SY09 Printemps 2020
TP 05 — Classification hiérarchique

Pour ce TP, il sera nécessaire de disposer de la dernière version de `scikit-learn` (version `0.22.1`).

1 Visualisation des données `Mutations`

On rappelle que l'AFTD calcule une représentation multidimensionnelle, dans un espace euclidien de dimension $p \leq n$, de données se présentant sous la forme d'un tableau $n \times n$ de dissimilarités δ_{ij} entre n individus ($i, j \in \{1, \ldots, n\}$), dont le tableau de dissimilarités ne donne qu'une description implicite. Cette représentation est exacte lorsque les dissimilarités sont des distances euclidiennes.

Une fois que des variables ont été retenues, la qualité de la représentation peut être évaluée numériquement par un critère similaire au pourcentage d'inertie de l'ACP, ou graphiquement au moyen d'un *diagramme de Shepard* représentant la distance $d_{ij} = d(\boldsymbol{x}_i, \boldsymbol{x}_j)$ entre les représentations de $\boldsymbol{x}_i$ et $\boldsymbol{x}_j$ déterminées par l'AFTD en fonction de la dissimilarité initiale δ_{ij}, pour tout couple $(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

(1) Charger les données `Mutations`. Vérifier que le tableau chargé est bien carré.

```
In  [1]: mut = pd.read_csv("data/mutations2.csv", index_col=0)
         mut.shape
Out [1]: (20, 20)
```

Pour réaliser une AFTD avec `scikit-learn`, il faut charger la classe `MDS` avec l'instruction suivante

```
from sklearn.manifold import MDS
```

Il faut ensuite instancier cette classe en spécifiant la dimension de la représentation voulue avec l'argument `n_components` et préciser que les données sont fournies sous la forme d'un tableau de distance avec l'argument `dissimilarity='precomputed'`.

Il suffit ensuite d'appeler la méthode `fit_transform` en fournissant le tableau de distance en argument. La méthode renvoie les coordonnées de la nouvelle représentation.

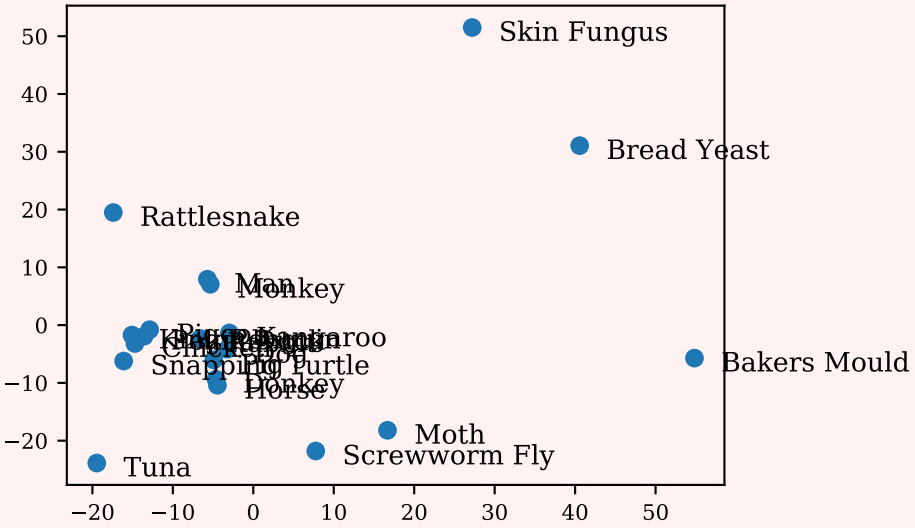
(2) Calculer une représentation euclidienne des données en $d = 2$ variables par AFTD.

```
In  [2]: from sklearn.manifold import MDS

         aftd = MDS(n_components=2, dissimilarity='precomputed')
         dist = aftd.fit_transform(mut)
```

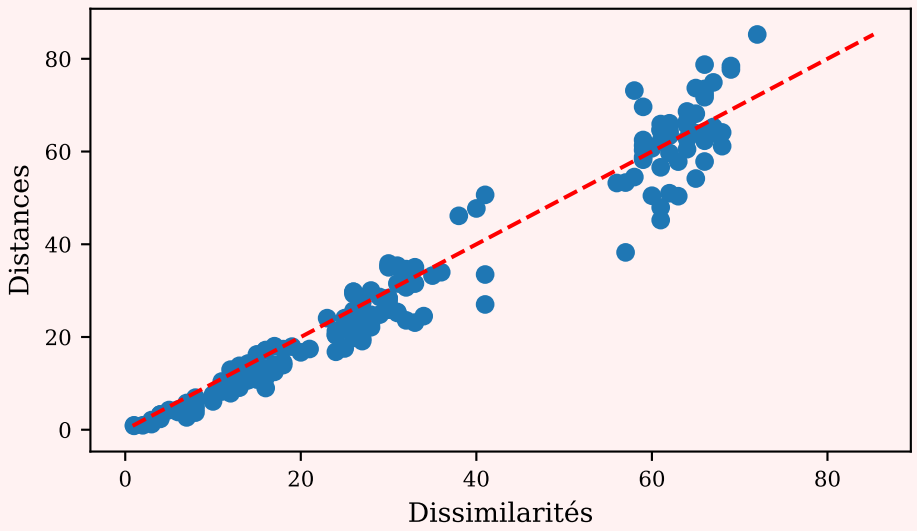
(3) Afficher la nouvelle représentation en deux dimensions. On pourra utiliser la fonction `add_labels` du TP03.

```
In  [3]: plt.scatter(*dist.T)
         add_labels(dist[:, 0], dist[:, 1], mut.index)
         plt.show()
```

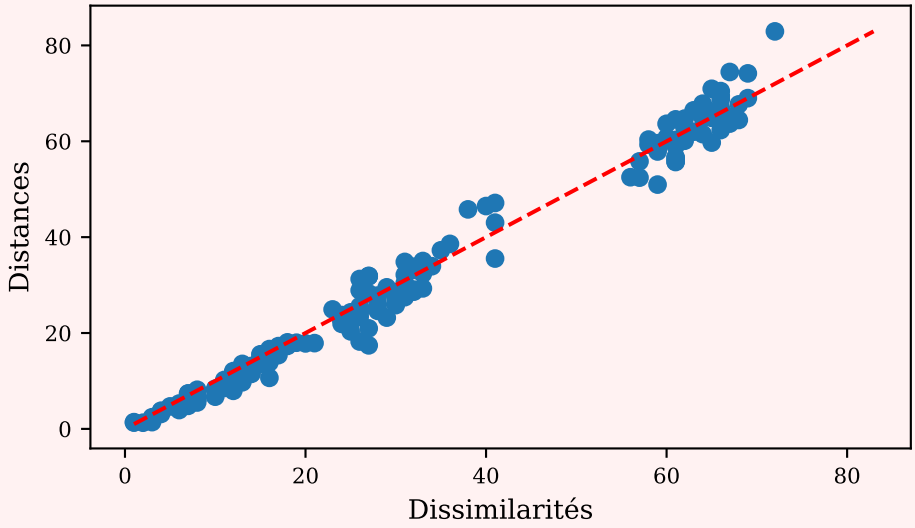


④ Afficher le diagramme de Shepard avec la fonction fournie `plot_Shepard`. Que peut-on dire ? Recommencer avec $d \in \{3, 4, 5\}$. Interpréter les résultats.

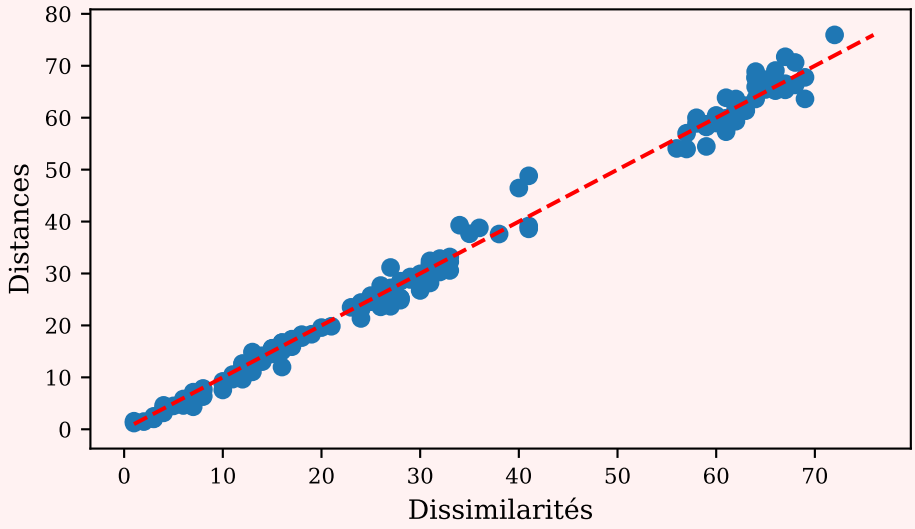
```
In  [4]: aftd = MDS(n_components=2, dissimilarity='precomputed')
         dist = aftd.fit_transform(mut)
         plot_Shepard(aftd)
         plt.show()
```



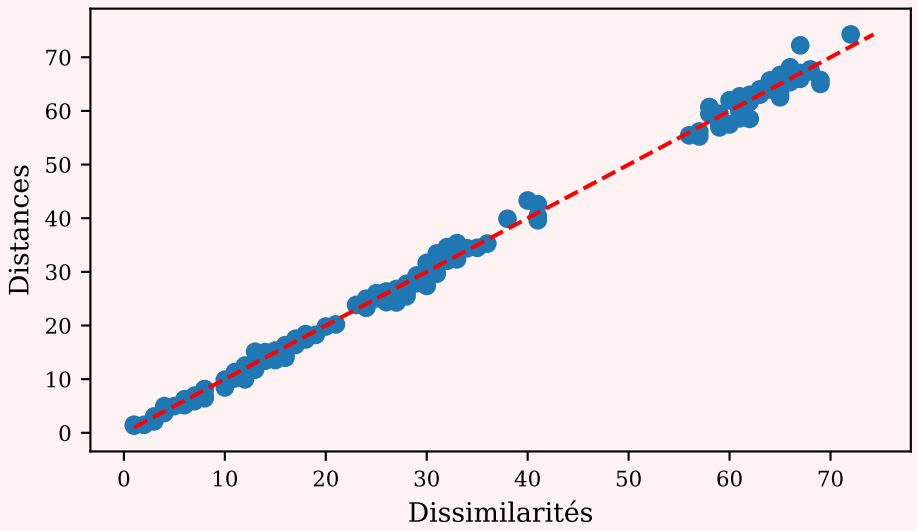

```
In  [5]: aftd = MDS(n_components=3, dissimilarity='precomputed')
         dist = aftd.fit_transform(mut)
         plot_Shepard(aftd)
         plt.show()
```



In [6]:
```
aftd = MDS(n_components=4, dissimilarity='precomputed')
dist = aftd.fit_transform(mut)
plot_Shepard(aftd)
plt.show()
```



In [7]:
```
aftd = MDS(n_components=5, dissimilarity='precomputed')
dist = aftd.fit_transform(mut)
plot_Shepard(aftd)
plt.show()
```



En ré-exécutant ces instructions pour des valeurs croissantes de d, on constate que les points sont de plus en plus proches de la première bissectrice.

(5) Retrouver le « stress » avec les distances fournies par `plot_Shepard`. On rappelle que la fonction « stress » que cherche à minimiser `scikit-learn` est définie par

$$\text{Stress} = \sum_{i<j} (d_{ij} - \delta_{ij})^2.$$

```
In  [8]: aftd.stress_
Out [8]: 287.6571287466462
In  [9]: diss, dist = plot_Shepard(aftd, plot=False)
         np.sum((diss - dist)**2)
Out [9]: 287.2327224904273
```

(6) Dans le diagramme de Shepard, vérifier que l'inertie I_{b_1} du nuage de points par rapport à la première bissectrice b_1 vérifie

$$\text{Stress} = 2mI_{b_1},$$

avec m le nombre de points dans le diagramme de Shepard. Pourquoi ce résultat ?

```
In  [10]: diss, dist = plot_Shepard(aftd, plot=False)
          stress = np.sum((diss - dist)**2)

          X = np.column_stack((diss, dist))
          n, p = X.shape
          feat_metric = np.eye(p)
          sample_metric = 1 / n * np.eye(n)
          feat_inner, inertia_point, inertia_axe = inertia_factory(feat_metric,
          ↪  sample_metric)
          inertia = inertia_axe(X, np.ones(2))

          n * inertia / stress
Out [10]: 0.5000000000000004
```

Les segments des inerties sont inclinés à 45 degrés et les segments du « stress » sont verticaux. Le rapport des deux vaut donc $\sin \pi/4 = \sqrt{2}/2$. Les segments étant au carré ont retrouve bien le rapport $1/2$.

(7) Peut-on choisir la matrice M de telle sorte que Stress $= mI_{b_1}$?

Il faudrait choisir la matrice M de telle sorte que les vecteurs

$$e_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

constituent une base orthonormale (au sens de M). Dès lors, la projection orthogonale (pour M) parallèlement à e_2 sur e_1 donne une projection verticale.
On veut donc

$$(e_1, e_2)^T M (e_1, e_2) = I_2,$$

d'où on tire

$$M = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}.$$

```
In  [11]: feat_metric = np.array([[2, -1], [-1, 1]])
          feat_inner, inertia_point, inertia_axe = inertia_factory(feat_metric,
          ↪  sample_metric)
          inertia = inertia_axe(X, np.ones(2))

          feat_inner(np.array([1, 1]), np.array([0, 1]))
Out [11]: 0
In  [12]: n * inertia / stress
Out [12]: 1.0000000000000009
```

2 Classification ascendante hiérarchique

La bibliothèque `scikit-learn` dispose d'un algorithme de classification ascendante hiérarchique. Pour cela, il faut importer la classe suivante :

```
from sklearn.cluster import AgglomerativeClustering
```

Il faut ensuite instancier cette classe. Les paramètres qui nous intéressent sont
- `linkage` : le critère d'agglomération,
- `affinity` : la distance utilisée pour calculer le critère d'agglomération.

Par exemple, pour faire la classification ascendante hiérarchique d'un tableau individus-variables `X` en utilisant la distance euclidienne et le critère d'agglomération maximum, on construit l'objet

```
cls = AgglomerativeClustering(linkage="maximum", affinity="euclidean")
```

Pour apprendre la classification, il faut utiliser la méthode `fit` en fournissant le jeu de données `X`

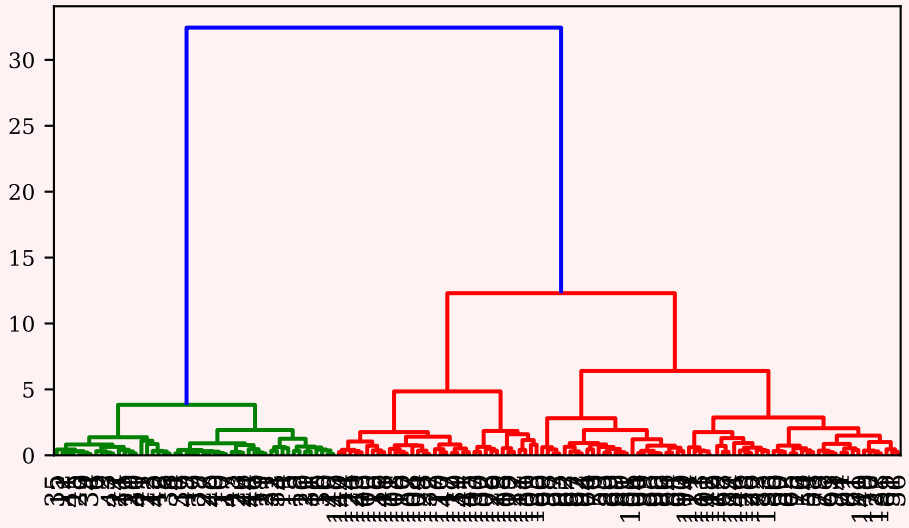
```
cls.fit(X)
```

Pour visualiser la hiérarchie indicée obtenue, la fonction `plot_dendrogram` fournie dans le fichier `src/utils.py` pourra être utilisée. Par défaut, `scikit-learn` ne calcule pas les distances permettant de tracer la hiérarchie. Il faut fournir les paramètres `distance_threshold` et `n_clusters` initialisés respectivement à `0` et `None`.

(8) Effectuer une classification ascendante hiérarchique des données `Iris`. Commenter les résultats obtenus, en vous appuyant sur votre connaissance de ce jeu de données.

```
In [13]: from sklearn.cluster import AgglomerativeClustering

         iris = sns.load_dataset("iris")
         cls = AgglomerativeClustering(
             affinity="euclidean", linkage="ward", distance_threshold=0,
             ↪ n_clusters=None
         )
         cls.fit(iris.drop(columns=["species"]))
         plot_dendrogram(cls)
         plt.show()
```

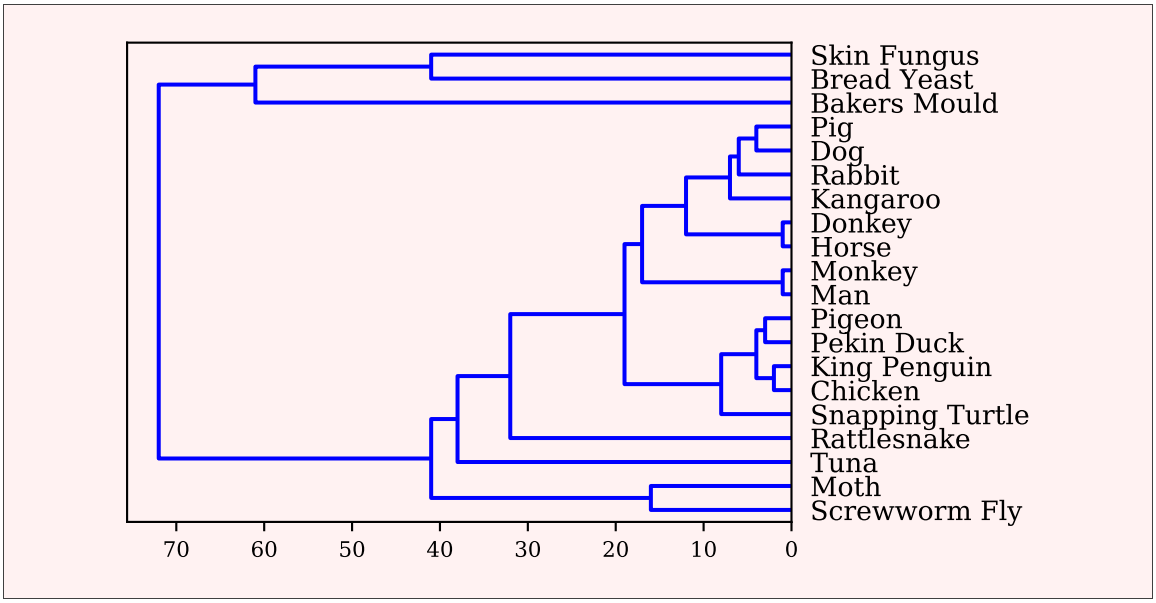


Lorsque les distances entre individus sont déjà calculées, le paramètre `affinity` est inutile. On lui attribue la valeur `"precomputed"` et au lieu de fournir le tableau individu-variable à la méthode `fit`, on lui donne directement le tableau de distances.

(9) Effectuer la classification hiérarchique ascendante des données de `Mutations` (avec les différents critères d'agrégation disponibles). Commenter et comparer les résultats obtenus, en vous appuyant sur la représentation obtenue par AFTD.

```
In [14]: mut = pd.read_csv("data/mutations2.csv", index_col=0)

         model = AgglomerativeClustering(
             affinity="precomputed", linkage="complete", distance_threshold=0,
             ↪ n_clusters=None
         ).fit(mut)
         plot_dendrogram(model, color_threshold=1, labels=mut.index,
         ↪ orientation="left")
         plt.show()
```



Dans certain cas, il n'est pas nécessaire voire souhaitable de mener la classification ascendante hiérarchique à son terme. Pour cela, on dispose des arguments `n_clusters` et `distance_threshold`. L'argument `n_clusters` signifie qu'il faut arrêter l'agglomération dès qu'on a obtenu `n_clusters` groupements. L'argument `distance_threshold` signifie qu'il faut arrêter l'agglomération dès que le critère d'agglomération descend sous ce seuil.

Dès lors, les attributs suivants sont disponibles depuis l'objet `cls`

- `n_clusters_` : le nombre de groupements obtenu,
- `labels_` : les étiquettes des individus données sous forme de nombre entier indiquant l'appartenance à un groupement.

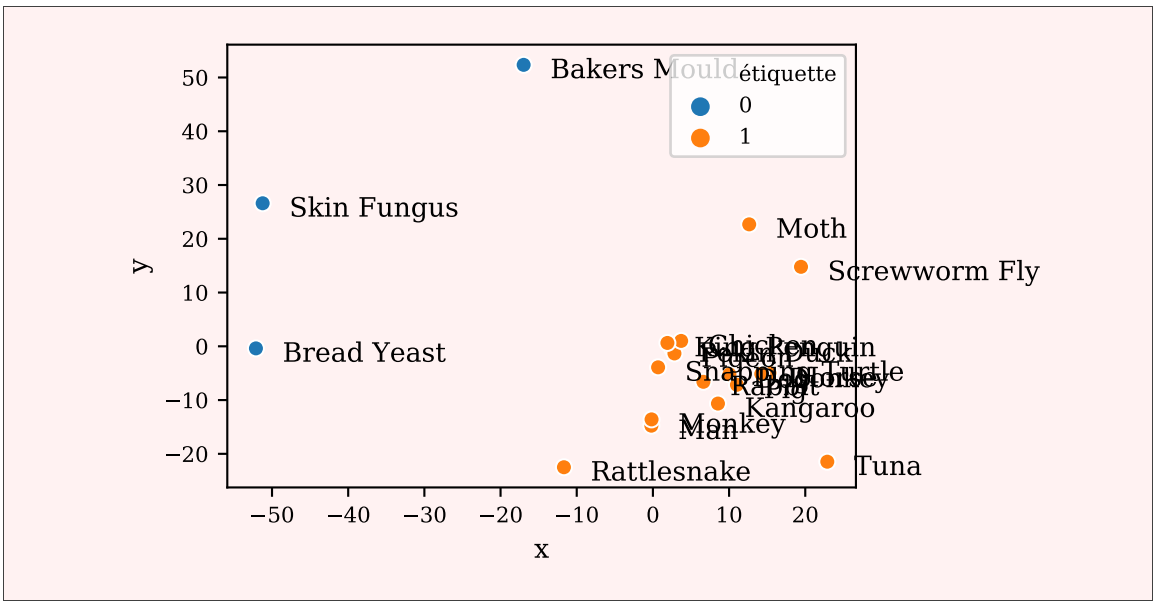
(10) Donner une partition en deux groupes à partir d'une classification ascendante hiérarchique. Visualiser cette partition en utilisant une AFTD en deux dimensions.

```
In [15]: model = AgglomerativeClustering(
             affinity="precomputed", linkage="complete", n_clusters=2
         ).fit(mut)
         labels = model.labels_

         aftd = MDS(n_components=2, dissimilarity="precomputed")
         dist = aftd.fit_transform(mut)

         df = pd.DataFrame({"x": dist[:, 0], "y": dist[:, 1], "étiquette":
         ↪  labels})

         sns.scatterplot("x", "y", hue="étiquette", data=df)
         add_labels(dist[:, 0], dist[:, 1], mut.index)
         plt.show()
```



3 Inertie intra-classe et critère de Ward

Dans cette section, on cherche à montrer expérimentalement la relation qui existe entre l'inertie intra-classe et le critère d'agglomération de Ward lors d'une classification ascendante hiérarchique. Plus précisément, on a

$$I_W(P') - I_W(P) = \frac{1}{n} D_{\text{Ward}}(A, B),$$

avec P la partition avant fusion, P' la partition après fusion et A et B les deux groupements minimisant le critère d'agglomération de Ward et qui vont être fusionnés.

On va d'abord chercher à calculer l'inertie intra-classe avant chaque fusion lors d'une classification ascendante hiérarchique. On va jouer sur le paramètre `n_clusters` de la classe `AgglomerativeClustering` et récupérer la classification pour calculer son inertie intra-classe.

(11) Créer une fonction `inertia` qui prend en argument un sous-jeu de données représentant un groupement ainsi que la taille du jeu de données total et renvoie l'inertie de ce sous-jeu de données.

On pourra utiliser la fonction `np.cov` ainsi que la trace `np.trace`. Attention toutefois au cas où le sous-jeu de données est réduit à un individu.

On contrôlera la justesse de la fonction avec les assertions suivantes :

```
import math
import seaborn as sns

iris = sns.load_dataset("iris")
iris0 = iris.drop(columns="species")
n = iris0.shape[0]
assert math.isclose(inertia(iris0, n), 4.542470666666669)
assert math.isclose(
    inertia(iris0.loc[iris.species == "setosa"], n), 0.10100666666666669
)
assert math.isclose(
    inertia(iris0.loc[iris.species == "versicolor"], n), 0.20410933333333345
)
assert math.isclose(
    inertia(iris0.loc[iris.species == "virginica"], n), 0.29019999999999996
)
```

```
In [16]: def inertia(cluster, n):
             nk = cluster.shape[0]

             if nk == 1:
                 return 0

             V = np.cov(cluster, rowvar=False, bias=True)
             return nk * np.trace(V) / n


         import math
         import seaborn as sns

         iris = sns.load_dataset("iris")
         iris0 = iris.drop(columns="species")
         n = iris0.shape[0]
         assert math.isclose(inertia(iris0, n), 4.542470666666669)
         assert math.isclose(
             inertia(iris0.loc[iris.species == "setosa"], n),
             ↪ 0.10100666666666669
         )
         assert math.isclose(
             inertia(iris0.loc[iris.species == "versicolor"], n),
             ↪ 0.20410933333333345
         )
         assert math.isclose(
             inertia(iris0.loc[iris.species == "virginica"], n),
             ↪ 0.29019999999999996
         )
```

(12) En utilisant la fonction précédente, créer une fonction `intra_class` qui prend en argument un nombre `n_clusters` de groupements et un jeu de données et renvoie l'inertie intra-classe de la classification ascendante hiérarchique avec critère d'agglomération de Ward en `n_clusters` groupements.

On pourra utiliser la méthode `groupby` sur le jeu de données afin de grouper le jeu de données par groupements et ensuite appliquer avec la méthode `apply` la fonction précédente.

Créer ensuite la liste des inerties intra-classe pour un nombre de groupements maximum jusqu'à un nombre de groupement minimum (c'est à dire 1).

```
In [17]: def intra_class(n_clusters, X):
             cls = AgglomerativeClustering(
                 affinity="euclidean", linkage="ward", n_clusters=n_clusters
             )
             cls.fit(X)
             n = X.shape[0]
             return X.groupby(cls.labels_).apply(inertia, n).sum()

         intra_class_list = [intra_class(i, iris0) for i in
         ↪ np.arange(iris0.shape[0], 0, -1)]
```

(13) L'argument `distances_` présent lorsque la classification ascendante hiérarchique est menée à son terme contient tous les critères d'agglomération successifs. Pour des raisons obscures, au lieu du critère de Ward c, `scikit-learn` renvoie la quantité $\sqrt{2c}$.

Calculer les critères de Ward successifs.

```
In  [18]: cls = AgglomerativeClustering(
              affinity="euclidean", linkage="ward", distance_threshold=0,
              ↪  n_clusters=None
          )
          cls.fit(iris0)
Out [18]: AgglomerativeClustering(affinity='euclidean', compute_full_tree='auto',
                                   connectivity=None, distance_threshold=0,
                                   ↪  linkage='ward',
                                   memory=None, n_clusters=None)
In  [19]: ward_list = cls.distances_**2/2
```

(14) Vérifier que les inerties intra-classe et les critères de Ward sont liés par la relation

$$I_W(P') - I_W(P) = \frac{1}{n} D_{\text{Ward}}(P, P').$$

```
In  [20]: n = iris0.shape[0]
          np.allclose(np.diff(intra_class_list), ward_list / n)
Out [20]: True
```