Travaux pratiques n°2
Programmation Orienté Objet pour le Calcul Scientifique

Sylvain Desroziers

Prérequis

Pour mener à bien ces travaux, il est nécessaire d'avoir sur sa machine :

* les outils nécessaires pour le TP n°1 (un compilateur `C++`, `CMake` et `Make`),
* la bibliothèque d'algèbre linéaire `Eigen`.

1 Présentation

Dans ce TP, nous allons tenter d'améliorer l'implémentation de notre algorithme de Machine Learning développé dans le TP n°1. L'idée est ici de définir une bibliothèque minimaliste d'algèbre linéaire visant à gérer les matrices et vecteurs sous-jacentes à l'algorithme de régression. D'une part, cela nous permettra de gérer des données de tailles quelconques. D'autre part, nous allons être vigilant quant à la performance de notre bibliothèque et mesurer l'impact de certains choix de l'orienté objet. Un compte-rendu **individuel** de ce TP devra être produit. Les différents éléments du compte-rendu (code source, rapport au format PDF) sont personnels, doivent être soignés et devront être envoyés à l'adresse `sylvain.desroziers@ifpen.fr`. Il est autorisé de travailler à plusieurs mais les **collègues de travail doivent être cités** dans les rapports.

La commande `tree` appliquée au répertoire `ml` du TP n°2 donne le résultat suivant :

```
desroziers> tree ml
ml
|-- CMakeLists.txt
|-- ref.cpp
|-- test_algebra.cpp
|-- test_performances.cpp

0 directories, 4 files
```

Votre travail dans ce TP va consister à compléter certains de ces fichiers et à en créer d'autres. Dans ce TP et comme dans le TP précédent, la compilation sera effectuée en utilisant les outils `CMake` et `Make`. Pour partir sur de bonnes bases, le fichier `ref.cpp` fourni contient une implémentation demandée dans la partie 2.2 du TP n°1. Ce fichier n'est pas à modifier et doit être la base de vos développements.

1. Si vous n'avez pas été au bout de ce TP, il convient d'étudier au préalable le fichier `ref.cpp` et de bien comprendre l'algorithme.
2. Créer un répertoire `build` et y compiler le projet en utilisant `CMake`. Lancer le binaire `ref`. Les résultats attendus des autres implémentations à effectuer dans ce TP devront être similaire.

```
Pour une femme d'une taille de 150cm, on prédit un poids de 47.426kg
Pour une femme d'une taille de 175cm, on prédit un poids de 75.0952kg
Pour un homme d'une taille de 168cm, on prédit un poids de 76.1764kg
Pour un homme d'une taille de 195cm, on prédit un poids de 106.059kg
```

2 Pour commencer, le vecteur

Votre travail dans cette partie va consister à créer un outil informatique pour décrire les vecteurs utilisés dans l'algorithme. Par exemple, dans notre algorithme de régression, $y \in \mathbb{R}^m$ est un vecteur de flottant de taille m (pouvant être grand) et $\theta = \{\theta_0, \theta_1, \theta_2\} \in \mathbb{R}^3$ est un vecteur de taille 3. Ici, on ne considère que des vecteurs de flottants de type `double`. Le but de cette partie est de construire une classe `ml::vector`, brique de base de notre bibliothèque d'algèbre linéaire. Pour cela, nous allons créer un fichier header `vector.h`, nous pourrons ainsi réutiliser les développements.

Les fonctionnalités souhaitées des objets de la classe `ml::vector` sont les suivantes :

- Construction à partir d'une taille :
  - ⋆ `vector(const std::size_t)`
- Accesseur à la taille :
  - ⋆ `std::size_t size() const`
- Accesseurs aux coefficients :
  - ⋆ `double& operator[](const std::size_t)`
  - ⋆ `const double& operator[](const std::size_t) const`
- Opérateurs d'affectation :
  - ⋆ `vector& operator=(const vector&)`
  - ⋆ `vector& operator=(double)`
- Opérateurs d'affectation composée :
  - ⋆ `vector& operator+=(const vector&)`
  - ⋆ `vector& operator-=(const vector&)`
- Opérateurs algébriques :
  - ⋆ `vector operator-(const vector&)`
  - ⋆ `vector operator+(const vector&)`
  - ⋆ `vector operator*(double)`
  - ⋆ `vector operator*(double, const vector&)`
- Opérateurs numériques :
  - ⋆ `auto dot(const vector&) const`
  - ⋆ `auto squaredNorm() const`
  - ⋆ `auto sum() const`
- Opérateur d'affichage :
  - ⋆ `std::ostream& operator<<(std::ostream&, const vector&)`

3. Créer le fichier `vector.h` et implémenter une classe `vector` répondant aux fonctionnalités précédents. Utiliser le namespace `ml`. Vous devrez utiliser les assertions pour protéger des mauvaises utilisations (par exemple les débordements d'indices). Ne pas oublier de protéger votre fichier de l'inclusion infinie.
4. Utiliser le fichier `test_algebra.cpp` pour tester votre implémentation.

Considérer le code suivant :

```
ml::vector v(3); // à remplir !!

// Que se passe t'il ici :
std::cout << "u = " << (v + v + 2.0 * v) << std::endl;
```

5. Combien de fois le constructeur `vector(const std::size_t)` est il appelé ? Qu'en pensez-vous ?
6. Utiliser le fichier `test_performance.cpp` pour tester la performance de votre implémentation. Typiquement, il convient de tester les opérateurs numériques et algébriques. Pour cela, vous pourrez utiliser la bibliothèque standard :

```
#include <chrono>

auto start = std::chrono::system_clock::now();
// code à intrumenter ...
auto end = std::chrono::system_clock::now();

auto elapsed =
  std::chrono::duration_cast<std::chrono::milliseconds>(end-start).count();

std::cout << "time = " << elapsed << " ms" << std::endl;
```

3 Pour continuer, matrice et transposée par recopie

De manière similaire, on souhaite construire un outil informatique pour décrire les matrices de notre algorithme. Par exemple, dans notre algorithme, $X \in \mathbb{R}^{m\times 3}$ est une matrice de m lignes et 3 colonnes. Ici, on ne considère que des matrices de flottants de type `double`. Le but de cette partie est de construire une classe `ml::matrix`, autre brique de base de notre bibliothèque. Pour cela, nous allons créer un fichier header `matrix.h`, nous pourrons ainsi réutiliser les développements.

Les fonctionnalités souhaitées des objets de la classe `ml::matrix` sont les suivantes :

- Construction à partir de tailles :
  - ⋆ `matrix(const std::size_t, const std::size_t)`
- Accesseurs à la taille :
  - ⋆ `std::size_t size_x() const`
  - ⋆ `std::size_t size_y() const`
- Accesseurs aux coefficients :
  - ⋆ `double& operator()(const std::size_t, const std::size_t)`
  - ⋆ `const double& operator()(const std::size_t, const std::size_t) const`
- Opérateurs d'affectation :
  - ⋆ `matrix& operator=(double)`
- Opérateurs algébriques :
  - ⋆ `vector operator*(const vector& v) const`
- Opérateur d'affichage :
  - ⋆ `std::ostream& operator<<(std::ostream&, const matrix&)`
- Accesseur à la transposée :
  - ⋆ `matrix transpose() const`

Ici, la méthode `transpose()` crée une nouvelle matrice. Il y a donc une allocations mémoire et une recopie *par transposition* de la matrice.

7. Créer le fichier `matrix.h` et implémenter une classe `matrix` répondant aux fonctionnalités précédentes. Utiliser le namespace imbriqué `ml::version1`. Vous devrez utiliser les assertions pour protéger des mauvaises utilisations (par exemple les débordements d'indices). Ne pas oublier de protéger votre fichier de l'inclusion infinie. Prenez soin de bien implémenter l'accès à la transposée en créant et initialisant une nouvelle instance de la classe `matrix`.
8. Utiliser le fichier `test_algebra.cpp` pour tester votre implémentation.
9. Utiliser le fichier `test_performance.cpp` pour tester la performance de votre implémentation. Typiquement, il convient de tester les opérateurs algébriques. Pour cela, vous pourrez comparer à une implémentation hors de la classe sur des tableaux classiques. Quelle est votre conclusion ?

4 Retour sur la régression

On peut maintenant essayer notre implémentation d'algèbre linéaire dans notre algorithme de régression issu du TP n°1 disponible dans le fichier `ref.cpp`.

10. Créer un fichier `ml.cpp` à partir de `ref.cpp`. Modifier l'algorithme défini sur des tableaux pour utiliser les structures algébriques définies dans la partie précédente.
11. Instrumenter la descente de gradient en utilisant 5000 itérations.
12. Déplacer les données dans un fichier `data.txt`. Dans ce fichier, on pourra définir au préalable un premier entier décrivant le nombre d'enregistrements (i.e. les lignes de la matrice) et un deuxième pour le nombre de caractéristique (i.e. les colonnes). La dernière caractéristique pourra être par convention la caractéristique à prédire. Implémenter le chargement des données par lecture de fichier.

```
#include <ifstream>

std::ifstream file("data.txt");

int m, int size;
file >> m >> size;
```

Nous avons maintenant une version de notre algorithme de régression complètement générique.

5 Pour continuer, matrice et transposée par héritage

On se propose maintenant d'améliorer la performance de la méthode `transpose()` de la classe `matrix` introduite dans la partie 3. En particulier, on souhaite supprimer l'allocation et recopie de la matrice et utiliser directement la matrice. L'idée générale est de se baser sur un pattern de programmation de type façade (ou proxy) pour résoudre ce problème. Pour cela, on va employer une approche objet classique (et naive) par héritage. On considère avoir deux types différents, un pour la matrice (définie dans la partie 3.) et un pour sa transposée `transpose_matrix` qui sera une façade à la matrice. Comme une matrice et sa transposée sont des *matrices*, on propose dans un premier temps d'implémenter un classe abstraite `matrix_base` qui sera la super-classe des deux types de matrices. Ainsi, une matrice et sa transposée seront des matrices de type `matrix_base`. Les fonctionnalités de la matrice de base sont :

- Accesseurs virtuels à la taille :
  - ⋆ `virtual std::size_t size_x() const = 0`
  - ⋆ `virtual std::size_t size_y() const = 0`
- Accesseurs virtuels aux coefficients :
  - ⋆ `virtual double& operator()(const std::size_t, const std::size_t) = 0`
  - ⋆ `virtual const double& operator()(const std::size_t, const std::size_t) const = 0`
- Opérateurs d'affectation :
  - ⋆ `matrix_base& operator=(double)`
- Opérateurs algébriques (non virtuels) :
  - ⋆ `vector operator*(const vector& v) const`
- Opérateur d'affichage (non virtuels) :
  - ⋆ `std::ostream& operator<<(std::ostream&, const matrix_base&)`

Ainsi, les opérateurs algébriques et d'affichage sont implémentés dans la super-classe et mutualisés par les classes filles. C'est une factorisation de code intéressante.

13. Créer le fichier `matrix.h` et implémenter une classe `matrix_base` répondant aux fonctionnalités précédentes. Utiliser le namespace `ml`. Vous devrez utiliser les assertions pour protéger des mauvaises utilisations (par exemple les débordements d'indices). Ne pas oublier de protéger votre fichier de l'inclusion infinie.
14. Implémenter la classe `matrix` héritant de la classe `matrix_base` et disposant des fonctionnalités manquantes de la partie précédente (sauf l'accès à la transposée pour le moment). Utiliser le namespace imbriqué `ml::version2`.
15. Implémenter la classe `transpose_matrix` héritant également de la classe `matrix_base`. Pour simplifier, la classe sera imbriquée dans la classe `matrix`. La classe `transpose_matrix` contiendra une référence vers un objet `matrix` comme attribut pour aider à son implémentation. Utiliser le namespace imbriqué `ml::version2`.
16. Implémenter finalement l'accès à la transposée de la classe `matrix`.
17. Utiliser le fichier `test_algebra.cpp` pour tester votre implémentation.
18. Utiliser le fichier `test_performance.cpp` pour tester la performance de votre implémentation. Typiquement, il convient de tester les opérateurs algébriques. Pour cela, vous pourrez comparer à une implémentation hors de la classe sur des tableaux classiques. Quelle est votre conclusion ?
19. Utiliser votre nouvelle implémentation dans le fichier `ml.cpp`. Pour cela, les fonctions devront s'adapter aux différentes types de matrices. On pourra factoriser le code des fonctions par l'utilisation de la programmation générique par template (voir le cours). Pour chaque fonction, on utilisera un paramètre template pour la matrice et éventuellement le vecteur. Par exemple :

```
template<typename M, typename V>
void gradientDescent(const M& X, const V& y, V& theta, double alpha, int iteration)
```

6 Version de matrice optimisée

Nous allons traiter le problème de performance de la version de matrice par héritage. Le but est de supprimer les méthodes virtuelles afin d'améliorer les performances. Ici, on accepte de supprimer la partie abstraite `matrix_base` de la partie précédente. D'un point de vue modélisation, on perd donc un élément de concept. Ainsi, il n'y aura plus de relation entre les classes `matrix` et `transpose_matrix`. Pour information, il convient de partir de l'implémentation effectuée dans la partie 3. et intégrer une classe imbriquée pour la transposée comme dans la partie 5.

20. Implémenter dans le fichier `matrix.h` une classe `matrix` optimisée dans le namespace imbriqué `ml::version3` (à faire cette fois-ci sans héritage) disposant de fonctionnalités de la partie 3. Il conviendra d'imbriquer une classe `transpose_matrix` (sans héritage encore) s'appuyant encore sur la classe `matrix` pour son implémentation (comme dans la partie 5).
21. Utiliser le fichier `test_algebra.cpp` pour tester votre implémentation.
22. Evaluer la performance en instrumentant le même code de test que dans la question 12. Que concluez-vous ?
23. Utiliser votre nouvelle implémentation dans le fichier `ml.cpp`.

7 Mon dernier conseil, ne pas réinventer la roue...

Par curiosité, nous allons maintenant utiliser une librairie d'algèbre linéaire ayant la même interface d'utilisation que les outils que nous avons construit dans ce TP. Ainsi, la généricité apportée par les templates introduite dans la partie précédente permettra d'utiliser directement cette bibliothèque. La librairie à utiliser est `Eigen`, `voir http://eigen.tuxfamily.org`, basée sur la programmation générique pour le calcul à hautes performances.

24. Télécharger et installer le répertoire `Eigen` dans vos sources.
25. Utiliser cette librairie dans le fichier `ml.cpp` et comparer les performances avec les différentes versions implémentées jusqu'ici. Que concluez-vous ?

8 Bonus pour ceux qui veulent

Une des différences de performance provient du produit matrice vecteur. On peut utiliser une librairie spécifique pour l'implémenter de manière performante, la librairie `blas` pour *Basic Linear Algebra Subprograms*. Voir `http://www.openblas.net/` par exemple. Attention, c'est une librairie Fortran mais `cblas.h` fournit une interface C.

26. Télécharger et installer la librairie `blas`.
27. Remplacer votre produit matrice vecteur par un appel à la librairie `blas` et instrumenter les performances. Qu'en pensez-vous ?
28. D'autres opérations sont certainement substituables par des appels à `blas`. Etudier l'amélioration de votre classe de vecteur.

Enfin, il est aussi possible de paralléliser en utilisant les threads de la bibliothèque standard...