Université du Québec à Chicoutimi **Session : A2023**
Département d'Informatique et de Mathématique **Professeur: A.Bouzouane**

Cours 8INF919: apprentissage automatique pour le BigData

Devoir #1 : Classification distribuée par arbre de décision

Le travail est individuel
À remettre le 19 octobre 2023[1]

1. But

- Se familiariser avec l'apprentissage automatique pour le BigData à l'aide d'un des algorithmes de classification distribuée basée sur les arbres de décision, tel que « MR-ID3,
- S'initier à l'environnement d'analyse de données massives comme Apache Spark http://spark.apache.org/ à travers l'utilisation d'outils disponibles pour l'apprentissage automatique comme la librairie MLlib, python pour spark (pyspark),
- Utiliser une des implémentations en pyspark de MR-ID3 en vue de montrer le respect du passage à l'échelle et l'amélioration de la précision du modèle appris,
- S'introduire à l'utilisation d'un cluster, comme celui de calcul Québec.

2. Travail demandé

Dans le contexte de l'article annexé « **MR-Tree – A Scalable MapReduce Algorithm for Building Decision Trees** », une expérimentation est montée à travers une douzaine (12) de nœuds–workers et un nœud-maître où on démontre, comme le montre la figure suivante, le passage à l'échelle avec une implémentation de l'algorithme MR-ID3 dans l'environnement Hadoop. L'échelle de variation du volume données, c'est de l'ordre de 1 pour 500 Mo.

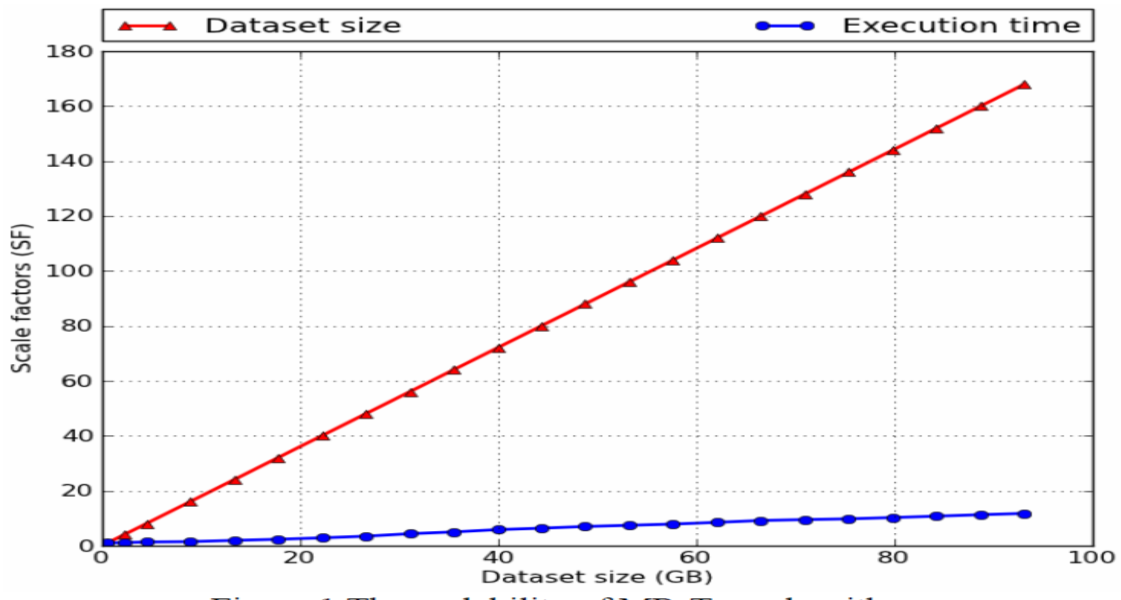


Figure 1.The scalability of MR-Tree algorithm

Dans le cas qui nous concerne, on vous demande de tenter de reproduire la même tendance dans l'environnement spark dans les deux modes local et cluster où l'échelle de variation du volume de

[1] Dans un dossier « remise des devoirs » du site moodle de cours.

données peut être différent d'un mode à un autre. Par ailleurs, il faut souligner que Spark est 100 fois plus rapide que Hadoop à cause des opérations d'E/S sur disque.

Le travail demandé se compose des parties suivantes:

2.1 Faire la synthèse de l'article qui est donné en annexe à partir de la page 4 **(1 point);**

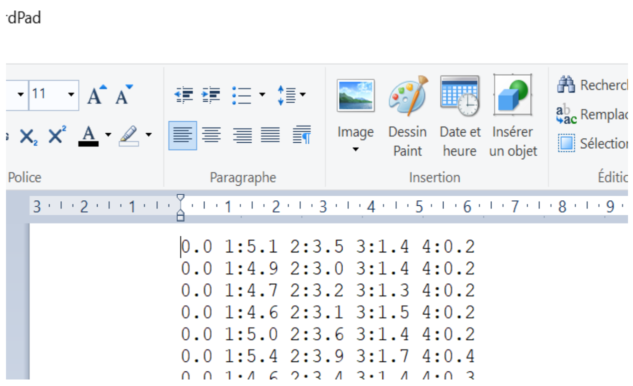
2.2 En utilisant une des implémentations en pyspark de la classification par arbre de décision, on vous demande de montrer la garantie du passage à l'échelle. Pour l'implémentation, vous pouvez vous inspirez de l'exemple du code fourni en pyspark:

2.2.1 Élaborer différents scénarios de surcharge selon le protocole d'expérimentation décrit dans l'article, en s'appuyant sur les deux configurations suivantes :

- **Mode local** : l'échelle sera moindre étant donné les contraintes du cluster en mode local. Vous pouvez opter pour une échelle de 1 pour 500Ko ou 250Ko de données pour une configuration donnée en termes de nombre de cœurs du processeur qui est à votre disposition. Vous pouvez utiliser votre portable s'il est doté d'un processeur quad-cœurs, prenez le nombre maximal de cœurs. Vous pouvez aussi utiliser celle proposée par Calcul Québec[2] en mode local avec 8 cœurs maximum. Veuillez noter que nous disposons d'une salle réservée pour les étudiants en Maîtrise (**P4-6600**) dotée de machines à **20 cœurs, 16Go de RAM et 3.6GHz** où pyspark est installé avec jupyter notebook. (**4 points**)

- **Mode Cluster** en lançant des jobs avec SSH. Vous avez la possibilité de reproduire une configuration avec un nombre de nœuds. L'échelle peut être de 1 pour 10Mo. (**4 points**)

Pour les données, le lien qui est indiqué dans l'article (référence bibliographie 13) n'est plus opérationnel, vous pouvez utiliser le lien suivant vers UCI : https://archive.ics.uci.edu/ml/datasets/adult. J'ai mis un exemple de code[3] sur le site du cours (devoir) qui utilise le dataset iris (150 individus) en format libsvm[4]. Ce dataset décrit la fleur d'iris dont le label est constitué de 3 classes : setosa, verginica et versicolor (0, 1, 2) où chaque individu est décrit par la classe suivi des 4 attributs (longueurs et largeurs de la sépale et de la pétale), comme le montre la figure suivante.



```
0.0 1:5.1 2:3.5 3:1.4 4:0.2
0.0 1:4.9 2:3.0 3:1.4 4:0.2
0.0 1:4.7 2:3.2 3:1.3 4:0.2
0.0 1:4.6 2:3.1 3:1.5 4:0.2
0.0 1:5.0 2:3.6 3:1.4 4:0.2
0.0 1:5.4 2:3.9 3:1.7 4:0.4
```

[2] A Consulter le guide d'introduction au cluster calcul Québec sur le site du cours.
[3] A consulter aussi la documentation de spark : http://spark.apache.org/docs/latest/mllib-decision-tree.html
[4] Un des formats les plus utilisés dont la structure est : <classe> <index1> :<valeur1> <index2> :<valeur2>.....

Vous pouvez dupliquer ce dataset pour créer un certain volume important de données, comme c'est le cas de l'article. Vous pouvez aussi vous appuyez sur n'importe quel jeu de données autre que le dataset Adult où celui que je vous propose.

2.2.2 Est-ce-que vous avez obtenu les mêmes tendances que l'article ? Justifiez votre réponse ? **(2 points);**

2.3 Pour montrer la nécessité de la distribution de l'apprentissage dans le cas du BigData, expérimenter le scénario suivant :

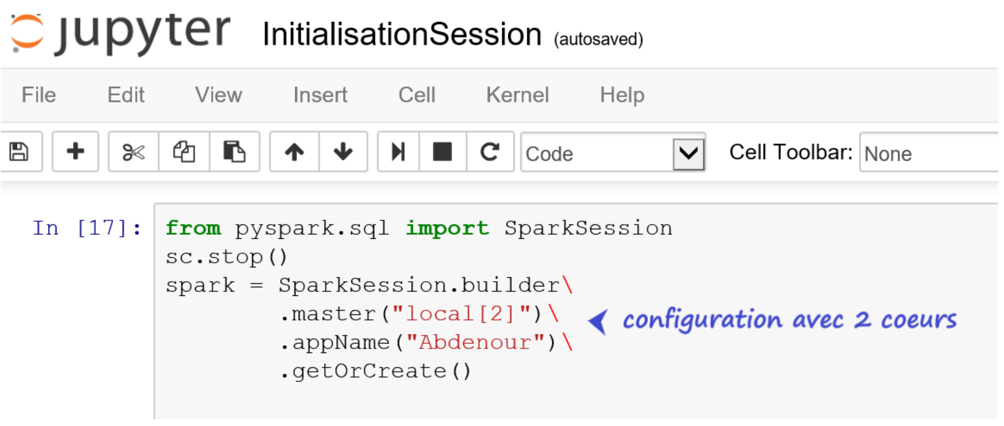
2.3.1 Utiliser le mode cluster pour mesurez le temps d'entrainement de l'algorithme MR-ID3, selon différentes configurations en nombre de nœuds (2, 4, 6, 8 et 10) avec un même volume de données pour toutes les configurations. Vous pouvez prendre n'importe quel jeu de données sur le net dont la taille ne dépassant pas le 2Go et, doit respecter les contraintes matérielles du cluster **(5 points)**.

2.3.2 Est-ce que la performance du modèle appris augmente du fait de l'entraînement sur un volume important de données ? Justifier votre réponse ? (**2 points)**

2.4 L'opération de réglage (tuning) des hyper-paramètres d'un modèle appris, comme par exemple la profondeur de l'arbre (maxDepth), le nombre minimum d'instances par nœud (minInstancesPerNode) nécessaire pour une division, est une opération très couteuse en temps et en particulier dans le contexte du bigdata. Montrer, en faisant la comparaison avec **scikit-learn en python et MLlib en pyspark,** pour un jeu de données important, que le recours à une validation croisée distribuée pour trouver le meilleur modèle, est indispensable. Justifier votre réponse à travers l'expérimentation avec scikit-leran et MLlib et le calcul du temps écoulé (**2 points**)

NB : **<u>pour la configuration vous pouvez utiliser les instructions suivantes :</u>**

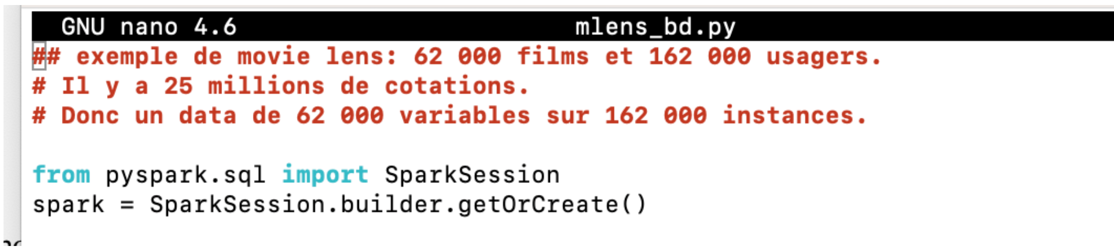
- En mode local avec jupyter notebook :


jupyter InitialisationSession (autosaved)
File Edit View Insert Cell Kernel Help
Code
Cell Toolbar: None
In [17]:
configuration avec 2 coeurs

```
from pyspark.sql import SparkSession
sc.stop()
spark = SparkSession.builder\
        .master("local[2]")\
        .appName("Abdenour")\
        .getOrCreate()
```

Veuillez noter que si vous travaillez sur le cloud avec Databriks, cette commande n'a aucun effet sur le cluster. La configuration du cluster est hors contrôle pour une édition gratuite. Vous pouvez juste faire des essais avec 2 cœurs.

- En mode cluster : rajouter les deux instructions suivantes dans vote code python pour l'ouverture d'une SparkSession :

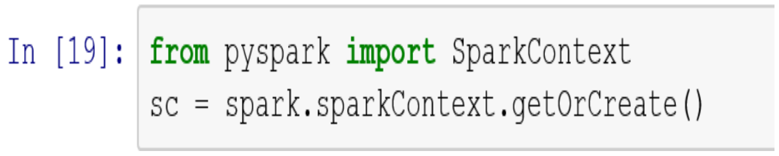


```
  GNU nano 4.6                          mlens_bd.py
## exemple de movie lens: 62 000 films et 162 000 usagers.
# Il y a 25 millions de cotations.
# Donc un data de 62 000 variables sur 162 000 instances.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

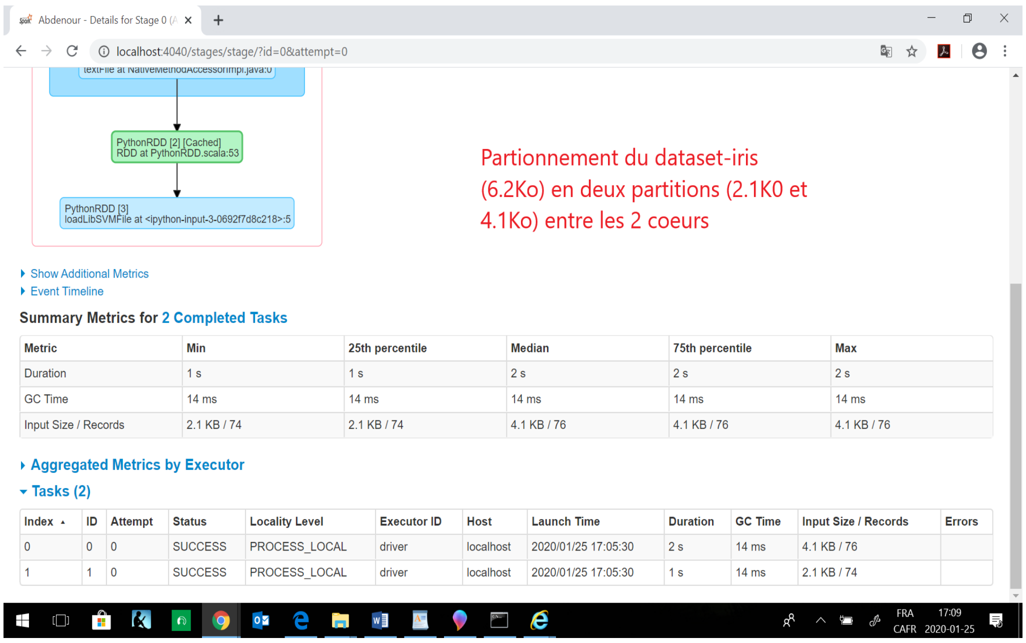
Vous pouvez vous inspirez de l'exemple de création d'un script pour définir un job et l'exemple du fichier python mlens_bd que j'ai mis à votre disposition dans la démnostration qui accompagne le guide d'introduction au cluster de calcul Québec.

Dans le cas où vous utiliserez une RDD, vous devez avoir un point d'entrée avec sparkContext :



```
In [19]: from pyspark import SparkContext
         sc = spark.sparkContext.getOrCreate()
```

Vous pouvez utiliser Spark-UI pour suivre le comportement des jobs dans le mode local. Ouvrir avec votre navigateur le lien http://localhost:4040. Ainsi, vous avez accès aux différentes métriques, comme le montre la figure suivante (**2 points**):


Abdenour - Details for Stage 0 (A
localhost:4040/stages/stage/?id=0&attempt=0
textFile at NativeMethodAccessorImpl.java:0
PythonRDD [2] [Cached]
RDD at PythonRDD.scala:53
PythonRDD [3]
loadLibSVMFile at <ipython-input-3-0692f7d8c218>:5
Show Additional Metrics
Event Timeline
FRA
CAFR
17:09
2020-01-25

Partionnement du dataset-iris (6.2Ko) en deux partitions (2.1K0 et 4.1Ko) entre les 2 coeurs

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 s	1 s	2 s	2 s	2 s
GC Time	14 ms	14 ms	14 ms	14 ms	14 ms
Input Size / Records	2.1 KB / 74	2.1 KB / 74	4.1 KB / 76	4.1 KB / 76	4.1 KB / 76

Aggregated Metrics by Executor

Tasks (2)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Input Size / Records	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2020/01/25 17:05:30	2 s	14 ms	4.1 KB / 76	
1	1	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2020/01/25 17:05:30	1 s	14 ms	2.1 KB / 74	

3. Livrables

3.1 Le code source en joignant un fichier en pdf décrivant la manière dont vous avez procédé pour mener l'expérimentation et les réponses aux questions posées.

3.2 Quelques copies d'écrans de résultats d'expérimentation.

MR-Tree - A Scalable MapReduce Algorithm for Building Decision Trees

[1]Vasile PURDILĂ, [2]Ştefan-Gheorghe PENTIUC
[1,2]Stefan cel Mare University of Suceava, Romania
[1]vasilep@eed.usv.ro, [2]pentiuc@usv.ro

Abstract–**Learning decision trees against very large amounts of data is not practical on single node computers due to the huge amount of calculations required by this process. Apache Hadoop is a large scale distributed computing platform that runs on commodity hardware clusters and can be used successfully for data mining task against very large datasets. This work presents a parallel decision tree learning algorithm expressed in MapReduce programming model that runs on Apache Hadoop platform and has a very good scalability with dataset size.**

Keywords: **big data, decision tree, Hadoop, MapReduce, pattern recognition.**

I. INTRODUCTION

In general pattern recognition applications a pattern is a mathematical representation of an object or phenomenon from the real world. The common representation of a pattern is an array with *p* numerical values representing the outputs of *p* measurement processes of measuring or observation. In database terms a pattern is a tuple with p attributes.

In the supervised learning context of pattern recognition problems the aim is to find a classifier model able to assign a class to a new pattern based on the information stored in a training set. One of the most difficult problems in this context is to classify patterns in non-linear separable classes. For solving such a problem the binary trees decision same to be a very good approach.

The papers deals with a parallel algorithm that use the programming model Map Reduce implemented on Apache Hadoop [14].

II. BACKGROUND AND RELATED WORK

A possible definition for big data is the following: *a very large amount of unstructured data that can be processed in real time by specific methods other than database management systems* [8].

Various technologies have developed to process big data in parallel such as Dryad [10], Sector/Sphere [11] and MapReduce [12] which is the most popular one and considered *de facto* standard for big data processing.

Apache Hadoop [14] is a Java implementation of MapReduce programming model and was originally developed at Yahoo. It runs on clusters of commodity hardware and has the following structure:

- Core – provides a set of API functions for platform management and MapReduce jobs execution
- HDFS – a distributed file system for data storage
- MapReduce – the MapReduce engine

Each file that is added to HDFS gets split into blocks of various sizes - the default size is 64 MB but it can be overridden by the user. These blocks are called splits and get replicated across the cluster on different data nodes. Hadoop is fault tolerant and has an automatic recovery mechanism for missing data. As the cluster consists of commodity hardware, the failure rate of individual data nodes may be high. In case of a data node failure, Hadoop will locate the replicas of the missing splits and continue the current MapReduce job without the user's intervention.

There are many algorithms for learning decision trees in literature [1-5] such as ID3, C4.5, C5.0, SPRINT, SLIQ and CART. However, they are optimized for small and medium sized datasets. To our knowledge, there are very few decision tree learning algorithms for big data [6-7] and they are optimized for ensemble learning [9].

In this paper we present a decision tree learning algorithm called MR-Tree that builds a decision tree in parallel and runs on MapReduce. This is an improved version of ID3 algorithm and can use both discrete and continuous attributes.

III. THE MR-TREE ALGORITHM

The algorithm has three sections. The first one is the controller section that processes the input parameters, runs the recursive tree induction function and also the pruning function. This part of the algorithm runs on the master node of the cluster and is not very computationally intensive.

The tree induction function is called ID3, and has the following parameters:

- ***filters*** – a list of pairs *(attribute, value)* for nominal attributes or *(attribute, threshold, sign)* for numeric attributes
- ***inputPath*** – the HDFS folder where the learning dataset can be found
- ***mostCommonClass*** – the most common class from the previous iteration
- ***attributes*** – the list of attributes that can be used for splitting a node

In the first call of ID3 function the *filters* parameter is an empty list while the *attributes* parameter contains all the attributes.

Within an ID3 call, if the *attributes* parameter is empty a new leaf node is created and labeled with the most common class from the previous iteration; else the function runs a MapReduce job that calculate the following:

- the class entropy
- information gain for each attribute
- the most common class for current dataset
- the number of patterns processed
- the splitting value (threshold) for each numeric attribute

If the class entropy is 0 (all patterns from current dataset belong to the same class) or no patterns were processed (all were excluded by the filters) the algorithm returns a new leaf node that is labeled with the most common class for current dataset; else it chooses the attribute *maxattr* that has the maximum information gain, creates a new node and uses *maxattr* to split it. The ID3 function is called recursively for each value of *maxattr* attribute. In case of numeric attributes, the ID3 function is called twice, once for values ≤ threshold value and once for values > threshold value.

ID3*(filters, inputPath, mostCommonClass, attributes)*

node is a node within the decision tree

```
1:  node = {createa new node}
2:  if (attributes is empty) then
3:      label node with mostCommonClass
4:      return node
5:  endif
6:  res = runHadoopJob (inputPath, filters, attributes)
7:  if (res.recProcessed == 0 or res.classEntropy == 0) then
8:      label node with res.mostCommonClass
9:      return node
10: endif
11: maxAttr = {the attribute with maximum gain}
12: if (maxAttr is nominal attribute) then
13:     foreach(v in maxAttr.values)
14:         if (filters contains maxAttr) then
15:             filters[maxAttr].value = v
16:         else
17:             filters.Add(maxAttr,v)
18:         endif
19:         remove maxAttr from attributes
20:         node.nodes.Add(ID3(filters,inputPath,
            res.mostCommonClass, attributes))
21:     endfor
22: else
23:     filters.Add(maxAttr, maxAttr.threshold, -1)
24:     node.nodes.Add(ID3(filters,inputPath,
        res.mostCommonClass, attributes))
25:     filters.Add(maxAttr, maxAttr.threshold, 1)
26:     node.nodes.Add(ID3(filters,inputPath,
        res.mostCommonClass, attributes))
27: endif
28: return node
```

A map task consists of four functions as follows:

- setup – various parameters from job's configuration table can be read here
- map – the function that gets called for each pattern from dataset
- cleanup – this function gets called after all patterns were processed by map function
- the main function from a map task that calls the other three

The map function processes one record at a time according to the filters passed down from ID3 function and calculates the class distribution of each attribute for current pattern. This class distribution is an array and is small enough to be stored in memory. The cleanup function emits a pair (key, value) for each attribute in the following format: *(attribute, class_distribution_for_attribute).*

map*(key, pattern)*
Input: *key* = the offset of a pattern from the input file
pattern = a pattern

filters is a set of filters read in function setup
attributes is a set of attributes read in function setup
distrib[][] is an array that holds the class distribution for each attribute

```
1:  if (acceptRecord(pattern, filters)) then
2:      distrib[][] = { calculate class distribution for each
        attribute from attributes }
3:  endif
```

cleanup*()*

```
1:  foreach(attr in attributes)
2:      emit(attr, distrib[attr])
3:  endfor
```

As one may see the *cleanup* function emits very little data which means less I/O operations.

The *reduce* function aggregates the total class distribution for each attribute.

reduce*(attr, ldistrib)*
Input: *attr* = an attribute
ldistrib = a list of vectors of class distribution

val is a vector of 0
ret is a structure of {recProcessed, classEntropy, threshold, mostCommonClass, informationGain}

```
1: foreach(distrib in ldistrib)
2:     val[] = val[] + distrib[]
3: endfor
4: ret.recProcessed = calculateTotalRecords(val)
5: ret.classEntropy = calculateClassEntropy(val)
6: ret.threshold = calculateThreshold(attr, val)
7: ret.mostCommonClass =
   calculateMostCommonClass(val)
8: ret.informationGain = calculateInformationGain(val)
9: emit(attr,ret)
```

As the tree that is induced by ID3 function may over fit data it is necessary to prune it. The pruning algorithm uses a test dataset that may also be very large, and therefore this operation must be expressed in MapReduce as well. In our experiment we have used the reduced error method that consists of replacing a node with a leaf labeled with the most common class within the sub-tree where that particular node is root. The pruning process is an iterative one and stops when the classification accuracy no longer improves as the nodes get replaced by leaves.

IV. Experimental results

In order to test the scalability of this algorithm a dataset derived from US Census Bureau database was used [13]. A number of 400 records from this dataset have been selected randomly and used as training and test datasets (200 records each set). In order to create various workloads for the experiments, each dataset was multiplied several times. We have defined a scale factor SF to measure the dataset size. The scale factor 1 was defined for a dataset size of 595,296,000 bytes (learning and test datasets combined). The biggest value for the scale factor was set to 168 which corresponds to a dataset size of 100,009,728,000 bytes.

The Hadoop cluster has been configured with 13 nodes, a master node and 12 slave nodes. Each slave node consists of two quad core Xeon processors @ 2 GHz, 2 GB of RAM and RedHat Linux operating system. The cluster has a total of 8 x 12 = 96 MapReduce slots. Each dataset was loaded into HDFS using a different split size in order to match the number of MapReduce slots; therefore a number of 96 splits were produced for each workload.

We have run 23 separate tests for various workloads and recorded the execution time against each workload. The experimental results show that MR-Tree algorithm has a linear scalability with dataset size.

This is caused by the fact that all map and reduce functions are very efficient in terms of data output which means very little overhead for I/O operations within the Hadoop cluster.

In Table I are shown the execution times against various workloads. A second scale factor was defined for the execution time as well; a scale factor of 1 corresponds to 40 minutes.

TABLE I. EXPERIMENTAL RESULTS

Dataset size (bytes)	SF - dataset size (1 = 595,296,000 bytes)	Execution time (minutes)	SF - execution time (1 = 40 minutes)
595,296,000	1	40	1.00
2,381,184,000	4	45	1.13
4,762,368,000	8	54	1.35
9,524,736,000	16	59	1.48
14,287,104,000	24	77	1.93
19,049,472,000	32	93	2.33
23,811,840,000	40	114	2.85
28,574,208,000	48	138	3.45
33,336,576,000	56	174	4.35
38,098,944,000	64	200	5.00
42,861,312,000	72	234	5.85
47,623,680,000	80	255	6.38
52,386,048,000	88	280	7.00
57,148,416,000	96	297	7.43
61,910,784,000	104	315	7.88
66,673,152,000	112	340	8.50
71,435,520,000	120	367	9.18
76,197,888,000	128	380	9.50
80,960,256,000	136	391	9.78
85,722,624,000	144	410	10.25
90,484,992,000	152	430	10.75
95,247,360,000	160	451	11.28
100,009,728,000	168	470	11.75

A graphical representation of data from Table I is shown in Fig. 1.

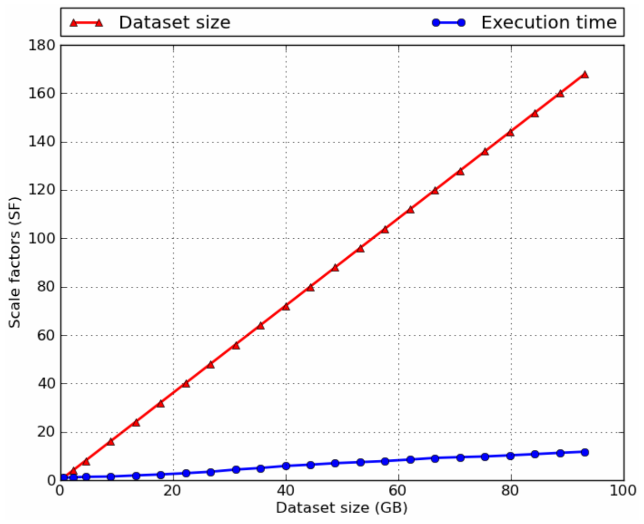


Figure 1.The scalability of MR-Tree algorithm

V. Conclusion

We presented a decision tree learning algorithm called MR-Tree that has an excellent scalability with dataset size. It can be used to learn decision trees against very large datasets and runs on Apache Hadoop platform.

The main disadvantage of the algorithm is that it uses a MapReduce iteration to choose the best attribute for splitting each tree node which means that it may be slow for very large trees.

As further research we plan to improve the algorithm in such a way that the tree growth is stopped at certain threshold so it doesn’t grow very large.

REFERENCES

[1] Quinlan, J. R. “Induction of Decision Trees.” Machine Learning 1, no. 1 (March 1, 1986): 81–106. doi:10.1007/BF00116251.

[2] J. R. Quinlan, "C4.5: Programs for Machine Learning", San Mateo, CA: Morgan Kaufmann, 1993

[3] Mehta, Manish, Rakesh Agrawal, and Jorma Rissanen. “SLIQ: A Fast Scalable Classifier for Data Mining.” In Proceedings of the 5th International Conference on EDT: Advances in Database Technology, 18–32. EDBT ’96. London, UK, UK: Springer-Verlag, 1996.

[4] BREIMAN, Leo, Jerome H. FRIEDMAN, Richard A. OLSHEN, and Charles J. STONE. “Classification and Regression Trees (POD)” (1999).

[5] Shafer, John C., Rakesh Agrawal, and Manish Mehta. “SPRINT: A Scalable Parallel Classifier for Data Mining.” In Proceedings of the 22th International Conference on Very Large Data Bases, 544–555. VLDB ’96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[6] Panda, Biswanath, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. “PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce.” Proc. VLDB Endow. 2, no. 2 (August 2009): 1426–1437.

[7] Yin, Wei, Yogesh Simmhan, and Viktor K. Prasanna. “Scalable Regression Tree Learning on Hadoop Using OpenPlanet.” In Proceedings of Third International Workshop on MapReduce and Its Applications Date, 57–64. MapReduce ’12. New York, NY, USA: ACM, 2012. doi:10.1145/2287016.2287027.

[8] http://adayinbigdata.com/

[9] Opitz, David, and Richard Maclin. “Popular Ensemble Methods: An Empirical Study.” Journal of Artificial Intelligence Research 11 (1999): 169–198.

[10] Isard, Michael, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks.” In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, 59–72. EuroSys ’07. New York, NY, USA: ACM, 2007. doi:10.1145/1272996.1273005.

[11] Grossman, Robert, and Yunhong Gu. “Data Mining Using High Performance Data Clouds: Experimental Studies Using Sector and Sphere.” In Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 920–927. KDD ’08. New York, NY, USA: ACM, 2008. doi:10.1145/1401890.1402000.

[12] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” Commun. ACM 51, no. 1 (January 2008): 107–113. doi:10.1145/1327452.1327492.

[13] US Census Bureau, adult dataset: http://www.sgi.com/tech/mlc/db/adult.names

[14] Apache Hadoop: http://hadoop.apache.org

Vasile Purdila received both MSc and BSc degrees in Computer Science and Engineering from “Stefan cel Mare” University of Suceava, Romania. He is now a PhD student at the same university. His research interests include Pattern Recognition, Parallel and Distributed Systems, Big data.

Ştefan Gheorghe Pentiuc is a professor at the “Stefan cel Mare” University of Suceava, Faculty of Electrical Engineering and Computer Science. His research interests include Pattern Recognition, Parallel and Distributed Systems.