SY09 Printemps 2022
TD/TP 06 — *K-means*

Introduction

Ce sujet a pour ambition d'étudier l'algorithme des K-means, ainsi que l'une de ses extensions. Le sujet est conséquent, en partie du fait de la quantité d'informations complémentaires fournies. une bonne préparation du TP en amont rendra la réalisation du TP beaucoup plus aisée.

On veillera à préparer les parties 1.1 (étude de l'algorithme des *K-means*) et 2.1 (convergence des *K-means*). On pourra également préparer la partie 1.2 (étude de la stabilité de l'algorithme des *K-means*). La partie 1.4 aborde des aspects de robustesse des *K-means*, et peut être laissée de côté dans un premier temps.

Les parties 1.3 et 2.2 concernent l'algorithme des *K-means* adaptatifs, extension des *K-means* au cas de classes « non sphériques » : ces parties demandent un peu plus de recul, et leur réalisation sera bien plus aisée une fois que la méthode des *K-means* « classique » aura été comprise.

1 Travaux pratiques

1.1 Méthode des centres mobiles (*K-means*)

L'algorithme des *K-means* est implémenté dans la bibliothèque `scikit-learn` au travers de la classe `KMeans` disponible grâce à l'instruction suivante

```
from sklearn.cluster import KMeans
```

Les paramètres intéressants lors de la création d'un objet de classe `KMeans` sont les suivants
- `n_clusters` : le nombre de groupes dans la classification,
- `init` : la stratégie de choix des centres de départ,
- `n_init` : le nombre d'essais successifs avant de retourner le meilleur résultat (10 par défaut),
- `random_state` : un entier qu'on spécifie pour initialiser le générateur de nombre aléatoire pour que la classification soit reproductible : même entier, même résultat.

Par exemple, pour appliquer l'algorithme des *K-means* avec trois groupements en choisissant les centres de départ aléatoirement de manière uniforme et en réitérant 10 fois, on définit l'objet

```
cls = KMeans(n_clusters=3, init="random")
```

Pour apprendre la classification, on appelle la méthode `fit` sur l'objet `cls` en fournissant le tableau de données `X`

```
cls.fit(X)
```

Une fois que la méthode `fit` a été appelée, les attributs suivants sont disponibles
- `labels_` : les affectations de chaque individu aux groupes sous forme d'entiers,
- `cluster_centers_` : les centres des `n_clusters` groupes,
- `inertia_` : l'inertie finale du nuages de points.

Pour visualiser les groupements obtenus, on utilisera une représentation dans le premier plan factoriel grâce à la fonction fournie `scatterplot_pca` :

```
scatterplot_pca(columns=<column list>, hue=labels, data=<Pandas DataFrame>)
```

où `columns` est la liste des colonnes présentes dans `data` à réduire par ACP, `labels` les étiquettes correspondantes et `data` le *Dataframe* `Pandas` contenant les données.

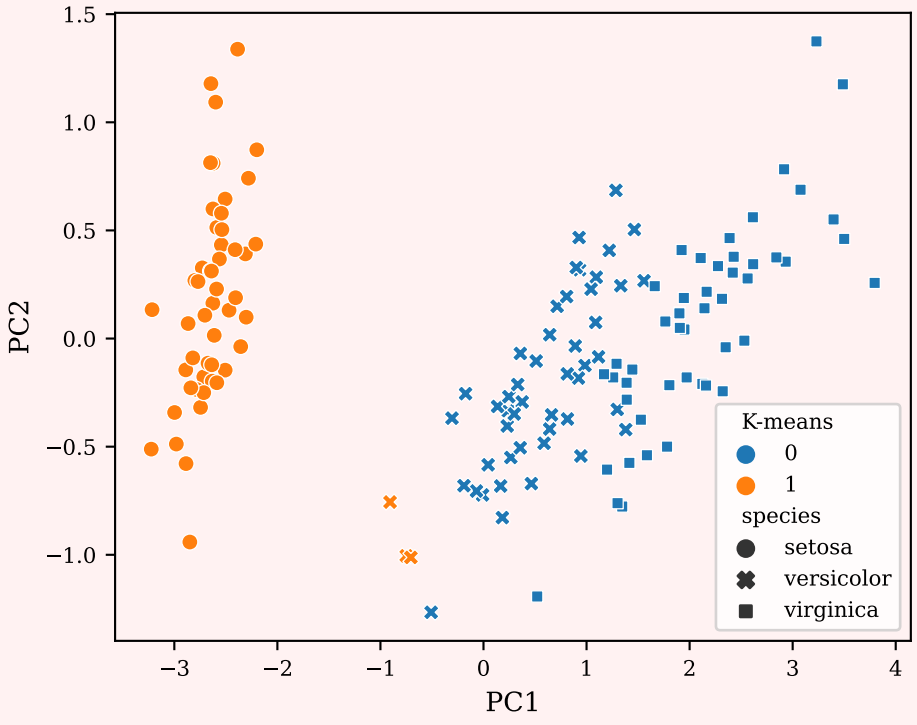
On peut également fournir une autre classification pour comparaison :

```
scatterplot_pca(
    columns=<column list>, hue=labels, style=labels2, data=<Pandas DataFrame>
)
```

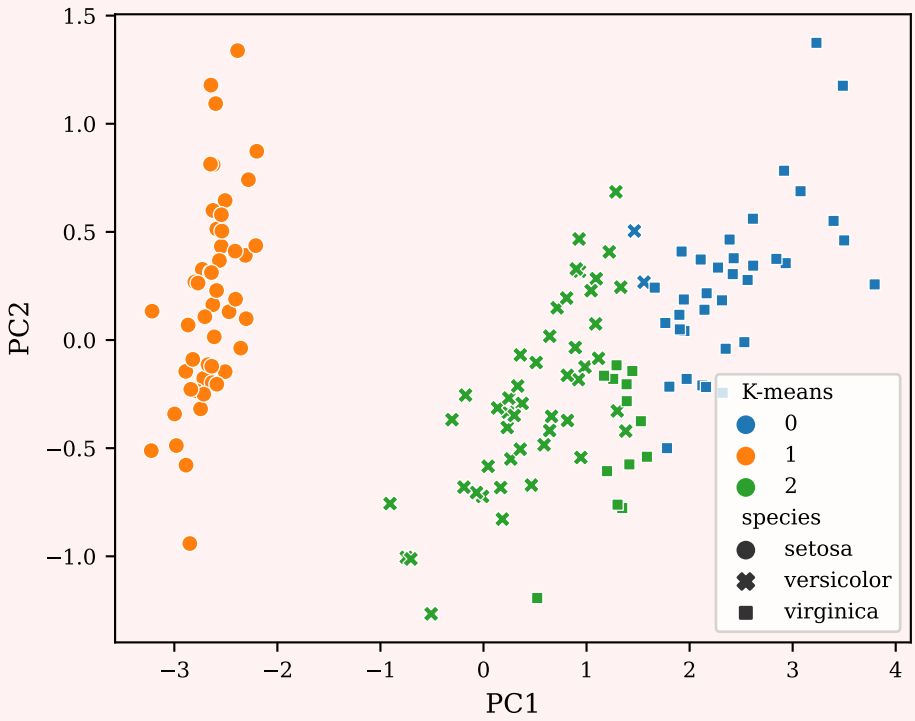
1.1.1 Étude des *K-means* sur Iris

1 Tenter une partition en $K \in \{2, 3, 4\}$ classes. Visualiser et commenter.

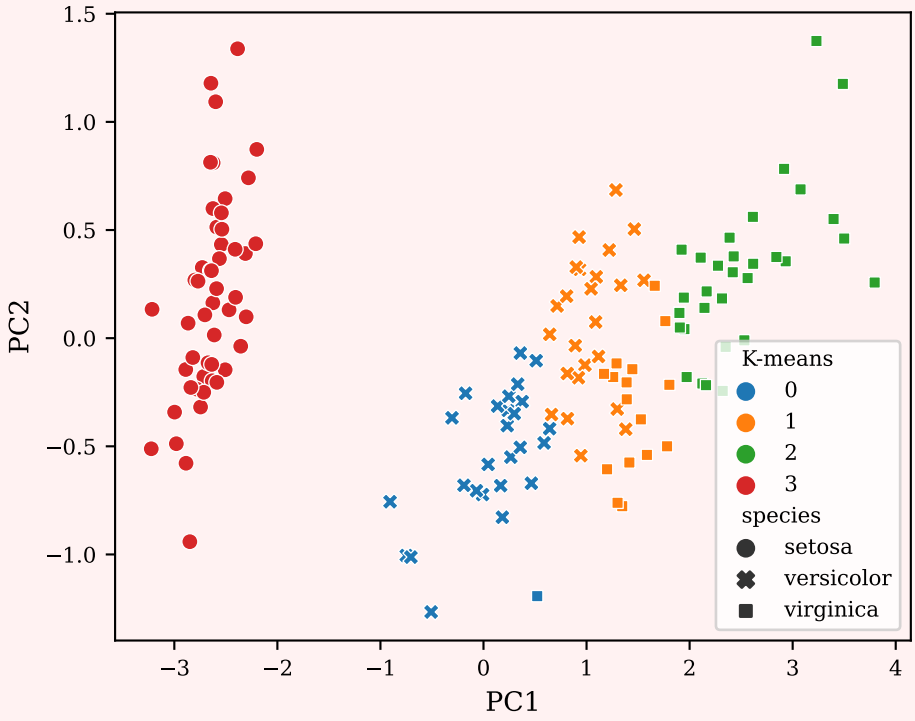
```
In [1]: iris = sns.load_dataset("iris")
        from sklearn.cluster import KMeans
        iris0 = iris.drop(columns=["species"])

        # 2 groupements
        cls = KMeans(n_clusters=2, init="random")
        cls.fit(iris0)
        labels = pd.Series(cls.labels_, name="K-means")
        scatterplot_pca(data=iris0, hue=labels, style=iris.species)
        plt.show()

        # 3 groupements
```



In [2]:
```
cls = KMeans(n_clusters=3, init="random")
cls.fit(iris0)
labels = pd.Series(cls.labels_, name="K-means")
scatterplot_pca(data=iris0, hue=labels, style=iris.species)
plt.show()

# 4 groupements
```



In [3]:
```
cls = KMeans(n_clusters=4, init="random")
cls.fit(iris0)
labels = pd.Series(cls.labels_, name="K-means")
scatterplot_pca(data=iris0, hue=labels, style=iris.species)
plt.show()
```



L'espèce Setosa située à gauche est toujours bien identifié car elle est esseulée. Les deux autres espèces sont proches. Pour $K = 3$, on retrouve à peu près les trois espèces. Pour $K = 4$, la partition ne fait pas grand sens d'après ce qu'on sait du jeu de données.

Pour étudier plus en détails l'algorithme des *K-means*, nous allons constituer un jeu de données portant sur les résultats obtenus en appliquant cette méthode au même jeu de données avec différents paramètres.

2 Compléter la fonction ci-dessous qui construit ce jeu de données. Les paramètres à faire varier sont les suivants :

— la liste de nombre de groupements considérés,
— la stratégie d'initialisation des centres,
— le nombre d'essais successifs lorsque les paramètres ci-dessus sont fixés.

Par défaut, `scikit-learn` lance 10 fois la méthode des *K-means* et sélectionne le meilleur résultat. Afin de ne pas fausser l'analyse, on n'oubliera pas de spécifier l'argument `n_init`.

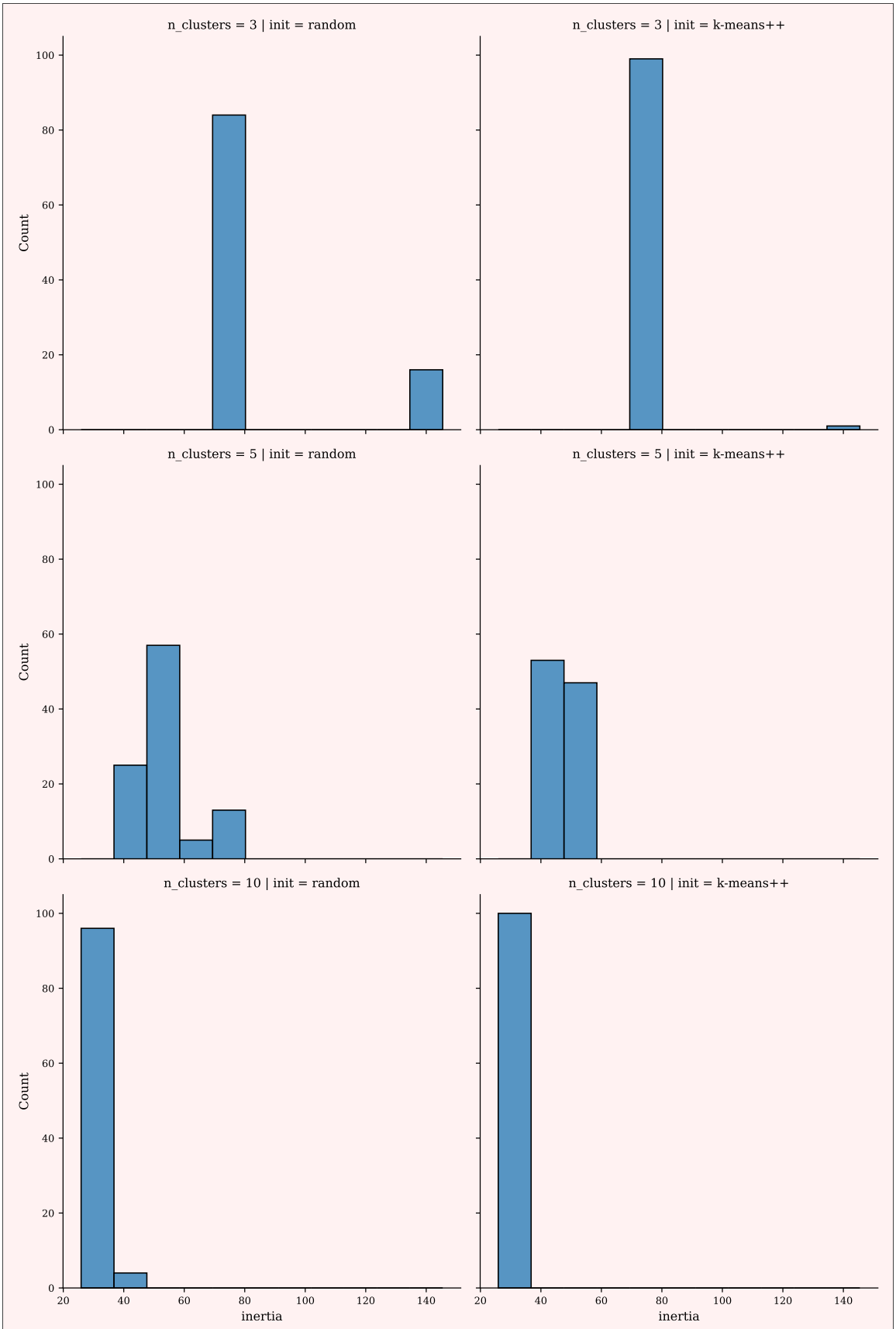
```
def kmeans_dataset(dataset, n_clusters_list, strategies, tries):
    for n_clusters in n_clusters_list:
        for strategy in strategies:
            for rs in range(tries):  # On utilisera `rs` pour fixer le
            ↪  `random_state`
                km = ...
                inertia = ...
                yield rs, strategy, n_clusters, inertia


gen = kmeans_dataset(...)

df = pd.DataFrame(gen, columns=["seed", "init", "n_clusters", "inertia"])
df = df.astype({
    "seed": "int32",
```

```
        "n_clusters": "int32"
    })
```

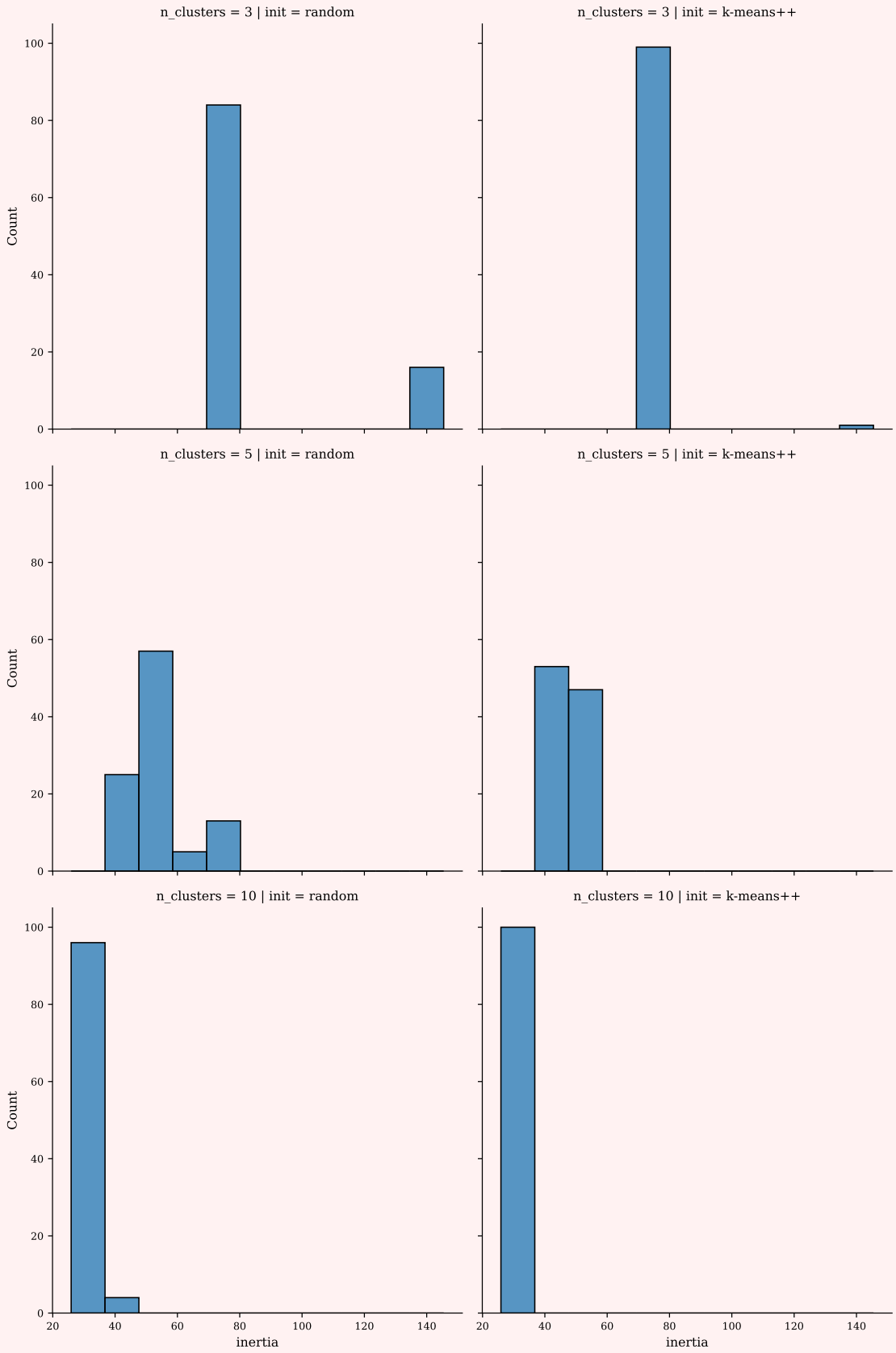
```
In [4]: def kmeans_dataset(dataset, n_clusters_list, strategies, tries):
            for n_clusters in n_clusters_list:
                for strategy in strategies:
                    for rs in range(tries):  # On utilisera `rs` pour fixer le
                    ↪ `random_state`
                        # <answer>
                        inertia = (
                            KMeans(
                                n_clusters=n_clusters,
                                n_init=1,
                                random_state=rs,
                                init=strategy,
                            )
                            .fit(dataset)
                            .inertia_
                        )
                        # </answer>
                        yield rs, strategy, n_clusters, inertia


        # <answer>
        gen = kmeans_dataset(iris0, [3, 5, 10], ["random", "k-means++"], 100)
        # </answer>

        df = pd.DataFrame(gen, columns=["seed", "init", "n_clusters",
        ↪ "inertia"])
        df = df.astype({
            "seed": "int32",
            "n_clusters": "int32"
        })
```

3 Visualiser la distribution des inerties en fonction du nombre de groupements et de la stratégie d'initialisation des centres. Quelles observations peut-on faire ?

```
In [5]: sns.displot(df, x="inertia", row="n_clusters", col="init")
```


n_clusters = 3 | init = random
n_clusters = 3 | init = k-means++
n_clusters = 5 | init = random
n_clusters = 5 | init = k-means++
n_clusters = 10 | init = random
n_clusters = 10 | init = k-means++
Count
inertia
0
20
40
60
80
100
120
140

In [6]: `plt.show()`



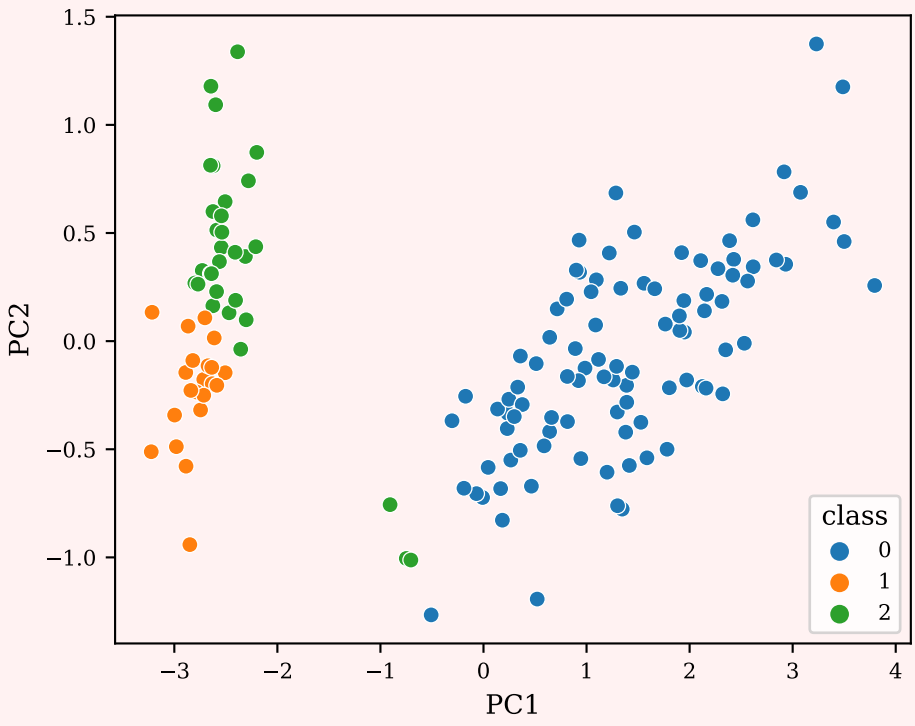
Tout d'abord, on observe que lorsque le nombre de groupements augmente, l'inertie intra-classe diminue quelque soit la stratégie d'initialisation des centres.

En revanche l'algorithme semble de plus en plus instable lorsque le nombre de groupements augmente. Cela se manifeste par une dispersion (relative) de la distribution des inerties.

Cette dispersion, qui peut donner des mauvaises classifications en terme d'inertie, est largement réduite en utilisant une stratégie d'initialisation des centres plus performante (`k-means++`).

4 Visualiser le résultat des *K-means* avec une *graine* issue de jeu de données et générant une mauvaise classification.

```
In  [7]: # Les indices des plus mauvaises classifications par stratégie
         # d'initialisation et nombre de groupements.
         bad = df.loc[df.groupby(["init", "n_clusters"])["inertia"].idxmax()]

         # Une mauvaise graine
         bad_seed = bad.loc[(bad.init == "random") & (bad.n_clusters ==
         ↪  3)].seed.iloc[0]

         cls = KMeans(n_clusters=3, n_init=1, random_state=bad_seed,
         ↪  init="random").fit(iris0)
         cls.inertia_

Out [7]: 145.45269176485027

In  [8]: scatterplot_pca(data=iris0, hue=cls.labels_)
         plt.show()
```

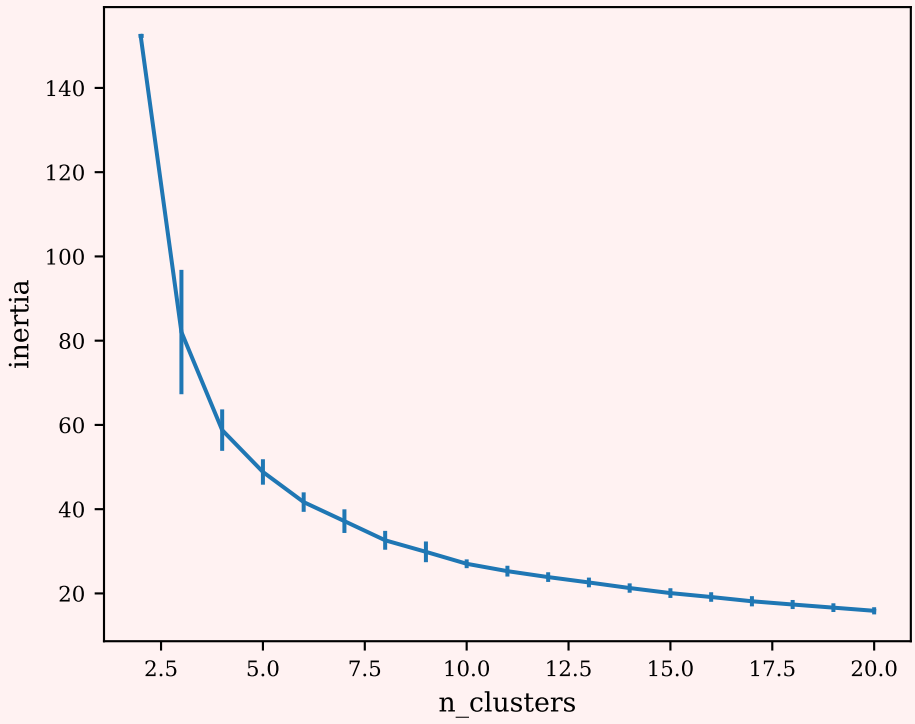


Un groupement naturel (à gauche) est coupé en deux d'où la mauvaise inertie. La solution est néanmoins stable (les affectations ne changent pas lors de la mise à jour des centres).

5 Visualiser les inerties pour la stratégie d'initialisation `k-means++` en fonction du nombre de groupements et sélectionner le nombre de groupements le plus vraisemblable avec la règle du coude.

On pourra générer un autre jeu de données avec des nombres de groupements contigus.

```
In [9]: gen = kmeans_dataset(iris0, range(2, 21), ["k-means++"], 20)
        df = pd.DataFrame(gen, columns=["seed", "init", "n_clusters",
        ↪  "inertia"])
        sns.lineplot(
            x="n_clusters",
            y="inertia",
            data=df.loc[df.init == "k-means++"],
            err_style="bars",
            ci="sd",
        )
        plt.show()
```

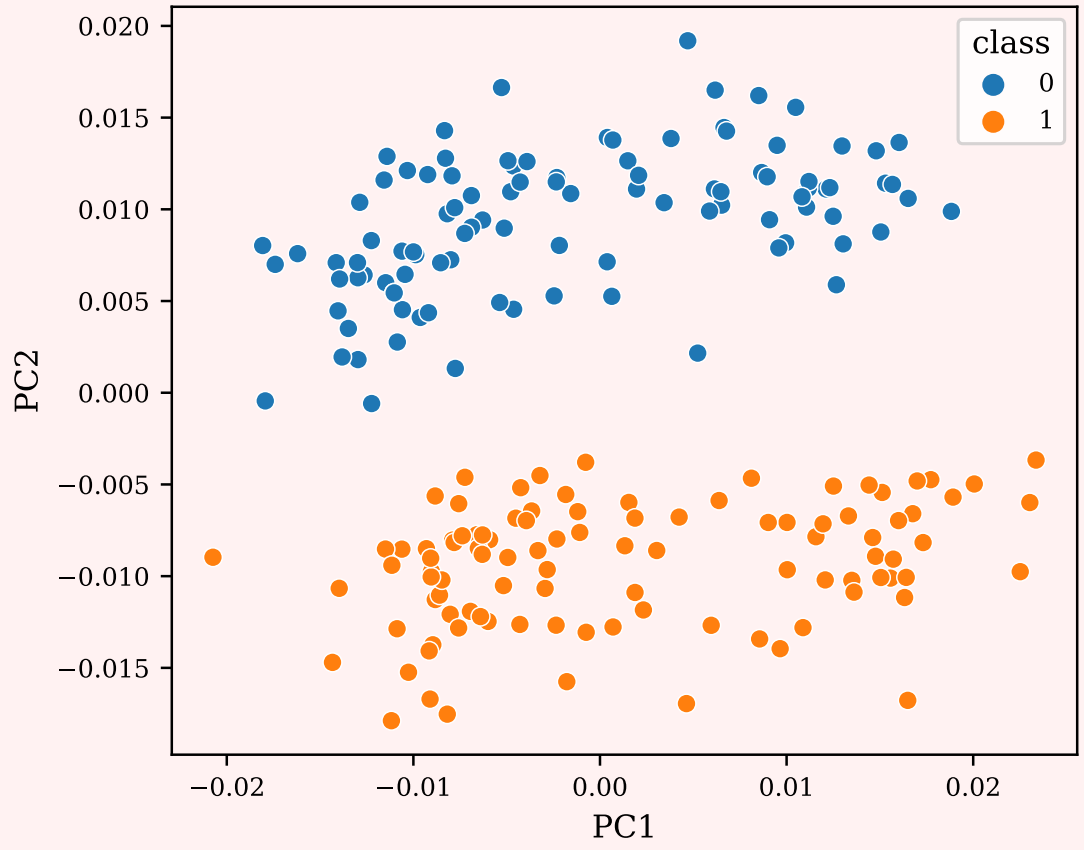


D'après la règle du coude un bon nombre de groupements est 3 voire 4.

1.1.2 Données Crabs

6 Effectuer plusieurs classifications en $K = 2$ classes des données pré-traitées de manière à supprimer l'effet taille.

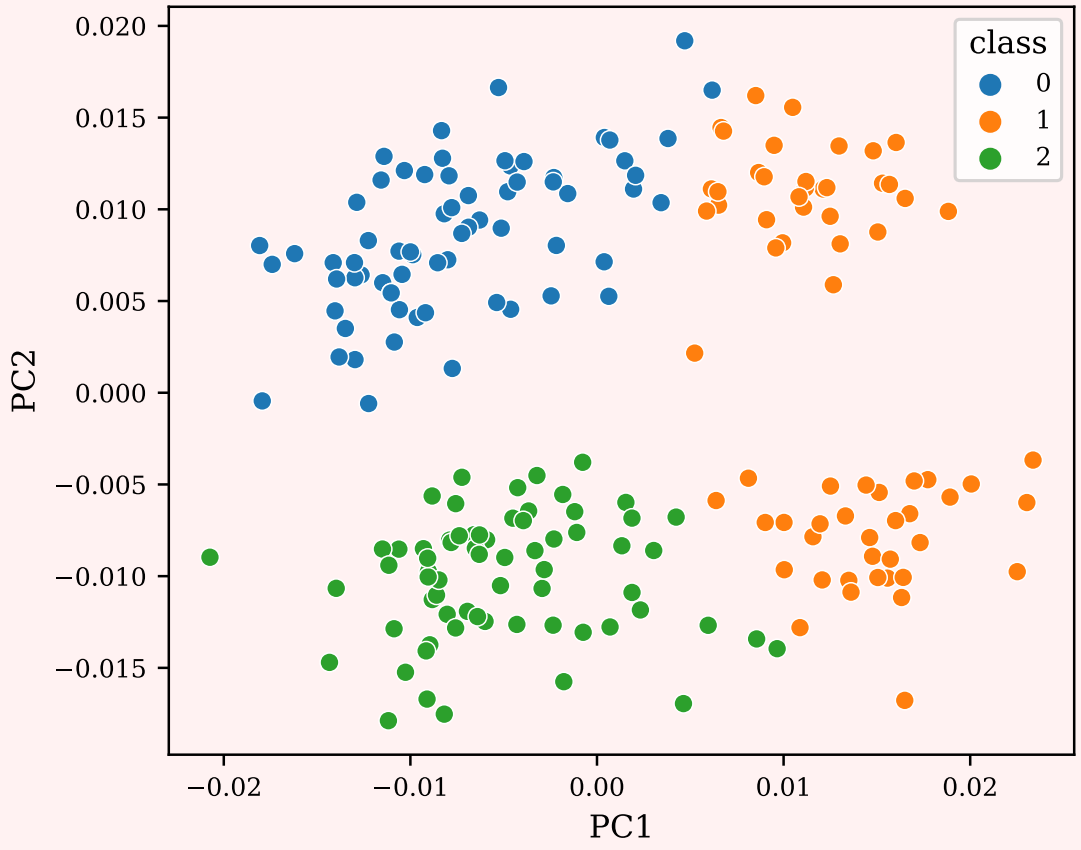
```
In [10]: # Données + prétraitement
         crabs = pd.read_csv("data/crabs.csv", sep="\s+")
         crabsquant = crabs.iloc[:, 3:8]
         crabsquant = crabsquant.apply(lambda row: row / sum(row), axis=1)

         cls = KMeans(n_clusters=2).fit(crabsquant)
         scatterplot_pca(data=crabsquant, hue=cls.labels_)
         plt.show()

         # Les inerties de 1000 K-means
```




```
In  [11]: inertias = [KMeans(n_clusters=2, init="random", n_init=1,
          ↪  random_state=i).fit(crabsquant).inertia_ for i in range(1000)]

          # Indices des classifications d'inerties différentes
          inertias, idxs_uniq, idxs = np.unique(inertias, return_index=True,
          ↪  return_inverse=True)

          # Fréquence des inerties
          np.bincount(idxs)

          # Les inerties uniques
Out [11]: array([439,   1, 544,  11,   5])

In  [12]: inertias

Out [12]: array([0.0276453 , 0.0276453 , 0.02826445, 0.02826445, 0.03149694])
```

7 Effectuer 1000 classifications des données en $K = 3$ classes. Qu'observe-t-on ?

```
In  [13]: cls = KMeans(n_clusters=3).fit(crabsquant)
          scatterplot_pca(data=crabsquant, hue=cls.labels_)
          plt.show()
```




```
In  [14]: inertias = [KMeans(n_clusters=3, init="random", n_init=1,
          ↪  random_state=i).fit(crabsquant).inertia_ for i in range(1000)]

          # Indices des classifications d'inerties différentes
          inertias, idxs_uniq, idxs = np.unique(inertias, return_index=True,
          ↪  return_inverse=True)

          # Fréquence des inerties
          np.bincount(idxs)

          # Les inerties uniques

Out [14]: array([225,  32,   5,  29,   1,   4,  12,  48,  70,   6,   9,   5,   2,
          ↪  3,   6,   2,   8,   4,   3,   7,  13,   8,  91,  12,   4,  18,   5,
          ↪  7,   4,  29,  20,   1,   1,   5, 151,  25,   2,   4, 114,   5])

In  [15]: inertias

Out [15]: array([0.01734867, 0.01735593, 0.01735593, 0.01736761, 0.01736761,
          ↪  0.01740171, 0.01740307, 0.01740325, 0.01740325, 0.01740441,
          ↪  0.017407  , 0.01797298, 0.01797706, 0.01800373, 0.01801256,
          ↪  0.01801487, 0.01801487, 0.01801556, 0.0180331 , 0.01803493,
          ↪  0.01804046, 0.01805939, 0.01806565, 0.01825852, 0.01825852,
          ↪  0.01826204, 0.01828601, 0.01830076, 0.0183031 , 0.01830477,
          ↪  0.01830477, 0.01833003, 0.01833003, 0.01833014, 0.01833167,
          ↪  0.01905302, 0.01905302, 0.01905309, 0.01915905, 0.02009844])
```

Il y a beaucoup plus d'inerties différentes avec des fréquences non négligeables. Cela montre l'instabilité de la classification en 3 classes. En effet, ce jeu de données est plus adapté à la classification

en 2 ou 4 classes.

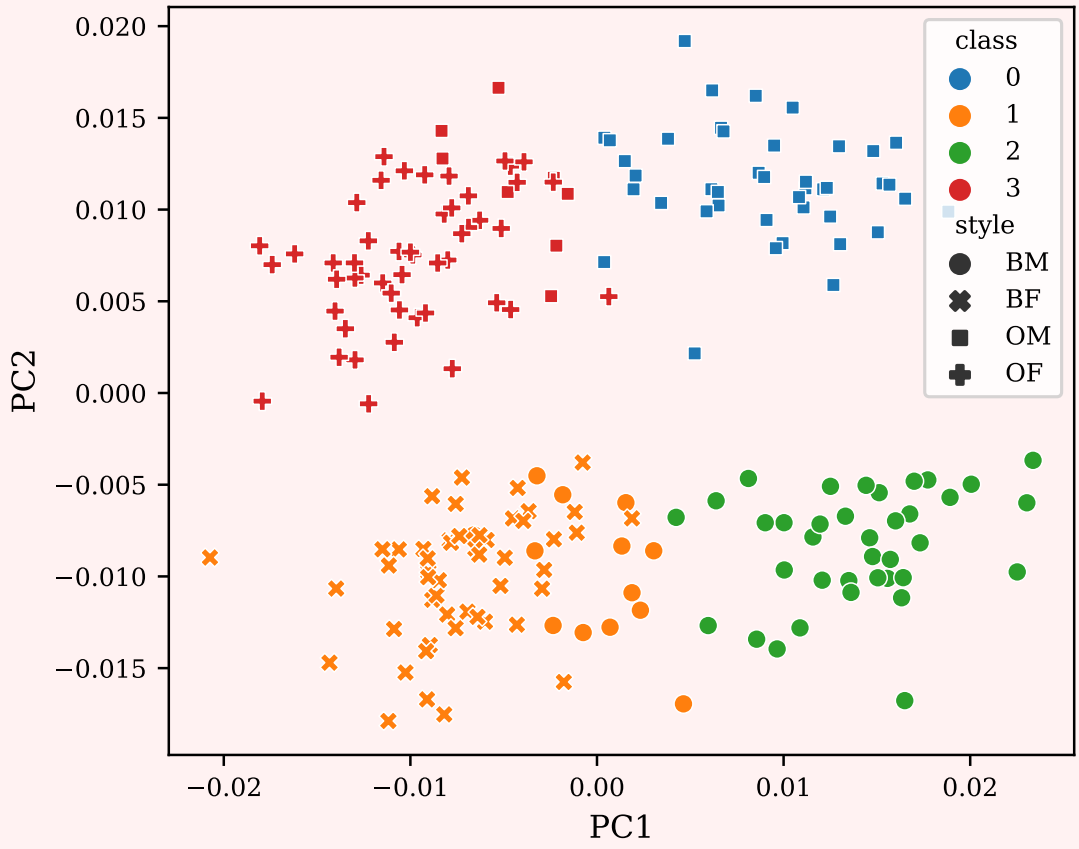
8 Effectuer une classification en $K = 4$ classes des données. Comparer à la partition réelle suivant l'espèce et le sexe. Que peut-on conclure ? On pourra utiliser l'indice de Rand ajusté :

```
from sklearn.metrics import adjusted_rand_score
```

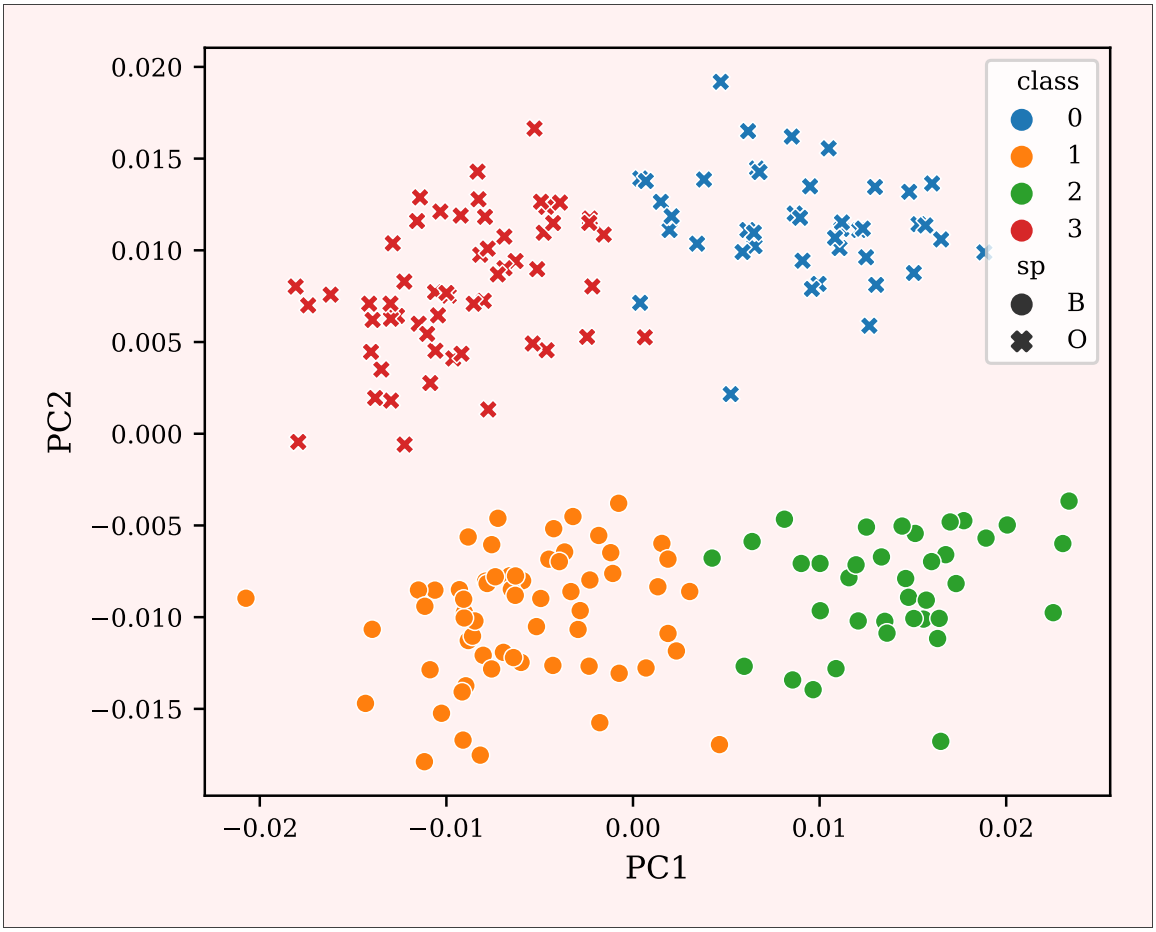
```
In [16]: from sklearn.metrics import adjusted_rand_score
         cls = KMeans(n_clusters=4).fit(crabsquant)
         adjusted_rand_score(crabs.sp+crabs.sex, cls.labels_)
```

```
Out [16]: 0.7512388410567427
```

```
In [17]: scatterplot_pca(data=crabsquant, hue=cls.labels_,
         ↪  style=crabs.sp+crabs.sex)
         plt.show()
```




```
In [18]: scatterplot_pca(data=crabsquant, hue=cls.labels_, style=crabs.sp)
         plt.show()
```

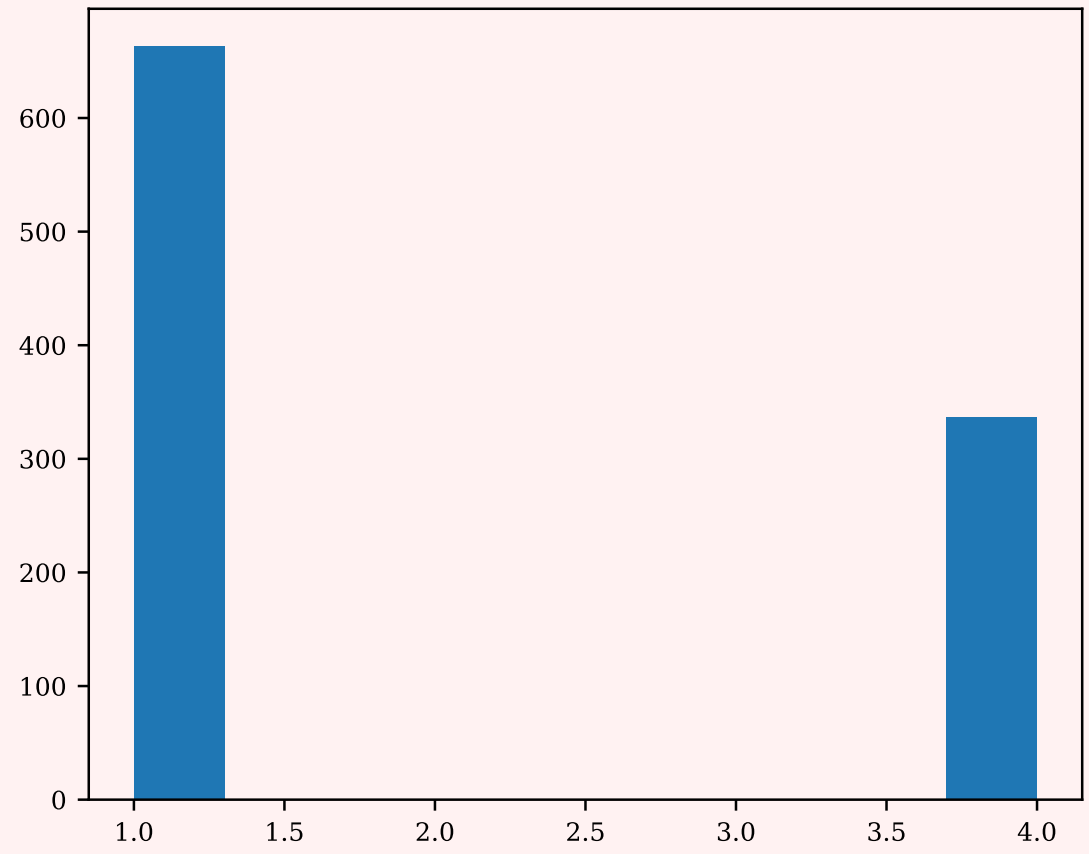


1.1.3 De l'intérêt d'une bonne initialisation

Soit un jeu de données de quatre points dans le plan formant un rectangle de grand côté a et de petit coté b. On veut obtenir une classification en deux groupes.

9 Vérifier expérimentalement que la proportion de mauvaise classification avec l'initialisation « random » est de 1/3 quel que soit a et $b < a$.

```
In [19]: a = 2
         b = 1
         X = np.array([[a / 2, b / 2], [a / 2, -b / 2], [-a / 2, b / 2], [-a /
         ↪ 2, -b / 2]])
         inertias = [
             KMeans(n_clusters=2, n_init=1, random_state=i,
             ↪ init="random").fit(X).inertia_
             for i in range(1000)
         ]
         plt.hist(inertias)
         plt.show()
```

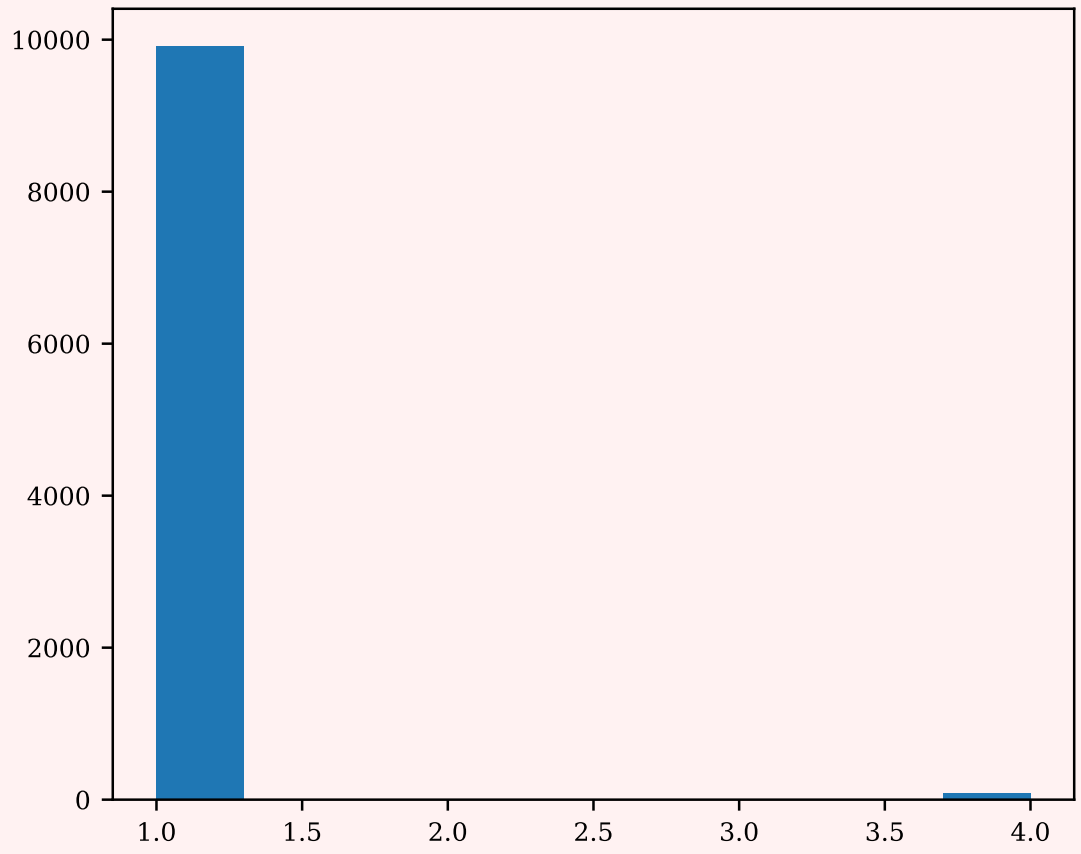



```
In  [20]: _, counts = np.unique(inertias, return_counts=True)
          counts
Out [20]: array([663, 337])
```

10 Vérifier expérimentalement que la proportion de mauvaise classification avec l'initialisation `k-means++` pour ce problème vaut

$$\frac{1}{4(1+a^2/b^2)^2}.$$

```
In  [21]: tries = 10000
          inertias = [
              KMeans(n_clusters=2, n_init=1, random_state=i).fit(X).inertia_ for
              ↪  i in range(tries)
          ]
          plt.hist(inertias)
          plt.show()
```




```
In  [22]: _, counts = np.unique(inertias, return_counts=True)
          min(counts) / tries
Out [22]: 0.0089
In  [23]: 1 / (4 * (1 + (a / b) ** 2) ** 2)
Out [23]: 0.01
```

1.2 Visualisation de classifications

L'inertie intra-classe est un indicateur de la qualité d'une classification mais elle ne renseigne en rien sur la similarité de deux classifications. Dans cette section, nous allons visualiser différentes classifications en nous appuyant sur l'indice de Rand et le positionnement multidimensionnel.

Cette section fait suite à la question 7 de la section 1.1.2.

11 On suppose $K = 3$ et on génère 1000 classifications. Parmi ces 1000 classifications, isoler un sous-ensemble de classifications uniques. On supposera que deux classifications sont identiques si les inerties correspondantes sont les mêmes.

On pourra utiliser la fonction `np.unique` avec les bons arguments.

```
In [24]: cls = KMeans(n_clusters=3).fit(crabsquant)
         inertias = [
             KMeans(n_clusters=3, init="random", n_init=1, random_state=i)
             .fit(crabsquant)
             .inertia_
             for i in range(1000)
         ]

         inertias, idxs_uniq, idxs = np.unique(inertias, return_index=True,
         ↪  return_inverse=True)
```

Le tableau `idxs_uniq` contient les « graines » donnant des classifications toutes différentes.

12 À l'aide de l'indice de Rand, calculer un tableau de similarité entre les classifications isolées à la question précédente. En déduire un tableau de dissimilarité.

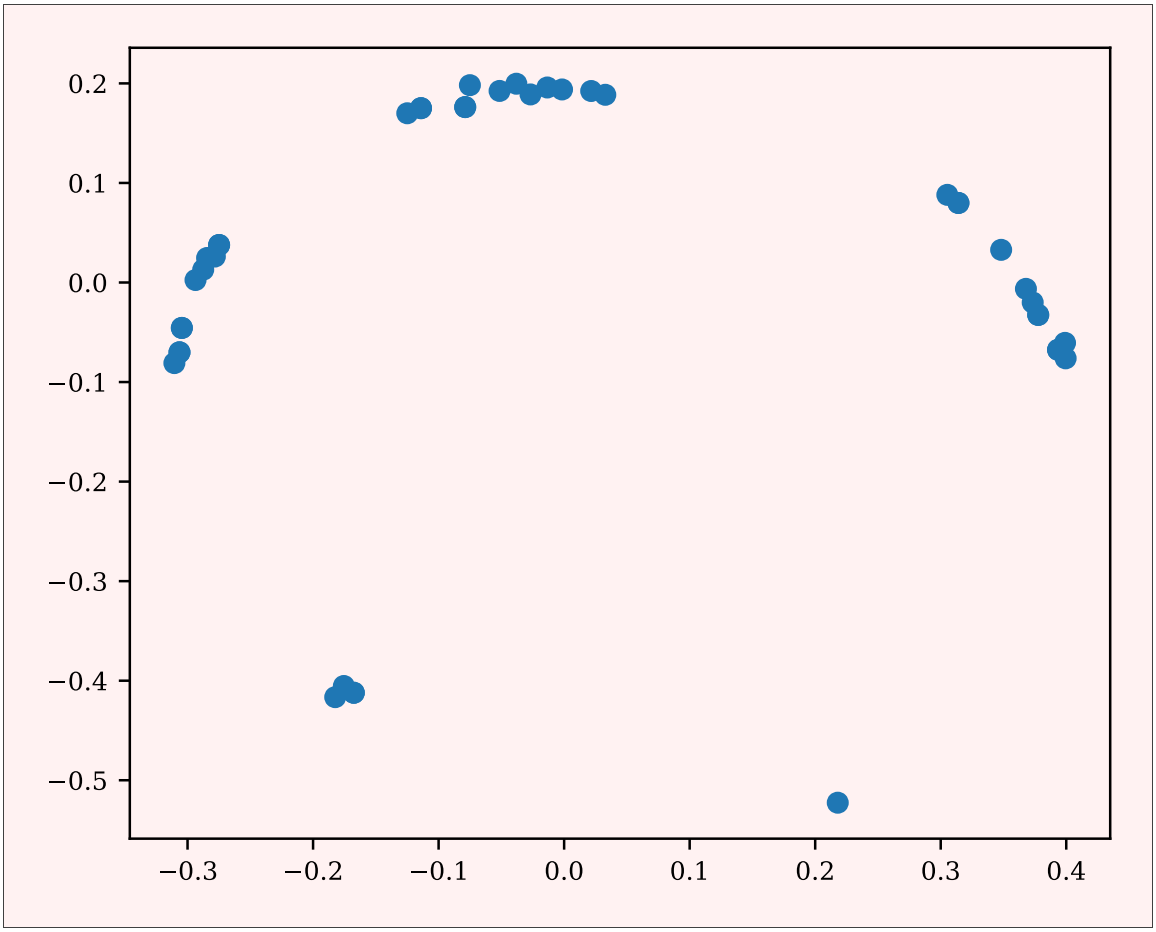
On pourra utiliser la fonction `pairwise_distances` présente dans le module `sklearn.metrics`.

```
In [25]: # On extrait les groupements différents (par leur inertie).
         # `clusterings` est de taille `n*200` avec `n` le nombre d'inerties
         # différentes.
         clusterings = np.array([
             KMeans(n_clusters=3, init="random", n_init=1, random_state=i)
             .fit(crabsquant)
             .labels_
             for i in idxs_uniq
         ])

         # Similarité des différents groupements précédemment extraits.
         from sklearn.metrics import pairwise_distances
         from sklearn.metrics import adjusted_rand_score
         S = pairwise_distances(clusterings, metric=adjusted_rand_score)

         # Transformation en dissimilarité avant AFTD (S <= 1)
         D = 1 - S
```

L'indice de Rand ajusté peut être négatif. On force donc à 0 et on transforme en distance.

13 Visualiser les classifications avec une AFTD.

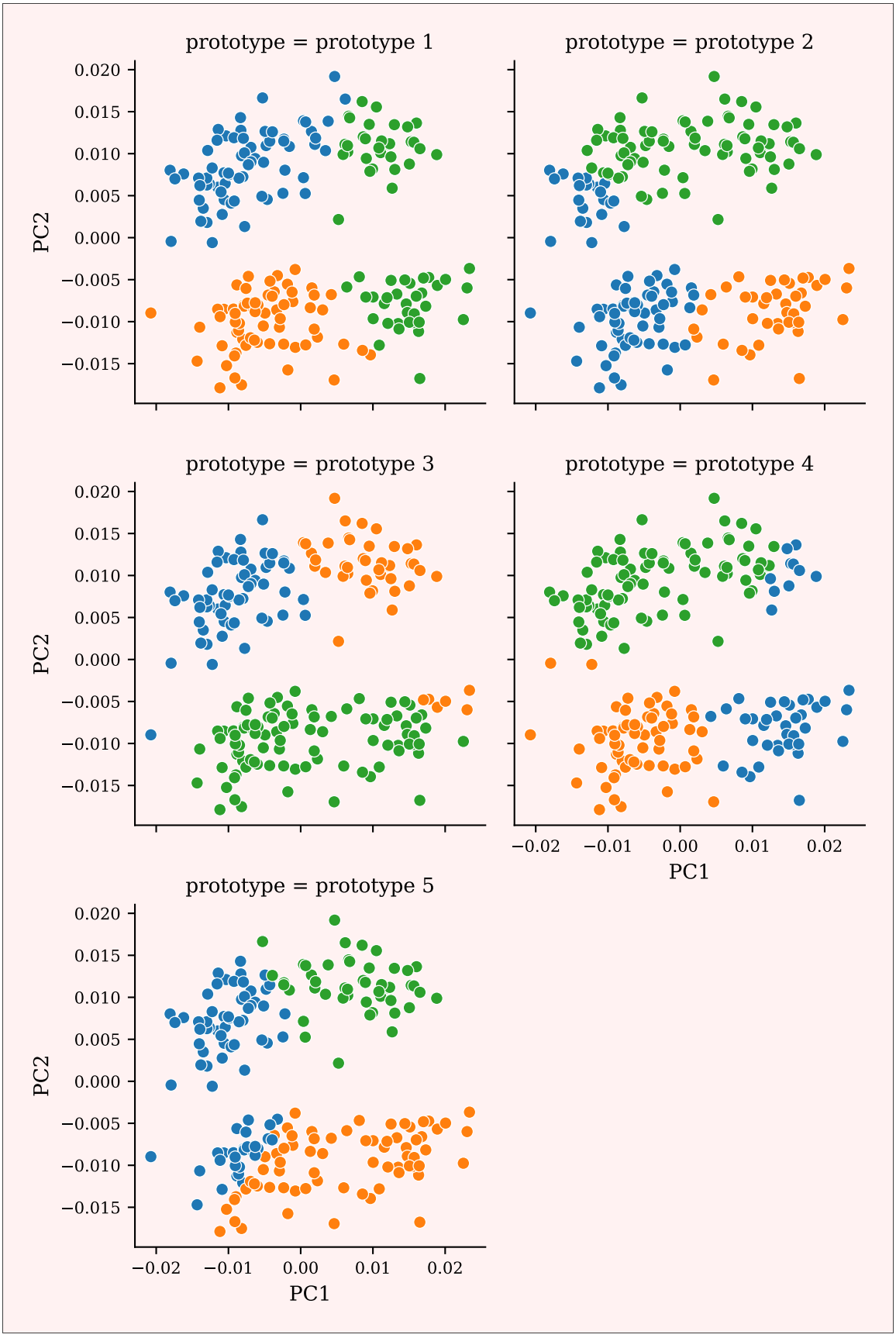
```
In [26]: # AFTD en 2 dimensions
         from sklearn.manifold import MDS
         aftd = MDS(n_components=2, dissimilarity='precomputed')
         dist = aftd.fit_transform(D)
         plt.scatter(*dist.T)
         plt.show()
```



14 En utilisant l'AFTD, isoler les classifications réellement différentes. Pour ce faire, on pourra utiliser à nouveau la méthode des *K-means*. Visualiser ces classifications.

```
In  [27]: # K-means sur la visualisation en 2D
          km = KMeans(n_clusters=5)
          km.fit(dist)

          # Choix d'un représentant au hasard par groupement.
          # `clustering_prototypes` est une matrice de taille `5*200`.

Out [27]: KMeans(n_clusters=5)
```

In [28]:
```
_, idxs_uniq = np.unique(km.labels_, return_index=True)
clustering_prototypes = clusterings[idxs_uniq, :]

# Tableau individu-variable 200*5 avec les étiquettes de chacun des
# 200 individus pour chacun des 5 groupements.
df_clustering_prototypes = pd.DataFrame(clustering_prototypes.T,
↪  columns=[f"prototype {i}" for i in range(1, 6)])

# Rajout des dimensions des crabes
df = pd.concat((crabsquant, df_clustering_prototypes), axis=1)

# Linéarisation pour représentation avec Seaborn
df2 = pd.melt(df, id_vars=["FL", "RW", "CL", "CW", "BD"],
↪  var_name="prototype", value_name="class")

# Représentation avec des facettes
g = sns.FacetGrid(df2, col="prototype", col_wrap=2)
g.map_dataframe(scatterplot_pca, columns=["FL", "RW", "CL", "CW",
↪  "BD"], hue="class")
plt.show()
```


prototype = prototype 1
prototype = prototype 2
prototype = prototype 3
prototype = prototype 4
prototype = prototype 5
PC2
PC1
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.02
−0.01
0.00
0.01
0.02

1.3 *K-means* adaptatifs

1.3.1 Introduction

L'algorithme des *K-means* avec distance adaptative consiste à utiliser l'algorithme des *K-means* en utilisant la distance de Mahalanobis à la place de la distance euclidienne.

Définition 1 (Distance de Mahalanobis)**.** *La distance entre deux points $\boldsymbol{x}$ et $\boldsymbol{y}$, au sens d'une métrique induite par une matrice M, est définie par*

$$d_M^2(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} - \boldsymbol{y})^T M (\boldsymbol{x} - \boldsymbol{y}). \tag{1}$$

La distance de Mahalanobis entre un point $\boldsymbol{x}$ et le centre μ_k de la classe ω_k est la distance de Mahalanobis entre ces points, au sens de la métrique induite par l'inverse $\tilde{V}_k^{-1}$ de la matrice de covariance empirique normalisée de la classe :

$$\begin{aligned} d_{V_k^{-1}}^2 &= (\boldsymbol{x} - \mu_k)^T \, V_k^{-1} \, (\boldsymbol{x} - \mu_k)\,, \\ \tilde{V}_k &= (\rho_k \, \det V_k)^{-1/p} \, V_k, \\ V_k &= \frac{1}{n_k} \sum_{i : \boldsymbol{x}_i \in P_k} (\boldsymbol{x}_i - \mu_k) \, (\boldsymbol{x}_i - \mu_k)^T. \end{aligned}$$

L'utilisation de cette métrique permet de tenir compte de la dispersion des points, qui peut varier selon la classe considérée. Ce surcroît de flexibilité a un prix : cela nécessite d'estimer davantage de paramètres (à chaque itération, les matrices de covariance empirique des classes, en plus des centres de gravité). Cette variante est donc plus sensible au manque de données que l'algorithme des *K-means* classique.

L'objectif de ce TP est d'implémenter puis d'étudier les propriétés (avantages, inconvénients) de l'utilisation d'une telle métrique en classification.

1.3.2 Programmation

On cherche à programmer la méthode des *K-means* avec distance adaptative, décrit dans l'algorithme 1. On prendra en compte les éléments suivants.

Arguments d'entrée La classe à développer accepte comme arguments d'entrée le nombre de groupements à apprendre `n_clusters`, le nombre d'essais `n_init`, le nombre d'itérations maximal `max_iter` et une précision `tol`.

L'argument `n_init` représente le nombre de fois où l'algorithme sera appliqué au jeu de données X à partir de différentes initialisations, afin d'augmenter les chances de converger vers l'optimum global du critère optimisé. L'argument `max_iter` a pour but d'empêcher l'algorithme d'itérer indéfiniment à la recherche d'un optimum ; le dernier vise à déterminer quand l'algorithme a convergé.

Initialisation Les centres des classes μ_k peuvent être initialisés par K points tirés au hasard dans le jeu de données X. Pour cela, on peut s'appuyer sur la fonction `rng.choice`, qui permet de tirer au hasard de nombres entiers avec ou sans remise.

Chaque matrice de covariance normalisée $\tilde{V}_k$ peut être initialisée par $(\rho_k)^{-1/p} \, I_p$, où I_p est la matrice identité de dimension p, et ρ_k est le volume souhaité (imposé) pour la matrice $\tilde{V}_k^{-1}$ (voir ci-dessous). On peut prendre par défaut $\rho_k = 1$, pour tout $k = 1, \ldots, K$.

Mise à jour des paramètres L'algorithme des *K-means* avec distance adaptative répète les opérations suivantes, jusqu'à convergence :

1. calcul d'une nouvelle partition des données $P = (P_1, \ldots, P_K)$ en K groupes : chaque point est affecté à la classe dont le centre μ_k est le plus proche au sens de la distance de Mahalanobis

(calculée avec la matrice de covariance normalisée $\tilde{V}_k$). On peut pour cela s'appuyer sur les fonction `linalg.inv`, `np.argmin`, `cdist`.

Voir le fichier `src/adaptive_kmeans.py`

2. À partir de cette nouvelle partition, les centres des classes et les matrices de covariance (normalisées) sont mis à jour ; plus particulièrement :

$$\begin{aligned}
\mu_k &\longleftarrow \frac{1}{n_k} \sum_{i:\boldsymbol{x}_i \in P_k} \boldsymbol{x}_i ; \\
V_k &\longleftarrow \frac{1}{n_k} \sum_{i:\boldsymbol{x}_i \in P_k} (\boldsymbol{x}_i - \mu_k)(\boldsymbol{x}_i - \mu_k)^T , \\
\tilde{V}_k &\longleftarrow (\rho_k \det V_k)^{-1/p} V_k ,
\end{aligned}$$

où $n_k = \operatorname{card}\{i : \boldsymbol{x}_i \in P_k\}$.

Voir le fichier `src/adaptive_kmeans.py`

Pour diverses raisons (qui seront en partie élucidées par la suite), il faut normaliser les matrices de covariance empiriques V_k utilisées dans le calcul de la distance de Mahalanobis. À chaque étape de l'algorithme, la matrice $\tilde{V}_k$ normalisée est ainsi calculée à partir de V_k de telle sorte que $\det \tilde{V}_k^{-1} = \rho_k$.

Convergence À chaque itération de l'algorithme, la convergence peut être testée en calculant la distance δ entre les centres μ_k des classes calculés à l'itération présente et ceux $\mu_{k,\text{prec}}$ calculés à l'itération précédente :

$$\delta = \sum_{k=1}^{K} \|\mu_k - \mu_{k,\text{prec}}\|^2 .$$

On supposera que la convergence est atteinte dès lors que $\delta \leq \varepsilon$, où ε est une valeur de précision fixée par l'utilisateur.

Essais À chaque essai, une fois que l'algorithme a convergé vers un optimum local — c'est-à-dire une partition définie par K centres $\mu_k^\star$ et matrices $\tilde{V}_k^\star$, on dispose également de la valeur du critère correspondante, c'est-à-dire la somme des distances des points au centre de gravité de leur classe :

$$J^\star = J(\mu_k^\star, \tilde{V}_k^\star) = \sum_{k=1}^{K} \sum_{i:\boldsymbol{x}_i \in P_k} d^2_{(\tilde{V}_k^\star)^{-1}}(\boldsymbol{x}_i, \mu_k).$$

Si la valeur de ce critère est plus faible que celle $J(\mu_k^{\text{prec}}, \tilde{V}_k^{\text{prec}})$ obtenue après une exécution précédente de l'algorithme *pour les mêmes valeurs de* ρ_k, alors l'optimum local considéré est meilleur que l'optimum local précédent.

À chaque nouvelle convergence de l'algorithme, il convient alors de conserver en mémoire les paramètres correspondant à l'optimum local courant, s'ils sont meilleurs que ceux correspondant au meilleur optimum local obtenu précédemment.

Architecture `scikit-learn` La méthode est implémentée en utilisant les classes fournies par `scikit-learn`. Le plus souvent, il suffit d'hériter des bonnes classes et d'implémenter la méthode `fit` qui prend en argument le tableau de données utilisé pour l'apprentissage. Dans notre cas, il s'agit d'un algorithme de *clustering*, on hérite donc de la classe `ClusterMixin` en plus de la classe de base `BaseEstimator`. Pour être tout à fait complet, il faudrait encore implémenter les fonctions `get_params` et `set_params` pour que l'algorithme puisse être utilisé dans un `pipeline` ou pour pouvoir faire de la recherche de paramètres automatique.

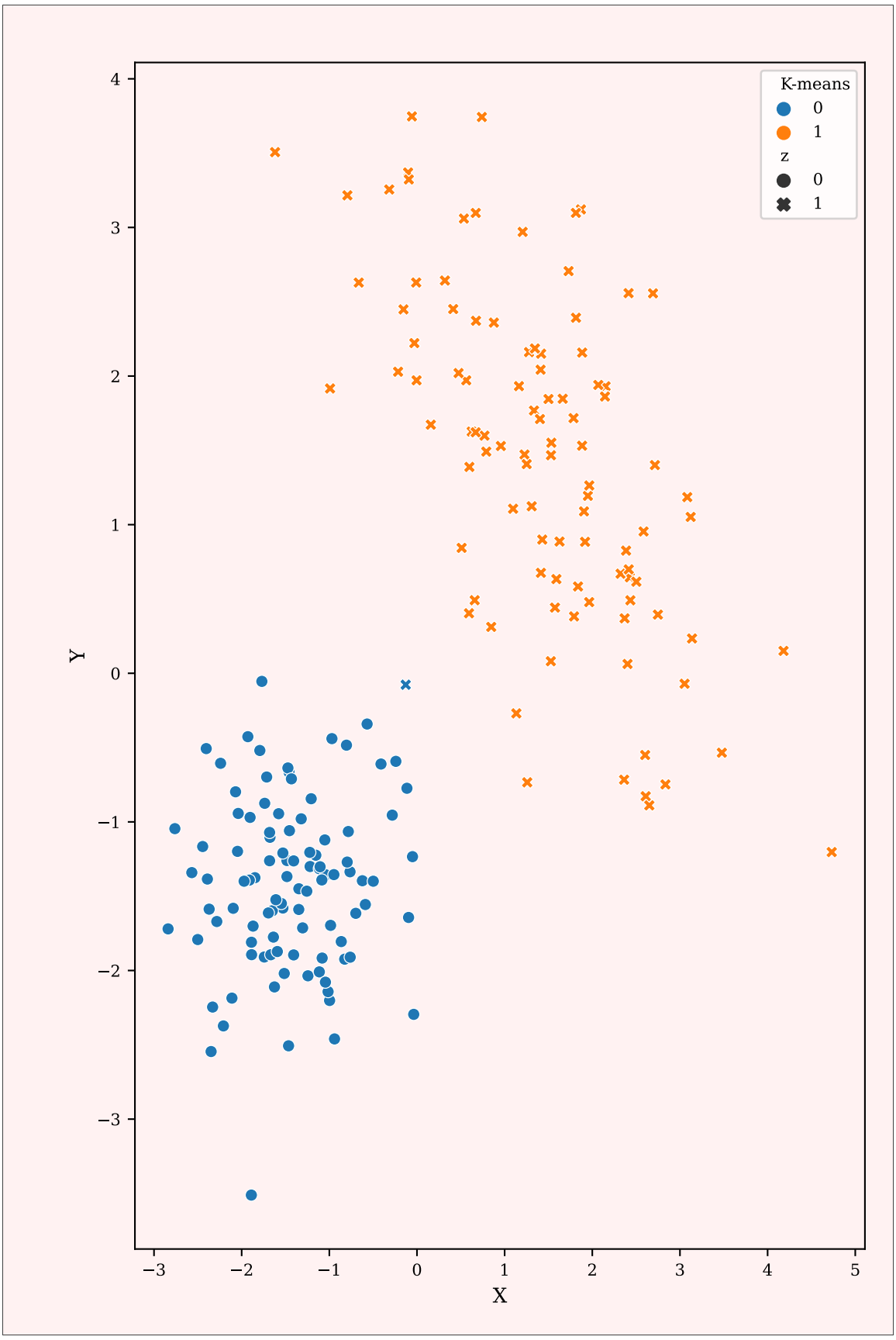
1.3.3 Applications

Données synthétiques On cherche dans un premier temps à tester l'algorithme développé sur diverses données synthétiques, de manière à appréhender son fonctionnement et ses limitations. On considère pour cela trois jeux de données `Synth1`, `Synth2` et `Synth3`. On pourra charger les données au moyen du code suivant :

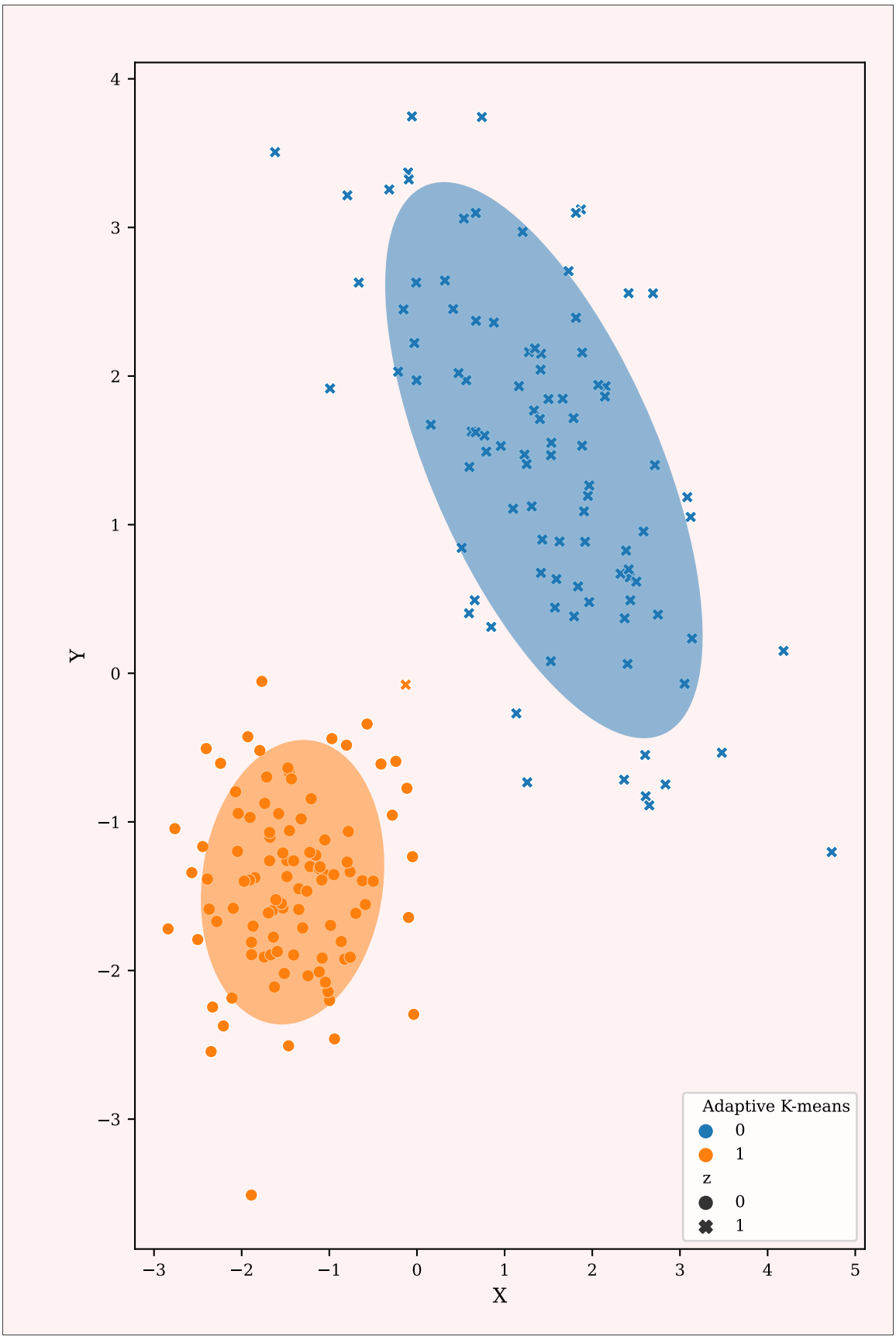
```
X = pd.read_csv("data/SynthN.csv")
z = X.z
X = X.drop(columns=["z"])
```

Pour chacun des jeux de données, effectuer une classification au moyen de l'algorithme des *K-means* classique, puis de l'algorithme des *K-means* avec distance adaptative. On comparera les résultats obtenus et on interprétera en fonction des données. On peut utiliser la fonction `adjusted_rand_score` disponible dans le module `sklearn.metrics` pour calculer l'adéquation de la partition trouvée à la partition réelle. Les arguments `centers` et `covars` de la fonction `plot_clustering` permettent d'afficher sous forme d'ellipse la matrice de covariance de chacun des groupements.

```
In  [29]: from sklearn.cluster import KMeans
          from sklearn.metrics import adjusted_rand_score

          X1 = pd.read_csv("data/Synth1.csv")
          z1 = X1.z
          X1 = X1.drop(columns=["z"])

          cls = KMeans(n_clusters=2)
          labels = cls.fit_predict(X1)
          s1 = adjusted_rand_score(z1, labels)

          labels = pd.Series(labels, name="K-means")
          plot_clustering(X1, clus1=labels, clus2=z1)

Out [29]: (<AxesSubplot:xlabel='X', ylabel='Y'>, None)

In  [30]: plt.show()
```


K-means
0
1
z
0
1
Y
X
4
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
4
5

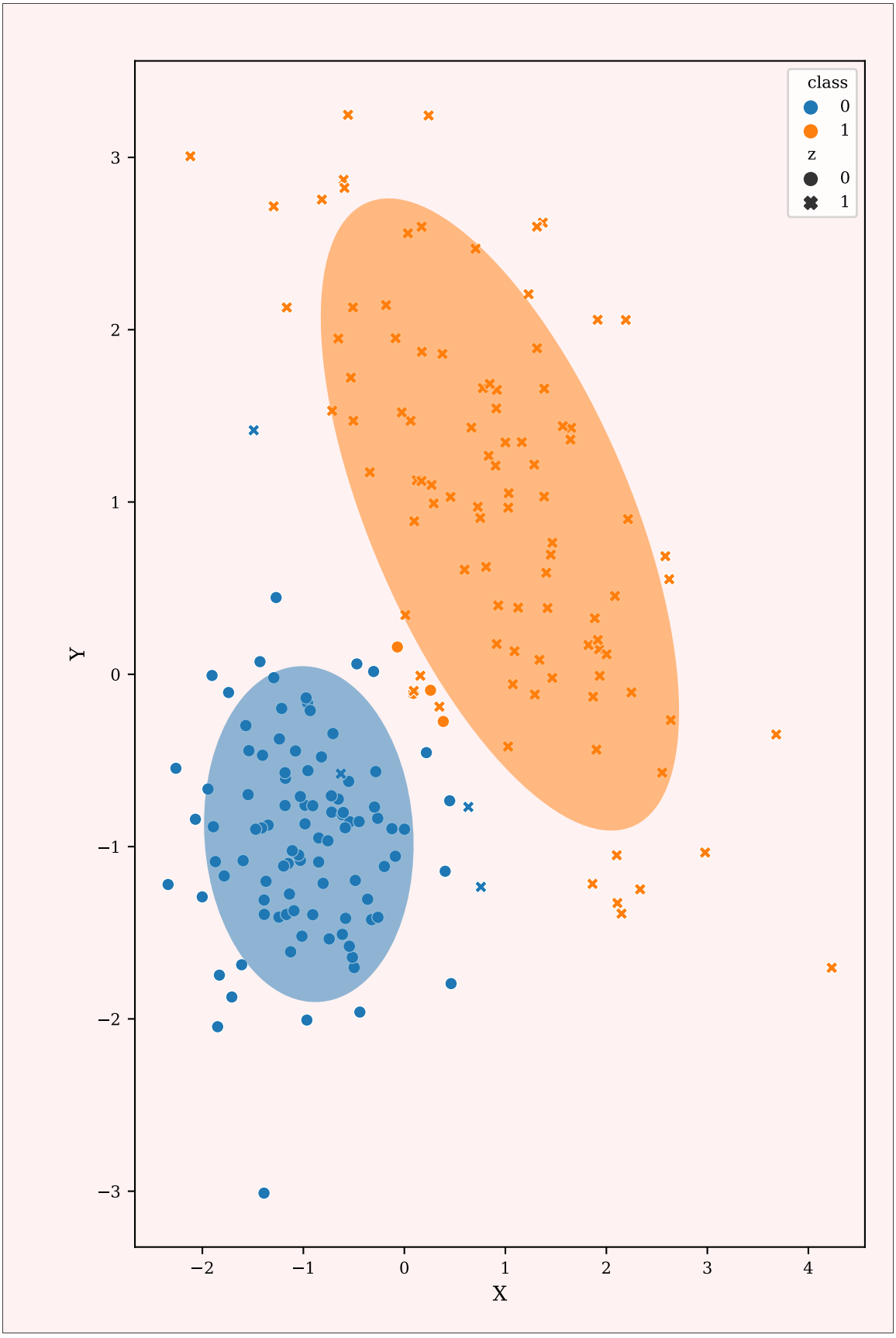
```
In  [31]: cls = AdaptiveKMeans(n_clusters=2)
          labels = cls.fit_predict(X1.to_numpy())
          s1a = adjusted_rand_score(z1, labels)

          labels = pd.Series(labels, name="Adaptive K-means")
          plot_clustering(X1, clus1=labels, clus2=z1,
           ↪  centers=cls.cluster_centers_, covars=cls.covars_)

Out [31]: (<AxesSubplot:xlabel='X', ylabel='Y'>, None)

In  [32]: plt.show()
```


Adaptive K-means
0
1
z
0
1
X
Y

In [33]:
```
X2 = pd.read_csv("data/Synth2.csv")
z2 = X2.z
X2 = X2.drop(columns=["z"])

cls = KMeans(n_clusters=2)
labels = cls.fit_predict(X2)
s2 = adjusted_rand_score(z2, labels)

cls = AdaptiveKMeans(n_clusters=2)
labels = cls.fit_predict(X2.to_numpy())
s2a = adjusted_rand_score(z2, labels)

plot_clustering(X2, labels, z2, centers=cls.cluster_centers_,
 ↪  covars=cls.covars_)
```

Out [33]:
```
(<AxesSubplot:xlabel='X', ylabel='Y'>, None)
```

In [34]:
```
plt.show()
```


class
0
1
z
0
1
Y
X
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4

In [35]:
```
X3 = pd.read_csv("data/Synth3.csv")
z3 = X3.z
X3 = X3.drop(columns=["z"])

cls = KMeans(n_clusters=2)
labels = cls.fit_predict(X3.to_numpy())
s3 = adjusted_rand_score(z3, labels)

cls = AdaptiveKMeans(n_clusters=2)
labels = cls.fit_predict(X3.to_numpy())
s3a = adjusted_rand_score(z3, labels)

plot_clustering(X3, labels, z3, centers=cls.cluster_centers_,
↪  covars=cls.covars_)
```

Out [35]:
```
(<AxesSubplot:xlabel='X', ylabel='Y'>, None)
```

In [36]:
```
plt.show()
```

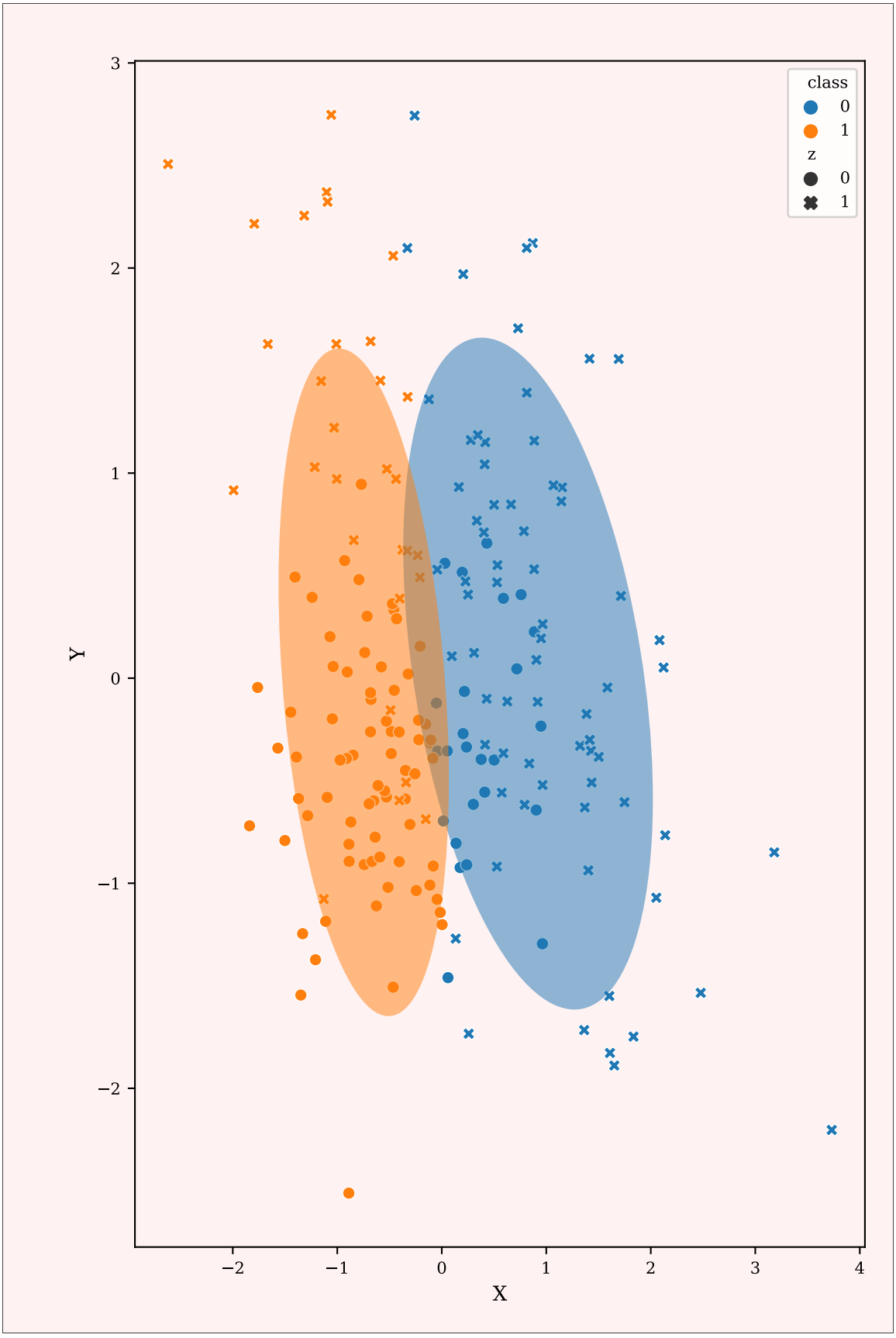

class
0
1
z
0
1
3
2
1
0
−1
−2
Y
−2
−1
0
1
2
3
4
X

Table 1 – ARI sur les trois jeux de données synthétiques `Synth1`, `Synth2` et `Synth3`.

	`Synth1`	`Synth2`	`Synth3`
ARI K-means	0.980	0.846	0.133
ARI K-means adaptatif	0.980	0.846	0.198

Iris Déterminer une classification des données `Iris` avec la méthode des *K-means* classique, puis avec les *K-means* adaptatifs, pour $K = 2, 3, 4, 5$. Calculer la valeur du critère optimisé (inertie dans le premier cas, distance totale dans le second) pour $K = 1$. Afficher les valeurs de critère en fonction de K. Quel semble être le meilleur nombre de classes dans chacun des cas ?

Effectuer une partition des données pour $K = 2$, puis pour $K = 3$. Comparer les résultats obtenus par les deux méthodes, et interpréter.

Il est à présent nécessaire de régulariser les matrices de covariance intra-classes lors des itérations. On obtient (`nstart` $= 25$ toujours) les valeurs d'inertie consignées dans le tableau 2. Les partitions en $K = 2$ et $K = 3$ groupes sont représentées dans la figure **??**.

Table 2 – Critères sur les Iris.

# groupements	1	2	3	4	5
K-means	681.371	152.348	78.851	57.282	46.446
K-means adaptatif	124.641	60.593	48.656	37.649	32.643

Crabs Déterminer une classification des données `Crabs` avec la méthode des *K-means* classique, puis avec les *K-means* adaptatifs, pour $K = 2, 3, \ldots, 8$. Calculer la valeur du critère optimisé (inertie dans le premier cas, distance totale dans le second) pour $K = 1$. Afficher les valeurs de critère en fonction de K. Conclure quant au nombre de classes qui semble le plus raisonnable.

Effectuer une partition des données pour $K = 2$, puis pour $K = 4$. Comparer les résultats obtenus par les deux méthodes ; interpréter.

Classification pixellaire On souhaite effectuer la classification pixellaire [1] d'une image, que l'on pourra charger et afficher au moyen du code suivant :

```
img_toucan = plt.imread("data/toucan.png")
plt.imshow(img_toucan)
plt.show()
```

On transformera l'image en un tableau individus-variables, chaque individu correspondant à un pixel et chaque variable à une composante couleur (normalisée entre 0 et 1) :

```
mat_toucan = img_toucan.reshape((-1, 3))
```

On effectuera une classification pixellaire avec les *K-means* et les *K-means* adaptatifs, pour un nombre de clusters variant entre $K = 2$ et $K = 5$. On prendra soin de comparer les résultats donnés par les deux algorithmes.

Une fois la classification obtenue, on pourra remplacer chaque pixel par le centre de sa classe et reconstituer ainsi une image simplifiée comme suit (si l'on suppose que les résultats des *K-means* et des *K-means* adaptatifs sont respectivement stockés dans les variables `km` et `akm`) :

```
img_quant = km.cluster_centers_[km.labels_, :].reshape(img_toucan.shape)
img_aquant = akm.cluster_centers_[akm.labels_, :].reshape(img_toucan.shape)
```

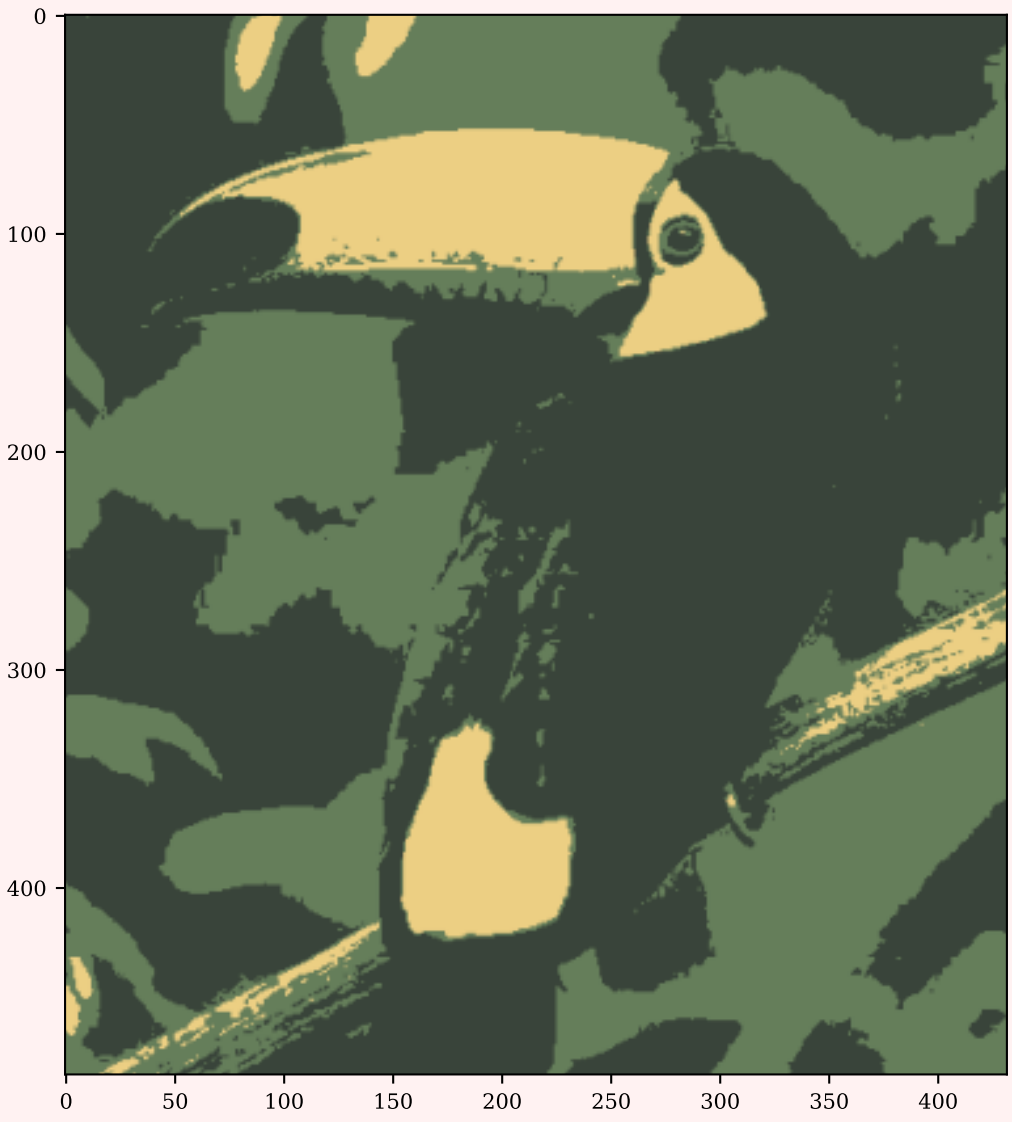
1. Contrairement à la segmentation, la classification pixellaire ne tient pas compte de l'information de voisinage des pixels.

On pourra ensuite afficher ces images comme précédemment.

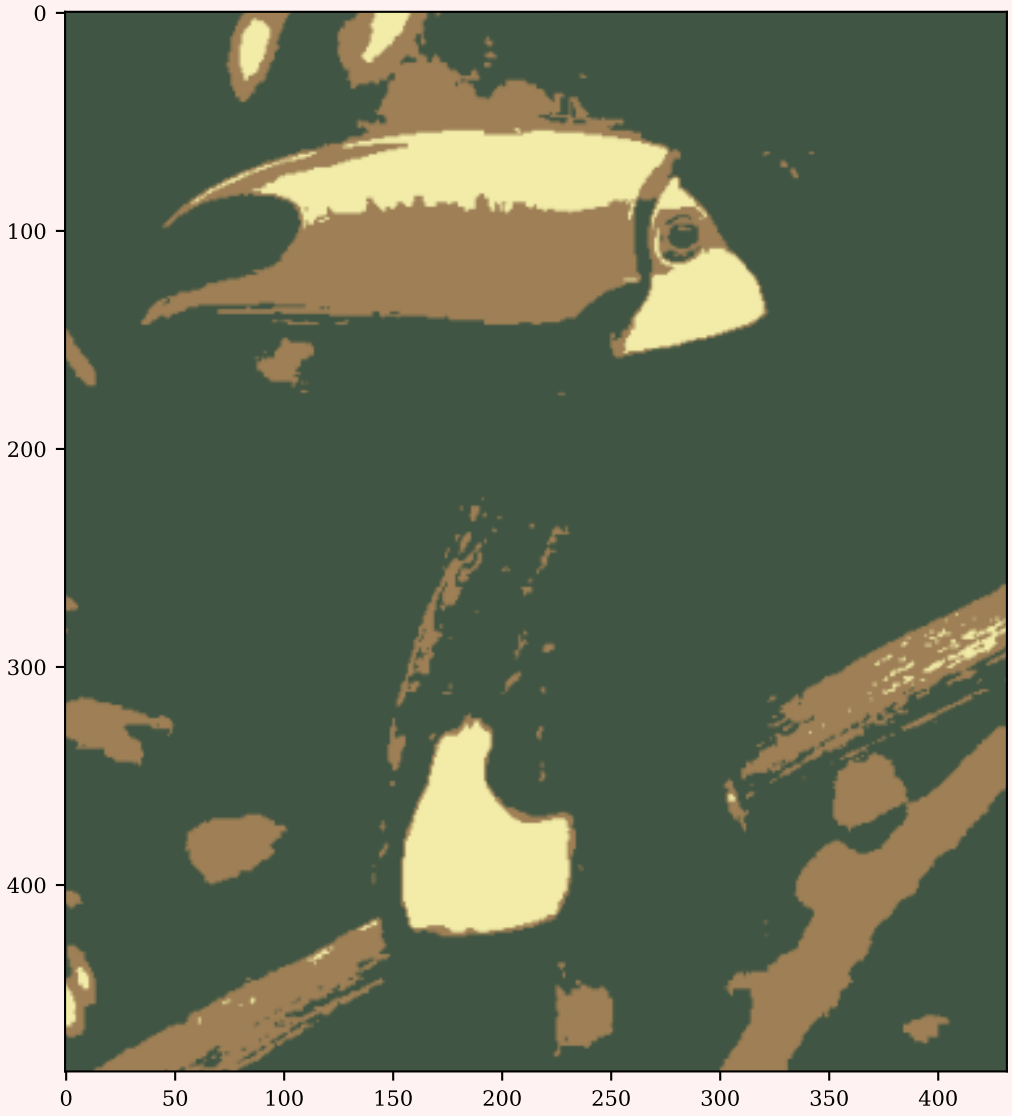
```
In  [37]: img_toucan = plt.imread("data/toucan.png")
          plt.imshow(img_toucan)

Out [37]: <matplotlib.image.AxesImage object at 0x7f5bee652290>

In  [38]: plt.show()
```


0
100
200
300
400
0
50
100
150
200
250
300
350
400

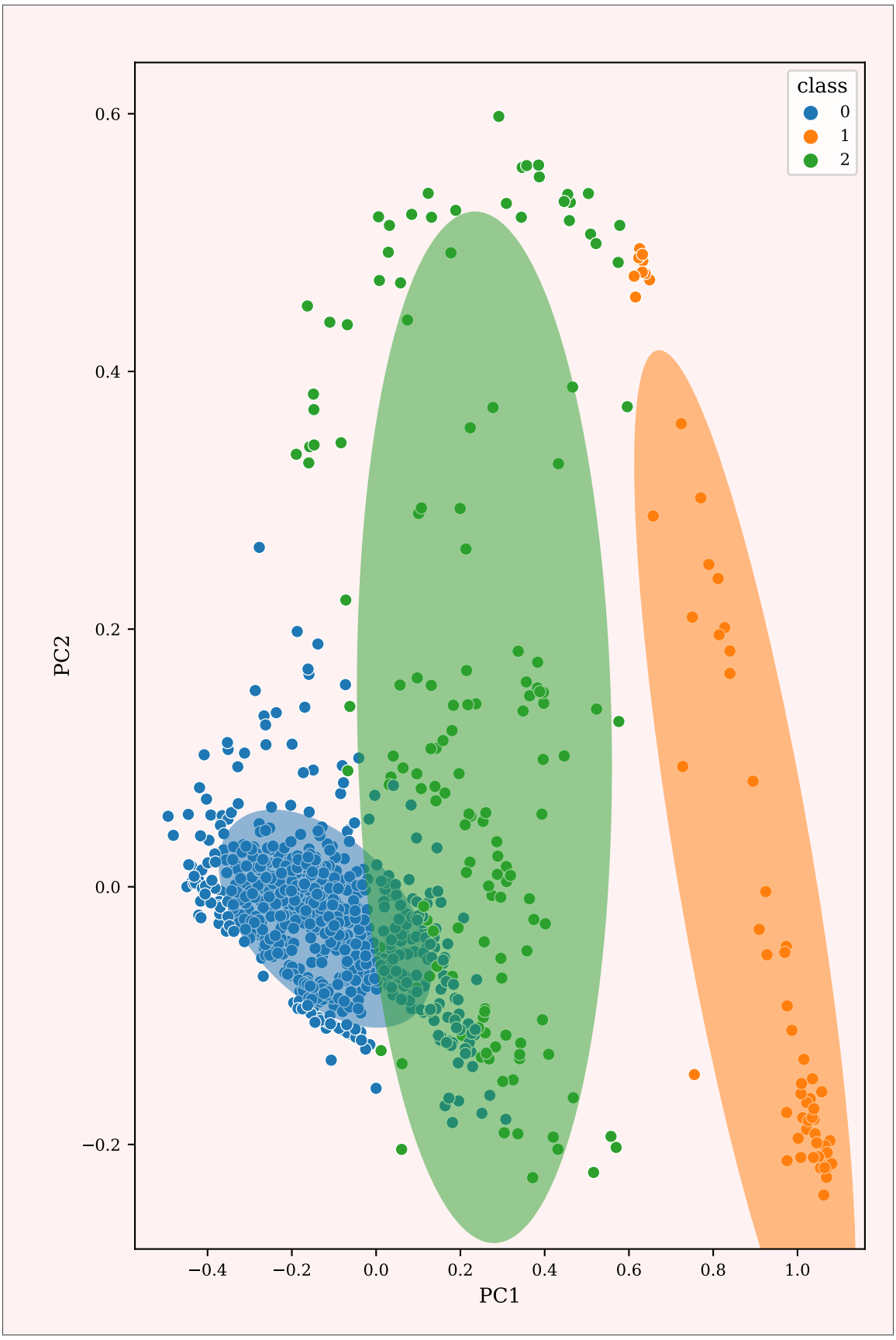
```
In  [39]: mat_toucan = img_toucan.reshape((-1, 3))

          km = KMeans(n_clusters=3)
          km.fit(mat_toucan)

Out [39]: KMeans(n_clusters=3)

In  [40]: img_quant = km.cluster_centers_[km.labels_,
          ↪  :].reshape(img_toucan.shape)
          plt.imshow(img_quant)

Out [40]: <matplotlib.image.AxesImage object at 0x7f5bee71e830>

In  [41]: plt.show()
```


0
100
200
300
400
0
50
100
150
200
250
300
350
400

```
In  [42]: kma = AdaptiveKMeans(n_clusters=3)
          kma.fit(mat_toucan)
          img_quant = kma.cluster_centers_[kma.labels_,
          ↪  :].reshape(img_toucan.shape)
          plt.imshow(img_quant)

Out [42]: <matplotlib.image.AxesImage object at 0x7f5bee71ea70>

In  [43]: plt.show()

          # Sous-échantillonnage de 1000 pixels
```


0
100
200
300
400
0
50
100
150
200
250
300
350
400

In [44]:
```
rng = np.random.default_rng()
idxs = rng.choice(range(mat_toucan.shape[0]), size=1000, replace=False)
mat_toucan_sub = pd.DataFrame(mat_toucan[idxs, :])


plot_clustering(mat_toucan_sub, kma.labels_[idxs],
 ↪  centers=kma.cluster_centers_, covars=kma.covars_)
plt.show()
```


class
0
1
2
PC2
PC1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
1.0

Données : Tableau de données X (matrice $n \times p$), nb. de classes K (entier), volumes des classes ρ_k (vecteur de longueur K), nb. max. d'itérations $n_{\text{iter.max}}$ (entier), nb. d'essais n_{start} (entier), tolérance ε pour vérifier la convergence (réel positif)

Résultat : valeur du critère, nb. d'itérations, partition des données, centres des classes, matrices de covariance empiriques normalisées des classes

pour $i = 1, 2, \ldots, n_{\text{start}}$ **faire**

initialiser μ_k et $\tilde{V}_k$, pour $k = 1, \ldots, K$

tant que la convergence n'est pas atteinte et $n_{\text{iter.max}}$ n'est pas dépassé **faire**

mettre à jour la partition des données : affecter chacun des points de X au centre μ_k le plus proche, au sens de la distance de Mahalanobis

mémoriser les anciens centres des classes : $\mu_{k,\text{prec}} \longleftarrow \mu_k$

mettre à jour les centres des classes μ_k et les matrices de covariance empiriques $\tilde{V}_k$ normalisées au moyen de la partition calculée précédemment

mettre à jour le critère de distance d_{tot} (somme des distances aux centres des classes)

mettre à jour le critère de convergence

fin

si $J^\star < J^{\text{opt}}$ (avec J^{opt} le meilleur optimum local donné par les essais précédents) **alors**

mettre à jour le meilleur optimum local : mémoriser la valeur du critère $J^\star$, le nombre d'itérations, la partition des données, les centres et les matrices de covariance

fin

fin

Algorithme 1 : Algorithme des *K-means* avec distance adaptative

1.4 Itérations en folie

Soit $n \geq 2$. On définit le jeu de données unidimensionnel consistant en $2n$ nombres $y_1 < y_2 < \cdots < y_n < x_n < x_{n-1} < \cdots < x_1$ tels que $y_i = -x_i$ pour $i = 1, \ldots, n$. On suppose également que $x_{i+1} = \beta_i x_i$ avec

$$\beta_i = \frac{n-i+1}{i(2n-i)}\left(1 + \frac{(i-1)(2n-i+1)}{n-i+2}\right),$$

pour $i = 1, \ldots, n-1$. On pourra prendre $x_1 = n$.

15 Définir une fonction qui construit ce jeu de données pour un n donné.

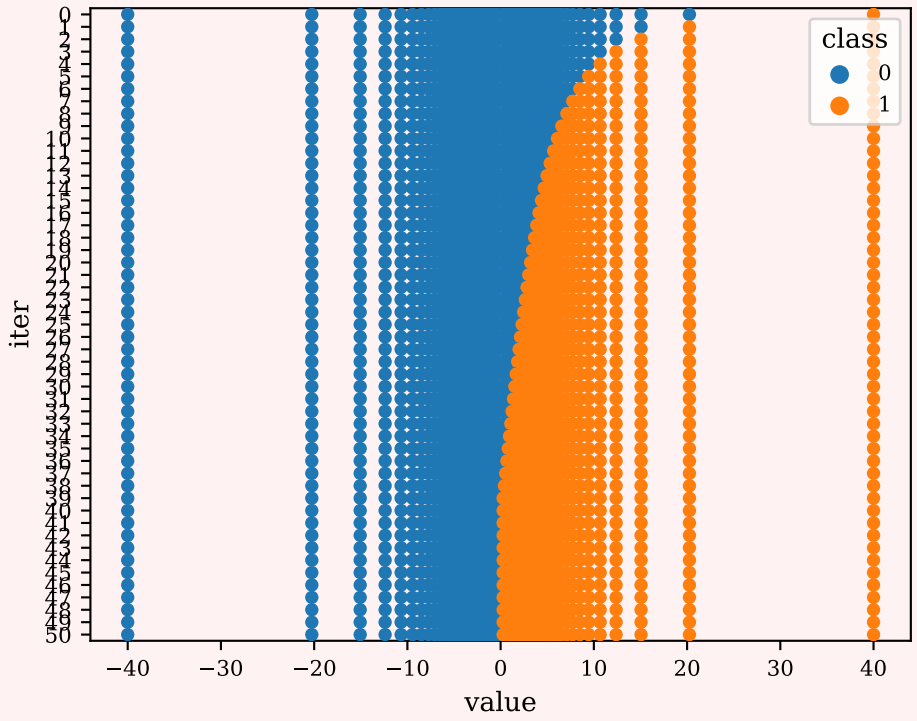
Plutôt que d'introduire des boucles `for`, on pourra utiliser les fonctions `np.cumprod` et `np.arange`.

```
In [45]: def km_iter(n):
             def betas(n):
                 i = np.arange(1, n)
                 u = (n - i + 1) / (i * (2 * n - i))
                 v = (i - 1) * (2 * n - i + 1) / (n - i + 2)
                 return u * (1 + v)

             x1 = n
             x = np.concatenate((np.array([x1]), x1 * np.cumprod(betas(n))))
             z = np.concatenate((-x, x[::-1]))

             return z
```

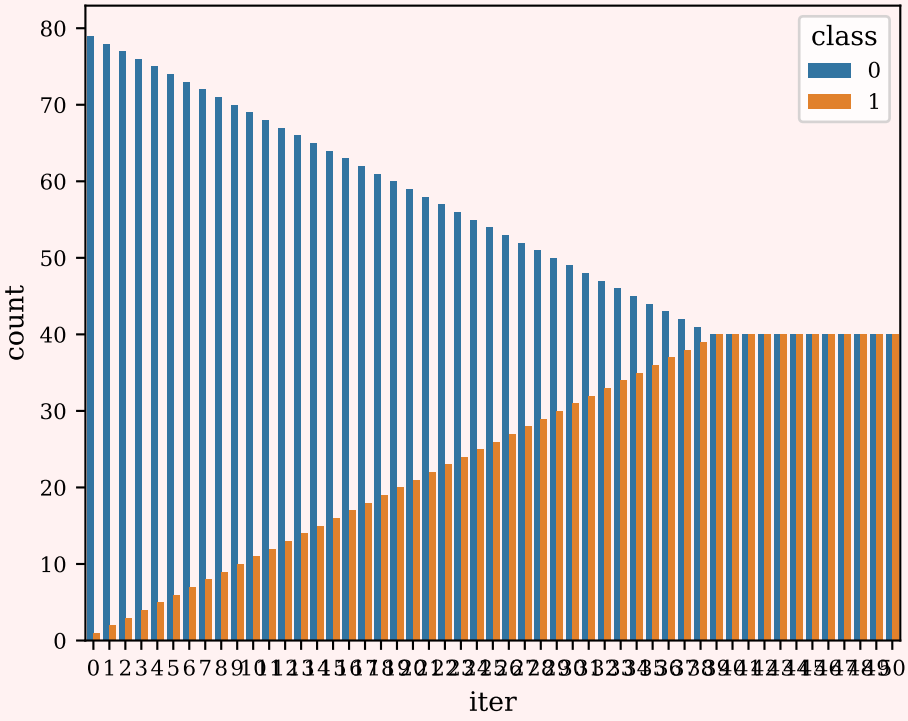
16 Appliquer la méthode des *K-means* sur ce jeu de données avec pour centres de départ x_1 et x_2. Visualiser l'évolution des groupements en fonction du nombre d'itérations.

On pourra faire varier `max_iter` en fixant `n_init` à 1.

In [46]:
```
from sklearn.cluster import KMeans
n = 40

def data_gen(X, max_iter):
    a = np.zeros(X.shape[0], dtype="int32")
    a[-1] = 1
    yield pd.Series(a)              # Le classement avant toute itération
    for i in range(1, max_iter+1):
        cls = KMeans(n_clusters=2, init=X[-2:, :], max_iter=i,
        ↪  n_init=1)
        cls.fit(X)
        yield pd.Series(cls.labels_, name=i)


X = km_iter(n)[:, None]
df = pd.concat(data_gen(X, 50), axis=1)
df = pd.concat((pd.Series(X.ravel(), name="value"), df), axis=1)
df = pd.melt(df, id_vars="value", var_name="iter", value_name="class")
df.iter = df.iter.astype("int32")

sns.stripplot(x="value", y="iter", hue="class", data=df, orient="h",
↪  jitter=False)
plt.show()
```



In [47]:
```
sns.countplot(x="iter", hue="class", data=df)
plt.show()
```



Ce jeu de données est construit pour être très lent à converger. Alors que classiquement, l'algorithme des *K-means* stationne en quelques dizaines d'itérations au plus, ce jeu données à $2n$ points converge en $n-1$ itérations.

2 Partie théorique

2.1 Convergence des *K-means*

On cherche ici à montrer que l'algorithme des K-means peut être interprété comme une procédure de minimisation alternée qui répète deux étapes :

- le calcul d'un représentant pour chaque groupe,
- le calcul d'une affectation des points à chacun des groupes.

17 On définit une affectation comme un ensemble de variables indicatrices $z_{ik} \in \{0,1\}$, pour tout $i = 1,\dots,n$ et $k = 1,\dots,K$, telles que $\sum_k z_{ik} = 1$ (une seule est non nulle). Exprimer le critère d'inertie optimisé par l'algorithme des K-means en fonction de ces variables d'affectation $\boldsymbol{z}_i$ et des représentants des groupes $\boldsymbol{\mu}_k$.

Le critère d'inertie est

$$I(\boldsymbol{\mu}_1,\dots,\boldsymbol{\mu}_K,\boldsymbol{z}_1,\dots,\boldsymbol{z}_n) = \frac{1}{n}\sum_{k=1}^{K}\sum_{\boldsymbol{x}_i\in\omega_k} d^2(\boldsymbol{x}_i,\boldsymbol{\mu}_k) = \frac{1}{n}\sum_{k=1}^{K}\sum_{i=1}^{n} z_{ik}d^2(\boldsymbol{x}_i,\boldsymbol{\mu}_k),$$

où la distance utilisée est la distance euclidienne :

$$d^2(\boldsymbol{x}_1,\boldsymbol{x}_2) = \|\boldsymbol{x}_1-\boldsymbol{x}_2\|^2 = (\boldsymbol{x}_1-\boldsymbol{x}_2)^T(\boldsymbol{x}_1-\boldsymbol{x}_2).$$

18 On suppose disposer d'une affectation $\boldsymbol{z}_1,\dots,\boldsymbol{z}_n$. Montrer que pour un groupe, le centre de gravité est le représentant qui minimise le critère d'inertie.

Pour choisir le représentant $\boldsymbol{\mu}_k$ d'un groupe, on peut calculer la dérivée partielle du critère d'inertie par rapport à $\boldsymbol{\mu}_k$:

$$\frac{\partial}{\partial \boldsymbol{\mu}_k} I(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = -\frac{2}{n} \sum_{i=1}^{n} z_{ik}(\boldsymbol{x}_i - \boldsymbol{\mu}_k).$$

L'annulation de ce vecteur de dérivées premières donne :

$$\frac{\partial}{\partial \boldsymbol{\mu}_k} I(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = 0 \Leftrightarrow \sum_{i=1}^{n} z_{ik}(\boldsymbol{x}_i - \boldsymbol{\mu}_k) = 0 \Leftrightarrow \boldsymbol{\mu}_k = \frac{\sum_{i=1}^{n} z_{ik} \boldsymbol{x}_i}{\sum_{i=1}^{n} z_{ik}}.$$

Remarquons que la matrice des dérivées secondes (matrice hessienne) est définie par blocs :

$$\frac{\partial^2 I(\boldsymbol{\mu}_k, \boldsymbol{z}_i)}{\partial \boldsymbol{\mu}_k \partial \boldsymbol{\mu}_k^T} = \frac{2}{n} \sum_{i=1}^{n} z_{ik} \mathrm{I_p}, \text{ et } \frac{\partial^2 I(\boldsymbol{\mu}_k, \boldsymbol{z}_i)}{\partial \boldsymbol{\mu}_k \partial \boldsymbol{\mu}_\ell^T} = 0_\mathrm{p} \text{ pour tout } \ell \neq k,$$

avec $\mathrm{I_p}$ la matrices identité, et 0_p la matrice nulle, de dimensions $p \times p$; elle est donc diagonale et définie positive si les classes ne sont pas vides (auquel cas $\sum_i z_{ik} > 0$).

19 Étant donné des centres $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$, montrer que l'affectation de chaque point au centre le plus proche minimise le critère d'inertie.

Remarquons tout d'abord que le critère d'inertie est séparable — il s'écrit comme une somme de termes dont chacun ne concerne qu'un exemple $\boldsymbol{x}_i$:

$$I(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) = \frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} z_{ik} d^2(\boldsymbol{x}_i, \boldsymbol{\mu}_k).$$

Pour chaque $\boldsymbol{x}_i$, il est alors évident qu'il faut choisir l'affectation $\boldsymbol{z}_i^\star$ définie par

$$z_{ik}^\star = \begin{cases} 1 & \text{pour } k^\star = \arg \min_{\ell=1,\ldots,K} d^2(\boldsymbol{x}_i, \boldsymbol{\mu}_\ell), \\ 0 & \text{pour } \ell \neq k, \end{cases}$$

car $d^2(\boldsymbol{x}_i, \boldsymbol{\mu}_\ell) \geq d^2(\boldsymbol{x}_i, \boldsymbol{\mu}_{k^\star})$ pour tout $\ell \neq k^\star$, et donc pour tout $\boldsymbol{z}_i \neq \boldsymbol{z}_i^\star$,

$$\sum_{k=1}^{K} z_{ik} d^2(\boldsymbol{x}_i, \boldsymbol{\mu}_k) \geq \sum_{k=1}^{K} z_{ik}^\star d^2(\boldsymbol{x}_i, \boldsymbol{\mu}_k).$$

2.2 Convergence des *K-means* adaptatifs

L'objectif de cette partie est de justifier l'algorithme d'un point de vue théorique.

Plus particulièrement, supposons que l'on dispose d'une partition des données $P = (P_1, \ldots, P_K)$, où la classe de chaque individu est codée par une variable binaire

$$z_{ik} = \begin{cases} 1 & \text{si } \boldsymbol{x}_i \in P_k, \\ 0 & \text{sinon.} \end{cases}$$

On cherche à caractériser chaque classe par un prototype $\boldsymbol{v}_k$ et une métrique définie par une matrice M_k. On souhaite montrer que la minimisation du critère de distance

$$J\left(\{v_k, M_k\}_{k=1,\ldots,K}\right) = \sum_{k=1}^{K} \sum_{i=1}^{n} z_{ik} d_{ik}^2 \tag{2}$$

sous la contrainte $\det M_k = \rho_k$ pour tout $k = 1, \ldots, K$, où la distance $d_{ik}^2 = d_{M_k}^2(\boldsymbol{x}_i, \boldsymbol{v}_k)$ est définie

par l'équation (1), donne

$$\boldsymbol{v}_k = \overline{\boldsymbol{x}}_k = \frac{1}{n_k}\sum_{i=1}^{n} z_{ik}\boldsymbol{x}_i, \tag{3}$$

$$M_k^{-1} = (\rho_k \det V_k)^{-1/p} V_k \text{ , avec } V_k = \frac{1}{n_k}\sum_{i=1}^{n} z_{ik}\left(\boldsymbol{x}_i - \boldsymbol{v}_k\right)\left(\boldsymbol{x}_i - \boldsymbol{v}_k\right)^T. \tag{4}$$

À chaque itération, les prototypes des classes sont donc obtenus en calculant les centres de gravité, et les métriques sont définies par les (inverses des) matrices de covariance normalisées.

Remarque 1 (Volume des classes)**.** *Fixer* $\det M_k = \rho_k$ *pour tout* $k = 1, \ldots, K$ *revient à fixer le volume de chaque classe : si l'on n'impose pas cette contrainte, il peut arriver qu'une ou plusieurs classes « absorbent » tous les points au détriment d'une ou plusieurs autres (ce qui cause des problèmes numériques, la matrice de covariance d'une classe d'effectif faible pouvant être de rang incomplet, et donc singulière).*

On admettra que pour minimiser le critère défini par (2)-(1) par rapport à $\boldsymbol{v}_k$ et M_k sous la contrainte $\det M_k = \rho_k$, on peut calculer le lagrangien

$$\mathcal{L}\left(\{v_k, M_k, \lambda_k\}_{k=1,\ldots,K}\right) = J\left(\{v_k, M_k\}_{k=1,\ldots,K}\right) - \sum_{k=1}^{K}\lambda_k(\det M_k - \rho_k), \tag{5}$$

où λ_k est le multiplicateur de Lagrange de la k^{e} contrainte (c'est-à-dire $\det M_k = \rho_k$), et de le minimiser par rapport à λ_k, $\boldsymbol{v}_k$ et M_k : pour cela, il faut vérifier conjointement

$$\frac{\partial\mathcal{L}\left(\{\boldsymbol{v}_k, M_k, \lambda_k\}\right)}{\partial\lambda_k} = 0, \quad \frac{\partial\mathcal{L}\left(\{\boldsymbol{v}_k, M_k, \lambda_k\}\right)}{\partial M_k} = 0, \quad \frac{\partial\mathcal{L}\left(\{\boldsymbol{v}_k, M_k, \lambda_k\}\right)}{\partial\boldsymbol{v}_k} = 0.$$

On définira la dérivée $\partial f(M)/\partial M$ d'une fonction $f(M)$ par rapport à une matrice M de terme général m_{ij} comme la matrice B de terme général $b_{ij} = \partial f(M)/\partial m_{ij}$.

Il faudrait de surcroît montrer que la matrice des dérivées secondes (matrice hessienne) est définie positive pour les valeurs de paramètres qui annulent le vecteur des dérivées premières (vecteur gradient). On admettra ici ce résultat.

Pour le calcul des dérivées, on pourra s'aider du « Matrix Cookbook », disponible à l'URL : `http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf`.

20 Calculer $\partial\mathcal{L}\left(\{\boldsymbol{v}_k, M_k, \lambda_k\}\right)/\partial\boldsymbol{v}_k$; montrer que la mise à jour des prototypes revient à calculer les centres de gravité des classes définis par l'équation (3).

La dérivée par rapport à $\boldsymbol{v}_k$ est aisément obtenue :

$$\begin{aligned}\frac{\partial\mathcal{L}\left(\{\boldsymbol{v}_k, M_k, \lambda_k\}\right)}{\partial\boldsymbol{v}_k} &= \sum_{i=1}^{n} z_{ik}\frac{\partial d_{ik}^2}{\partial\boldsymbol{v}_k},\\ &= -2M_k^{-1}\sum_{i=1}^{n} z_{ik}(\boldsymbol{x}_i - \boldsymbol{v}_k);\end{aligned}$$

l'annulation de ce vecteur de dérivées premières donne

$$\frac{\partial\mathcal{L}\left(\{\boldsymbol{v}_k, M_k, \lambda_k\}\right)}{\partial\boldsymbol{v}_k} = 0 \Leftrightarrow \sum_{i=1}^{n} z_{ik}(\boldsymbol{x}_i - \boldsymbol{v}_k) = 0 \Leftrightarrow \boldsymbol{v}_k = \frac{1}{n_k}\sum_{i=1}^{n} z_{ik}\boldsymbol{x}_i = \overline{\boldsymbol{x}_k},$$

où $n_k = \sum_{i=1}^{n} z_{ik}$.

21 Calculer $\partial\mathcal{L}\left(\{v_k, M_k, \lambda_k\}\right)/\partial M_k$, puis $\partial\mathcal{L}\left(\{v_k, M_k, \lambda_k\}\right)/\partial\lambda_k$. Montrer que la mise à jour des matrices M_k définissant la métrique spécifique à chaque classe revient à calculer les inverses des matrices de covariance empiriques normalisées définies par l'équation (4) (plus difficile !).

Commençons par annuler la dérivée par rapport à λ_k :

$$\frac{\partial \mathcal{L}\left(\{\boldsymbol{v}_k, M_k, \lambda_k\}\right)}{\partial \lambda_k} = 0 \Leftrightarrow \det M_k - \rho_k = 0 \Leftrightarrow \det M_k = \rho_k.$$

Le calcul de la matrice des dérivées secondes par rapport aux éléments de M_k donne

$$\begin{aligned}\frac{\partial \mathcal{L}\left(\{\boldsymbol{v}_k, M_k, \lambda_k\}\right)}{\partial M_k} &= \sum_{i=1}^{n} z_{ik} \frac{\partial d_{ik}^2}{\partial M_k} - \lambda_k \frac{\partial \det M_k}{\partial M_k} \\ &= \sum_{i=1}^{n} z_{ik} (\boldsymbol{x}_i - \boldsymbol{v}_k)(\boldsymbol{x}_i - \boldsymbol{v}_k)^T - \lambda_k \det M_k (M_k^{-1})^T.\end{aligned}$$

L'annulation conjointe des dérivées premières par rapport à λ_k et M_k donne

$$\begin{aligned}\left\{\frac{\partial \mathcal{L}}{\partial \lambda_k} = 0, \frac{\partial \mathcal{L}}{\partial M_k} = 0\right\} &\Leftrightarrow \sum_{i=1}^{n} z_{ik} (\boldsymbol{x}_i - \boldsymbol{v}_k)(\boldsymbol{x}_i - \boldsymbol{v}_k)^T = \lambda_k \rho_k (M_k^{-1})^T \\ &\Leftrightarrow M_k^{-1} = \lambda_k^{-1} \rho_k^{-1} \sum_{i=1}^{n} z_{ik} (\boldsymbol{x}_i - \boldsymbol{v}_k)(\boldsymbol{x}_i - \boldsymbol{v}_k)^T \\ &\Leftrightarrow M_k^{-1} = \lambda_k^{-1} \rho_k^{-1} n_k V_k.\end{aligned}$$

Il reste à déterminer la valeur du multiplicateur de Lagrange λ_k ; on calcule pour cela $\det M_k$:

$$\begin{aligned}\det M_k &= \det\left((\lambda_k^{-1} \rho_k^{-1} n_k V_k)^{-1}\right) \\ &= \det\left(\lambda_k \rho_k n_k^{-1} V_k^{-1}\right) \\ &= \lambda_k^p \rho_k^p n_k^{-p} \det\left(V_k^{-1}\right); \\ \Leftrightarrow \quad \lambda_k &= \left(\rho_k^{-p} \left(\det\left(V_k^{-1}\right)\right)^{-1} \det M_k n_k^p\right)^{1/p} \\ &= \left(\rho_k^{1-p} n_k^p \det V_k\right)^{1/p} \\ &= \rho_k^{1/p-1} n_k \left(\det V_k\right)^{1/p},\end{aligned}$$

ce qui, utilisé dans l'expression précédente de M_k^{-1}, donne

$$\begin{aligned}\left\{\frac{\partial \mathcal{L}}{\partial \lambda_k} = 0, \frac{\partial \mathcal{L}}{\partial M_k} = 0\right\} \Leftrightarrow \; M_k^{-1} &= \rho_k^{1-1/p} n_k^{-1} \left(\det V_k\right)^{-1/p} \rho_k^{-1} n_k V_k \\ &= \rho_k^{-1/p} \left(\det V_k\right)^{-1/p} V_k.\end{aligned}$$