K-moyennes

17 juin

L'algorithme des k-moyennes est un algorithme qui cherche à construire un nombre de clusters défini à l'avance (appelé k dans la littérature, mais qu'on notera m en cohérence avec les parties précédentes). L'objectif est de résoudre le problème d'optimisation `CLUSTERING` :

- **Instance** : un ensemble de données $E = \{x_1, \ldots, x_N\}$ et un entier m.
- **Solution** : une partition de E en m classes $C_1, \ldots, C_m$.
- **Optimisation** : minimiser $\sum_{j=1}^{m} \sum_{x \in C_j} \delta(x, b_j)^2$, où b_j est le barycentre de la classe C_j.

Cependant, ce problème d'optimisation est NP-difficile donc l'algorithme présenté ici est une heuristique considérée comme satisfaisante mais qui peut renvoyer un résultat non optimal. L'idée de l'algorithme des k-moyennes est la suivante :

— créer m points $b_1, \ldots, b_m$, soit au hasard dans l'espace considéré, soit choisis au hasard parmi les x_i ;
— tant que les b_j changent :
  — affecter à chaque point x_i la classe j correspondant au barycentre b_j qui lui est le plus proche ;
  — modifier chaque b_j en le barycentre de la nouvelle classe j.

On représente une partition d'un ensemble E en m classes comme un tableau `classes` de taille N telle que `classes[i]` vaut l'entier j tel que $x_i \in C_j$. On se place dans le cas où $d = 2$. On représente un point de $\mathbb{R}^2$ par le type :

```
struct Point {
    double abs;
    double ord;
};

typedef struct Point point;
```

tel qu'un point $x = (a, b) \in \mathbb{R}^2$ est représenté par un objet `x` de type `point` tel que `x.abs` vaut a et `x.ord` vaut b.

1. Écrire une fonction `point* echantillon(point* donnees, int N, int m)` qui prend en argument un tableau `donnees` contenant N points et un entier m et renvoie un tableau correspondant à un échantillon de taille m dans le tableau `donnees`. On autorisera la modification du tableau et on rappelle qu'un appel à `rand() % n` renvoie un entier choisi aléatoirement et uniformément dans $[\![0, n-1]\!]$.
2. Écrire une fonction `int plus_proche(point x, point* bary, int m)` qui prend en arguments un point x et un tableau contenant m points et renvoie l'indice j du point du tableau le plus proche de x.
3. En déduire une fonction

   `bool modif_classes(point* donnees, point* bary, int* classes, int N, int m)`

   qui prend en arguments l'ensemble E (un tableau de N points), un tableau de m barycentres`bary` et un tableau de classes de taille N et modifie `classes` de telle sorte qu'après modification, `classes[i]` correspond à l'indice j du barycentre le plus proche de `donnees[i]`. La fonction renverra `true` s'il y a eu une modification de `classes`
4. Écrire une fonction

   `void modif_bary(point* donnees, point* bary, int* classes, int N, int m)`

   qui prend les mêmes arguments que la fonction précédente et modifie le tableau `bary` de telle sorte qu'après modification `bary[j]` correspond au barycentre de la classe j.
5. En déduire une fonction `int* k_moyennes(point* donnees, int N, int m, int itermax)` qui prend en arguments un ensemble E de données de taille N, un entier m et un entier `itermax` et renvoie un pointeur vers un tableau `classes` correspondant à une partition de E de taille m selon l'algorithme des k-moyennes. On arrêtera l'algorithme soit lorsqu'il y a convergence, soit lorsqu'on a dépassé le nombre maximal d'itérations.