PROJET D'APPRENTISSAGE STATISTIQUE - GRANDE DIMENSION

Nous disposons d'un jeu de données relatives à l'usage de carte de crédit. Chaque observation représente un client et chaque colonne représente un usage / une mesure de l'usage de sa carte de crédit. Le principe ici est d'établir une segmentation des clients à des fins de besoin marketing. La particularité de ce jeu de données est notamment sa taille, nous travaillons ici en grande dimension.

I) PRE-PROCESSING

In [1]:
```
import pandas as pd
from tqdm.notebook import tqdm
import warnings
warnings.filterwarnings('ignore')
from sklearn import metrics
```

In [2]:
```
data_credit_brut = pd.read_csv("C:/Users/arian/OneDrive/Bureau/M2 SORBONNE S1 ARIAN
```

In [3]:
```
data_credit_brut
```

Out[3]:

	CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLM
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	
...	...	...	...	...	...	
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 [4]:
```
data_credit_brut.shape
```

Out[4]:
```
(8950, 18)
```

In [5]:
```
#affichage du nombre de valeurs uniques et du type de données
def unique_column_values(df):
    for column in df:
        print("{} | {} | {}".format(
            df[column].name, len(df[column].unique()), df[column].dtype))

unique_column_values(data_credit_brut)
```

```
CUST_ID | 8950 | object
BALANCE | 8871 | float64
BALANCE_FREQUENCY | 43 | float64
PURCHASES | 6203 | float64
ONEOFF_PURCHASES | 4014 | float64
INSTALLMENTS_PURCHASES | 4452 | float64
CASH_ADVANCE | 4323 | float64
PURCHASES_FREQUENCY | 47 | float64
ONEOFF_PURCHASES_FREQUENCY | 47 | float64
PURCHASES_INSTALLMENTS_FREQUENCY | 47 | float64
CASH_ADVANCE_FREQUENCY | 54 | float64
CASH_ADVANCE_TRX | 65 | int64
PURCHASES_TRX | 173 | int64
CREDIT_LIMIT | 206 | float64
PAYMENTS | 8711 | float64
MINIMUM_PAYMENTS | 8637 | float64
PRC_FULL_PAYMENT | 47 | float64
TENURE | 7 | int64
```

In [6]:
```
#on regarde s'il y a des doublons
print('There are ' + str(data_credit_brut.duplicated(subset=data_credit_brut.column
data_credit_brut[data_credit_brut.duplicated(subset=data_credit_brut.columns[1:], k
```

```
There are 0 exact duplicates.
```

Out[6]:

CUST_ID	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_P

On voit qu'il n'y a pas de doublons. Néanmoins, nous n'avons pas exclu la variable CUST_ID. Il y a peut-être des clients différents qui ont exactement les mêmes données associées. Nous allons refaire le test après avoir retiré cette variable.

In [7]:
```
data_credit = data_credit_brut.drop(columns=['CUST_ID'])
```

In [8]:
```
#on regarde s'il y a des doublons après avoir retiré CUST_ID
print('There are ' + str(data_credit.duplicated(subset=data_credit.columns[1:], kee
data_credit[data_credit.duplicated(subset=data_credit.columns[1:], keep=False)]
```

```
There are 0 exact duplicates.
```

Out[8]:

BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES

MISSING VALUES

In [9]:
```
missing_values_count = data_credit.isnull().sum()
total_missing = missing_values_count.sum()
total_missing
```

Out[9]: 314

In [10]:
```
import numpy as np
```

In [11]:
```
total_cells = np.product(data_credit.shape)
total_cells
```

Out[11]: 152150

In [12]:
```
#calcul % de valeurs manquantes
percent_missing = (total_missing/total_cells) * 100
percent_missing
```

Out[12]: 0.20637528754518566

Il y a 0.2 % de valeurs manquantes dans le dataset.

In [13]:
```
#diagramme en bâtons représentant le % de valeurs manquantes par colonne
size = data_credit.shape
nan_values = data_credit.isna().sum()
nan_values = nan_values.sort_values(ascending=True)*100/size[0]

ax = nan_values.plot(kind='barh',
                     figsize=(20, 40),
                     color='#0000FF',
                     zorder=2,
                     width=0.85)

ax.spines['top'].set_visible(False)
ax.spines['left'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)

ax.tick_params(axis="both",
               which="both",
               bottom="off",
               top="off",
               labelbottom="on",
               left="off",
               right="off",
               labelleft="on")
ax.set_title("Taux de valeurs manquantes pour chaque variable du dataframe df")
vals = ax.get_xticks()

for tick in vals:
  ax.axvline(x=tick, linestyle='dashed', alpha=0.4, color='#eeeeee', zorder=1)
```


Taux de valeurs manquantes pour chaque variable du dataframe df
MINIMUM_PAYMENTS
CREDIT_LIMIT
TENURE
PURCHASES_FREQUENCY
BALANCE_FREQUENCY
PURCHASES
ONEOFF_PURCHASES
INSTALLMENTS_PURCHASES
CASH_ADVANCE
PURCHASES_INSTALLMENTS_FREQUENCY
ONEOFF_PURCHASES_FREQUENCY
PRC_FULL_PAYMENT
CASH_ADVANCE_FREQUENCY
CASH_ADVANCE_TRX
PURCHASES_TRX
PAYMENTS
BALANCE

In [14]:
```
#on affiche le nombre total de valeurs manquantes par colonne
data_credit.isnull().sum().sort_values(ascending=False).head(3)
```

Out[14]:
```
MINIMUM_PAYMENTS    313
CREDIT_LIMIT          1
BALANCE               0
dtype: int64
```

In [15]:
```
data_credit['MINIMUM_PAYMENTS'].describe()
```

Out[15]:
```
count     8637.000000
mean       864.206542
std       2372.446607
min          0.019163
25%        169.123707
50%        312.343947
75%        825.485459
max      76406.207520
Name: MINIMUM_PAYMENTS, dtype: float64
```

In [16]:
```
#Pour mieux comprendre les observations où il y a des valeurs manquantes pour MINIM
data_credit[data_credit['MINIMUM_PAYMENTS'].isna()]
```

Out[16]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PUR
3	1666.670542	0.636364	1499.00	1499.00	
45	2242.311686	1.000000	437.00	97.00	
47	3910.111237	1.000000	0.00	0.00	
54	6.660517	0.636364	310.00	0.00	
55	1311.995984	1.000000	1283.90	1283.90	
...	...	...	...	...	
8919	14.524779	0.333333	152.00	152.00	
8929	371.527312	0.333333	0.00	0.00	
8935	183.817004	1.000000	465.90	0.00	
8944	193.571722	0.833333	1012.73	1012.73	
8946	19.183215	1.000000	300.00	0.00	

313 rows × 17 columns

In [17]:
```
data_credit["MINIMUM_PAYMENTS"].fillna(value=data_credit["MINIMUM_PAYMENTS"].median
```

In [18]:
```
#Pour mieux comprendre les observations où il y a des valeurs manquantes pour CREDI
data_credit[data_credit['CREDIT_LIMIT'].isna()]
```

Out[18]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCH
5203	18.400472	0.166667	0.0	0.0	

In [19]:
```
data_credit['CREDIT_LIMIT'].describe()
```

Out[19]:
```
count     8949.000000
mean      4494.449450
std       3638.815725
min         50.000000
25%       1600.000000
50%       3000.000000
75%       6500.000000
max      30000.000000
Name: CREDIT_LIMIT, dtype: float64
```

In [20]:
```
#on peut se permettre de supprimer l'observation où il y a une valeur manquante pou
data_credit = data_credit.dropna(subset = ['CREDIT_LIMIT'])
```

In [21]:
```
data_credit[data_credit['CREDIT_LIMIT'].isna()]
```

Out[21]:

	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PURCHASES

In [22]:
```
data_credit.isnull().sum().sort_values(ascending=False).head(3)
```

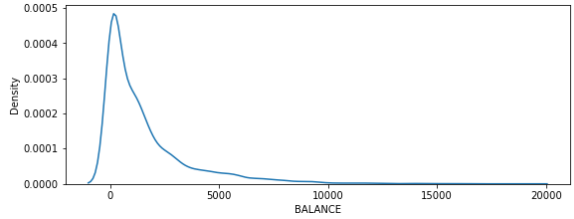
Out[22]:
```
BALANCE                   0
CASH_ADVANCE_FREQUENCY    0
PRC_FULL_PAYMENT          0
dtype: int64
```

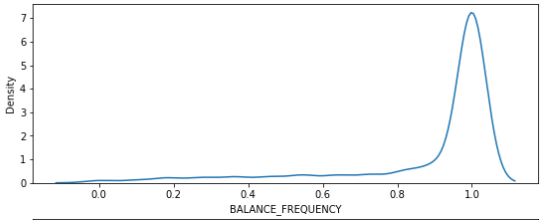
OUTLIERS (VARIABLES QUANTITATIVES CONTINUES)

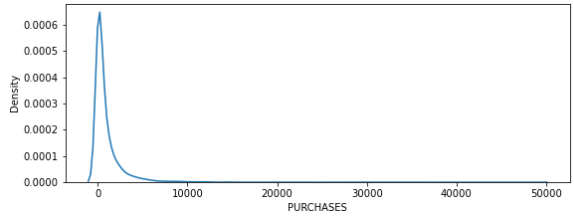
In [23]:
```
data_credit.columns
```

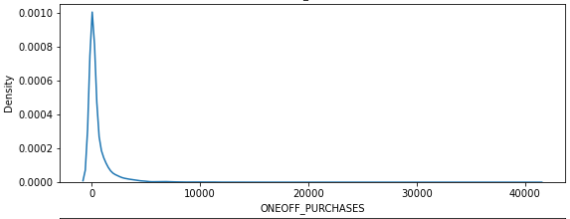
Out[23]:
```
Index(['BALANCE', 'BALANCE_FREQUENCY', 'PURCHASES', 'ONEOFF_PURCHASES',
       'INSTALLMENTS_PURCHASES', 'CASH_ADVANCE', 'PURCHASES_FREQUENCY',
       'ONEOFF_PURCHASES_FREQUENCY', 'PURCHASES_INSTALLMENTS_FREQUENCY',
       'CASH_ADVANCE_FREQUENCY', 'CASH_ADVANCE_TRX', 'PURCHASES_TRX',
       'CREDIT_LIMIT', 'PAYMENTS', 'MINIMUM_PAYMENTS', 'PRC_FULL_PAYMENT',
       'TENURE'],
      dtype='object')
```

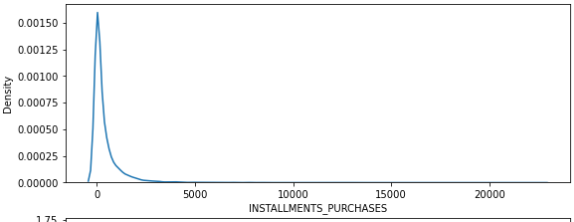
In [24]:
```
#distributions
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=(20,35))
for i, col in enumerate(data_credit.columns):
    if data_credit[col].dtype != 'object':
        ax = plt.subplot(9, 2, i+1)
        sns.kdeplot(data_credit[col], ax=ax)
        plt.xlabel(col)

plt.show()
```

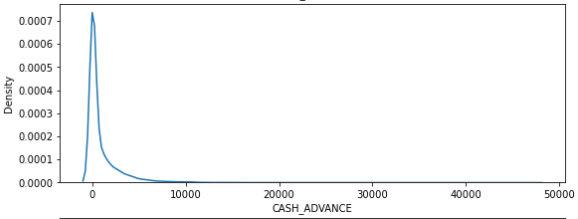

Density
BALANCE

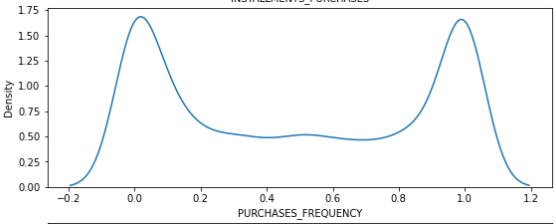

Density
BALANCE_FREQUENCY

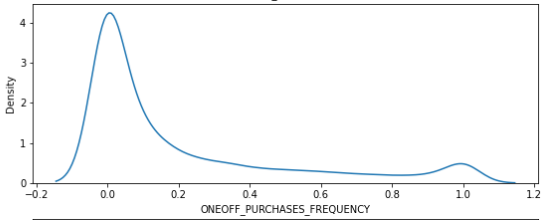

Density
PURCHASES

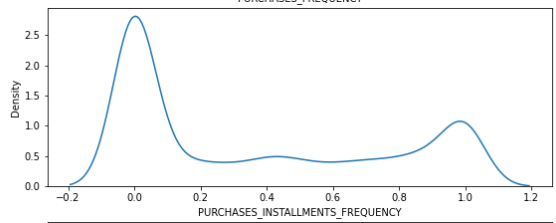

Density
ONEOFF_PURCHASES

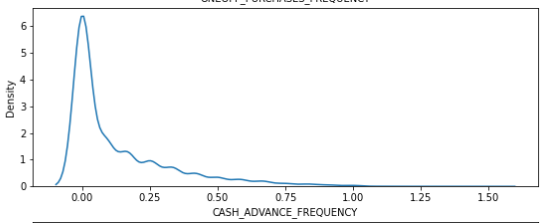

Density
INSTALLMENTS_PURCHASES

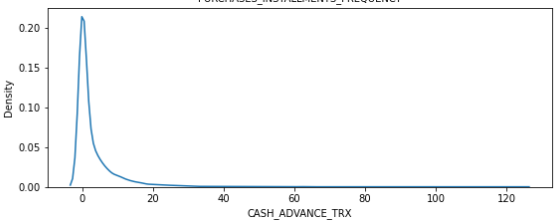

Density
CASH_ADVANCE

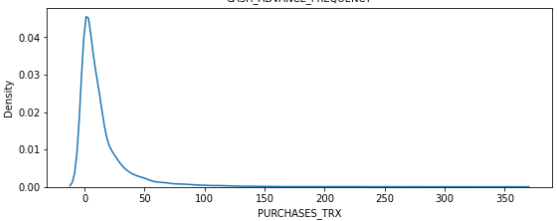

Density
PURCHASES_FREQUENCY

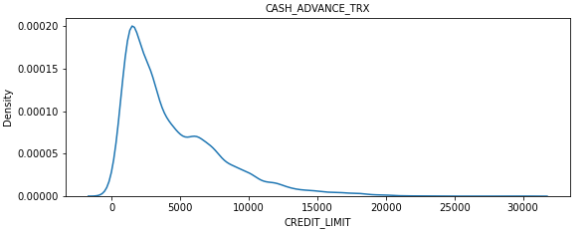

Density
ONEOFF_PURCHASES_FREQUENCY

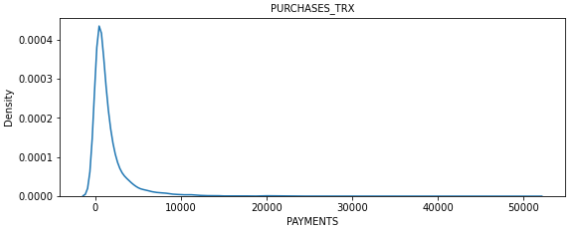

Density
PURCHASES_INSTALLMENTS_FREQUENCY

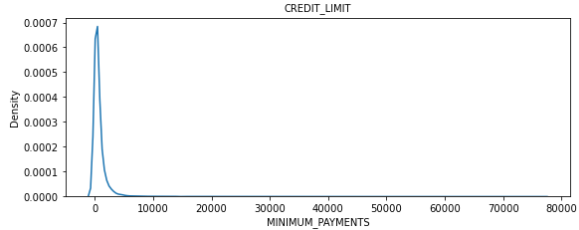

Density
CASH_ADVANCE_FREQUENCY

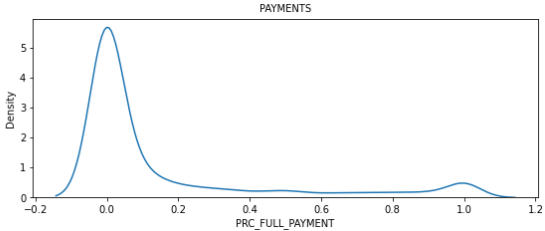

Density
CASH_ADVANCE_TRX

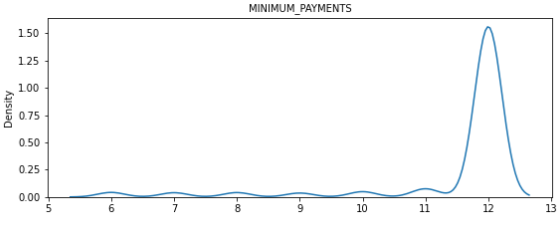

Density
PURCHASES_TRX


Density
CREDIT_LIMIT


Density
PAYMENTS


Density
MINIMUM_PAYMENTS

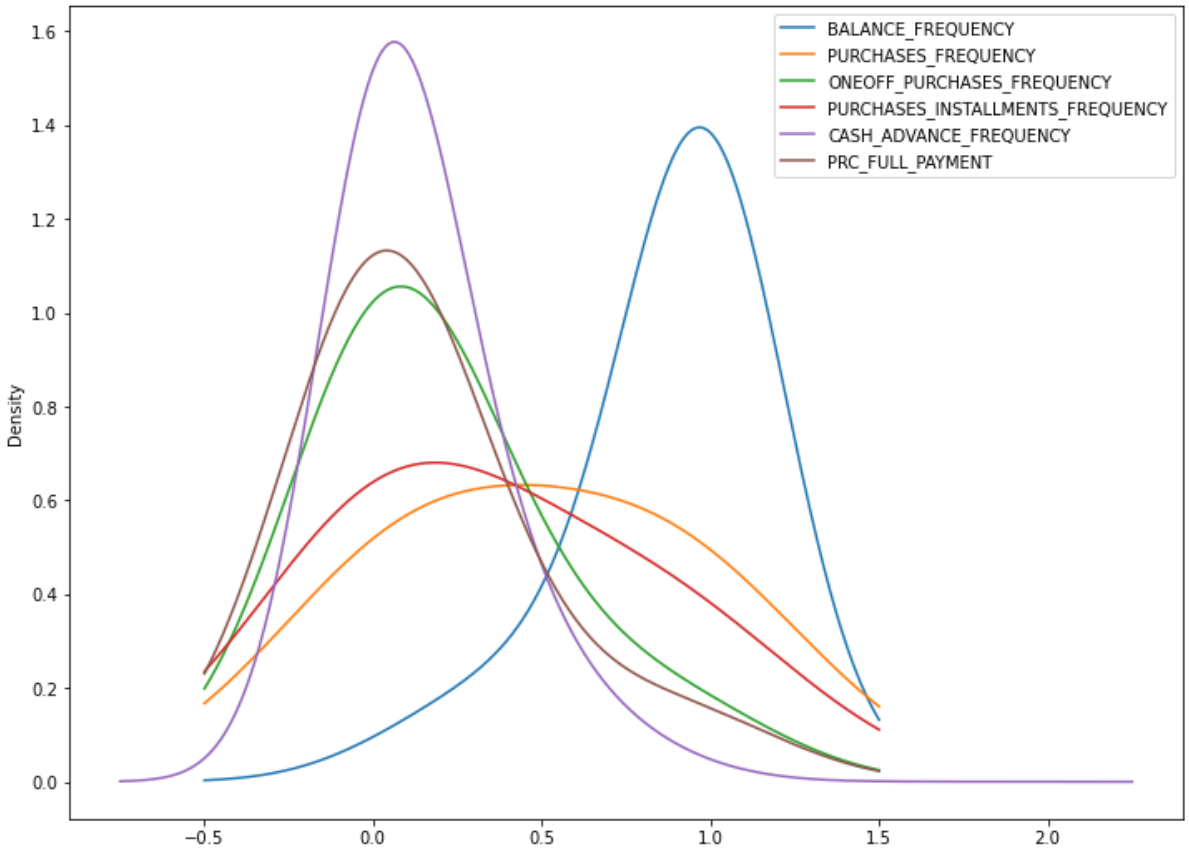

Density
PRC_FULL_PAYMENT


Density

TENURE

On peut remarquer que les distributions sont non gaussiennes.

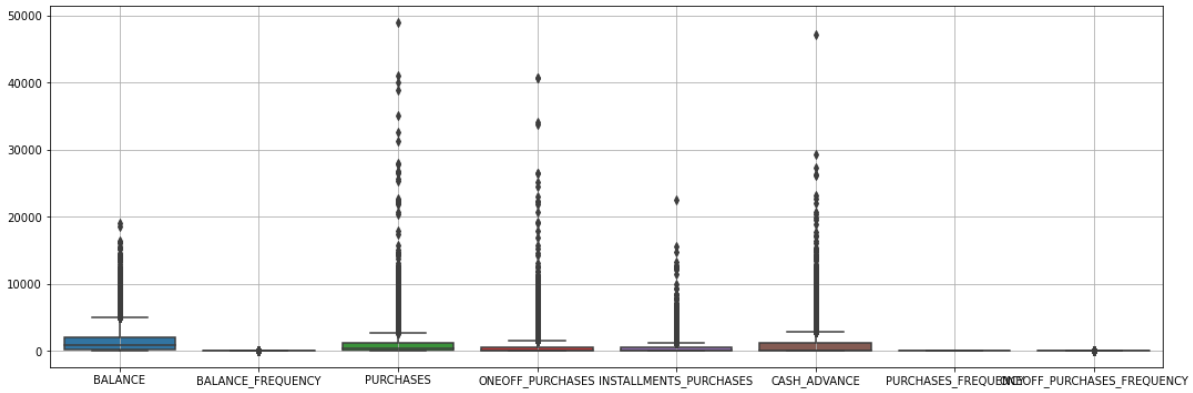
In [25]:
```
ax = data_credit[['BALANCE_FREQUENCY', 'PURCHASES_FREQUENCY',
        'ONEOFF_PURCHASES_FREQUENCY', 'PURCHASES_INSTALLMENTS_FREQUENCY',
        'CASH_ADVANCE_FREQUENCY','PRC_FULL_PAYMENT']].plot.kde(figsize=(12,9), bw_
```



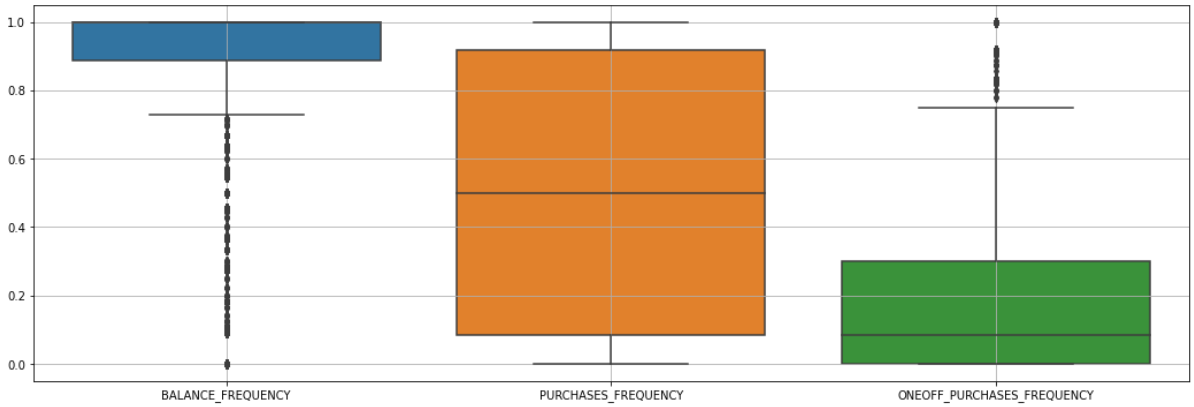
In [26]:
```
#on détecte les outliers grâce à des boxplots, pour les variables continues (on ret
```

In [27]:
```
def boxplot(data):
    plt.figure(figsize=(18,6))
    sns.boxplot(data=data)
    plt.grid()
```

In [28]:
```
boxplot(data_credit.loc[:,['BALANCE', 'BALANCE_FREQUENCY', 'PURCHASES', 'ONEOFF_PUR
       'INSTALLMENTS_PURCHASES', 'CASH_ADVANCE', 'PURCHASES_FREQUENCY',
       'ONEOFF_PURCHASES_FREQUENCY',]])
```



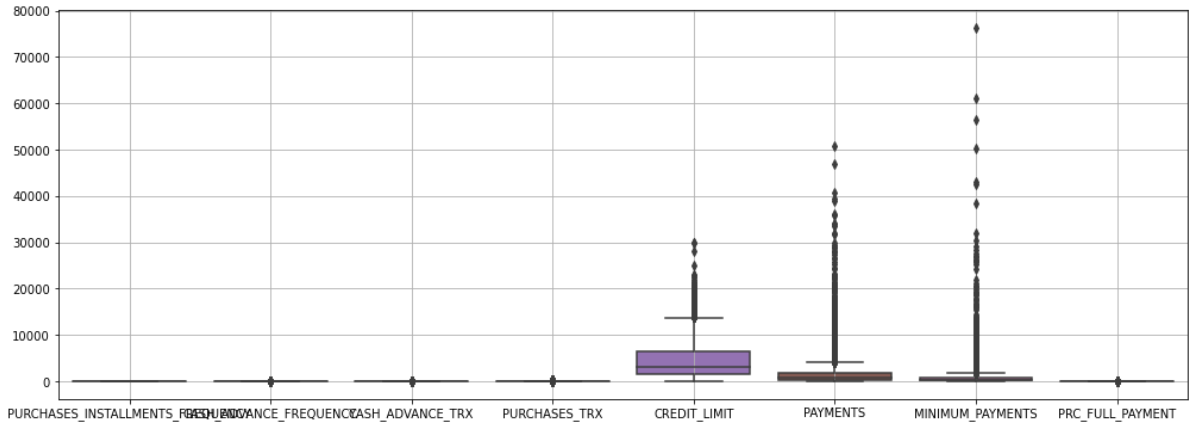
In [29]:
```
#pour avoir plus de précision, on se concentre sur les boxplots écrasés
boxplot(data_credit.loc[:,['BALANCE_FREQUENCY','PURCHASES_FREQUENCY',
       'ONEOFF_PURCHASES_FREQUENCY',]])
```



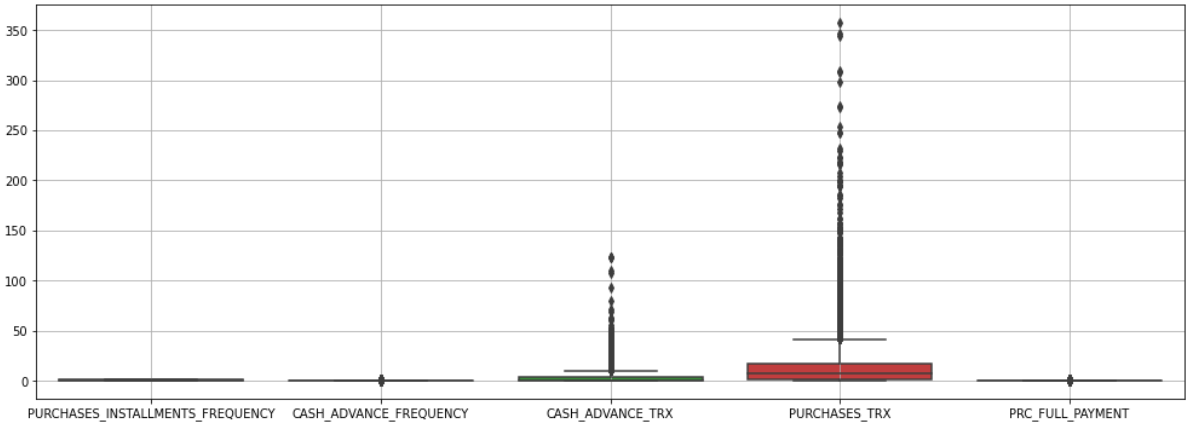
On voit des outliers dans les colonnes balance, purchases, oneoff_purchases, installmentts_purchases & cash_advance.

In [30]:
```
def boxplot(data):
    plt.figure(figsize=(17,6))
    sns.boxplot(data=data)
    plt.grid()
```

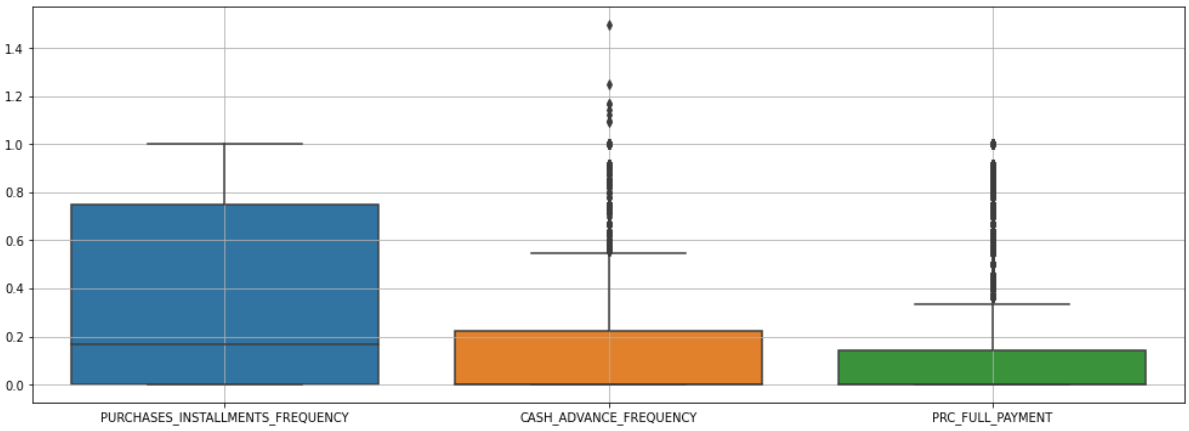
In [31]:
```
boxplot(data_credit.loc[:,['PURCHASES_INSTALLMENTS_FREQUENCY',
       'CASH_ADVANCE_FREQUENCY', 'CASH_ADVANCE_TRX', 'PURCHASES_TRX',
       'CREDIT_LIMIT', 'PAYMENTS', 'MINIMUM_PAYMENTS', 'PRC_FULL_PAYMENT']])
```



In [32]:
```
#pour avoir plus de précision, on se concentre sur les boxplots écrasés
boxplot(data_credit.loc[:,['PURCHASES_INSTALLMENTS_FREQUENCY',
       'CASH_ADVANCE_FREQUENCY', 'CASH_ADVANCE_TRX', 'PURCHASES_TRX', 'PRC_FULL_PAY
```



In [33]:
```
#pour avoir plus de précision, on se concentre sur les boxplots écrasés
boxplot(data_credit.loc[:,['PURCHASES_INSTALLMENTS_FREQUENCY',
       'CASH_ADVANCE_FREQUENCY','PRC_FULL_PAYMENT']])
```



On voit des outliers dans les colonnes credit_limit, payments, minimum_payments.

In [34]:
```
outliers = ['BALANCE','BALANCE_FREQUENCY','CASH_ADVANCE_TRX','PURCHASES_TRX','CASH_
```

On voit qu'il y a beaucoup de variables affectées par des outliers. Les algorithmes de clustering que nous utiliserons seront affectés par ces outliers.
Pour traiter les outliers, il y a plusieurs solutions : les supprimer ou les remplacer (quartiles).
Ici nous avons décidé de les laisser dans un dataframe (data_credit) et de les remplacer par le premier (si valeur très loin en-dessous de la moyenne) ou troisième quartile (au-dessus) dans un autre (data_credit_out).
Nous ne les avons pas supprimés car ils sont très nombreux et les supprimer nous retirerait beaucoup d'information.

In [35]:
```
data_credit_out = data_credit.copy()
```

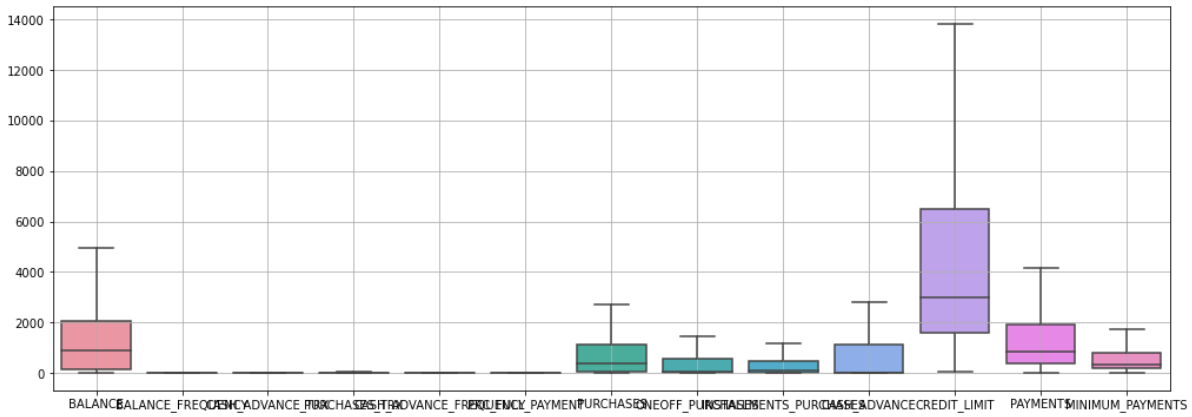
In [36]:
```
def outliers_imput(data,feature):
    q1 = np.percentile(data[feature],25)
    q3 = np.percentile(data[feature],75)
```

```
    iqr = q3-q1
    lower_bound = q1 - 1.5*iqr
    upper_bound = q3 + 1.5*iqr


    data.loc[data[feature] < lower_bound,feature] = lower_bound
    data.loc[data[feature] > upper_bound,feature] = upper_bound
```

In [37]:
```
for feature in outliers:
    outliers_imput(data_credit_out,feature)
```

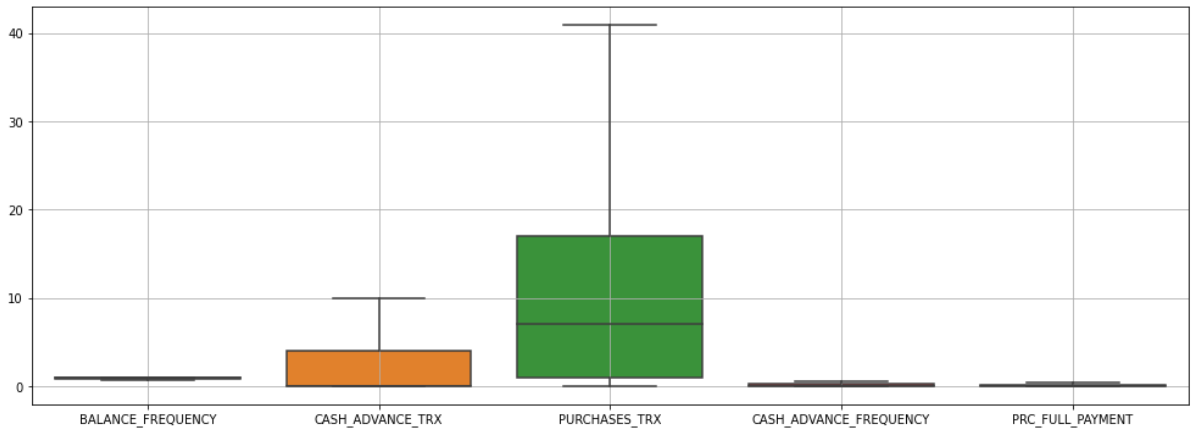
In [38]:
```
def boxplot(data):
    plt.figure(figsize=(17,6))
    sns.boxplot(data=data_credit_out[outliers])
    plt.grid()
```

In [39]:
```
boxplot(data_credit_out[outliers])
```

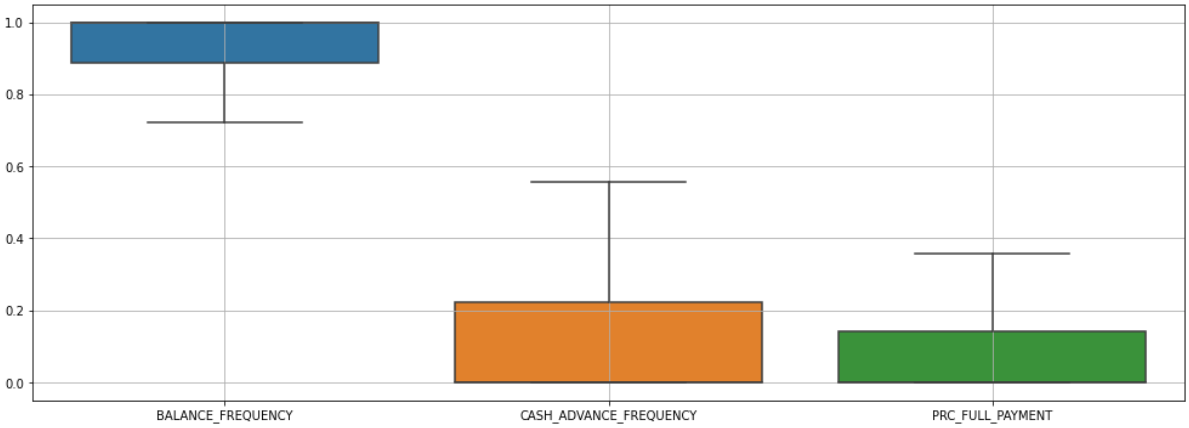


In [40]:
```
def boxplot(data):
    plt.figure(figsize=(17,6))
    sns.boxplot(data=data)
    plt.grid()
```

In [41]:
```
#pour avoir plus de précision, on se concentre sur les boxplots écrasés
boxplot(data_credit_out.loc[:,['BALANCE_FREQUENCY','CASH_ADVANCE_TRX','PURCHASES_TR
```



In [42]:
```
#pour avoir plus de précision, on se concentre sur les boxplots écrasés
boxplot(data_credit_out.loc[:,['BALANCE_FREQUENCY','CASH_ADVANCE_FREQUENCY','PRC_FU
```



Néanmoins, les avoir remplacés n'est pas une solution parfaite car on avait peut-être dans nos données (avant imputation) des valeurs extrêmes mais bien réelles. Cela change et fausse dans une certaine mesure la distribution des données et peut biaiser la constitution des clusters.

In [43]:
```
#on regarde s'il y a des doublons après avoir retiré CUST_ID et avoir imputé les ou
print('There are ' + str(data_credit_out.duplicated(subset=data_credit_out.columns[
data_credit_out[data_credit_out.duplicated(subset=data_credit_out.columns[1:], keep
```

```
There are 6 exact duplicates.
```

Out[43]:

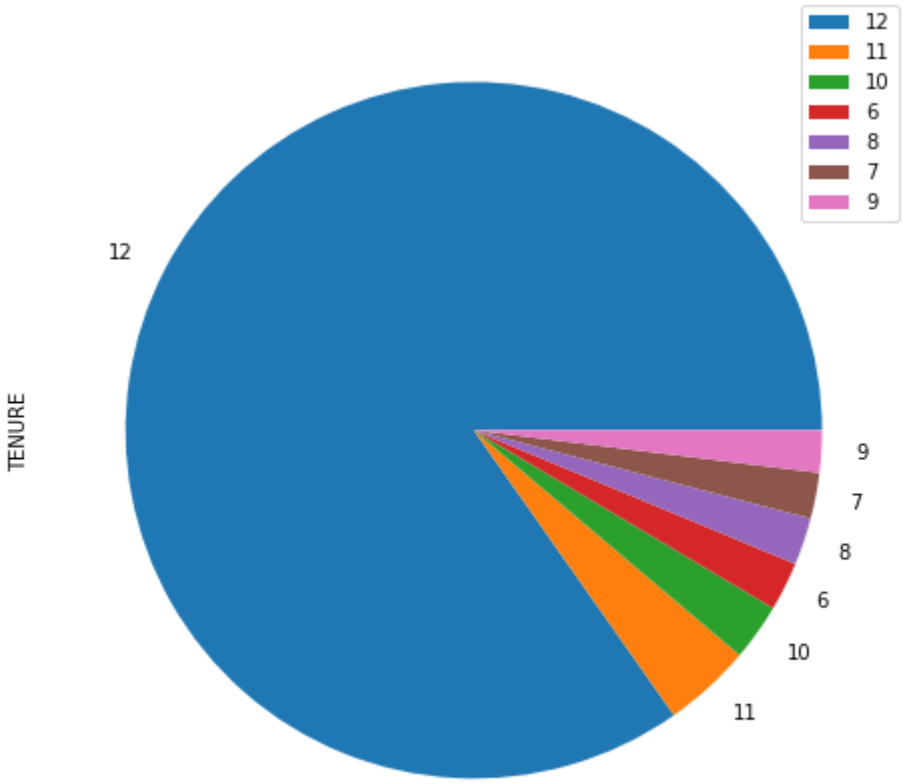
	BALANCE	BALANCE_FREQUENCY	PURCHASES	ONEOFF_PURCHASES	INSTALLMENTS_PUR
643	4943.383447	1.0	2715.725	1444.575	1
883	4943.383447	1.0	0.000	0.000	
1778	4943.383447	1.0	0.000	0.000	
2454	4943.383447	1.0	0.000	0.000	
4140	4943.383447	1.0	2715.725	1444.575	1
4426	4943.383447	1.0	0.000	0.000	

On voit bien que l'imputation a pour effet de confondre certains clients entre eux.

On va finalement préférer conserver les outliers, ceux-ci pouvant d'autant plus correspondre à des comportements bancaires frauduleux ou douteux.

VARIABLES CATEGORIELLES

In [44]:
```
df_TENURE=pd.DataFrame(data_credit['TENURE'].value_counts())
plot = df_TENURE.plot.pie(y='TENURE', figsize=(8, 8))
```



NORMALISATION

Nous préférons normaliser plutôt que standardiser étant donné les distributions de nos variables (non gaussiennes).
L'inconvénient de cette méthode est que cela va compresser les valeurs proches de la médiane à cause des valeurs extrêmes (min/max).

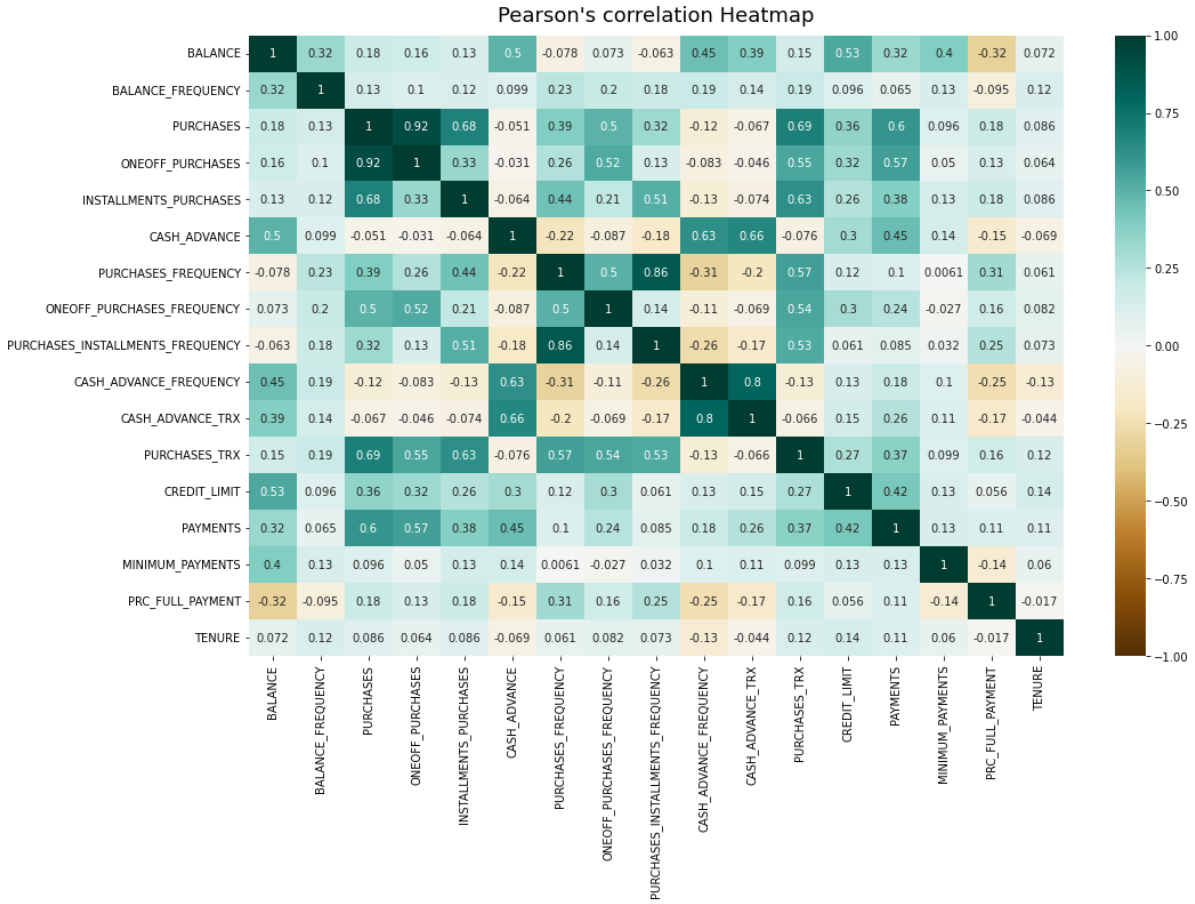
In [45]:
```
data_credit_norm = (data_credit-data_credit.min())/(data_credit.max()-data_credit.m
```

In [46]:
```
data_credit_norm_out = (data_credit_out-data_credit_out.min())/(data_credit_out.max
```

ANALYSE DES CORRELATIONS / REDUCTION DE DIMENSION (PCA)

In [47]:
```
plt.figure(figsize=(16, 10))
heatmap = sns.heatmap(data_credit_norm.corr(method='pearson'), vmin=-1, vmax=1, ann
heatmap.set_title("Pearson's correlation Heatmap", fontdict={'fontsize':18}, pad=12
```

Out[47]:
```
Text(0.5, 1.0, "Pearson's correlation Heatmap")
```



On choisit le coefficient de corrélation de Pearson car nous avons une variable catégorielle. On remarque qu'il y a des corrélations entre variables. Il convient alors de réduire la dimension des données.

In [48]:
```
from sklearn.decomposition import PCA
pca = PCA()
data_PCA_norm = pca.fit_transform(data_credit_norm)
print(pca.explained_variance_ratio_.cumsum())
```

```
[0.49602436 0.6365885  0.7650565  0.84168251 0.9113391  0.94723117
 0.96542845 0.97896048 0.98551728 0.99102779 0.99386978 0.99572875
 0.99699616 0.9982223  0.99927965 0.99999997 1.         ]
```

On voit qu'il faut beaucoup de composantes principales pour expliquer une grande partie de la variance, il convient alors mieux de ne pas réduire la dimensionnalité des données avant le clustering mais de le faire plutôt après chaque algorithme de clustering implémenté dans le but de pouvoir tracer nos graphes. Par ailleurs, ce n'est une nécessité dans notre cas de réduire la dimensionnalité dans la mesure où nous avons beaucoup d'observations.

In [49]:
```
from sklearn.decomposition import PCA
pca = PCA()
data_PCA_norm_out = pca.fit_transform(data_credit_norm_out)
print(pca.explained_variance_ratio_.cumsum())
```

```
[0.34213762 0.56912545 0.66902221 0.75771732 0.81381603 0.86130078
 0.89103865 0.91397787 0.93391162 0.95259069 0.96581539 0.97748753
 0.98509762 0.99097986 0.99466759 0.99815936 1.         ]
```

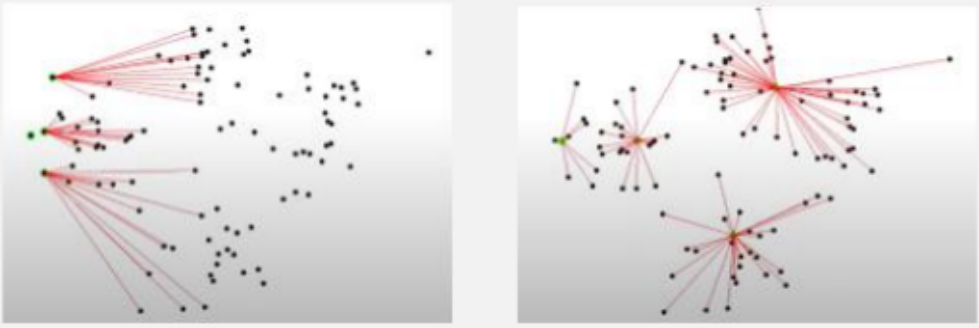
On fait de même pour les données avec outliers imputés pour comparer les analyses.

II) MODELISATION

In [50]:
```
from sklearn.metrics import silhouette_score
```

MODELE DE CLUSTERING - KMEANS (min somme des carrés d'inertie)

L'algorithme classe les données en k clusters, le nombre de clusters k étant fixé à l'avance. Les k clusters doivent minimiser la somme des carrés d'inertie. L'inertie désigne la somme des carrés des distances des points au centre du cluster (centroïde). L'algorithme identifie des centroïdes aléatoirement (c'est-à-dire les centres des clusters qui correspondront aux moyennes arithmétiques de tous les points, pour chaque cluster) et affecte chaque individu au centre le plus proche. Ce processus aléatoire est répété par itération jusqu'à que les individus ne soient plus réaffectés à de nouveaux clusters.
Il est adapté pour de grands échantillons et relativement peu de clusters, de taille uniforme, efficace et simple à implémenter.



Graphiques représentant le processus itéré de l'algorithme kmeans

In [51]:
```
X = data_credit_norm
```

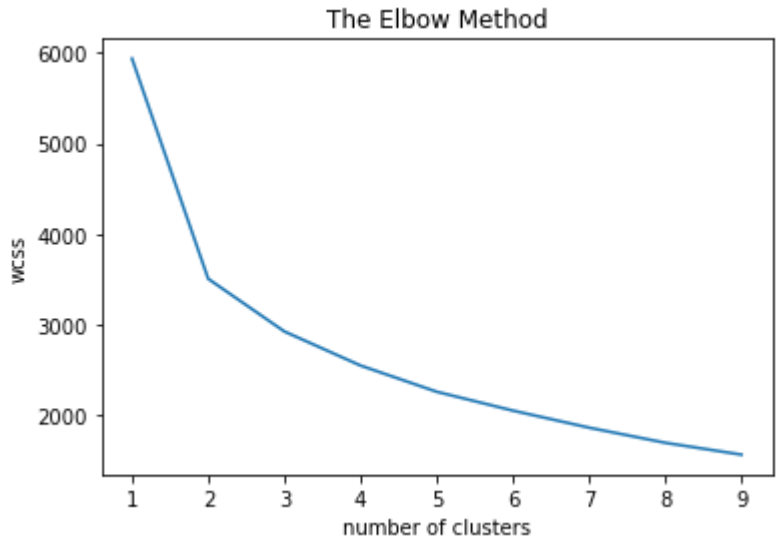
In [52]:
```
Y = data_credit_norm_out
```

In [53]:
```
from sklearn.cluster import KMeans
```

In [54]:
```
wcss=[]
for i in tqdm(range(1,10)):
    kmeans = KMeans(n_clusters= i, init='k-means++', random_state=42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)

plt.plot(range(1,10), wcss)
plt.title('The Elbow Method')
```

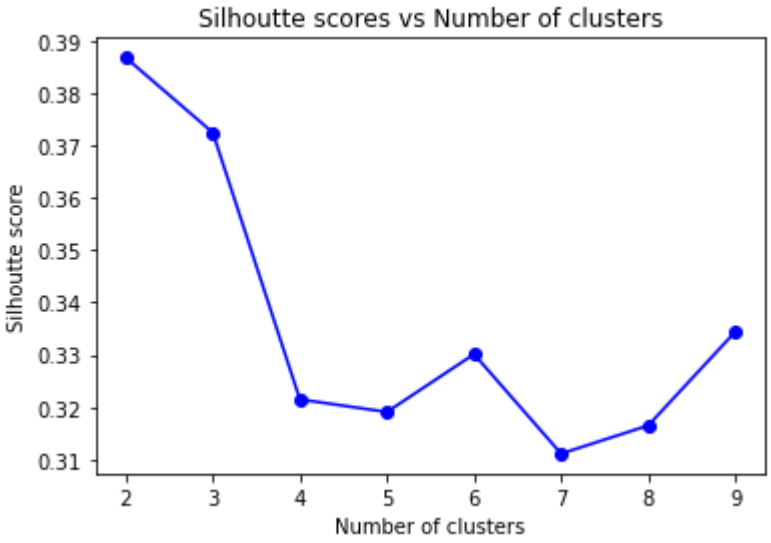
```
plt.xlabel('number of clusters')
plt.ylabel('wcss')
plt.show()
```

```
  0%|          | 0/9 [00:00<?, ?it/s]
```



In [55]:
```
silhoutte_scores =[]
for i in tqdm(range(2,10)):
    kmeans = KMeans(n_clusters= i, init='k-means++', random_state=42)
    kmeans.fit(X)
    silhoutte_scores.append(silhouette_score(X, kmeans.labels_))

plt.plot(range(2,10), silhoutte_scores, "bo-")
plt.xticks(range(2,10))
plt.title('Silhoutte scores vs Number of clusters')
plt.xlabel('Number of clusters')
plt.ylabel('Silhoutte score')
plt.show()
```

```
  0%|          | 0/8 [00:00<?, ?it/s]
```

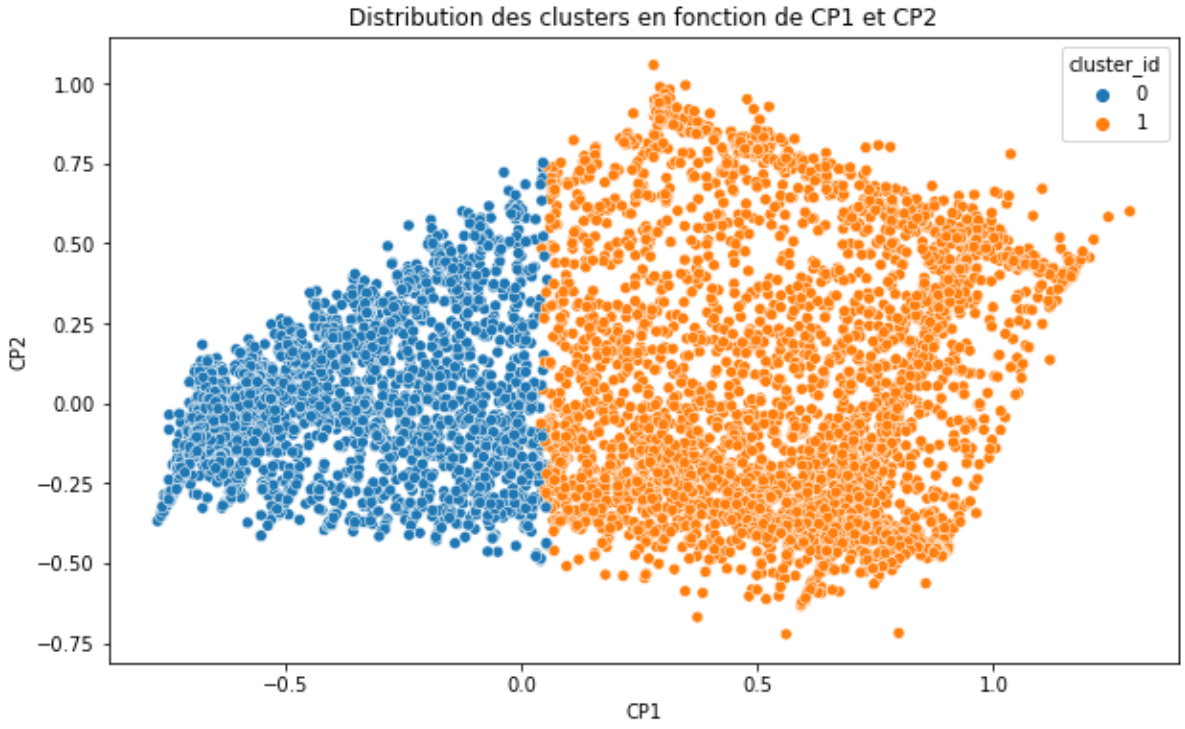


Nous utilisons ici comme métrique le coefficient de silhouette. Celui-ci permet de savoir à quel point les points d'un cluster sont ressérés sur eux-mêmes et loin de ceux des autres

clusters. Il y a donc ici deux notions clefs : similarité et distance. On voit que le nombre optimal de clusters d'après ces deux méthodes est 2. Néanmoins, on pourrait prendre davantage de clusters et faire des regroupements manuels selon les besoins commerciaux. Nous verrons graphiquement après la PCA si deux catégories de clients semblent suffisantes.

In [56]:
```
kmeans = KMeans(n_clusters=2)
kmeans.fit_predict(X)
X['cluster_id'] = kmeans.labels_
```

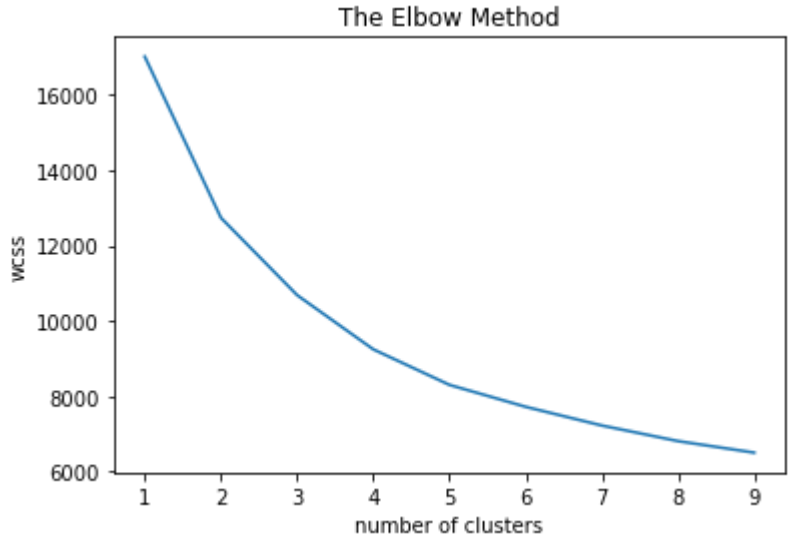
In [57]:
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = kmeans.labels_

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



Comparons maintenant avec les résultats obtenus avec des outliers imputés.

In [58]:
```
wcss=[]
for i in tqdm(range(2,10)):
    kmeans = KMeans(n_clusters= i, init='k-means++', random_state=42)
    kmeans.fit(Y)
    wcss.append(kmeans.inertia_)

plt.plot(range(1,10), wcss)
plt.title('The Elbow Method')
plt.xlabel('number of clusters')
plt.ylabel('wcss')
plt.show()
```

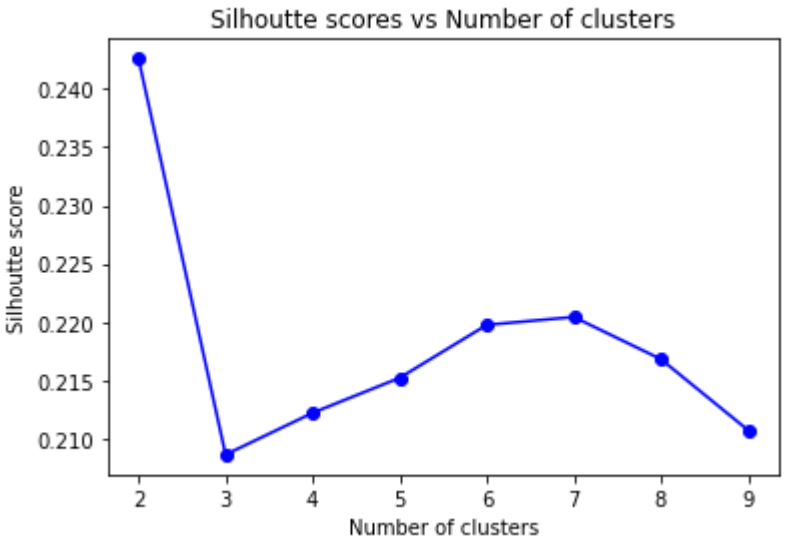
```
0%|          | 0/9 [00:00<?, ?it/s]
```



On peut voir grâce au temps de chargement que l'algorithme a tourné très rapidement, ce qui explique notamment pourquoi il est efficace.

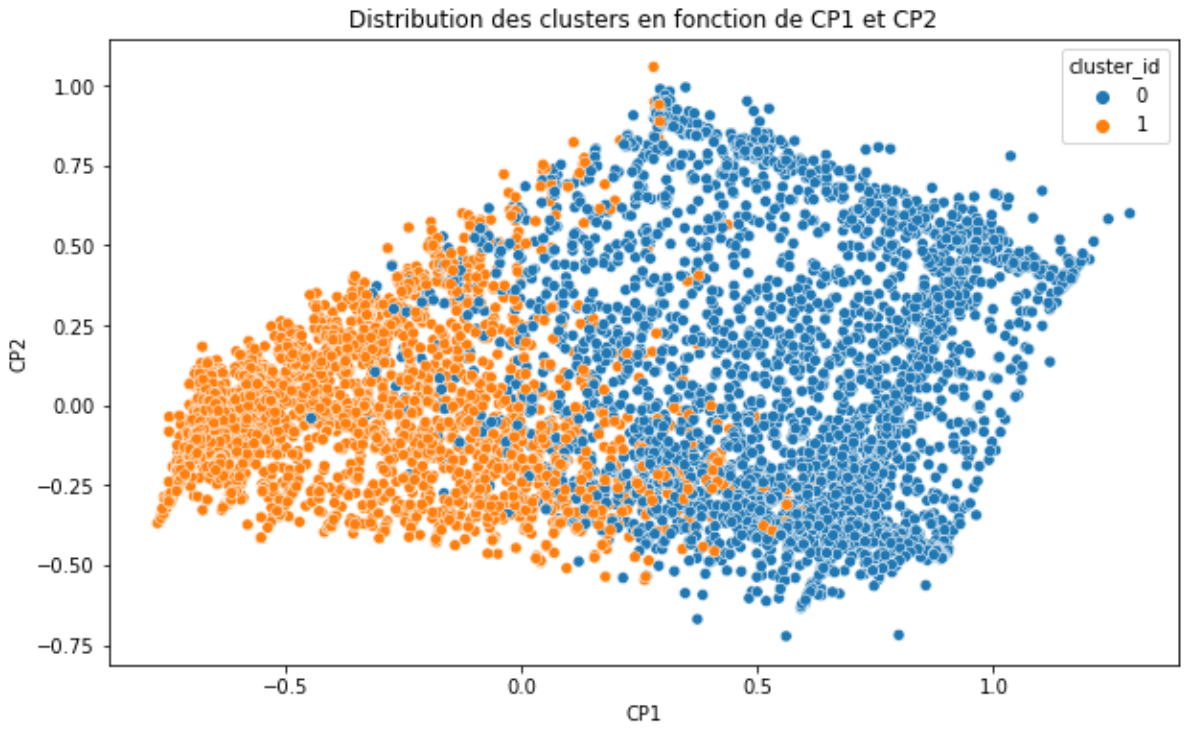
In [59]:
```
silhoutte_scores =[]
for i in tqdm(range(2,10)):
    kmeans = KMeans(n_clusters= i, init='k-means++', random_state=42)
    kmeans.fit(Y)
    silhoutte_scores.append(silhouette_score(Y, kmeans.labels_))

plt.plot(range(2,10), silhoutte_scores, "bo-")
plt.xticks(range(2,10))
plt.title('Silhoutte scores vs Number of clusters')
plt.xlabel('Number of clusters')
plt.ylabel('Silhoutte score')
plt.show()
```

```
0%|          | 0/8 [00:00<?, ?it/s]
```



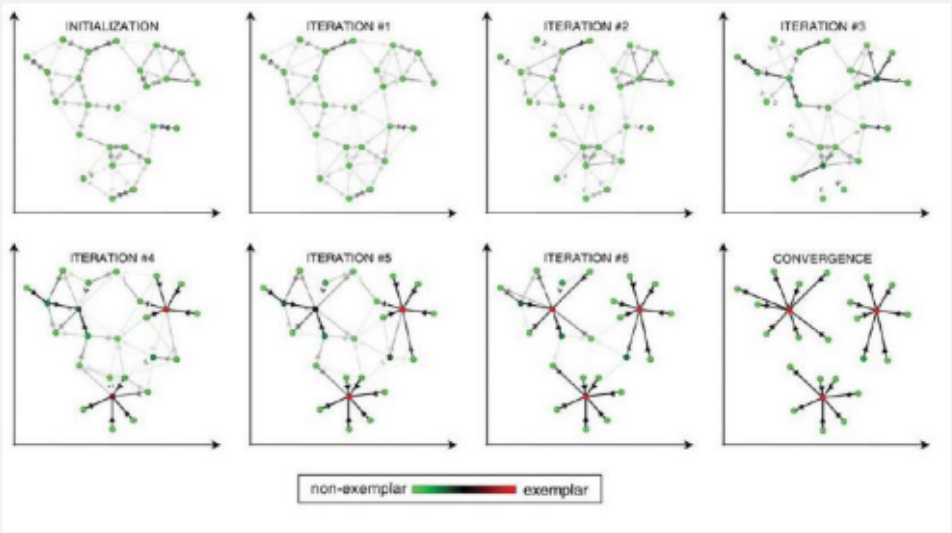
In [60]:
```
kmeans = KMeans(n_clusters=2)
kmeans.fit_predict(Y)
Y['cluster_id'] = kmeans.labels_
```

In [61]:
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = kmeans.labels_

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



MODELE DE CLUSTERING - AFFINITY PROPAGATION (similarité entre paires jusqu'à convergence)

Cet algorithme crée des clusters en envoyant des messages entre des paires d'échantillons jusqu'à convergence. Les messages sont envoyés par itérations entre les paires représentent l'aptitude d'un échantillon à être l'exemplaire de l'autre, qui est mise à jour en réponse aux valeurs des autres paires. Cette mise à jour s'effectue de manière itérative jusqu'à convergence, moment auquel les exemplaires finaux sont choisis, et donc le clustering final donné. Il est adapté dans les cas de nombreux clusters de tailles inégales.
L'avantage de cette méthode est donc que le nombre de clusters est donné en fonction des données a posteriori



Graphique représentant le processus itéré de l'algorithme affinity propagation

In [62]:
```
X = data_credit_norm
```

In [63]:
```
Y = data_credit_norm_out
```

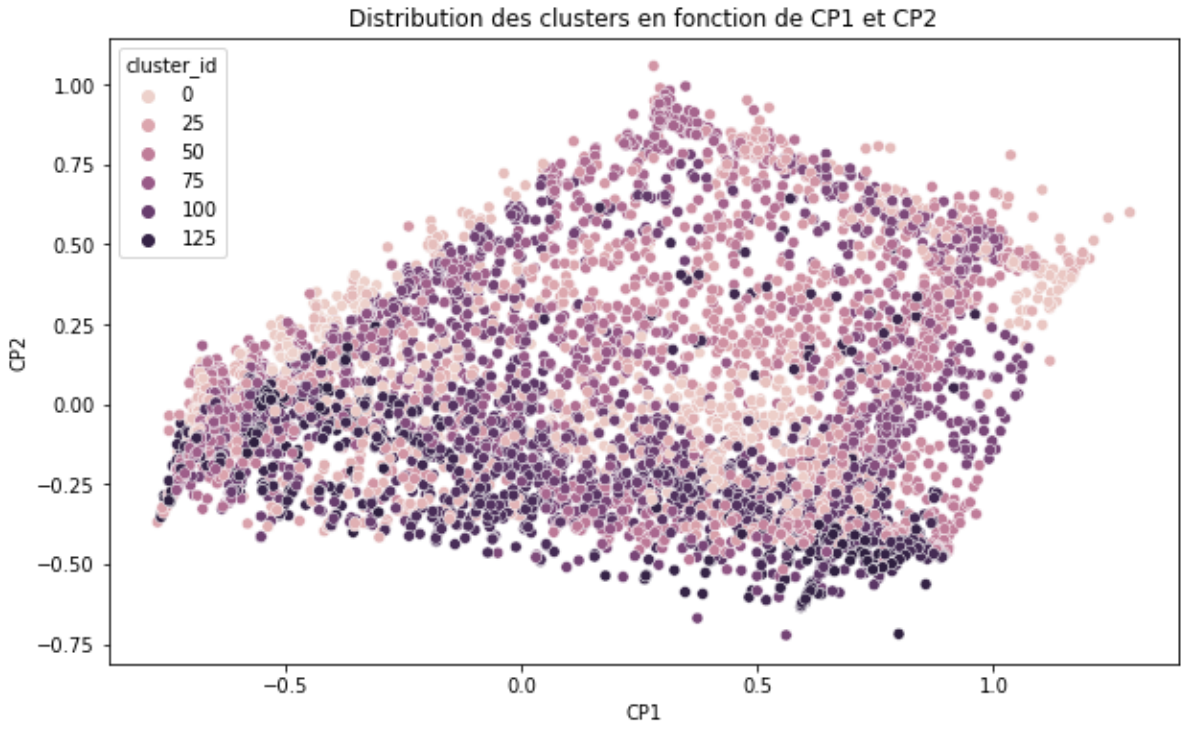
In [64]:
```
from sklearn.cluster import AffinityPropagation
```

In [65]:
```
af = AffinityPropagation(random_state=42).fit(X)
cluster_centers_indices = af.cluster_centers_indices_
n_clusters_ = len(cluster_centers_indices)
X['cluster_id'] = af.labels_
```

In [66]:
```
print("Estimated number of clusters: %d" % n_clusters_)
print("Silhouette Coefficient: %0.3f" % metrics.silhouette_score(X, af.labels_, met
```

```
Estimated number of clusters: 131
Silhouette Coefficient: 0.947
```

In [67]:
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = af.labels_

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



Nous pouvons voir que les clusters ne sont pas très bien définis dans cet espace en deux dimensions puisque une réduction de dimension a été faite pour tracer le graphe alors qu'aucune réduction n'a été faite au préalable sur les données modélisées. Il est donc normal que le coefficient de silhouette soit élevé alors que les clusters sont diffus graphiquement.

Comparons maintenant avec outliers imputés.

In [68]:
```
af = AffinityPropagation(random_state=42).fit(Y)
cluster_centers_indices = af.cluster_centers_indices_
n_clusters_ = len(cluster_centers_indices)
Y['cluster_id'] = af.labels_
```

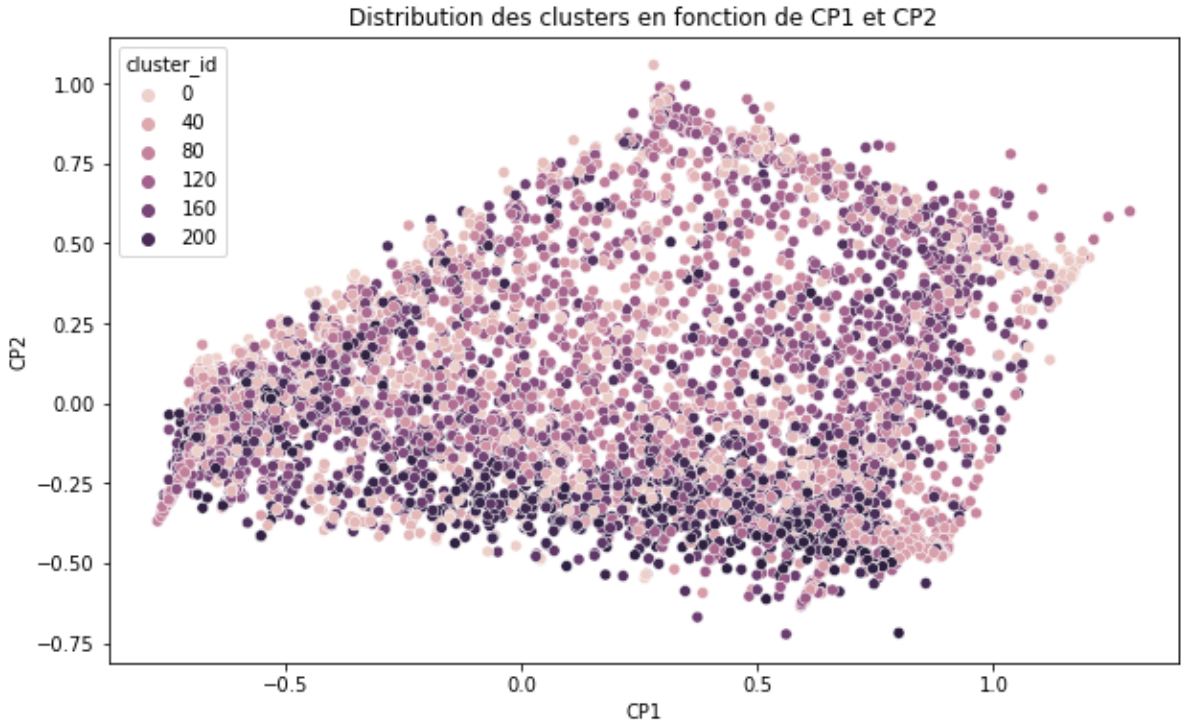
In [69]:
```
print("Estimated number of clusters: %d" % n_clusters_)
print("Silhouette Coefficient: %0.3f" % metrics.silhouette_score(Y, af.labels_, met
```

```
Estimated number of clusters: 225
Silhouette Coefficient: 0.870
```

On remarque qu'avec imputation des outliers le nombre de clusters est plus élevé et la performance diminue légèrement. Cela peut être dû au fait que les groupes sont plus diffus entre eux : les distances entre clusters sont moins élevées et les groupes moins homogènes car l'imputation est "forcée", d'où le coefficient de silhouette moindre.

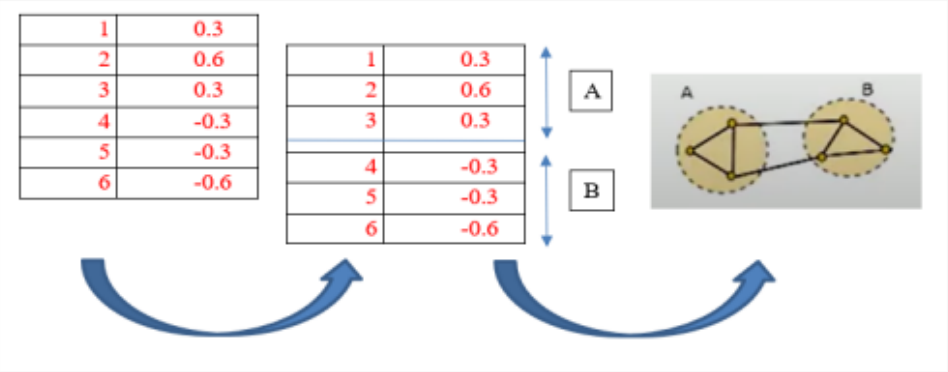
In [70]:
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = af.labels_

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
```

```
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



MODELE DE CLUSTERING - SPECTRAL CLUSTERING

L'algorithme consiste à tout d'abord construire une matrice représentant le graphique des points (grâce à des liaisons entre les points, que l'on appelle des nœuds) : il s'agit d'une matrice de Laplace. Elle illustre le nombre de connexions (nœuds) entre les différents points. L'algorithme cherche les valeurs propres et vecteurs propres de celle-ci et retient la plus petite valeur propre ainsi que le vecteur propre associé à celle-ci. Il s'agit ensuite de découper le vecteur propre réduit de dimension (n,1) en deux pour créer deux groupes (ou clusters). Ce qui détermine le point de découpage peut être la médiane des valeurs ou bien zéro pour avoir d'un côté les valeurs de la matrice négatives et de l'autre les positives. Pour créer plus de deux clusters (que l'on peut noter A et B), on peut partitionner en deux puis repartitionner autant de fois que souhaité en fonction du nombre de clusters désiré. Il est également possible de travailler avec plusieurs vecteurs propres.
Le nombre de clusters doit être spécifié à l'avance. Il fonctionne bien pour un petit nombre de clusters mais n'est pas conseillé pour de nombreux clusters. Cet algorithme est adapté pour des bases de données de grande dimension, ce qui est notre cas. Il a également l'avantage de ne pas faire d'hypothèses trop contraignantes sur la forme de clusters et aussi de pouvoir gérer des variables catégorielles (ici, 'TENURE' est une variable catégorielle). En revanche, cet algorithme est particulièrement lent à faire tourner.



Graphique représentant le processus itéré de l'algorithme spectral clustering

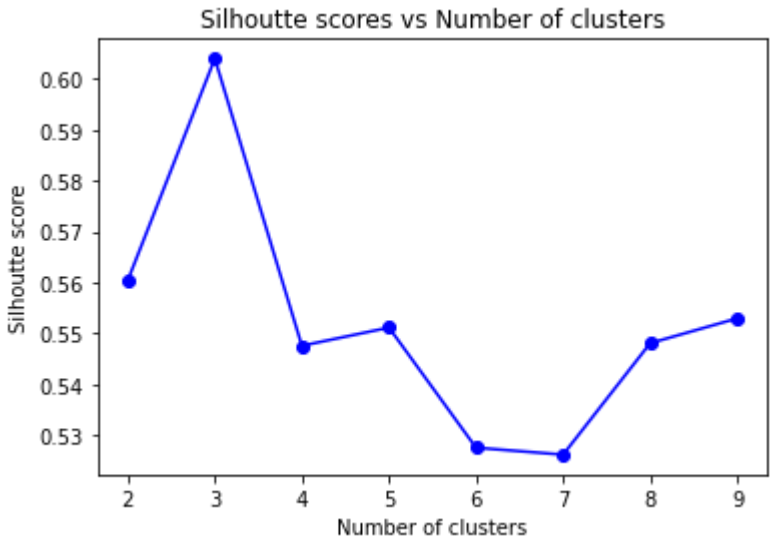
In [71]:
```
X = data_credit_norm
```

In [72]:
```
Y = data_credit_norm_out
```

In [73]:
```
from sklearn.cluster import SpectralClustering
```

In [74]:
```
silhoutte_scores =[]
for i in tqdm(range(2,10)):
    sc_model = SpectralClustering(n_clusters= i, random_state=42)
    sc_model.fit(X)
    silhoutte_scores.append(silhouette_score(X, sc_model.labels_))

plt.plot(range(2,10), silhoutte_scores, "bo-")
plt.xticks(range(2,10))
plt.title('Silhoutte scores vs Number of clusters')
plt.xlabel('Number of clusters')
plt.ylabel('Silhoutte score')
plt.show()
```

```
  0%|          | 0/8 [00:00<?, ?it/s]
```

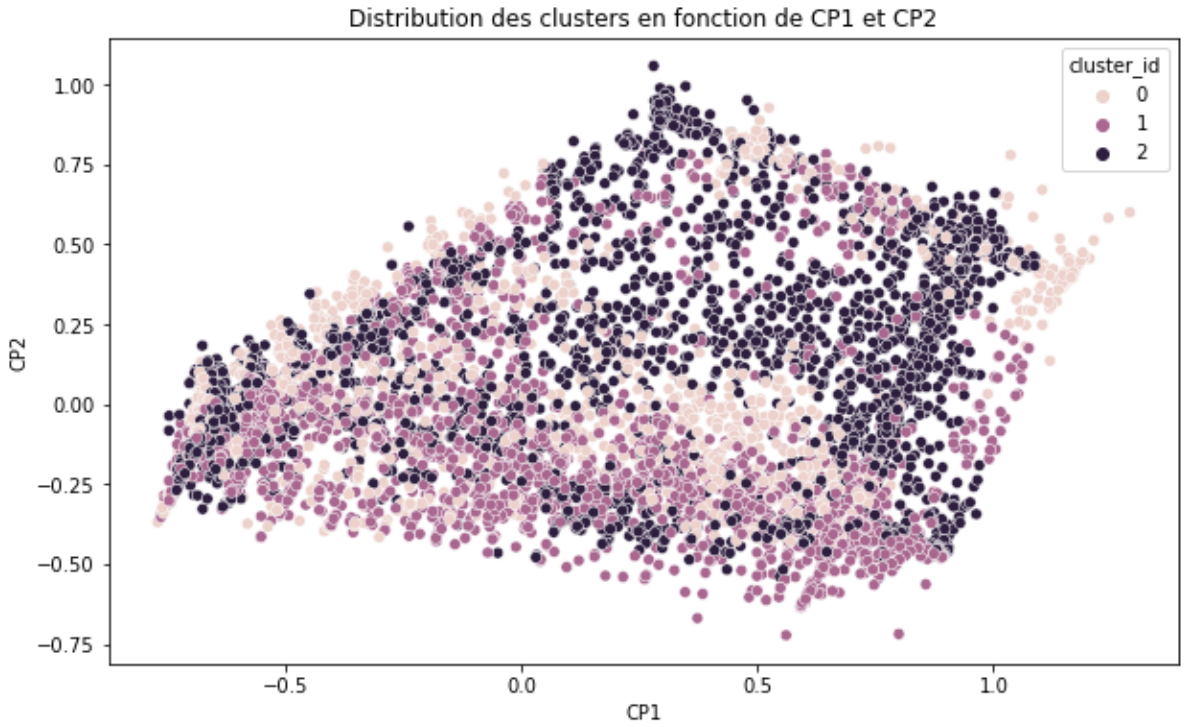


On peut voir la lenteur de l'algorithme à tourner par son temps de chargement (plus de 30

minutes!).

In [76]:
```
sc_model = SpectralClustering(n_clusters=3)
sc_model.fit_predict(X)
X['cluster_id'] = sc_model.labels_
```

In [77]:
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = sc_model.labels_

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



Comparons maintenant avec les résultats obtenus avec des outliers imputés.

In [ ]:
```
silhoutte_scores =[]
for i in tqdm(range(2,10)):
    sc_model = SpectralClustering(n_clusters= i, random_state=42)
    sc_model.fit(Y)
    silhoutte_scores.append(silhouette_score(Y, sc_model.labels_))

plt.plot(range(2,10), silhoutte_scores, "bo-")
plt.xticks(range(2,10))
plt.title('Silhoutte scores vs Number of clusters')
plt.xlabel('Number of clusters')
plt.ylabel('Silhoutte score')
plt.show()
```

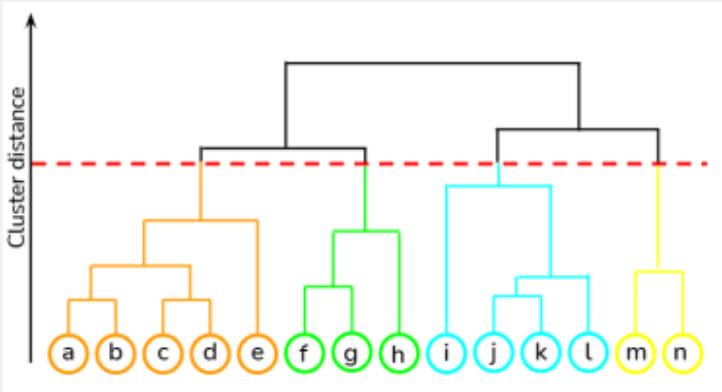
L'algorithme peine à tourner (lenteur extrême).

In [ ]:
```
sc_model = SpectralClustering(n_clusters=2)
sc_model.fit_predict(Y)
Y['cluster_id'] = sc_model.labels_
```

In [ ]:
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = sc_model.labels_

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```

MODELE DE CLUSTERING - AGGLOMERATIVE (HIERARCHICAL) CLUSTERING

Il s'agit d'un type d'algorithme appartenant au hierarchical clustering. Le hierarchical clustering construit des clusters imbriqués en les fusionnant ou en les divisant successivement. Cette hiérarchie de clusters est représentée sous la forme d'un arbre (ou dendrogramme). Les individus sont représentés en abscisse et on retrouve la distance entre les points en ordonnée. L'agglomerative clustering effectue un clustering hiérarchique en utilisant une approche ascendante : chaque observation commence dans son propre cluster, et les clusters sont successivement fusionnés ensemble.
Cet algorithme est adapté pour des situations où il y a des données numériques et catégorielles. Dans nos données, nous avons une variable catégorielle en même temps. Cette méthode nécessite de spécifier un nombre de clusters en amont. Cet algorithme est sensible aux outliers, nous chercherons à le démontrer.



Graphique représentant le processus itéré de l'algorithme agglomerative clustering

In [78]:
```
X = data_credit_norm
```

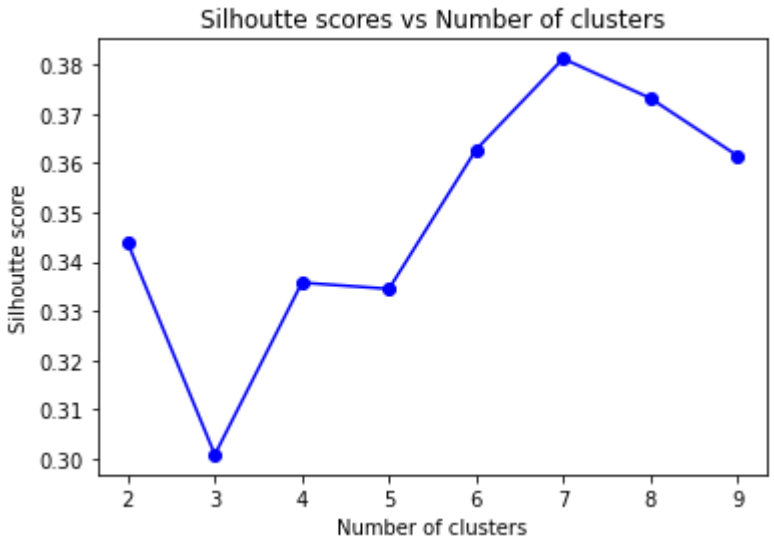
In [79]:
```
Y = data_credit_norm_out
```

In [80]:
```
from sklearn.cluster import AgglomerativeClustering
```

```
from scipy.cluster.hierarchy import dendrogram, linkage
import scipy.cluster.hierarchy as hier
```

In [81]:
```
silhoutte_scores =[]
for i in tqdm(range(2,10)):
    ag_model = AgglomerativeClustering(n_clusters = i)
    ag_model.fit(X)
    silhoutte_scores.append(silhouette_score(X,ag_model.labels_))

plt.plot(range(2,10), silhoutte_scores, "bo-")
plt.xticks(range(2,10))
plt.title('Silhoutte scores vs Number of clusters')
plt.xlabel('Number of clusters')
plt.ylabel('Silhoutte score')
plt.show()
```

```
  0%|          | 0/8 [00:00<?, ?it/s]
```

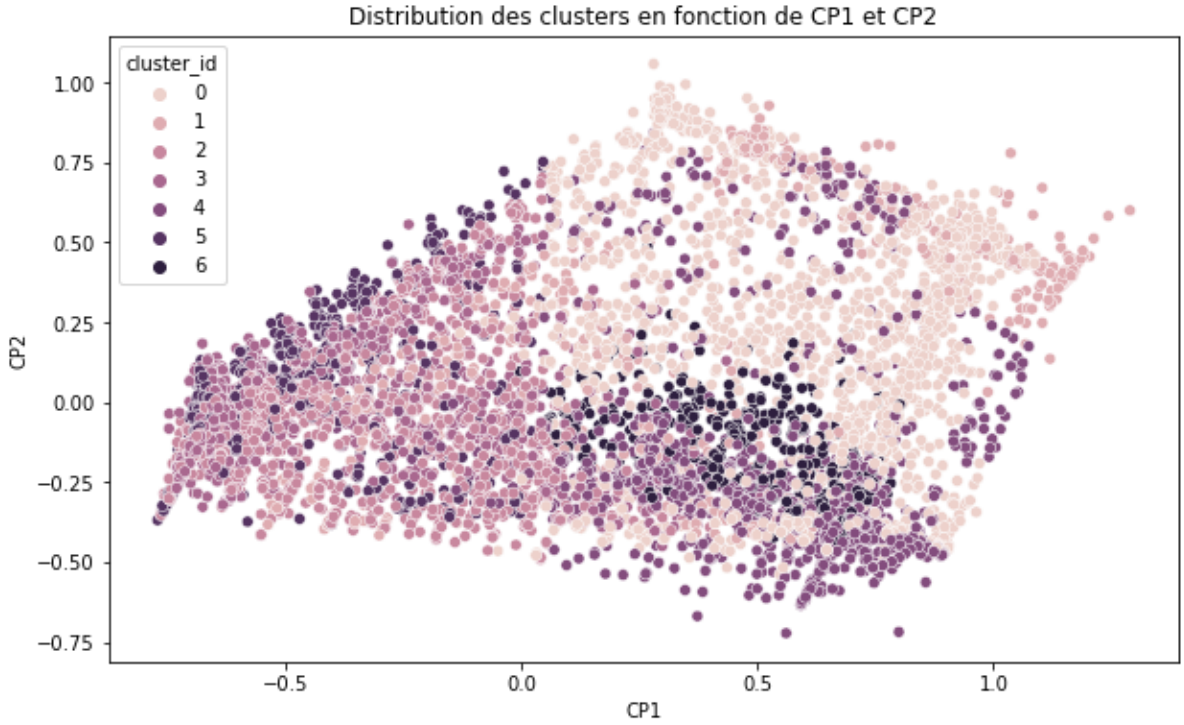


In [82]:
```
ag_model = AgglomerativeClustering(n_clusters=7)
ag_model.fit(X)
X['cluster_id'] = ag_model.labels_
```

In [83]:
```
print("Silhouette Coefficient: %0.3f" % metrics.silhouette_score(X, ag_model.labels
```

```
Silhouette Coefficient: 0.484
```

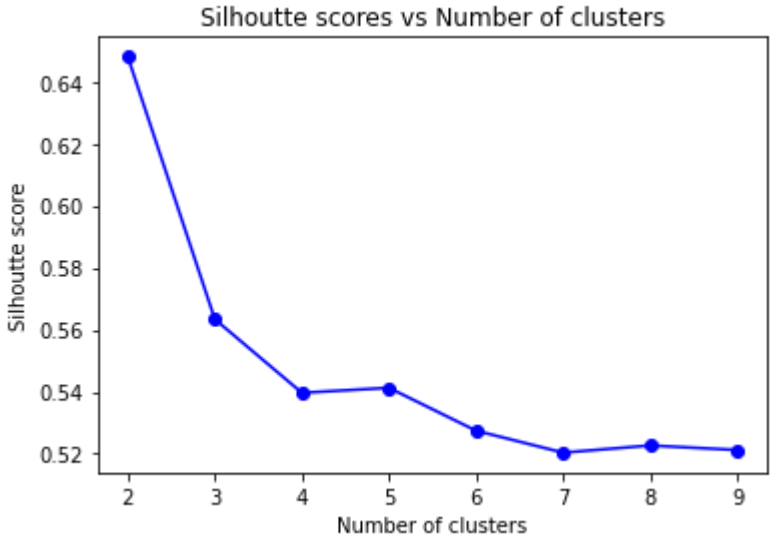
In [85]:
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = ag_model.labels_

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



Comparons maintenant avec les résultats obtenus avec des outliers imputés.

In [86]:
```
silhoutte_scores =[]
for i in tqdm(range(2,10)):
    ag_model = AgglomerativeClustering(n_clusters = i)
    ag_model.fit(Y)
    silhoutte_scores.append(silhouette_score(Y,ag_model.labels_))

plt.plot(range(2,10), silhoutte_scores, "bo-")
plt.xticks(range(2,10))
plt.title('Silhoutte scores vs Number of clusters')
plt.xlabel('Number of clusters')
plt.ylabel('Silhoutte score')
plt.show()
```

```
  0%|          | 0/8 [00:00<?, ?it/s]
```

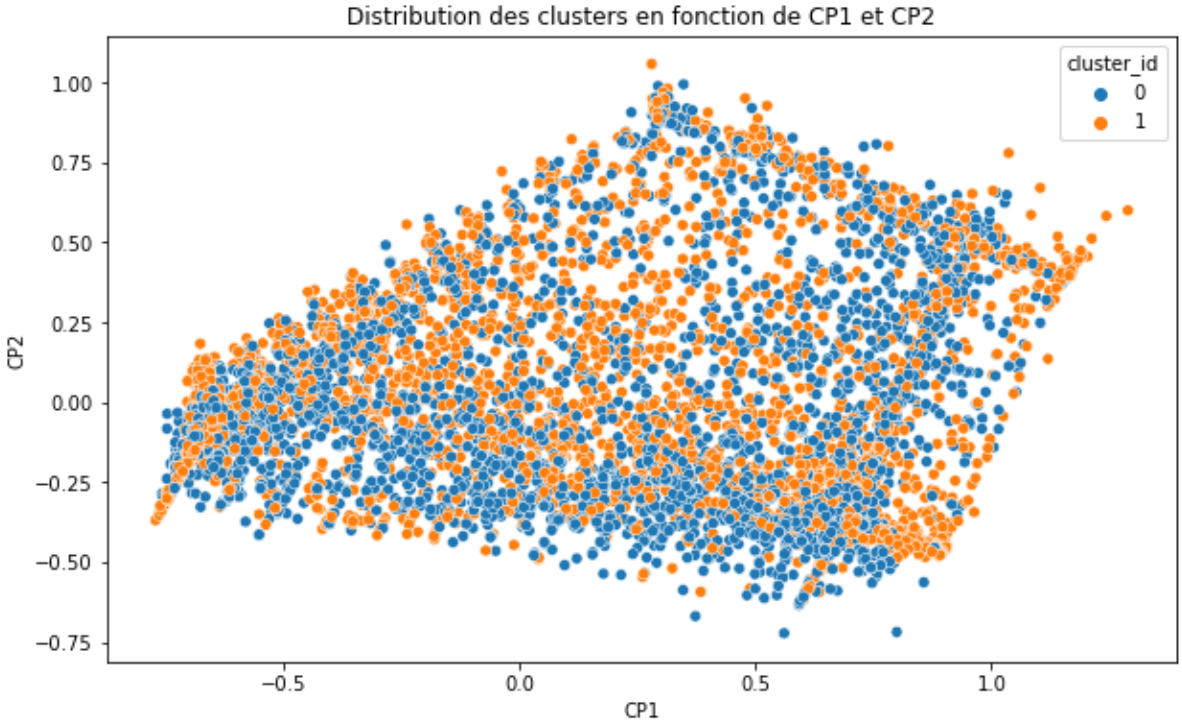


In [87]:
```
ag_model = AgglomerativeClustering(n_clusters=2)
ag_model.fit(Y)
Y['cluster_id'] = ag_model.labels_
```

In [88]:
```
print("Silhouette Coefficient: %0.3f" % metrics.silhouette_score(Y, ag_model.labels
```

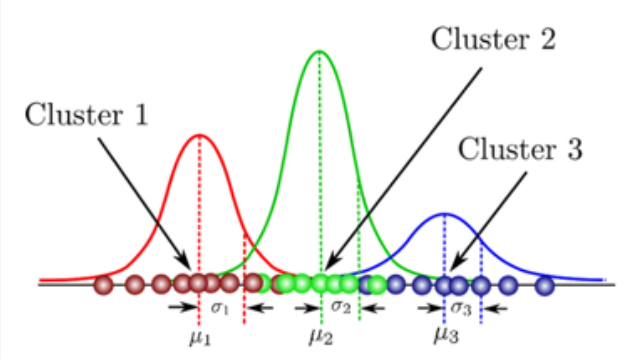
```
Silhouette Coefficient: 0.139
```

In [90]:
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = ag_model.labels_

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



MODELE DE CLUSTERING - GAUSSIANMIXTURE

Un modèle de mélange gaussien est un modèle probabiliste qui suppose que tous les points de données sont générés à partir d'un mélange d'un nombre fini de distributions gaussiennes avec des paramètres inconnus. La variance, la moyenne et l'amplitude de chaque gaussienne sont optimisés selon un critère de maximum de vraisemblance (processus d'espérance-maximisation itéré). Chaque cluster est modélisé par des densités de mélange de gaussiennes.
Une donnée peut donc appartenir à plusieurs clusters avec des probabilités d'appartenance, ce qui est approprié dans notre cas car nous avons de nombreuses données (observations et variables). Ce modèle permet ainsi de ne pas biaiser la taille et la forme des clusters, ceux-ci pouvant se chevaucher.



Graphique représentant le processus itéré du modèle de mélange gaussien

In [92]:
```
X = data_credit_norm
```

In [93]:
```
Y = data_credit_norm_out
```

Pour le GMM, nous pouvons établir une sélection de modèle grâce au BIC. La sélection va nous permettre de mieux choisir le type de covariance ainsi que le nombre de composantes.

In [94]:
```
from sklearn.mixture import GaussianMixture
```

In [95]:
```
from sklearn.model_selection import GridSearchCV
def gmm_bic_score(estimator, X):
    return -estimator.bic(X)
param_grid = {
    "n_components": range(1, 7),
    "covariance_type": ["spherical", "tied", "diag", "full"],
}
grid_search = GridSearchCV(
    GaussianMixture(), param_grid=param_grid, scoring=gmm_bic_score
)
grid_search.fit(X)
```

Out[95]:
```
GridSearchCV(estimator=GaussianMixture(),
             param_grid={'covariance_type': ['spherical', 'tied', 'diag',
                                             'full'],
                         'n_components': range(1, 7)},
             scoring=<function gmm_bic_score at 0x000001B9E840C9D0>)
```

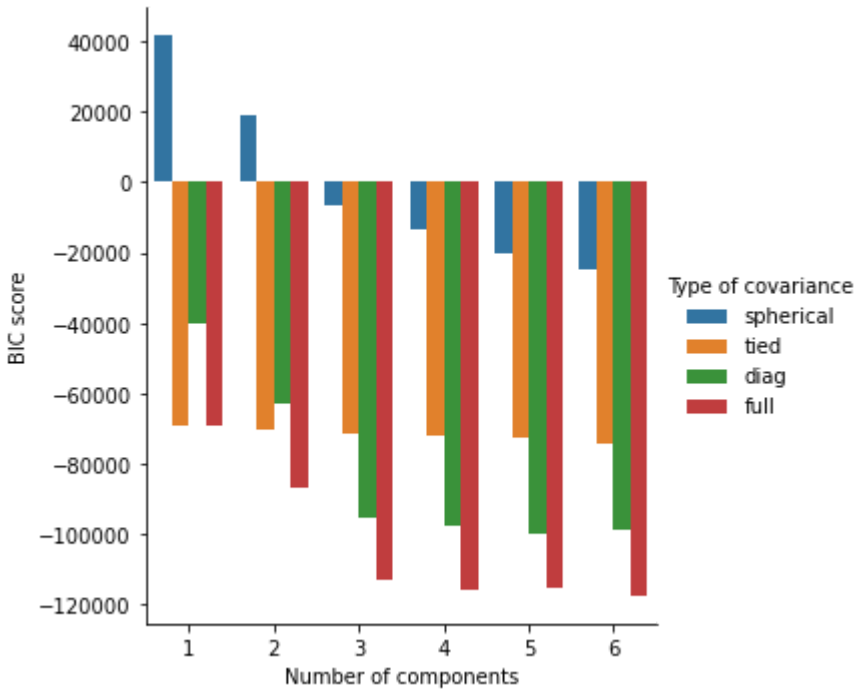
In [96]:
```
import pandas as pd
df = pd.DataFrame(grid_search.cv_results_)[
    ["param_n_components", "param_covariance_type", "mean_test_score"]
]
df["mean_test_score"] = -df["mean_test_score"]
df = df.rename(
    columns={
        "param_n_components": "Number of components",
        "param_covariance_type": "Type of covariance",
        "mean_test_score": "BIC score",
    }
```

```
)
df.sort_values(by="BIC score").head()
```

Out[96]:

	Number of components	Type of covariance	BIC score
23	6	full	-117376.073640
21	4	full	-115757.416172
22	5	full	-115390.997704
20	3	full	-112930.682146
16	5	diag	-99875.182514

In [97]:
```
sns.catplot(
    data=df,
    kind="bar",
    x="Number of components",
    y="BIC score",
    hue="Type of covariance",
)
plt.show()
```

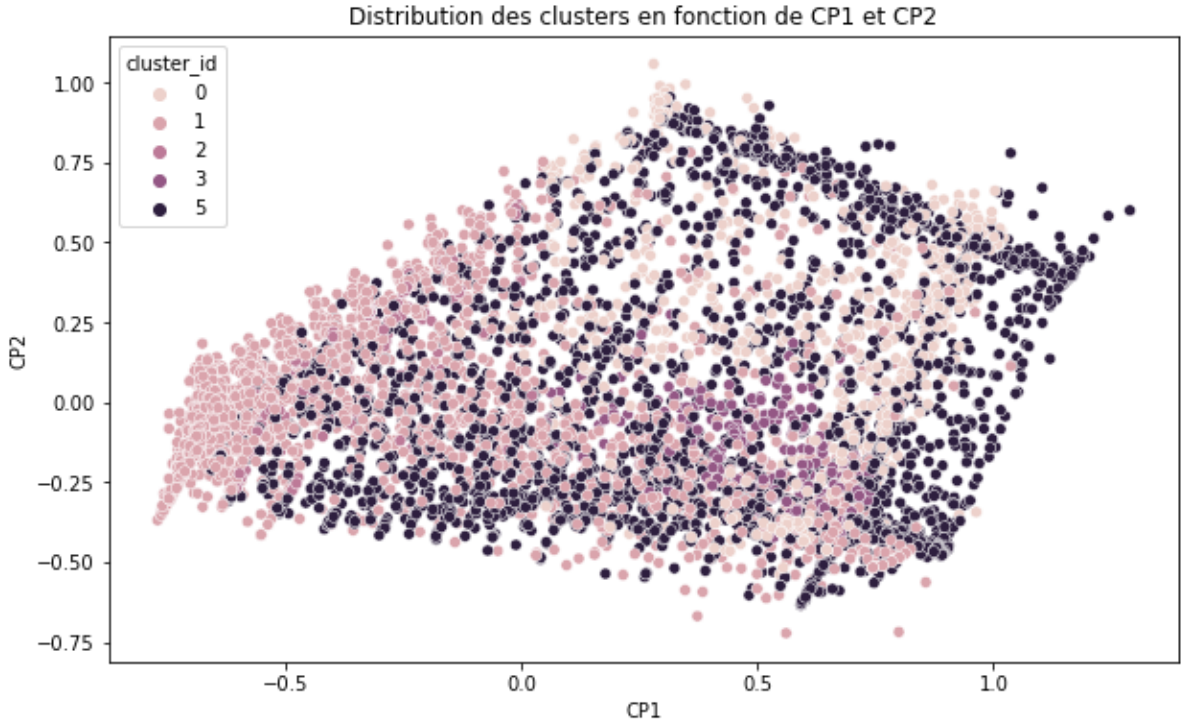


On voit que le score BIC est le plus petit pour le type de covariance "full" (cas où chaque composante a sa propre matrice de covariance), et avec 6 composantes (qui seront nos clusters dans ce cadre).

In [98]:
```
gmm=GaussianMixture(n_components=6,covariance_type='full',random_state = 42)
gmm.fit(X)
X['cluster_id'] = gmm.predict(X)
```

In [99]:
```
plt.figure(figsize=(10,6))
```

```
data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = gmm.predict(X)

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



In [100…
```
metrics.silhouette_score(X,gmm.predict(X))
```

Out[100]:
```
0.18039061859641004
```

Le score est bas. On pourrait alors penser à faire une ACP sur les données avant de les modéliser par un modèle de mélange gaussien.

Comparons maintenant avec les résultats obtenus avec des outliers imputés.

In [101…
```
from sklearn.model_selection import GridSearchCV
def gmm_bic_score(estimator, Y):
    return -estimator.bic(Y)
param_grid = {
    "n_components": range(1, 7),
    "covariance_type": ["spherical", "tied", "diag", "full"],
}
grid_search = GridSearchCV(
    GaussianMixture(), param_grid=param_grid, scoring=gmm_bic_score
)
grid_search.fit(Y)
```

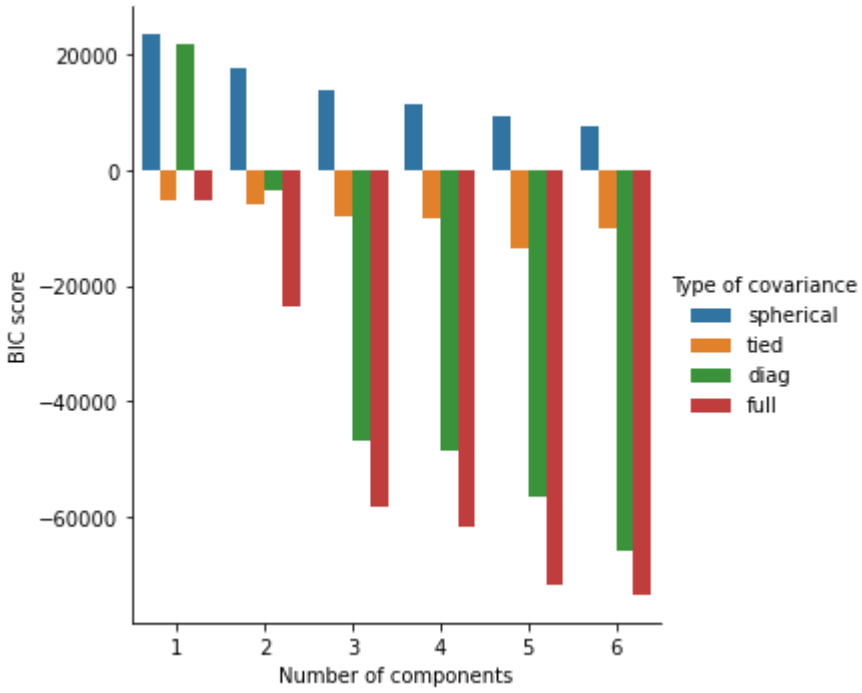
Out[101]:
```
GridSearchCV(estimator=GaussianMixture(),
             param_grid={'covariance_type': ['spherical', 'tied', 'diag',
                                             'full'],
                         'n_components': range(1, 7)},
             scoring=<function gmm_bic_score at 0x000001B9E840C8B0>)
```

In [102…
```
import pandas as pd
df = pd.DataFrame(grid_search.cv_results_)[
    ["param_n_components", "param_covariance_type", "mean_test_score"]
]
df["mean_test_score"] = -df["mean_test_score"]
df = df.rename(
    columns={
        "param_n_components": "Number of components",
        "param_covariance_type": "Type of covariance",
        "mean_test_score": "BIC score",
    }
)
df.sort_values(by="BIC score").head()
```

Out[102]:

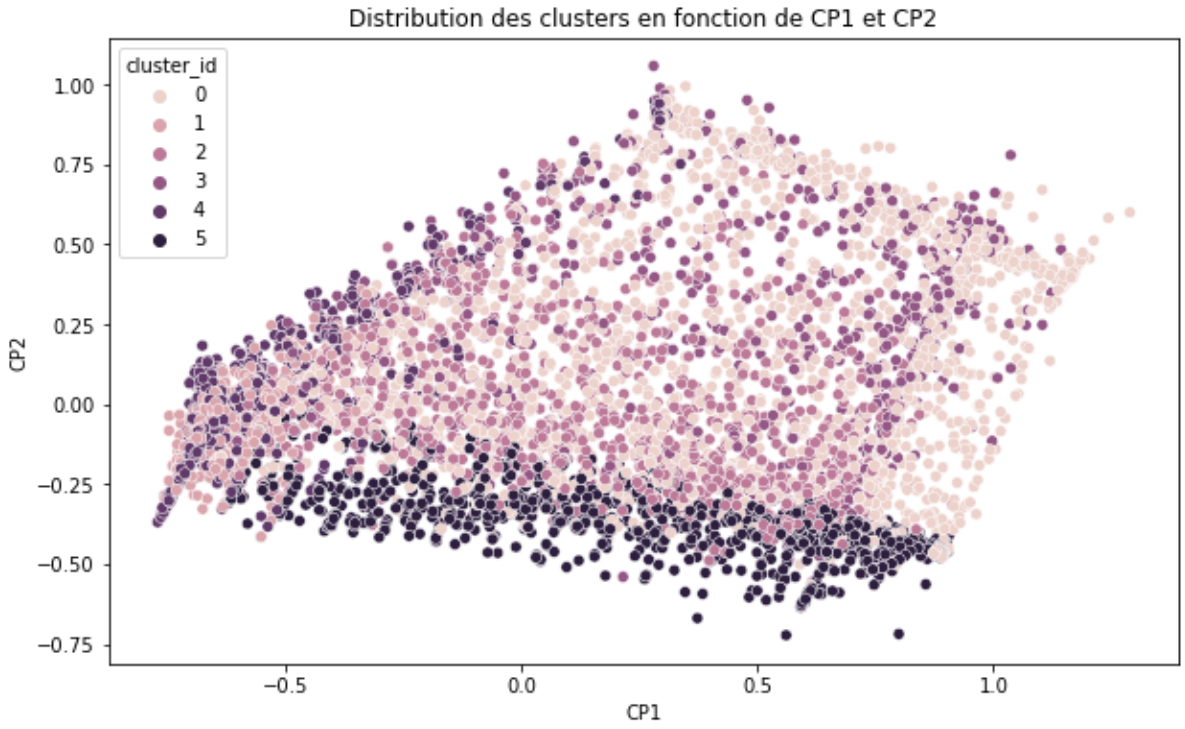
	Number of components	Type of covariance	BIC score
23	6	full	-73535.677372
22	5	full	-71759.130006
17	6	diag	-65910.497219
21	4	full	-61892.089244
20	3	full	-58319.978170

In [103…
```
sns.catplot(
    data=df,
    kind="bar",
    x="Number of components",
    y="BIC score",
    hue="Type of covariance",
)
plt.show()
```



In [104…
```
gmm=GaussianMixture(n_components=6,covariance_type='full',random_state = 42)
gmm.fit(Y)
X['cluster_id'] = gmm.predict(Y)
```

In [105…
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = gmm.predict(Y)

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



In [106…
```
metrics.silhouette_score(Y,gmm.predict(Y))
```

Out[106]:
```
0.07683708601322455
```

Même remarque que pour le cas des données sans imputation des outliers. Le score est encore plus bas.

In [107…
```
from sklearn.decomposition import PCA
pca = PCA()
data_credit_norm = pca.fit_transform(data_credit_norm)
print(pca.explained_variance_ratio_.cumsum())
```

```
[0.84014749 0.91995717 0.9413188  0.9617199  0.97425299 0.98559136
 0.99145324 0.99436531 0.99655254 0.99762777 0.99853109 0.99899733
 0.99930215 0.99951008 0.99970839 0.99988184 0.99999999 1.        ]
```

In [108…
```
X = data_credit_norm
```

In [109…
```
from sklearn.model_selection import GridSearchCV
def gmm_bic_score(estimator, X):
    return -estimator.bic(X)
param_grid = {
    "n_components": range(1, 7),
    "covariance_type": ["spherical", "tied", "diag", "full"],
}
grid_search = GridSearchCV(
    GaussianMixture(), param_grid=param_grid, scoring=gmm_bic_score
)
grid_search.fit(X)
```

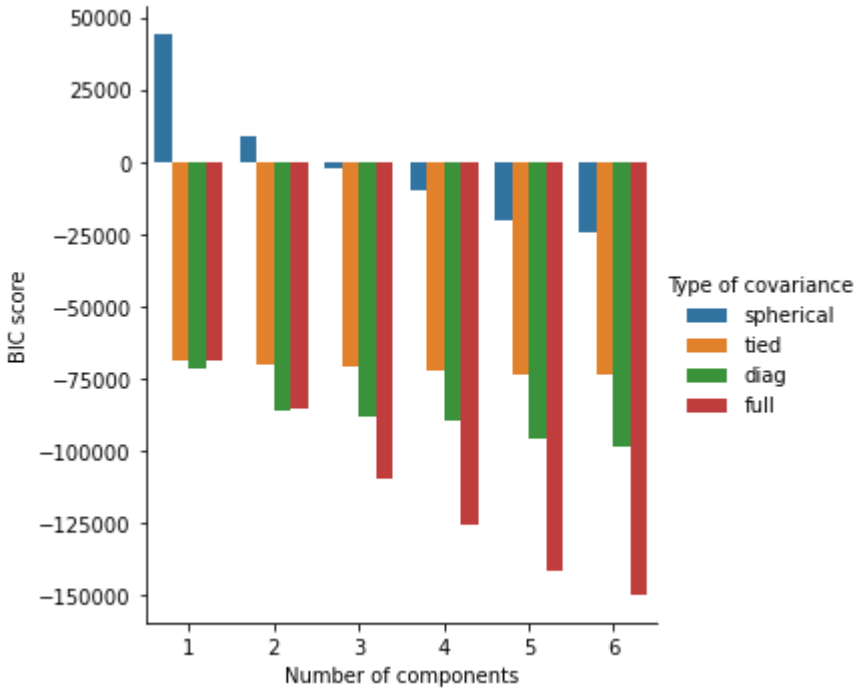
Out[109]:
```
GridSearchCV(estimator=GaussianMixture(),
             param_grid={'covariance_type': ['spherical', 'tied', 'diag',
                                             'full'],
                         'n_components': range(1, 7)},
             scoring=<function gmm_bic_score at 0x000001B9E3243F70>)
```

In [110…
```
import pandas as pd
df = pd.DataFrame(grid_search.cv_results_)[
    ["param_n_components", "param_covariance_type", "mean_test_score"]
]
df["mean_test_score"] = -df["mean_test_score"]
df = df.rename(
    columns={
        "param_n_components": "Number of components",
        "param_covariance_type": "Type of covariance",
        "mean_test_score": "BIC score",
    }
)
df.sort_values(by="BIC score").head()
```

Out[110]:

	Number of components	Type of covariance	BIC score
23	6	full	-149695.327773
22	5	full	-141464.517493
21	4	full	-125465.329909
20	3	full	-109492.935018
17	6	diag	-98636.226801

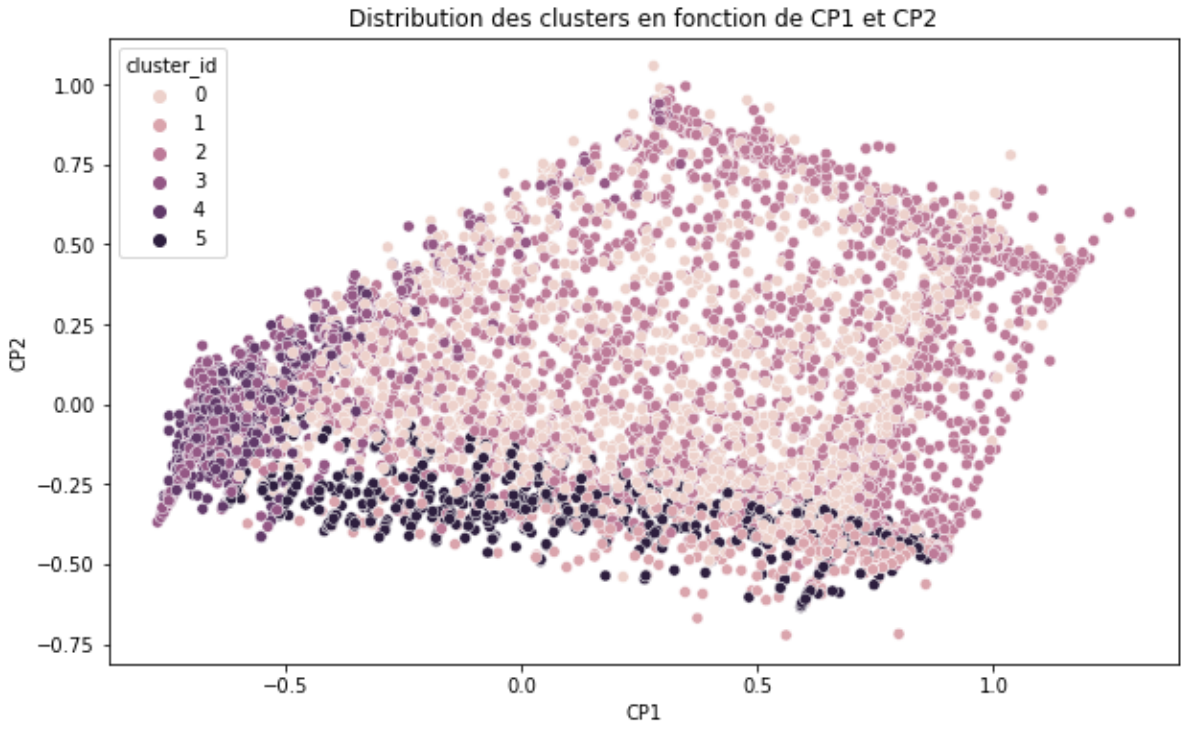
In [111…
```
sns.catplot(
    data=df,
    kind="bar",
    x="Number of components",
    y="BIC score",
    hue="Type of covariance",
)
plt.show()
```



In [114…
```
gmm=GaussianMixture(n_components=6,covariance_type='full',random_state = 42)
gmm.fit(X)
```

Out[114]:
```
GaussianMixture(n_components=6, random_state=42)
```

In [115…
```
plt.figure(figsize=(10,6))

data_PCA_plot = pd.DataFrame(data_PCA_norm[:,0:2], columns=['CP1', 'CP2'])
data_PCA_plot['cluster_id'] = gmm.predict(X)

sns.scatterplot(data=data_PCA_plot, x='CP1', y='CP2', hue='cluster_id')
plt.title('Distribution des clusters en fonction de CP1 et CP2')
plt.show()
```



In [116…
```
metrics.silhouette_score(X,gmm.predict(X))
```

Out[116]:
```
0.39886466772683987
```

On voit que la performance a nettement augmenté en ayant effectué une ACP sur les données avant de les modéliser par GMM. Réduire la dimension des données peut ainsi augmenter la performance d'un GMM.

CONCLUSION

On obtient de bons résultats avec l'algorithmes AffinityPropagation qui indique un nombre élevé de clusters. Cela semble cohérent car nous travaillons avec un très grand nombre de données (environ 9000 clients). Etablir une centaine de clusters au lieu de quelques uns n'est pas dénué de sens. Nous pouvons penser que peu de clusters pour de nombreuses données amène à des regroupements superficiels donc un clustering moins pertinent. On voit là tout l'intérêt de cet algorithme : ne pas spécifier de nombre de clusters a priori. Il n'y a ainsi pas de limite au nombre de clusters ce qui permet de maximiser la performance. Néanmoins, un besoin marketing particulier peut justifier un nombre restreit de clusters. D'autre part, on remarque que les résultats sont différents en prenant en compte l'imputation des outliers. La performance diminue de manière générale avec l'imputation des outliers ce qui n'est pas non plus dénué de sens dans la mesure où les valeurs extrêmes peuvent représenter une catégorie à part entière de clients.