MINOR PROJECT : DATA SCIENCE JANUARY MINOR PROJECT DONE BY: RITHU

```
import numpy as np
import pandas as pd

from matplotlib import pyplot as plt
import seaborn as sns
%matplotlib inline

from matplotlib import style
style.use('ggplot')

# to suppress warningss
from warnings import filterwarnings
filterwarnings('ignore')

# setting the plot size for graphs:
plt.rcParams['figure.figsize'] = (8,6)

#importing data
df = pd.read_csv('credit_card.csv')
df.head()
```

```
  CUST_ID      BALANCE  ...  PRC_FULL_PAYMENT  TENURE
0  C10001    40.900749  ...          0.000000      12
1  C10002  3202.467416  ...          0.222222      12
2  C10003  2495.148862  ...          0.000000      12
3  C10004  1666.670542  ...          0.000000      12
4  C10005   817.714335  ...          0.000000      12

[5 rows x 18 columns]
```

```
#DATA EXPLORATION

# shape and info of the data
df.shape
```

```
(8950, 18)
```

```
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

# Summary Statistics for  the Numerical Variables:
df.describe()

            BALANCE  BALANCE_FREQUENCY  ...  PRC_FULL_PAYMENT
TENURE
count   8950.000000        8950.000000  ...       8950.000000
8950.000000
mean    1564.474828           0.877271  ...          0.153715
11.517318
std     2081.531879           0.236904  ...          0.292499
1.338331
min        0.000000           0.000000  ...          0.000000
6.000000
25%      128.281915           0.888889  ...          0.000000
12.000000
50%      873.385231           1.000000  ...          0.000000
12.000000
75%     2054.140036           1.000000  ...          0.142857
12.000000
max    19043.138560           1.000000  ...          1.000000
12.000000

[8 rows x 17 columns]

# Data Summary for continuous variables:

def var_summary(x):
    return pd.Series([x.count(), x.isnull().sum(), x.sum(), x.mean(),
x.median(), x.std(), x.var(), x.min(),
        x.quantile(0.01), x.quantile(0.05),
x.quantile(0.10),x.quantile(0.25),x.quantile(0.50),x.quantile(0.75),
                              x.quantile(0.90),x.quantile(0.95),
x.quantile(0.99),x.max()],

                  index = ['N', 'NMISS', 'SUM', 'MEAN','MEDIAN',
```

```
'STD', 'VAR', 'MIN', 'P1',
                              'P5' ,'P10' ,'P25' ,'P50' ,'P75' ,'P90'
,'P95' ,'P99' ,'MAX'])

num_features = df.select_dtypes([np.number])

num_features.apply(var_summary).T
```

```
                                       N  NMISS  ...           P99
MAX
BALANCE                           8950.0    0.0  ...   9338.804814
19043.13856
BALANCE_FREQUENCY                 8950.0    0.0  ...      1.000000
1.00000
PURCHASES                         8950.0    0.0  ...   8977.290000
49039.57000
ONEOFF_PURCHASES                  8950.0    0.0  ...   6689.898200
40761.25000
INSTALLMENTS_PURCHASES            8950.0    0.0  ...   3886.240500
22500.00000
CASH_ADVANCE                      8950.0    0.0  ...   9588.163357
47137.21176
PURCHASES_FREQUENCY               8950.0    0.0  ...      1.000000
1.00000
ONEOFF_PURCHASES_FREQUENCY        8950.0    0.0  ...      1.000000
1.00000
PURCHASES_INSTALLMENTS_FREQUENCY  8950.0    0.0  ...      1.000000
1.00000
CASH_ADVANCE_FREQUENCY            8950.0    0.0  ...      0.833333
1.50000
CASH_ADVANCE_TRX                  8950.0    0.0  ...     29.000000
123.00000
PURCHASES_TRX                     8950.0    0.0  ...    116.510000
358.00000
CREDIT_LIMIT                      8949.0    1.0  ...  17000.000000
30000.00000
PAYMENTS                          8950.0    0.0  ...  13608.715541
50721.48336
MINIMUM_PAYMENTS                  8637.0  313.0  ...   9034.098737
76406.20752
PRC_FULL_PAYMENT                  8950.0    0.0  ...      1.000000
1.00000
TENURE                            8950.0    0.0  ...     12.000000
12.00000

[17 rows x 18 columns]
```

```
# Summary Statistics for Categorical Variables:

df.describe(exclude=[np.number])
```

```
       CUST_ID
count     8950
unique    8950
top     C10001
freq         1
```

```
# dropping Customer Id as is unique and not needed for model building:
df.drop('CUST_ID', axis=1, inplace=True)


# Checking for Missing Values


count_missing = df.isnull().sum()

percent_missing = (df.isnull().sum()/len(df))*100

missing_values = pd.concat([percent_missing,count_missing], axis=1,
                           keys=['Percent_of_Missing_Values',
'Count_of_Missing_Values'])
missing_values
```

```
                                  Percent_of_Missing_Values
Count_of_Missing_Values
BALANCE                                            0.000000
0
BALANCE_FREQUENCY                                  0.000000
0
PURCHASES                                          0.000000
0
ONEOFF_PURCHASES                                   0.000000
0
INSTALLMENTS_PURCHASES                             0.000000
0
CASH_ADVANCE                                       0.000000
0
PURCHASES_FREQUENCY                                0.000000
0
ONEOFF_PURCHASES_FREQUENCY                         0.000000
0
PURCHASES_INSTALLMENTS_FREQUENCY                   0.000000
0
CASH_ADVANCE_FREQUENCY                             0.000000
0
CASH_ADVANCE_TRX                                   0.000000
0
PURCHASES_TRX                                      0.000000
0
CREDIT_LIMIT                                       0.011173
1
```

```
PAYMENTS                                                   0.000000
0
MINIMUM_PAYMENTS                                           3.497207
313
PRC_FULL_PAYMENT                                           0.000000
0
TENURE                                                     0.000000
0
```

```
# checking the value which is Null for Credit Limit
df[df['CREDIT_LIMIT'].isnull()]
```

```
        BALANCE  BALANCE_FREQUENCY  ...  PRC_FULL_PAYMENT  TENURE
5203  18.400472           0.166667  ...               0.0       6

[1 rows x 17 columns]
```

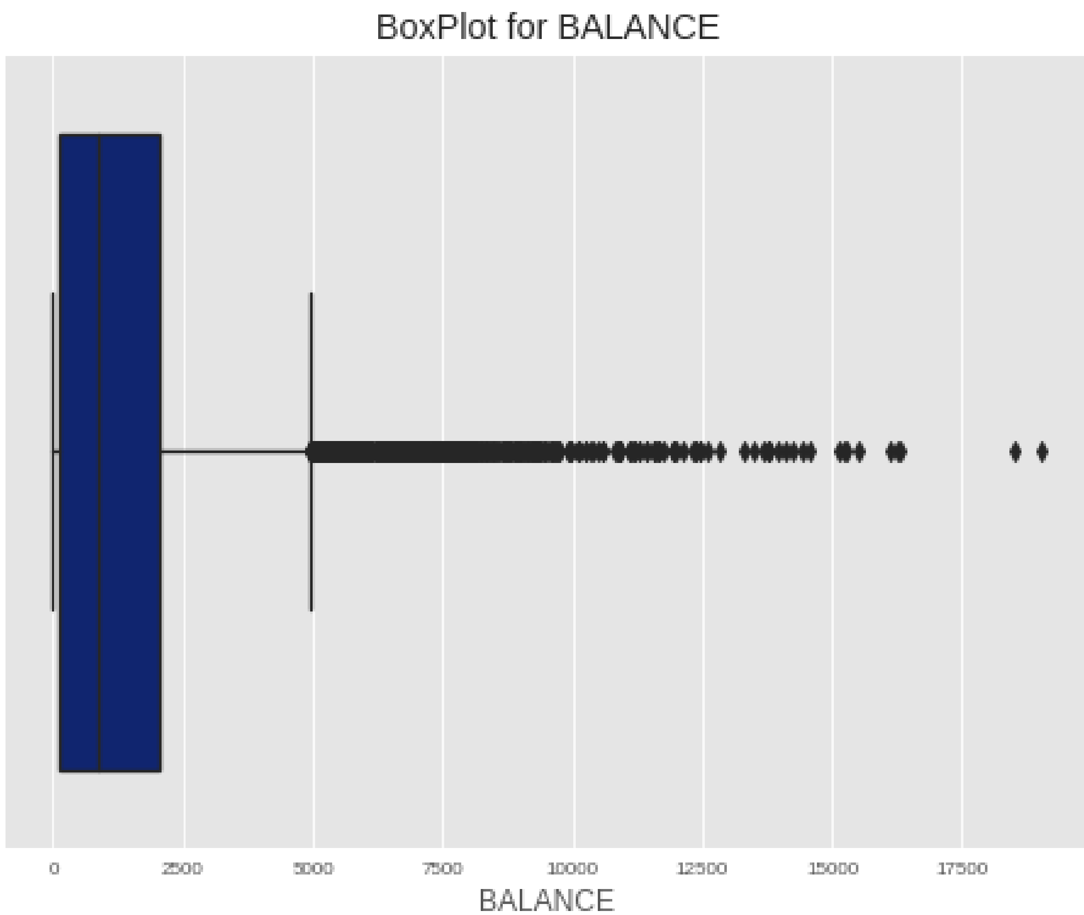
```
# dropping off the missing value for Credit Limit

df = df.drop(5203)

# resetting the index after dropping the record:
df = df.reset_index(drop=True)

# Impute Using Median for Minimum Payments

df['MINIMUM_PAYMENTS']=
df['MINIMUM_PAYMENTS'].fillna(df['MINIMUM_PAYMENTS'].median())

# Checking again to confirm if missing values are present or not:
df.isnull().sum()
```

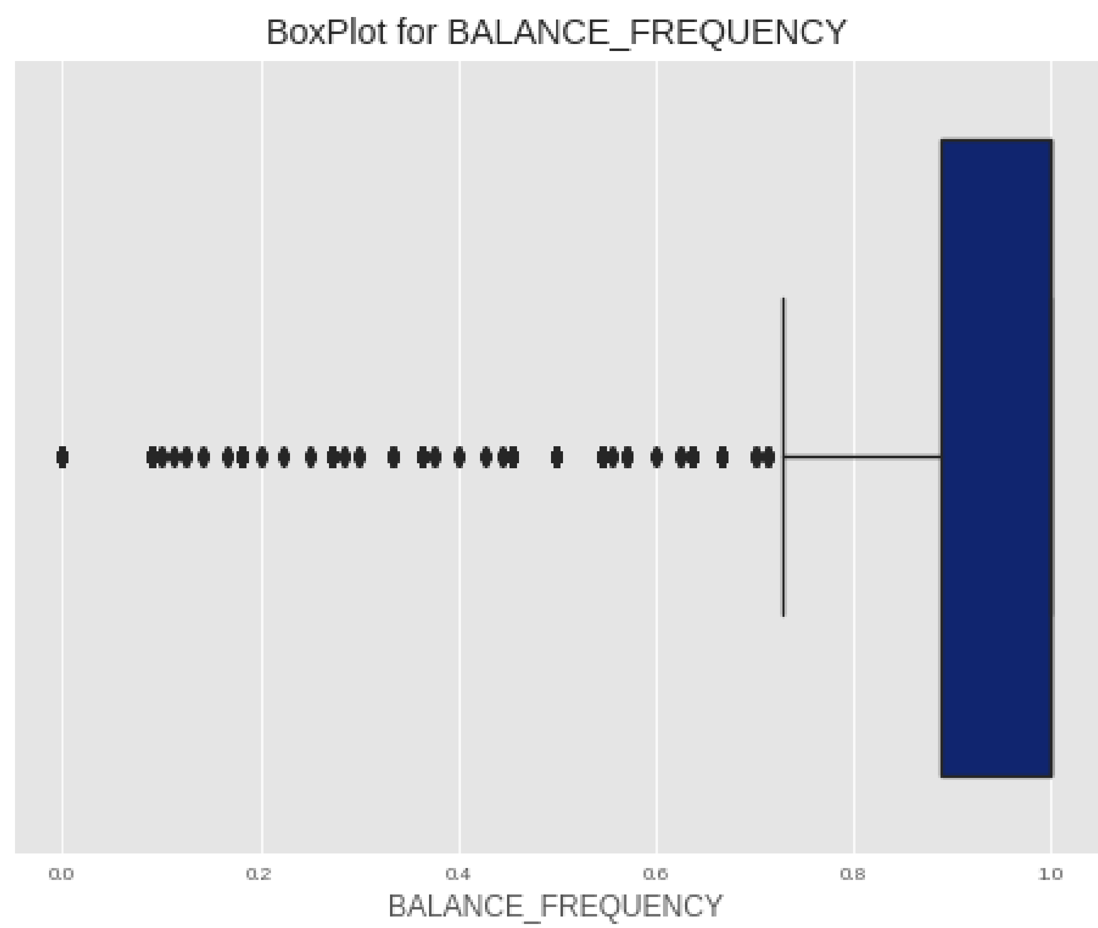
```
BALANCE                             0
BALANCE_FREQUENCY                   0
PURCHASES                           0
ONEOFF_PURCHASES                    0
INSTALLMENTS_PURCHASES              0
CASH_ADVANCE                        0
PURCHASES_FREQUENCY                 0
ONEOFF_PURCHASES_FREQUENCY          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
```

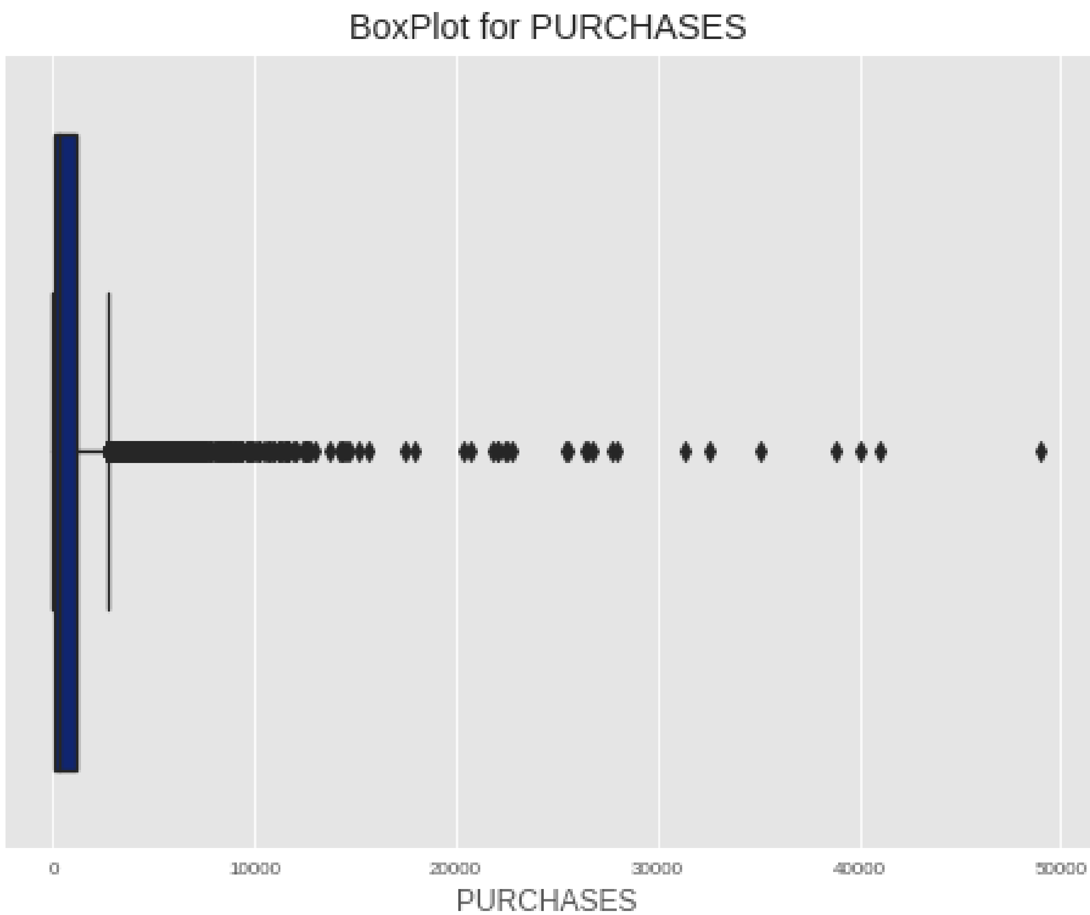
```
#To check for outliers of numerical columns, plotting box plot for
each of the variable.

num_vars = df.columns
num_vars
```

```
Index(['BALANCE', 'BALANCE_FREQUENCY', 'PURCHASES',
'ONEOFF_PURCHASES',
       'INSTALLMENTS_PURCHASES', 'CASH_ADVANCE',
'PURCHASES_FREQUENCY',
       'ONEOFF_PURCHASES_FREQUENCY',
'PURCHASES_INSTALLMENTS_FREQUENCY',
       'CASH_ADVANCE_FREQUENCY', 'CASH_ADVANCE_TRX', 'PURCHASES_TRX',
       'CREDIT_LIMIT', 'PAYMENTS', 'MINIMUM_PAYMENTS',
'PRC_FULL_PAYMENT',
       'TENURE'],
      dtype='object')
```

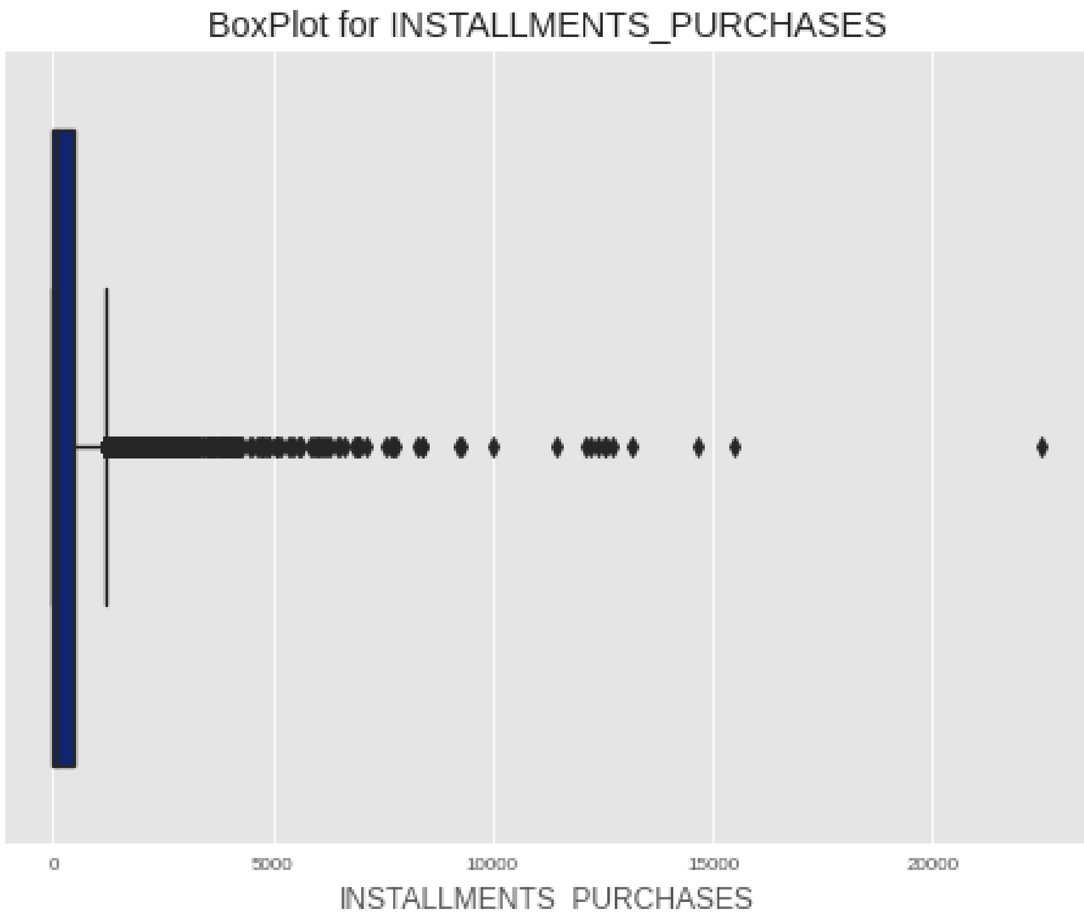
```
# Box Plot:
for i in num_vars:
    sns.boxplot(df[i], palette='dark')
    plt.title('BoxPlot for {}'.format(i))
    plt.show()
```

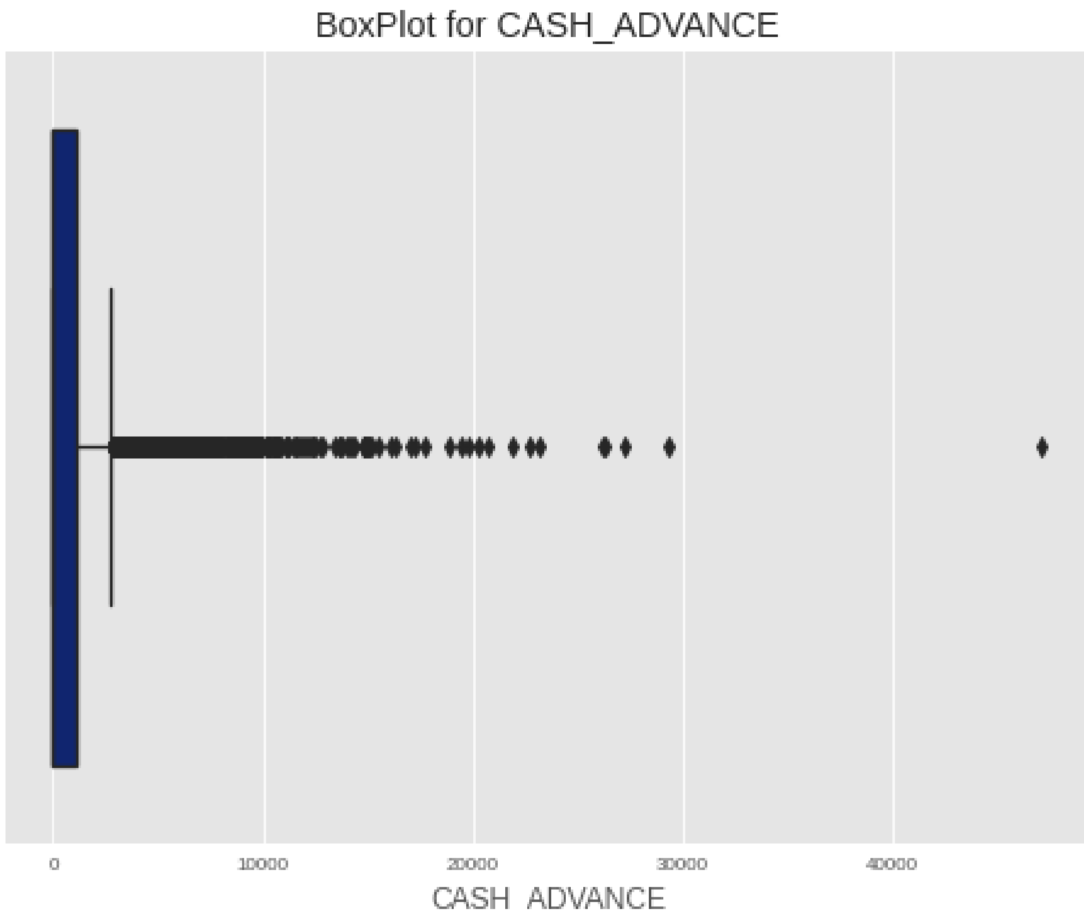

BoxPlot for BALANCE
0
2500
5000
7500
10000
12500
15000
17500
BALANCE

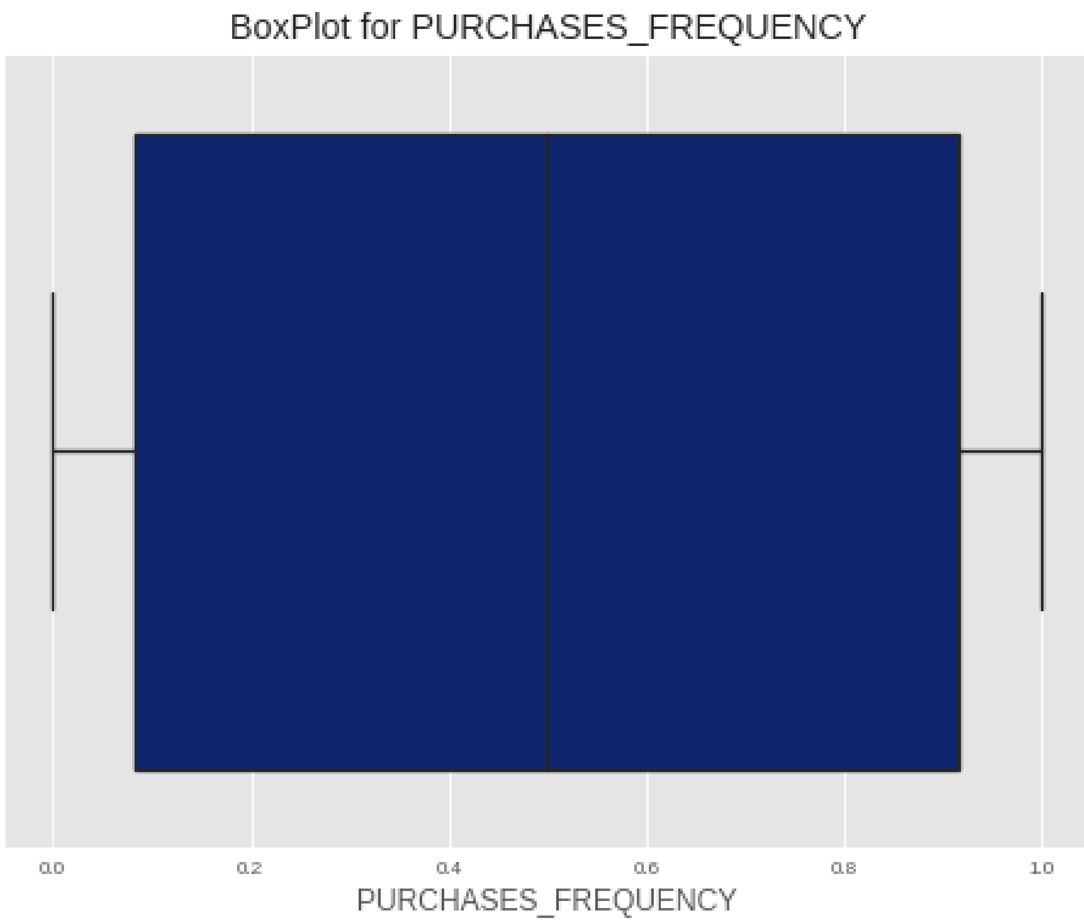

BoxPlot for BALANCE_FREQUENCY
0.0
0.2
0.4
0.6
0.8
1.0
BALANCE_FREQUENCY

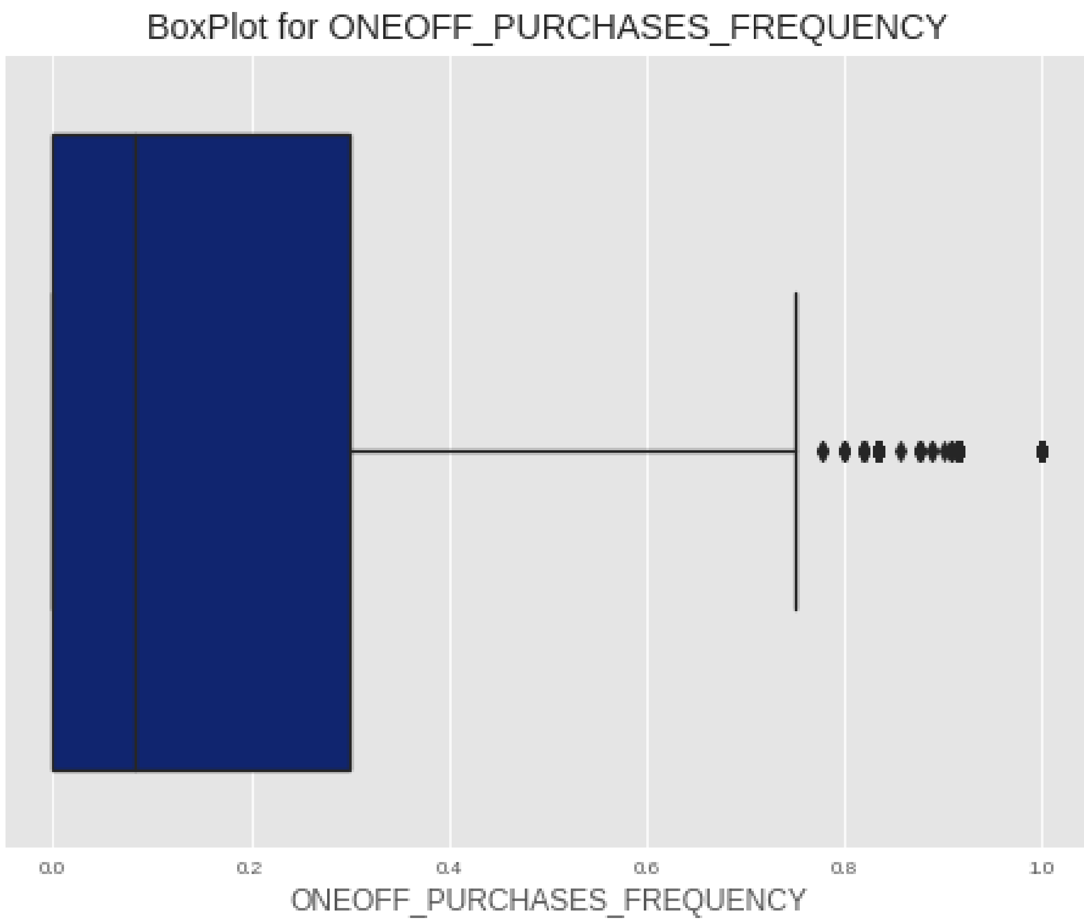

BoxPlot for PURCHASES
0
10000
20000
30000
40000
50000
PURCHASES

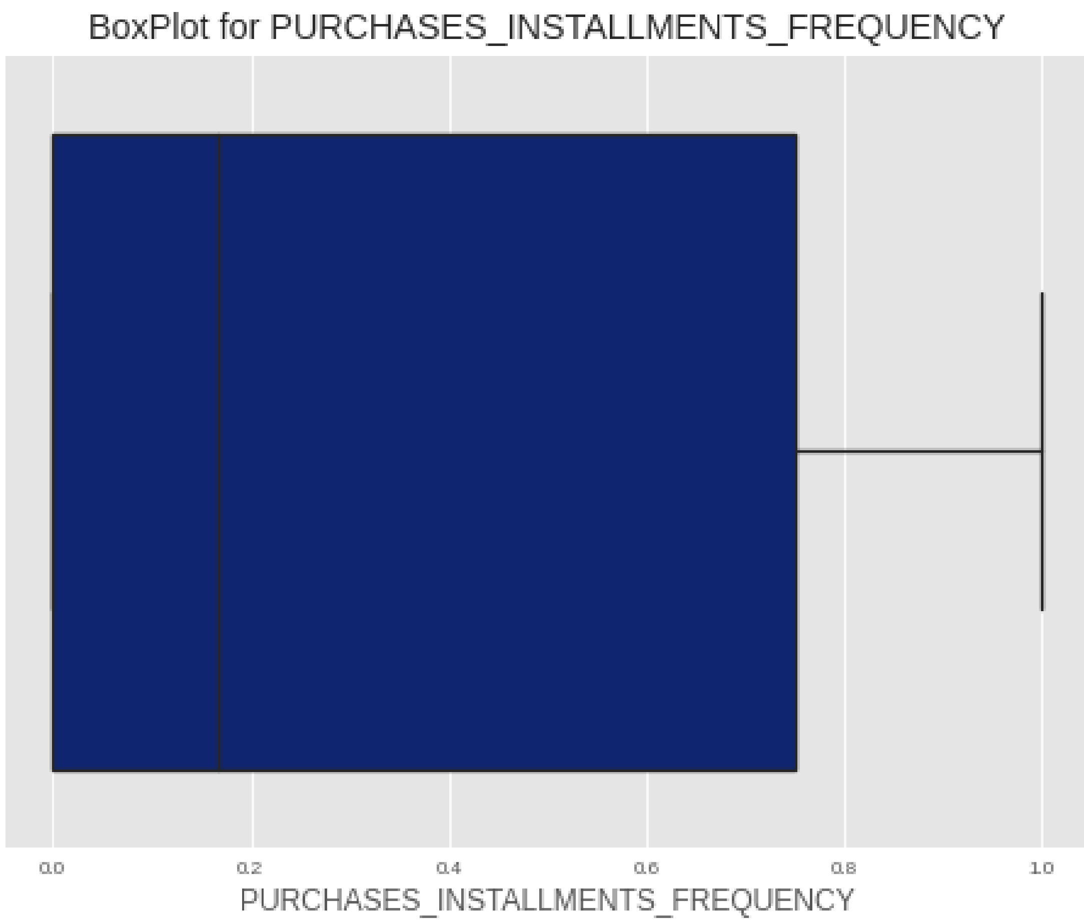

BoxPlot for ONEOFF_PURCHASES
0
5000
10000
15000
20000
25000
30000
35000
40000
ONEOFF_PURCHASES

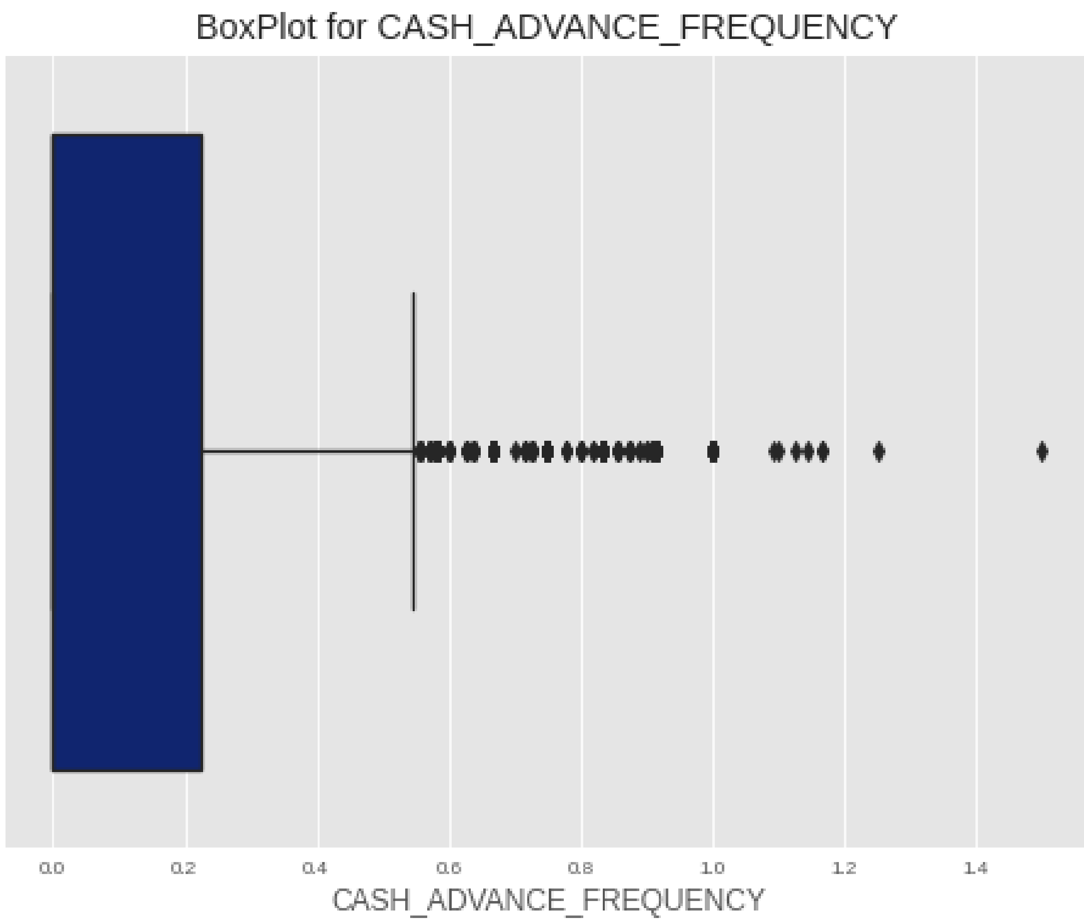

BoxPlot for INSTALLMENTS_PURCHASES
0
5000
10000
15000
20000
INSTALLMENTS_PURCHASES

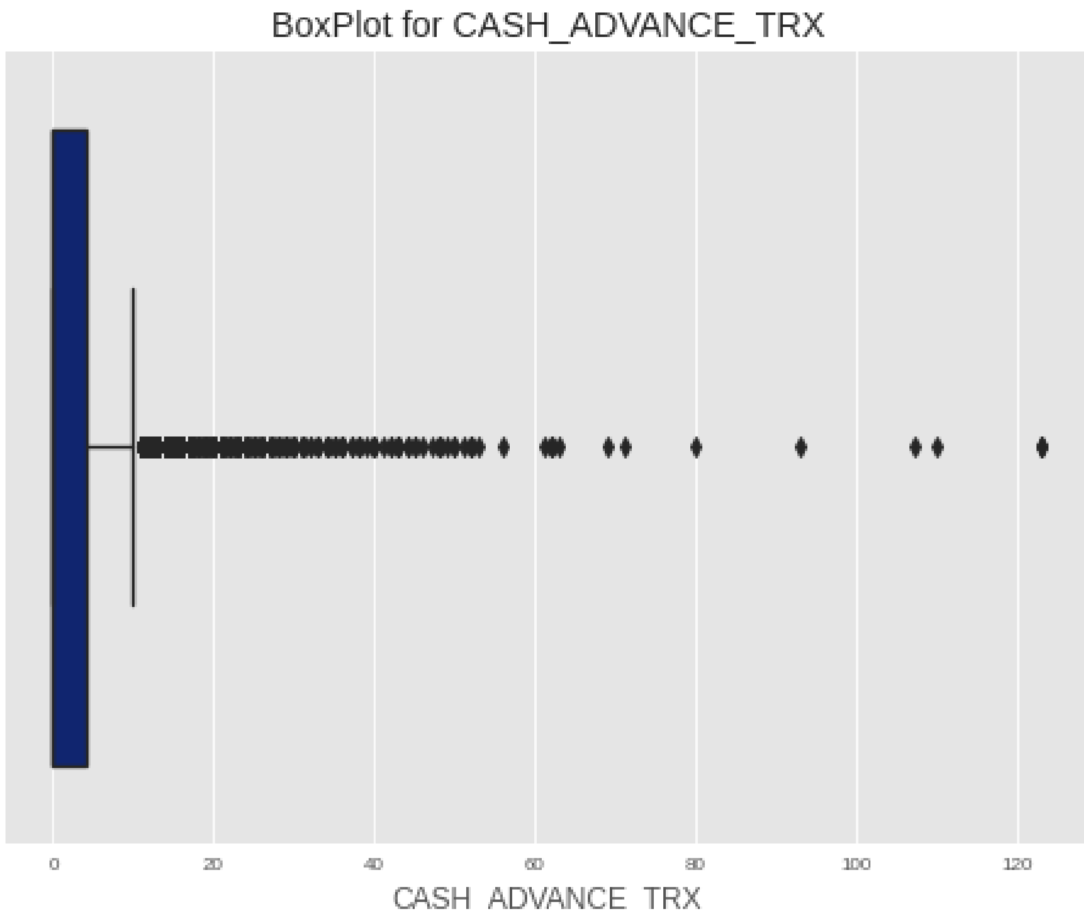

BoxPlot for CASH_ADVANCE
0
10000
20000
30000
40000
CASH_ADVANCE


BoxPlot for PURCHASES_FREQUENCY
0.0
0.2
0.4
0.6
0.8
1.0
PURCHASES_FREQUENCY

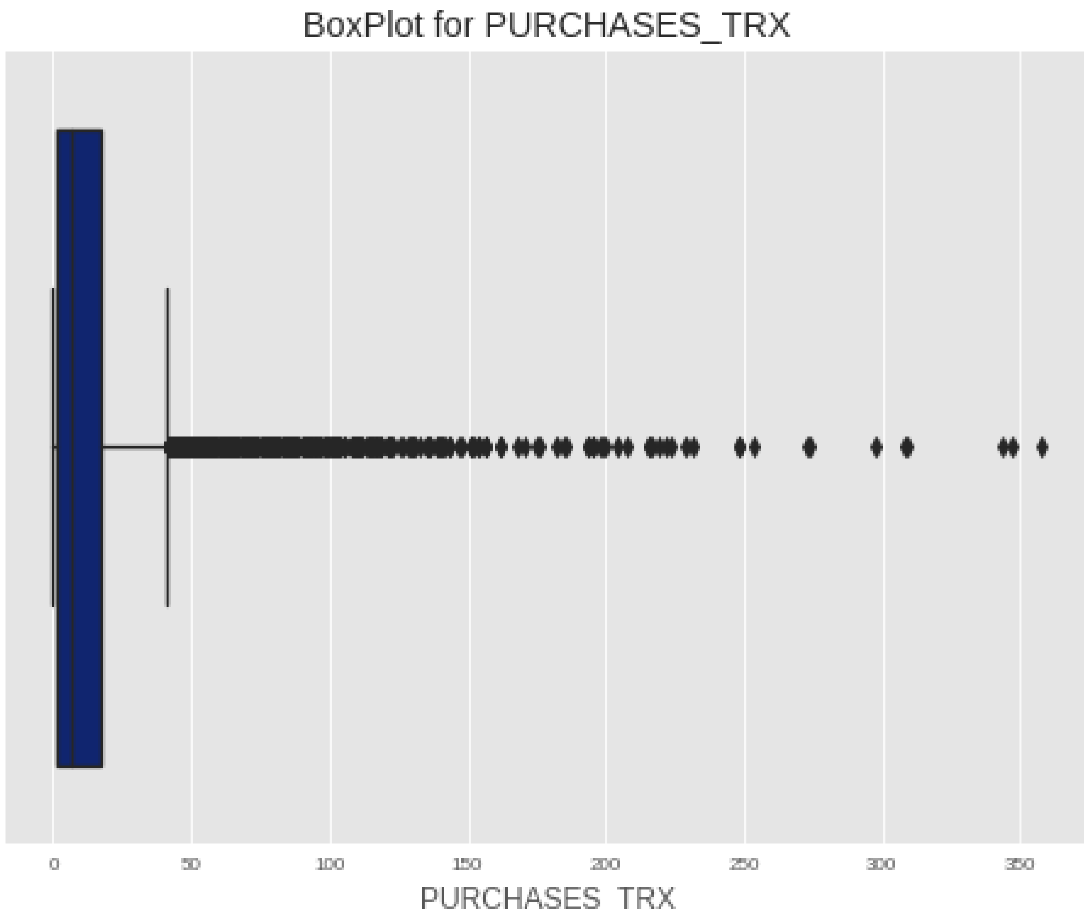

BoxPlot for ONEOFF_PURCHASES_FREQUENCY
0.0
0.2
0.4
0.6
0.8
1.0
ONEOFF_PURCHASES_FREQUENCY


BoxPlot for PURCHASES_INSTALLMENTS_FREQUENCY
0.0
0.2
0.4
0.6
0.8
1.0
PURCHASES_INSTALLMENTS_FREQUENCY

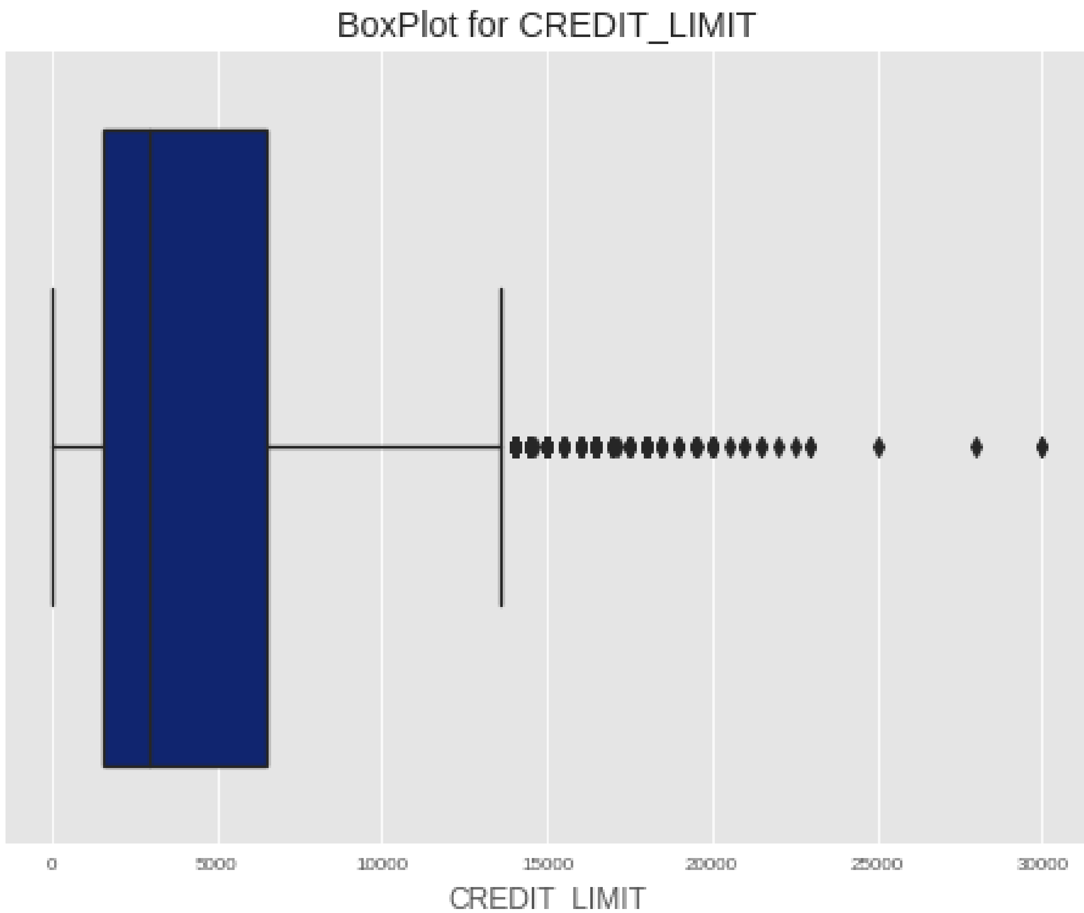

BoxPlot for CASH_ADVANCE_FREQUENCY
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
CASH_ADVANCE_FREQUENCY

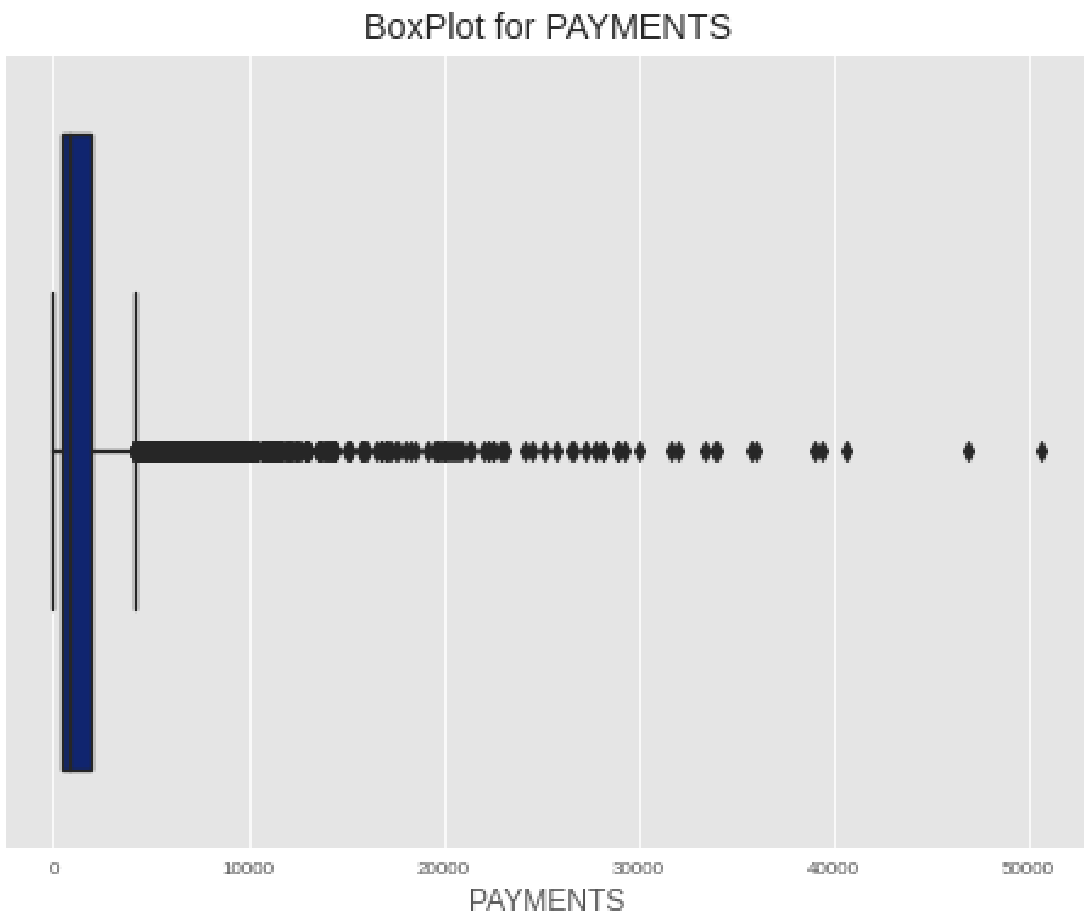


BoxPlot for CASH_ADVANCE_TRX
0
20
40
60
80
100
120
CASH_ADVANCE_TRX

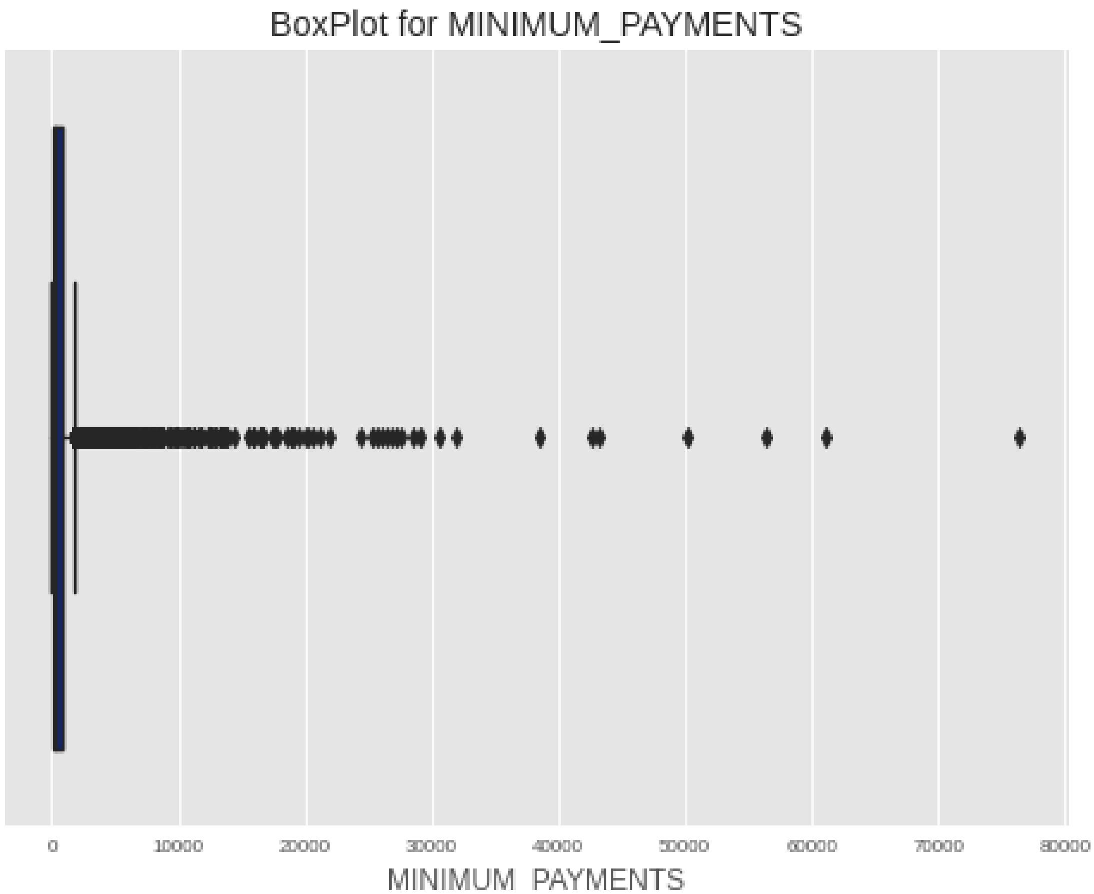


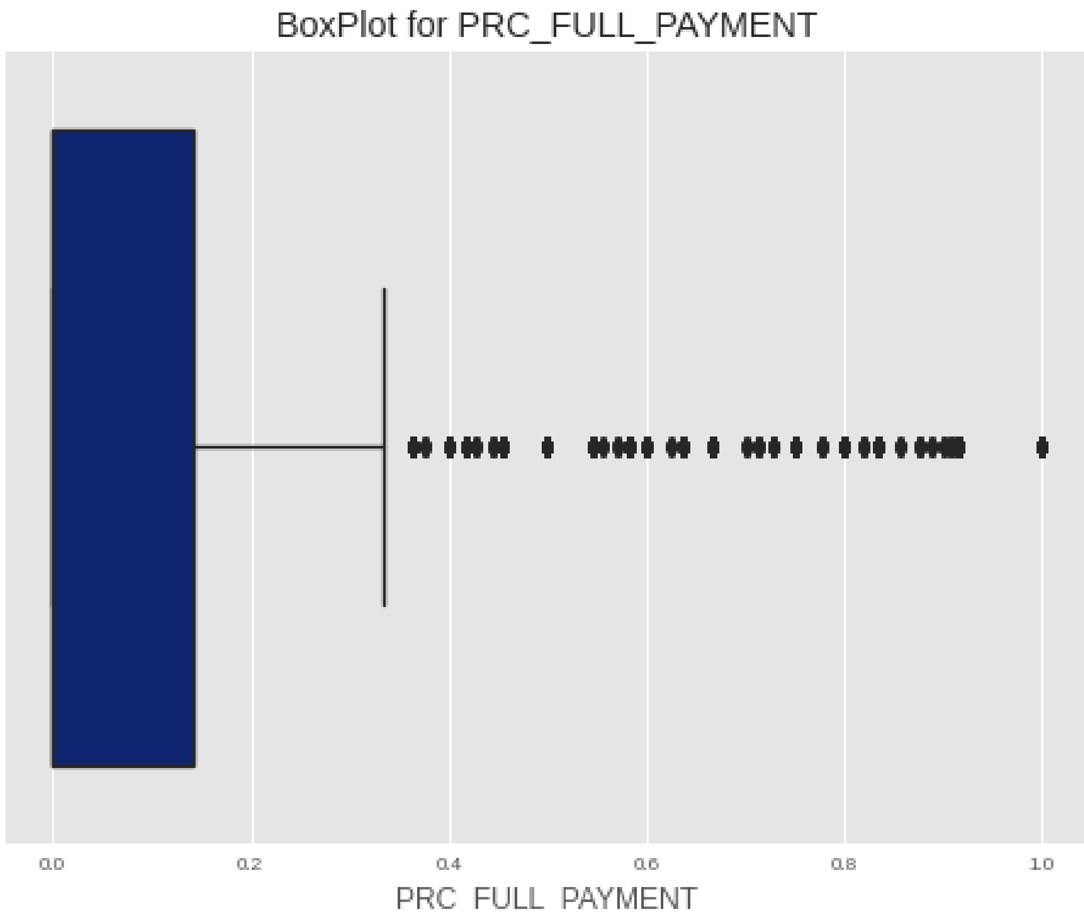
BoxPlot for PURCHASES_TRX
0
50
100
150
200
250
300
350
PURCHASES_TRX

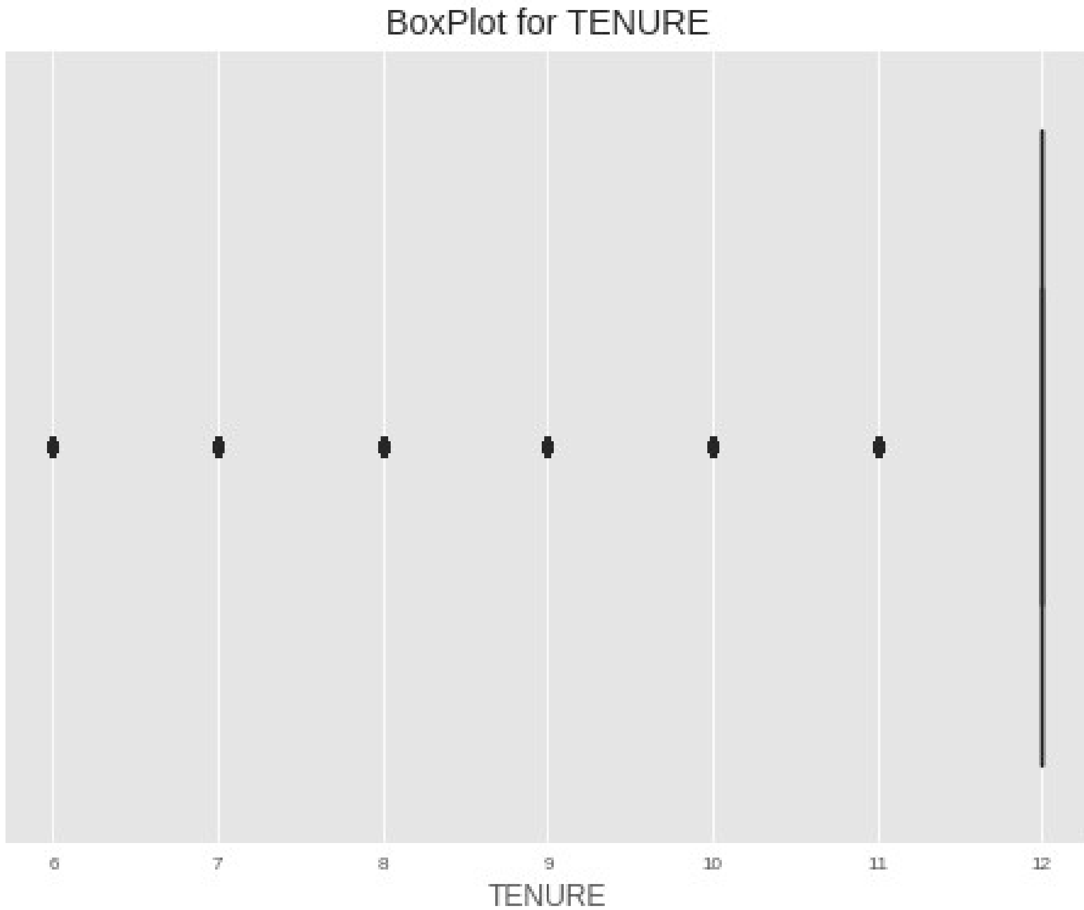

BoxPlot for CREDIT_LIMIT
0
5000
10000
15000
20000
25000
30000
CREDIT_LIMIT



BoxPlot for PAYMENTS
0
10000
20000
30000
40000
50000
PAYMENTS


BoxPlot for MINIMUM_PAYMENTS
0
10000
20000
30000
40000
50000
60000
70000
80000
MINIMUM_PAYMENTS


BoxPlot for PRC_FULL_PAYMENT
0.0
0.2
0.4
0.6
0.8
1.0
PRC_FULL_PAYMENT




```
#From the above boxplots, can see there are outliers present.


# transforming the data by applying PowerTransformer to treat for
outliers.
#By using IQR Method or Z-score to cap the outliers would have deleted
those respective values.
#Hence, transforming the data.

from sklearn.preprocessing import PowerTransformer

PT = PowerTransformer()

print(PT.fit_transform(df))

[[-1.23833786 -1.0801604  -0.36831098 ... -0.82502551 -0.67793662
   0.42210751]
 [ 1.05188287 -0.4256199  -1.50536123 ...  0.91748237  1.23484635
   0.42210751]
 [ 0.86050618  0.62852726  0.52149237 ...  0.4759187  -0.67793662
   0.42210751]
 ...
```

```
 [-1.40957025 -0.9921333  -0.21655169 ... -1.30177584  1.32828513
  -2.52719186]
 [-1.55874115 -0.9921333  -1.50536123 ... -1.66214063  1.32828513
  -2.52719186]
 [-0.32454944 -1.6469605   0.70189133 ... -1.23886969 -0.67793662
  -2.52719186]]
```

```python
df.shape
```

```
(8949, 17)
```

```python
#Building KPIs to understand customer profiles
#1. Monthly Average Purchase

print('The average monthly purchase for the customers are as
follows:')

Monthly_Avg_Purchase = df['PURCHASES']/df['TENURE']

print(Monthly_Avg_Purchase)
```

```
The average monthly purchase for the customers are as follows:
0         7.950000
1         0.000000
2        64.430833
3       124.916667
4         1.333333
           ...
8944     48.520000
8945     50.000000
8946     24.066667
8947      0.000000
8948    182.208333
Length: 8949, dtype: float64
```

```python
# adding Monthly Average Purchase to the df
df['Monthly_Avg_Purchase'] = df['PURCHASES']/df['TENURE']
```

```python
#2. Monthly Average Cash Advance Amount

print('The average monthly cash advance for the customers are as
follows:')

Monthly_Avg_Cash = df['CASH_ADVANCE']/df['TENURE']

print(Monthly_Avg_Cash)
```

```
The average monthly cash advance for the customers are as follows:
0         0.000000
1       536.912124
2         0.000000
```

```
3        17.149001
4         0.000000
            ...
8944      0.000000
8945      0.000000
8946      0.000000
8947      6.093130
8948     21.173335
Length: 8949, dtype: float64
```

```
# adding Monthly Average Cash Advance Amount to the df
df['Monthly_Avg_Cash'] = df['CASH_ADVANCE']/df['TENURE']

#3. Division of Customers based on the type of Purchases (One-Off,
Installments)
#how the customers spend on the basis of the type of purchases: One-
Off purchase, do they make purchases on installments. They are spender
of both the categories or none.

# Step 1: Seperating the Type of Purchases data in another datframe:
df_purchases = df[['ONEOFF_PURCHASES','INSTALLMENTS_PURCHASES']]
df_purchases
```

```
      ONEOFF_PURCHASES  INSTALLMENTS_PURCHASES
0                 0.00                   95.40
1                 0.00                    0.00
2               773.17                    0.00
3              1499.00                    0.00
4                16.00                    0.00
...                ...                     ...
8944              0.00                  291.12
8945              0.00                  300.00
8946              0.00                  144.40
8947              0.00                    0.00
8948           1093.25                    0.00

[8949 rows x 2 columns]
```

```
df_purchases.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8949 entries, 0 to 8948
Data columns (total 2 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   ONEOFF_PURCHASES        8949 non-null   float64
 1   INSTALLMENTS_PURCHASES  8949 non-null   float64
dtypes: float64(2)
memory usage: 140.0 KB
```

```
# Step 2: Filtering on the categories and taking the count of those
categories:

# 1:
df_purchases[(df_purchases['ONEOFF_PURCHASES'] == 0) &
(df_purchases['INSTALLMENTS_PURCHASES'] == 0)].shape
```

```
(2041, 2)
```

```
# 2:
df_purchases[(df_purchases['ONEOFF_PURCHASES'] > 0) &
(df_purchases['INSTALLMENTS_PURCHASES'] == 0)].shape
```

```
(1874, 2)
```

```
# 3.
df_purchases[(df_purchases['ONEOFF_PURCHASES'] == 0) &
(df_purchases['INSTALLMENTS_PURCHASES'] > 0)].shape
```

```
(2260, 2)
```

```
# 4.
df_purchases[(df_purchases['ONEOFF_PURCHASES'] > 0) &
(df_purchases['INSTALLMENTS_PURCHASES'] > 0)].shape
```

```
(2774, 2)
```

8949 customers have credit card are divided into 4 parts.

The 4 categories based on purchase type are:

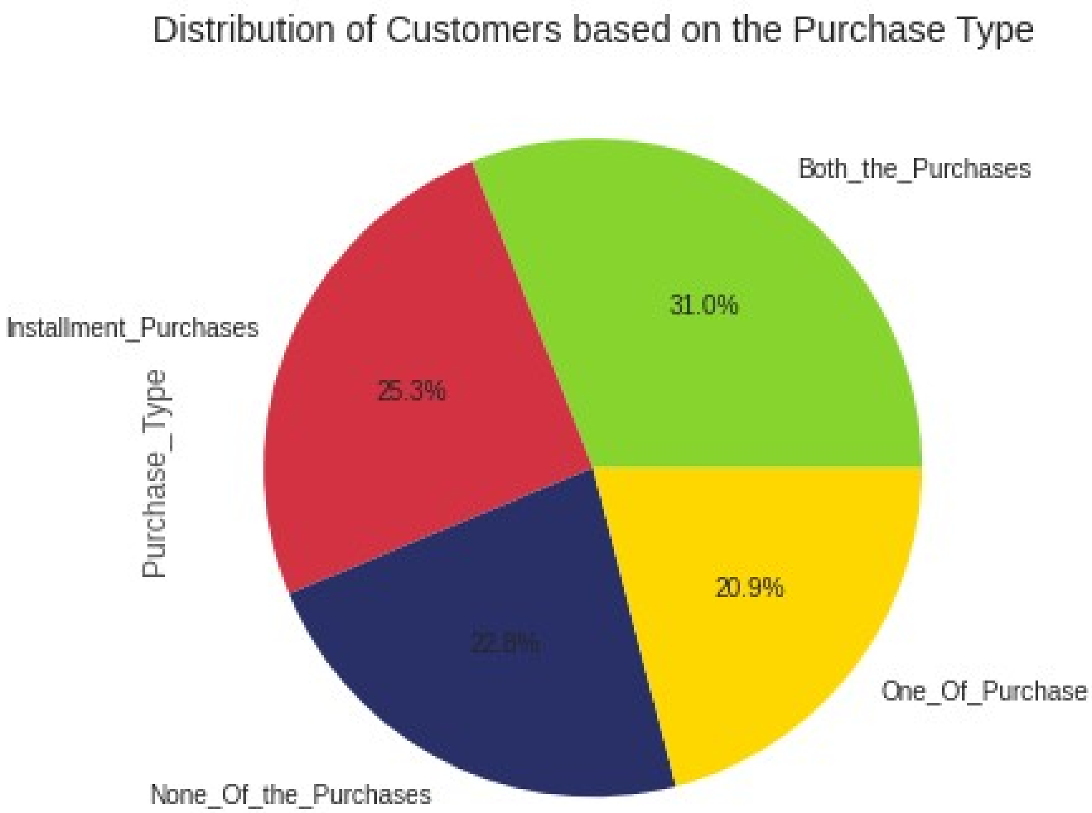
1) Both_the_Purchases 2) Installment_Purchases 3) None_Of_the_Purchases 4) One_Of_Purchase

```
df['Purchase_Type'] = np.where((df['ONEOFF_PURCHASES'] == 0) &
(df['INSTALLMENTS_PURCHASES'] == 0),'None_Of_the_Purchases',
                    np.where((df['ONEOFF_PURCHASES'] > 0) &
(df['INSTALLMENTS_PURCHASES'] == 0), 'One_Of_Purchase',
np.where((df_purchases['ONEOFF_PURCHASES'] == 0) &
(df_purchases['INSTALLMENTS_PURCHASES'] >
0),'Installment_Purchases','Both_the_Purchases')))

# Purchase Type Categories are as follows:
df['Purchase_Type'] .value_counts()
```

```
Both_the_Purchases       2774
Installment_Purchases    2260
None_Of_the_Purchases    2041
One_Of_Purchase          1874
Name: Purchase_Type, dtype: int64
```

```
# Plotting the distribution of customer on basis of Purhcase Type
df['Purchase_Type'].value_counts().sort_index().plot(kind='pie',autopc
```

```
t='%1.01f%%',
                              colors
=['#87d42f','#d33243','#292f67','#FFD700'],fontsize=10,textprops =
{'fontsize': 18})
plt.title('Distribution of Customers based on the Purchase Type')
plt.show()
```



31% of the customers make purchases for both the types: One Off and Installment Purchases followed by 25.3% customer who make only installment purchases.

Average Amount per purchase transaction and average amount per cash-advance transaction is provided to us in the data in attributes as PURCHASES_TRX and CASH_ADVANCE_TRX.

1. Estimating the Limit Usage of customers Computing the ratio of balance to credit limit to estimate the balance-to-limit ratio for each customer. Balance-to-Limit-ratio is also known as the utilization rate. A higher utilization rate indicates presense of credit risk. Hence, a lower utilization rate (balance-to-limit ratio) is desirable.

```
df['Limit_Usage'] = df['BALANCE']/df['CREDIT_LIMIT']
df['Limit_Usage']
```

```
0       0.040901
1       0.457495
```

```
2       0.332687
3       0.222223
4       0.681429
          ...
8944    0.028494
8945    0.019183
8946    0.023399
8947    0.026915
8948    0.310590
Name: Limit_Usage, Length: 8949, dtype: float64
```

```
#6. Payments to Minimum_Payments Ratio
df['Pay_to_MinimumPay'] = df['PAYMENTS']/df['MINIMUM_PAYMENTS']
df['Pay_to_MinimumPay']
```

```
0       1.446508
1       3.826241
2       0.991682
3       0.000000
4       2.771075
          ...
8944    6.660231
8945    0.882891
8946    0.986076
8947    0.942505
8948    0.715439
Name: Pay_to_MinimumPay, Length: 8949, dtype: float64
```

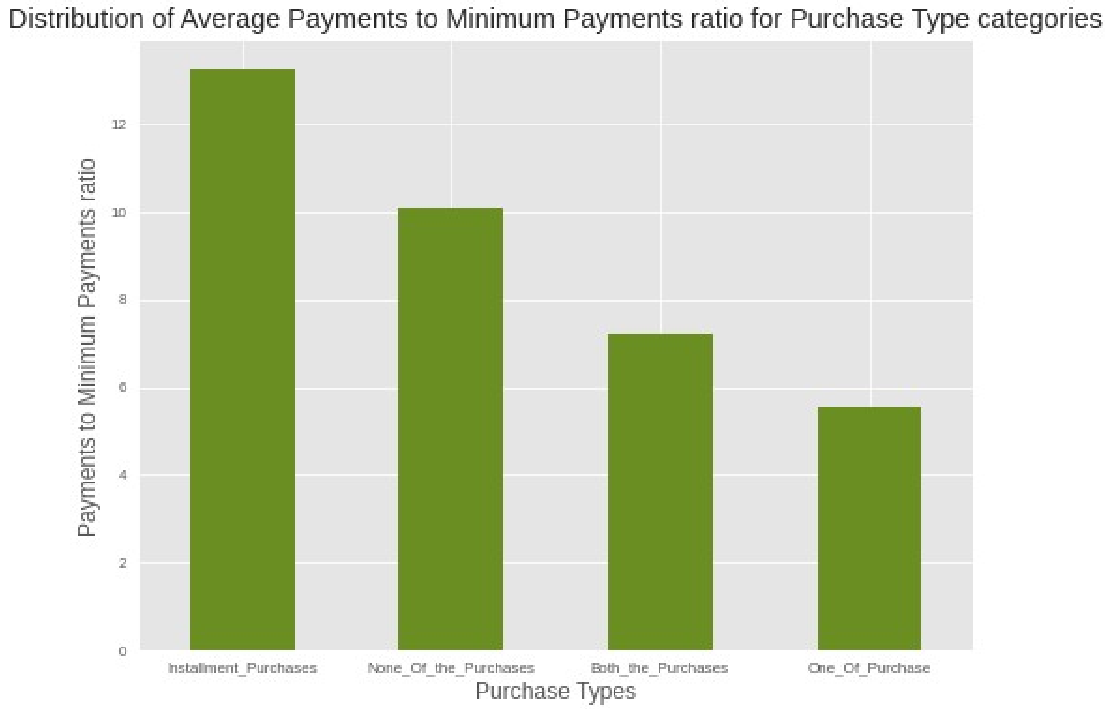
Insights using KPIs To gain insights on the customer profiles, explore the data using the Purchase_Type feature over other attributes to understand how the customers behave.

```
# average of Pay_to_MinimumPay for each of the Purchase Type

t1 = df.groupby(by=['Purchase_Type'])
['Pay_to_MinimumPay'].mean().sort_values(ascending=False)
t1
```

```
Purchase_Type
Installment_Purchases    13.258996
None_Of_the_Purchases    10.092080
Both_the_Purchases        7.236979
One_Of_Purchase           5.571042
Name: Pay_to_MinimumPay, dtype: float64
```

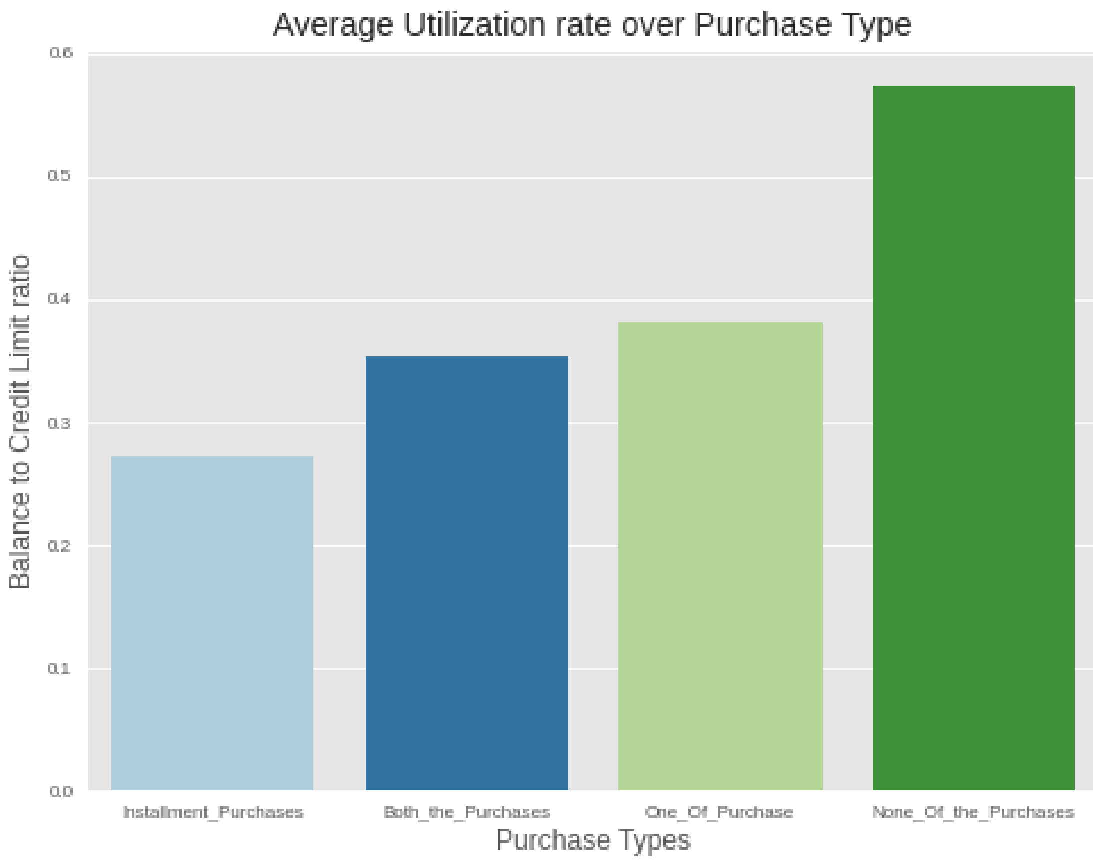
```
# Plot the graph

t1.plot(kind='bar',color='olivedrab')
plt.title('Distribution of Average Payments to Minimum Payments ratio
for Purchase Type categories')
plt.xlabel('Purchase Types')
plt.ylabel('Payments to Minimum Payments ratio')
```

```
plt.xticks(rotation=0)
plt.show()
```



Inference: Customers who made the installment purchases paid the highest average minimum payment dues.

```
# Balance to Credit Limit ratio (or Utilization rate) over Purchase
Type
#Find the average of Limit Usage i.e of the credit card score for each
of the Purchase Type:

t2 = df.groupby(['Purchase_Type'])
['Limit_Usage'].mean().sort_values(ascending = True).reset_index()
t2

            Purchase_Type  Limit_Usage
0  Installment_Purchases     0.271678
1     Both_the_Purchases     0.353548
2        One_Of_Purchase     0.381074
3  None_Of_the_Purchases     0.574049

# Plot the graph of Average Utilization rate over Purchase type

sns.barplot(t2['Purchase_Type'], t2['Limit_Usage'], palette='Paired')
plt.title('Average Utilization rate over Purchase Type')
plt.xlabel('Purchase Types')
plt.ylabel('Balance to Credit Limit ratio')
plt.show()
```

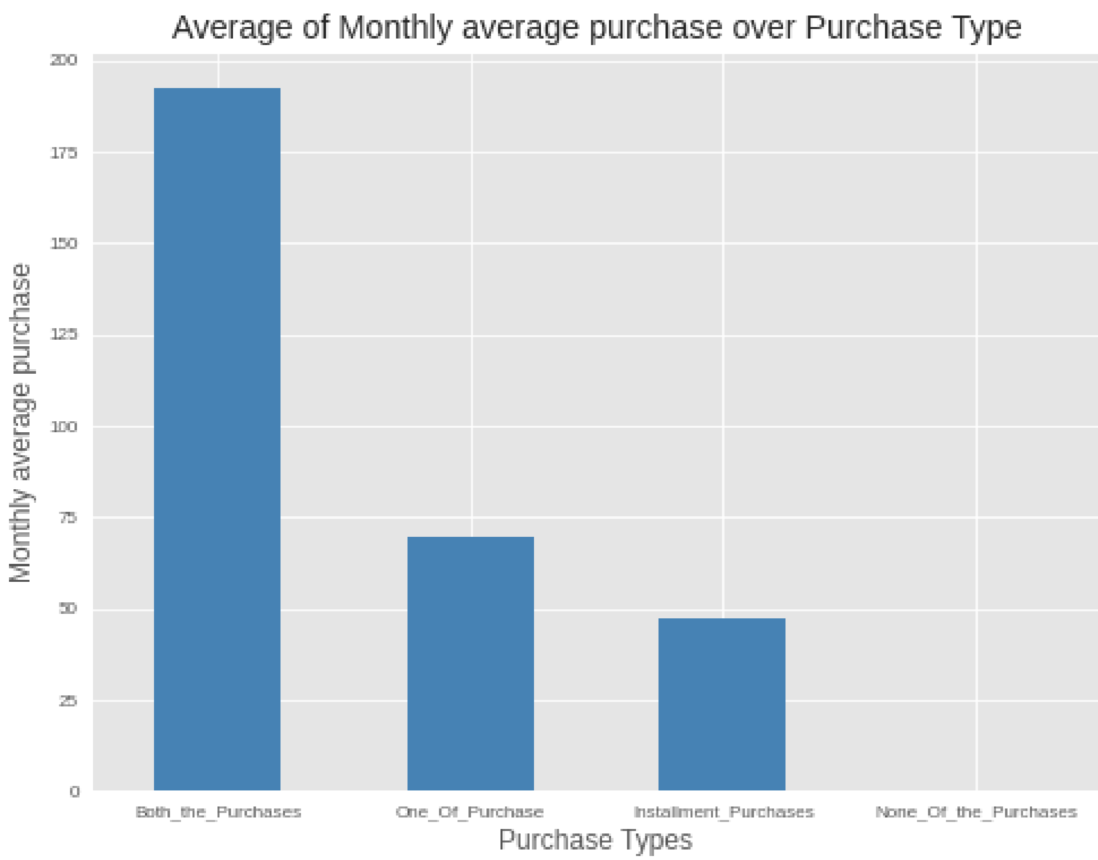


A lower balance-to-limit ratio is desirable which indicates there is low credit risk. The customers who make installment purchases have the lowest utilization rate.

```
# Monthly_Avg_Purchase over Purchase Type
df1 = df.copy()
df1.head()

       BALANCE  BALANCE_FREQUENCY  ...  Limit_Usage  Pay_to_MinimumPay
0    40.900749           0.818182  ...     0.040901           1.446508
1  3202.467416           0.909091  ...     0.457495           3.826241
2  2495.148862           1.000000  ...     0.332687           0.991682
3  1666.670542           0.636364  ...     0.222223           0.000000
4   817.714335           1.000000  ...     0.681429           2.771075

[5 rows x 22 columns]

df1.shape

(8949, 22)

# Find the average of Monthly Average Purchase for each Purchase Type

t3 = df.groupby(by=['Purchase_Type'])
['Monthly_Avg_Purchase'].mean().sort_values(ascending=False)
t3
```

```
Purchase_Type
Both_the_Purchases        192.685172
One_Of_Purchase            69.688958
Installment_Purchases      46.974347
None_Of_the_Purchases       0.000000
Name: Monthly_Avg_Purchase, dtype: float64
```

```
#  Plot the graph

t3.plot(kind='bar',color='steelblue')
plt.title('Average of Monthly average purchase over Purchase Type')
plt.xlabel('Purchase Types')
plt.ylabel('Monthly average purchase')
plt.xticks(rotation=0)
plt.show()
```



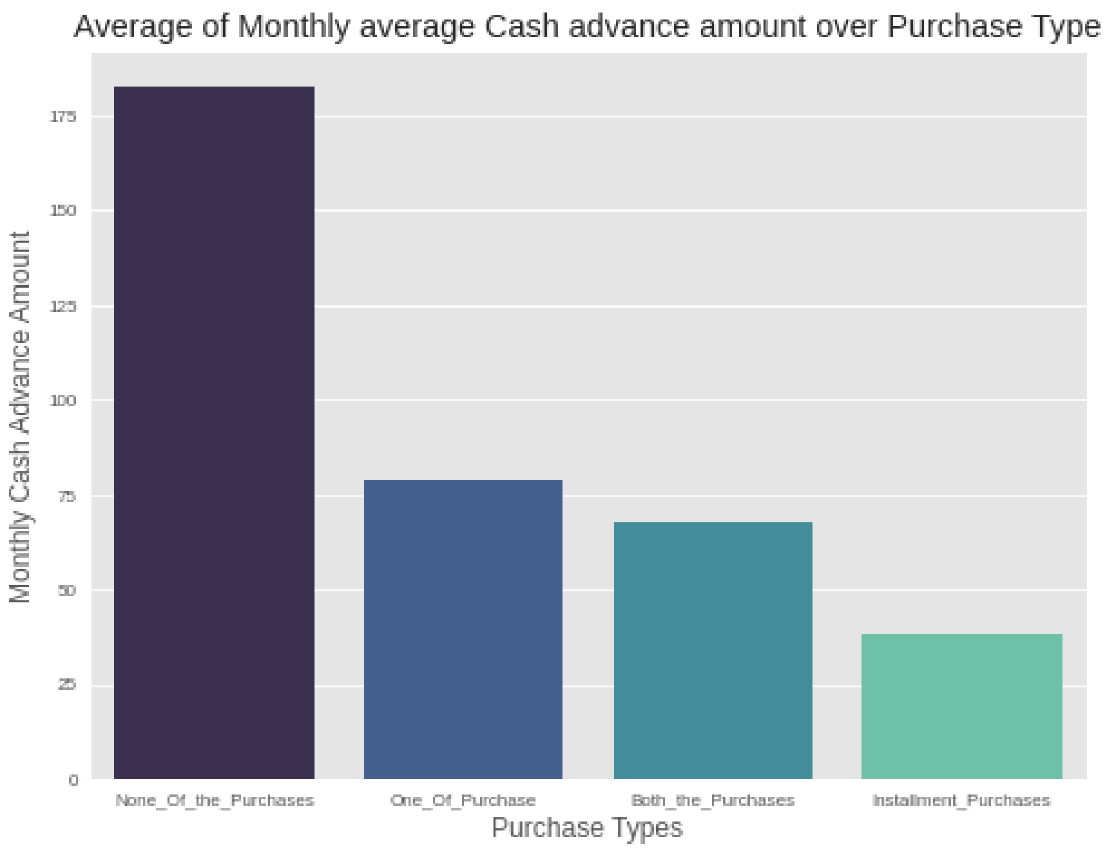
Inference: The customers who made both the one off and installment purchases have made the highhest total average purchase amount over the last 12 months.

```
# Monthly_Cash_Advance over Purchase Type
# Find the average of Monthly Average Cash Advance for each Purchase
Type

t4 = df.groupby(['Purchase_Type'])
```

```
['Monthly_Avg_Cash'].mean().sort_values(ascending=False).reset_index()
t4
```

```
          Purchase_Type  Monthly_Avg_Cash
0  None_Of_the_Purchases        182.932504
1        One_Of_Purchase         78.995966
2      Both_the_Purchases        67.821985
3  Installment_Purchases         38.398206
```

```
#  Plot the graph:

sns.barplot(t4['Purchase_Type'], t4['Monthly_Avg_Cash'],
palette='mako')
plt.title('Average of Monthly average Cash advance amount over
Purchase Type')
plt.xlabel('Purchase Types')
plt.ylabel('Monthly Cash Advance Amount')
plt.show()
```



The customers who made neither the one off purchase nor the installments purchase have made the highest monthly average cash in advance amount.

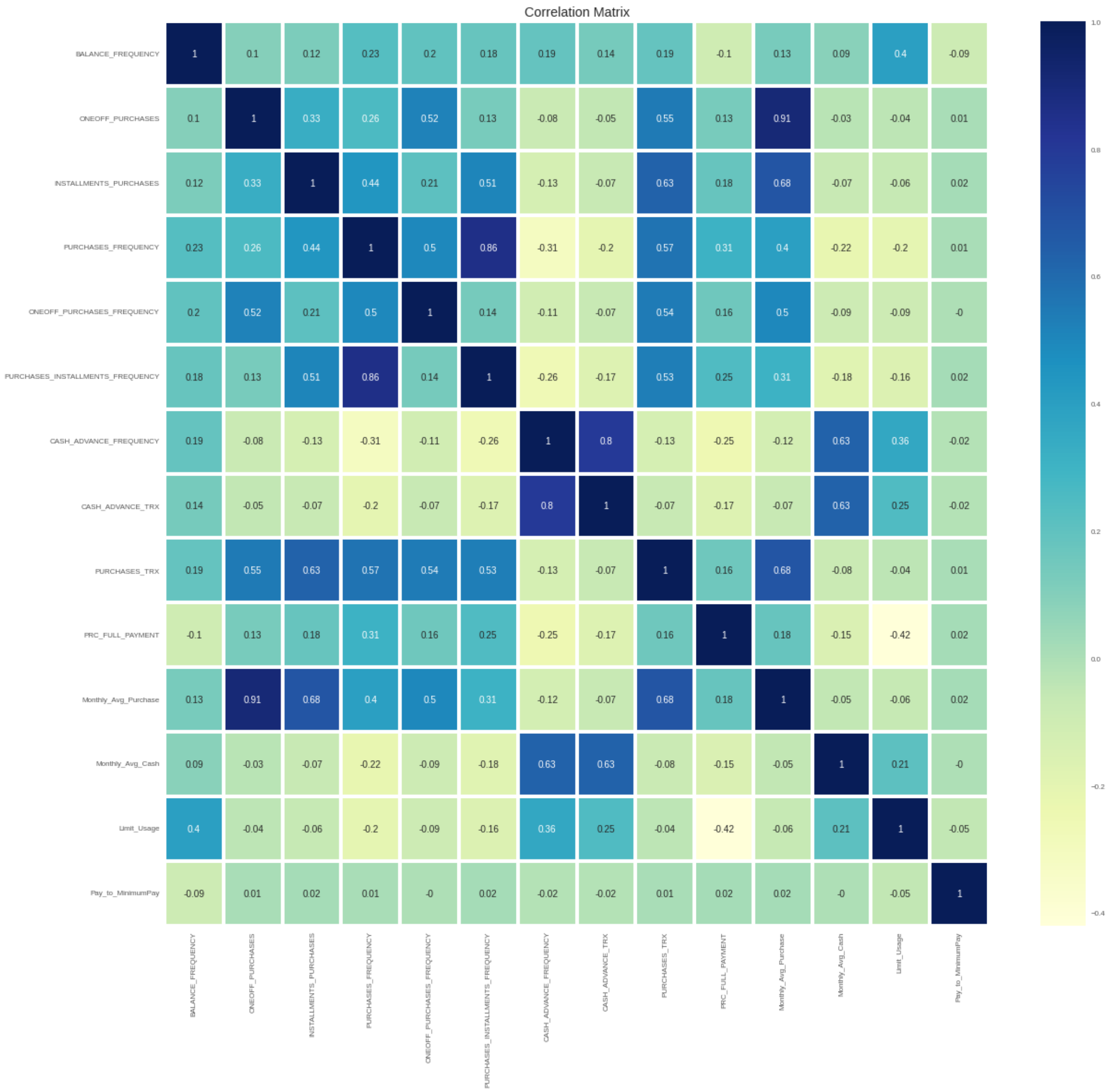
Dropping the original variables 'BALANCE','PURCHASES','PAYMENTS','MINIMUM_PAYMENTS','TENURE','CASH_ADVANCE' which were used to create the new variables. These variables will be correlated with derived variables increasing the redudancy in the data.

```
df.drop(['BALANCE','CREDIT_LIMIT','PURCHASES','PAYMENTS','MINIMUM_PAYMENTS','TENURE','CASH_ADVANCE'], axis=1, inplace=True)
```

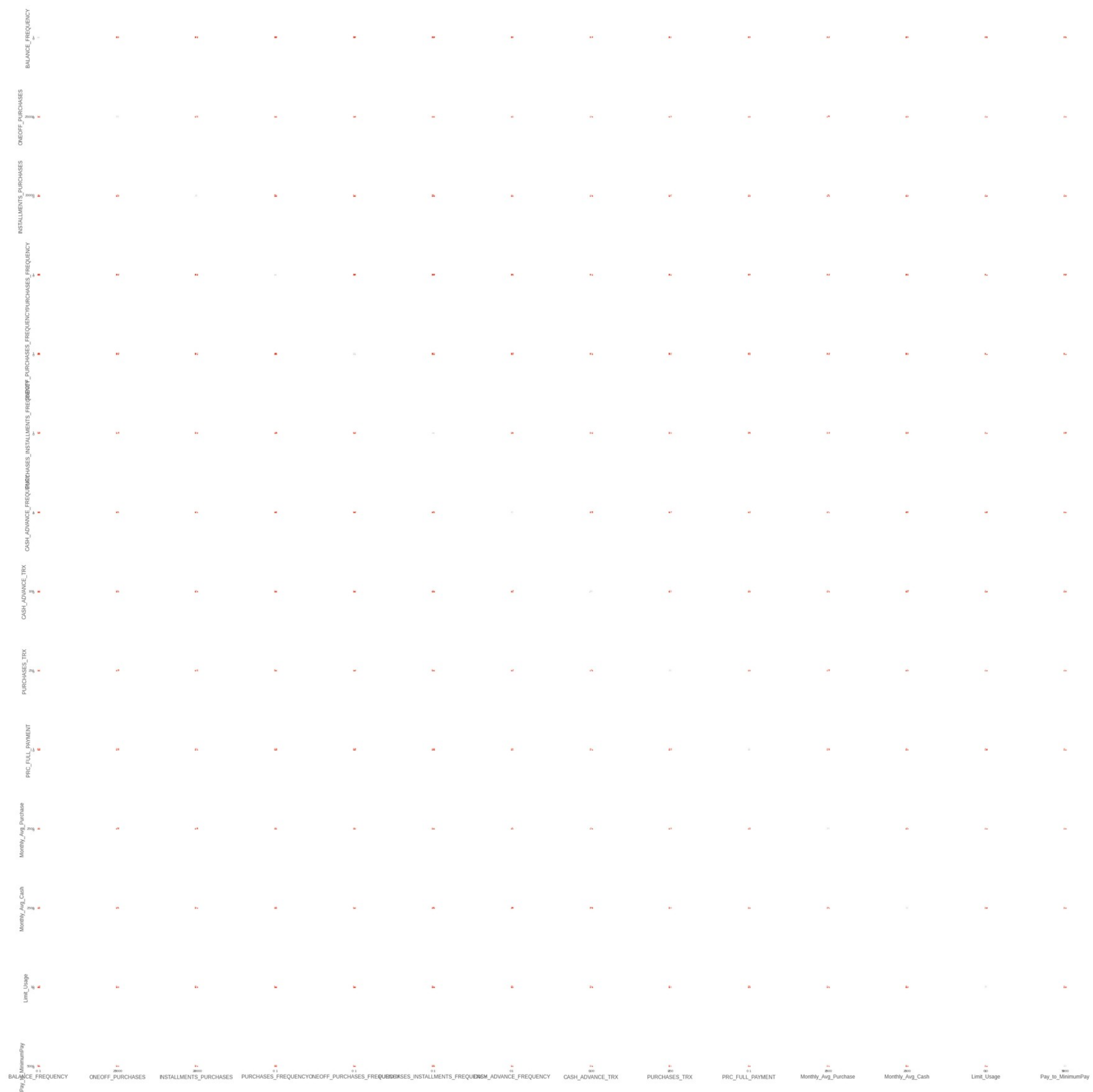
```
corr_df = df.corr()
corr_df
```

```
                                  BALANCE_FREQUENCY  ...
Pay_to_MinimumPay
BALANCE_FREQUENCY                          1.000000  ...          -
0.089340
ONEOFF_PURCHASES                           0.104257  ...
0.010298
INSTALLMENTS_PURCHASES                     0.124204  ...
0.020618
PURCHASES_FREQUENCY                        0.229440  ...
0.011399
ONEOFF_PURCHASES_FREQUENCY                 0.202295  ...          -
0.004556
PURCHASES_INSTALLMENTS_FREQUENCY           0.175869  ...
0.017915
CASH_ADVANCE_FREQUENCY                     0.192022  ...          -
0.021861
CASH_ADVANCE_TRX                           0.141516  ...          -
0.016119
PURCHASES_TRX                              0.189527  ...
0.013472
PRC_FULL_PAYMENT                          -0.095308  ...
0.018459
Monthly_Avg_Purchase                       0.131188  ...
0.016266
Monthly_Avg_Cash                           0.085963  ...          -
0.004345
Limit_Usage                                0.404557  ...          -
0.054659
Pay_to_MinimumPay                         -0.089340  ...
1.000000

[14 rows x 14 columns]
```

```
#finding Correlation among the variables:

plt.figure(figsize=(20,18))
sns.heatmap(round(df.corr(),2),annot=True, cmap='YlGnBu',
linewidths=3, fmt='.2g')
plt.title('Correlation Matrix')
plt.show()
```




```python
# Plotting Pair Plot
sns.pairplot(df)
plt.show()
```



```
#Preparing the data for Modeling
# Creating dummy variables for Purchase_Type

x_cat = pd.get_dummies(df['Purchase_Type'], drop_first=True)
x_cat

# using drop_first = True as will create one dimension less and the
4th category can be computed using the first 3 categories
```

	Installment_Purchases	None_Of_the_Purchases	One_Of_Purchase
0	1	0	0
1	0	1	0
2	0	0	1
3	0	0	1
4	0	0	1
...	...	...	...
8944	1	0	0

```
8945                          1                       0                 0
8946                          1                       0                 0
8947                          0                       1                 0
8948                          0                       0                 1

[8949 rows x 3 columns]
```

```python
#Preparing the data for Modeling
# Creating dummy variables for Purchase_Type

x_cat = pd.get_dummies(df['Purchase_Type'], drop_first=True)
x_cat

# using drop_first = True as will create one dimension less and the
4th category can be computed using the first 3 categories
```

```
      Installment_Purchases  None_Of_the_Purchases  One_Of_Purchase
0                         1                      0                0
1                         0                      1                0
2                         0                      0                1
3                         0                      0                1
4                         0                      0                1
...                     ...                    ...              ...
8944                      1                      0                0
8945                      1                      0                0
8946                      1                      0                0
8947                      0                      1                0
8948                      0                      0                1

[8949 rows x 3 columns]
```

```python
# Filtering out the Numerical variables:

x_num = df.dtypes[df.dtypes != 'object'].index.to_list()
x_num
```

```
['BALANCE_FREQUENCY',
 'ONEOFF_PURCHASES',
 'INSTALLMENTS_PURCHASES',
 'PURCHASES_FREQUENCY',
 'ONEOFF_PURCHASES_FREQUENCY',
 'PURCHASES_INSTALLMENTS_FREQUENCY',
 'CASH_ADVANCE_FREQUENCY',
 'CASH_ADVANCE_TRX',
 'PURCHASES_TRX',
 'PRC_FULL_PAYMENT',
 'Monthly_Avg_Purchase',
 'Monthly_Avg_Cash',
 'Limit_Usage',
 'Pay_to_MinimumPay']
```

```
# Filtering out the Numerical variables in the original df from df1
(Copy ofdf):

x_num_df1 = df1.dtypes[df1.dtypes != 'object'].index.to_list()
x_num_df1

['BALANCE',
 'BALANCE_FREQUENCY',
 'PURCHASES',
 'ONEOFF_PURCHASES',
 'INSTALLMENTS_PURCHASES',
 'CASH_ADVANCE',
 'PURCHASES_FREQUENCY',
 'ONEOFF_PURCHASES_FREQUENCY',
 'PURCHASES_INSTALLMENTS_FREQUENCY',
 'CASH_ADVANCE_FREQUENCY',
 'CASH_ADVANCE_TRX',
 'PURCHASES_TRX',
 'CREDIT_LIMIT',
 'PAYMENTS',
 'MINIMUM_PAYMENTS',
 'PRC_FULL_PAYMENT',
 'TENURE',
 'Monthly_Avg_Purchase',
 'Monthly_Avg_Cash',
 'Limit_Usage',
 'Pay_to_MinimumPay']

# Original Variables Concatenated with dummy variables but without
Scaling the Numerical variables :

original_df = pd.concat([x_cat, df1[x_num_df1]], axis=1)
original_df.head()

   Installment_Purchases  None_Of_the_Purchases  ...  Limit_Usage
Pay_to_MinimumPay
0                      1                      0  ...     0.040901
1.446508
1                      0                      1  ...     0.457495
3.826241
2                      0                      0  ...     0.332687
0.991682
3                      0                      0  ...     0.222223
0.000000
4                      0                      0  ...     0.681429
2.771075

[5 rows x 24 columns]
```

```
# Scaling the numerical variables

from sklearn.preprocessing import StandardScaler

SS = StandardScaler()

x_scaled = pd.DataFrame(SS.fit_transform(df[x_num]), columns=x_num)
x_scaled.head()

   BALANCE_FREQUENCY  ONEOFF_PURCHASES  ...  Limit_Usage
Pay_to_MinimumPay
0          -0.249881         -0.356957  ...    -0.893059          -
0.064423
1           0.134049         -0.356957  ...     0.175953          -
0.044287
2           0.517980          0.108843  ...    -0.144316          -
0.068272
3          -1.017743          0.546123  ...    -0.427774          -
0.076663
4           0.517980         -0.347317  ...     0.750582          -
0.053215

[5 rows x 14 columns]

# Combining the Categorical and Numerical dataset

concat_df = pd.concat([x_cat, x_scaled], axis=1)
concat_df.head()

   Installment_Purchases  None_Of_the_Purchases  ...  Limit_Usage
Pay_to_MinimumPay
0                      1                      0  ...    -0.893059
-0.064423
1                      0                      1  ...     0.175953
-0.044287
2                      0                      0  ...    -0.144316
-0.068272
3                      0                      0  ...    -0.427774
-0.076663
4                      0                      0  ...     0.750582
-0.053215

[5 rows x 17 columns]
```

Applying PCA Will be performing Prinicipal Component Analysis(PCA) to reduce the dimensions.

```
from sklearn.decomposition import PCA

#Steps to perform PCA:
#Scaled the data:
```

```
concat_df.head()
```

```
   Installment_Purchases  None_Of_the_Purchases  ...  Limit_Usage
Pay_to_MinimumPay
0                      1                      0  ...    -0.893059
-0.064423
1                      0                      1  ...     0.175953
-0.044287
2                      0                      0  ...    -0.144316
-0.068272
3                      0                      0  ...    -0.427774
-0.076663
4                      0                      0  ...     0.750582
-0.053215

[5 rows x 17 columns]
```

```
#  Find the covariance Matrix:

cov_matrix = np.cov(concat_df.T)
print(cov_matrix.shape)
print('Covariance Matrix:', cov_matrix)
```

```
(17, 17)
Covariance Matrix: [[ 0.18878572 -0.05760378 -0.05289048 -0.05855883 -
0.09015665  0.03540427
   0.12856082 -0.17142362  0.1868402  -0.1030289  -0.07341032 -
0.02855805
   0.08574557 -0.05486451 -0.06615361 -0.07599021  0.00897372]
 [-0.05760378  0.17607385 -0.04776526  0.00518779 -0.08142023 -
0.10369386
  -0.27862371 -0.15481222 -0.20909041  0.15684113  0.10204653 -
0.13498277
  -0.08526858 -0.10890678  0.11095414  0.10835407  0.0019918 ]
 [-0.05289048 -0.04776526  0.1655753  -0.03628075  0.02451854 -
0.09520935
  -0.08840125  0.08304779 -0.19206814 -0.00992445 -0.00972336 -
0.06404633
  -0.03673949 -0.01913927 -0.01083133 -0.00422013 -0.00618309]
 [-0.05855883  0.00518779 -0.03628075  1.00011176  0.1042684
0.12421758
   0.22946609  0.20231737  0.17588838  0.19204378  0.14153225
0.18954776
  -0.09531866  0.13120311  0.08597276  0.40460218 -0.08935015]
 [-0.09015665 -0.08142023  0.02451854  0.1042684   1.00011176
0.33064653
   0.26494216  0.52493992  0.12771371 -0.0826314  -0.04623107
0.5455752
   0.13275964  0.91316095 -0.0345611  -0.04225896  0.0102995 ]
```

```
 [ 0.03540427 -0.10369386 -0.09520935  0.12421758  0.33064653
1.00011176
   0.44244707  0.21403986  0.51139132 -0.13232689 -0.07402549
0.62816721
   0.18256799  0.67709213 -0.0678062  -0.05832416  0.02062031]
 [ 0.12856082 -0.27862371 -0.08840125  0.22946609  0.26494216
0.44244707
   1.00011176  0.50136122  0.86301739 -0.3085176  -0.20356387
0.56847109
   0.30579499  0.39535408 -0.21587028 -0.20196655  0.01139991]
 [-0.17142362 -0.15481222  0.08304779  0.20231737  0.52493992
0.21403986
   0.50136122  1.00011176  0.14228589 -0.1117194  -0.06912335  0.54491
   0.15751502  0.49974981 -0.08906929 -0.09209901 -0.00455627]
 [ 0.1868402  -0.20909041 -0.19206814  0.17588838  0.12771371
0.51139132
   0.86301739  0.14228589  1.00011176 -0.26298409 -0.16926908
0.53000852
   0.25007715  0.31414127 -0.17939345 -0.16155451  0.01791683]
 [-0.1030289   0.15684113 -0.00992445  0.19204378 -0.0826314  -
0.13232689
  -0.3085176  -0.1117194  -0.26298409  1.00011176  0.79966188 -
0.13117544
  -0.24979609 -0.11611516  0.62839126  0.36020827 -0.0218638 ]
 [-0.07341032  0.10204653 -0.00972336  0.14153225 -0.04623107 -
0.07402549
  -0.20356387 -0.06912335 -0.16926908  0.79966188  1.00011176 -
0.06618758
  -0.16982639 -0.06572332  0.63336182  0.25262383 -0.01612085]
 [-0.02855805 -0.13498277 -0.06404633  0.18954776  0.5455752
0.62816721
   0.56847109  0.54491     0.53000852 -0.13117544 -0.06618758
1.00011176
   0.16205527  0.68264939 -0.08341994 -0.04379938  0.01347378]
 [ 0.08574557 -0.08526858 -0.03673949 -0.09531866  0.13275964
0.18256799
   0.30579499  0.15751502  0.25007715 -0.24979609 -0.16982639
0.16205527
   1.00011176  0.18177538 -0.15140284 -0.41574788  0.01846083]
 [-0.05486451 -0.10890678 -0.01913927  0.13120311  0.91316095
0.67709213
   0.39535408  0.49974981  0.31414127 -0.11611516 -0.06572332
0.68264939
   0.18177538  1.00011176 -0.04577493 -0.05710301  0.01626812]
 [-0.06615361  0.11095414 -0.01083133  0.08597276 -0.0345611  -
0.0678062
  -0.21587028 -0.08906929 -0.17939345  0.62839126  0.63336182 -
0.08341994
  -0.15140284 -0.04577493  1.00011176  0.21118346 -0.00434547]
 [-0.07599021  0.10835407 -0.00422013  0.40460218 -0.04225896 -
```

```
0.05832416
  -0.20196655 -0.09209901 -0.16155451  0.36020827  0.25262383 -
0.04379938
  -0.41574788 -0.05710301  0.21118346  1.00011176 -0.05466551]
 [ 0.00897372  0.0019918  -0.00618309 -0.08935015  0.0102995
0.02062031
   0.01139991 -0.00455627  0.01791683 -0.0218638  -0.01612085
0.01347378
   0.01846083  0.01626812 -0.00434547 -0.05466551  1.00011176]]
```

```python
#  Calculate the eigenvalues and eigenvectors:

eig_val, eig_vec = np.linalg.eig(cov_matrix)
print(len(eig_val))
print(eig_vec.shape)
```

```
17
(17, 17)
```

```python
print('Eigen Vectors:', eig_vec)
print('Eigen Values:', eig_val)
```

```
Eigen Vectors: [[-2.16214195e-02 -9.60677046e-02  1.67381856e-01
8.19928834e-02
  -3.57899991e-02  9.84994683e-02  2.44027882e-02  2.11899671e-01
   5.56474223e-02  3.60667634e-02  4.02520605e-02  1.48532851e-01
   2.33858365e-01  8.34225681e-01  3.69341072e-03  3.46267956e-01
  -2.16081494e-02]
 [ 1.11657548e-01  4.90282698e-02 -5.65895947e-02 -1.17156465e-02
  -3.87011677e-02  7.96232725e-02  1.32863134e-01 -6.48391206e-02
  -4.72425183e-02 -1.49445350e-04 -2.50198107e-01 -2.89863493e-01
  -4.53616465e-01  1.23152526e-02  5.81280017e-03  5.27284617e-01
   5.65031484e-01]
 [ 3.02717103e-02 -8.40584979e-03 -1.46726683e-01 -5.29253320e-02
   5.12449464e-02 -1.02322672e-01 -5.16997856e-02 -8.43400519e-02
  -4.57082809e-03 -3.69491736e-02  2.27176821e-01  3.25503001e-01
   5.61519210e-01 -3.70742145e-01 -4.02097658e-03  5.39101108e-01
   2.22381854e-01]
 [-7.17343323e-02  2.91641417e-01  3.20955201e-01 -4.22057416e-01
   9.44842223e-02 -2.96062607e-01  4.60078481e-01  4.64286783e-02
  -5.46034733e-01 -1.33766372e-02 -7.67929086e-02  9.81443201e-02
   4.41503243e-02  3.25010755e-02  4.05332971e-03 -1.19073730e-04
  -3.69088395e-02]
 [-3.02709015e-01  2.26553855e-01 -4.62987554e-01 -5.62581847e-02
   2.32435869e-03  4.89496653e-02  8.22062237e-02  5.23430585e-01
   1.52681849e-02  1.18991424e-01 -4.32751214e-02 -7.03756623e-03
  -9.13643020e-03 -2.52407867e-02 -5.82478187e-01  7.51452763e-03
  -2.64835097e-03]
 [-3.27399292e-01  1.39030379e-01  1.10014048e-01  9.78938040e-02
  -2.09276305e-01  4.73947468e-01  1.31399215e-01 -5.03896443e-01
  -1.84255741e-01 -8.21955615e-02  4.04715947e-01 -4.22464139e-02
```

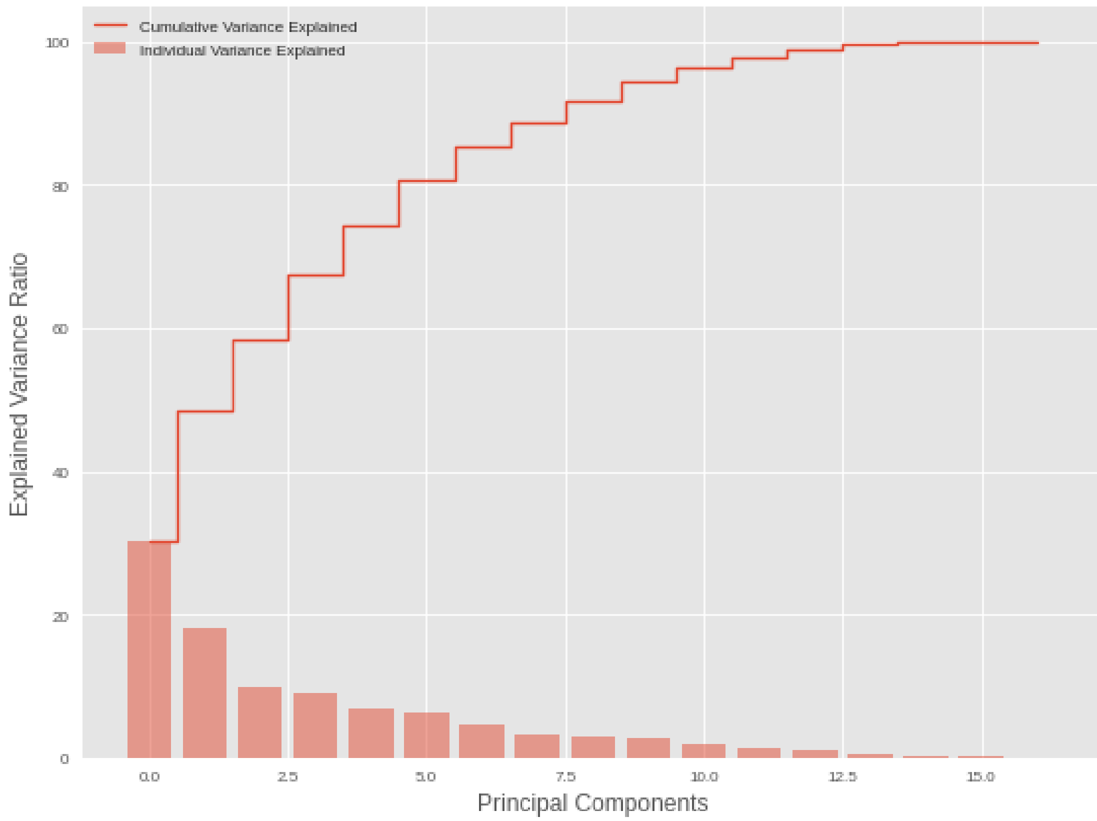
```
 -2.94260077e-02  4.58311043e-02 -3.20610796e-01  7.16524181e-03
  9.93818414e-03]
[-3.81713386e-01 -1.17134993e-02  3.71278149e-01  5.13803819e-02
  8.15381081e-02 -2.46191533e-01 -2.27259634e-01  1.81420284e-01
  1.13808626e-01 -1.62603313e-03  2.35407535e-01 -5.31235389e-02
  5.42879453e-02  1.60666419e-02 -1.80783606e-02 -2.93633036e-01
  6.33170523e-01]
[-2.88618327e-01  1.56733795e-01 -2.48066055e-01 -1.16115150e-01
  2.27363312e-01 -5.38702476e-01 -2.69548478e-01 -3.35047328e-01
  9.86327937e-02 -1.05456354e-01  1.96714383e-01 -1.64382028e-01
 -2.16785131e-01  2.17513862e-01  2.23894287e-03  2.23554598e-01
 -2.43823177e-01]
[-3.31828597e-01 -3.09342808e-02  5.25153914e-01  1.19608960e-01
 -6.31925453e-02  6.45350039e-02 -1.45780515e-01  2.74993009e-01
  7.92956934e-02  6.59572459e-02 -7.46784837e-02 -1.36617222e-01
 -1.29462303e-01 -3.30921434e-01  9.43261191e-03  4.12748423e-01
 -4.02752668e-01]
[ 2.18531315e-01  4.60877619e-01  7.19479233e-02  2.18215866e-01
  4.58069956e-02 -4.38323083e-02 -8.00097578e-03 -7.18149513e-02
  3.27412814e-02  3.66462939e-01 -1.41046149e-02 -5.98252940e-01
  4.27809320e-01  2.42352990e-02 -5.63112375e-03 -1.69029257e-02
 -2.00519841e-02]
[ 1.73516316e-01  4.51291532e-01  9.24044469e-02  3.36164027e-01
  8.07475796e-02 -5.58540886e-02 -8.05935405e-02 -3.63288401e-02
  3.26132436e-02  4.16856926e-01  1.06905637e-01  5.65694305e-01
 -3.46749058e-01 -7.53073366e-03  8.13011892e-03  1.36785417e-02
  2.02792466e-02]
[-3.81520071e-01  1.96365009e-01  3.29271253e-03 -4.89703746e-03
 -6.27196960e-02  6.84310714e-02 -1.53351805e-01 -3.22912289e-01
  1.16384414e-01 -1.43557517e-02 -7.47610122e-01  2.16010915e-01
  2.21071303e-01  4.53110368e-02  1.26555678e-02 -6.45224982e-02
  6.97190396e-02]
[-1.89752176e-01 -1.73649657e-01 -9.24793371e-03  4.32288808e-01
  3.00074245e-01 -1.75220527e-01  7.05613415e-01 -9.97443130e-02
  3.33700776e-01 -3.67286478e-02 -3.57050966e-02  7.98199363e-03
  2.88987556e-02 -5.29482762e-02 -1.26416522e-03 -8.21765389e-03
 -2.33069247e-02]
[-3.72565181e-01  2.36424079e-01 -3.13257636e-01  2.18007874e-03
 -8.39569013e-02  2.36033200e-01  1.17777494e-01  2.00471079e-01
 -6.31096994e-02  4.31797052e-02  1.60751071e-01 -3.39900321e-02
 -7.54740572e-03 -3.11432596e-03  7.46354444e-01 -6.21671571e-03
  1.86896076e-02]
[ 1.64987875e-01  4.02636504e-01  4.43023659e-02  3.55638903e-01
  5.38400601e-02  5.92174919e-03 -7.99187317e-02  1.79041931e-01
 -7.29696704e-02 -7.96024033e-01 -3.36687459e-02  1.17752320e-02
  1.85693802e-02 -2.25440954e-04 -1.08114169e-02 -2.74641059e-03
 -1.11312797e-02]
[ 1.32105947e-01  3.20832874e-01  1.55438619e-01 -4.97828414e-01
 -1.90835848e-01  7.93337854e-02  1.88651709e-01 -2.75069044e-05
  7.04477555e-01 -1.24976168e-01  1.18780084e-01  4.38488786e-02
```

```
  -3.48893100e-02 -9.05790431e-03 -2.31891417e-03 -3.46336902e-03
  -7.91018446e-03]
 [-1.26059885e-02 -3.45580289e-02 -6.74008776e-02  2.17233240e-01
  -8.54732457e-01 -4.48743514e-01  1.18378653e-01 -2.20809998e-04
  -2.66751806e-02  2.86735252e-03  3.10655632e-03  1.15224078e-02
   7.35093722e-03 -5.67744189e-04  4.97603183e-04 -3.41297177e-03
  -5.77465275e-03]]
Eigen Values: [4.39676109 2.63000669 1.46128961 1.30796423 1.00416405
0.91491162
 0.69314392 0.47418763 0.43347583 0.41105916 0.28858693 0.19672233
 0.15896349 0.08921346 0.00446266 0.02813756 0.0389492 ]
```

```
# Making the Eigen Pairs:

eigen_pairs = [(eig_val[i], eig_vec[:,i]) for i in
range(len(eig_val))]
eigen_pairs_sorted = sorted(eigen_pairs, reverse = True)

#  Sort the Eigen Vectors and Eigen Values

eig_val_sorted = [eigen_pairs_sorted[i][0] for i in
range(len(eig_val))]
eig_vec_sorted = [eigen_pairs_sorted[i][1] for i in
range(len(eig_val))]

#  Calculating Cumulative Variance Explained:

tot = np.sum(eig_val)
exp_var = [(i/tot)*100 for i in sorted(eig_val, reverse = True)]   #
explained variance
tot_var = np.cumsum(exp_var)                                       #
total variance explained
print('Cumulative Variance explained', tot_var)
```

```
Cumulative Variance explained [ 30.25572014  48.35375746  58.40942539
67.41000536  74.32002523
  80.61586649  85.38564319  88.64870161  91.63160722  94.46025558
  96.44612772  97.799846    98.89373187  99.5076423   99.77566598
  99.96929084 100.        ]
```

```
# Plotting the Summary Plot of the Cumulative Variance Explained:

plt.bar(range(17), exp_var, alpha=0.50, align = 'center',
label='Individual Variance Explained')
plt.step(range(17), tot_var, where ='mid', label='Cumulative Variance
Explained')
plt.ylabel('Explained Variance Ratio')
plt.xlabel('Principal Components')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```




```
pca_model = PCA(n_components = 17)
X_PCA = pca_model.fit_transform(concat_df)

X_PCA.shape
```

```
(8949, 17)
```

```
# Cumulative Variance explained:
pca_var = pca_model.explained_variance_ratio_
np.cumsum(pca_var)
```

```
array([0.3025572 , 0.48353757, 0.58409425, 0.67410005, 0.74320025,
       0.80615866, 0.85385643, 0.88648702, 0.91631607, 0.94460256,
       0.96446128, 0.97799846, 0.98893732, 0.99507642, 0.99775666,
       0.99969291, 1.        ])
```

```
# Cumulative Variance explains
var1 = np.cumsum(np.round(pca_model.explained_variance_ratio_,
decimals=6)*100)
var1
```

```
array([30.2557, 48.3537, 58.4094, 67.41  , 74.32  , 80.6158, 85.3856,
       88.6487, 91.6316, 94.4602, 96.4461, 97.7998, 98.8937, 99.5076,
       99.7756, 99.9692, 99.9999])
```

```
# Summary table showing the Eigen Vectors, Eigen Values and the
variance explained by each of the component(eigenvector)

vec_val = pd.DataFrame({'Eigen_Values':pca_model.explained_variance_,
'Cumulative_Variance':var1},
                       index=range(1,18)).round(4)
vec_val
```

```
    Eigen_Values  Cumulative_Variance
1         4.3968              30.2557
2         2.6300              48.3537
3         1.4613              58.4094
4         1.3080              67.4100
5         1.0042              74.3200
6         0.9149              80.6158
7         0.6931              85.3856
8         0.4742              88.6487
9         0.4335              91.6316
10        0.4111              94.4602
11        0.2886              96.4461
12        0.1967              97.7998
13        0.1590              98.8937
14        0.0892              99.5076
15        0.0389              99.7756
16        0.0281              99.9692
17        0.0045              99.9999
```

```
# PCA with 8 components:

PCA_7 = PCA(n_components=7)
X_PCA_7 = PCA_7.fit_transform(concat_df)

PC = pd.DataFrame(X_PCA_7, columns=['PC1 PC2 PC3 PC4 PC5 PC6
PC7'.split()])
PC
```

```
           PC1       PC2       PC3       PC4       PC5       PC6
PC7
0    -0.963797 -1.466063  0.395737  0.147147  0.113362 -0.452822 -
0.161024
1    -2.209442  0.931402  0.333874 -0.799582 -0.191202 -0.214703
0.703046
2     1.007415 -0.162731  0.841429  1.325162 -0.822963  2.170478 -
1.088306
3    -0.866781 -0.770916  1.725054  0.148277  0.109396 -0.527342 -
0.380985
4    -1.306443 -0.603864  0.469885  1.500899  0.189492  0.023789
0.418797
...        ...       ...       ...       ...       ...       ...
...
8944  1.055194 -1.518576 -1.542865 -0.646465 -0.416310  0.319131
```

```
0.680914
8945  0.739634 -1.224406 -1.556767  0.090250  0.053046 -0.007028 -
0.533323
8946  0.427028 -1.622618 -0.976104 -0.482658 -0.177350 -0.023084 -
0.129768
8947 -1.473409 -0.962832  0.786342 -0.353589 -0.230467 -0.234425
0.658129
8948  0.311737  0.472111  1.473544  0.126844 -0.404786  0.845758 -
1.277894

[8949 rows x 7 columns]
```

```
# Taking out the list of columns:

list_cols = concat_df.columns

PC_with_all_variables = pd.DataFrame(PCA_7.components_.T, columns =
['PC_'+str(i) for i in range(1,8)], index = list_cols)
PC_with_all_variables
```

```
                                       PC_1      PC_2  ...      PC_6
PC_7
Installment_Purchases              0.021621 -0.096068  ... -0.098499
0.024403
None_Of_the_Purchases             -0.111658  0.049028  ... -0.079623
0.132863
One_Of_Purchase                   -0.030272 -0.008406  ...  0.102323 -
0.051700
BALANCE_FREQUENCY                  0.071734  0.291641  ...  0.296063
0.460078
ONEOFF_PURCHASES                   0.302709  0.226554  ... -0.048950
0.082206
INSTALLMENTS_PURCHASES             0.327399  0.139030  ... -0.473947
0.131399
PURCHASES_FREQUENCY                0.381713 -0.011713  ...  0.246192 -
0.227260
ONEOFF_PURCHASES_FREQUENCY         0.288618  0.156734  ...  0.538702 -
0.269548
PURCHASES_INSTALLMENTS_FREQUENCY  0.331829 -0.030934  ... -0.064535 -
0.145781
CASH_ADVANCE_FREQUENCY            -0.218531  0.460878  ...  0.043832 -
0.008001
CASH_ADVANCE_TRX                  -0.173516  0.451292  ...  0.055854 -
0.080594
PURCHASES_TRX                      0.381520  0.196365  ... -0.068431 -
0.153352
PRC_FULL_PAYMENT                   0.189752 -0.173650  ...  0.175221
0.705613
Monthly_Avg_Purchase               0.372565  0.236424  ... -0.236033
0.117777
Monthly_Avg_Cash                  -0.164988  0.402637  ... -0.005922 -
```

```
0.079919
Limit_Usage                         -0.132106  0.320833  ... -0.079334
0.188652
Pay_to_MinimumPay                    0.012606 -0.034558  ...  0.448744
0.118379

[17 rows x 7 columns]
```

```
# Exporting the output:
PC_with_all_variables.to_csv('PC_with_all_variables.csv')

# Variance explained by each of the Component:

pd.Series(PCA_7.explained_variance_ratio_*100, index = ['PC_' + str(i)
for i in range(1,8)])
```

```
PC_1    30.255720
PC_2    18.098037
PC_3    10.055668
PC_4     9.000580
PC_5     6.910020
PC_6     6.295841
PC_7     4.769777
dtype: float64
```

```
Loadings = pd.DataFrame((pca_model.components_.T *
np.sqrt(pca_model.explained_variance_)).T, index=
list_cols,columns=['PC1 PC2 PC3 PC4 PC5 PC6 PC7 PC8 PC9 PC10 PC11 PC12
PC13 PC14 PC15 PC16 PC17'.split()])

Loadings
```

```
                                       PC1       PC2  ...      PC16
PC17
Installment_Purchases             0.045337 -0.234129  ... -0.277006
0.026433
None_Of_the_Purchases            -0.155796  0.079511  ...  0.520304 -
0.056044
One_Of_Purchase                  -0.202338  0.068408  ... -0.187900
0.081477
BALANCE_FREQUENCY                -0.093772  0.013399  ...  0.569348 -
0.248442
ONEOFF_PURCHASES                  0.035864  0.038782  ...  0.191233
0.856510
INSTALLMENTS_PURCHASES           -0.094216 -0.076160  ... -0.075884
0.429228
PURCHASES_FREQUENCY               0.020317  0.110616  ...  0.157062
0.098556
ONEOFF_PURCHASES_FREQUENCY        0.145917 -0.044649  ... -0.000019 -
0.000152
PURCHASES_INSTALLMENTS_FREQUENCY  0.036638 -0.031104  ...  0.463820 -
```

```
0.017563
CASH_ADVANCE_FREQUENCY           -0.023124  0.000096  ...  0.080127 -
0.001838
CASH_ADVANCE_TRX                  0.021624 -0.134407  ...  0.063809
0.001669
PURCHASES_TRX                     0.065879 -0.128564  ...  0.019448
0.005111
PRC_FULL_PAYMENT                 -0.093240  0.180858  ...  0.013910 -
0.002931
Monthly_Avg_Purchase              0.249172  0.003678  ... -0.002705 -
0.000170
Monthly_Avg_Cash                 -0.004264  0.111512  ... -0.001561 -
0.001140
Limit_Usage                      -0.058084 -0.088448  ...  0.000581
0.000573
Pay_to_MinimumPay                 0.000247  0.000388  ... -0.000155
0.000033

[17 rows x 17 columns]
```

```
# Exporting the output:
Loadings.to_csv('Loadings1.csv')
```

```
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.metrics import silhouette_score
from scipy.cluster.hierarchy import linkage, cophenet, dendrogram
from scipy.spatial.distance import pdist
```

```
# Step 1: Finding the Cophenetic Distance Correlation Coefficient for
different Linkages

for i  in ['single', 'complete', 'average']:
    print('Linkage is', i)
    for j in ['euclidean', 'cityblock', 'cosine']:
            Z= linkage(X_PCA_7, method = i , metric = j)
            c , coph_dist = cophenet(Z, pdist(X_PCA_7))
            print('Cophenetic Distance Correlation Coefficient for',
j, 'distance is\t:', c)
    print()

Z= linkage(X_PCA_7, 'ward')
c , coph_dist = cophenet(Z, pdist(X_PCA_7))
print('Cophenetic Distance Correlation Coefficient for ward linkage is
\t:', c)
```

```
Linkage is single
Cophenetic Distance Correlation Coefficient for euclidean distance is
      : 0.7630936998688107
Cophenetic Distance Correlation Coefficient for cityblock distance is
      : 0.7516099250964815
Cophenetic Distance Correlation Coefficient for cosine distance is
```

```
      : 0.0332067352490848

Linkage is complete
Cophenetic Distance Correlation Coefficient for euclidean distance is
      : 0.803293695300964
Cophenetic Distance Correlation Coefficient for cityblock distance is
      : 0.804782193826038
Cophenetic Distance Correlation Coefficient for cosine distance is
      : 0.26890794541141255

Linkage is average
Cophenetic Distance Correlation Coefficient for euclidean distance is
      : 0.8598689038991882
Cophenetic Distance Correlation Coefficient for cityblock distance is
      : 0.869806244869372
Cophenetic Distance Correlation Coefficient for cosine distance is
      : 0.3398016312224923

Cophenetic Distance Correlation Coefficient for ward linkage is  :
0.3619897408064154
```

```python
# Step 2: Finding the Optimal clusters using KMeans, Silhouette
Coefficient Score for both KMeans and Agglomerative Clustering

wcss = []
sil_kmeans = []
sil_agc = []

for i in range(3,9):

    # K-Means Clustering:
    kmeans = KMeans(n_clusters = i, n_init = 100, init='k-means++',
random_state = 0)
    kmeans.fit(X_PCA_7)

     # Inertia and Silhouette Score for Clusters using K-Means:
    in_km = kmeans.inertia_
    wcss.append(in_km)
    sil_km = silhouette_score(X_PCA_7, kmeans.labels_)
    sil_kmeans.append(sil_km)

    # Agglomerative Clusters and its Silhouette Score
    agc = AgglomerativeClustering(n_clusters = i, affinity =
'cityblock', linkage = 'average')
    agc.fit(X_PCA_7)
    sil_ag = silhouette_score(X_PCA_7, agc.labels_)
    sil_agc.append(sil_ag)

    print('Number of clusters:', i)
    print('KMeans Inertia', in_km)
```

```python
    print('Silhouette Score for KMeans:', sil_km)
    print('Silhouette Score for AGC(HCA):', sil_ag)
    print()
```

```
Number of clusters: 3
KMeans Inertia 74229.6994039248
Silhouette Score for KMeans: 0.24784561379166523
Silhouette Score for AGC(HCA): 0.8499511136280253

Number of clusters: 4
KMeans Inertia 63708.48198506316
Silhouette Score for KMeans: 0.25984073654156464
Silhouette Score for AGC(HCA): 0.8276634947082336

Number of clusters: 5
KMeans Inertia 55828.66164055138
Silhouette Score for KMeans: 0.2887017139510505
Silhouette Score for AGC(HCA): 0.7930548693789233

Number of clusters: 6
KMeans Inertia 48750.07890336938
Silhouette Score for KMeans: 0.28855784749665525
Silhouette Score for AGC(HCA): 0.7907514561247028

Number of clusters: 7
KMeans Inertia 42422.40591665273
Silhouette Score for KMeans: 0.3023889650761613
Silhouette Score for AGC(HCA): 0.7550012654147571

Number of clusters: 8
KMeans Inertia 38669.582949106465
Silhouette Score for KMeans: 0.3074856699207035
Silhouette Score for AGC(HCA): 0.7545763220623827
```

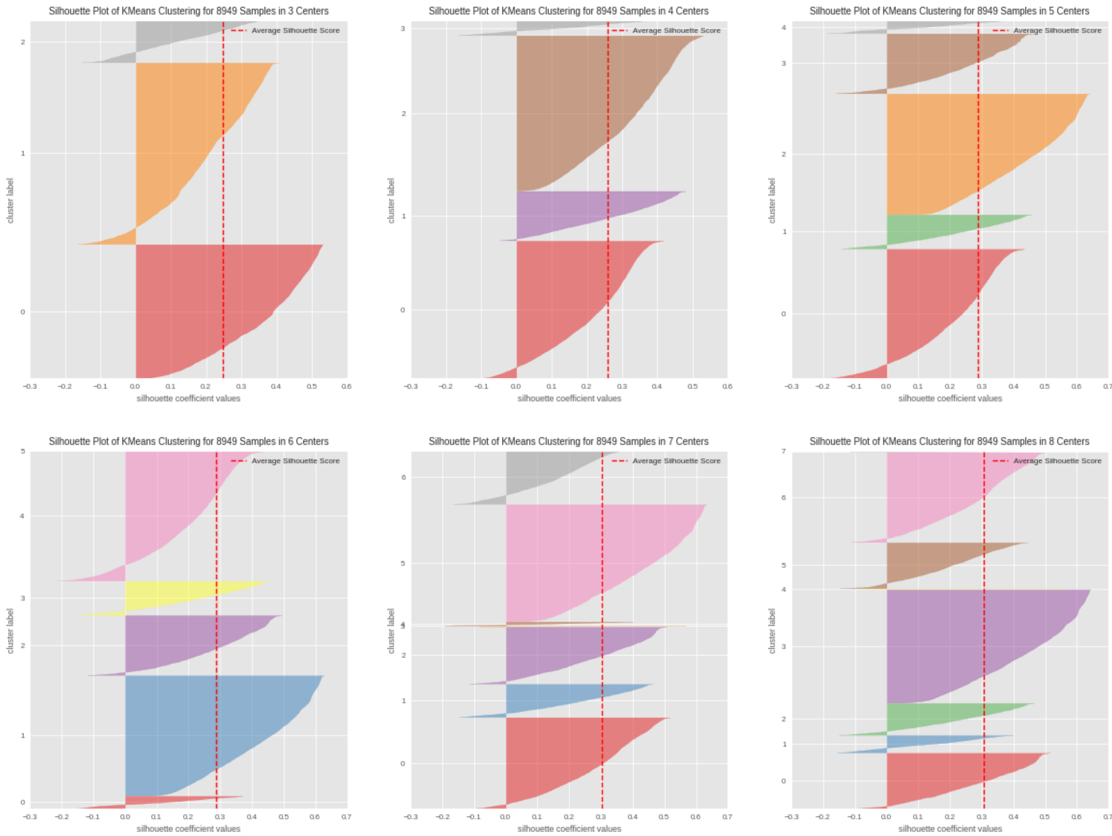
```
!pip install yellowbrick
```

```
Requirement already satisfied: yellowbrick in
/usr/local/lib/python3.7/dist-packages (1.4)
Requirement already satisfied: scikit-learn>=1.0.0 in
/usr/local/lib/python3.7/dist-packages (from yellowbrick) (1.0.2)
Requirement already satisfied: scipy>=1.0.0 in
/usr/local/lib/python3.7/dist-packages (from yellowbrick) (1.4.1)
Requirement already satisfied: matplotlib!=3.0.0,>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from yellowbrick) (3.2.2)
Requirement already satisfied: cycler>=0.10.0 in
/usr/local/lib/python3.7/dist-packages (from yellowbrick) (0.11.0)
Requirement already satisfied: numpy>=1.16.0 in
/usr/local/lib/python3.7/dist-packages (from yellowbrick) (1.21.5)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!
```

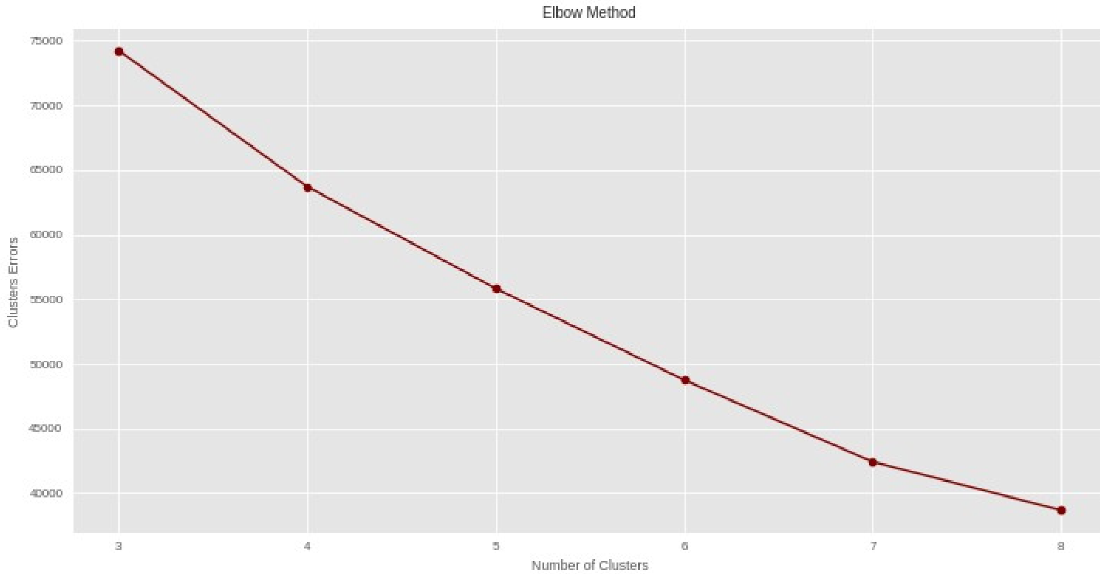
```
=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from
matplotlib!=3.0.0,>=2.0.2->yellowbrick) (3.0.7)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib!
=3.0.0,>=2.0.2->yellowbrick) (1.3.2)
Requirement already satisfied: python-dateutil>=2.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib!
=3.0.0,>=2.0.2->yellowbrick) (2.8.2)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.7/dist-packages (from python-dateutil>=2.1-
>matplotlib!=3.0.0,>=2.0.2->yellowbrick) (1.15.0)
Requirement already satisfied: joblib>=0.11 in
/usr/local/lib/python3.7/dist-packages (from scikit-learn>=1.0.0-
>yellowbrick) (1.1.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.7/dist-packages (from scikit-learn>=1.0.0-
>yellowbrick) (3.1.0)
```

```
from yellowbrick.cluster import SilhouetteVisualizer
```

```
plt.style.use('seaborn-paper')
fig, axs = plt.subplots(2, 3, figsize=(20, 15))
axs = axs.reshape(6)
for i, k in enumerate(range(3, 9)):
    ax = axs[i]
    sil = SilhouetteVisualizer(KMeans(n_clusters = k, n_init = 100,
init='k-means++', random_state = 0), ax=ax)
    sil.fit(X_PCA_7)
    sil.finalize()
```

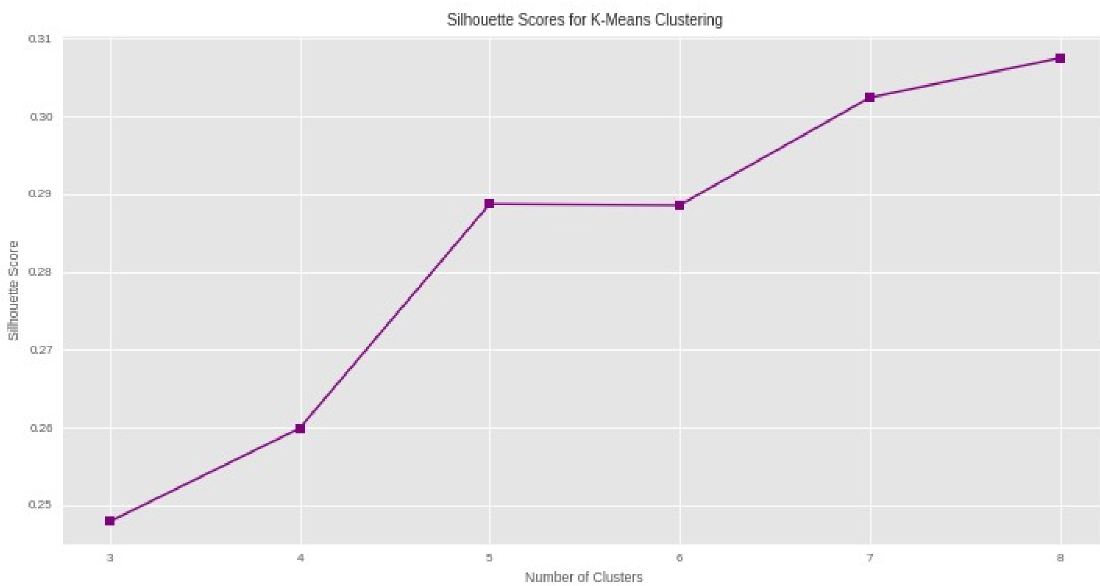



```python
# Plotting graph of Elbow Method

plt.figure(figsize=(12,6))
plt.plot(range(3,9), wcss, c ='#800000', marker='o')
plt.title('Elbow Method')
plt.xlabel('Number of Clusters')
plt.ylabel('Clusters Errors')
plt.show()

# Inertia or Sum of Squared Errors within the Clusters is also known
as the Cluster Errors

# CLuster error will decrease after some Clusters but
```

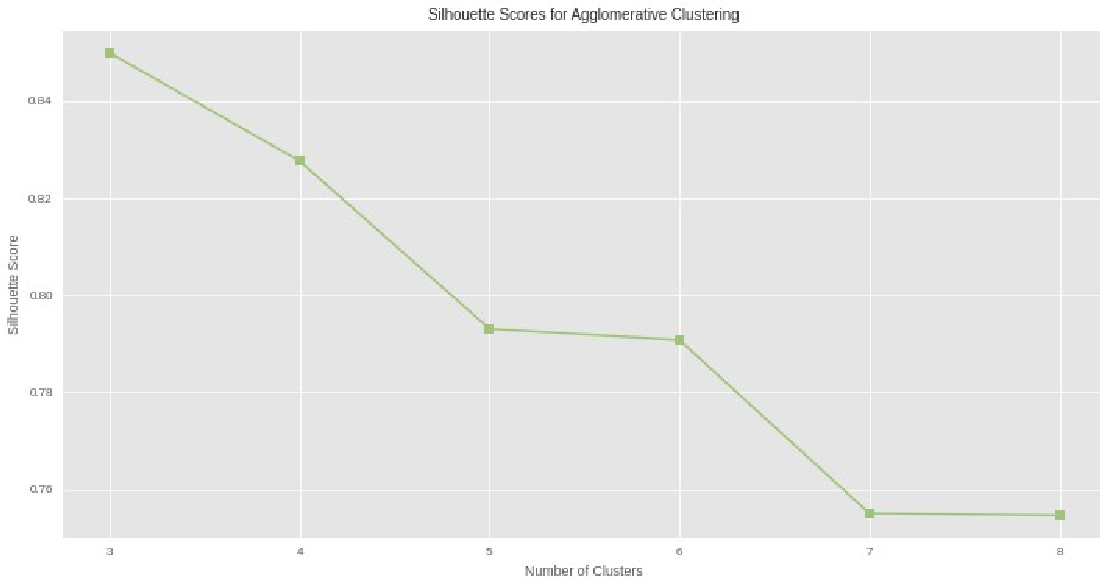


Inference of Elbow Method: The sum of squared distances of each data point within a cluster from its respective centroid is called the inertia. The K at which the inertia stops to drop significantly (using the above elbow method) is the best K.

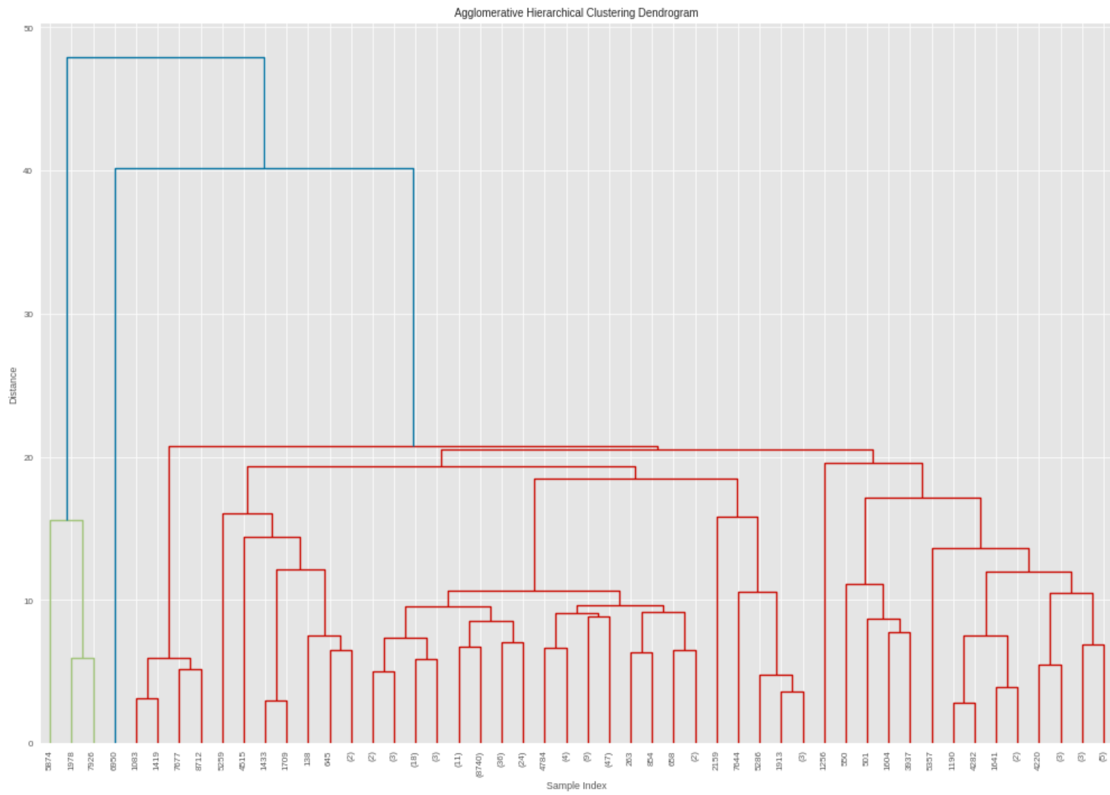
```
# Plotting the Silhouette Score for the clusters found from K-Means
and Agglomerative Clustering

plt.figure(figsize=(12,6))
plt.plot(range(3,9), sil_kmeans, marker='s', c='purple')
plt.title('Silhouette Scores for K-Means Clustering')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')
plt.show()
```



```
# Plotting the Silhouette Score for the clusters found from K-Means
and Agglomerative Clustering

plt.figure(figsize=(12,6))
plt.plot(range(3,9), sil_agc, marker='s', c='g')
plt.title('Silhouette Scores for Agglomerative Clustering')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')
plt.show()
```




```
# Plotting Dendrogram

Z= linkage(concat_df, method = 'average', metric = 'euclidean')
plt.figure(figsize=(14,10))
plt.title('Agglomerative Hierarchical Clustering Dendrogram')
plt.xlabel('Sample Index')
plt.ylabel('Distance')
dendrogram(Z, leaf_rotation=90, leaf_font_size=8,
truncate_mode='level', p =9)
plt.tight_layout()
plt.show()

# p-value tells how the deep the Dendrogram goes. The lesser the p
value then the values would be far away on the x-axis
```



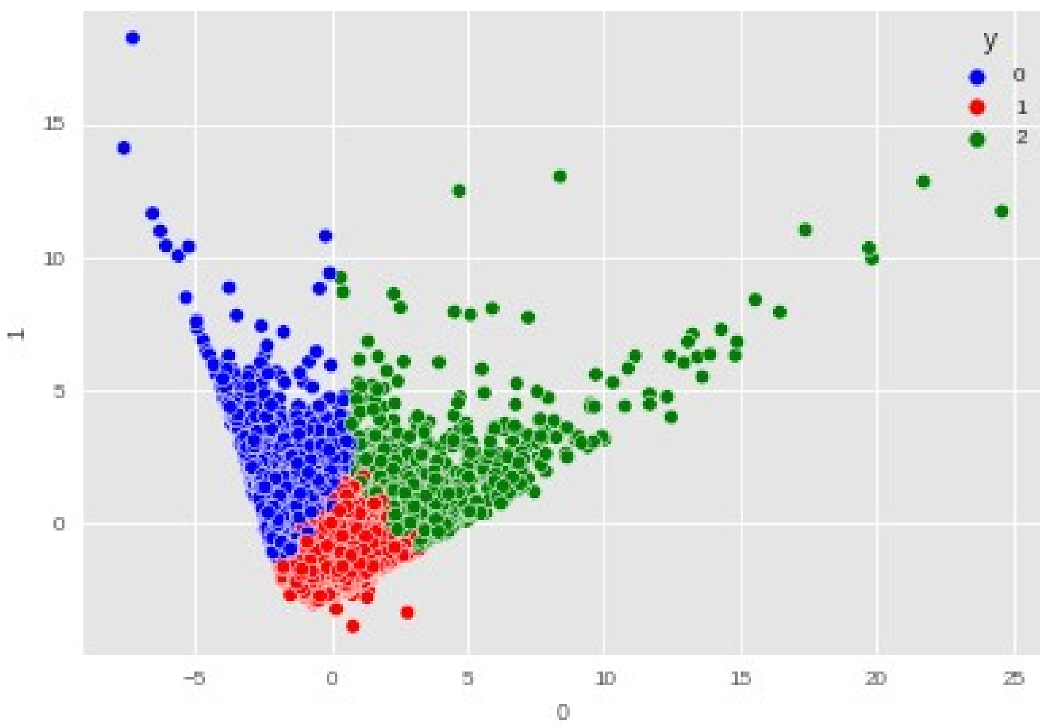
Conclusion: From the above Elbow method, Silhouette Coefficient Scores for K-Means & Agglomerative Clustering, and from Dendrogram, can see that the clusters 4 and 5 look similar.

We see that the Silhouette Scores for K = 5 is the highest (0.28857) and then the Silhouette Coefficient for K = 4 is 0.26015, which also gives the nearby score. The clusters K = 4 or K = 5 look very similar so now will use the other methods and best practices that is by finding out the Segment Distribution and performing Profiling, will check the similarities and dissimilarities between the segments and see which cluster is giving the best solution.

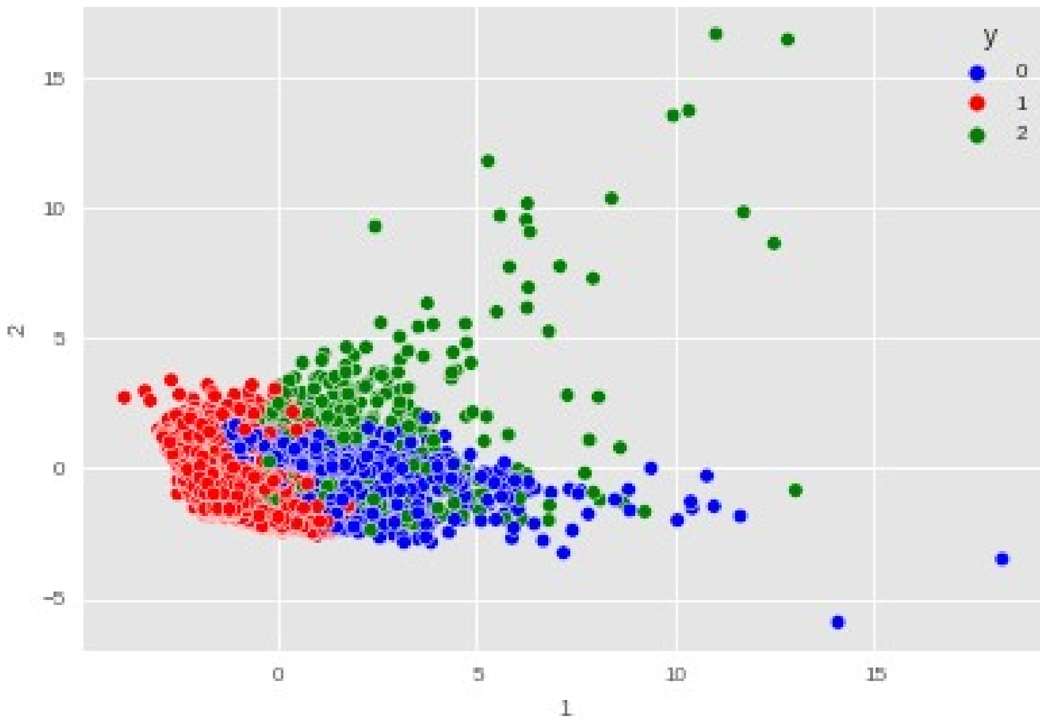
```
### Applying Clustering and visualizing the spread of the data
(finding out if the data points have been clustered correctly through
visualization)

#K-Means Clusters:   For K= 3
kmeans = KMeans(n_clusters = 3, n_init = 100, init='k-means++',
random_state = 0)
kmeans.fit(X_PCA_7)

# Taking into each dataframes
df_pca = pd.DataFrame(X_PCA_7)
y_lab = pd.Series(kmeans.labels_, name = 'y') # labels for clusters

#concatenating the dataframe:
df_final = pd.concat([df_pca, y_lab], axis = 1)
```

```
# As there are 7 dimensions, hence we need to plot for each of the
different pairs to visualize the spread of the data:

for i in range(6):
    print('Scatter plot for Principal Components', i, 'and', i+1)
    sns.scatterplot(df_pca[i], df_pca[i+1], hue = df_final['y'],
palette = ['blue', 'red', 'green'])
    plt.show()
```

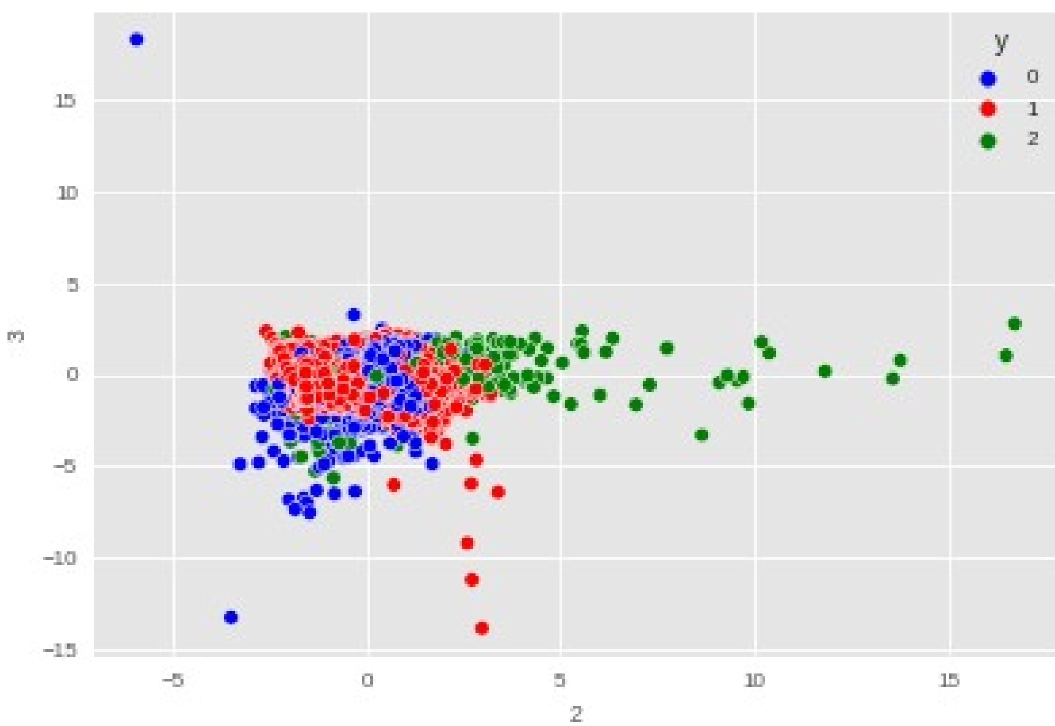
```
Scatter plot for Principal Components 0 and 1
```



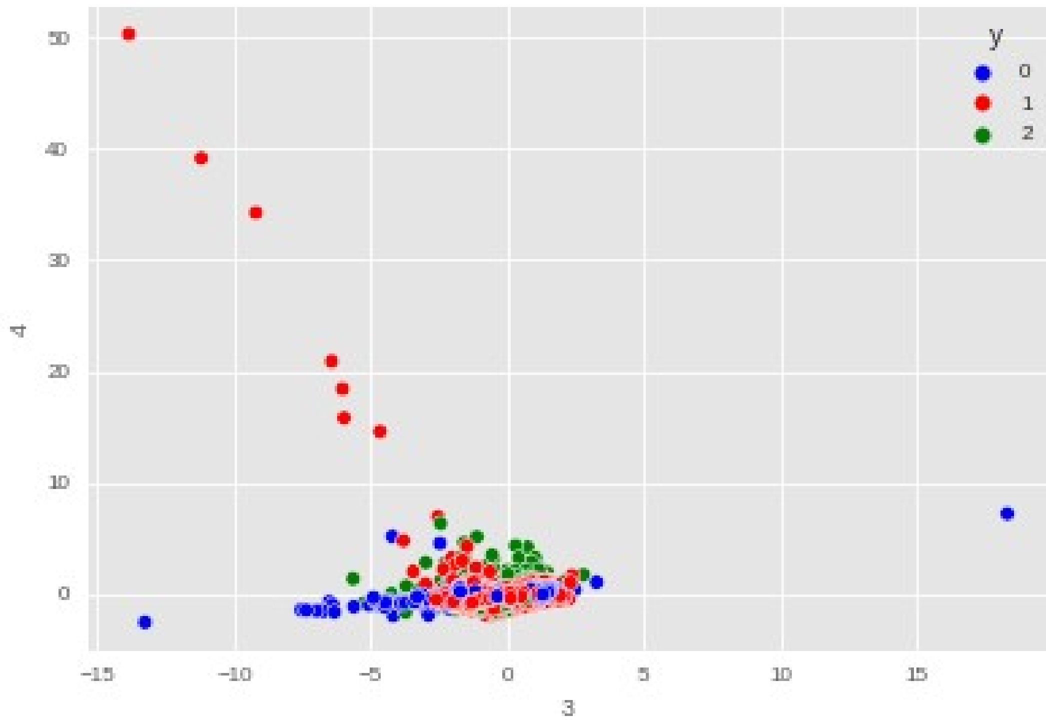

```
Scatter plot for Principal Components 1 and 2
```



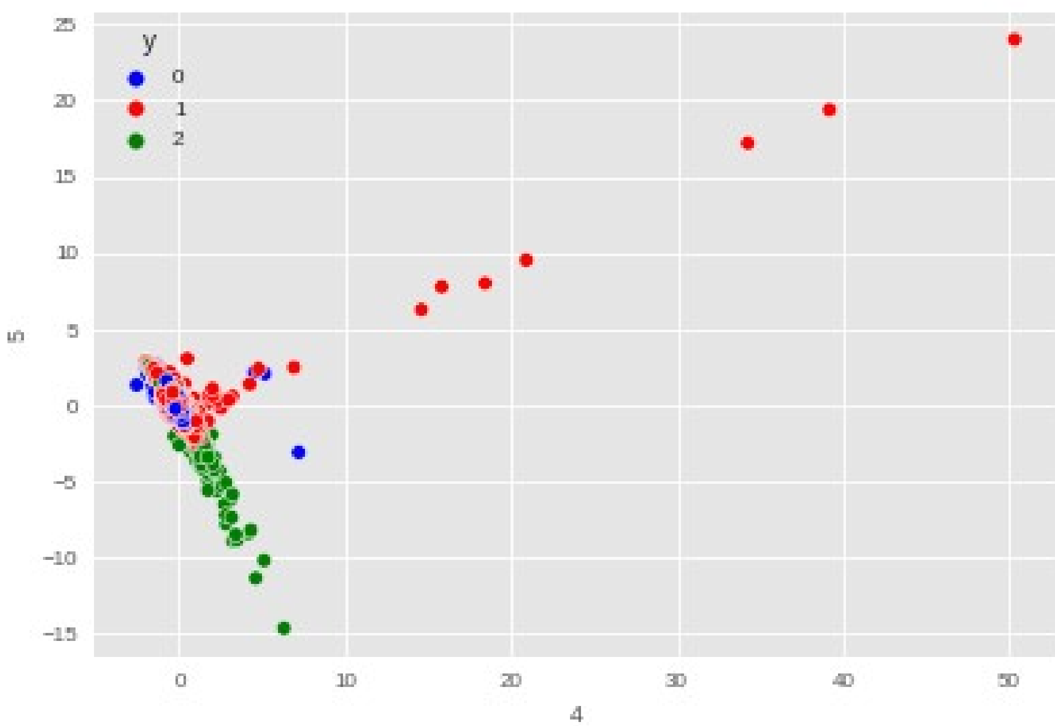
Scatter plot for Principal Components 2 and 3



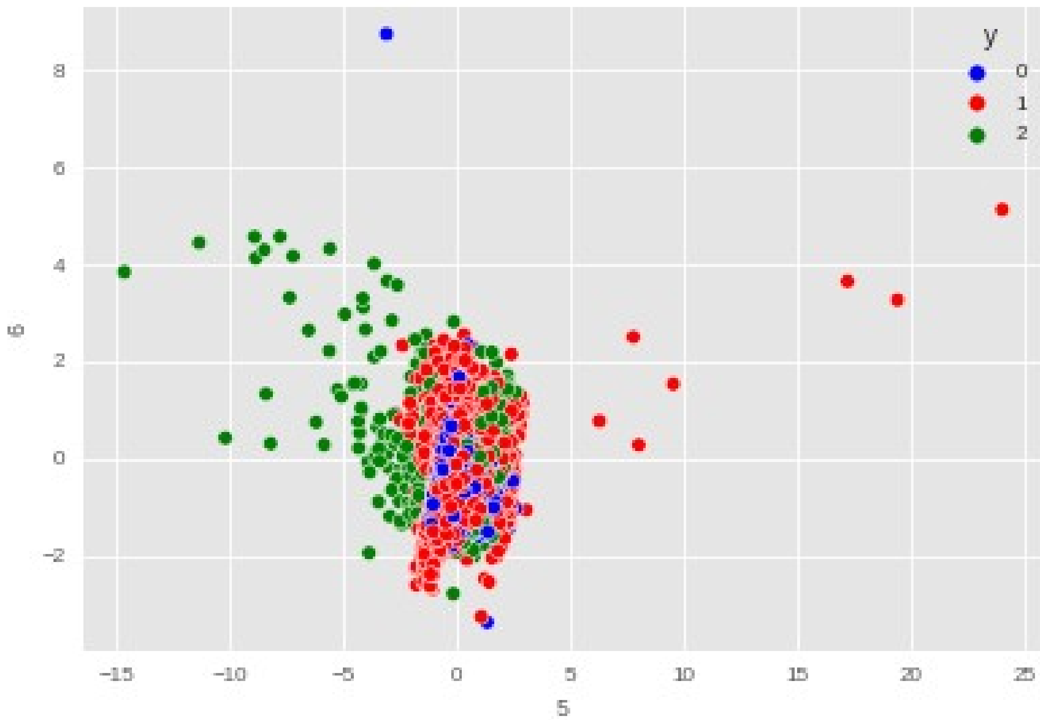
Scatter plot for Principal Components 3 and 4



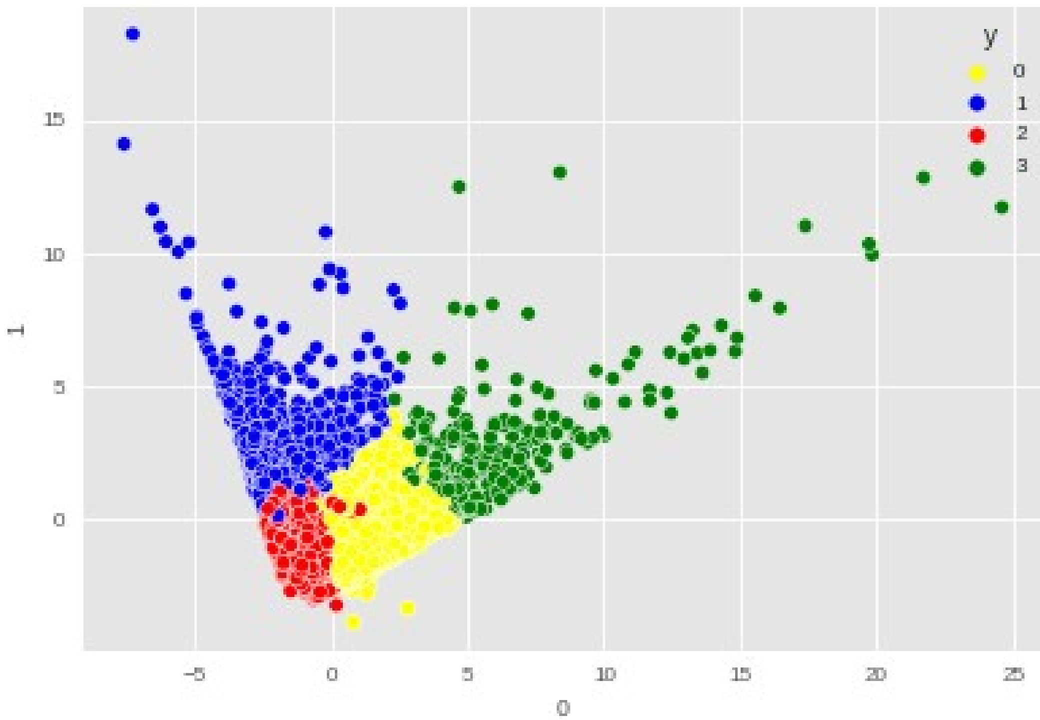
Scatter plot for Principal Components 4 and 5



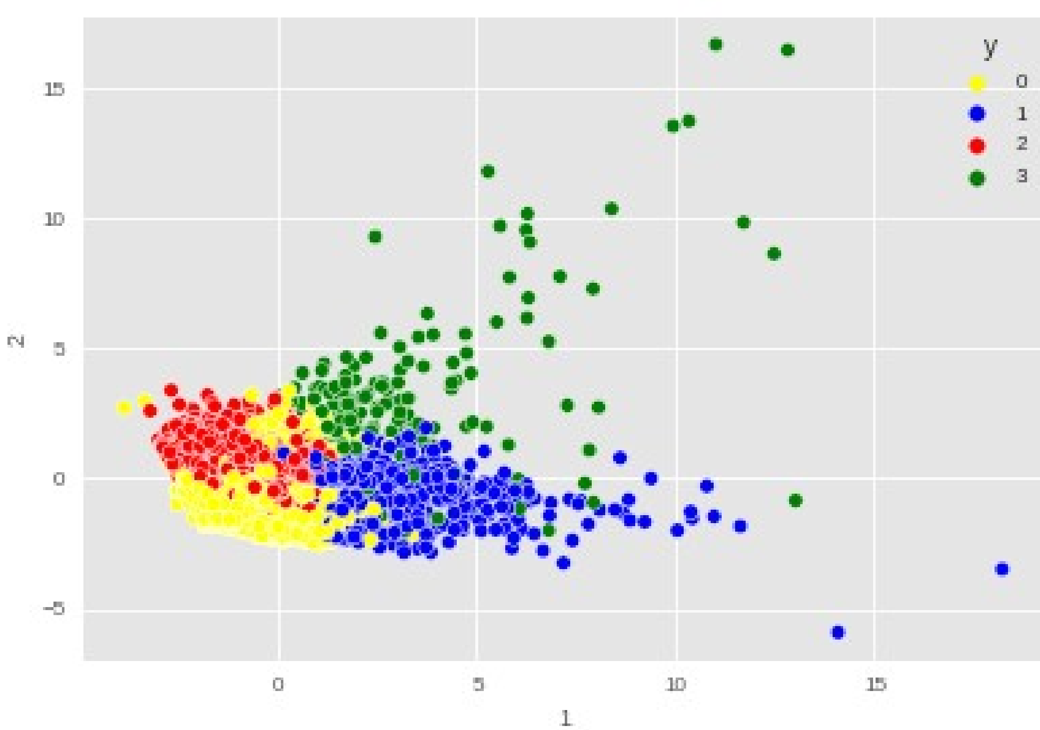
Scatter plot for Principal Components 5 and 6



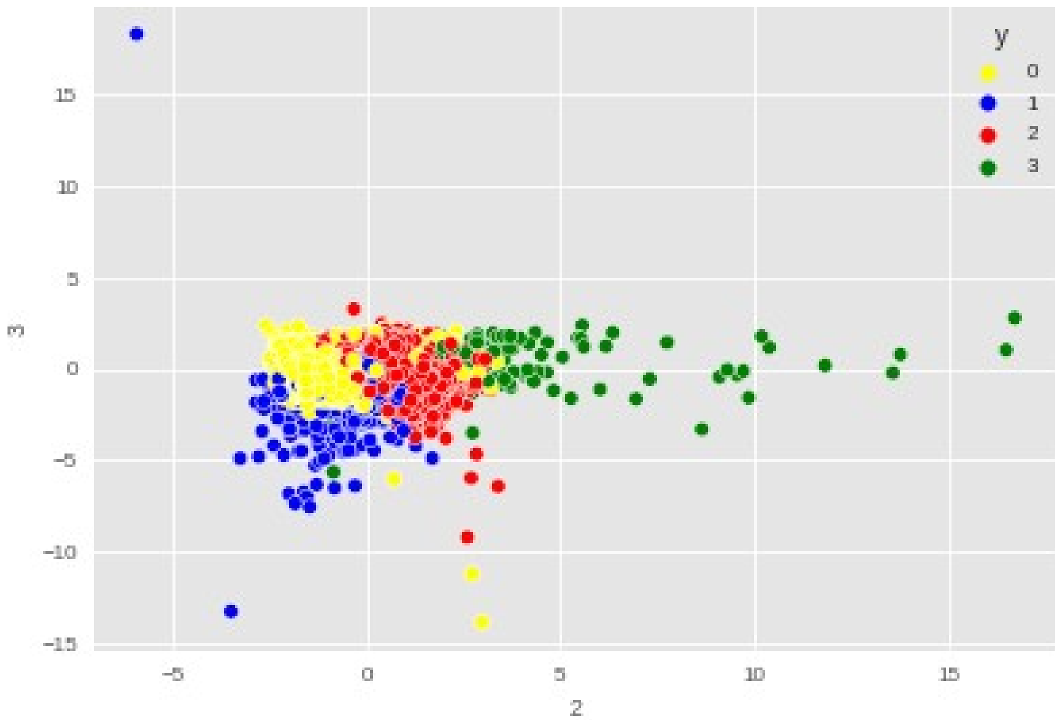

```
### Applying Clustering and visualizing the spread of the data
(finding out if the data points have been clustered correctly through
visualization)

#K-Means Clusters:   For K= 4
kmeans = KMeans(n_clusters = 4, n_init = 100, init='k-means++',
random_state = 0)
kmeans.fit(X_PCA_7)

# Taking into each dataframes
df_pca = pd.DataFrame(X_PCA_7)
y_lab = pd.Series(kmeans.labels_, name = 'y') # labels for clusters

#concatenating the dataframe:
df_final = pd.concat([df_pca, y_lab], axis = 1)

# As there are 7 dimensions, hence we need to plot for each of the
different pairs to visualize the spread of the data:

for i in range(6):
    print('Scatter plot for Principal Components', i, 'and', i+1)
    sns.scatterplot(df_pca[i], df_pca[i+1], hue = df_final['y'],
palette = ['yellow', 'blue', 'red', 'green'])
    plt.show()

Scatter plot for Principal Components 0 and 1
```



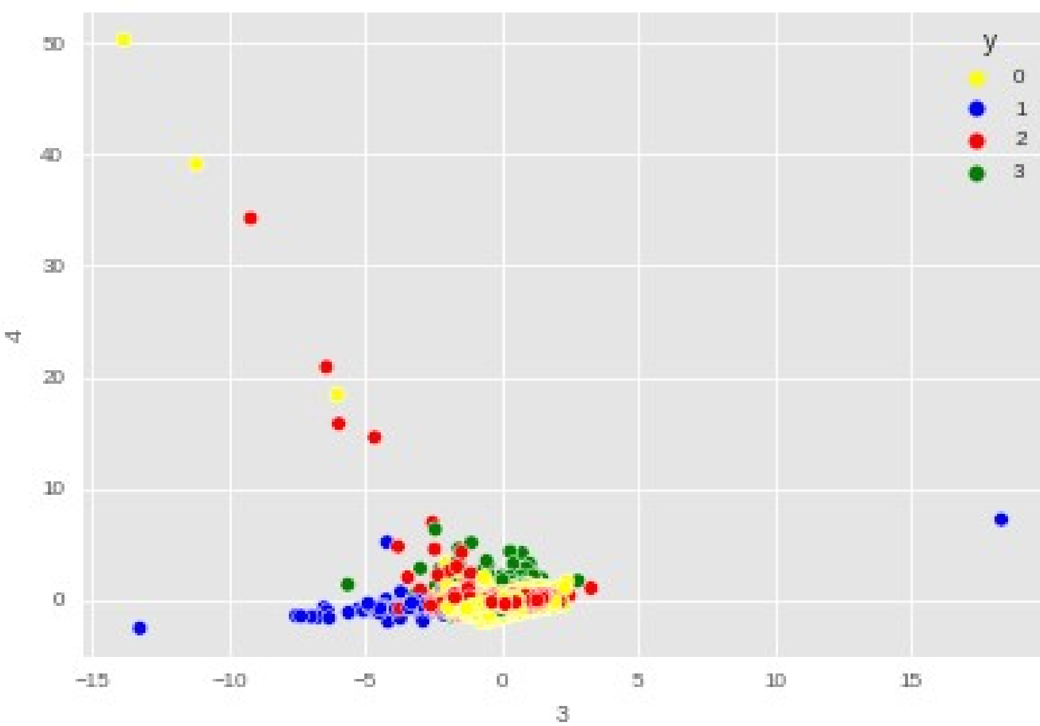
Scatter plot for Principal Components 1 and 2



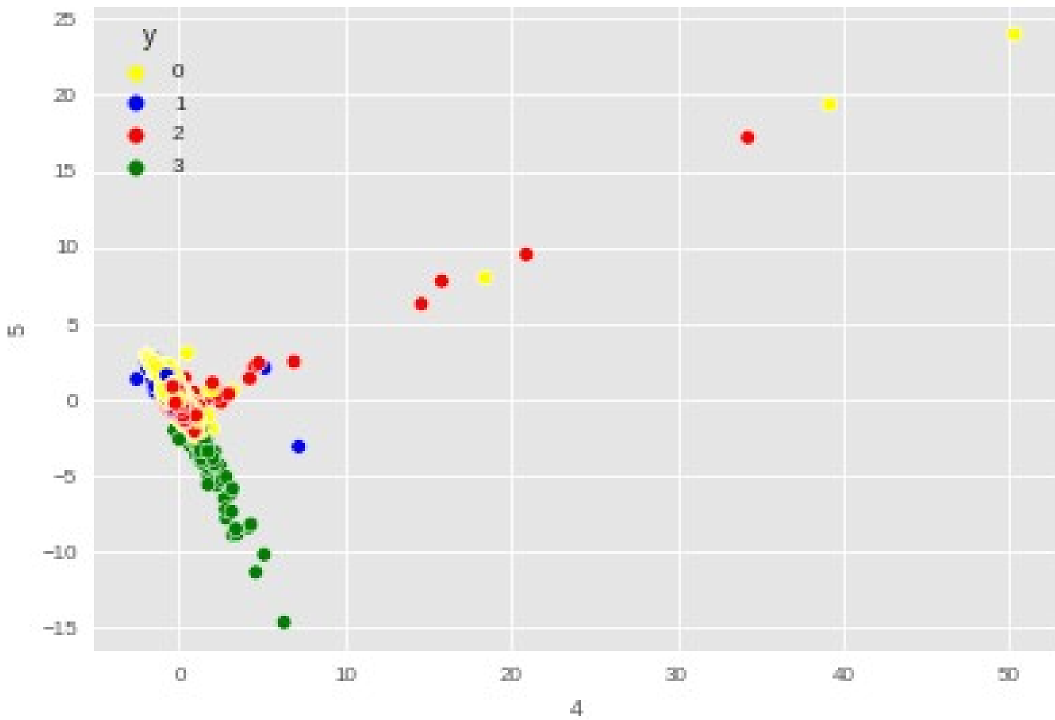
Scatter plot for Principal Components 2 and 3



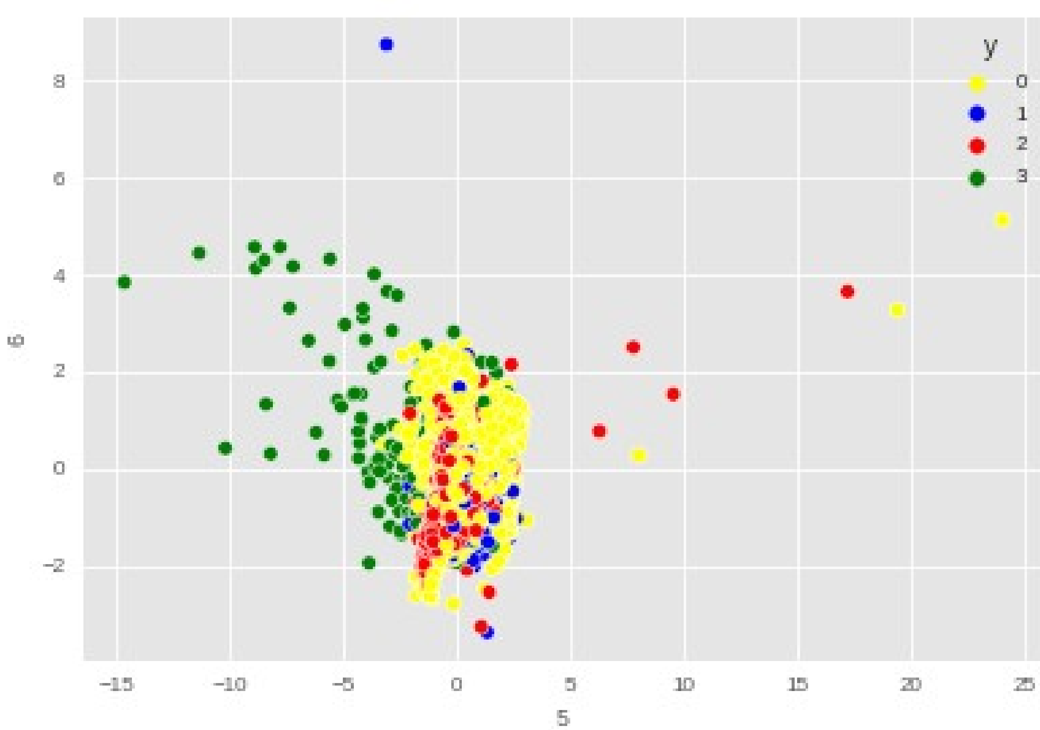
Scatter plot for Principal Components 3 and 4



Scatter plot for Principal Components 4 and 5



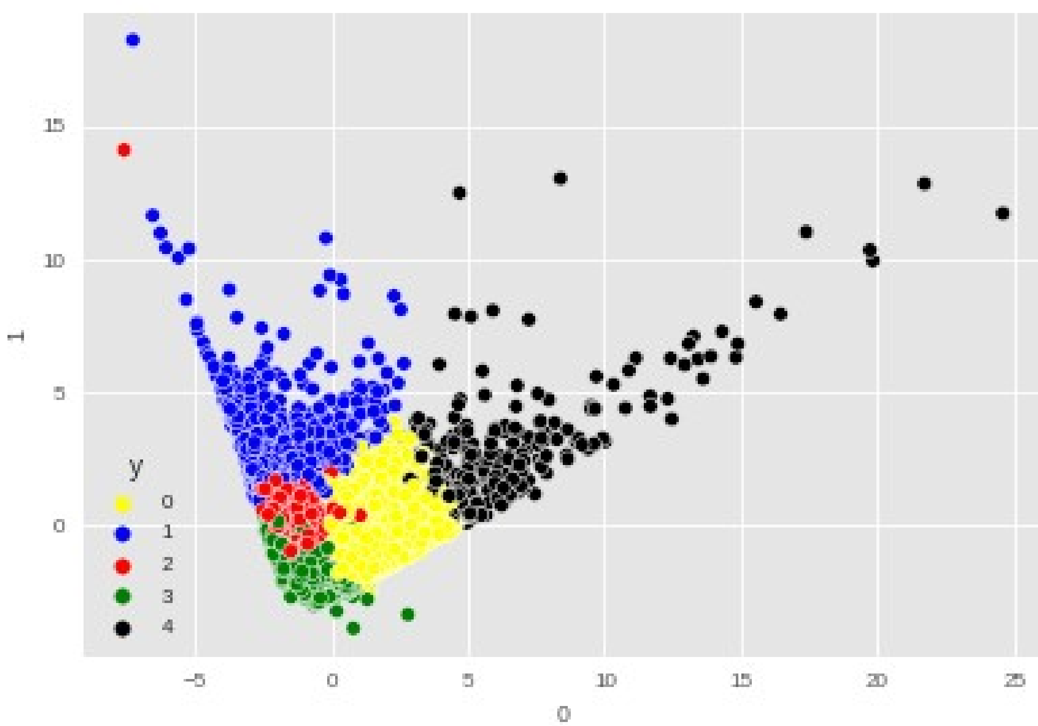
Scatter plot for Principal Components 5 and 6



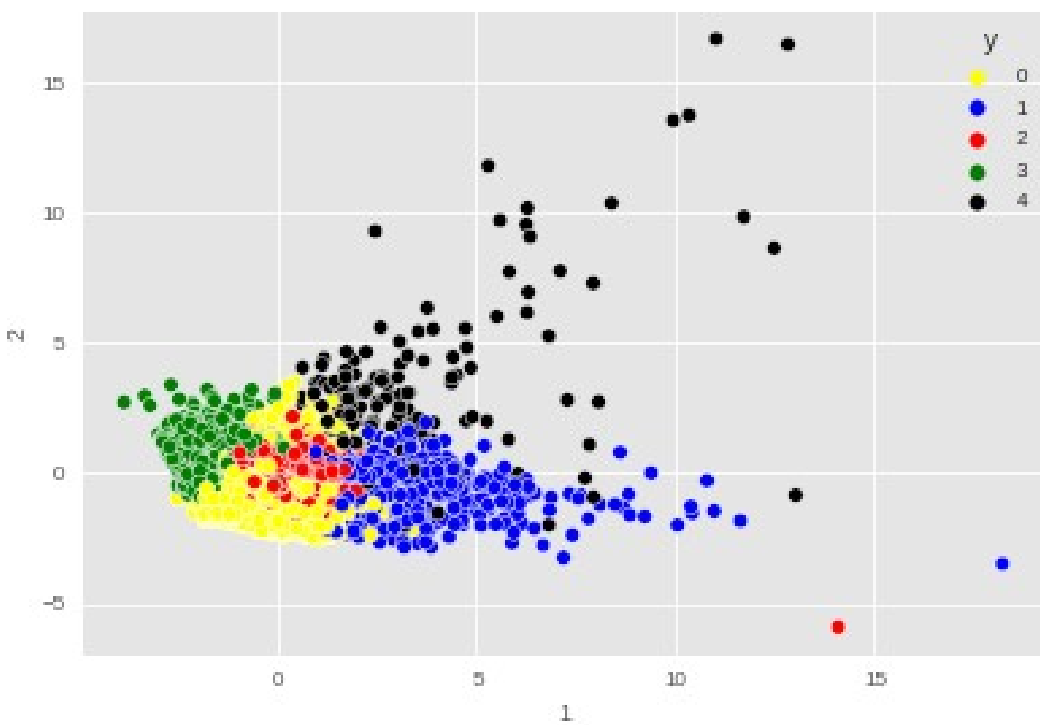
Applying Clustering and visualizing the spread of the data (finding out if the data points have been clustered correctly through visualization)

```python
#K-Means Clusters:   For K= 5
kmeans = KMeans(n_clusters = 5, n_init = 100, init='k-means++',
random_state = 0)
kmeans.fit(X_PCA_7)

# Taking into each dataframes
df_pca = pd.DataFrame(X_PCA_7)
y_lab = pd.Series(kmeans.labels_, name = 'y') # labels for clusters

#concatenating the dataframe:
df_final = pd.concat([df_pca, y_lab], axis = 1)

# As there are 7 dimensions, hence we need to plot for each of the
different pairs to visualize the spread of the data:

for i in range(6):
    print('Scatter plot for Principal Components', i, 'and', i+1)
    sns.scatterplot(df_pca[i], df_pca[i+1], hue = df_final['y'],
palette = ['yellow', 'blue', 'red', 'green','black'])
    plt.show()
```

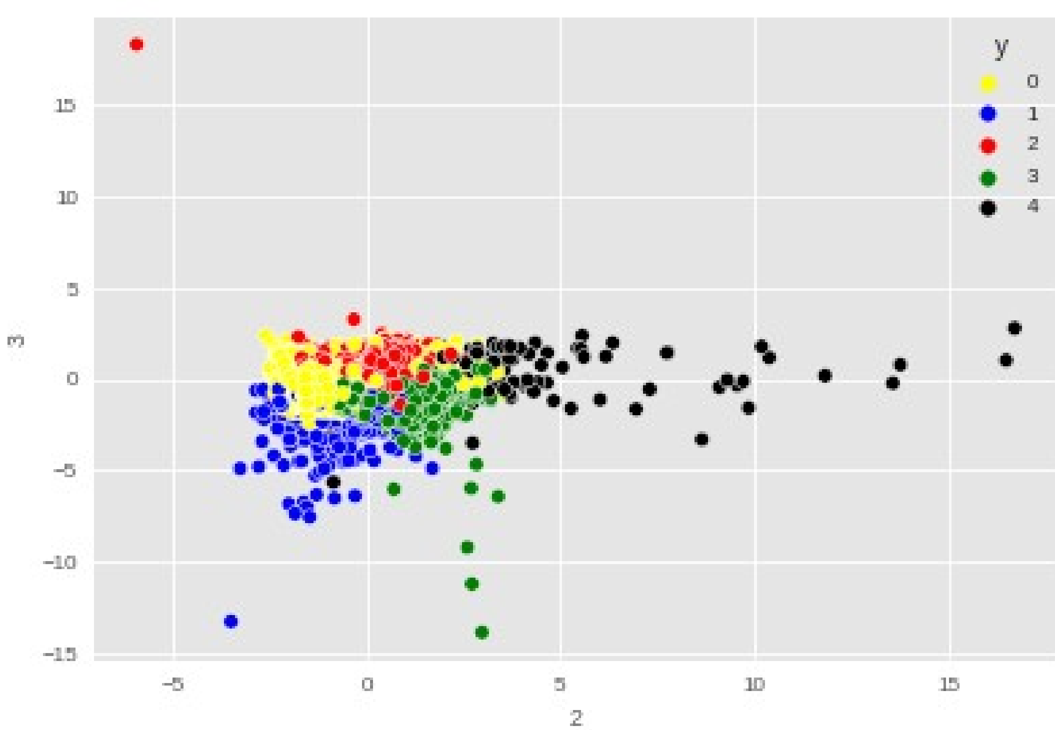
Scatter plot for Principal Components 0 and 1



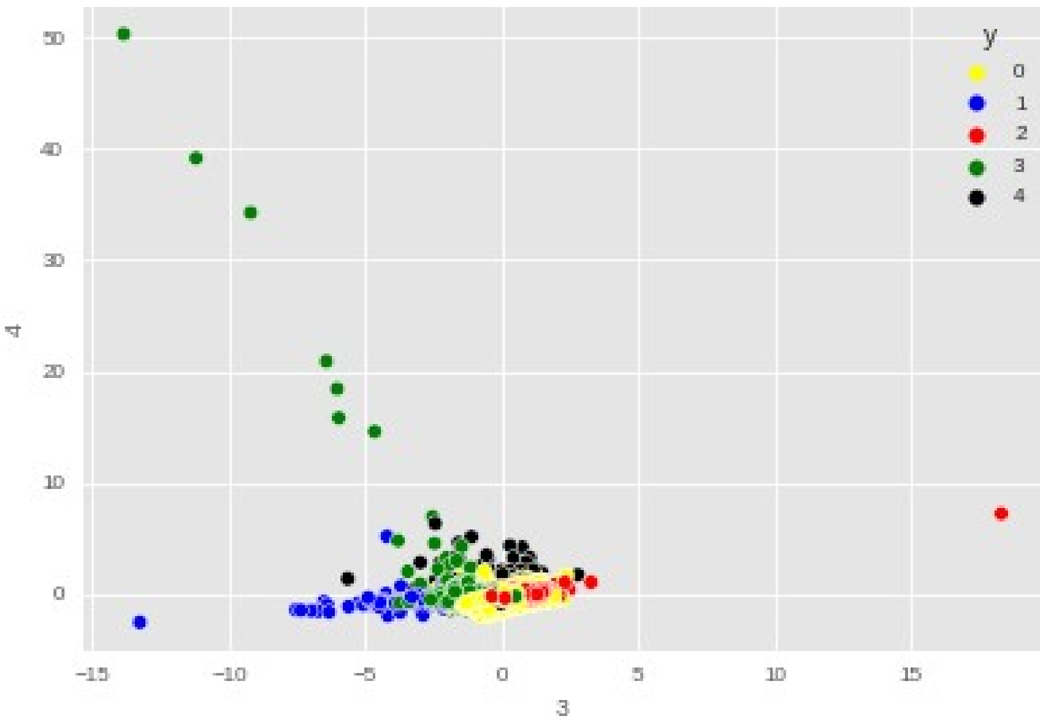
Scatter plot for Principal Components 1 and 2



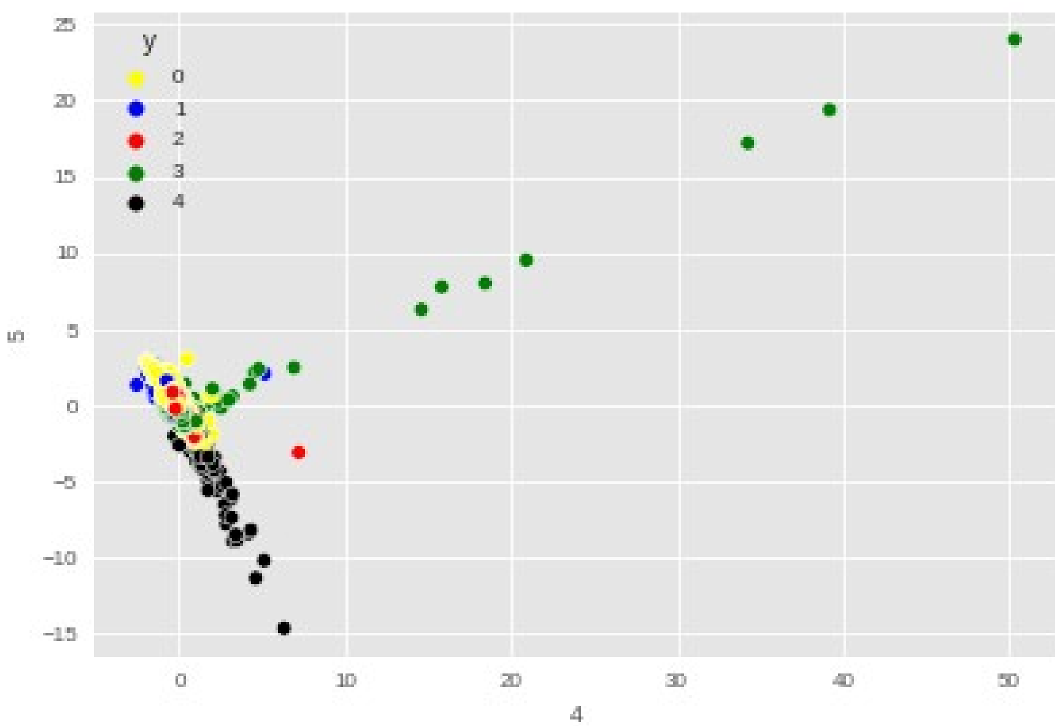
Scatter plot for Principal Components 2 and 3



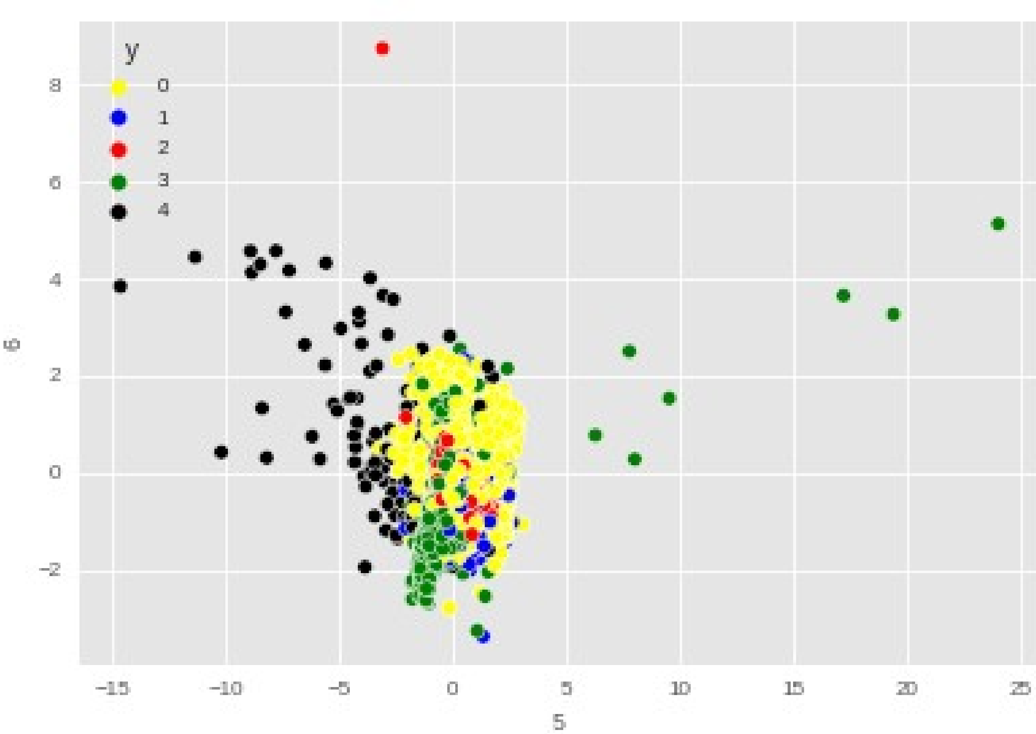
Scatter plot for Principal Components 3 and 4



Scatter plot for Principal Components 4 and 5



Scatter plot for Principal Components 5 and 6




```
# Step 1: Making K-Means Cluster and Labels for finding out the
distribution of Segments and then performing Profiling

# K = 3

km_3 = KMeans(n_clusters = 3, n_init = 100, init='k-means++',
random_state = 0)
KM_3 = km_3.fit(X_PCA_7)

# Labels of Cluster 3

KM_3.labels_

array([1, 0, 1, ..., 1, 0, 1], dtype=int32)

# Centroids for Cluster 3:

KM_3.cluster_centers_

array([[-1.76479658,  0.93587491,  0.13620598,  0.18538487,
0.04654055,
        -0.06395186,  0.23937914],
       [ 0.43005603, -0.99827666, -0.21376279, -0.15160496, -0.025684
,
         0.01894025, -0.13917789],
       [ 3.79706665,  1.3431978 ,  0.49391727,  0.06497734, -
0.03762582,
         0.12296967, -0.16269885]])
```

```
KM_4 = KMeans(n_clusters = 4, n_init = 100, init='k-means++',
random_state = 0).fit(X_PCA_7)
KM_5 = KMeans(n_clusters = 5, n_init = 100, init='k-means++',
random_state = 0).fit(X_PCA_7)
KM_6 = KMeans(n_clusters = 6, n_init = 100, init='k-means++',
random_state = 0).fit(X_PCA_7)
KM_7 = KMeans(n_clusters = 7, n_init = 100, init='k-means++',
random_state = 0).fit(X_PCA_7)
KM_8 = KMeans(n_clusters = 8, n_init = 100, init='k-means++',
random_state = 0).fit(X_PCA_7)

# Appending the Cluster labels to the Original Data: (not to
Standardized data)

original_df['cluster_3'] = KM_3.labels_
original_df['cluster_4'] = KM_4.labels_
original_df['cluster_5'] = KM_5.labels_
original_df['cluster_6'] = KM_6.labels_
original_df['cluster_7'] = KM_7.labels_
original_df['cluster_8'] = KM_8.labels_

# the new data set has Original variables + the Cluster Labels from
each of the clusters got from K-Means

original_df.head()

   Installment_Purchases  None_Of_the_Purchases  ...  cluster_7
cluster_8
0                      1                      0  ...          2
0
1                      0                      1  ...          5
3
2                      0                      0  ...          6
5
3                      0                      0  ...          2
0
4                      0                      0  ...          5
3

[5 rows x 30 columns]

# Finding the Segment Distribution for cluster K = 3 :

pd.Series.sort_index(original_df.cluster_3.value_counts())/sum(origina
l_df.cluster_3.value_counts())

0    0.375237
1    0.507990
2    0.116773
Name: cluster_3, dtype: float64
```

```
# Segment Distribution for cluster K = 4 :

pd.Series.sort_index(original_df.cluster_4.value_counts())/sum(origina
l_df.cluster_4.value_counts())

0    0.385741
1    0.138339
2    0.435132
3    0.040787
Name: cluster_4, dtype: float64

# Segment Distribution for cluster K = 5 :

pd.Series.sort_index(original_df.cluster_5.value_counts())/sum(origina
l_df.cluster_5.value_counts())

0    0.363169
1    0.095430
2    0.338474
3    0.167728
4    0.035199
Name: cluster_5, dtype: float64

# Segment Distribution for cluster K = 6 :

pd.Series.sort_index(original_df.cluster_6.value_counts())/sum(origina
l_df.cluster_6.value_counts())

0    0.035199
1    0.337691
2    0.168063
3    0.095206
4    0.363057
5    0.000782
Name: cluster_6, dtype: float64

# Segment Distribution for cluster K = 7 :

pd.Series.sort_index(original_df.cluster_7.value_counts())/sum(origina
l_df.cluster_7.value_counts())

0    0.256341
1    0.092748
2    0.159906
3    0.000782
4    0.011510
5    0.328975
6    0.149737
Name: cluster_7, dtype: float64
```

```
# Segment Distribution for cluster K = 8 :

pd.Series.sort_index(original_df.cluster_8.value_counts())/sum(origina
l_df.cluster_8.value_counts())

0    0.156554
1    0.048721
2    0.089395
3    0.319365
4    0.000782
5    0.129065
6    0.252989
7    0.003129
Name: cluster_8, dtype: float64

# Step 1a: Get the total size of the cluster:
original_df.cluster_3.size

# Step 1b: Get the break up of the values in each segment:
# which gives how many observations are there in each of the
respective segment:

original_df.cluster_3.value_counts()

1    4546
0    3358
2    1045
Name: cluster_3, dtype: int64

# by using the Sort Index provides:
# the value counts based on the Segment Label (0,1,2 depending upon
the K-value) in the index
# and not based on the highest value within the segments

pd.Series.sort_index(original_df.cluster_3.value_counts())

0    3358
1    4546
2    1045
Name: cluster_3, dtype: int64

# combining the size for each cluster K value into one single array:

size=pd.concat([pd.Series(original_df.cluster_3.size),
pd.Series.sort_index(original_df.cluster_3.value_counts()),
pd.Series.sort_index(original_df.cluster_4.value_counts()),
           pd.Series.sort_index(original_df.cluster_5.value_counts()),
pd.Series.sort_index(original_df.cluster_6.value_counts()),
           pd.Series.sort_index(original_df.cluster_7.value_counts()),
pd.Series.sort_index(original_df.cluster_8.value_counts())])
```

```
# Gives the size of Segments for each of the Clusters :

size

0    8949
0    3358
1    4546
2    1045
0    3452
1    1238
2    3894
3     365
0    3250
1     854
2    3029
3    1501
4     315
0     315
1    3022
2    1504
3     852
4    3249
5       7
0    2294
1     830
2    1431
3       7
4     103
5    2944
6    1340
0    1401
1     436
2     800
3    2858
4       7
5    1155
6    2264
7      28
dtype: int64

# Segment Size:
Seg_size=pd.DataFrame(size, columns=['Seg_size'])

# Segment Distribtuion % wise:
Seg_Pct = pd.DataFrame(size/original_df.cluster_3.size,
columns=['Seg_Pct'])

# Taking Transpose of Segment Percentage :
Seg_Pct.T
```

```
           0         0        1  ...         5         6         7
Seg_Pct  1.0  0.375237  0.50799  ...  0.129065  0.252989  0.003129

[1 rows x 34 columns]
```

```
# Concatenating the Segment Size and Segment Percentage:
pd.concat([Seg_size.T, Seg_Pct.T], axis=0)
```

```
               0            0           1  ...            5
6          7
Seg_size  8949.0  3358.000000  4546.00000  ...  1155.000000
2264.000000  28.000000
Seg_Pct      1.0     0.375237     0.50799  ...     0.129065
0.252989   0.003129

[2 rows x 34 columns]
```

```
# Overall each variables wise Avg:
original_df.apply(np.mean).T
```

```
Installment_Purchases                       0.252542
None_Of_the_Purchases                       0.228070
One_Of_Purchase                             0.209409
BALANCE                                  1564.647593
BALANCE_FREQUENCY                           0.877350
PURCHASES                                1003.316936
ONEOFF_PURCHASES                          592.503572
INSTALLMENTS_PURCHASES                    411.113579
CASH_ADVANCE                              978.959616
PURCHASES_FREQUENCY                         0.490405
ONEOFF_PURCHASES_FREQUENCY                  0.202480
PURCHASES_INSTALLMENTS_FREQUENCY            0.364478
CASH_ADVANCE_FREQUENCY                      0.135141
CASH_ADVANCE_TRX                            3.249078
PURCHASES_TRX                              14.711476
CREDIT_LIMIT                             4494.449450
PAYMENTS                                 1733.336511
MINIMUM_PAYMENTS                          845.003358
PRC_FULL_PAYMENT                            0.153732
TENURE                                     11.517935
Monthly_Avg_Purchase                       86.184802
Monthly_Avg_Cash                           88.984447
Limit_Usage                                 0.388926
Pay_to_MinimumPay                           9.060094
cluster_3                                   0.741535
cluster_4                                   1.130964
cluster_5                                   1.416359
cluster_6                                   2.415577
cluster_7                                   3.004246
cluster_8                                   3.373897
dtype: float64
```

```
# Grouping-by over each cluster to find the Segment wise average for
each variable
original_df.groupby('cluster_3').apply(np.mean).T
```

```
cluster_3                                      0            1
2
Installment_Purchases                   0.073258     0.439067
0.017225
None_Of_the_Purchases                   0.565515     0.031236
0.000000
One_Of_Purchase                         0.252531     0.208755
0.073684
BALANCE                              2504.779770   672.690120
2423.857947
BALANCE_FREQUENCY                       0.944531     0.803102
0.984468
PURCHASES                             221.712067   720.561518
4744.977493
ONEOFF_PURCHASES                      164.722534   331.415458
3102.929694
INSTALLMENTS_PURCHASES                 57.142716   389.491912
1642.621962
CASH_ADVANCE                         2172.651223   163.805328
689.270600
PURCHASES_FREQUENCY                     0.149574     0.635632
0.953860
ONEOFF_PURCHASES_FREQUENCY              0.083552     0.173038
0.712726
PURCHASES_INSTALLMENTS_FREQUENCY        0.072448     0.490396
0.755114
CASH_ADVANCE_FREQUENCY                  0.294685     0.027915
0.088917
CASH_ADVANCE_TRX                        7.248064     0.486362
2.417225
PURCHASES_TRX                           3.011316    12.474263
62.041148
CREDIT_LIMIT                         4306.796073  3934.559375
7533.110048
PAYMENTS                             1766.018317  1073.394798
4499.221224
MINIMUM_PAYMENTS                     1146.964278   532.175664
1235.558319
PRC_FULL_PAYMENT                        0.026135     0.218709
0.281082
TENURE                                 11.371352    11.547074
11.862201
Monthly_Avg_Purchase                   20.261795    62.548595
400.844760
Monthly_Avg_Cash                      199.044571    14.503674
59.326736
```

```
Limit_Usage                                  0.633871     0.220250
0.335603
Pay_to_MinimumPay                            3.605150    12.537488
11.461496
cluster_3                                    0.000000     1.000000
2.000000
cluster_4                                    1.614056     0.783546
1.089952
cluster_5                                    1.782609     1.184558
1.247847
cluster_6                                    1.565813     2.965904
2.752153
cluster_7                                    3.852889     1.906951
5.050718
cluster_8                                    2.698332     3.869556
3.388517
```

```python
# Concatinating the above two averages:

Profiling_output = pd.concat([original_df.apply(lambda x: x.mean()).T,

                original_df.groupby('cluster_3').apply(lambda x:
x.mean()).T,
                original_df.groupby('cluster_4').apply(lambda x:
x.mean()).T,
                original_df.groupby('cluster_5').apply(lambda x:
x.mean()).T,
                original_df.groupby('cluster_6').apply(lambda x:
x.mean()).T,
                original_df.groupby('cluster_7').apply(lambda x:
x.mean()).T,
                original_df.groupby('cluster_8').apply(lambda x:
x.mean()).T], axis=1)

Profiling_output
```

```
                                                0  ...            7
Installment_Purchases                    0.252542  ...     0.071429
None_Of_the_Purchases                    0.228070  ...     0.000000
One_Of_Purchase                          0.209409  ...     0.071429
BALANCE                               1564.647593  ...  5761.648320
BALANCE_FREQUENCY                        0.877350  ...     0.980195
PURCHASES                             1003.316936  ... 25651.435714
ONEOFF_PURCHASES                       592.503572  ... 18455.715357
INSTALLMENTS_PURCHASES                 411.113579  ...  7195.720357
CASH_ADVANCE                           978.959616  ...  1459.599916
PURCHASES_FREQUENCY                      0.490405  ...     0.933929
ONEOFF_PURCHASES_FREQUENCY               0.202480  ...     0.799405
PURCHASES_INSTALLMENTS_FREQUENCY         0.364478  ...     0.772619
CASH_ADVANCE_FREQUENCY                   0.135141  ...     0.071429
CASH_ADVANCE_TRX                         3.249078  ...     3.250000
```

```
PURCHASES_TRX                              14.711476  ...    154.107143
CREDIT_LIMIT                             4494.449450  ...  15432.142857
PAYMENTS                                 1733.336511  ...  24033.806368
MINIMUM_PAYMENTS                          845.003358  ...   3630.480428
PRC_FULL_PAYMENT                            0.153732  ...      0.497700
TENURE                                     11.517935  ...     11.928571
Monthly_Avg_Purchase                       86.184802  ...   2153.442464
Monthly_Avg_Cash                           88.984447  ...    121.633326
Limit_Usage                                 0.388926  ...      0.407671
Pay_to_MinimumPay                           9.060094  ...     24.662878
cluster_3                                   0.741535  ...      2.000000
cluster_4                                   1.130964  ...      3.000000
cluster_5                                   1.416359  ...      4.000000
cluster_6                                   2.415577  ...      0.000000
cluster_7                                   3.004246  ...      4.000000
cluster_8                                   3.373897  ...      7.000000

[30 rows x 34 columns]

# Combining the outputs from steps 1 and 2:
# Concatenating the segment size, segment distribution, the overall
averages, and the individual segment-wise average

Profiling_output_final=pd.concat([Seg_size.T, Seg_Pct.T,
Profling_output], axis=0)

# Adding column names

Profiling_output_final.columns = ['Overall', 'KM3_1', 'KM3_2',
'KM3_3',
                                'KM4_1', 'KM4_2', 'KM4_3', 'KM4_4',
                                'KM5_1', 'KM5_2', 'KM5_3', 'KM5_4',
'KM5_5',
                                'KM6_1', 'KM6_2', 'KM6_3', 'KM6_4',
'KM6_5','KM6_6',
                                'KM7_1', 'KM7_2', 'KM7_3', 'KM7_4',
'KM7_5','KM7_6','KM7_7',
                                'KM8_1', 'KM8_2', 'KM8_3', 'KM8_4',
'KM8_5','KM8_6','KM8_7','KM8_8',]

Profling_output_final

# Exporting the output:
Profiling_output_final.to_csv('Profiling_output_final.csv')

# Predicting for the new data using the 5 clusters class
KM_5.predict(concat_new_cust)

# adding Segment or Group to the data as column:
new_customer_data['Segment'] = KM_5.predict(concat_new_cust)
```

```
new_customer_data
```