Clustering com dados de Cartão de Crédito.

[1]
```
import pandas as pd

dataframe = pd.read_csv('/content/CC GENERAL.csv')
dataframe.drop(columns=["CUST_ID", "TENURE"], inplace=True)
dataframe.head()
```

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES	CASH_ADVANCE	PURCHASES_FREQUENCY	ONEOFF_PURCHASES_FREQUENCY	PURCHASES_INS
0	40.900749	0.818182	95.40	0.00	95.4	0.000000	0.166667	0.000000	
1	3202.467416	0.909091	0.00	0.00	0.0	6442.945483	0.000000	0.000000	
2	2495.148862	1.000000	773.17	773.17	0.0	0.000000	1.000000	1.000000	
3	1666.670542	0.636364	1499.00	1499.00	0.0	205.788017	0.083333	0.083333	
4	817.714335	1.000000	16.00	16.00	0.0	0.000000	0.083333	0.083333	

[2]
```
dataframe.isna().sum()
```

```
BALANCE                           0
BALANCE_FREQUENCY                 0
PURCHASES                         0
ONEOFF_PURCHASES                  0
INSTALLMENTS_PURCHASES            0
CASH_ADVANCE                      0
PURCHASES_FREQUENCY               0
ONEOFF_PURCHASES_FREQUENCY        0
PURCHASES_INSTALLMENTS_FREQUENCY  0
CASH_ADVANCE_FREQUENCY            0
CASH_ADVANCE_TRX                  0
PURCHASES_TRX                     0
CREDIT_LIMIT                      1
PAYMENTS                          0
MINIMUM_PAYMENTS                313
PRC_FULL_PAYMENT                  0
dtype: int64
```

[3]
```
# preenchendo todos os NaN com os valores de mediana do dataset.
dataframe.fillna(dataframe.median(), inplace=True)
dataframe.isna().sum()
```

```
BALANCE                           0
BALANCE_FREQUENCY                 0
PURCHASES                         0
ONEOFF_PURCHASES                  0
INSTALLMENTS_PURCHASES            0
CASH_ADVANCE                      0
PURCHASES_FREQUENCY               0
ONEOFF_PURCHASES_FREQUENCY        0
PURCHASES_INSTALLMENTS_FREQUENCY  0
CASH_ADVANCE_FREQUENCY            0
CASH_ADVANCE_TRX                  0
PURCHASES_TRX                     0
CREDIT_LIMIT                      0
PAYMENTS                          0
MINIMUM_PAYMENTS                  0
PRC_FULL_PAYMENT                  0
dtype: int64
```

[4]
```
# normalizando os dados para a clusterização
from sklearn.preprocessing import Normalizer

values = Normalizer().fit_transform(dataframe.values)
values
```

```
array([[3.93555441e-02, 7.87271593e-04, 9.17958473e-02, ...,
        1.94178127e-01, 1.34239194e-01, 0.00000000e+00],
       [2.93875903e-01, 8.34231560e-05, 0.00000000e+00, ...,
```

```
       3.76516684e-01, 9.84037959e-02, 2.03923046e-05],
      [3.10798149e-01, 1.24560965e-04, 9.63068011e-02, ...,
       7.74852335e-02, 7.81351982e-02, 0.00000000e+00],
      ...,
      [2.27733092e-02, 8.11060955e-04, 1.40540698e-01, ...,
       7.90986945e-02, 8.02156174e-02, 2.43318384e-04],
      [2.65257948e-02, 1.64255731e-03, 0.00000000e+00, ...,
       1.03579625e-01, 1.09898221e-01, 4.92767391e-04],
      [1.86406219e-01, 3.33426837e-04, 5.46778061e-01, ...,
       3.15915455e-02, 4.41568390e-02, 0.00000000e+00]])
```

```
[5] from sklearn.cluster import KMeans

    kmeans = KMeans(n_clusters=5, n_init=10, max_iter=300)
    y_pred = kmeans.fit_predict(values)
```

```
[6] from sklearn import metrics

    labels = kmeans.labels_
    silhouette = metrics.silhouette_score(values, labels, metric='euclidean')
    print(silhouette)
```

```
0.3644845919974304
```

```
[7] dbs = metrics.davies_bouldin_score(values, labels)
    print(dbs)
```

```
1.0757138590613295
```

```
[8] calinski = metrics.calinski_harabasz_score(values, labels)
    print(calinski)
```

```
3431.79374284143
```

```
[9] def clustering_algorithm(n_clusters, dataset):
      kmeans = KMeans(n_clusters=n_clusters, n_init=10, max_iter=300)
      labels = kmeans.fit_predict(dataset)
      s = metrics.silhouette_score(dataset, labels, metric='euclidean')
      dbs = metrics.davies_bouldin_score(dataset, labels)
      calinski = metrics.calinski_harabasz_score(dataset, labels)
      return s, dbs, calinski
```

```
[10] s1, dbs1, calinski1 = clustering_algorithm(3, values)
     print(s1, dbs1, calinski1)
```

```
0.32718018646452657 1.3099263090078583 3526.457038744187
```

```
[11] s2, dbs2, calinski2 = clustering_algorithm(5, values)
     print(s2, dbs2, calinski2)
```

```
0.36454314826693845 1.075812734747931 3431.800540734652
```

```
[12] s3, dbs3, calinski3 = clustering_algorithm(10, values)
     print(s3, dbs3, calinski3)
```

```
0.3513538592938944 1.1172981433812788 3019.1684605115156
```

```
[13] s4, dbs4, calinski4 = clustering_algorithm(20, values)
     print(s4, dbs4, calinski4)
```

```
0.2746878273393849 1.2150556690307737 2416.262439885008
```

```
[14] s5, dbs5, calinski5 = clustering_algorithm(50, values)
     print(s5, dbs5, calinski5)
```

```
0.26072153685038213 1.1834863591294742 1610.441421276917
```

Escolhemos trabalhar com 5 clusters por conta da melhoria dos resultados de Silhouette, a medida mais utilizada para esses casos, e também por ser mais interessante dividir os clientes em 5 grupos distintos do que apenas 3, já que os resultados não só da primeira métrica supracitada melhorar, mas também a Davies Bouldin e a Calinski Harabasz não apresentarem grandes alterações de valor e mantendo suas particularidades de cálculo.

```
[15] dataframe.count()
```

```
BALANCE                             8950
BALANCE_FREQUENCY                   8950
PURCHASES                           8950
ONEOFF_PURCHASES                    8950
INSTALLMENTS_PURCHASES              8950
CASH_ADVANCE                        8950
PURCHASES_FREQUENCY                 8950
ONEOFF_PURCHASES_FREQUENCY          8950
PURCHASES_INSTALLMENTS_FREQUENCY    8950
CASH_ADVANCE_FREQUENCY              8950
CASH_ADVANCE_TRX                    8950
PURCHASES_TRX                       8950
CREDIT_LIMIT                        8950
PAYMENTS                            8950
```

```
MINIMUM_PAYMENTS                  8950
PRC_FULL_PAYMENT                  8950
dtype: int64
```

```
[16] import numpy as np
```

```
[17] random_data = np.random.rand(8950, 16)
     s, dbs, calinski = clustering_algorithm(5, random_data)
     print(s, dbs, calinski)
     print(s2, dbs2, calinski2)
```

```
0.04007168309625099 3.491801217352981 303.04654633026036
0.36454314826693845 1.075812734747931 3431.800540734652
```

Os resultados acima mostram uma diferença grande entre números aleatórios gerados e os números do dataset para as mesma atribuições de cálculo, o que é mais um ponto de validação dos resultados obtidos quando consideramos seus critérios interpretativos.

```
[18] set1, set2, set3 = np.array_split(values, 3)
     s1, dbs1, calinski1 = clustering_algorithm(5, set1)
     s2, dbs2, calinski2 = clustering_algorithm(5, set2)
     s3, dbs3, calinski3 = clustering_algorithm(5, set3)
     print(s1, dbs1, calinski1)
     print(s2, dbs2, calinski2)
     print(s3, dbs3, calinski3)
```

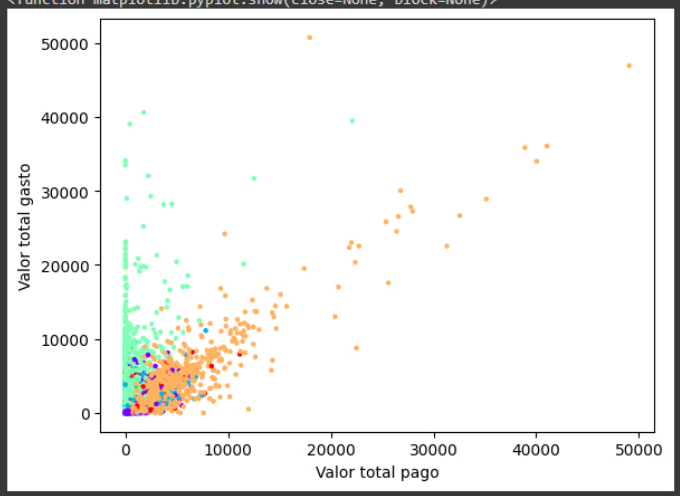
```
0.36870710489440606 1.0593734845912124 1204.0515886736605
0.3532183246410258 1.142940860955218 1194.9401425417566
0.367206263825063 1.0978523283321115 1167.5102424823506
```

A similaridade dos valores validam a estabilidade dos nossos clusters.

```
[19] import matplotlib.pyplot as plt
```

```
[20] plt.scatter(dataframe['PURCHASES'], dataframe['PAYMENTS'], c=labels, s=5, cmap='rainbow')
     plt.xlabel('Valor total pago')
     plt.ylabel('Valor total gasto')
     plt.show
```

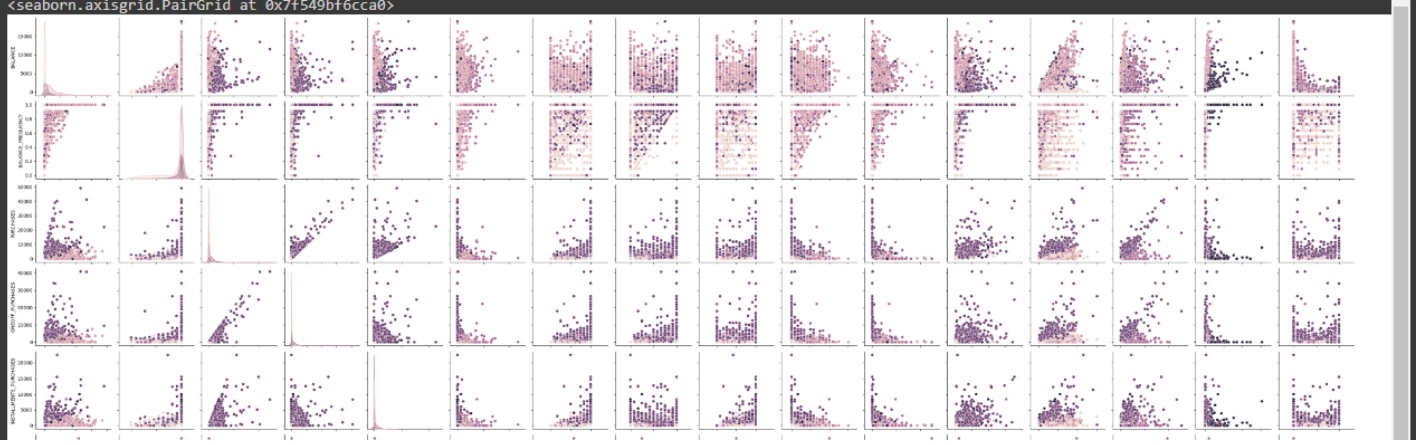
```
<function matplotlib.pyplot.show(close=None, block=None)>
```

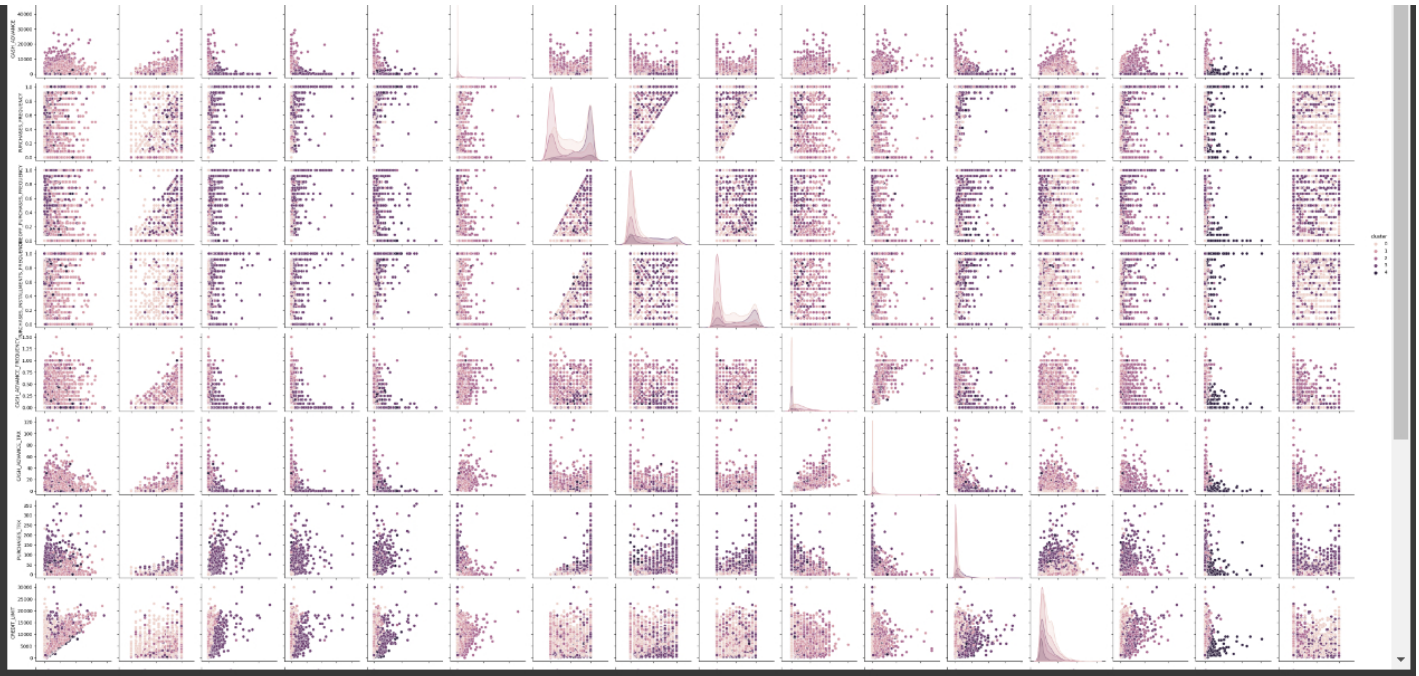



```
[21] import seaborn as sns
```

```
[22] dataframe['cluster'] = labels
     sns.pairplot(dataframe[0:], hue='cluster')
```

```
<seaborn.axisgrid.PairGrid at 0x7f549bf6cca0>
```





Dada uma visualização em que não foi possível tirar nenhuma conclusão interpretativa dos dados, partimos para outra estratégia de análise.

```
[23] dataframe.groupby('cluster').describe()
```

	BALANCE								BALANCE_FREQUENCY		...	MINIMUM_PAYMENTS		PRC_FULL_PA	
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mea
cluster															
0	3278.0	443.992358	846.597972	0.000000	23.315538	82.992153	411.193793	9630.367575	3278.0	0.774417	...	248.800040	4483.503862	3278.0	0.24
1	2649.0	3037.962543	2478.838161	6.824905	1256.612223	2174.018945	4223.867789	18495.558550	2649.0	0.963740	...	1330.593643	8345.641905	2649.0	0.0(
2	1072.0	1794.024195	1982.950586	0.000000	506.925348	1087.920500	2308.762488	14581.459140	1072.0	0.862131	...	884.718306	21235.065300	1072.0	0.1
3	1540.0	1142.222657	1664.035755	0.000000	211.566656	554.332740	1370.859355	19043.138560	1540.0	0.928053	...	524.184580	18621.013310	1540.0	0.27
4	411.0	1987.501586	1854.459156	70.794108	1018.958891	1337.287314	2138.678431	11670.179850	411.0	0.989486	...	7112.618584	76406.207520	411.0	0.0

5 rows × 128 columns

```
[24] centroids = kmeans.cluster_centers_
     print(centroids)
```

```
[[6.91386971e-02 2.30552524e-04 1.21267798e-01 4.80255998e-02
  7.33425496e-02 2.43607736e-02 1.72581896e-04 3.37097813e-05
  1.40021986e-04 7.93309218e-06 1.24113667e-04 2.86037895e-03
  9.43728649e-01 1.53557708e-01 5.68217245e-02 7.57977264e-05]
 [5.04263792e-01 2.72194282e-04 6.17808431e-02 3.62269020e-02
  2.55756543e-02 2.52903017e-01 6.73170559e-05 2.56975193e-05
  4.50651626e-05 5.74388410e-05 1.15687334e-03 1.33194105e-03
  7.19174706e-01 1.66205262e-01 1.79245673e-01 5.52303554e-07]
 [2.35141479e-01 2.16555293e-04 6.03857074e-02 3.44170171e-02
  2.59739049e-02 4.00128071e-01 5.26401668e-05 2.04882397e-05
  3.51921361e-05 6.54864556e-05 1.57199032e-03 1.12217156e-03
  5.35089146e-01 5.98040052e-01 1.04348171e-01 2.77498236e-05]
 [1.59699490e-01 2.45649360e-04 4.38116859e-01 2.56325660e-01
  1.81962376e-01 2.57889184e-02 2.04826443e-04 8.87914942e-05
  1.51162866e-04 9.36296056e-06 1.69350945e-04 6.17145756e-03
  6.04336246e-01 4.02584838e-01 8.54128174e-02 7.26645255e-05]
 [3.32918803e-01 2.68423552e-04 1.58240376e-01 3.63059801e-02
  1.22319068e-01 5.83557153e-02 1.66754597e-04 2.64938445e-05
  1.47063187e-04 1.93217566e-05 4.27934836e-04 3.86197017e-03
  3.76718873e-01 2.51861450e-01 6.80311114e-01 4.23993819e-06]]
```

```
[25] max = len(centroids[0])
     for i in range(max):
       print(dataframe.columns.values[i], '\n{:4f}'.format(centroids[:, i].var()))
```

```
BALANCE
0.022418
BALANCE_FREQUENCY
0.000000
PURCHASES
0.019621
ONEOFF_PURCHASES
0.007598
INSTALLMENTS_PURCHASES
0.003588
CASH_ADVANCE
0.022548
PURCHASES_FREQUENCY
0.000000
```

```
ONEOFF_PURCHASES_FREQUENCY
0.000000
PURCHASES_INSTALLMENTS_FREQUENCY
0.000000
CASH_ADVANCE_FREQUENCY
0.000000
CASH_ADVANCE_TRX
0.000000
PURCHASES_TRX
0.000003
CREDIT_LIMIT
0.036005
PAYMENTS
0.027994
MINIMUM_PAYMENTS
0.054331
PRC_FULL_PAYMENT
0.000000
```

Clique duas vezes (ou pressione "Enter") para editar

- BALANCE 0.022405
- PURCHASES 0.019680
- CASH_ADVANCE 0.022534
- CREDIT_LIMIT 0.035973
- PAYMENTS 0.027942

```
[32] description = dataframe.groupby("cluster")["BALANCE", "PURCHASES", "CASH_ADVANCE", "CREDIT_LIMIT", "PAYMENTS"]
     n_clients = description.size()
     description = description.mean()
     description['n_clients'] = n_clients
     print(description)
```

```
              BALANCE    PURCHASES  CASH_ADVANCE  CREDIT_LIMIT     PAYMENTS  \
cluster
0          443.992358   629.345171    141.569301   5132.945933   814.477150
1         3037.962543   385.248630   1636.917210   4495.771989   968.890376
2         1794.024195   475.494823   3270.246792   3976.372399  4709.556601
3         1142.222657  3268.886929    183.019398   4097.701397  3037.241495
4         1987.501586   854.865815    421.129352   2227.737226  1336.238911

          n_clients
cluster
0              3278
1              2649
2              1072
3              1540
4               411
<ipython-input-32-09694d8613d0>:1: FutureWarning: Indexing with multiple keys (implicitly converted to a tuple of keys) will be deprecated, use a list instead.
  description = dataframe.groupby("cluster")["BALANCE", "PURCHASES", "CASH_ADVANCE", "CREDIT_LIMIT", "PAYMENTS"]
```

```
[34] #atributo invocado para ajudar na análise do dataframe acima.
     dataframe.groupby("cluster")["PRC_FULL_PAYMENT"].describe()
```

cluster	count	mean	std	min	25%	50%	75%	max
0	3278.0	0.246549	0.346601	0.0	0.0	0.000000	0.444444	1.0
1	2649.0	0.001799	0.023260	0.0	0.0	0.000000	0.000000	0.6
2	1072.0	0.117384	0.186853	0.0	0.0	0.083333	0.142857	1.0
3	1540.0	0.278582	0.367203	0.0	0.0	0.083333	0.545455	1.0
4	411.0	0.019318	0.090901	0.0	0.0	0.000000	0.000000	1.0

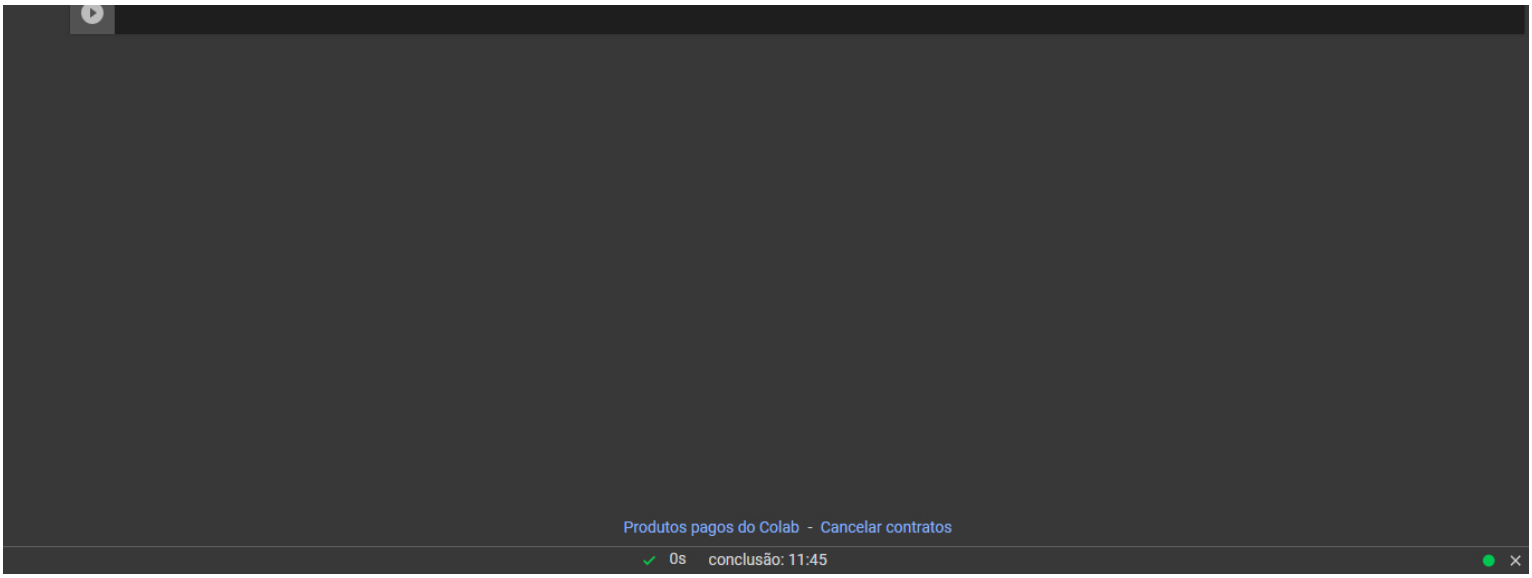
Interpretação dos dados dos clusters.

Anotação dos valores extremos.

- CLUSTER 0: Clientes que gastam pouco. Clientes com o maior limite. Bons pagadores. Maior número de clientes.
- CLUSTER 1: Clientes que mais gastam. O foco deles é o saque. Piores pagadores. Boa quantidade de clientes.
- CLUSTER 2: Clientes que gastam muito com compras. Melhores pagadores.
- CLUSTER 3: Clientes que gastam muito com saques. Pagam as vezes.
- CLUSTER 4: Clientes com o menor limite. Não são bons pagadores. Menor quantidade de clientes.

Se fossemos aplicar uma regra de negócio para, por exemplo, aumentar consumo de determinado cluster através de uma campanha de marketing ou oferecimento de um determinado serviço como um programa de pontos, provavelmente faríamos com o CLUSTER 0, pois são bons pagadores e uma base grande de clientes, apesar de sabermos que gastam pouco.

Aplicaríamos um teste AB por um determinado período, separando-os em Grupo Teste e Grupo controle, para podermos concluir sobre os efeitos de consumo, se houve ou não efetividade na medida.


Produtos pagos do Colab - Cancelar contratos
0s conclusão: 11:45