Sorbonne Université **M1 Mathématiques**
2023–2024 **MU4M056 : Programmation en C++**

TP noté 1 : Clustering de Voronoï

L'objectif de ce TP est d'implémenter en C++ un exemple de classification de points dans l'espace selon des cellules de Voronoï, à implémenter sous la forme d'une classe, et à tester la classe sur un jeu de données.

On veillera à inclure dans chaque fichier toutes les bibliothèques nécessaires et les options de compilation nécessaires. **Il est également impératif de mettre en commentaire, dans tous les fichiers, les nom, prénom et n° d'étudiant.**

1 Introduction

1.1 Description mathématique

On se donne des points $(c_k)_{0\leq k<K}$ de $\mathbb{R}^2$, appelés *centres* ainsi qu'une distance d sur $\mathbb{R}^2$. Ces points permettent de diviser tout l'espace en K régions $(R_k)_{0\leq k<K}$ de telle sorte que $x \in R_k$ si et seulement si, pour tout $l \neq k$, $d(x, c_k) \leq d(x, c_l)$, autrement dit $x \in R_k$ si et seulement si x est plus proche de c_k que de n'importe quel autre centre.

Étant donné un ensemble fini de points P de $\mathbb{R}^2$, le but de ce TP est de construire un algorithme qui permette de partitionner P en K ensembles $(P_k)_{0\leq k<K}$ ave $P_k \subset R_k$. Cela revient, pour chaque k, à regrouper les points de P les plus proches de c_k dans un ensemble P_k. C'est ce qu'on appelle du *clustering*.

1.2 Structures et classes en C++

Afin de décrire les points de $\mathbb{R}^2$, nous introduisons la structure suivante, dans laquelle *tout vous est donné* :

```
struct Point2D {
   double x;
   double y;
   double dist_2(Point2D u) const { return (x-u.x)*(x-u.x)+(y-u.y)*(y-u.y); }
   // usage:    p1.dist_2(p2) donne la distance entre p1 et p2.
   bool operator<(Point2D u) const {
      if( x != u.x) return (x<u.x); else return (y<u.y);  }
   // vous n'aurez pas à l'utiliser: c'est nécessaire pour std::set ci-dessous.
};
```

Chaque cluster de points sera décrit par la donnée d'un centre c (un point de $\mathbb{R}^2$) et les points proches de ce centre donnés sous la forme d'un ensemble de points :

```
struct Cluster {
        Point2D center;//centre du cluster c
        std::set<Point2D> close_points;//ens. des points x_i proches du centre c
};
```

Remarque importante : **toute la documentation nécessaire pour `std::set` est résumée dans la dernière page du sujet.**

Le but du TP est de compléter et tester la classe suivante :

```
class Voronoi_classifier {
   private:
      std::vector< Cluster > clusters; //ensemble de cluster (c_k, P_k)
      int nb_points;//nombr de points de données déjà classés.
      // Invariants de classes:
      // (1) tous les points de clusters[i].close_points sont
      //    plus proches de clusters[i].center que de tout autre centre
      // (2) chaque point n'apparaît qu'une fois au total
      // (3) nb_points est le nombre total de points, i.e. la somme des tailles
      //     des clusters.
   public:
     // A COMPLETER
};
```

Toutes ces déclarations de classes sont écrites dans un fichier `voronoi_classif.hpp` et tous les codes associés seront écrits dans un fichier `voronoi_classif.cpp` .

2 Questions

1. Compléter tous les fichiers avec les inclusions de fichiers et de bibliothèques nécessaires.

2.1 Premières méthodes et premiers tests

Le but de cette section est de faire fonctionner le code suivant donné dans le fichier `test1.cpp` :

```
vector<Point2D> centers{ Point2D{0.2,0.2},Point2D{0.8,0.8},Point2D{0.3,0.7} };
Voronoi_classifier VC(centers);
VC.add_point(Point2D{0.5,0.});
VC.add_point(Point2D{0.75,0.6});
VC.add_point(Point2D{1.,0.2});
cout << "--- VC contient " << VC.nb_of_clusters() <<" clusters et "
     << VC.nb_of_data_points() << " points.\n";
vector<int> nbs = VC.cluster_sizes();
cout << "--- Les tailles des trois clusters sont :\n";
   for(int n : nbs) { cout << n << "\n"; }
cout << "--- Les clusters sont donnés par:\n" << VC << "\n";
VC.clear_clusters();
cout << "--- Après effacement, il y a " << VC.nb_of_clusters() << " centres et "
     << VC.nb_of_data_points() << " point.\n";
```

2. Écrire le code du constructeur de la ligne 2, qui prend comme argument un vecteur de points —les centres des clusters— et construit un classifieur dont tous les ensembles de points de données sont encore vides.

3. Écrire un accesseur `nb_of_clusters()` au nombre de clusters et un accesseur `nb_of_data_points()` au champ `nb_points` .

4. Écrire le code de la méthode `void add_point(Point2D p)` qui ajoute un nouveau point de donnée dans l'ensemble correspondant selon l'algorithme suivant :

- on cherche de quel centre le point `p` est le plus proche ;
- on insère le point `p` dans l'ensemble associé.

Pour cela, on pourra soit chercher l'entier `i` tel que `p` est le plus proche de `clusters[i].center`, soit utiliser `std::min_element` (qui prend un argument deux itérateurs et un opérateur de comparaison) de `<algorithm>` qui renvoie un itérateur sur la case qui correspond au plus petit élément (ici l'ordre est relié à la distance à `p`)

5. Ajouter un opérateur `<<` qui affiche le contenu d'un objet de la classe `Voronoi_classifier` avec `out << VC` (cf. ligne 13) sous la forme :

```
Center: ( 0.2, 0.2 )  //1er centre
      ( 0.5, 0 )      //   point du premier cluster
Center: ( 0.8, 0.8 )  //2ème centre
      ( 0.75, 0.6 )   //   points du
      ( 1, 0.2 )      //   deuxième cluster
Center: ( 0.3, 0.7 )  //3ème centre
      Vide            // le mot "Vide" si le cluster est vide
```

6. Ajouter une méthode

```
std::vector<int> Voronoi_classifier::cluster_sizes() const;
```

qui renvoie un vecteur ave la taille de chaque cluster, i.e. le cardinal de P_k, le nombre de points de données les plus proches de c_k. *Il y aura un bonus de point pour l'usage de* ***`std::transform`***.

7. Ajouter une méthode `clear_clusters()` qui vide les points de données de chaque ensemble P_k mais conserve les centres des clusters.

8. Tester à présent le code de `test1.cpp` qui doit fonctionner et comparer au résultat attendu (cf. fin du fichier `test1.cpp`).

Il y a trois situations possibles :

- soit tout fonctionne sans problème et vous pouvez passer à la suite
- soit ça ne marche pas, et dans ce cas :
  - soit vous insistez pour faire fonctionner ce qui ne marche pas (nous vous le conseillons fortement)
  - soit vous commentez les lignes de `test1.cpp` qui ne marchent pas afin d'avoir un fichier exécutable qui marche suffisamment et passez à la suite à vos risques et périls.

2.1.1 Méthodes plus élaborées

9. Écrire une méthode

```
void Voronoi_classifier::classify( std::istream & input, int nb);
```

qui lit `nb` points de données dans un flux de lecture `input` et les insère dans le bon cluster P_k. On supposera que le fichier derrière `input` est formaté de la manière suivante :

```
x1   y1
x2   y2
x3   y3
....
```

où les `x_i` et `y_i` sont des réels qui correspondent aux coordonnées d'un point de $\mathbb{R}^2$.

10. Écrire une méthode

```
std::vector<double> Voronoi_classifier::mean_distance_square() const;
```

qui renvoie un vecteur qui contient pour chaque cluster la distance carrée moyenne des points au centre du cluster, i.e., pour chaque $0 \leq k < K$, le vecteur résultat contient

$$v_k = \frac{1}{|P_k|} \sum_{x \in P_k} d(c_k, x)^2$$

Attention : `p.dist_2(q)` *renvoie déjà le carré de la distance.*

11. Écrire un fichier `test2.cpp` qui classifie les 500 points d'un fichier `cloud.dat` par rapport aux 4 centres $(0,0)$, $(5,3)$, $(-2,4)$, $(-1,-4)$, affiche le nombre de points de chaque cluster et affiche les quatre moyennes des carrés des distances de chaque cluster. Exécuter le fichier et vérifier qu'on obtient bien :

```
Taille des 4 clusters: 138 127 107 128
Valeurs des v_k: 4.8247  7.98358  7.25505  6.43587
```

12. Ajout d'un nouveau centre. Il se peut qu'on ait déjà classé un ensemble P de N points autour de K centres $(c_k)_{0 \leq k < K}$ en construisant les K ensembles $(P_k)_{0 \leq k < K}$. On décide alors d'insérer un nouveau centre c' et passer ainsi de K centres à $K+1$ centres avec $c_K = c'$. Il n'est alors pas nécessaire de refaire tous les calculs : il suffit de parcourir chaque ancien cluster et de déplacer les points nécessaires vers le nouveau cluster . L'algorithme est alors le suivant :

- on introduit un ensemble S vide (le futur cluster associé à c')
- pour chaque entier $0 \leq k < K$,
  - on introduit un ensemble P'_k vide
  - on parcourt tous les éléments de P_k et, pour chaque point p, on l'ajoute à S si $d(c', p) < d(c_k, p)$ et sinon on l'ajoute à P'_k
  - on remplace P_k par P'_k
- à la fin, le cluster P_K associé au nouveau point $c_K = c'$ est donné par l'ensemble S.

Écrire le code de la méthode

```
int Voronoi_classifier::add_center(Point2D new_center);
```

qui ajoute un nouveau cluster à la fin de `clusters` de centre c' donné par `new_center` et met à jour les clusters selon l'algorithme précédent. On renverra la taille du nouveau cluster ainsi formé.

13. Compléter votre fichier `test2.cpp` en ajoutant le point $(0,-1)$ et afficher les tailles des 5 clusters. Vous devriez obtenir

```
Taille des 4 clusters après l'ajout: 75 126 107 115 77
```

14. Afin de pouvoir faire une représentation graphique simple des clusters (avec Python ou gnuplot), on souhaite écrire tous les points (centres et points de données) dans un fichier sous la forme d'une suite de lignes

```
x   y   index
x'  y'  index'
....
```

où `x` et `y` sont les coordonnées du point et `index` est un entier qui vaut -1 si le point est un centre et i si le point est un point de donnée appartenant au i-ème cluster.

Écrire la méthode

```
void Voronoi_classifier::print(std::ostream &) const
```

qui réalise cette écriture. Compléter votre fichier `test2.cpp` pour tester cette méthode en écrivant le résultat dans un fichier `clusters_rep.dat` .

Bonus : si vous le souhaitez, vous pouvez visualiser le fichier en utilisant le logiciel "gnuplot". Pour cela, taperz "gnuplot" dans le terminal puis, à l'intérieur de gnuplot, tapez

```
plot "clusters_rep.dat" u 1:2:3 pt 7 ps 5 palette"
```

et vous verrez apparaître une représentation graphique colorée des clusters.

15. Ajouter une méthode `+=` qui fait la même chose que `add_point(p)` sous la nouvelle syntaxe `VC += p;` . Tester en modifiant vos fichiers de test précédents.

Fin du sujet

Rappels sur le conteneur `std::set` .

Le conteneur `std::set<T>` permet de décrire un ensemble d'objets de type `T` . Les fonctionnalités sont les suivantes :

- le constructeur par défaut crée un ensemble vide
- la taille de l'ensemble est accessible par la méthode `size()`
- on insère un point `t` à un ensemble `E` par `E.insert(t);`
- on peut parcourir tous les élements d'un ensemble par

```
for(T t: E) {...} // ou bien const T & pour les gros objets
```

- on peut utiliser la bibliothèque `<algorithm>` avec les itérateurs `E.begin()` et `E.end()` .