OPENCLASSROOM

Parcours data scientist en alternance

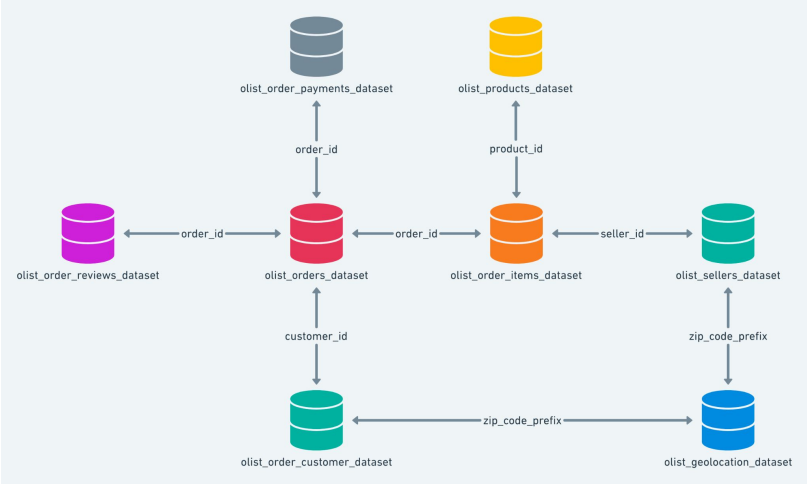
03 juillet 2022

Maxime Dupouy

Livrable 5

Missions	Objectifs
Fournir à Olist (e-commerce) une **segmentation des clients** tenant compte **des différents types d'utilisateurs et** utilisable au quotidien pour les campagnes de communication **Fournir une description** de votre segmentation et de sa logique sous-jacente, ainsi qu'une **proposition de contrat de maintenance**	- Mettre en place le modèle d'apprentissage non supervisé adapté au problème métier - Transformer les variables pertinentes - Adapter les hyperparamètres - Évaluer les performances

La source



Données commerciales réelles provenant d'Olist et hébergée sur **Kaggle.**

- 9 tables reliées par des ID
- beaucoup d'informations

Data Cleaning/engineering 🧹

Identifier les besoins

Sélectionner les colonnes ayant un intérêt marketing *(suppression des colonnes du vendeur/taille produit,...)*

Identifier la colonne id pour un client

Comprendre la logique de la base Olist

Création base de données marketing

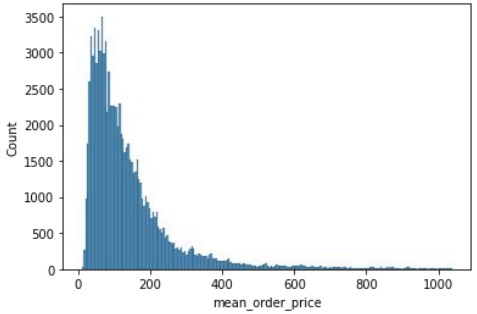
```
class CreateDataset:
    def __init__(self):
        pass

    def go(self, path):
        data = self.read_file(path)
        data['olist_order_items_dataset'] = self.pre_traitement(
            data['olist_order_items_dataset'])
        self.df = self.merging(data)
        return self.df

    def pre_traitement(self, order_items):
        '''create a new columns for total price for an order'''
        orders_value = order_items.groupby('order_id',
                                           as_index=False)[
            ['price', 'freight_value']
        ].sum()
        orders_value['total_order_value'] = orders_value[
                                            'price'] + orders_value[
                                                    'freight_value']
        orders_value = orders_value[
            ['order_id', 'total_order_value']
                                    ].drop_duplicates()
        order_items = order_items.merge(orders_value, on='order_id')
        return order_items

    def read_file(self, path) -> dict:
        data = {}
        for file_name in TO_OPEN:
            path_file = path + file_name+EXTENSION
            data[file_name] = pd.read_csv(path_file)
        return data

    def merging(self, data: dict):
        '''create data from all the datasets
        filter on order_status == delivered '''
        self.df = data['olist_orders_dataset'].merge(
                        data['olist_order_reviews_dataset'],
                        on='order_id',
                        how='left').merge(
                        data['olist_order_payments_dataset'],
                        on='order_id',
```

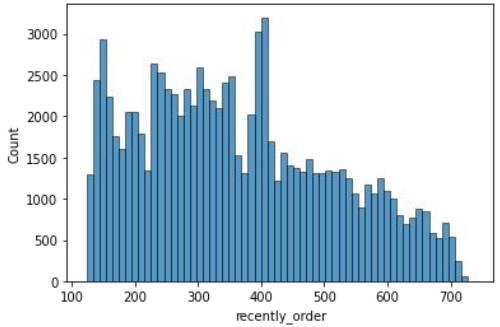
Agrégation de la donnée dans une classe, data engineering :

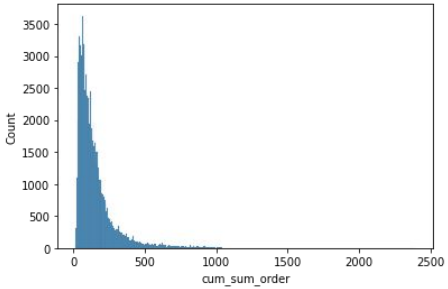
- conversion des dates en format DateTime
- moyenne des notes des commandes
- panier moyen
- somme cumulative des dépenses
- nombre de commandes effectuées
- état du client (encodage par fréquence)
- récense : nombre de jours écoulés depuis dernière commande

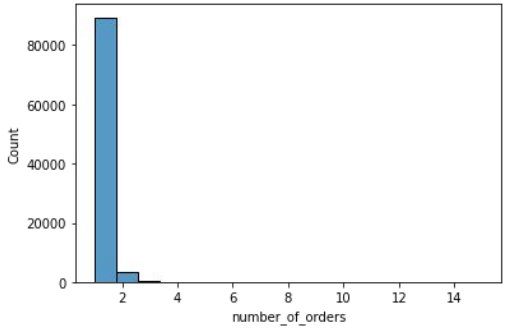
Enlever les outliers :

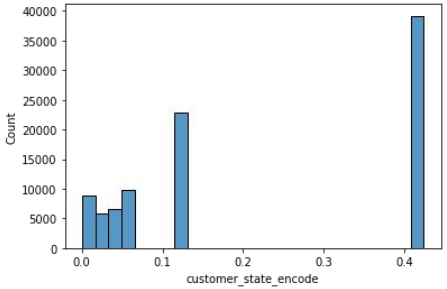
- commandes très chères
- commandes trop anciennes

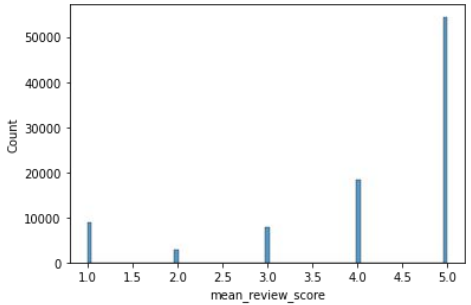




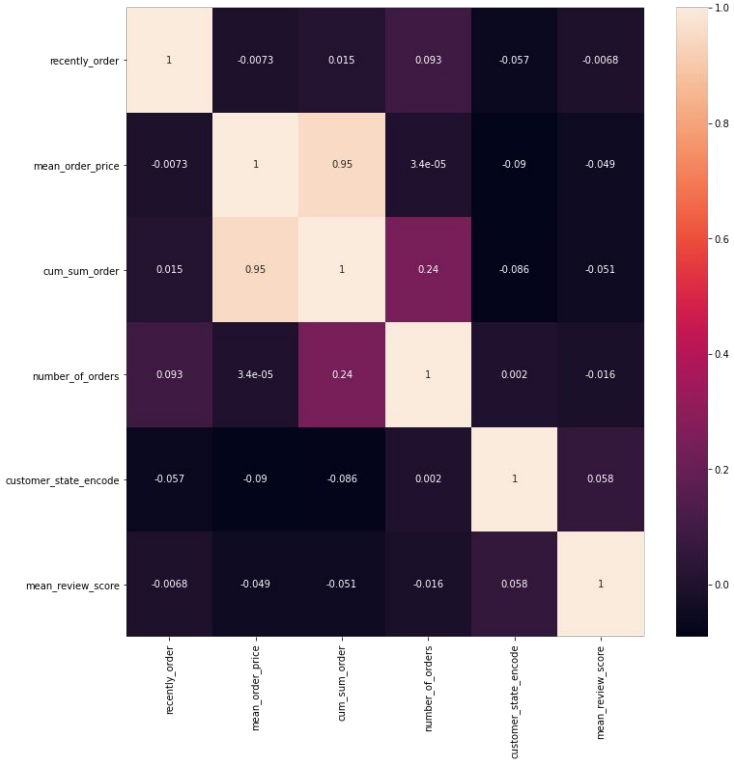








Visualisation

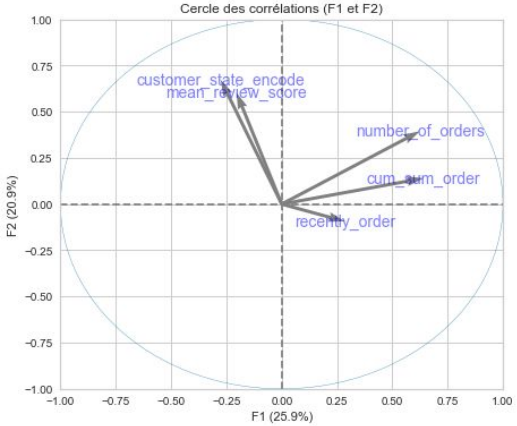


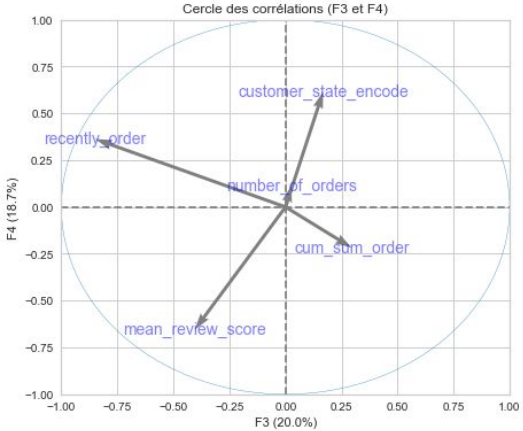
Somme cumulative et moyenne du prix des commandes trop corrélés -> suppression de la moyenne du panier moyen

Visualisation

Entrainements Modèles 

PCA

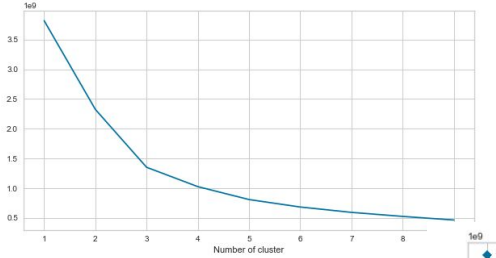


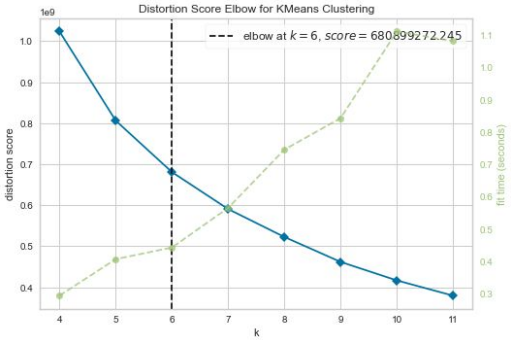


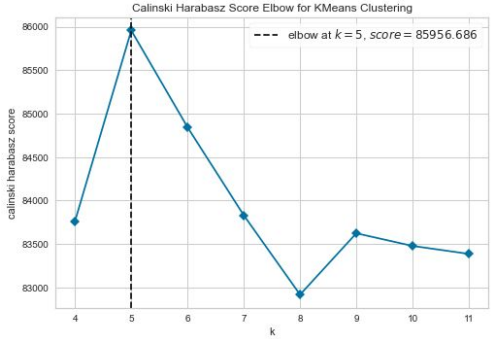
Réalisation d'une PCA :

- **comprendre l'importance des features**
- **explorer la donnée**
- **mieux la séparer**

Nombre de clusters





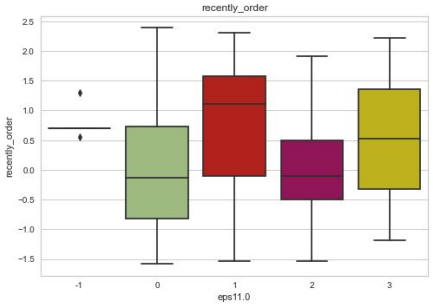


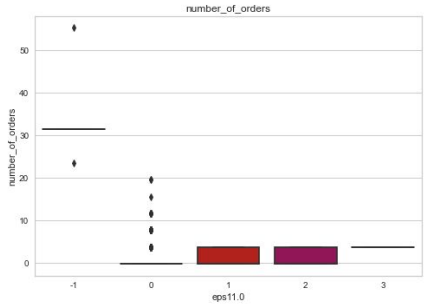
Via KMeans et méthode du coude avec différentes métriques de distance :

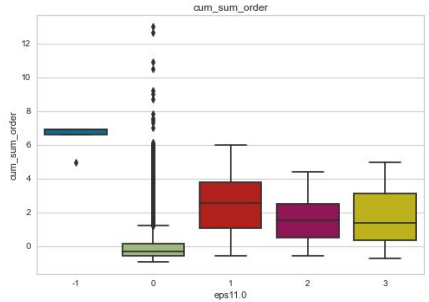
- Somme des distances au carré des échantillons à leur centre de cluster le plus proche
- distortion
- calinski harabasz

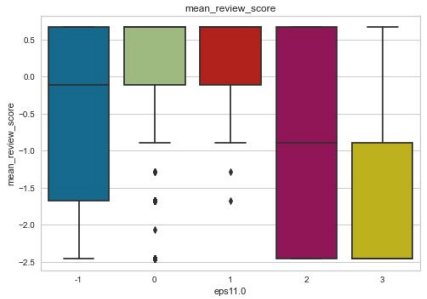
-> score retenu : 5

DBScan









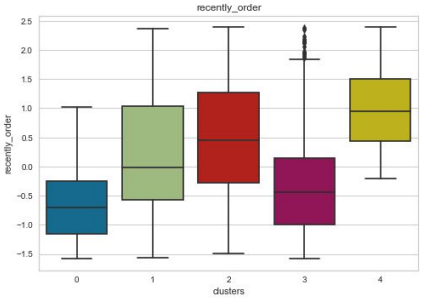
Réduction de la donnée (⅕)

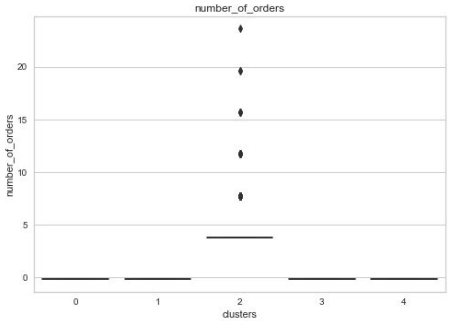
min_sample fixé à 100 et **optimisation d'Epsilon**

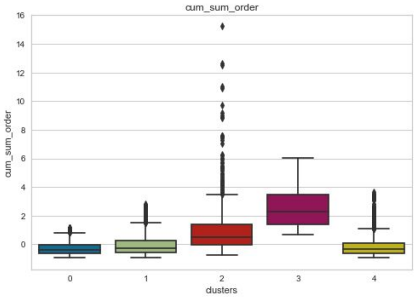
Résultats pas très satisfaisants : clusters **mal répartis** (+ peu séparés)

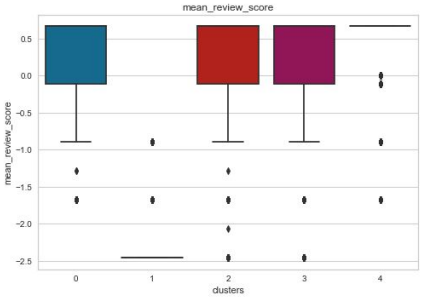
```
]:  1  X_db_scan['eps11.0'].value_counts()

 0    17936
 2      338
 1      249
 3      114
-1        5
```

Clustering Hiérarchique



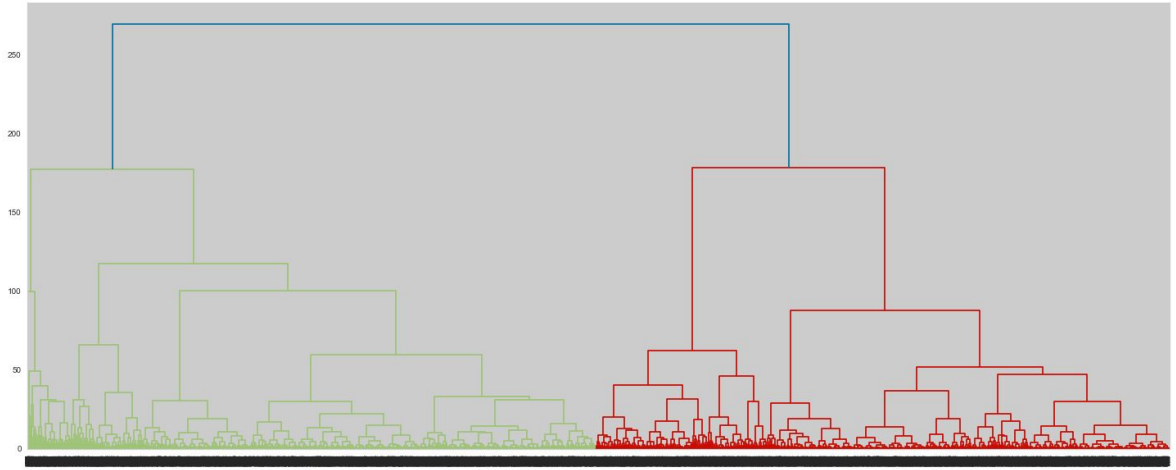






- Réduction de la donnée (⅕)
- **optimisation** de la distance_threshold pour optimiser le nombre de clusters

```
]:  1  X_hierarchcal['clusters'].value_counts()

0    8826
4    5702
1    2141
3    1204
2     769
```



Clustering Hiérarchique

```
4]:
1  a = test[['recently_order',
2            'cum_sum_order',
3            'number_of_orders',
4            'mean_review_score']]
5  b = test['clusters']
6  silhouette_score(a, b)
```

```
0.34382542934720917
```

-> bons résultats mais problème de scalabilité/performance

KMeans

Différents tests :

- features RFM (Récence, Fréquence, Monetary) et plus de features (moyenne des notes, états)
- donnée non normalisée/normalisée
- donnée provenant de la pca

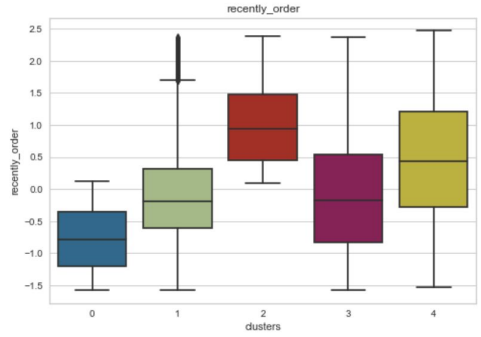
KMeans

	Standardisé	Silhouette score
RFM	non	0.4
RFM	oui	0.38
RFM + état du client + moyenne des notes	non	0.4
RFM + moyenne des notes	non	0.41
RFM + moyenne des notes	oui	0.40
Données PCA	oui	0.35

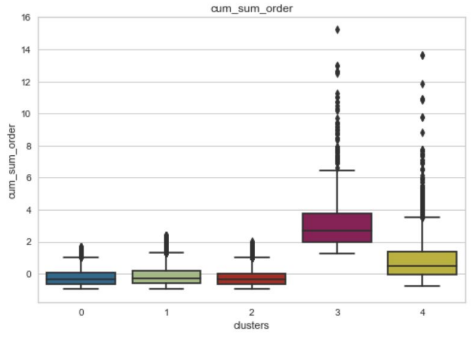
Meilleurs résultats :

- donnée normalisée (Standarscaler)
- colonnes : somme_cumulative, récense, nombre de commande, moyenne des notes

-> permet de mieux séparer la donnée en fonction des features

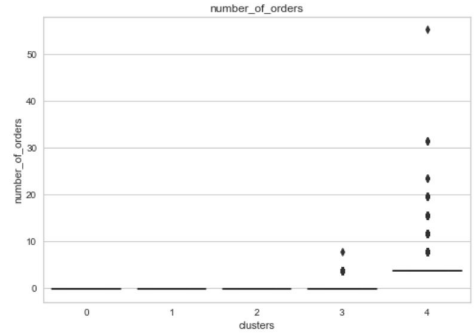



```
4 --> 0.45626389709105053
0 --> -0.7789950999255428
2 --> 1.0000079811770992
1 --> -0.06226435077267382
3 --> -0.06942881714432039
```

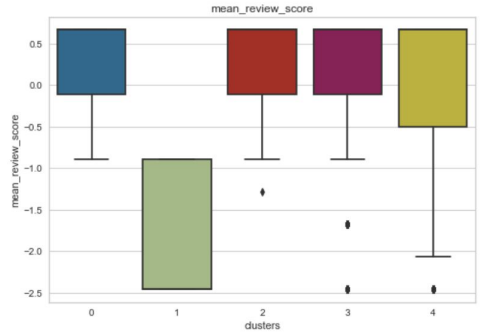



```
4 --> 0.9013508707488282
0 --> -0.23988234679509535
2 --> -0.2316313145093146
1 --> -0.1212281382287244
3 --> 3.0343194137342255
```

```
]: 1 df_k_mean_norm.clusters.value_counts()

1    39426
0    30274
2    14837
4     4893
3     3730
```



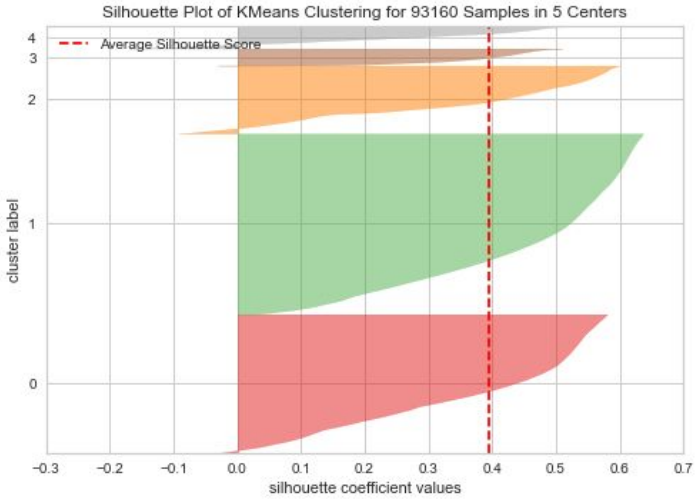

```
4 --> 4.320641355307969
0 --> -0.18653092209393043
2 --> -0.1865309220939458
1 --> -0.18653092209401428
3 --> -0.11525162228519807
```

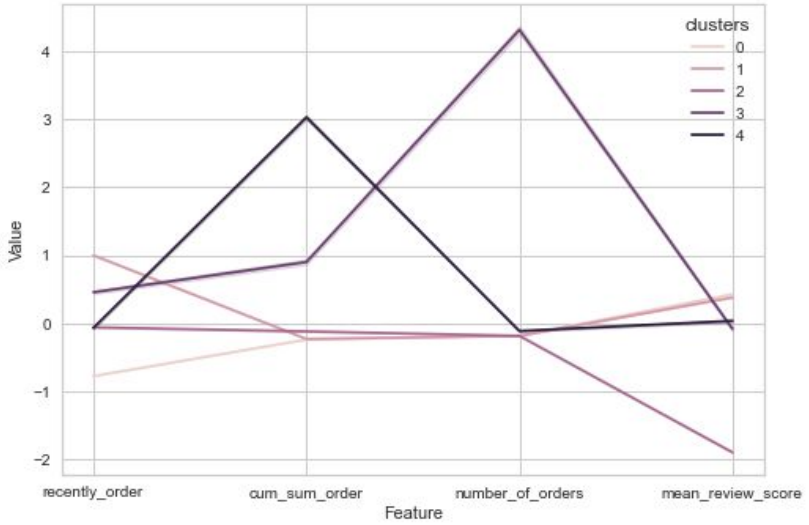



```
4 --> -0.08251818171627795
0 --> 0.42786277532999367
2 --> 0.3790950728054445
1 --> -1.899150421685898
3 --> 0.03388921596190389
```

-> clusters semblent différents en fonction des features

Visualisation features

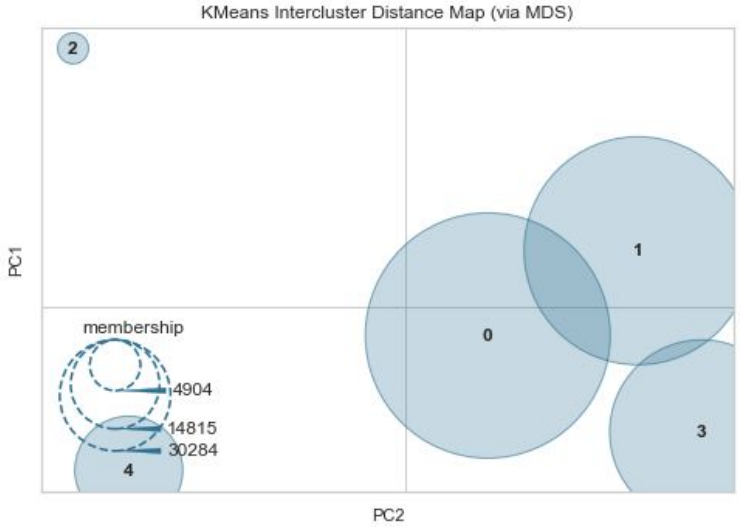


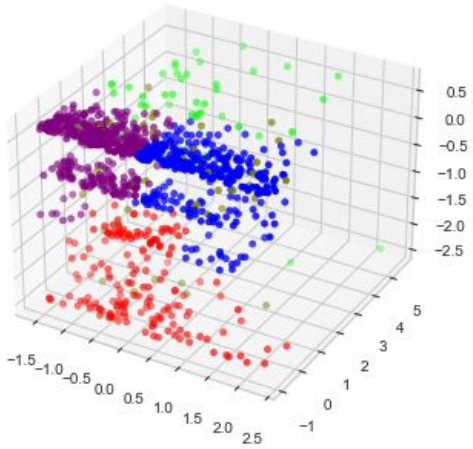


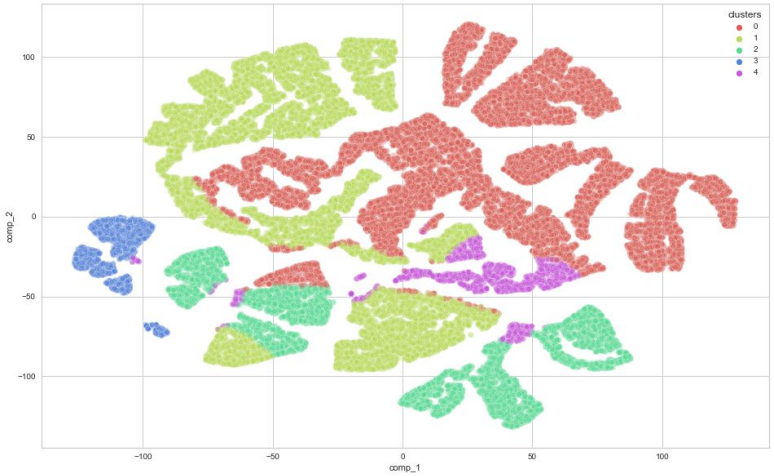
Visualisation

0 : clients inactifs, petites commandes, une seule commande, très satisfaits
1 : clients les plus actifs, petites commandes, une seule fois, très satisfaits
2 : clients peu actifs, petites commandes, une seule fois, insatisfaits
3 : clients actifs, grosses commandes, habitués et assez satisfaits
4 : clients actifs, commandes conséquentes,une seule fois, assez satisfaits

Clusters semblent assez éloignés



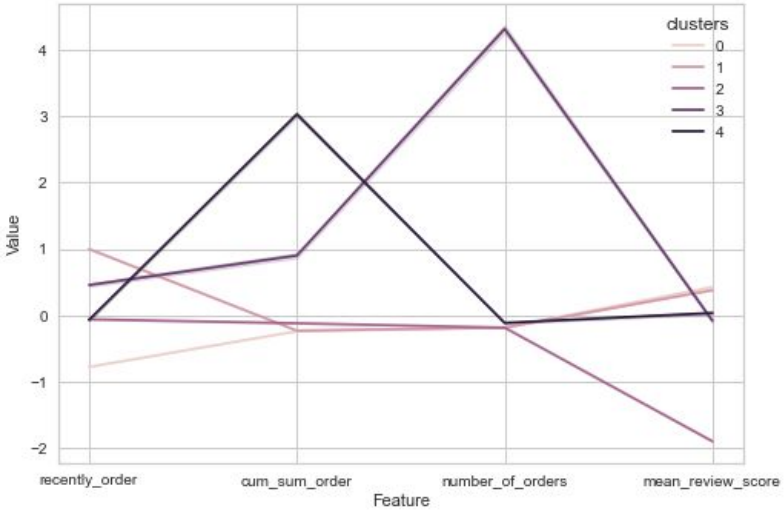




Le TSNE avec deux composantes ne semble pas bien différencier les clusters

Visualisation

Recommandations




```
]:  1  df_k_mean_norm.clusters.value_counts()

0    39431
1    30281
2    14817
4     4901
3     3730
```

Au vu de la clusterisation, afin de cibler les clients les plus attractifs :

- **Cluster 3** (clients endormis mais dépensier et ayant commandés plusieurs fois) [GOLD]
- **Cluster 0** (clients actifs et satisfaits) [SYLVER]
- **Cluster 4** (clients actifs, dépensiers) [BRONZE]

Le cluster 2 étant le **cluster à éviter** (clients endormis et insatisfaits) et 1 (clients anciens)

Maintenabilité 

Bases

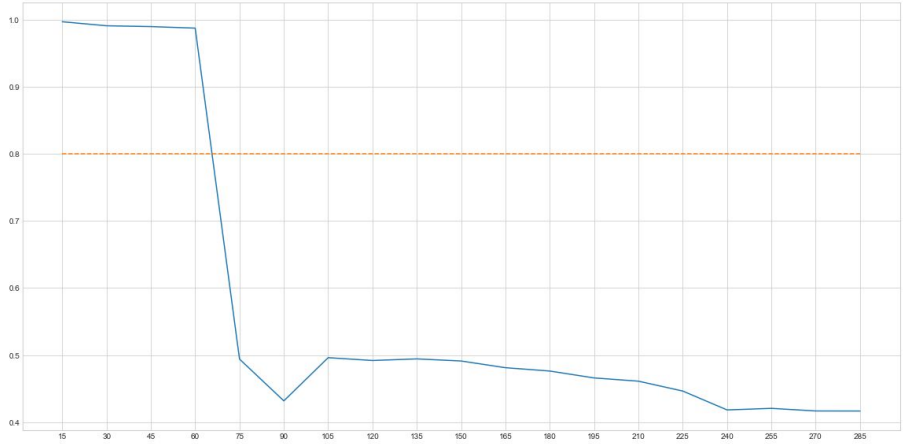
```
BEGINING = '2017-12-31'


class Maintenance:
    def __init__(self, df: pd.core.frame.DataFrame):
        self.df = df

    def fill_na(self):
        '''fill the missing values for the review'''
        value = self.df.mean_review_score.mean()
        self.df.mean_review_score.fillna(value=value, inplace=True)

    def date_convertor(self,
                       serie: pd.core.series.Series) -> pd.core.series.Series:
        '''take a pandas serie and return it in format datetime'''
        return pd.to_datetime(serie, format="%Y-%m-%d %H:%M:%S", )

    def normalisation(self, features_to_normalized, default=False):
        '''if set to true, normalised data'''
        if default:
            for feature in features_to_normalized:
                scaler = StandardScaler()
                self.df[feature] = scaler.fit_transform(self.df[[feature]])
        return self.df

    def clean_data(self,
                   data: pd.core.frame.DataFrame) -> pd.core.frame.DataFrame:
        '''take data with potencialy multiple orders for a customer and return
        data with only '''
        data = data[['customer_unique_id',
                     'recently_order',
                     'mean_order_price',
                     'cum_sum_order',
                     'number_of_orders',
                     'mean_review_score']].drop_duplicates()
        return data

    def creating_ari_data(self) -> dict:
        '''create a dictionary key=timedelta and value=dataframe'''
        self.df.order_delivered_customer_date = self.date_convertor(
            self.df.order_delivered_customer_date)
        MAX_DATE = max(self.df.order_delivered_customer_date)
```

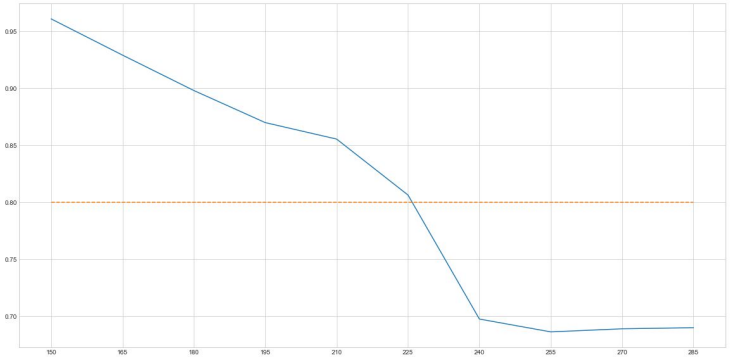
ARI (Adjusted Rand Score)

- définition d'une date avec suffisamment de données (**31-12-2017**)
- définition d'un **intervalle de temps** pour calculer ARI (**15 jours**)

-> le tout sous forme de classe pour faciliter l'implémentation par les équipes Olist

Résultats



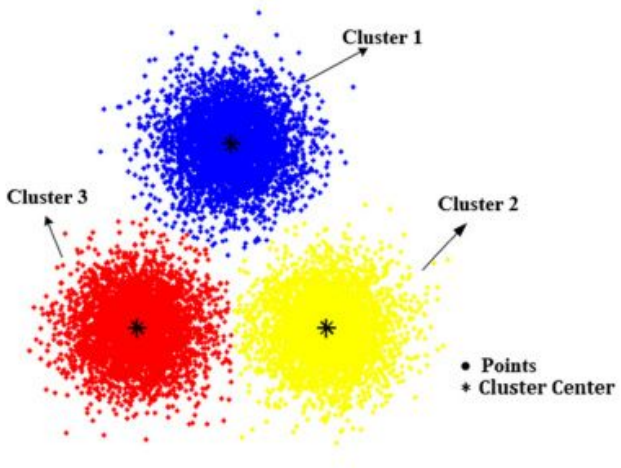


ARI supérieur à 0.8 : remodélisation tous les 60 jours

Recommandation : **remodélisation tous les 45 jours**

Test avec même système mais à une date ultérieure : OK

Conclusion



Clusterisation avec **KMeans(n_clusters=5)**

Clusterisation selon :

- **récense**
- **somme cumulative commandes**
- **moyenne des notes attribués**
- **fréquence**

-> RFM + satisfaction

Remodélisation **tous les 45 jours**

Pistes pour l'avenir :

- *inclure le panier moyen (aujourd'hui trop corrélé à la somme cumulative) -> renforce feature monetary*
- *NLP + clusterisation sur les commentaires -> renforce feature satisfaction*

Je vous remercie pour votre attention