Traitement des Données Distribuées

Multithreading & Multiprocessing

BRY Nathan
VONIN Cédric

Mai 2021

Introduction

Dans un monde où les volumes annuels de données créées, capturées, copiése et consommées augmentent de façon exponentielle (ils doublent environ tous les 3 ans selon le site *Statista*) et où la Loi de Moore (selon laquelle la capacité de calcul fdes processeurs double tous les ans) n'est plus aussi vraie que dans les années 1990, il est impératif de trouver de nouveaux moyens afin de pouvoir traiter ces données, qui sont clés dans la stratégie de nombreuses entreprises. Le parallélisme et le calcul distribué se sont alors imposés comme 2 solutions très efficaces afin d'augmenter la puissance de calcul à notre disposition sans dépendre des limitations technologies. En tant qu'étudiants aspirant à des rôles de Data Scientists ou autres experts de la donnée, il nous est de plus en plus essentiel de nous former à ces notions et de les maîtriser. Nous étudierons dans ce rapport 2 concepts liés au parallélisme : le multithreading et le multiprocessing. Fondamentalement, le multiprocessing et le threading sont deux façons de réaliser un calcul parallèle, en utilisant respectivement des processus et des threads comme agents de traitement. Nous décrirons d'abord le fonctionnement théorique de ces technique avant d'ensuite les implémenter avec le langage *Python*.

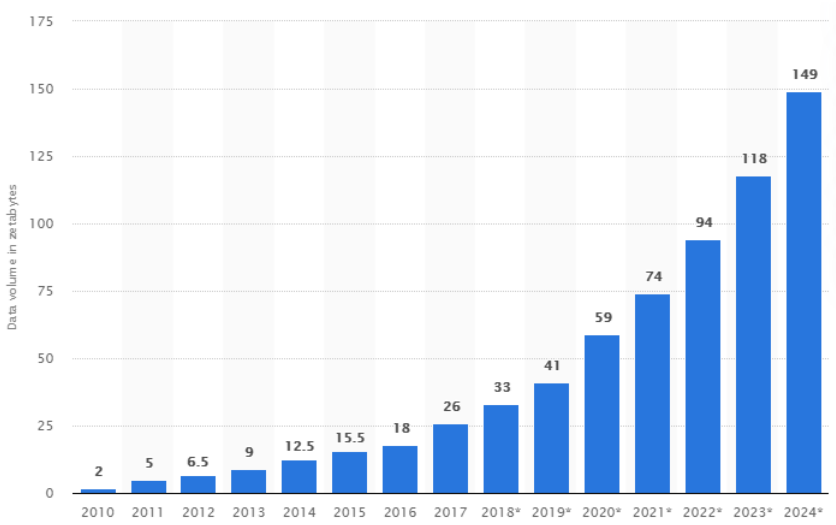

Figure 1: Evolution des volumes de données dans le monde

1 Multithreading

En informatique, un thread d'exécution est la plus petite séquence d'instructions programmées qui peut être gérée indépendamment par un planificateur, qui fait généralement partie du système d'exploitation. L'implémentation des threads et des processus diffère selon les systèmes d'exploitation, mais dans la plupart des cas, un thread est un composant d'un processus. Plusieurs threads peuvent exister au sein d'un processus, s'exécutant simultanément et partageant des ressources telles que la mémoire, alors que différents processus ne partagent pas ces ressources. En particulier, les threads d'un processus partagent son code exécutable et les valeurs de ses variables allouées dynamiquement et des variables globales non locales au thread à tout moment.

Dans l'architecture des ordinateurs, le multithreading est la capacité d'un CPU (ou d'un seul cœur dans un processeur multicœur) à fournir plusieurs threads d'exécution simultanés, pris en charge par le système d'exploitation. Cette approche diffère du multiprocessing. Dans une application multithread, les threads partagent les ressources d'un ou de plusieurs cœurs, qui comprennent les unités de calcul, les caches de CPU et la translation lookaside buffer (TLB).

Si les systèmes de multiprocessing comprennent plusieurs unités de traitement complètes dans un ou plusieurs cœurs, le multithreading vise à augmenter l'utilisation d'un seul cœur en utilisant le parallélisme au niveau des threads d'exécution. Comme les deux techniques sont complémentaires, elles sont combinées dans presque toutes les architectures de systèmes modernes avec des CPU multithreading multiples et avec des CPU avec des cœurs multithreading multiples.

2 Multiprocessing

En informatique, un processus est l'instance d'un programme informatique qui est exécutée par un ou plusieurs threads. Il contient le code du programme et son activité. Selon le système d'exploitation (OS), un processus peut être composé de plusieurs threads d'exécution qui exécutent des instructions simultanément.

Le multiprocessing est un mode de fonctionnement dans lequel deux ou plusieurs processeurs d'un ordinateur traitent simultanément deux ou plusieurs parties différentes du même programme (ensemble d'instructions). Le multiprocessing est généralement effectué par deux microprocesseurs ou plus, chacun d'entre eux étant en fait un CPU sur une seule petite puce. Les superordinateurs combinent généralement des milliers de microprocesseurs de ce type pour interpréter et exécuter les instructions. Au niveau du système d'exploitation, le multiprocessing est parfois utilisé pour désigner l'exécution de plusieurs processus simultanés dans un système, chaque processus s'exécutant sur un CPU ou un noyau distinct, par opposition à un seul processus à un instant donné.

Le principal avantage du multiprocessing est la vitesse, et donc la capacité à gérer de plus grandes quantités d'informations. Comme chaque processeur d'un tel système est affecté à une fonction spécifique, il peut accomplir sa tâche, transmettre le jeu d'instructions au processeur suivant et commencer à travailler sur un nouveau jeu d'instructions.

3 Les données

Afin d'implémenter ces 2 concepts, nous avons choisi de façon presque aléatoire un dataset. Notre seul critère était que le dataset contienne une quantité non négligeable de données (au minimum plusieurs dizaines de milliers d'instances) afin que l'entraînement de modèles et le GridSearch utilisés ne soient pas instantanés.

Nous avons ainsi utilisé un dataset extrait de Kaggle *Health Insurance Cross Sell Prediction*[1]. Une compagnie d'assurance indienne qui a fourni une assurance maladie à ses clients souhaite construire un modèle pour prédire si les souscripteurs de l'année précédente seront également intéressés par l'assurance automobile proposée par la société.

Ainsi, construire un modèle pour prédire si un client serait intéressé par l'assurance automobile est extrêmement utile pour la compagnie d'assurance car elle peut alors planifier sa stratégie de communication pour atteindre ces clients et optimiser son modèle économique et son chiffre d'affaire. Nous disposons pour cela d'informations sur l'assuré (sexe, âge, code régional), son véhicule (âge du véhicule, dommages) et sa police d'assurance souscrite (prime, moyen de contact).

Le véritable objectif de ce rapport étant d'illustrer les 2 concepts de parallélisme, nous ne décriverons que très brievement les données utilisées.

3.1 Généralités

Les données consistent en 381 109 instances des onze variables suivantes (features), associées au label à prédire :

'id'	Variable unique d'identification de l'assuré
'Gender'	Sexe de l'assuré
'Age'	Âge de l'assuré
'Driving_License'	L'assuré possède le permis de conduire
'Region_Code'	Code d'identification de la région
'Previously_Insured'	L'assuré est déjà couvert par une assurance automobile
'Vehicle_Age'	Âge du véhicule de l'assuré
'Vehicle_Damage'	Le véhicule de l'assuré a déjà subi des dommages par le passé
'Annual_Premium'	Prime annuelle payée par l'assuré (en roupies)
'Policy_Sales_Channel'	Code représentant le moyen utilisé pour contacter l'assuré
'Vintage'	Ancienneté de l'assuré auprès de la société d'assurance (en jours)
'Response'	Variable à prédire : l'assuré est intéressé par l'offre d'assurance automobile

L'analyse exploratoire révèle que 88% des instances appartiennent à la classe 0 : ces individus ne sont pas intéressés par l'assurance proposée. Nous sommes donc face à un problème de forte asymétrie des classes (*"Imbalanced Classification Problem"*). La classe d'intérêt est celle des individus intéressés par l'offre d'assurance. Elle est ici minoritaire et sera donc difficile à prédire.

[1] https://www.kaggle.com/anmolkumar/health-insurance-cross-sell-prediction

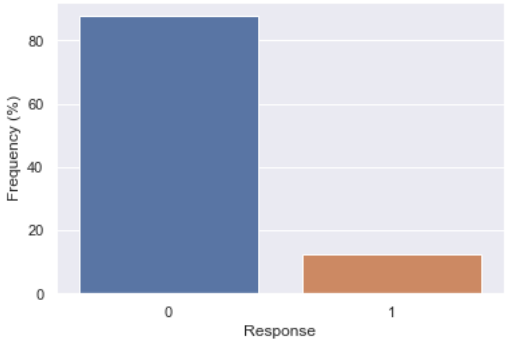

Figure 2: Fréquence des classes

3.2 Undersampling

Après avoir exploré nos données et fait le preprocessing nécessaire (notamment l'encoding de certaines features), nous avons séparé nos données en un jeu d'entraînement et un autre de test selon le ratio 80-20 (avec l'attribut *stratify* afin de conserver la répartition des classes).

Afin de remédier à ce problème de dataset déséquilibré, nous avons recours au sous-échantillonnage (Undersampling), qui consiste à supprimer certaines observations de la classe majoritaire pour équilibrer les données. *RandomUnderSampler* (RUS) est l'une des techniques de sous-échantillonnage, qui choisit au hasard certains échantillons de la classe majoritaire et les supprime. Nous appliquons ainsi cette technique sur notre training set (nous ne touchons jamais au testing set) afin de passer d'une répartition 88%-12% à une répartition 50%-50% (37368 échantillons de chaque classe).

4 Implémentation

Une fois notre dataset undersamplé, nous avons choisi un modèle à implémenter afin d'illustrer les concepts de multithreading et multiprocessing.

4.1 eXtreme Gradient Boosting (XGBoost)

Nous avons choisi la méthode XGBoost, méthode d'agrégation de modèles qui est une implémentation optimisée de l'algorithme d'arbres de boosting de gradient. Le boosting de gradient est un algorithme d'apprentissage supervisé dont le principe est de combiner les résultats d'un ensemble de modèles plus simples et plus faibles afin de fournir une meilleur prédiction.
Contrairement au Random Forest, cet algorithme travaille de manière séquentielle. L'algorithme commence par construire un premier modèle qu'il va évaluer. A partir de cette première évaluation, chaque individu va être alors pondéré en fonction de la performance de la prédiction. Cette approche permet à l'algorithme de s'améliorer par capitalisation par rapport aux exécutions précédentes.

Le parallélisme dans le boosting de gradient peut être mis en œuvre dans la construction d'arbres individuels, plutôt que dans la création d'arbres en parallèle comme la forêt aléatoire. En effet, dans

le boosting, les arbres sont ajoutés au modèle de manière séquentielle. La rapidité de XGBoost réside à la fois dans l'ajout du parallélisme dans la construction d'arbres individuels et dans la préparation efficace des données d'entrée pour accélérer la construction des arbres. En fait, la librairie *xgboost* comporte un paramètre appelé ***nthread***, qui permet d'utiliser plusieurs thread en parallèle afin de run un XGBoost.

4.2 Multithreading

Afin de d'abord illustrer l'impact d'une stratégie de multithreading, nous avons donc entraîné un XGBoost en spécifiant à chaque fois le nombre de threads utilisés, avec des valeurs allant de 1 à 6. Pour chaque modèle, nous avons mesuré le temps (en secondes) pris pour le training.

Figure 3: Vitesse de training du XGBoost vs nombre de threads utilisés

Bien que les vitesses de training soient relativement faibles (de l'ordre de la dizaine de secondes), on mesure assez bien l'impact du multithreading sur les durées de training. En effet, en passant de 1 à 6 threads utilisés on peut presque diviser par 3 le temps de training (de 16 à 6 secondes).

Essayons maintenant de mettre en place des opérations plus conséquentes afin de voir si on observe un impact encore plus fort.

4.3 GridSearchCV

La fonction *GridSearchCV* de la librairie *scikit-learn* est une méthode permettant de trouver les paramètres optimaux d'un modèle. A partir d'une grille contenant plusieurs valeurs pour chaque paramètre recherché, la fonction va entraîner et évaluer (selon un critère spécifié) une version du

modèle pour chaque combinaison de valeurs des paramètres en question, avec une technique de Cross-Validation (on divise le dataset en k subsets, entraîne le modèle sur $k-1$ subsets et teste le modèle sur le dernier, et ce pour chaque combinaison de subsets).

En raison du grand nombre de modèles entraînés puis testés sur un grand nombre de combinaisons de subsets, l'opération *GridSearchCV* est relativement coûteuse en terme de temps d'exécution. Cependant cette tâche est parallélisable. En effet, l'argument ***n_jobs*** de la fonction *GridSearchCV* utilisée pour évaluer un modèle sur un ensemble de données en utilisant la k-fold Cross Validation permet de spécifier le nombre de tâches parallèles à exécuter.
Par défaut, cette valeur est fixée à 1, mais elle peut être fixée à -1 pour utiliser tous les cœurs du processeur de notre système, ce qui est une bonne pratique.

4.4 Multiprocessing & Multithreading

Nous allons donc implémenter un GridSearchCV avec notre modèle XGBoost, ce qui permettra de combiner les possibilités de parallélisation de ces 2 techniques. Nous allons mesurer les temps d'exécution dans 4 situations différentes :

1. Multithreading désactivé pour le XGBoost et Multiprocessing désactivé pour le GridSearchCV
2. Multithreading désactivé pour le XGBoost et Multiprocessing activé pour le GridSearchCV
3. Multithreading activé pour le XGBoost et Multiprocessing désactivé pour le GridSearchCV
4. Multithreading activé pour le XGBoost et Multiprocessing activé pour le GridSearchCV

Voici les résultats observés. On constate à nouveau l'impact conséquent du multithreading et du multiprocessing. En effet, si notre *GridSearchCV* prend à l'origine 1h pour tourner sans aucun parallélisme, on remarque que l'utilisation du multithreading pour notre modèle (comme dans l'exemple vu plus haut) permet de diviser par 2 le temps d'exécution. Cependant, le multiprocessing mis en place grâce à l'argument ***n_jobs = -1*** de la fonction *GridSearchCV*, permettant d'utiliser tous les coeurs de notre processeur, a un impact encore plus important puisqu'il permet de diviser notre temps d'exécution par environ 6. En effet, lorsqu'on parallélise uniquement le GridSearch notre temps d'exécution est de seulement 8 minutes, et de 10 minutes lorsqu'on parallélise le modèle ET le GridSearch.

Implémentation	Vitesse (secondes)
1. Single Thread XGBoost, Single Job GridSearchCV	3658
2. Single Thread XGBoost, Parallel Job GridSearchCV	492
3. Parallel Thread XGBoost, Single Job GridSearchCV	1573
4. Parallel Thread XGBoost, Parallel Job GridSearchCV	584

Conclusion

Nous avons pu voir un aperçu de l'impact des méthodes de parallélisme telles que le **Multithreading** et le **Multiprocessing** grâce à une implémentation relativement simple en Python. En effet, sans avoir eu à installer d'autres packages / librairies que ceux dont nous avions initialement besoin et en n'utilisant que les arguments nativement intégrés à un modèle parallélisable (le *XGBoost*) et à une fonction elle aussi parallélisable (*GridSearchCV*), nous avons pu aller jusqu'à diviser notre temps d'exécution par presque 7 (**de 1h à 8mn**). L'impact de telles stratégies serait peut-être encore plus impactantes sur de plus gros volumes de données, jusqu'au point où une seule machine n'est plus suffisante. Des méthodes de calcul distribué, telles *Hadoop et MapReduce* pourraient alors être pertinentes afin de répartir la charge sur plusieurs machines.

References

[1] *Calcul distribué*. URL: https://fr.wikipedia.org/wiki/Calcul_distribu%C3%A9.

[2] *Data Volumes Worldwilde*. URL: https://www.statista.com/statistics/871513/worldwide-data-created/.

[3] *GridSearchCV*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.

[4] *Loi de Moore*. URL: https://fr.wikipedia.org/wiki/Loi_de_Moore.

[5] *Multiprocessing*. URL: https://fr.wikipedia.org/wiki/Multiprocesseur.

[6] *Multithreading*. URL: https://fr.wikipedia.org/wiki/Multithreading.

[7] *Parallélisme*. URL: https://fr.wikipedia.org/wiki/Parall%C3%A9lisme_(informatique).

[8] *XGBoost*. URL: https://xgboost.readthedocs.io/en/latest/.