Data Source: https://www.kaggle.com/arjunbhasin2013/ccdata

IMPORTAR LAS LIBRERÍAS Y LOS DATASETS

In [3]:
```
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
import warnings
warnings.filterwarnings("ignore")
```

Carga Dataset

In [4]:
```
creditcard_df = pd.read_csv('A:/a_Google Drive/01_Oriol/02_Formación IA/05_GitHub_Oriol-Marco/04_Proyectos_Mach

# CUSTID: Identificación del titular de la tarjeta de crédito
# BALANCE: Cantidad de saldo que queda en la cuenta del cliente para hacer compras
# BALANCE_FREQUENCY: Frecuencia de la actualización del saldo, puntuación entre 0 y 1 (1 = actualizado con frec
# PURCHASES: Cantidad de compras realizadas desde la cuenta
# ONEOFFPURCHASES: Importe máximo de compra realizado en una sola vez
# INSTALLMENTS_PURCHASES: Importe de la compra realizada en cuotas
# CASH_ADVANCE: Anticipo otorgado al usuario
# PURCHASES_FREQUENCY: Frecuencia con la que se realizan las compras, puntuación entre 0 y 1 (1 = compras frec
# ONEOFF_PURCHASES_FREQUENCY: Con qué frecuencia las compras se están realizando, puntuación entre 0 y 1 (1 = compra con f
# PURCHASES_INSTALLMENTS_FREQUENCY: Con qué frecuencia se realizan las compras a plazos (1 = se realizan con f
# CASH_ADVANCE_FREQUENCY: Con qué frecuencia se paga por adelantado
# CASH_ADVANCE_TRX: número de transacciones realizadas con "Efectivo por adelantado"
# PURCHASES_TRX: número de transacciones de compras realizadas
# CREDIT_LIMIT: límite de tarjeta de crédito para el usuario
# PAYMENTS: Número de pagos realizados por el usuario
# MINIMUM_PAYMENTS: cantidad mínima de pagos realizados por el usuario
# PRC_FULL_PAYMENT: porcentaje del pago total pagado por el usuario
# TENURE: Años que el usuario lleva usando el servicio de tarjeta de crédito
```

In [5]:
```
creditcard_df.head()
```

Out[5]:

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE
0	C10001	40.900749	0.818182	95.40	0.00	95.4	0.000000
1	C10002	3202.467416	0.909091	0.00	0.00	0.0	6442.945483
2	C10003	2495.148862	1.000000	773.17	773.17	0.0	0.000000
3	C10004	1666.670542	0.636364	1499.00	1499.00	0.0	205.788017
4	C10005	817.714335	1.000000	16.00	16.00	0.0	0.000000

In [6]:
```
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
```

In [7]:
```
creditcard_df.describe()

# 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 de ONEOFF_PURCHASES_FREQUENCY, PURCHASES_INSTALLMENTS_FREQUENCY, y CASH_ADVANCE_FREQUENCY es en g
# El promedio del límite de crédito es ~ 4500
# El porcentaje de pago completo es 15%
# Los clientes llevan de promedio en el servicio 11 años
```

Out[7]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE
count	8950.000000	8950.000000	8950.000000	8950.000000	8950.000000	8950.000000
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

In [7]:
```
# De investiga quien ha hecho la mayor compra de $40761!

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

# Es un tipo de cliente que no necesita publicidad de una nueva targeta que le puedan adelantar el dinero.
```

Out[7]:

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE
550	C10574	11547.52001	1.0	49039.57	40761.25	8278.32	558.166886

In [8]:
```
# De investiga quien ha solicitado el mayor adelanto de dinero al banco

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

Out[8]: 47137.211760000006

In [9]:
```
# Vamos a ver quien pago por anticipado $47137!
# Este cliente hizo un total de 123 transacciones por adelantado!!
# Nunca pagaba sus compras completamente con la tarjeta

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

Out[9]:

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE
2159	C12226	10905.05381	1.0	431.93	133.5	298.43	47137.21176

VISUALIZACIÓN DEL DATASET

In [10]:
```
# Comprobamos si tenemos datos faltantes

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

# Se observa que hay algunos datos faltantes en la columna de Minimum Payments.
```

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd883441d0>

In [11]:
```
creditcard_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                          1
PAYMENTS                              0
MINIMUM_PAYMENTS                    313
PRC_FULL_PAYMENT                      0
TENURE                                0
dtype: int64
```

In [12]:
```
# Se rellenan los datos faltantes con el promedio del campo 'MINIMUM_PAYMENT'

creditcard_df.loc[(creditcard_df['MINIMUM_PAYMENTS'].isnull() == True), 'MINIMUM_PAYMENTS'] = creditcard_df['MIN
```

In [13]:
```
creditcard_df.loc[(creditcard_df['CREDIT_LIMIT'].isnull() == True), 'CREDIT_LIMIT'] = creditcard_df['CREDIT_LIMI
```

In [14]:
```
sns.heatmap(creditcard_df.isnull(), yticklabels=False, cbar=False, cmap="Blues")
```

Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3d85a993e8>

In [15]:
```
# Verificamos si tenemos entradas duplicadas en nuestros datos

creditcard_df.duplicated().sum()
```

Out[15]: 0

In [16]:
```
# Eliminamos el campo Customer ID ya que no sirve para nada

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

In [17]:
```
creditcard_df.head()
```

Out[17]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENC
0	40.900749	0.818182	95.40	0.00	95.4	0.000000	0.16666
1	3202.467416	0.909091	0.00	0.00	0.0	6442.945483	0.00000
2	2495.148862	1.000000	773.17	773.17	0.0	0.000000	1.00000
3	1666.670542	0.636364	1499.00	1499.00	0.0	205.788017	0.08333
4	817.714335	1.000000	16.00	16.00	0.0	0.000000	0.08333

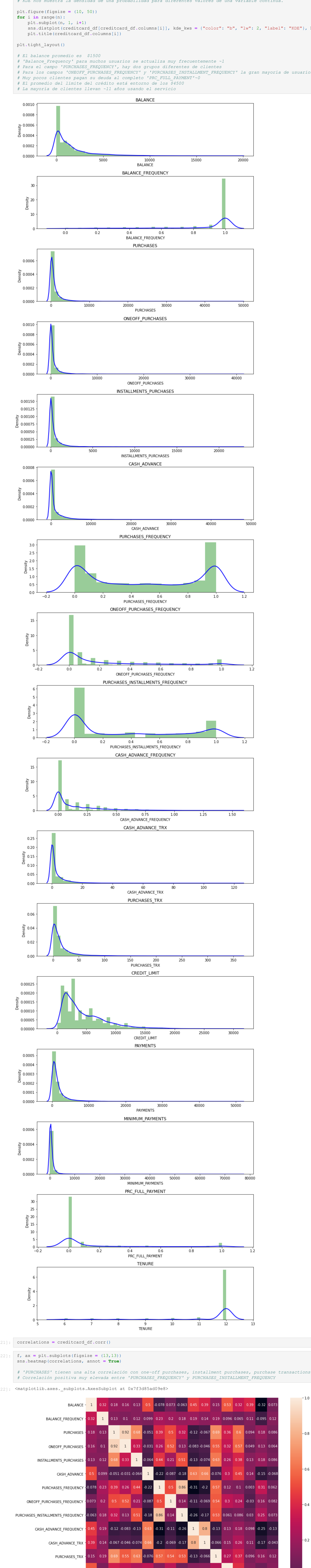
In [18]:
```
n = len(creditcard_df.columns)
n
```

Out[18]: 17

In [19]:
```
creditcard_df.columns
```

Out[19]:
```
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')
```

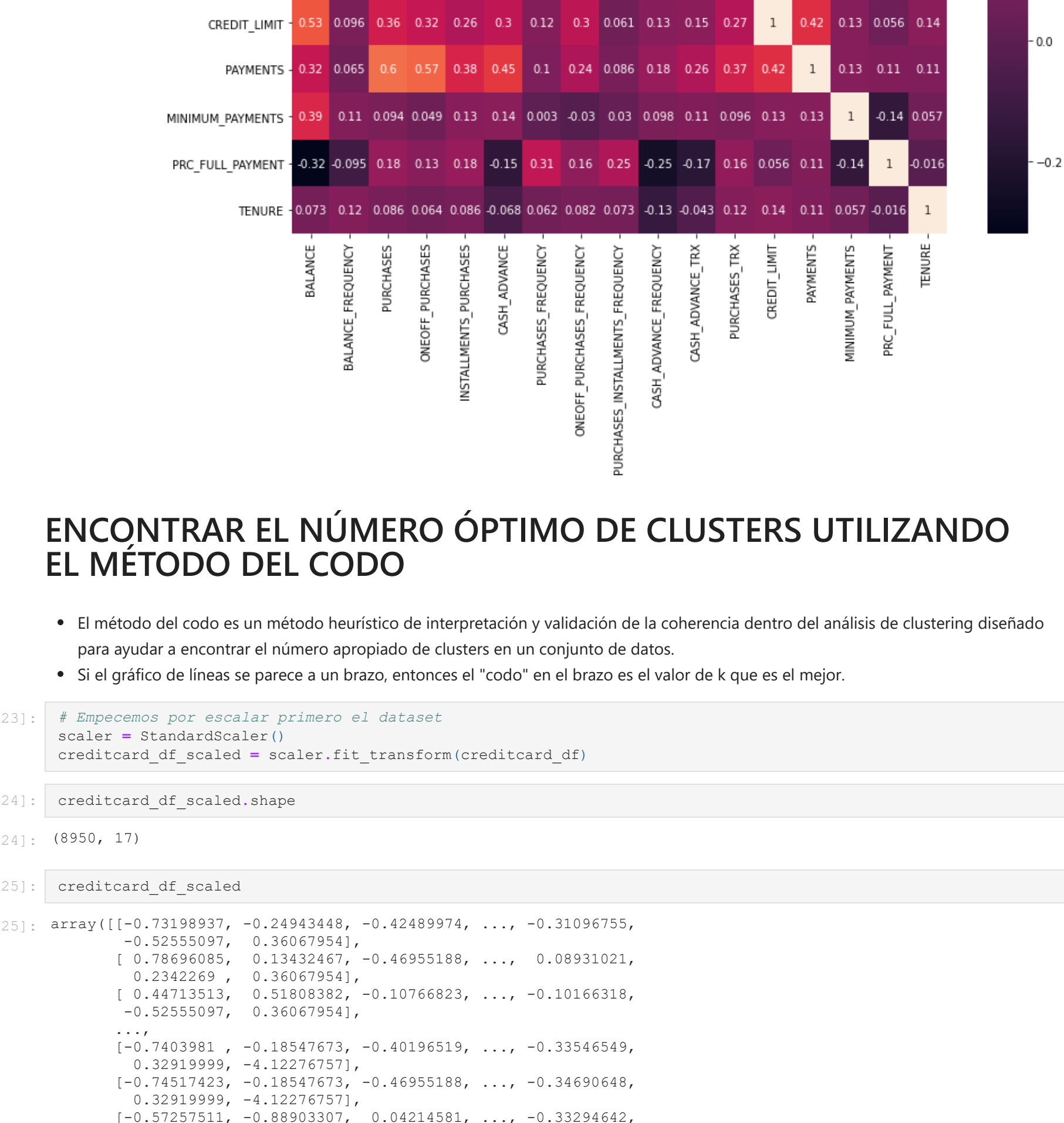
In [20]:
```
# distplot combina la función matplotlib.hist con la de seaborn kdeplot()
# KDE Plot representa la Kernel Density Estimate
# KDE se utiliza para visualizar la densidad de una probabilidad de una variable continua.
# KDE nos muestra la densidad de una probabilidad para diferentes valores de una variable continua.

plt.figure(figsize = (10, 50))
for i in range(n):
    plt.subplot(n, 1, i+1)
    sns.distplot(creditcard_df[creditcard_df.columns[i]], kde_kws = {"color": "b", "lw": 2, "label": "KDE"}, bi
    plt.title(creditcard_df.columns[i])

plt.tight_layout()

# El balance promedio es  $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
# 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
```



In [21]:
```
correlations = creditcard_df.corr()
```

In [22]:
```
f, ax = plt.subplots(figsize = (20,20))
sns.heatmap(correlations, annot = True)

# 'PURCHASES' tienen una alta correlación con one-off purchases, installment purchases, purchase transactions,
# Correlación positiva muy elevada entre 'PURCHASES_FREQUENCY' y 'PURCHASES_INSTALLMENT_FREQUENCY'
```

Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3d85a09e8>



ENCONTRAR EL NÚMERO ÓPTIMO DE CLUSTERS UTILIZANDO EL MÉTODO DEL CODO

- El método del codo es un método heurístico de interpretación y validación de la coherencia dentro del análisis de clustering diseñado para ayudar a encontrar el número apropiado de clusters en un conjunto de datos.
- Si el gráfico de líneas se parece a un brazo, entonces el "codo" en el brazo es el valor de k que es el mejor.

In [23]:
```
# Empezamos por escalar primero el dataset
scaler = StandardScaler()
creditcard_df_scaled = scaler.fit_transform(creditcard_df)
```

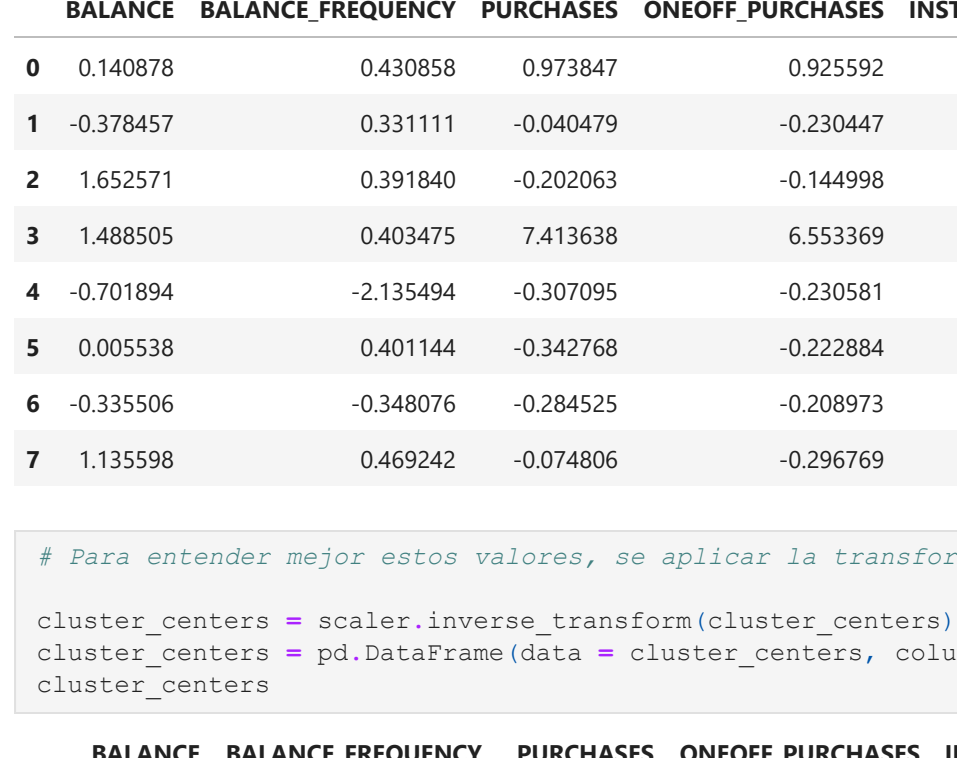
In [24]:
```
creditcard_df_scaled.shape
```

Out[24]: (8950, 17)

In [25]:
```
creditcard_df_scaled
```

Out[25]:
```
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 [26]:
```
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_)

plt.plot(range_values, scores_1, 'bx-')
plt.title("Encontrar el número óptimo de Clusters")
plt.xlabel("Clusters")
plt.ylabel("WCSS(k)")
plt.show()

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



TAREA #6: APLICAR EL MÉTODO DE K-MEANS

In [27]:
```
kmeans = KMeans(8)
kmeans.fit(creditcard_df_scaled)
labels = kmeans.labels_
```

In [28]:
```
kmeans.cluster_centers_.shape
```

Out[28]: (8, 17)

In [29]:
```
cluster_centers = pd.DataFrame(data = kmeans.cluster_centers_, columns=[creditcard_df.columns])

cluster_centers
```

Out[29]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY
0	0.140878	0.430858	0.973847	0.925592	0.602157	-0.307917	1.099450
1	-0.378457	0.331111	-0.040479	-0.230447	0.327492	-0.366848	0.974571
2	1.652571	0.391840	-0.202063	-0.144998	-0.211468	2.014035	-0.448415
3	1.488505	0.403475	7.413638	6.553369	5.486972	0.028557	1.072872
4	-0.701894	-2.135494	-0.307095	-0.230581	-0.302387	-0.322957	-0.547410
5	0.005538	0.401144	-0.342768	-0.222884	-0.400881	-0.103146	-0.812419
6	-0.335506	-0.348076	-0.284525	-0.208973	-0.288475	0.065539	-0.198735
7	1.135598	0.469242	-0.074806	-0.296769	0.367637	-0.042415	-0.071366

In [30]:
```
# Para entender mejor estos valores, se aplicar la transformación inversa.

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

Out[30]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUEN
0	1857.699590	0.979337	3083.844658	2128.731272	955.590334	333.153753	0.9316
1	776.747744	0.955708	916.721809	209.942613	707.214729	209.574097	0.8814
2	5004.162748	0.970094	571.494091	351.770261	219.839364	5202.397488	0.3103
3	4662.671853	0.972850	16842.556892	11469.688108	5372.868784	1038.757441	0.9209
4	103.540821	0.371392	347.092201	209.719739	137.622715	301.615215	0.2706
5	1576.002651	0.972298	270.876081	222.495612	48.556145	762.568330	0.1642
6	866.148306	0.794815	395.311749	245.585564	150.203132	1116.308792	0.4105
7	3928.125451	0.988430	843.379636	99.862000	743.517636	889.924775	0.4617

In [31]:
```
# Primer Cluster de Clientes (Cluster4): Estos son los clientes que pagan la menor cantidad de cargos por inter
# Segundo Cluster de Clientes (Cluster2) que usan tarjeta de crédito como préstamo (sector más lucrativo): sald
# Tercer Cluster de Clientes (Cluster 3): límite de crédito alto $ 13K y porcentaje más alto de pago completo,
# Cuarto Cluster de clientes (Cluster 6): estos son clientes con baja antigüedad (7 años), saldo bajo
```

In [32]:
```
labels.shape
```

Out[32]: (8950,)

In [33]:
```
labels.min()
```

Out[33]: 0

In [34]:
```
labels.max()
```

Out[34]: 7

In [35]:
```
y_kmeans = kmeans.fit_predict(creditcard_df_scaled)

y_kmeans
```

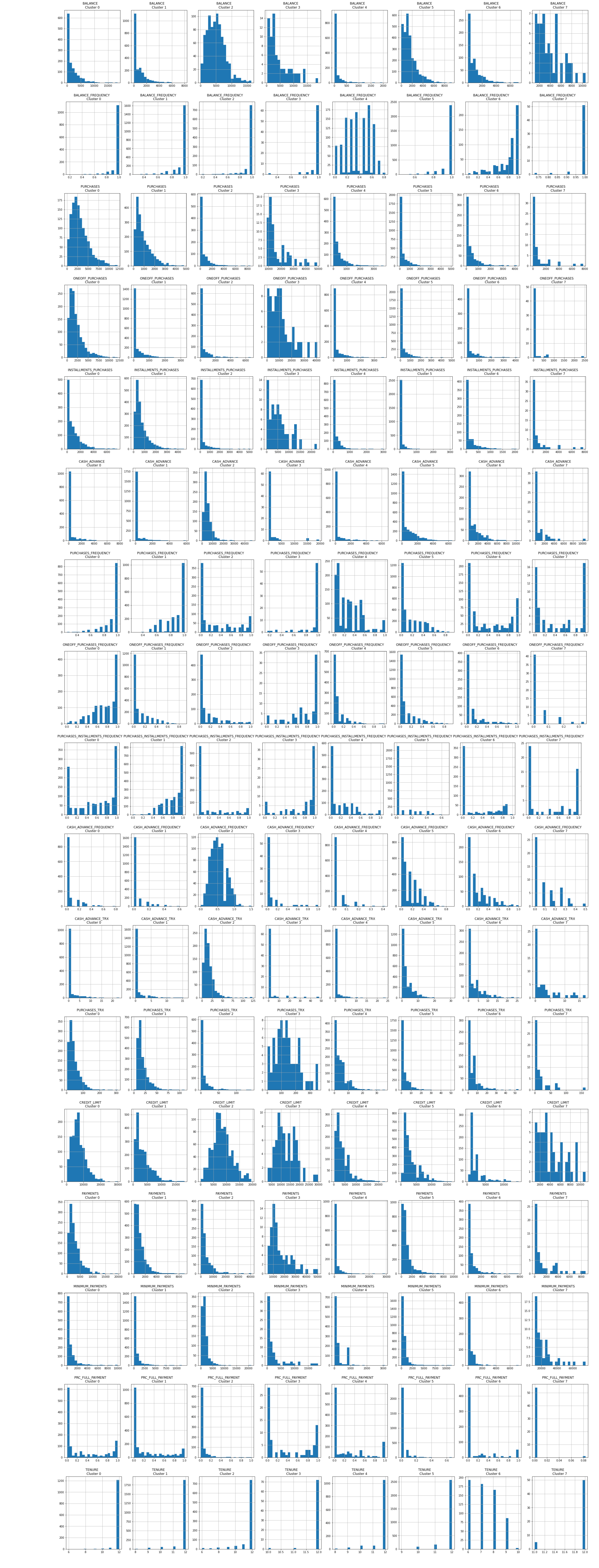
Out[35]: array([3, 0, 1, ..., 2, 2, 2], dtype=int32)

In [36]:
```
# Se concatenan las etiquetas de los clusters con el dataset original

creditcard_df_cluster = pd.concat([creditcard_df, pd.DataFrame({'CLUSTER': labels})], axis = 1)
creditcard_df_cluster.head(10)
```

Out[36]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUEN
0	40.900749	0.818182	95.40	0.00	95.40	0.000000	0.16666
1	3202.467416	0.909091	0.00	0.00	0.00	6442.945483	0.00000
2	2495.148862	1.000000	773.17	773.17	0.00	0.000000	1.00000
3	1666.670542	0.636364	1499.00	1499.00	0.00	205.788017	0.08333
4	817.714335	1.000000	16.00	16.00	0.00	0.000000	0.08333
5	1809.828751	1.000000	1333.28	0.00	1333.28	0.000000	0.66666
6	627.260806	1.000000	7091.01	6402.63	688.38	0.000000	1.00000
7	1823.652743	1.000000	436.20	0.00	436.20	0.000000	1.00000
8	1014.926473	1.000000	861.49	661.49	200.00	0.000000	0.33333
9	152.225975	0.545455	1281.60	1281.60	0.00	0.000000	0.16666

In [37]:
```
# Se visualiza el histogramas para cada cluster

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))
    plt.show()
```



APLICAR ANÁLISIS DE LAS COMPONENTES PRINCIPALES Y VISUALIZAR LOS RESULTADOS

In [38]:
```
# Obtenemos las componentes principales

pca = PCA(n_components = 2)
principal_comp = pca.fit_transform(creditcard_df_scaled)
principal_comp
```

Out[38]:
```
array([[-1.68222173, -1.07644783],
       [-1.13829889,  2.50648306],
       [ 0.96968161, -0.38350836],
       ...,
       [-0.92620373, -1.81078524],
       [-2.33654905, -0.65797389],
       [-0.55641761, -0.40047692]])
```

In [39]:
```
# Se crea un dataframe con las dos componentes principales

pca_df = pd.DataFrame(data = principal_comp, columns=["pca1", "pca2"])
pca_df.head()
```

Out[39]:

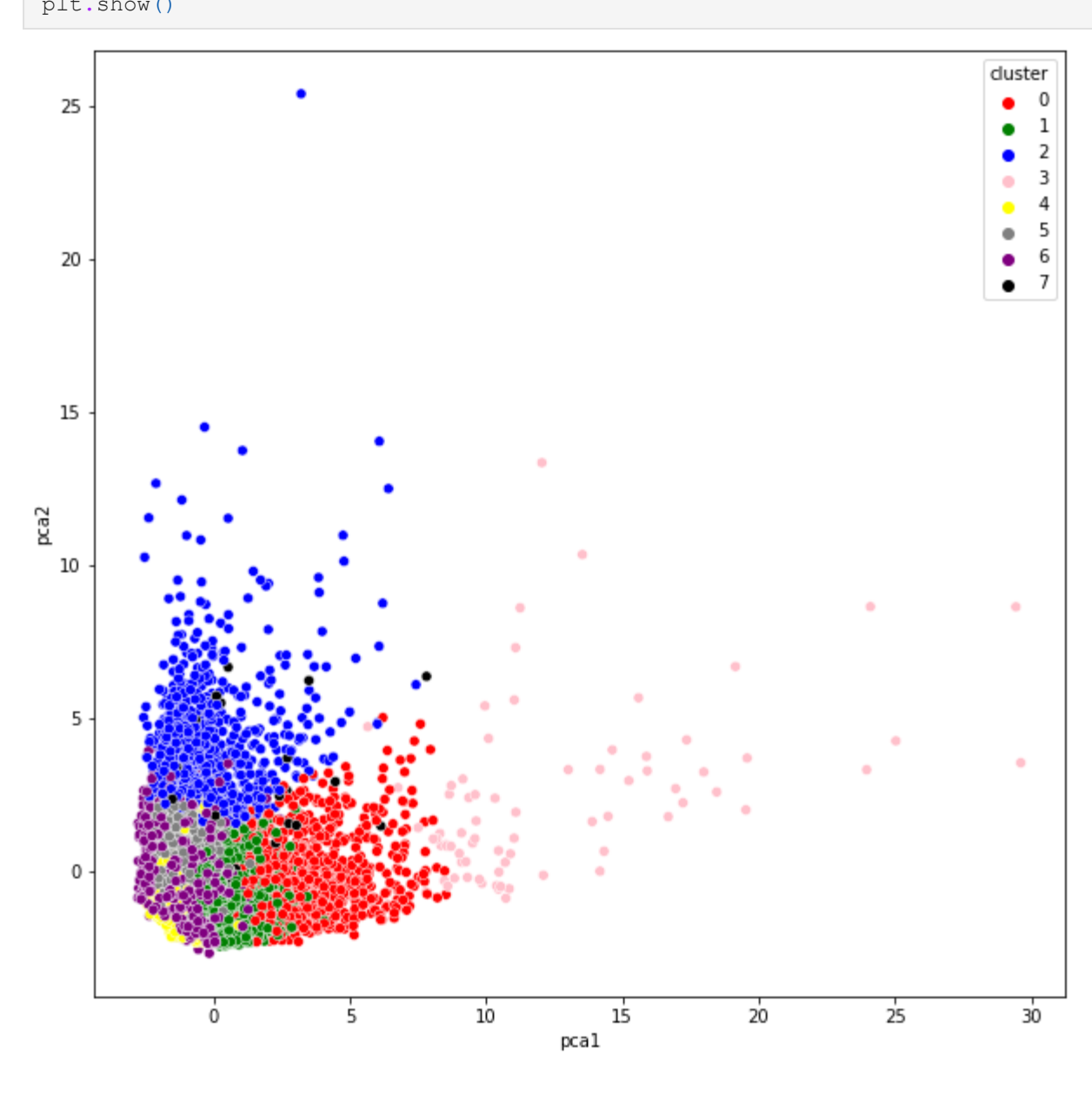
	pca1	pca2
0	-1.682222	-1.076448
1	-1.138299	2.506483
2	0.969682	-0.383508
3	-0.873630	0.043167
4	-1.599435	-0.688577

In [40]:
```
# Concatenamos las etiquetas de los clusters con el dataframe de las componentes principales

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

pca_df.head()
```

Out[40]:

	pca1	pca2	cluster
0	-1.682222	-1.076448	5
1	-1.138299	2.506483	2
2	0.969682	-0.383508	0
3	-0.873630	0.043167	5
4	-1.599435	-0.688577	5

In [41]:
```
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", "black"])
plt.show()
```



APLICAR LOS AUTOENCODERS (REDUCIR LA DIMENSIÓN DE LOS DATOS CON LOS AUTOENCODERS)

In [43]:
```
from tensorflow.keras.layers import Input, Add, Dense, Activation, ZeroPadding2D, BatchNormalization, Flatten,
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.initializers import glorot_uniform
from keras.optimizers import SGD

encoding_dim = 7

input_df = Input(shape = (17, ))

# Glorot normal inicializador (Xavier normal initializer) tomar muestras aleatorias de una distribución normal

x = Dense(encoding_dim, activation = 'relu')(input_df)
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)

encoded = Dense(10, activation = 'relu', kernel_initializer = 'glorot_uniform')(x)

x = Dense(2000, activation = 'relu', kernel_initializer = 'glorot_uniform')(encoded)
x = Dense(500, activation = 'relu', kernel_initializer = 'glorot_uniform')(x)

decoded = Dense(17, kernel_initializer = 'glorot_uniform')(x)


autoencoder = Model(input_df, decoded)

encoder = Model(input_df, encoded)

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

In [44]:
```
creditcard_df_scaled.shape
```

Out[44]:
```
(8950, 17)
```

In [45]:
```
autoencoder.summary()
```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 17)]              0
_________________________________________________________________
dense (Dense)                (None, 7)                 126
_________________________________________________________________
dense_1 (Dense)              (None, 500)               4000
_________________________________________________________________
dense_2 (Dense)              (None, 500)               250500
_________________________________________________________________
dense_3 (Dense)              (None, 2000)              1002000
_________________________________________________________________
dense_4 (Dense)              (None, 10)                20010
_________________________________________________________________
dense_5 (Dense)              (None, 2000)              22000
_________________________________________________________________
dense_6 (Dense)              (None, 500)               1000500
_________________________________________________________________
dense_7 (Dense)              (None, 17)                8517
=================================================================
Total params: 2,307,653
Trainable params: 2,307,653
Non-trainable params: 0
_________________________________________________________________
```

In [46]:
```
autoencoder.fit(creditcard_df_scaled, creditcard_df_scaled, batch_size=128, epochs = 25, verbose = 1)
```

```
Epoch 1/25
70/70 [==============================] - 4s 55ms/step - loss: 0.4904
Epoch 2/25
70/70 [==============================] - 4s 55ms/step - loss: 0.2395
Epoch 3/25
70/70 [==============================] - 4s 55ms/step - loss: 0.1884
Epoch 4/25
70/70 [==============================] - 4s 55ms/step - loss: 0.1666
Epoch 5/25
70/70 [==============================] - 4s 55ms/step - loss: 0.1443
Epoch 6/25
70/70 [==============================] - 4s 55ms/step - loss: 0.1368
Epoch 7/25
70/70 [==============================] - 4s 56ms/step - loss: 0.1225
Epoch 8/25
70/70 [==============================] - 4s 56ms/step - loss: 0.1131
Epoch 9/25
70/70 [==============================] - 4s 55ms/step - loss: 0.1123
Epoch 10/25
70/70 [==============================] - 4s 55ms/step - loss: 0.1046
Epoch 11/25
70/70 [==============================] - 4s 55ms/step - loss: 0.1013
Epoch 12/25
70/70 [==============================] - 4s 55ms/step - loss: 0.0935
Epoch 13/25
70/70 [==============================] - 4s 55ms/step - loss: 0.0920
Epoch 14/25
70/70 [==============================] - 4s 55ms/step - loss: 0.0863
Epoch 15/25
70/70 [==============================] - 4s 57ms/step - loss: 0.0832
Epoch 16/25
70/70 [==============================] - 4s 62ms/step - loss: 0.0819
Epoch 17/25
70/70 [==============================] - 4s 60ms/step - loss: 0.0779
Epoch 18/25
70/70 [==============================] - 4s 59ms/step - loss: 0.0749
Epoch 19/25
70/70 [==============================] - 4s 60ms/step - loss: 0.0743
Epoch 20/25
70/70 [==============================] - 4s 58ms/step - loss: 0.0703
Epoch 21/25
70/70 [==============================] - 4s 56ms/step - loss: 0.0659
Epoch 22/25
70/70 [==============================] - 4s 55ms/step - loss: 0.0619
Epoch 23/25
70/70 [==============================] - 4s 56ms/step - loss: 0.0598
Epoch 24/25
70/70 [==============================] - 4s 55ms/step - loss: 0.0611
Epoch 25/25
70/70 [==============================] - 4s 55ms/step - loss: 0.0572
```

Out[46]:
```
<tensorflow.python.keras.callbacks.History at 0x7f3d4d3865c0>
```

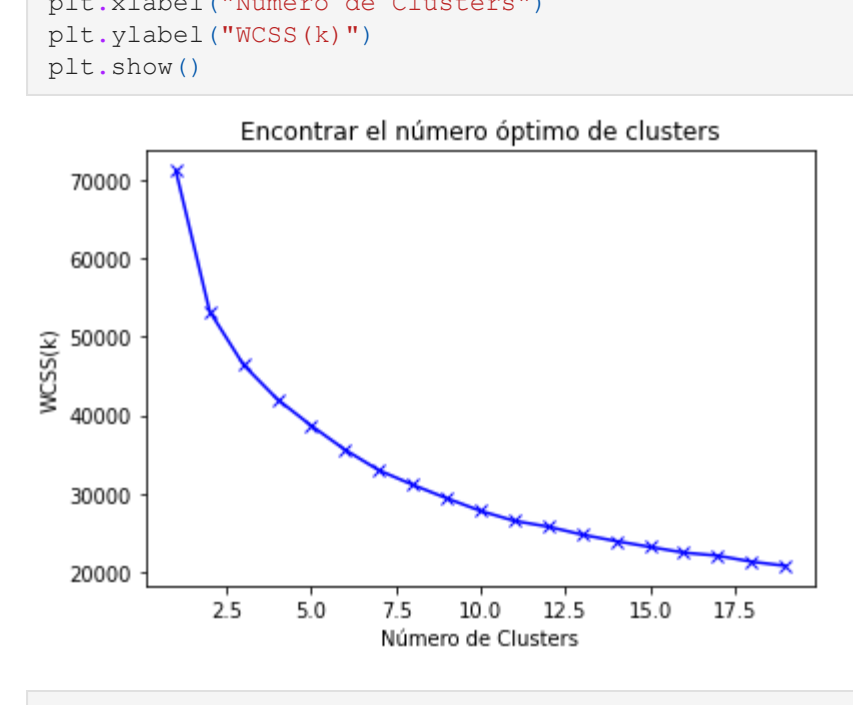
In [47]:
```
autoencoder.save_weights('autoencoder.h5')
```

In [48]:
```
pred = encoder.predict(creditcard_df_scaled)
```

In [49]:
```
pred.shape
```

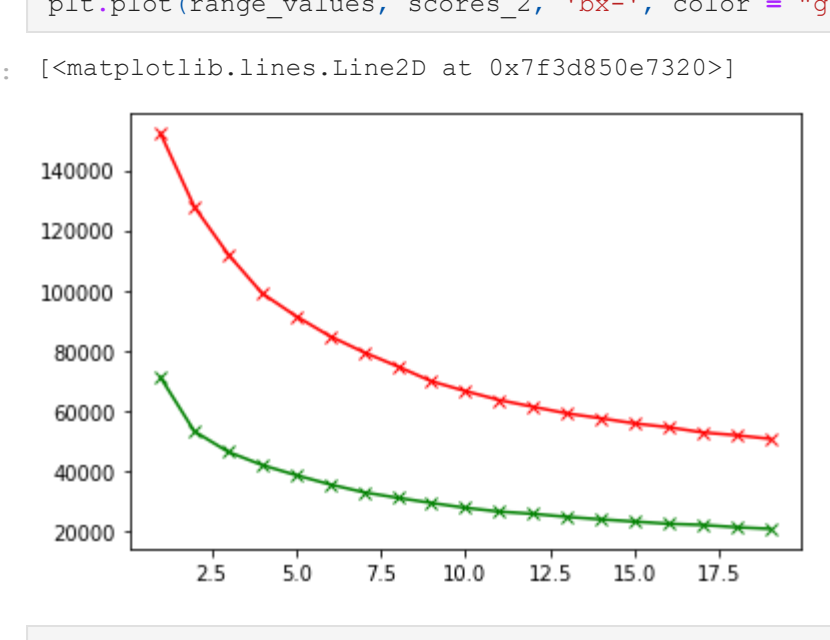
Out[49]:
```
(8950, 10)
```

In [50]:
```
scores_2 = []

range_values = range(1,20)

for i in range_values:
    kmeans = KMeans(n_clusters = i)
    kmeans.fit(pred)
    scores_2.append(kmeans.inertia_)


plt.plot(range_values, scores_2, 'bx-')
plt.title("Encontrar el número óptimo de clusters")
plt.xlabel("Número de Clusters")
plt.ylabel("WCSS(k)")
plt.show()
```



In [51]:
```
plt.plot(range_values, scores_1, 'bx-', color = "r")
plt.plot(range_values, scores_2, 'bx-', color = "g")
```

Out[51]:
```
[<matplotlib.lines.Line2D at 0x7f3d850e7320>]
```



In [52]:
```
kmeans = KMeans(4)
kmeans.fit(pred)
labels = kmeans.labels_
y_kmeans = kmeans.fit_predict(pred)
```

In [53]:
```
df_cluster_dr = pd.concat([creditcard_df, pd.DataFrame({'cluster': labels})], axis = 1)
df_cluster_dr.head()
```

Out[53]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENC
0	40.900749	0.818182	95.40	0.00	95.4	0.000000	0.16666
1	3202.467416	0.909091	0.00	0.00	0.0	6442.945483	0.00000
2	2495.148862	1.000000	773.17	773.17	0.0	0.000000	1.00000
3	1666.670542	0.636364	1499.00	1499.00	0.0	205.788017	0.08333
4	817.714335	1.000000	16.00	16.00	0.0	0.000000	0.08333

In [54]:
```
pca = PCA(n_components=2)
princ_comp = pca.fit_transform(pred)
pca_df = pd.DataFrame(data = princ_comp, columns=["pca1", "pca2"])
pca_df.head()
```

Out[54]:

	pca1	pca2
0	-1.229043	-0.458952
1	1.962612	-1.228787
2	-0.465310	0.961974
3	-0.717040	-0.495402
4	-1.348035	-0.388251

In [55]:
```
pca_df = pd.concat([pca_df, pd.DataFrame({"cluster":labels})], axis = 1)
pca_df
```

Out[55]:

	pca1	pca2	cluster
0	-1.229043	-0.458952	1
1	1.962612	-1.228787	3
2	-0.465310	0.961974	1
3	-0.717040	-0.495402	1
4	-1.348035	-0.388251	1
...	...	...	...
8945	0.208941	-0.741651	0
8946	0.760663	-0.548579	0
8947	0.460611	-1.016577	0
8948	0.951057	-2.003083	0
8949	2.596555	-1.001330	0

8950 rows × 3 columns

In [56]:
```
plt.figure(figsize=(10,10))
ax = sns.scatterplot(x="pca1", y = "pca2", hue="cluster", data = pca_df, palette=["red", "green", "blue", "yell
plt.show()
```