# Credit Card Customer Segmentation - Targeted Marketing Ad Campaign

By: Katrina James, Christian Urday, Justin Kerry

## Introductory Statement

Banks often use many different tactics to define their data and allow a deeper understanding of the data for realistic and accurate decision making. One of these tactics that many people are familiar with is called Customer Segmentation.

**Customer Segmentation**, also known by many as *Market Segmentation*, is the process of dividing a heterogeneous market into relatively more homogeneous segments based on certain parameters involving human behaviour and trends. This process is crucial for maximizing marketing campaign conversion rates. It is a process that is not only used in commerical business cases, but also a process that is used by banks as stated earlier. One of the ways that banks can utilize customer segmentation is when they are analyzing their credit card customers. They can segment their credit card customers into a few different groups to develop a better understanding regarding the traits and patterns of these credit card customers.

## Background

Through the process of customer segmentation, banks divide credit card customers into a few different groups:

- **Transactors**: customers who pay the least amount of interest charges and are more financially cautious.
- **Revolvers**: customers who use their credit card as a loan. This group is the most lucrative sector for banks since they usually pay 20%+ interest.
- **VIP/Prime**: customers with high credit limit and percentage of full payment. They can be targeted to increase their credit limit/spending.
- **New Customers**: customers with low tenure who can be targeted for alternative bank services enrollment.

## Objective

We will be using an unsupervised machine learning algorithm (K-Means) to segment the credit card customers featured in our dataset into different groups as clusters. We will then use Principal Components Analysis (PCA) for dimension reduction to provide us with a simplified overview of our data without significant loss of accuracy.

In order to provide the theoretical bank marketing team with a successful targeted marketing ad campaign that is effectively tailored to specific groups of customers, we will need to divide these customers into at least 3 distinctive groups. This means that we will need at least 3 clusters or components in our machine learning model.

## Dataset Analysis

We will be using a sample dataset for customer segmentation that can be found at the following weblink: https://www.kaggle.com/arjunbhasin2013/ccdata

Now we will take a look at the dataset as a dataframe table for initial analysis.

First we will import all of the libraries that we will need and then we can read our dataset as a viewable dataframe table.

In [1]:

```
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, normalize
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from jupyterthemes import jtplot
jtplot.style(theme='monokai',context='notebook',ticks=True, grid=False)
```

In [2]:

```
df=pd.read_csv('Bank_marketing.csv')
df
```

Out[2]:

| | CUST_ID | BALANCE | BALANCE_FREQUENCY | PURCHASES | ONEOFF_PURCHASES | INSTALLMENTS_PURCHASES | CASH_ADVANCE | PURCHA |
|---|---|---|---|---|---|---|---|---|
| **0** | C10001 | 40.900749 | 0.818182 | 95.40 | 0.00 | 95.40 | 0.000000 | |
| **1** | C10002 | 3202.467416 | 0.909091 | 0.00 | 0.00 | 0.00 | 6442.945483 | |
| **2** | C10003 | 2495.148862 | 1.000000 | 773.17 | 773.17 | 0.00 | 0.000000 | |
| **3** | C10004 | 1666.670542 | 0.636364 | 1499.00 | 1499.00 | 0.00 | 205.788017 | |
| **4** | C10005 | 817.714335 | 1.000000 | 16.00 | 16.00 | 0.00 | 0.000000 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | |
| **8945** | C19186 | 28.493517 | 1.000000 | 291.12 | 0.00 | 291.12 | 0.000000 | |
| **8946** | C19187 | 19.183215 | 1.000000 | 300.00 | 0.00 | 300.00 | 0.000000 | |
| **8947** | C19188 | 23.398673 | 0.833333 | 144.40 | 0.00 | 144.40 | 0.000000 | |
| **8948** | C19189 | 13.457564 | 0.833333 | 0.00 | 0.00 | 0.00 | 36.558778 | |
| **8949** | C19190 | 372.708075 | 0.666667 | 1093.25 | 1093.25 | 0.00 | 127.040008 | |

8950 rows × 18 columns

This sample dataset contains the usage behaviour of just under 9000 active credit card holders during the last 6 months. The data is at a customer level with 18 attributes for each object in the dataset.

Let us take a look at each attribute alongside an explanation in order to see what exactly each attribute tells us about each object:

- **CUST_ID**: identification of Credit Card holder *(categorical)*.
- **BALANCE**: balance amount left in their account to make purchases *(numerical)*.
- **BALANCE_FREQUENCY**: range of how often balance is updated *(1 = often, 0 = not often)* .
- **PURCHASES**: amount of purchases made from account *(numerical)*.
- **ONEOFF_PURCHASES**: maximum purchase amount done in a single purchase *(numerical)*.
- **INSTALLMENTS_PURCHASES**: amount of purchase done in an installment *(numerical)*.
- **CASH_ADVANCE**: cash advance given by the user *(numerical)*.
- **PURCHASES_FREQUENCY**: range of how frequently purchases are being made *(1 = often, 0 = not often)* .
- **ONEOFF_PURCHASES_FREQUENCY**: range of how often purchases are happening in a single installment *(1 = often, 0 = not often)* .
- **PURCHASES_INSTALLMENTS_FREQUENCY**: range of how often purchases in installments are occuring *(1 = often, 0 = not often)* .
- **CASH_ADVANCE_FREQUENCY**: range of how often the cash is being paid in advance *(1 = often, 0 = not often)* .
- **CASH_ADVANCE_TRX**: number of transactions made with "Cash in Advance" *(numerical)*.
- **PURCHASES_TRX**: number of purchase transactions made *(numerical)*.
- **CREDIT_LIMIT**: limit of credit for a customer's credit card *(numerical)*.
- **PAYMENTS**: amount of payments done by credit card user *(numerical)*.
- **MINIMUM_PAYMENTS**: minimum amount of payments made by the user *(numerical)*.
- **PRC_FULL_PAYMENT**: percentage of full payment paid by user *(numerical percentage)*.
- **TENURE**: tenure of credit card service for user in months *(numerical)*.

--

Now we can visualize our dataset and have a better understanding of the attributes for each object. We can proceed with data preprocessing.

## Data Preprocessing

Since we are beginning to preprocess our data, let us gather some information about the dataset such as *non-null count* and *data types*.

This will give us an idea on any missing data as well as which types of data we are going to be dealing with when building our model later on.

In [3]:

```
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
```

--

We can see that we have some missing data for *MINIMUM_PAYMENTS* and *CREDIT_LIMIT*.

This could've happened through human error or possibly a customer refusing to disclose that information.

Regardless, we will have to fill in these missing data values since they are crucial for our model to achieve a higher accuracy when segmentating each object.

--

First let us use the .describe() function in python to acquire some statistical insights on our dataframe.

In [4]:

```
df.describe()
```

Out[4]:

| | BALANCE | BALANCE_FREQUENCY | PURCHASES | ONEOFF_PURCHASES | INSTALLMENTS_PURCHASES | CASH_ADVANCE | PURCHASES_FI |
|---|---|---|---|---|---|---|---|
| **count** | 8950.000000 | 8950.000000 | 8950.000000 | 8950.000000 | 8950.000000 | 8950.000000 | 89 |
| **mean** | 1564.474828 | 0.877271 | 1003.204834 | 592.437371 | 411.067645 | 978.871112 | |
| **std** | 2081.531879 | 0.236904 | 2136.634782 | 1659.887917 | 904.338115 | 2097.163877 | |
| **min** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | |
| **25%** | 128.281915 | 0.888889 | 39.635000 | 0.000000 | 0.000000 | 0.000000 | |
| **50%** | 873.385231 | 1.000000 | 361.280000 | 38.000000 | 89.000000 | 0.000000 | |
| **75%** | 2054.140036 | 1.000000 | 1110.130000 | 577.405000 | 468.637500 | 1113.821139 | |
| **max** | 19043.138560 | 1.000000 | 49039.570000 | 40761.250000 | 22500.000000 | 47137.211760 | |

--

We now have an understanding of the important statistics regarding our data. We can see the min values and the max values. We can also see the mean which can be used to fill in the missing data. **We are assuming no data entry rather than zero values which is why we will be filling the missing data with the mean value of each column**.

We can also see the count for the two attributes that do not equal 8950, which is our missing data values. Let us address this missing data now.

### Missing Data

Here is a heatmap plot from our seaborn library that will show us our missing data values. We will use this plot twice. Once for confirming that the data values are missing. A second time for confirming that the missing data values have been filled.

In [5]:

```
sns.heatmap(df.isnull(),yticklabels=False, cbar=False, cmap="Blues")
```

Out[5]:

```
<AxesSubplot:>
```


We can see the missing data values of our dataset in the above plot.

--

Let us check to see exactly how many values are missing.

In [6]:

```
df.isnull().sum()
```

Out[6]:

```
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
```

--

We can see that there are 313 missing values for *MINIMUM_PAYMENTS*.

We can see that there is 1 missing value for *CREDIT_LIMIT*.

--

Let us use the mean of *MINIMUM_PAYMENTS* to fill in the 313 missing values for that attribute. **As mentioned before, we will be assuming no data entry rather than zero payments. This is why we will be filling the missing data values with the mean of the column as opposed to filling missing values with 0.**

In [7]:

```
df.loc[(df['MINIMUM_PAYMENTS'].isnull()==True),'MINIMUM_PAYMENTS']=df['MINIMUM_PAYMENTS'].mean()
```

Now let us check to see if the 313 missing values for *MINIMUM_PAYMENTS* have been filled with the mean of the attribute.

In [8]:

```
df.isnull().sum()
```

Out[8]:

```
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
```

--

The above table shows no more missing values for MINIMUM_PAYMENTS meaning that missing values have been successfully filled. We can do the same for the missing value of the *CREDIT_LIMIT* attribute.

In [9]:

```
df.loc[(df['CREDIT_LIMIT'].isnull()==True),'CREDIT_LIMIT']=df['CREDIT_LIMIT'].mean()
```

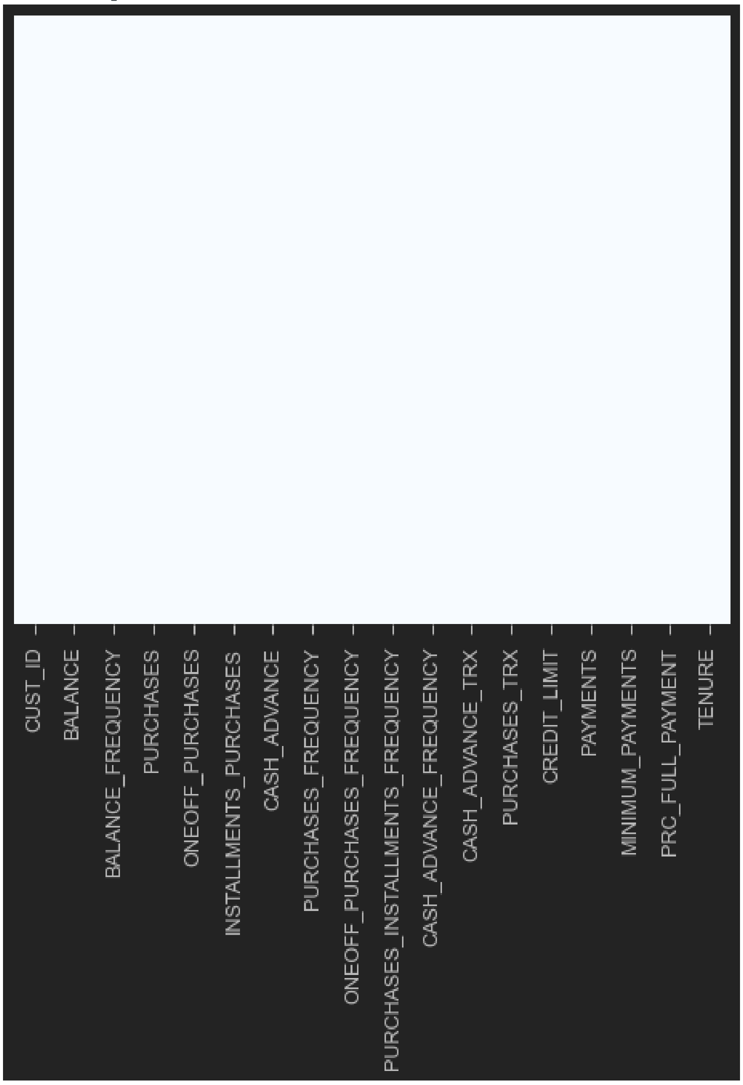
We have now filled the missing value in CREDIT_LIMIT with the mean of that column. Our second heatmap will show us that there are no more missing values in our dataset.

In [10]:

```
sns.heatmap(df.isnull(),yticklabels=False, cbar=False, cmap="Blues")
```

Out[10]:

```
<AxesSubplot:>
```




This can be proven by using a final sum for isnull.

In [11]:

```
df.isnull().sum()
```

Out[11]:

```
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
```

We can clearly see that there are no more missing data in the dataset.

### Checking for Duplicate Data

Let us use a sum function to check for duplicated data.

In [12]:

```
df.duplicated().sum()
```

Out[12]:

```
0
```

We can see that there are no duplicated data.

### Dropping Unneccessary Variables

After looking over each of the attributes in the dataset, we can see that there is one attribute that is unneccessary and may actually contribute to some noise in our model.

This attribute is customer ID (CUST_ID). Let us drop that attribute so we can have a cleaner dataset.

In [13]:

```
df.drop('CUST_ID',axis=1,inplace=True)
df.head()
```

Out[13]:

| | BALANCE | BALANCE_FREQUENCY | PURCHASES | ONEOFF_PURCHASES | INSTALLMENTS_PURCHASES | CASH_ADVANCE | PURCHASES_FREQUEN |
|---|---|---|---|---|---|---|---|
| **0** | 40.900749 | 0.818182 | 95.40 | 0.00 | 95.4 | 0.000000 | 0.166 |
| **1** | 3202.467416 | 0.909091 | 0.00 | 0.00 | 0.0 | 6442.945483 | 0.000 |
| **2** | 2495.148862 | 1.000000 | 773.17 | 773.17 | 0.0 | 0.000000 | 1.000 |
| **3** | 1666.670542 | 0.636364 | 1499.00 | 1499.00 | 0.0 | 205.788017 | 0.083 |
| **4** | 817.714335 | 1.000000 | 16.00 | 16.00 | 0.0 | 0.000000 | 0.083 |

Perfect. The CUST_ID column that was not providing us with any valuable information is now eliminated.

## Feature Plotting

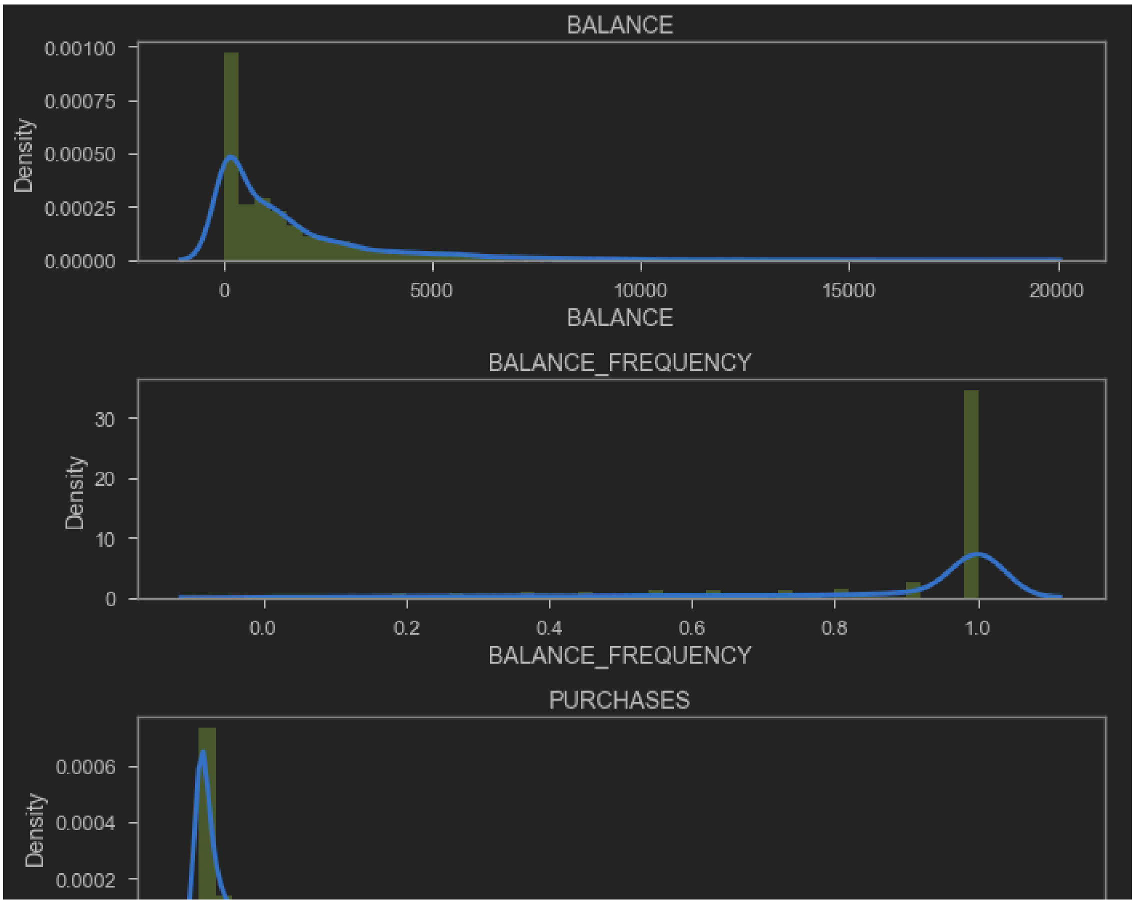
Let us plot each feature to visualize the probability density of each variable being continuous.

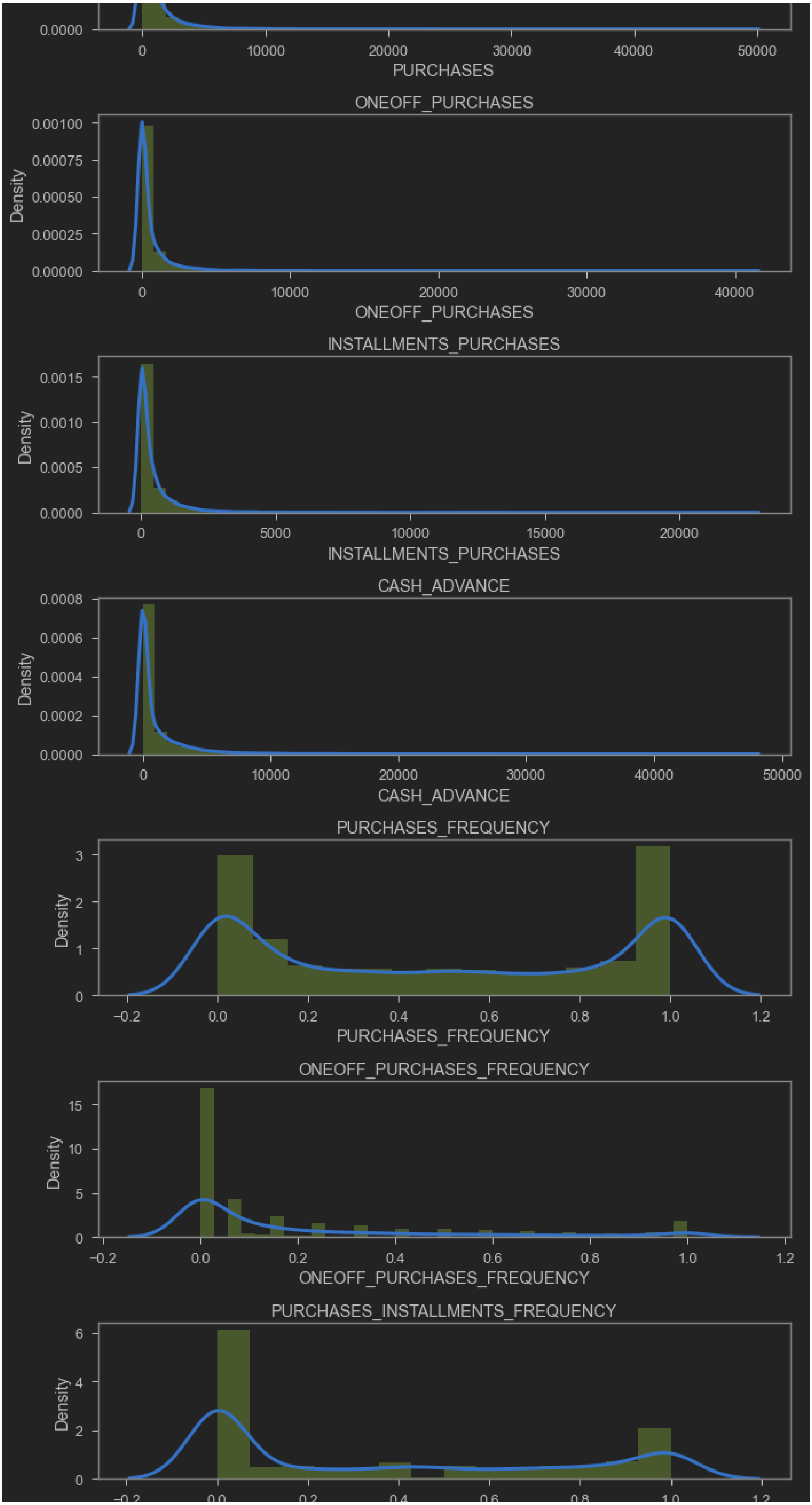
*Sidenote: future error messages have been displayed, but they will not interfere with our data or our feature plots so they can be safely disregarded. We have used an ignore parameter for warnings in our library imports so the warning messages won't show allowing for a cleaner report.*

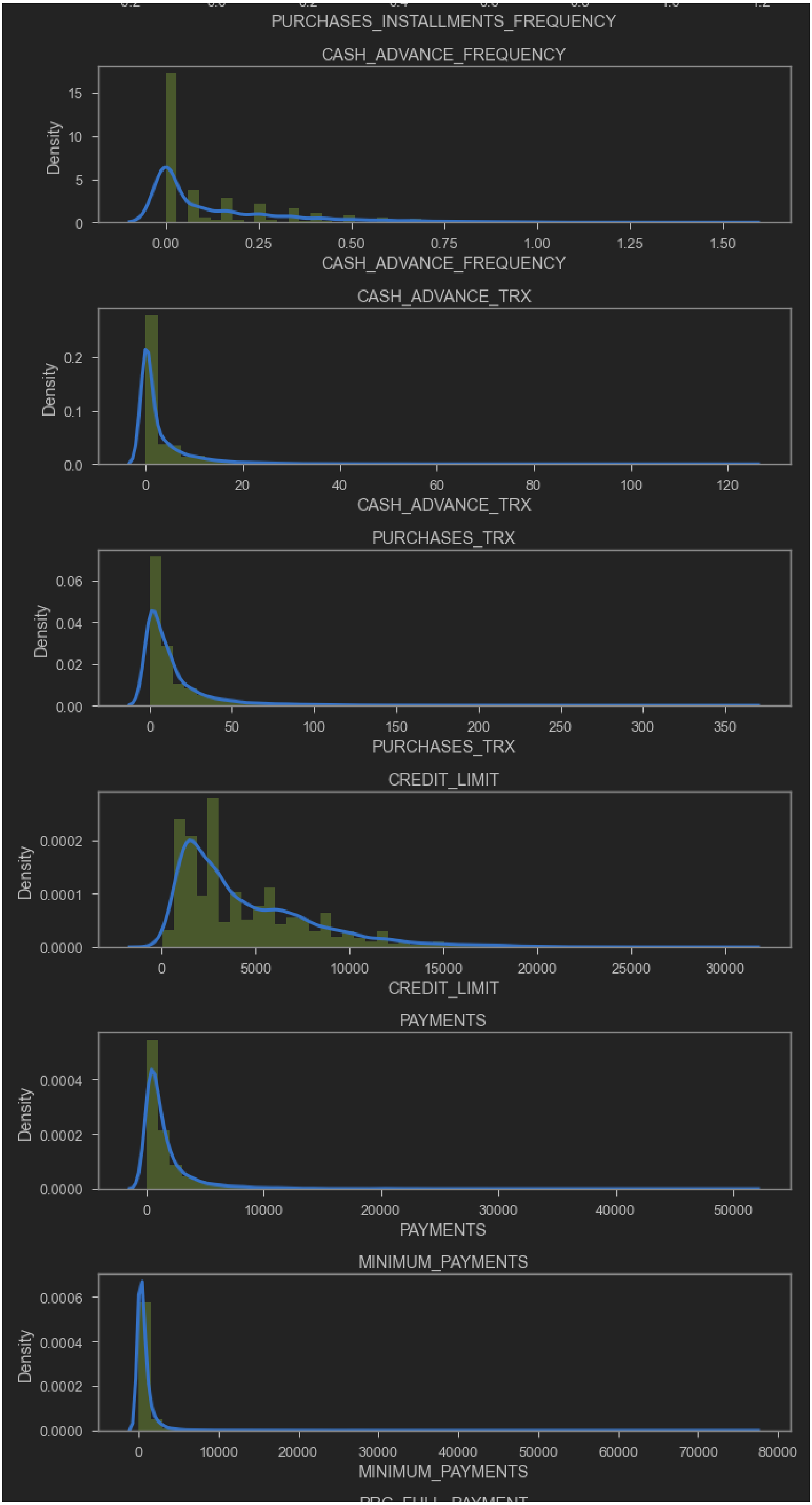
In [14]:

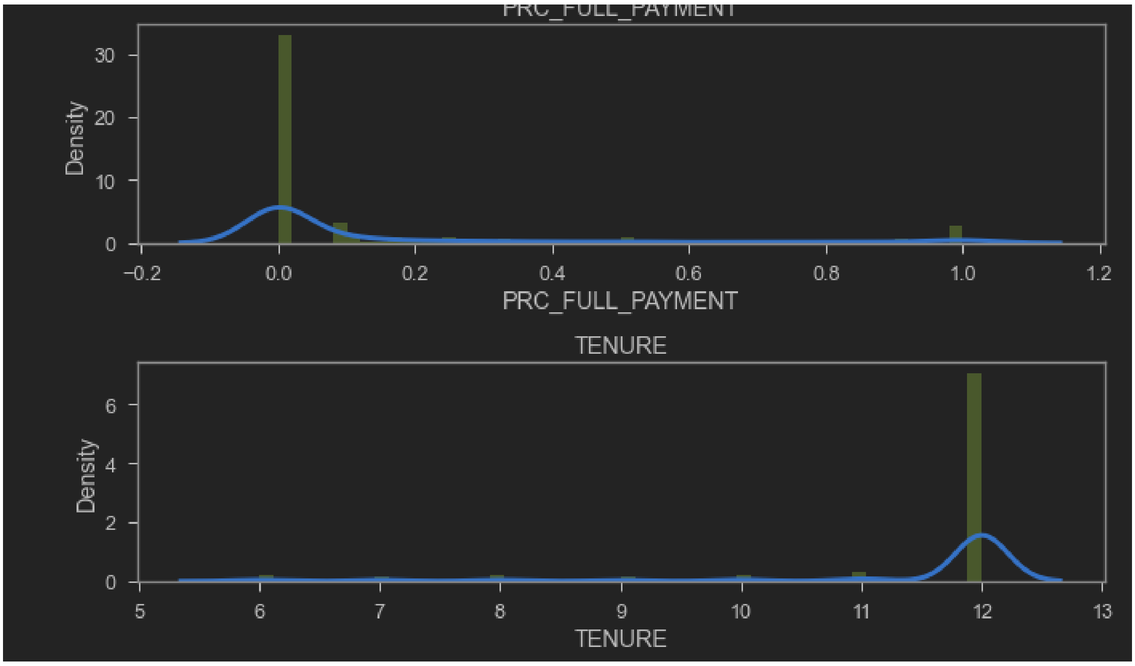
```
plt.figure(figsize=(10,50))
for i in range(len(df.columns)):
    plt.subplot(17, 1, i+1)
    sns.distplot(df[df.columns[i]], kde_kws={"color":"b","lw":3,"label":"KDE"}, hist_kws={"color":"g"})
    plt.title(df.columns[i])

plt.tight_layout()
```




PURCHASES
ONEOFF_PURCHASES
Density
ONEOFF_PURCHASES
INSTALLMENTS_PURCHASES
Density
INSTALLMENTS_PURCHASES
CASH_ADVANCE
Density
CASH_ADVANCE
PURCHASES_FREQUENCY
Density
PURCHASES_FREQUENCY
ONEOFF_PURCHASES_FREQUENCY
Density
ONEOFF_PURCHASES_FREQUENCY
PURCHASES_INSTALLMENTS_FREQUENCY
Density


PURCHASES_INSTALLMENTS_FREQUENCY
CASH_ADVANCE_FREQUENCY
Density
15
10
5
0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
CASH_ADVANCE_FREQUENCY
CASH_ADVANCE_TRX
Density
0.2
0.1
0.0
0
20
40
60
80
100
120
CASH_ADVANCE_TRX
PURCHASES_TRX
Density
0.06
0.04
0.02
0.00
0
50
100
150
200
250
300
350
PURCHASES_TRX
CREDIT_LIMIT
Density
0.0002
0.0001
0.0000
0
5000
10000
15000
20000
25000
30000
CREDIT_LIMIT
PAYMENTS
Density
0.0004
0.0002
0.0000
0
10000
20000
30000
40000
50000
PAYMENTS
MINIMUM_PAYMENTS
Density
0.0006
0.0004
0.0002
0.0000
0
10000
20000
30000
40000
50000
60000
70000
80000
MINIMUM_PAYMENTS




--

We can also use a Correlation Matrix to visualize the level of correlation between the features. This matrix will also display a level bar on the right side to show how correlated each feature is with each other.

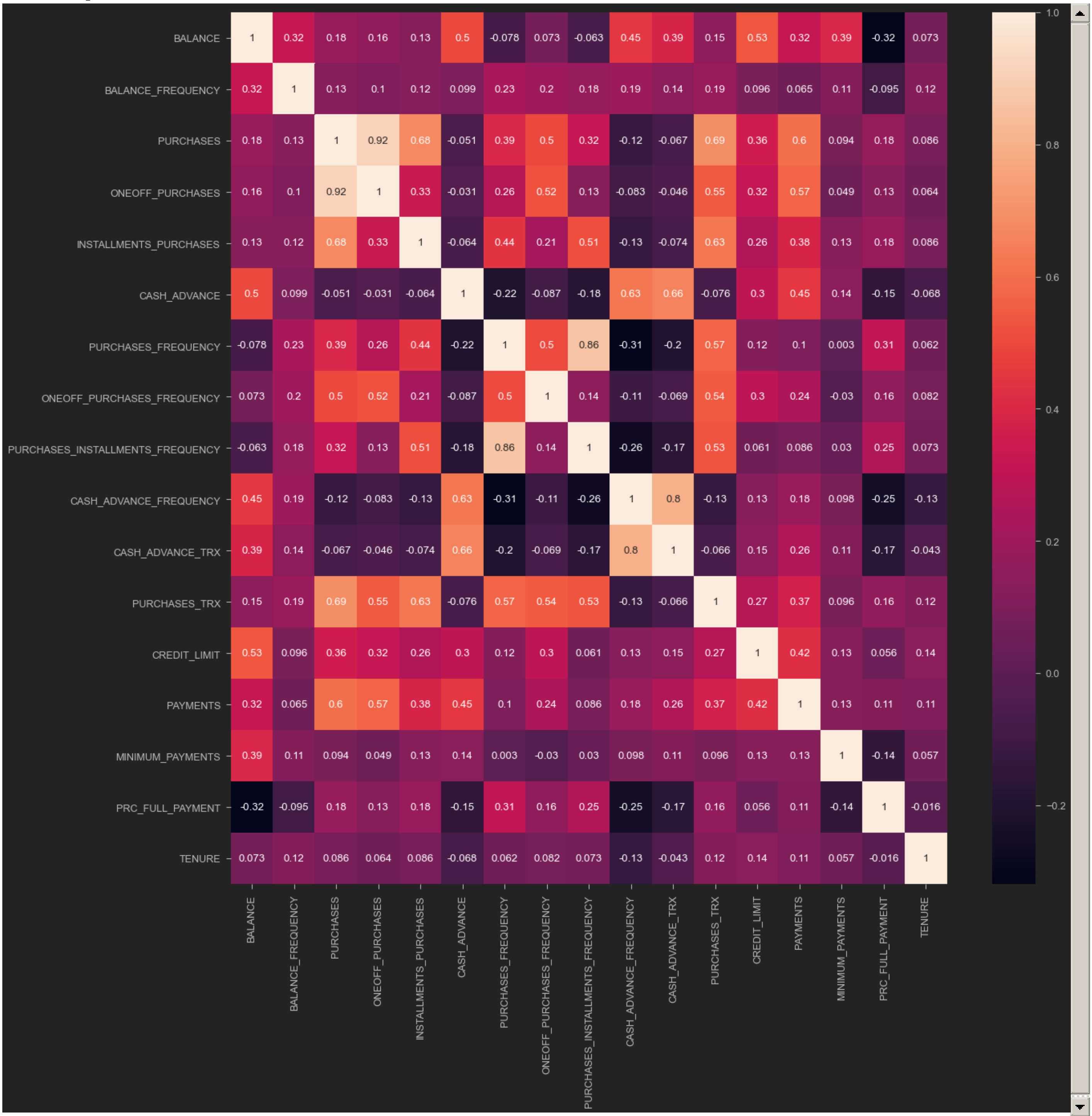
*Sidenote: 1 can be considered the most correlated and -0.32 can be considered the least correlated .*

In [15]:

```
correlation=df.corr()
f, ax=plt.subplots(figsize=(20,20))
sns.heatmap(correlation, annot=True)
```

Out[15]:

```
<AxesSubplot:>
```



--

We are almost ready to use the K-Means Clustering algorithm. First we must scale and transform the data appropriately. We must also determine the optimal amount of clusters for the K-Means algorithm.

## Scaling the Data

We can scale our data using the Standard Scaler in python.

In [16]:

```
scaler=StandardScaler()
df_scaled=scaler.fit_transform(df)

df_scaled
```

Out[16]:

```
array([[-0.73198937, -0.24943448, -0.42489974, ..., -0.31096755,
        -0.52555097,  0.36067954],
       [ 0.78696085,  0.13432467, -0.46955188, ...,  0.08931021,
         0.2342269 ,  0.36067954],
       [ 0.44713513,  0.51808382, -0.10766823, ..., -0.10166318,
        -0.52555097,  0.36067954],
       ...,
       [-0.7403981 , -0.18547673, -0.40196519, ..., -0.33546549,
         0.32919999, -4.12276757],
       [-0.74517423, -0.18547673, -0.46955188, ..., -0.34690648,
         0.32919999, -4.12276757],
       [-0.57257511, -0.88903307,  0.04214581, ..., -0.33294642,
        -0.52555097, -4.12276757]])
```

--

Let us confirm the correct number of objects and attributes in our dataset before we continue.

In [17]:

```
df_scaled.shape
```

Out[17]:

```
(8950, 17)
```

## Determining Optimal Number of Clusters

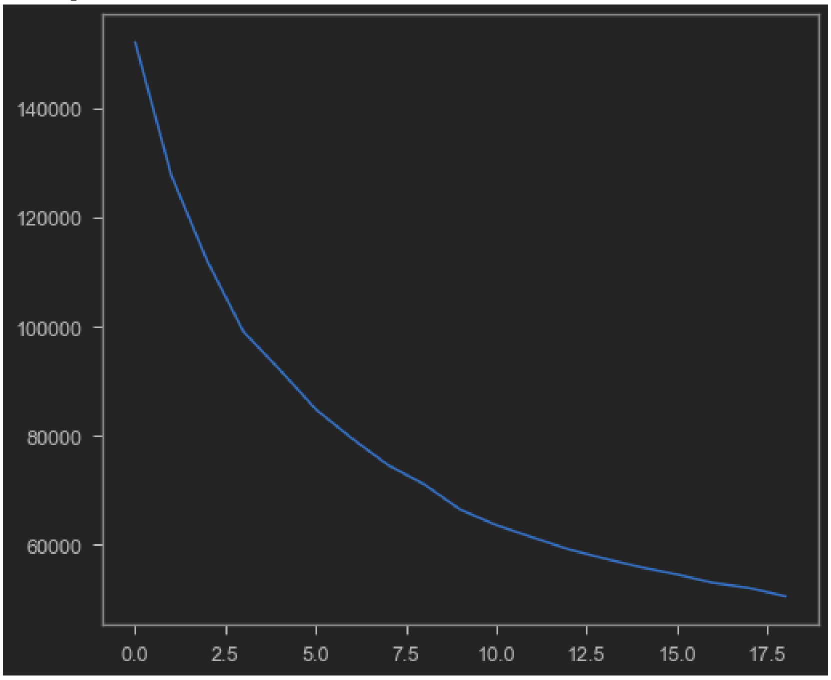
In order to determine the optimal number of clusters for our K-Means algorithm, we will use the elbow method.

In [18]:

```
scores=[]
range_val=range(1,20)
for i in range_val:
    kmeans=KMeans(n_clusters=i)
    kmeans.fit(df_scaled)
    scores.append(kmeans.inertia_)
plt.plot(scores,'bx-')
```

Out[18]:

```
[<matplotlib.lines.Line2D at 0x111a30c70>]
```




Based on this plot, we can see that the Elbow forms around the 3rd or 4th cluster.

Since the elbow is non-definitive and the linearity of the graph does not present itself until the 7th or 8th cluster, we will choose 7 clusters for our K-Means algorithm. This will provide our model with more than enough segmentation for the customer business case.

## Application of K-Means Method

Now we can start applying the K-Means method as a process for our model construction using the selected parameters.

In [19]:

```
kmeans=KMeans(7)
kmeans.fit(df_scaled)
labels=kmeans.labels_
```

```
kmeans.cluster_centers_.shape
```

Out[19]:

```
(7, 17)
```

--

We have now applied K-Means to our scaled dataset.

Let us now view the scaled dataset as a dataframe post K-Means application.

*Sidenote: We will not actually see the clusters listed in our dataframe until after we concatenate the dataset with the clusters labels. We will do that very shortly.*

In [20]:

```
cluster_centers=pd.DataFrame(data=kmeans.cluster_centers_,columns=[df.columns])
cluster_centers
```

Out[20]:

| | BALANCE | BALANCE_FREQUENCY | PURCHASES | ONEOFF_PURCHASES | INSTALLMENTS_PURCHASES | CASH_ADVANCE | PURCHASES_FREQUENCY |
|---|---|---|---|---|---|---|---|
| **0** | -0.335506 | -0.348076 | -0.284525 | -0.208973 | -0.288475 | 0.065539 | -0.198735 |
| **1** | 0.008364 | 0.402565 | -0.344339 | -0.225641 | -0.399534 | -0.103008 | -0.811133 |
| **2** | 0.126924 | 0.430008 | 0.936481 | 0.893318 | 0.573104 | -0.308187 | 1.092347 |
| **3** | -0.368610 | 0.330287 | -0.039975 | -0.235223 | 0.337450 | -0.368847 | 0.980184 |
| **4** | -0.701872 | -2.134325 | -0.306924 | -0.230292 | -0.302515 | -0.323078 | -0.547138 |
| **5** | 1.430238 | 0.419467 | 6.915048 | 6.083034 | 5.172266 | 0.038778 | 1.090699 |
| **6** | 1.672609 | 0.393258 | -0.204140 | -0.148982 | -0.209063 | 1.996728 | -0.453351 |

--

Before we concatenate the clusters labels to the dataframe, we need to apply an inverse transformation to understand the differentiation between the numerical values.

### Transformations

Let us apply the inverse transformation to the dataframe now.

In [21]:

```
cluster_centers=scaler.inverse_transform(cluster_centers)
cluster_centers=pd.DataFrame(data=cluster_centers,columns=[df.columns])
cluster_centers
```

Out[21]:

| | BALANCE | BALANCE_FREQUENCY | PURCHASES | ONEOFF_PURCHASES | INSTALLMENTS_PURCHASES | CASH_ADVANCE | PURCHASES_FREQUE |
|---|---|---|---|---|---|---|---|
| **0** | 866.148306 | 0.794815 | 395.311749 | 245.585564 | 150.203132 | 1116.308792 | 0.41 |
| **1** | 1581.883580 | 0.972635 | 267.520239 | 217.920025 | 49.774223 | 762.859357 | 0.16 |
| **2** | 1828.656208 | 0.979136 | 3004.010071 | 2075.162956 | 929.318813 | 332.589357 | 0.92 |
| **3** | 797.244565 | 0.955513 | 917.797767 | 202.015524 | 716.219696 | 205.381298 | 0.88 |
| **4** | 103.587241 | 0.371669 | 347.456361 | 210.199629 | 137.506773 | 301.361116 | 0.27 |
| **5** | 4541.393882 | 0.976638 | 15777.311395 | 10689.027791 | 5088.283605 | 1060.190695 | 0.92 |
| **6** | 5045.869096 | 0.970430 | 567.057618 | 345.157607 | 222.014247 | 5166.103258 | 0.30 |

--

Now we are ready to concatenate the clusters labels with our original dataframe.

In [22]:

```
df_cluster=pd.concat([df,pd.DataFrame({'CLUSTER':labels})], axis=1)
df_cluster.head(50)
```

Out[22]:

| | BALANCE | BALANCE_FREQUENCY | PURCHASES | ONEOFF_PURCHASES | INSTALLMENTS_PURCHASES | CASH_ADVANCE | PURCHASES_FREQU |
|---|---|---|---|---|---|---|---|
| **0** | 40.900749 | 0.818182 | 95.40 | 0.00 | 95.40 | 0.000000 | 0.1 |

| | BALANCE | BALANCE_FREQUENCY | PURCHASES | ONEOFF_PURCHASES | INSTALLMENTS_PURCHASES | CASH_ADVANCE | PURCHASES_FREQU |
|---|---|---|---|---|---|---|---|
| 1 | 3202.467416 | 0.909091 | 0.00 | 0.00 | 0.00 | 6442.945483 | 0.00 |
| 2 | 2495.148862 | 1.000000 | 773.17 | 773.17 | 0.00 | 0.000000 | 1.00 |
| 3 | 1666.670542 | 0.636364 | 1499.00 | 1499.00 | 0.00 | 205.788017 | 0.08 |
| 4 | 817.714335 | 1.000000 | 16.00 | 16.00 | 0.00 | 0.000000 | 0.08 |
| 5 | 1809.828751 | 1.000000 | 1333.28 | 0.00 | 1333.28 | 0.000000 | 0.66 |
| 6 | 627.260806 | 1.000000 | 7091.01 | 6402.63 | 688.38 | 0.000000 | 1.00 |
| 7 | 1823.652743 | 1.000000 | 436.20 | 0.00 | 436.20 | 0.000000 | 1.00 |
| 8 | 1014.926473 | 1.000000 | 861.49 | 661.49 | 200.00 | 0.000000 | 0.33 |
| 9 | 152.225975 | 0.545455 | 1281.60 | 1281.60 | 0.00 | 0.000000 | 0.16 |
| 10 | 1293.124939 | 1.000000 | 920.12 | 0.00 | 920.12 | 0.000000 | 1.00 |
| 11 | 630.794744 | 0.818182 | 1492.18 | 1492.18 | 0.00 | 0.000000 | 0.25 |
| 12 | 1516.928620 | 1.000000 | 3217.99 | 2500.23 | 717.76 | 0.000000 | 1.00 |
| 13 | 921.693369 | 1.000000 | 2137.93 | 419.96 | 1717.97 | 0.000000 | 0.75 |
| 14 | 2772.772734 | 1.000000 | 0.00 | 0.00 | 0.00 | 346.811390 | 0.00 |
| 15 | 6886.213231 | 1.000000 | 1611.70 | 0.00 | 1611.70 | 2301.491267 | 0.50 |
| 16 | 2072.074354 | 0.875000 | 0.00 | 0.00 | 0.00 | 2784.274703 | 0.00 |
| 17 | 41.089489 | 0.454545 | 519.00 | 0.00 | 519.00 | 0.000000 | 0.41 |
| 18 | 1989.072228 | 1.000000 | 504.35 | 166.00 | 338.35 | 0.000000 | 0.66 |
| 19 | 3577.970933 | 1.000000 | 398.64 | 0.00 | 398.64 | 0.000000 | 1.00 |
| 20 | 2016.684686 | 1.000000 | 176.68 | 0.00 | 176.68 | 0.000000 | 0.66 |
| 21 | 6369.531318 | 1.000000 | 6359.95 | 5910.04 | 449.91 | 229.028245 | 1.00 |
| 22 | 132.342240 | 0.636364 | 815.90 | 0.00 | 815.90 | 0.000000 | 1.00 |
| 23 | 3800.151377 | 0.818182 | 4248.35 | 3454.56 | 793.79 | 7974.415626 | 1.00 |
| 24 | 5368.571219 | 1.000000 | 0.00 | 0.00 | 0.00 | 798.949863 | 0.00 |
| 25 | 169.781679 | 1.000000 | 399.60 | 0.00 | 399.60 | 0.000000 | 1.00 |
| 26 | 1615.967240 | 1.000000 | 102.00 | 102.00 | 0.00 | 244.840485 | 0.16 |
| 27 | 125.694817 | 1.000000 | 233.28 | 0.00 | 233.28 | 0.000000 | 1.00 |
| 28 | 7152.864372 | 1.000000 | 387.05 | 204.55 | 182.50 | 2236.145259 | 0.66 |
| 29 | 22.063490 | 1.000000 | 100.00 | 0.00 | 100.00 | 0.000000 | 0.41 |
| 30 | 12136.219960 | 1.000000 | 3038.01 | 1013.20 | 2024.81 | 3183.583301 | 1.00 |
| 31 | 1162.273324 | 1.000000 | 1347.71 | 400.00 | 947.71 | 175.815755 | 1.00 |
| 32 | 6732.823064 | 1.000000 | 324.95 | 324.95 | 0.00 | 1189.533753 | 0.08 |
| 33 | 125.660453 | 1.000000 | 636.79 | 636.79 | 0.00 | 0.000000 | 0.91 |
| 34 | 3517.101616 | 0.727273 | 547.28 | 0.00 | 547.28 | 0.000000 | 1.00 |
| 35 | 1656.350781 | 1.000000 | 0.00 | 0.00 | 0.00 | 99.264367 | 0.00 |
| 36 | 7427.076941 | 1.000000 | 0.00 | 0.00 | 0.00 | 8873.375046 | 0.00 |
| 37 | 4047.480828 | 1.000000 | 2380.55 | 1642.17 | 738.38 | 1697.660901 | 0.75 |
| 38 | 6269.418144 | 1.000000 | 204.00 | 204.00 | 0.00 | 2925.699862 | 0.08 |
| 39 | 1411.602230 | 0.454545 | 963.24 | 963.24 | 0.00 | 6173.682877 | 0.08 |
| 40 | 663.447810 | 0.727273 | 0.00 | 0.00 | 0.00 | 3133.063934 | 0.00 |
| 41 | 784.889762 | 0.909091 | 1526.59 | 786.30 | 740.29 | 2188.419607 | 0.58 |
| 42 | 4104.710798 | 1.000000 | 203.82 | 203.82 | 0.00 | 0.000000 | 0.08 |
| 43 | 1360.742377 | 1.000000 | 0.00 | 0.00 | 0.00 | 1481.587093 | 0.00 |
| 44 | 5315.945594 | 1.000000 | 1525.44 | 469.80 | 1055.64 | 0.000000 | 1.00 |
| 45 | 2242.311686 | 1.000000 | 437.00 | 97.00 | 340.00 | 184.648692 | 0.33 |
| 46 | 474.447149 | 0.500000 | 109.74 | 0.00 | 109.74 | 1013.659552 | 0.37 |
| 47 | 3910.111237 | 1.000000 | 0.00 | 0.00 | 0.00 | 1980.873201 | 0.00 |

| | BALANCE | BALANCE_FREQUENCY | PURCHASES | ONEOFF_PURCHASES | INSTALLMENTS_PURCHASES | CASH_ADVANCE | PURCHASES_FREQU |
|---|---|---|---|---|---|---|---|
| **48** | 22[illegible] | 1.000000 | [illegible] | [illegible] | [illegible] | 0.000000 | 1.0[illegible] |
| **49** | 863.809833 | 1.000000 | 507.50 | 507.50 | 0.00 | 695.097159 | 0.1[illegible] |

--

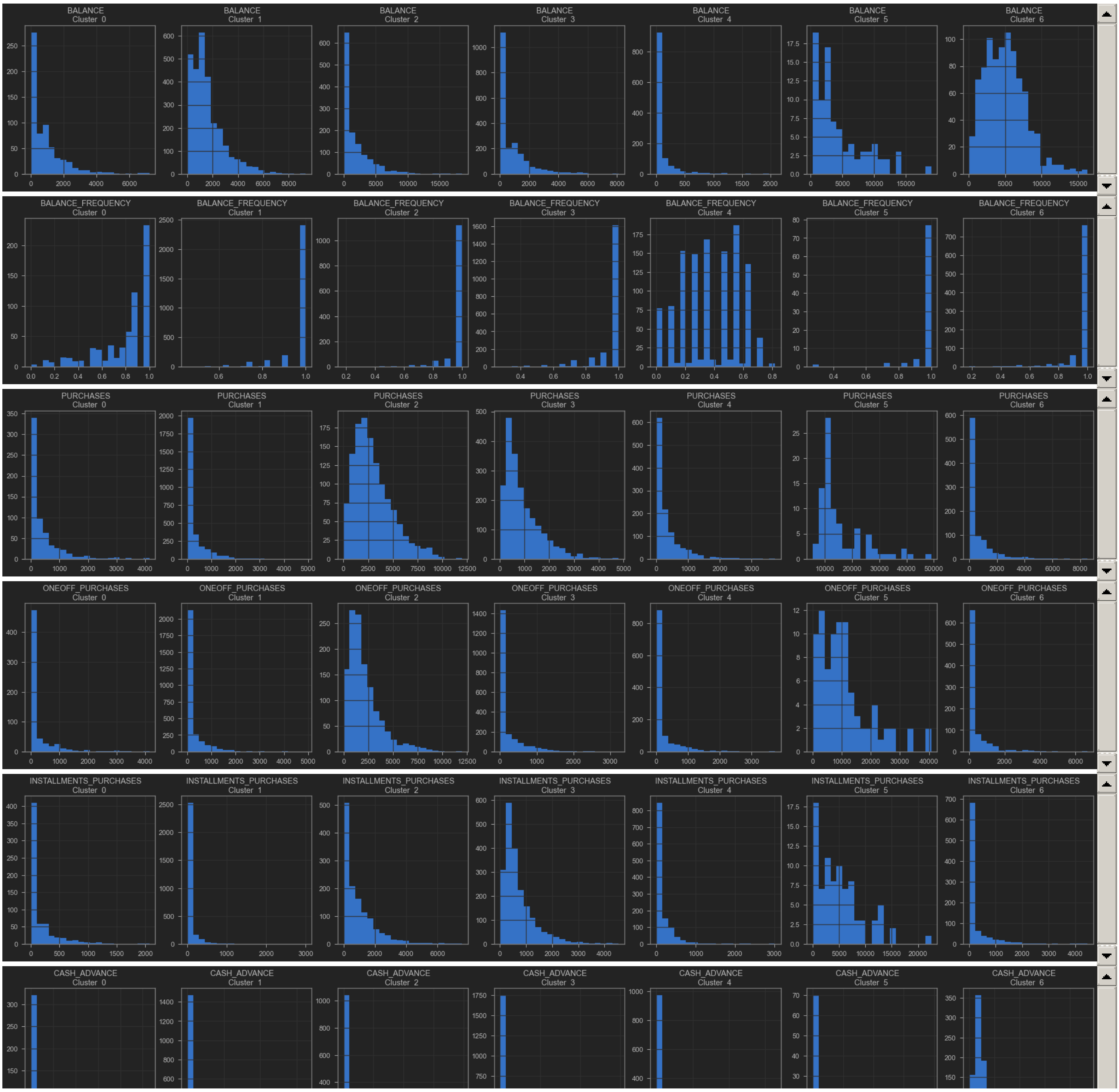
Now that clusters labels have been concatenated with original dataframe, we can apply Principal Components Analysis to reduce our dimensions and allow our unsupervised machine learning model to transition to a more simplified overview without a significant loss in accuracy.

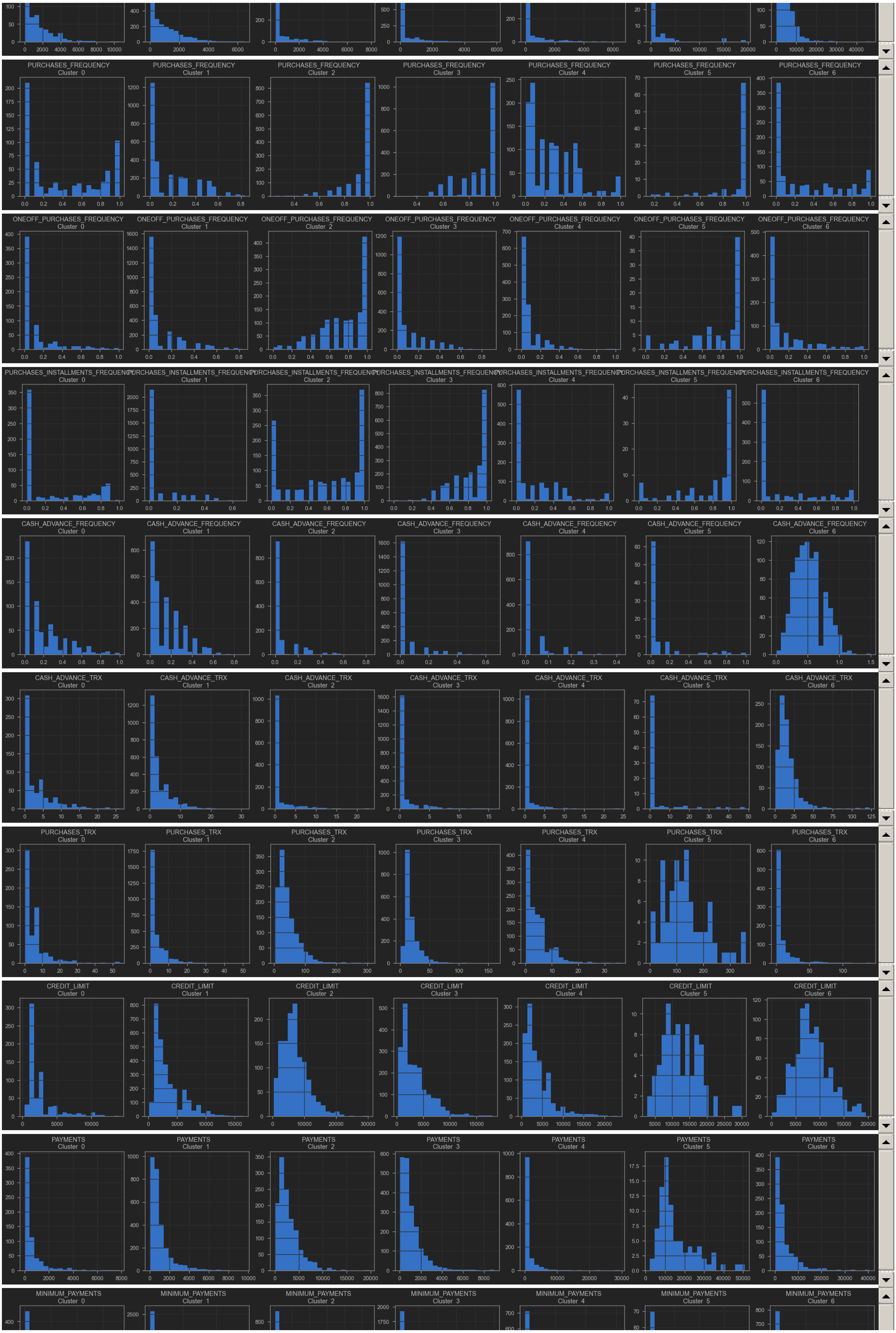
Right before we do that we will plot the clusters as a histogram matrix to visualize each of the clusters against each of the attributes featured in our dataframe.

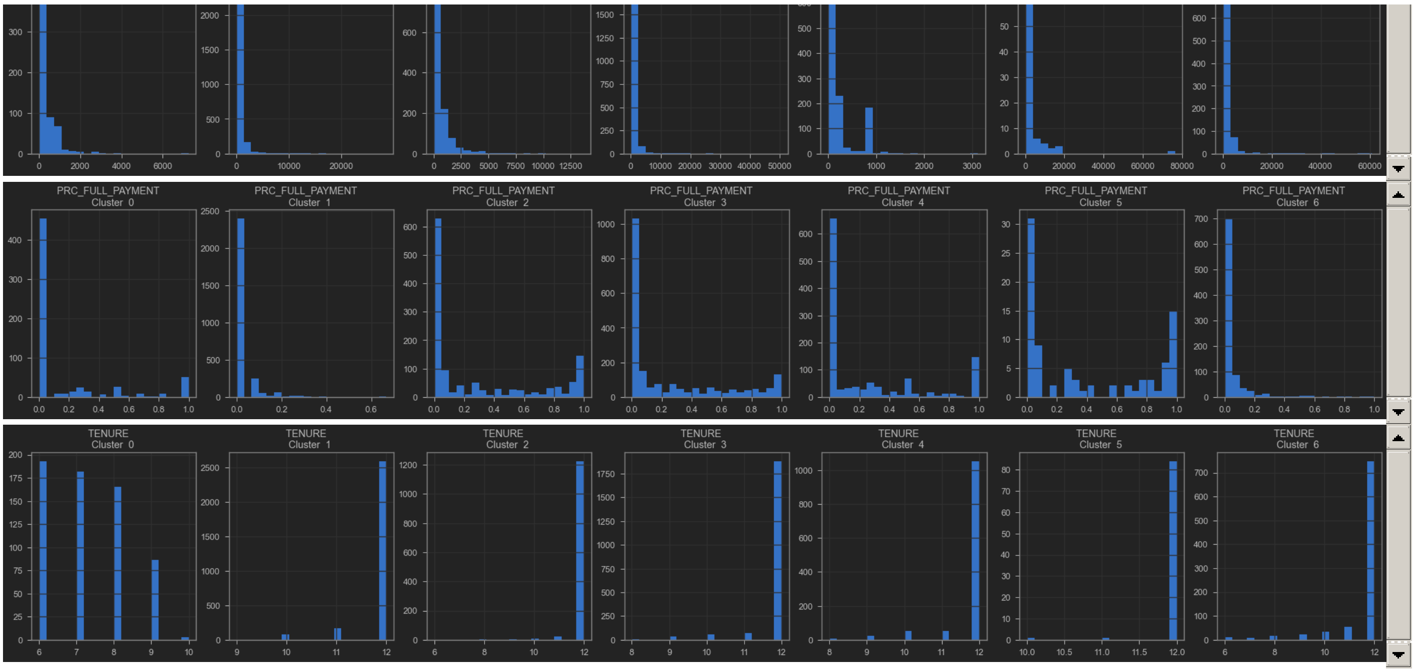
*Sidenote: python will list these clusters as 0-6 which is the same as listing them as clusters 1-7 since we do have 7 clusters derived from K-Means.*

*In [23]:*

```
for i in df.columns:
    plt.figure(figsize=(35,5))
    for j in range(7):
        plt.subplot(1,7,j+1)
        cluster=df_cluster[df_cluster['CLUSTER']==j]
        cluster[i].hist(bins=20)
        plt.title('{}    \nCluster  {} '.format(i,j))
    plt.show()
```




PURCHASES_FREQUENCY
Cluster 0
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cluster 6
ONEOFF_PURCHASES_FREQUENCY
PURCHASES_INSTALLMENTS_FREQUENCY
CASH_ADVANCE_FREQUENCY
CASH_ADVANCE_TRX
PURCHASES_TRX
CREDIT_LIMIT
PAYMENTS
MINIMUM_PAYMENTS




--

*Now that we have achieved a clear visualization of the relationship between our clusters, we can begin the application of Principal Components Analysis to our model.*

## Application of Principal Components Analysis

*Before we began this report, we had ran some experiments to determine the total variance between the principal components.*

*We discovered that a minimum of 6 principal components was required to achieve the minimum recommended 75% variance in our model.*

*Let us apply PCA using the 6 required components. We will show the variance contributions between the components very soon in this report.*

In [24]:

```
pca=PCA(n_components=6)
pca.fit(df_scaled)
```

Out[24]:

```
PCA(n_components=6)
```

In [25]:

```
scores=pca.transform(df_scaled)

scores_df=pd.DataFrame(scores,columns=['PC1','PC2','PC3','PC4','PC5','PC6'])
scores_df
```

Out[25]:

| | PC1 | PC2 | PC3 | PC4 | PC5 | PC6 |
|---|---|---|---|---|---|---|
| **0** | -1.682220 | -1.076451 | 0.488507 | 0.665552 | 0.018225 | 0.050629 |
| **1** | -1.138295 | 2.506477 | 0.601212 | -0.120437 | 0.605803 | -1.136841 |
| **2** | 0.969684 | -0.383520 | 0.102371 | 1.209266 | -2.172584 | -0.217222 |
| **3** | -0.873628 | 0.043166 | 1.460167 | 1.151980 | 0.295632 | -0.123689 |
| **4** | -1.599434 | -0.688581 | 0.365094 | 0.990232 | -0.487039 | 0.075060 |
| **...** | ... | ... | ... | ... | ... | ... |
| **8945** | -0.359629 | -2.016145 | -0.995355 | -2.727433 | 0.268860 | 2.673305 |
| **8946** | -0.564369 | -1.639123 | -1.290238 | -1.860551 | 0.187104 | 3.384215 |
| **8947** | -0.926204 | -1.810786 | -0.474723 | -2.280239 | 0.386553 | 2.976948 |
| **8948** | -2.336552 | -0.657966 | 0.974725 | -1.861279 | 0.069779 | 3.174380 |
| **8949** | -0.556422 | -0.400467 | 1.015196 | -1.953237 | -1.243000 | 3.753291 |

*8950 rows × 6 columns*

*In the above dataframe table, we have shown the scores provided for each of the 6 clusters in relation to every object in our dataset.*

*Sidenote: As previously mentioned before, we have a grand total of 8950 objects in the dataset. The table reaches 8949 but with the automatic inclusion of object 0, all of the objects are truly present in this table.*

### Explaining the Variance

*Let us now display the variance between each of the 6 PCA components. This will fully show the variance contributions and provide an explanation as to why we have selected 6 for the number of principal components included in this algorithm of our model.*

In [26]:

```
explained_variance=pca.explained_variance_ratio_
explained_variance
```

Out[26]:

```
array([0.27290037, 0.20301991, 0.08791979, 0.07479975, 0.06262792,
       0.05750211])
```

In [27]:

```
explained_variance=np.insert(explained_variance,0,0)
```

*Inside of the array, we can see the variance shown for each component. They are shown in order from the 1st to the 6th component.*

*Now let us show the cumulative variance values to get a better insight on how they stack on each other. This will also prove that the 6 components will give us our 75% variance target.*

In [28]:

```
cumulative_variance=np.cumsum(np.round(explained_variance,decimals=2))
```

*We have just prepared the cumulative variance values.*

*Now let us combine these values with our Dataframe.*

In [29]:

```
pc_df=pd.DataFrame(['','PC1','PC2','PC3','PC4','PC5','PC6'], columns=['PC'])
explained_variance_df=pd.DataFrame(explained_variance, columns=['Explained Variance'])
cumulative_variance_df=pd.DataFrame(cumulative_variance, columns=['Cumulative Varaince'])
```

*Now that the values have been combined with our Dataframe, we can display these cumulative variance values as a table. This is the part where we can see how the first 6 principal components can explain at least 75% of the variance in our model after dimension reduction.*

In [30]:

```
df_explained_variance=pd.concat([pc_df,explained_variance_df,cumulative_variance_df], axis=1)
df_explained_variance
```

Out[30]:

| | PC | Explained Variance | Cumulative Varaince |
|---|---|---|---|
| **0** | | 0.000000 | 0.00 |
| **1** | PC1 | 0.272900 | 0.27 |
| **2** | PC2 | 0.203020 | 0.47 |
| **3** | PC3 | 0.087920 | 0.56 |
| **4** | PC4 | 0.074800 | 0.63 |
| **5** | PC5 | 0.062628 | 0.69 |
| **6** | PC6 | 0.057502 | 0.75 |

*We can see how the first 2 components are highly significant explaining a combined amount of approximately 47% of the variance.*

*The 3rd component is not nearly as significant as it explains approximately 8% of the variance. The component is still necessary though since it is only when we reach the 6th component that we reach a combined total of 75% of the variance explained.*

--

*Next let us utilize a scree plot to visualize the results of the explained variance between the principal components.*

### Utilizing a Scree Plot

*First we will import the required plotly library in python.*

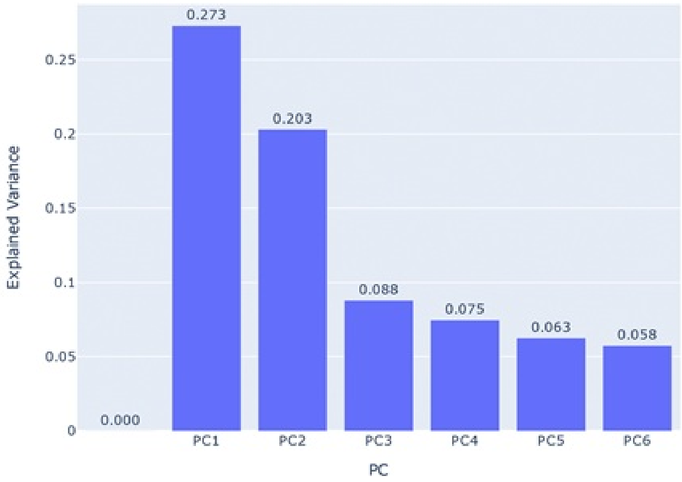
In [31]:

```
import plotly.express as px
```

*Now we can generate the scree plot.*

In [ ]:

```
fig=px.bar(df_explained_variance,x='PC', y='Explained Variance', text='Explained Variance', width=800)
fig.update_traces(texttemplate='%{text:.3f}', textposition='outside')
fig.show()
```




## Visualization of Major Components

*Since Principal Component 1 and 2 are major components for explained variance, we have decided to show them both in a plane. This will allow us to plot them in a scatterplot, which will be an effective way to visualize the two major components which have a combined explained variance of approximately 47%.*

### Visualizing Principal Components in a Plane

In [33]:

```
pca_df=pd.DataFrame(data=scores_df,columns=['PC1','PC2'])
pca_df.head()
```

Out[33]:

| | PC1 | PC2 |
|---|---|---|
| 0 | -1.682220 | -1.076451 |
| 1 | -1.138295 | 2.506477 |
| 2 | 0.969684 | -0.383520 |
| 3 | -0.873628 | 0.043166 |
| 4 | -1.599434 | -0.688581 |

*We have now determined the scores for the first 2 components and are now able to move on with the creation of the scatter plot.*

*First we will have to concatenate these 2 principal components with the clusters created from K-Means against the objects in our dataset.*

In [34]:

```
pca_df=pd.concat([pca_df,pd.DataFrame({'CLUSTER':labels})],axis=1)
pca_df.head()
```

Out[34]:

| | PC1 | PC2 | CLUSTER |
|---|---|---|---|
| 0 | -1.682220 | -1.076451 | 1 |
| 1 | -1.138295 | 2.506477 | 6 |
| 2 | 0.969684 | -0.383520 | 2 |
| 3 | -0.873628 | 0.043166 | 1 |
| 4 | -1.599434 | -0.688581 | 1 |

*Now that we have achieved concatenation, let us plot the scatterplot to show the major components in a plane.*

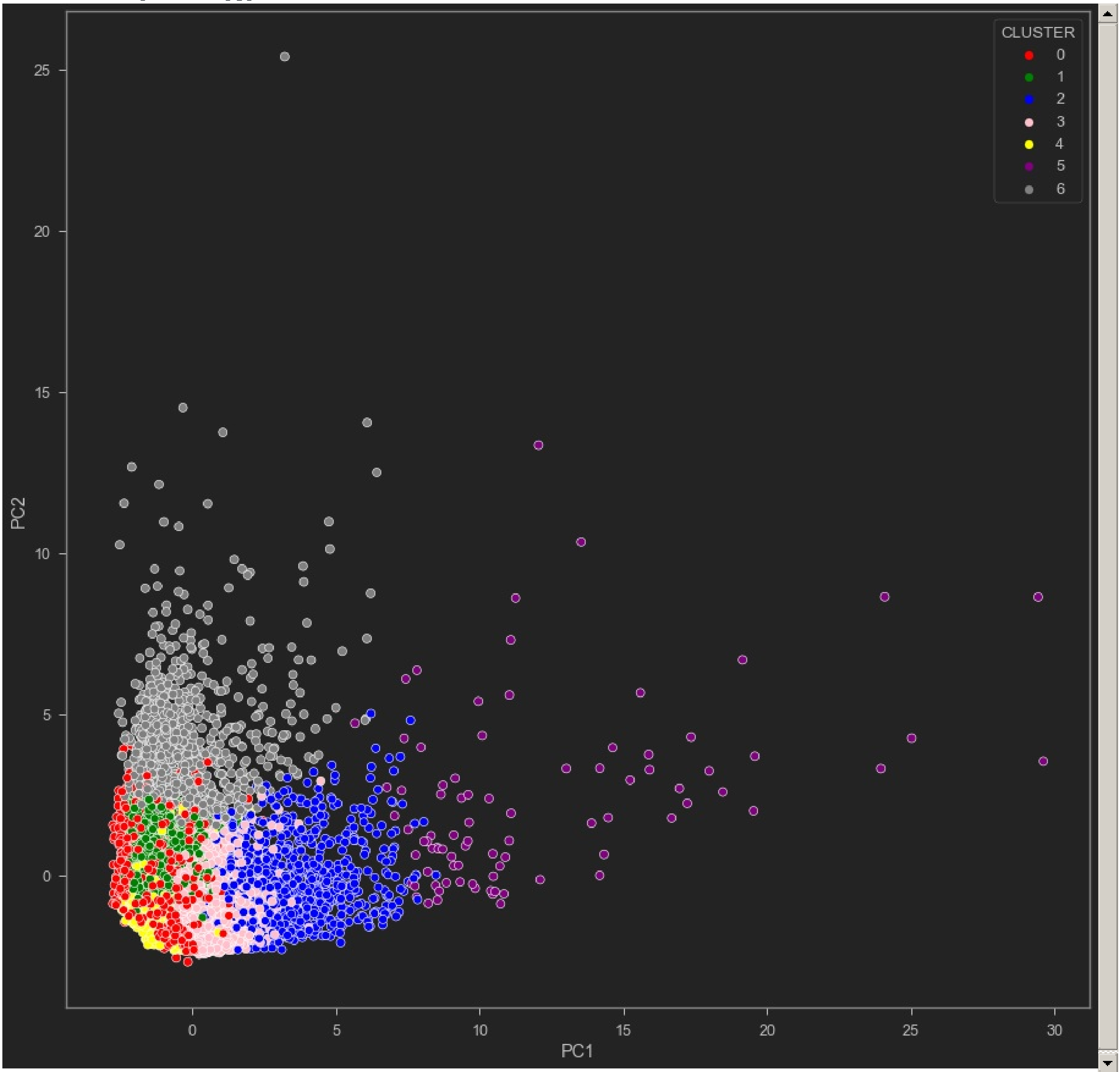
### Scatterplot for Clustering Major Components

In [35]:

```
plt.figure(figsize=(15,15))
ax=sns.scatterplot(x="PC1",y="PC2",hue="CLUSTER", data=pca_df,palette=['red','green','blue','pink','yello
plt.show
```

Out[35]:

```
<function matplotlib.pyplot.show(close=None, block=None)>
```




## Baseline Cluster Analysis

- ***First Customers Cluster (Transactors):*** *Those are customers who pay the least amount of interest charges and are cautious with their finances. Cluster with lowest balance (104 dollars) and cash advance (303 dollars). Percentage of full payment = 23%.*
- ***Second Customers Cluster (Revolvers):*** *Customers who use credit card as a loan (most lucrative sector): highest balance (5000 dollars) and cash advance (~5000 dollars), low purchase frequency, high cash advance frequency (0.5), high cash advance transactions (16) and low percentage of full payment (3%).*
- ***Third Customer Cluster (VIP/Prime):*** *High credit limit (16K dollars) and highest percentage of full payment. Target for increase of credit limit and increase of spending habits.*
- ***Fourth Customer Cluster (low tenure):*** *These are customers with low tenure (7 months or lower) and low balance.*

## Deployment

*Since our objective for this **unsupervised machine learning model** has been to cluster the credit card customers into at least 3 distinctive groups, we have decided with the elbow method to use our 7 clusters to build our app. This will provide us with plenty of segmentation for the bank's marketing campaign.*

*The model itself has been effectively built with **7 clusters** and **6 principal components** for segmentation. This model will provide enough segmentation as previously mentioned and it will also provide enough explained variance.*

*However we will only show the top 3 components in the app. This has been decided in order for us to display our 7 clusters in a 3-D plot for visualization. This will provide the bank with a clear visualization for the marketing campaign ad.*

*Lastly, we have also included 2 additional plots alongside the 3-D plot. One to show the Explained Variance of all 6 components. Another to show the Total Explained Variance of all 6 components. These 2 additional plots will compliment the 3-D plot very well since they will show the cumulative and non-cumulative variance explained through all 6 components, which is not shown in the 3-D plot.*

*Sidenote: Explained Variance will show non-cumulative variance explained. Total Explained Variance will show cumulative variance explained.*

## Performance Evaluation

*What we have done well:*

- *Our representation of our analysis through the use of plots was effective.*
- *Using the combined algorithms of both K-Means and PCA was effective in visualizing our clusters and dimensions.*
- *The production of the App was visually appealing and the use of both a 3-D plot with the 2-D plots thoroughly showed many aspects of the extent of our data research.*

*What we could have done better:*

- *With more clear knowledge of bank finance, we could have had better identification of the clusters in relation to customer segmentation groups.*
- *With knowledge of autoencoders, we could have had more accurate results with our dimension reduction process.*
- *Knowledge of these mentioned areas would have refined our results.*

## Bibliography

***For Our Dataset*** *https://www.kaggle.com/arjunbhasin2013/ccdata*

***Market Segmentation Analysis*** *https://en.wikipedia.org/wiki/Market_segmentation*