Marketing. ¿Qué cliente se parece a que cliente?

José Luis Higuera Caraveo

Marzo 12 2022

1 Planteamiento del Problema

El Marketing es fundamental para el crecimiento y la sostenibilidad de cualquier negocio.

Los especialistas en marketing pueden ayudar a desarrollar la marca de la empresa, atraer clientes, aumentar los ingresos y aumentar las ventas. Unos de los puntos críticos para los especialistas en marketing es conocer a sus clientes e identificar sus necesidades.

Al comprender al cliente, los especialistas en marketing pueden lanzar campañas especializadas y dirigidas con el fin de que se adapte a las especificaciones y necesidades de cada cliente.

Al tener a la disponibilidad de los datos, refiriéndose al comportamiento del cliente, se puede usar herramientas relacionadas a la ciencia de datos con el fin de conocer las necesidades específicas de los clientes.

En este caso de estudio se va a simular que un banco de la ciudad de New York desea realizar un caso de estudio con el lanzar una campaña de marketing, pero no cualquier campaña, quiere que esta vaya dirigida en específico para cada tipo de cliente. Tiene como objetivo segmentarlos en al menos 3 grupos. En los últimos 6 meses la empresa ha recabado datos respecto a sus clientes, y serán estos los que se usarán para crear un modelo de clasificación y conocer esta segmentación.

El dataset contiene la siguiente información:

- CUSTID: Identificación del titular de la Tarjeta de Crédito.
- BALANCE: Monto de saldo que queda en su cuenta para realizar compras.
- BALANCEFREQUENCY: Con qué frecuencia se actualiza el Saldo, puntuación entre 0 y 1 (1 = actualizado con frecuencia, 0 = no actualizado con frecuencia). PURCHASES: Importe de las compras realizadas desde la cuenta.
- ONEOFFPURCHASES: Importe máximo de compra realizado de una sola vez.
- INSTALLMENTSPURCHASES: Importe de la compra realizada a plazos.
- CASHADVANCE: Efectivo por adelantado dado por el usuario.
- PURCHASESFREQUENCY: Con qué frecuencia se realizan las Compras, puntuación entre 0 y 1 (1 = compra frecuente, 0 = compra no frecuente).
- ONEOFFPURCHASESFREQUENCY: con qué frecuencia se realizan compras de una sola vez (1 = compra frecuente, 0 = compra no frecuente).
- PURCHASESINSTALLMENTSFREQUENCY: Frecuencia con la que se realizan las compras a plazos (1 = frecuente, 0 = no frecuente).
- CASHADVANCEFREQUENCY: Con qué frecuencia se paga el anticipo de efectivo.
- CASHADVANCETRX: Número de Transacciones realizadas con “Cash in Advanced”.
- PURCHASESTRX: Número de transacciones de compra realizadas,
- CREDITLIMIT: límite de tarjeta de crédito para el usuario.

- PAYMENTS: Importe del pago realizado por el usuario.
- MINIMUM_PAYMENTS : Cantidad mínima de pagos realizados por el usuario.
- PRCFULLPAYMENT: Porcentaje del pago total pagado por el usuario.
- TENURE: Tenencia del servicio de tarjeta de crédito para el usuario.

Los datos que se usarán para este estudio con abiertos y de dominio público, disponibles en el siguiente enlace.

Acceder a los datos.

2 Importar Librerías y los datos

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt
matplotlib.style.use('ggplot')

from sklearn.preprocessing import StandardScaler, normalize
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
```

[2]:
```
creditcard_df = pd.read_csv('./Marketing_data.csv')
creditcard_df.head()
```

[2]:

	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
0	C10001	40.900749	0.818182	95.40	0.00	95.4	0.000000	0.166667	0.000000	0.083333				
1	C10002	3202.467416	0.909091	0.00	0.00	0.0	6442.945483	0.000000	0.000000	0.000000				
2	C10003	2495.148862	1.000000	773.17	773.17	0.0	0.000000	1.000000	1.000000	0.000000				
3	C10004	1666.670542	0.636364	1499.00	1499.00	0.0	205.788017	0.083333	0.083333	0.000000				
4	C10005	817.714335	1.000000	16.00	16.00	0.0	0.000000	0.083333	0.083333	0.000000				

```
0                0.000000                    0              2        1000.0
1                0.250000                    4              0        7000.0
2                0.000000                    0             12        7500.0
3                0.083333                    1              1        7500.0
4                0.000000                    0              1        1200.0

      PAYMENTS  MINIMUM_PAYMENTS  PRC_FULL_PAYMENT  TENURE
0   201.802084        139.509787          0.000000      12
1  4103.032597       1072.340217          0.222222      12
2   622.066742        627.284787          0.000000      12
3     0.000000               NaN          0.000000      12
4   678.334763        244.791237          0.000000      12
```

Sabemos ahora las columnas y los datos con los que nos estamos enfrentando, podemos hacer un análisis rápido del nombre de las columnas, rango de los datos, comprender las características en general.

Analizar estas características puede que no sea suficiente para terminar de entender la data, es por ello que a continuación se hace un análisis más profundo de la misma.

[3]:
```
creditcard_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
```

[4]:
```
creditcard_df.describe()
```

[4]:

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

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

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

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

	PAYMENTS	MINIMUM_PAYMENTS	PRC_FULL_PAYMENT	TENURE
count	8950.000000	8637.000000	8950.000000	8950.000000
mean	1733.143852	864.206542	0.153715	11.517318
std	2895.063757	2372.446607	0.292499	1.338331
min	0.000000	0.019163	0.000000	6.000000

```
25%      383.276166        169.123707          0.000000    12.000000
50%      856.901546        312.343947          0.000000    12.000000
75%     1901.134317        825.485459          0.142857    12.000000
max    50721.483360      76406.207520          1.000000    12.000000
```

Con las últimas dos funciones aplicadas al dataset podemos acceder a información más específica de los datos, podemos analizar la media, desviación estándar, cuantiles, etcétera, de las variables que son numéricas.

También podemos darnos cuenta que algunas columnas contienen datos nulos, por lo que se tiene que trabajar con ellos y tomarla decisión si se eliminan los clientes que los contienen o llenamos esos datos con alguna medida de tendencia central.

Algunos puntos importantes son:

- El balance medio es $1564.
- La frecuencia del balance se actualiza bastante a menudo, en promedio ~ 0.9.
- El promedio de las compras es $1000.
- El importe máximo de compra no recurrente es en promedio ~ $600.
- El promedio de la frecuencia de las compras está cerca de 0.5.
- El promedio del límite de crédito es ~ $4500.
- El porcentaje de pago completo es 15%.
- Los clientes llevan 11 años de promedio en el servicio.

[5]:
```
# Analizando la compra máxima
creditcard_df[creditcard_df["ONEOFF_PURCHASES"] == 40761.25]
```

[5]:
```
     CUST_ID      BALANCE  BALANCE_FREQUENCY  PURCHASES  ONEOFF_PURCHASES  \
550   C10574  11547.52001                1.0   49039.57          40761.25

     INSTALLMENTS_PURCHASES  CASH_ADVANCE  PURCHASES_FREQUENCY  \
550                 8278.32    558.166886                  1.0

     ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
550                         1.0                          0.916667

     CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PURCHASES_TRX  CREDIT_LIMIT  \
550                0.083333                 1            101       22500.0

        PAYMENTS  MINIMUM_PAYMENTS  PRC_FULL_PAYMENT  TENURE
550  46930.59824       2974.069421              0.25      12
```

El comportamiento de los movimientos de este cliente justifica la cantidad de compras que ha realizado.

- El cliente aún tiene más de $11000 para seguir comprando.
- Su compra máxima fue de $40761.25, una compra elevada.
- Tiene una alta frecuencia de compras (1).
- Ha realizado 101 compras y solo 1 compra pidiendo efectivo por adelantado al banco.
- Tiene un límite de crédito de $22500.

En general, este cliente compra con mucha frecuencia, es una persona fiel al banco, usa con mucha frecuencia los servicios, por lo que crear una campaña de marketing para este tipo de cliente posiblemente no sea necesaria si el objetivo es incentivar a comprar. Una enfocada en usar efectivo anticipado podría funcionar mejor.

Analizemos la otra cara de la moneda, aquel cliente cuyas compras las realiza con efectivo por adelantado.

[6]:
```
creditcard_df[creditcard_df['CASH_ADVANCE'] > 47000]
```

[6]:
```
     CUST_ID      BALANCE  BALANCE_FREQUENCY  PURCHASES  ONEOFF_PURCHASES  \
2159  C12226  10905.05381                1.0     431.93             133.5

      INSTALLMENTS_PURCHASES  CASH_ADVANCE  PURCHASES_FREQUENCY  \
2159                  298.43   47137.21176             0.583333

      ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
2159                        0.25                               0.5

      CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PURCHASES_TRX  CREDIT_LIMIT  \
2159                     1.0               123             21       19600.0

         PAYMENTS  MINIMUM_PAYMENTS  PRC_FULL_PAYMENT  TENURE
2159  39048.59762       5394.173671               0.0      12
```

Este cliente cuenta con las siguientes características:

- Tiene un alto saldo para realizar compras.
- Su saldo se actualiza contantemente.
- Sus comprar con efectivo anticipado son mayores respecto a las compras directas.
- Tiene un límite de crédito alto.

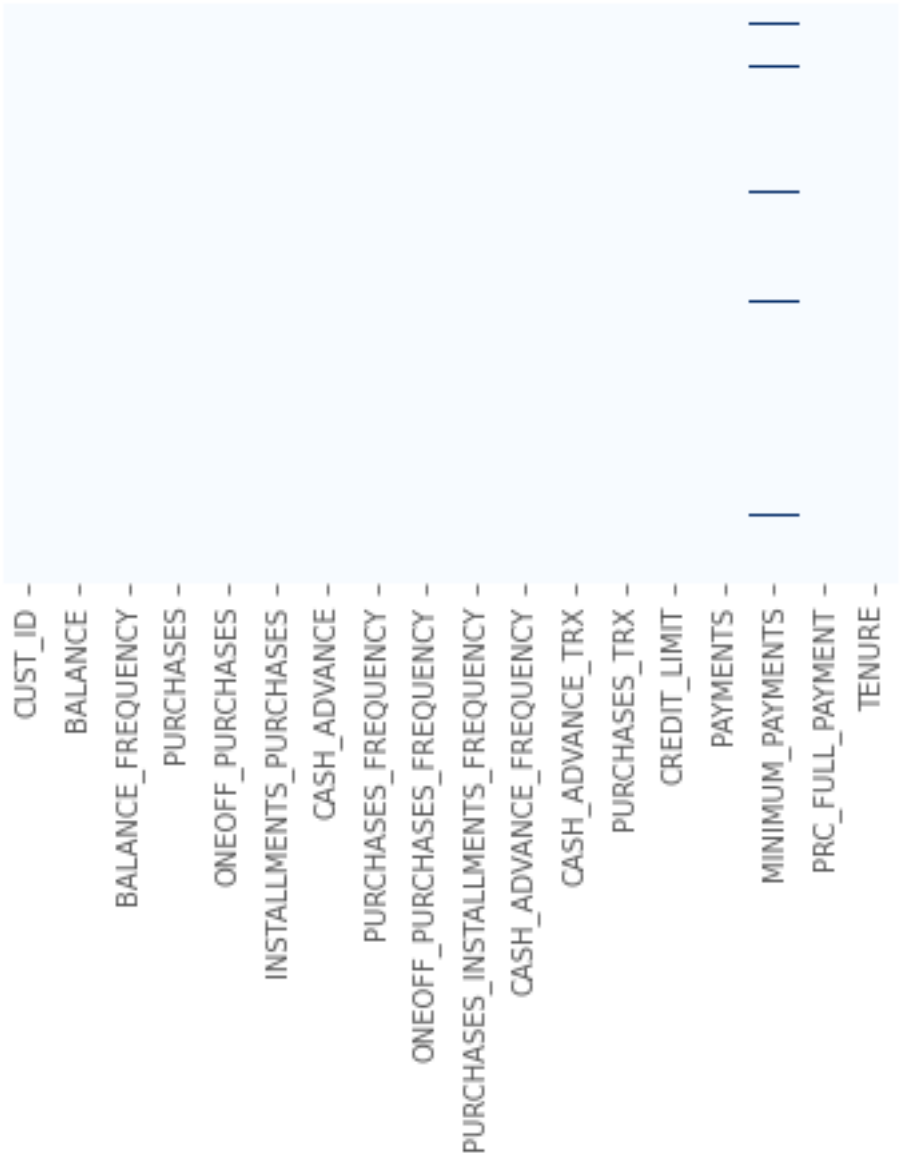
3 Visualización de datos

3.1 Limpieza de datos Nulos y duplicados

3.1.1 Datos Nulos

Antes de iniciar a entrenar el modelo de Machine Learning es necesario verificar si la data tiene datos nulos. Para ello se realiza la siguiente visualización para comprobarlo.

[7]:
```
sns.heatmap(creditcard_df.isnull(), yticklabels=False, cbar=False, cmap='Blues')
plt.show()
```



La visualización nos ayuda a comprender que la columna MINIMUM_PAYMENTS tiene datos faltantes. Otra manera de comprobarlo es como sigue.

[8]:
```
creditcard_df.isnull().sum()
```

[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                      313
PRC_FULL_PAYMENT                        0
TENURE                                  0
dtype: int64
```

Es importante realizar la operación anterior ya que nos arroja la información más detallada de los datos faltantes. En la visualización no se alcanza a apreciar que la columna CREDIT_LIMIT también cuanta con datos faltantes, solo 1, el cual también se tiene que trabajar.

Para este caso de estudio se van a rellenar los datos faltantes con la media de la columna correspondiente.

[9]:
```
# Eliminando los faltantes para la columna MINIMUM_PAYMENTS
creditcard_df.loc[(creditcard_df['MINIMUM_PAYMENTS'].isnull() == True),␣
↪'MINIMUM_PAYMENTS'] = creditcard_df['MINIMUM_PAYMENTS'].mean()

# Eliminando los faltantes para la columna CREDIT_LIMIT
creditcard_df.loc[(creditcard_df['CREDIT_LIMIT'].isnull() == True),␣
↪'CREDIT_LIMIT'] = creditcard_df['CREDIT_LIMIT'].mean()

# Comprobando datos faltantes
creditcard_df.isnull().sum()
```

[9]:
```
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
```

3.1.2 Datos duplicados

Otro proceso importante al momento de estar analizando los datos es si tenemos datos duplicados.

[10]:
```
duplicates = creditcard_df.duplicated().sum()

print('Datos duplicados: {}'.format(duplicates))
```

```
Datos duplicados: 0
```

En este caso no tenemos datos duplicados.

3.2 Limpieza de datos no necesarios

La columna CUST_ID no es necesaria para el modelado, por lo que se procede a eliminarla, esto para no seguir trabajando con ella.

[11]:
```
creditcard_df.drop('CUST_ID', axis=1, inplace=True)
creditcard_df.head()
```

[11]:

	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
0	40.900749	0.818182	95.40	0.00	95.4	0.000000	0.166667	0.000000	0.083333	0.000000	0	2	1000.0
1	3202.467416	0.909091	0.00	0.00	0.0	6442.945483	0.000000	0.000000	0.000000	0.250000	4	0	7000.0
2	2495.148862	1.000000	773.17	773.17	0.0	0.000000	1.000000	1.000000	0.000000	0.000000	0	12	7500.0
3	1666.670542	0.636364	1499.00	1499.00	0.0	205.788017	0.083333	0.083333	0.000000				
4	817.714335	1.000000	16.00	16.00	0.0	0.000000	0.083333	0.083333	0.000000				

```
3                0.083333                     1               1        7500.0
4                0.000000                     0               1        1200.0

      PAYMENTS  MINIMUM_PAYMENTS  PRC_FULL_PAYMENT  TENURE
0   201.802084        139.509787          0.000000      12
1  4103.032597       1072.340217          0.222222      12
2   622.066742        627.284787          0.000000      12
3     0.000000        864.206542          0.000000      12
4   678.334763        244.791237          0.000000      12
```

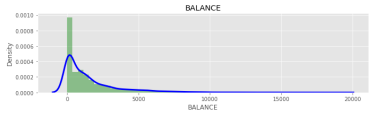
Al eliminar la columna CUST_ID nos quedamos con 17 para seguir trabajando.

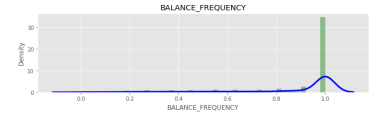
3.3 Visualización de datos

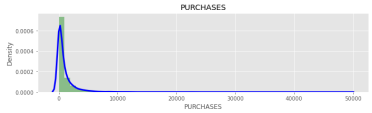
3.3.1 Histogramas y KDE Density

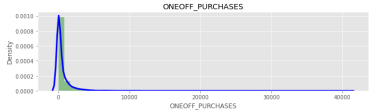
Una manera útil de observar la densidad de una variable es con un gráfico KDE, el cual nos ayuda a visualizar la densidad de probabilidad de una variable continua.

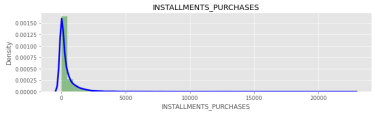
[13]:
```
plt.figure(figsize = (10, 50))
for i in range(len(creditcard_df.columns)):
    plt.subplot(len(creditcard_df.columns), 1, i+1)
    sns.distplot(creditcard_df[creditcard_df.columns[i]], kde_kws = {"color":␣
 ↪"b", "lw": 3, "label": "KDE"}, hist_kws={"color": "g"})
    plt.title(creditcard_df.columns[i])

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

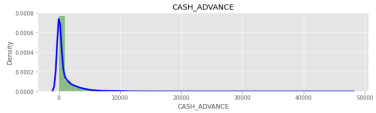

BALANCE
Density
BALANCE

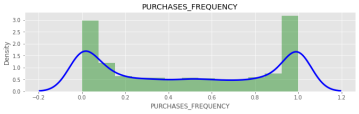

BALANCE_FREQUENCY
Density
BALANCE_FREQUENCY

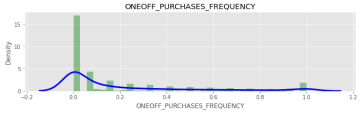

PURCHASES
Density
PURCHASES


ONEOFF_PURCHASES
Density
ONEOFF_PURCHASES

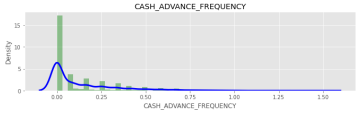

INSTALLMENTS_PURCHASES
Density
INSTALLMENTS_PURCHASES

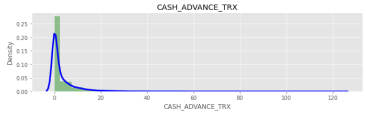

CASH_ADVANCE
Density
CASH_ADVANCE

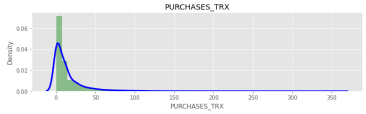

PURCHASES_FREQUENCY
Density
PURCHASES_FREQUENCY

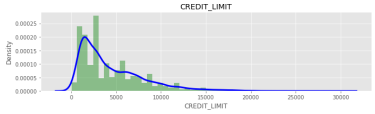

ONEOFF_PURCHASES_FREQUENCY
Density
ONEOFF_PURCHASES_FREQUENCY


PURCHASES_INSTALLMENTS_FREQUENCY
Density
PURCHASES_INSTALLMENTS_FREQUENCY


CASH_ADVANCE_FREQUENCY
Density
CASH_ADVANCE_FREQUENCY


CASH_ADVANCE_TRX
Density
CASH_ADVANCE_TRX

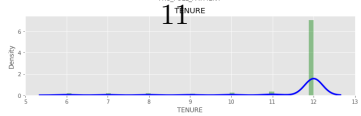

PURCHASES_TRX
Density
PURCHASES_TRX


CREDIT_LIMIT
Density
CREDIT_LIMIT


PAYMENTS
Density
PAYMENTS


MINIMUM_PAYMENTS
Density
MINIMUM_PAYMENTS


PRC_FULL_PAYMENT
Density
PRC_FULL_PAYMENT

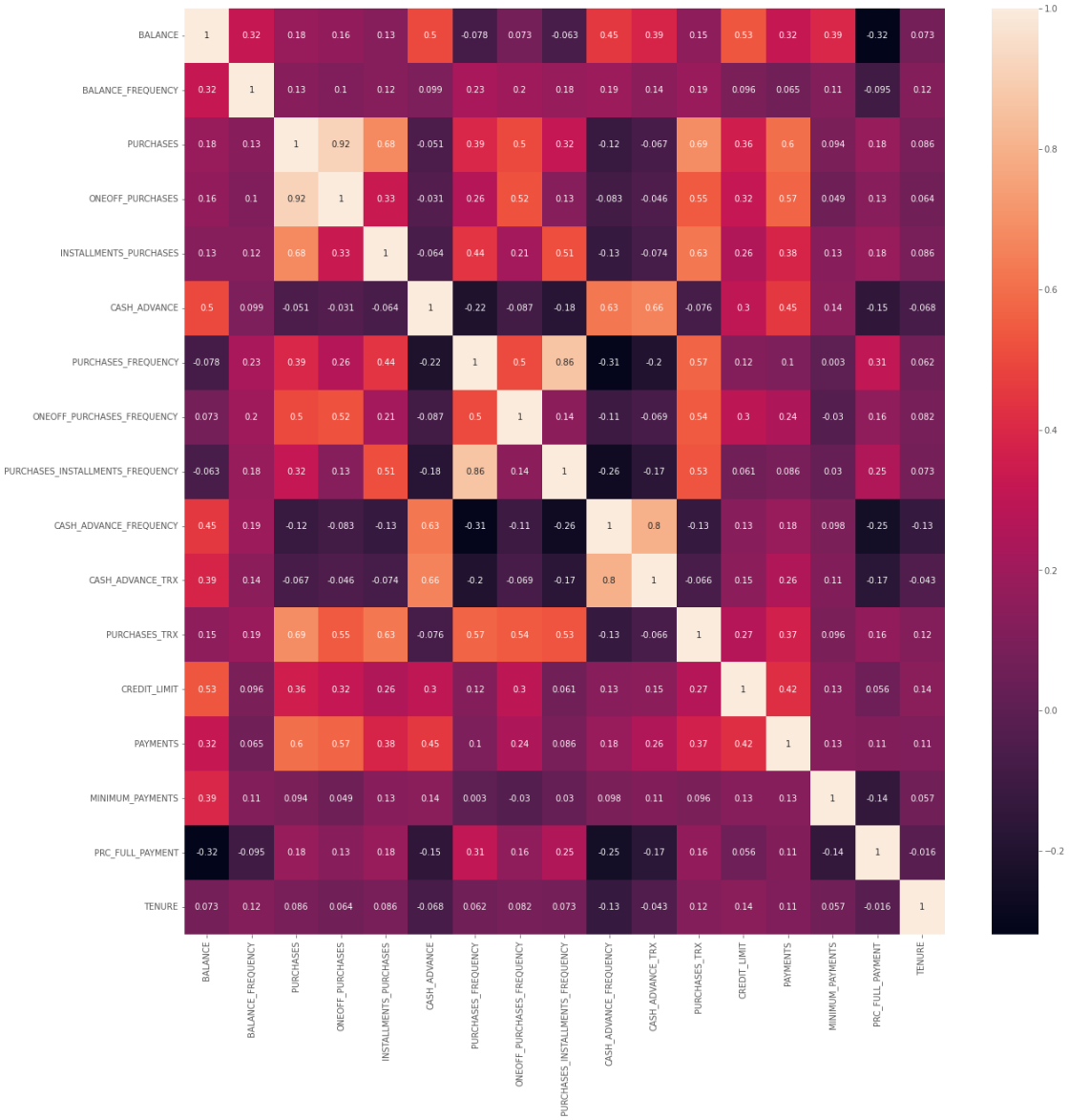

TENURE
Density
TENURE

- El balance promedio es de $1500.
- 'Balance_Frequency' para muchos usuarios se actualiza muy frecuentemente ~ 1.
- Para el campo 'PURCHASES_FREQUENCY', hay dos grupos diferentes de clientes.
- Para los campos 'ONEOFF_PURCHASES_FREQUENCY' y 'PURCHASES_INSTALLMENT_FREQUENCY' la gran mayoría de usuarios no pagan todo de golpe ni a plazos.
- Muy pocos clientes pagan su deuda al completo 'PRC_FULL_PAYMENT' ~ 0.
- El promedio del límite del crédito está entorno de los $4500.
- La mayoría de clientes llevan ~11 años usando el servicio.

3.3.2 Matriz de coorrelaciones

Necesitamos entender cómo se correlacionan las variables, esto nos ayuda a encontrar posibles tendencias en los datos.

[14]:
```
correlations = creditcard_df.corr()

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



- Hay correlación entre 'PURCHASES' y ONEOFF_PURCHASES & INSTALLMENT_PURCHASES
- Se ve una tendencia entre 'PURCHASES' y 'CREDIT_LIMIT' & 'PAYMENTS'
- 'PURCHASES' tienen una alta correlación con ONEOFF_PURCHASES, INSTALLMENTS_PURCHASES, PURCHASES_TRX, CREDIT_LIMIT y PAYMENTS.
- Correlación positiva muy elevada entre 'PURCHASES_FREQUENCY' y 'PURCHASES_INSTALLMENT_FREQUENCY'

4 Entrenamiento del modelo

4.1 K-Means

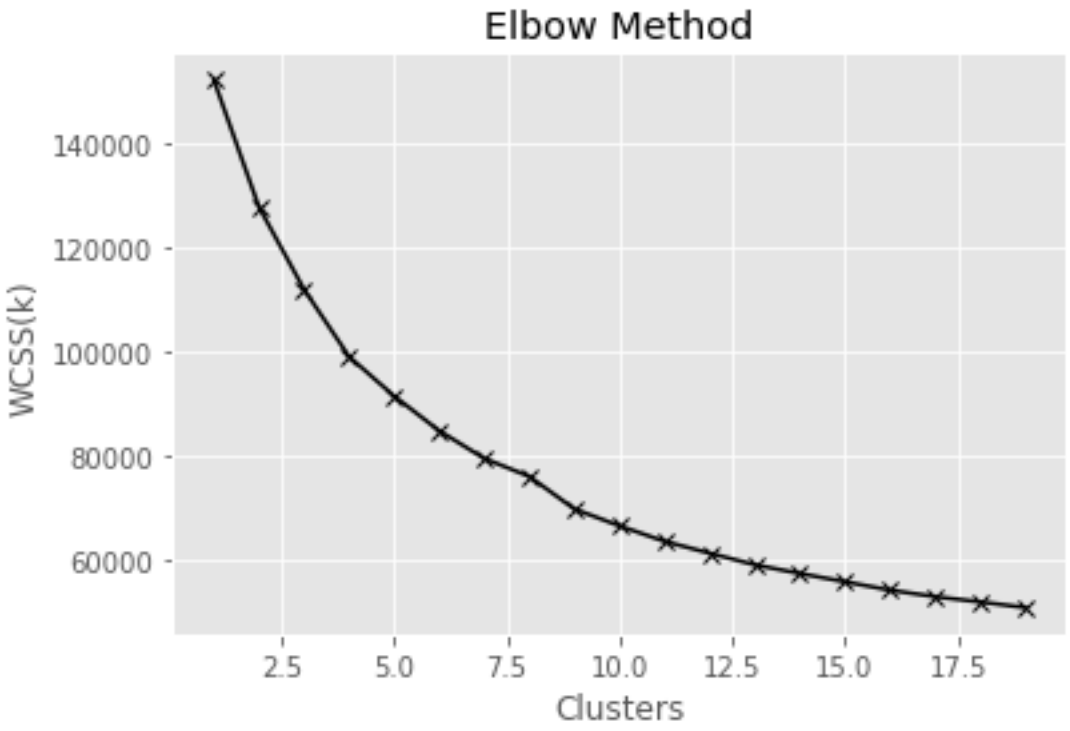
Antes de empezar a modelar, es necesario escalar los datos, esto para evitar que variables con rango mayor dominen versus las otras con dominios más pequeños.

[15]:
```
scaler = StandardScaler()
creditcard_df_scaled = scaler.fit_transform(creditcard_df)
```

4.1.1 Número óptimo de centroides (Método del codo).

Al aplicar un modelo de K-Means es necesario encontrar el número óptimo de clusters en los que la data se va a dividir, no existe una fórmula matemática para procesar este dato, hasta el momento, la mejor manera de hacerlo es con la visualización del método del codo. Este método toma como primicia la varianza intra-cluster entre el centroide y los datos que lo componen. A un número grande de k (centroides), menos varianza. El método del codo, visualizando esta varianza toma como k óptimo aquel cuya diferencia entre k y k + 1 ya no mejore considerablemente.

[16]:
```
# Encontrar K óptimo
scores_1 = []
range_values = range(1,20)

for i in range_values:
  kmeans = KMeans(n_clusters=i)
  kmeans.fit(creditcard_df_scaled)
  scores_1.append(kmeans.inertia_)

# Visualizar para elegir el K óptimo
plt.plot(range_values, scores_1, 'kx-')
plt.title('Elbow Method')
plt.xlabel('Clusters')
plt.ylabel('WCSS(k)')
plt.show()
```



Con el gráfico podemos ver que en 4 clústers es donde se forma el codo de la curva. Sin embargo, los valores no se reducen a una forma lineal hasta el 8º cluster. Elijamos un número de clústers igual a 8.

4.1.2 Entrenamiento

[17]:
```
k = 8

kmeans = KMeans(n_clusters=k)
kmeans.fit(creditcard_df_scaled)
labels = kmeans.labels_
```

Los cluster centers son aquellos centroides que ayudan a separar la data en diferentes segmentos. Para visualizarlos los incluimos en un Dataframe.

[18]:
```
cluster_centers = pd.DataFrame(data = kmeans.cluster_centers_, columns =␣
↪[creditcard_df.columns])
cluster_centers
```

[18]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	\
0	0.901817	0.466986	2.270963	1.756936	
1	0.019493	0.403153	-0.361863	-0.246971	
2	1.698325	0.393098	-0.215463	-0.154529	
3	1.923051	0.337717	11.212042	10.600367	

```
4 -0.701229         -2.144116  -0.311099        -0.235720
5 -0.336050         -0.347078  -0.289267        -0.215966
6 -0.165253          0.392196   0.453349         0.593167
7 -0.364778          0.333613  -0.037381        -0.244339

  INSTALLMENTS_PURCHASES CASH_ADVANCE PURCHASES_FREQUENCY  \
0               2.141920    -0.195512            1.158629
1              -0.401779    -0.086621           -0.867503
2              -0.225632     2.025668           -0.471452
3               7.033118     0.419625            1.046983
4              -0.302414    -0.321905           -0.556586
5              -0.286835     0.068284           -0.203078
6              -0.017967    -0.333914            0.943302
7               0.360316    -0.363589            0.990669

  ONEOFF_PURCHASES_FREQUENCY PURCHASES_INSTALLMENTS_FREQUENCY  \
0                   1.583889                          1.226198
1                  -0.410513                         -0.758672
2                  -0.210500                         -0.409161
3                   1.915501                          0.981334
4                  -0.444989                         -0.439730
5                  -0.288661                         -0.224549
6                   1.878357                          0.089014
7                  -0.387079                          1.206081

  CASH_ADVANCE_FREQUENCY CASH_ADVANCE_TRX PURCHASES_TRX CREDIT_LIMIT  \
0              -0.312181        -0.212097      2.781452     1.238947
1               0.115631        -0.020700     -0.486861    -0.305126
2               1.920837         1.941432     -0.263115     1.040171
3              -0.258912         0.061229      5.362438     3.044064
4              -0.520844        -0.376103     -0.419790    -0.177161
5               0.308663         0.000996     -0.388117    -0.567159
6              -0.407665        -0.323378      0.523732     0.373578
7              -0.475238        -0.361153      0.187666    -0.260925

   PAYMENTS MINIMUM_PAYMENTS PRC_FULL_PAYMENT    TENURE
0  1.290295         0.441655         0.304778  0.334182
1 -0.248169        -0.008412        -0.456474  0.271801
2  0.828342         0.557352        -0.392330  0.071341
3  8.098975         1.120318         1.110132  0.310863
4 -0.202048        -0.256658         0.281550  0.199199
5 -0.392680        -0.209145         0.014011 -3.203733
6  0.086557        -0.162605         0.406347  0.261047
7 -0.216886        -0.032660         0.313849  0.257637
```

Los datos anteriores están escalados, por lo que resulta complicado entender realmente que significan. Para ello aplicaremos la transformación inversa del escalado para obtener los valores reales y

así poder analizar más clara y adecuadamente.

[19]:
```
cluster_centers = scaler.inverse_transform(cluster_centers)
cluster_centers = pd.DataFrame(data = cluster_centers, columns = [creditcard_df.
 ↪columns])
cluster_centers
```

[19]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES
0	3441.530986	0.987896	5855.151608	3508.591111
1	1605.047605	0.972774	230.077907	182.515426
2	5099.393953	0.970392	542.864477	335.950907
3	5567.142164	0.957273	24957.905000	18186.875667
4	104.925267	0.369349	338.537483	201.190254
5	865.015978	0.795051	385.181720	233.977974
6	1220.514994	0.970178	1971.792676	1576.972447
7	805.220083	0.956301	923.338824	186.885283

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY
0	2347.978936	568.874079	0.955365
1	47.744156	797.223294	0.142179
2	207.031791	5226.790667	0.301134
3	6771.029333	1858.844605	0.910556
4	137.598754	303.821813	0.266966
5	151.686061	1122.064941	0.408846
6	394.820228	278.637458	0.868943
7	736.896637	216.408238	0.887954

	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INSTALLMENTS_FREQUENCY
0	0.674962	0.851760
1	0.079994	0.062922
2	0.139661	0.201826
3	0.773889	0.754444
4	0.069709	0.189677
5	0.116344	0.275196
6	0.762808	0.399814
7	0.086985	0.843765

	CASH_ADVANCE_FREQUENCY	CASH_ADVANCE_TRX	PURCHASES_TRX	CREDIT_LIMIT
0	0.072674	1.801418	83.846336	9002.245863
1	0.158283	3.107562	2.608297	3384.275575
2	0.519523	16.497674	8.169767	8279.016913
3	0.083333	3.666667	148.000000	15570.000000
4	0.030918	0.682203	4.275424	3849.863936
5	0.196911	3.255627	5.062701	2430.891398
6	0.053566	1.042009	27.727854	5853.677875
7	0.040044	0.784226	19.374504	3545.099307

```
       PAYMENTS MINIMUM_PAYMENTS PRC_FULL_PAYMENT     TENURE
0   5468.421612      1893.464949         0.242857  11.964539
1   1014.718178       844.603245         0.020204  11.881057
2   4131.114001      2163.092995         0.038965  11.612791
3  25178.882690      3475.059479         0.478409  11.933333
4   1148.234177       266.075424         0.236063  11.783898
5    596.373827       376.802926         0.157813   7.229904
6   1983.717894       485.262318         0.272564  11.866667
7   1105.280930       788.094852         0.245510  11.862103
```

Analicemos los clúster más relevantes.

- Segundo Clúster de Clientes: Estos son clientes que hacen uso del servicio muy poco, tiene una frecuencia de compra de apenas 0.14, aunque tienen un saldo de $1605, por lo que se considera como aquellos clientes que ahorran.
- Cuarto Clúster de Clientes: Tienen un saldo mayor que el promedio, pero sin ser muy elevado, frecuencia alta de compra, sus compras de una sola transacción son altas, además usan el crédito con frecuencia.
- Sexto clúster de Clientes: Son aquellos que su nivel de compra es la más alta, su importe máximo de compra en una sola vez es el más alto. Es decir, aquellos clientes con mayor flujo de compras. Por ende, los que más pagan impuestos.
- Séptimo clúster de Clientes: Son aquellos que aportan mayor efectivo, es decir, no lo piden por adelantado al banco, por lo que sus transacciones con Cash in Advance son bajas.
- Octavo clúster de Clientes: Son clientes que se asemejan a los anteriores, solo que estos compran menos y su saldo en cuenta no es muy elevado.

Ahora ya podemos agregar al dataset original a que cluster corresponde cada cliente.

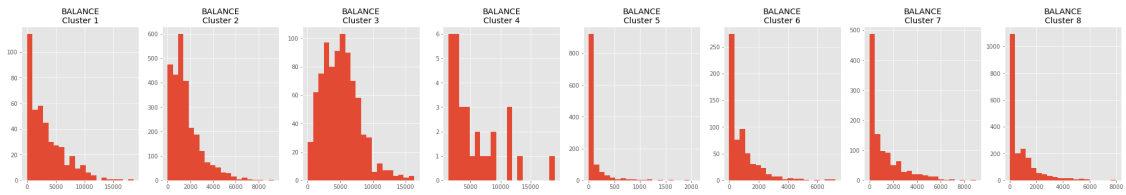
[20]:
```
creditcard_df_cluster = pd.concat([creditcard_df, pd.DataFrame({'cluster':␣
↪labels})], axis = 1)
creditcard_df_cluster.head()
```

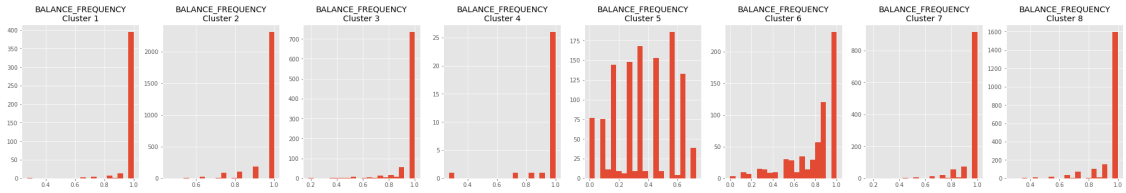
[20]:
```
        BALANCE  BALANCE_FREQUENCY  PURCHASES  ONEOFF_PURCHASES  \
0    40.900749           0.818182      95.40              0.00
1  3202.467416           0.909091       0.00              0.00
2  2495.148862           1.000000     773.17            773.17
3  1666.670542           0.636364    1499.00           1499.00
4   817.714335           1.000000      16.00             16.00

   INSTALLMENTS_PURCHASES  CASH_ADVANCE  PURCHASES_FREQUENCY  \
0                    95.4      0.000000             0.166667
1                     0.0   6442.945483             0.000000
2                     0.0      0.000000             1.000000
3                     0.0    205.788017             0.083333
4                     0.0      0.000000             0.083333

   ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
0                    0.000000                          0.083333
```

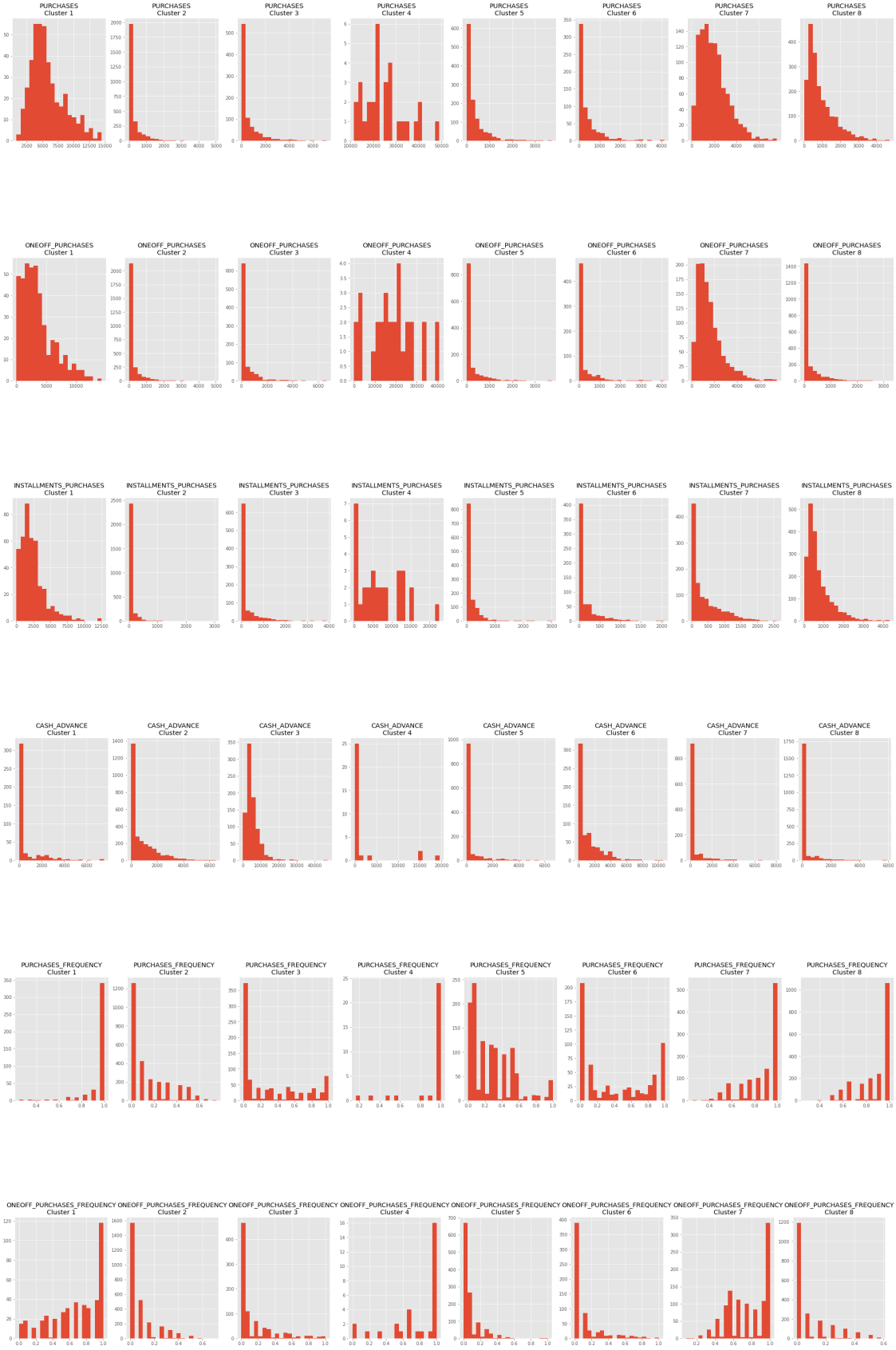
```
1                    0.000000                             0.000000
2                    1.000000                             0.000000
3                    0.083333                             0.000000
4                    0.083333                             0.000000

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PURCHASES_TRX  CREDIT_LIMIT  \
0                0.000000                 0              2        1000.0
1                0.250000                 4              0        7000.0
2                0.000000                 0             12        7500.0
3                0.083333                 1              1        7500.0
4                0.000000                 0              1        1200.0

      PAYMENTS  MINIMUM_PAYMENTS  PRC_FULL_PAYMENT  TENURE  cluster
0   201.802084        139.509787          0.000000      12        1
1  4103.032597       1072.340217          0.222222      12        2
2   622.066742        627.284787          0.000000      12        6
3     0.000000        864.206542          0.000000      12        1
4   678.334763        244.791237          0.000000      12        1
```

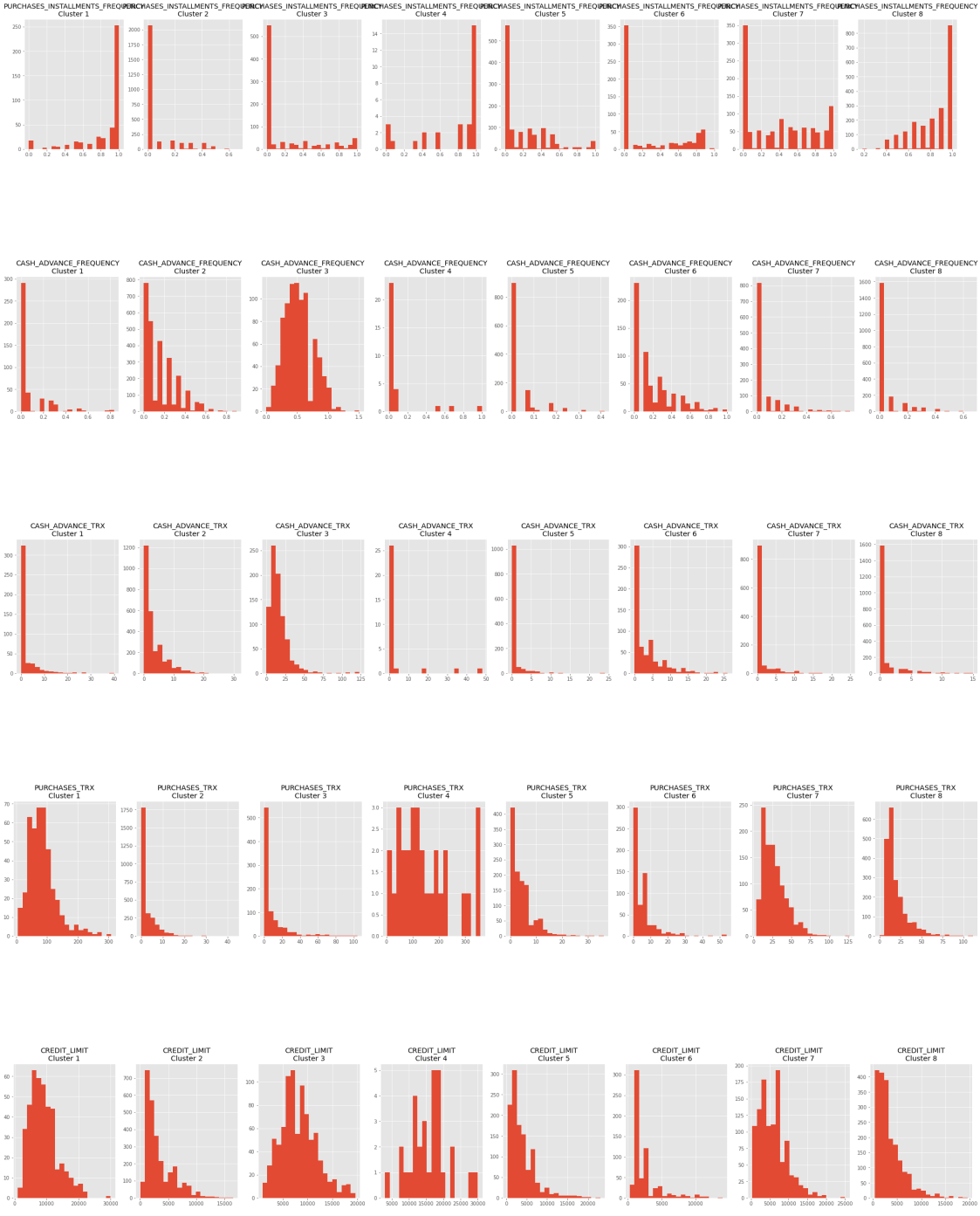
Visualizando el histograma de cada variables respecto a los clústers.

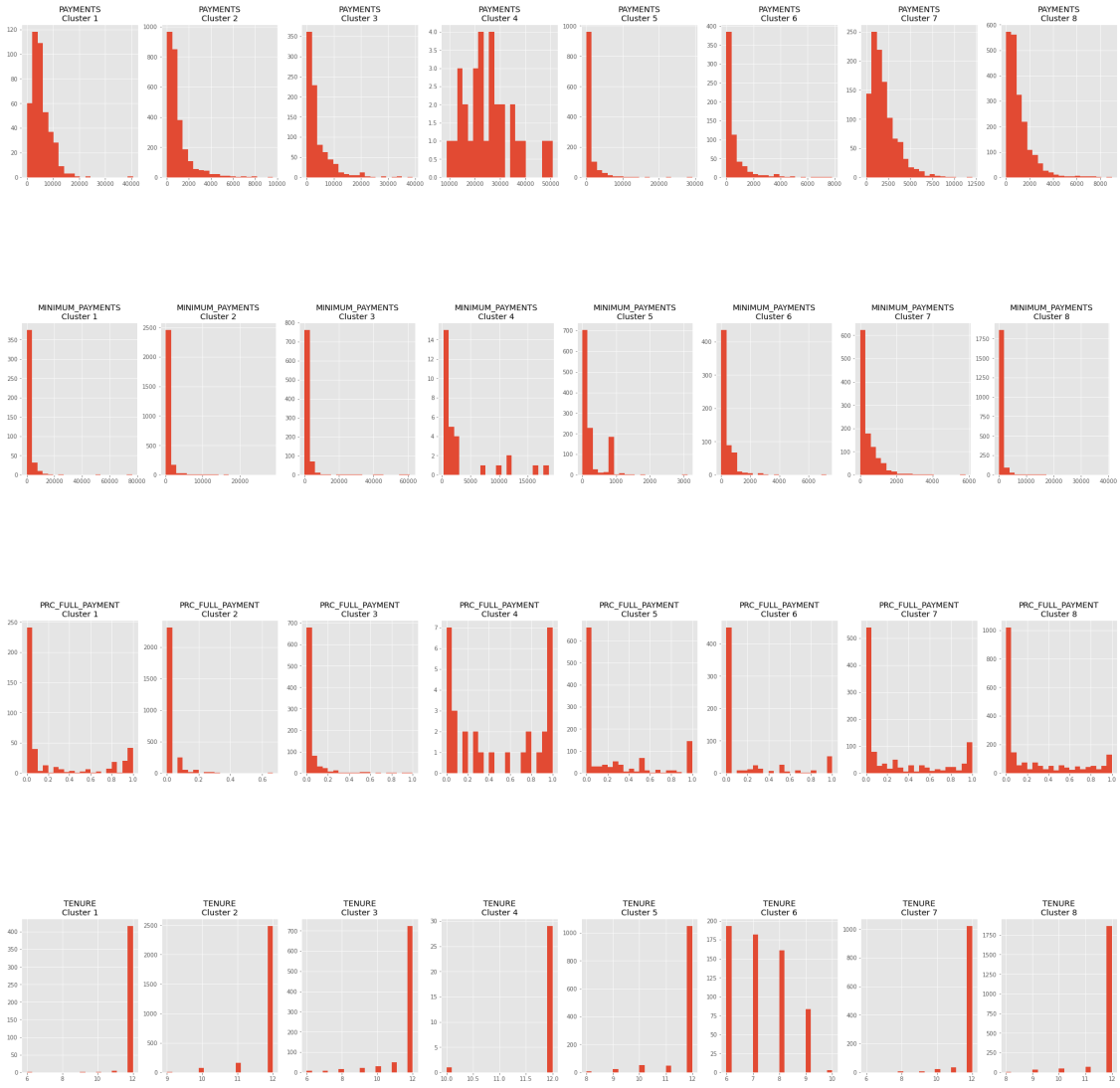
[22]:
```
for i in creditcard_df.columns:
  plt.figure(figsize=(35,5))
  for j in range(8):
    plt.subplot(1,8,j+1)
    cluster = creditcard_df_cluster[creditcard_df_cluster['cluster'] == j]
    cluster[i].hist(bins = 20)
    plt.title('{}\nCluster {}'.format(i, j+1))
  plt.show()
```






PURCHASES Cluster 1
PURCHASES Cluster 2
PURCHASES Cluster 3
PURCHASES Cluster 4
PURCHASES Cluster 5
PURCHASES Cluster 6
PURCHASES Cluster 7
PURCHASES Cluster 8
ONEOFF_PURCHASES Cluster 1
ONEOFF_PURCHASES Cluster 2
ONEOFF_PURCHASES Cluster 3
ONEOFF_PURCHASES Cluster 4
ONEOFF_PURCHASES Cluster 5
ONEOFF_PURCHASES Cluster 6
ONEOFF_PURCHASES Cluster 7
ONEOFF_PURCHASES Cluster 8
INSTALLMENTS_PURCHASES Cluster 1
INSTALLMENTS_PURCHASES Cluster 2
INSTALLMENTS_PURCHASES Cluster 3
INSTALLMENTS_PURCHASES Cluster 4
INSTALLMENTS_PURCHASES Cluster 5
INSTALLMENTS_PURCHASES Cluster 6
INSTALLMENTS_PURCHASES Cluster 7
INSTALLMENTS_PURCHASES Cluster 8
CASH_ADVANCE Cluster 1
CASH_ADVANCE Cluster 2
CASH_ADVANCE Cluster 3
CASH_ADVANCE Cluster 4
CASH_ADVANCE Cluster 5
CASH_ADVANCE Cluster 6
CASH_ADVANCE Cluster 7
CASH_ADVANCE Cluster 8
PURCHASES_FREQUENCY Cluster 1
PURCHASES_FREQUENCY Cluster 2
PURCHASES_FREQUENCY Cluster 3
PURCHASES_FREQUENCY Cluster 4
PURCHASES_FREQUENCY Cluster 5
PURCHASES_FREQUENCY Cluster 6
PURCHASES_FREQUENCY Cluster 7
PURCHASES_FREQUENCY Cluster 8
ONEOFF_PURCHASES_FREQUENCY Cluster 1
ONEOFF_PURCHASES_FREQUENCY Cluster 2
ONEOFF_PURCHASES_FREQUENCY Cluster 3
ONEOFF_PURCHASES_FREQUENCY Cluster 4
ONEOFF_PURCHASES_FREQUENCY Cluster 5
ONEOFF_PURCHASES_FREQUENCY Cluster 6
ONEOFF_PURCHASES_FREQUENCY Cluster 7
ONEOFF_PURCHASES_FREQUENCY Cluster 8


PURCHASES_INSTALLMENTS_FREQUENCY Cluster 1
PURCHASES_INSTALLMENTS_FREQUENCY Cluster 2
PURCHASES_INSTALLMENTS_FREQUENCY Cluster 3
PURCHASES_INSTALLMENTS_FREQUENCY Cluster 4
PURCHASES_INSTALLMENTS_FREQUENCY Cluster 5
PURCHASES_INSTALLMENTS_FREQUENCY Cluster 6
PURCHASES_INSTALLMENTS_FREQUENCY Cluster 7
PURCHASES_INSTALLMENTS_FREQUENCY Cluster 8
CASH_ADVANCE_FREQUENCY Cluster 1
CASH_ADVANCE_FREQUENCY Cluster 2
CASH_ADVANCE_FREQUENCY Cluster 3
CASH_ADVANCE_FREQUENCY Cluster 4
CASH_ADVANCE_FREQUENCY Cluster 5
CASH_ADVANCE_FREQUENCY Cluster 6
CASH_ADVANCE_FREQUENCY Cluster 7
CASH_ADVANCE_FREQUENCY Cluster 8
CASH_ADVANCE_TRX Cluster 1
CASH_ADVANCE_TRX Cluster 2
CASH_ADVANCE_TRX Cluster 3
CASH_ADVANCE_TRX Cluster 4
CASH_ADVANCE_TRX Cluster 5
CASH_ADVANCE_TRX Cluster 6
CASH_ADVANCE_TRX Cluster 7
CASH_ADVANCE_TRX Cluster 8
PURCHASES_TRX Cluster 1
PURCHASES_TRX Cluster 2
PURCHASES_TRX Cluster 3
PURCHASES_TRX Cluster 4
PURCHASES_TRX Cluster 5
PURCHASES_TRX Cluster 6
PURCHASES_TRX Cluster 7
PURCHASES_TRX Cluster 8
CREDIT_LIMIT Cluster 1
CREDIT_LIMIT Cluster 2
CREDIT_LIMIT Cluster 3
CREDIT_LIMIT Cluster 4
CREDIT_LIMIT Cluster 5
CREDIT_LIMIT Cluster 6
CREDIT_LIMIT Cluster 7
CREDIT_LIMIT Cluster 8



En la visualización anterior podemos observar cómo se comporta cada variable respecto al clúster, así como las frecuencias que se le asigna a cada clúster. por ejemplo:

- Los clientes con mayor saldo con catalogados en los clúster 3 y 4.
- Los clientes que hacen compras altas en una sola exibición se encuentran en el clúster 4.
- Los clientes que menos actualizan su saldo están en el clúster 5.
- Los clientes de los clúster 3 y 6 son los que piden más efectivo por adelantado al banco.

4.2 Análisis de Componentes Principales

Analizar un conjunto de datos donde se involucran muchas variables, en este caso de estudio 17, puede llegar a ser complicado.

El ACP o PCA en inglés, nos ayuda a reducir la dimensionalidad del problema, esto reduciendo el número de variables, tal que, se pierda la menor varianza posible de los datos y así perder la menos información posible.

El objetivo de este caso de estudio es reducir a 2 componentes principales, esto con el fin de poder visualizar a los usuarios en un gráfico de dispersión y que sea más sencillo analizarlo.

[23]:
```
pca = PCA(n_components=2)
principal_comp = pca.fit_transform(creditcard_df_scaled)
```

[24]:
```
pca_df = pd.DataFrame(data=principal_comp, columns=['pca_1', 'pca_2'])
pca_df.head()
```

[24]:
```
      pca_1     pca_2
0 -1.682222 -1.076444
1 -1.138299  2.506500
2  0.969687 -0.383521
3 -0.873628  0.043176
4 -1.599436 -0.688578
```

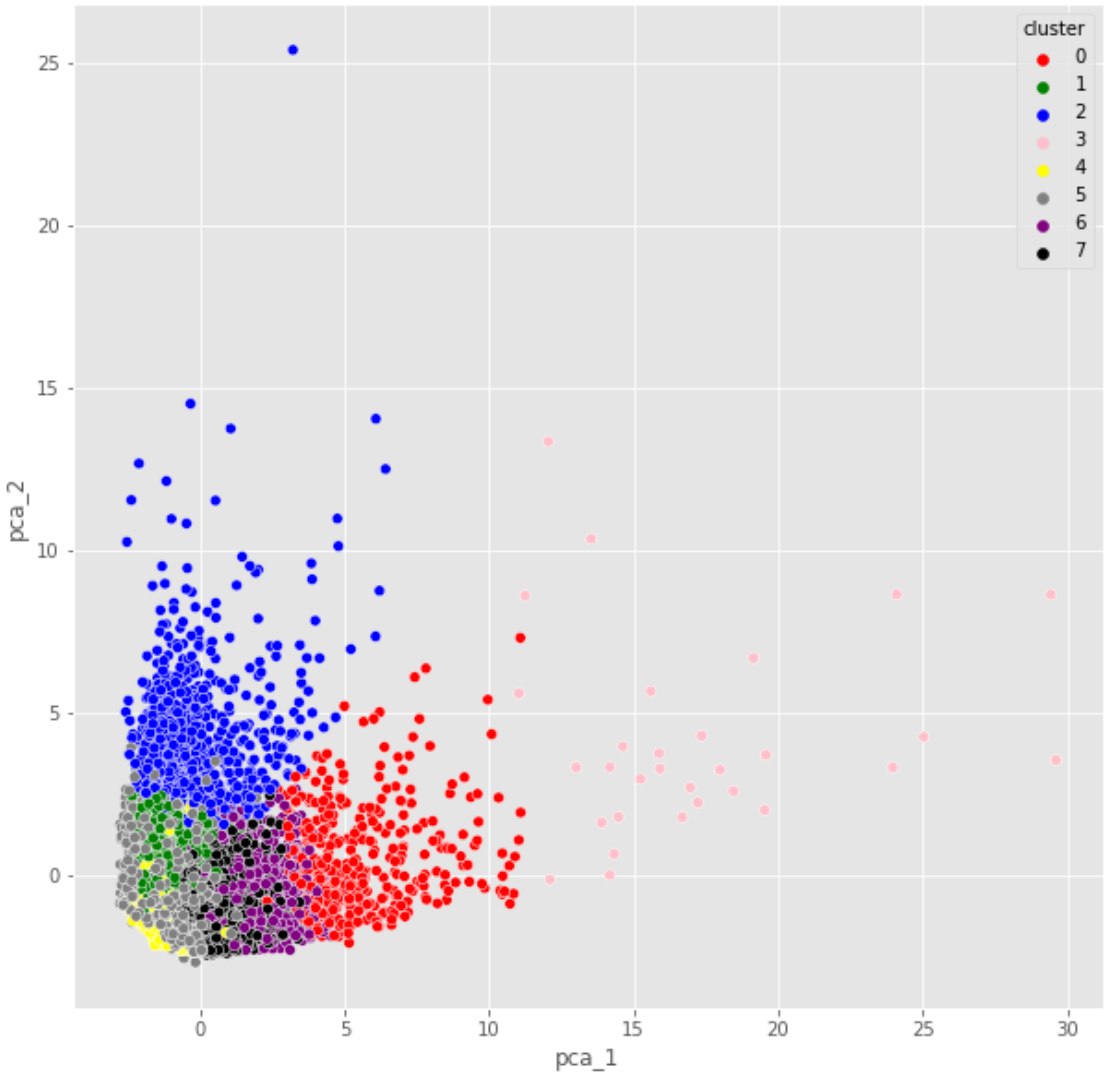
Ya tenemos los datos proyectados a solo 2 dimensiones, ahora concatenaremos los cluster a los que pertenece cada cliente.

[25]:
```
pca_df = pd.concat([pca_df, pd.DataFrame({'cluster': labels})], axis = 1)
pca_df.head()
```

[25]:
```
      pca_1     pca_2  cluster
0 -1.682222 -1.076444        1
1 -1.138299  2.506500        2
2  0.969687 -0.383521        6
3 -0.873628  0.043176        1
4 -1.599436 -0.688578        1
```

Con un dataframe de solo dos características podemos visualizar más claro el resultado del algoritmo de K-Means.

[26]:
```
plt.figure(figsize=(10,10))
ax = sns.scatterplot(x='pca_1', y='pca_2', hue='cluster', data=pca_df,
                     palette=['r', 'g', 'b', 'pink', 'yellow', 'gray',␣
 ↪'purple', 'k'])

plt.show()
```



Como podemos observar, ahora podemos visualizar como el algoritmo K-Means ha clusterizado los datos.

El siguiente paso es llevar a la práctica esta información e ir actualizando el número k de clústers respecto al comportamiento del mercado.

El algoritmo de K-Means, al ser no supervisado, resulta difícil predecir con una exactitud elevada el número correcto de clústers, el método del codo es útil y ayuda a dar una primera impresión, pero será el comportamiento del algoritmo en la práctica el que ayude a mantener el código y hacer los cambios pertinentes.

4.3 Autoencoders

Analizar una cantidad grande de variables para entender lo que se está estudiando puede resultar en una tarea difícil.

Nos ayudaremos de los autoencoders para reducir la dimensionalidad del problema y así hacer sencillo analizar las variables.

[27]:
```
from tensorflow.keras.layers import Input, Add, Dense, Activation,␣
↪ZeroPadding2D, BatchNormalization, Flatten, Conv2D, AveragePooling2D,␣
↪MaxPooling2D, Dropout
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.initializers import glorot_uniform
from tensorflow.keras.optimizers import SGD
```

4.3.1 Arquitectura del Autoencoder

Para este caso de estudio partiremos de 17 variables (Las 17 columnas originales del dataset) y las comprimiremos para obtener 10.

[32]:
```
input_df = Input(shape = (17,))
encoding_dim = 7

# Hacemos una primera reducción a 7 variables
x = Dense(encoding_dim, activation='relu')(input_df)
# Trabajamos con esa reducción
x = Dense(500, activation='relu', kernel_initializer='glorot_uniform')(x)
x = Dense(500, activation='relu', kernel_initializer='glorot_uniform')(x)
x = Dense(2000, activation='relu', kernel_initializer='glorot_uniform')(x)

# Aumentamos a 10 variables
encoded = Dense(10, activation='relu', kernel_initializer='glorot_uniform')(x)

# Trabajamos con esas 10 variables de modo inverso, sesgando a no tener una␣
↪capa de 500 neuronas
x = Dense(2000, activation='relu', kernel_initializer='glorot_uniform')(encoded)
x = Dense(500, activation='relu', kernel_initializer='glorot_uniform')(x)

# Decodificamos para obtener de nuevo las 17 variables
decoded = Dense(17, kernel_initializer='glorot_uniform')(x)

# Modelo para codificar y decodificar
autoencoder = Model(input_df, decoded)
# Modelo sólo para codificar
encoder = Model(input_df, encoded)

autoencoder.compile(optimizer='adam', loss='mean_squared_error')
```

[33]:
```
# Un resumen de como el modelo compilar la información
autoencoder.summary()
```

```
Model: "model_2"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 17)]              0
_________________________________________________________________
dense_8 (Dense)              (None, 7)                 126
_________________________________________________________________
dense_9 (Dense)              (None, 500)               4000
_________________________________________________________________
dense_10 (Dense)             (None, 500)               250500
_________________________________________________________________
dense_11 (Dense)             (None, 2000)              1002000
_________________________________________________________________
dense_12 (Dense)             (None, 10)                20010
_________________________________________________________________
dense_13 (Dense)             (None, 2000)              22000
_________________________________________________________________
dense_14 (Dense)             (None, 500)               1000500
_________________________________________________________________
dense_15 (Dense)             (None, 17)                8517
=================================================================
Total params: 2,307,653
Trainable params: 2,307,653
Non-trainable params: 0
_________________________________________________________________
```

[41]:
```
# Entrenamiento
hist = autoencoder.fit(creditcard_df_scaled, creditcard_df_scaled,␣
 ↪batch_size=64, epochs=15,
                verbose=1)
```

```
Epoch 1/15
140/140 [==============================] - 10s 69ms/step - loss: 0.0588
Epoch 2/15
140/140 [==============================] - 13s 93ms/step - loss: 0.0566
Epoch 3/15
140/140 [==============================] - 8s 60ms/step - loss: 0.0501
Epoch 4/15
140/140 [==============================] - 8s 60ms/step - loss: 0.0466
Epoch 5/15
140/140 [==============================] - 6s 46ms/step - loss: 0.0447
Epoch 6/15
140/140 [==============================] - 14s 102ms/step - loss: 0.0635
Epoch 7/15
140/140 [==============================] - 13s 90ms/step - loss: 0.0523
Epoch 8/15
140/140 [==============================] - 10s 69ms/step - loss: 0.0414
Epoch 9/15
140/140 [==============================] - 13s 92ms/step - loss: 0.0369
```

```
Epoch 10/15
140/140 [==============================] - 12s 88ms/step - loss: 0.0336
Epoch 11/15
140/140 [==============================] - 8s 59ms/step - loss: 0.0342
Epoch 12/15
140/140 [==============================] - 12s 87ms/step - loss: 0.0402
Epoch 13/15
140/140 [==============================] - 10s 75ms/step - loss: 0.0462
Epoch 14/15
140/140 [==============================] - 11s 78ms/step - loss: 0.0316
Epoch 15/15
140/140 [==============================] - 12s 84ms/step - loss: 0.0293
```

[42]:
```
# Guardar pesos del autoencoder entrenado
autoencoder.save_weights('autoencoder.h5')
```

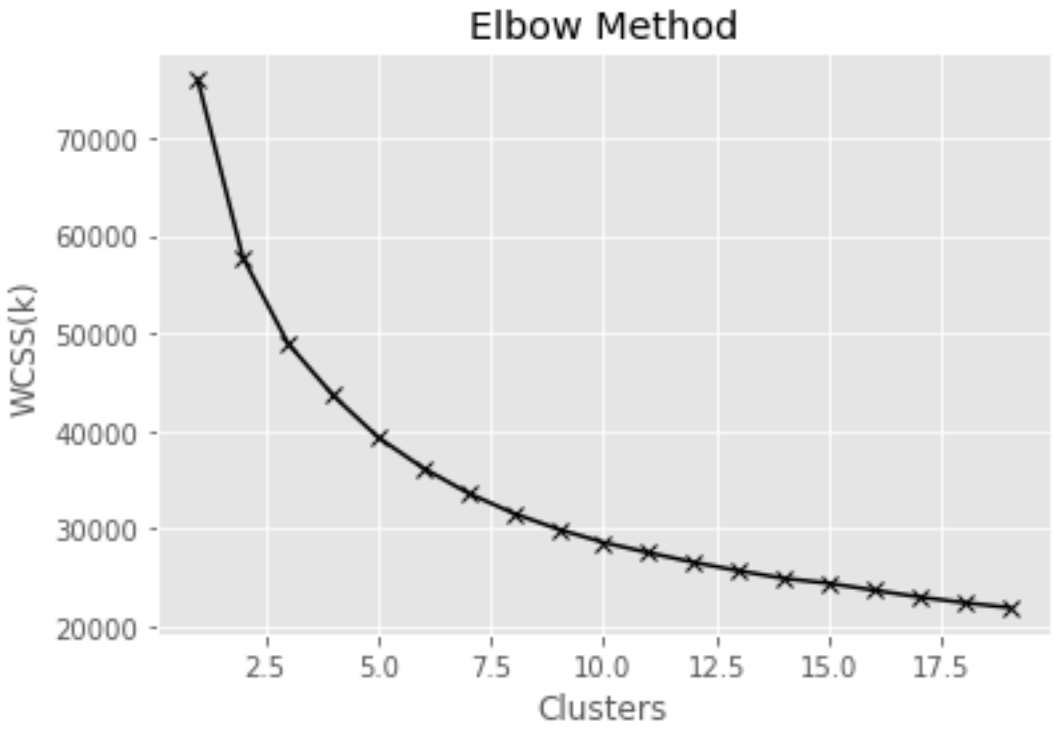
Ahora que tenemos el codificador automático que entrenamos, es hora de reducir las variables de 17 a 10.

Llamaremos a este nuevo grupo de datos pred.

[43]:
```
pred = encoder.predict(creditcard_df_scaled)
```

Volvemos ahora al algoritmo K-Means, esto para ahora aplicarlo a la reducción de variables.

[44]:
```
# Encontrar K óptimo
scores_1 = []
range_values = range(1,20)

for i in range_values:
  kmeans = KMeans(n_clusters=i)
  kmeans.fit(pred)
  scores_1.append(kmeans.inertia_)

# Visualizar para elegir el K óptimo
plt.plot(range_values, scores_1, 'kx-')
plt.title('Elbow Method')
plt.xlabel('Clusters')
plt.ylabel('WCSS(k)')
plt.show()
```



Ahora que ya tenemos el autoencoder entrenamos, es momento de reducir las variables de 17 a 10.

A este nuevo grupo de datos lo llamaremos pred.

[45]:
```
kmeans = KMeans(5)
kmeans.fit(pred)
labels = kmeans.labels_
y_kmeans = kmeans.fit_predict(pred)

df_cluster_ae = pd.concat([creditcard_df, pd.DataFrame({'cluster': labels})],␣
 ↪axis=1)
df_cluster_ae.head()
```

[45]:

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

	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	\
0	95.4	0.000000	0.166667	
1	0.0	6442.945483	0.000000	
2	0.0	0.000000	1.000000	

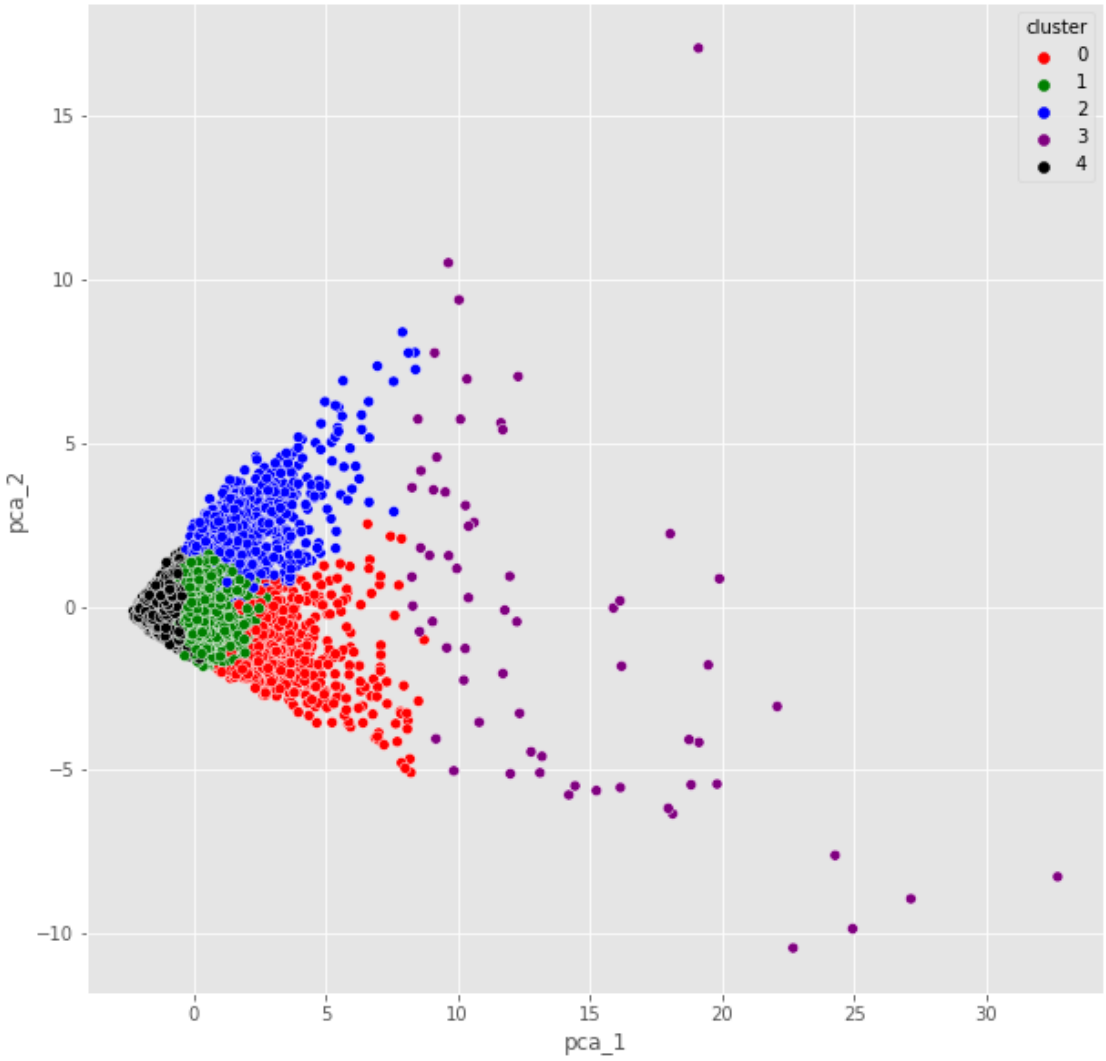
```
3                     0.0    205.788017             0.083333
4                     0.0      0.000000             0.083333

   ONEOFF_PURCHASES_FREQUENCY  PURCHASES_INSTALLMENTS_FREQUENCY  \
0                    0.000000                          0.083333
1                    0.000000                          0.000000
2                    1.000000                          0.000000
3                    0.083333                          0.000000
4                    0.083333                          0.000000

   CASH_ADVANCE_FREQUENCY  CASH_ADVANCE_TRX  PURCHASES_TRX  CREDIT_LIMIT  \
0                0.000000                 0              2        1000.0
1                0.250000                 4              0        7000.0
2                0.000000                 0             12        7500.0
3                0.083333                 1              1        7500.0
4                0.000000                 0              1        1200.0

      PAYMENTS  MINIMUM_PAYMENTS  PRC_FULL_PAYMENT  TENURE  cluster
0   201.802084        139.509787          0.000000      12        4
1  4103.032597       1072.340217          0.222222      12        2
2   622.066742        627.284787          0.000000      12        4
3     0.000000        864.206542          0.000000      12        1
4   678.334763        244.791237          0.000000      12        4
```

Para visualizar estos nuevos cluster aplicamos de nuevo PCA a 2 dimensiones.

[46]:
```
pca = PCA(n_components=2)
pca_comp2 = pca.fit_transform(pred)
pca_df = pd.DataFrame(data=pca_comp2, columns=['pca_1', 'pca_2'])

# Concatenamos los cluster calculados con k-means
pca_df = pd.concat([pca_df, pd.DataFrame({'cluster': labels})], axis=1)
pca_df.head()
```

[46]:
```
      pca_1     pca_2  cluster
0 -1.776384 -0.187245        4
1  0.192772  1.718288        2
2 -0.488257 -1.082752        4
3 -0.229187 -0.191664        1
4 -2.063338 -0.241279        4
```

[47]:
```
plt.figure(figsize=(10,10))
ax = sns.scatterplot(x='pca_1', y='pca_2', hue='cluster', data=pca_df,
                     palette=['r', 'g', 'b', 'purple', 'k'])

plt.show()
```



La visualización nos deja claro cómo se segmentan los datos con 5 clústeres.

Los siguientes pasos son desplegar el algoritmo en producción, observar el comportamiento e ir mejorándolo conforme a este. Recordemos que no existe una fórmula exacta para calcular los clústeres óptimos, por lo que queda es ir mejorando conforme los clientes van respondiendo a las diferentes campañas.

[ ]: