Projet E4 : Netflix Clustering

Guillaume GAY | Kevin TOULCANON
Professeur référent : Victor RABIET

Sommaire

I. Introduction

- Entreprises produisent beaucoup de données

- Machine Learning et Data Scientist jouent un rôle important

- Apprentissage supervisée et apprentissage non supervisée

- Netflix Clustering

II. Clustering et KMeans

KMeans

Trois métriques :

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

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

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

INPUT: S, k **where** S = *set of classified instances*, k = *integer*
OUTPUT: k *Clusters*
Require: $S \neq \varnothing$, $k > 0$

```
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
```

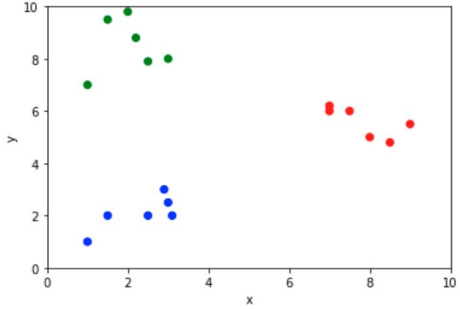
KMeans - fonction *fit()*

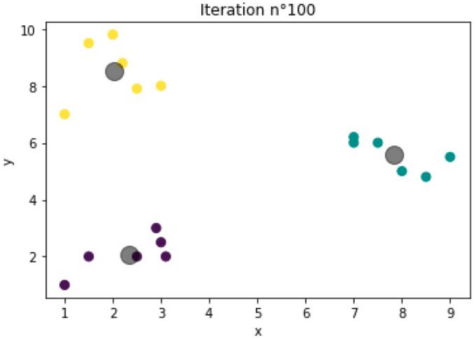
```
21      def fit(self, X, drawing=False):
22          """
23          On effectue un nombre d'itérations fixe défini par self.n_iter
24          """
25          if not isinstance(X, np.ndarray):
26              X = np.array(X)
27
28          # On choisi K points aléatoirement pour être les centroides
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
```

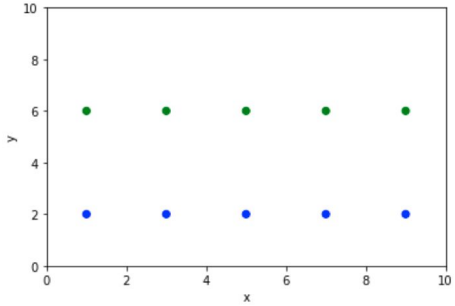
KMeans - *update_centroid()* et *predict_labels()*

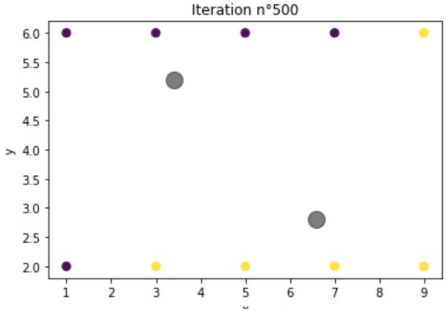
```python
    def predict_labels(self, X, centroids, dist='euc'):
        if dist in ['euc', 'l1']:
            return self.distance(X, centroids, dist).argmin(axis=1)
        elif dist=='cos':
            return abs(self.distance(X, centroids, dist)).argmin(axis=1)

    def distance(self, X, centroids, dist='euc'):
        map = {'euc':'euclidean', 'cos':'cosine', 'l1':'cityblock'}
        return sp.spatial.distance.cdist(X, centroids, metric=map[dist])

    def update_centroid(self, X):
        self.centroids = np.array([np.mean(X[self.labels == k], axis=0)  for k in range(self.K)])
        # On pourrait utiliser np.median au lieu de np.mean pour L1

    def draw_graph(self, X, i=0, show=True):
        plt.scatter(X[:, 0], X[:, 1], c=self.labels, s=50, cmap='viridis')
        plt.scatter(self.centroids[:, 0], self.centroids[:, 1], c='black', s=200, alpha=0.5)
        plt.xlabel('x'); plt.ylabel('y'); plt.title(f'Iteration n°{i}')
        self.graph = plt.gcf()
        if show: plt.show()
        else: plt.close(self.graph)
```

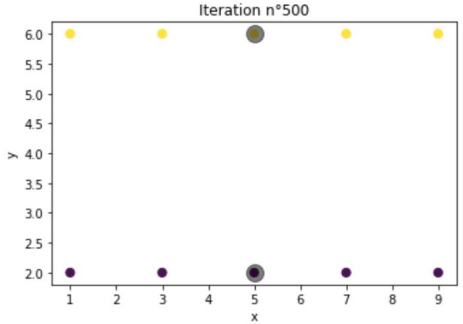
Différents tests

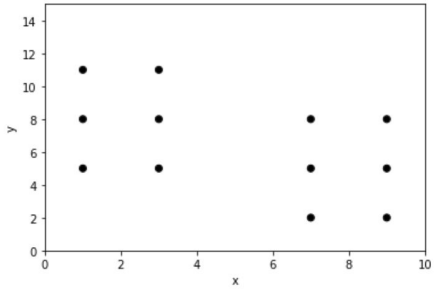




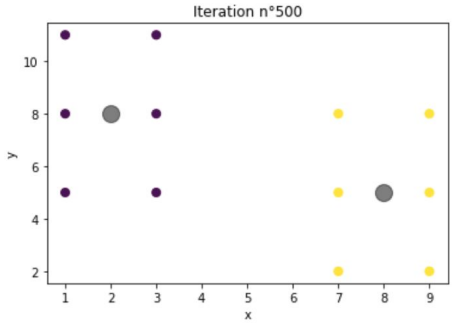




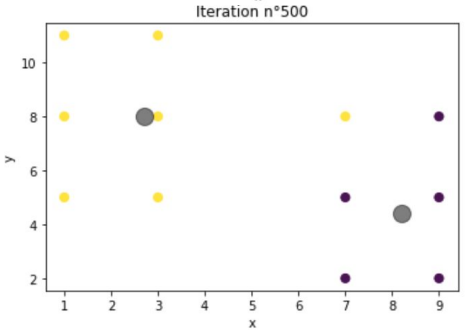




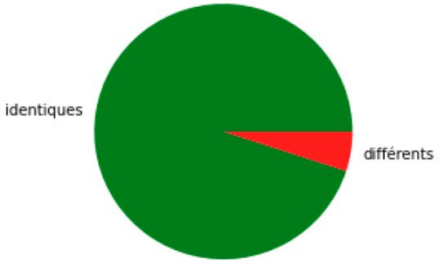
cos



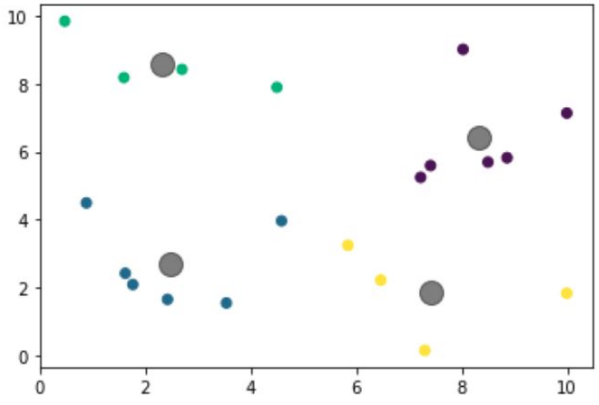
L1



Comparaison avec SKlearn

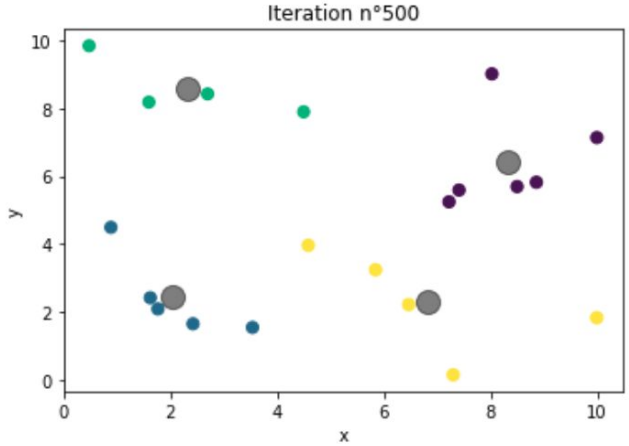



```
Labels : [0 1 2 3 3 2 1 1 0 0 1 1 2 0 3 1 0 2 3 0]
Inertie : 58.51195436463284
```



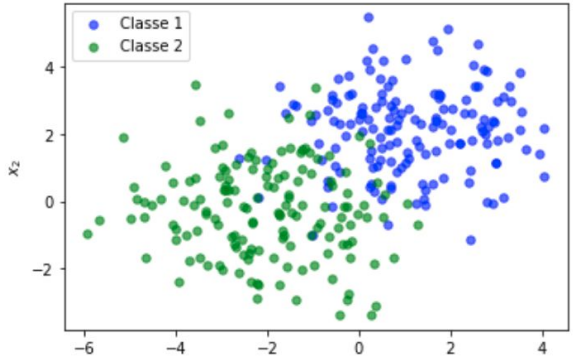
KMeans de SKlearn

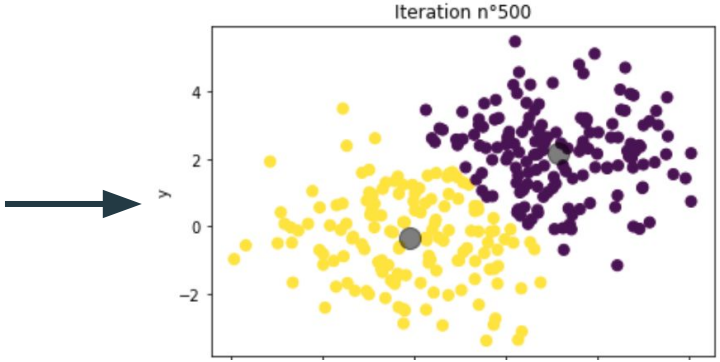
```
[0 1 2 3 3 2 1 1 0 0 3 1 2 0 3 1 0 2 3 0]
```



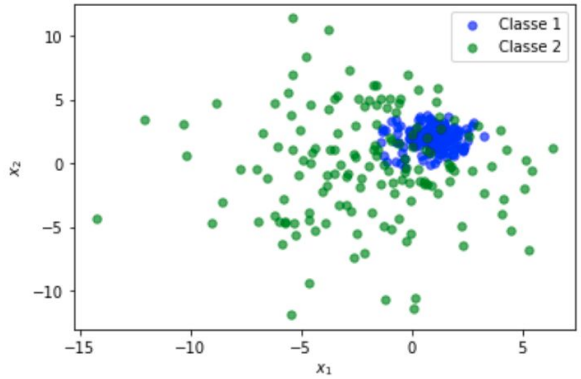
KMeans custom

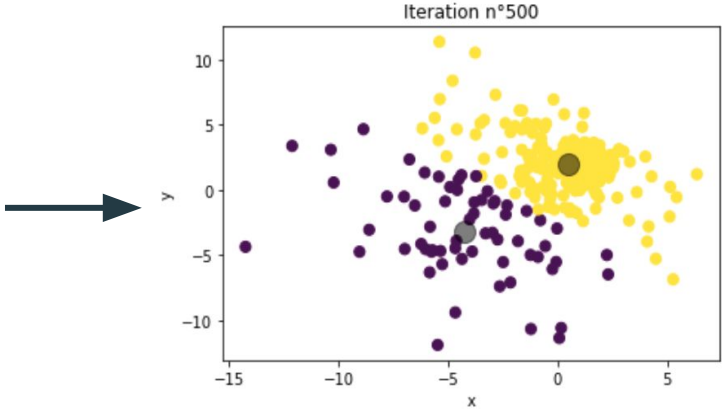
Travail sur des gaussiennes





$\sigma 1 \simeq \sigma 2$





$\sigma 1 << \sigma 2$

III. Variante de KMeans : KMedoids

KMedoids

- Reprend le fonctionnement de KMeans
- Les centroids sont forcés d'être des points des données

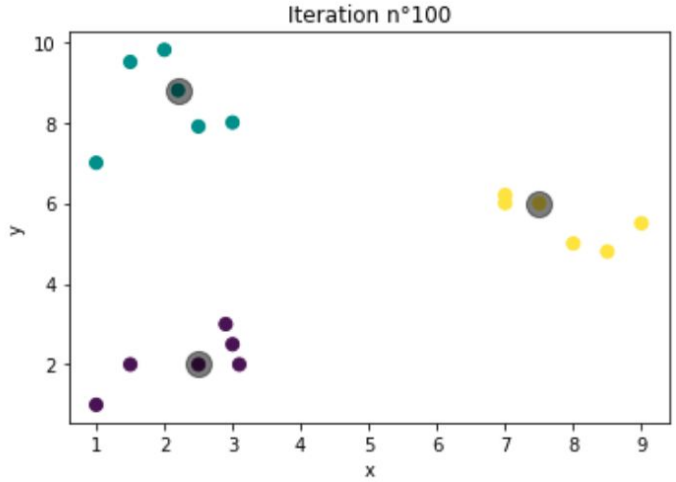
```
46      def distance(self, X, centroids, dist='euc'):
47          map = {'euc':'euclidean', 'cos':'cosine', 'l1':'cityblock'}
48          return sp.spatial.distance.cdist(X, centroids, metric=map[dist])
49
50      def update_centroid(self, X, dist='euc'):
51          self.centroids = np.array([np.mean(X[self.labels == 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
```

KMedoids

```
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
```

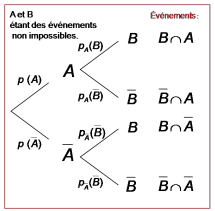


IV. Classification Naïve Bayésienne

Plusieurs étapes :

- Étape 1 : Le compte des mots

- Étape 2 : Calcul des probabilités



D'après le calcul de la probabilité d'un parcours, on a :

$$p\,(B \cap A) = p\,(A) \times p_A(B)$$

D'où :

$$p_A(B) = \frac{p\,(B \cap A)}{p\,(A)}$$

- Étape 3: Test de la classification

Étape 1

```
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
```

Étape 2

```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
```

Étape 3

```
        #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
```

Résultat

- 81.5 % prédictions justes données anglaises : 90% pour la classe neg et 73% pour la classe pos

- 71.5% prédictions justes données françaises

V. Algorithme E-M

Deux étapes :

- Étape E : Calcul de la vraisemblance des données
- Étape M : Maximisation de la vraisemblance et actualisation des paramètres
- Itérations tant que la log-vraisemblance varie significativement

Étape E

```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
```

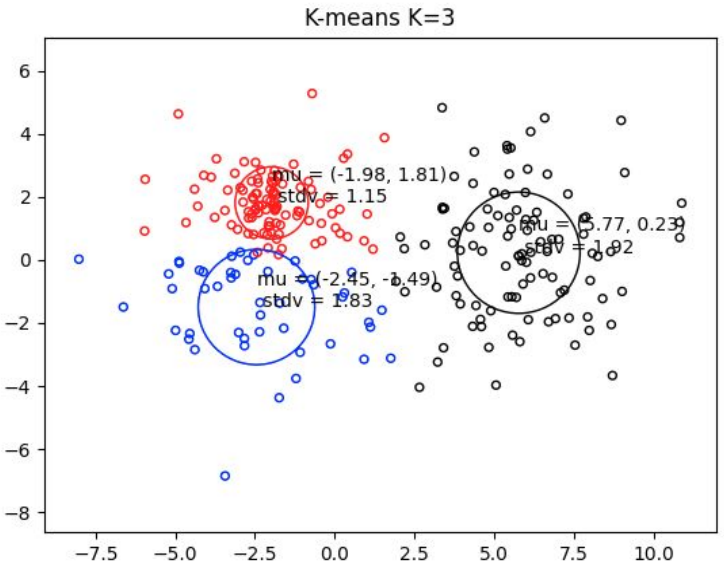
Étape M

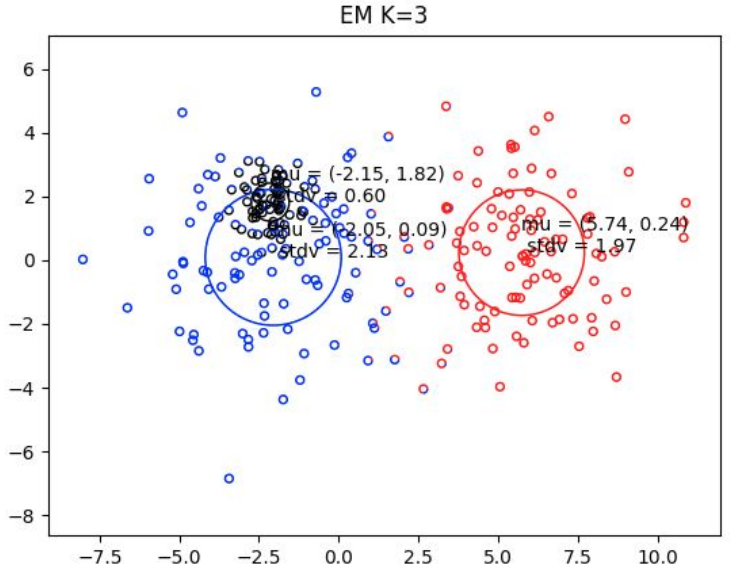
```python
def mstep(X: np.ndarray, post: np.ndarray) -> GaussianMixture:
    """M-step: Updates the gaussian mixture by maximizing the log-likelihood
    of the weighted dataset

    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)
```

Itérations

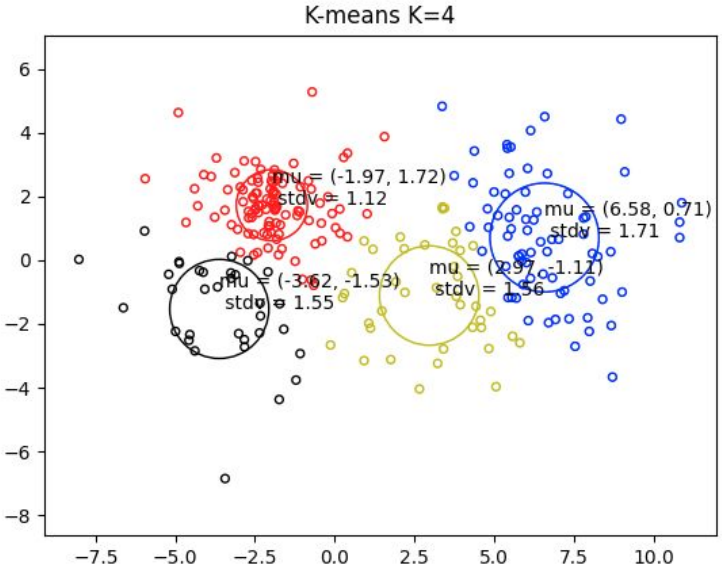
```python
def run(X: np.ndarray, mixture: GaussianMixture,
        post: np.ndarray) -> Tuple[GaussianMixture, np.ndarray, float]:
    """Runs the mixture model

    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
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: log-likelihood of the current assignment
    """
    gauss_mixture = mixture
    old_log_vraisemblance = np.inf
    log_vraisemblance = 0
    while abs(old_log_vraisemblance - log_vraisemblance) > 1e-6*abs(log_vraisemblance):
        old_log_vraisemblance = log_vraisemblance
        post, log_vraisemblance = estep(X, gauss_mixture)
        gauss_mixture = mstep(X, post)
    return gauss_mixture, post, log_vraisemblance
```

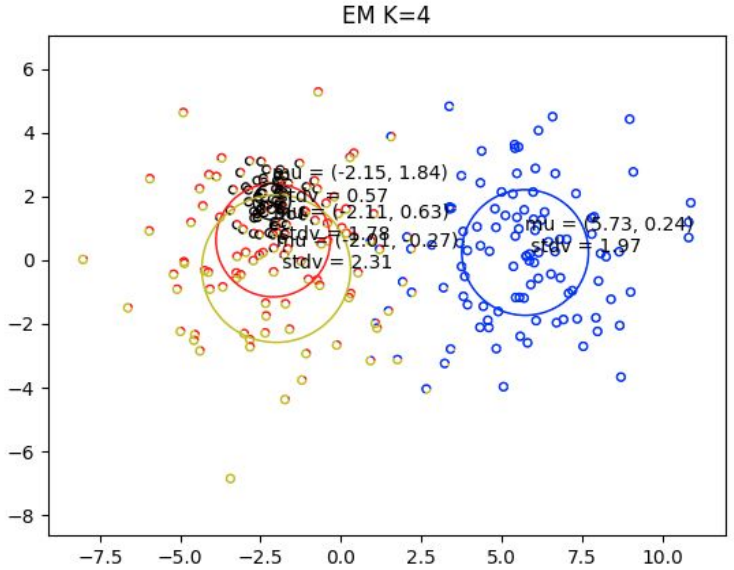
Résultats


K-means K=3
mu = (-1.98, 1.81)
stdv = 1.15
mu = (5.77, 0.23)
stdv = 1.92
mu = (-2.45, -1.49)
stdv = 1.83


EM K=3
mu = (-2.15, 1.82)
stdv = 0.60
mu = (2.05, 0.09)
stdv = 2.13
mu = (5.74, 0.24)
stdv = 1.97

Résultats


K-means K=4
mu = (-1.97, 1.72)
stdv = 1.12
mu = (6.58, 0.71)
stdv = 1.71
mu = (-3.62, -1.53)
stdv = 1.55
mu = (2.97, -1.11)
stdv = 1.56


EM K=4
mu = (-2.15, 1.84)
stdv = 0.57
mu = (-2.11, 0.63)
stdv = 1.78
mu = (5.73, 0.24)
stdv = 1.97
mu = (-2.01, 0.27)
stdv = 2.31

Conclusion et bilan