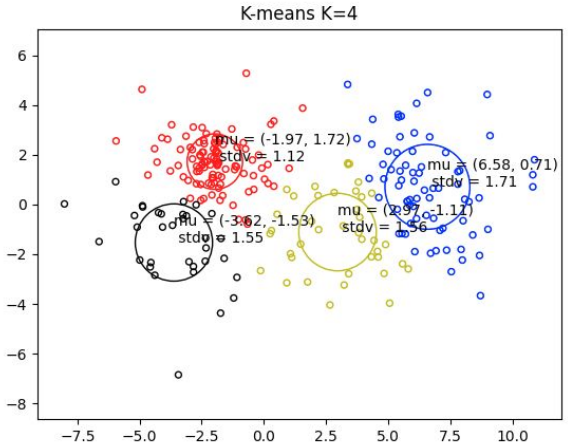
ESIEE PARIS

Rapport Projet 4ème année

Machine Learning

Netflix Clustering



Guillaume GAY et Kevin TOULCANON

Professeur référent : Victor Rabiet

Sommaire

I. Introduction

De nos jours les entreprises produisent une quantité astronomique de données et être capable de les analyser et les valoriser représente un enjeu crucial et un avantage compétitif indéniable. C'est pourquoi le machine learning et le métier de Data Scientist vont jouer un rôle important dans le monde que nous connaissons aujourd'hui. Le machine learning (ou apprentissage automatique en français) représente l'ensemble des d'algorithmes informatiques qui s'améliorent automatiquement par l'entraînement. Il est considéré comme un sous-ensemble de l'intelligence artificielle (*Wikipédia*). Le Data Scientist aura pour but d'utiliser des algorithmes de machine learning afin de créer par exemple des moteurs de recommandations, des prédictions pour son entreprise ou encore des intelligences artificielles.

Il existe deux grands types d'algorithme de machine learning : *l'apprentissage supervisée* et ***l'apprentissage non-supervisée***. La différence entre ces deux types d'algorithme est très simple. Dans le cas de l'apprentissage supervisée, le Data Scientist va guider l'algorithme en lui fournissant des exemples. L'algorithme va alors « apprendre » de chaque exemple en ajustant ses paramètres afin de limiter les erreurs que ce dernier pourrait générer. Concernant l'apprentissage non-supervisée, on ne va pas fournir d'exemples, c'est l'algorithme qui va apprendre de façon autonome en décortiquant les interactions entre les données.

Il est important de signaler que le type d'algorithme utilisé dépend de la situation dans laquelle on se trouve, c'est-à-dire, de la base de données et de ce que l'on cherche à déterminer.

Le projet qui nous a été attribué consiste à travailler sur une base de données contenant des données de la célèbre plateforme de streaming **Netflix**. Nous allons à partir de cette base de données nous intéresser à un problème connu auprès des Data Scientist : **le clustering**. Avec le clustering et les données de Netflix, il sera possible de **prédire les films** que les utilisateurs vont préférer. Au cours du projet nous avons pu étudier divers algorithmes que nous allons vous présenter dans ce rapport.

II. Clustering et KMeans

A. Définition

Le clustering est une méthode d'analyse statistique utilisée pour organiser des données brutes en ce que l'on nomme des « clusters ». Les clusters vont regroupés des données qui possèdent des similarités entre elles. Pour avoir un bon clustering, l'algorithme utilisé va minimiser l'inertie intraclasse de chaque cluster et maximiser l'inertie interclasse entre chaque cluster. En d'autres mots, plus les données d'un même cluster sont proches les unes des autres mais éloignées des données des autres clusters, meilleur sera le clustering effectué.

Le clustering sert donc principalement à segmenter ou classifier une base de données. Il est utilisé par exemple pour regrouper des articles par mots-clé sur un site. Pour se lancer dans le projet avec une première approche du clustering, nous avons implémenté un algorithme très classique de clustering :l'algorithme KMeans.

B. L'algorithme de clustering : KMeans

KMeans permet de regrouper en K clusters distincts (le nombre de cluster est défini à priori) les données en notre possession. Chaque donnée appartiendra à un cluster et ne pourra appartenir à deux clusters différents. Pour regrouper des données en clusters on va se servir de la notion de similarité entre ces dernières. Cette notion va se traduire en terme de distance au sens mathématiques : c'est une application qui doit satisfaire les trois propriétés de *séparation*, *symétrie* et *inégalité triangulaire*. Il existe plusieurs manières de calculer la distance entre deux éléments, nous avons implémenté trois des distances les plus classiques pour cet algorithme.

- Distance euclidienne : La distance euclidienne est la distance la plus communément utilisée. Entre deux points A et B, de coordonnées respectives a_i et b_i, $\forall i \in [1, n]$ dans un espace à n dimensions, la distance euclidienne entre ces deux points se calcule comme suit :

$$d(A, B) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

- La distance de Manhattan (taxi-distance) : est la distance entre deux points parcourus par un taxi lorsqu'il se déplace dans une ville où les rues sont agencées selon un réseau ou un quadrillage. Entre deux points A et B, de coordonnées respectives a_i et b_i, $\forall i \in [1, n]$, la distance de Manhattan est définie par :

$$d(A, B) = \sum_{i=1}^{n} |a_i - b_i|$$

- La similarité cosinus : donne une mesure de la similarité entre deux vecteurs en calculant le cosinus de l'angle entre eux. La similarité cosinus est une mesure de distance particulièrement efficace quand l'ordre de grandeur des vecteurs importe peu, car les vecteurs sont normalisés contrairement aux distances euclidienne et de Manhattan.

$$cos(A, B) = \frac{A \bullet B}{||A||.||B||} = \frac{\sum_{i=1}^{n} a_i \bullet b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}}$$

  La métrique que nous utiliserons est la "distance" cosinus. Cependant notons que ce n'est pas exactement une distance au sens mathématique car elle ne satisfait pas l'inégalité triangulaire.

$$d(A, B) = 1 - cos(A, B)$$

KMeans est un algorithme dit itératif ce qui signifie que c'est un algorithme qui va effectuer une boucle. Dans l'algorithme que nous allons utiliser la boucle aura pour but de minimiser la distance entre chaque individu et le centroid. Le centroid est un point qui sert de représentant entre chaque cluster. La fin de l'algorithme est atteinte lorsque que l'on a convergé vers la distance minimum entre chaque donnée et son centroid qui lui a été affecté.

```
INPUT: S, k where S = set of classified instances, k = integer
OUTPUT: k Clusters
Require: S ≠ ∅, k > 0
 1: procedure GENERATECLUSTERS
 2:     initialize k random Centroids
 3:     repeat
 4:         for all Instance i in S do
 5:             shortest ← 0
 6:             membership ← null
 7:             for all Centroid c do
 8:                 dist ← Distance(c)
 9:                 if dist < shortest then
10:                     shortest ← dist
11:                     membership ← c
12:                 end if
13:             end for
14:         end for
15:         RecalculateCentroids(c)
16:     until convergence
17: end procedure
```

source : [Comparing Clustering Techniques: A Concise Technical Overview]

Pour regrouper les points en clusters nous allons utiliser la notion de centre de classe, ou centroid : tous les points les plus proches des centres de chaque classe sont affectés à ces classes. L'algorithme KMeans prend en paramètre l'entier K, le nombre de clusters que l'on veut former. On commence par choisir aléatoirement K centroids avant d'entrer dans la partie itérative de l'algorithme. Tant qu'une condition n'est pas remplie (nombre d'itérations fixé, pas d'évolution d'une itération à une autre, etc), on calcule la distance entre chaque point et les centroids et on affecte chaque point à la classe de centroid le plus proche. Une fois que les clusters ont été formés, on recalcule les centroids, souvent en faisant une moyenne des points appartenant à la classe, mais d'autres méthodes peuvent être envisagées. Quand les centroids sont recalculés, on peut itérer de nouveau en calculant les distances avec les nouveaux centroids.

KMeans a l'avantage d'être simple d'utilisation et facilement transposable sur des datasets de nature et de taille très différentes. Il a également un temps d'exécution rapide et un faible coût de calculs, avec une complexité linéaire en O(K*d*n) où K est le nombre de clusters et d la dimensions des données. Cependant l'algorithme a aussi des défauts comme le fait qu'il ne fournisse pas forcément l'ensemble optimal de clusters, il faut d'ailleurs préciser le nombre de clusters au début de l'algorithme. L'initialisation aléatoire des centroids conduit à des résultats différents d'une exécution à une autre et l'algorithme a besoin de clusters de tailles équivalentes pour être efficace.

KMeans est donc une des algorithmes de clustering les plus simples à mettre en place et est facile et rapide à entraîner. Mais si ses performances permettent une bonne première approche d'un problème, les performances seront souvent moins bonnes que d'autres algorithmes plus sophistiqués car les résultats dépendent beaucoup de l'initialisation et des données en elles-même.

C. Implémentation de l'algorithme

Une fois le principe de l'algorithme assimilé nous l'avons implémenté en utilisant le langage informatique python. Pour ce faire nous nous sommes aidés de différents packages :

- *pandas* : permet de manipuler facilement des données à analyser.
- *numpy* : permet d'effectuer des calculs numériques avec une certaine facilité sur des tableaux de nombres pouvant atteindre de grandes dimensions.
- *Scipy* : implémente beaucoup d'outils utiles pour travailler, entre autres, sur des tableaux ou images.
- *Scikit-learn* : bibliothèque destinée à l'apprentissage automatique qui nous fournira des implémentations déjà réalisées des algorithmes que nous allons écrire.
- *Matplotlib.pyplot* : permet d'afficher des graphes.

Le code de tout ce qui suit dans cette partie est situé dans le notebook *KMeans_KMedoids.ipynb*.

La première étape a été de créer une classe Points sur laquelle nous effectueront les tests sur des points 2D. Ensuite nous avons créé la classe KMeans qui implémente l'algorithme. Cette classe est bâtie sur le même modèle que les algorithmes de machine learning de scikit learn : instancier la classe permet d'initialiser le modèle, ensuite la fonction *fit()* permet d'entraîner le modèle sur des données d'entraînement et de retourner les clusters.

Voici le code de la fonction *fit()* qui implémente l'algorithme tel que vu précédemment. L'initialisation du modèle demande le nombre de clusters, le nombre d'itérations et la métrique à utiliser pour les stocker dans des attributs.

```
21      def fit(self, X, drawing=False):
22          """
23          On effectue un nombre d'itérations fixe défini par self.n_iter
24          """
25          if not isinstance(X, np.ndarray):
26              X = np.array(X)
27
28          # On choisi K points aléatoirement pour être les centroides
29          self.centroids = X[np.random.choice(len(X), self.K, replace=False)]
30          self.initial_centroids = self.centroids
31          self.prev_label,  self.labels = None, np.zeros(len(X))
32          # On effectue n_iter itérations
33          for i in range(1,self.n_iter+1):
34              self.prev_label = self.labels
35              # On calcule la distance de chaque point aux centroids
36              # et on affecte un point à la classe ayant le centroide le plus proche
37              self.labels = self.predict_labels(X, self.centroids, self.dist)
38              # On change le centroide de chaque classe en prenant la moyenne des points de cette classe
39              self.update_centroid(X)
40
41              if i % 10 == 0: #On plot toutes les 10 itérations
42                  self.draw_graph(X, i=i, show=drawing)
43          return self
44
```

Pour le calcul des distances/labels et des nouveaux centroids, on utilise dans la fonction *fit()* les fonctions *predict_labels()* et *distance()* et la fonction *update_centroid()*.

```
45      def predict_labels(self, X, centroids, dist='euc'):
46          if dist in ['euc', 'l1']:
47              return self.distance(X, centroids, dist).argmin(axis=1)
48          elif dist=='cos':
49              return abs(self.distance(X, centroids, dist)).argmin(axis=1)
50
51      def distance(self, X, centroids, dist='euc'):
52          map = {'euc':'euclidean', 'cos':'cosine', 'l1':'cityblock'}
53          return sp.spatial.distance.cdist(X, centroids, metric=map[dist])
54
55      def update_centroid(self, X):
56          self.centroids = np.array([np.mean(X[self.labels == k], axis=0)  for k in range(self.K)])
57          # On pourrait utiliser np.median au lieu de np.mean pour L1
58
59      def draw_graph(self, X, i=0, show=True):
60          plt.scatter(X[:, 0], X[:, 1], c=self.labels, s=50, cmap='viridis')
61          plt.scatter(self.centroids[:, 0], self.centroids[:, 1], c='black', s=200, alpha=0.5)
62          plt.xlabel('x'); plt.ylabel('y'); plt.title(f'Iteration n°{i}')
63          self.graph = plt.gcf()
64          if show: plt.show()
65          else: plt.close(self.graph)
```

Les distances sont calculées dans *distance()*. Les métriques utilisées sont extraites de la bibliothèque distance de *Scipy*. Nous avons choisi de n'utiliser que les trois métriques vues précédemment : 'euc', 'cos' et 'l1'. Cependant l'implémentation permet facilement de rajouter d'autres métriques, extraites de *Scipy* ou recodées en partant de rien.

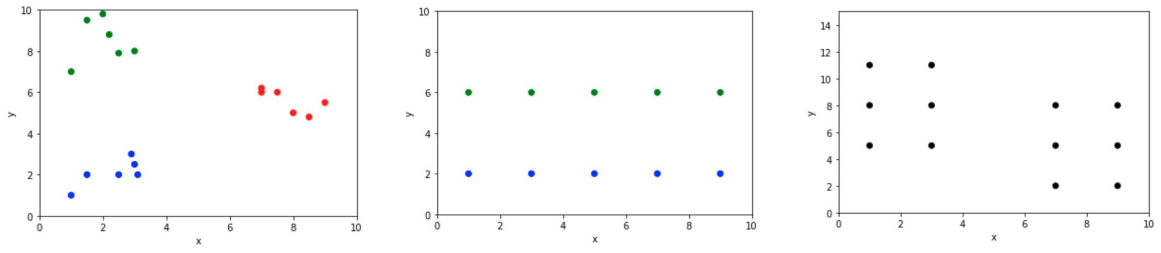
La fonction *predict_labels()* utilise distance pour récupérer le point le plus proche du centroid. Pour les distances euclidienne et L1, on récupère la distance minimale. Pour la "distance" cosinus, on choisit l'élément le plus proche de 0 (la "distance" cosinus donne des nombres entre -1 et 1).

Point clé de l'algorithme, les centroids sont mis à jour dans la fonction *update_centroid()*. Une fois que tous les points ont été affectés à des classes, les nouveaux centroids sont calculés en prenant la moyenne de ces points, le point "moyen" ou centre de classe. Utiliser la moyenne est la pratique la plus courante et est acceptable pour nos trois métriques, mais nous pourrions envisager d'autres manières de recalculer les centroids, comme utiliser la médiane pour la distance L1 par exemple (ce qui donnerait l'algorithme appelé KMedian).

Une fonction *draw_graph()* est appelé dans *fit()* et permet de créer un graph pendant l'exécution de l'algorithme pour suivre l'évolution des clusters. Un graph est généré toutes les 10 itérations et le graph final est accessible via l'attribut *self.graph*.

D. Test sur différents cas de figure

Nous allons maintenant tester les performances de notre algorithme KMeans sur différentes configurations de nuages de points. Le graphe de gauche présente un nuage de point classique et les deux suivants illustrent des situations où les distances cosinus et L1 sont censées être plus adaptées. Nous appliquerons ensuite l'algorithme sur des nuages gaussiens, plus représentatifs de la suite du projet.

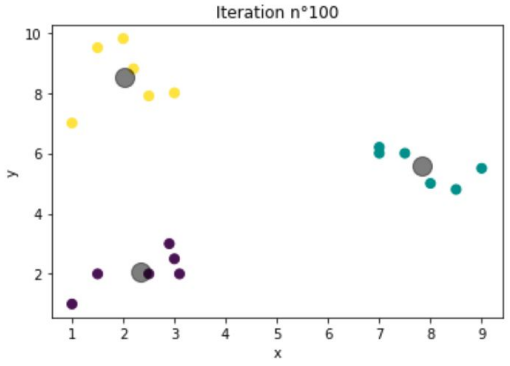


Sur le premier nuage de points, où les clusters sont bien démarqués et sphériques (ce qui est le cas idéal pour KMeans et la métrique euclidienne) notre algorithme sépare bien les différents clusters.

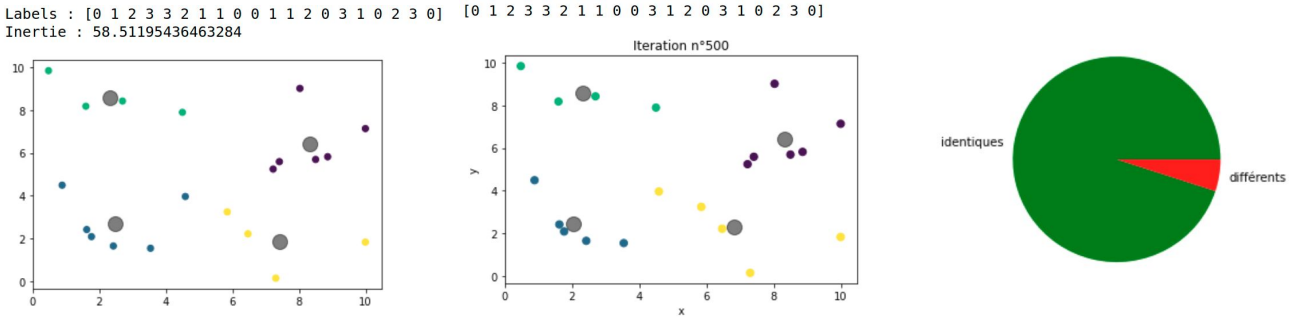
```
kmeans = KMeans(n_clusters=3, n_iter=100, dist='euc')
kmeans.fit(df_X, drawing=False)
kmeans.labels
```

```
array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2])
```

```
kmeans.graph
```



Les résultats ont d'ailleurs été équivalents à la version de KMeans de *sklearn* lorsque nous les avons comparés sur une autre nuage de 20 points (à gauche, le clustering donné par s*klearn*, au milieu celui donnée par notre algorithme, à droite la comparaison entre les deux classifications).

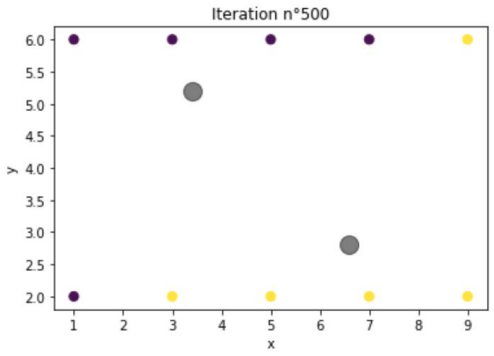


Les cas suivants montrent que la classification peut être différente (et de qualité variable également) en fonction de la disposition des points ou de la métrique utilisée.

```
1 kmeans = KMeans(n_clusters=2, n_iter=500, dist='cos')
2 kmeans.fit(df_X, drawing=False)
3 kmeans.labels
```

```
array([0, 1, 1, 1, 1, 0, 0, 0, 0, 1])
```

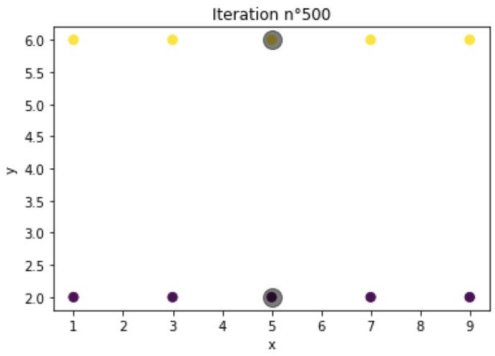
```
1 kmeans.graph
```




```
1 kmeans = KMeans(n_clusters=2, n_iter=500, dist='l1')
2 kmeans.fit(df_X, drawing=False)
3 kmeans.labels
```

```
array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```

```
1 kmeans.graph
```

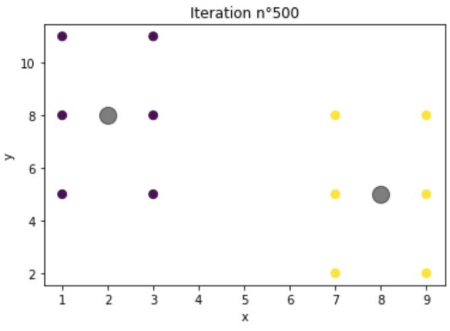


Ci-dessus, la distance cosinus (à gauche) classifie mal deux points et la distance L1 donne bien les deux clusters attendus.

```
1 kmeans = KMeans(n_clusters=2, n_iter=500, dist='euc')
2 kmeans.fit(df_X, drawing=False)
3 kmeans.labels
```

```
array([1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0])
```

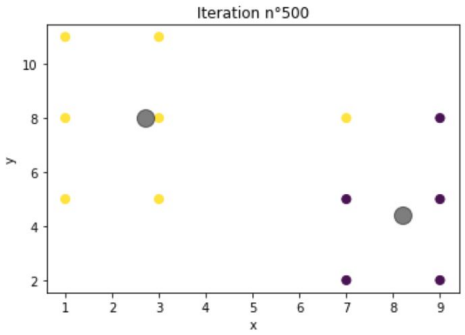
```
1 kmeans.graph
```




```
1 kmeans = KMeans(n_clusters=2, n_iter=500, dist='l1')
2 kmeans.fit(df_X, drawing=False)
3 kmeans.labels
```

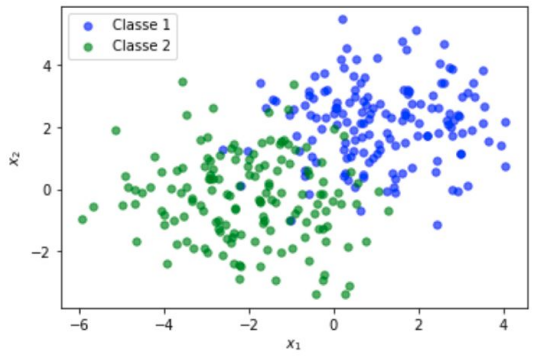
```
array([0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1])
```

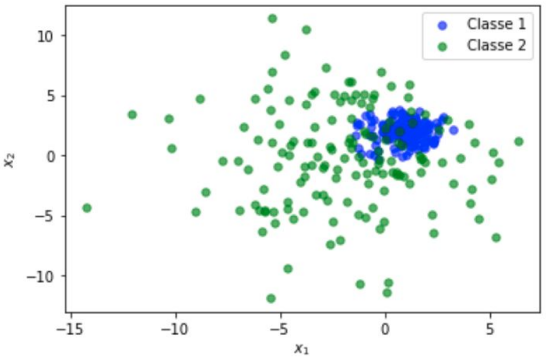
```
1 kmeans.graph
```



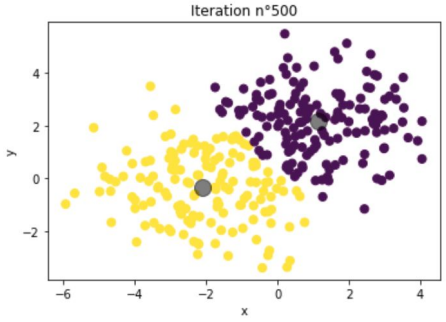
Dans cet exemple, les clusters sont bien identifiés par les distances euclidienne et cosinus, mais un point est mal classé par la distance L1. Ce qui ne correspond pas à nos attentes car ce nuage de points construit en quadrillage devrait être propice à la distance de Manhattan.

Nous allons maintenant appliquer l'algorithme KMeans à deux clusters gaussiens. Tout d'abord dans le cas où les variances des deux nuages gaussiens sont équivalentes, puis dans le cas où la variance d'un des deux nuages est bien plus grande que l'autre, et où les deux nuages se recouvrent.

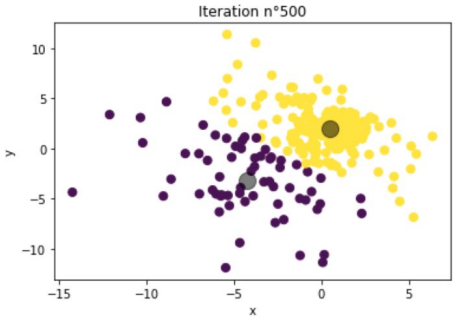





```
kmeans = KMeans(n_clusters=2, n_iter=500, dist='euc')
kmeans.fit(data, drawing=False)
kmeans.graph
```




```
kmeans = KMeans(n_clusters=2, n_iter=500, dist='euc')
kmeans.fit(data, drawing=False)
kmeans.graph
```



On remarque que la classification est globalement bien faite, surtout la séparation spatiale entre les deux clusters est bien marquée. Cependant les points du cluster 1 qui recouvrent le cluster 2 sont attribués au cluster 2. En effet, c'est dû au fait que ces points sont plus proche spatialement du mauvais centre de classe. Nous voyons ici une des limites de l'algorithme KMeans. D'autres algorithmes comme l'algorithme EM que nou verrons en dernier permettent d'éviter ce problème.

III. Variante de KMeans : KMedoids

Les centroids calculés par l'algorithme KMeans sont des moyennes des points du cluster associé. Ils ne sont donc pas des points appartenant aux données. Dans certaines applications, on peut vouloir définir les clusters en fonction d'un point des données qui expliquerait le mieux tous les autres. C'est précisément ce que permet de faire l'algorithme KMedoids.

KMedoids reprend le fonctionnement de KMeans, la fonction *fit()* notamment est la même pour les deux algorithmes, mais diffère dans le calcul des centroids (que l'on appelle medoids ici). Il n'y a donc que la méthode *update_centroid()* qui est modifiée.

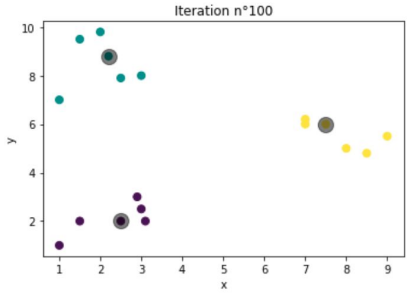
```
46      def distance(self, X, centroids, dist='euc'):
47          map = {'euc':'euclidean', 'cos':'cosine', 'l1':'cityblock'}
48          return sp.spatial.distance.cdist(X, centroids, metric=map[dist])
49
50      def update_centroid(self, X, dist='euc'):
51          self.centroids = np.array([np.mean(X[self.labels == k], axis=0)  for k in range(self.K)])
52
53          distances = self.distance(X, self.centroids, dist=dist)
54          self.centroids = X[distances.argmin(axis=0)]
55
56      def draw_graph(self, X, i=0, show=True):
57          plt.scatter(X[:, 0], X[:, 1], c=self.labels, s=50, cmap='viridis')
58          plt.scatter(self.centroids[:, 0], self.centroids[:, 1], c='black', s=200, alpha=0.5)
59          plt.xlabel('x'); plt.ylabel('y'); plt.title(f'Iteration n°{i}')
60          self.graph = plt.gcf()
61          if show: plt.show()
62          else: plt.close(self.graph)
63
```

Les medoids sont donc fixés sur les points des données qui définissent le mieux les clusters au sens des distances utilisées.

```
1  kmed = KMedoids(n_clusters=3, n_iter=100, dist='euc')
2  kmed.fit(df_X, drawing=False)
3  kmed.labels
```

```
array([0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1])
```

```
1  kmed.graph
```



KMedoids a des avantages par rapport à l'algorithme KMeans : il est plus flexible car il peut être utilisé avec n'importe quelle métrique alors que la convergence de KMeans n'est pas assurée avec certaines métriques. KMedoids est également moins sensible aux valeurs aberrantes. Si un point est très éloigné des autres (une erreur de mesure par exemple), la moyenne dans KMeans le prendra en compte et le centroid en sera affecté, alors que le medoid sera le plus centré du cluster. Cependant le défaut de KMedoids par rapport à KMeans est son coût en calculs. Là où KMeans a une complexité linéaire, celle de KMedoid est quadratique.

IV. Classification Naïve Bayésienne

A. Première approche

Puisque le temps nous le permettait, nous nous sommes attardés sur un autre algorithme de machine learning : la classification Naïve Bayésienne. Contrairement à K-Means et K-Medoid, cet algorithme est un algorithme d'apprentissage supervisée et doit son nom au théorème qu'il exploite afin de fonctionner : *le théorème de Bayes*. L'idée est de calculer la probabilité qu'un document appartienne à un cluster, et d'affecter le document au cluster ayant la plus haute probabilité. Pour calculer cette probabilité, nous allons nous baser sur les probabilités conditionnelles.

Le théorème de Bayes est un théorème se basant sur les probabilités conditionnelles, c'est-à-dire, la probabilité qu'un évènement se produise sachant qu'un autre événement s'est déjà produit que l'on note P(A|B). Le théorème est le suivant :

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A) \times P(A)}{P(B)}$$

Prenons un petit exemple pour illustrer ce théorème. Supposons qu'on ait une classe de collégiens. Soit A et B les deux évènements suivants :

- l'événement A : l'élève est un garçon.
- l'événement B : l'élève pratique l'espagnol.

	Garçon	Fille	Total
Espagol	12	9	21
Autre langue	5	3	8
Total	17	12	29

Quelle est la probabilité qu'on tire au hasard un élève parlant Espagnol sachant que c'est un garçon ?

Cette question se traduit ainsi avec les notations :

$P(B|A) = \frac{P(B \cap A)}{P(A)}$

- P(A) est ce que l'on appelle la *prior probability*. Ici il s'agit de calculer la probabilité de prendre au hasard un garçon dans le jeu de donnés que l'on a.

  $P(A) = \frac{Cardinal(A)}{Cardinal(\Omega)} = \frac{17}{29} = 0.59$

  * cardinal(Ω) représente l'ensemble des lycéens

- P(B∩A) est la probabilité de choisir au hasard dans la population totale un garçon faisant espagnol.

  $P(B \cap A) = \frac{Nombre\ de\ garçons\ faisant\ espagnol}{Cardinal(\Omega)} = \frac{12}{30} = 0.4$

Donc on a :

$$P(B|A) = \frac{0.4}{0.59} = 0.68$$

B. Implémentation de l'algorithme

Lors de l'exemple précédent, le théorème de Bayes a été appliqué qu'à une seule variable. Pour ce qui va suivre, on va calculer le résultat en utilisant plusieurs variables et cela rend les choses beaucoup plus complexes au niveau des calculs. Pour contourner ce problème, nous supposons que toutes les variables que nous utiliserons sont indépendantes entre elles ce qui est totalement faux et c'est pour cela qu'on attribue à cette classification l'adjectif "naïve" qui traduit l'absurdité de cette hypothèse. Néanmoins, malgré cette fausse hypothèse l'algorithme donne de bons résultats.

Pour exploiter la classification Naïve Bayésienne, notre responsable de projet nous a fourni des données de critiques de films. Ces critiques sont classifiées en deux groupes :

	content	label
0	films adapted from comic books have had plenty...	pos
1	every now and then a movie comes along from a ...	pos
2	you've got mail works alot better than it dese...	pos
3	" jaws " is a rare film that grabs your atten...	pos
4	moviemaking is a lot like being the general ma...	pos
...	...	...
1795	2 days in the valley is more or less a pulp fi...	neg
1796	what would inspire someone who cannot write or...	neg
1797	synopsis : a novelist struggling with his late...	neg
1798	okay , okay . \nmaybe i wasn't in the mood to ...	neg
1799	in life , eddie murphy and martin lawrence pla...	neg

apperçu des données

- Pos : pour les critiques dites positives
- Neg : pour les critiques dites négatives

Les données étaient réparties en deux dossiers selon la langue utilisée pour la critique. Puisque l'algorithme utilisé est du type apprentissage supervisée, dans ces deux dossiers on y trouvait des données pour l'apprentissage (train) et des données pour la validation de notre algorithme (test).

Avant de commencer à travailler sur l'implémentation de notre propre algorithme, nous avons utilisé la bibliothèque scikit learn pour avoir une idée du résultat que nous devions obtenir.

```
vectorizer = CountVectorizer()

counts = vectorizer.fit_transform(a_train['content'].values)
```

```
classifier = MultinomialNB()
targets = a_train['label'].values
classifier.fit(counts, targets)
```

```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

```
test = vectorizer.transform(a_test['content'])
predictions = classifier.predict(test)
```

```
print('Le pourcentage de bonnes prédictions avec la libraire scikitlearn est de : ', pourcentage(predictions,a_test), '%')
```

```
Le pourcentage de bonnes prédictions avec la libraire scikitlearn est de :  83.0 %
```

Première approche avec scikitlearn

Comme nous pouvons l'observer, il y a 83% de bonnes prédictions. Notre objectif à nous a alors été de nous en approcher le plus possible.

La première étape a été de créer une fonction pour compter les mots, la fonction est la suivante :

```
def freqMots(df):
    l = []
    dic = {}
    for e in df.itertuples():
        l.append(e[1])
    L = traitement_texte(l)
    for phr in L: #création du dictionnaire
        mots = nltk.word_tokenize(phr)
        for mot in mots:
            if mot not in dic.keys():
                dic[mot] = 1
            else:
                dic[mot] += 1
    return dic
```

fonction freqMots()

La fonction traitement_texte() à la ligne 6 qui prend en entrée une chaîne de caractères est une fonction que nous avons mis en place afin de supprimer les caractères inutiles comme la ponctuation et les espaces blancs mais aussi afin de mettre tous les caractères en minuscules pour ne pas avoir de différence entre par exemple "Agréable" et "agréable".

La boucle for va prendre chaque phrase que contient la liste L, ensuite dans la variable *mots* on va diviser la phrase en tous les mots qu'elle contient grâce à la fonction word_tokenize() de la bibliothèque nltk. Ensuite de cette liste de mots on va pouvoir compter les mots, si le mot n'est pas dans le dictionnaire on va l'ajouter et lui attribuer le compte 1 mais si il appartient déjà au dictionnaire on va rajouter au compte +1. On obtient alors le résultat suivant pour tous les mots des données :

{'films': 1357, 'adapted': 40, 'from': 4461, 'comic': 352, 'books': 70, 'have': 4404, 'had': 1395,
'plenty': 123, 'of': 30547, 'success': 192, 'whether': 197, 'they': 4338, 're': 1048, 'about': 3146,
'superheroes': 12, 'batman': 189, 'superman': 24, 'spawn': 82, 'or': 2824, 'geared': 13, 'toward': 9
0, 'kids': 290, 'casper': 17, 'the': 68583, 'arthouse': 2, 'crowd': 74, 'ghost': 70, 'world': 933,
'but': 7779, 'there': 3378, 's': 16691, 'never': 1208, 'really': 1402, 'been': 1851, 'a': 34297, 'bo
ok': 303, 'like': 3326, 'hell': 219, 'before': 891, 'for': 8920, 'starters': 6, 'it': 14531, 'was':
4426, 'created': 154, 'by': 5649, 'alan': 90, 'moore': 80, 'and': 31848, 'eddie': 143, 'campbell': 9
0, 'who': 5150, 'brought': 140, 'medium': 31, 'to': 28663, 'whole': 447, 'new': 1154, 'level': 241,
'in': 19622, 'mid': 53, '80s': 50, 'with': 9664, '12': 55, 'part': 644, 'series': 493, 'called': 35
3, 'watchmen': 1, 'say': 712, 'thoroughly': 66, 'researched': 1, 'subject': 146, 'jack': 265, 'rippe
r': 6, 'would': 1907, 'be': 5551, 'saying': 150, 'michael': 369, 'jackson': 115, 'is': 22531, 'start
ing': 53, 'look': 749, 'little': 1359, 'odd': 98, 'graphic': 81, 'novel': 238, 'if': 2524, 'you': 48
35, 'will': 1996, 'over': 1284, '500': 15, 'pages': 16, 'long': 753, 'includes': 96, 'nearly': 282,
'30': 67, 'more': 2997, 'that': 14274, 'consist': 8, 'nothing': 717, 'footnotes': 2, 'other': 1743,
'words': 194, 'don': 1162, 't': 5708, 'dismiss': 12, 'this': 8581, 'film': 8569, 'because': 1511, 'i
ts': 2042, 'source': 56, 'can': 2704, 'get': 1749, 'past': 306, 'thing': 724, 'might': 582, 'find':
695, 'another': 1001, 'stumbling': 14, 'block': 42, 'directors': 106, 'albert': 35, 'allen': 128, 'h
ughes': 43, 'getting': 368, 'brothers': 178, 'direct': 73, 'seems': 931, 'almost': 735, 'as': 10212,
'ludicrous': 37, 'casting': 107, 'carrot': 11, 'top': 346, 'well': 1702, 'anything': 557, 'riddle':

Compte des mots des données anglaises

Il a fallu ensuite faire le compte des mots pour la classe pos et la classe neg afin de calculer les probabilités conditionnelles.

Le calcul des prior probability s'est fait facilement puisque dans les données d'apprentissage nous avions la moitié qui était de la classe pos et la moitié de la classe neg. Donc P(pos) = P(neg) =0,5. En ce qui concerne les probabilités conditionnelles le calcul était un peu plus complexe. Nous avons utilisé la formule suivante :

$$P(f|c) = \frac{count(f,c)+1}{\Sigma f' count(f',C)+1}$$

Ici count est l'équivalent de cardinal, f représente une caractéristique de la classe c (soit un mot dans notre cas). En dénominateur on retrouve le nombre de mots total selon la classe (pos ou neg). On ajoute au numérateur et au dénominateur +1 afin de ne pas avoir un zéro si le mot n'appartient à la classe pour lequel on calcul sa probabilité. Pour calculer les probabilités conditionnelles de chaque mot nous avons créé une fonction conditionnal_prob() qui prend en entré un dictionnaire et qui retourne aussi un dictionnaire avec les mots et leurs probabilités respectives.

```python
def conditionnal_prob(dic):
    """
        fonction qui va calculer les probabilités conditionnelles de chaque mot selon la classe dont elle appartient
    """
    dpos_prob = {} #dictionnaire qui contiendra les probabilités du vocabulaire pour la classe pos
    dneg_prob = {} #dictionnaire qui contiendra les probabilités du vocabulaire pour la classe neg
    a = 0
    for cle, valeur in dic.items():
        if cle in dic_pos:
            a = (dic_pos[cle]+1.0)/(nbr_mots_pos+1) #utilisation de la formule
            dpos_prob[cle]=a
        else: #si le mot n'appartient pas à la classe
            a = 1.0/(nbr_mots_pos+1)
            dpos_prob[cle]=a
        if cle in dic_neg:
            a = (dic_neg[cle]+1.0)/(nbr_mots_neg+1)
            dneg_prob[cle]=a
        else:
            a = 1.0/(nbr_mots_neg+1)
            dneg_prob[cle]=a
    return dpos_prob, dneg_prob
```

Une fois les probabilités conditionnelles calculées, il nous restait plus qu'à appliquer la classification bayésienne aux données de test. Pour ce faire, on a créé une fonction classe qui prend en entré les données test en format dataframe et qui retourne une liste contenant la classe de chaque critique. La fonction va calculer pour chaque document d la probabilité d'appartenir à la classe pos et à la classe neg. Cette probabilité s'obtient en multipliant la prior probability de la classe (ici 0.5 pour les deux classes) et toutes les probabilités de chaque mot d'appartenir à telle ou telle classe. On a alors créé une variable qui va avoir pour rôle de prendre la probabilité de chaque mot est de les multiplier entre eux. Pour éviter d'avoir des résultats trop faibles nous avons pris le log. Une fois calculée, cette variable est ajouté à une liste puis elle va ensuite être réinitialisée afin de calculer la probabilité pour le document suivant. C'est donc cette liste qui va être retourné.

```
def classe(df):
    """
        fonction qui va déterminer selon la base d'apprentissage si un document de la base de test
        est de la classe pos ou neg
    """
    L=[]
    Lp = []
    Ln = []
    l = []
    for e in df.itertuples():
        L.append(e[1])
    for i in range(len(df)): #traitement de textes
        L[i] = L[i].lower() #lettres en micuscules
        L[i] = re.sub(r'\W',' ',L[i]) #suppression de la ponctuation
        L[i] = re.sub(r'\s+',' ',L[i]) #suppresion de espaces blancs
        #compte des mots
        mots = L[i].split()
        compte = {}.fromkeys(set(mots),0)
        for valeur in mots:
            compte[valeur] += 1
        #Propriétés d'appartenir à la classe pos ou neg
        pp = np.log(0.5) #0.5 est une prior probability pour la classe pos
        pn = np.log(0.5) #0.5 est une prior probability pour la classe neg
        for cle in compte.keys():
            if cle in dic:
                pp+=np.log(dpos_prob[cle]**(compte[cle]))
                #en puissance compte[cle] correspond à la fréquence du mot dans le document
            if cle in dic:
                pn+=np.log(dneg_prob[cle]**(compte[cle]))
        Lp.append(pp)
        Ln.append(pn)
    for i in range(len(Lp)):
        if(Lp[i]>Ln[i]):
            l.append('pos')
        else:
            l.append('neg')
    return l
```

En appliquant cette fonction aux données test on obtient alors 81.5% de bonnes prédictions contre 83% avec scikitlearn ce qui est un résultat satisfaisant. Ce qu'on peut retenir c'est que la classification est meilleure pour la classe neg. En effet, pour les documents qui sont de la classe neg dans les données test, en appliquant l'algorithme on obtient 90% de bons résultats contre 73% pour les données de la classe pos. Cela pourrait s'expliquer par le fait qu'il est plus aisé de caractériser un document rempli de termes négatifs, souvent marqués et à connotations fortes, que le contraire. De plus, lorsqu'on applique l'algorithme sur les données françaises on obtient de moins bons résultats : 71.5% de prédictions justes.

V. Algorithme E-M *(Espérance-maximisation)*

A. Approche théorique

On peut définir l'algorithme EM comme une méthode d'estimation paramétrique s'inscrivant dans le cadre général du maximum de vraisemblance. C'est un algorithme itératif qui permet donc de trouver les paramètres du maximum de vraisemblance. À chaque itération, l'algorithme va effectuer deux étapes : étape E (« expectation ») et étape M (« maximisation). L'étape E va estimer les données inconnues en fonction des données observées, c'est-à-dire, estimer (une première fois ou de nouveau) la valeur des paramètres qui ont été trouvée à l'itération précédente. Quant à l'étape M, elle correspond comme son nom l'indique à la maximisation de la vraisemblance rendue possible grâce à l'estimation des données inconnues à l'étape précédente.

Pour appliquer l'algorithme on va utiliser le modèle de mélange Gaussien. Le modèle de mélange gaussien est un modèle qui se compose de k distributions de probabilités représentant k clusters. Il combine plusieurs distributions normales. Ces distributions dépendent de leurs espérances μ et de leurs écart-type σ. La densité de probabilité de la loi normale est donnée par la formule suivante :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

On utilise une distribution normale (ou aussi appelé distribution gaussienne) car la loi normale est la plus adaptée pour modéliser un phénomène naturel issu de plusieurs évènements aléatoires. En effet, d'après le théorème central limite, une suite de variables aléatoires indépendantes converge vers une loi normale.

Pour pouvoir exploiter l'algorithme pour un modèle de mélange gaussien on considère que les données sont composées de n individus différents appartenant chacun à une des classes $C_1,...,C_g$ suivant une loi normale de moyenne : μ_k et de matrice de covariance $\sum_k$ (k = 1,...,g). On note le paramètre de chaque loi $\Theta_k(\mu_k, \sum_k)$ et les probabilités des différentes classes $\pi_1,...,\pi_g$. C'est le paramètre Θ_k qu'on va chercher à déterminer en utilisant l'algorithme E-M.

Une fois toutes les conditions réunies on peut suivre les étapes décrites ci-dessous :

Algorithm 3: E-M Algorithm

Data: Dataset $\mathcal{D} = \{X_1, \ldots, X_n\} \subset \mathbb{R}^p$. Parametric model $(f_\theta)_\theta$.

Initialization : Choose an initial distribution π^0 in the probability simplex and pick a parameter θ^0 randomly in its state space.

Denote $\Theta^0 = (\pi^0, \theta^0)$.

for $t = 1 \ldots + \infty$ **do**

E Step Use formula (5) to compute the expected responsabilities on each observation :

$$\forall i \in \{1, \ldots, n\} \quad \forall k \in \{1, \ldots, K\} \qquad \gamma_{i,k}^t = \frac{\pi_k^{t-1} f_{\theta_k^{t-1}}(X_i)}{\sum_{k=1}^{K} \pi_k^{t-1} f_{\theta_k^{t-1}}(X_i)}.$$

M Step Update the parameters $\Theta^t = (\pi^t, \theta^t)$ with

$$\pi^t := \arg \max_{\pi : \sum \pi_k = 1} \sum_{k=1}^{K} \log[\pi_k] \left(\sum_{i=1}^{n} \gamma_{i,k}^t \right) \tag{6}$$

and

$$\forall k \in \{1 \ldots K\} \qquad \theta_k^t := \arg \max_{\theta} \sum_{i=1}^{n} \gamma_{i,k}^t \log f_{\theta_k}(X_i). \tag{7}$$

end

Output : Limiting values in Θ^t.

Source : Unsupervised clustering with EM, université de Toulouse

À l'étape d'initialisation, lorsque l'on initialise Θ_0 cela revient aussi à initialiser les probabilités de chaque classe π_k. À partir de l'échantillon qu'on obtient on va calculer le log de vraisemblance. Ensuite on peut entamer la partie itératif de l'algorithme avec l'étape E et l'étape M. On calcule donc à l'étape E les probabilités conditionnelles π_k pour la valeur courante de Θ. Ensuite à l'étape M on va actualiser l'estimation de Θ puis recalculer le log de vraisemblance. L'algorithme s'arrêtera lorsque que l'algorithme aura convergé selon une condition que l'on s'est fixé. Ce dernier retournera alors Θ_k, les paramètres que l'on cherchait à estimer.

B. Code de l'algorithme

L'algorithme EM que nous avons codé s'apparente à une première version simplifiée de l'algorithme EM qui serait nécessaire pour traiter les reviews des films Netflix. En effet, étant donné la grande dimension des données (d=1600), il nous est impossible de calculer directement la densité de la loi gaussienne car le calcul demandé est trop grand. La version actuelle de l'algorithme permet toutefois de travailler sur des données de dimensions plus faibles, comme le dataset sur les jouets que nous avons utilisé pour travailler sur cette partie du projet.

```python
def estep(X: np.ndarray, mixture: GaussianMixture) -> Tuple[np.ndarray, float]:
    """E-step: Softly assigns each datapoint to a gaussian component

    Args:
        X: (n, d) array holding the data
        mixture: the current gaussian mixture

    Returns:
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: log-likelihood of the assignment
    """
    n, d = X.shape
    K = mixture.mu.shape[0]
    densite = np.zeros([n, K])
    posterior = np.zeros([n, K])
    for i in range(K):
        # Densité gaussienne
        densite[:,i] = multivariate_normal.pdf(X, mixture.mu[i,:], mixture.var[i])
        # P(x_i|theta) - loi à posteriori
        posterior[:,i] = densite[:,i]*mixture.p[i]
    vraisemblance = np.sum(posterior, axis=1)
    post = posterior / vraisemblance[:,None]
    log_vraisemblance = np.log(posterior.sum(axis=1)).sum()
    return post, log_vraisemblance
```

```python
def mstep(X: np.ndarray, post: np.ndarray) -> GaussianMixture:
    """M-step: Updates the gaussian mixture by maximizing the log-likelihood
    of the weighted dataset

    Args:
        X: (n, d) array holding the data
        post: (n, K) array holding the soft counts
            for all components for all examples

    Returns:
        GaussianMixture: the new gaussian mixture
    """
    d = X.shape[1]
    K = post.shape[1]
    mu = np.zeros([K, d])
    sigma = np.zeros(K)
    # p est la moyenne des p(j|i)
    p = post.mean(axis=0)

    for i in range(K):
        mu[i,:] = (post[:,i] @ X)/post[:,i].sum()
        sigma[i] = ( (post[:,i] @ (X - mu[i,:])**2) / post[:,i].sum() ).mean()

    return GaussianMixture(mu, sigma, p)
```

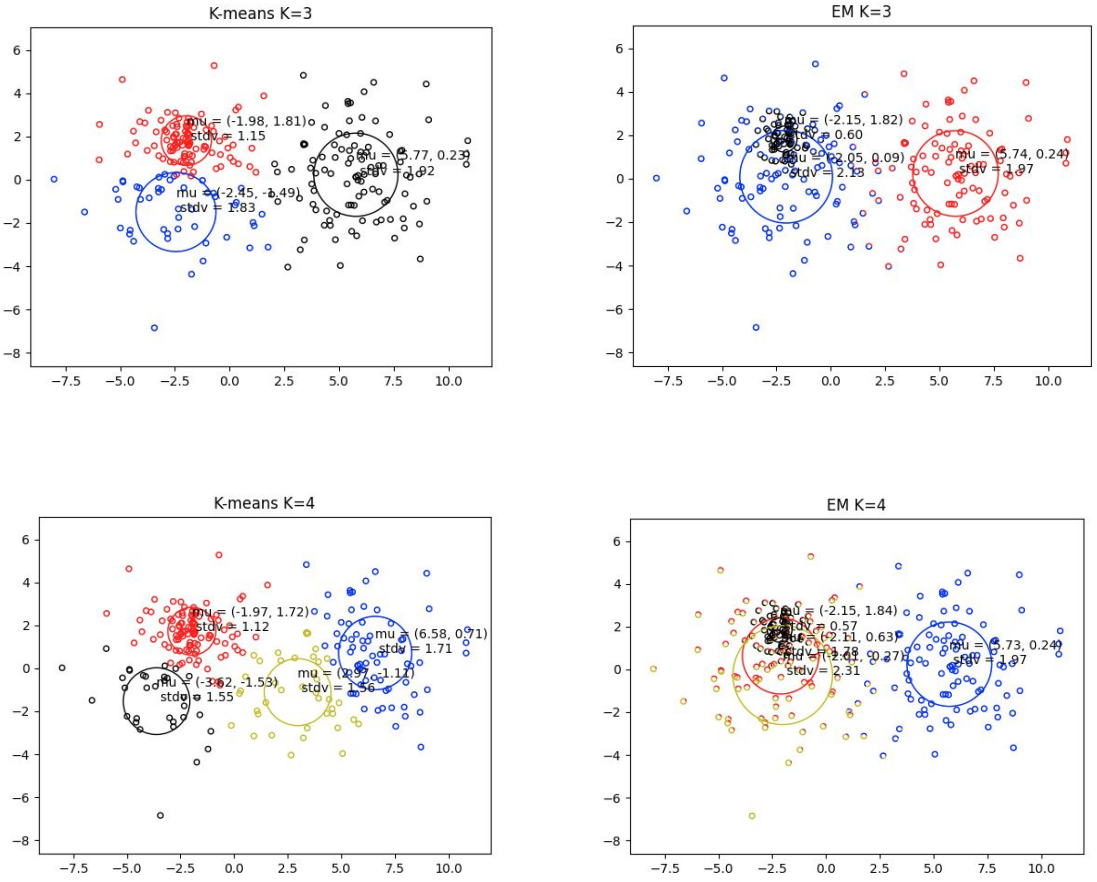
Le code de l'étape E reprend directement l'algorithme vu ci-dessus, on estime la probabilité que le point x_i appartienne au cluster j et on renvoie également la log-vraisemblance du modèle. L'étape M recalcule les paramètres de la mixture gaussienne (moyenne, variance et poids des composants de la gaussienne) pour pouvoir calculer de nouveau la probabilité à posteriori à l'itération suivante.

```
62  def run(X: np.ndarray, mixture: GaussianMixture,
63          post: np.ndarray) -> Tuple[GaussianMixture, np.ndarray, float]:
64      """Runs the mixture model
65
66      Args:
67          X: (n, d) array holding the data
68          post: (n, K) array holding the soft counts
69              for all components for all examples
70
71      Returns:
72          GaussianMixture: the new gaussian mixture
73          np.ndarray: (n, K) array holding the soft counts
74              for all components for all examples
75          float: log-likelihood of the current assignment
76      """
77      gauss_mixture = mixture
78      old_log_vraisemblance = np.inf
79      log_vraisemblance = 0
80      while abs(old_log_vraisemblance - log_vraisemblance) > 1e-6*abs(log_vraisemblance):
81          old_log_vraisemblance = log_vraisemblance
82          post, log_vraisemblance = estep(X, gauss_mixture)
83          gauss_mixture = mstep(X, post)
84      return gauss_mixture, post, log_vraisemblance
```

La fonction *run()* effectue la boucle dans laquelle on appelle alternativement les étapes E et M. Le critère d'arrêt de l'algorithme se porte sur la log-vraisemblance calculée à l'étape E : si la log-vraisemblance évolue peu d'une itération à une autre (écart inférieur à 10^{-6}), on sort de la boucle et on renvoie les paramètres estimés de la mixture gaussienne ainsi que les probabilités d'appartenance aux clusters et la log-vraisemblance.

Pour trouver les meilleurs paramètres, on peut faire tourner l'algorithme pour différents nombre de clusters K et pour des initialisations différentes. En effet, l'algorithme EM ne garantit qu'une convergence locale et non globale. La solution donnée peut alors ne pas être la meilleure et une initialisation, aléatoire, différente peut amener à un clustering différent. Pour limiter ce problème on peut essayer d'estimer au préalable les paramètres du modèles avec un autre algorithme plus simple comme KMeans, puis initialiser les paramètres avec les centres des clusters renvoyés par KMeans.

Voici les résultats renvoyés, pour le dataset *toy_data.txt* avec K=3 et K=4 et pour les meilleures seed qui sont choisies en fonction de la log-vraisemblance.



On remarque que les résultats donnés par EM et KMeans sont très différents. Là où KMeans ne pourrait pas déceler les classes qui se superposeraient, EM fonctionne parfaitement car il ne repose pas sur des distances entre les points mais sur les probabilités d'appartenance à une classe.

VI. Bilan du projet

Ce projet aura été l'occasion pour nous de revenir sur la notion de clustering, nécessaire pour un ingénieur en data science et/ou IA et sur les différentes manière d'aborder cette problématique. Nous avons pu revoir l'algorithme KMeans et les méthodes d'inférence Bayésienne et même compléter notre bagage dans ce domaine avec de nouveaux algorithmes pour nous comme l'algorithme de Maximisation-Espérance. En ce sens il est dommage que le contexte de pandémie actuelle ait dû raccourcir la durée de notre projet ce qui ne nous a pas laissé le temps de retravailler notre version de Naive Bayes et d'approfondir le développement de l'algorithme EM qui aurait pu être amélioré. Nous avons cependant pris le temps de découvrir la théorie qui se cache derrière ces algorithmes ce qui est pour nous, futurs ingénieurs, la réelle valeur ajoutée de ce projet. Nous tenons par ailleurs à remercier M. Victor Rabiet, qui a encadré notre projet, pour l'aide qu'il nous a apporté et pour la qualité de ses enseignements dans ce contexte particulier.

Bibliographie

- *Différence entre apprentissage supervisé et apprentissage non supervisé*, **ActuIA,** www.actuia.com/vulgarisation/difference-entre-apprentissage-supervise-apprentissage-non-supervise/
- *A quoi sert le clustering des données ?*, **JDN**, www.journaldunet.fr/web-tech/dictionnaire-du-webmastering/1203345-clustering-definition/
- *Tout ce que vous voulez savoir sur l'algorithme K-Means*, **Mr.Mint**, https://mrmint.fr/algorithme-k-means
- *6 6 Multinomial Naive Bayes A Worked Example*, **Mausam Jain**, https://www.youtube.com/watch?v=OWGVQfuvNMk&t=396s