

# TASK #1: UNDERSTAND THE PROBLEM STATEMENT AND BUSINESS CASE

- In this project, you have been hired as a data scientist at a bank and you have been provided with extensive data on the bank's customers for the past 6 months.
- Data includes transactions frequency, amount, tenure..etc.
- The bank marketing team would like to leverage AI/ML to launch a targeted marketing ad campaign that is tailored to specific group of customers.
- In order for this campaign to be successful, the bank has to divide its customers into at least 3 distinctive groups.
- This process is known as “marketing segmentation” and it crucial for maximizing marketing campaign conversion rate.



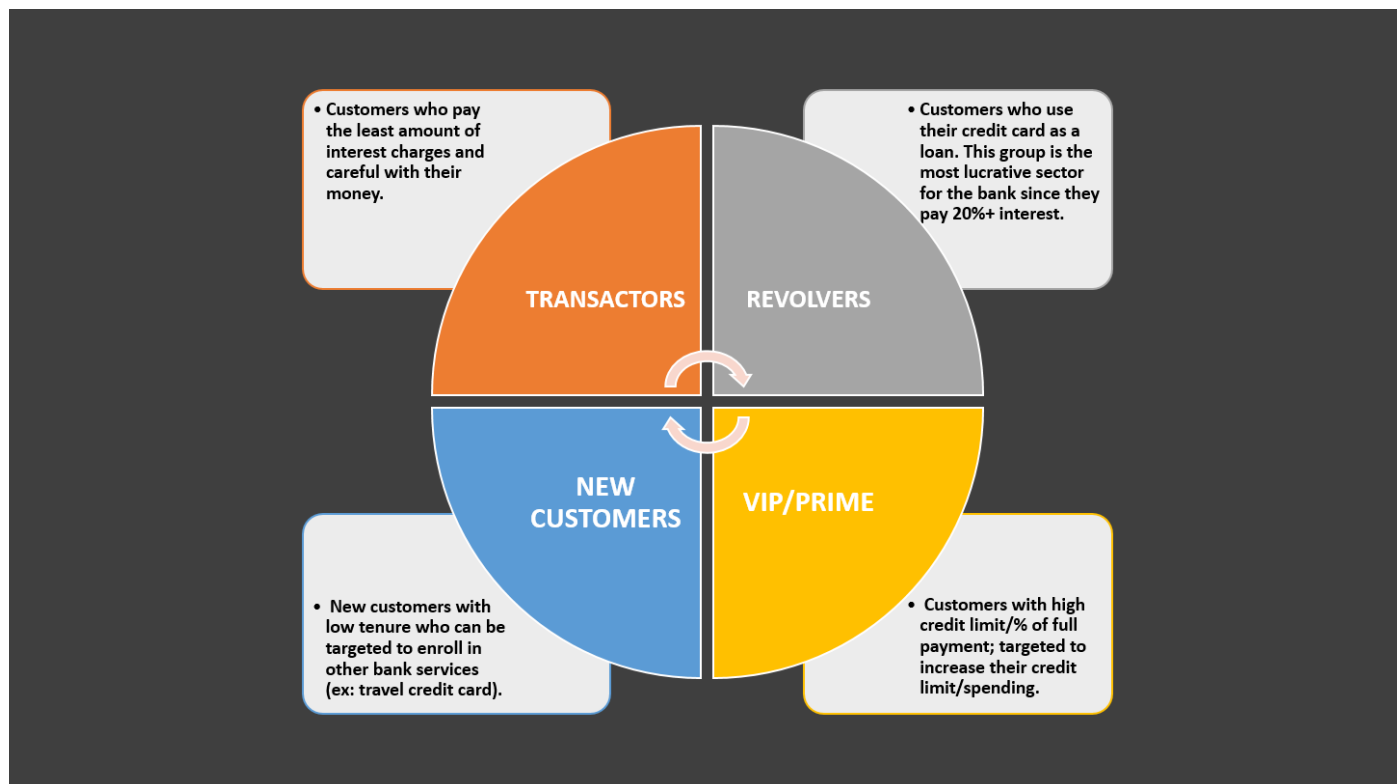
- Data Source: <https://www.kaggle.com/arjunbhasin2013/ccdata>
- Photo Credit: <https://www.needpix.com/photo/1011172/marketing-customer-polaroid-center-presentation-online-board-target-economy>

## INSTRUCTOR

- Adjunct professor & online instructor
- Passionate about artificial intelligence, machine learning, and electric vehicles
- Taught 80,000+ students globally
- MBA (2018), Ph.D. (2014), M.A.Sc (2011)



Ryan Ahmed, Ph.D.



Data Source: <https://www.kaggle.com/arjunbhasin2013/ccdata> (<https://www.kaggle.com/arjunbhasin2013/ccdata>)

## TASK #2: IMPORT LIBRARIES AND DATASETS

In [2]:

```
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)
# setting the style of the notebook to be monokai theme
# this line of code is important to ensure that we are able to see the x and y axes clearly
# If you don't run this code line, you will notice that the xlabel and ylabel on any plot
# is black on black and it will be hard to see them.
```

In [3]:

```
# You have to include the full link to the csv file containing your dataset
creditcard_df = pd.read_csv('marketing_data.csv')

# CUSTID: Identification of Credit Card holder
# BALANCE: Balance amount left in customer's account to make purchases
# BALANCE_FREQUENCY: How frequently the Balance is updated, score between 0 and 1 (1 = frequently updated, 0 = not frequently updated)
# PURCHASES: Amount of purchases made from account
# ONEOFFPURCHASES: Maximum purchase amount done in one-go
# INSTALLMENTS_PURCHASES: Amount of purchase done in installment
# CASH_ADVANCE: Cash in advance given by the user
# PURCHASES_FREQUENCY: How frequently the Purchases are being made, score between 0 and 1 (1 = frequently purchased, 0 = not frequently purchased)
# ONEOFF_PURCHASES_FREQUENCY: How frequently Purchases are happening in one-go (1 = frequently purchased, 0 = not frequently purchased)
# PURCHASES_INSTALLMENTS_FREQUENCY: How frequently purchases in installments are being done (1 = frequently done, 0 = not frequently done)
# CASH_ADVANCE_FREQUENCY: How frequently the cash in advance being paid
# CASH_ADVANCE_TRX: Number of Transactions made with "Cash in Advance"
# PURCHASES_TRX: Number of purchase transactions made
# CREDIT_LIMIT: Limit of Credit Card for user
# PAYMENTS: Amount of Payment done by user
# MINIMUM_PAYMENTS: Minimum amount of payments made by user
# PRC_FULL_PAYMENT: Percent of full payment paid by user
# TENURE: Tenure of credit card service for user
```

In [4]:

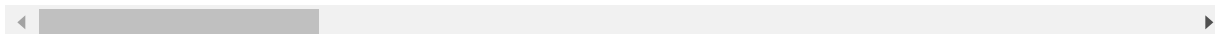
```
creditcard_df
```

Out[4]:

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INST
0	C10001	40.900749	0.818182	95.40	0.00	
1	C10002	3202.467416	0.909091	0.00	0.00	
2	C10003	2495.148862	1.000000	773.17	773.17	
3	C10004	1666.670542	0.636364	1499.00	1499.00	
4	C10005	817.714335	1.000000	16.00	16.00	
5	C10006	1809.828751	1.000000	1333.28	0.00	
6	C10007	627.260806	1.000000	7091.01	6402.63	
7	C10008	1823.652743	1.000000	436.20	0.00	
8	C10009	1014.926473	1.000000	861.49	661.49	
9	C10010	152.225975	0.545455	1281.60	1281.60	
10	C10011	1293.124939	1.000000	920.12	0.00	
11	C10012	630.794744	0.818182	1492.18	1492.18	
12	C10013	1516.928620	1.000000	3217.99	2500.23	
13	C10014	921.693369	1.000000	2137.93	419.96	
14	C10015	2772.772734	1.000000	0.00	0.00	
15	C10016	6886.213231	1.000000	1611.70	0.00	
16	C10017	2072.074354	0.875000	0.00	0.00	
17	C10018	41.089489	0.454545	519.00	0.00	
18	C10019	1989.072228	1.000000	504.35	166.00	
19	C10020	3577.970933	1.000000	398.64	0.00	
20	C10021	2016.684686	1.000000	176.68	0.00	
21	C10022	6369.531318	1.000000	6359.95	5910.04	
22	C10023	132.342240	0.636364	815.90	0.00	
23	C10024	3800.151377	0.818182	4248.35	3454.56	
24	C10025	5368.571219	1.000000	0.00	0.00	
25	C10026	169.781679	1.000000	399.60	0.00	
26	C10027	1615.967240	1.000000	102.00	102.00	
27	C10028	125.694817	1.000000	233.28	0.00	
28	C10029	7152.864372	1.000000	387.05	204.55	
29	C10030	22.063490	1.000000	100.00	0.00	
...	...	...	...	...	...	
8920	C19161	1055.087681	0.666667	0.00	0.00	
8921	C19162	3.417407	0.500000	57.42	0.00	
8922	C19163	33.812837	1.000000	145.98	0.00	

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INST
8923	C19164	486.661842	0.833333	1898.88	939.09	
8924	C19165	7.336804	0.333333	74.00	74.00	
8925	C19166	101.564003	1.000000	418.59	0.00	
8926	C19167	53.097361	0.833333	580.00	0.00	
8927	C19168	91.639086	1.000000	315.20	147.80	
8928	C19169	62.320028	1.000000	500.00	0.00	
8929	C19170	371.527312	0.333333	0.00	0.00	
8930	C19171	229.540018	1.000000	84.00	0.00	
8931	C19172	46.814144	0.833333	235.80	0.00	
8932	C19173	39.552396	1.000000	180.00	0.00	
8933	C19174	735.652303	1.000000	619.60	255.62	
8934	C19175	20.260716	0.833333	110.50	0.00	
8935	C19176	183.817004	1.000000	465.90	0.00	
8936	C19177	108.977282	1.000000	712.50	0.00	
8937	C19178	163.001629	0.666667	0.00	0.00	
8938	C19179	78.818407	0.500000	0.00	0.00	
8939	C19180	728.352548	1.000000	734.40	734.40	
8940	C19181	130.838554	1.000000	591.24	0.00	
8941	C19182	5967.475270	0.833333	214.55	0.00	
8942	C19183	40.829749	1.000000	113.28	0.00	
8943	C19184	5.871712	0.500000	20.90	20.90	
8944	C19185	193.571722	0.833333	1012.73	1012.73	
8945	C19186	28.493517	1.000000	291.12	0.00	
8946	C19187	19.183215	1.000000	300.00	0.00	
8947	C19188	23.398673	0.833333	144.40	0.00	
8948	C19189	13.457564	0.833333	0.00	0.00	
8949	C19190	372.708075	0.666667	1093.25	1093.25	

8950 rows × 18 columns



In [5]:

```
# Let's apply info and get additional insights on our dataframe
# 18 features with 8950 points
creditcard_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8950 entries, 0 to 8949
Data columns (total 18 columns):
CUST_ID                8950 non-null object
BALANCE                8950 non-null float64
BALANCE_FREQUENCY      8950 non-null float64
PURCHASES              8950 non-null float64
ONEOFF_PURCHASES       8950 non-null float64
INSTALLMENTS_PURCHASES 8950 non-null float64
CASH_ADVANCE           8950 non-null float64
PURCHASES_FREQUENCY    8950 non-null float64
ONEOFF_PURCHASES_FREQUENCY 8950 non-null float64
PURCHASES_INSTALLMENTS_FREQUENCY 8950 non-null float64
CASH_ADVANCE_FREQUENCY 8950 non-null float64
CASH_ADVANCE_TRX       8950 non-null int64
PURCHASES_TRX          8950 non-null int64
CREDIT_LIMIT           8949 non-null float64
PAYMENTS               8950 non-null float64
MINIMUM_PAYMENTS       8637 non-null float64
PRC_FULL_PAYMENT       8950 non-null float64
TENURE                 8950 non-null int64
dtypes: float64(14), int64(3), object(1)
memory usage: 1.2+ MB
```

MINI CHALLENGE #1:

- What is the average, minimum and maximum "BALANCE" amount?

In [6]:

```
print("Average = ", creditcard_df['BALANCE'].mean())
print("Min = ", creditcard_df['BALANCE'].min())
print("Max = ", creditcard_df['BALANCE'].max())
```

```
Average = 1564.4748276781006
Min = 0.0
Max = 19043.13856
```

In [7]:

```
# Let's apply describe() and get more statistical insights on our dataframe
# Mean balance is $1564
# Balance frequency is frequently updated on average ~0.9
# Purchases average is $1000
# one off purchase average is ~$600
# Average purchases frequency is around 0.5
# average ONEOFF_PURCHASES_FREQUENCY, PURCHASES_INSTALLMENTS_FREQUENCY, and CASH_ADVANCE_F
# FREQUENCY are generally low
# Average credit limit ~ 4500
# Percent of full payment is 15%
# Average tenure is 11 years
creditcard_df.describe()
```

Out[7]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_FREQUENCY
count	8950.000000	8950.000000	8950.000000	8950.000000	8950.000000
mean	1564.474828	0.877271	1003.204834	592.437371	0.500000
std	2081.531879	0.236904	2136.634782	1659.887917	0.500000
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	128.281915	0.888889	39.635000	0.000000	0.000000
50%	873.385231	1.000000	361.280000	38.000000	0.000000
75%	2054.140036	1.000000	1110.130000	577.405000	0.000000
max	19043.138560	1.000000	49039.570000	40761.250000	0.000000

MINI CHALLENGE #2:

- Obtain the features (row) of the customer who made the maximum "ONEOFF\_PURCHASES"
- Obtain the features of the customer who made the maximum cash advance transaction? how many cash advance transactions did that customer make? how often did he/she pay their bill?

In [8]:

```
creditcard_df[creditcard_df['ONEOFF_PURCHASES'] == 40761.25]
```

Out[8]:

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_FREQUENCY
550	C10574	11547.52001	1.0	49039.57	40761.25	0.000000



In [9]:

```
creditcard_df['CASH_ADVANCE'].max()
```

Out[9]:

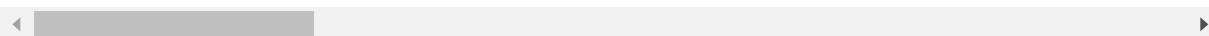
47137.211760000006

In [10]:

```
creditcard_df[creditcard_df['CASH_ADVANCE'] == 47137.211760000006]
```

Out[10]:

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INST
2159	C12226	10905.05381	1.0	431.93	133.5	



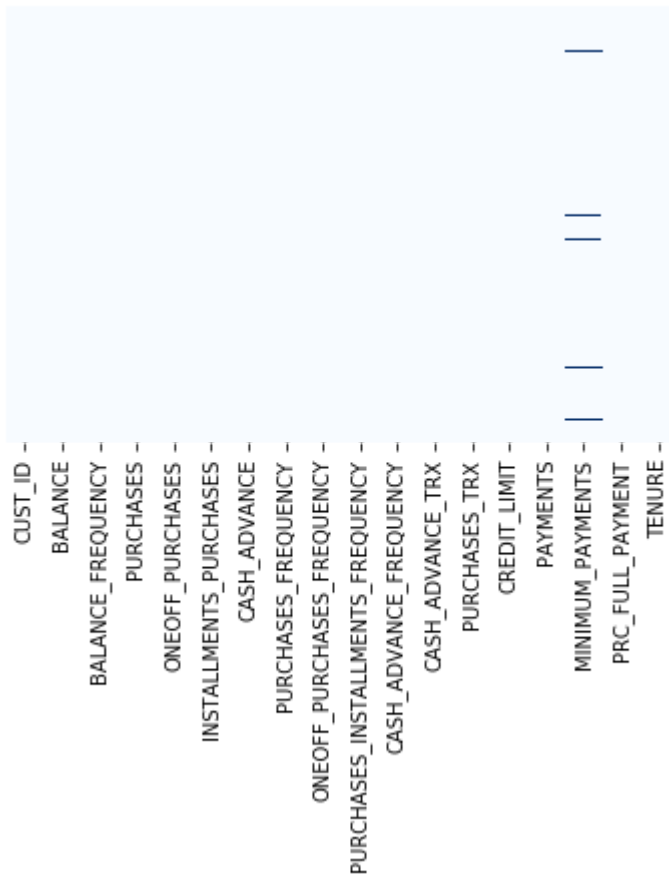
## TASK #3: VISUALIZE AND EXPLORE DATASET

In [11]:

```
# Let's see if we have any missing data, luckily we don't have many!
sns.heatmap(creditcard_df.isnull(), yticklabels = False, cbar = False, cmap="Blues")
```

Out[11]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7fb947ba9590>



In [12]:

```
creditcard_df.isnull().sum()
```

Out[12]:

CUST_ID	0
BALANCE	0
BALANCE_FREQUENCY	0
PURCHASES	0
ONEOFF_PURCHASES	0
INSTALLMENTS_PURCHASES	0
CASH_ADVANCE	0
PURCHASES_FREQUENCY	0
ONEOFF_PURCHASES_FREQUENCY	0
PURCHASES_INSTALLMENTS_FREQUENCY	0
CASH_ADVANCE_FREQUENCY	0
CASH_ADVANCE_TRX	0
PURCHASES_TRX	0
CREDIT_LIMIT	1
PAYMENTS	0
MINIMUM_PAYMENTS	313
PRC_FULL_PAYMENT	0
TENURE	0

dtype: int64

In [13]:

```
# Fill up the missing elements with mean of the 'MINIMUM_PAYMENT'  
creditcard_df.loc[(creditcard_df['MINIMUM_PAYMENTS'].isnull() == True), 'MINIMUM_PAYMENTS']  
= creditcard_df['MINIMUM_PAYMENTS'].mean()
```

In [14]:

```
creditcard_df.isnull().sum()
```

Out[14]:

CUST_ID	0
BALANCE	0
BALANCE_FREQUENCY	0
PURCHASES	0
ONEOFF_PURCHASES	0
INSTALLMENTS_PURCHASES	0
CASH_ADVANCE	0
PURCHASES_FREQUENCY	0
ONEOFF_PURCHASES_FREQUENCY	0
PURCHASES_INSTALLMENTS_FREQUENCY	0
CASH_ADVANCE_FREQUENCY	0
CASH_ADVANCE_TRX	0
PURCHASES_TRX	0
CREDIT_LIMIT	1
PAYMENTS	0
MINIMUM_PAYMENTS	0
PRC_FULL_PAYMENT	0
TENURE	0

dtype: int64

MINI CHALLENGE #3:

- Fill out missing elements in the "CREDIT\_LIMIT" column
- Double check and make sure that no missing elements are present

In [15]:

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

In [16]:

```
creditcard_df.isnull().sum()
```

Out[16]:

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

In [17]:

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

Out[17]:

<matplotlib.axes.\_subplots.AxesSubplot at 0x7fb94915ae90>



CUST\_ID -  
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 -

In [18]:

```
# Let's see if we have duplicated entries in the data  
creditcard_df.duplicated().sum()
```

Out[18]:

0

MINI CHALLENGE #4:

- Drop Customer ID column 'CUST\_ID' and make sure that the column has been removed from the dataframe

In [21]:

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



In [22]:

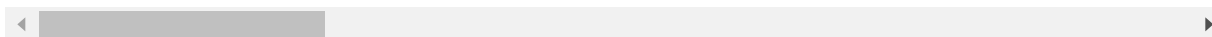
```
creditcard_df
```

Out[22]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS
0	40.900749	0.818182	95.40	0.00	
1	3202.467416	0.909091	0.00	0.00	
2	2495.148862	1.000000	773.17	773.17	
3	1666.670542	0.636364	1499.00	1499.00	
4	817.714335	1.000000	16.00	16.00	
5	1809.828751	1.000000	1333.28	0.00	
6	627.260806	1.000000	7091.01	6402.63	
7	1823.652743	1.000000	436.20	0.00	
8	1014.926473	1.000000	861.49	661.49	
9	152.225975	0.545455	1281.60	1281.60	
10	1293.124939	1.000000	920.12	0.00	
11	630.794744	0.818182	1492.18	1492.18	
12	1516.928620	1.000000	3217.99	2500.23	
13	921.693369	1.000000	2137.93	419.96	
14	2772.772734	1.000000	0.00	0.00	
15	6886.213231	1.000000	1611.70	0.00	
16	2072.074354	0.875000	0.00	0.00	
17	41.089489	0.454545	519.00	0.00	
18	1989.072228	1.000000	504.35	166.00	
19	3577.970933	1.000000	398.64	0.00	
20	2016.684686	1.000000	176.68	0.00	
21	6369.531318	1.000000	6359.95	5910.04	
22	132.342240	0.636364	815.90	0.00	
23	3800.151377	0.818182	4248.35	3454.56	
24	5368.571219	1.000000	0.00	0.00	
25	169.781679	1.000000	399.60	0.00	
26	1615.967240	1.000000	102.00	102.00	
27	125.694817	1.000000	233.28	0.00	
28	7152.864372	1.000000	387.05	204.55	
29	22.063490	1.000000	100.00	0.00	
...	...	...	...	...	
8920	1055.087681	0.666667	0.00	0.00	
8921	3.417407	0.500000	57.42	0.00	
8922	33.812837	1.000000	145.98	0.00	

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS
8923	486.661842	0.833333	1898.88		939.09
8924	7.336804	0.333333	74.00		74.00
8925	101.564003	1.000000	418.59		0.00
8926	53.097361	0.833333	580.00		0.00
8927	91.639086	1.000000	315.20		147.80
8928	62.320028	1.000000	500.00		0.00
8929	371.527312	0.333333	0.00		0.00
8930	229.540018	1.000000	84.00		0.00
8931	46.814144	0.833333	235.80		0.00
8932	39.552396	1.000000	180.00		0.00
8933	735.652303	1.000000	619.60		255.62
8934	20.260716	0.833333	110.50		0.00
8935	183.817004	1.000000	465.90		0.00
8936	108.977282	1.000000	712.50		0.00
8937	163.001629	0.666667	0.00		0.00
8938	78.818407	0.500000	0.00		0.00
8939	728.352548	1.000000	734.40		734.40
8940	130.838554	1.000000	591.24		0.00
8941	5967.475270	0.833333	214.55		0.00
8942	40.829749	1.000000	113.28		0.00
8943	5.871712	0.500000	20.90		20.90
8944	193.571722	0.833333	1012.73		1012.73
8945	28.493517	1.000000	291.12		0.00
8946	19.183215	1.000000	300.00		0.00
8947	23.398673	0.833333	144.40		0.00
8948	13.457564	0.833333	0.00		0.00
8949	372.708075	0.666667	1093.25		1093.25

8950 rows × 17 columns



In [23]:

```
n = len(creditcard_df.columns)
n
```

Out[23]:

17

In [24]:

```
creditcard_df.columns
```

Out[24]:

```
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')
```

In [25]:

```
creditcard_df['TENURE'] = creditcard_df['TENURE'].astype(int)    #for some reason "tenure"  
throws some "scott" error, so eh dont draw its graph  
creditcard_df['CASH_ADVANCE_TRX'] = creditcard_df['CASH_ADVANCE_TRX'].astype(float)  
creditcard_df['PURCHASES_TRX'] = creditcard_df['PURCHASES_TRX'].astype(float)
```

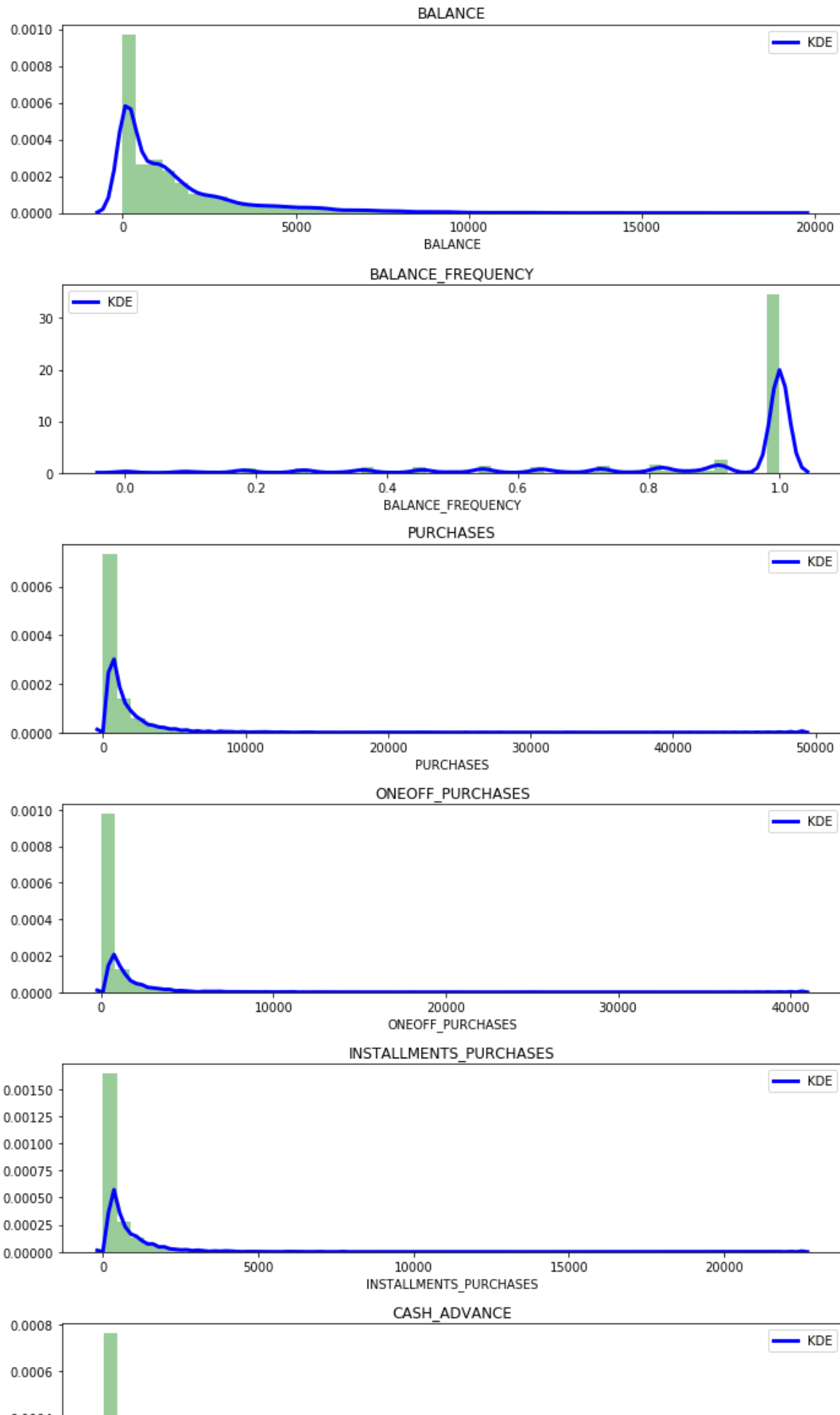
In [26]:

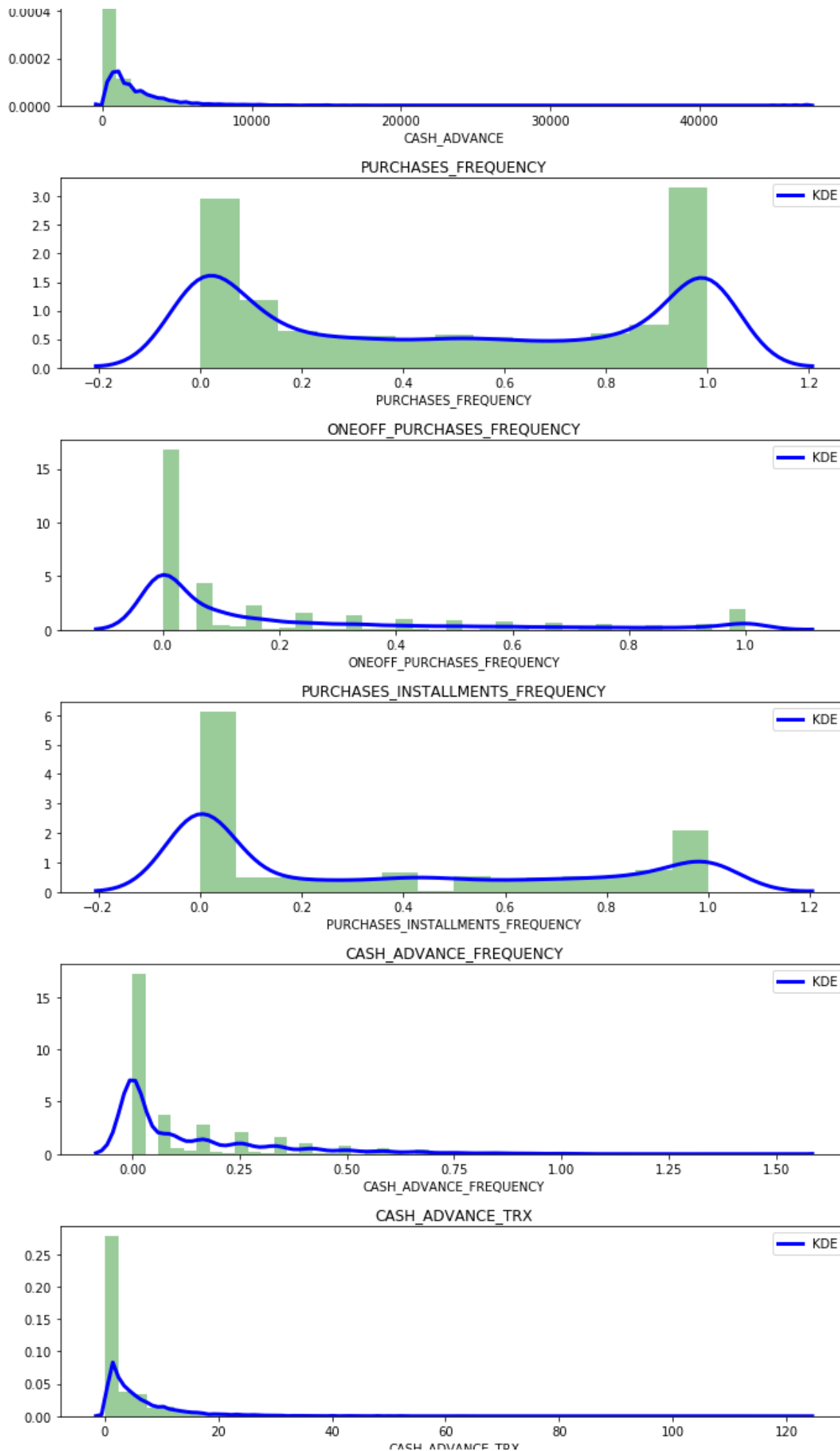
```
# distplot combines the matplotlib.hist function with seaborn kdeplot()
# KDE Plot represents the Kernel Density Estimate
# KDE is used for visualizing the Probability Density of a continuous variable.
# KDE demonstrates the probability density at different values in a continuous variable.

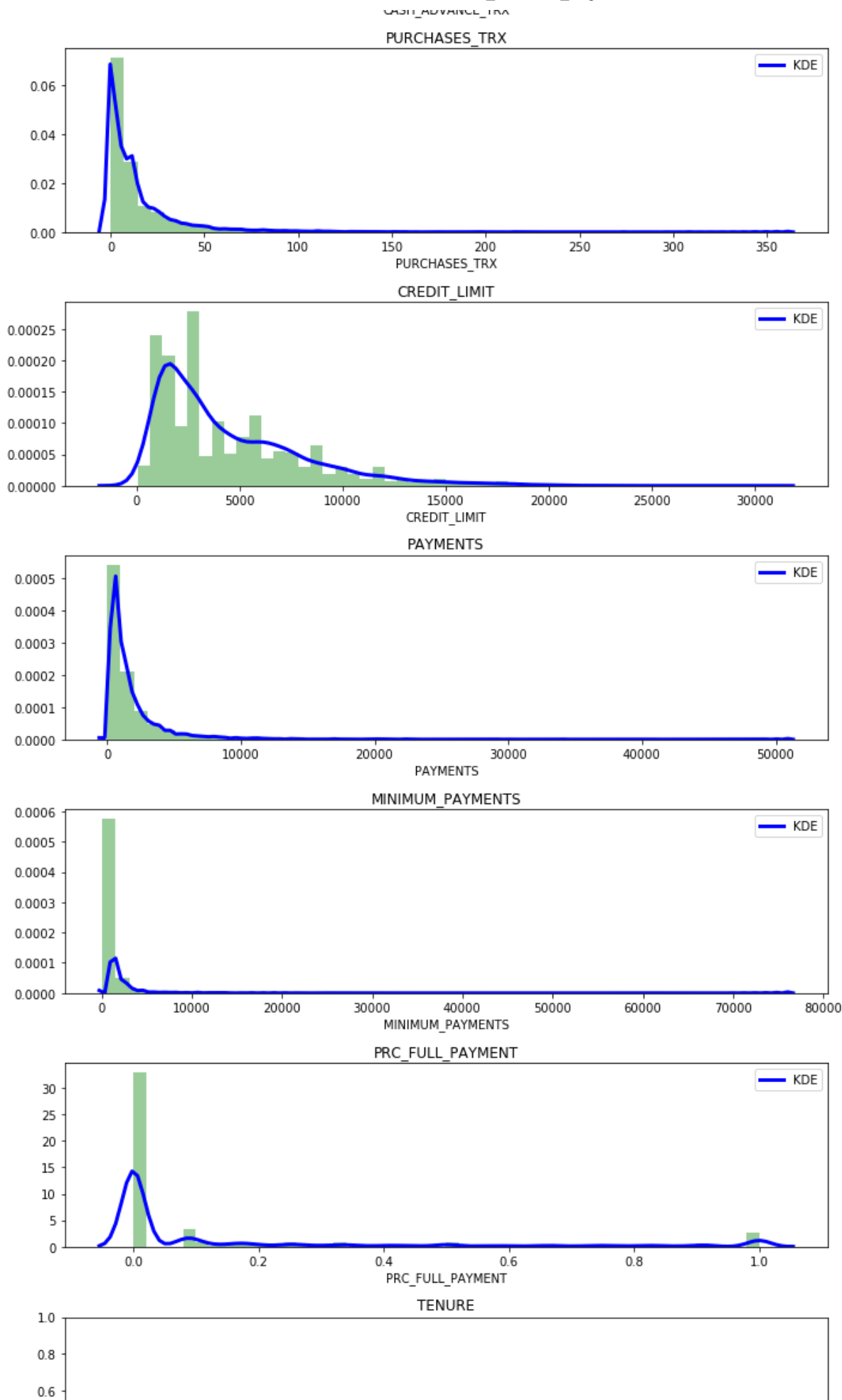
# Mean of balance is $1500
# 'Balance_Frequency' for most customers is updated frequently ~1
# For 'PURCHASES_FREQUENCY', there are two distinct group of customers
# For 'ONEOFF_PURCHASES_FREQUENCY' and 'PURCHASES_INSTALLMENT_FREQUENCY' most users don't
  do one off purchases or installment purchases frequently
# Very small number of customers pay their balance in full 'PRC_FULL_PAYMENT'~0
# Credit limit average is around $4500
# Most customers are ~11 years tenure

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

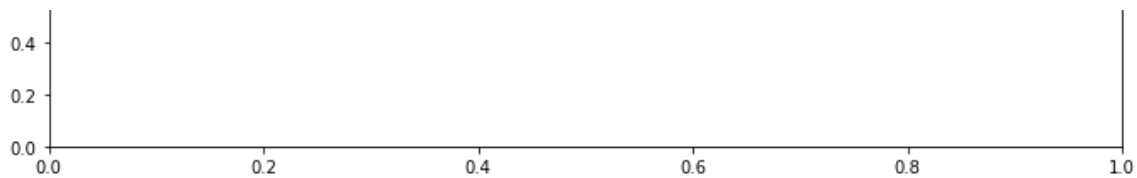
plt.tight_layout()
```











### MINI CHALLENGE #5:

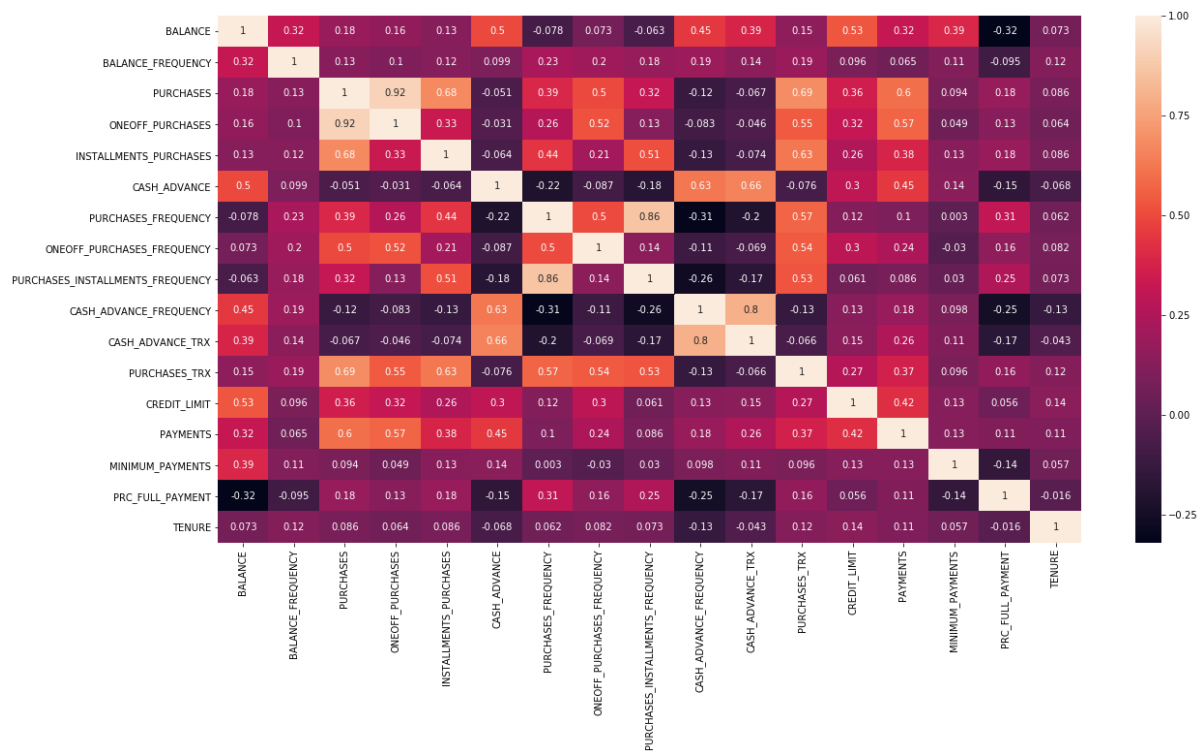
- Obtain the correlation matrix between features

In [68]:

```
correlations = creditcard_df.corr()
f, ax = plt.subplots(figsize = (20, 10))
sns.heatmap(correlations, annot=True)
```

Out[68]:

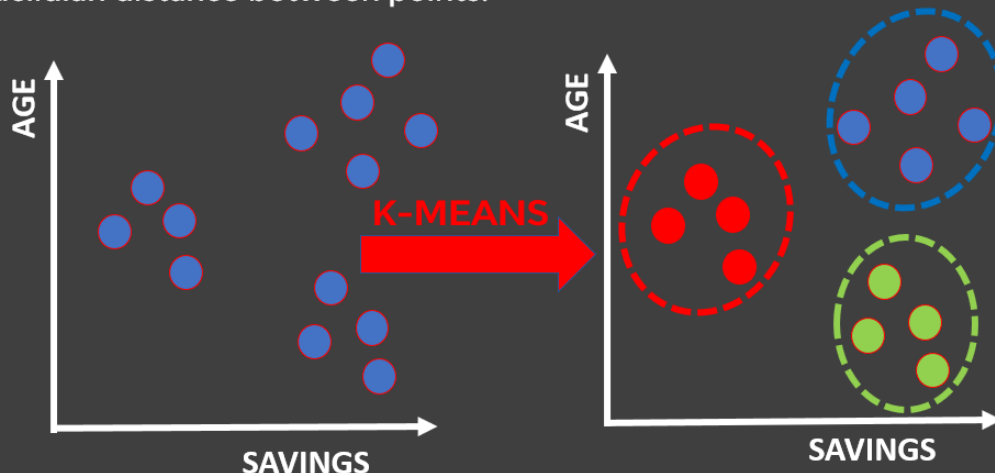
<matplotlib.axes.\_subplots.AxesSubplot at 0x7fd5e17cae50>



## TASK #4: UNDERSTAND THE THEORY AND INTUITON BEHIND K-MEANS

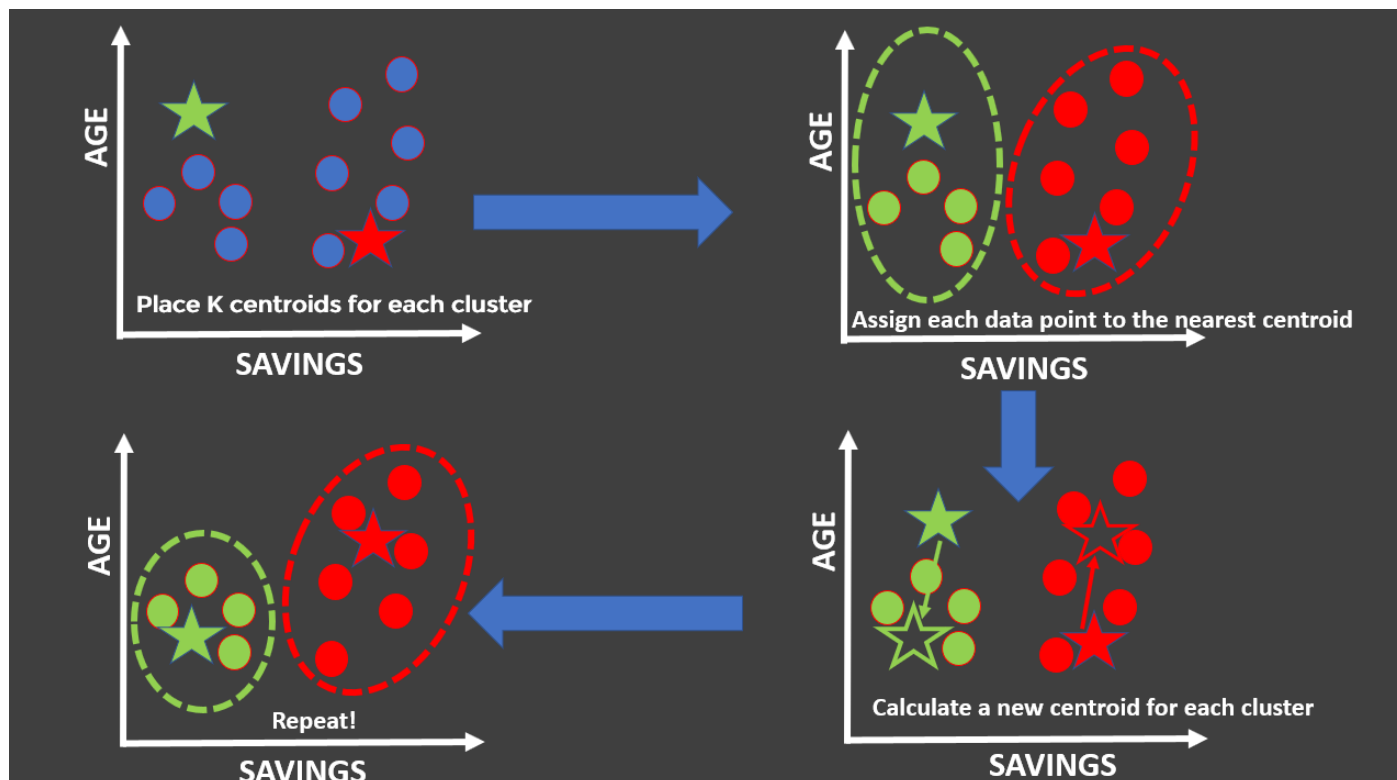
## K-MEANS INTUITION

- K-means is an unsupervised learning algorithm (clustering).
- K-means works by grouping some data points together (clustering) in an unsupervised fashion.
- The algorithm groups observations with similar attribute values together by measuring the Euclidian distance between points.



## K-MEANS ALGORITHM STEPS

1. Choose number of clusters “K”
2. Select random K points that are going to be the centroids for each cluster
3. Assign each data point to the nearest centroid, doing so will enable us to create “K” number of clusters
4. Calculate a new centroid for each cluster
5. Reassign each data point to the new closest centroid
6. Go to step 4 and repeat.



#### MINI CHALLENGE #6:

- Which of the following conditions could terminate the K-means clustering algorithm? (choose 2)
  - K-means terminates after a fixed number of iterations is reached
  - K-means terminates when the number of clusters does not increase between iterations
  - K-means terminates when the centroid locations do not change between iterations

In [29]:

```
#K-means terminates when the centroid locations do not change between iterations
#K-means terminates after a fixed number of iterations is reached
```

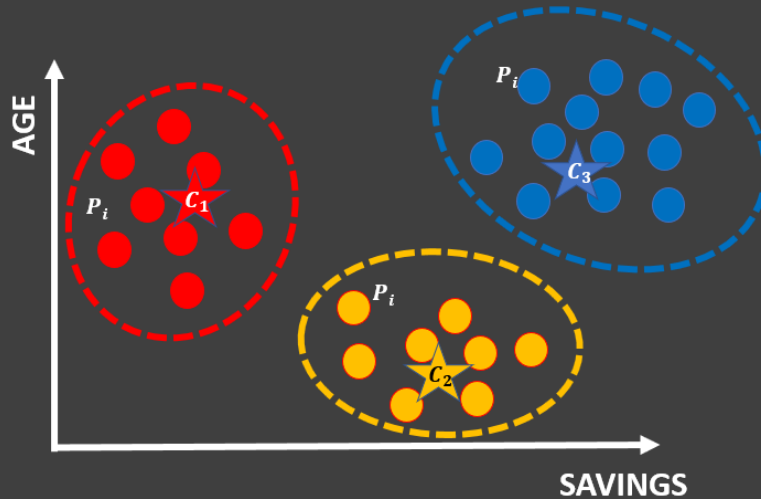
## TASK #5: LEARN HOW TO OBTAIN THE OPTIMAL NUMBER OF CLUSTERS (ELBOW METHOD)

# HOW TO SELECT THE OPTIMAL NUMBER OF CLUSTERS (K)?

## “ELBOW METHOD”

Within Cluster Sum of Squares (WCSS)

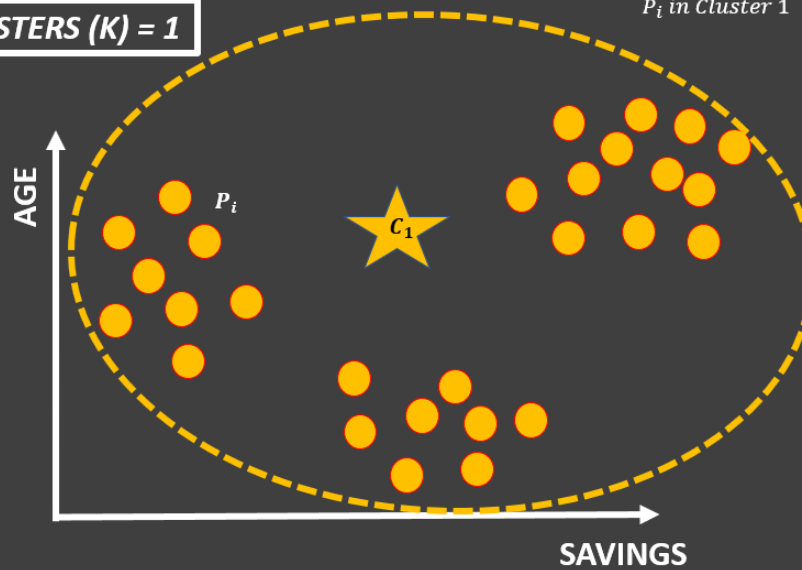
$$= \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster 2}} \text{distance}(P_i, C_2)^2 + \sum_{P_i \text{ in Cluster 3}} \text{distance}(P_i, C_3)^2$$



# HOW TO SELECT THE OPTIMAL NUMBER OF CLUSTERS (K)? “ELBOW METHOD”

$$\text{Within Cluster Sum of Squares (WCSS)} = \sum_{P_i \text{ in Cluster } 1} \text{distance}(P_i, C_1)^2$$

NUMBER OF CLUSTERS (K) = 1

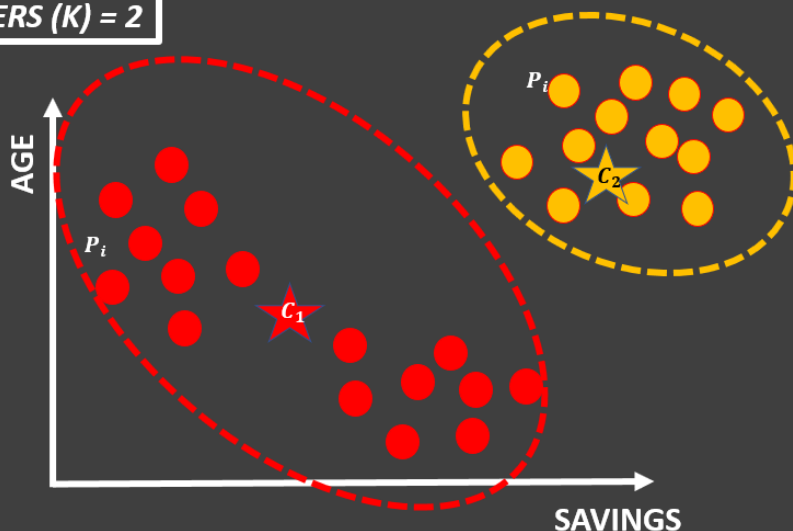


# HOW TO SELECT THE OPTIMAL NUMBER OF CLUSTERS (K)?

## “ELBOW METHOD”

$$\text{Within Cluster Sum of Squares (WCSS)} = \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster 2}} \text{distance}(P_i, C_2)^2$$

**NUMBER OF CLUSTERS (K) = 2**



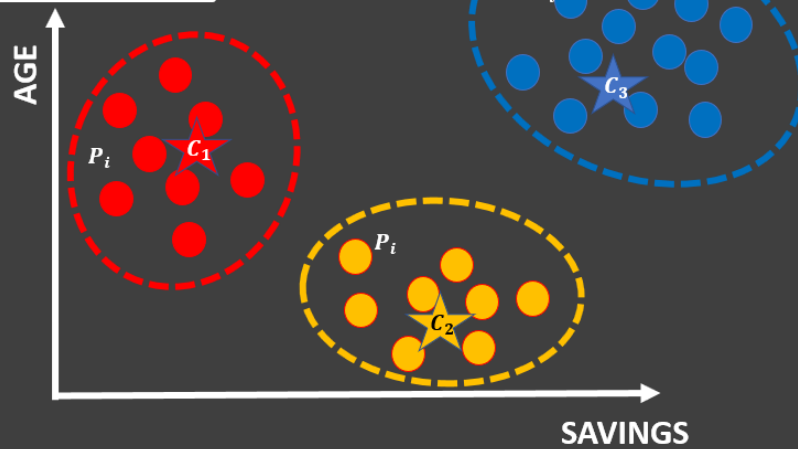
# HOW TO SELECT THE OPTIMAL NUMBER OF CLUSTERS (K)?

## “ELBOW METHOD”

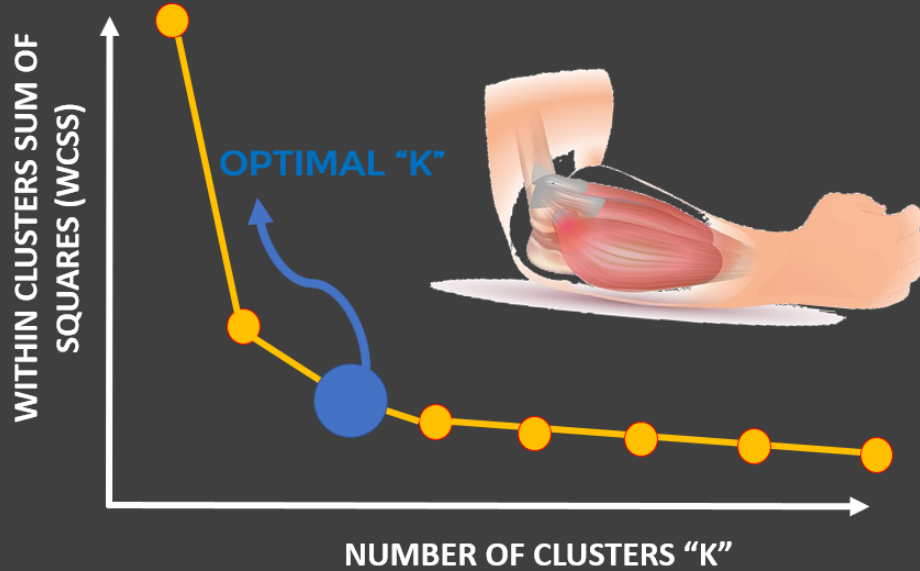
Within Cluster Sum of Squares (WCSS)

$$= \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster 2}} \text{distance}(P_i, C_2)^2 + \sum_{P_i \text{ in Cluster 3}} \text{distance}(P_i, C_3)^2$$

**NUMBER OF CLUSTERS (K) = 3**



## HOW TO SELECT THE OPTIMAL NUMBER OF CLUSTERS (K)? “ELBOW METHOD”



Source: [https://commons.wikimedia.org/wiki/File:Tennis\\_Elbow\\_Illustration.jpg](https://commons.wikimedia.org/wiki/File:Tennis_Elbow_Illustration.jpg)

## TASK #6: FIND THE OPTIMAL NUMBER OF CLUSTERS USING ELBOW METHOD

- The elbow method is a heuristic method of interpretation and validation of consistency within cluster analysis designed to help find the appropriate number of clusters in a dataset.
- If the line chart looks like an arm, then the "elbow" on the arm is the value of k that is the best.
- Source:
  - [https://en.wikipedia.org/wiki/Elbow\\_method\\_\(clustering\)](https://en.wikipedia.org/wiki/Elbow_method_(clustering)) ([https://en.wikipedia.org/wiki/Elbow\\_method\\_\(clustering\)](https://en.wikipedia.org/wiki/Elbow_method_(clustering)))
  - <https://www.geeksforgeeks.org/elbow-method-for-optimal-value-of-k-in-kmeans/> (<https://www.geeksforgeeks.org/elbow-method-for-optimal-value-of-k-in-kmeans/>)



In [34]:

```
# Let's scale the data first  
#bring all data to similar scaling  
scaler = StandardScaler()  
creditcard_df_scaled = scaler.fit_transform(creditcard_df)
```

In [38]:

```
creditcard_df_scaled.shape
```

Out[38]:

```
(8950, 17)
```

In [40]:

```
creditcard_df_scaled
```

Out[40]:

```
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]])
```

In [39]:

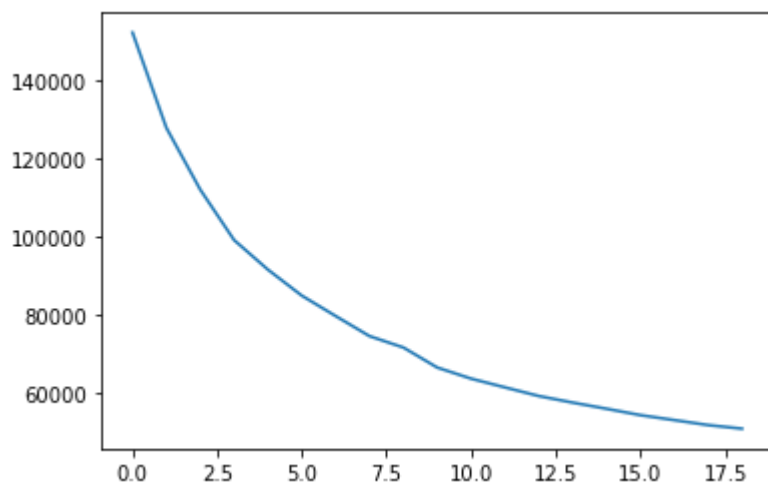
```
# Index(['BALANCE', 'BALANCE_FREQUENCY', 'PURCHASES', 'ONEOFF_PURCHASES',
#       'INSTALLMENTS_PURCHASES', 'CASH_ADVANCE', 'PURCHASES_FREQUENCY',
#       'ONEOFF_PURCHASES_FREQUENCY', 'PURCHASES_INSTALLMENTS_FREQUENCY',
#       'CASH_ADVANCE_FREQUENCY', 'CASH_ADVANCE_TRX', 'PURCHASES_TRX',
#       'CREDIT_LIMIT', 'PAYMENTS', 'MINIMUM_PAYMENTS', 'PRC_FULL_PAYMENT',
#       'TENURE'], dtype='object')
wcss_scores = []
range_values = range(1, 20)

for i in range_values:
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(creditcard_df_scaled)
    #intertia = sum of squared distances to closest centroid
    wcss_scores.append(kmeans.inertia_)
plt.plot(wcss_scores)

# From this we can observe that, 4th cluster seems to be forming the elbow of the curve.
# However, the values does not reduce linearly until 8th cluster.
# Let's choose the number of clusters to be 7 or 8.
```

Out[39]:

[&lt;matplotlib.lines.Line2D at 0x7fb943e6ea90&gt;]



## MINI CHALLENGE #7:

- Let's assume that our data only consists of the first 7 columns of "creditcard\_df\_scaled", what is the optimal number of clusters would be in this case? modify the code and rerun the cells.

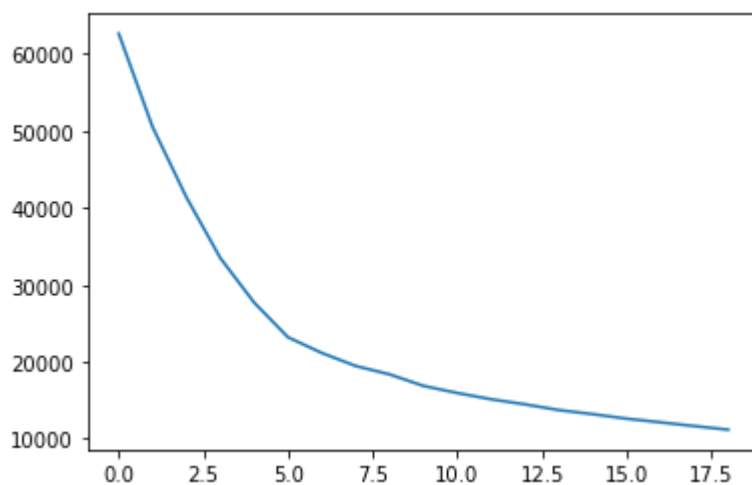
In [41]:

```
wcss_scores = []
range_values = range(1, 20)

for i in range_values:
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(creditcard_df_scaled[:, :7])
    #inertia = sum of squared distances to closest centroid
    wcss_scores.append(kmeans.inertia_)
plt.plot(wcss_scores)
#around 5 clusters is best here
```

Out[41]:

[<matplotlib.lines.Line2D at 0x7fb94331b990>]



## TASK #7: APPLY K-MEANS METHOD

In [42]:

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

In [43]:

```
kmeans.cluster_centers_.shape
```

Out[43]:

(7, 17)

In [44]:

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

Out[44]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PUR
0	-0.370523	0.331560	-0.042700	-0.233421	
1	0.136638	0.430241	0.947732	0.899644	
2	1.675127	0.394566	-0.200657	-0.147856	-
3	1.443847	0.414656	7.093187	6.244031	
4	0.010411	0.402063	-0.344508	-0.225326	-
5	-0.334843	-0.343573	-0.284231	-0.208737	-
6	-0.701934	-2.133938	-0.307125	-0.230482	-

In [45]:

```
# In order to understand what these numbers mean, Let's perform inverse transformation
cluster_centers = scaler.inverse_transform(cluster_centers)
cluster_centers = pd.DataFrame(data = cluster_centers, columns = [creditcard_df.columns])
cluster_centers

# First Customers cluster (Transactors): Those are customers who pay Least amount of interest charges and careful with their money, Cluster with lowest balance ($104) and cash advance ($303), Percentage of full payment = 23%
# Second customers cluster (revolvers) who use credit card as a loan (most lucrative sector): highest balance ($5000) and cash advance (~$5000), 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 and highest percentage of full payment, target for increase credit limit and increase spending habits
# Fourth customer cluster (Low tenure): these are customers with low tenure (7 years), Low balance
```

Out[45]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PAY
0	793.263383	0.955814	911.975010	205.006628	
1	1848.876256	0.979191	3028.049284	2085.661330	
2	5051.110362	0.970740	574.498681	347.027204	
3	4569.720859	0.975499	16157.907683	10956.249146	5
4	1586.144799	0.972516	267.157499	218.442487	
5	867.527490	0.795881	395.941226	245.976624	
6	103.458514	0.371760	347.028335	209.883894	

In [46]:

```
labels.shape # Labels associated to each data point
```

Out[46]:

(8950,)

In [47]:

```
labels.max()
```

Out[47]:

6

In [48]:

```
labels.min()
```

Out[48]:

0

In [49]:

```
y_kmeans = kmeans.fit_predict(creditcard_df_scaled)
y_kmeans
```

Out[49]:

array([1, 2, 4, ..., 6, 6, 6], dtype=int32)

In [50]:

```
# concatenate the clusters labels to our original dataframe
creditcard_df_cluster = pd.concat([creditcard_df, pd.DataFrame({'cluster':labels})], axis
= 1)
creditcard_df_cluster.head()
```

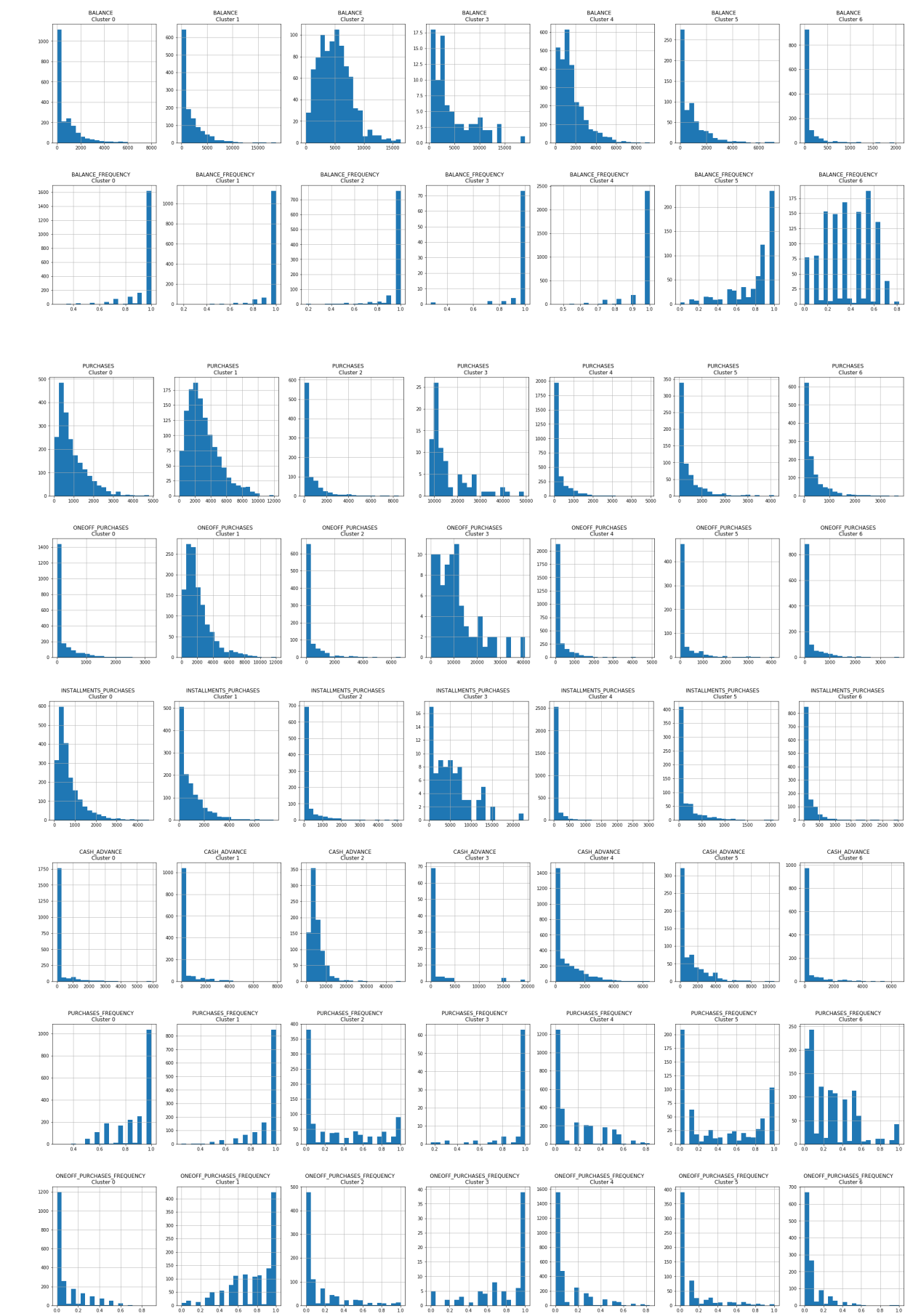
Out[50]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PL
0	40.900749	0.818182	95.40	0.00	
1	3202.467416	0.909091	0.00	0.00	
2	2495.148862	1.000000	773.17	773.17	
3	1666.670542	0.636364	1499.00	1499.00	
4	817.714335	1.000000	16.00	16.00	

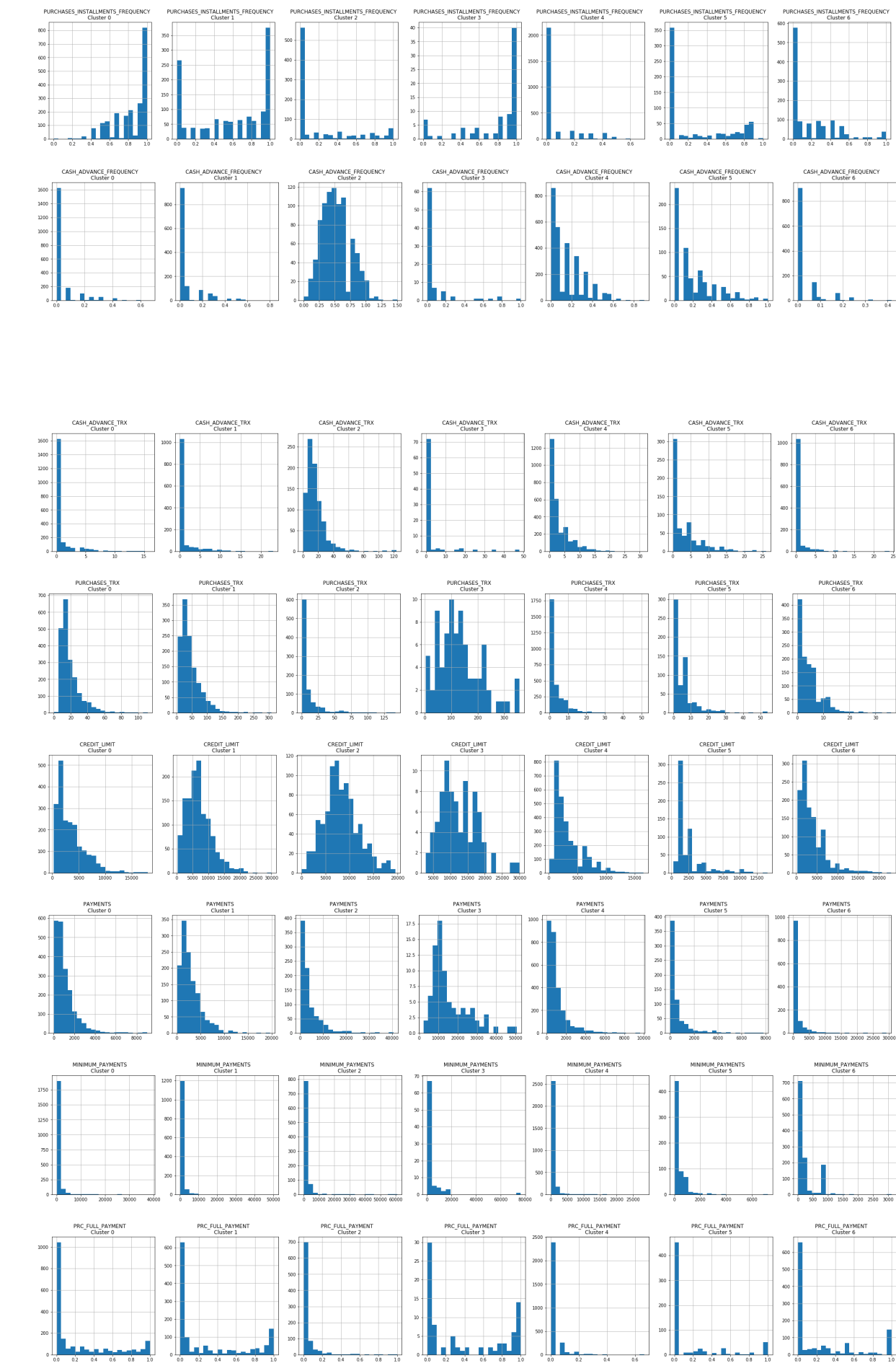
In [51]:

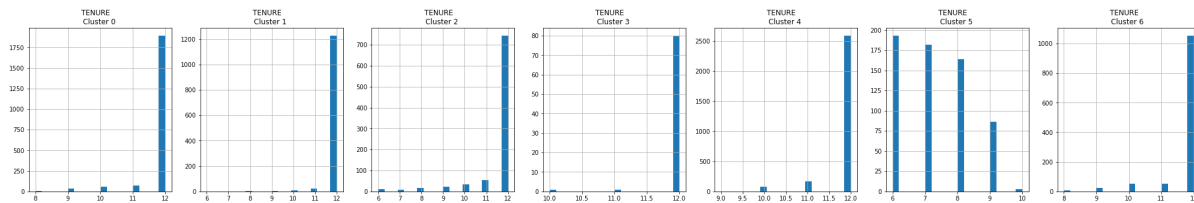
```
# Plot the histogram of various clusters
for i in creditcard_df.columns:
    plt.figure(figsize = (35, 5))
    for j in range(7):
        plt.subplot(1,7,j+1)
        cluster = creditcard_df_cluster[creditcard_df_cluster['cluster'] == j]
        cluster[i].hist(bins = 20)
        plt.title('{} \nCluster {}'.format(i,j))

plt.show()
```









### MINI CHALLENGE #8:

- Repeat the same procedure with 8 clusters instead of 7

## TASK 8: APPLY PRINCIPAL COMPONENT ANALYSIS AND VISUALIZE THE RESULTS

### PRINCIPAL COMPONENT ANALYSIS (PCA)

- PCA is an unsupervised machine learning algorithm.
- PCA performs dimensionality reductions while attempting at keeping the original information unchanged.
- PCA works by trying to find a new set of features called components.
- Components are composites of the uncorrelated given input features.

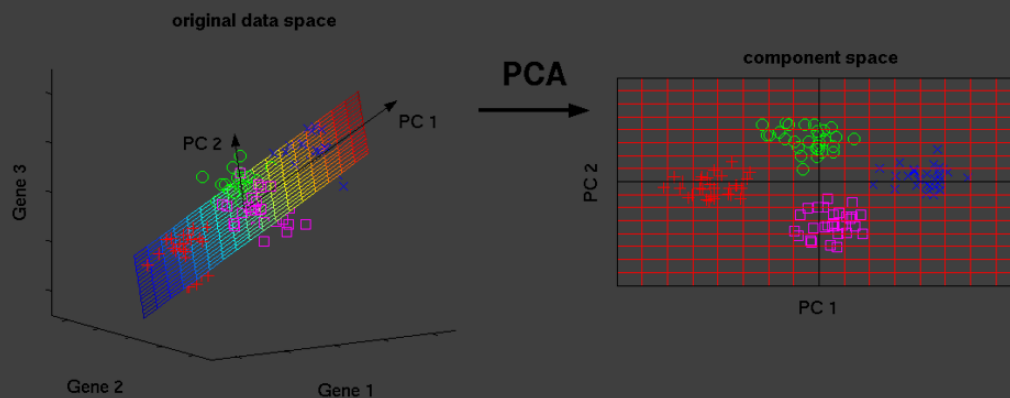


Photo Credit: <http://phdthesis-bioinformatics-maxplanckinstitute-molecularplantphys.matthias-scholz.de/>

In [53]:

```
# Obtain the principal components
pca = PCA(n_components=2)
principal_comp = pca.fit_transform(creditcard_df_scaled)
principal_comp
```

Out[53]:

```
array([[ -1.68222041, -1.07643892],
       [ -1.13829517,  2.50650536],
       [  0.96968413, -0.3835291 ],
       ...,
       [ -0.92620374, -1.81078066],
       [ -2.3365516 , -0.65797677],
       [ -0.5564217 , -0.40050417]])
```

In [54]:

```
# Create a dataframe with the two components
pca_df = pd.DataFrame(data = principal_comp, columns = ['pca1', 'pca2'])
pca_df.head()
```

Out[54]:

	pca1	pca2
0	-1.682220	-1.076439
1	-1.138295	2.506505
2	0.969684	-0.383529
3	-0.873628	0.043194
4	-1.599434	-0.688577

In [55]:

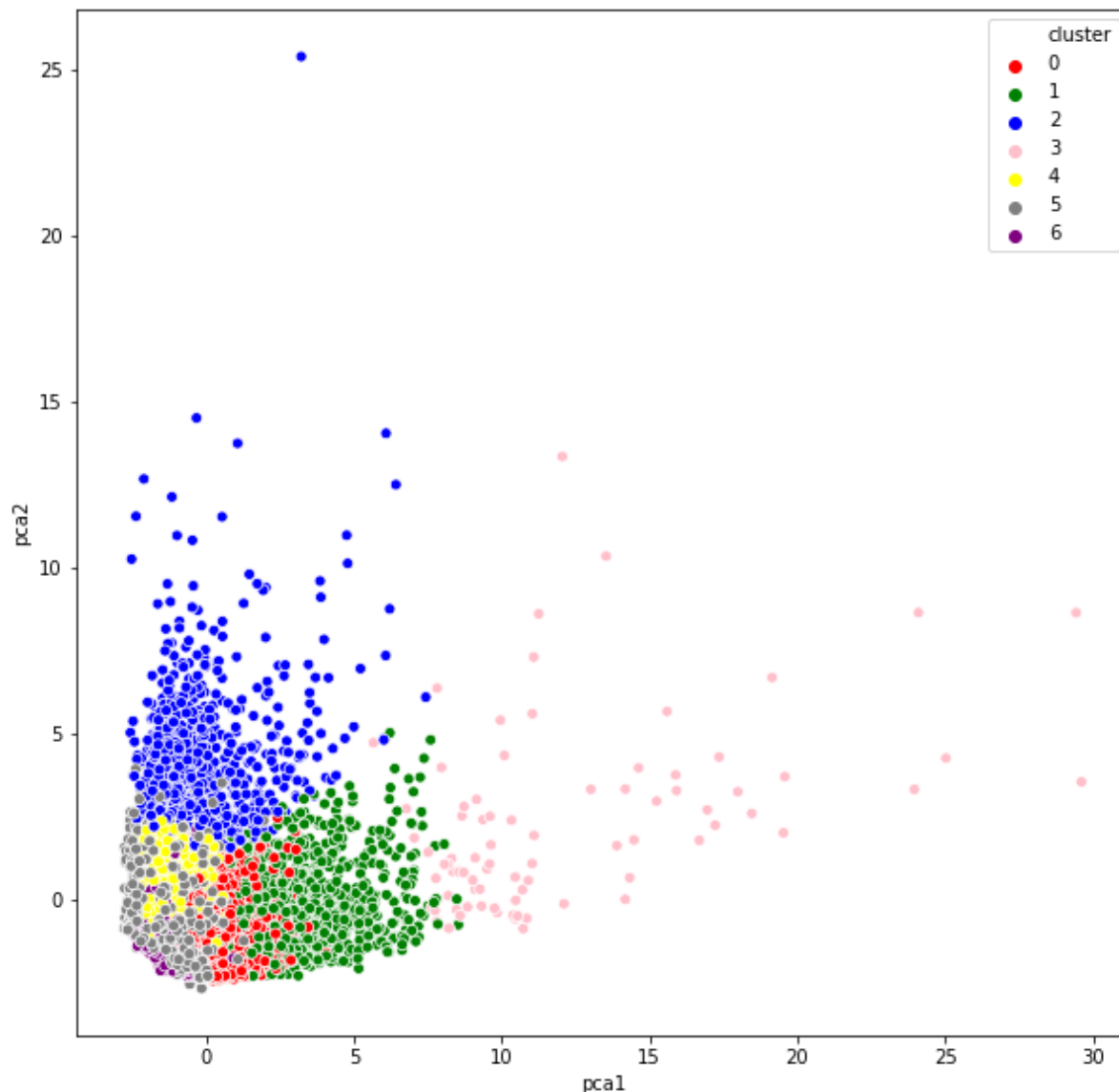
```
# Concatenate the clusters labels to the dataframe
pca_df = pd.concat([pca_df, pd.DataFrame({'cluster': labels})], axis = 1)
pca_df.head()
```

Out[55]:

	pca1	pca2	cluster
0	-1.682220	-1.076439	4
1	-1.138295	2.506505	2
2	0.969684	-0.383529	1
3	-0.873628	0.043194	4
4	-1.599434	-0.688577	4

In [58]:

```
plt.figure(figsize=(10,10))
ax = sns.scatterplot(x="pca1", y="pca2", hue = "cluster", data = pca_df, palette = ['red',
'green', 'blue', 'pink', 'yellow', 'gray', 'purple'])
plt.show()
```



MINI CHALLENGE #9:

- Repeat task #7 and #8 with number of clusters = 7 and 4

## EXCELLENT JOB! YOU SHOULD BE PROUD OF YOUR NEWLY ACQUIRED SKILLS

MINI CHALLENGE SOLUTIONS

## MINI CHALLENGE #1

In [ ]:

```
# Average, minimum and maximum balance amounts
print('The average, minimum and maximum balance amount are:', creditcard_df['BALANCE'].mean(), creditcard_df['BALANCE'].min(), creditcard_df['BALANCE'].max())
```

## MINI CHALLENGE #2

In [ ]:

```
# Let's see who made one off purchase of $40761!
creditcard_df[creditcard_df['ONEOFF_PURCHASES'] == 40761.25]
```

In [ ]:

```
creditcard_df['CASH_ADVANCE'].max()
```

In [ ]:

```
# Let's see who made cash advance of $47137!
# This customer made 123 cash advance transactions!!
# Never paid credit card in full

creditcard_df[creditcard_df['CASH_ADVANCE'] == 47137.211760000006]
```

## MINI CHALLENGE #3

In [ ]:

```
# Fill up the missing elements with mean of the 'CREDIT_LIMIT'
creditcard_df.loc[(creditcard_df['CREDIT_LIMIT'].isnull() == True), 'CREDIT_LIMIT'] = creditcard_df['CREDIT_LIMIT'].mean()
sns.heatmap(creditcard_df.isnull(), yticklabels = False, cbar = False, cmap="Blues")
```

## MINI CHALLENGE #4

In [ ]:

```
# Let's drop Customer ID since it has no meaning here
creditcard_df.drop("CUST_ID", axis = 1, inplace= True)
creditcard_df.head()
```

## MINI CHALLENGE #5

In [ ]:

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

# 'PURCHASES' have high correlation between one-off purchases, 'installment purchases, purchase transactions, credit limit and payments.
# Strong Positive Correlation between 'PURCHASES_FREQUENCY' and 'PURCHASES_INSTALLMENT_FREQUENCY'
```

#### MINI CHALLENGE #6:

- Which of the following conditions could terminate the K-means clustering algorithm? (choose 2)
  - K-means terminates after a fixed number of iterations is reached (True)
  - K-means terminates when the number of clusters does not increase between iterations (False)
  - K-means terminates when the centroid locations do not change between iterations (True)

#### MINI CHALLENGE #7:

In [ ]:

```
# code modification
kmeans.fit(creditcard_df_scaled[:, :7])
# optimal number of clusters would be = 5
```

#### MINI CHALLENGE #8 & #9:

- simply change the values requested in the question and rerun the cells