Parallélisation de l'algorithme KNN avec Cython

Hashley Ramanankatsoina, Ivanna Savonik

Eléments logiciels pour le traitement des données massives
ENSAE - Institut Polytechnique de Paris
2022

Keywords: Cython, Python, Parallélisation, KNN, Machine Learning

RÉSUMÉ

Dans ce rapport, nous avons étudié le gain de la parallélisation (*thread-based parallelism*) de l'algorithme de classification KNN en utilisant Cython. Les résultats expérimentaux montrent que la version Cython a un bon effet d'accélération du temps d'exécution en comparaison de l'implémentation de l'algorithme en Python, surtout lorsque la taille du jeu de données augmente. Nous obtenons un *speedup* moyen de 75.93.

1 INTRODUCTION

Le *Machine Learning*, apprentissage automatique en français, est un type d'Intelligence Artificielle qui donne aux ordinateurs la capacité d'"apprendre", sans intervention ou assistance humaine, en leur fournissant une grande quantité de données. Avec l'augmentation constante du volume de données disponibles, pouvoir traiter les données efficacement est un enjeu majeur dans le domaine du *Machine Learning*.

L'algorithme des K plus proches voisins (KNN) est un algorithme classique de *Machine Learning* dont le coût du calcul de distance sur lequel il est fondé augmente fortement lorsque la taille du jeu de données à traiter augmente. Afin d'améliorer le temps d'exécution de l'algorithme initialement sous Python, son implémentation en Cython peut être une solution. Cython est une extension du langage Python. Sa puissance provient de la manière dont il combine Python et C. L'intérêt de mélanger ces deux langages vient du fait qu'ils sont complémentaires : Python est un langage dynamique et interprété, tandis que C est un langage compilé à typage statique et plus rapide.

2 KNN

Développé par Evelyn Fix et Joseph Hodges, ce modèle utilise la "similarité des caractéristiques" pour prédire les valeurs de toutes nouvelles observations. Cela signifie qu'une valeur est attribuée à la nouvelle observation en fonction de sa ressemblance avec ses K plus proches voisins. Quand on parle de voisin, cela implique la notion de distance. Une métrique de distance couramment utilisée pour les variables continues est la distance euclidienne.

Dans notre cas, nous utilisons KNN dans un contexte de la classification binaire. Avec un entier K donné et une observation des données de test $\boldsymbol{x}_0 \in \mathbb{R}^d$, cet algorithme cherche dans les données d'entraînement les K points les plus proches de $\boldsymbol{x}_0$. Pour trouver cette "proximité", plusieurs mesures de distance existent. Par exemple, la distance euclidienne entre une observation $\boldsymbol{x} \in \mathbb{R}^d$ et la nouvelle observation à prédire est la suivante :

$$D_i(\boldsymbol{x}_i, \boldsymbol{x}_0) = \sqrt{\sum_{j=1}^{d} (x_{ij} - (x_0)_j)^2}$$

où $i \in 1...n$.

Les K observations les plus proches sont sélectionnées, c'est-à-dire les K observations dont les distances avec la nouvelle observation sont les plus petites. Dans un contexte de classification, la classe la plus représentée par ces K observations sera la prédiction du modèle (vote majoritaire).

Dans un premier temps, nous avons développé le code Python de l'algorithme KNN. A partir de cette base, nous avons ensuite développé l'équivalent du KNN en Cython afin de tester l'efficacité de la parallélisation au niveau du temps d'exécution.

3 PARALLÉLISATION

La parallélisation est possible grâce au CPU (*central processing unit*) de notre ordinateur. Le CPU est le circuit électronique qui exécute les instructions d'un programme informatique. Il est composé de cœurs et de threads. Les cœurs sont des composants matériels physiques, tandis que les threads sont des composants virtuels qui correspondent à une séquence d'instructions qu'un cœur doit exécuter.

Après inspection de l'algorithme KNN, il est clair que la distance entre les points du jeu de données de test et ceux du jeu de données d'entraînement peut être calculée de manière indépendante. Cela fait de notre boucle impliquant ce calcul un candidat idéal pour la parallélisation. La parallélisation que nous avons implémentée consiste donc à paralléliser les calculs de distance du KNN. Pour ce faire, nous avons exploité les fonctionnalités de multithreading de Cython afin d'accéder au parallélisme basé sur les threads (*thread-based parallelism*). La fonction Cython "prange" permet de transformer des boucles for, initialement en série, en boucles parallèles sur plusieurs threads et exploitant tous les cœurs de CPUs disponibles (par défaut "prange" utilise autant de threads qu'il y a de cœurs disponibles). Nos données dans la boucle sont alors divisées en parties égales, et chaque partie est associé à un thread.

Il est important de préciser que la parallélisation n'implique pas que le temps d'exécution va être réduit proportionnellement au nombre de cœurs utilisés. Cela est dû au fait qu'il faut aussi prendre en compte le temps d'initialisation (qui prend le plus de temps dans notre cas), ainsi que le temps de communication entre les threads (celui-ci est peu élevé en comparaison avec le temps de communication entre CPUs — des clusters HPC plus traditionnels — puisque la distance est plus petite).

4 JEU DE DONNÉES

Dans le but de comparer les résultats du KNN avec Cython (C-KNN) et KNN avec Python (P-KNN) dans un contexte de classification, nous avons utilisé le générateur de données de la librairie Scikit-Learn ”make_classification”. Ces jeux de données artificielles nous permettent de faire varier facilement leurs nombres d’observations afin de tester la robustesse de nos deux algorithmes.

5 RÉSULTATS EXPÉRIMENTAUX

Nous présentons sur la Figure 1 la comparaison du temps d’éxécution de C-KNN vs P-KNN en fonction de la taille du jeu de données initial (sachant que 25% a été utilisé comme données de test). Le Tableau 1 présente le détail du temps d’exécution et le *speedup*.

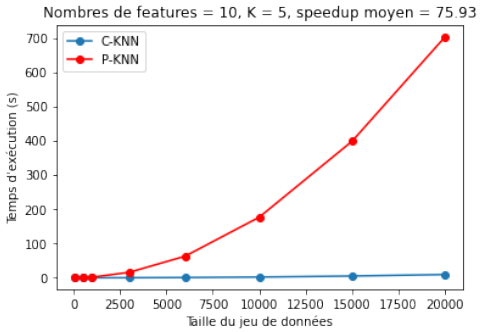

FIGURE 1. Evolution du temps d’exécution en fonction de la taille de l’échantillon pour C-KNN et P-KNN.

Table 1. Le temps de l’exécution et le speedup en fonction de la taille de l’échantillon pour C-KNN et P-KNN.

Taille du jeu de données	Temps d’éxecution (s)		
	P-KNN	C-KNN	Speedup
1 000	1.69	0.02	77
3 000	15.96	0.20	80
6 000	62.88	0.79	80
10 000	176.94	2.19	81
15 000	398.80	5.21	77
20 000	703.25	9.50	75

D’après la Figure 1, nous constatons que plus la taille de l’échantillon est grande, plus la différence entre les deux temps d’exécution est grande. Nous pouvons aussi voir que la courbe associée à C-KNN est presque plate donc le temps d’exécution augmente très lentement avec la taille de l’échantillon. Cette même dynamique est visible lorsque nous avons fait varier vers la hausse les dimensions du jeu de données (*features*).
Le Tableau 1 confirme cette analyse. Le *speedup* apporté par l’utilisation du C-KNN est en moyenne autour de 75.93.

La Figure 2 présente les résultats de la différence de vitesse d’exécution entre fonction Cython sans parallélisation (Cyt-KNN) et avec (C-KNN).

D’après la Figure 2, nous pouvons voir que le *speedup* moyen est de 1.34. Bien que la différence soit légère, le temps d’exécution

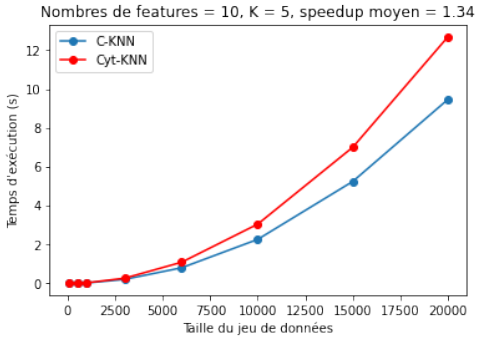

FIGURE 2. Evolution du temps d’exécution en fonction de la taille de l’échantillon pour C-KNN et Cyt-KNN.

a bien accéléré. Ce *speedup* peut potentiellement augmenter avec la parallélisation de la partie où les distances sont triées dans le KNN.

6 CONCLUSION

Dans cette étude, nous proposons une mise en œuvre rapide de l’algorithme des K plus proches voisins (KNN) en utilisant plusieurs threads (multithreading). Nous avons montré que l’utilisation de cette parallélisation accélère le temps d’exécution d’un *speedup* moyen de 75.93. En particulier, cette amélioration permet de réduire la restriction de taille généralement nécessaire lorsque nous décidons d’utiliser le KNN.

Une piste intéressante à exploiter serait de paralléliser la partie sélectionnant les K plus proches voisins car elle inclut du tri. Cela devrait nous permettre de réduire davantage le temps d’exécution. Une autre suggestion serait d’utiliser des algorithmes de tri de différentes complexités afin de voir comment le temps d’exécution évolue (dans notre cas le ”np.argsort” utilise ”quicksort” comme algorithme de tri).

RÉFÉRENCES

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn : Machine learning in Python. *Journal of Machine Learning Research*, 12 :2825–2830, 2011.

Kurt W. Smith. *Cython*. O’Reilly, 2015.

Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython : The best of both worlds. *Computing in Science & Engineering*, 13(2) :31–39, 2011.

GeeksforGeeks. What are threads in computer processor or cpu ?

Princeton University. Cos 511 : Theoretical machine learning.

MITOPENCOURSEWARE. Lecture 1 k-nearest neighbor algorithms for classification and prediction.

J L Hodges Evelyn Fix. Discriminatory analysis : nonparametric discrimination, consistency properties.