SY09 P2025
TD/TP 5 — Classification hiérarchique

`numpy`=2.2.3; `seaborn`=0.13.2; `matplotlib`=3.10.1; `pandas`=2.2.3; `sklearn`=1.6.1

Pour ce TP, il sera nécessaire de disposer d'une version récente de `scikit-learn` (au moins 0.22.1).

1 Travaux pratiques

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.

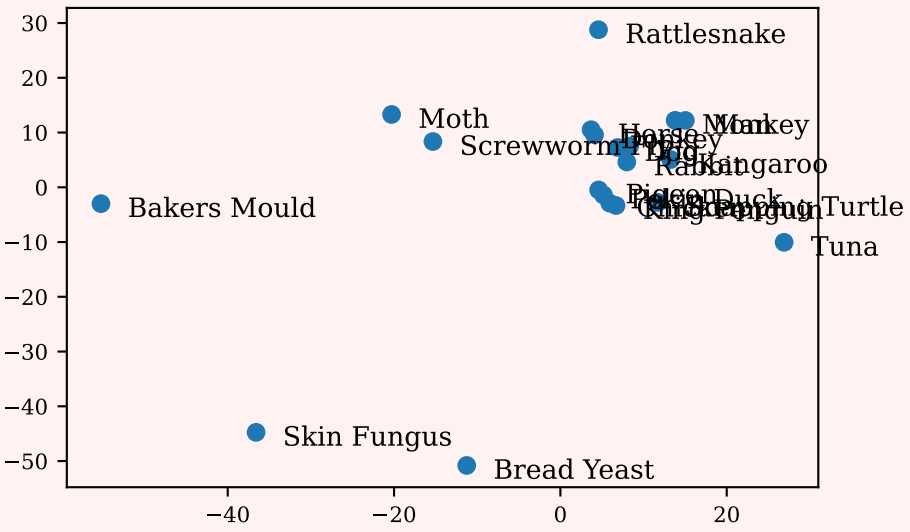
2 Calculer une représentation euclidienne des données en $d = 2$ variables par AFTD.

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

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

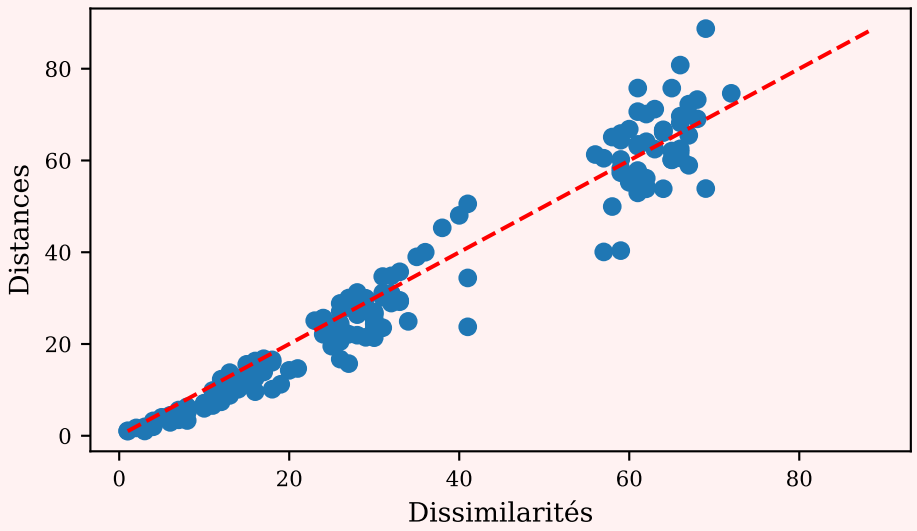
3 Afficher la nouvelle représentation en deux dimensions. On pourra utiliser la fonction `add_labels` du TP03.

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

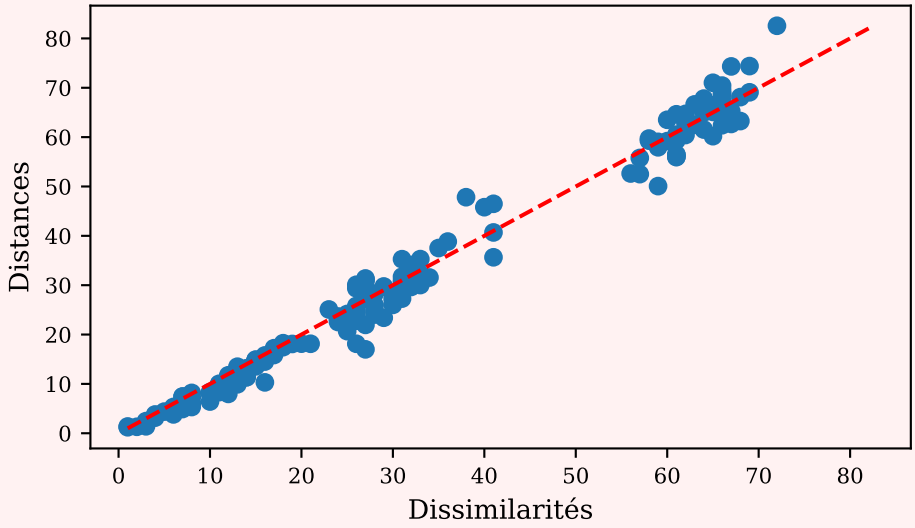


4 Afficher le diagramme de Shepard avec la fonction fournie `plot_Shepard`. Que peut-on dire ? Recommencer avec $d \in \{3, 4, 5\}$. Interpréter les résultats.

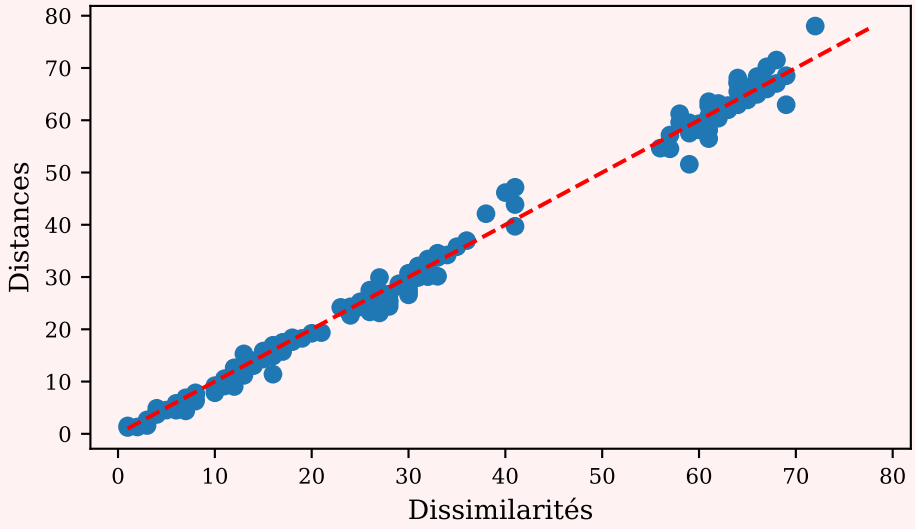
```
In [4]: aftd = MDS(n_components=2, dissimilarity='precomputed')
        dist = aftd.fit_transform(mut)
        plot_Shepard(aftd)
        plt.show()
```



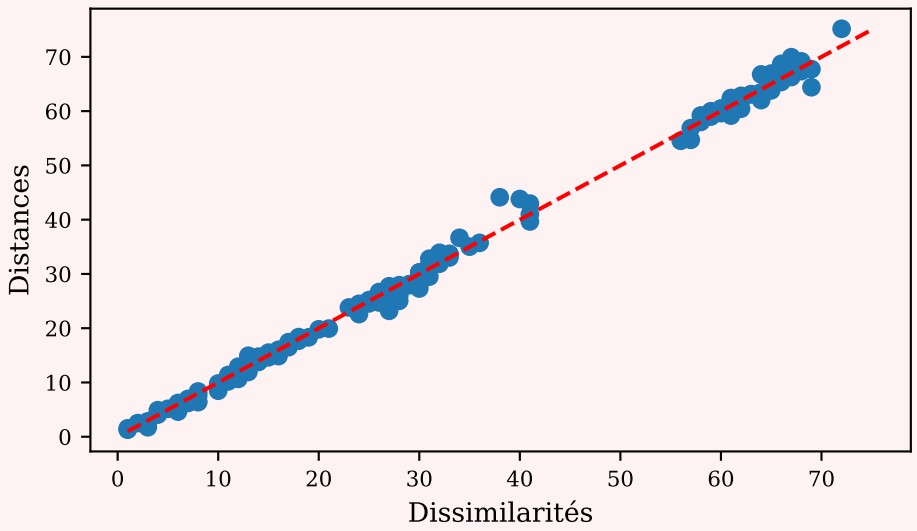

```
In [5]: aftd = MDS(n_components=3, dissimilarity='precomputed')
        dist = aftd.fit_transform(mut)
        plot_Shepard(aftd)
        plt.show()
```



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]: np.float64(280.18896991409105)
In  [9]: diss, dist = plot_Shepard(aftd, plot=False)
         np.sum((diss - dist)**2)
Out [9]: np.float64(279.78710941842644)
```

1.2 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,
- `metric` : 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="complete", metric="euclidean")
```

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

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

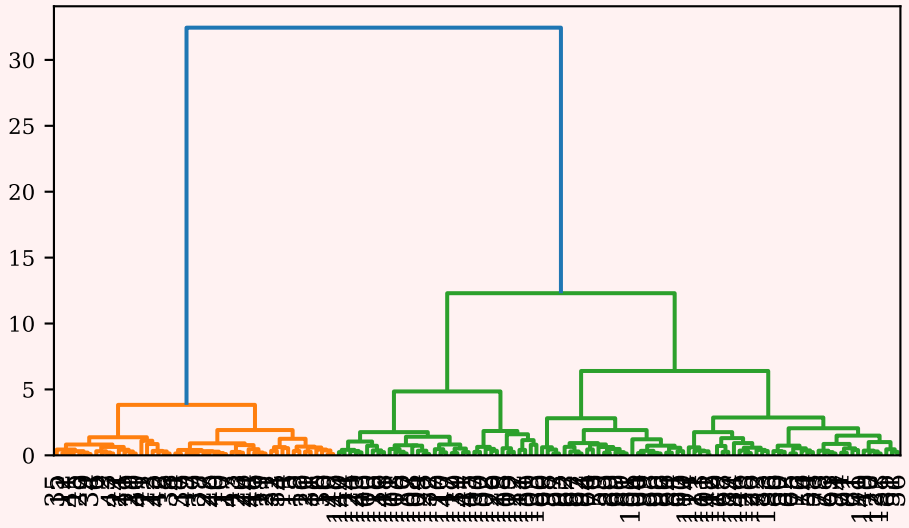
Pour visualiser la hiérarchie indicée obtenue, la fonction `plot_dendrogram` fournie dans le fichier `src/utils.py` pourra être utilisée. Par défaut, `scikit-learn` ne calcule pas les distances permettant

de tracer la hiérarchie. Il faut fournir les paramètres `distance_threshold` et `n_clusters` initialisés respectivement à `0` et `None`.

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

```
In [10]: from sklearn.cluster import AgglomerativeClustering

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

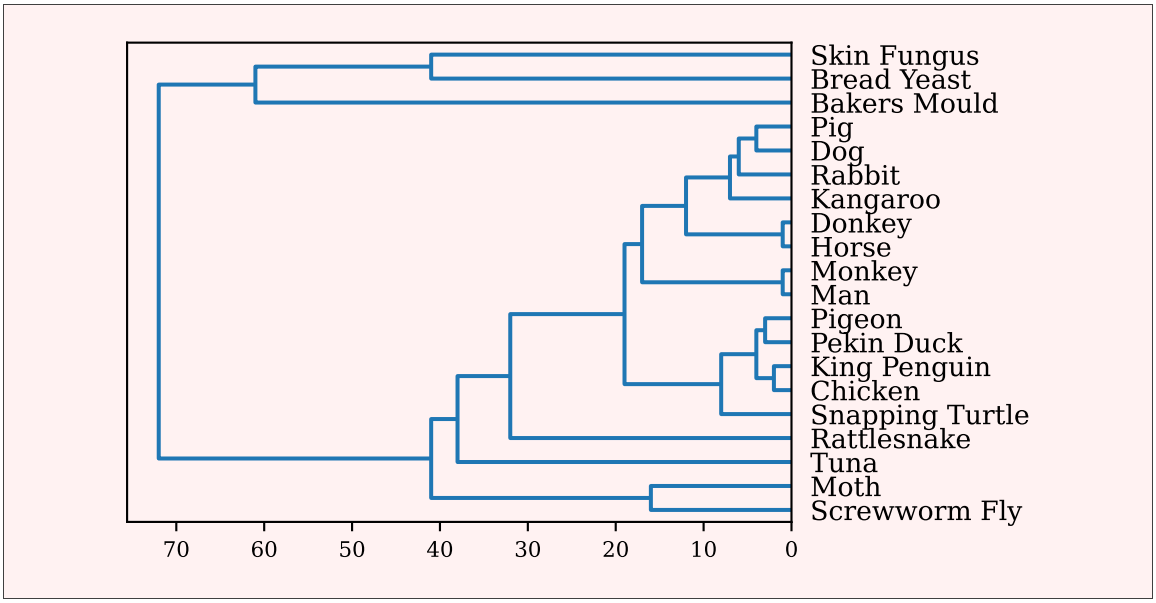


Lorsque les distances entre individus sont déjà calculées, le paramètre `metric` est inutile. On lui attribue la valeur `"precomputed"` et au lieu de fournir le tableau individu-variable à la méthode `fit`, on lui donne directement le tableau de distances.

7 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 [11]: mut = pd.read_csv("data/mutations2.csv", index_col=0)

         model = AgglomerativeClustering(
             metric="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.

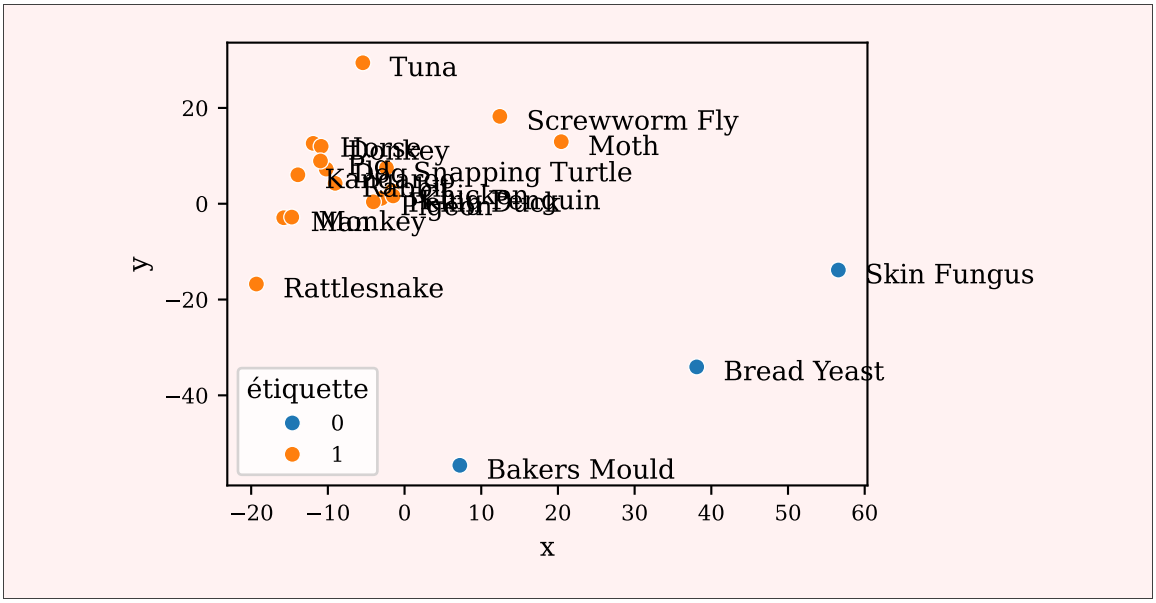
8 Donner une partition en deux groupes à partir d'une classification ascendante hiérarchique. Visualiser cette partition en utilisant une AFTD en deux dimensions.

```
In [12]: model = AgglomerativeClustering(
             metric="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="x", y="y", hue="étiquette", data=df)
         add_labels(dist[:, 0], dist[:, 1], mut.index)
         plt.show()
```



1.3 Inertie intra-classe et critère de Ward

Dans cette partie, 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.

9 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  [13]: 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
          )
```

10 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 CAH avec critère 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  [14]: def intra_class(n_clusters, X):
              cls = AgglomerativeClustering(
                  metric="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)]
```

11 L'argument `distances_` présent lorsque la classification ascendante hiérarchique est menée à son terme contient tous les critères d'agglomération successifs. Au lieu du critère de Ward c, `scikit-learn` renvoie la quantité $\sqrt{2c}$). Ce recalage est nécessaire si on veut que le critère d'agglomération entre deux feuilles coincide avec la distance entre deux feuilles.

Calculer les critères de Ward successifs.

```
In  [15]: cls = AgglomerativeClustering(
              metric="euclidean", linkage="ward", distance_threshold=0,
              ↪  n_clusters=None
          )
          cls.fit(iris0)

Out [15]: AgglomerativeClustering(distance_threshold=0, n_clusters=None)

In  [16]: ward_list = cls.distances_**2/2
```

12 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  [17]: n = iris0.shape[0]
          np.allclose(np.diff(intra_class_list), ward_list / n)

Out [17]: True
```

1.4 Euclidianité et euclidianisation

L'AFTD permet de trouver une représentation euclidienne si elle existe. Au passage, on dispose d'un test pour savoir si une dissimilarité admet une représentation euclidienne. En effet, si D est une matrice de dissimilarité, elle est euclidienne si et seulement si la matrice W_D suivante

$$W_D = -\frac{1}{2} Q_n D_2 Q_n, \tag{1}$$

est semi-définie positive avec Q_n la matrice de centrage et D_2 la matrice D où chaque entrée est mise au carré. Il suffit donc de calculer la valeur propre la plus petite de W_D et de vérifier si elle est positive ou non.

13 Créer deux fonctions, `Wd` et `smallest_eigenvalue` qui renvoient respectivement la matrice W_D et la plus petite valeur propre de W_D.

On pourra utiliser la fonction `eigvalsh` disponible dans le module suivant

```
import scipy.linalg as linalg
```

qui calcule les valeurs propres d'une matrice symétrique.

```
In  [18]: import scipy.linalg as linalg


          def Wd(D):
              D = D ** 2

              # Double centrage (on utilise le broadcasting)
              D = D - D.mean(axis=0)
              D = D - D.mean(axis=1, keepdims=True)

              return -0.5 * D


          def smallest_eigenvalue(D):
              W = Wd(D)
              return min(linalg.eigvalsh(W))
```

14 On considère la matrice de distance suivante

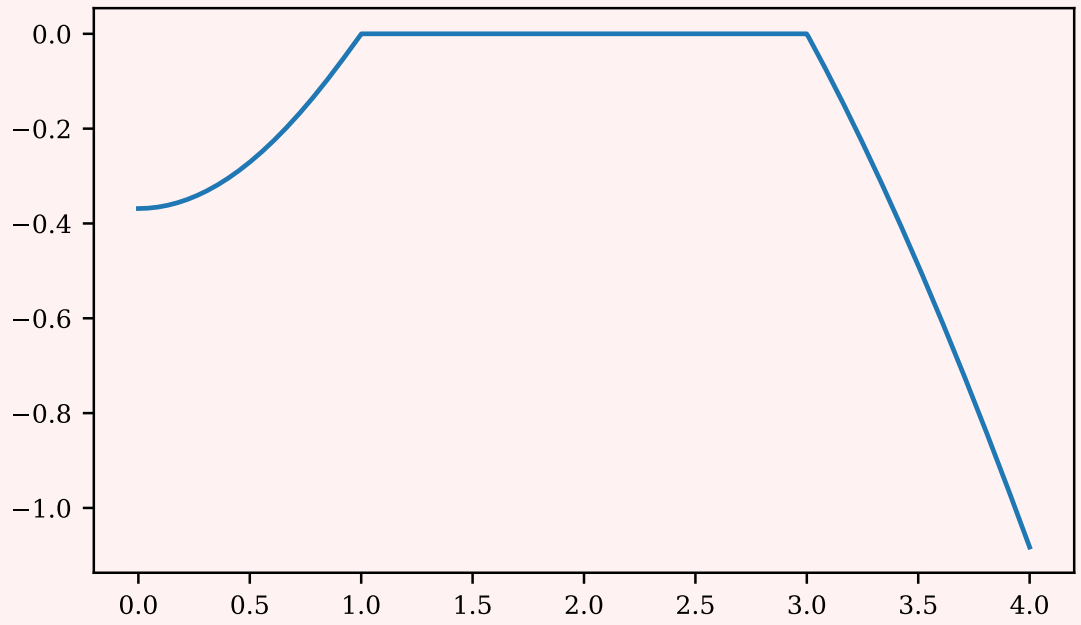
$$D = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & a \\ 2 & a & 0 \end{pmatrix}$$

À quelle condition sur a la matrice D est-elle euclidienne ?

Il nous faut 3 points dont les inter-distances sont 1, 2 et a. D'après l'inégalité triangulaire, il faut que $a \leq 3$ et $a \geq 1$. On vérifie sans mal que si c'est le cas, on peut trouver 3 points dans le plan vérifiant de telles inter-distances.

15 À l'aide de la fonction `smallest_eigenvalue`, vérifier expérimentalement le résultat précédent.

```
In [19]: a_s = np.linspace(0, 4, 1000)
         smallest_eigenvalues = [
             smallest_eigenvalue(np.array([[0, 1, 2], [1, 0, a], [2, a, 0]]))
             for a in a_s
         ]
         plt.plot(a_s, smallest_eigenvalues)
         plt.show()
```



À partir d'une dissimilarité quelconque D, on définit une autre dissimilarité D^γ comme suit

$$D^\gamma_{ij} = \begin{cases} 0 & \text{si } i = j, \\ D_{ij} + \gamma & \text{sinon,} \end{cases}$$

avec $\gamma > -\min_{i \neq j} D_{ij}$.

16 Montrer expérimentalement que D^γ est une matrice euclidienne à partir d'une certaine valeur de γ.

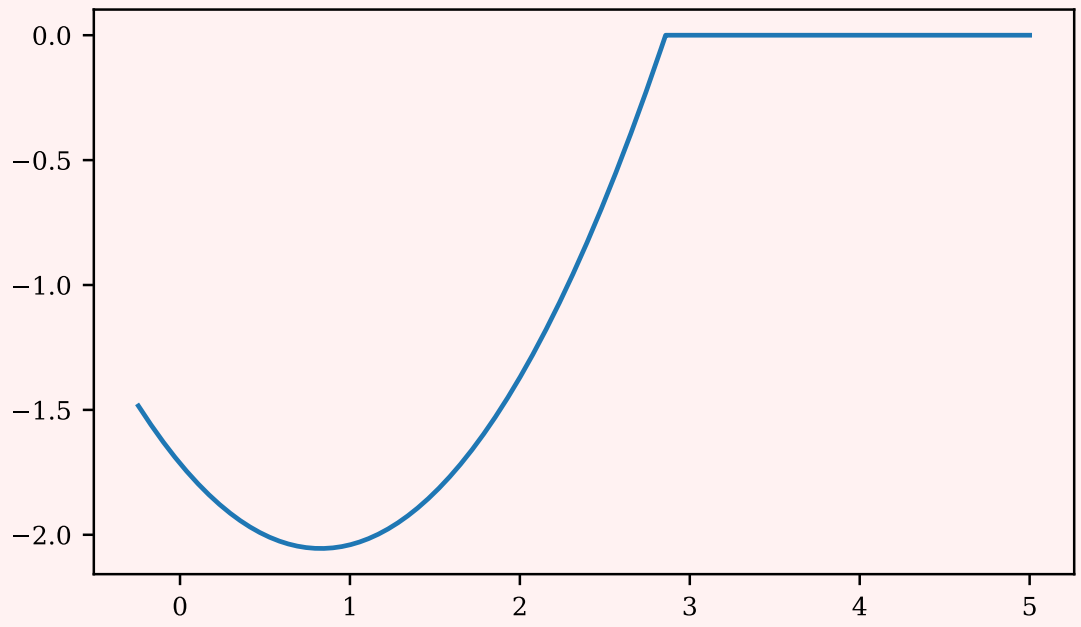
In [20]:
```
def ndiagadd(d, e):
    "Ajoute la quantité `e` hors diagonale"
    d = d + e
    np.fill_diagonal(d, 0)
    return d


# Matrice de distance aléatoire
from numpy.random import default_rng
rng = default_rng(43)
N = 5
D = rng.exponential(scale=1, size=(N, N))
D = (D + D.T) / 2
np.fill_diagonal(D, 0)

# Valeur g_min minimale telle que D + g_min >= 0
g_min = -min(D[D > 0])

gammas = np.linspace(g_min, 5, 1000)
smallest_eigenvalues = np.array([smallest_eigenvalue(ndiagadd(D, e))
↪   for e in gammas])

plt.plot(gammas, smallest_eigenvalues)
plt.show()
```



17 Vérifier expérimentalement que cette valeur est la plus grande valeur propre de la matrice suivante

$$B = \begin{pmatrix} 0 & 2W_D \\ -I & -4W_{D^{1/2}} \end{pmatrix},$$

avec $W_{D^{1/2}}$ la matrice décrite par (1) avec la matrice D au lieu de D_2.

```
In  [21]: W = 2 * Wd(D)
          Wsqrt = -4 * Wd(np.sqrt(D))
          B = np.block([[np.zeros((N, N)), W], [-np.eye(N), Wsqrt]])
          max(linalg.eigvals(B))

Out [21]: np.complex128(2.8576381702999076+0j)
```

2 Exercices théoriques

2.1 AFTD des données `Mutations`

On reprend le problème de la représentation des données `Mutations`.

18 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  [22]: 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 [22]: np.float64(0.5000000000000001)
```

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 $1/2$.

19 Peut-on choisir la matrice M de telle sorte que $\text{Stress} = mI_{b_1}$?

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  [23]: 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 [23]: 0

In  [24]: n * inertia / stress

Out [24]: np.float64(1.0000000000000002)
```

2.2 Classification ascendante hiérarchique

20 On considère le tableau individus-variables suivant :

$$X = \begin{pmatrix} 2 & 1 \\ 1 & 2 \\ 3 & 5 \\ 5 & 2 \\ 7 & 3 \end{pmatrix}$$

Calculer la matrice de distances euclidiennes associée ; faire une CAH avec lien minimum, puis lien maximum.

On obtient la matrice de distances euclidiennes suivante :

$$D(X) = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & & & & \\ \sqrt{2} & 0 & & & \\ \sqrt{17} & \sqrt{13} & 0 & & \\ \sqrt{10} & \sqrt{16} & \sqrt{13} & 0 & \\ \sqrt{29} & \sqrt{37} & \sqrt{20} & \sqrt{5} & 0 \end{bmatrix} \end{array}.$$

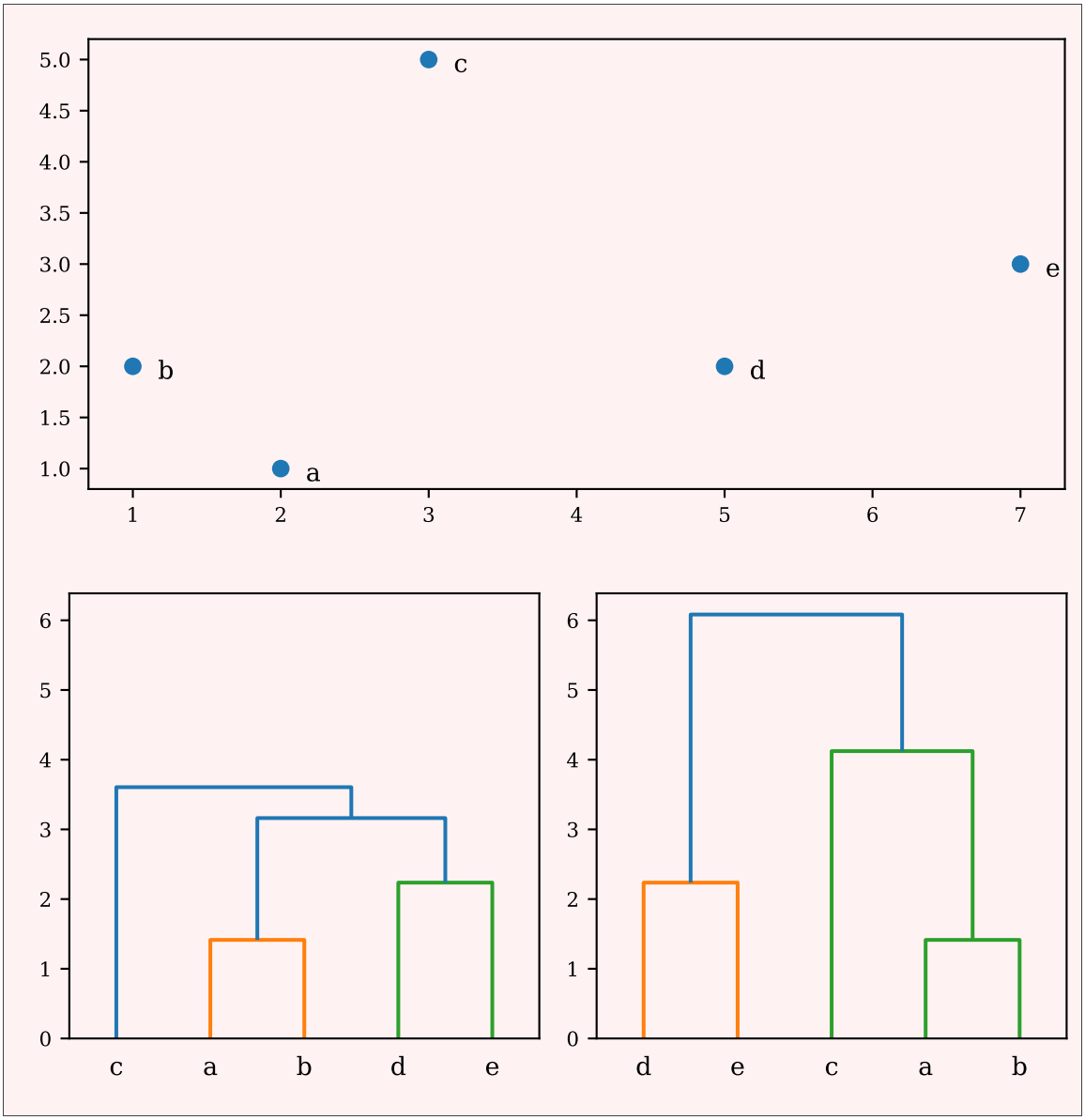
On peut ensuite calculer les tableaux successifs obtenus en fusionnant les groupes, en comparant les valeurs des colonnes ou lignes correspondant aux éléments fusionnés ; pour le lien minimum :

$$\underbrace{\begin{array}{c} \\ \{a,b\} \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{cccc} \{a,b\} & c & d & e \end{array} \\ \begin{bmatrix} 0 & & & \\ \sqrt{13} & 0 & & \\ \sqrt{10} & \sqrt{13} & 0 & \\ \sqrt{29} & \sqrt{20} & \sqrt{5} & 0 \end{bmatrix} \end{array}}_{\text{fusion de } a \text{ et } b}, \underbrace{\begin{array}{c} \\ \{a,b\} \\ c \\ \{d,e\} \end{array} \begin{array}{c} \begin{array}{ccc} \{a,b\} & c & \{d,e\} \end{array} \\ \begin{bmatrix} 0 & & \\ \sqrt{13} & 0 & \\ \sqrt{10} & \sqrt{13} & 0 \end{bmatrix} \end{array}}_{\text{fusion de } d \text{ et } e}, \underbrace{\begin{array}{c} \\ \{a,b,d,e\} \\ c \end{array} \begin{array}{c} \begin{array}{cc} \{a,b,d,e\} & c \end{array} \\ \begin{bmatrix} 0 & \\ \sqrt{13} & 0 \end{bmatrix} \end{array}}_{\text{fusion de } \{a,b\} \text{ et } \{d,e\}};$$

pour le lien maximum :

$$\underbrace{\begin{array}{c} \\ \{a,b\} \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{cccc} \{a,b\} & c & d & e \end{array} \\ \begin{bmatrix} 0 & & & \\ \sqrt{17} & 0 & & \\ \sqrt{16} & \sqrt{13} & 0 & \\ \sqrt{37} & \sqrt{20} & \sqrt{5} & 0 \end{bmatrix} \end{array}}_{\text{fusion de } a \text{ et } b}, \underbrace{\begin{array}{c} \\ \{a,b\} \\ c \\ \{d,e\} \end{array} \begin{array}{c} \begin{array}{ccc} \{a,b\} & c & \{d,e\} \end{array} \\ \begin{bmatrix} 0 & & \\ \sqrt{17} & 0 & \\ \sqrt{37} & \sqrt{20} & 0 \end{bmatrix} \end{array}}_{\text{fusion de } d \text{ et } e}, \underbrace{\begin{array}{c} \\ \{a,b,c\} \\ \{d,e\} \end{array} \begin{array}{c} \begin{array}{cc} \{a,b,c\} & \{d,e\} \end{array} \\ \begin{bmatrix} 0 & \\ \sqrt{37} & 0 \end{bmatrix} \end{array}}_{\text{fusion de } \{a,b\} \text{ et } c}.$$

On peut tracer les dendrogrammes correspondants :



21 Prouver les formules de récurrence de Lance & Williams, pour les trois critères d'agrégation du lien minimum, du lien maximum, et du lien moyen :

$$
\begin{aligned}
D_{\min}(A, B \cup C) &= \min\{D_{\min}(A, B), D_{\min}(A, C)\}, \\
D_{\max}(A, B \cup C) &= \max\{D_{\max}(A, B), D_{\max}(A, C)\}, \\
D_{\text{moy}}(A, B \cup C) &= \frac{n_B\, D_{\text{moy}}(A, B) + n_C\, D_{\text{moy}}(A, C)}{n_B + n_C}.
\end{aligned}
$$

On a

$$
\begin{aligned}
D_{\min}(A, B \cup C) &= \min\{d(i, i'),\ i \in A,\ i' \in B \cup C\}, \\
&= \min\big\{\min\{d(i, i'),\ i \in A,\ i' \in B\}, \min\{d(i, i'),\ i \in A,\ i' \in C\}\big\}, \\
&= \min\{D_{\min}(A, B), D_{\min}(A, C)\};
\end{aligned}
$$

on peut appliquer exactement le même raisonnement pour prouver la seconde formule de récurrence : les deux opérateurs min et max sont tous deux distributifs par rapport à l'union.

Pour le lien moyen, on a

$$\begin{aligned} D_{\text{moy}}(A, B \cup C) &= \frac{1}{n_A\, n_{B\cup C}} \sum_{i\in A, i'\in B\cup C} d(i,i'), \\ &= \frac{1}{n_A\,(n_B+n_C)} \left(\sum_{i\in A, i'\in B} d(i,i') + \sum_{i\in A, i'\in C} d(i,i') \right), \\ &= \frac{1}{n_B+n_C} \left(\frac{n_B}{n_A\, n_B} \sum_{i\in A, i'\in B} d(i,i') + \frac{n_C}{n_A\, n_C} \sum_{i\in A, i'\in C} d(i,i') \right), \\ &= \frac{n_B\, D_{\text{moy}}(A,B) + n_C\, D_{\text{moy}}(A,C)}{n_B+n_C}. \end{aligned}$$