Antoine Pradier
Axel Marchand
Ralph Saidi

ENSAE 3ème année
Éléments logiciels pour le traitement des données massives

Comparaison de performances : produit matriciel, régression linéaire LASSO, forêts aléatoires

Table des matières

Introduction

Sur la base de tests de performance de différentes méthodes utilisant Cython pour le produit vectoriel, nous avons implémenté diverses méthodes de produit matriciel, de régression linéaire LASSO ainsi que de forêt aléatoire. Pour chacun de ses trois algorithmes, nous présentons les temps de calcul d'éxécution sur des données générées aléatoirement.

Nous nous sommes basés sur la base du package *td3a_cpp*. Nous avons laissé le code déjà rédigé intact mais nous avons rajouté les nouvelles fonctions de la façon suivante en les rajoutant sous forme de package dans le fichier *td3a_cpp*. Nous avons en conséquence modifié les imports dans *setup.py*.

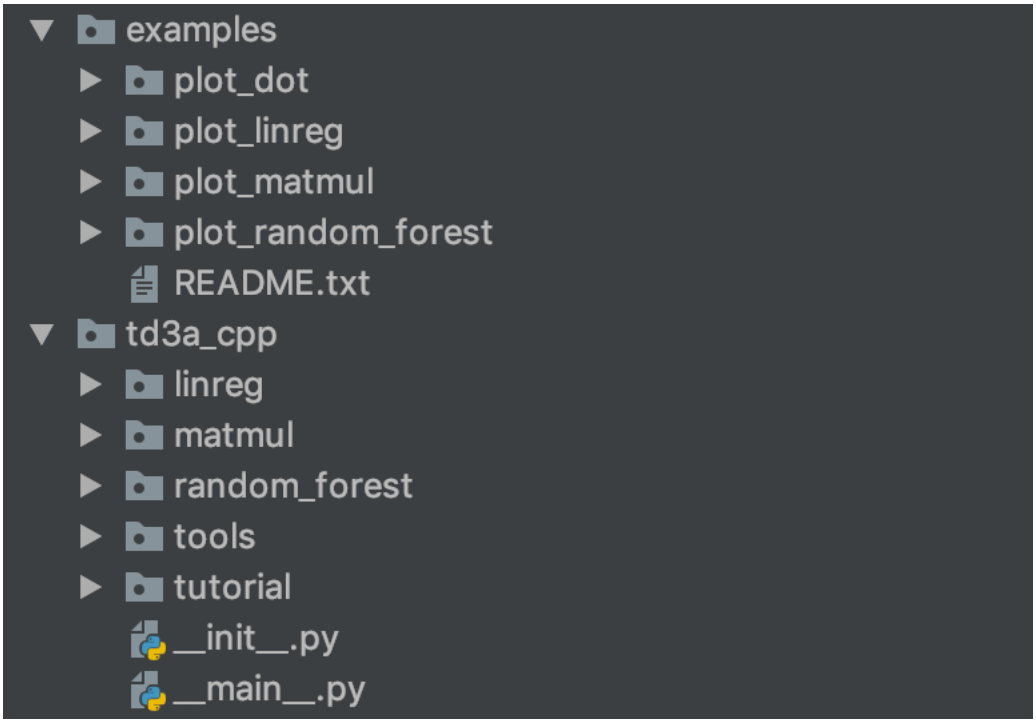


FIGURE 1: Arborescence du projet td3a_cpp

1 Produit terme à terme

1.1 Expression mathématique

Soit $A, B \in (\mathbb{R}^{d\times d})^2$, le produit matriciel est définit comme la matrice C,

$$C_{i,j} = A_{i,j} \times B_{i,j}$$

1.2 Implémentations

Nous avons retenu plusiers implémentations pour le produit matriciel :

- une méthode naïve directement codée sous Python (`matmulpy.py`)
- trois méthodes utilisant Cython (`matmulcython.pyx`)
- la méthode proposée par Numpy

1.3 Résultats

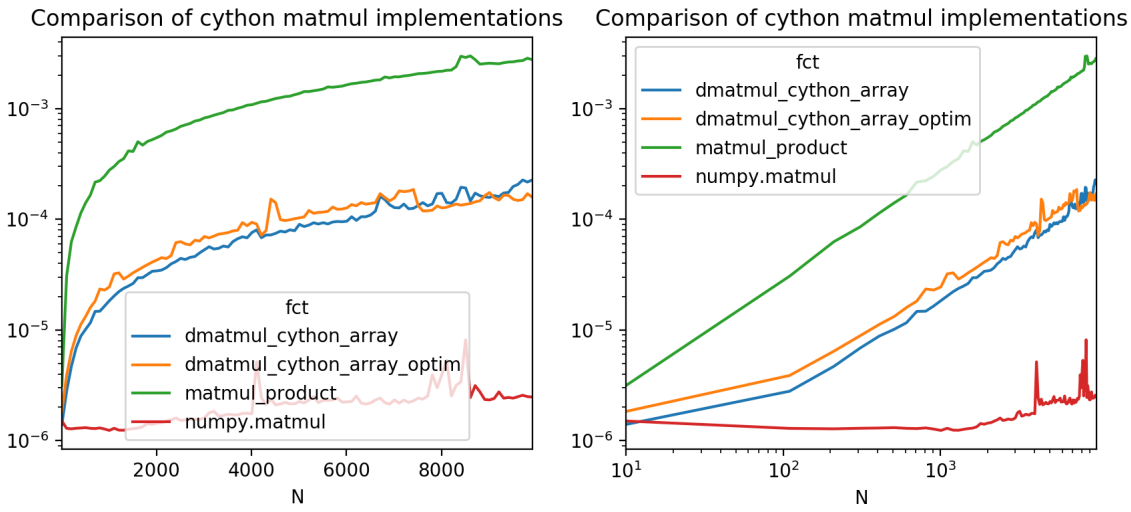


FIGURE 2: Résultats obtenus pour la multiplication terme à terme dans le cas double

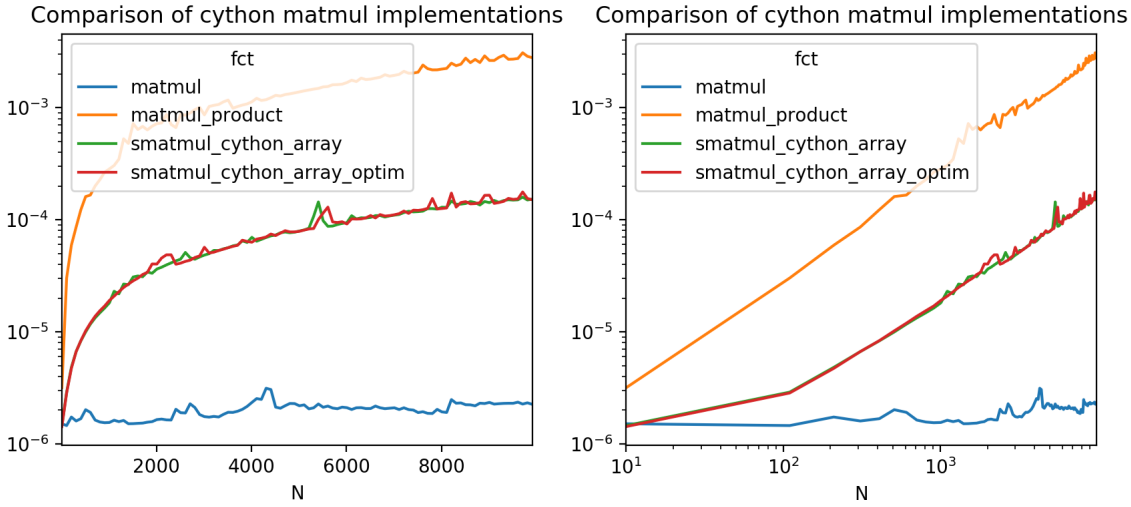


FIGURE 3: Résultats obtenus pour la multiplication terme à terme dans le cas float

On peut voir qu'aucune de nos méthodes ne parvient à approcher les performances de numpy. Même si elles sont plus rapides que celles obtenues dans le cas naïf.

2 Régression linéaire

2.1 Expression mathématique

On cherche à exprimer nos données sous la forme

$$y = \sum_{i=1}^{p} \beta_i X$$

avec $X \in \mathbb{R}^p$. On effectue donc une régression Lasso en minimisant la somme résiduelle des carrés

$$\sum_{i=1}^{n} \left(y_i - \sum_{j=0}^{p} \beta_j X_{ij}\right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

On détermine le coefficient en effectuant une coordinate descent de la manière suivante à l'aide de la fonction $S()$ suivante :

$$\begin{cases} \theta_j = \rho_j + \lambda & \text{for } \rho_j < -\lambda \\ \theta_j = 0 & \text{for } -\lambda \leqslant \rho_j \leqslant \lambda \\ \theta_j = \rho_j - \lambda & \text{for } \rho_j > \lambda \end{cases} \quad (1)$$

Et ensuite, on répète jusqu'à convergence ou nombre maximum d'itérations

1. Pour $j = 0, 1, ..., n$
2. On calcule $\rho_j = \sum_{i=1}^{m} x_j^{(i)}(y^{(i)} - \sum_{k \neq j}^{n} \beta_k x_k^{(i)}) = \sum_{i=1}^{m} x_j^{(i)}(y^{(i)} - \hat{y}_{pred}^{(i)} + \beta_j x_j^{(i)})$
3. On pose $\beta_j = S(\rho_j, \lambda)$

2.2 Implémentations

Pour la régression linéaire LASSO, nous avons retenu les implémentations suivantes :

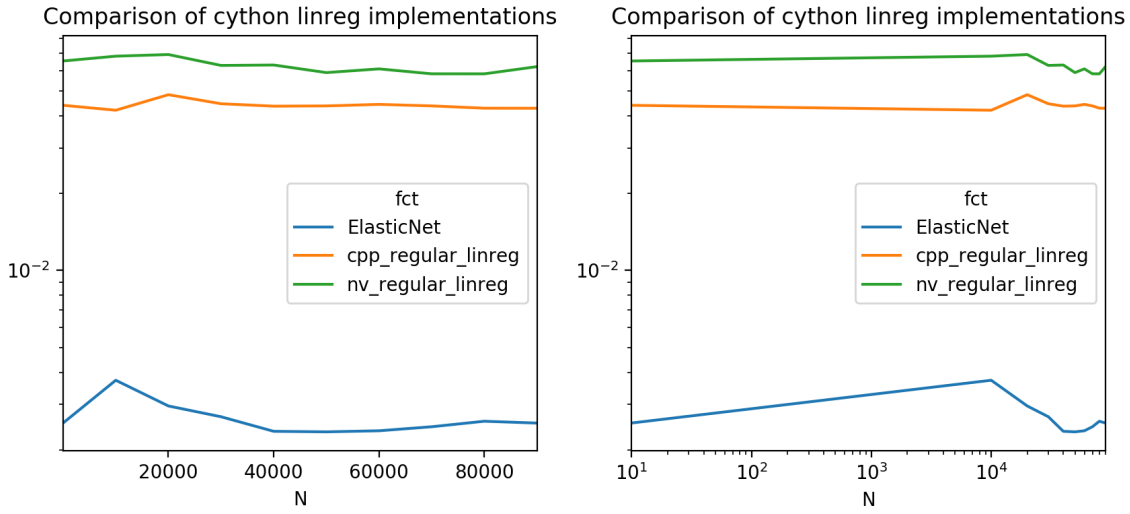
• une méthode naïve directement codée avec des fonctions Python et Numpy (`nv_regular_linreg.py`)

• une méthode Multi-Processing en utilisant **ProcessPoolExecutor** (`mp_py_regular_linreg.py`)

• la méthode `ElasticNet` proposée par scikit-learn

• une méthode utilisant Cython. Notre méthode utilisant cython consiste en un fichier `cy_regularized_linreg_.cpp`, dans lequel la logique est implémentée, qui est ensuite appelée dans les fichiers (`cy_regularized_linreg.pyx`; `cpp_py_regular_linreg.py`)

2.3 Résultats



On voit bien que notre implémentation est très inférieure à celle de sklearn, même si elle obtient des meilleurs résultats que celle naïve. On note également que nous avons du enlever la méthode multiprocessing de nos calculs, celle ci prenant en effet beaucoup trop de temps.

3 Forêt aléatoire

3.1 Expression mathématique

Pour notre dernière expérience, nous avons décidé de nous intéresser à la fonction random forest de sklearn. Nous ne détaillerons pas intégralement la méthode, mais il nous faut définir plusieurs fonctions :

— $test_split : splitthedataset$

3.2 Implémentations

Pour la random forest, nous avons retenu les implémentations suivantes :

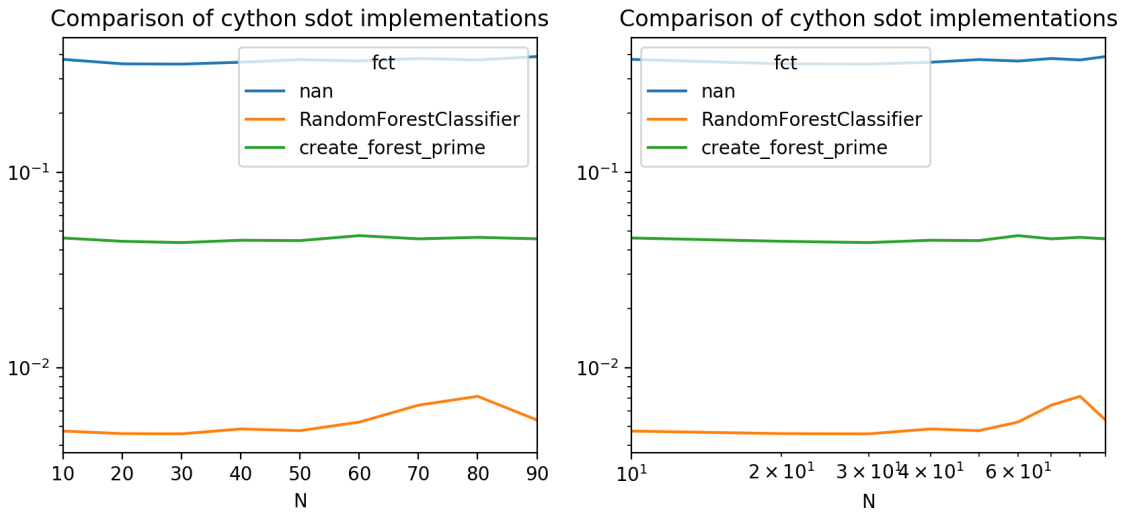
- une méthode naïve directement codée avec des fonctions Python

(`nv_random_forest.py`)

• la méthode `RandomForestClassifier` proposée par scikit-learn

• une méthode utilisant Cython, la fonction *build_forest* a été recodée en prenant avantage des types. Nous avons utilisé l'implémentation issue de ce lien github. Nous l'avons ensuite adaptée et comparée à notre méthode naïve. (`random_forest.pyx`)

3.3 Résultats



Notre implémentation est une nouvelle fois bien meilleure que celle naïve. Cependant elle reste largement inférieure à celle de sklearn qui est beaucoup plus performante.

4 Conclusion

Nous avons testé sur trois types d'algorithme la capacité de Cython à accélérer les calculs, souvent longs lorsqu'on utilise seulement des fonctions Python.

Dans chaque cas, nous pouvons constater qu'utiliser la librairie Cython permet une amélioration nette de la vitesse de calcul par rapport à des méthodes naïves directement codé avec Python. En utilisant les bibliothèques numpy et scikit-learn, nous avons choisi

de comparer les vitesses de calcul des algorithmes utilisant Cython à celles des techniques déjà implémentées et largement utilisées. La simple utilisation de Cython ne permet pas d'atteindre la rapidité des méthodes de ces bibliothèques qui ont subi un certain nombre d'optimisation.