BUT1 – S.A.E. S2
ALGORITHMES DE PARTITIONNEMENT DE DONNÉES

IUT DE NANTES – DÉPARTEMENT D'INFORMATIQUE – 2021–2022

Contact : Johan Leray, Email : johan.leray@univ-nantes.fr

Le but de cette partie de SAÉ est de développer des algorithmes de partitionnement, basés sur deux méthodes distinctes, d'en étudier les avantages et les inconvénients, et de les utiliser dans un contexte d'étude de données géographiques.

Les deux premières parties de ce document sont consacrées à la présentation de ces deux méthodes de partitionnement. Dans la dernière partie, on présentera les attendus de la SAÉ et les contraintes de votre travail.

TABLE DES MATIÈRES

PROBLÉMATIQUE

Considérons une population $\mathcal{P}$ de cardinal fini, et n variables statistiques X_i quantitatives définies sur cette population, c'est-à-dire que pour tout $i \in [\![1, n]\!]$, on a une application $X_i : \mathcal{P} \to \mathbb{R}$. Pour chaque individu de cette population, c'est-à-dire pour tout $i \in \mathcal{P}$, on associe donc un point de $\mathbb{R}^n$ donné par

$$i \in \mathcal{P} \longmapsto \big(X_1(i), X_2(i), \dots, X_n(i)\big) \in \mathbb{R}^n.$$

Pour toute la suite, on considère $\mathcal{S} \subset \mathbb{R}^n$ le sous-ensemble fini de $\mathbb{R}^n$ défini par

$$\mathcal{S} \coloneqq \left\{ \big(X_1(i), X_2(i), \dots, X_n(i)\big) \in \mathbb{R}^n \mid i \in \mathcal{P} \right\}.$$

Par exemple, on pourra considérer $\mathcal{P}$, un ensemble de locations saisonnières de Loire-Atlantique et le couple de variables $(X, Y) : \mathcal{P} \to \mathbb{R}^2$ qui, à une location $i \in \mathcal{P}$, associe ses coordonnées géographiques, c'est-à-dire la longitude et la latitude de sa position.

Le but des algorithmes que l'on vous propose d'étudier dans ce document est de répondre au problème suivant.

PROBLÈME – PROBLÈME DE PARTITIONNEMENT – *Étant donné un ensemble fini $\mathcal{S} \subset \mathbb{R}^n$, on cherche une partition $\mathcal{S}_1, \dots, \mathcal{S}_K$ de $\mathcal{S}$, c'est-à-dire K sous-ensembles (que l'on appellera* classes *de $\mathcal{S}$ tels que pour*

tous $i, j \in [\![1, K]\!]$ *avec* $i \neq j$, $\mathcal{S}_i \cap \mathcal{S}_j = \varnothing$ *et tels que la réunion des* $\mathcal{S}_j$ *est* $\mathcal{S}$ *tout entier, i.e.*

$$\mathcal{S} = \bigcup_{k=1}^{K} \mathcal{S}_k.$$

De plus, on souhaite que chaque élément de $\mathcal{S}_i$ *soit "plus similaire" aux autres éléments de* $\mathcal{S}_i$ *qu'aux éléments des autres* $\mathcal{S}_j$ *pour* $i \neq j$.

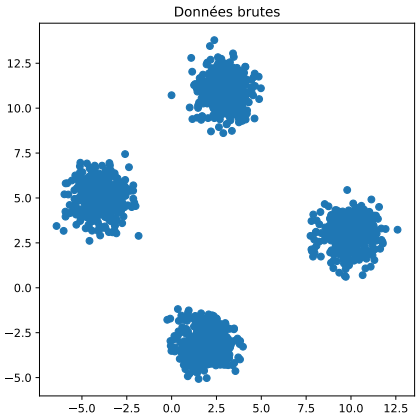
L'idée est donc de diviser la population totale en sous-populations (en classes) présentant des caractéristiques communes. Par la suite, pour toutes les applications, on fera apparaître ces classes graphiquement en leur appliquant à chacune une couleur différente.

Partie 1 – Une première méthode de partitionnement de données

Dans cette section, on présente un premier algorithme qui permet de partitionner nos données en K classes, avec $K \in \mathbb{N}^*$, un nombre entier fixé par l'utilisateur. Dans la suite, on considère toujours un ensemble $\mathcal{S}$ de points $P \in \mathbb{R}^n$.

Par exemple, on pourra considérer un ensemble de points du plan $\mathbb{R}^2$, comme représenter en Figure 1. En visualisant ces données, on distingue clairement quatre groupes de points. On aimerait donc que notre algorithme retrouve ces 4 groupes que l'on voit sur la représentation graphique.

Figure 1 – Visualisation des données



On va donc mettre en place un algorithme qui va séparer cet ensemble $\mathcal{S}$ en K sous-ensembles $\mathcal{S}_1, \ldots, \mathcal{S}_K$ tels que, pour tous $i, j \in [\![1, k]\!]$, avec $i \neq j$, $\mathcal{S}_i \cap \mathcal{S}_j = \varnothing$ et tels que

$$\bigcup_{i=1}^{K} \mathcal{S}_i = \mathcal{S}.$$

Ainsi, dans la figure 1, on souhaiterait un algorithme capable de distinguer les quatre sous-ensembles de points. Avant de donner une description de celui-ci, on rappelle que la *distance euclidienne* entre deux points $P = (p_1, \ldots, p_n)$ et $Q = (q_1, \ldots, q_n)$ de $\mathbb{R}^n$ est définie comme suit :

$$d(P, Q) := \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}.$$

Dans la suite, on considérera plutôt le carré de cette distance :

$$\mathrm{dist}(P, Q) := (p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2.$$

L'idée de cette méthode est la suivante :

1. on choisit $m_1, \dots, m_K$, K points de notre ensemble $\mathcal{S}$;
2. pour tous les $i \in [\![1, K]\!]$, on retient dans $\mathcal{S}_i$ tous les points de $\mathcal{S}$ qui sont le plus proche de m_i ;
3. pour chaque $\mathcal{S}_i$, on calcule le point moyen de cet ensemble que l'on retient dans m_i ;
4. on retourne à 2 tant que notre cas d'arrêt n'est pas atteint.

1.1. **L'algorithme**. On fournit ici un algorithme pour appliquer numériquement cette méthode de partitionnement.

Algorithme 1 : Méthode A de partitionnement

Entrées : une liste de points $\mathcal{S} \subset \mathbb{R}^n$
une liste de K points $[m_1, \dots, m_K]$
un nombre d'itérations maximal N

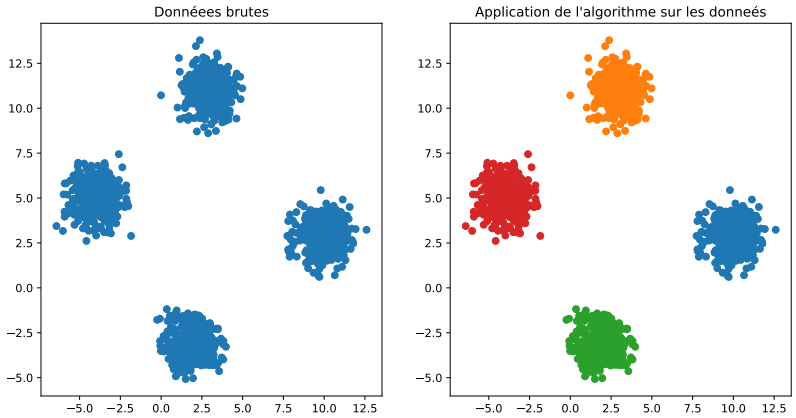
Sortie : une liste de K listes de points $[\mathcal{S}_1, \dots, \mathcal{S}_k]$
la liste $m_1, \dots, m_K$ de K points où m_i est le point moyen de $\mathcal{S}_i$

On initialise $\mathcal{S}_1, \dots, \mathcal{S}_K$ par des listes vides.
`/* Boucle principale */`
pour $i \leftarrow 1$ **à** N **faire**
  pour $k \leftarrow 1$ **à** K **faire**
    `/* on met dans` $\mathcal{S}_k$ `les points les plus proches de` m_k `*/`
    $\mathcal{S}_k \leftarrow \left\{ p \in \mathcal{S} \;\middle|\; \forall j \in [\![1, K]\!], \operatorname{dist}(p, m_k) \leqslant \operatorname{dist}(p, m_j) \right\}$
  pour $k \leftarrow 1$ **à** K **faire**
    `/* on affecte dans` m_k `le point moyen de l'ensemble` $\mathcal{S}_k$ `*/`
    si $\operatorname{Card}(\mathcal{S}_k) \neq \emptyset$ **alors**
      $m_k \leftarrow \frac{1}{\operatorname{Card}(\mathcal{S}_k)} \sum_{p \in \mathcal{S}_k} p$
    sinon
      on tire de nouveau m_k au hasard dans $\mathcal{S}$

Pour les premières applications, on choisira la liste $[m_1, \dots, m_K]$ en tirant au hasard K points distincts de $\mathcal{S}$. L'utilisateur n'aura ainsi qu'à choisir le nombre de classes, c'est-à-dire le nombre K. Par la suite, nous verrons une méthode permettant de choisir ces K premiers points.

En choisissant un nombre de classes égal à 4, ainsi que 25 itérations de la boucle principale, on obtient sur notre premier jeu de données quatre sous-ensembles de points que l'on colore de différentes couleurs et que l'on représente en Figure 2. Dans ce cas, on est capable de discriminer informatiquement les 4 sous-ensembles que l'on voyait apparaître graphiquement.

FIGURE 2 – Visualisation du rendu de la méthode A de partitionnement



CONSIGNE D'IMPLÉMENTATION

Vous implémenterez cet algorithme en une fonction python `MethodA` qui devra respecter la syntaxe suivante. Cette fonction pourra faire appel à des fonctions auxiliaires qui seront définies en amont de celle-ci.

```
def MethodA(data, ListPtsInit, NbMaxIter):
    # data : une matrice à m lignes et n colonnes à chaque ligne
    # correspond à un point de Rn
    # ListPtsInit : une liste de points pour initialiser l'algorithme
    # NbMaxIter : un entier fixant le nombre maximal
    # d'itération de la boucle principale
    # ListOfClasses : une liste de classes (listes de points)
    # ListOfCentroids : la liste des points moyens de chaque classe
    return ListOfClasses, ListOfCentroids
```

La fonction `MethodA` devra être accompagnée d'une fonction python `MethodAExample()` qui affichera une illustration graphique du partitionnement d'un jeu de données.

```
def MethodAExample(Title):
    # Titre : chaîne de caractères
    # sauvegarde une illustration graphique de la MethodA dans
    # un fichier Title.pdf
```

On peut améliorer la rapidité d'exécution de cet algorithme en ajoutant un cas d'arrêt pour la boucle principale. En effet, si après une itération de cette boucle, aucun des ensembles $\mathcal{S}_k$ n'a été modifié par rapport à l'itération précédente, alors il est inutile de continuer l'algorithme : le résultat ne sera pas modifié aux itérations suivantes de cette boucle.

1.2. **Limitations de cette méthode**. Comme vous avez pu le constater, cette méthode de partitionnement se met facilement en place. Cependant, elle possède un certain nombre de problèmes majeurs :

— on doit fixer le nombre de partitions à l'avance ;

— elle est assez dépendante du choix des K premiers points, comme vous l'aurez peut-être constaté lors de vos tests ;

— elle n'est pas efficace dans tous les cas, car cette méthode revient à diviser l'espace par des segments et des demi-droites.

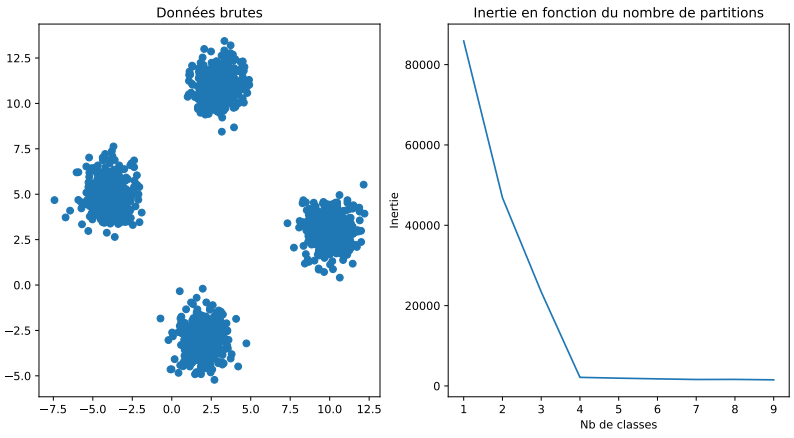
1.2.1. *Détermination du nombre de partitions.* Pour estimer le nombre de partitions adapté à notre jeu de données, on peut utiliser la méthode empirique suivante, appelée "la méthode du coude". Pour cela, on va appliquer notre algorithme de partitionnement sur nos données en faisant varier le nombre de partitions et regarder évoluer une quantité associée en fonction du nombre.

Fixons $K \in \mathbb{N}^*$ le nombre de partitions et notons $\mathcal{S}_1, \dots, \mathcal{S}_K$ les K sous-ensembles composant la partition de notre ensemble $\mathcal{S}$ renvoyés par notre algorithme. À une telle partition, on lui associe *son inertie*, qui est un nombre que l'on détermine de la manière suivante. À chaque sous-ensemble $\mathcal{S}_k$, on associe son point moyen s_k. L'inertie de la partition $\mathcal{S}_1, \dots, \mathcal{S}_K$ est le nombre réel positif défini par

$$\text{Inertie}(\mathcal{S}_1, \dots, \mathcal{S}_K) \coloneqq \sum_{k=1}^{K} \sum_{p \in \mathcal{S}_k} \text{dist}(p, s_k).$$

Lorsque le nombre de classes de notre partition augmente, l'inertie aura tendance à diminuer. On peut tracer le graphique de l'évolution de l'inertie en fonction du nombre de classes. Lorsque cette diminution est moins forte, c'est-à-dire que graphiquement "la courbe fait un coude", on choisira l'abscisse de ce coude comme notre nombre de classes. Par exemple, on constate en Figure 3 que le coude de la courbe est en 4 et qu'il correspond au nombre de groupes de points que l'on observe sur la représentation graphique des données.

FIGURE 3 – Méthode "du coude"



Consigne d'implémentation

Vous implémenterez cette méthode graphique de détermination du nombre idéal de classes en une fonction python `MethodAElbow` qui devra respecter la syntaxe suivante. Cette fonction affichera la représentation graphique des données et le graphique d'évolution de l'inertie en fonction du nombre de nombre de classes. Elle pourra faire appel à des fonctions auxiliaires qui seront définies en amont de celle-ci.

```
def MethodAElbow(data, NbClassMax, Titlee):
    # data : une matrice à m lignes et n colonnes à chaque ligne
    # correspond à un point de Rn
    # NbClassMax : un entier égal au nombre de classes maximal
    # Titre : chaîne de caractères
    # Sauvegarde une illustration sous le nom Title.pdf
```

1.2.2. *Amélioration de l'initialisation.* Comme vous l'avez constaté sur vos différents tests, même sur des données test, il arrive que ces algorithmes ne donnent pas le résultat voulu. Cela est notamment dû à l'initialisation des points $m_1, \dots, m_K$. Nous allons donc présenter un algorithme qui va nous permettre d'optimiser l'initialisation de ces points et mieux, de ne pas renseigner à l'avance le nombre de classes de notre partition.

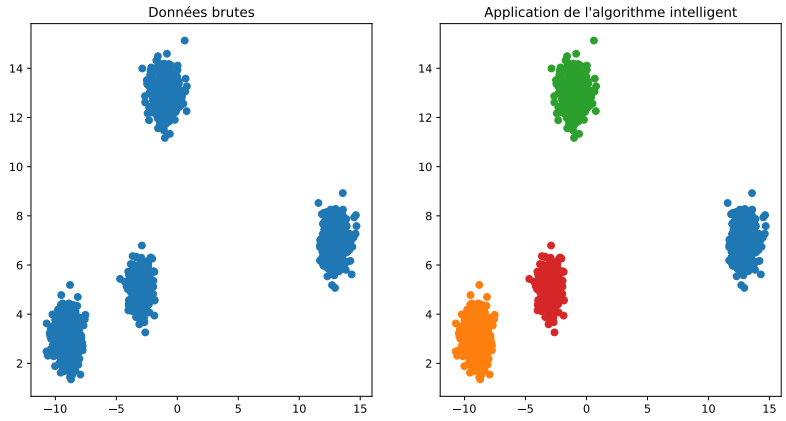
Pour cela, on va itérer un certain nombre de fois le partitionnement en 2 classes de nos données $\mathcal{S}$: une première classe donnée par les points les plus proches de m_{iso}, le point moyen de $\mathcal{S}$; une seconde classe $\mathscr{C}$ déterminée par l'algorithme de partitionnement en partant des points les plus proches de m_{far}, le point de $\mathcal{S}$ le plus loin de m_{iso}. Une fois l'algorithme, on réitère le procédé sur l'ensemble $\mathcal{S} \setminus \mathscr{C}$ sans modifier le point m_{iso}. On construit comme cela un certain nombre de classes $\mathcal{S}_1, \dots, \mathcal{S}_M$. Pour toutes les classes $\mathcal{S}_j$ de cardinal dépassant un certain seuil fixé par l'utilisateur, on calcule son point moyen. On obtient ainsi K points $m_1, \dots, m_K$ avec $K \leqslant M$ que l'on va utiliser pour initialiser notre premier algorithme.

Algorithme 2 : Algorithme d'initialisation de notre méthode A

Entrées : une liste de points $\mathcal{S} \subset \mathbb{R}^n$
un nombre entier positif T qui est un seuil
Sortie : une liste $\mathscr{L} = [m_1, \dots, m_k]$ avec $k \leqslant K_{\max}$
$m_{\text{fixe}} \longleftarrow \frac{1}{\text{Card}(\mathcal{S})} \sum_{p \in \mathcal{S}} p$ `/* Calcul du point moyen */`
tant que $\text{Card}(\mathcal{S}) > T$ **faire**
  $m_{\text{temp}} \longleftarrow$ le point de $\mathcal{S}$ le plus loin de m_{fixe}
  tant que *Cas d'arrêt* **faire**
    $\mathcal{S}_{\text{temp}} \longleftarrow \left\{ p \in \mathcal{S} \;\middle|\; \forall j \in [\![1, K]\!], \text{dist}(p, m_{\text{temp}}) \leqslant \text{dist}(p, m_{\text{fixe}}) \right\}$
    $m_{\text{temp}} \longleftarrow \frac{1}{\text{Card}(\mathcal{S}_{\text{temp}})} \sum_{p \in \mathcal{S}_{\text{temps}}} p$
  si $\text{Card}(\mathcal{S}_{\text{temp}}) > T$ **alors**
    on ajoute m_{temp} à $\mathscr{L}$
  $\mathcal{S} \longleftarrow \mathcal{S} \setminus \mathcal{S}_{\text{temp}}$

On peut noter ici que le seuil est fixé de manière arbitraire. Cet algorithme nous permet d'initialiser notre algorithme 1, et nous pouvons ainsi nous passer de demander à l'utilisateur de spécifier le nombre de classes pour la partition attendu (on lui demande seulement de fixer le seuil de tolérance pour l'algorithme). On peut voir une illustration du résultat de la combinaison des algorithmes 1 et 2 en Figure 4.

FIGURE 4 – Illustration de l'algorithme "intelligent"



CONSIGNE D'IMPLÉMENTATION

Vous implémenterez cet algorithme en une fonction python `MethodAIntel` qui devra respecter la syntaxe suivante. Cette fonction pourra faire appel à des fonctions auxiliaires qui seront définies en amont de celle-ci.

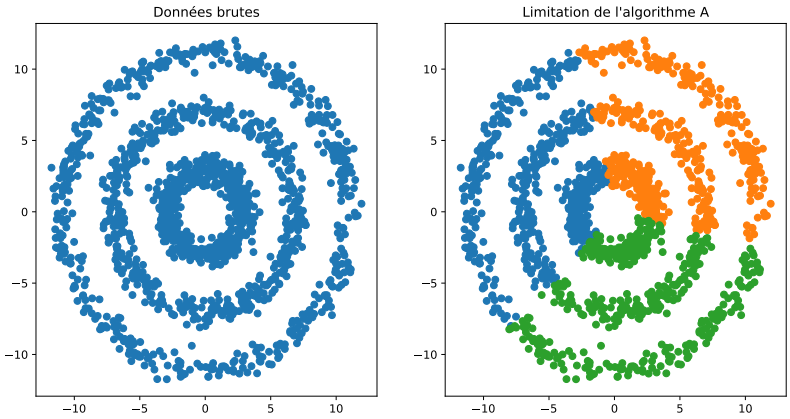
```
def MethodAIntel(data, NbMaxIter):
    # data : une matrice à m lignes et n colonnes à chaque ligne
    # correspond à un point de Rn
    # NbMaxIter : un entier fixant le nombre maximal d'itération
    #  de la boucle principale
    # ListOfClasses : une liste de classes (listes de points)
    # ListOfCentroids : la liste des points moyens de chaque classe
    return ListOfClasses, ListOfCentroids
```

La fonction `MethodAIntel` devra être accompagnée d'une fonction python `MethodAIntelExample` qui affichera une illustration graphique du partitionnement d'un jeu de données.

```
def MethodAIntelExample(Title):
    # Title : chaîne de caractères
    # Sauvegarde une illustration graphique de MethodAIntel dans
    # un fichier Title.pdf
```

1.2.3. *Un dernier exemple de limitation.* Une dernière limitation de cette méthode provient de la forme des données que l'on souhaite exploiter. En effet, dans certains cas, cette méthode ne peut tout bonnement être efficace, comme on peut le constater sur l'exemple en Figure 5.

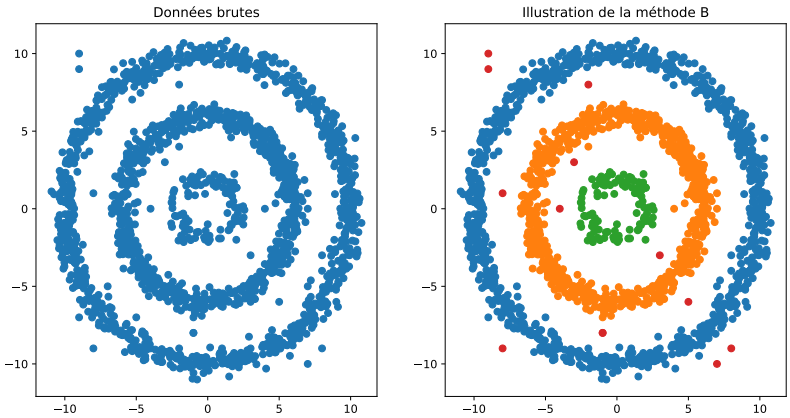
FIGURE 5 – Limitation de la méthode A de partitionnement



PARTIE 2 – UNE SECONDE MÉTHODE DE PARTITIONNEMENT

Dans cette seconde partie, on propose une nouvelle méthode de partitionnement de données, qui est complémentaire de la première. Cette méthode de partitionnement va utiliser la densité du nuage de points que l'on étudie pour former les classes de la partition. Essentiellement, on va étudier le voisinage de chaque point du jeu de données (c'est-à-dire un disque autour du point) et étudier le nombre de points que ce voisinage contient. Avant de décrire cette méthode, on peut constater sur l'exemple où la méthode A montrait ses limites, cette seconde méthode de partitionnement fonctionne assez bien (cf. Figure 6).

FIGURE 6 – Illustration de la méthode B de partitionnement



2.1. **Présentation de la méthode.** L'algorithme que nous allons présenter demande à l'utilisateur de spécifier deux quantités. Pour commencer, l'utilisateur spécifiera un nombre réel positif $\varepsilon \in \mathbb{R}_+^*$ qui correspond à la taille des voisinages des points, c'est-à-dire au rayon du disque autour de chacun des points. L'utilisateur spécifiera également un nombre entier $N_{\min} \in \mathbb{N}^*$, qui correspond au nombre de points minimal de points dans un voisinage d'un autre afin de considérer qu'ils appartiennent à la même classe. Ces deux nombres permettent donc une mesure de la densité des classes.

Cette méthode de partitionnement de données ne demande pas à l'utilisateur de spécifier à l'avance le nombre de classes qu'il souhaite avoir dans sa partition, ce qui est un avantage par rapport à la méthode A. Cependant, les paramètres ε et $N_{\min}$ peuvent s'avérer difficiles à estimer, car ils sont intrinsèquement liés à la topologie de l'espace à partitionner.

On présente ci-dessous un algorithme correspondant à notre méthode de partitionnement. L'algorithme principal va parcourir l'ensemble des points de notre jeu de données. Lorsqu'un point aura été visité, il sera *marqué*, afin de spécifier que l'algorithme a déjà considéré ce point. Par exemple, on pourra, à chaque point, ajouter une coordonnée booléenne qui vaudra `False` si le point n'est pas marqué, et `True` sinon.

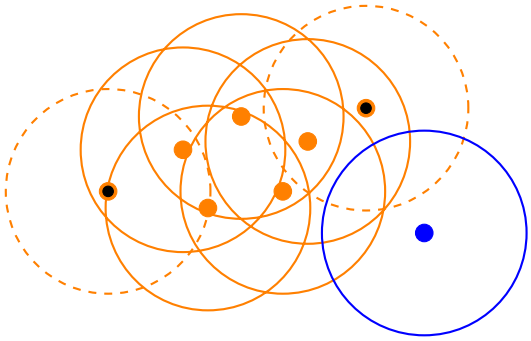
L'algorithme va construire les différentes classes de notre partition les unes après les autres. Une de ces classes sera particulière, car elle contiendra, à la fin de l'algorithme, tous les points isolés, c'est-à-dire qui se situe dans une zone trop peu dense en points. Dans la description de l'algorithme, on désignera cette classe par $\mathcal{S}_{\text{bruit}}$.

Pour construire une classe, on va procéder comme suit : on choisit un premier point P qui n'appartient à aucune classe

— si P possède moins de $N_{\min}$ points à une distance inférieure à ε, alors on met P dans la classe $\mathcal{S}_{\text{bruit}}$: c'est un *point aberrant* ;

— si P possède assez de points voisins, c'est-à-dire s'il possède plus de $N_{\min}$ points à une distance inférieure à ε, alors on lui attribue une nouvelle classe ; P est actuellement un *point central* et les points qui lui sont à une distance inférieure à ε sont alors des *points frontières* ; on étend ensuite cette classe de proche en proche : pour tout point frontière n'appartenant à aucune classe, on lui attribue la classe C et s'il possède plus de $N_{\min}$ points ε-proche, on ajoute ces points dans l'ensemble des points frontières à étudier.

FIGURE 7 – Illustration des trois types de points avec $N_{\min} = 3$



Les points oranges sont les points centraux de la classe (ils possèdent assez de points voisins) ; les points noirs sont les points frontières de la classe (ils n'ont pas assez de points voisins mais sont dans le voisinage d'un point de la classe) ; le point bleu est un point aberrant, qui ne fait pas partie de la classe.

2.2. **L'algorithme**. On fournit ci-dessous l'algorithme correspondant à cette méthode de partitionnement en le décomposant en un algorithme principal et deux fonctions. On illustre le résultat de celui-ci en Figure 6.

Algorithme 3 : Algorithme de la méthode B de partitionnement

Entrées : une liste de points $\mathcal{S} \subset \mathbb{R}^n$
un nombre réel positif ε
un nombre entier positif $N_{\min}$
Sortie : une liste de listes de points $[\mathcal{S}_1, \ldots, \mathcal{S}_K]$

$C \longleftarrow 0$ `/* Compteur du nombre de classes */`

pour *tout point P de $\mathcal{S}$ non marqué* **faire**
  marquer le point P
  $PtsVois \longleftarrow$ `Voisinage`$(\mathcal{S}, P, \varepsilon)$
  si Card$(PtsVois) < N_{\min}$ **alors**
    Ajouter P à $\mathcal{S}_{\text{bruit}}$
  sinon
    $C \longleftarrow C + 1$
    `ExtensionClasse`$(\mathcal{S}, P, PtsVois, C, \varepsilon, N_{\min})$

Algorithme 4 : Fonctions pour la méthode B de partitionnement

Fonction `Voisinage`$(\mathcal{S}, Pt, \varepsilon)$:
  Renvoie les points de $\mathcal{S}$ qui sont à une distance inférieure à ε de P

Fonction `ExtensionClasse`$(\mathcal{S}, Pt, PtsVois, C, \varepsilon, N_{\min})$:
  `/* Modifie la classe` C `*/`
  Ajouter Pt à la classe C
  pour $p \in PtsVois$ **faire**
    si *p n'est pas marqué* **alors**
      marquer p
      $PtsVoisPrime \longleftarrow$ `Voisinage`$(\mathcal{S}, p, \varepsilon)$
      si Card$(PtsVoisPrime) \geqslant N_{\min}$ **alors**
        $PtsVois \longleftarrow PtsVois \cup PtsVoisPrime$
    si *p n'appartient à aucune classe* **alors**
      Ajouter p à la classe C

En choisissant de manière adéquate les nombres ε et $N_{\min}$, cette méthode permet de faire une partition des données en ensembles de même densité. L'algorithme montrera ses limites avec des données qui ne présentent pas de densité uniforme sur les différentes partitions que l'on voudrait détecter.

Consigne d'implémentation

Vous implémenterez cet algorithme en une fonction python `MethodB` qui devra respecter la syntaxe suivante. Cette fonction pourra faire appel à des fonctions auxiliaires qui seront définies en amont de celle-ci.

```
def MethodB(data, epsilon, Nmin):
    # data : une matrice à m lignes et n colonnes
    # epsilon : un flottant
    # Nmin : un entier positif
    # ListOfMarkedPts : la liste des points marqués avec leur
    # numéro de classe. Un élément de ListOfMarkerPts est
    # [p1,...,pn,True,C] avec [p1,...pn] est un element de data
    # et C correspond à son numéro de classe
    return ListOfMarkedPts
```

La fonction `MethodB` devra être accompagnée d'une fonction python `MethodBExample` qui sauvegardera une illustration graphique du partitionnement d'un jeu de données.

```
def MethodBExample(Title):
    # Title : chaîne de caractères
    # Sauvegarde une illustration graphique de MethodB dans un
    # fichier Title.pdf
```

Pour finir, on compare en Figure 8 et Figure 9, les résultats des deux méthodes sur deux exemples de données. On constate qu'en fonction du type de données et de l'organisation de celle-ci, on préférera l'une ou l'autre méthode pour les partitionner.

Consigne d'implémentation

Vous implémenterez une fonction python `CompareMethods` qui affichera des comparaisons graphiques des deux différentes méthodes.

```
def CompareMethods(Title):
    # Title : chaîne de caractères
    # sauvegarde des graphiques illustrant les différences de résultat
    # des méthodes A et B dans un fichier Title.pdf
```

FIGURE 8 – Comparaison des deux méthodes : la méthode A est plus efficace

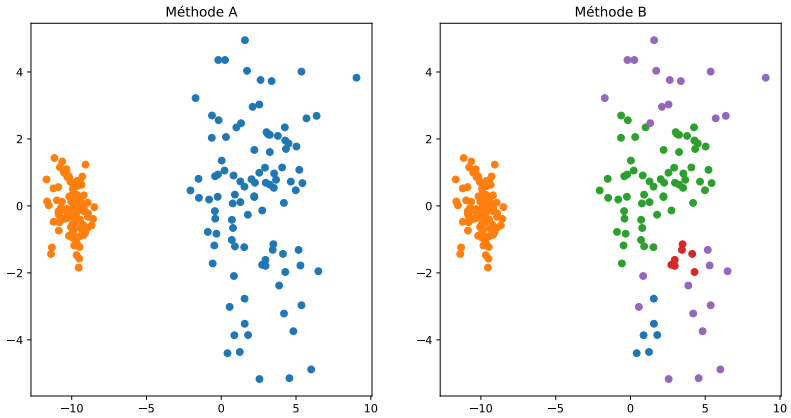
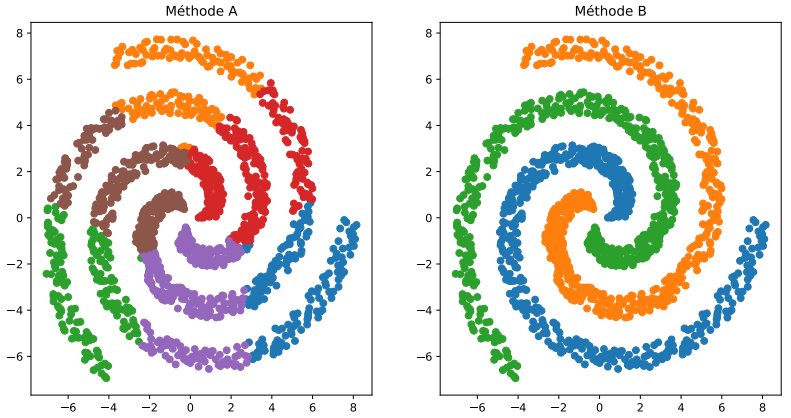


FIGURE 9 – Comparaison des deux méthodes : la méthode B est plus efficace



PARTIE 3 – ATTENDUS DE LA SAÉ

Votre travail consistera en plusieurs tâches.

1. Vous devrez comprendre les différents algorithmes présentés ici en les implémentant en python. Afin de tester leur bon fonctionnement, vous pourrez utiliser les données tests fournies sur MADOC. Vous déposerez un fichier python `SAE_partitionnement.py` contenant vos différentes fonctions. **Vous porterez notamment une attention toute particulière aux différentes consignes d'implémentations : le non-respect de celles-ci sera sanctionné.**
2. Vous devrez notamment être capable d'expliquer en détail le fonctionnement de l'une des deux méthodes (au choix) lors de la soutenance finale. Vous vous appuierez notamment sur des applications de celle-ci sur des exemples bien choisis pour en illustrer les points forts, mais également les limitations. Par exemple, vous pourriez programmer une animation illustrant les différentes itérations faites par l'algorithme.
3. Vous appliquerez l'une ou ces méthodes à des données géographiques étudiées dans cette SAÉ. Vous pourrez vous inspirer de l'exemple ci-dessous mais vous pouvez également choisir d'autres données, d'autres problématiques.

Exemple On considère les données géographiques étudiées dans la partie *Base de données* de cette Saé. On cherche à aire apparaître les secteurs du département qui contiennent une densité assez élevé de locations saisonnières. Pour cela, on peut appliquer notre méthode B comme en Figure 10. On constate aisément que cette manière de partitionner les données n'est pas entièrement satisfaisante. On peut donc filtrer l'affichage des classes, en retirant le bruit (la classe qui ne contient que les locations qui se trouvent dans une zone peu dense) et les classes trop petites. On illustre le résultat obtenu en Figure 11.

Figure 10 – Partitionnement des données de locations saisonnières

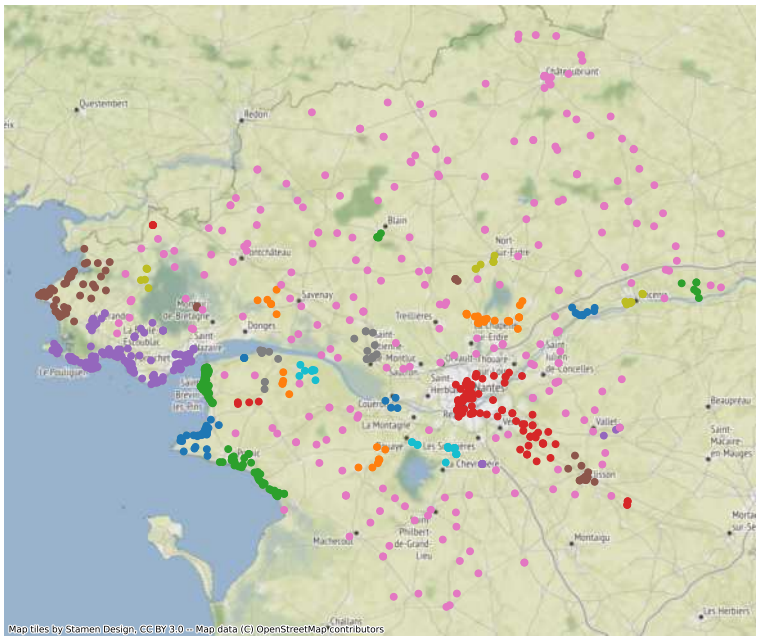


Figure 11 – Partitionnement des données de locations saisonnières avec filtres