Projet : Algorithme des k-moyennes

Par Alexandre PETIT,

le 04 janvier 2017

Licence Informatique, 2ème année

UFR des Sciences et Techniques de Rouen

Démarrage rapide

Deux programmes à lancer en mode interactif on été développé pour présenter le projet.

Le premier, `basicexample.ml` est un exemple d'utilisation **pratique** et **minimal** de l'application. Il est documenté pas à pas et montre comment, à partir de données fournies par l'utilisateur, il est possible d'obtenir une liste de multi-ensembles à l'aide de la bibliothèque `kmoy.cma` . La démarche nécessaire à visualiser le processus est ensuite explicitée.

Le programme `benchmark.ml` propose lui un aperçu plus large des possibilités de la bibliothèque. Plusieurs cas d'utilisation sont expérimentés et les cas limites sont notamment identifiés.

Description générale du programme

Le programme a été compilé sous la forme d'une bibliothèque, nommée `kmoy.cma`.

L'implantation du programme des k-moyennes repose autour de deux strucutures de données : celle des vecteurs de $\mathbb{R}^n$ et celle de multi-ensemble - implantée par des arbres binaires de recherche automatiquement équilibrés. Un troisième module tiens une place de choix dans le projet : le module kmoy qui opère le calcul des k-moyennes et manipule les deux structures de données susmentionnées.

Dans la suite de ce document, nous donnerons un aperçu générale de ces trois modules. Notons que l'interface fournie avec les modules documente de façon exhaustive et détaillée les méthodes, types et exceptions proposés par chaque module. Aussi, le style des commentaires est inspiré du code source d'Ocaml et compatible avec ocamldoc.

Nous nous atterderons en en particulier sur la description des principaux algorithme, afin notamment d'en analyser la complexité. Enfin à une exception près, justifiée et mentionnée par la suite, la récursivité des algorithmes est terminale.

Descriptions des modules et analyse des algorithmes principaux

Le module Kmoy

interface : *kmoy.mli*

implantation: *kmoy.ml*

Description

Le module kmoy implante l'algorithme des k-moyennes ainsi que ses fonctions auxiliaires. Il effectue à cet effet une paramétrisation du module Mset, polymorphe, sur le type Vector des vecteurs de $\mathbb{R}^n$.

Algorithmes

Notations pour la suite

- M un nuage de m points, dont $l \leq m$ sont distincts. M est implanté par un multi-ensemble.
- $N = [N_0, \ldots, N_k]$ un recouvrement de M constitué d'une liste de k sous multi-ensembles $N_0, \ldots, N_k$. Les N_j constituent des partitions de M. Afin de distinguer les multi-ensembles implantés par des arbre binaires et la liste des valeurs du multi-ensemble (avec lesquelles vont travailler les algorithmes auxiliires) qui sont eux implantés par des listes, ont utilisera parfois la notation suivante pour la seconde : $P = [P_0, \ldots, P_k]$.
- $\mu = [\mu_0, \ldots, \mu_k]$ la liste des barycentres associés aux partitions N_j (et donc aux aux P_j)

Calcul du barycentre

Algorithme : calcul du barycentre d'un ensemble vecteurs pondérés.

Reçoit : une liste de vecteurs pondérés de la forme $P_0 = [(u_0, q_0); \ldots; (v_l, q_l)]$ composé de l éléments, où les u_i sont des vecteurs de $\mathbb{R}^n$, avec $n = n_0$ fixé, et les q_i sont des valeurs entières qui représentent la pondération des vecteurs.

Transmet : le barycentre de ces vecteurs, sous la forme d'un vecteur de $\mathbb{R}^n$.

Contrainte sur les données : le cardinal de la liste doit être au minimum d'une unité.

Complexité : $\Theta(l)$.

Explicitons ce résultat :

- L'algorithme parcours les l éléments de la liste de façon séquentielle afin sommer les vecteurs et mémoriser le nombre total d'unités (soit la somme des pondérations) :

$$u = \sum_{i=0}^{n} (u_i \times m_i) \quad \text{où } \times \text{ désigne la multiplication d'un vecteur par un scalaire}$$
$$m = \sum_{i=0}^{n} m_i$$

- La valeur du barycentre est alors immédiate :

$$v_b = u \times (1/m)$$

L'algorithme présente ainsi une **complexité linéaire**.

Attention

On remarque cependant que l'algorithme présente une limite évidente. Sans contraintes préalables, le calcul de la somme des vecteurs peut rapidement engendrer un **débordement** de valeurs. Pour prévenir ce cas, il est sera recommandé de passer en argument une liste de **vecteurs normalisés**, c'est à dire dont les valeurs réelles des composantes sont comprises entre 0 et 1. Cette opération est à réaliser par l'utilisateur au préalable. Considérons max_{float} la valeur flottante maximale et max_{int} la plus grande valeur entière. L'emploi de vecteurs normalisés assure alors de calculer le barycentre de jusqu'à $q = min(\lfloor max_{float} \rfloor, max_{int})$ unités de vecteurs.

Calcul des partitions

Algorithme : partitionnement d'une liste de vecteurs pondérés, selon une liste de pivots

Reçoit :

- $\boldsymbol{elts}$*, les éléments à partitioner par une liste de vecteurs pondérés de la forme* $[(u_0, m_0); \ldots; (v_l, m_l)]$ *;*
- $\boldsymbol{\mu}$, les pivots, sous la forme d'une liste de vecteurs de $\mathbb{R}^{n_0}$.
- $\boldsymbol{d}$ *une fonction de distance ;*

Transmet : $\boldsymbol{Card(pvts)}$ *partitions qui constituent un recouvrement de l'ensemble des éléments de* $\boldsymbol{elts}$*.*

Contraintes sur les données : aucune

Complexité : $\boldsymbol{\Theta(lk)}$*, où* $\boldsymbol{l}$ *est le nombre de points distincts et* $\boldsymbol{k}$ *le nombre de partitions*.

L'algorithme procède de la façon suivante. Pour commencer, $\boldsymbol{k = Card(pvts)}$ listes vides sont crées $(*)$. Notons $\boldsymbol{P_j}$ ces listes; ce sont les partitions qui seront renvoyés à la fin.

Ensuite, pour chaque élément $\boldsymbol{u_i}$ de $\boldsymbol{elts}$: on recherche lequel des pivots de $\boldsymbol{pvts}$ est le plus proche de $\boldsymbol{u_i}$ $(**)$. On insère alors $\boldsymbol{u_i}$ dans la $\boldsymbol{n}$-ième partition, c'est à dire dans $\boldsymbol{P_n}$ $(***)$, où $\boldsymbol{n}$ correspond à l'indice du pivot qui est le plus proche de l'élément $\boldsymbol{u_i}$, d'après la distance calculée par $\boldsymbol{d}$.

Synthétisons : pour chaque élément de $\boldsymbol{elts}$,

$(*)$ la création de $\boldsymbol{k}$ listes vides est clairement $\boldsymbol{\Theta(k)}$;

$(**)$ de même, la recherche du pivot le plus proche est $\boldsymbol{\Theta(k)}$;

$(***)$ on admettra que l'insertion à la $\boldsymbol{n}$-ième position est elle aussi $\boldsymbol{\Theta(k)}$, ni plus, ni moins puisque l'algorithme reconstruit en faite la liste des partitions, de façon stable.

On rappel que $\boldsymbol{l}$ est le nombre d'éléments distincts de $\boldsymbol{elts}$. Par relation asymptotique, la compléxité de l'algorithme est donc $\boldsymbol{\Theta(l)\Theta(k) = \Theta(lk)}$, autrement dit **linéaire**.

Calcul de la dispersion

Algorithme : calcul de la dispertion.

Reçoit :

- $\boldsymbol{prts} = [N_0, \ldots, N_k]$*, une liste de* $\boldsymbol{k} \in \mathbb{N}\backslash\{0\}$ *partitions générés par la fonction* `partition` *;*
- $\boldsymbol{bars} = [\mu_0, \ldots, \mu_k]$ *la liste des barycentres respectifs ces partitions ;*
- $\boldsymbol{d}$ *une fonction de distance.*

Transmet : $\boldsymbol{V = \sum_{j=0}^{j=k} \sum_{u \in N_j} d(\mu_j, u)}$*, un nombre réel.*

Contraintes sur les données : aucune, si ce n'est que le nombre de partitions doit être égal au nombre de barycentres, ce qui est assuré puisque `dispersion` *est une fonction privée et que l'implantation respecte cette précondition.*

Compléxité : $\boldsymbol{\Theta(l)}$*, où* $\boldsymbol{l}$ *est le nombre de points distincts.*

L'algorithme procède selon un parcours séquentiel des données et présente une **compléxité linéaire**.

Calcul des k-moyennes

Algorithme : Calcul des k-moyennes.

Reçoit : un multi-ensemble M_{set} de m points de $\mathbb{R}^n, n \in \mathbb{N}$; un entier strictement positif $k \leq m$; une fonction de distance d prenant deux points et renvoyant un nombre réel positif.

Transmet : une liste de k multi-ensembles $N = [N_1, \ldots, N_k]$ telle que pour chaque multi-ensemble N_j la valeur $V = dispersion(N, \mu, d)$ est minimale.

Contraintes sur les données : $0 < k \leq m$ et il existe un entier naturel n_0 tel que $\forall c \in M_{set}, c \in \mathbb{R}^{n_0}$.

*Complexité: basée sur la complexité de `partition`, `barycentre`, `dispersion` et sur **la courbre de réduction de la dispersion**.*

Regardons de plus près le déroulement de cet algorithme.

Initialisation

Dans un premier temps, l'algorithme casse le multi-ensemble M_{set} pour travailler avec la liste de ses éléments. Cette opération, implantée par `Mset.tolist` procède en fait à un parcours infixe de la structure d'arbre AVL et renvoie la liste strictement croissante des éléments du multi-ensemble. Les éléments de la liste sont sous la forme de couples (c, m) où $c \in \mathbb{R}^n$ est un vecteur et $m \in \mathbb{N}^*$ désigne le nombre de vecteurs c présents dans le multi-ensemble. Dans toute la suite, l'algorithme travail avec des listes. Les sous-multi-ensembles ne sont reconstruits qu'une fois les partitions optimisées.

De la liste de éléments de M_{set}, on extrait ensuite les k premiers éléments à l'aide de la fonction `Listutils.slice`. Les structures de vecteur de ces éléments servent alors de pivots pour le premier partitionnement de M_{set}.

Raffinement

La première partition effectuée, l'algorithme procède alors de façon récursive au raffinement des partitions de N. En prenant comme nouveaux pivots les barycentres des partitions, la valeur V décroit strictement à chaque étape, jusqu'à se stabiliser. L'algorithme est alors terminé.

Analyse de la complexité

Si l'on met de côte le problème d'analyse que pose la diminution de V. L'algorithme séquence des algorithmes dont on a montré que la complexité est au pire linéaire. Notons g la compléxité de la courbe de réduction de la dispersion. Par relation asymptotique, il s'ensuit que la complexité de l'algorithme est produit d'une complexité linéaire par g.

Conclusion de l'analyse

L'algorithme esquisse une compléxité linéaire. Cependant cette analyse nous a permis de relever deux facteurs qui impactent directement le cout :

- l le nombre d'éléments distincts présent dans l'ensemble de départ ;
- k le nombre de partitions à éffectuer.

Le module Mset

interface : *mset.mli*

implantation : *mset.ml*

Description

Ce module implante la structure de multi-ensemble sur des arbres binaires de recherche automatiquement équilibrés (AVL). L'implantation est polymorphe est fait usage des functor afin d'être spécialisée selon les besoins.

La hauteur de chaque sous-arbre est mémorisée en vue d'optimiser les opérations de maintiens de l'équilibre. Aussi, la fonction d'ajout n'est pas récursive terminale. Cependant, la hauteur de l'arbre est logarithmique par rapport à sa taille, la hauteur de pile est donc minimale. En effet, pour un multi-ensemble de 4 Milliards d'éléments distincts, la pile n'atteind qu'une hauteur de 64 dans le cas d'un ajout en bout de chemin, c'est à dire dans le pire des cas.

On remarquera que seules les fonctions nécessaires au projet ont été implantés. Pour une implantation complète, il restera à définir la fonction *remove*, accompagnée des les fonctions auxiliaires sous-jacentes. Les fonctions de parcours prefixe et postfixe semblables à celle du parcours infixe (ici nommé *tolist*) peuvent également être implantés.

Le module Vector

interface : *vector.mli*

implantation : *vector.ml*

Description

Le module vector propose une implantation succinte d'une structure de vecteur de $\mathbb{R}^n$. Afin que la dimension soit dynamique, le choix a été fait d'employer la structure récursive de liste.

En plus des générateurs de base *add*, *sub* et *scale* . Le module contient certaines fonctions qui sont relativement spécifiques au projet, comme le calcul de la distance entre deux points, la recherche du point le plus proche parmis un ensemble de points, la génération d'un vecteur aléatoire, et l'affichage sur la sortie standard.

Problèmes rencontrés

La difficulté majeur de ce projet a été de comprendre et d'utiliser les functor afin de paramétrer le multi-ensemble.

Divers

[re]compiler le projet

méthode dépreciée :

Pour recompiler la bibliothèque, éxécuter les commandes suivante depuis le dossier source `src` :

```
ocamlc -c vector.mli;
ocamlc -c vector.ml;
ocamlc -c mset.mli;
ocamlc -c mset.ml;
ocamlc -c listutils.mli;
ocamlc -c listutils.ml;
ocamlc -c kmoy.mli;
ocamlc -c graphics.cma listutils.cmo vector.cmo mset.cmo kmoy.ml;

ocamlc -a vector.cmo mset.cmo listutils.cmo graphics.cma kmoy.cmo;
```

nouvelle méthode :

Depuis le dossier `src` lancer la commande `make` puis `make install`

La bibliothèque est alors compilée et installée. Il est dès lors possible de faire appel au modules contenus dans l'archive par l'appel aux instructions suivantes dans le *toplevel* :

```
#use "topfind";;
#require "kmoy";;
```

ressource :

http://pleac.sourceforge.net/pleac_ocaml/packagesetc.html

Découvertes

- L'utilisation de la directive #use afin de charger un fichier .ml dans le toplevel.
- ocamlc -i pour générer l'interface par défaut d'un module.

Ressources

Le site officiel d'Ocaml propose une riche documentation du language et de l'utilisation qu'il s'en fait. Pour ce qui est de l'utilisation, la partie tutoriels présente plusieurs chapitre très intéressants. Aussi, de nombreux exemples sont fournis et permettent de rapidement saisir chaque concept.

http://ocaml.org/learn/tutorials/

Pour ce qui est du language, pur et dur, la documentation des bibliothèques standards sont d'une aide remarquable. Elle s'avère être un outils de choix pour parvenir au bon développement du projet.

http://caml.inria.fr/pub/docs/manual-ocaml/libref/index.html

Enfin, pour la compilation en *bytecode* du projet :

http://caml.inria.fr/pub/docs/manual-ocaml/comp.html