UQAC

Devoir 1 - Apprentissage automatique pour le BigData

Classification distribuée par arbre de décision

Université du Québec
à Chicoutimi

Benoît Faure FAUB20060100
Octobre 2023

Table des matières

Table des figures

1 Synthèse de l'article

L'article présente l'algorithme MR-Tree[2], une méthode de construction d'arbres de décision en parallèle destinée à traiter de vastes ensembles de données à l'aide du modèle de programmation MapReduce s'exécutant sur la plate-forme Apache Hadoop. L'objectif principal de cette approche est de résoudre le défi de l'apprentissage d'arbres de décision sur des ensembles de données massifs. Il est expliqué que les technologies comme MapReduce et Apache Hadoop sont essentielles pour le traitement parallèle des données massives. L'algorithme MR-Tree est conçu pour avoir une excellente extensibilité en fonction de la taille de l'ensemble de données.

L'article décrit en détail la structure de l'algorithme MR-Tree. Il se compose de trois sections principales : le contrôleur, la fonction d'induction d'arbre (ID3) et la taille de l'arbre. Le contrôleur gère les paramètres d'entrée, exécute la fonction d'induction d'arbre récursive et la taille de l'arbre. La fonction ID3 est responsable de la croissance de l'arbre de décision, tandis que la tâche de la taille de l'arbre est de réduire les arbres pour éviter le surajustement. Les résultats expérimentaux montrent que l'algorithme MR-Tree affiche une scalabilité linéaire avec la taille de l'ensemble de données, grâce à des fonctions de map et de réduction efficaces avec un faible coût d'E/S.

En conclusion, l'article met en avant MR-Tree comme une approche évolutive pour la construction d'arbres de décision sur de vastes ensembles de données grâce à l'utilisation du modèle de programmation MapReduce sur la plate-forme Apache Hadoop. Il souligne toutefois un potentiel ralentissement lors de la manipulation d'arbres très volumineux et suggère des axes de recherche futurs pour améliorer l'algorithme en limitant la croissance de l'arbre.

2 Expérience 1 - Surcharge

2.1 Mode local

2.1.1 Méthodologie

Pour cette expérience en mode local, le code a été développé au sein d'un environnement Jupyter Notebook, avec la gestion de l'environnement Python et de ses dépendances assurée par Poetry. Pour gérer les données, les DataFrames PySpark ont été utilisés. En ce qui concerne les données, j'ai utilisé le dataset "Adult"[1] mentionné dans l'article.

Les données ont été chargées à l'aide des algorithmes de création de DataFrames PySpark. Ensuite, un prétraitement des données a été effectué. Les étapes suivantes ont été suivies :

1. Remplissage des valeurs manquantes (NA) avec des zéros.
2. Identification des colonnes de caractéristiques et de la colonne d'étiquettes.
3. Identification des valeurs catégorielles.
4. Indexation des valeurs catégorielles à l'aide de StringIndexer pour les colonnes de caractéristiques catégorielles.
5. Transformation de l'ensemble de données en un DataFrame PySpark final, prêt pour l'apprentissage automatique.

Pour l'entraînement, j'ai utilisé le modèle de classification par arbre de décision de MLlib, qui implémente la solution MapReduce pour paralléliser l'expansion de l'arbre de décision, comme décrit dans l'article. À chaque itération, le nombre de données dans le DataFrame a été augmenté en copiant les données prétraitées. Les performances, mesurées sous forme de temps d'entraînement (dt), ont été enregistrées pour chaque itération, puis sauvegardées dans un fichier CSV en vue d'une analyse ultérieure.

La figure 1 montre les jobs exécutés et en cours d'exécution sur mon ordinateur.

2.1.2 Résultats

La figure 2 montre comment le scale factor du temp d'entrainment augmente en fonction du nombre de données quand le programme est lancé sur 1 coeur de ma machine. Le rapport entre le temp d'entrainement et le temp d'entrainement initial augmente et se rapproche du linéaire le plus la taille du jeu de donnés augmente. Cette difference avec les résultats de l'aticle est du au fait que le calcul ne peut pas etre distribue pour utilise les avantages de l'aglorithme propose dans l'article en distribuant les calculs.

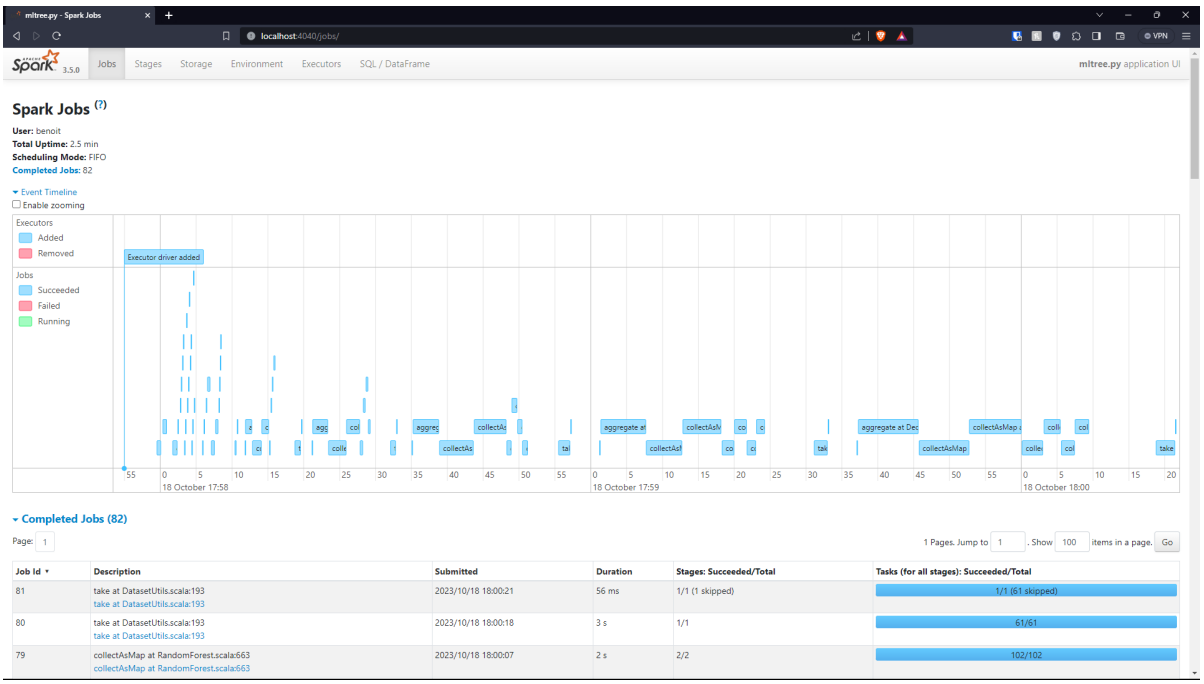

FIGURE 1 – SparkUI affichant les jobs

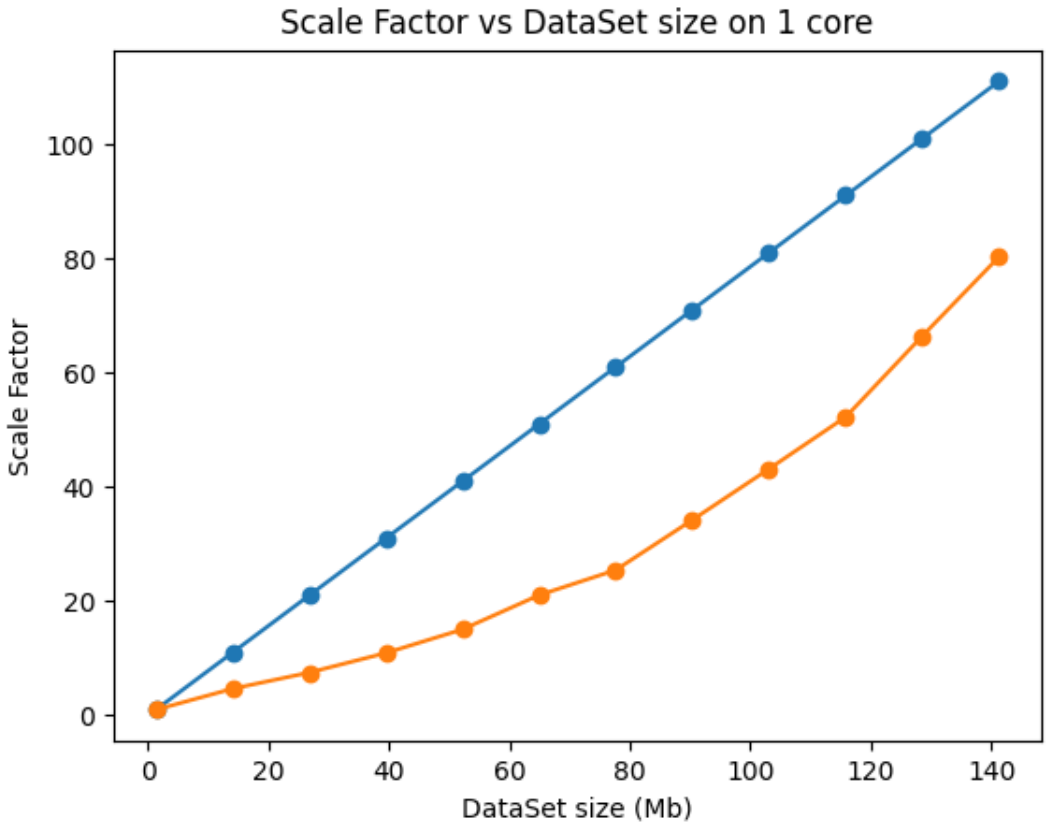

FIGURE 2 – Scale Factor vs Volume sur 1 coeurs

2.2 Mode cluster

2.2.1 Méthodologie

Pour le mode cluster, j'ai utilisé la même méthode que pour le mode local. En mettant le code dans un script Python, j'ai pu exécuter ce code sur le cluster Beluga de Calcul Québec. Le script prend en argument le nombre de nœuds ainsi que le nombre de cœurs par tâche et sauvegarde les résultats dans un fichier CSV sur le cluster. Le fichier est écrit de façon dynamique au cas où le job s'arrêterait pendant son exécution. Pour cela, la configuration du job sont accessibles avec les variables `$SLURM_JOB_NUM_NODES` et `$SLURM_CPUS_PER_TASK` qui sont écrites dans le fichier Slurm. Ensuite, ce fichier est exécuté avec la commande `sbatch`.

J'ai rencontré des difficultés pour lancer le code, car il y avait parfois du monde sur le cluster, en plus de cela, les jobs se trouvaient parfois bloqués en mode de configuration `CF`. Un exemple de ce problème est illustré dans la sortie de la commande `squeue`, visible dans la figure 3. Pourtant, la commande `scontrol` montrait bien que les ressources avaient été allouées, comme le montre la figure 4.

```
[bfaure@login1 bfaure]$ squeue
             JOBID     USER      ACCOUNT           NAME  ST  TIME_LEFT NODES CPUS TRES_PER_N MIN_MEM NODELIST (REASON)
               940      yet def-sponsor0 spawner-jupyte   R       1:53     1    6        N/A   3776M nodepool1 (None)
               951 baptiste def-sponsor0 spawner-jupyte   R      18:37     1    2        N/A   7616M nodepool13 (None)
               965  joeltia def-sponsor0 spawner-jupyte   R      44:51     1    8        N/A   3776M nodepool14 (None)
               954  hrimoux def-sponsor0 spawner-jupyte   R    4:24:54     1    4        N/A   3776M nodepool13 (None)
               968   bfaure def-sponsor0   job-spark.sh  CF      10:27     2   16        N/A      4G nodepool[5-6] (None)
               964      yet def-sponsor0   job-spark.sh  CF      11:12     2   16        N/A      4G node1,nodepool2 (None)
               966 baptiste def-sponsor0     Cluster.sh  CF      15:46     2   16        N/A      4G nodepool[3,15] (None)
               969 mfdorime def-sponsor0 spawner-jupyte  CF      55:56     1    6        N/A   3776M nodepool7 (None)
[bfaure@login1 bfaure]$
```

FIGURE 3 – Sortie de `squeue`

```
             JOBID     USER      ACCOUNT           NAME  ST  TIME_LEFT NODES CPUS TRES_PER_N MIN_MEM NODELIST (REASON)
              1026   bfaure def-sponsor0   job-spark.sh  CF      14:03     4   32        N/A      4G node1,nodepool[1,4-5] (None)
[bfaure@login1 bfaure]$ scontrol show job -dd 1026
JobId=1026 JobName=job-spark.sh
   UserId=bfaure(60047) GroupId=bfaure(60047) MCS_label=N/A
   Priority=4294900735 Nice=0 Account=def-sponsor00 QOS=normal
   JobState=CONFIGURING Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   DerivedExitCode=0:0
   RunTime=00:02:09 TimeLimit=00:15:00 TimeMin=N/A
   SubmitTime=2023-10-20T00:03:42 EligibleTime=2023-10-20T00:03:42
   AccrueTime=2023-10-20T00:03:42
   StartTime=2023-10-20T00:03:42 EndTime=2023-10-20T00:18:42 Deadline=N/A
   SuspendTime=None SecsPreSuspend=0 LastSchedEval=2023-10-20T00:03:42 Scheduler=Main
   Partition=cpubase_bycore_b1 AllocNode:Sid=login1:852656
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=node1,nodepool[1,4-5]
   BatchHost=node1
   NumNodes=4 NumCPUs=32 NumTasks=4 CPUs/Task=8 ReqB:S:C:T=0:0:*:*
   TRES=cpu=32,mem=16G,node=4,billing=32
   Socks/Node=* NtasksPerN:B:S:C=1:0:*:* CoreSpec=*
   JOB_GRES=(null)
     Nodes=node1,nodepool[1,4-5] CPU_IDs=0-7 Mem=4096 GRES=
   MinCPUsNode=8 MinMemoryNode=4G MinTmpDiskNode=0
   Features=(null) DelayBoot=00:00:00
   OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/project/60015/bfaure/job-spark.sh
   WorkDir=/project/60015/bfaure
   StdErr=/project/60015/bfaure/slurm-1026.out
   StdIn=/dev/null
   StdOut=/project/60015/bfaure/slurm-1026.out
   Power=
   SELinuxContext=user_u:user_r:user_t:s0

[bfaure@login1 bfaure]$
```

FIGURE 4 – Debugging des jobs avec `scontrol`

J'ai utilisé `scp` pour téléverser mes codes, comme illustré dans la figure 5. De plus, j'ai rencontré quelques bugs dans Spark lorsque je suis passé à un nombre de nœuds supérieur

à 2. J'ai fréquemment utilisé `nano` pour modifier et déboguer le code directement sur le cluster.

```
PS C:\Users\benoi\Documents\zz_UQAC\Apprentissage Automatique pour Données Maasives\TP_8INF919> scp mltree_cluster.py bfaure@uqac-8inf919.calculquebec.cloud:~/projects/def-sponsor00
/bfaure/mltree.py
(bfaure@uqac-8inf919.calculquebec.cloud) Password:
mltree_cluster.py                                                                    100% 2849    22.9KB/s   00:00
```

Figure 5 – Uploading avec `scp`

Une fois les données traitées, j'ai pu télécharger les résultats avec `scp`, comme le montre la figure 6.

```
[bfaure@login1 bfaure]$ scp bfaure@uqac-8inf919.calculquebec.cloud:~/projects/def-sponsor00/bfaure/out/ml_tree_app-20231020035226-0000_800.csv .
The authenticity of host 'uqac-8inf919.calculquebec.cloud (<no hostip for proxy command>)' can't be established.
ED25519 key fingerprint is SHA256:kH5JcNVGIwgJudd3gxAHoeoptpLPdX+o13UnXooSOpc.
Matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'uqac-8inf919.calculquebec.cloud' (ED25519) to the list of known hosts.
Password:
ml_tree_app-20231020035226-0000_800.csv
                      100%  196    84.7KB/s   00:00
```

Figure 6 – Download avec `scp`

Les jobs prenaient parfois du temps. Pendant les vacances, je pouvais facilement déboguer et travailler sur le cluster sans avoir besoin d'accéder à mon ordinateur grâce à l'application Terminus (termius.com). La connexion SSH se faisait facilement grâce aux paramètres déjà enregistrés dans l'application (voir la figure 7).

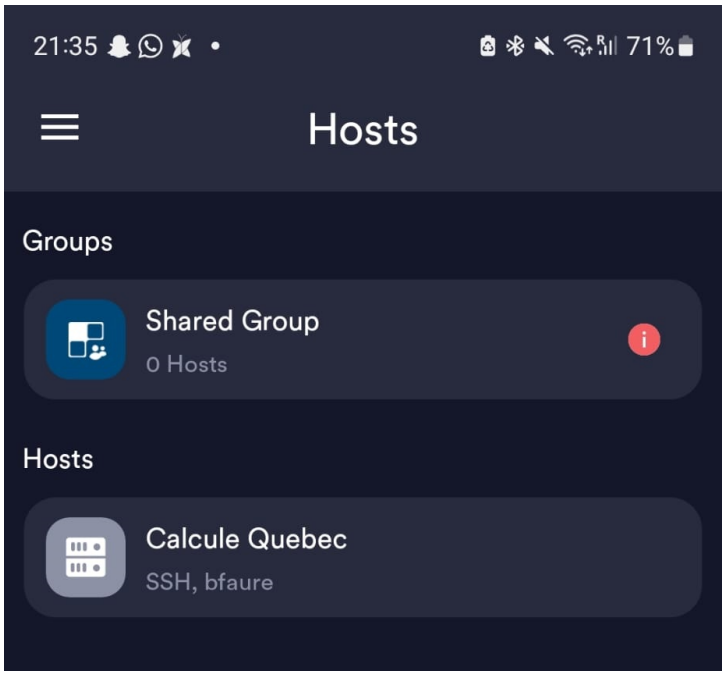

Figure 7 – Page d'accueil de Terminus

2.2.2 Résultats

Les résultats pour 4 nœuds sont présentés dans la figure 8. Nous pouvons observer la tendance décrite dans l'article au début de la courbe, notamment dans les petites

tailles de la base de données. Cependant, une fois qu'il y a trop de données, la courbe rattrape l'évolution linéaire du facteur d'échelle, et nous perdons l'avantage de la montée en échelle.

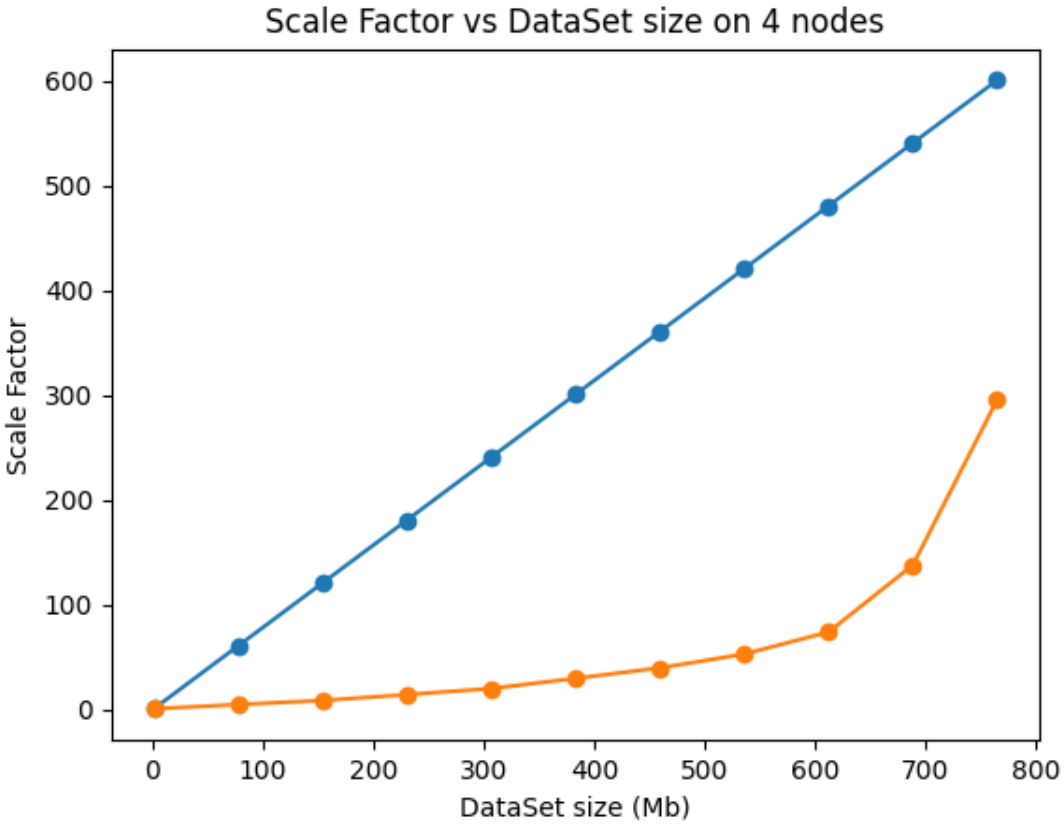

Figure 8 – Scale factor vs Volume sur 4 nodes a 8 coeurs

3 Expérience 2 - Configurations

3.1 Méthodologie

Pour tester le temps d'entraînement sur différentes configurations, j'ai simplement créé plusieurs fichiers Slurm avec les configurations de 2, 4, 6, 8 et 10 nœuds. Ensuite, j'ai attendu que le cluster soit vide afin de ne pas déranger personne, puis j'ai lancé les jobs comme illustré dans la figure 9.

```
[bfaure@login1 bfaure]$ squeue
             JOBID     USER      ACCOUNT           NAME  ST  TIME_LEFT NODES CPUS TRES_PER_N MIN_MEM NODELIST (REASON)
              1046   bfaure def-sponsor0 job-spark-2.sh   R      29:34     2   16        N/A      4G node1,nodepool1 (None)
              1047   bfaure def-sponsor0 job-spark-4.sh  PD      30:00     4   32        N/A      4G  (Resources)
              1048   bfaure def-sponsor0 job-spark-6.sh  PD      30:00     6   48        N/A      4G  (ReqNodeNotAvail, UnavailableNodes:nodepool[2-12])
              1049   bfaure def-sponsor0 job-spark-8.sh  PD      30:00     8   64        N/A      4G  (ReqNodeNotAvail, UnavailableNodes:nodepool[2-12])
              1050   bfaure def-sponsor0 job-spark-10.s  PD      30:00    10   80        N/A      4G  (ReqNodeNotAvail, UnavailableNodes:nodepool[2-12])
```

Figure 9 – Plusieurs jobs avec différentes configurations

3.2 Résultats

Les résultats de la comparaison sur un jeu de données de 827 Mo sont présentés dans la figure 10. Nous observons que le temps d'exécution diminue drastiquement jusqu'à atteindre un plateau à 10 nœuds.

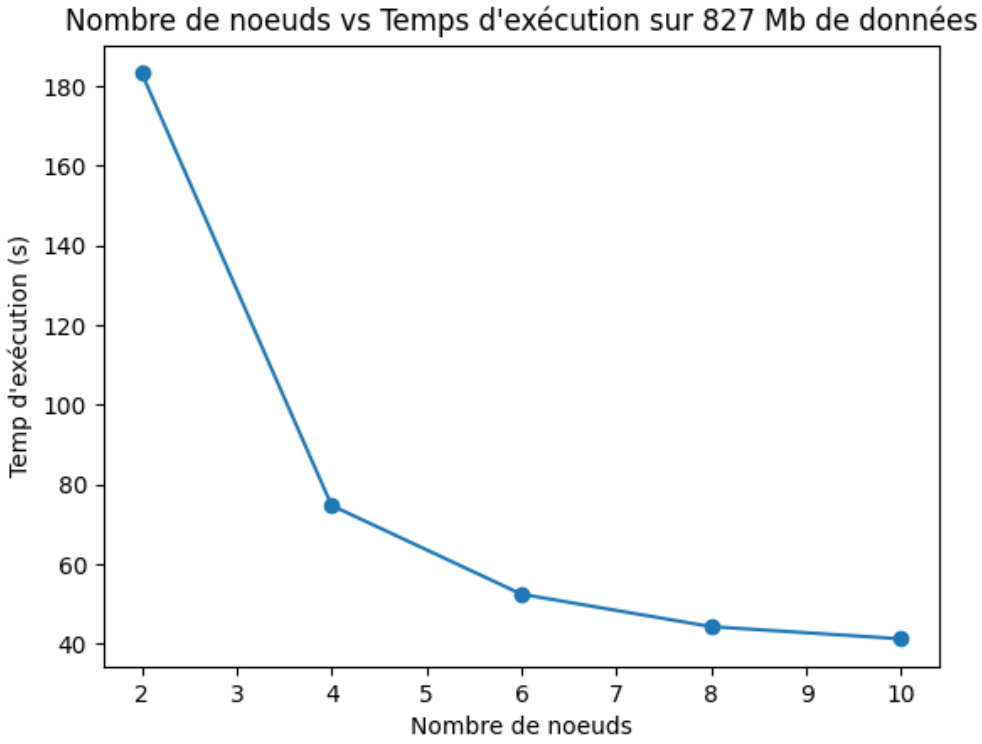

Figure 10 – Comparaison du temps d'exécution par rapport au nombre de noeuds utilisés

3.3 Accuracy

La figure 11 montre l'évolution de l'accuracy du modèle en fonction du nombre de données utilisées. Pour réaliser ce test, les données ont été séparées en un ensemble d'entraînement (70 %) et un ensemble de test (30 %). Nous observons que l'accuracy ne varie pas significativement et reste aux alentours de 84 %. Cette constance s'explique par le fait que nous réutilisons plusieurs fois les mêmes données pour augmenter le volume, ce qui équivaut à réentraîner simplement l'arbre de décision plusieurs fois, comme le montrent les données, ce qui s'avère inutile.

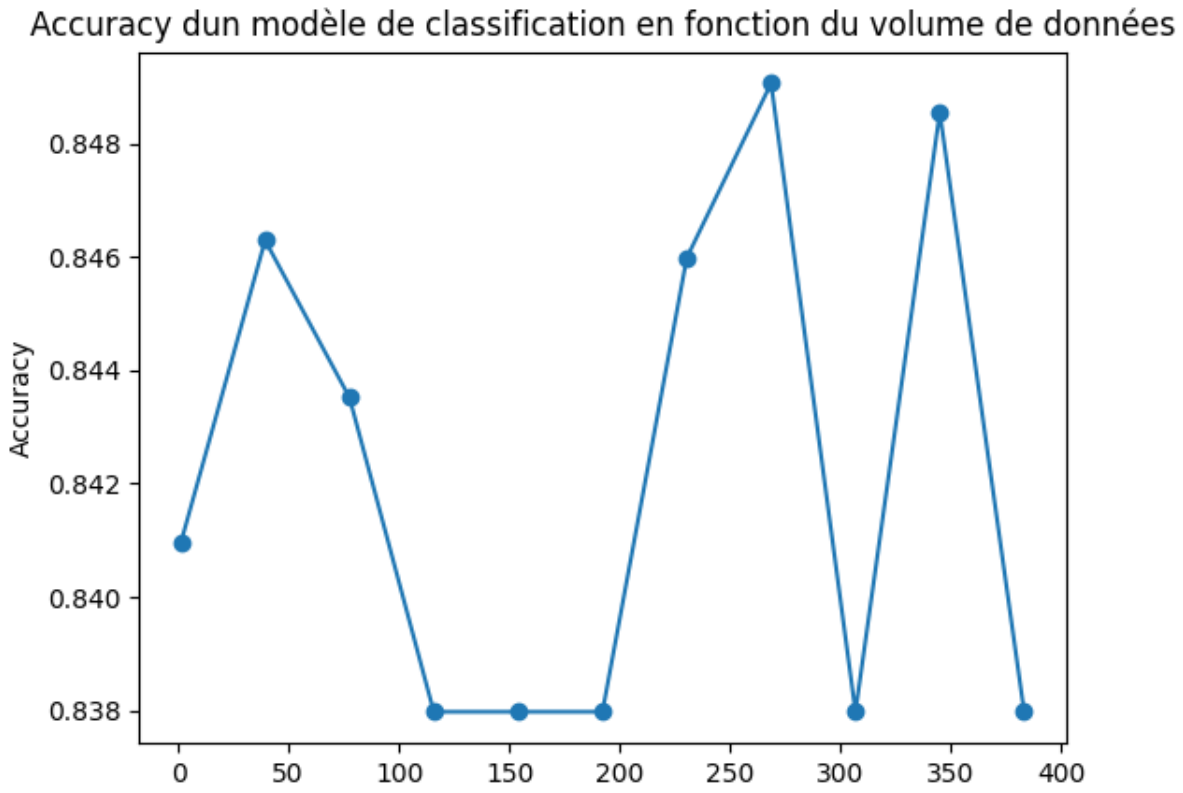

Figure 11 – Accuracy vs Volume de données

4 Expérience 3 - Comparaison avec sklearn

Pour optimiser les hyperparamètres, il est courant de réaliser une recherche systématique (sweep). Cela consiste à explorer l'espace des hyperparamètres afin de déterminer lesquels ont le plus d'influence sur les performances du modèle, ainsi que dans quelle direction. Pour ce faire, nous utilisons la validation croisée, ce qui implique l'entraînement de nombreux modèles avec des hyperparamètres différents. Il est alors nécessaire d'entraîner les arbres de décision le plus rapidement possible en parallélisant leur entraînement. Sur un jeu de données de 318 Mo, le temps d'exécution de scikit-learn était de **3282 secondes**, tandis que MLlib n'a pris que **1521 secondes**. Avec ce jeu de données de cette taille, MLlib était deux fois plus rapide.

5 Références

[1] Barry Becker and Ronny Kohavi. Adult. UCI Machine Learning Repository, 1996. DOI : https ://doi.org/10.24432/C5XW20.

[2] Vasile Purdil and tefan Gheorghe Pentiuc. Mr-tree-a scalable mapreduce algorithm for building decision trees. 2014.