

# SY09 Printemps 2020

## TP 06 — *K-means*

### 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 :

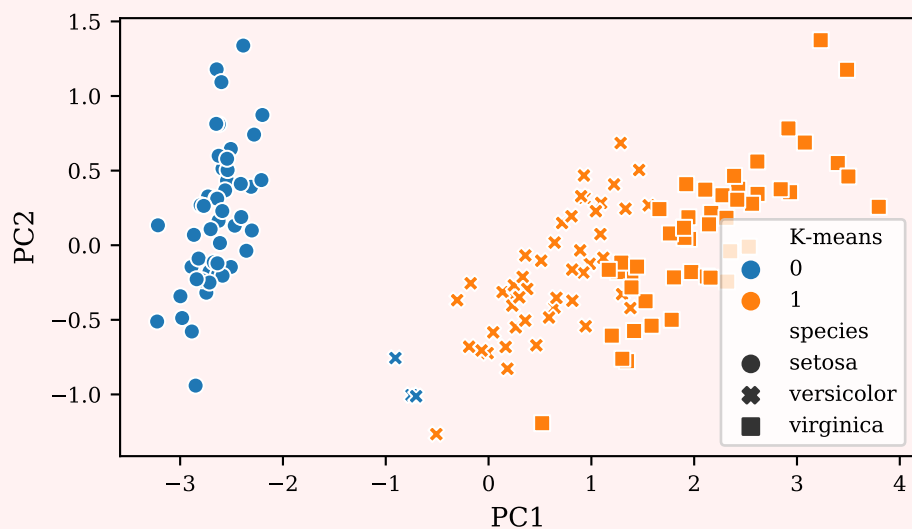
```
scatterplot_pca(  
    columns=<column list>, hue=labels, style=labels2, data=<Pandas DataFrame>  
)
```

#### 1.1 Étude des *K-means* sur Iris

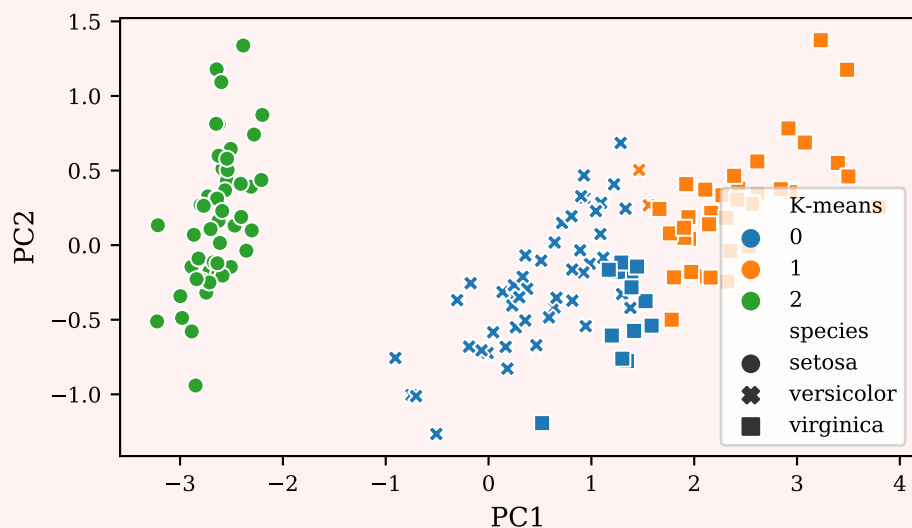
- ① Tenter une partition en  $K \in \{2, 3, 4\}$  classes. Visualiser et commenter.

```
In [1]: iris = sns.load_dataset("iris")

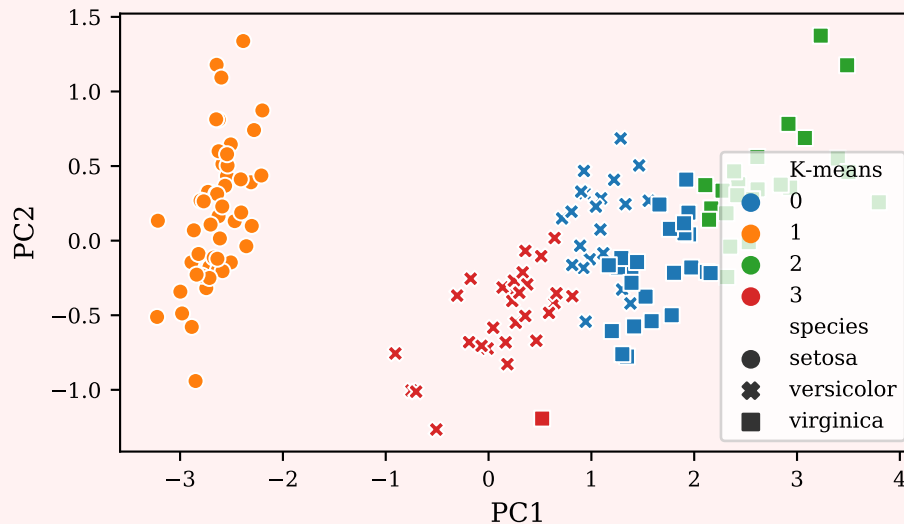
from sklearn.cluster import KMeans
iris0 = iris.drop(columns=["species"])
cls = KMeans(n_clusters=2, init="random")
cls.fit(iris0)
labels = pd.Series(cls.labels_, name="K-means")
scatterplot_pca(data=iris0, hue=labels, style=iris.species)
plt.show()
```



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



```
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 identifiée car elle est essulée. 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.

Afin d'étudier plus en détails la méthode des *K-means*, nous allons constituer un jeu de données portant sur les résultats successifs de cette méthode sur un même jeu de données et pour 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`
                    inertia = (
                        KMeans(
                            n_clusters=n_clusters,
                            n_init=1,
                            random_state=rs,
                            init=strategy,
                        )
                        .fit(dataset)
                        .inertia_
                    )
                    yield rs, strategy, n_clusters, inertia

gen = kmeans_dataset(iris0, [3, 5, 10], ["random", "k-means++"], 100)

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

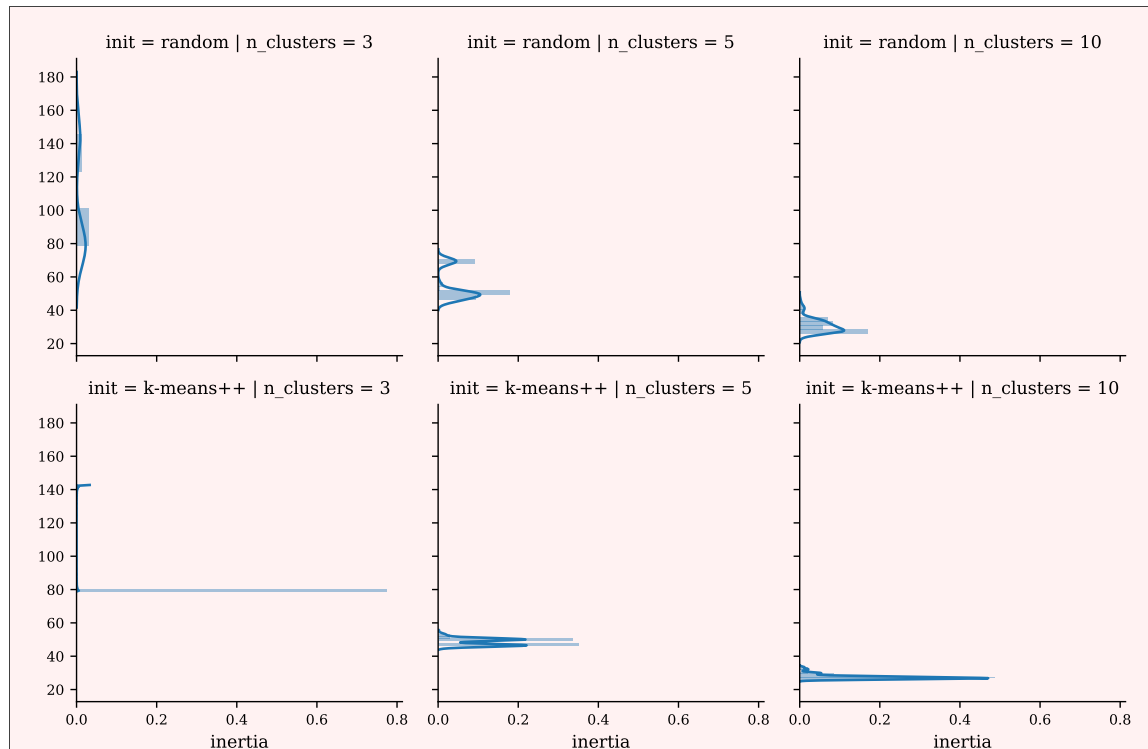
```

③ À l'aide de la fonction `sns.FacetGrid`, 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]: g = sns.FacetGrid(df, row="init", col="n_clusters")
        g.map(sns.distplot, "inertia", vertical=True)
        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.

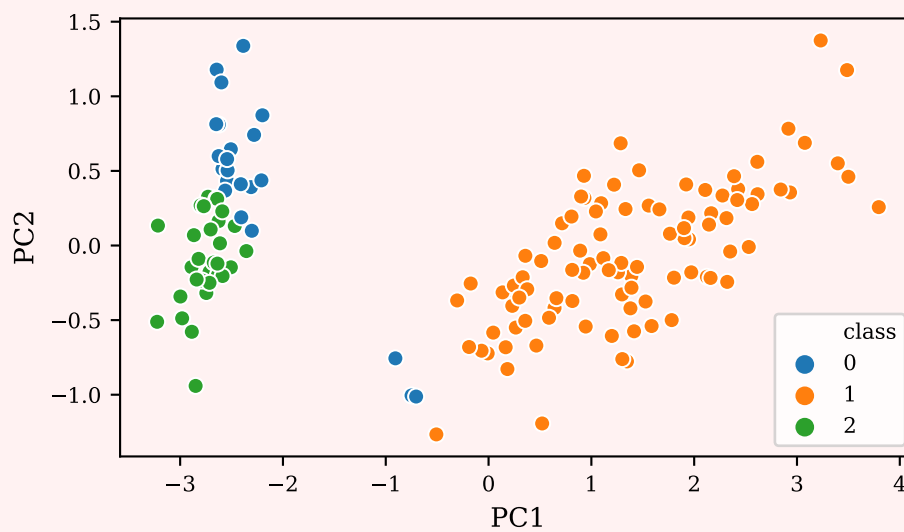
Cette dispersion, qui peut donner des mauvaises classifications en terme d'inertie, est largement réduite en utilisant une stratégie d'initialisation des centres plus performante (k-means++).

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

```
In [6]: # Les plus mauvaises classifications
        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 [6]: 145.7649378722375
In [7]: scatterplot_pca(data=iris0, hue=cls.labels_)
        plt.show()
```

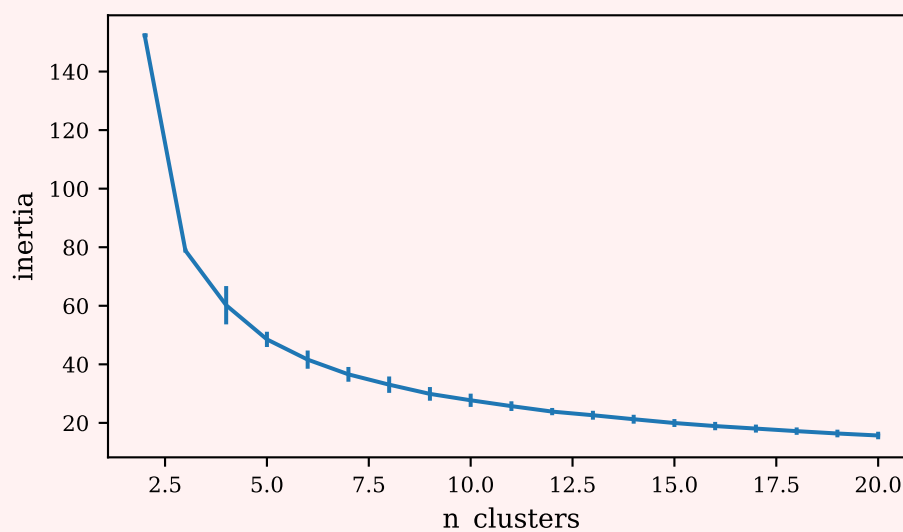


Un groupement naturel (à gauche) est coupé en deux d'où la mauvaise inertie. La solution est néanmoins stable (les affectations ne changent pas lors de la mise à jour des centres).

- 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 [8]: 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()
```

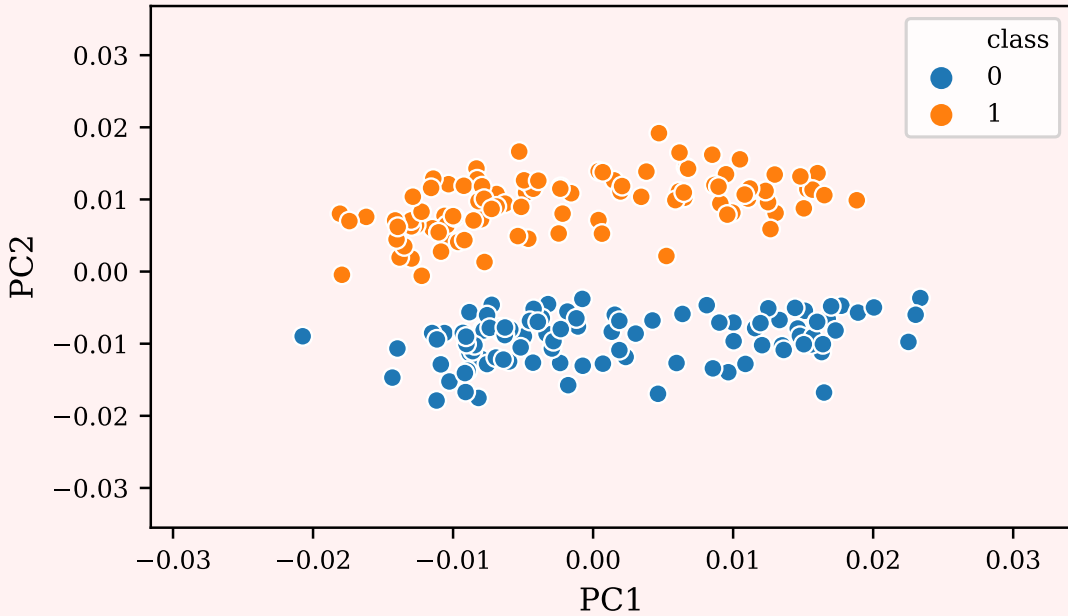


D'après la règle du coude un bon nombre de groupements est 3 voire 4.

## 1.2 Données Crabs

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

```
In [9]: 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()
```

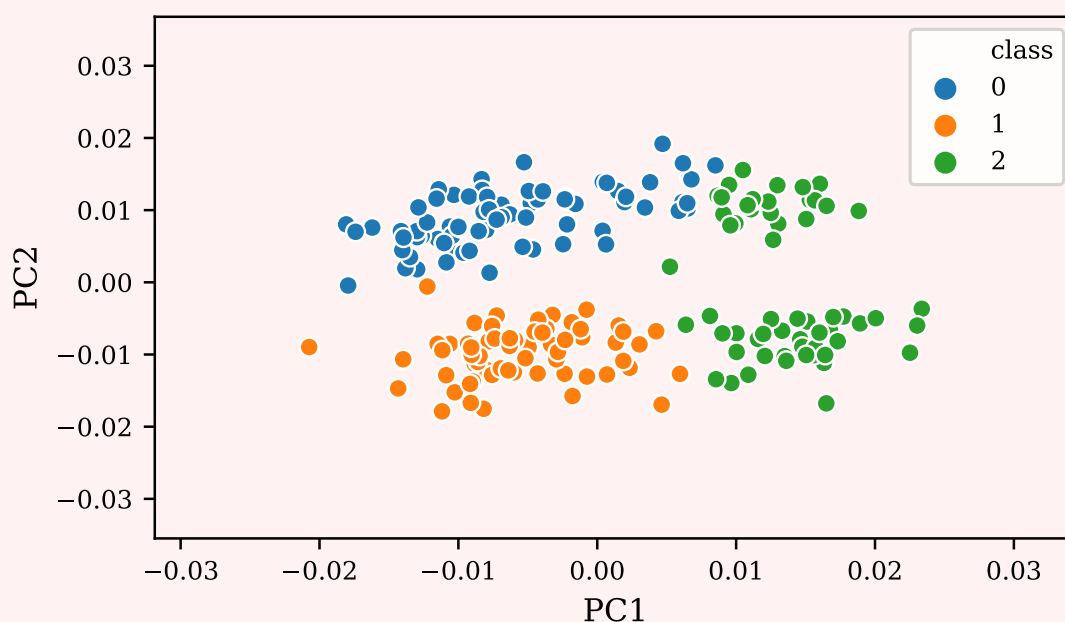


```
In [10]: inertias = [KMeans(n_clusters=2, init="random", n_init=1,
    → random_state=i).fit(crabsquant).inertia_ for i in range(1000)]

# Indices des classifications d'inerties différentes
inertias, idxs_uniq, idxs = np.unique(inertias, return_index=True,
    → return_inverse=True)
np.bincount(idxs)
Out [10]: array([444, 556])
In [11]: inertias
Out [11]: array([0.0276453 , 0.02826445])
```

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

```
In [12]: cls = KMeans(n_clusters=3).fit(crabsquant)
        scatterplot_pca(data=crabsquant, hue=cls.labels_)
        plt.show()
```



```
In [13]: 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)
np.bincount(idxs)
Out [13]: array([205, 45, 20, 2, 5, 119, 9, 3, 9, 3, 5, 8, 3,
    → 1, 2, 9, 12, 7, 95, 20, 23, 5, 9, 4, 49, 11, 168,
    → 1, 24, 4, 113, 6, 1])

In [14]: inertias
Out [14]: array([0.01734867, 0.01735593, 0.01736761, 0.01740171, 0.01740307,
    → 0.01740325, 0.01740441, 0.017407 , 0.01797298, 0.01800373,
    → 0.01801256, 0.01801487, 0.01801556, 0.01801751, 0.0180331 ,
    → 0.01803493, 0.01804046, 0.01805939, 0.01806565, 0.01825852,
    → 0.01826204, 0.01828601, 0.01830076, 0.0183031 , 0.01830477,
    → 0.01833014, 0.01833167, 0.01833763, 0.01905302, 0.01905309,
    → 0.01915905, 0.02009844, 0.02018774])
```

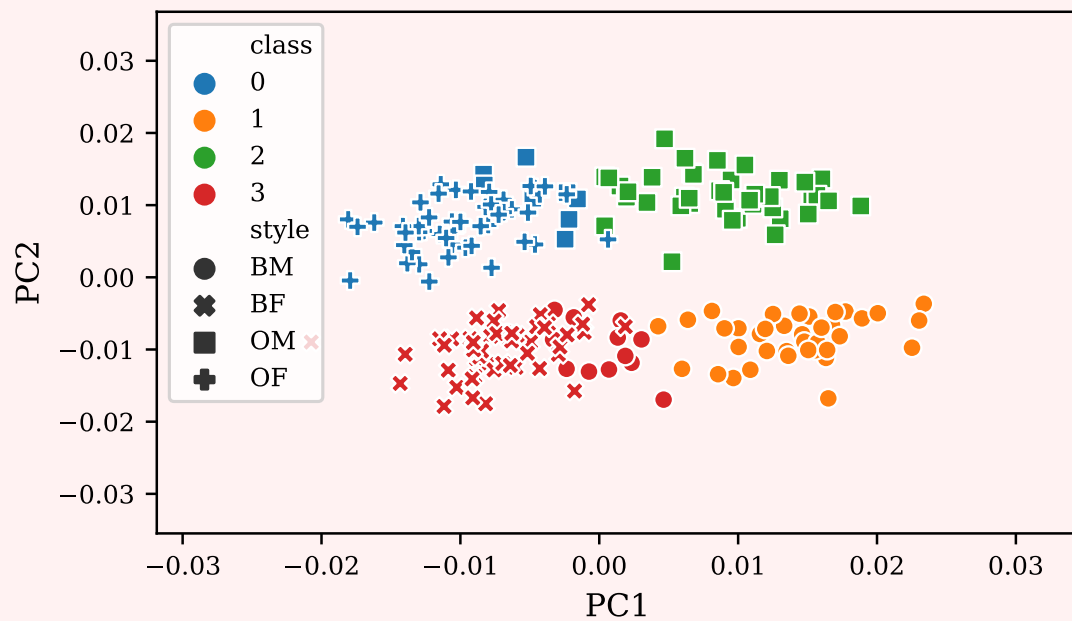
⑧ Effectuer une classification en  $K = 4$  classes des données. Comparer à la partition réelle suivant l'espèce et le sexe. Que peut-on conclure ?

On pourra utiliser l'indice de Rand ajusté avec l'instruction

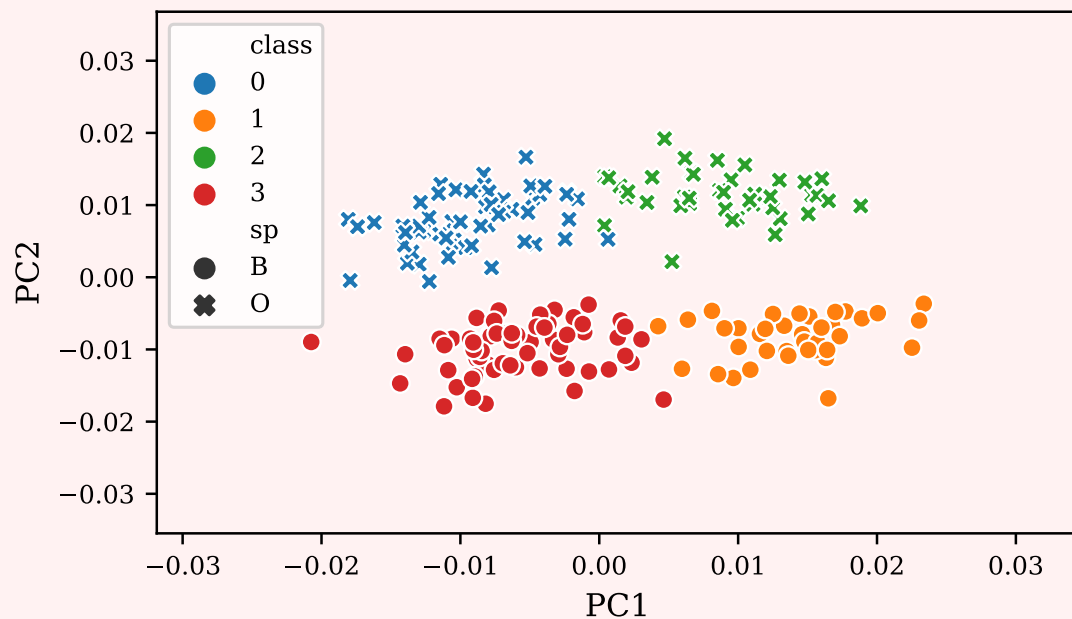
```
from sklearn.metrics import adjusted_rand_score
```

```
In [15]: from sklearn.metrics import adjusted_rand_score
    cls = KMeans(n_clusters=4).fit(crabsquant)
    adjusted_rand_score(crabs.sp+crabs.sex, cls.labels_)
Out [15]: 0.7512388410567427
In [16]: scatterplot_pca(data=crabsquant, hue=cls.labels_,
    → style=crabs.sp+crabs.sex)
plt.show()
```





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



## 2 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.

1. 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). Expri-

mer le critère d'inertie optimisé par l'algorithme des  $K$ -means en fonction de ces variables d'affectation  $z_i$  et des représentants des groupes  $\mu_k$ .

Le critère d'inertie est

$$I(\mu_1, \dots, \mu_K, z_1, \dots, z_n) = \frac{1}{n} \sum_{k=1}^K \sum_{x_i \in \omega_k} d^2(x_i, \mu_k) = \frac{1}{n} \sum_{k=1}^K \sum_{i=1}^n z_{ik} d^2(x_i, \mu_k),$$

où la distance utilisée est la distance euclidienne :

$$d^2(x_1, x_2) = \|x_1 - x_2\|^2 = (x_1 - x_2)^T (x_1 - x_2).$$

2. On suppose disposer d'une affectation  $z_1, \dots, z_n$ . Montrer que pour un groupe, le centre de gravité est le représentant qui minimise le critère d'inertie.

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

$$\frac{\partial}{\partial \mu_k} I(\mu_1, \dots, \mu_K, z_1, \dots, z_n) = -\frac{2}{n} \sum_{i=1}^n z_{ik} (x_i - \mu_k).$$

L'annulation de ce vecteur de dérivées premières donne :

$$\frac{\partial}{\partial \mu_k} I(\mu_1, \dots, \mu_K, z_1, \dots, z_n) = 0 \Leftrightarrow \sum_{i=1}^n z_{ik} (x_i - \mu_k) = 0 \Leftrightarrow \mu_k = \frac{\sum_{i=1}^n z_{ik} x_i}{\sum_{i=1}^n z_{ik}}.$$

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

$$\frac{\partial^2 I(\mu_k, z_i)}{\partial \mu_k \partial \mu_k^T} = \frac{2}{n} \sum_{i=1}^n z_{ik} I_p, \text{ et } \frac{\partial^2 I(\mu_k, z_i)}{\partial \mu_k \partial \mu_\ell^T} = 0_p \text{ pour tout } \ell \neq k,$$

avec  $I_p$  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  $\sum_i z_{ik} > 0$ ).

3. Étant donné des centres  $\mu_1, \dots, \mu_K$ , 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  $x_i$  :

$$I(\mu_1, \dots, \mu_K, z_1, \dots, z_n) = \frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K z_{ik} d^2(x_i, \mu_k).$$

Pour chaque  $x_i$ , il est alors évident qu'il faut choisir l'affectation  $z_i^*$  définie par

$$z_{ik}^* = \begin{cases} 1 & \text{pour } k^* = \arg \min_{\ell=1, \dots, K} d^2(x_i, \mu_\ell), \\ 0 & \text{pour } \ell \neq k, \end{cases}$$

car  $d^2(x_i, \mu_\ell) \geq d^2(x_i, \mu_{k^*})$  pour tout  $\ell \neq k^*$ , et donc pour tout  $z_i \neq z_i^*$ ,

$$\sum_{k=1}^K z_{ik} d^2(x_i, \mu_k) \geq \sum_{k=1}^K z_{ik}^* d^2(x_i, \mu_k).$$

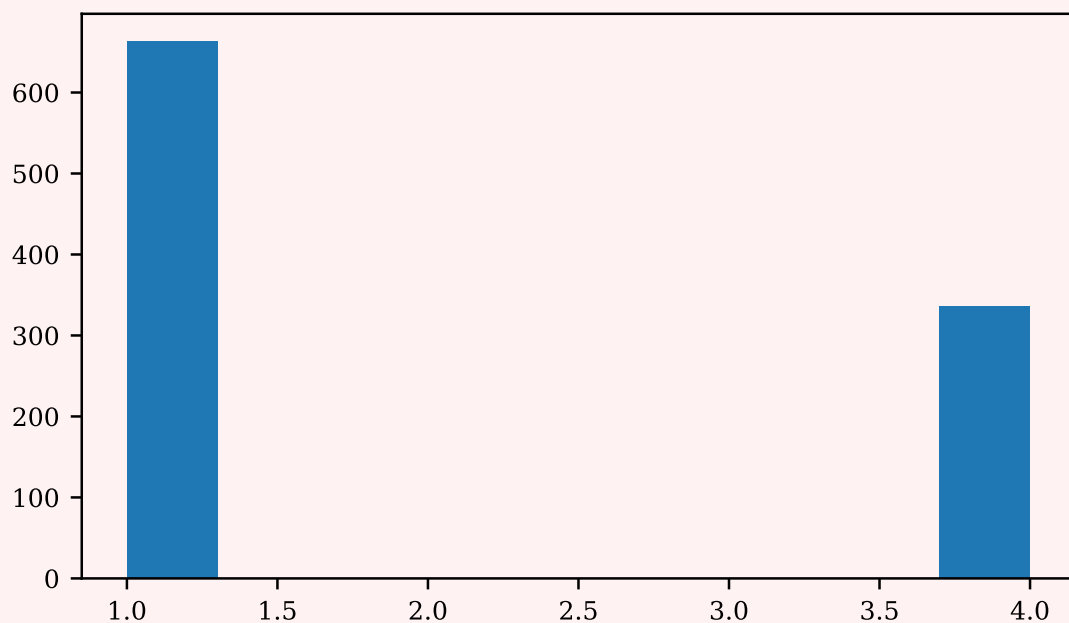


### 3 De l'intérêt d'une bonne initialisation

Soit un jeu de données de quatre points dans le plan formant un rectangle de grand côté  $a$  et de petit côté  $b$ . 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 [18]: 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()
```

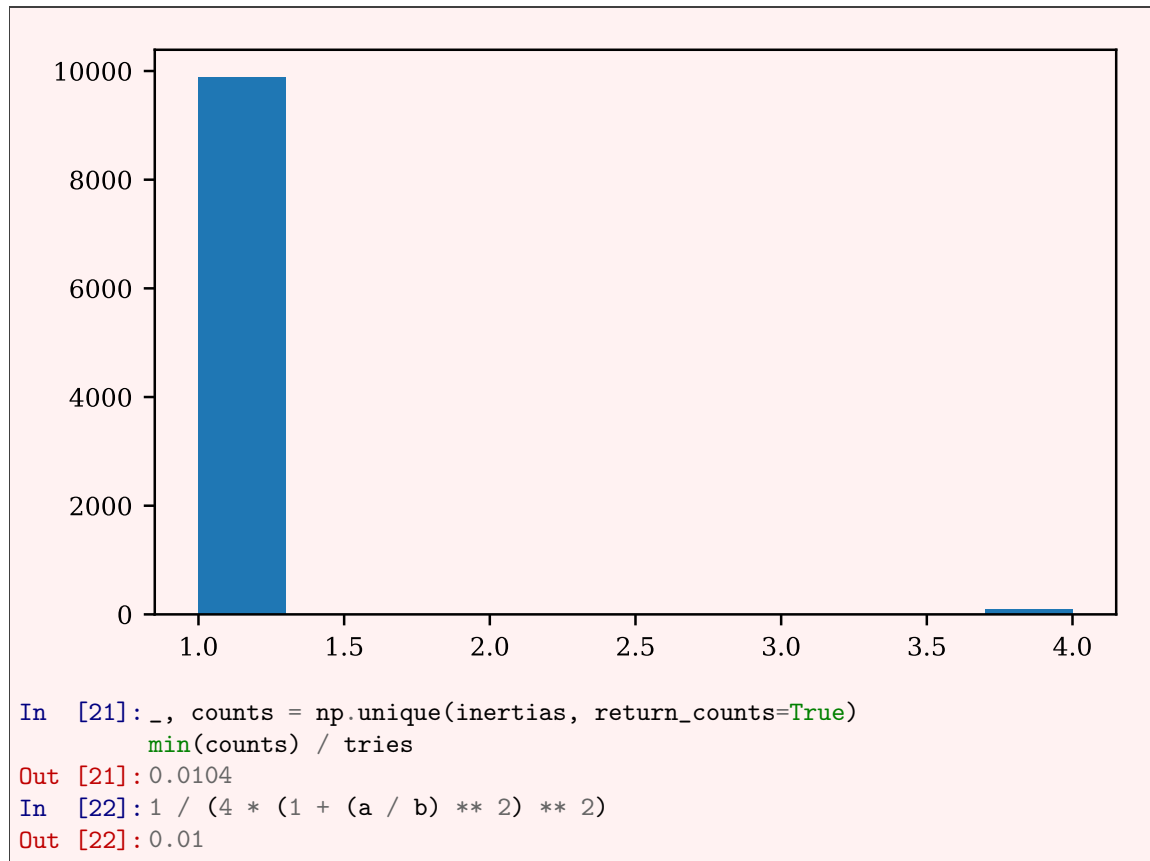


```
In [19]: _, counts = np.unique(inertias, return_counts=True)
         counts
Out [19]: array([664, 336])
```

- 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 [20]: 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()
```



## 4 *K-means* adaptatif

### 4.1 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  $\mathbf{x}$  et  $\mathbf{y}$ , au sens d'une métrique induite par une matrice  $M$ , est définie par

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

La distance de Mahalanobis entre un point  $\mathbf{x}$  et le centre  $\mu_k$  de la classe  $\omega_k$  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 :

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

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.

## 4.2 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  $\mu_k$  peuvent être initialisés par  $K$  points tirés au hasard dans le jeu de données  $X$ . Pour cela, on peut s'appuyer sur la fonction `np.random.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  $\rho_k$  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  $k = 1, \dots, K$ .

**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  $P = (P_1, \dots, P_K)$  en  $K$  groupes : chaque point est affecté à la classe dont le centre  $\mu_k$  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 fonctions `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 :

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

où  $n_k = \text{card} \{i : \mathbf{x}_i \in P_k\}$ .

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  $\delta$  entre les centres  $\mu_k$  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  $\varepsilon$  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  $\mu_k^*$  et matrices  $\tilde{V}_k^*$ , 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 :

$$J^* = J(\mu_k^*, \tilde{V}_k^*) = \sum_{k=1}^K \sum_{i: \mathbf{x}_i \in P_k} d_{(\tilde{V}_k^*)^{-1}}^2(\mathbf{x}_i, \mu_k).$$

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  $\rho_k$* , 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.

## 4.3 Applications

### 4.3.1 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"])
```

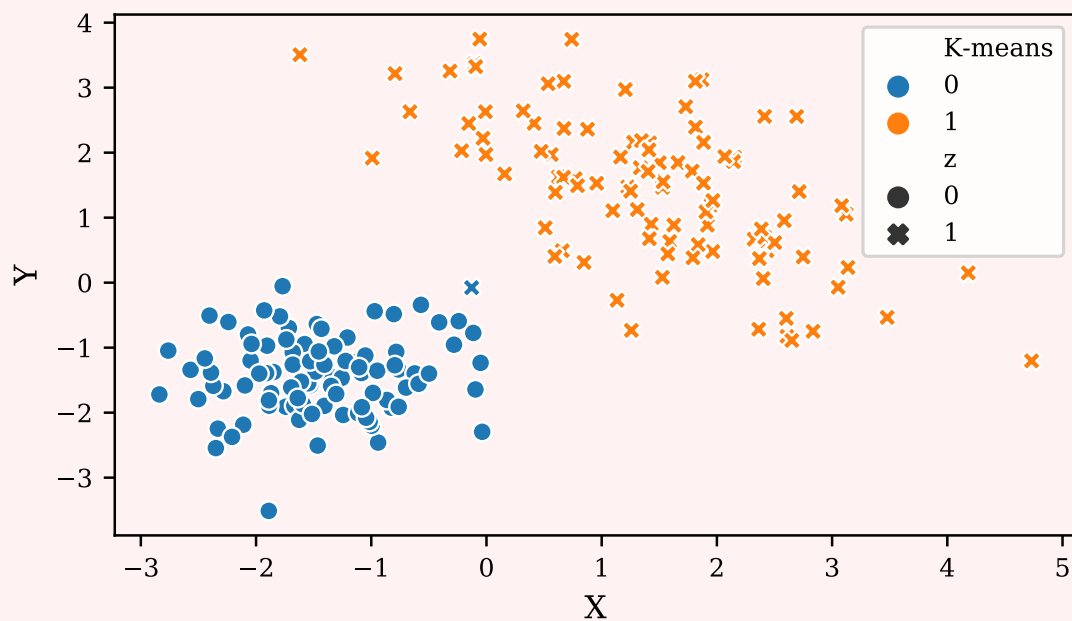
Pour chacun des jeux de données, effectuer une classification au moyen de l'algorithme des *K-means* classique, puis de l'algorithme des *K-means* avec distance adaptative. On comparera les résultats obtenus et on interprétera en fonction des données. On peut utiliser la fonction `adjusted_rand_score` 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 [23]: 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"])

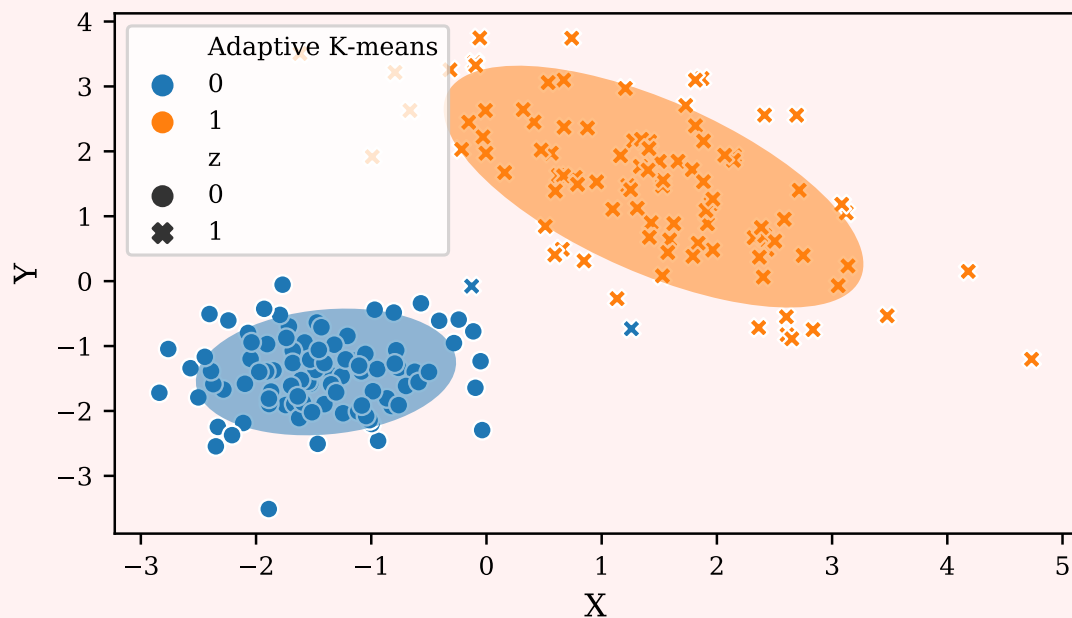
         cls = KMeans(n_clusters=2)
         labels = cls.fit_predict(X1)
         s1 = adjusted_rand_score(z1, labels)

         labels = pd.Series(labels, name="K-means")
         plot_clustering(X1, clus1=labels, clus2=z1)
Out [23]: (<matplotlib.axes._subplots.AxesSubplot object at 0x1a29b66790>, None)
In [24]: plt.show()
```



```
In [25]: cls = AdaptiveKMeans(n_clusters=2)
          labels = cls.fit_predict(X1.to_numpy())
          sia = 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 [25]: (<matplotlib.axes._subplots.AxesSubplot object at 0x1a28a90e50>, None)
In [26]: plt.show()
```



```

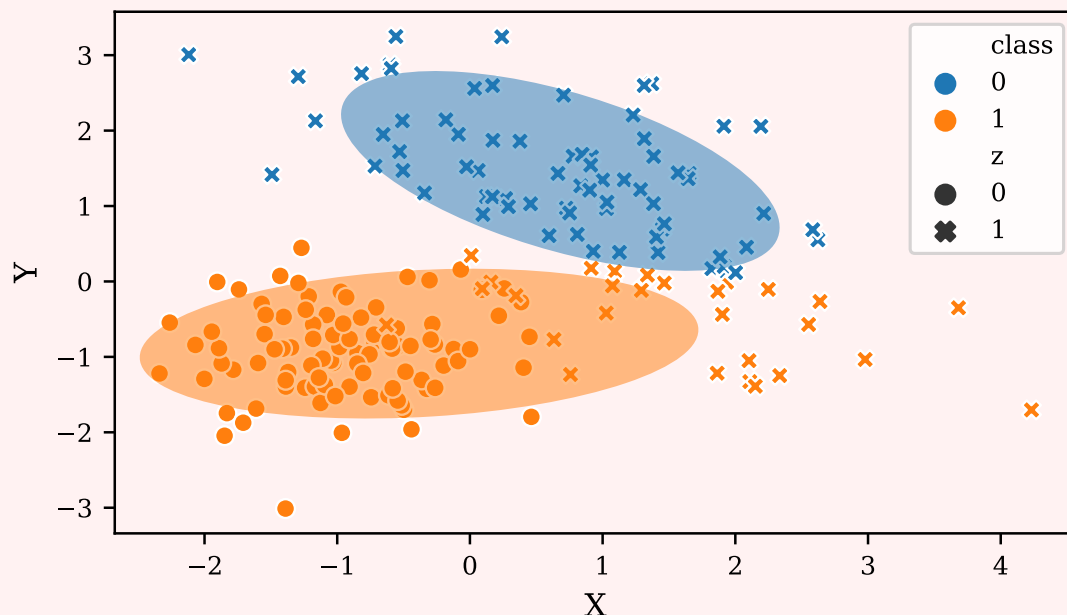
In [27]: X2 = pd.read_csv("data/Synth2.csv")
        z2 = X2.z
        X2 = X2.drop(columns=["z"])

        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 [27]: (<matplotlib.axes._subplots.AxesSubplot object at 0x1a29c8c1d0>, None)
In [28]: plt.show()

```



```

In [29]: 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 [29]: (<matplotlib.axes._subplots.AxesSubplot object at 0x1a29cd2190>, None)
In [30]: plt.show()

```



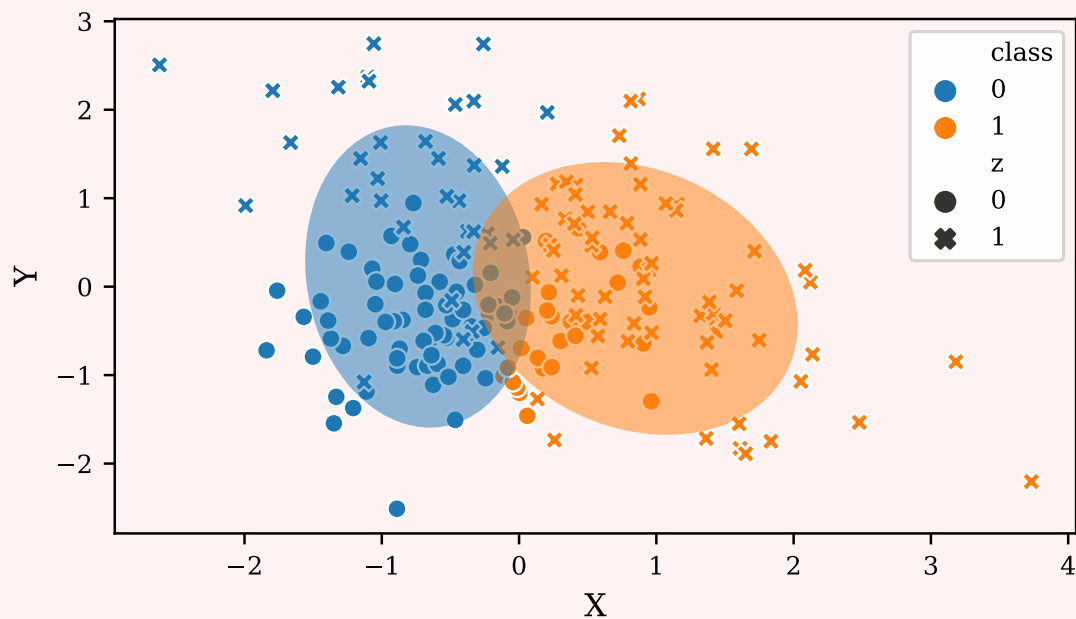


TABLE 1 – ARI sur les trois jeux de données synthétiques Synth1, Synth2 et Synth3.

	Synth1	Synth2	Synth3
ARI K-means	0.9800	0.845625	0.132860
ARI K-means adaptatif	0.9602	0.516159	0.132562

#### 4.3.2 Données réelles

**Iris** Déterminer une classification des données **Iris** avec la méthode des *K-means* classique, puis avec les *K-means* adaptatifs, pour  $K = 2, 3, \dots, 5$ . 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 ???. Les partitions en  $K = 2$  et  $K = 3$  groupes sont représentées dans la figure ??.

TABLE 2 – Critères sur les Iris.

# voisins	1	2	3	4	5
K-means	681.370600	152.347952	78.851441	57.256009	46.472230
K-means adaptatif	124.640638	60.593256	45.868037	37.671429	30.763948

**Crabs** Déterminer une classification des données **Crabs** avec la méthode des *K-means* classique, puis avec les *K-means* adaptatifs, pour  $K = 2, 3, \dots, 8$ . 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<sup>1</sup> 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))
```

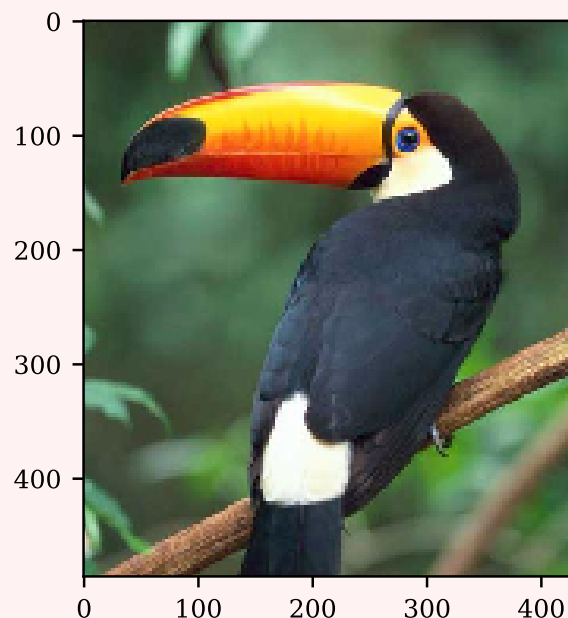
On effectuera une classification pixellaire avec les *K-means* et les *K-means* adaptatifs, pour un nombre de clusters variant entre  $K = 2$  et  $K = 5$ . On prendra soin de comparer les résultats donnés par les deux algorithmes.

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)
```

On pourra ensuite afficher ces images comme précédemment.

```
In [31]: img_toucan = plt.imread("data/toucan.png")
         plt.imshow(img_toucan)
Out [31]: <matplotlib.image.AxesImage object at 0x1a25637f90>
In [32]: plt.show()
```

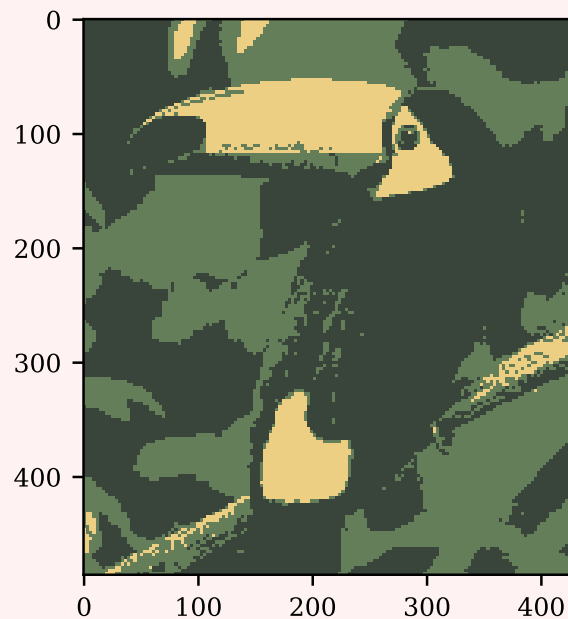


```
In [33]: mat_toucan = img_toucan.reshape((-1, 3))

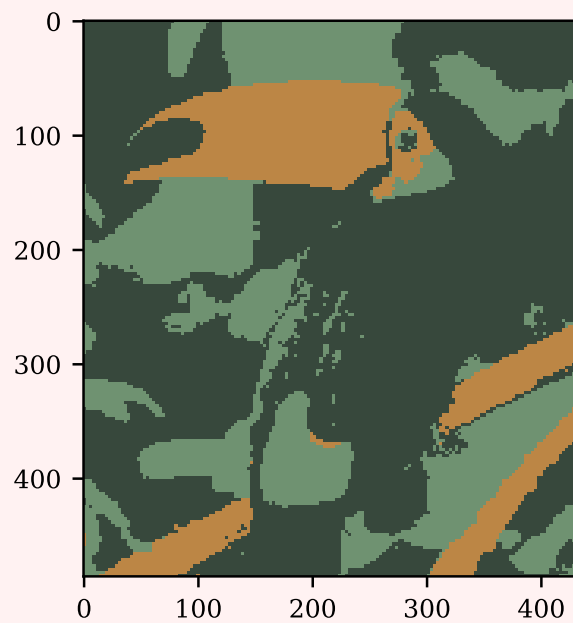
         km = KMeans(n_clusters=3)
         km.fit(mat_toucan)
```

1. La classification pixellaire ne tient pas compte de l'information de voisinage des pixels.

```
Out [33]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
                n_clusters=3, n_init=10, n_jobs=None,
                ↪ precompute_distances='auto',
                random_state=None, tol=0.0001, verbose=0)
In [34]: img_quant = km.cluster_centers_[km.labels_,
    ↪ :].reshape(img_toucan.shape)
        plt.imshow(img_quant)
Out [34]: <matplotlib.image.AxesImage object at 0x11f71d790>
In [35]: plt.show()
```

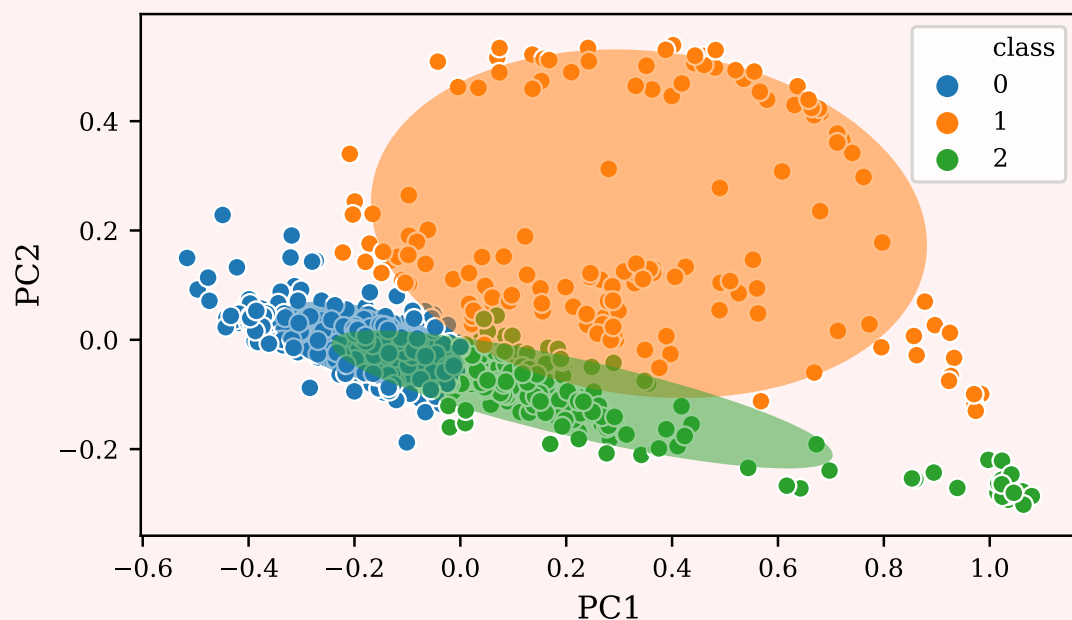


```
In [36]: kma = AdaptiveKMeans(n_clusters=3)
        kma.fit(mat_toucan)
        img_quant = kma.cluster_centers_[kma.labels_,
    ↪ :].reshape(img_toucan.shape)
        plt.imshow(img_quant)
Out [36]: <matplotlib.image.AxesImage object at 0x11f71d6d0>
In [37]: plt.show()
```



```
In [38]: idxs = np.random.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()
```



#### 4.4 Preuves de convergence

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  $P = (P_1, \dots, P_K)$ , où la classe de chaque individu est codée par une variable binaire

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

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

$$J(\{v_k, M_k\}_{k=1, \dots, K}) = \sum_{k=1}^K \sum_{i=1}^n z_{ik} d_{ik}^2 \quad (2)$$

sous la contrainte  $\det M_k = \rho_k$  pour tout  $k = 1, \dots, K$ , où la distance  $d_{ik}^2 = d_{M_k}^2(\mathbf{x}_i, \mathbf{v}_k)$  est définie par l'équation (1), donne

$$\mathbf{v}_k = \bar{\mathbf{x}}_k = \frac{1}{n_k} \sum_{i=1}^n z_{ik} \mathbf{x}_i, \quad (3)$$

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

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.

Remarquons que fixer  $\det M_k = \rho_k$  pour tout  $k = 1, \dots, K$  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 non inversible).

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

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

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

$$\frac{\partial \mathcal{L}(\{\mathbf{v}_k, M_k, \lambda_k\})}{\partial \lambda_k} = 0, \quad \frac{\partial \mathcal{L}(\{\mathbf{v}_k, M_k, \lambda_k\})}{\partial M_k} = \not= 0, \quad \frac{\partial \mathcal{L}(\{\mathbf{v}_k, M_k, \lambda_k\})}{\partial \mathbf{v}_k} = \not= 0.$$

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 pourra admettre ce résultat.

**Questions** 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](http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf).

1. Calculer  $\partial \mathcal{L}(\{\mathbf{v}_k, M_k, \lambda_k\})/\partial \mathbf{v}_k$  ; 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  $\mathbf{v}_k$  est aisément obtenue :

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

l'annulation de ce vecteur de dérivées premières donne

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

où  $n_k = \sum_{i=1}^n z_{ik}$ .

2. Calculer  $\partial \mathcal{L}(\{\mathbf{v}_k, M_k, \lambda_k\}) / \partial M_k$ , puis  $\partial \mathcal{L}(\{\mathbf{v}_k, M_k, \lambda_k\}) / \partial \lambda_k$ . 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  $\lambda_k$  :

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

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

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

L'annulation conjointe des dérivées premières par rapport à  $\lambda_k$  et  $M_k$  donne

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

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

$$\begin{aligned}\det M_k &= \det ((\lambda_k^{-1} \rho_k^{-1} n_k V_k)^{-1}) \\ &= \det (\lambda_k \rho_k n_k^{-1} V_k^{-1}) \\ &= \lambda_k^p \rho_k^p n_k^{-p} \det (V_k^{-1});\end{aligned}$$

cela donne

$$\lambda_k = \left( \rho_k^{-p} (\det (V_k^{-1}))^{-1} \det M_k n_k^p \right)^{1/p} = \left( \rho_k^{1-p} n_k^p \det V_k \right)^{1/p} = \rho_k^{1/p-1} n_k (\det V_k)^{1/p},$$

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

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

**Données :** Tableau de données  $X$  (matrice  $n \times p$ ), nb. de classes  $K$  (entier), volumes des classes  $\rho_k$  (vecteur de longueur  $K$ ), nb. max. d'itérations  $n_{\text{iter.max}}$  (entier), nb. d'essais  $n_{\text{start}}$  (entier), tolérance  $\varepsilon$  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  $i = 1, 2, \dots, n_{\text{start}}$  faire
    initialiser  $\mu_k$  et  $\tilde{V}_k$ , pour  $k = 1, \dots, K$ 
    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  $\mu_k$  le
            plus proche, au sens de la distance de Mahalanobis
        mémoriser les anciens centres des classes :  $\mu_{k,\text{prec}} \leftarrow \mu_k$ 
        mettre à jour les centres des classes  $\mu_k$  et les matrices de covariance empiriques  $\tilde{V}_k$ 
            normalisées au moyen de la partition calculée précédemment
        mettre à jour le critère de distance  $d_{\text{tot}}$  (somme des distances aux centres des classes)
        mettre à jour le critère de convergence
    fin
    si  $J^* < J^{\text{opt}}$  (avec  $J^{\text{opt}}$  le meilleur optimum local donné par les essais précédents) alors
        mettre à jour le meilleur optimum local : mémoriser la valeur du critère  $J^*$ , le nb.
            d'itérations, la partition des données, les centres et les matrices de covariance
    fin
fin

```

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



## 5 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 1.2.

- (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 [39]: 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`.

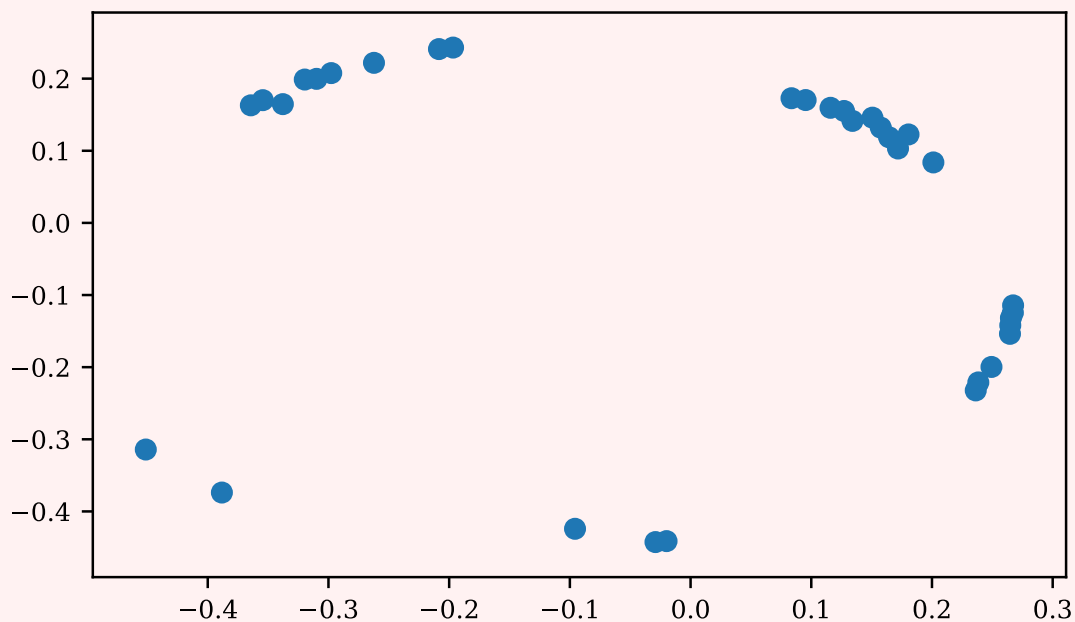
```
In [40]: clusterings = np.array([
    KMeans(n_clusters=3, init="random", n_init=1, random_state=i)
    .fit(crabsquant)
    .labels_
    for i in idxs_uniq
])

from sklearn.metrics import pairwise_distances
from sklearn.metrics import adjusted_rand_score
S = pairwise_distances(clusterings, metric=adjusted_rand_score)
D = 1 - np.maximum(S, 0)
```

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 [41]: 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'ATFD, isoler les classifications réellement différentes. Pour ce faire, on pourra utiliser à nouveau la méthode des *K-means*. Visualiser ces classifications.

```
In [42]: km = KMeans(n_clusters=5)
km.fit(dist)

Out [42]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=5, n_init=10, n_jobs=None,
    ↪ precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

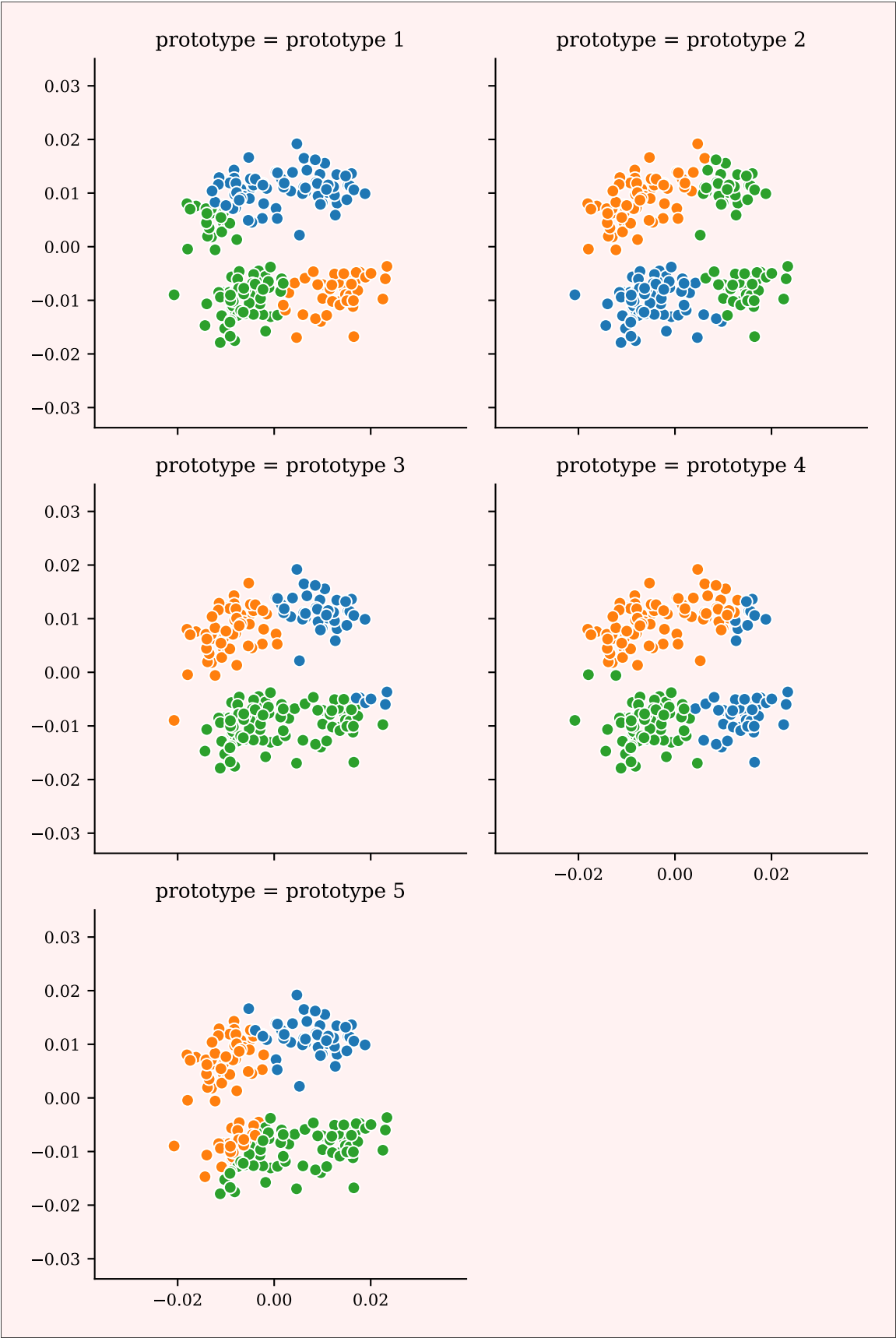


```
In [43]: _, idxs_uniq = np.unique(km.labels_, return_index=True)
         clustering_prototypes = clusterings[idxs_uniq, :]

         df_clustering_prototypes = pd.DataFrame(clustering_prototypes.T,
         ↪ columns=[f"prototype {i}" for i in range(1, 6)])

         df = pd.concat((crabsquant, df_clustering_prototypes), axis=1)
         df2 = pd.melt(df, id_vars=["FL", "RW", "CL", "CW", "BD"],
         ↪ var_name="prototype", value_name="class")

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





## 6 Itérations en folie

Soit  $n \geq 2$ . On définit le jeu de données unidimensionnel consistant en  $2n$  nombres  $y_1 < y_2 < \dots < y_n < x_n < x_{n-1} < \dots < x_1$  tels que  $y_i = -x_i$  pour  $i = 1, \dots, n$ . 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  $i = 1, \dots, n-1$ . 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 [44]: 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.

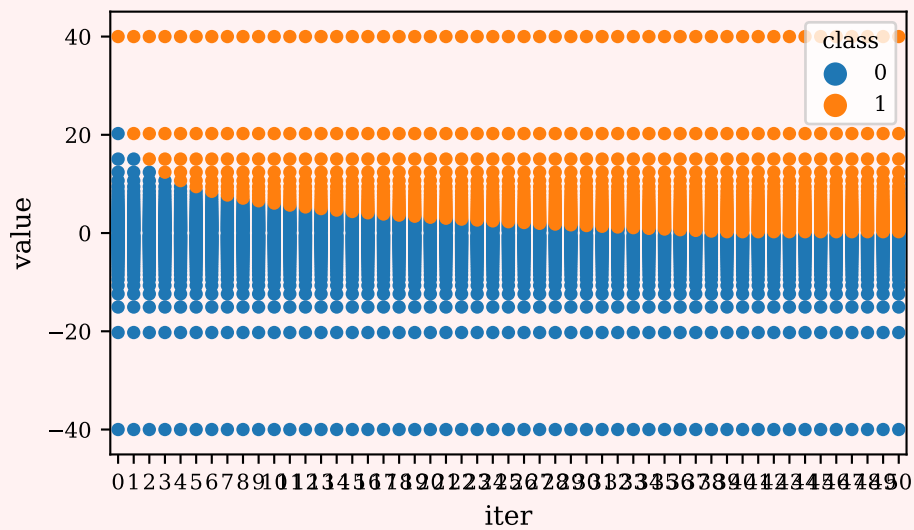
Ou pourra faire varier `max_iter` en fixant `n_init` à 1.

```
In [45]: from sklearn.cluster import KMeans
         n = 40

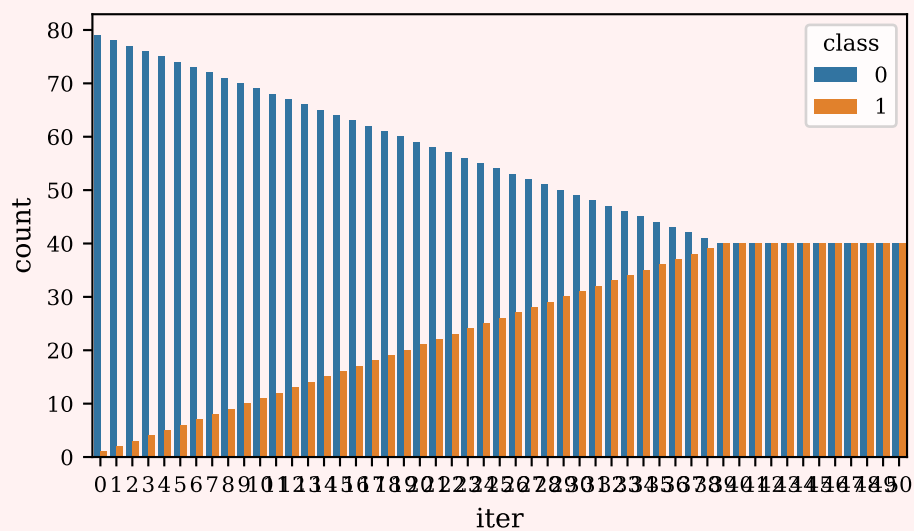
         def data_gen(X, max_iter):
             a = np.zeros(X.shape[0], dtype="int")
             a[-1] = 1
             yield pd.Series(a) # Le classement avant toute itération
             for i in range(1, max_iter+1):
                 cls = KMeans(n_clusters=2, init=X[-2:, :], max_iter=i,
                               tol=1e-10, 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")

         sns.stripplot(y="value", x="iter", hue="class", data=df, jitter=False)
Out [45]: <matplotlib.axes._subplots.AxesSubplot object at 0x1a28ac3350>
In [46]: plt.show()
```



```
In [47]: sns.countplot(x="iter", hue="class", data=df)
Out [47]: <matplotlib.axes._subplots.AxesSubplot object at 0x1a29c3cad0>
In [48]: 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 de données à  $2n$  points converge en  $n - 1$  itérations.



17 Construire une extension non triviale au cas multidimensionnel.