TP n°4 - Algorithme des k moyennes

Notions abordées

- Algorithme des k moyennes
- Utilisation du module Graphics en OCaml
- Passage d'argument à un programme en OCaml

Exercice 1 : Algorithme des k moyennes en OCaml

Dans cet exercice on se propose d'implémenter l'algorithme des k moyennes en OCaml. Dans tout l'exercice p représente la dimension de l'espace dans lequel vivent les vecteurs à regrouper. L'algorithme prend en arguments un entier k et un ensemble de vecteurs $\mathcal{D} \subseteq \mathbb{R}^p$ et essaie de regrouper ces vecteurs en k groupes de sorte à minimiser la somme des carrés des distances des vecteurs au barycentre de leur groupe.
Avant de rappeler l'algorithme des k moyennes, on introduit une notation pour l'ensemble des vecteurs du jeu de données se trouvant dans le i-ème groupe.

Notation C_i^m. Étant donné une famille $(m_i)_{i\in[\![1,k]\!]}$ de k vecteurs de $\mathbb{R}^p$ et un entier $i \in [\![1,k]\!]$, on note C_i^m l'ensemble des vecteurs du jeu de données tels que le vecteur de $(m_j)_{j\in[\![1,k]\!]}$ le plus proche de v est celui d'indice i, ce qui s'écrit comme suit♣.

$$C_i^m = \{v \in \mathcal{D} \mid \underset{j\in[\![1,k]\!]}{\arg\min} \|v - m_j\|_2 = i\}$$

On rappelle l'algorithme des k moyennes.

Algorithme 1 : Algorithme des k moyennes

Entrée : Un jeu de données $\mathcal{D}$, un paramètre k
Sortie : Une partition de $\mathcal{D}$ en k groupes qui minimise localement la somme des carrés des distances de chaque point de $\mathcal{D}$ au barycentre de son groupe

```
(m_i)_{i∈[1,k]} ← k points parmi les données D ;
Stable ← faux ;
tant que nonStable faire
    C ← (C_i^m)_{i∈[1,k]} ;
    m' ← (barycentre(C_i))_{i∈[1,k]} ;
    Stable ← m ≟ m' ;
    m ← m' ;
retourner C
```

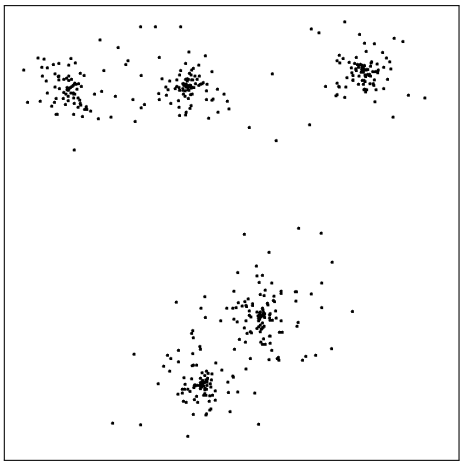
♣. bien que $\arg\min$ désigne classiquement l'ensemble des points en lesquels le minimum est atteint, il désigne ici le plus petit élément de cet ensemble qui est ici un sous-ensemble non vide de $\{1, k\}$

Représentation des données. Dans le reste de cet exercice, le groupe d'un vecteur du jeu de données sera représenté par un entier de l'intervalle $[\![0, k-1]\!]$ puisqu'il y a k tels groupes. Ainsi l'algorithme des k moyennes manipule 3 tableaux :

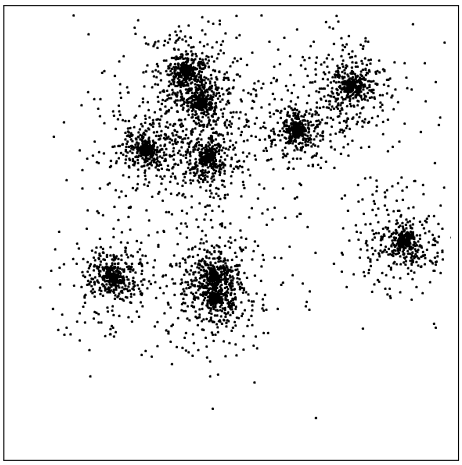
- un tableau `d` de n vecteurs qui n'est jamais modifié et qui représente le jeu de données à regrouper $\mathcal{D}$;
- un tableau `cl` de n entiers indiquant dans sa case d'indice i le numéro du groupe auquel appartient la donnée `d.(`i`)` ;
- un tableau `ce` de k vecteurs contenant dans sa case d'inidce i le barycentre du groupe numéro i décrit par `cl`.

Code de `compagnon.ml`. Dans le fichier `compagnon.ml` disponible sur cahier de prépa, on fournit :

- un module `module Vector` permettant la manipulation de vecteurs de $\mathbb{R}^p$;
- un type `type data = Vector.t array` permettant de représenter un jeu de données ou plus généralement une famille de vecteurs ;
- des fonctions de visualisation graphique (décrites dans le fichier) :
  - `draw_data (d: data) : unit;`
  - `draw_clustering (d: data) (cl: int array): unit;`
  - `draw_clustering_w_centers (ce: data) (d: data) (cl: int array): unit;`
- une fonction `gen_data : int -> int -> float -> data` permettant de générer un jeu de données dans $\mathbb{R}^2$. Plus précisément l'appel `(gen_data nb_elem nb_cluster deviation)` génère un jeu de `nb_elem` vecteurs de $[0,1]^2$ répartis dans `nb_cluster` clusters dont l'étendue spatiale est définie par l'argument `deviation`. Les données générées étant en 2 dimensions il est possible de les afficher en utilisant les fonctions pré-citées. On obtient par exemple les images ci-après.



`(gen_data 500 5 0.05)`



`(gen_data 5000 10 0.05)`

Q. 1 Définir une fonction `find_closest_center (ce: data) (v: Vector.t): int` prenant en arguments : un tableau de vecteurs `ce`, et un vecteur `v` et calculant l'indice dans `ce` du vecteur de `ce` minimisant la distance à `v`.

Q. 2 Définir une fonction `modify_clustering (ce: data) (d: data) (cl: int array) : bool` prenant en arguments : un tableau de vecteurs `ce` de taille k, un tableau `data` de taille n et un tableau `cl` de taille n. La fonction doit remplir le tableau `cl` pour y stocker, dans chaque case d'indice i, l'indice dans `ce` du vecteur le plus proche de `d.(`i`)`. Votre fonction devra renvoyer `true` si et seulement si une des données a changé de groupe, c'est à dire si au moins une case du tableau `cl` a été modifiée.

Q. 3 Définir une fonction `recompute_centers (ce: data) (d: data) (cl: int array) : unit` prenant en arguments les mêmes données que la fonction précédente et mettant à jour les

centres de chacun des groupes. Aussi le tableau `ce` ne doit pas être lu par l'algorithme, chaque case `ce.(i)` doit recevoir le barycentre des vecteurs appartenant au groupe numéro `i`, selon le tableau `cl`.

Q. 4 Définir une fonction `k_means_update (ce: data) (d: data) (cl: int array): bool` effectuant une itération de l'algorithme des k moyennes et renvoyant si cette étape a modifié `cl`.

Q. 5 Définir l'algorithme des k moyennes dans une fonction `k_means (k: int) (d: data) : int array`, prenant en arguments un entier k et un jeu de données et retournant la partition calculée par l'algorithme des k moyennes en choisissant comme centres initiaux les k premiers vecteurs du jeu de données.

`Graphics` et pause dans l'exécution. Lorsqu'une fenêtre graphique est ouverte (voir ci-dessous) l'évaluation de l'expression OCaml `Graphics.wait_next_event [Graphics.Key_pressed]` met en pause l'exécution d'un programme OCaml en attendant que l'utilisateur appuie sur une touche du clavier.

Q. 6 Dupliquer la fonction `k_means` en une fonction `k_means_anim` et modifier cette dernière de manière à ce qu'elle réalise l'affichage pas à pas de l'algorithme. Plus précisément cette fonction affiche d'abord le jeu de données (sans k centres choisis initialement qui ne sont pas pertinents), et attend qu'une lettre soit appuyée pour passer à l'étape suivante : elle affiche alors le jeu de données colorié selon les k centres après une itération de l'algorithme, attend à nouveau qu'une touche soit appuyée et ainsi de suite. On pourra éventuellement proposer un affichage spécial pour la partition finalement retournée par l'algorithme, afin que lorsqu'on appuye successivement sur une touche pour avancer, on ne risque pas de louper le résultat en appuyant une fois de trop.

Q. 7 Ajouter un point d'entrée à votre programme (une valeur `main` de type `unit`, qui n'est donc pas une fonction), de sorte que la compilation de votre fichier produise un exécutable qui, lorsqu'il est lancé en ligne de commande, prend en paramètres[♣] un entier `n`, un entier `k` et un flottant `deviation`, génère un jeu de données avec `gen_data` selon ces 3 paramètres, et affiche l'exécution pas à pas de l'algorithme des k moyennes (avec k valant `k`) sur le jeu de données généré.

On précise que pour compiler un fichier `code.ml` utilisant `Graphics` en un exécutable `code`, on peut taper la commande suivante.

```
ocamlfind ocamlopt -o code -linkpkg -package graphics code.ml
```

♣. on pourra se référer au topo à ce sujet

Créer un exécutable qui prend des arguments en ligne de commande en OCaml

Il est possible de passer des paramètres à un programme OCaml compilé `prgm` au moment où on lance son exécution en ligne de commande. On rappelle que la commande ci-dessous compile le fichier `code.ml` en un exécutable `prgm`.

```
ocamlc code.ml -o prgm
```

Contrairement à ce qui est fait en C, le point d'entrée du programme n'est pas une fonction particulière. En effet l'exécution de `prgm` a pour effet l'évaluation de toutes les valeurs définies dans `code.ml`, en particulier celles de type **`unit`** dont on peut observer les effets de bord. Ainsi, de même qu'on peut définir un jeu de tests par **`let`** `test_f:`**`unit`** `=...`, on peut définir la suite d'instructions principales par **`let`** `main:`**`unit`** `=...`.
On accède aux arguments passés au programme à travers le tableau `Sys.argv`.
Comme en C, il s'agit d'un tableau de chaînes de caractères (*i.e.* de type **`string array`**) initialisé selon les mots de la ligne de commande. En particulier `Sys.argv(`**`0`**`)` est le nom de la commande (rarement utile), et `Array.length Sys.argv` est le nombre de paramètres +1.

Deux exemples

```
let main : unit =
  print_string ("ici on a lancé la commande \" "^Sys.argv.(0)^"\" \n");
  let n = Array.length Sys.argv in
  for i=1 to n-1 do
    print_string ("le paramètre "^string_of_int i^" est "^Sys.argv.(i)^"\n")
  done;
```

Si `affiche_param` est le programme obtenu en compilant le code ci-dessus, `./affiche_param 12 bjr` produit l'affichage suivant.

```
ici on a lancé la commande " ./affiche_param"
le paramètre 1 est 12
le paramètre 2 est bjr
```

```
let main : unit =
  let n = Array.length Sys.argv in
  let s = ref 0 in
  for i=1 to n-1 do
    s := !s + int_of_string Sys.argv.(i)
  done;
  print_string("la somme des paramètres vaut "^string_of_int !s^"\n")
```

Si `add` est le programme obtenu en compilant le code ci-dessus, alors

- `./add` affiche `la somme des paramètres vaut 0`
- `./add 1 2 6` affiche `la somme des paramètres vaut 9`
- `./add 1 2 salut` affiche l'erreur `Fatal error: exception Failure("int_of_string")`

NB : lors de l'import avec `#use` sous `utop`, il n'est pas possible de passer des paramètres, seul `Sys.argv.(0)` est rempli avec le chemin jusqu'à l'exécutable `utop`.