Devoir #1 — INF5171 (gr. 20)

Automne 2017

Date de remise: Mardi, 17 octobre, 13h30. Exceptionnellement, **aucun travail en retard ne sera accepté** car la solution sera présentée et discutée en classe, mardi 17 octobre, lors de la séance de cours avant l'examen intra — l'examan a lieu le mardi 24 octobre!

Classification de données par la méthode des k-moyennes (*k-means clustering*)

> «*Machine learning means learning from data; AI is a buzzword. Machine learning lives up to the hype: there are an incredible number of problems that you can solve by providing the right training data to the right learning algorithms. Call it AI if that helps you sell it, but know that AI is a buzzword that can mean whatever people want it to mean.*»
>
> Source : `https://www.forbes.com/sites/quora/2017/09/06/ten-things-everyone-should-know-about-machine-learning/#690ccd7c4e9e`

1 Le problème : Classification de données en k groupes

Depuis quelques années, on entend souvent parler d'*apprentissage machine* (*machine learning*) — on dit aussi «apprentissage automatique» — un champ d'étude de l'intelligence artificielle.

Une des formes d'apprentissage machine est celle de l'*apprentissage non supervisé* :

> Dans le domaine informatique, **l'apprentissage non supervisé** (parfois dénommé «*clustering*») est une méthode d'apprentissage automatique. Il s'agit pour un logiciel de diviser un groupe hétérogène de données, en sous-groupes de manière que les données considérées comme les plus similaires soient associées au sein d'un groupe homogène et qu'au contraire les données considérées comme différentes se retrouvent dans d'autres groupes distincts ; l'objectif étant de permettre une extraction de connaissance organisée à partir de ces données.
>
> Source : `https://fr.wikipedia.org/wiki/Apprentissage_non_supervis%C3%A9`

Partitionnement en k-moyennes

Étant donné un ensemble de n points $P = \{x_1, x_2, \ldots, x_n\}$, on cherche à *partitionner*[1] les n points en k groupes $S = \{S_1, S_2, \ldots, S_k\}$ — typiquement avec k *beaucoup plus petit* que n — de façon à minimiser une certaine fonction de coût associée aux groupes sélectionnés.

La fonction de coût souvent utilisée est la somme du carré de la distances entre chacun des points et un point qui est la «moyenne» des points du groupe, et qu'on appelle **le représentant** du groupe. C'est la méthode dite du **partitionnement en k-moyennes** (*k-means clustering*) :

> Étant donné un ensemble de points $(x_1, x_2, \ldots, x_n)$, on cherche à partitionner les n points en k ensembles $S = \{S_1, S_2, \ldots, S_k\}$ $(k \leq n)$ en minimisant la distance entre les points à l'intérieur de chaque partition :
>
> $$\arg\min_{S} \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2 \tag{1}$$
>
> où μ_i est le barycentre[2] des points dans S_i.
>
> Source : `https://fr.wikipedia.org/wiki/K-moyennes`

Dans le présent devoir, nous allons supposer qu'un point de donnée est un vecteur — à 1, 2 ou plusieurs dimensions (selon les données traitées par le problème). La distance entre deux points x et y, notée $||x-y||$, est alors simplement la distance vectorielle (euclidienne). Par exemple, pour des vecteurs à deux dimensions $x = (x_0, x_1)$ et $y = (y_0, y_1)$, la distance (en notation mathématique) est définie comme suit :

$$||x - y|| = \sqrt{(x_0 - y_0)^2 + (x_1 - y_1)^2}$$

Par contre, le code Ruby fourni utilise une classe `Point`, possédant un attribut `position`, lequel est un `Vector`. La distance entre deux `Point`s est alors obtenue par une méthode `distance`, et on écrira `x.distance(y)`.

Note : Signalons que cette méthode `distance` sur des `Point`s s'évalue en appelant ensuite la méthode `distance` définie sur les objets `Vector`.

Remarque 1: Au sujet des points traités et de la distance entre ces points.

[1] Une **partition** de P en k sous-ensembles $S_1, S_2, \ldots, S_k$ implique deux propriétés :

1. Les S_i sont disjoints : $i \neq j \Rightarrow S_i \cap S_j = \emptyset$
2. La combinaison des S_i redonne P : $P = S_1 \cup S_2 \cup \ldots \cup S_k$

[2] On dit aussi «le centre de masse», typiquement, la «moyenne» des points.

Une heuristique pour le partitionnement en k-moyennes

Étant donné une valeur k, il n'existe pas **d'algorithme** en temps raisonnable (polynomial) pour calculer **de façon exacte** les k meilleurs groupes, i.e., les k groupes qui *minimisent* la fonction mentionnée plus haut (Équation 1).

Pour obtenir une solution «pas trop mauvaise» en temps raisonnable, on a plutôt recours à une *heuristique* :

> Une **heuristique** est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile. C'est un concept utilisé entre autres en optimisation combinatoire, en théorie des graphes, en théorie de la complexité des algorithmes et en intelligence artificielle.
>
> Une heuristique s'impose quand les algorithmes de résolution exacte sont de complexité exponentielle, et dans beaucoup de problèmes difficiles.
>
> Source : `https://fr.wikipedia.org/wiki/Heuristique_(math%C3%A9matiques)`

L'heuristique utilisée pour le partitionnement en k-moyennes procède comme suit :

1. On choisit k points qui, initialement, vont servir de **représentants** des divers groupes (voir Remarque 2).
2. On répète les étapes suivantes jusqu'à ce qu'il n'y ait plus de modification[3] :
   - Pour chaque point de données, on détermine lequel des k représentants courants est **le plus près** et on associe le point au groupe de ce représentant.
   - Pour l'ensemble des points faisant partie d'un même groupe, on calcule la nouvelle moyenne, i.e., **le nouveau représentant.**

Le Code Ruby 1 donne le code Ruby (partiel) pour la méthode `Classificateur#run` qui met en oeuvre cette heuristique.

Dans le code Ruby fourni, le choix des représentants initiaux peut être fait de deux façons :

- Dans les tests (pour assurer leur reproducibilité) et dans l'exemple ci-bas, on choisit simplement les k premiers points **distincts** parmi les points à traiter.
- Dans certains exemples, avec affichage notamment, on choisit **aléatoirement** k points **distinct** parmi les points à traiter.

Remarque 2: Au sujet du choix des représentants initiaux.

[3] Ou, si on veut limiter le temps d'exécution, jusqu'à ce qu'on ait atteint un nombre **maximal** d'itérations spécifié lors de l'appel à la méthode `run` : voir plus bas.

Code Ruby 1 Code (partiel) pour la méthode `Classificateur#run`, qui met en oeuvre l'heuristique pour le partitionnement en k-moyennes.

```ruby
def run( nb_groupes = nil, nb_iterations_max = nil )
  @representants = definir_representants( nb_groupes )\
                   if representants_doivent_etre_generes?( nb_groupes )

  anciens = nil
  nb_iterations = 0
  until Classificateur.memes_representants?( anciens, @representants ) ||
        iterations_completees?( nb_iterations, nb_iterations_max )

    classifier_les_points( points, @representants )

    anciens = @representants
    @representants = nouveaux_representants( @nb_groupes, points )

    nb_iterations += 1
  end

  nb_iterations
end
```

Un exemple : Classification d'informations sur des âges

Nous allons illustrer l'approche des k-moyennes à l'aide d'un petit exemple, tel qu'il devrait être traité par le programme Ruby.

Soit les données suivantes, qui représentent l'âge de diverses personnes :

```
9, 10, 10, 10, 10, 11, 22, 23, 23, 24, 55, 56, 57, 59, 60, 61
```

Note : Dans le programme Ruby, ces valeurs sont représentées par des `Vector` à une dimension, e.g., `Vector[9]`. Dans le présent exemple, pour simplifier, nous indiquerons directement les valeurs.

On veut classifier ces données en trois (3) groupes, c'est-à-dire $k = 3$.

Voici alors les différentes étapes suivies par le programme :[4]

0. On sélectionne tout d'abord les trois (3) représentants initiaux, un représentant par groupe. Ici, nous allons simplement choisir les trois premiers points **distincts**.[5]

   ```
   Représentants initiaux:
   0: 9
   1: 10
   2: 11
   ```

1. On associe chaque point de données au groupe dont le représentant est le plus près, et donc on obtient les trois groupes suivants :

   ```
   0: [9]
   1: [10, 10, 10, 10]
   2: [11, 22, 23, 23, 24, 55, 56, 57, 59, 60, 61]
   ```

   On calcule les nouveaux représentants (points «moyens») des groupes ainsi produits :

   ```
   Représentants:
   0: 9.0
   1: 10.0
   2: 41.0
   ```

[4] Les graphiques ont été produits par «`make afficher_ages`». Voir le fichier `afficher_classification.rb`.

[5] L'approche ne fonctionne pas si deux représentants ont la même valeur, puisque leurs groupes sont alors confondus, et donc la valeur effective de k est inférieure à celle souhaitée.

2. Les représentants ont changé, donc on répète le processus en fonction des nouveaux représentants :

   ```
   0: [9]
   1: [10, 10, 10, 10, 11, 22, 23, 23, 24]
   2: [55, 56, 57, 59, 60, 61]
   ```

   On calcule les nouveaux représentants :

   ```
   Représentants:
   0: 9.00
   1: 15.89
   2: 58.00
   ```

3. Les représentants ont changé, donc on répète :

   ```
   0: [9, 10, 10, 10, 10, 11]
   1: [22, 23, 23, 24]
   2: [55, 56, 57, 59, 60, 61]
   ```

   On calcule les nouveaux représentants :

   ```
   Représentants:
   0: 10.00
   1: 23.00
   2: 58.00
   ```

4. Les représentants ont changé, donc on répète le processus :

   ```
   0: [9, 10, 10, 10, 10, 11]
   1: [22, 23, 23, 24]
   2: [55, 56, 57, 59, 60, 61]
   ```

   Les groupes et les représentants n'ont pas changé, donc on a trouvé la «solution» ; on termine avec ces groupes et leurs représentants.

2 Ce que vous devez faire

Divers fichiers vous sont fournis, sous forme d'un dépôt `git` :

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/KMoyennes.git
```

La classe principale, `Classificateur`, est documentée (avec `yard`) à l'adresse suivante :
`http://www.labunix.uqam.ca/~tremblay_gu/INF5171/Devoirs/KMoyennesDoc/Classificateur.html`

Cette classe définit diverses méthodes publiques, déjà mises en oeuvre. La méthode la plus importante est `Classificateur#run`. Cette méthode est simplement un répartiteur (*dispatcher*) qui fait appel à diverses méthodes, privées, dont deux que vous aurez à compléter :

1. `"classifier_les_points_#{mode}"` : classifie les différents points à traiter en fonction des représentants actuellement associés à chacun des groupes — donc modifie possiblement l'attribut `groupe` des divers points de données.
2. `"nouveaux_representants_#{mode}"` : produit une nouvelle liste de représentants en fonction de la classification qui vient d'être effectuée par l'appel à la méthode `"classifier_les_points_#{mode}"`.

Ces deux méthodes sont appelées **de façon dynamique**, i.e., avec `send`, en fonction de la valeur de la variable `mode`. En d'autres mots, c'est la valeur de la variable `mode` qui détermine quelle méthode, privée, sera effectivement appelée.

Dans le fichier fourni, les opérations (privées) en version séquentielle sont définies avec en-tête seulement, donc sans mise en oeuvre. Quant aux versions parallèles, elles sont simplement mises en oeuvre en appelant la version séquentielle. C'est donc à vous **de compléter ces diverses méthodes, selon les indications données plus bas.**

- Séquentielle
  1. `classifier_les_points_seq` et `nouveaux_representants_seq` : Mise en oeuvre *sequentielle*, sans parallélisme d'aucune sorte.
- Parallélisme *fork/join* à granularité (très!) fine :
  2. `classifier_les_points_par_fj_fin` et `nouveaux_representants_par_fj_fin` : Mise en oeuvre avec parallélisme à granularité fine, la plus fine permise par le problème — i.e., autant de tâches que possible, sans *agglomération de tâches*, même si ce n'est pas nécessairement efficace.[6]
- Parallélisme à granularité grossière avec du parallélisme *fork/join* — donc utilisant `PRuby.pcall` ou `PRuby.future`.
  3. `classifier_les_points_par_fj_adj` et `nouveaux_representants_par_fj_adj` : Répartition statique par tranches d'éléments adjacents.
  4. `classifier_les_points_par_fj_cyc` et `nouveaux_representants_par_fj_cyc` : Répartition statique **cyclique** des éléments, par groupes de taille spécifiée par `taille_tache`.
- Parallélisme à granularité grossière avec du parallélisme **autre** que *fork/join* — **donc sans utiliser** `PRuby.pcall` **ou** `PRuby.future`.
  5. `classifier_les_points_par_sta` et `nouveaux_representants_par_sta` : Répartition statique des éléments, par tranches d'éléments adjacents si `taille_tache` n'est pas spécifiée, cyclique par groupe si `taille_tache` est spécifiée.
  6. `classifier_les_points_par_dyn` et `nouveaux_representants_par_dyn` : Répartition dynamique des éléments entre les *threads*, par groupes de taille spécifiée par `taille_tache`.

  Pour ces deux dernières versions parallèles (`*_par_{sta,dyn}`), vous ne pouvez pas non plus utiliser `Thread.new`, puisqu'il s'agit d'une autre forme de *fork/join*. Vous devez donc utiliser **d'autres méthodes de la bibliothèque** `PRuby`. Si vous faites les bons choix, alors ce sont des versions (très!) **faciles** à écrire.

[6] Les mesures de temps d'exécution et d'accélération ne seront pas exécutées avec cette version!

3 Ce qui vous est fourni

Plusieurs fichiers vous sont fournis, que vous n'avez pas à modifier, sauf évidemment pour le fichier `classificateur.rb` :

1. `vector.rb` : une extension de la classe `Vector`, qui définit (ou redéfinit) diverses méthodes ainsi que deux assertions (utilisées dans les tests).

   Pour la documentation des autres méthodes sur les vecteurs, voir la documentation en ligne de cette classe :

   `http://ruby-doc.org/stdlib-2.2.1/libdoc/matrix/rdoc/Vector.html`

2. `point.rb` : la classe `Point` et ses diverses méthodes et attributs.

3. `classificateur.rb` : la classe `Classificateur` et ses diverses méthodes et attributs.

4. `*_spec.rb` : trois (3) suites de tests définies à l'aide de `MiniUnit`.

Fichier	**Classe testée**
`point_spec.rb`	`Point`
`classificateur_spec.rb`	`Classificateur` version séquentielle
`classificateur_par_spec.rb`	`Classificateur` versions parallèles

   Plus précisément, les tests pour les versions parallèles **vérifient que les résultats obtenus sont bien les mêmes que pour la version séquentielle.** Au préalable, il faut donc que les tests de la version séquentielle s'exécutent avec succès.

5. `afficher_classification.rb` : un programme qui permet de visualiser le processus de classification d'une collection de points. Un certain nombre de collections sont définies et leur classification peut être visualisée.

6. Des fichiers pour effectuer des mesures de temps d'exécution et d'accéleration :

   - `mesurer_temps.rb` : mesure le temps d'exécution de diverses mises en oeuvre — séquentielle vs. parallèles — en fonction de divers paramètres.
   - `calculer_accelerations.rb` : lorsque des temps d'exécution ont été calculés, permet de calculer les accélérations — sous forme textuelle.
   - `gnuplot-temps.sh` et `gnuplot-acc.sh` (et `temps_max.rb`) : lorsque des temps d'exécution ont été calculés, permet de générer des graphes (en format `png`) des temps d'exécution et des accélérations.

7. `makefile`, où les principales cibles sont les suivantes :

   - `make tests_seq` : exécute *l'ensemble des tests* pour la mise en oeuvre **séquentielle.**
   - `make tests_par` : exécute les tests pour **toutes les mises en oeuvre parallèles**, donc toutes les méthodes parallèles privées appelées par `run`.

     On peut tester une version parallèle spécifique avec une des cibles suivantes :

     ```
     tests_par_fj_fin
     tests_par_fj_adj
     tests_par_fj_cyc
     tests_par_sta
     tests_par_dyn
     ```

     Pour ce faire, il suffit de modifier la variable `TEST` dans le fichier `makefile`.
   - `make afficher` : visualise le processus de classification d'une collection de points.

     Par défaut, c'est la collection de points `nombreux_points_en_trois_groupes` qui est classifiée, en trois (3) groupes.

     Pour visualiser la classification de l'une des autres collections avec un nombre différents de groupes, par exemple `points4` en deux groupes, on utilise la commande suivante :

     ```
     $ make afficher POINTS=points4 NB_GROUPES=2
     ```
   - Cibles pour les mesures de temps d'exécution et d'accélération — où vous devez remplacer `N` par une valeur entière :
     - `make temps_N` : exécute le programme `mesurer-temps.rb` pour `N` points.
     - `make acc_N` : calcule et affiche (format textuel) les accélérations obtenues pour les mesures avec `N` points.
     - `make graphes_N` : génère (format `png`) les graphes de temps d'exécution et d'accélérations pour les mesures obtenues pour `N` points.
     - `make benchmarks` : génère les données et graphes pour différentes tailles de collection — 64, 128, etc. À n'exécuter que lorque le programme produit les bons résultats (tous les tests s'exécutent avec succès) et que vous avez réussi à produire les graphes pour des petits collections de points.
   - `make remise` : remet la copie électronique de votre code — voir plus bas.

4 Ce que vous devez remettre

Remise papier

Vous devez remettre un document **papier** (remis dans la boîte de remise des travaux du secrétariat du département) contenant les éléments suivants :

1. Le listage de votre fichier `classificateur.rb` — **et uniquement ce fichier!**
2. Des **graphes** donnant les temps d'exécution et les accélérations obtenus pour vos versions parallèles (**sauf pour la version à granularité fine!**), et ce pour divers nombres de points — voir `makefile` et la cible `benchmarks`.

   Remarque importante : Si vous n'avez pas réussi à mettre en oeuvre une version parallèle, alors indiquez-le clairement dans votre rapport et faites simplement un appel à la version séquentielle dans le code Ruby de la méthode parallèle (tel que cela est fait par défaut dans le code fourni). Ceci vous permettra de quand même générer les graphes de temps d'exécution et d'accélération, et ce sans devoir modifier les cibles et scripts — le programme `mesurer_temps.rb` vérifie que le résultat parallèle est bien le même que celui séquentiel.
3. Un «rapport» écrit où vous répondez aux questions suivantes, avec de brèves justifications :
   (a) Quelles sont les mises en oeuvre parallèles que vous avez complétées correctement?
   (b) Quelle semble être la meilleure accélération obtenue et pour quelle(s) version(s) cette meilleure accélération est-elle obtenue?
   (c) Si une ou des versions semblent meilleures que les autres (?), pouvez-vous expliquer pourquoi il en est ainsi?
   (d) Est-ce que l'accélération semble dépendre du nombre de points traitées?
   Si oui, pour quel nombre de points l'accélération semble-t-elle la plus élevée?
   (e) Dans la version avec répartition dynamique, quel est l'effet de la taille des tâches (blocs)? Pour quelle(s) taille(s) semble-t-on obtenir de meilleures accélérations?

Pour la remise de ce document papier, vous devez utiliser **la page couverture (avec grille de correction)** disponible à la fin du présent du document.

Remise électronique

Vous devez remettre une version électronique du fichier `classificateur.rb` en exécutant la commande suivante sur `japet.labunix.uqam.ca` :

```
$ make remise
```

Note : Vous devez *auparavant* modifier la variable `CODES_PERMANENTS` dans le `makefile` pour indiquer vos codes permanents (si vous travaillez en équipe, deux personnes maximum) ou votre code permanent (si vous travaillez seul).

5 Remarques sur la correction, suggestions et indices

Remarques sur la correction

- La correction des travaux se fera sur la machine `japet.labunix.uqam.ca`, qui comporte 64 coeurs/processeurs. Donc, vous devez absolument vous assurer que votre programme fonctionne correctement sur cette machine avant de le remettre.

- Si le fichier remis contient des erreurs de syntaxe, alors vous perdrez une très (très!) grande partie des points — voir plus bas.

- Une partie des points sera accordée pour la *qualité* et le style de votre code — présentation, clarté et simplicité, structure, choix des identificateurs — ainsi que le respect des conventions Ruby[7] , etc. Par contre, même si la présence de commentaires est suggérée, je n'évaluerai pas la présence, ou l'absence, de tels commentaires.

  Deux principes de base pour la clarté du code : KISS et DRY.[8]

  Autre principe pour la correction du style : plus je mets de rouge parce que je ne comprends pas le code ou parce que le code est mal écrit, plus la note est faible ☹

- Je pourrai effectuer la vérification du bon fonctionnement *avec d'autres tests* que ceux qui vous sont fournis — donc avec des *tests privés* additionnels.

  De plus, la vérification du «bon fonctionnement» consistera aussi à vérifier que, pour chacune des versions parallèles, **vous avez effectivement utilisé la stratégie de programmation parallèle indiquée**. Dans le cas contraire, aucun point ne sera accordé pour la version concernée.

  En gros, la répartition des points sera la suivante : compilation correcte ≈ 20 %, bons résultats sur les tests publics ≈ 60 %, bons résultats sur les tests privés ≈ 20 %.

- **BONUS ;** L'équipe (ou les équipes) qui remettra (remettront) le fichier `classificateur.rb` dont les opérations parallèles de simulation fonctionnant sur plusieurs processeurs seront les plus rapides recevra (recevront) un **bonus** (pouvant aller jusqu'à 10 %).

[7] `https://github.com/styleguide/ruby`
[8] KISS = *Keep It Simple, Stupid*; DRY = *Don't Repeat Yourself.*

Suggestions et indices

- Le nombre de points à traiter **ne sera pas nécessairement un multiple du nombre de *threads***. Il vous faudra donc faire les calculs appropriés qui permettront d'assurer que les éléments — dans le cas de la répartition par blocs d'éléments adjacents — seront répartis le plus *uniformément* possible entre les différents *threads*.

- Assurez-vous de n'introduire qu'un seul niveau de parallélisme. En d'autres mots, vous devriez lancer une série de *threads* selon l'approche requise — méthodes `classifier_les_points_par_*` et `nouveaux_representants_par_*` — mais ensuite ces divers *threads* ne devraient pas lancer de nouveaux *threads* — sinon votre solution sur une machine telle que `japet`[9] ne sera pas efficace car trop de *threads* seront créés.

- Si vous avez besoin d'utiliser un verrou, alors **c'est que vous n'avez pas utilisé la bonne approche ☹**

- N'hésitez pas à introduire des méthodes auxiliaires — gardez votre code **DRY**, c'est-à-dire, évitez le plus possible le même code qui se répète.

- Je vous suggère fortement d'utiliser `git` pour gérer l'évolution de votre projet et éviter sa régression, puisque le code qui vous est fourni est déjà sous forme d'un dépôt local `git`. Donc, lorsque vous avez réussi à faire fonctionner une partie de votre code, faites un `commit`!

  Plus de détails sur `git` sont disponibles via la page suivante :

  `http://www.labunix.uqam.ca/~tremblay/INF5171/Liens/`

- N'espérez pas obtenir des accélérations tellement plus grandes que 5 ou 6 — et pas nécessairement pour toutes les versions!

  Si vous obtenez des accélérations nettement plus grandes que cela, contactez-moi : soit vous avez utilisé une/des astuces intéressante/s de programmation (??), soit (plus probablement!) votre version séquentielle est vraiment mauvaise, ce qui produit des accélérations plus fortes.

[9] Ou un MacBook, ou une machine Linux!

6 Compléments d'information

Les éléments qui suivent ne sont pas nécessaires pour la réalisation du devoir. Il s'agit plutôt de remarques **complémentaires**, pour mieux comprendre le problème du partitionnement en k groupes et l'heuristique des k-moyennes.

Un avantage de la méthode des k-moyennes est qu'elle est relativement simple à mettre à oeuvre — et à paralléliser. Elle possède toutefois quelques désavantages :

- La méthode suppose qu'on connait ou qu'on fixe *a priori* la valeur de k.

  Si k est trop petit, la classification ne sera pas assez fine, alors que si k est trop grand, on n'effectuera pas suffisamment de regroupement — aux cas limites, si $k = 1$, tous les points se retrouvent dans le même groupe, alors que si $k = n$, chaque point se retrouve seul dans son propre groupe ☹

  Il existe une approche heuristique relativement simple pour choisir une valeur appropriée pour k : Voir Remarque 2.

- La méthode peut ne pas produire une solution optimale.

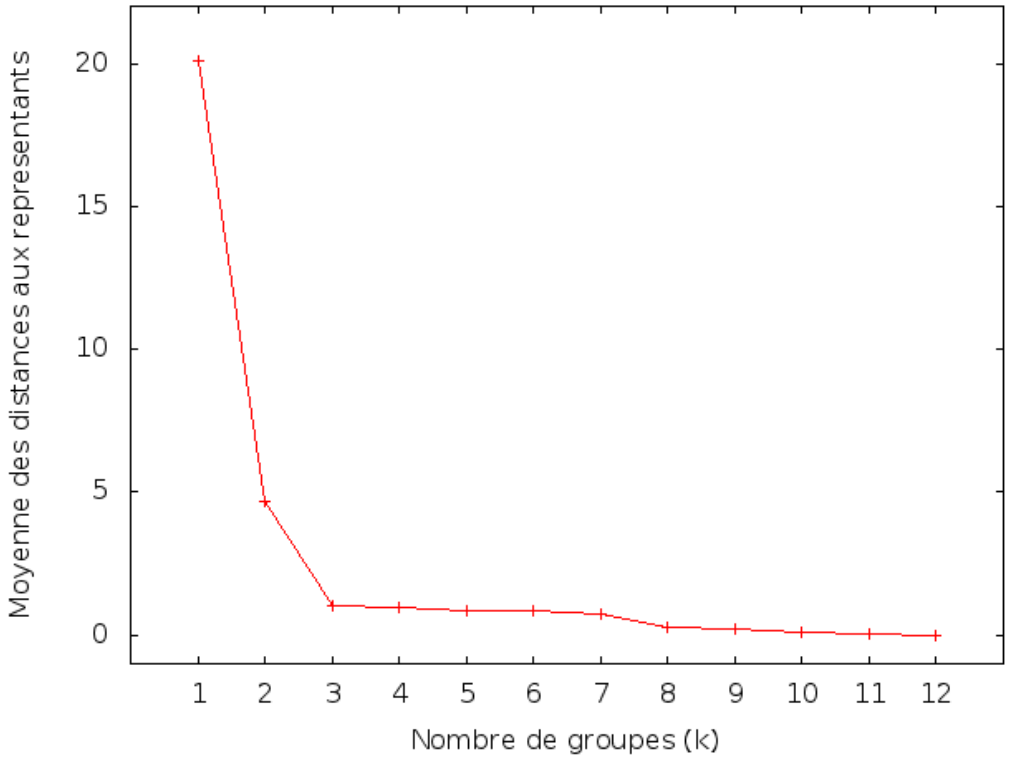
  Notamment, la solution obtenue dépend souvent du choix fait pour les représentants initiaux. C'est-à-dire que des choix différents de représentants initiaux peuvent produire des groupes résultants différents, et que ces groupes peuvent avoir des valeurs globales différentes.

  Un petit exemple (avec quatre points) est présenté dans le fichier `classificateur_spec.rb` : `"un exemple qui montre que les representants et les groupes varient selon l'ordre des elements"`. Dans cet exemple, le choix des (2) représentants initiaux se fait en prenant les deux premiers éléments distincts de la collection de données ; or, si on varie l'ordre de ces quatre élements, on arrive à des solutions différentes!

Complément d'information 1: Désavantages de la méthode des k-moyennes.

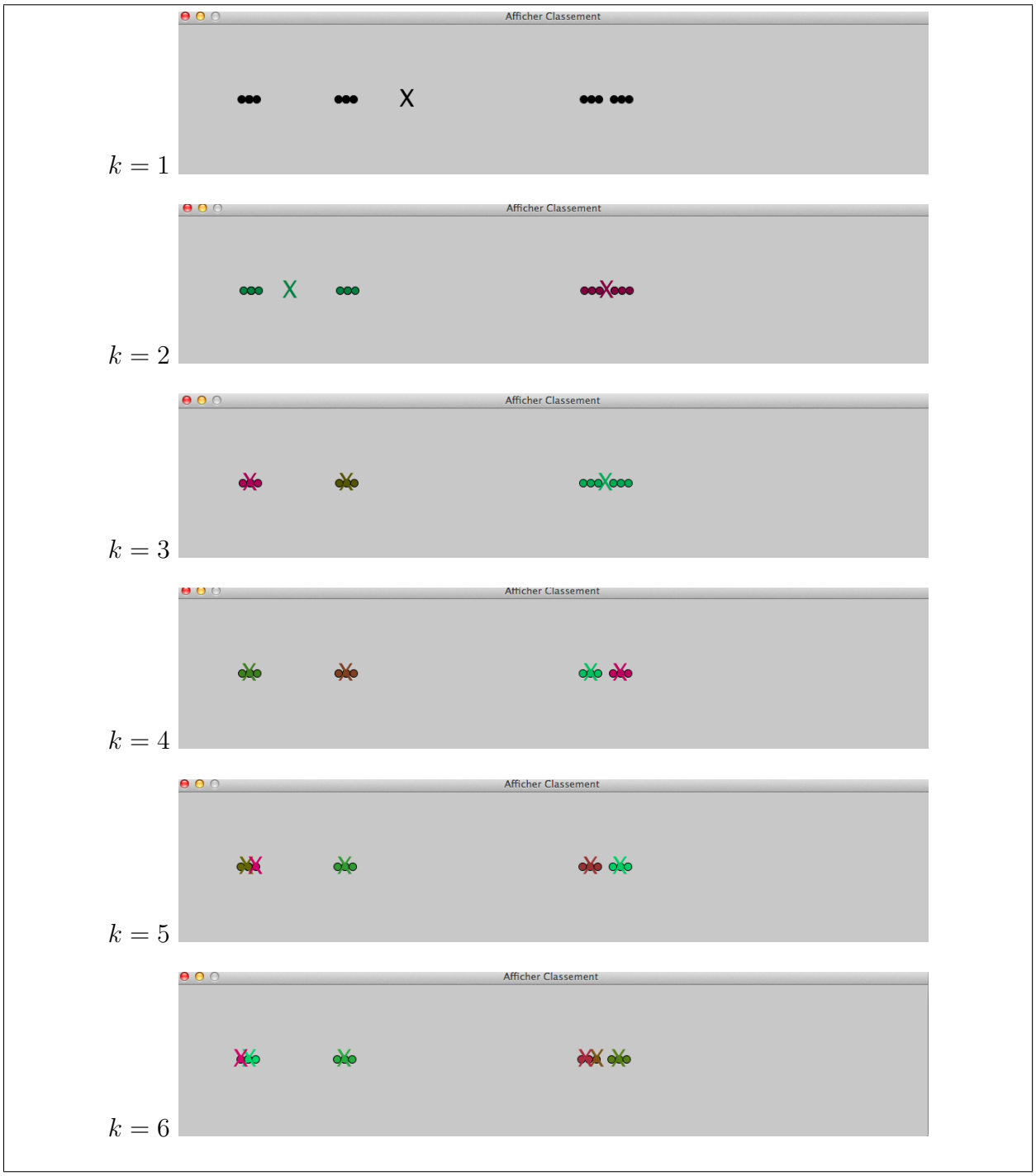
Il n'existe pas de règle précise pour choisir/déterminer k. Une heuristique souvent utilisée est présentée ci-bas. Les figures de la page 16 illustrent la classification des données sur les ages pour différentes valeurs de k allant de 1 à 6 — celles pour 7 à 12, utilisées dans la graphique ci-bas, ne sont pas illustrées.

On peut associer une valeur à chacune de ces classifications, valeur définie comme étant la moyenne de la distance entre chaque point et le représentant de son groupe. La figure ci-bas illustre l'évolution de cette valeur en fonction du nombre de groupes. On considère que le nombre préférable de groupes est celui avec le point d'inflexion dans la courbe, i.e., lorsque le taux de diminution de la valeur diminue, donc ici $k = 3$.



Graphique indiquant les différentes valeurs pour la moyenne des distances aux représentants pour différents k pour la classification des données sur les âges.

Complément d'information 2: Comment choisir une valeur k pas trop mauvaise.



Complément d'information 3: Solutions obtenues pour $k = 1, \ldots, 6$ pour les données sur les âges. Les solutions pour $k = 7, \ldots, 12$ ont été omises.

Tel qu'indiqué plus haut (et illustré dans un des cas de test), la solution obtenue par l'heuristique des k-moyennes dépend souvent des k représentants qui sont initialement choisis. Or, certains choix peuvent ne pas produire la meilleure solution.

Une stratégie souvent utilisée pour trouver la meilleure solution avec une plus grande probabilité est la suivante : *i*) on choisit les k représentants initiaux **de façon aléatoire** ; *ii*) on exécute l'heuristique et on prend note de la solution obtenue et de sa valeur. On répète alors ce processus **à plusieurs reprises**, pour ultimement sélectionner la meilleure solution obtenue après un certain nombre d'exécutions.

Complément d'information 4: Comment obtenir la solution optimale avec une plus grande probabilité.

Travail remis à Guy Tremblay

INF5171-20 : Devoir 1

À remettre au plus tard **Mardi, 17 octobre, 13h30** (début du cours)

Aucun travail en retard ne sera accepté!

NE PAS METTRE DANS UNE ENVELOPPE

Nom	
Prénom	
Code permanent	
Courriel	
Nom	
Prénom	
Code permanent	
Courriel	

Qualité générale du code (KISS, DRY)	/ 5
Respect du style Ruby	/ 5
`{classifier_les_points,nouveaux_representants}_seq`	/ 5
`{classifier_les_points,nouveaux_representants}_par_fin`	/ 5
`{classifier_les_points,nouveaux_representants}_par_fj_adj`	/ 5
`{classifier_les_points,nouveaux_representants}_par_fj_cyc`	/ 5
`{classifier_les_points,nouveaux_representants}_par_sta`	/ 5
`{classifier_les_points,nouveaux_representants}_par_dyn`	/ 5
Bonnes accélérations, graphiques (temps et accélération), etc.	/ 10
Réponses aux questions.	/ 5
Total	/ 55
Note globale	**/ 10**