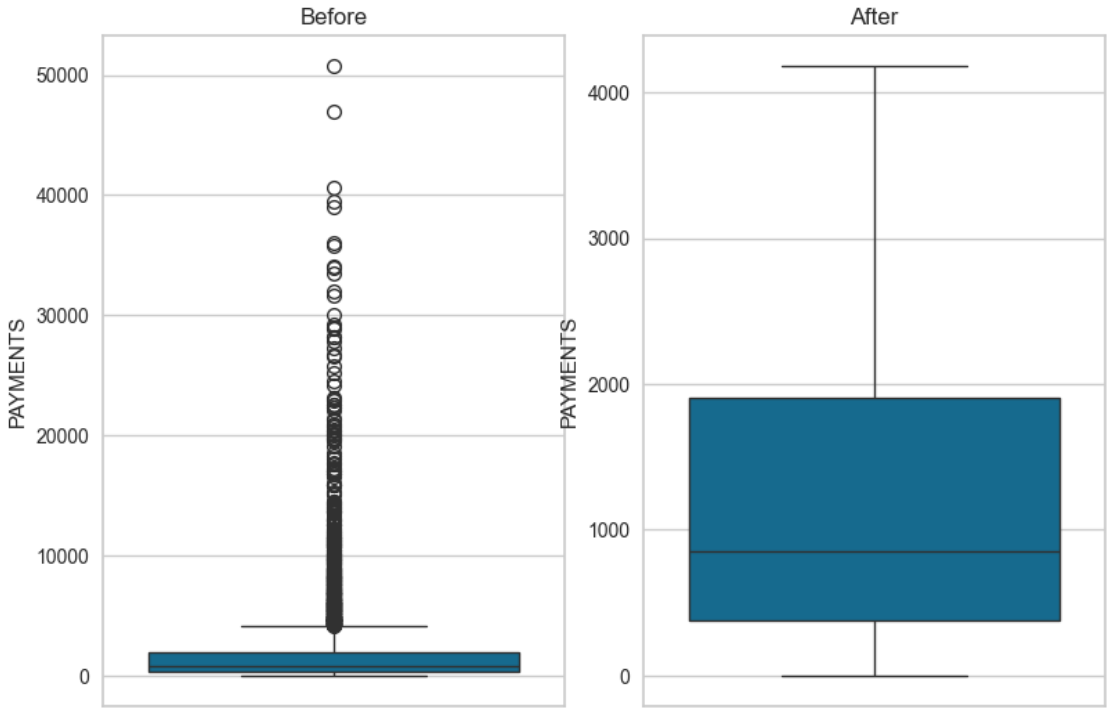

```
Before removing outliers the shape of PURCHASES_TRX:(8950, 17)
Q1: 1.0, Q3: 17.0
IQR: 16.0
Lower limit: -23.0, Upper limit: 41.0
After removing outliers the shape of PURCHASES_TRX:(8950, 17)
```



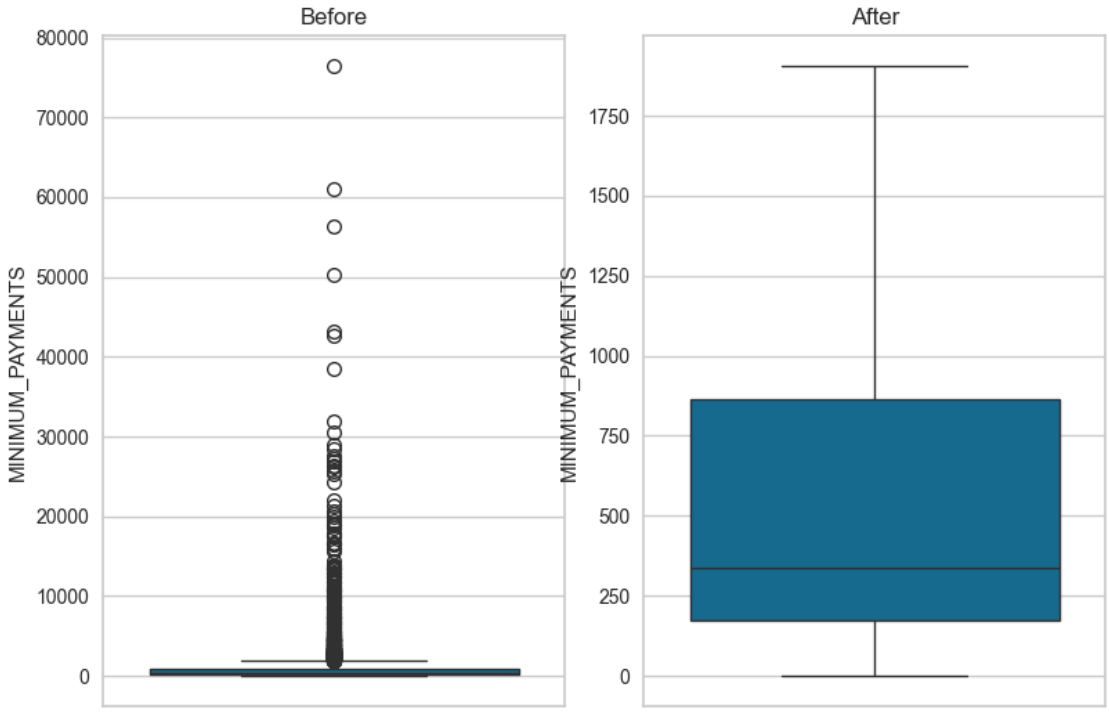

```
Before removing outliers the shape of CREDIT_LIMIT:(8950, 17)
Q1: 1600.0, Q3: 6500.0
IQR: 4900.0
Lower limit: -5750.0, Upper limit: 13850.0
After removing outliers the shape of CREDIT_LIMIT:(8950, 17)
```



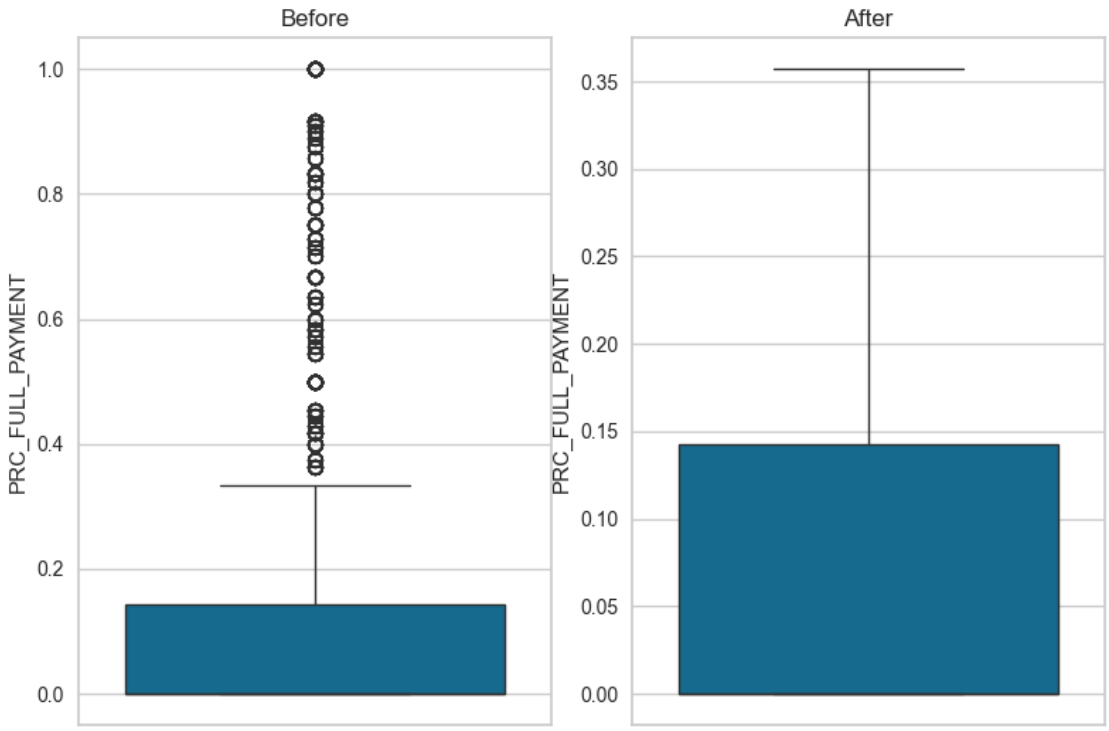

```
Before removing outliers the shape of PAYMENTS:(8950, 17)
Q1: 383.276166, Q3: 1901.1343167500002
IQR: 1517.8581507500003
Lower limit: -1893.5110601250003, Upper limit: 4177.921542875
After removing outliers the shape of PAYMENTS:(8950, 17)
```




```
Before removing outliers the shape of MINIMUM_PAYMENTS:(8950, 17)
Q1: 170.85765425, Q3: 864.2065423050828
IQR: 693.3488880550829
Lower limit: -869.1656778326242, Upper limit: 1904.229874387707
After removing outliers the shape of MINIMUM_PAYMENTS:(8950, 17)
```




```
Before removing outliers the shape of PRC_FULL_PAYMENT:(8950, 17)
Q1: 0.0, Q3: 0.142857
IQR: 0.142857
Lower limit: -0.21428550000000002, Upper limit: 0.35714250000000003
After removing outliers the shape of PRC_FULL_PAYMENT:(8950, 17)
```



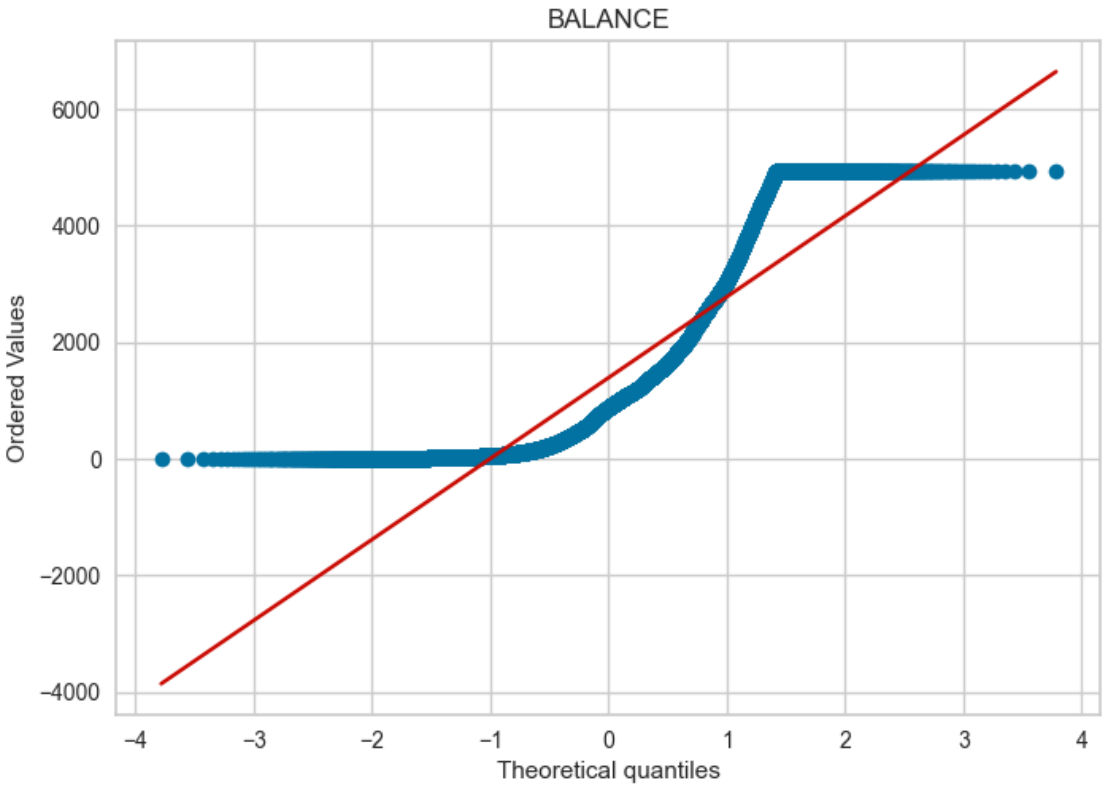
From the results, we can see that `IQR` is better since it does not reduce the dataset drastically.

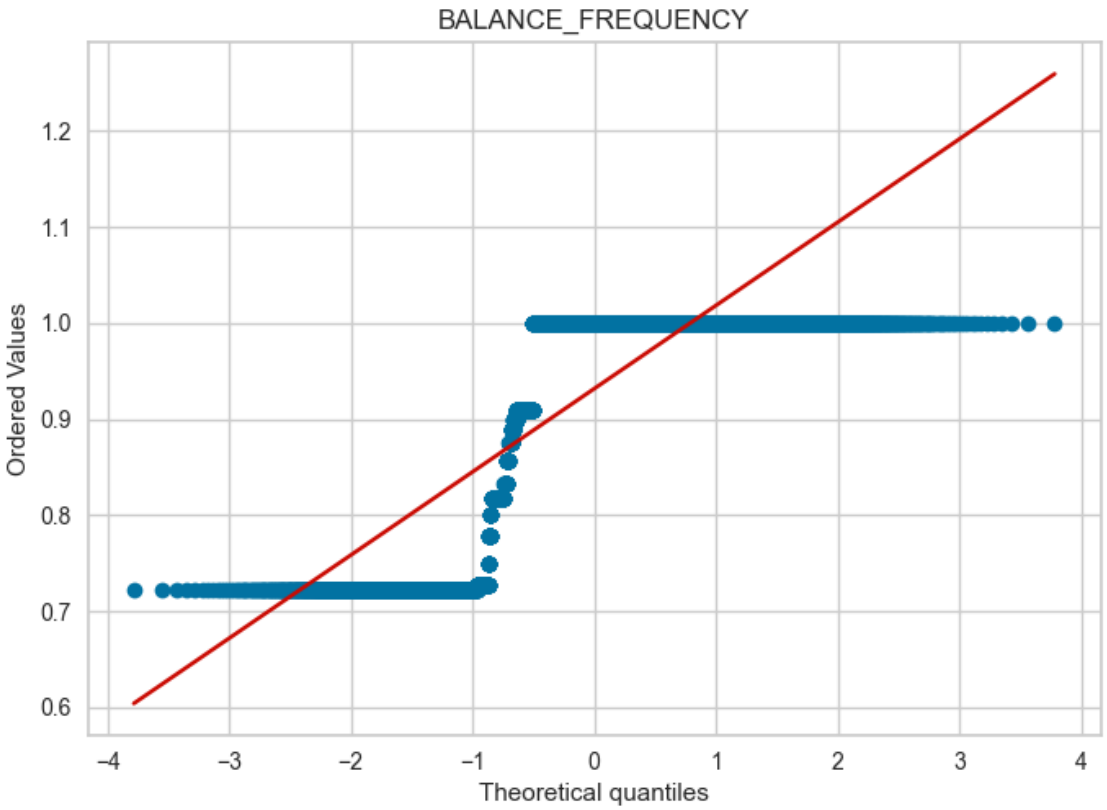
1.2.2 Producing Q-Q Plots, Histograms, and Applying Necessary Transformations

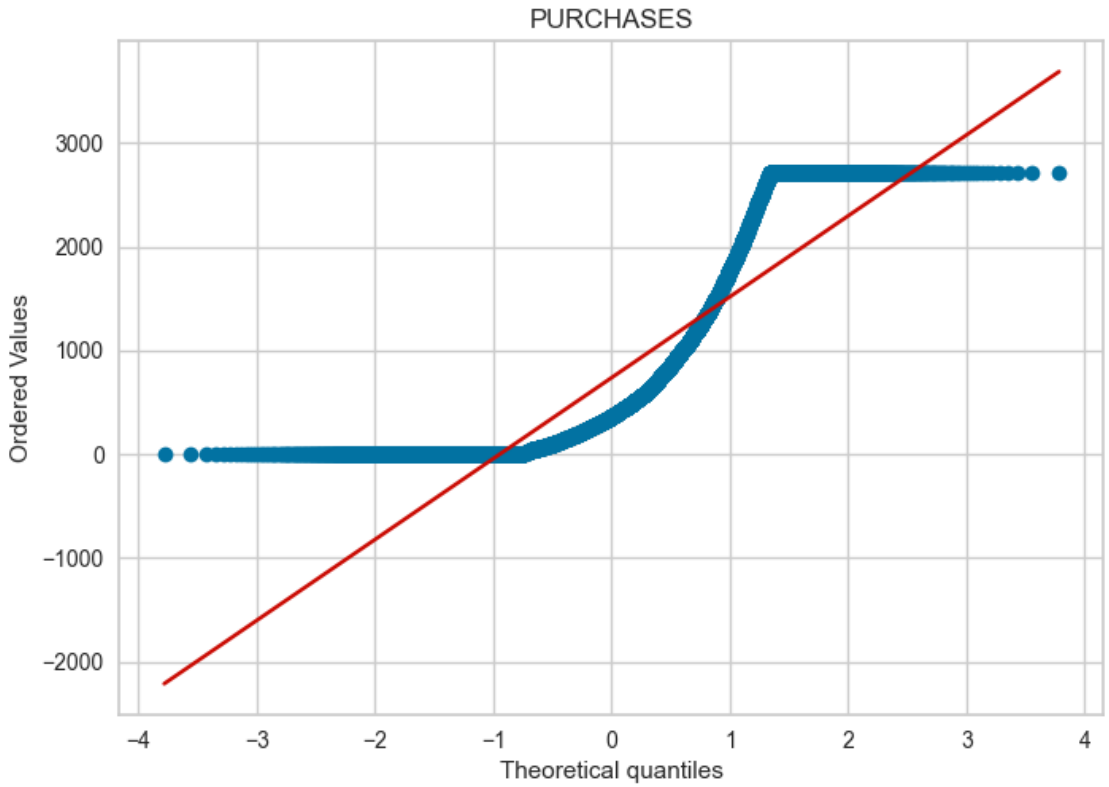
First, we need to produce Q-Q plots and histograms to check whether the data distribution in the dataset is normal or not.

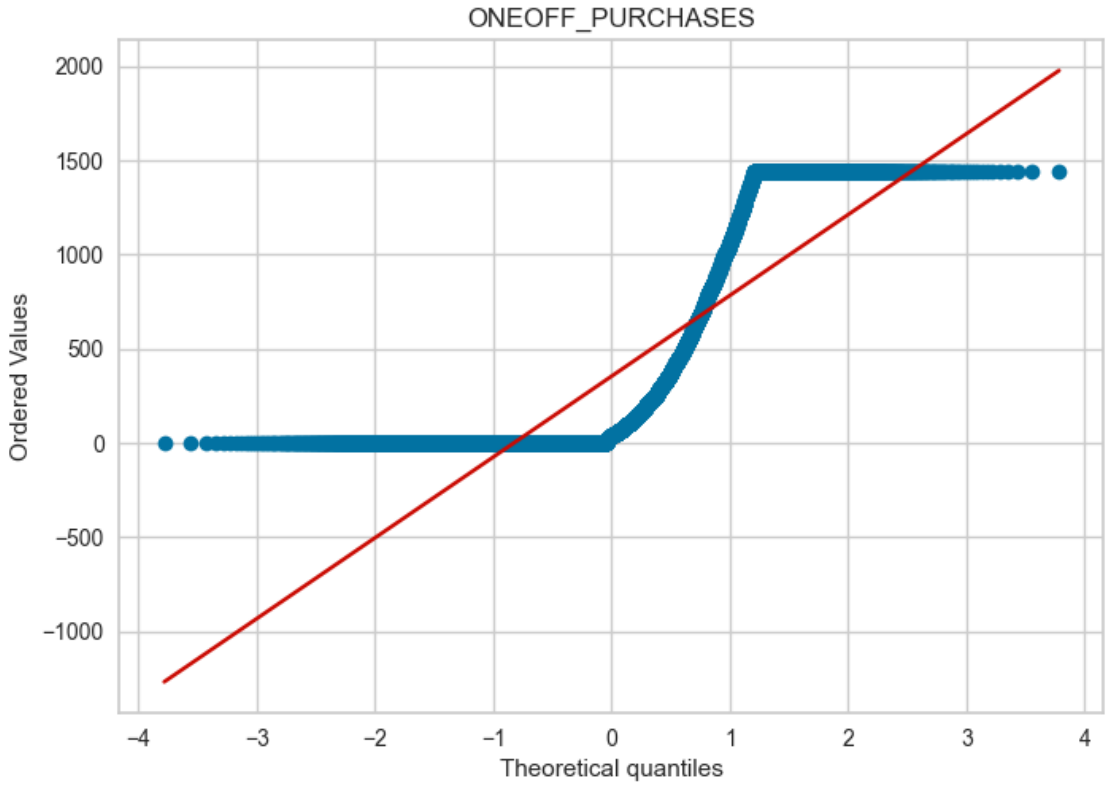
Step 1 - Q-Q Plots Q-Q(Quantile-Quantile) plots are typically used to assess whether a given dataset follows a particular distribution or not. It usually checks whether a dataset follows a normal(Gaussian) distribution and based on the exploratory data extracted from it, we can transform the data and normalize the dataset for more accurate results. The following code snippet is used to make Q-Q plots for the dataset.

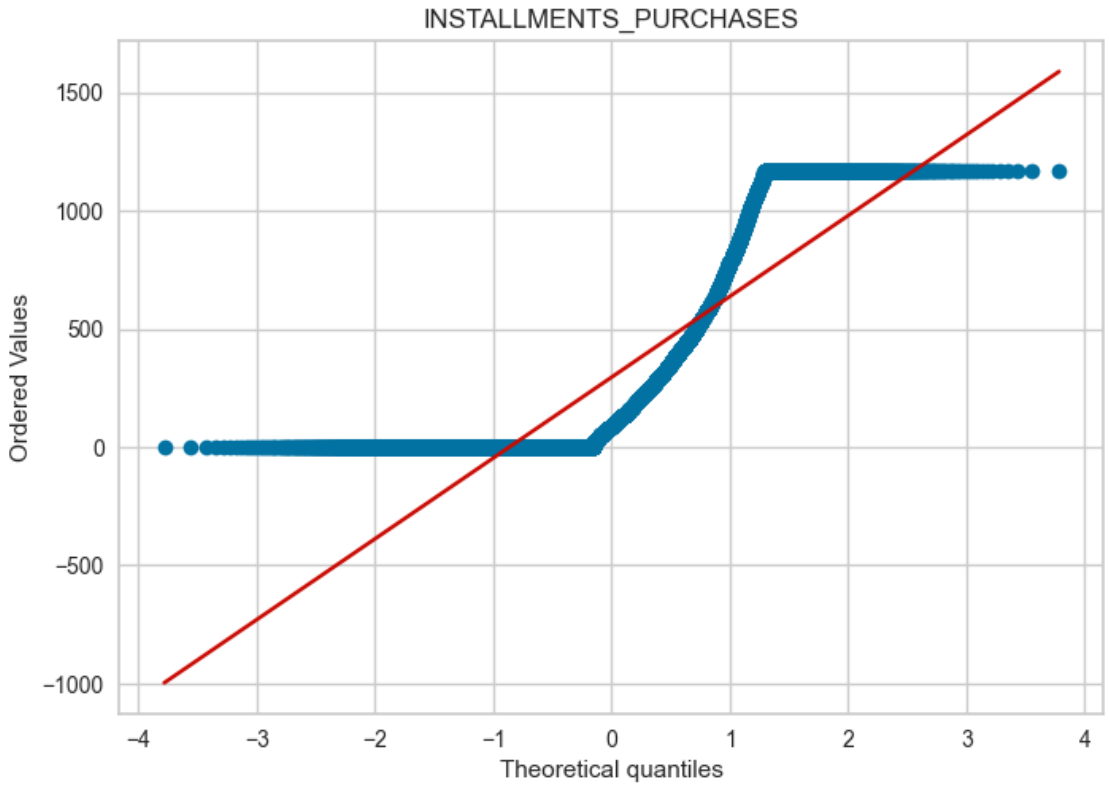
[19]:
```
# We are using the dataset, which doesn't have any outliers
def draw_qq_plots(df, field):
    # create and show the plots
    stats.probplot(df[field], dist="norm", plot=plt)
    plt.title(field)
    plt.show()

# Draw q-q plots
for field in numerical_fields:
    draw_qq_plots(df3, field)
```

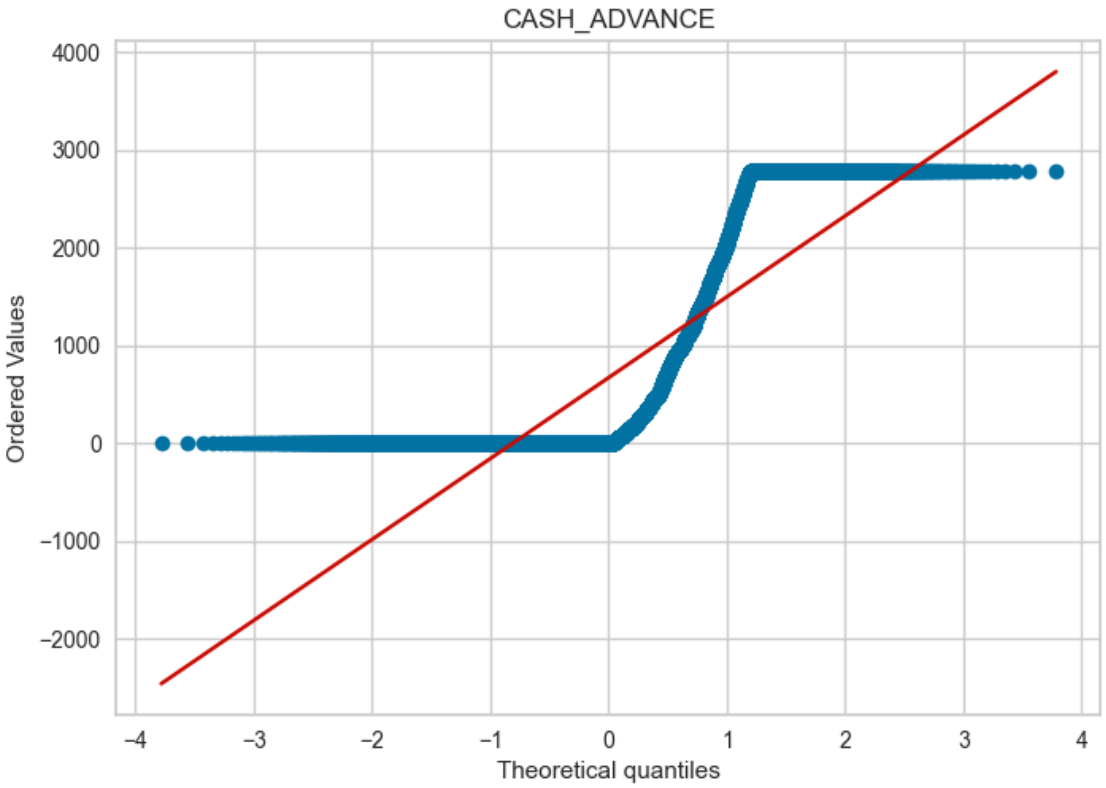

BALANCE
Ordered Values
6000
4000
2000
0
−2000
−4000
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles

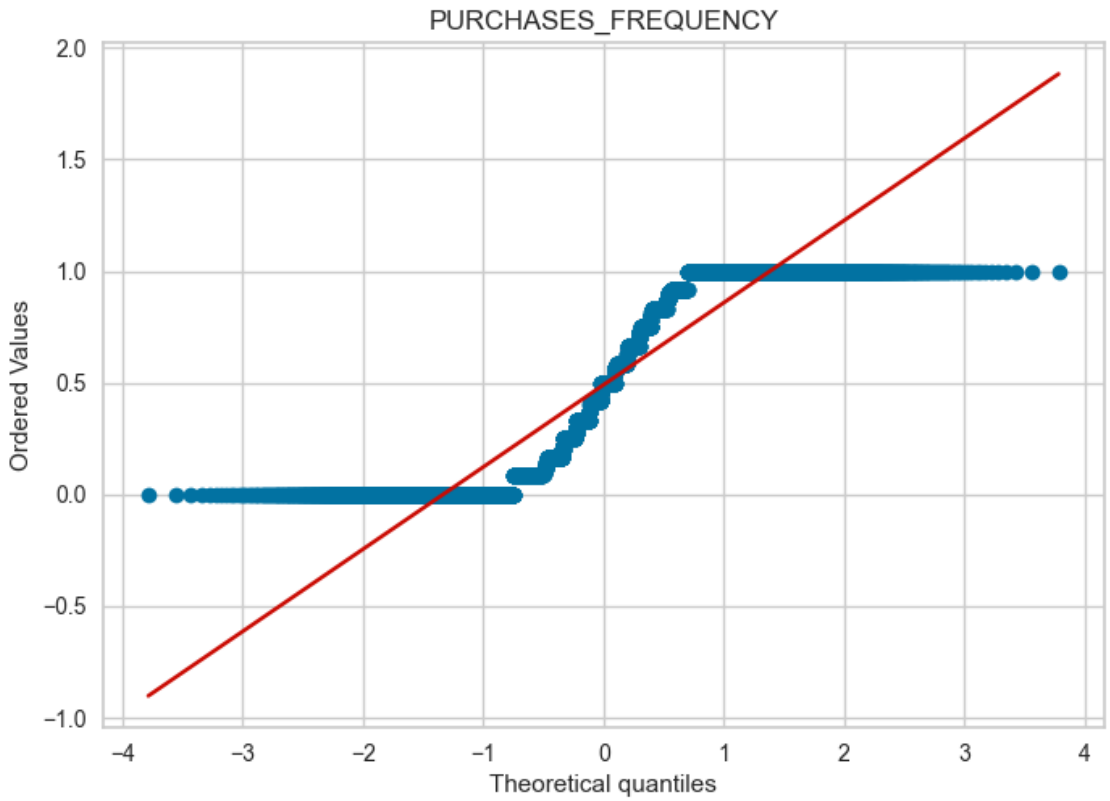

BALANCE_FREQUENCY
Ordered Values
1.2
1.1
1.0
0.9
0.8
0.7
0.6
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles

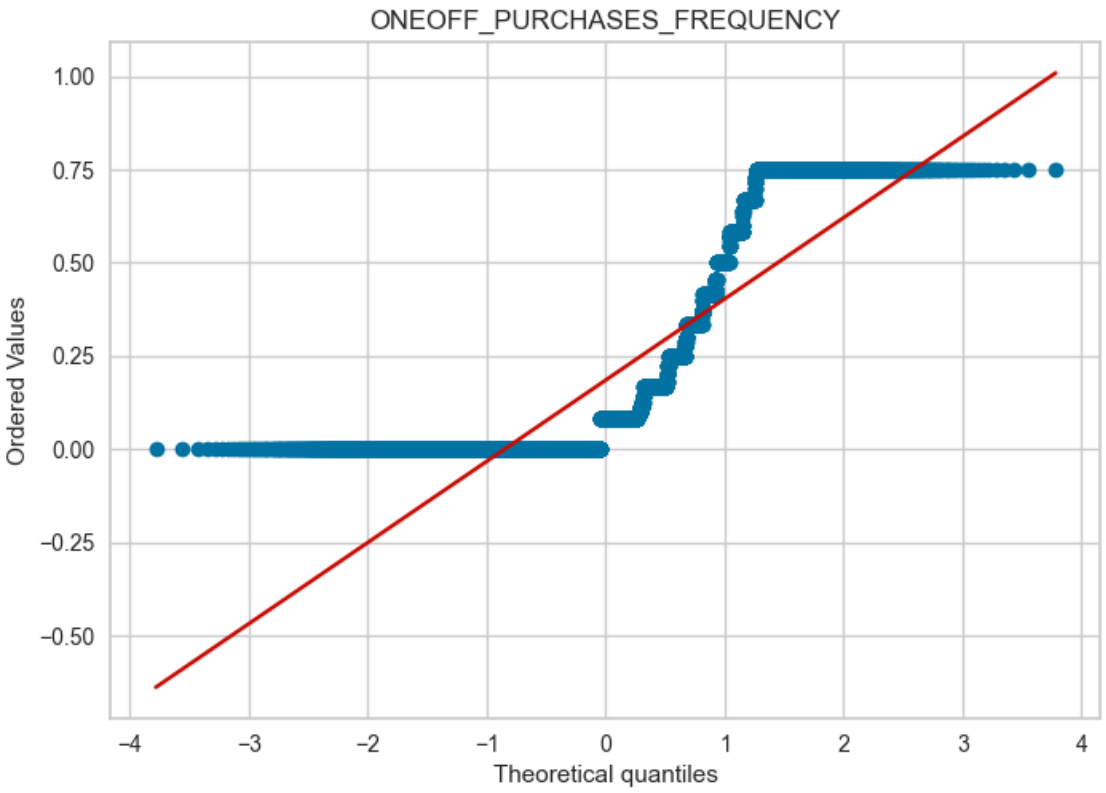

PURCHASES
Ordered Values
3000
2000
1000
0
−1000
−2000
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles

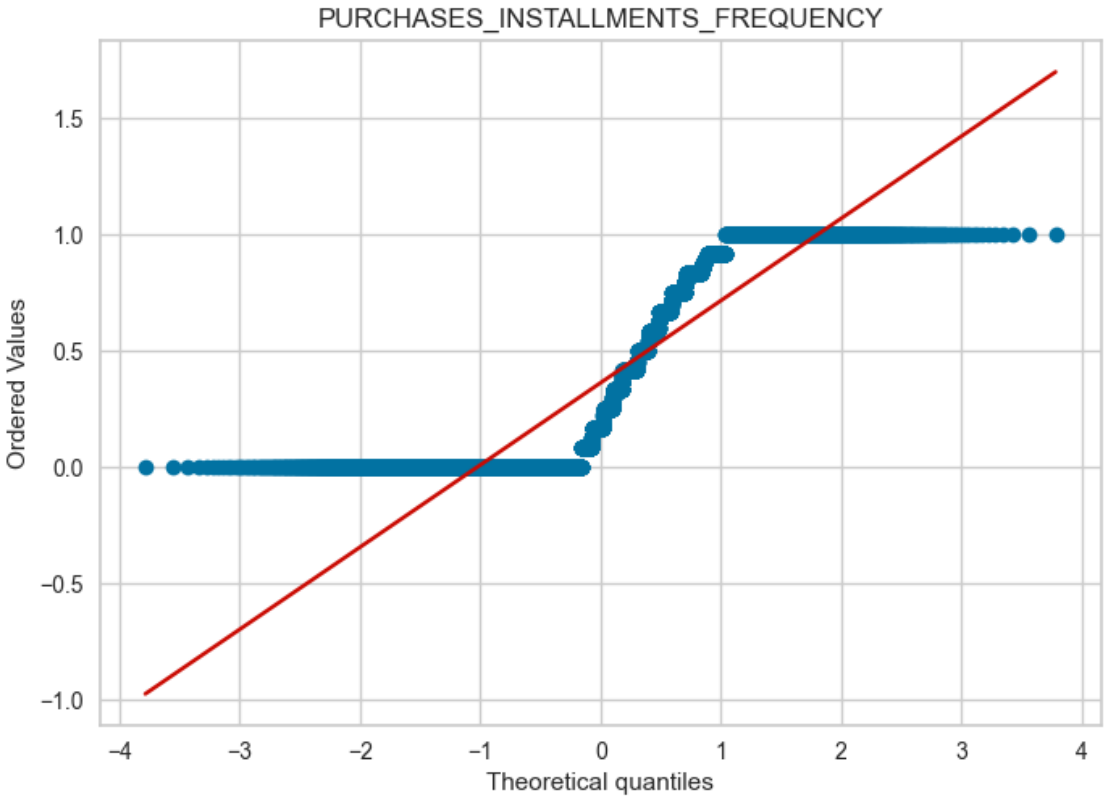

ONEOFF_PURCHASES
Ordered Values
2000
1500
1000
500
0
−500
−1000
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles

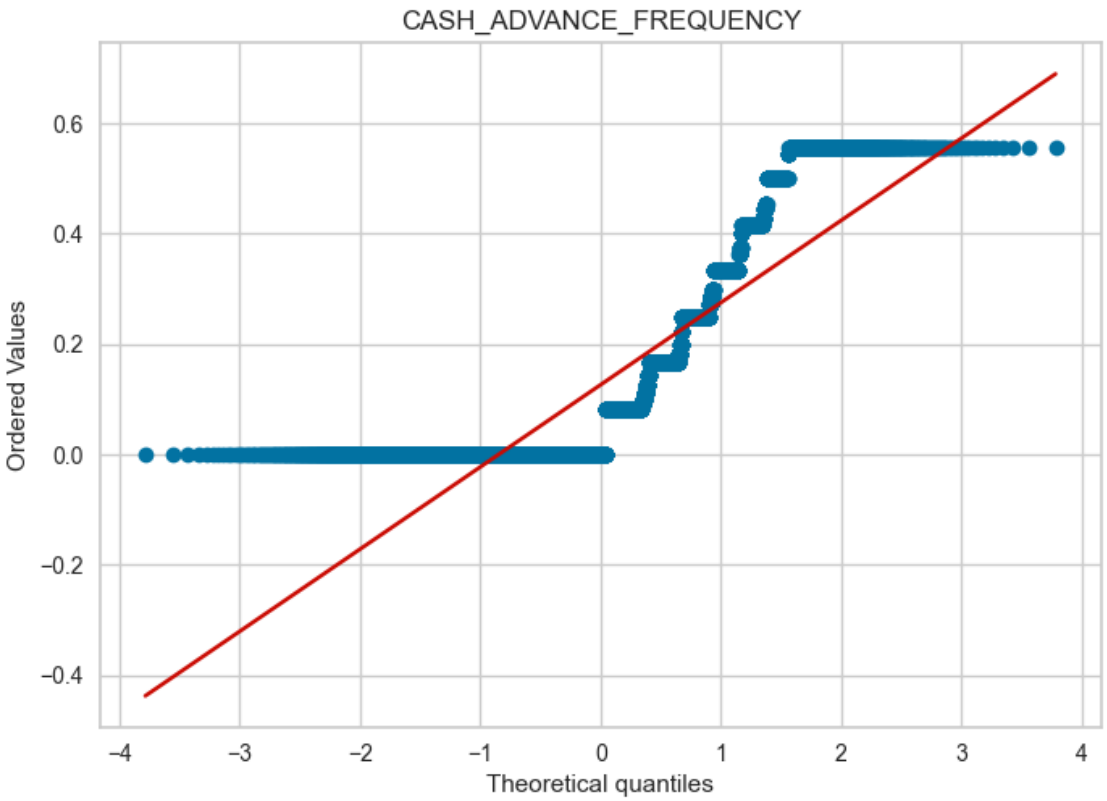

INSTALLMENTS_PURCHASES
1500
1000
500
0
−500
−1000
Ordered Values
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles

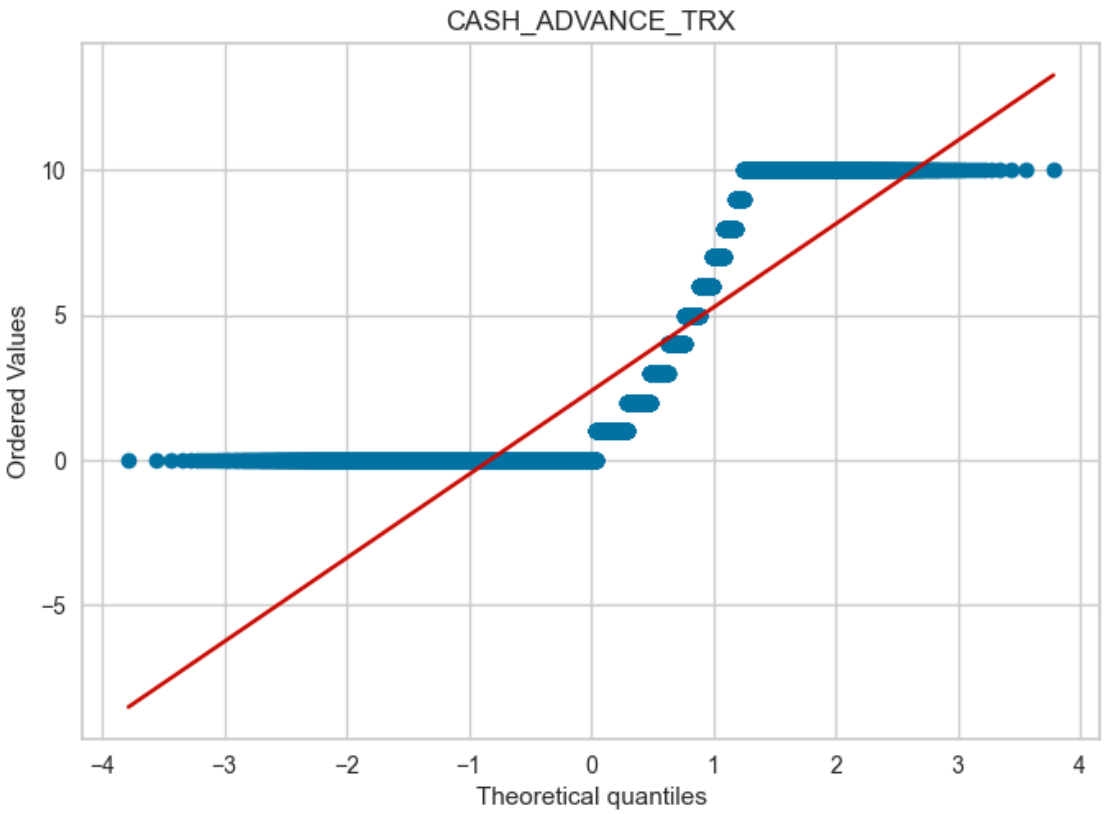

CASH_ADVANCE
Ordered Values
Theoretical quantiles
4000
3000
2000
1000
0
−1000
−2000
−4
−3
−2
−1
0
1
2
3
4

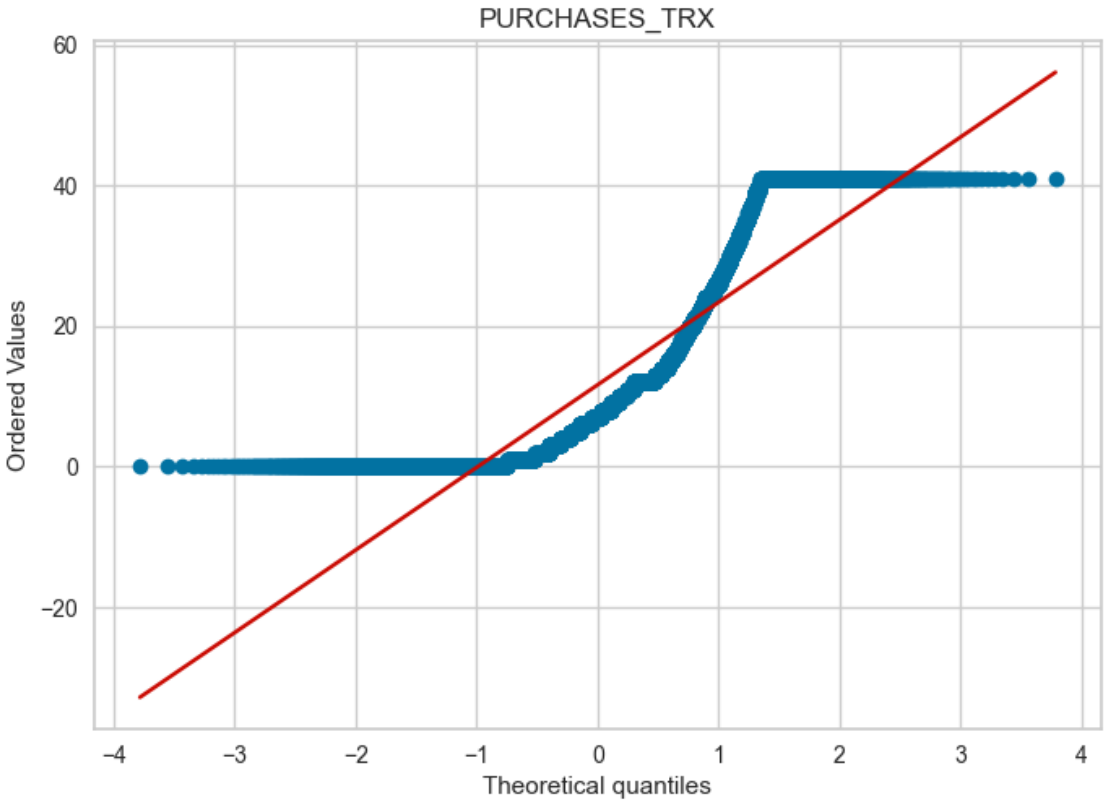

PURCHASES_FREQUENCY
Ordered Values
Theoretical quantiles
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−4
−3
−2
−1
0
1
2
3
4

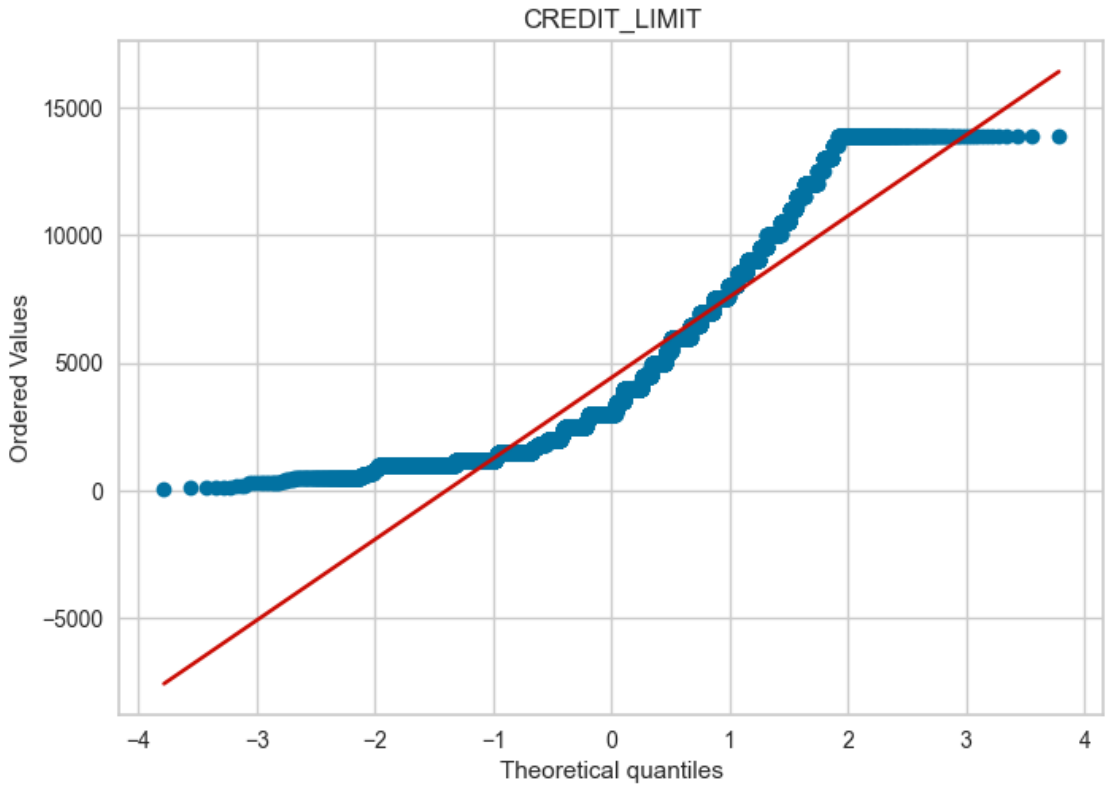

ONEOFF_PURCHASES_FREQUENCY
Ordered Values
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles

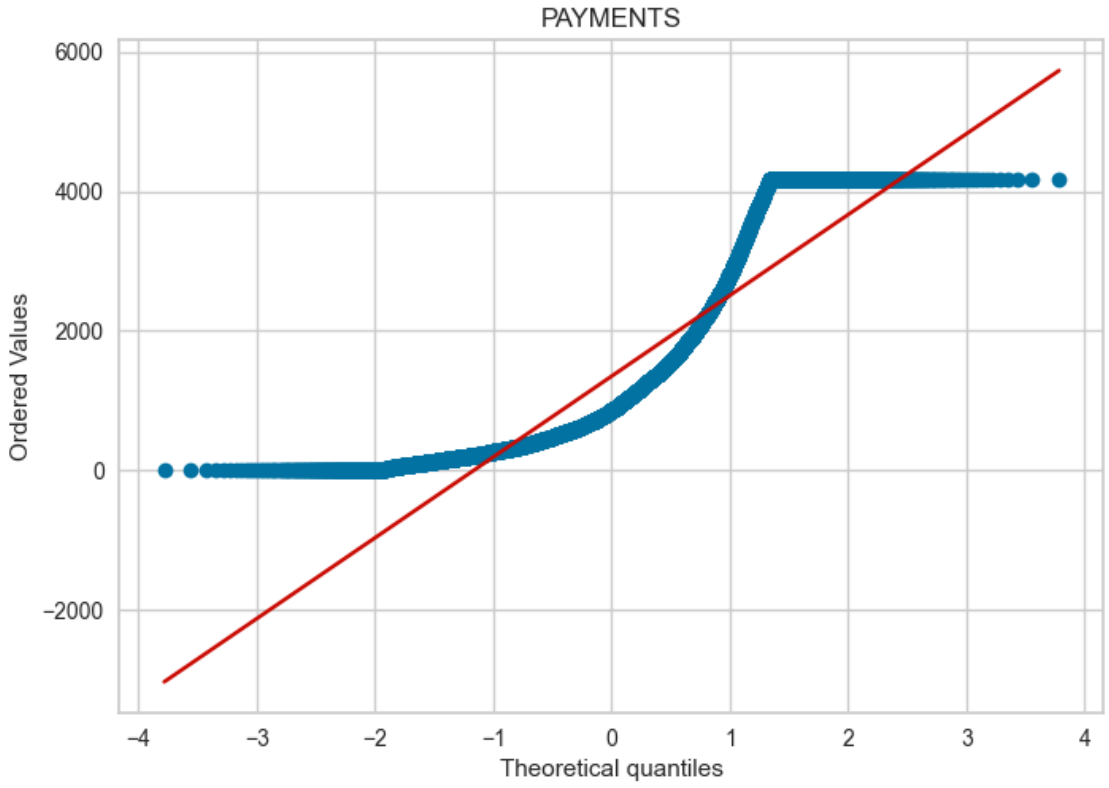

PURCHASES_INSTALLMENTS_FREQUENCY
Ordered Values
Theoretical quantiles
1.5
1.0
0.5
0.0
−0.5
−1.0
−4
−3
−2
−1
0
1
2
3
4

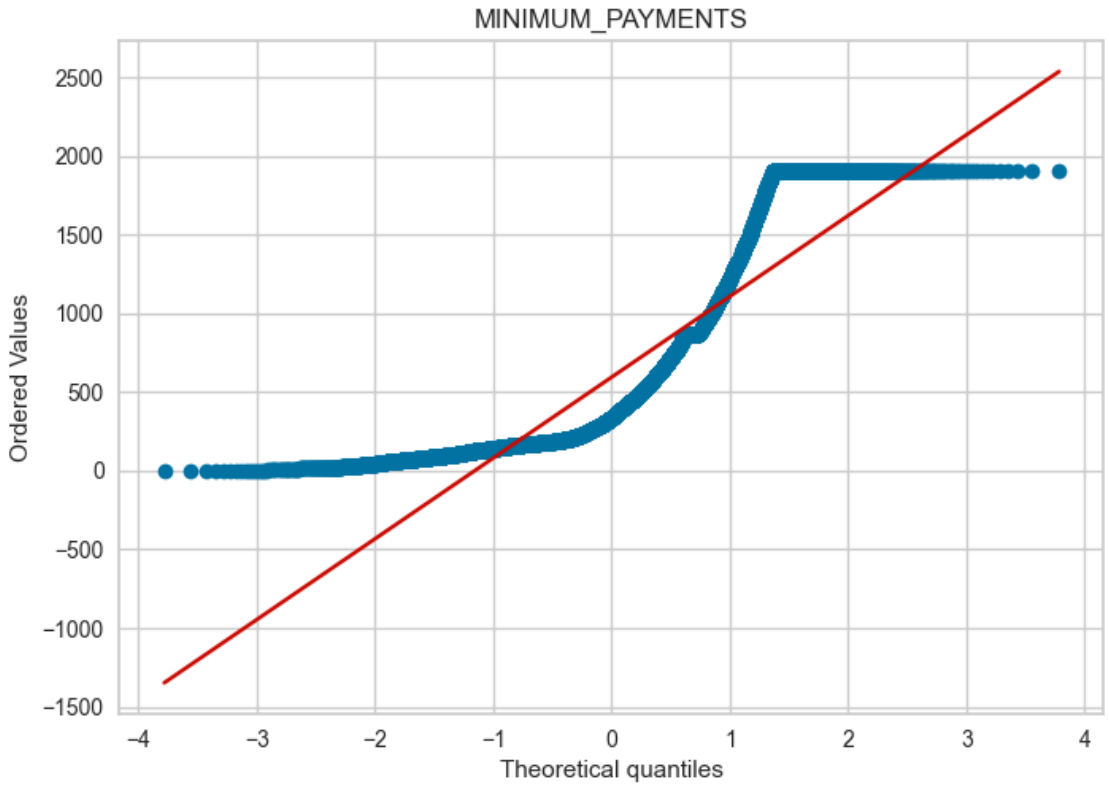

CASH_ADVANCE_FREQUENCY
Ordered Values
Theoretical quantiles
0.6
0.4
0.2
0.0
−0.2
−0.4
−4
−3
−2
−1
0
1
2
3
4

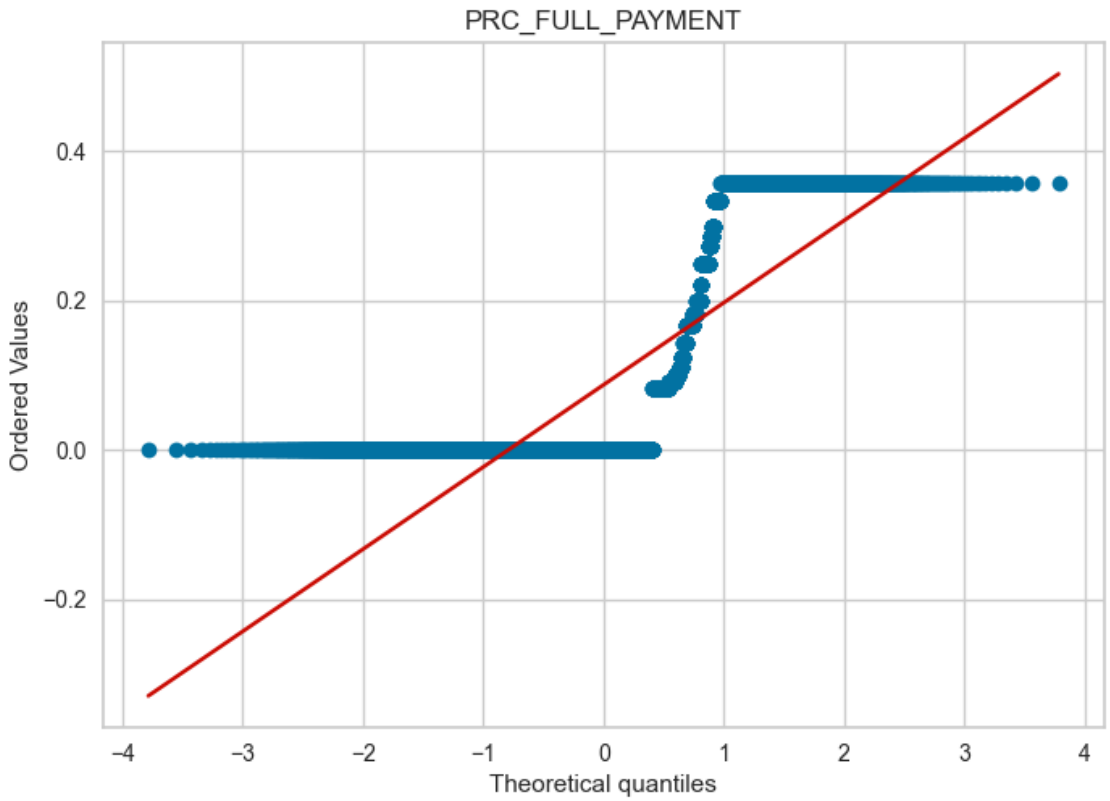

CASH_ADVANCE_TRX
Ordered Values
Theoretical quantiles
10
5
0
−5
−4
−3
−2
−1
0
1
2
3
4


PURCHASES_TRX
Ordered Values
Theoretical quantiles
60
40
20
0
−20
−4
−3
−2
−1
0
1
2
3
4


CREDIT_LIMIT
Ordered Values
15000
10000
5000
0
−5000
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles


PAYMENTS
Ordered Values
6000
4000
2000
0
−2000
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles

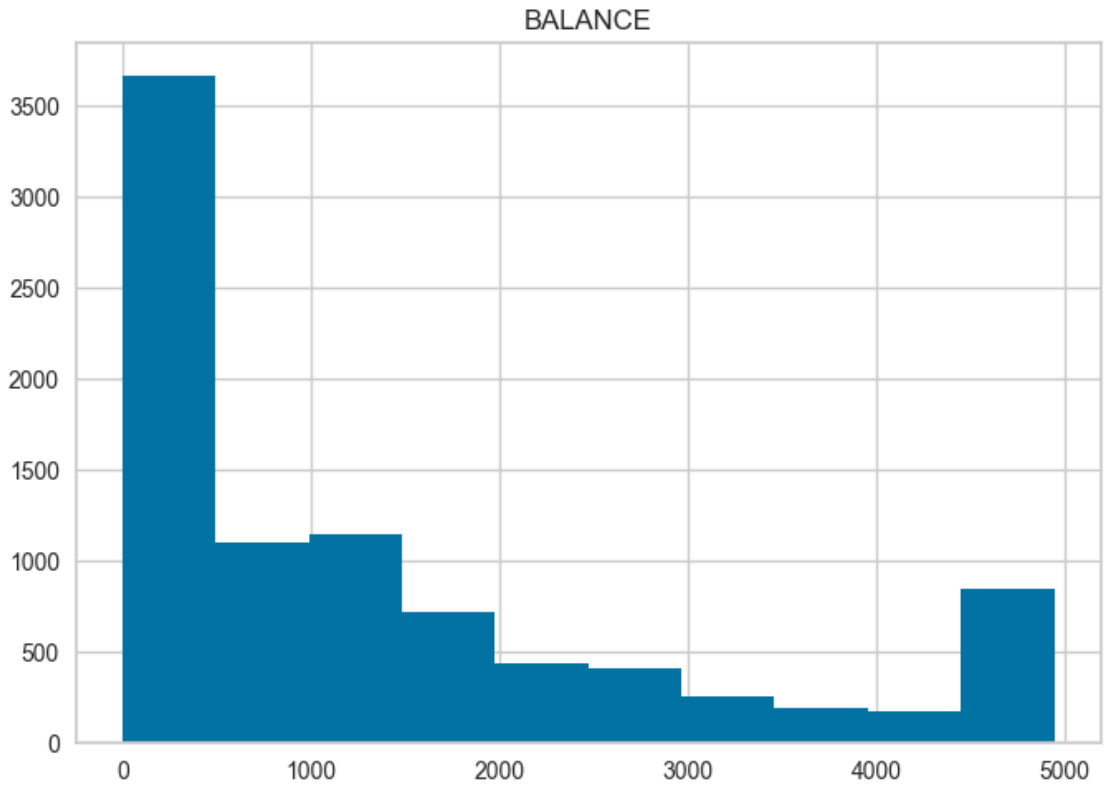

MINIMUM_PAYMENTS
Ordered Values
2500
2000
1500
1000
500
0
−500
−1000
−1500
−4
−3
−2
−1
0
1
2
3
4
Theoretical quantiles



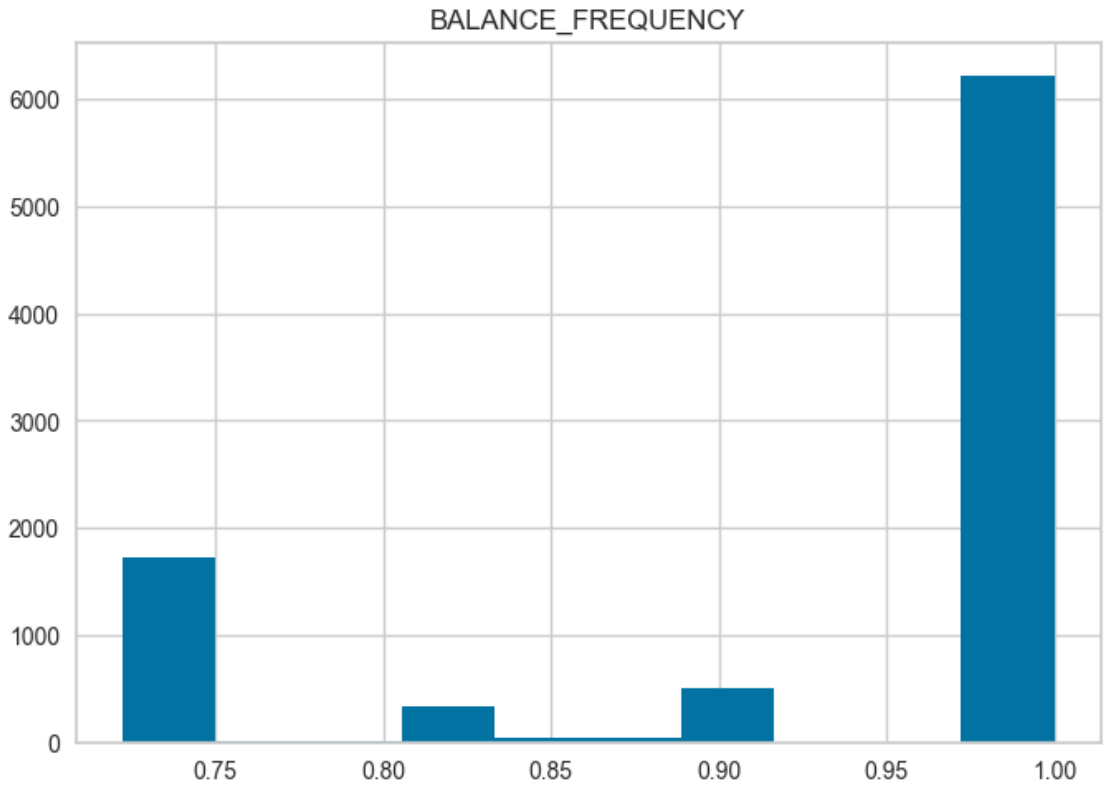
As you can see most of the Q-Q plot data are above the 45-degree line, which indicates that there is normal distribution in the dataset. Now, let's check the histograms to see whether there are any skewed columns in the dataset.

Step 2 - Histograms Histograms allow us to check the data distribution. And using them we can find out whether a dataset is right-skewed or left-skewed. The following code is used to plot the histograms.

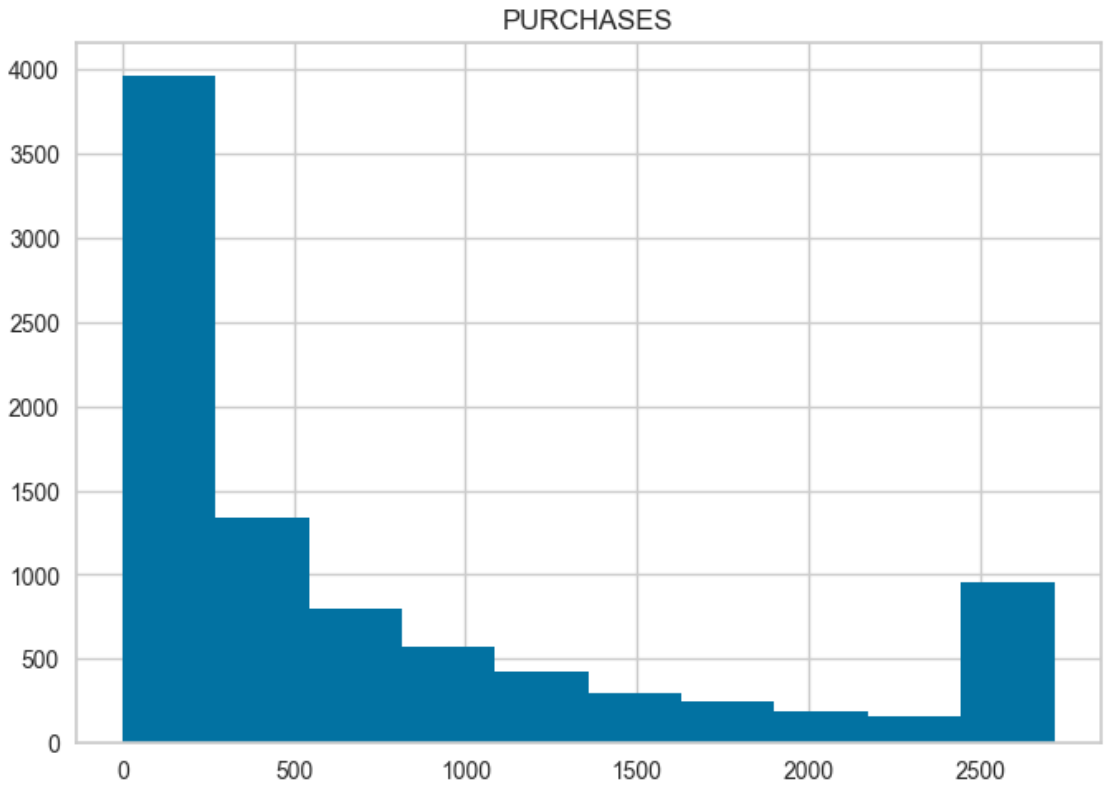
[20]:
```
# Generate histograms for the numerical fields
def draw_histograms(df, field):
    plt.hist(df[field], color='b')
    plt.title(field)
    plt.figure()
    plt.show()

# Draw histograms
for field in numerical_fields:
    draw_histograms(df3, field)
```



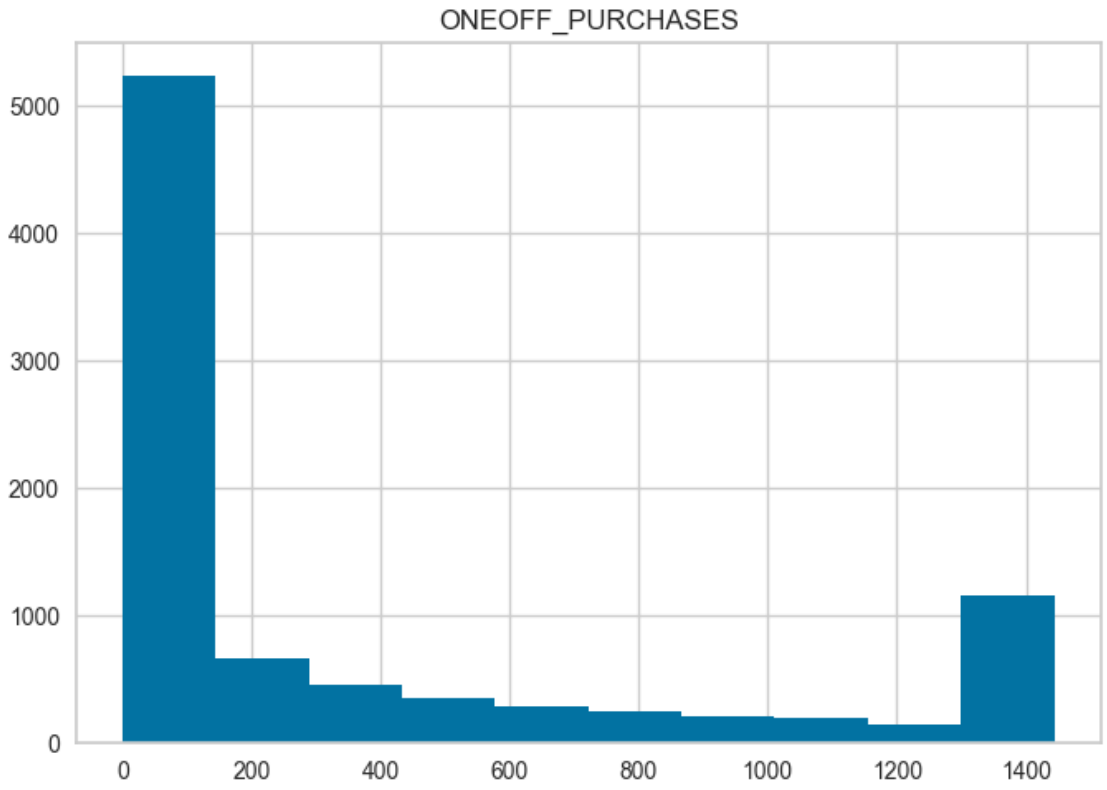

```
<Figure size 800x550 with 0 Axes>
```



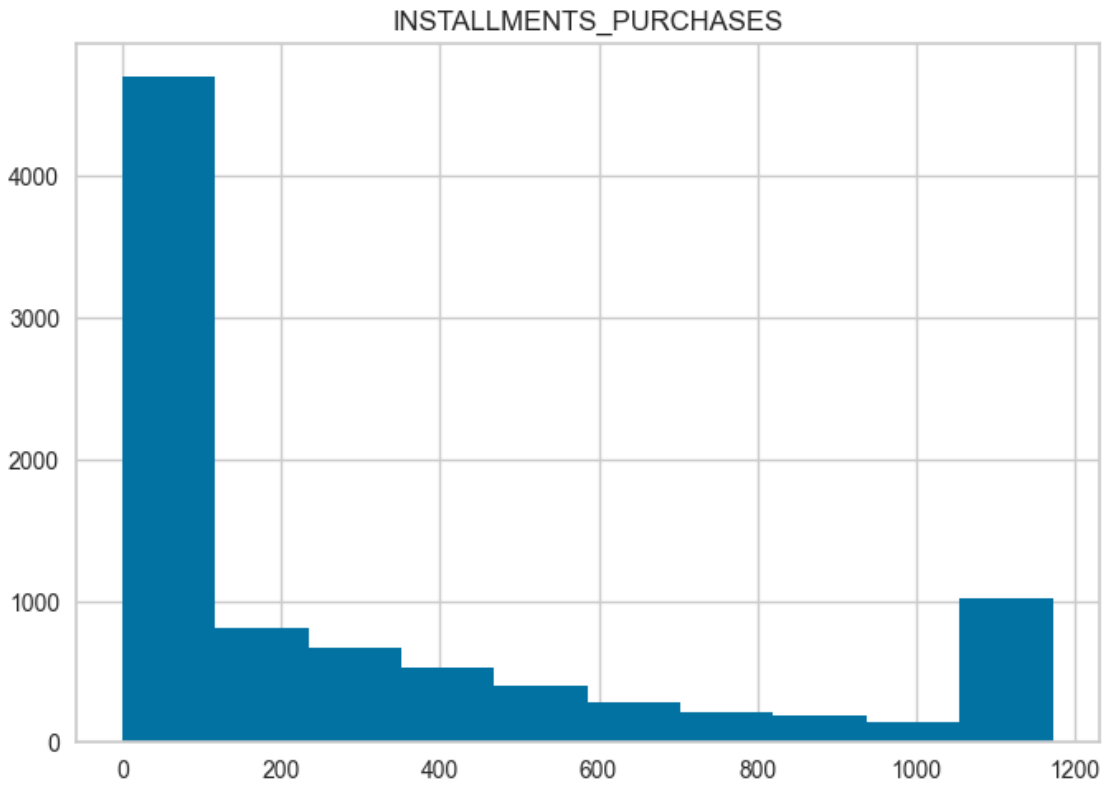

```
<Figure size 800x550 with 0 Axes>
```



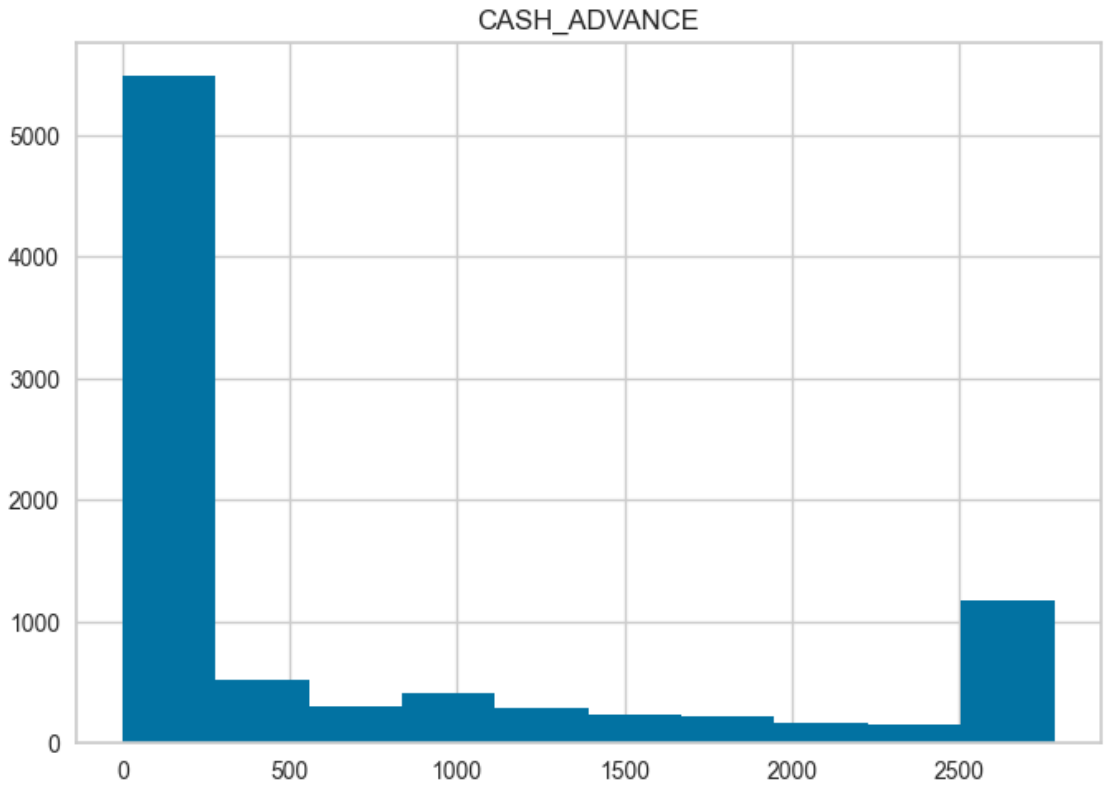

```
<Figure size 800x550 with 0 Axes>
```



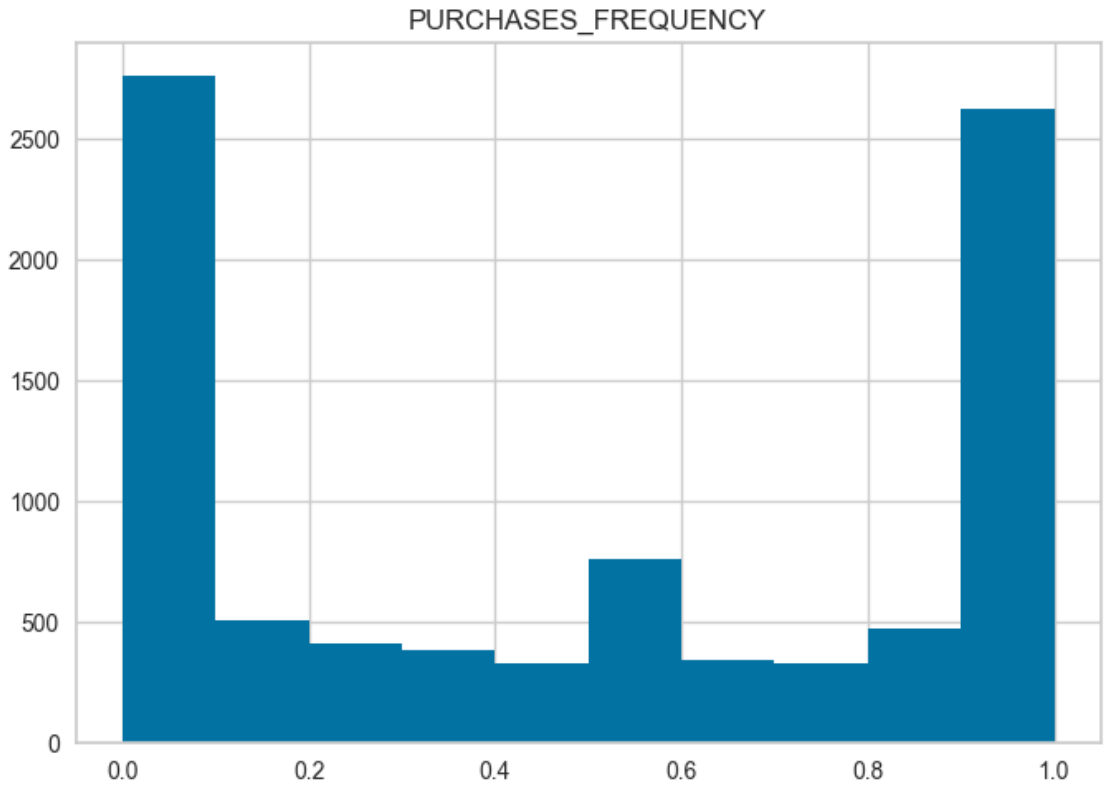

```
<Figure size 800x550 with 0 Axes>
```



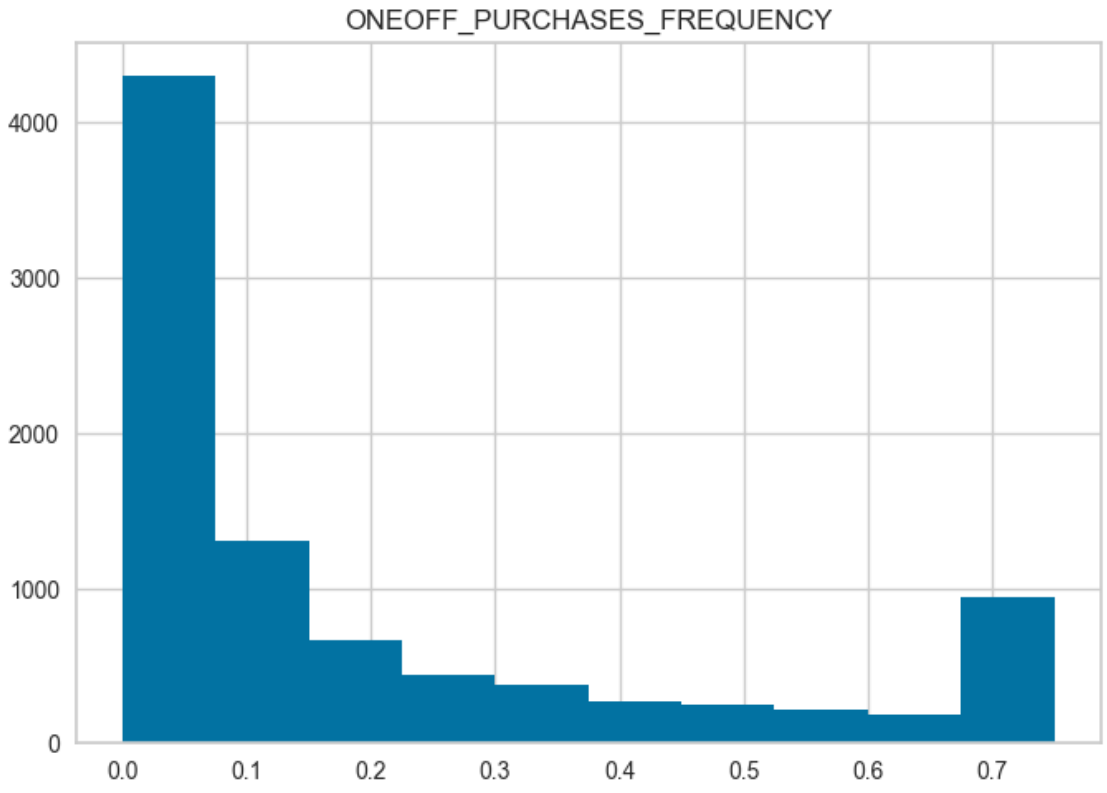

```
<Figure size 800x550 with 0 Axes>
```



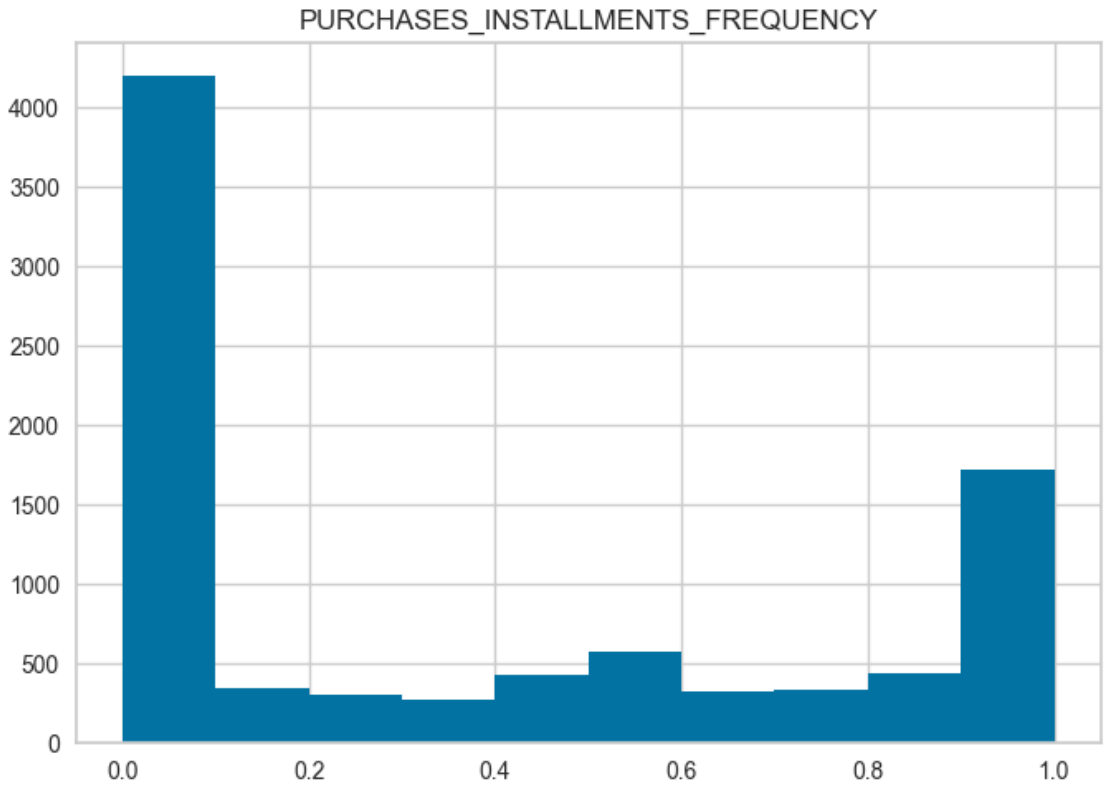

```
<Figure size 800x550 with 0 Axes>
```



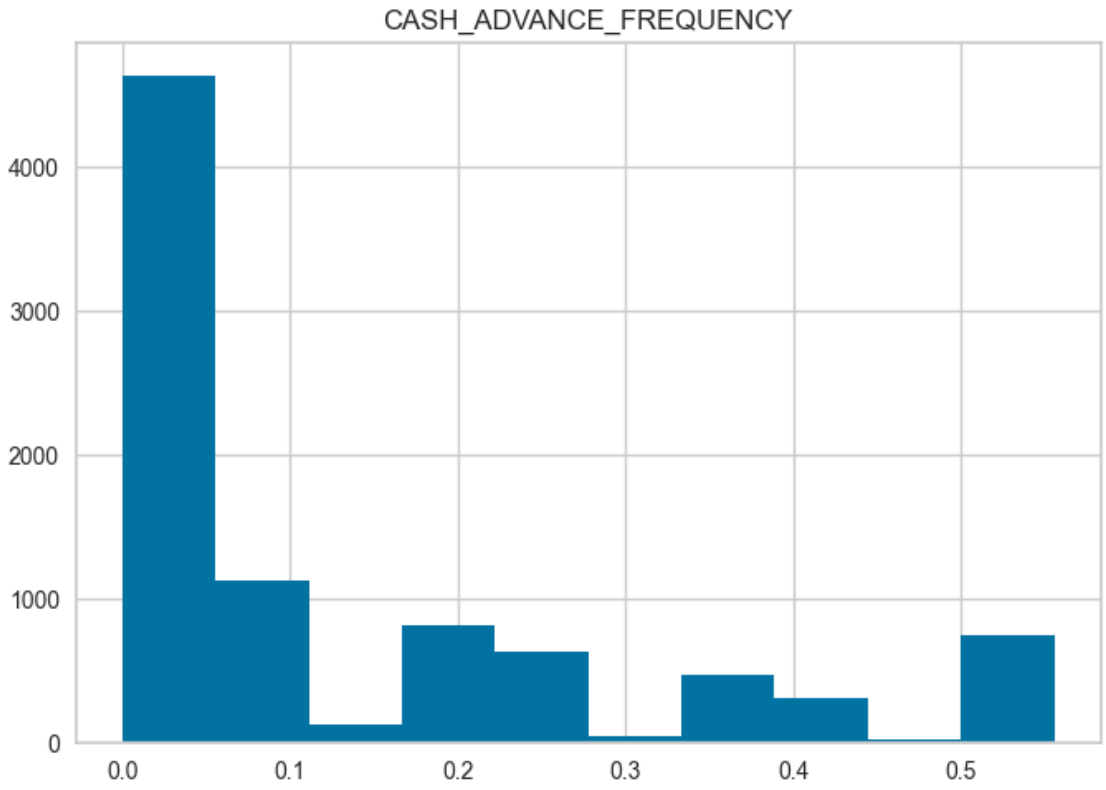

```
<Figure size 800x550 with 0 Axes>
```



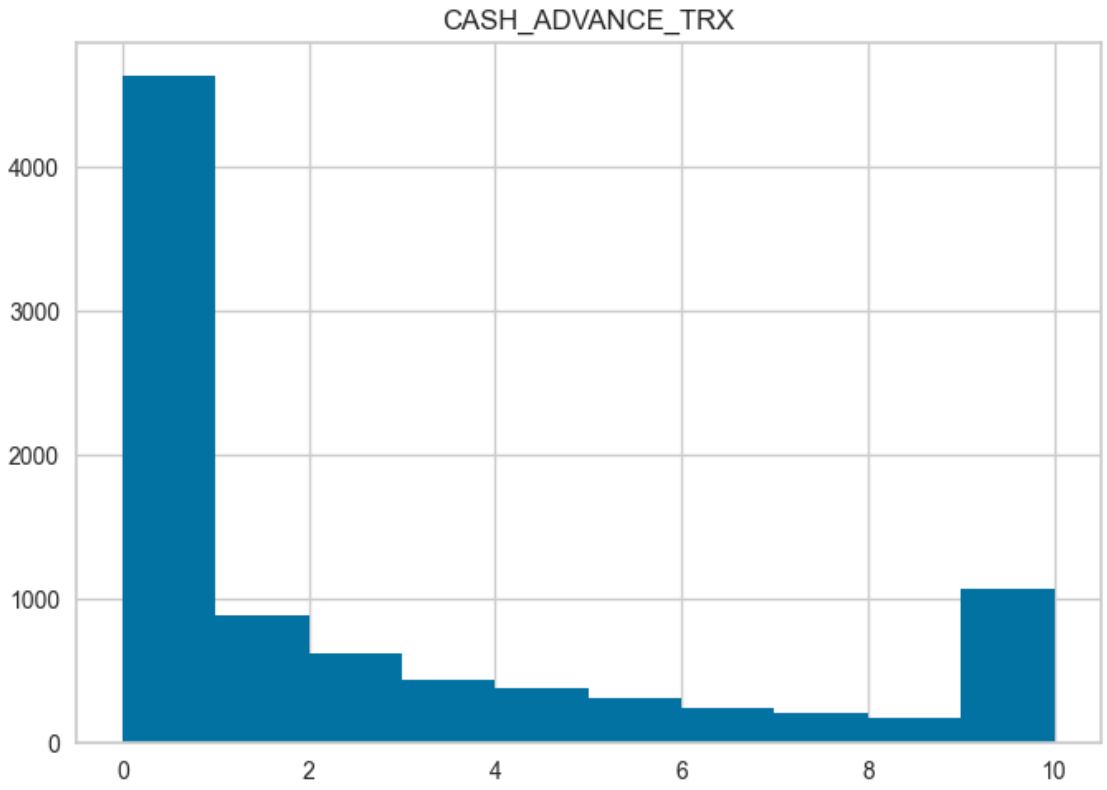

```
<Figure size 800x550 with 0 Axes>
```



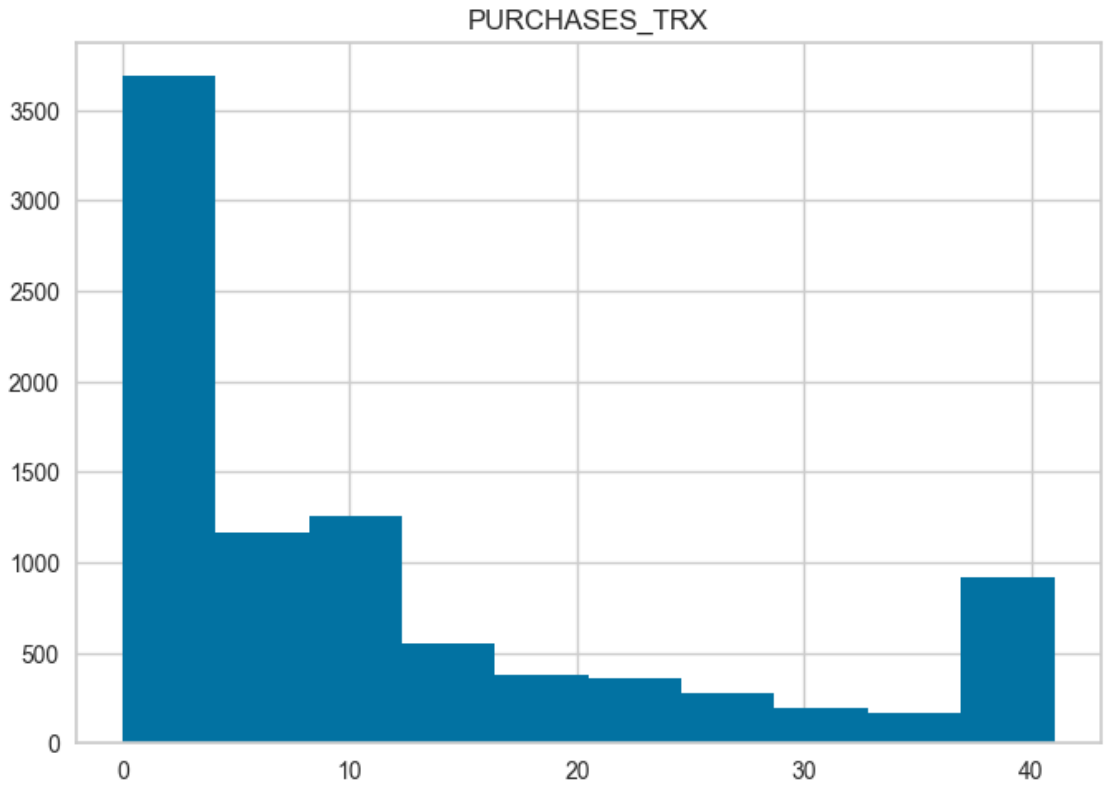

```
<Figure size 800x550 with 0 Axes>
```



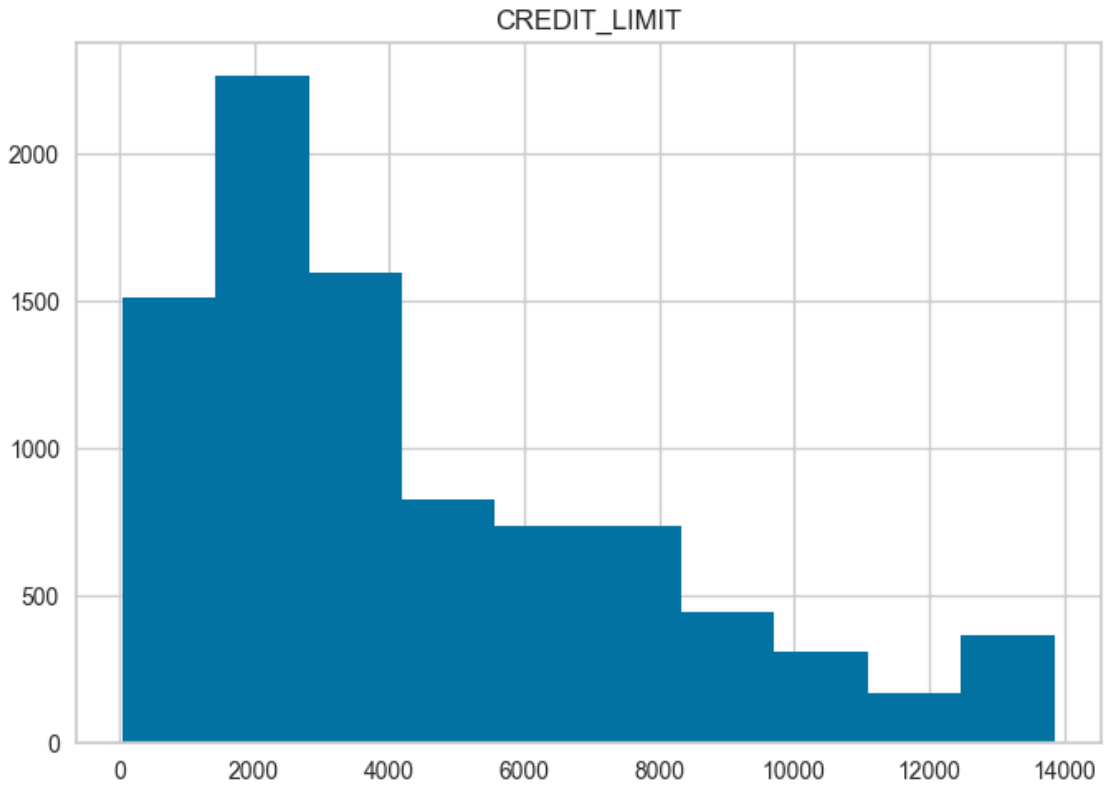

```
<Figure size 800x550 with 0 Axes>
```



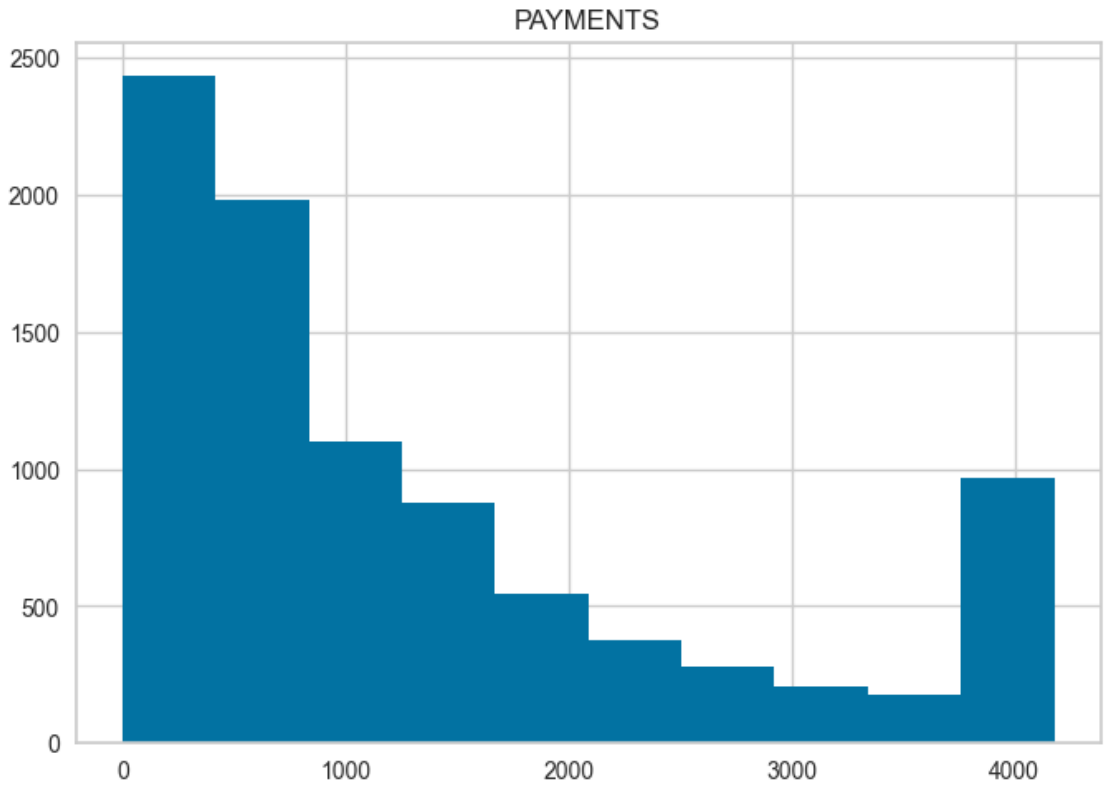

```
<Figure size 800x550 with 0 Axes>
```



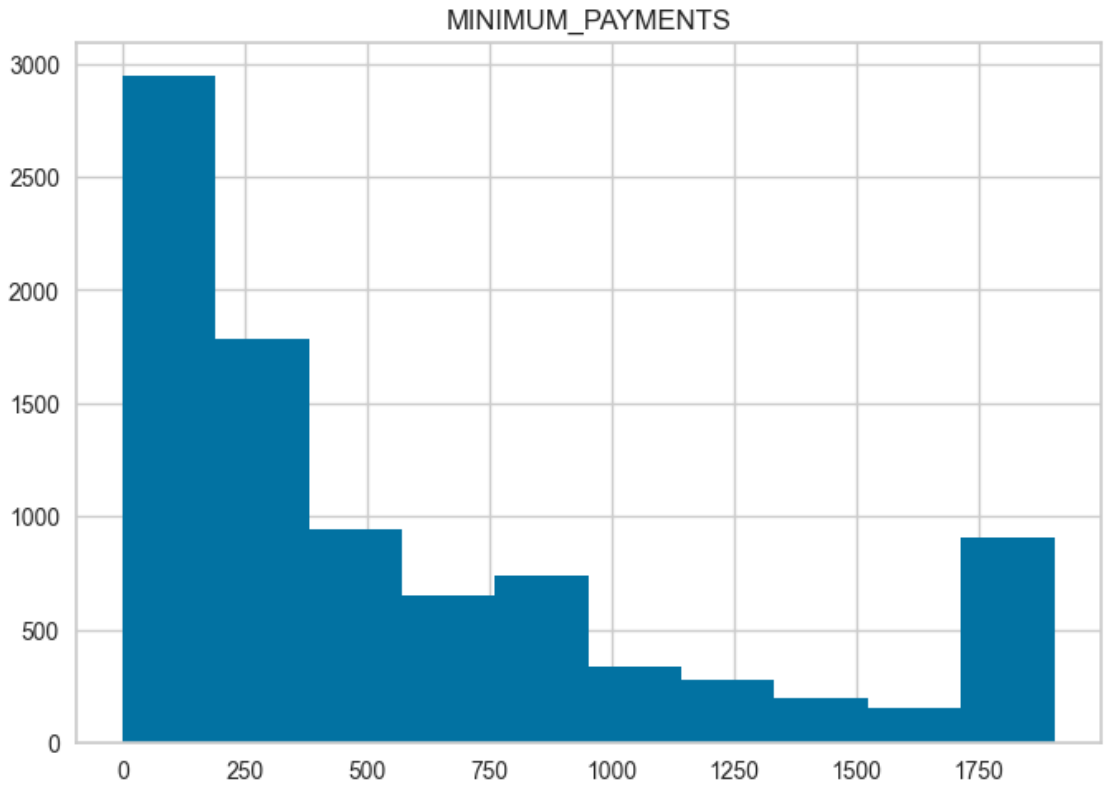

```
<Figure size 800x550 with 0 Axes>
```



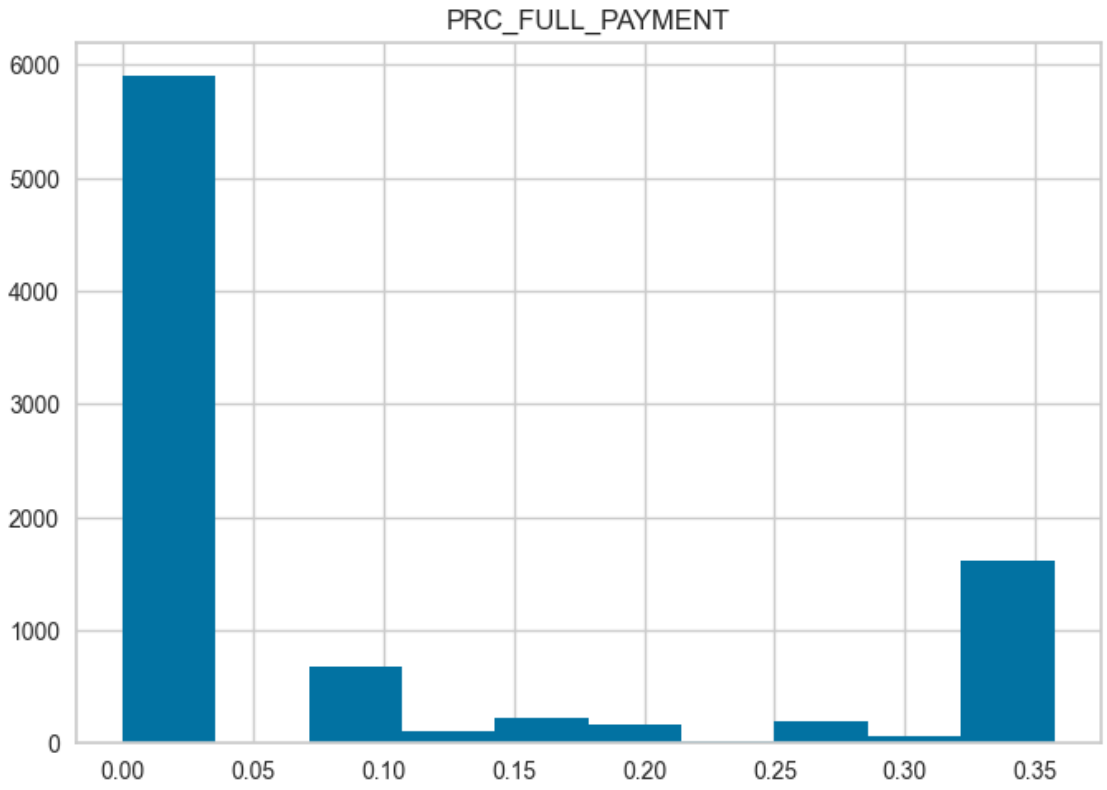

```
<Figure size 800x550 with 0 Axes>
```




```
<Figure size 800x550 with 0 Axes>
```




```
<Figure size 800x550 with 0 Axes>
```

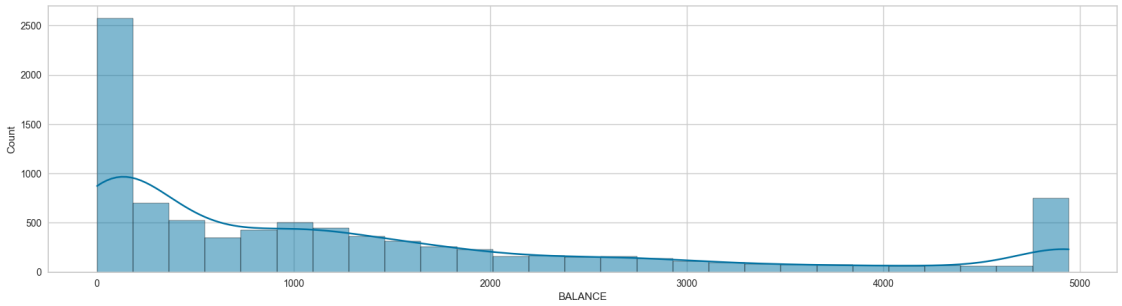


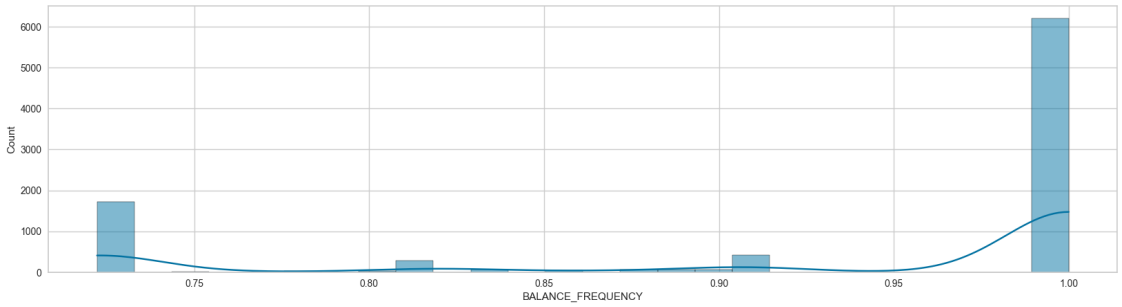

```
<Figure size 800x550 with 0 Axes>
```

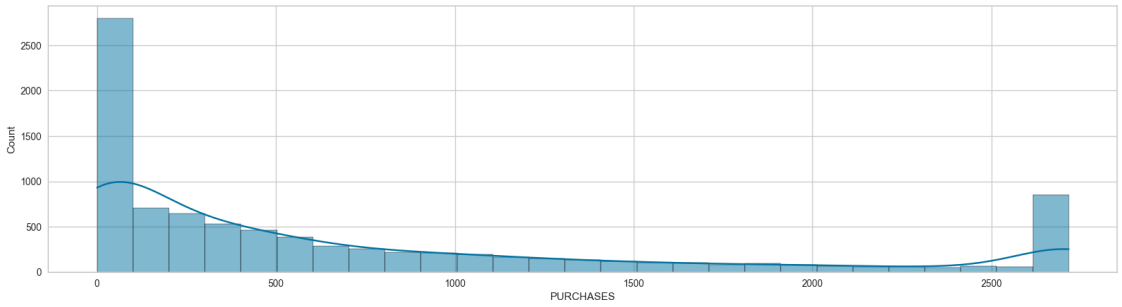
Since it is hard to understand whether there is a skew or not in the dataset, we can use the `histplot()` function in the `Yellowbrick` library.

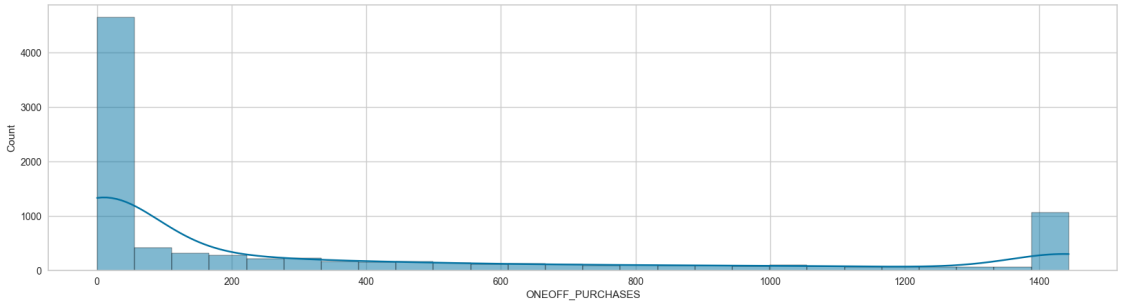
Step 3 - Histplots The function to get histplots is shown below.

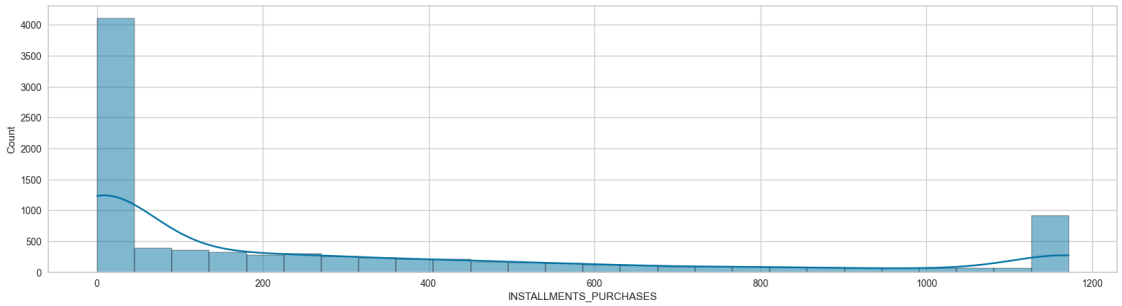
[21]:
```
# Generate histplot to analyze more closely
def draw_histplots(df, field):
    plt.figure(figsize=(20,5))
    sns.histplot(df[field], kde=True)
    plt.show()

# Draw histplots
for field in numerical_fields:
    draw_histplots(df3, field)
```

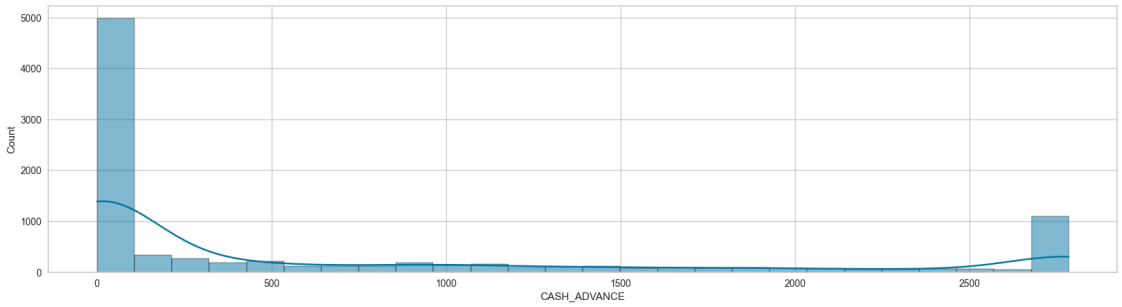

Count
BALANCE

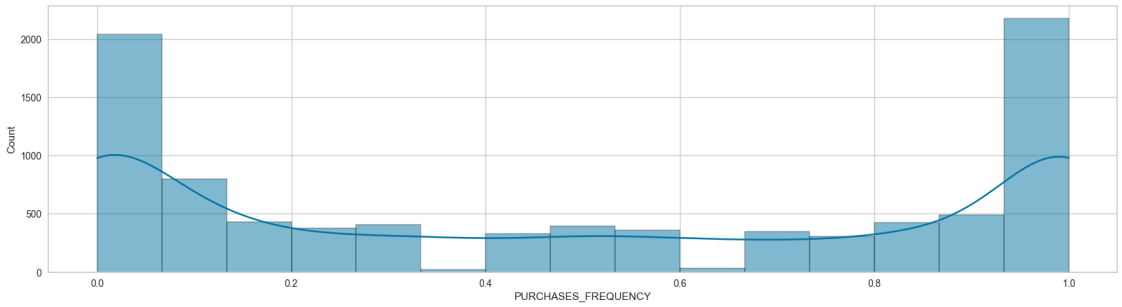

Count
BALANCE_FREQUENCY

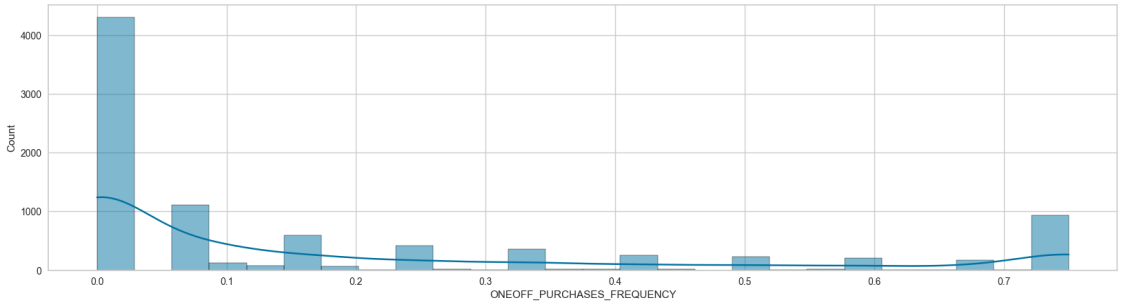

Count
PURCHASES

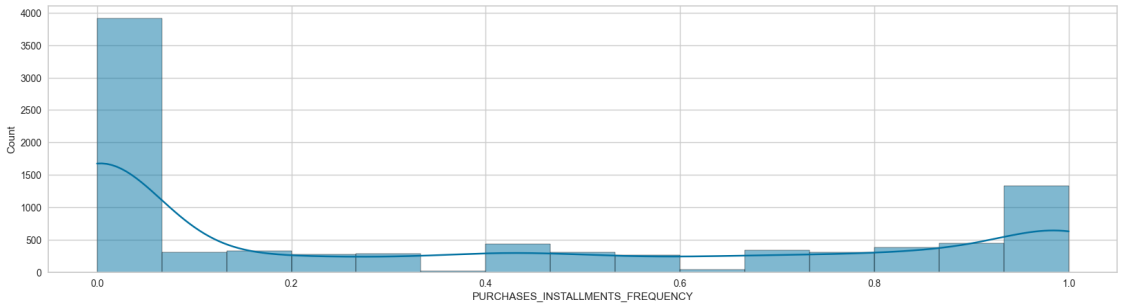

Count
ONEOFF_PURCHASES

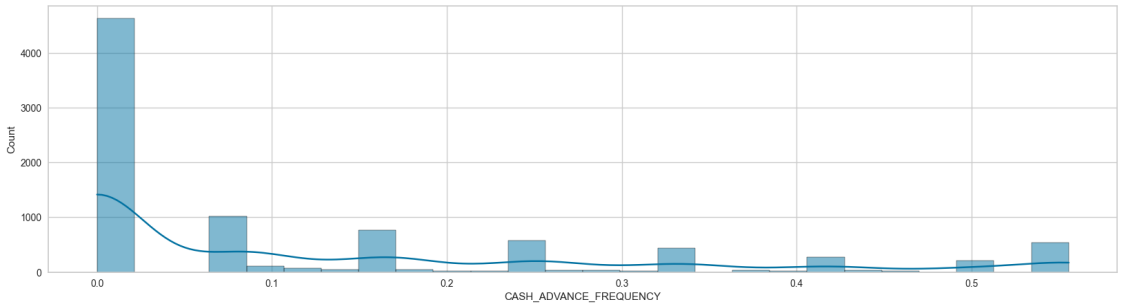

Count
INSTALLMENTS_PURCHASES

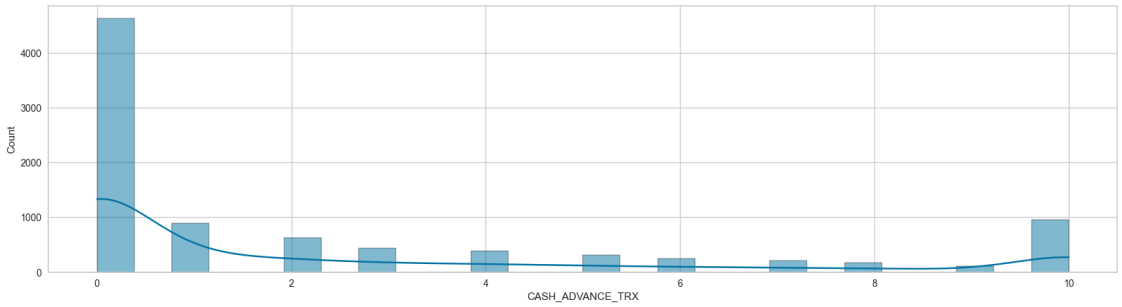

Count
CASH_ADVANCE

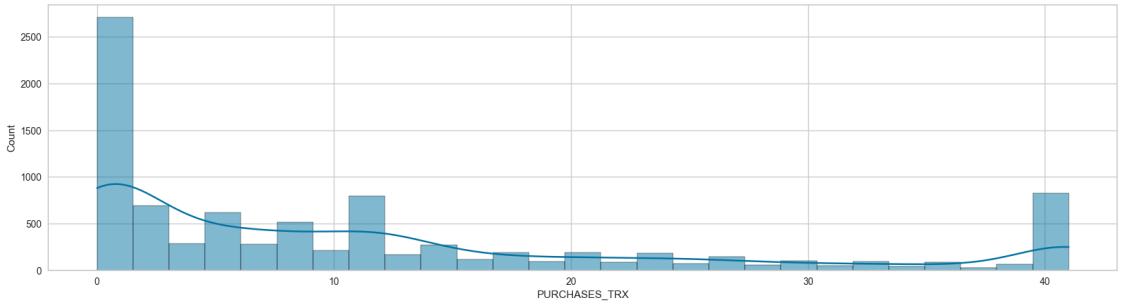

Count
PURCHASES_FREQUENCY

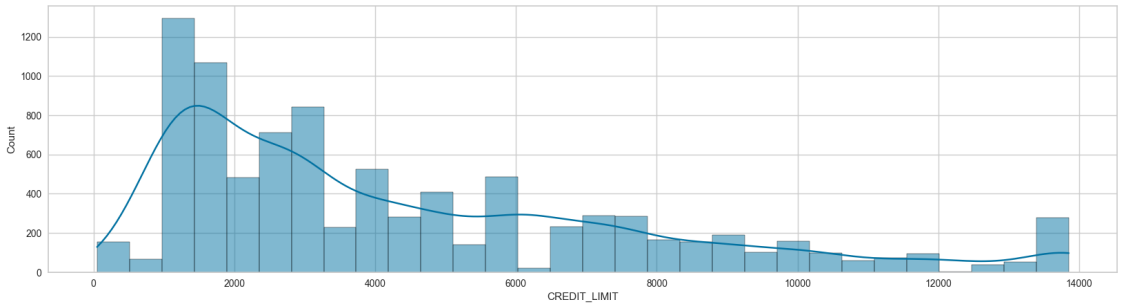

Count
ONEOFF_PURCHASES_FREQUENCY

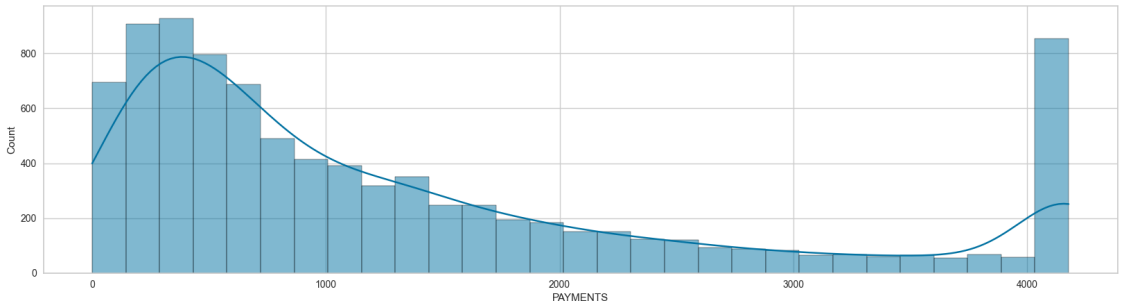

Count
PURCHASES_INSTALLMENTS_FREQUENCY

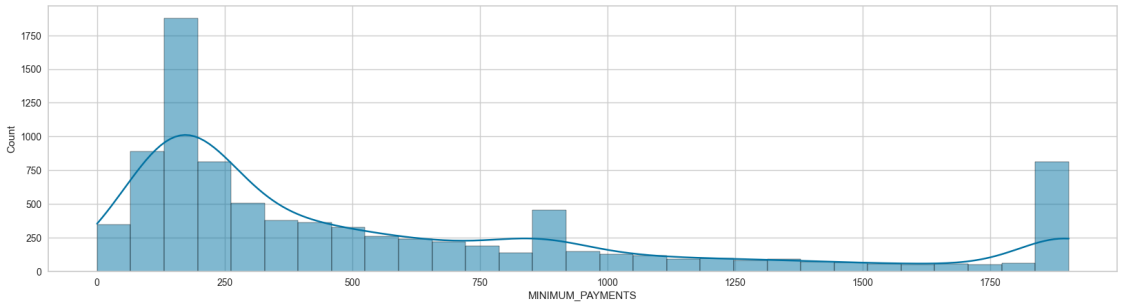

Count
CASH_ADVANCE_FREQUENCY

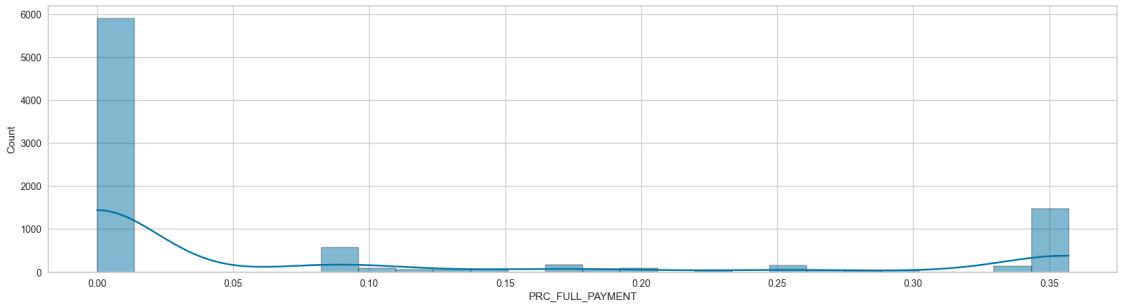

Count
CASH_ADVANCE_TRX


Count
PURCHASES_TRX


Count
CREDIT_LIMIT


Count
PAYMENTS


Count
MINIMUM_PAYMENTS


Count
PRC_FULL_PAYMENT

Step 4 - Applying Transformations From the graphs, we can see that the following columns are right-skewed.

- BALANCE
- PURCHASES
- PURCHASES_TRX
- CREDIT_LIMIT
- PAYMENTS
- MINIMUM_PAYMENTS

The following techniques are used to address the skewed data. 1. Right Skewed Data →
(a) Logarithmic Transformation
(b) Square-root Transformation 3. Left Skewed Data →
(a) Exponential Transformation The `Logarithmic Transformation` cannot be used with 0 and negative values. Therefore, to address those values we can use the `Square-root Transformation` with a little tweak as shown below.

```
FunctionTransformer(lambda x:np.sqrt(abs(x)), validate=True)
```

The following code explains how you can transform the necessary data.

[22]:
```
# Columns identified via histograms
right_skewed = ['BALANCE', 'PURCHASES', 'PURCHASES_TRX',␣
 ↪'CREDIT_LIMIT','PAYMENTS', 'MINIMUM_PAYMENTS']

# Combine the right-skewed and left-skewed columns
additional_columns = [col for col in numerical_fields if col not in␣
 ↪right_skewed]
```

[23]:
```
# Copy the data frame
data = df3.copy()

# Create a function transformer object with square root transformation for␣
 ↪right-skewed data
square_root_transformer = FunctionTransformer(lambda x:np.sqrt(abs(x)),␣
 ↪validate=True)

# Apply the transformations
data_new1 = square_root_transformer.transform(data[right_skewed])

# Create a new data frame
df4 = pd.DataFrame(data_new1, columns=right_skewed)
df4.head()
```

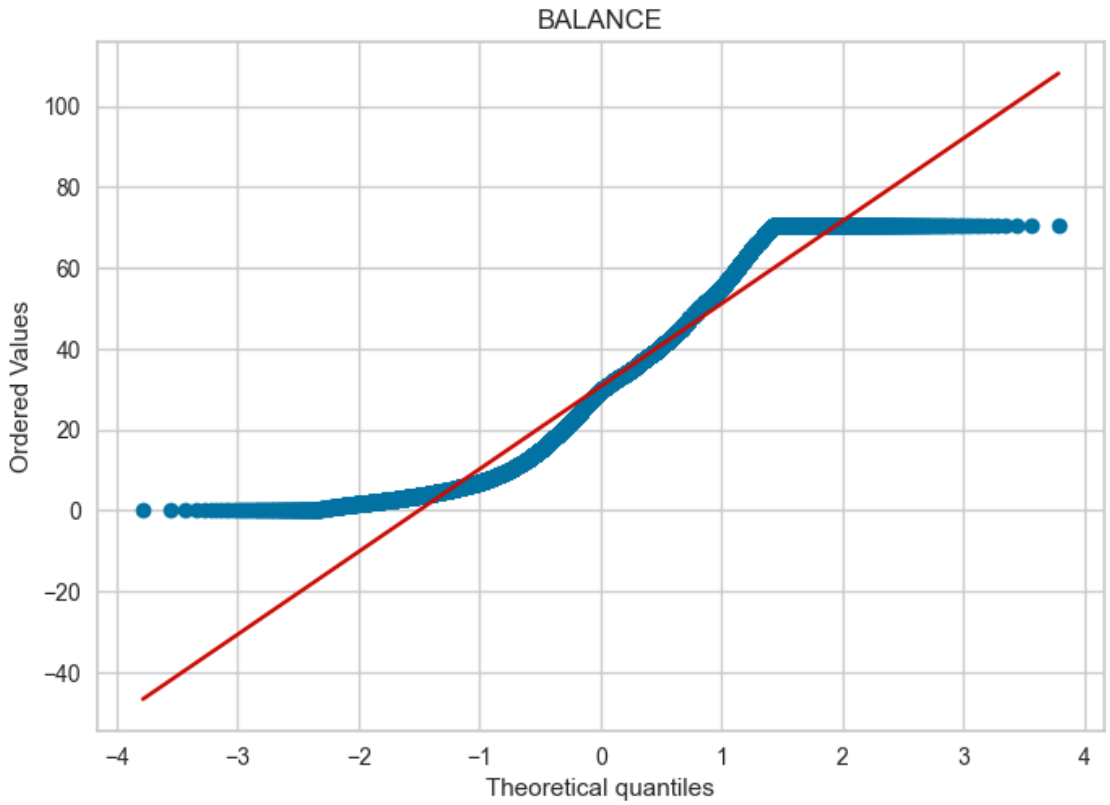
[23]:

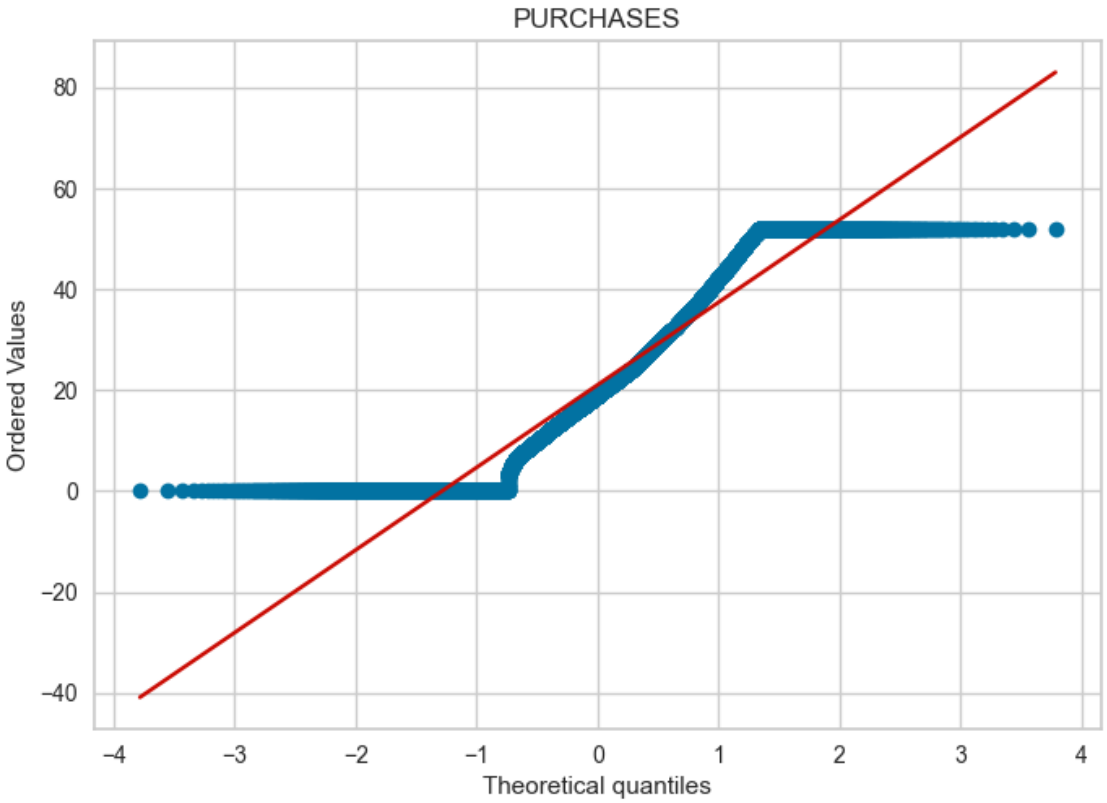
	BALANCE	PURCHASES	PURCHASES_TRX	CREDIT_LIMIT	PAYMENTS	\
0	6.395369	9.767292	1.414214	31.622777	14.205706	
1	56.590347	0.000000	0.000000	83.666003	64.054919	
2	49.951465	27.805935	3.464102	86.602540	24.941266	
3	40.824877	38.716921	1.000000	86.602540	0.000000	

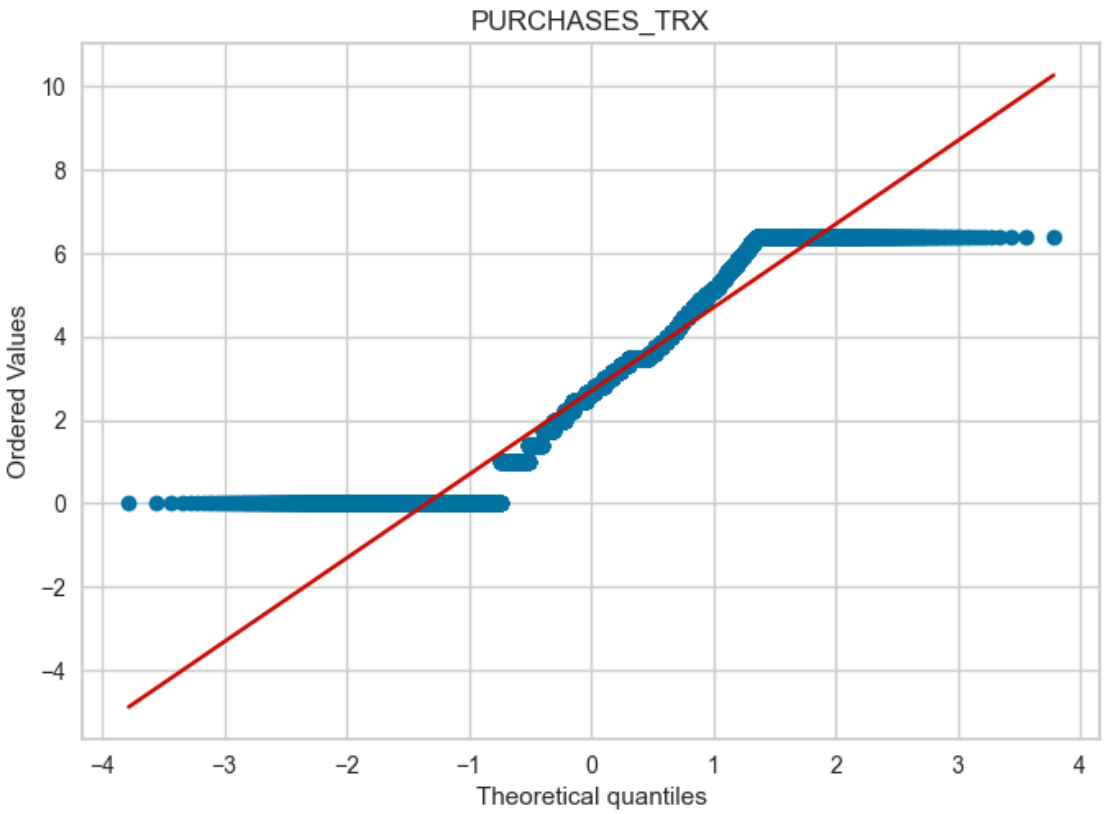
```
4  28.595705    4.000000        1.000000      34.641016   26.044861

   MINIMUM_PAYMENTS
0         11.811426
1         32.746606
2         25.045654
3         29.397390
4         15.645806
```

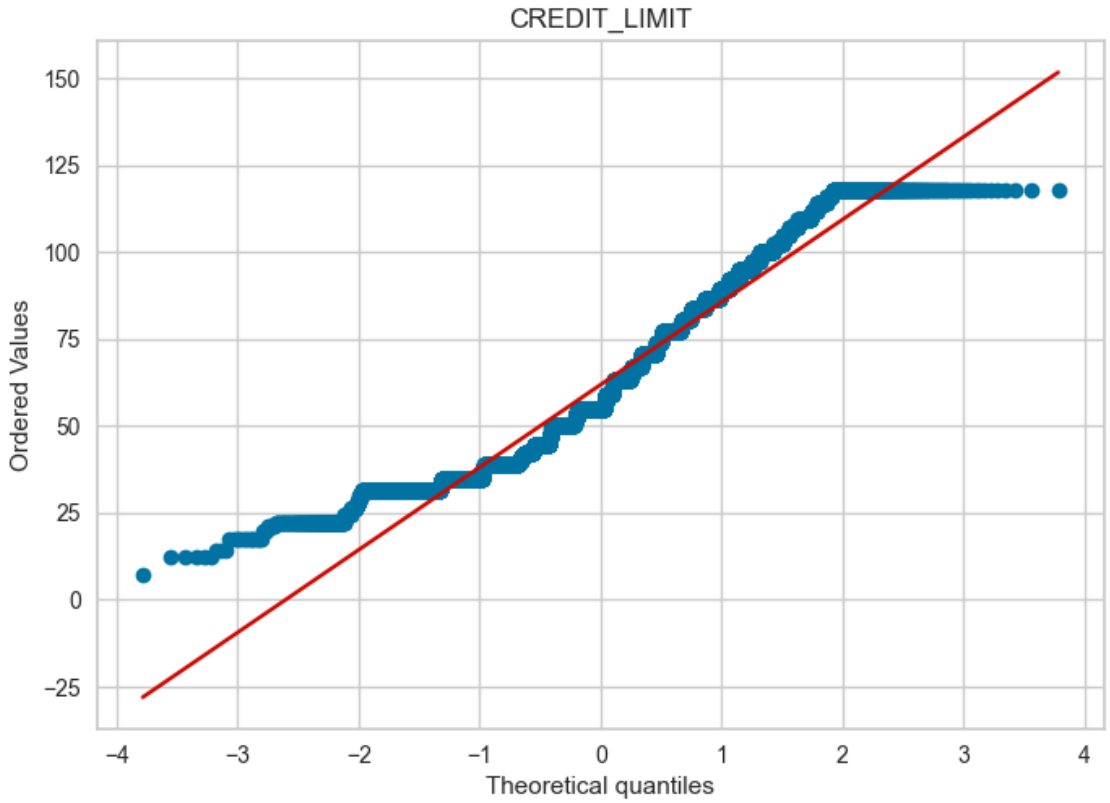
Step 5 - Verifying Let's check the Q-Q plots again to see if the transformation applied properly for the transformed columns.

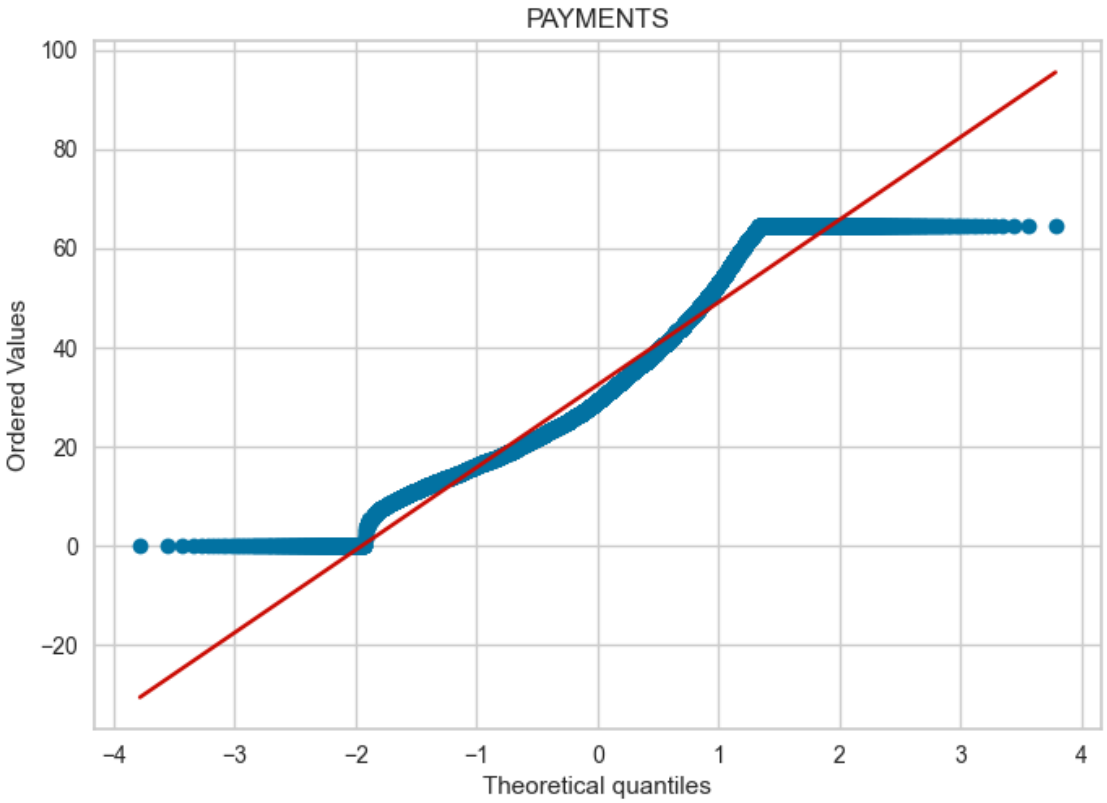
[24]:
```
# Check the q-q plots for the transformed values
for field in right_skewed:
    draw_qq_plots(df4, field)
```

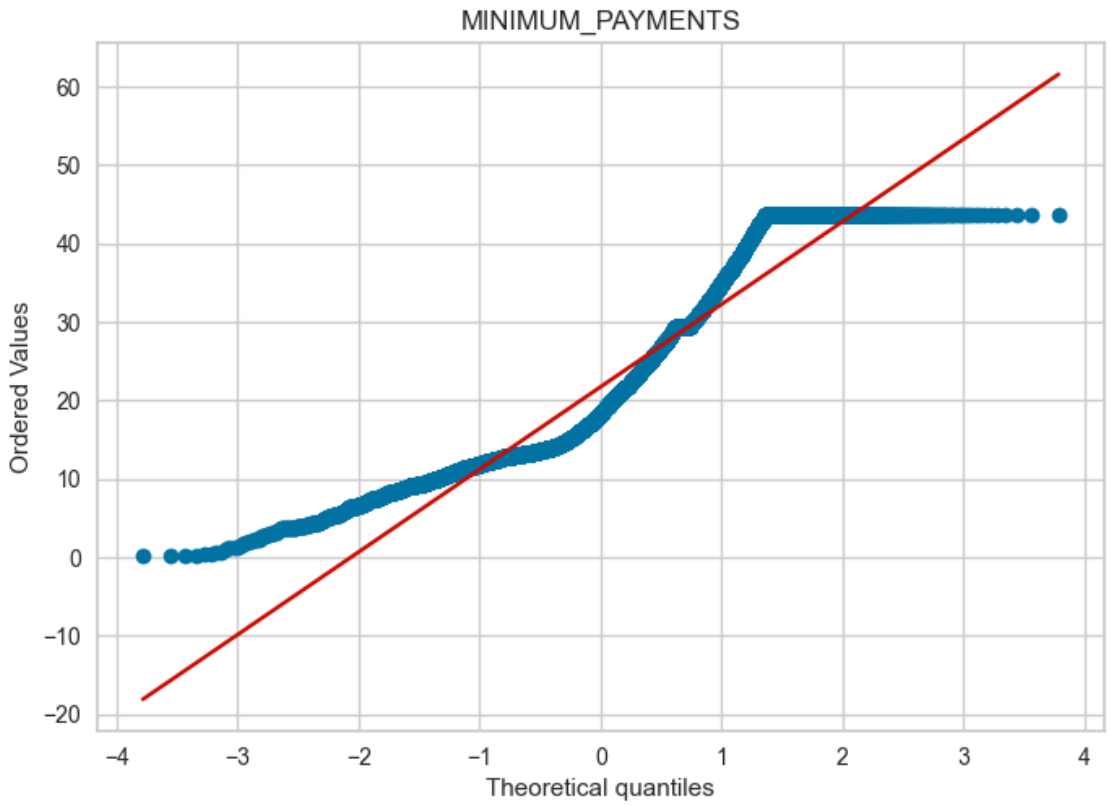



PURCHASES
Ordered Values
Theoretical quantiles
80
60
40
20
0
−20
−40
−4
−3
−2
−1
0
1
2
3
4


PURCHASES_TRX
Ordered Values
Theoretical quantiles
10
8
6
4
2
0
−2
−4
−4
−3
−2
−1
0
1
2
3
4

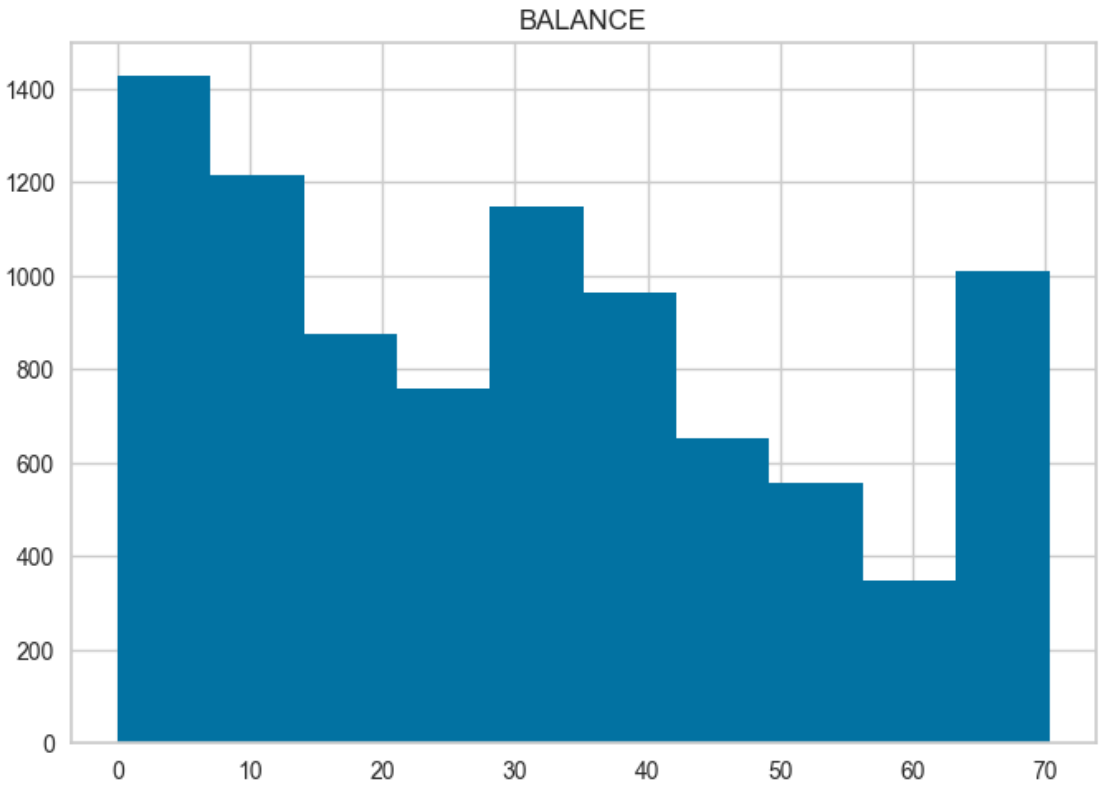

CREDIT_LIMIT
Ordered Values
Theoretical quantiles
150
125
100
75
50
25
0
−25
−4
−3
−2
−1
0
1
2
3
4


PAYMENTS
Ordered Values
Theoretical quantiles
100
80
60
40
20
0
−20
−4
−3
−2
−1
0
1
2
3
4

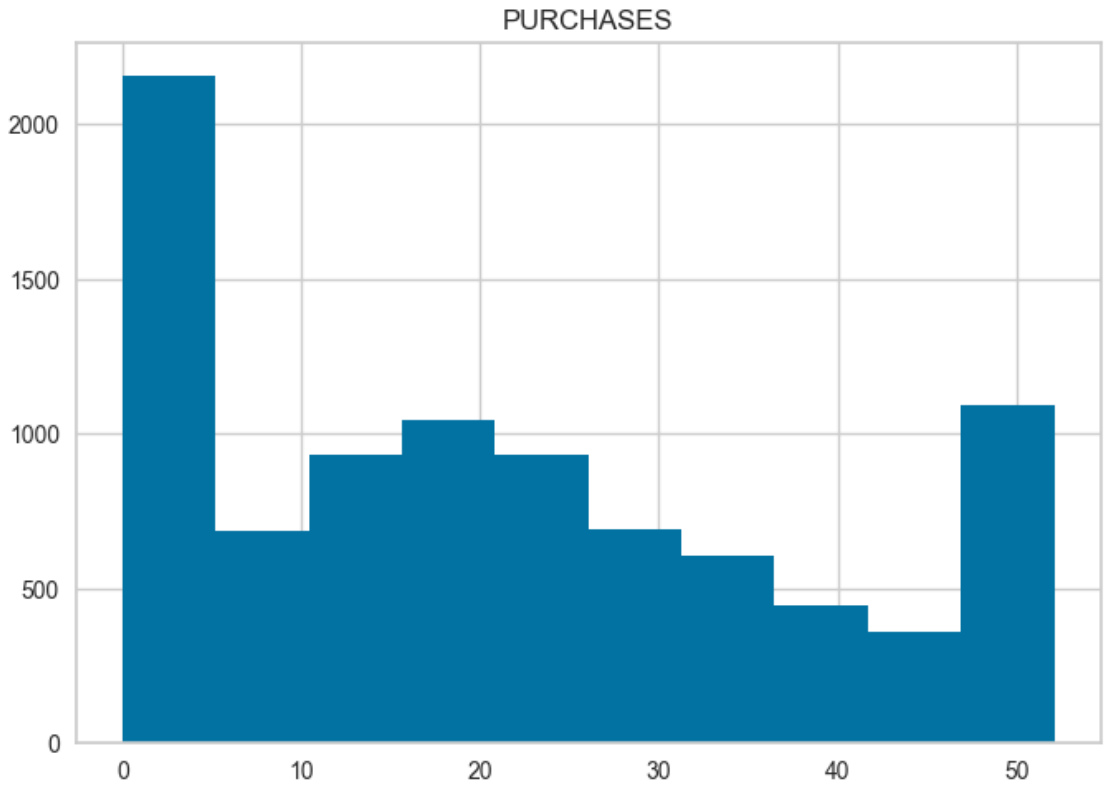


Similarly, let's check the Histograms to see if the transformation applied properly for the transformed columns.

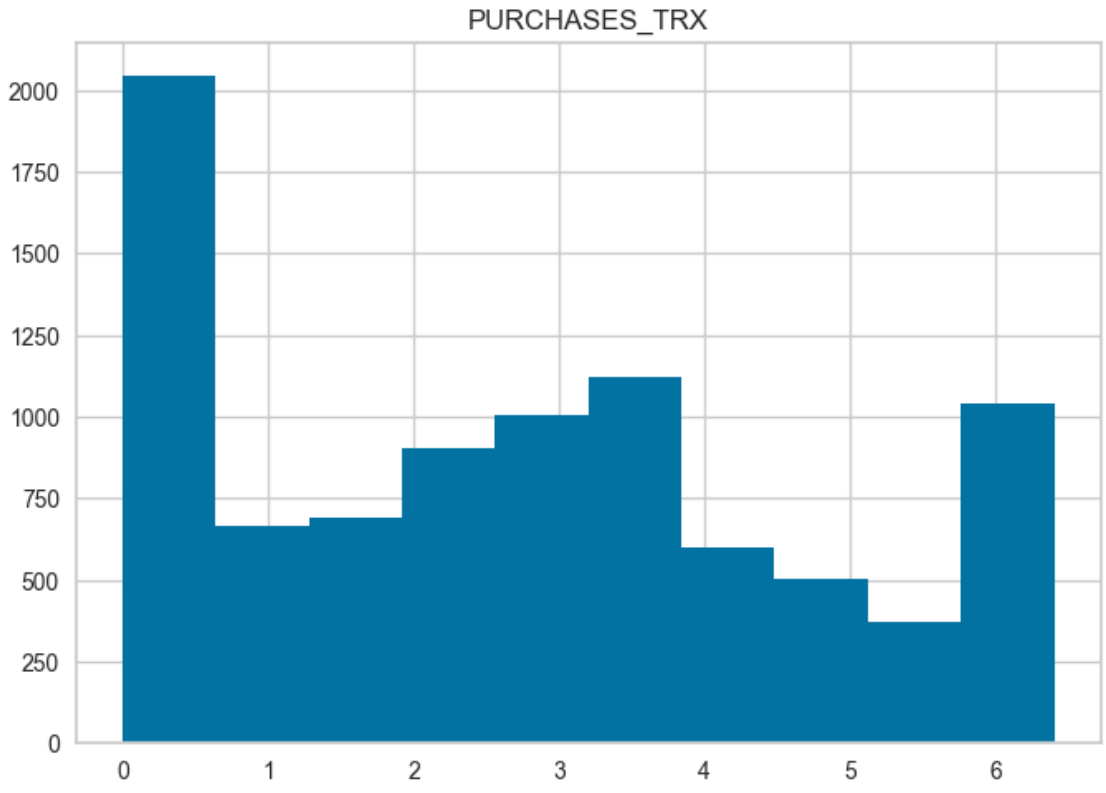
[25]:
```
# Check the histograms for the transformed values
for field in right_skewed:
    draw_histograms(df4, field)
```



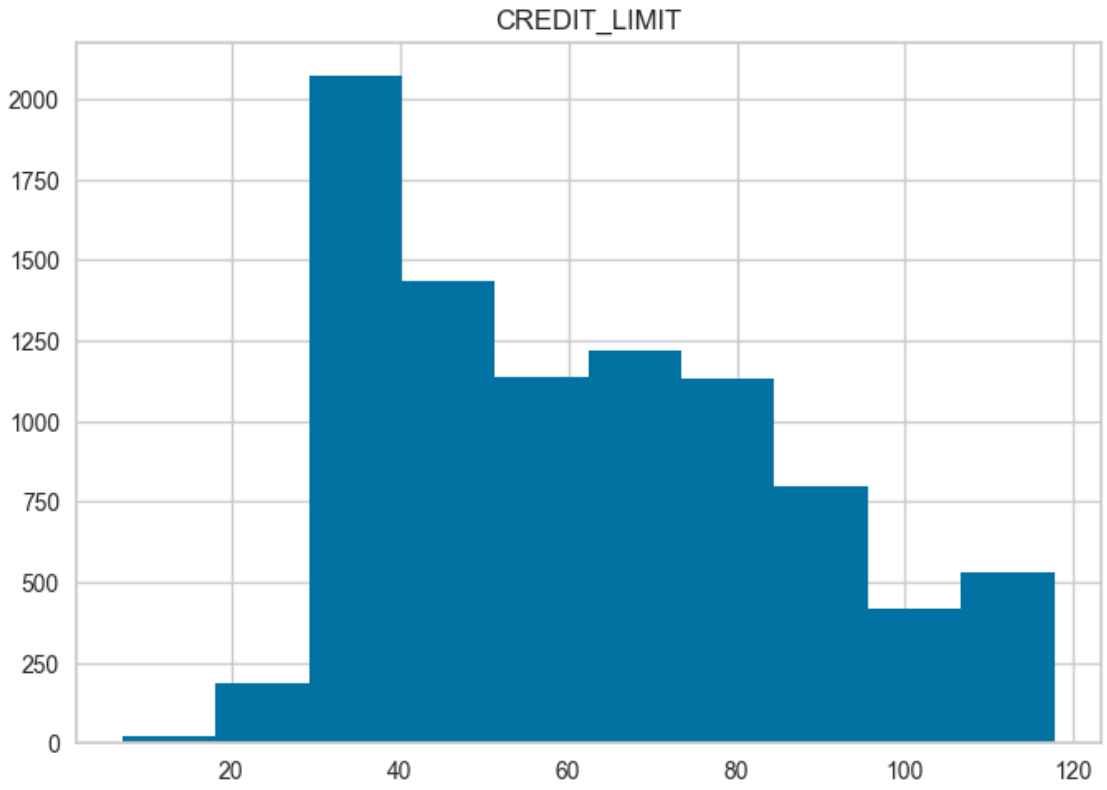

```
<Figure size 800x550 with 0 Axes>
```



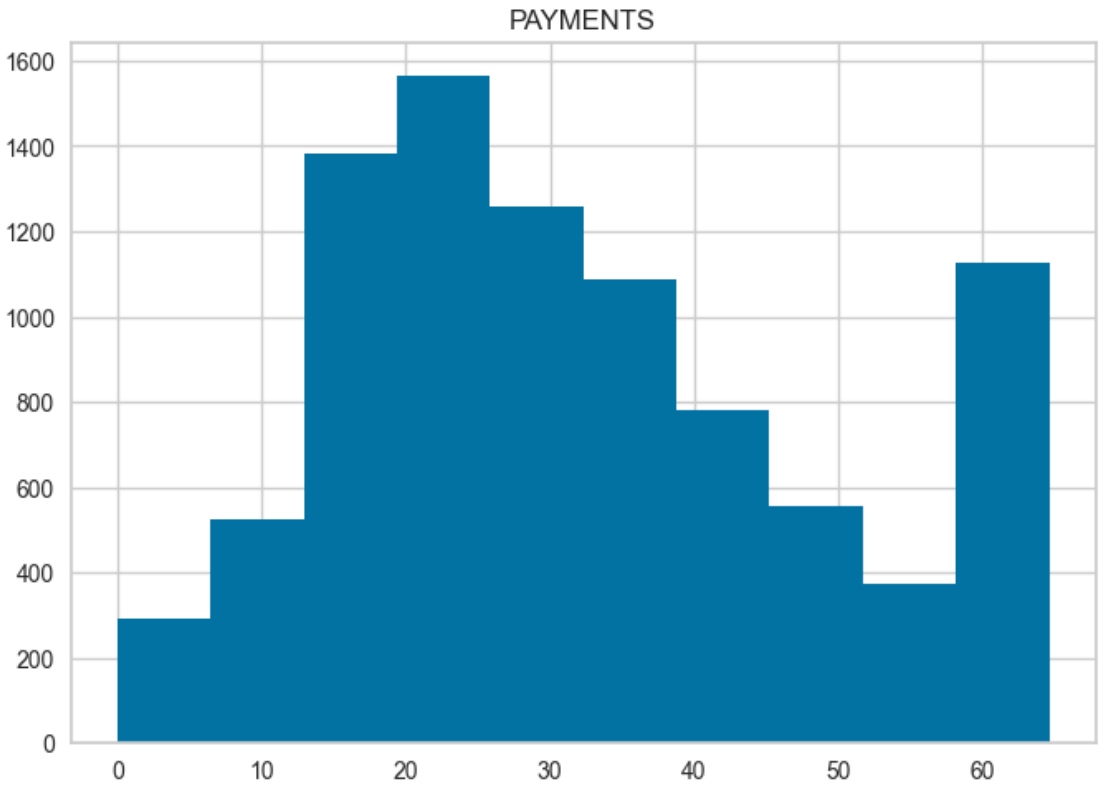

```
<Figure size 800x550 with 0 Axes>
```



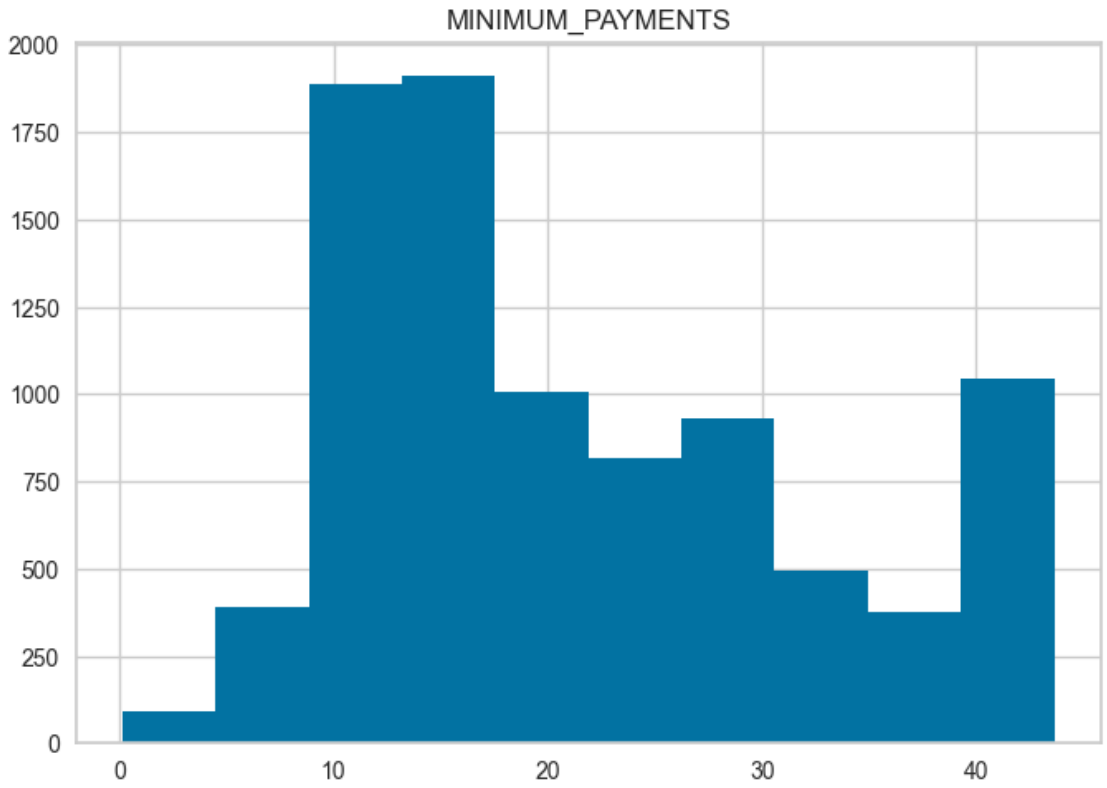

```
<Figure size 800x550 with 0 Axes>
```




```
<Figure size 800x550 with 0 Axes>
```



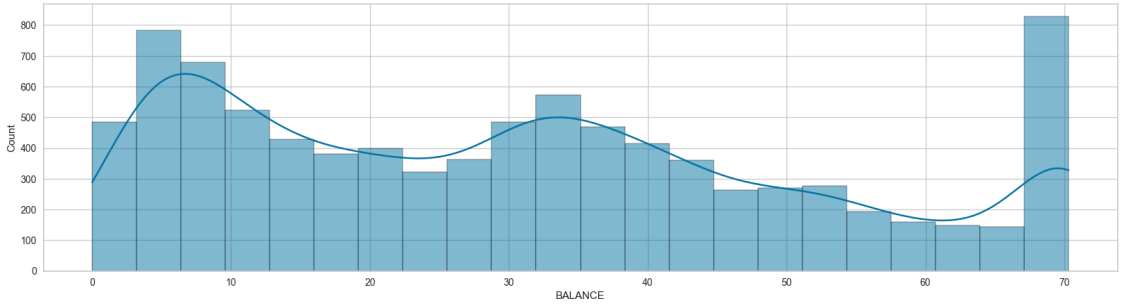

```
<Figure size 800x550 with 0 Axes>
```

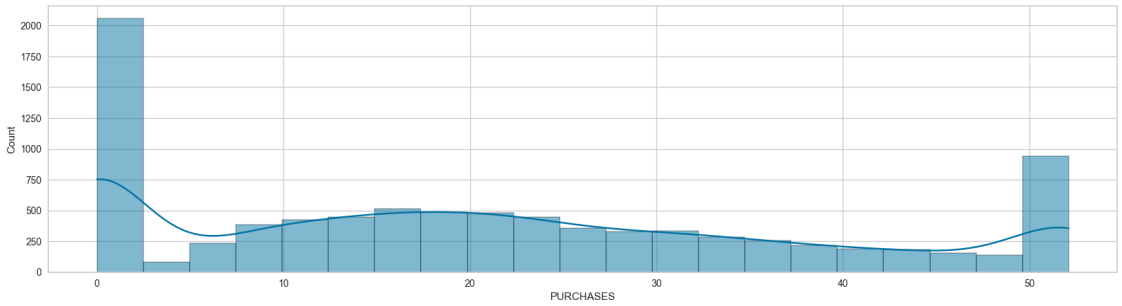


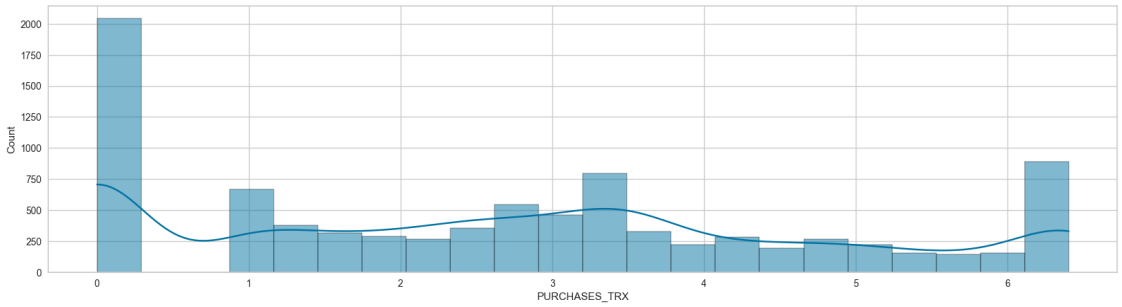

```
<Figure size 800x550 with 0 Axes>
```

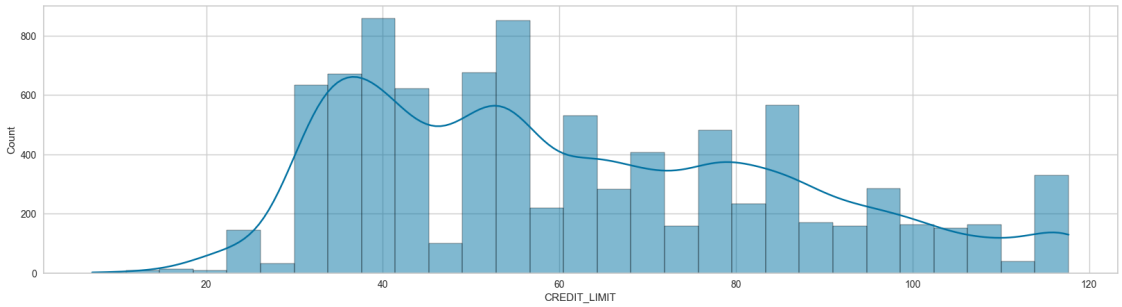
Similarly, let's check the Histplots to see if the transformation applied properly for the transformed columns.

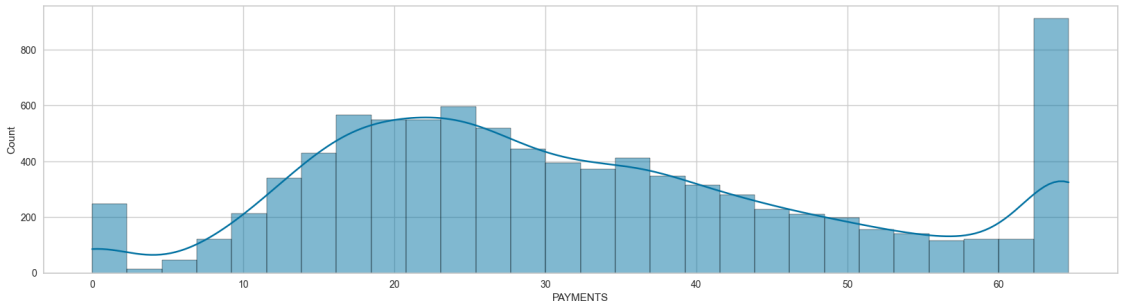
[26]:
```
# Check histplots for the transformed values
for field in right_skewed:
    draw_histplots(df4, field)
```

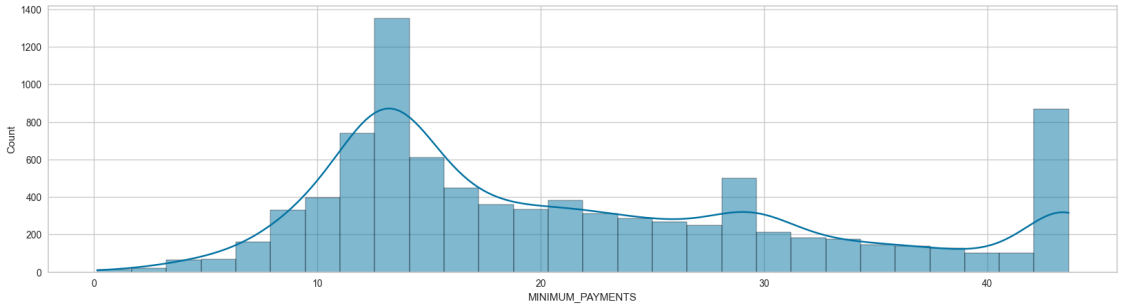



Count
PURCHASES


Count
PURCHASES_TRX


Count
CREDIT_LIMIT


Count
PAYMENTS



Append the non-transformed columns back to the data frame again.

[27]:
```
# Get additional data as a data frame
additional_data = df3[additional_columns]

# Get 'TENURE' data as a data frame
tenure_data = df3['TENURE']
tenure_data = tenure_data.to_frame()

# Concatenate the data frames together
df5 = pd.concat([df4, additional_data, tenure_data], join='outer', axis=1).
 ↪reset_index(drop=True)

# Check whether the concatenation is done properly
df5.head()
```

[27]:

	BALANCE	PURCHASES	PURCHASES_TRX	CREDIT_LIMIT	PAYMENTS	\
0	6.395369	9.767292	1.414214	31.622777	14.205706	
1	56.590347	0.000000	0.000000	83.666003	64.054919	
2	49.951465	27.805935	3.464102	86.602540	24.941266	
3	40.824877	38.716921	1.000000	86.602540	0.000000	
4	28.595705	4.000000	1.000000	34.641016	26.044861	

	MINIMUM_PAYMENTS	BALANCE_FREQUENCY	ONEOFF_PURCHASES	\
0	11.811426	0.818182	0.0000	
1	32.746606	0.909091	0.0000	
2	25.045654	1.000000	773.1700	
3	29.397390	0.722223	1443.5125	
4	15.645806	1.000000	16.0000	

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	\
0	95.4	0.000000	0.166667	
1	0.0	2784.552848	0.000000	
2	0.0	0.000000	1.000000	
3	0.0	205.788017	0.083333	
4	0.0	0.000000	0.083333	

```
   ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
0                    0.000000                          0.083333
1                    0.000000                          0.000000
2                    0.750000                          0.000000
3                    0.083333                          0.000000
4                    0.083333                          0.000000

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PRC_FULL_PAYMENT  TENURE
0                0.000000               0.0          0.000000      12
1                0.250000               4.0          0.222222      12
2                0.000000               0.0          0.000000      12
3                0.083333               1.0          0.000000      12
4                0.000000               0.0          0.000000      12
```

Check the data frame information as well.

[28]:
```
df5.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8950 entries, 0 to 8949
Data columns (total 17 columns):
 #   Column                            Non-Null Count  Dtype
---  ------                            --------------  -----
 0   BALANCE                           8950 non-null   float64
 1   PURCHASES                         8950 non-null   float64
 2   PURCHASES_TRX                     8950 non-null   float64
 3   CREDIT_LIMIT                      8950 non-null   float64
 4   PAYMENTS                          8950 non-null   float64
 5   MINIMUM_PAYMENTS                  8950 non-null   float64
 6   BALANCE_FREQUENCY                 8950 non-null   float64
 7   ONEOFF_PURCHASES                  8950 non-null   float64
 8   INSTALLMENTS_PURCHASES            8950 non-null   float64
 9   CASH_ADVANCE                      8950 non-null   float64
 10  PURCHASES_FREQUENCY               8950 non-null   float64
 11  ONEOFF_PURCHASES_FREQUENCY        8950 non-null   float64
 12  PURCHASES_INSTALLMENTS_FREQUENCY  8950 non-null   float64
 13  CASH_ADVANCE_FREQUENCY            8950 non-null   float64
 14  CASH_ADVANCE_TRX                  8950 non-null   float64
 15  PRC_FULL_PAYMENT                  8950 non-null   float64
 16  TENURE                            8950 non-null   int64
dtypes: float64(16), int64(1)
memory usage: 1.2 MB
```

1.2.3 Applying Suitable Feature Encoding Techniques

Since our data frame contains only numerical values(according to `df.info()` function), we do not need to do feature encodings.

1.2.4 Scale/Standardize Data

Feature Scaling, also known as data normalization, is a technique used to standardize the range of independent variables of features of data. There are a few ways to scale or standardize data.

- Min/Max Scaling(Normalization) → Rescales features to a range between 0 and 1.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

- Standardization(Z-score Normalization) → Centers the data around 0 with a standard deviation of 1.

$$X_{std} = \frac{X - mean(X)}{std(X)}$$

- Robust Scaling → Similar to standardization but uses the median and the inter-quartile range.

$$X_{robust} = \frac{X - median(X)}{Q_3(X) - Q_1(X)}$$

In this case, we are using the `Standard Scaling` to center the data around 0 with a standard deviation of 1, since it helps more in the clustering process. The code snippet used to scale the data is shown below.

[29]:
```
# Create the scaler object
scaler = StandardScaler()

# Copy the data frame
df_scaled = df5.copy()
df_scaled[df_scaled.columns] = scaler.fit_transform(df_scaled[df_scaled.
 ↪columns])

# Check the data
df_scaled.head()
```

[29]:

	BALANCE	PURCHASES	PURCHASES_TRX	CREDIT_LIMIT	PAYMENTS	\
0	-1.151173	-0.660913	-0.615419	-1.239281	-1.071948	
1	1.224025	-1.231647	-1.293663	0.894879	1.849656	
2	0.909877	0.393144	0.367690	1.015299	-0.442750	
3	0.478012	1.030708	-0.814072	1.015299	-1.904528	
4	-0.100666	-0.997914	-0.814072	-1.115511	-0.378069	

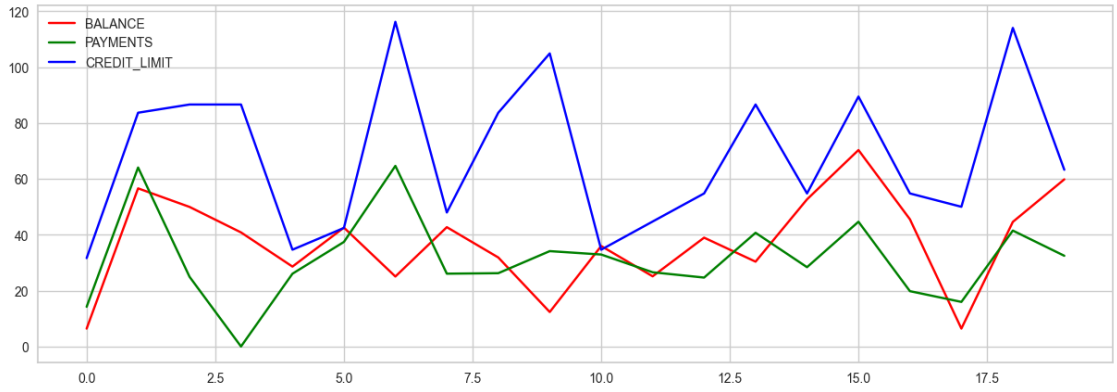
	MINIMUM_PAYMENTS	BALANCE_FREQUENCY	ONEOFF_PURCHASES	\
0	-0.899715	-1.021875	-0.692383	
1	0.999354	-0.202708	-0.692383	
2	0.300786	0.616459	0.818320	
3	0.695540	-1.886552	2.128108	
4	-0.551892	0.616459	-0.661121	

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	\
0	-0.505216	-0.673507	-0.806490	

```
1               -0.746029      2.115343           -1.221758
2               -0.746029     -0.673507            1.269843
3               -0.746029     -0.467401           -1.014125
4               -0.746029     -0.673507           -1.014125

   ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
0                   -0.722749                         -0.707313
1                   -0.722749                         -0.916995
2                    2.210909                         -0.916995
3                   -0.396788                         -0.916995
4                   -0.396788                         -0.916995

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PRC_FULL_PAYMENT   TENURE
0               -0.730084         -0.697293         -0.629277  0.36068
1                0.717792          0.473089          0.973961  0.36068
2               -0.730084         -0.697293         -0.629277  0.36068
3               -0.247460         -0.404697         -0.629277  0.36068
4               -0.730084         -0.697293         -0.629277  0.36068
```

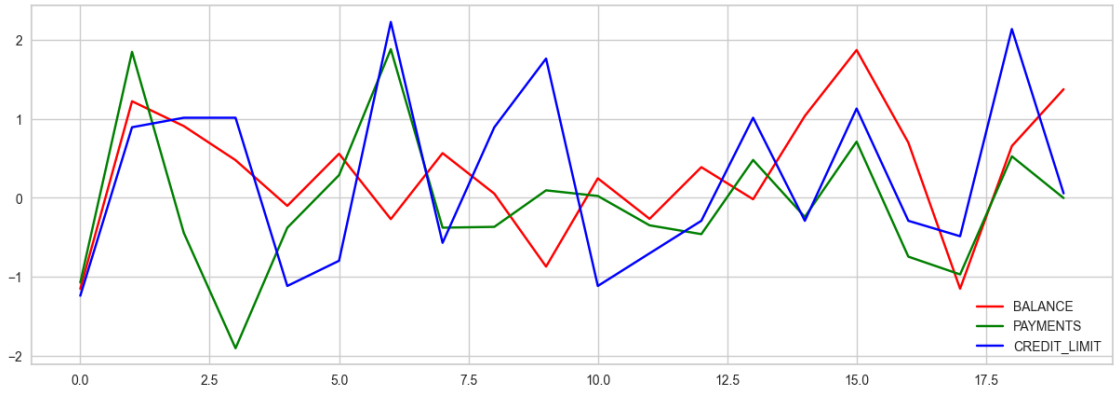
The graphs of before and after scaling the data are shown below.

[30]:
```
# Visualizing the before and after scaling graphs for features.
# Before
plt.figure(figsize=(15,5))
plt.plot(df5.BALANCE[:20], color='red', label='BALANCE')
plt.plot(df5.PAYMENTS[:20], color='green', label='PAYMENTS')
plt.plot(df5.CREDIT_LIMIT[:20], color='blue', label='CREDIT_LIMIT')
plt.legend(loc='best')
plt.show()
```



[31]:
```
# After
plt.figure(figsize=(15,5))
plt.plot(df_scaled.BALANCE[:20], color='red', label='BALANCE')
```

```
plt.plot(df_scaled.PAYMENTS[:20], color='green', label='PAYMENTS')
plt.plot(df_scaled.CREDIT_LIMIT[:20], color='blue', label='CREDIT_LIMIT')
plt.legend(loc='best')
plt.show()
```



1.2.5 Data Discretization

Feature discretization, also known as binning or discretization, is the process of transforming continuous numerical features into discrete intervals or bins. This helps to handle continuous data more effectively, simplify models, and improve their interpretability. In the clustering process, this allows us to create better clusters since clusters are created around centroids. - Equal-Width Binning(Fixed Width) → Divides the range of values into equally sized bins. - Equal-Frequency Binning(Fixed Frequency) → Divides the data into bins with approximately the same number of data points in each bin. - Custom Binning → Bins are defined based on domain knowledge or specific requirements. - Quantile Binning → Bins are created based on quantiles of the data. - Cluster-Based Binning → Uses clustering algorithms to group similar data points into bins. In our case, we will be using `KBinsDiscretizer` for data discretization.

[32]:
```
# Copy the data frame
df_scaled_discretization = df_scaled.copy()

# Create KBinsDiscretizer
kbins = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='uniform')
df_scaled_discretization[df_scaled_discretization.columns] = kbins.
 ↪fit_transform(df_scaled_discretization[df_scaled_discretization.columns])

# Check data
df_scaled_discretization.head()
```

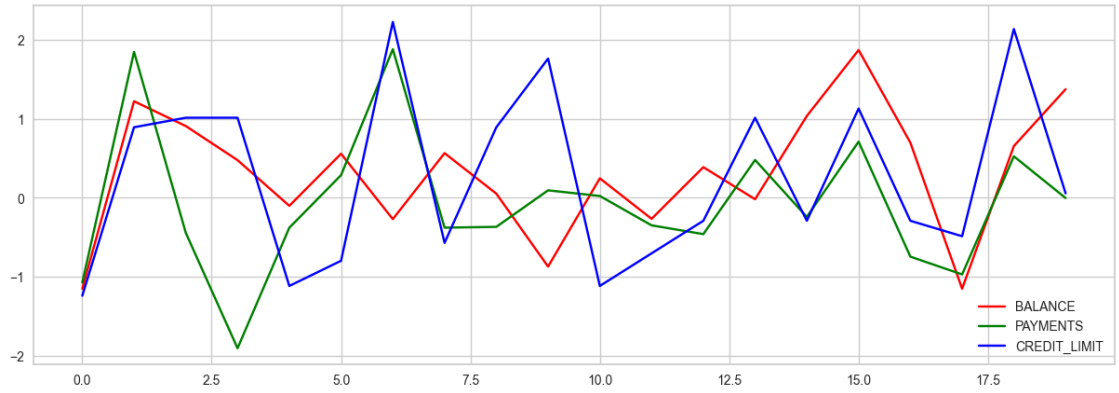
[32]:

	BALANCE	PURCHASES	PURCHASES_TRX	CREDIT_LIMIT	PAYMENTS	\
0	0.0	1.0	2.0	2.0	2.0	
1	8.0	0.0	0.0	6.0	9.0	
2	7.0	5.0	5.0	7.0	3.0	

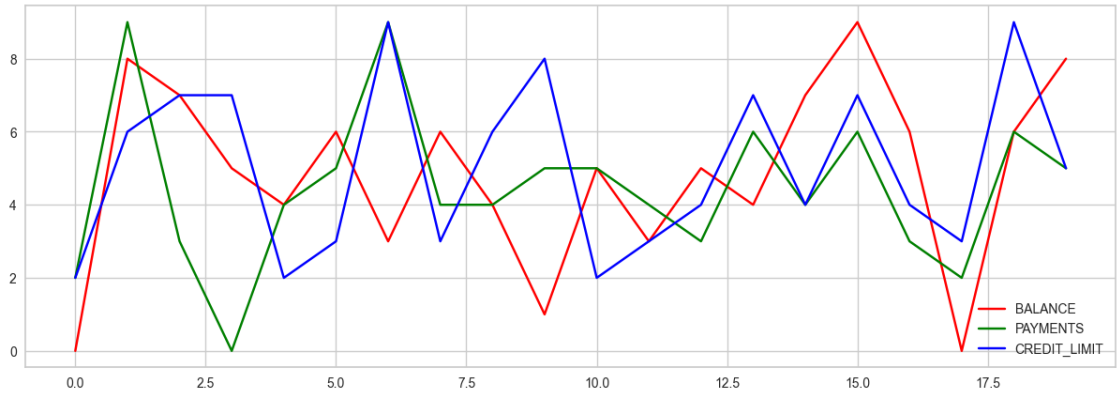
```
3      5.0        7.0            1.0           7.0       0.0
4      4.0        0.0            1.0           2.0       4.0

   MINIMUM_PAYMENTS  BALANCE_FREQUENCY  ONEOFF_PURCHASES  \
0               2.0                3.0               0.0
1               7.0                6.0               0.0
2               5.0                9.0               5.0
3               6.0                0.0               9.0
4               3.0                9.0               0.0

   INSTALLMENTS_PURCHASES  CASH_ADVANCE  PURCHASES_FREQUENCY  \
0                     0.0           0.0                  1.0
1                     0.0           9.0                  0.0
2                     0.0           0.0                  9.0
3                     0.0           0.0                  0.0
4                     0.0           0.0                  0.0

   ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
0                         0.0                               0.0
1                         0.0                               0.0
2                         9.0                               0.0
3                         1.0                               0.0
4                         1.0                               0.0

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PRC_FULL_PAYMENT  TENURE
0                     0.0               0.0               0.0     9.0
1                     4.0               4.0               6.0     9.0
2                     0.0               0.0               0.0     9.0
3                     1.0               1.0               0.0     9.0
4                     0.0               0.0               0.0     9.0
```

The graphs of before and after discretization the data are shown below.

[33]:
```
# Before
plt.figure(figsize=(15,5))
plt.plot(df_scaled.BALANCE[:20], color='red', label='BALANCE')
plt.plot(df_scaled.PAYMENTS[:20], color='green', label='PAYMENTS')
plt.plot(df_scaled.CREDIT_LIMIT[:20], color='blue', label='CREDIT_LIMIT')
plt.legend(loc='best')
plt.show()
```



[34]:
```
# After
plt.figure(figsize=(15,5))
plt.plot(df_scaled_discretization.BALANCE[:20], color='red', label='BALANCE')
plt.plot(df_scaled_discretization.PAYMENTS[:20], color='green',␣
 ↪label='PAYMENTS')
plt.plot(df_scaled_discretization.CREDIT_LIMIT[:20], color='blue',␣
 ↪label='CREDIT_LIMIT')
plt.legend(loc='best')
plt.show()
```



1.3 Perform Feature Engineering

1. Identify significant and independent features using appropriate techniques.
2. Show how you selected the features using suitable graphs.

1.3.1 Identifying Significant and Independent Variables

We can identify the significant features in the dataset by performing feature engineering. We are using the following feature engineering methods on our dataset.

1. PCA(Principal Component Analysis) → PCA is used for dimension reduction in the dataset. It is useful when dealing with dealing with datasets that have many highly correlating variables.
2. SVD(Single Value Decomposition) → SVD is a matrix factorization technique that decomposes a matrix into three simple matrices, ∪, ∨, and $\sum$. This also allows us to reduce dimensionality.

Step 1 - PCA(Principal Component Analysis) PCA is used for dimension reduction in the dataset. It is useful when dealing with dealing with datasets that have many highly correlating variables.

The following code is used to perform PCA.

[35]:
```
# Copy data frame
df_pca_scaled = df_scaled_discretization.copy()

# Create a PCA object
pca = PCA()

# Apply the transform to the dataset
df_pca_scaled[df_pca_scaled.columns] = pca.
 ↪fit_transform(df_pca_scaled[df_pca_scaled.columns])

# Shape of the df_pca_scaled
df_pca_scaled.shape
```

[35]:
```
(8950, 17)
```

[36]:
```
df_pca_scaled.head()
```

[36]:

	BALANCE	PURCHASES	PURCHASES_TRX	CREDIT_LIMIT	PAYMENTS	\
0	-5.424098	-8.235778	1.406150	-1.803849	1.893730	
1	-8.994678	4.366142	0.058750	5.392989	-2.671634	
2	4.212950	2.569625	6.414961	-5.414136	-0.756905	
3	-2.788429	-2.400597	8.369653	-1.003194	6.179858	
4	-6.045446	-3.826287	0.903521	-5.209828	-2.358177	

	MINIMUM_PAYMENTS	BALANCE_FREQUENCY	ONEOFF_PURCHASES	\
0	0.536220	-1.554429	-1.427039	
1	-5.811712	2.569535	0.660693	
2	1.810996	4.607691	-1.160287	
3	-1.592537	0.100046	1.138640	
4	-1.184562	-1.525047	0.148748	

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	\
0	0.240992	0.614772	0.425296	
1	-1.022840	2.640337	0.806268	
2	0.620563	-0.826130	-1.388430	

```
3               -0.713468     -5.267590             3.929801
4                0.244583      1.690520            -0.537904

   ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
0                    0.664810                          0.603416
1                    0.409229                         -1.224936
2                    2.102297                          0.212080
3                   -4.293793                         -0.269293
4                    0.795709                         -1.197224

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PRC_FULL_PAYMENT    TENURE
0                0.007705         -0.574741         -0.124425 -0.519374
1               -0.541644          0.090286         -0.559080 -0.143043
2               -1.491156          2.649925          0.265688 -0.686067
3               -0.118928          1.530771         -0.185934  0.506732
4               -0.460919         -0.570811          0.219043 -0.185949
```

Step 2 - SVD(Single Value Decomposition) SVD is a matrix factorization technique that decomposes a matrix into three simple matrices, $\cup$, $\vee$, and $\sum$. This also allows us to reduce dimensionality.

The following code is used to perform SVD. First, we get the optimal number of components and perform the SVD using them.

[37]:
```
# Copy the data frame
df_scaled_svd_tmp = df_scaled_discretization.copy()

# Make sparse matrix
X_sparse = csr_matrix(df_scaled_svd_tmp)

tsvd = TruncatedSVD(n_components=X_sparse.shape[1]-1)
X_tsvd = tsvd.fit(df_scaled_svd_tmp)

tsvd_var_ratios = tsvd.explained_variance_ratio_

def select_n_components(var_ratio, goal_var: float) -> int:
    # Set initial variance explained so far
    total_variance = 0.0

    # Set initial number of features
    n_components = 0

    # For the explained variance of each feature:
    for explained_variance in var_ratio:
        # Add the explained variance to the total
        total_variance += explained_variance
        n_components += 1
```

```
        # If we reach our goal level of explained variance
        if total_variance >= goal_var:
            break

    return n_components

number_of_components = select_n_components(tsvd_var_ratios, 0.95)
print("Number of components : ", number_of_components)
```

```
Number of components :  10
```

[38]:
```
# Copy the data frame
df_for_scaled_svd = df_pca_scaled.copy()

svd = TruncatedSVD(n_components=number_of_components)
svd.fit(df_for_scaled_svd)

# Apply transformations
transformed = svd.transform(df_for_scaled_svd)

# Get the df
df_svd_scaled = pd.DataFrame(transformed)

# View Data
df_svd_scaled.head()
```

[38]:

	0	1	2	3	4	5	6	7	8	9
0	-5.424098	-8.235778	1.406150	-1.803849	1.893730	0.536220	-1.554429	-1.427039	0.240992	0.614772
1	-8.994678	4.366142	0.058750	5.392989	-2.671634	-5.811712	2.569535	0.660693	-1.022840	2.640337
2	4.212950	2.569625	6.414961	-5.414136	-0.756905	1.810996	4.607691	-1.160287	0.620563	-0.826130
3	-2.788429	-2.400597	8.369653	-1.003194	6.179858	-1.592537	0.100046	1.138640	-0.713468	-5.267590
4	-6.045446	-3.826287	0.903521	-5.209828	-2.358177	-1.184562	-1.525047	0.148748	0.244583	1.690520

Step 3 - Correlation Values Next, we need to check for the correlation values. Because the lower the correlation values are, we can separate the data into unique clusters. The following code shows how you can get the correlation values.

[39]:
```
# Find the correlations in the df_for_scaled_svd data frame
df_for_scaled_svd.corr()
```

[39]:

	BALANCE	PURCHASES	PURCHASES_TRX
BALANCE	1.000000e+00	2.134524e-15	1.919957e-15
PURCHASES	2.134524e-15	1.000000e+00	5.930965e-16
PURCHASES_TRX	1.919957e-15	5.930965e-16	1.000000e+00
CREDIT_LIMIT	-2.459508e-16	2.981297e-15	-7.154199e-16
PAYMENTS	5.693911e-16	-5.740043e-16	-3.299517e-16
MINIMUM_PAYMENTS	-1.312757e-15	2.353705e-15	9.540805e-16
BALANCE_FREQUENCY	-9.442609e-16	-1.949660e-15	9.602371e-16
ONEOFF_PURCHASES	1.294693e-15	-1.399139e-15	-1.020802e-15
INSTALLMENTS_PURCHASES	-5.068382e-17	9.709217e-16	-4.150047e-16
CASH_ADVANCE	1.884719e-17	-9.368012e-17	3.109948e-16
PURCHASES_FREQUENCY	7.463796e-16	2.620828e-16	3.609075e-18
ONEOFF_PURCHASES_FREQUENCY	-7.373859e-16	-2.608142e-16	3.823294e-16
PURCHASES_INSTALLMENTS_FREQUENCY	8.528059e-16	1.676782e-15	-1.257466e-15
CASH_ADVANCE_FREQUENCY	-2.103946e-16	2.105484e-15	-1.028119e-16
CASH_ADVANCE_TRX	1.748859e-15	-2.172521e-16	-2.203427e-15
PRC_FULL_PAYMENT	6.644085e-17	-4.428475e-16	1.076116e-16
TENURE	1.240445e-15	4.113701e-17	-1.589134e-15

	CREDIT_LIMIT	PAYMENTS
BALANCE	-2.459508e-16	5.693911e-16
PURCHASES	2.981297e-15	-5.740043e-16
PURCHASES_TRX	-7.154199e-16	-3.299517e-16
CREDIT_LIMIT	1.000000e+00	-2.040439e-16
PAYMENTS	-2.040439e-16	1.000000e+00
MINIMUM_PAYMENTS	7.275274e-17	8.576690e-16
BALANCE_FREQUENCY	6.684454e-16	3.030743e-16
ONEOFF_PURCHASES	-2.114718e-16	1.124896e-16
INSTALLMENTS_PURCHASES	-1.773794e-16	2.215344e-16
CASH_ADVANCE	-3.153678e-16	1.113208e-16
PURCHASES_FREQUENCY	-7.155431e-16	7.098130e-17
ONEOFF_PURCHASES_FREQUENCY	-1.882144e-16	1.989070e-16
PURCHASES_INSTALLMENTS_FREQUENCY	3.676042e-16	4.321127e-16
CASH_ADVANCE_FREQUENCY	-2.331994e-16	-2.222103e-17
CASH_ADVANCE_TRX	-2.657758e-16	-1.962437e-16
PRC_FULL_PAYMENT	-2.506646e-17	1.852417e-16
TENURE	-3.980454e-17	1.448769e-16

	MINIMUM_PAYMENTS	BALANCE_FREQUENCY
BALANCE	-1.312757e-15	-9.442609e-16
PURCHASES	2.353705e-15	-1.949660e-15
PURCHASES_TRX	9.540805e-16	9.602371e-16
CREDIT_LIMIT	7.275274e-17	6.684454e-16
PAYMENTS	8.576690e-16	3.030743e-16
MINIMUM_PAYMENTS	1.000000e+00	-1.151033e-15
BALANCE_FREQUENCY	-1.151033e-15	1.000000e+00
ONEOFF_PURCHASES	1.088916e-15	1.255805e-15

```
INSTALLMENTS_PURCHASES                1.681419e-16       8.820032e-16
CASH_ADVANCE                         -1.318533e-15       6.229350e-16
PURCHASES_FREQUENCY                  -5.656507e-16       1.144192e-15
ONEOFF_PURCHASES_FREQUENCY           -1.197448e-15      -6.183697e-16
PURCHASES_INSTALLMENTS_FREQUENCY      3.270544e-17      -3.766122e-16
CASH_ADVANCE_FREQUENCY                3.066550e-16       1.282294e-15
CASH_ADVANCE_TRX                      1.169334e-15       1.716539e-15
PRC_FULL_PAYMENT                     -1.495256e-16       2.065743e-16
TENURE                                1.021825e-16       2.002564e-16

                                  ONEOFF_PURCHASES  INSTALLMENTS_PURCHASES  \
BALANCE                              1.294693e-15           -5.068382e-17
PURCHASES                           -1.399139e-15            9.709217e-16
PURCHASES_TRX                       -1.020802e-15           -4.150047e-16
CREDIT_LIMIT                        -2.114718e-16           -1.773794e-16
PAYMENTS                             1.124896e-16            2.215344e-16
MINIMUM_PAYMENTS                     1.088916e-15            1.681419e-16
BALANCE_FREQUENCY                    1.255805e-15            8.820032e-16
ONEOFF_PURCHASES                     1.000000e+00            9.488112e-16
INSTALLMENTS_PURCHASES               9.488112e-16            1.000000e+00
CASH_ADVANCE                         1.002577e-15            6.413976e-16
PURCHASES_FREQUENCY                 -1.319378e-15           -1.650258e-16
ONEOFF_PURCHASES_FREQUENCY           3.445513e-16            4.586687e-16
PURCHASES_INSTALLMENTS_FREQUENCY     7.607138e-16           -1.626507e-16
CASH_ADVANCE_FREQUENCY               1.934126e-16           -9.550783e-16
CASH_ADVANCE_TRX                    -1.919482e-15           -2.699901e-16
PRC_FULL_PAYMENT                     7.909209e-17            2.518357e-16
TENURE                              -4.945931e-16            2.159899e-16

                                   CASH_ADVANCE  PURCHASES_FREQUENCY  \
BALANCE                            1.884719e-17         7.463796e-16
PURCHASES                         -9.368012e-17         2.620828e-16
PURCHASES_TRX                      3.109948e-16         3.609075e-18
CREDIT_LIMIT                      -3.153678e-16        -7.155431e-16
PAYMENTS                           1.113208e-16         7.098130e-17
MINIMUM_PAYMENTS                  -1.318533e-15        -5.656507e-16
BALANCE_FREQUENCY                  6.229350e-16         1.144192e-15
ONEOFF_PURCHASES                   1.002577e-15        -1.319378e-15
INSTALLMENTS_PURCHASES             6.413976e-16        -1.650258e-16
CASH_ADVANCE                       1.000000e+00         1.758002e-15
PURCHASES_FREQUENCY                1.758002e-15         1.000000e+00
ONEOFF_PURCHASES_FREQUENCY         9.130367e-16         1.360117e-15
PURCHASES_INSTALLMENTS_FREQUENCY -7.375994e-16        -9.726898e-19
CASH_ADVANCE_FREQUENCY             2.964986e-16        -1.856248e-16
CASH_ADVANCE_TRX                   1.558396e-16        -4.865915e-16
PRC_FULL_PAYMENT                   3.448697e-16         4.434009e-16
TENURE                             6.720388e-17         2.725732e-16
```

	ONEOFF_PURCHASES_FREQUENCY
BALANCE	-7.373859e-16
PURCHASES	-2.608142e-16
PURCHASES_TRX	3.823294e-16
CREDIT_LIMIT	-1.882144e-16
PAYMENTS	1.989070e-16
MINIMUM_PAYMENTS	-1.197448e-15
BALANCE_FREQUENCY	-6.183697e-16
ONEOFF_PURCHASES	3.445513e-16
INSTALLMENTS_PURCHASES	4.586687e-16
CASH_ADVANCE	9.130367e-16
PURCHASES_FREQUENCY	1.360117e-15
ONEOFF_PURCHASES_FREQUENCY	1.000000e+00
PURCHASES_INSTALLMENTS_FREQUENCY	5.220099e-16
CASH_ADVANCE_FREQUENCY	-1.726873e-16
CASH_ADVANCE_TRX	3.055484e-16
PRC_FULL_PAYMENT	1.005387e-16
TENURE	8.599551e-16

	PURCHASES_INSTALLMENTS_FREQUENCY
BALANCE	8.528059e-16
PURCHASES	1.676782e-15
PURCHASES_TRX	-1.257466e-15
CREDIT_LIMIT	3.676042e-16
PAYMENTS	4.321127e-16
MINIMUM_PAYMENTS	3.270544e-17
BALANCE_FREQUENCY	-3.766122e-16
ONEOFF_PURCHASES	7.607138e-16
INSTALLMENTS_PURCHASES	-1.626507e-16
CASH_ADVANCE	-7.375994e-16
PURCHASES_FREQUENCY	-9.726898e-19
ONEOFF_PURCHASES_FREQUENCY	5.220099e-16
PURCHASES_INSTALLMENTS_FREQUENCY	1.000000e+00
CASH_ADVANCE_FREQUENCY	-3.562126e-16
CASH_ADVANCE_TRX	-7.265584e-16
PRC_FULL_PAYMENT	1.181659e-16
TENURE	7.829731e-17

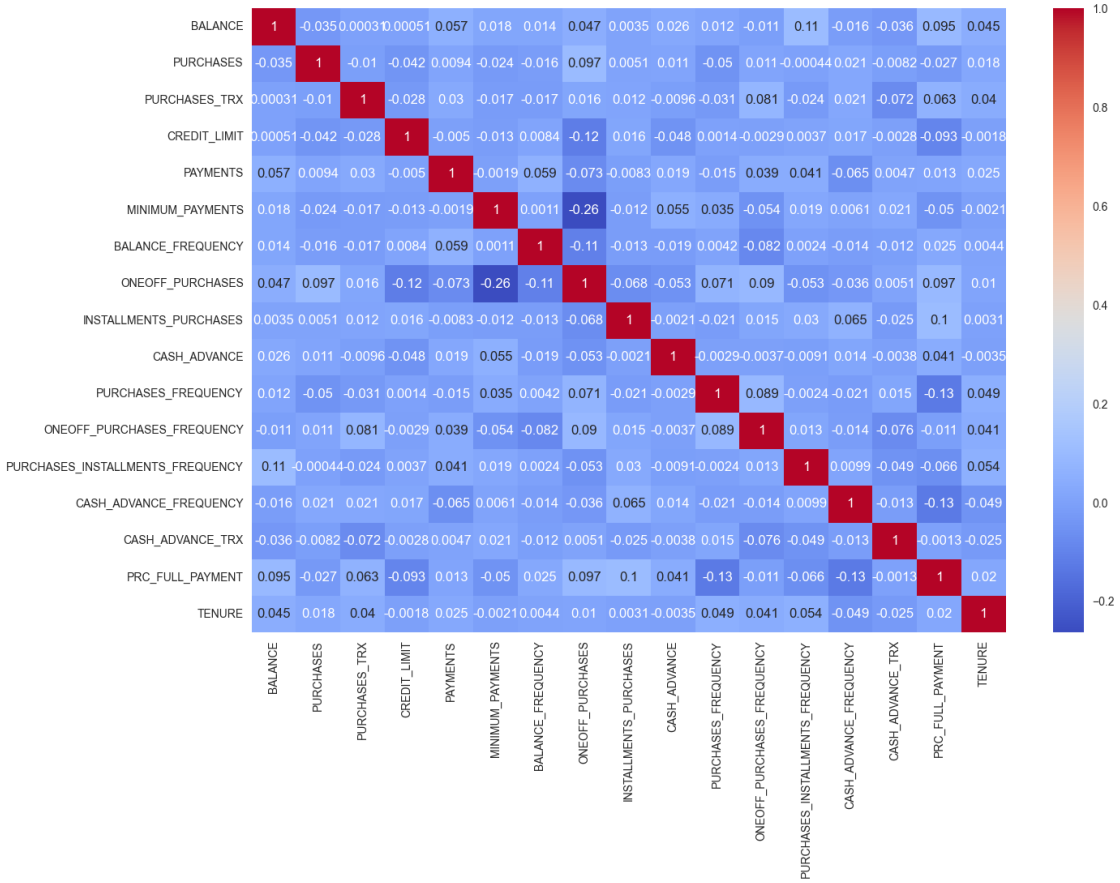
	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX
BALANCE	-2.103946e-16	1.748859e-15
PURCHASES	2.105484e-15	-2.172521e-16
PURCHASES_TRX	-1.028119e-16	-2.203427e-15
CREDIT_LIMIT	-2.331994e-16	-2.657758e-16
PAYMENTS	-2.222103e-17	-1.962437e-16
MINIMUM_PAYMENTS	3.066550e-16	1.169334e-15
BALANCE_FREQUENCY	1.282294e-15	1.716539e-15

```
ONEOFF_PURCHASES                      1.934126e-16     -1.919482e-15
INSTALLMENTS_PURCHASES               -9.550783e-16     -2.699901e-16
CASH_ADVANCE                          2.964986e-16      1.558396e-16
PURCHASES_FREQUENCY                  -1.856248e-16     -4.865915e-16
ONEOFF_PURCHASES_FREQUENCY           -1.726873e-16      3.055484e-16
PURCHASES_INSTALLMENTS_FREQUENCY     -3.562126e-16     -7.265584e-16
CASH_ADVANCE_FREQUENCY                1.000000e+00     -4.921966e-16
CASH_ADVANCE_TRX                     -4.921966e-16      1.000000e+00
PRC_FULL_PAYMENT                      2.765778e-16      1.188726e-16
TENURE                               -2.861441e-16     -8.525290e-16

                                  PRC_FULL_PAYMENT        TENURE
BALANCE                               6.644085e-17  1.240445e-15
PURCHASES                            -4.428475e-16  4.113701e-17
PURCHASES_TRX                         1.076116e-16 -1.589134e-15
CREDIT_LIMIT                         -2.506646e-17 -3.980454e-17
PAYMENTS                              1.852417e-16  1.448769e-16
MINIMUM_PAYMENTS                     -1.495256e-16  1.021825e-16
BALANCE_FREQUENCY                     2.065743e-16  2.002564e-16
ONEOFF_PURCHASES                      7.909209e-17 -4.945931e-16
INSTALLMENTS_PURCHASES                2.518357e-16  2.159899e-16
CASH_ADVANCE                          3.448697e-16  6.720388e-17
PURCHASES_FREQUENCY                   4.434009e-16  2.725732e-16
ONEOFF_PURCHASES_FREQUENCY            1.005387e-16  8.599551e-16
PURCHASES_INSTALLMENTS_FREQUENCY      1.181659e-16  7.829731e-17
CASH_ADVANCE_FREQUENCY                2.765778e-16 -2.861441e-16
CASH_ADVANCE_TRX                      1.188726e-16 -8.525290e-16
PRC_FULL_PAYMENT                      1.000000e+00  1.062697e-16
TENURE                                1.062697e-16  1.000000e+00
```

To identify the correlation values better we can get a heatmap.

[40]:
```
# Create heatmap
f, ax = plt.subplots(figsize=(15, 10))
sns.heatmap(df_for_scaled_svd.corr(method='spearman'), annot=True,␣
 ↪cmap='coolwarm')
```

[40]:
```
<Axes: >
```



Since all the features in the dataset have low correlation values with each other, we can determine all of the features are independent variables and they can be used for the clustering algorithms. We cannot determine significant variables in this case, since we do not have a `y` variable for clustering.

1.4 Clustering

1. Justifies the choice of clustering algorithm for the dataset.
2. Consider and apply alternative algorithms to the dataset and explain why they were chosen.
3. Using suitable evaluation matrices, compare the applicability of different clustering algorithms on the given dataset.
4. Relates the clusters to the original problem and provides actionable insights.

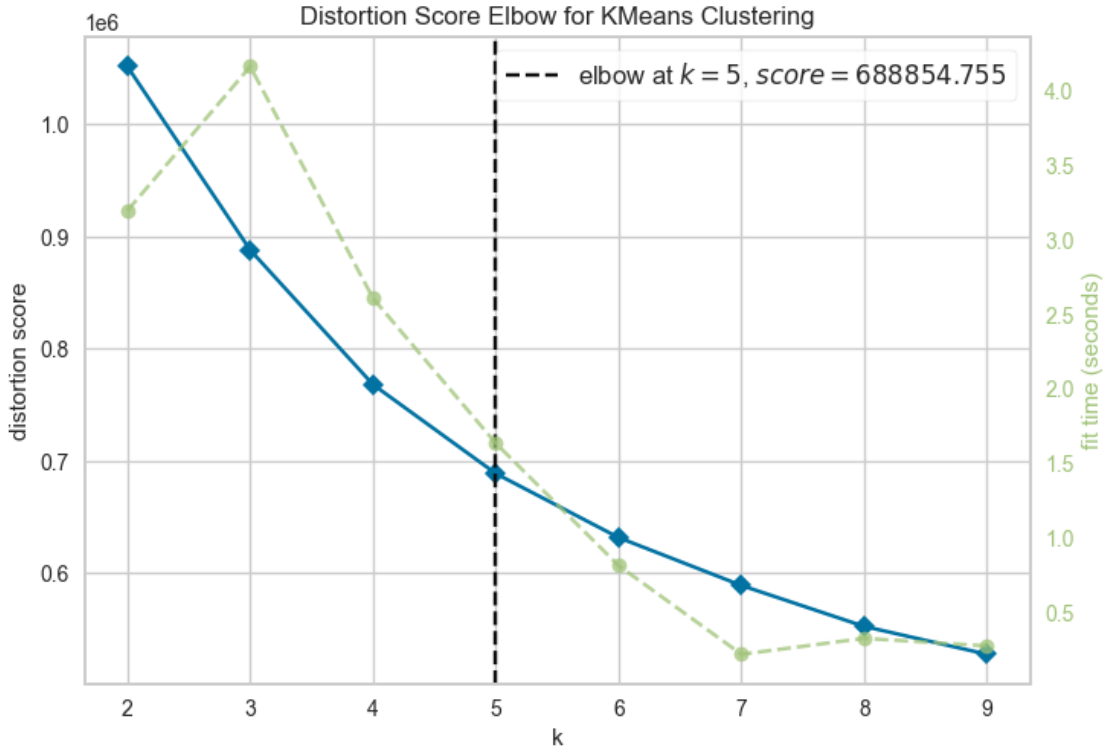
1.4.1 Chosen Clustering Algorithm for the Dataset

The Clustering algorithm that was chosen for this is the `KMeans Clustering` Algorithm. The reasons for choosing that algorithm for this dataset are mentioned below. - KMeans is a direct and easy algorithm to implement which provides good results since it assigns each data point to the nearest centeroid, updating the centeroids to minimize the sum of the squared distances. - KMeans is also efficient and can handle large datasets with many variables due to its linear time complexity. - KMeans can work with a variety of data types and distribution types, which makes

it a versatile algorithm. - The clusters formed by KMeans are easy to interpret since each cluster is associated with a centroid which represents the average of all the points that are assigned to that cluster. - KMeans is widely used in segmentation processes like image segmentation and customer segmentation.

Step 1 - Elbow Method First, we need to use the KMeans algorithm. The following code explains how you can find the optimal number of clusters for `KMeans` using the `Elbow Method`(using the `Yellowbrick` library).

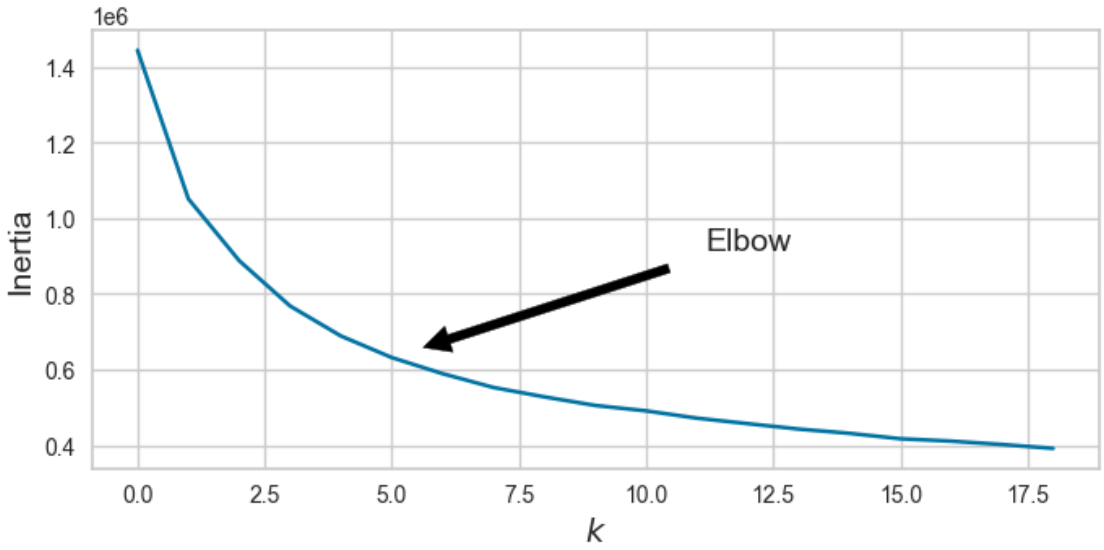
[41]:
```
# Select the model
model = KMeans(random_state=42)

# Selet the KElbowVisualizer from Yellowbrick library
visualizer = KElbowVisualizer(model, k=(2,10))

visualizer.fit(df_pca_scaled)
visualizer.show()
plt.show()
```



And if we need to plot `Elbow Curve` without using the Yellowbrick library, we can do that as well. The following code explains how you can convert the data frame to a `numpy` array and use it to get the Elbow Curve. The place where the curve tends to form an Elbow joint contains the optimal number of clusters.

[42]:
```
# Transform the data to a NumPy array
X = df_pca_scaled.to_numpy()
# Get cluster-wise-inertias
kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X)
                for k in range(1, 20)]
inertias = [model.inertia_ for model in kmeans_per_k]
```

[43]:
```
# Plot the Elbow curve
plt.figure(figsize=(8, 3.5))
plt.plot(inertias, "bx-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Inertia", fontsize=14)
plt.annotate('Elbow',
             xy=(5, inertias[5]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=14,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )
plt.show()
```



After selecting the optimal number of clusters(in my case its 5) we can fit and predict it to get the associated labels in the dataset.

[44]:
```
# KMeans
num_of_clusters = 5
kmeans = KMeans(num_of_clusters)
y_pred = kmeans.fit(X)
```

```
labels = kmeans.labels_
```

[45]:
```
labels
```

[45]:
```
array([3, 2, 0, …, 3, 3, 3], dtype=int32)
```

[46]:
```
labels.shape
```

[46]:
```
(8950,)
```

Let's examine the cluster centers/centroids as well.

[47]:
```
kmeans.cluster_centers_
```

[47]:
```
array([[ 9.57218899e+00,  4.05211070e+00,  3.70736709e+00,
        -6.56871773e-02, -1.54058816e-01,  4.69300268e-01,
        -1.44914891e-02,  1.05066178e-03,  1.99307163e-02,
        -8.93658748e-03, -2.24390327e-02,  1.20438509e-01,
        -3.83117901e-01,  1.78951248e-01, -6.19573260e-03,
         1.30263240e-02, -1.36109623e-01],
       [ 5.89151698e+00, -2.34427017e+00, -5.01705475e+00,
        -3.45159834e-01, -2.00407534e-01,  1.46215979e-01,
         2.24053270e-01, -1.05644381e-02, -1.65742698e-03,
         2.43390561e-01,  2.30505107e-02, -1.98776821e-01,
         3.93162671e-01, -8.67125179e-02,  1.66373875e-01,
        -1.07996306e-02,  2.53091798e-02],
       [-8.09075330e+00,  6.80402025e+00, -9.89098938e-01,
         2.45046025e+00, -2.69463295e-01,  2.10927145e-01,
         2.82849893e-01, -1.17191880e-01,  1.97881275e-01,
         5.36538109e-02,  1.00383888e-01, -5.56741888e-03,
         1.05761389e-01, -1.55483042e-02,  1.91191649e-03,
         4.51568143e-02,  3.28467363e-02],
       [-2.93713913e+00, -7.21318707e+00,  2.13315847e+00,
         2.67265891e+00,  2.54232254e+00, -4.05161935e-01,
         1.91014190e-01, -3.27933287e-01, -2.69240066e-01,
        -2.51653428e-01, -6.21192643e-02,  2.10830294e-02,
        -2.82842613e-02,  2.52267012e-02, -5.58936439e-02,
        -3.70059057e-02,  3.10550806e-02],
       [-5.07532936e+00, -1.19613123e+00,  6.70902235e-01,
        -3.75563308e+00, -1.47027198e+00, -4.05439306e-01,
        -5.87296633e-01,  3.66465741e-01,  3.48375799e-02,
        -6.99497309e-02, -3.57750980e-02,  7.25540403e-02,
        -1.04725299e-01, -8.15658652e-02, -1.11361278e-01,
        -8.93250329e-03,  4.37797817e-02]])
```

We can further analyze the centers to extract more information as well.

[48]:
```
kmeans.cluster_centers_.shape
```

[48]:
```
(5, 17)
```

[49]:
```
cluster_centers = pd.DataFrame(data=kmeans.cluster_centers_,␣
↪columns=[df_pca_scaled.columns])
cluster_centers
```

[49]:

	BALANCE	PURCHASES	PURCHASES_TRX	CREDIT_LIMIT	PAYMENTS	MINIMUM_PAYMENTS
0	9.572189	4.052111	3.707367	-0.065687	-0.154059	0.469300
1	5.891517	-2.344270	-5.017055	-0.345160	-0.200408	0.146216
2	-8.090753	6.804020	-0.989099	2.450460	-0.269463	0.210927
3	-2.937139	-7.213187	2.133158	2.672659	2.542323	-0.405162
4	-5.075329	-1.196131	0.670902	-3.755633	-1.470272	-0.405439

	BALANCE_FREQUENCY	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE
0	-0.014491	0.001051	0.019931	-0.008937
1	0.224053	-0.010564	-0.001657	0.243391
2	0.282850	-0.117192	0.197881	0.053654
3	0.191014	-0.327933	-0.269240	-0.251653
4	-0.587297	0.366466	0.034838	-0.069950

	PURCHASES_FREQUENCY	ONEOFF_PURCHASES_FREQUENCY
0	-0.022439	0.120439
1	0.023051	-0.198777
2	0.100384	-0.005567
3	-0.062119	0.021083
4	-0.035775	0.072554

	PURCHASES_INSTALLMENTS_FREQUENCY	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX
0	-0.383118	0.178951	-0.006196
1	0.393163	-0.086713	0.166374
2	0.105761	-0.015548	0.001912
3	-0.028284	0.025227	-0.055894
4	-0.104725	-0.081566	-0.111361

	PRC_FULL_PAYMENT	TENURE
0	0.013026	-0.136110
1	-0.010800	0.025309
2	0.045157	0.032847
3	-0.037006	0.031055
4	-0.008933	0.043780

Now, we can predict the KMeans values by fitting it to the KMeans algorithm we have created.

[50]:
```
y_kmeans = kmeans.fit_predict(X)
y_kmeans
```

[50]:
```
array([0, 1, 3, …, 0, 0, 0], dtype=int32)
```

[51]:
```
df_cluster = pd.concat([df_pca_scaled, pd.DataFrame(labels,␣
 ↪columns=['clusterKMeans'])],axis=1)
df_cluster.head()
```

[51]:

	BALANCE	PURCHASES	PURCHASES_TRX	CREDIT_LIMIT	PAYMENTS
0	-5.424098	-8.235778	1.406150	-1.803849	1.893730
1	-8.994678	4.366142	0.058750	5.392989	-2.671634
2	4.212950	2.569625	6.414961	-5.414136	-0.756905
3	-2.788429	-2.400597	8.369653	-1.003194	6.179858
4	-6.045446	-3.826287	0.903521	-5.209828	-2.358177

	MINIMUM_PAYMENTS	BALANCE_FREQUENCY	ONEOFF_PURCHASES
0	0.536220	-1.554429	-1.427039
1	-5.811712	2.569535	0.660693
2	1.810996	4.607691	-1.160287
3	-1.592537	0.100046	1.138640
4	-1.184562	-1.525047	0.148748

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY
0	0.240992	0.614772	0.425296
1	-1.022840	2.640337	0.806268
2	0.620563	-0.826130	-1.388430
3	-0.713468	-5.267590	3.929801
4	0.244583	1.690520	-0.537904

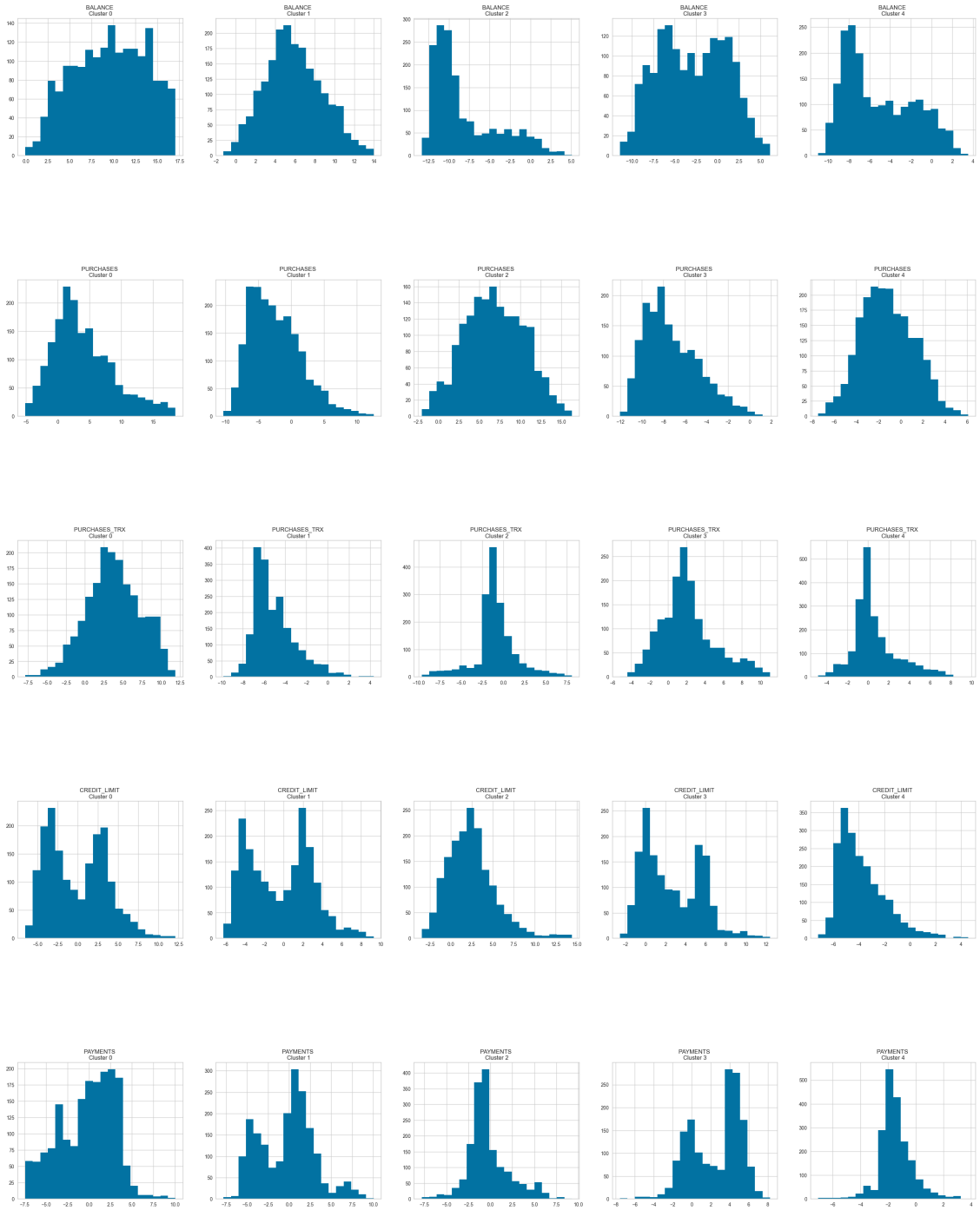
	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY
0	0.664810	0.603416
1	0.409229	-1.224936
2	2.102297	0.212080
3	-4.293793	-0.269293
4	0.795709	-1.197224

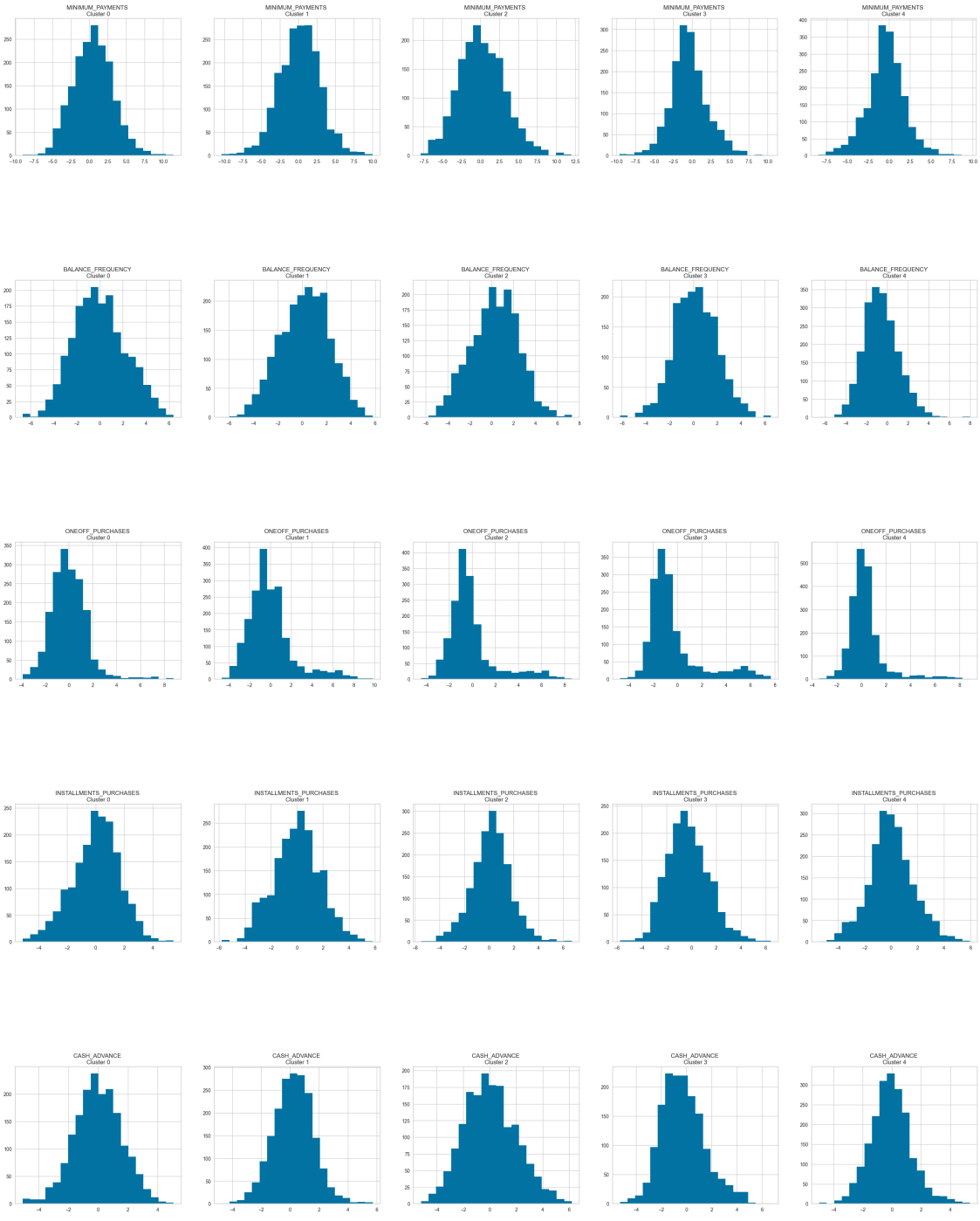
	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX	PRC_FULL_PAYMENT	TENURE
0	0.007705	-0.574741	-0.124425	-0.519374
1	-0.541644	0.090286	-0.559080	-0.143043
2	-1.491156	2.649925	0.265688	-0.686067
3	-0.118928	1.530771	-0.185934	0.506732
4	-0.460919	-0.570811	0.219043	-0.185949

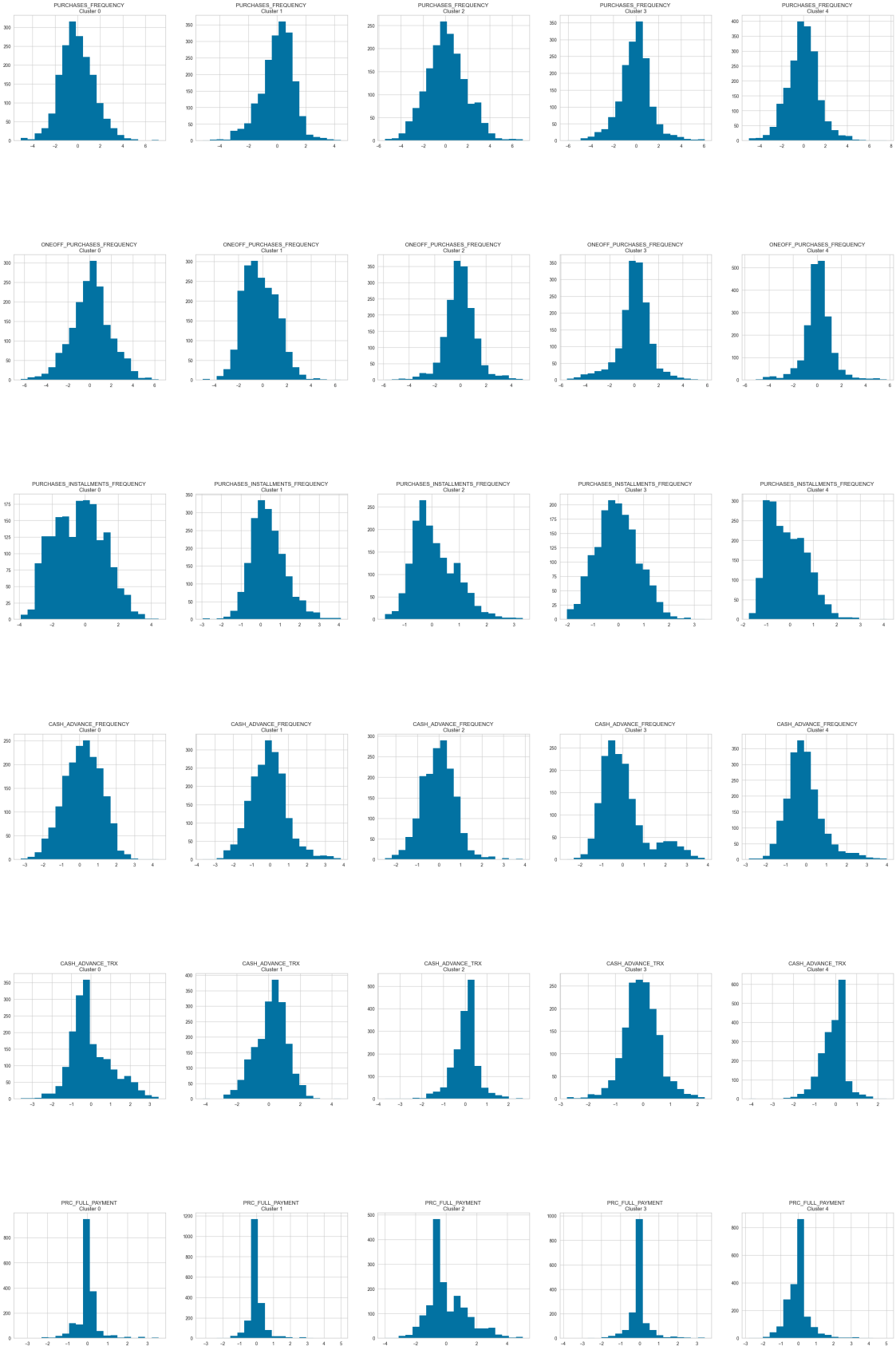
	clusterKMeans
0	3
1	2
2	0
3	3
4	4

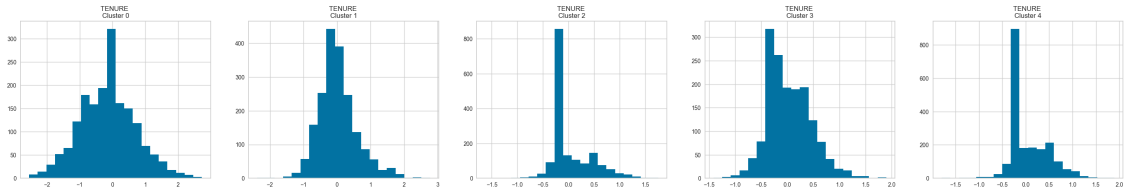
Let's plot the graphs to see the predictions more clearly.

[52]:
```
for i in df_pca_scaled:
    plt.figure(figsize=(35,5))
    for j in range(5):
        plt.subplot(1,5,j+1)
        cluster = df_cluster[df_cluster['clusterKMeans']==j]
        cluster[i].hist(bins=20)
        plt.title(f'{i} \nCluster {j}')
    plt.show()
```




MINIMUM_PAYMENTS
Cluster 0
MINIMUM_PAYMENTS
Cluster 1
MINIMUM_PAYMENTS
Cluster 2
MINIMUM_PAYMENTS
Cluster 3
MINIMUM_PAYMENTS
Cluster 4
BALANCE_FREQUENCY
Cluster 0
BALANCE_FREQUENCY
Cluster 1
BALANCE_FREQUENCY
Cluster 2
BALANCE_FREQUENCY
Cluster 3
BALANCE_FREQUENCY
Cluster 4
ONEOFF_PURCHASES
Cluster 0
ONEOFF_PURCHASES
Cluster 1
ONEOFF_PURCHASES
Cluster 2
ONEOFF_PURCHASES
Cluster 3
ONEOFF_PURCHASES
Cluster 4
INSTALLMENTS_PURCHASES
Cluster 0
INSTALLMENTS_PURCHASES
Cluster 1
INSTALLMENTS_PURCHASES
Cluster 2
INSTALLMENTS_PURCHASES
Cluster 3
INSTALLMENTS_PURCHASES
Cluster 4
CASH_ADVANCE
Cluster 0
CASH_ADVANCE
Cluster 1
CASH_ADVANCE
Cluster 2
CASH_ADVANCE
Cluster 3
CASH_ADVANCE
Cluster 4


PURCHASES_FREQUENCY
Cluster 0
PURCHASES_FREQUENCY
Cluster 1
PURCHASES_FREQUENCY
Cluster 2
PURCHASES_FREQUENCY
Cluster 3
PURCHASES_FREQUENCY
Cluster 4
ONEOFF_PURCHASES_FREQUENCY
Cluster 0
ONEOFF_PURCHASES_FREQUENCY
Cluster 1
ONEOFF_PURCHASES_FREQUENCY
Cluster 2
ONEOFF_PURCHASES_FREQUENCY
Cluster 3
ONEOFF_PURCHASES_FREQUENCY
Cluster 4
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 0
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 1
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 2
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 3
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 4
CASH_ADVANCE_FREQUENCY
Cluster 0
CASH_ADVANCE_FREQUENCY
Cluster 1
CASH_ADVANCE_FREQUENCY
Cluster 2
CASH_ADVANCE_FREQUENCY
Cluster 3
CASH_ADVANCE_FREQUENCY
Cluster 4
CASH_ADVANCE_TRX
Cluster 0
CASH_ADVANCE_TRX
Cluster 1
CASH_ADVANCE_TRX
Cluster 2
CASH_ADVANCE_TRX
Cluster 3
CASH_ADVANCE_TRX
Cluster 4
PRC_FULL_PAYMENT
Cluster 0
PRC_FULL_PAYMENT
Cluster 1
PRC_FULL_PAYMENT
Cluster 2
PRC_FULL_PAYMENT
Cluster 3
PRC_FULL_PAYMENT
Cluster 4

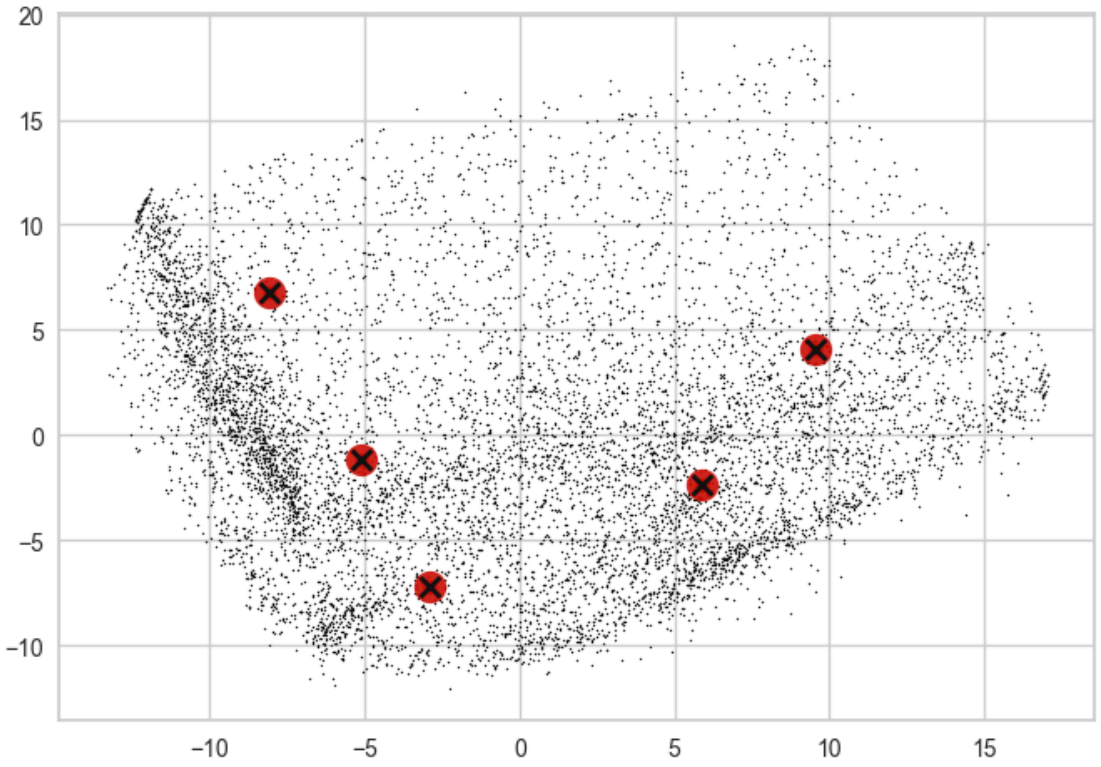


We can also check the centeroids and the data distribution to get an idea about the clustering as well.

[53]:
```
def plot_data(X):
    plt.plot(X[:,0], X[:,1], 'k.', markersize=2)
```

[54]:
```
def plot_centroids(centroids, weights=None, circle_color='r', cross_color='k'):
    if weights is not None:
        centroids = centroids[weights > weights.max() / 10]
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='o', s=35, linewidths=8,
                color=circle_color, zorder=10, alpha=0.9)
    plt.scatter(centroids[:, 0], centroids[:, 1],
                marker='x', s=2, linewidths=12,
                color=cross_color, zorder=11, alpha=1)
```

[55]:
```
plot_data(X)
plot_centroids(kmeans.cluster_centers_)
```



Step 2 - Silhouette Score Now we can get the KMeans Score to find out the `Silhouette Score`. `Silhouette Score`is a measure used to evaluate a clustering algorithm by assessing how well each object lies in the clusters. The Silhouette Score ranges from -1 to 1, where, - Score 1 indicates that the object is well matched to its own cluster and poorly matched to the neighboring clusters. - Score 0 indicates that the object is on the boundary between two clusters. - Score -1 indicates that the object is poorly assigned to the wrong cluster. To get the `Silhouette Score` we can use the following code snippet.

KMeans Score for Clusters = 5

[56]:
```
kmeans.score(X)
```

[56]:
```
-688854.9198279636
```

Silhouette Score for Clusters = 5

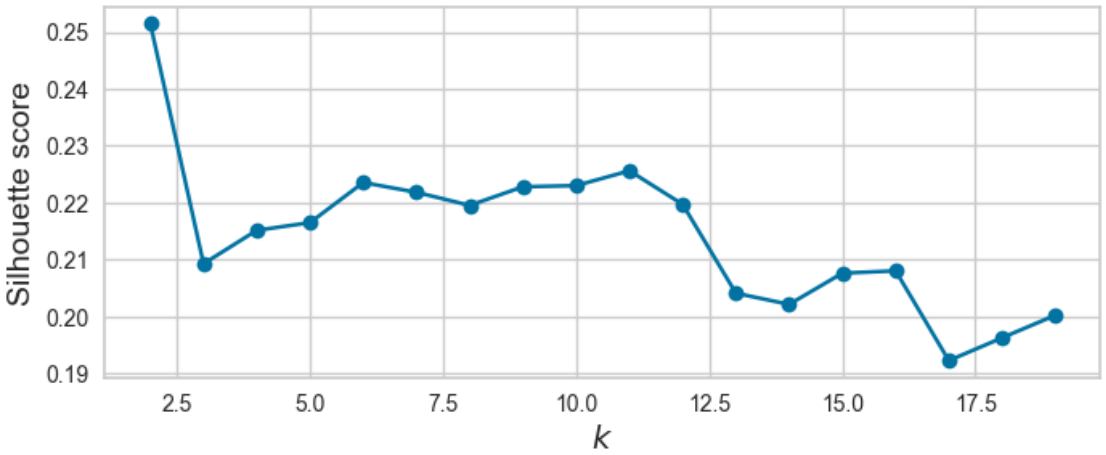
[57]:
```
silhouette_score(X, kmeans.labels_)
```

[57]:
```
0.2165475102195835
```

We are plotting the `Silhouette Score` for the KMeans clusters 1 to 20 which we tested earlier using the `Elbow Method` to find the optimal number of clusters.

[58]:
```
silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]
```

[59]:
```
plt.figure(figsize=(8, 3))
plt.plot(range(2, 20), silhouette_scores, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Silhouette score", fontsize=14)
plt.show()
```

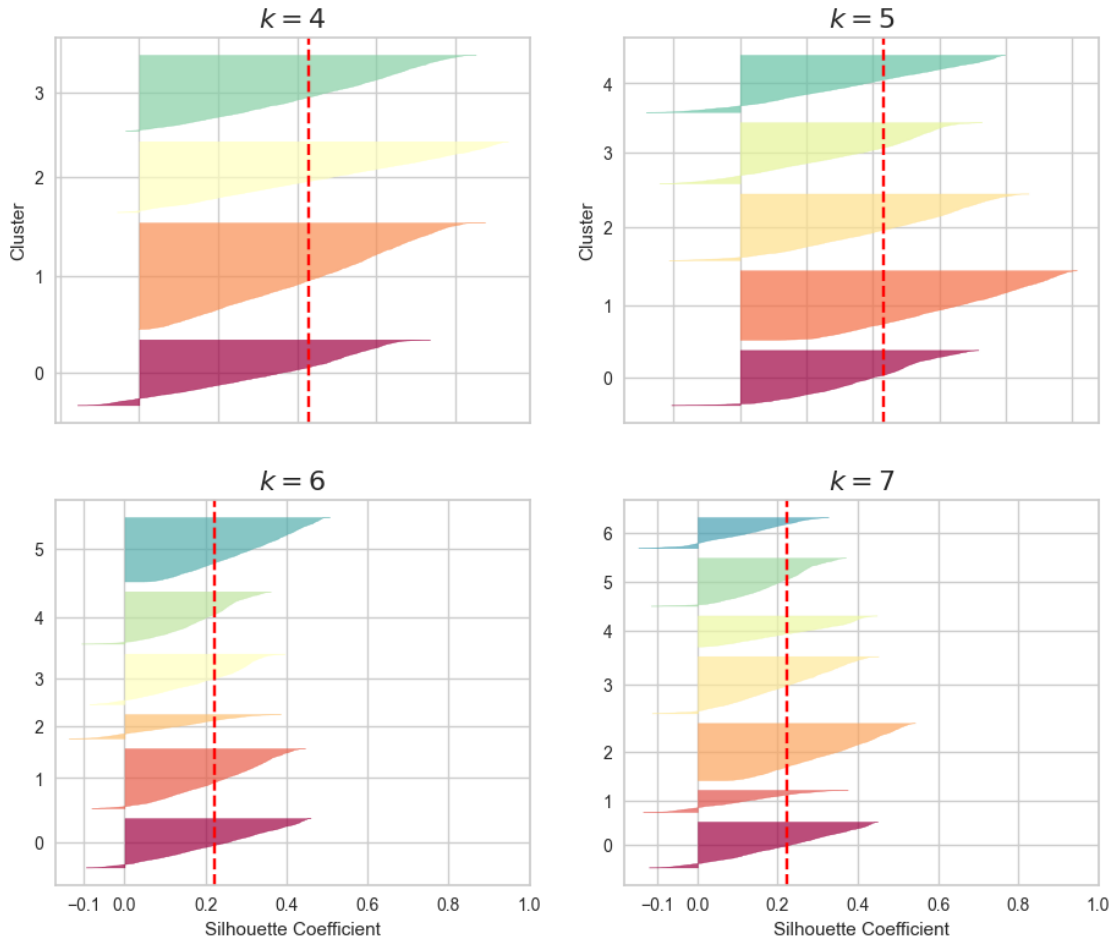


From the graph, we can see, that the clusters 1, and 5-11 have high `Silhouette Scores > 0`. Therefore, we can select the optimal number of clusters as 5, since it agrees to both `Elbow Method` and the `Silhouette Scores`.

Since the above graph does not show the variations of `Silhoutte Scores` well, we can use the following code to create graphs and find the optimal clusters and the Silhoutte Score values. This graph is called the `Silhouette Coefficient` graph. The wider the values, better the `Silhouette Score` is. The line shows the mean `Silhouette Coefficients`.

[60]:
```
plt.figure(figsize=(11, 9))

for k in (4, 5, 6, 7):
    plt.subplot(2, 2, k - 3)

    y_pred = kmeans_per_k[k - 1].labels_
    silhouette_coefficients = silhouette_samples(X, y_pred)

    padding = len(X) // 30
    pos = padding
    ticks = []
    for i in range(k):
        coeffs = silhouette_coefficients[y_pred == i]
```

```
        coeffs.sort()

        color = mpl.cm.Spectral(i / k)
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
    plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
    if k in (4,5):
        plt.ylabel("Cluster")

    if k in (6,7):
        plt.gca().set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
        plt.xlabel("Silhouette Coefficient")
    else:
        plt.tick_params(labelbottom=False)

    plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
    plt.title("$k={}$".format(k), fontsize=16)

plt.show()
```



Therefore, the number of clusters we can select for the KMeans algorithm is 5.

1.5 Alternative Clustering Algorithms

Rather than using the KMeans algorithm, we can use other algorithms for clustering as well. The two of the clustering alternative clustering algorithms that were used in this project are,

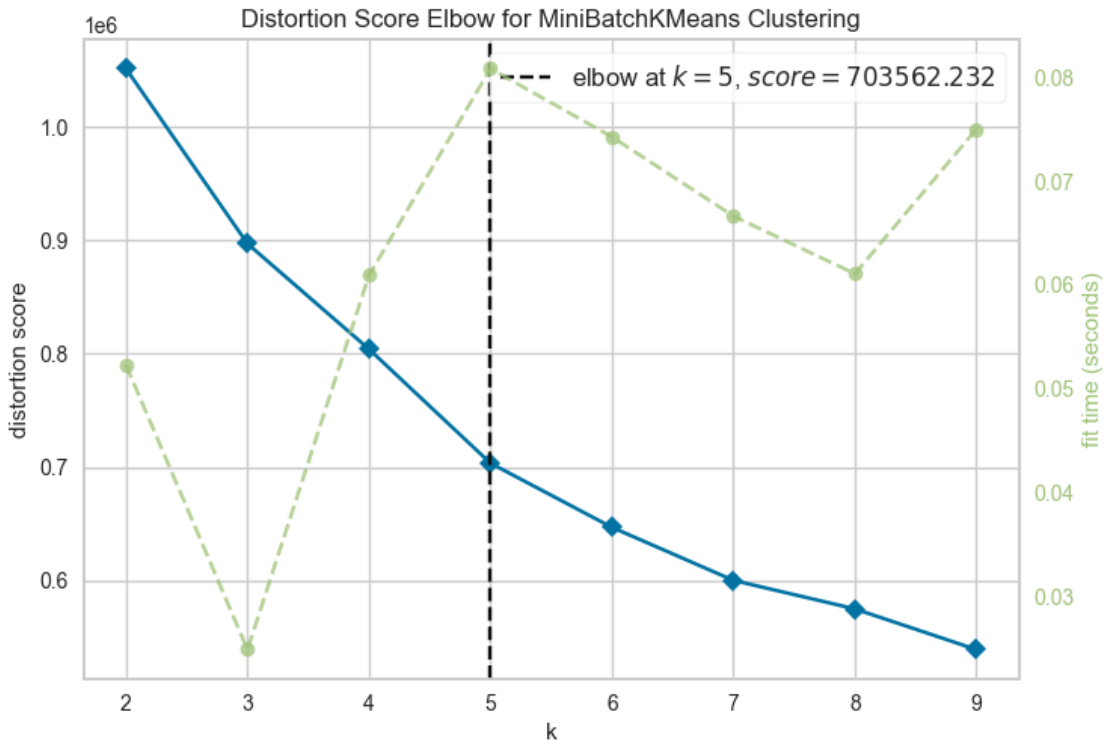
- **Mini-batch KMeans** → This is a variation of the traditional KMeans algorithm that is designed to work with large datasets. Instead of using the entire dataset to update the centroids, it uses random subsets or mini-batches of the data. The advantages of the Mini-batch KMeans are mentioned below.
  1. Efficiency → This is faster and more scalable than the traditional KMeans algorithm, making it optimal for large datasets.
  2. Less Memory Usage → Since it works with mini-batches of data it uses less memory.
  3. Good Results → In spite of its stochastic nature, it provides similar results to KMeans.
- **Density-Based Spatial Clusterings of Applications with Noise(DBScan)** → This is a density-based clustering algorithm that divides datasets into clusters based on density. The points in high-density are considered core points and those points are used for clustering. The advantages of DBSCAN are mentioned below.

1. Robust to noise → This can effectively identify outliers and noise points in the dataset.
2. No need to specify the number of clusters → Unlike KMeans. DBSCAN does not need to specify the number of clusters in advance.
3. Handles Arbitrary Cluster Shapes → DBSCAN can identify clusters of arbitrary shapes and is not sensitive to outliers.

Mini-batch KMeans First, let's proceed with the Mini-batch KMeans and get the cluster labels. We can use the same `Elbow Method` to find the clusters and we can notice the optimal clusters, in this case, is 5 as well.

Step 1 - Elbow Method Let's first find the optimal number of clusters using the `Elbow Method` for the Mini-batch KMeans algorithm.

[61]:
```
# Select the Model
model = MiniBatchKMeans(random_state=42)
# Get cluster-wise-inertias
minibatch_kmeans_per_k = [MiniBatchKMeans(n_clusters=k, random_state=42).fit(X)
                for k in range(1, 20)]
inertias = [model.inertia_ for model in minibatch_kmeans_per_k]

# Selet the KElbowVisualizer from Yellowbrick library
visualizer = KElbowVisualizer(model, k=(2,10))

visualizer.fit(df_pca_scaled)
visualizer.show()
plt.show()
```



After selecting the optimal number of clusters(in my case its 5) we can fit and predict it to get the associated labels in the dataset.

[62]:
```
# Mini-batch KMeans
minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)
minibatch_kmeans.fit(X)
```

[62]:
```
MiniBatchKMeans(n_clusters=5, random_state=42)
```

[63]:
```
labels = minibatch_kmeans.labels_
```

[64]:
```
labels.shape
```

[64]:
```
(8950,)
```

After getting the labels we can check for the centroids of the data set as well.

[65]:
```
minibatch_kmeans.cluster_centers_
```

[65]:
```
array([[-5.19084583e+00, -1.19426375e+00,  8.16943580e-01,
        -3.71420311e+00, -1.46543561e+00, -4.35690324e-01,
        -5.88458174e-01,  3.76911205e-01,  4.53440511e-02,
        -5.00463997e-02,  3.08950389e-02,  4.65548721e-02,
        -1.14819068e-01, -5.09248295e-02, -9.39742871e-02,
```

```
       -1.11266487e-02,  3.09995439e-02],
      [ 5.43475243e+00,  1.12152408e+00, -4.41076156e+00,
       -1.38937399e+00,  8.74237232e-01,  4.99633800e-01,
        1.58684335e-01, -1.92880390e-01, -1.18042278e-01,
        2.92432931e-01,  1.95809056e-02, -1.55701793e-01,
        2.91559143e-01, -1.01242787e-01,  6.85568424e-02,
        6.32259245e-03,  1.59384387e-02],
      [-8.40098697e+00,  6.75694549e+00, -6.37968336e-01,
        2.45643260e+00, -3.67635657e-01,  2.39036696e-01,
        2.58552511e-01, -1.25713448e-01,  1.78991420e-01,
        5.25529342e-02,  1.30646367e-01, -2.18004717e-02,
        7.99276759e-02,  1.21265322e-02,  1.73401377e-02,
        3.18752212e-02,  1.67911720e-02],
      [ 1.00117187e+01,  1.53128304e+00,  2.64017640e+00,
        6.66578997e-01, -1.35135613e+00,  9.41032228e-02,
       -5.09610910e-02,  1.35572169e-01,  1.79230979e-01,
       -4.47449404e-02, -6.25931044e-02,  1.75672235e-01,
       -2.33917738e-01,  1.86712116e-01,  2.97209248e-02,
        7.14602268e-03, -9.36548899e-02],
      [-2.02435842e+00, -7.29357316e+00,  1.41531189e+00,
        2.66207628e+00,  2.33640281e+00, -3.10032616e-01,
        2.95862554e-01, -3.20785200e-01, -2.75871429e-01,
       -2.19416359e-01, -6.86329793e-02, -7.28454211e-02,
       -3.00444225e-02, -6.64444134e-03, -9.63792948e-03,
       -5.43911589e-02,  1.77128698e-02]])
```

We can further analyze labels to extract more data about the centroids as well.

[66]:
```
minibatch_kmeans.cluster_centers_.shape
```

[66]:
```
(5, 17)
```

[67]:
```
df_cluster = pd.concat([df_pca_scaled, pd.DataFrame(labels,␣
 ↪columns=['cluster'])],axis=1)
df_cluster.head()
```

[67]:

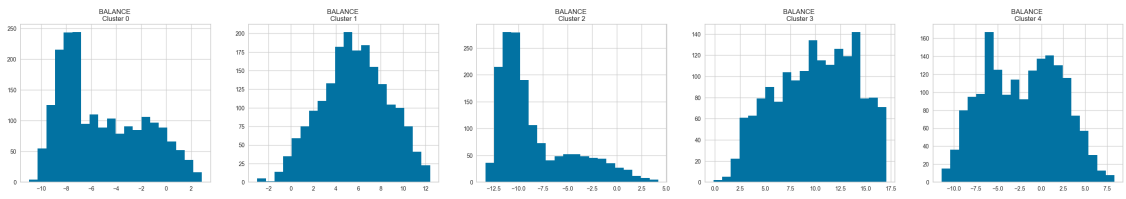
	BALANCE	PURCHASES	PURCHASES_TRX	CREDIT_LIMIT	PAYMENTS	\
0	-5.424098	-8.235778	1.406150	-1.803849	1.893730	
1	-8.994678	4.366142	0.058750	5.392989	-2.671634	
2	4.212950	2.569625	6.414961	-5.414136	-0.756905	
3	-2.788429	-2.400597	8.369653	-1.003194	6.179858	
4	-6.045446	-3.826287	0.903521	-5.209828	-2.358177	

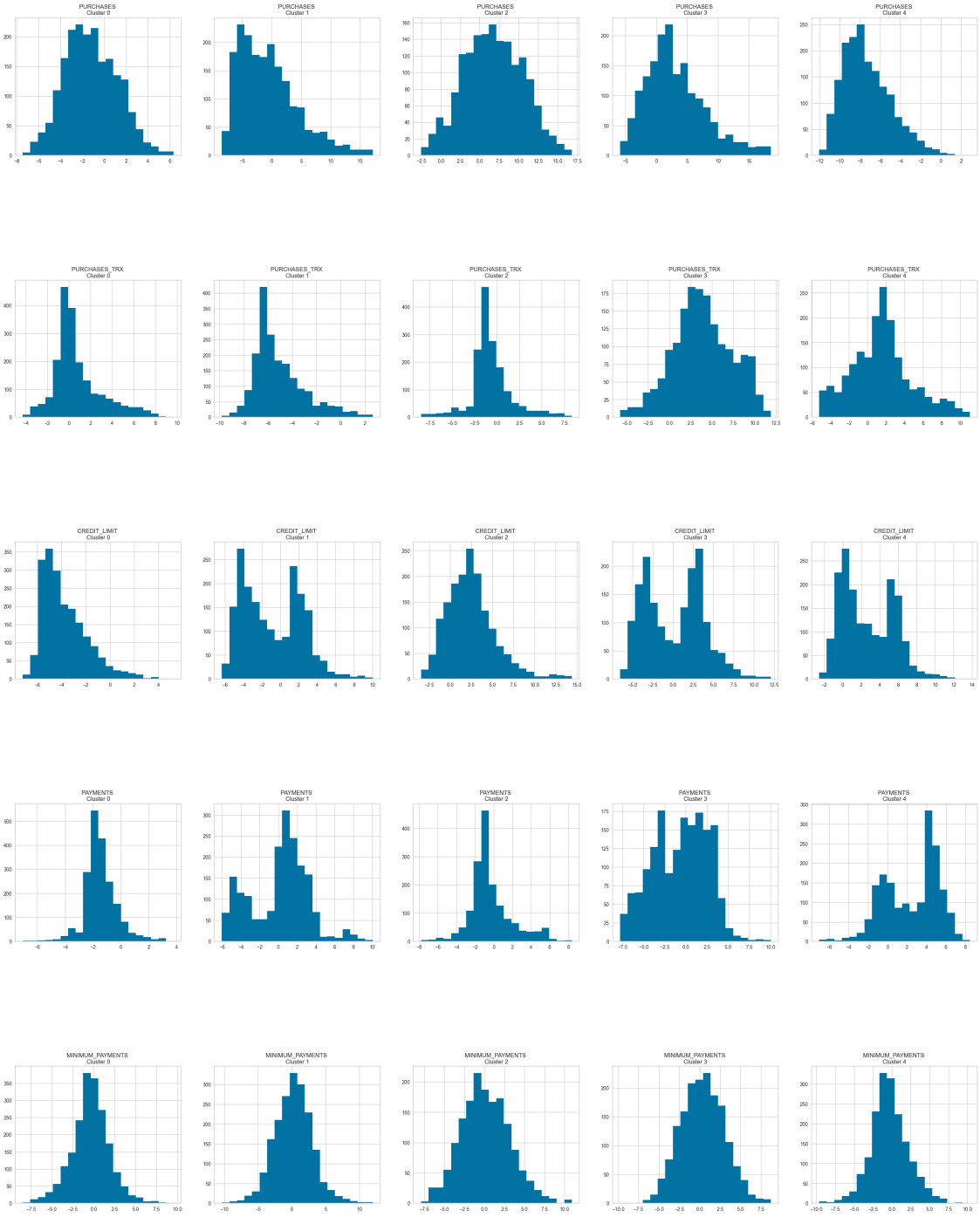
	MINIMUM_PAYMENTS	BALANCE_FREQUENCY	ONEOFF_PURCHASES	\
0	0.536220	-1.554429	-1.427039	
1	-5.811712	2.569535	0.660693	
2	1.810996	4.607691	-1.160287	
3	-1.592537	0.100046	1.138640	

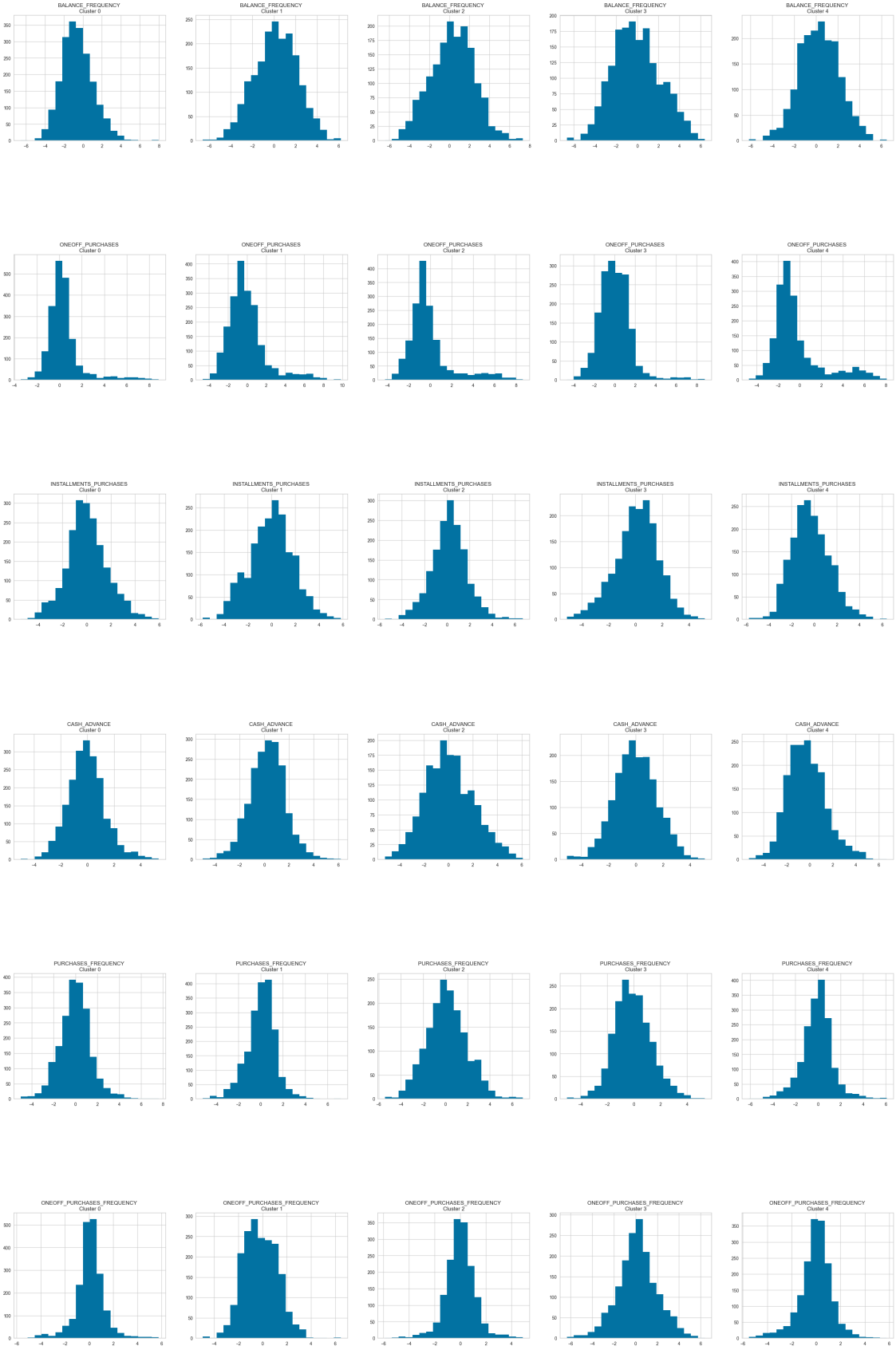
```
4         -1.184562          -1.525047          0.148748

   INSTALLMENTS_PURCHASES  CASH_ADVANCE  PURCHASES_FREQUENCY  \
0                0.240992      0.614772             0.425296
1               -1.022840      2.640337             0.806268
2                0.620563     -0.826130            -1.388430
3               -0.713468     -5.267590             3.929801
4                0.244583      1.690520            -0.537904

   ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
0                    0.664810                          0.603416
1                    0.409229                         -1.224936
2                    2.102297                          0.212080
3                   -4.293793                         -0.269293
4                    0.795709                         -1.197224

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PRC_FULL_PAYMENT    TENURE  \
0                0.007705         -0.574741         -0.124425 -0.519374
1               -0.541644          0.090286         -0.559080 -0.143043
2               -1.491156          2.649925          0.265688 -0.686067
3               -0.118928          1.530771         -0.185934  0.506732
4               -0.460919         -0.570811          0.219043 -0.185949

   cluster
0        4
1        2
2        3
3        4
4        0
```

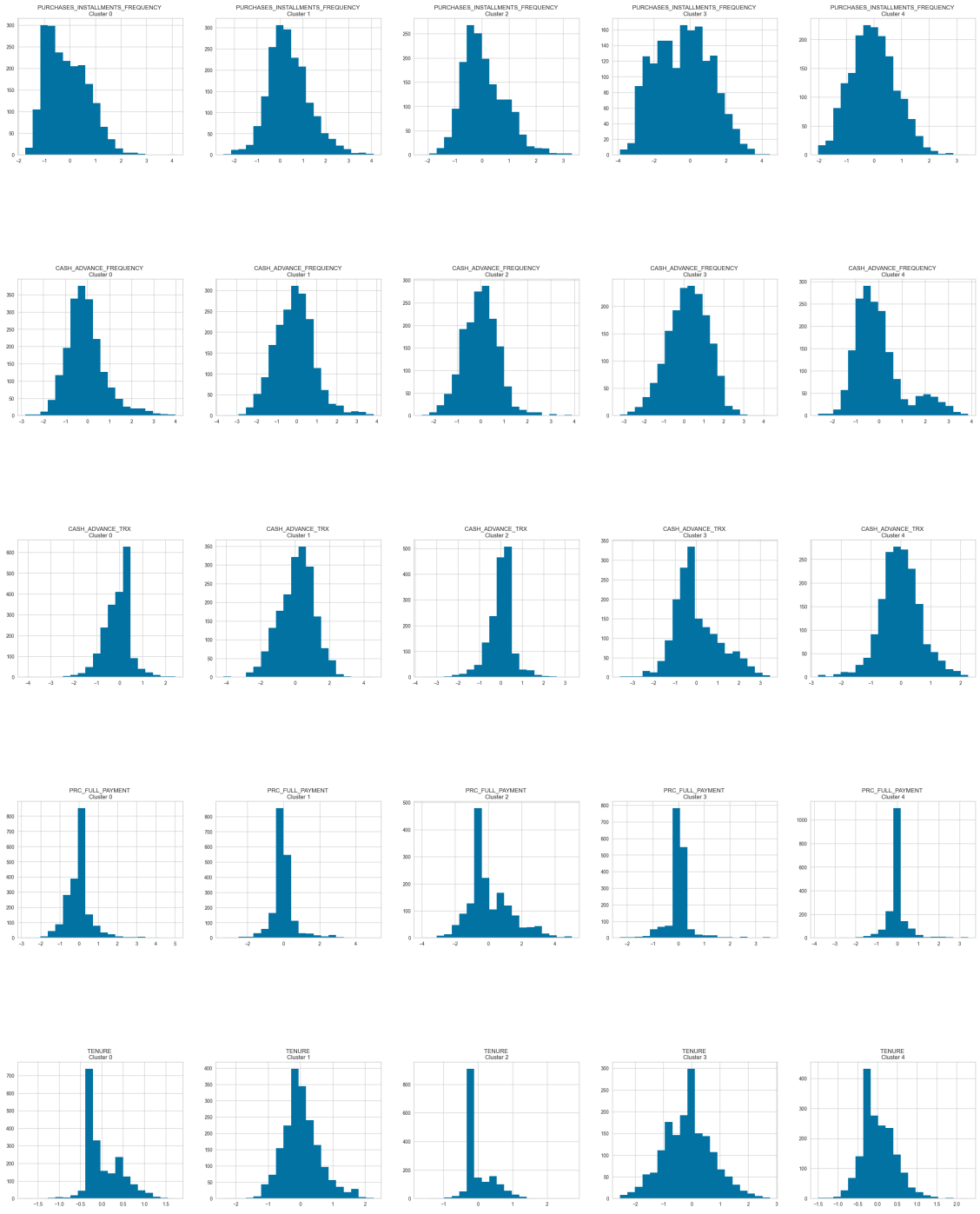
Let's plot the graphs to see the predictions more clearly.

[68]:
```
for i in df_pca_scaled:
    plt.figure(figsize=(35,5))
    for j in range(5):
        plt.subplot(1,5,j+1)
        cluster = df_cluster[df_cluster['cluster']==j]
        cluster[i].hist(bins=20)
        plt.title(f'{i} \nCluster {j}')
    plt.show()
```




PURCHASES
Cluster 0
PURCHASES
Cluster 1
PURCHASES
Cluster 2
PURCHASES
Cluster 3
PURCHASES
Cluster 4
PURCHASES_TRX
Cluster 0
PURCHASES_TRX
Cluster 1
PURCHASES_TRX
Cluster 2
PURCHASES_TRX
Cluster 3
PURCHASES_TRX
Cluster 4
CREDIT_LIMIT
Cluster 0
CREDIT_LIMIT
Cluster 1
CREDIT_LIMIT
Cluster 2
CREDIT_LIMIT
Cluster 3
CREDIT_LIMIT
Cluster 4
PAYMENTS
Cluster 0
PAYMENTS
Cluster 1
PAYMENTS
Cluster 2
PAYMENTS
Cluster 3
PAYMENTS
Cluster 4
MINIMUM_PAYMENTS
Cluster 0
MINIMUM_PAYMENTS
Cluster 1
MINIMUM_PAYMENTS
Cluster 2
MINIMUM_PAYMENTS
Cluster 3
MINIMUM_PAYMENTS
Cluster 4


BALANCE_FREQUENCY
Cluster 0
BALANCE_FREQUENCY
Cluster 1
BALANCE_FREQUENCY
Cluster 2
BALANCE_FREQUENCY
Cluster 3
BALANCE_FREQUENCY
Cluster 4
ONEOFF_PURCHASES
Cluster 0
ONEOFF_PURCHASES
Cluster 1
ONEOFF_PURCHASES
Cluster 2
ONEOFF_PURCHASES
Cluster 3
ONEOFF_PURCHASES
Cluster 4
INSTALLMENTS_PURCHASES
Cluster 0
INSTALLMENTS_PURCHASES
Cluster 1
INSTALLMENTS_PURCHASES
Cluster 2
INSTALLMENTS_PURCHASES
Cluster 3
INSTALLMENTS_PURCHASES
Cluster 4
CASH_ADVANCE
Cluster 0
CASH_ADVANCE
Cluster 1
CASH_ADVANCE
Cluster 2
CASH_ADVANCE
Cluster 3
CASH_ADVANCE
Cluster 4
PURCHASES_FREQUENCY
Cluster 0
PURCHASES_FREQUENCY
Cluster 1
PURCHASES_FREQUENCY
Cluster 2
PURCHASES_FREQUENCY
Cluster 3
PURCHASES_FREQUENCY
Cluster 4
ONEOFF_PURCHASES_FREQUENCY
Cluster 0
ONEOFF_PURCHASES_FREQUENCY
Cluster 1
ONEOFF_PURCHASES_FREQUENCY
Cluster 2
ONEOFF_PURCHASES_FREQUENCY
Cluster 3
ONEOFF_PURCHASES_FREQUENCY
Cluster 4



Let's check the Mini-batch KMeans inertia and score as well.

[69]:
```
minibatch_kmeans.inertia_
```

[69]: 719608.4544140967

Step 2 - Silhouette Score KMeans Score for Clusters = 5

[70]:
```
minibatch_kmeans.score(X)
```

[70]: -719608.4544140967

Silhouette Score for Clusters = 5

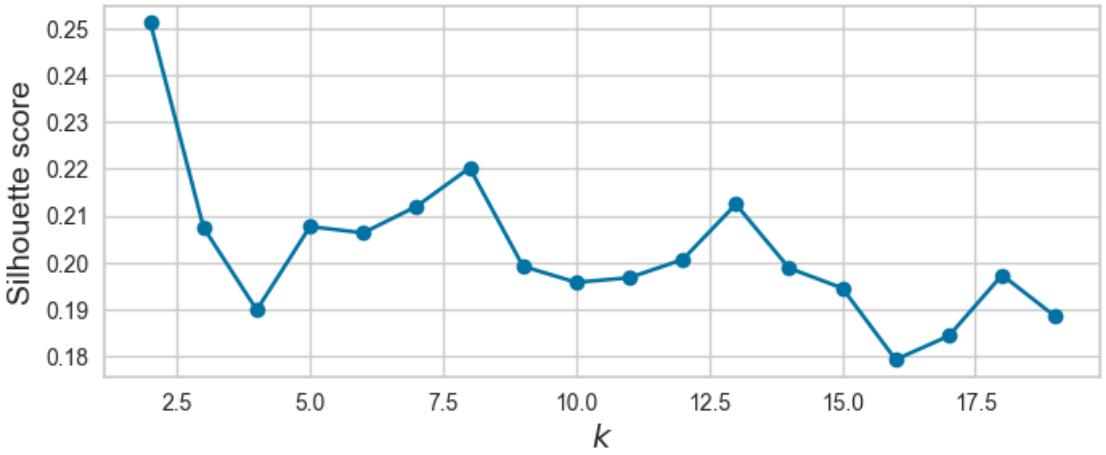
[71]:
```
silhouette_score(X, minibatch_kmeans.labels_)
```

[71]: 0.20769327678276242

We are plotting the `Silhouette Score` for the Mini-batch KMeans clusters 1 to 20 which we tested earlier using the `Elbow Method` to find the optimal number of clusters.

[72]:
```
silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in minibatch_kmeans_per_k[1:]]
```

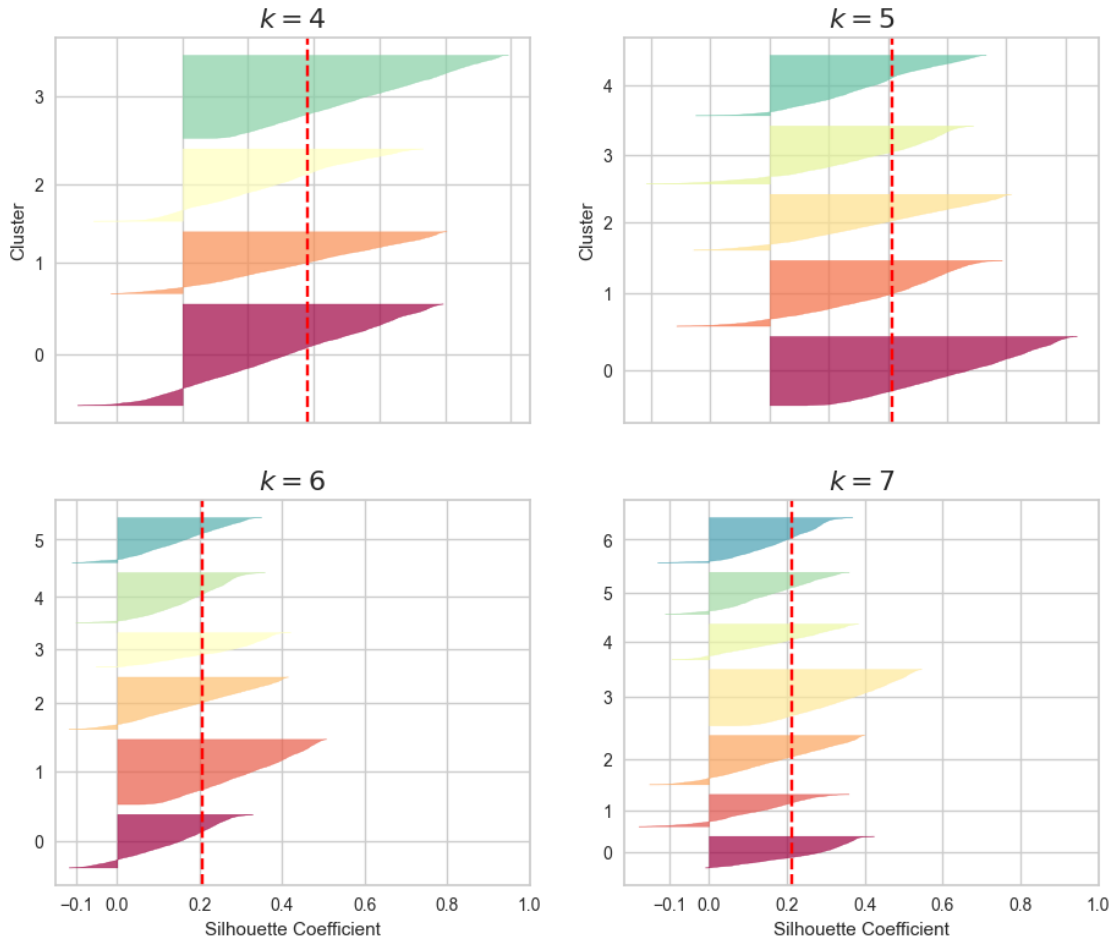
[73]:
```
plt.figure(figsize=(8, 3))
plt.plot(range(2, 20), silhouette_scores, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Silhouette score", fontsize=14)
plt.show()
```



Plot `Silhouette Coefficients` graph.

[74]:
```
plt.figure(figsize=(11, 9))

for k in (4, 5, 6, 7):
    plt.subplot(2, 2, k - 3)
```

```
    y_pred = minibatch_kmeans_per_k[k - 1].labels_
    silhouette_coefficients = silhouette_samples(X, y_pred)

    padding = len(X) // 30
    pos = padding
    ticks = []
    for i in range(k):
        coeffs = silhouette_coefficients[y_pred == i]
        coeffs.sort()

        color = mpl.cm.Spectral(i / k)
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
    plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
    if k in (4,5):
        plt.ylabel("Cluster")

    if k in (6,7):
        plt.gca().set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
        plt.xlabel("Silhouette Coefficient")
    else:
        plt.tick_params(labelbottom=False)

    plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
    plt.title("$k={}$".format(k), fontsize=16)

plt.show()
```



Density-Based Spatial Clusterings of Applications with Noise(DBScan) First, let's proceed with the DBSCAN and get the cluster labels.

Step 1 - Silhoutte Scores

[75]:
```
# DBSCAN
for eps in range(1, 10):
    for min_samples in range (1, 10):
        dbscan = DBSCAN(eps = eps, min_samples = min_samples)
        labels = dbscan.fit_predict(df_pca_scaled)
        score = silhouette_score(X, labels)

        # We need to at least have the silhouette score > 0
        if(score > 0):
            print(f'\neps {eps}')
            print(f'\nmin samples {min_samples}')
            print(f'clusters present: {np.unique(labels)}')
            print(f'clusters sizes: {np.bincount(labels + 1)}')
```

```
            print(f'Silhouette Score: {score}')
```

```
eps 1

min samples 1
clusters present: [   0    1    2 … 8140 8141 8142]
clusters sizes: [0 1 1 … 1 1 1]
Silhouette Score: 0.038647184062067075


eps 5

min samples 7
clusters present: [-1  0]
clusters sizes: [1188 7762]
Silhouette Score: 0.08605581206812085


eps 6

min samples 4
clusters present: [-1  0  1  2]
clusters sizes: [ 365 8578    3    4]
Silhouette Score: 0.0450578006950032


eps 6

min samples 6
clusters present: [-1  0]
clusters sizes: [ 454 8496]
Silhouette Score: 0.1011788233398261


eps 6

min samples 7
clusters present: [-1  0]
clusters sizes: [ 482 8468]
Silhouette Score: 0.10285223752204822


eps 6

min samples 8
clusters present: [-1  0]
clusters sizes: [ 514 8436]
Silhouette Score: 0.10241549471191899
```

```
eps 6

min samples 9
clusters present: [-1  0]
clusters sizes: [ 544 8406]
Silhouette Score: 0.10006226476577831


eps 7

min samples 4
clusters present: [-1  0  1]
clusters sizes: [ 157 8789    4]
Silhouette Score: 0.051500691059250474


eps 7

min samples 5
clusters present: [-1  0]
clusters sizes: [ 170 8780]
Silhouette Score: 0.12066004929319041


eps 7

min samples 6
clusters present: [-1  0]
clusters sizes: [ 176 8774]
Silhouette Score: 0.12031833448341936


eps 7

min samples 7
clusters present: [-1  0]
clusters sizes: [ 191 8759]
Silhouette Score: 0.12092606565654407


eps 7

min samples 8
clusters present: [-1  0]
clusters sizes: [ 205 8745]
Silhouette Score: 0.12575843901039038


eps 7

min samples 9
```

```
clusters present: [-1  0]
clusters sizes: [ 214 8736]
Silhouette Score: 0.12551210887392897


eps 8

min samples 2
clusters present: [-1  0  1  2  3  4  5]
clusters sizes: [  24 8913    2    3    2    4    2]
Silhouette Score: 0.022146357371238802


eps 8

min samples 3
clusters present: [-1  0  1  2]
clusters sizes: [  30 8913    4    3]
Silhouette Score: 0.04311454563848459


eps 8

min samples 4
clusters present: [-1  0  1]
clusters sizes: [  39 8907    4]
Silhouette Score: 0.12101611591862231


eps 8

min samples 5
clusters present: [-1  0]
clusters sizes: [  49 8901]
Silhouette Score: 0.14744152920052206


eps 8

min samples 6
clusters present: [-1  0]
clusters sizes: [  53 8897]
Silhouette Score: 0.1471749479797535


eps 8

min samples 7
clusters present: [-1  0]
clusters sizes: [  58 8892]
Silhouette Score: 0.1583479195327679
```

```
eps 8

min samples 8
clusters present: [-1  0]
clusters sizes: [  60 8890]
Silhouette Score: 0.15676830165744476


eps 8

min samples 9
clusters present: [-1  0]
clusters sizes: [  62 8888]
Silhouette Score: 0.1604190352227765


eps 9

min samples 2
clusters present: [-1  0]
clusters sizes: [   8 8942]
Silhouette Score: 0.15684115236947135


eps 9

min samples 3
clusters present: [-1  0]
clusters sizes: [   8 8942]
Silhouette Score: 0.15684115236947135


eps 9

min samples 4
clusters present: [-1  0]
clusters sizes: [   9 8941]
Silhouette Score: 0.1651108531287837


eps 9

min samples 5
clusters present: [-1  0]
clusters sizes: [  11 8939]
Silhouette Score: 0.17320188257716032


eps 9
```

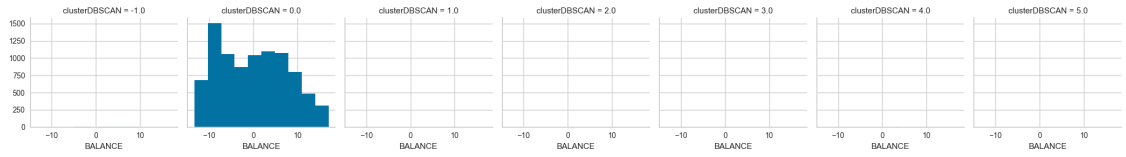
```
min samples 6
clusters present: [-1  0]
clusters sizes: [  11 8939]
Silhouette Score: 0.17320188257716032


eps 9

min samples 7
clusters present: [-1  0]
clusters sizes: [  14 8936]
Silhouette Score: 0.16950078539660343


eps 9

min samples 8
clusters present: [-1  0]
clusters sizes: [  15 8935]
Silhouette Score: 0.1659836675423123


eps 9

min samples 9
clusters present: [-1  0]
clusters sizes: [  16 8934]
Silhouette Score: 0.16334808592122696
```

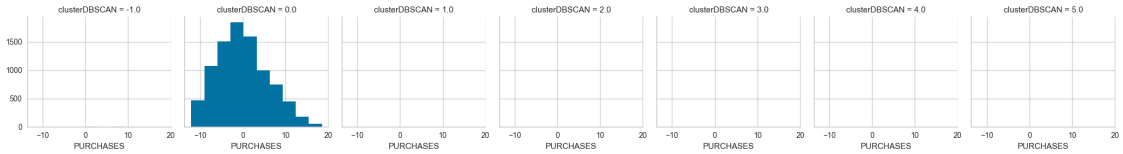
From the above values we can see the best clustering value is given at `eps=8` and `min_samples=2`.

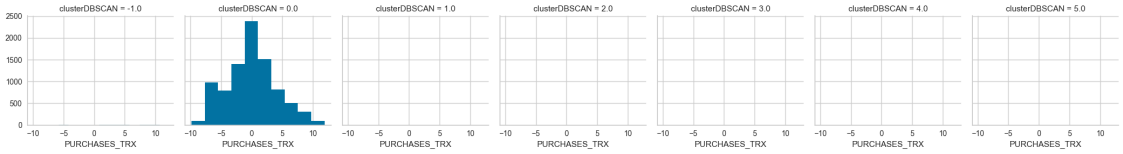
[76]:
```
# Optimal DBSCAN Model
db = DBSCAN(eps=8, min_samples=2).fit(df_pca_scaled)
labels = db.labels_
```

[77]:
```
df_cluster['clusterDBSCAN'] = labels.astype(float)
```

[78]:
```
for c in df_cluster:
    grid= sns.FacetGrid(df_cluster, col='clusterDBSCAN')
    grid.map(plt.hist, c)
```



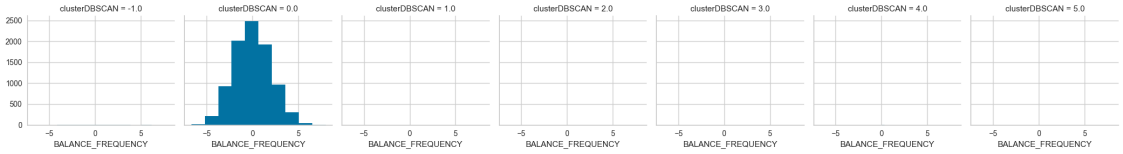

clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
PURCHASES

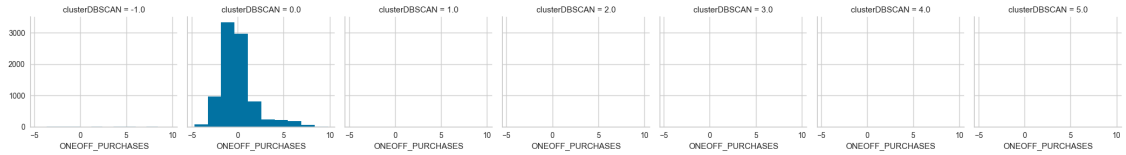

clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
PURCHASES_TRX

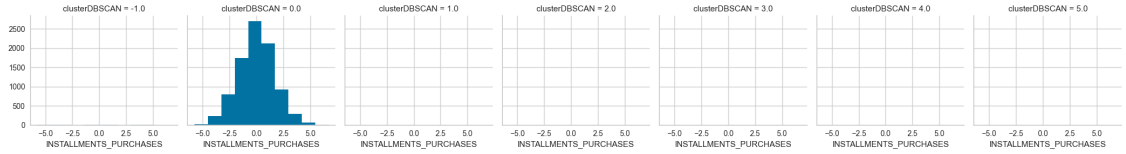

clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
CREDIT_LIMIT

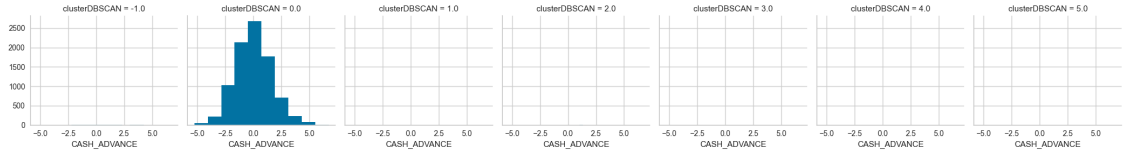

clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
PAYMENTS

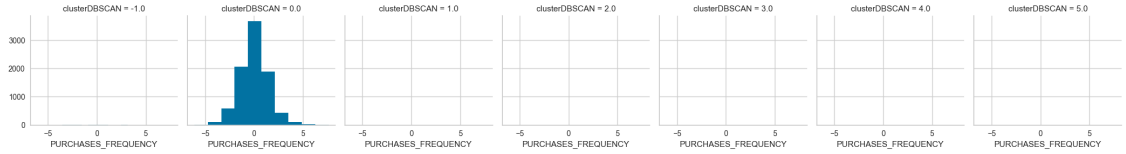

clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
MINIMUM_PAYMENTS

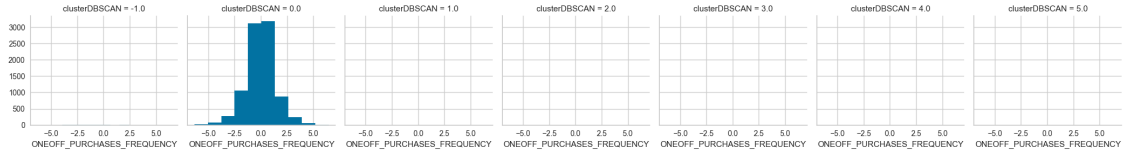

clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
BALANCE_FREQUENCY

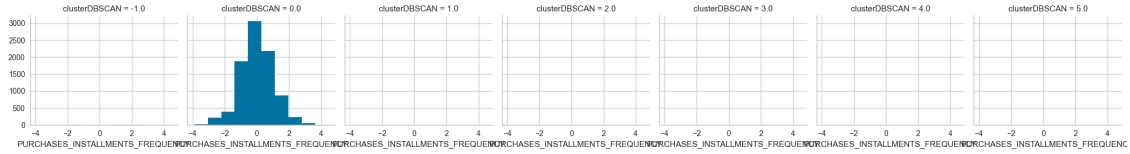

clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
ONEOFF_PURCHASES

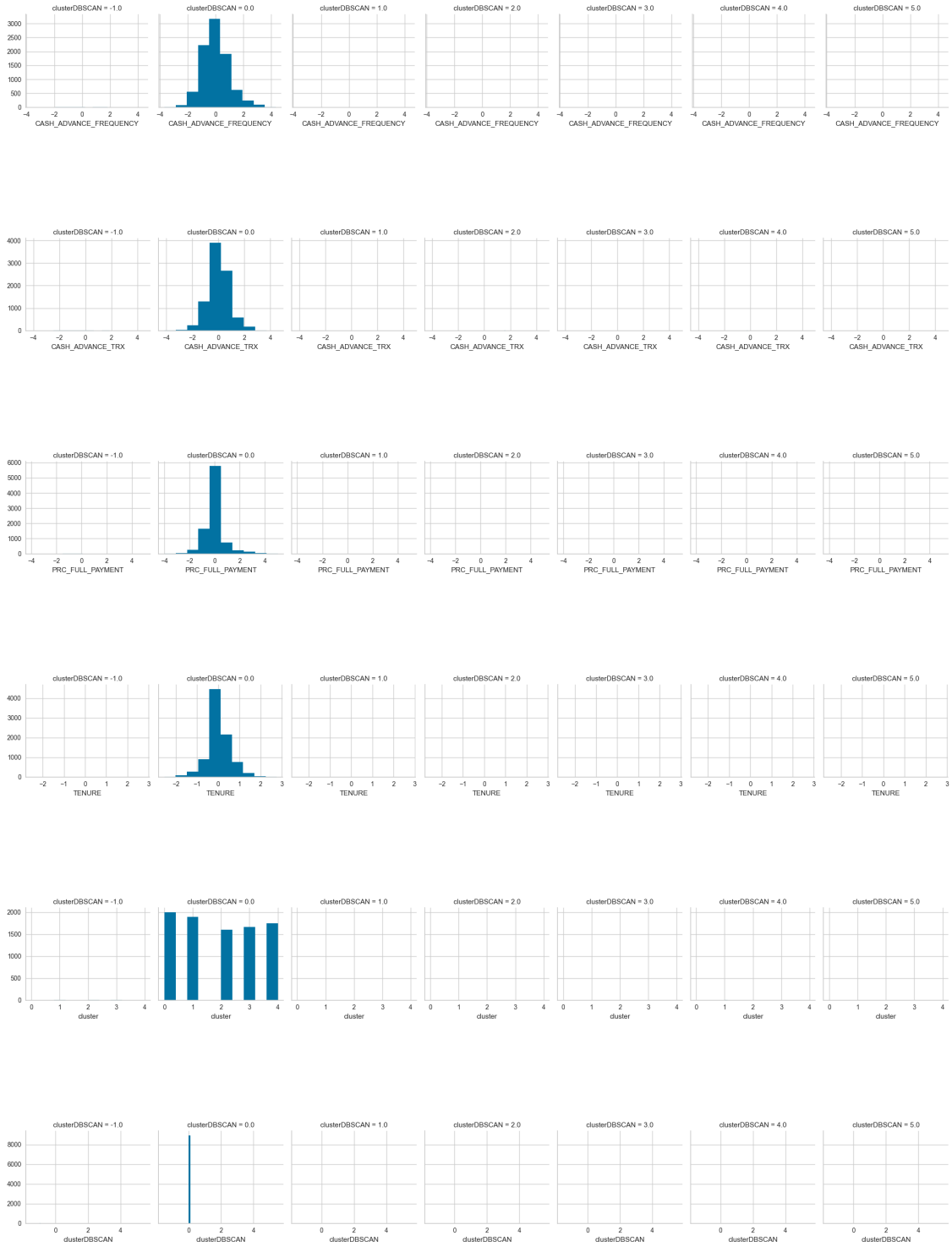

clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
INSTALLMENTS_PURCHASES


clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
CASH_ADVANCE


clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
PURCHASES_FREQUENCY


clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
ONEOFF_PURCHASES_FREQUENCY


clusterDBSCAN = -1.0
clusterDBSCAN = 0.0
clusterDBSCAN = 1.0
clusterDBSCAN = 2.0
clusterDBSCAN = 3.0
clusterDBSCAN = 4.0
clusterDBSCAN = 5.0
PURCHASES_INSTALLMENTS_FREQUENCY



Plot the estimated number of clusters

[79]:
```
labels = db.labels_
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1) # -1 is not count since it's considered as␣
 ↪noise

print("Estimated number of clusters: %d" % n_clusters_)
```

```
Estimated number of clusters: 6
```

1.5.1 Comparing the Classification Algorithms

For the KMeans and Mini-batch KMeans algorithms, we were able to get 5 as the number of clusters. This number was taken by considering both the `Elbow Method` and `Silhouette Scores`. But for the DBSCAN method, we selected `eps=8` and `min_samples=2` since they gave the best `Silhouette Scores`.

Algorithm	Optimal Values
KMeans	Number of Clusters = 5
Mini-batch KMeans	Number of Clusters = 5
DBSCAN	Number of Clusters = 6, with `eps=8` and `min_sample=2`

Performance First, let's check the performance of the three algorithms with their respective optimal values

[80]:
```
%timeit KMeans(n_clusters=5, random_state=42).fit(X)
```

```
204 ms ± 15.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[81]:
```
%timeit MiniBatchKMeans(n_clusters=5, random_state=42).fit(X)
```

```
73.7 ms ± 4.36 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[82]:
```
%timeit DBSCAN(eps=8, min_samples=2).fit(df_pca_scaled)
```

```
90.9 ms ± 1.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

It can be seen that Mini-batch KMeans algorithm runs much faster than DBSCAN and KMeans algorithms. Since it is a variation of KMeans algorithm let's plot the performances of KMeans and Mini-batch KMeans algorithms to get more insights.

[83]:
```
num_of_times = 100
times = np.empty((num_of_times, 2))
inertias = np.empty((num_of_times, 2))
for k in range(1, num_of_times + 1):
    kmeans_ = KMeans(n_clusters=k, random_state=42)
    minibatch_kmeans_ = MiniBatchKMeans(n_clusters=k, random_state=42)
    print("\r{}/{}".format(k, num_of_times), end="")
    times[k-1, 0] = timeit("kmeans_.fit(X)", number=10, globals=globals())
```

```
    times[k-1, 1]  = timeit("minibatch_kmeans_.fit(X)", number=10,␣
↪globals=globals())
    inertias[k-1, 0] = kmeans_.inertia_
    inertias[k-1, 1] = minibatch_kmeans_.inertia_
```

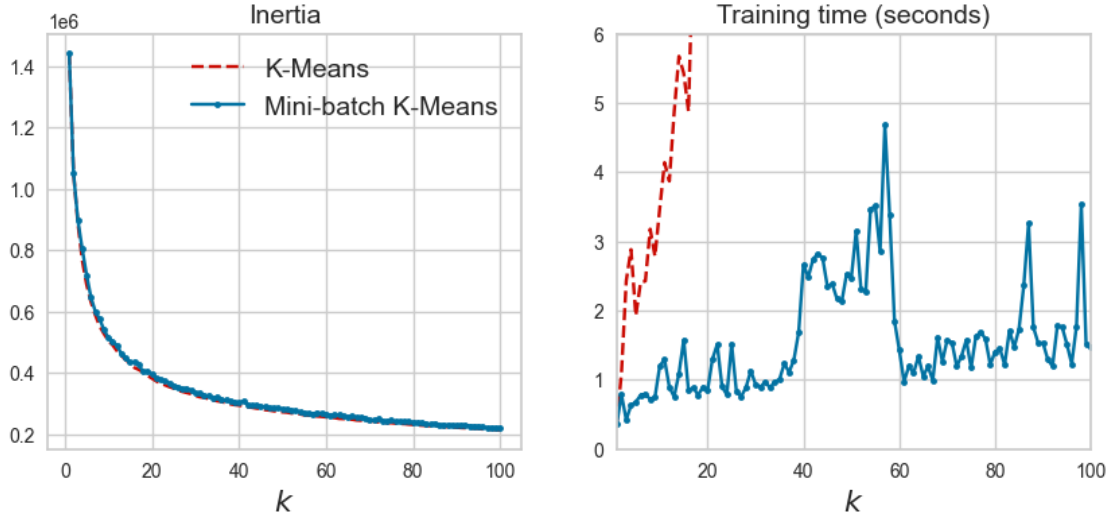
```
1/100
2/100
3/100
4/100
5/100
6/100
7/100
8/100
9/100
10/100
11/100
12/100
13/100
14/100
15/100
16/100
17/100
18/100
19/100
20/100
21/100
22/100
23/100
24/100
25/100
26/100
27/100
28/100
29/100
```

30/100
31/100
32/100
33/100
34/100
35/100
36/100
37/100
38/100
39/100
40/100
41/100
42/100
43/100
44/100
45/100
46/100
47/100
48/100
49/100
50/100
51/100
52/100
53/100
54/100
55/100
56/100
57/100
58/100
59/100
60/100
61/100

62/100
63/100
64/100
65/100
66/100
67/100
68/100
69/100
70/100
71/100
72/100
73/100
74/100
75/100
76/100
77/100
78/100
79/100
80/100
81/100
82/100
83/100
84/100
85/100
86/100
87/100
88/100
89/100
90/100
91/100
92/100
93/100

```
94/100
95/100
96/100
97/100
98/100
99/100
100/100
```

[84]:
```
plt.figure(figsize=(10,4))

plt.subplot(121)
plt.plot(range(1, 101), inertias[:, 0], "r--", label="K-Means")
plt.plot(range(1, 101), inertias[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.title("Inertia", fontsize=14)
plt.legend(fontsize=14)

plt.subplot(122)
plt.plot(range(1, 101), times[:, 0], "r--", label="K-Means")
plt.plot(range(1, 101), times[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.title("Training time (seconds)", fontsize=14)
plt.axis([1, 100, 0, 6])

plt.show()
```



We can see that the Intertia values of KMeans and Mini-batch KMeans values are equal but

training times of the two are different. Therefore, it is better to use Mini-batch KMeans than KMeans algorithm.

Silhouette Scores Next, let's see the Silhouette Scores of each algorithm. To do that, we can use the following Python snippet.

[85]:
```
# Create an instance of PrettyTable
table = PrettyTable()

# Define table headers
table.field_names = ["Algorithm", "Silhouette Score"]

# Enumerate the dictionary
graph_dict = dict()
table.add_row(["KMeans", silhouette_score(X,  kmeans.labels_)])
table.add_row(["Mini-batch KMeans", silhouette_score(X, minibatch_kmeans.
 ↪labels_)])
table.add_row(["DBSCAN", silhouette_score(X, db.labels_)])

# Set table formatting options
table.align = "l"  # Align text to the left
table.border = True  # Add border to the table
table.hrules = True  # Add horizontal rules between rows

# Print the table
print(table)
```

```
+------------------+----------------------+
| Algorithm        | Silhouette Score     |
+------------------+----------------------+
| KMeans           | 0.2165475102195835   |
+------------------+----------------------+
| Mini-batch KMeans | 0.20769327678276242  |
+------------------+----------------------+
| DBSCAN           | 0.022146357371238802 |
+------------------+----------------------+
```

We can see that all the `Silhouette Scores` are positive and therefore the clusters are defined properly. Since the highest `Silhouette Score` is for `KMeans` algorithm we can deduce that it gives better results.

Davies-Bouldin Index The Davies-Bouldin index measures the average similarity between each cluster and its most similar cluster, normalized by the cluster size. Lower values indicate better clustering. It provides a measure of the compactness and separation of clusters. We can calculate and print the Davies-Bouldin Index values for our algorithms as shown below.

[86]:
```
# Define Davies-Bouldin Indexes for three algorithms
dbi_kmeans = davies_bouldin_score(X, kmeans.labels_)
```

```
dbi_minibatch_kmeans = davies_bouldin_score(X, minibatch_kmeans.labels_)
dbi_dbscan = davies_bouldin_score(X, db.labels_)

# Create an instance of PrettyTable
table = PrettyTable()

# Define table headers
table.field_names = ["Algorithm", "Davies-Bouldin Index"]

# Enumerate the dictionary
graph_dict = dict()
table.add_row(["KMeans", dbi_kmeans])
table.add_row(["Mini-batch KMeans", dbi_minibatch_kmeans])
table.add_row(["DBSCAN", dbi_dbscan])

# Set table formatting options
table.align = "l"  # Align text to the left
table.border = True  # Add border to the table
table.hrules = True  # Add horizontal rules between rows

# Print the table
print(table)
```

```
+------------------+----------------------+
| Algorithm        | Davies-Bouldin Index |
+------------------+----------------------+
| KMeans           | 1.5146499720660827   |
+------------------+----------------------+
| Mini-batch KMeans | 1.5719225993454213   |
+------------------+----------------------+
| DBSCAN           | 1.6733455630235472   |
+------------------+----------------------+
```

Since the lowest `Davies-Bouldin Index` is for `KMeans` algorithm we can deduce that it gives better results.

Calinski Harabasz Score The Calinski-Harabasz index is the ratio of the sum of between-cluster dispersion to within-cluster dispersion for all clusters. Higher values indicate better-defined, more separate clusters. We can calculate and print the Calinski Harabasz Scores for our algorithms as shown below.

[87]:
```
# Define Calinski Harabasz Score for three algorithms
chs_kmeans = calinski_harabasz_score(X, kmeans.labels_)
chs_minibatch_kmeans = calinski_harabasz_score(X, minibatch_kmeans.labels_)
chs_dbscan = calinski_harabasz_score(X, db.labels_)

# Create an instance of PrettyTable
table = PrettyTable()
```

```
# Define table headers
table.field_names = ["Algorithm", "Calinski Harabasz Scores"]

# Enumerate the dictionary
graph_dict = dict()
table.add_row(["KMeans", chs_kmeans])
table.add_row(["Mini-batch KMeans", chs_minibatch_kmeans])
table.add_row(["DBSCAN", chs_dbscan])

# Set table formatting options
table.align = "l"  # Align text to the left
table.border = True  # Add border to the table
table.hrules = True  # Add horizontal rules between rows

# Print the table
print(table)
```

```
+------------------+--------------------------+
| Algorithm        | Calinski Harabasz Scores |
+------------------+--------------------------+
| KMeans           | 2453.718411981619        |
+------------------+--------------------------+
| Mini-batch KMeans | 2355.678998982053        |
+------------------+--------------------------+
| DBSCAN           | 5.99185601655639         |
+------------------+--------------------------+
```

Since the higest `Calinski HarabaszS` is for `KMeans` algorithm we can deduce that it gives better results.

Metrics Results Following is the summarization of metrics.

Metric	Best Algorithm
Performance	Mini-batch KMeans
Silhouette Scoring	KMeans
Davies-Boulding Index	KMeans
Calinski Harabasz Score	KMeans

Therefore, with the above metrics we can deduce that although the `KMeans` algorithm is low in the `performance` aspect, it gives the best results. But if the performance aspect is also needed to be considered, the `Mini-batch KMeans` also gives good results, as you can see from the above tables.

1.5.2 Insights

The following code shows, how you can append the cluster labels to the data frame for the three algorithms, view them in plots and provide insights related to the **Data Mining Problem** we have discussed in section 1.

[88]:
```
# Concatenate the labels
kmeans_labels = kmeans.labels_
minibatch_kmeans_labels = minibatch_kmeans.labels_
dbscan_labels = db.labels_

df_cluster = pd.concat([df1, pd.DataFrame(kmeans_labels, columns=['kmeans'])],␣
 ↪axis=1)
df_cluster = pd.concat([df_cluster, pd.DataFrame(minibatch_kmeans_labels,␣
 ↪columns=['minibatch_kmeans'])], axis=1)
df_cluster = pd.concat([df_cluster, pd.DataFrame(dbscan_labels,␣
 ↪columns=['dbscan'])], axis=1)

# Show first 10 rows
df_cluster.head(10)
```

[88]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	\
0	40.900749	0.818182	95.40	0.00	
1	3202.467416	0.909091	0.00	0.00	
2	2495.148862	1.000000	773.17	773.17	
3	1666.670542	0.636364	1499.00	1499.00	
4	817.714335	1.000000	16.00	16.00	
5	1809.828751	1.000000	1333.28	0.00	
6	627.260806	1.000000	7091.01	6402.63	
7	1823.652743	1.000000	436.20	0.00	
8	1014.926473	1.000000	861.49	661.49	
9	152.225975	0.545455	1281.60	1281.60	

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	\
0	95.40	0.000000	0.166667	
1	0.00	6442.945483	0.000000	
2	0.00	0.000000	1.000000	
3	0.00	205.788017	0.083333	
4	0.00	0.000000	0.083333	
5	1333.28	0.000000	0.666667	
6	688.38	0.000000	1.000000	
7	436.20	0.000000	1.000000	
8	200.00	0.000000	0.333333	
9	0.00	0.000000	0.166667	

	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY	\
0	0.000000	0.083333	
1	0.000000	0.000000	
2	1.000000	0.000000	
3	0.083333	0.000000	
4	0.083333	0.000000	
5	0.000000	0.583333	
6	1.000000	1.000000	

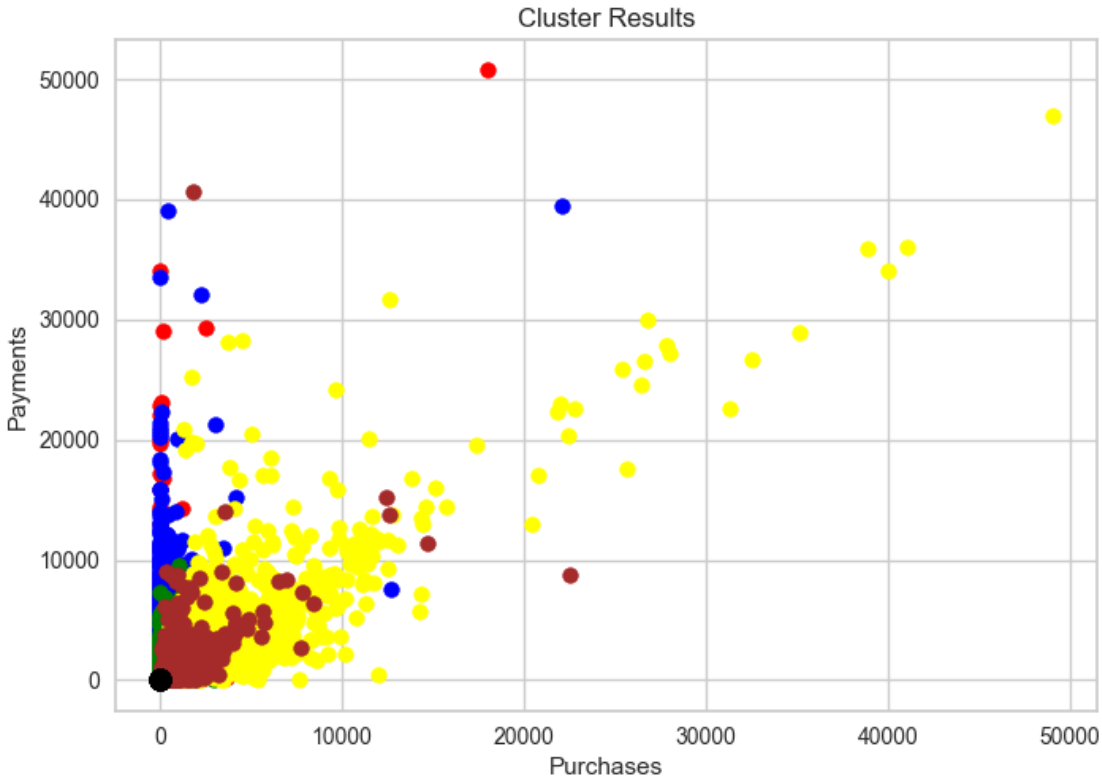
```
7                    0.000000                          1.000000
8                    0.083333                          0.250000
9                    0.166667                          0.000000

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PURCHASES_TRX  CREDIT_LIMIT  \
0                0.000000                 0              2        1000.0
1                0.250000                 4              0        7000.0
2                0.000000                 0             12        7500.0
3                0.083333                 1              1        7500.0
4                0.000000                 0              1        1200.0
5                0.000000                 0              8        1800.0
6                0.000000                 0             64       13500.0
7                0.000000                 0             12        2300.0
8                0.000000                 0              5        7000.0
9                0.000000                 0              3       11000.0

      PAYMENTS  MINIMUM_PAYMENTS  PRC_FULL_PAYMENT  TENURE  kmeans  \
0   201.802084        139.509787          0.000000      12       0
1  4103.032597       1072.340217          0.222222      12       1
2   622.066742        627.284787          0.000000      12       3
3     0.000000        864.206542          0.000000      12       0
4   678.334763        244.791237          0.000000      12       2
5  1400.057770       2407.246035          0.000000      12       4
6  6354.314328        198.065894          1.000000      12       3
7   679.065082        532.033990          0.000000      12       4
8   688.278568        311.963409          0.000000      12       2
9  1164.770591        100.302262          0.000000      12       0

   minibatch_kmeans  dbscan
0                 4       0
1                 2       0
2                 3       0
3                 4       0
4                 0       0
5                 1       0
6                 3       0
7                 1       0
8                 0       0
9                 4       0
```

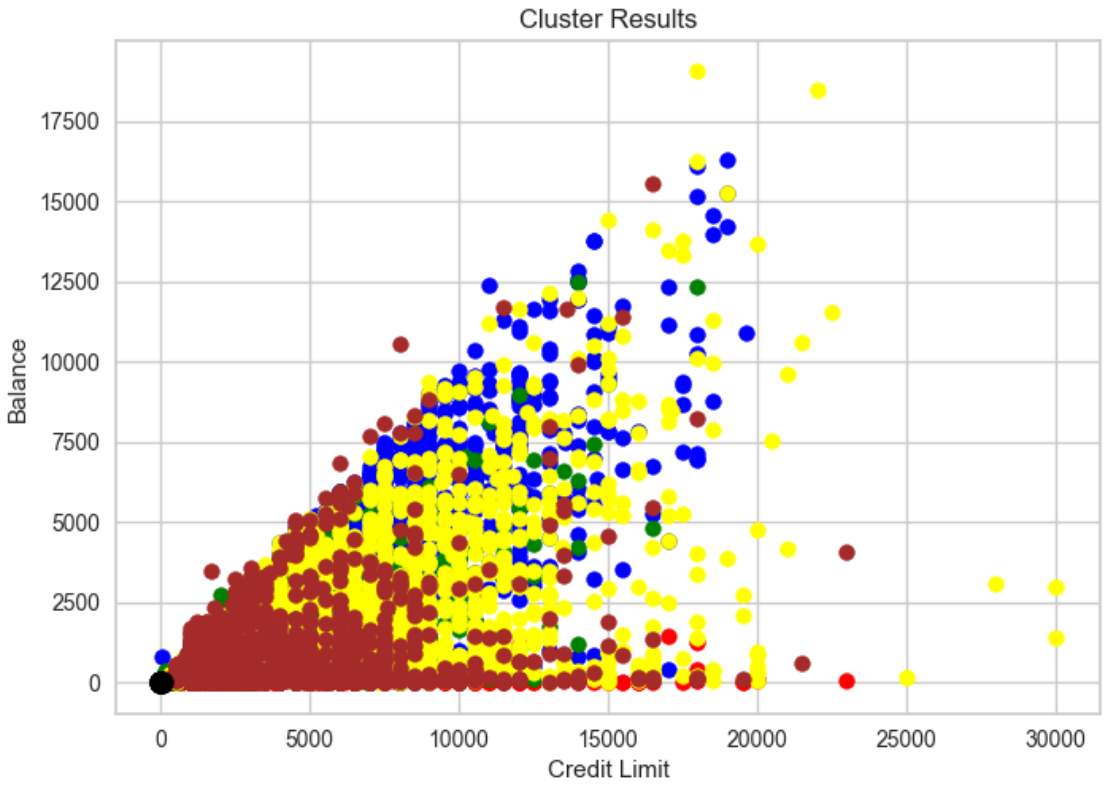
Now, let's check few graphs using the labels we have gotten from KMeans algorithm.

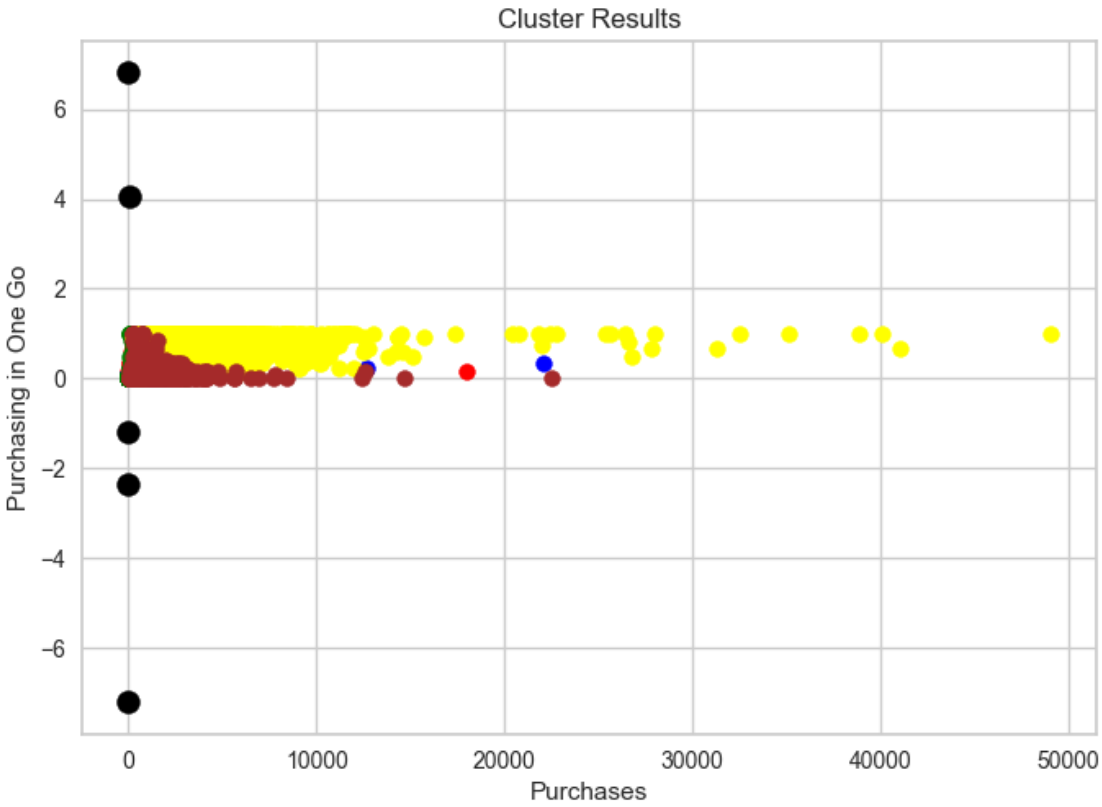
[89]:
```
# KMeans
df = df_cluster.copy()

# Purchases vs. Payments
```

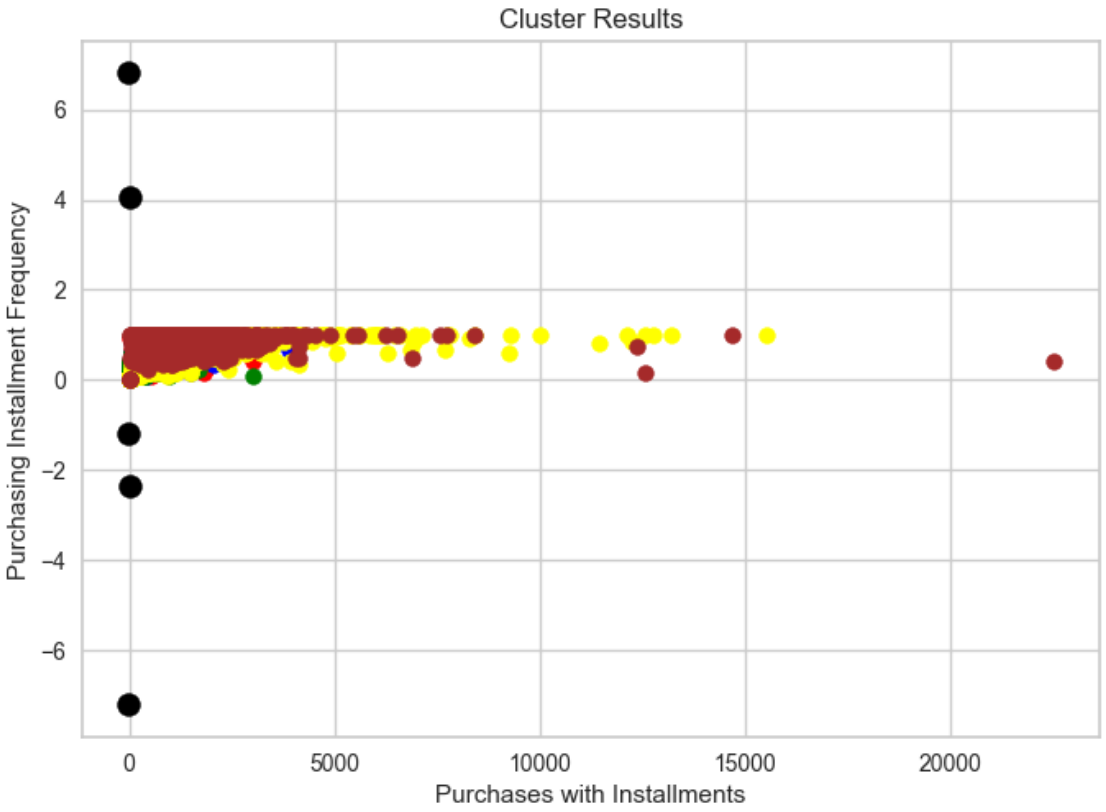
```
plt.scatter(df.iloc[kmeans_labels==0,2], df.iloc[kmeans_labels==0,13], s=50, c=␣
 ↪"red")
plt.scatter(df.iloc[kmeans_labels==1,2], df.iloc[kmeans_labels==1,13], s=50, c=␣
 ↪"blue")
plt.scatter(df.iloc[kmeans_labels==2,2], df.iloc[kmeans_labels==2,13], s=50, c=␣
 ↪"green")
plt.scatter(df.iloc[kmeans_labels==3,2], df.iloc[kmeans_labels==3,13], s=50, c=␣
 ↪"yellow")
plt.scatter(df.iloc[kmeans_labels==4,2], df.iloc[kmeans_labels==4,13], s=50, c=␣
 ↪"brown")
plt.scatter(kmeans.cluster_centers_[:,0],
           kmeans.cluster_centers_[:,1], s=100, c= "black")
plt.title("Cluster Results")
plt.xlabel("Purchases")
plt.ylabel("Payments")
plt.show()

# Credit Limit vs. Balance
plt.scatter(df.iloc[kmeans_labels==0,12], df.iloc[kmeans_labels==0,0], s=50, c=␣
 ↪"red")
plt.scatter(df.iloc[kmeans_labels==1,12], df.iloc[kmeans_labels==1,0], s=50, c=␣
 ↪"blue")
plt.scatter(df.iloc[kmeans_labels==2,12], df.iloc[kmeans_labels==2,0], s=50, c=␣
 ↪"green")
plt.scatter(df.iloc[kmeans_labels==3,12], df.iloc[kmeans_labels==3,0], s=50, c=␣
 ↪"yellow")
plt.scatter(df.iloc[kmeans_labels==4,12], df.iloc[kmeans_labels==4,0], s=50, c=␣
 ↪"brown")
plt.scatter(kmeans.cluster_centers_[:,0],
           kmeans.cluster_centers_[:,1], s=100, c= "black")
plt.title("Cluster Results")
plt.xlabel("Credit Limit")
plt.ylabel("Balance")
plt.show()

# Purchases vs. OnOff Purchases Frequency
plt.scatter(df.iloc[kmeans_labels==0,2], df.iloc[kmeans_labels==0,7], s=50, c=␣
 ↪"red")
plt.scatter(df.iloc[kmeans_labels==1,2], df.iloc[kmeans_labels==1,7], s=50, c=␣
 ↪"blue")
plt.scatter(df.iloc[kmeans_labels==2,2], df.iloc[kmeans_labels==2,7], s=50, c=␣
 ↪"green")
plt.scatter(df.iloc[kmeans_labels==3,2], df.iloc[kmeans_labels==3,7], s=50, c=␣
 ↪"yellow")
plt.scatter(df.iloc[kmeans_labels==4,2], df.iloc[kmeans_labels==4,7], s=50, c=␣
 ↪"brown")
```

```
plt.scatter(kmeans.cluster_centers_[:,0],
            kmeans.cluster_centers_[:,1], s=100, c= "black")
plt.title("Cluster Results")
plt.xlabel("Purchases")
plt.ylabel("Purchasing in One Go")
plt.show()

# Purchases with Installments vs. Installment Purchases Frequency
plt.scatter(df.iloc[kmeans_labels==0,4], df.iloc[kmeans_labels==0,8], s=50, c=␣
 ↪"red")
plt.scatter(df.iloc[kmeans_labels==1,4], df.iloc[kmeans_labels==1,8], s=50, c=␣
 ↪"blue")
plt.scatter(df.iloc[kmeans_labels==2,4], df.iloc[kmeans_labels==2,8], s=50, c=␣
 ↪"green")
plt.scatter(df.iloc[kmeans_labels==3,4], df.iloc[kmeans_labels==3,8], s=50, c=␣
 ↪"yellow")
plt.scatter(df.iloc[kmeans_labels==4,4], df.iloc[kmeans_labels==4,8], s=50, c=␣
 ↪"brown")
plt.scatter(kmeans.cluster_centers_[:,0],
            kmeans.cluster_centers_[:,1], s=100, c= "black")
plt.title("Cluster Results")
plt.xlabel("Purchases with Installments")
plt.ylabel("Purchasing Installment Frequency")
plt.show()
```




Cluster Results
Balance
Credit Limit
17500
15000
12500
10000
7500
5000
2500
0
0
5000
10000
15000
20000
25000
30000


Cluster Results
Purchasing in One Go
Purchases
6
4
2
0
−2
−4
−6
0
10000
20000
30000
40000
50000

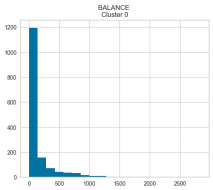


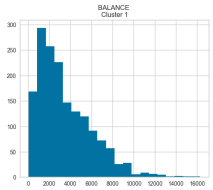
From the graphs, we can understand that, - There are a considerable number of customers who have credit cards for making purchases only and they tend to pay the bank the amounts they use to make purchases. - We can see most customers are making payments according to the credit limits they have and not trying to max out the credit cards. - We can also see most of the purchases are being done in one go. - Similarly, most of the users are not making installment purchases as well.

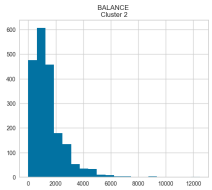
By understanding these points, we can deduce that most of the customers are not maxing out their credit cards and most of them try to pay the credit cards as well.

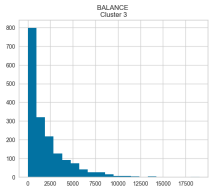
Rather than using these general points, we can find the insights of our clusters as well. The following code plots the clusters and we can check the clusters then and find insights.

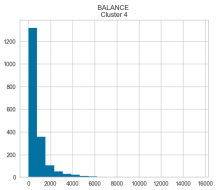
[90]:
```
for i in df_pca_scaled:
    plt.figure(figsize=(35,5))
    for j in range(5):
        plt.subplot(1,5,j+1)
        cluster = df_cluster[df_cluster['kmeans']==j]
        cluster[i].hist(bins=20)
        plt.title(f'{i} \nCluster {j}')
    plt.show()
```

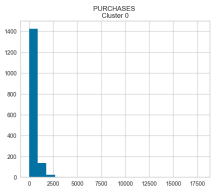

BALANCE
Cluster 0

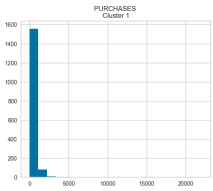

BALANCE
Cluster 1

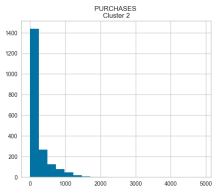

BALANCE
Cluster 2

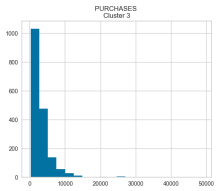

BALANCE
Cluster 3

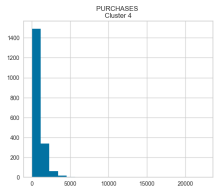

BALANCE
Cluster 4

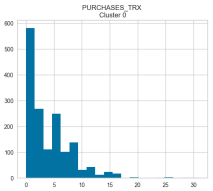

PURCHASES
Cluster 0

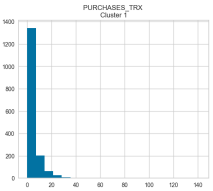

PURCHASES
Cluster 1

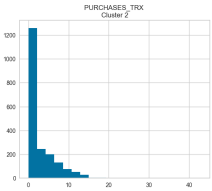

PURCHASES
Cluster 2

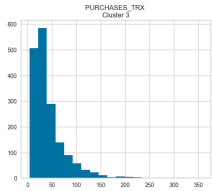

PURCHASES
Cluster 3

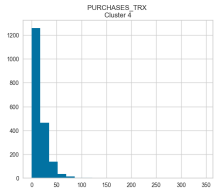

PURCHASES
Cluster 4

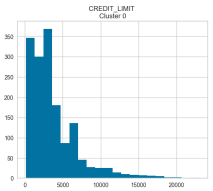

PURCHASES_TRX
Cluster 0

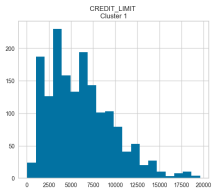

PURCHASES_TRX
Cluster 1

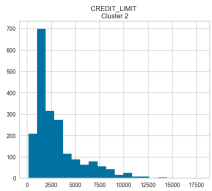

PURCHASES_TRX
Cluster 2


PURCHASES_TRX
Cluster 3


PURCHASES_TRX
Cluster 4

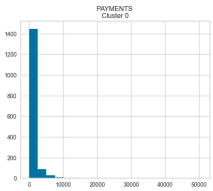

CREDIT_LIMIT
Cluster 0

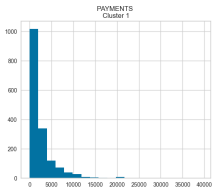

CREDIT_LIMIT
Cluster 1

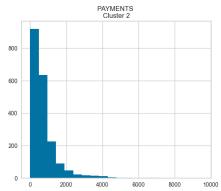

CREDIT_LIMIT
Cluster 2

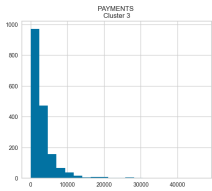

CREDIT_LIMIT
Cluster 3

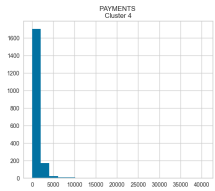

CREDIT_LIMIT
Cluster 4

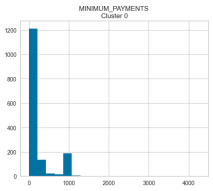

PAYMENTS
Cluster 0

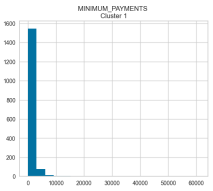

PAYMENTS
Cluster 1

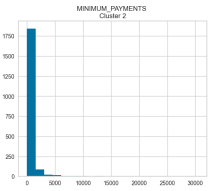

PAYMENTS
Cluster 2

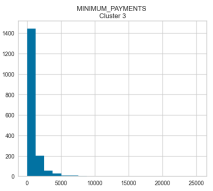

PAYMENTS
Cluster 3

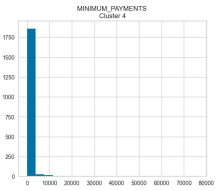

PAYMENTS
Cluster 4

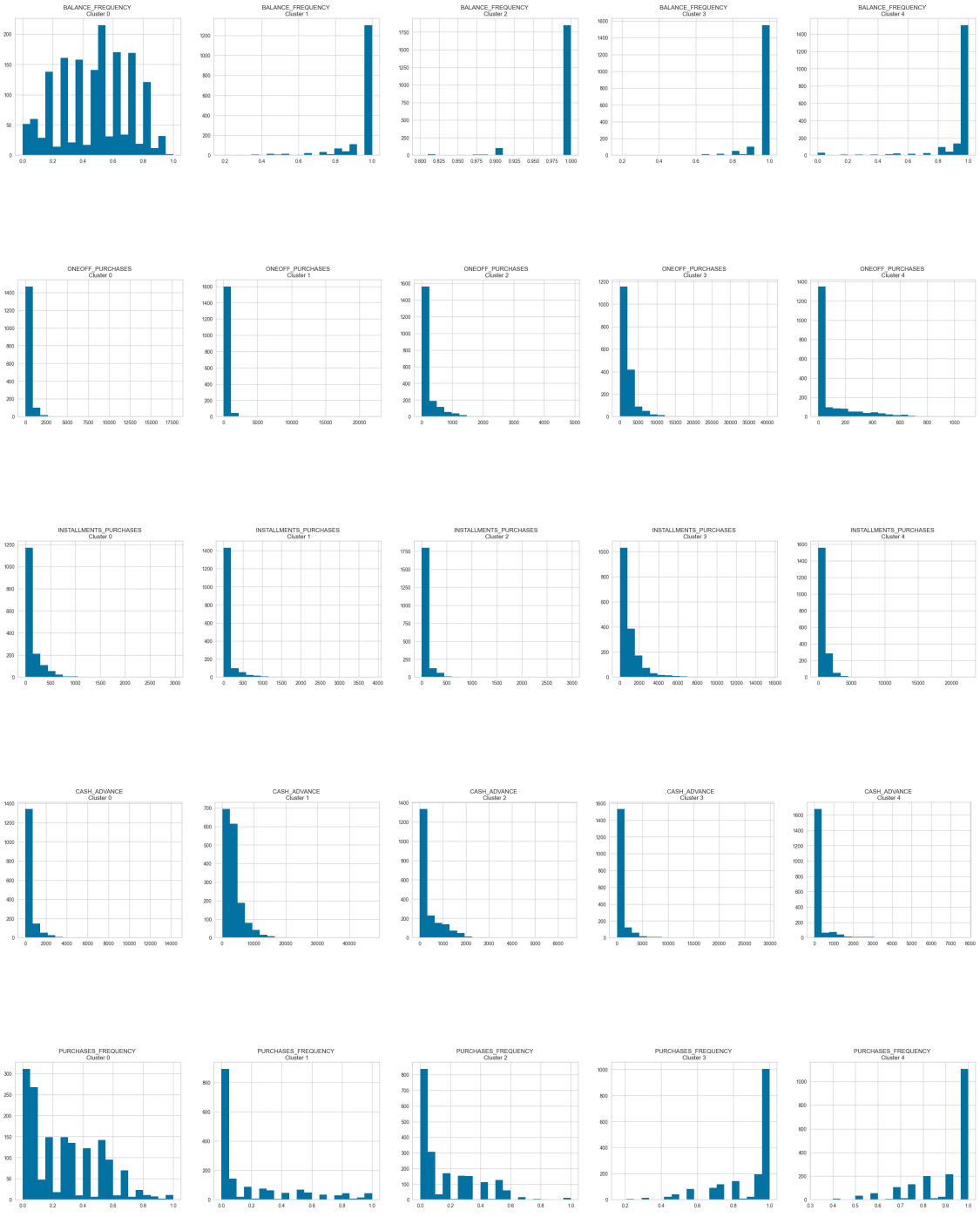

MINIMUM_PAYMENTS
Cluster 0

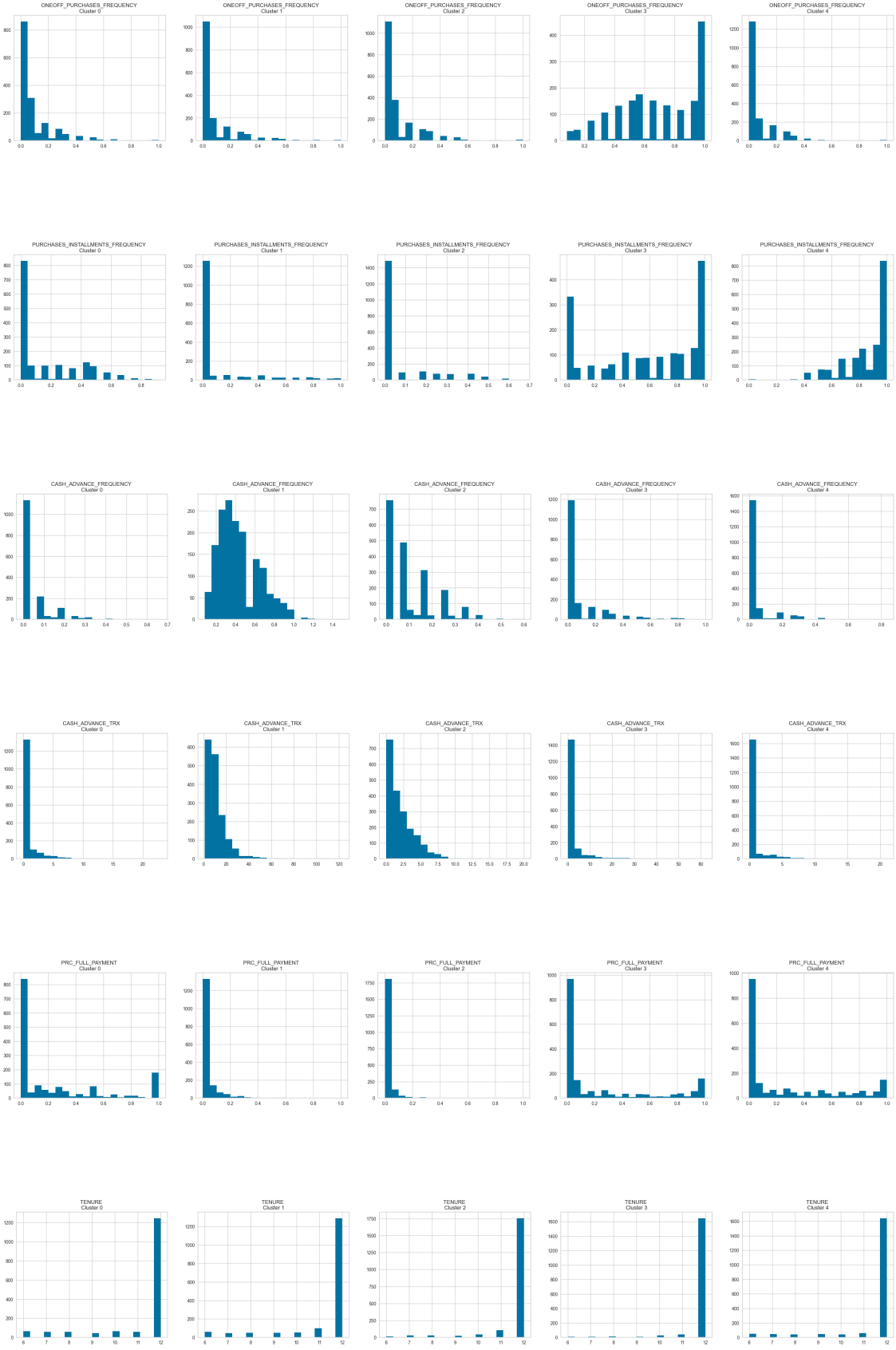

MINIMUM_PAYMENTS
Cluster 1


MINIMUM_PAYMENTS
Cluster 2


MINIMUM_PAYMENTS
Cluster 3


MINIMUM_PAYMENTS
Cluster 4


BALANCE_FREQUENCY
Cluster 0
BALANCE_FREQUENCY
Cluster 1
BALANCE_FREQUENCY
Cluster 2
BALANCE_FREQUENCY
Cluster 3
BALANCE_FREQUENCY
Cluster 4
ONEOFF_PURCHASES
Cluster 0
ONEOFF_PURCHASES
Cluster 1
ONEOFF_PURCHASES
Cluster 2
ONEOFF_PURCHASES
Cluster 3
ONEOFF_PURCHASES
Cluster 4
INSTALLMENTS_PURCHASES
Cluster 0
INSTALLMENTS_PURCHASES
Cluster 1
INSTALLMENTS_PURCHASES
Cluster 2
INSTALLMENTS_PURCHASES
Cluster 3
INSTALLMENTS_PURCHASES
Cluster 4
CASH_ADVANCE
Cluster 0
CASH_ADVANCE
Cluster 1
CASH_ADVANCE
Cluster 2
CASH_ADVANCE
Cluster 3
CASH_ADVANCE
Cluster 4
PURCHASES_FREQUENCY
Cluster 0
PURCHASES_FREQUENCY
Cluster 1
PURCHASES_FREQUENCY
Cluster 2
PURCHASES_FREQUENCY
Cluster 3
PURCHASES_FREQUENCY
Cluster 4


ONEOFF_PURCHASES_FREQUENCY
Cluster 0
ONEOFF_PURCHASES_FREQUENCY
Cluster 1
ONEOFF_PURCHASES_FREQUENCY
Cluster 2
ONEOFF_PURCHASES_FREQUENCY
Cluster 3
ONEOFF_PURCHASES_FREQUENCY
Cluster 4
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 0
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 1
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 2
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 3
PURCHASES_INSTALLMENTS_FREQUENCY
Cluster 4
CASH_ADVANCE_FREQUENCY
Cluster 0
CASH_ADVANCE_FREQUENCY
Cluster 1
CASH_ADVANCE_FREQUENCY
Cluster 2
CASH_ADVANCE_FREQUENCY
Cluster 3
CASH_ADVANCE_FREQUENCY
Cluster 4
CASH_ADVANCE_TRX
Cluster 0
CASH_ADVANCE_TRX
Cluster 1
CASH_ADVANCE_TRX
Cluster 2
CASH_ADVANCE_TRX
Cluster 3
CASH_ADVANCE_TRX
Cluster 4
PRC_FULL_PAYMENT
Cluster 0
PRC_FULL_PAYMENT
Cluster 1
PRC_FULL_PAYMENT
Cluster 2
PRC_FULL_PAYMENT
Cluster 3
PRC_FULL_PAYMENT
Cluster 4
TENURE
Cluster 0
TENURE
Cluster 1
TENURE
Cluster 2
TENURE
Cluster 3
TENURE
Cluster 4

The cluster details are as follows.

Cluster 0 - Low high balance - Low cash advance - High purchase frequency - High purchase installment frequency - Low cash advance frequency - Low cash advance transaction - High minimum payments - Low full payment - High tenure

The people in cluster 0 are making purchases frequently and make low cash advances utilizing credit cards to the maximum.

Cluster 1 - Low high balance - Low cash advance - Little purchase frequency - Low purchase installment frequency - Low cash advance frequency - Low cash advance transaction - Little minimum payments - Low full payment - High tenure

The people in cluster 1 are not making purchases and they don't have a high balance, these can be transactors, who are using the credit cards carefully.

Cluster 2 - Low high balance - Low cash advance - Low purchase frequency - Low purchase installment frequency - High cash advance frequency - Little advance transaction - High minimum payments - Low full payment - High tenure

The people in cluster 2 are making high minimum payments and have high cash advance frequencies. Therefore we can assume these are frequent credit card users. These people can be VIP members who are using credit cards for every transaction and make sure to fill the credit card without maxing it out.

Cluster 3 - High high balance - Low cash advance - Low purchase frequency - High purchase installment frequency - High cash advance frequency - Low cash advance transaction - High minimum payments - Low full payment - High tenure

The people in cluster 3 have high balances and high minimum payments. These can be travelers who have gotten credit cards for the offers.

Cluster 4 - Low high balance - Low cash advance - High purchase frequency - High purchase installment frequency - Low cash advance frequency - Low cash advance transaction - High minimum payments - Little full payment - High tenure

The people in cluster 4 have a low high balance, high purchases, and high purchase frequencies. These can be those who are mainly using credit cards for shopping, and for making big purchases.

2 References

1. Géron, A. (2019). Hands-on machine learning with Scikit-Learn and TensorFlow concepts, tools, and techniques to build intelligent systems. 2nd ed. O'Reilly Media, Inc.
2. Witten, I.H. and Al, E. (2017). Data mining : practical machine learning tools and techniques. Amsterdam: Morgan Kaufmann.
3. Priy, S. (2018). Clustering in Machine Learning - GeeksforGeeks. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/clustering-in-machine-learning/.
4. Zaki, M.J. and Wagner Meira (2020). Data mining and machine learning : fundamental concepts and algorithms. Cambridge, United Kingdom ; New York, Ny: Cambridge University Press.

5. Everitt, B. and Al, E. (2011). Cluster analysis. Chichester, West Sussex, U.K.: Wiley.

[ ]: