Classification On The Clouds Using MapReduce

Simão Martins
Instituto Superior Técnico
Lisbon, Portugal
simao.martins@tecnico.ulisboa.pt

Cláudia Antunes
Instituto Superior Técnico
Lisbon, Portugal
claudia.antunes@tecnico.ulisboa.pt

1 ABSTRACT

The parallelization of mining algorithms under MapReduce (MR) became a reality in the last years, but algorithms for training single decision trees, like ID3 [1]or C4.5 [2], remain unexplored. Decision trees continue to play an important role in data mining, mainly in the cases where the model itself is used to understand or just validate existing domain knowledge. In this paper, we discuss the different issues to deal with when trying to parallelize ID3 under MR, and propose a new algorithm, MRID4, for training a single decision tree, based on ID3 but that is able to deal with numerical attributes, as in C4.5. Experimental results show that our algorithm can scale to very high values.

2 INTRODUCTION

In the last years, it has been shown that many machine learning and data-mining (ML-DM) algorithms are suited to be implemented under MR [1]. In the majority of the identified cases, the problem consists in defining each of the algorithms as a summation over data points, which may be easily parallelized in MR.

With this work we aim to implement a classification algorithm named MRID4 that is heavily based on ID3 but is also able to deal with continuous attributes in the same way that C4.5 does.

3 MAPREDUCE

A MapReduce (MR) program consists in the execution of various jobs. Each job is comprised by the execution of two user-defined functions: the *Map* and the *Reduce*. In MR the input is interpreted as a set of key/value pairs and the Map and Reduce functions perform operations over them. At the beginning of each job the framework splits the input dataset into independent chunks, called *splits*, which are then processed by the mappers in a completely parallel manner. Each mapper, that is, the execution of the user defined map function over one of splits, will emit an intermediate set of key/value pairs. This intermediate data will be sorted by key by the framework and then passed as input to the reducers. The reducers will perform the final aggregation of the data and/or any other type of data processing procedure the algorithm may require. Please note that, for each unique key in the intermediate data, the framework will only assign a single reducer to process all the values associated with that key. Each reducer can run in parallel because it will be processing a different portion of the intermediate data. But the reducers can only begin to execute when all the maps have finished and the framework has sorted all the intermediate data. The most widely used implementation of MR is Hadoop.

4 LITERATURE REVIEW

The task of building decision tree learners in parallel is an area of many research, however there aren't many papers providing implementations for decision tree learners using MR.

Gongqing Wu et al. [2] proposed the MReC4.5 algorithm, that partitions the data into m subsets (m is a user defined variable), applies the standard C4.5 algorithm in each subset (the algorithm is applied in the mapper) and finally each of the decision trees, one from each mapper, are assembled in the reducer by relying on the bagging ensemble strategy. This strategy classifies an example with class y, if the majority of the decision trees classify the example with class y.

Biswanath et al. created PLANET [3], which stands for Parallel Learner for Assembling Numerous Ensemble Trees. The process is as follows: a

single machine, called the Controller, maintains a *ModelFile* containing the tree constructed so far and two queues: *MapReduceQueue* (MRQ) and *InMemoryQueue* (InMemQ). Whenever a dataset associated with a node is too large to fit in the main memory, that node it is added to the MRQ, similarly if the dataset fits in memory then the node is added to the InMemQ.

For each node in the MRQ, a *MR_ExpandNodes* job is scheduled. This job receives the nodes to expand, N, the model file, M, and the entire training dataset. Then the map executes two actions. First, it determines if the record is part of the input dataset for any node in N, by traversing the current model with the record. Then the possible splits for each node are evaluated and the best one is selected. When a *MR_ExpandNodes* job returns, the queues are updated with new nodes that can now be expanded, if these new nodes are in the same level in the tree, PLANET will schedule a single job passing in the new nodes (as N), as opposed to scheduling a job for each node, this means the tree is expanded in a breadth first manner.

Similarly to the nodes in the MRQ, for each node in the InMemQ a *MR_InMemory* job is scheduled, with the same arguments as the previous job. As before, in the map, the training dataset is filtered to records that are input to the node, but now it is the reducer that performs the entire sub tree construction using the traditional greedy top-down approach. It is able to do so, because all the input records fit in the main memory.

The focus of PLANET is on numerical attributes, so their implementation is more focused on regression trees.

Wei Yin et al. [4] implemented an open-source version of PLANET called OpenPlanet. This implementation distinguishes itself from PLANET by leveraging the Weka machine learning library.

5 ID3

The ID3 algorithm [5] generates a decision tree from a given dataset in a recursive manner. In the beginning the algorithm starts with the original dataset $\mathbb{S}$ as the root node; then in each step it calculates the information gain (IG) of every unused attribute in the set $\mathbb{S}$. It then splits the set into subsets using the attribute for which the information gain is maximal; finally, it recurses on the subsets considering only the attributes never selected before. The algorithm stops the recursion on the following cases: 1) every instance in the subset belongs to the same class; 2) there are no more attributes to be selected; 3) there are no more instances in the subset.

6 Challenges of Implementing ID3 In Hadoop

Before presenting our algorithm, we will go through a list of issues that arise when ID3 is to be implemented in Hadoop.

6.1 Subset Recursion And Computation

Let's assume that we had a MR job, J, which given a dataset $\mathbb{S}$, would be able to compute the attribute that best splits $\mathbb{S}$, and the subsets resulting from splitting $\mathbb{S}$ using that attribute. With this job, we could replicate the recursive implementation of the ID3 algorithm, by invoking J for the sub sets returned by the previous execution of J. Although this seems like a good solution to the problem we will show that it has a lot of shortcomings.

If we were to iterate through a big dataset with MR, the framework would divide the dataset, and each mapper would only have access to a split of the dataset. So, if one of the mappers finds that all its instances have the same class label (stop condition number 1 of ID3), this might not be true globally since one of the other mappers might have found that the instances in its split do not belong to the same class. Also, if the computed sub sets in a previous execution of J, were to have very different sizes, the invocations of J for the very small sub sets would do very little work compared to the invocations of J for the very large sub sets, or in other words, it would lead to an uneven load distribution.

In this manner, it is clear that invoking J for each computed sub set would be a bad option. Now let's

show that such job J cannot be implemented in Hadoop. J needs to do two different things: to compute the attribute that best splits the dataset and to compute the sub sets resulting from splitting the data set using that attribute. Since the job must output the sub sets in the end, the reducer(s) must have access to the sub sets instances, which in turn means the mappers have to send the instances to the reducer(s). Even without delving deeper one can see this will incur in lots of overheads since the entire data set has to be sent through the network.

If we were to assume the number of reducers, in J, is equal to the number of sub sets, that is, each reducer would output a single sub set of the data, it would mean the mappers would be capable of computing the sub sets by themselves, and the reducers would only copy the input they received to the output. This however would be impossible because it would mean each mapper would have to choose the same attribute as the one that best splits the data set, but since each mapper only has access to a split of the data set, this would not be possible.

Assuming the mappers by themselves cannot compute the attribute that best splits the dataset (which we will show in the next issue). The entire data set would have to be sent to a single reducer which would then compute the sub sets, but this is the same as executing the algorithm in a single machine, although with a lot of overhead.

Since sending the instances between jobs is clearly a very costly option, one of the best options seems to be to iterate the entire data set in each job, and for each example transverse the decision tree constructed so far, to see if the example falls within a node we are trying to compute the best attribute. Although this also seems a costly approach we will show that it is in fact a very good one in the last issue.

6.2 COMPUTING THE IG

Given that the time consuming part of the algorithm is iterating through the instances to compute the IG for each attribute, we could leverage MR to distribute this work. We could launch a job where each map determines, for its split, the information gain for each attribute, and then the reducer sums the partial IG values and chooses the attribute with the highest information gain.

Unfortunately this is not possible since the information gain for an attribute in a set is not necessarily equal to the sum of its information gain in its subsets (equation 1).

$$IG(\mathbb{S} + \mathbb{T}, a) \neq IG(\mathbb{S}, a) + IG(\mathbb{T}, a),$$

Equation 1 – Inequality of the IG.

6.3 ONE ATTRIBUTE PER MAPPER

It seems reasonable to distribute the work of computing the IG for each attribute in a column-wise approach, that is each mapper would calculate the IG for an attribute (a column in the data set) and the reduce would choose the one with the highest value. However this distribution also lacks in various aspects.

In Hadoop, we can't reliably ascertain how many map tasks will be launched, this coupled with the fact that we cannot distribute different data to each mapper (besides the dataset split or any data in the distributed cache or job configuration), makes it so that we cannot inform each mapper what attribute it should process. Even if we could do this, each mapper would have to iterate the entire dataset to be able to compute the IG for its given attribute, however since each mapper only gets a split of the input this would be impossible, not to mention very inefficient since it requires that the entire dataset is iterated for each existing attribute.

Here the best approach is to calculate the IG for every attribute simultaneously in each mapper of the job.

6.4 WHEN TO LAUNCH A MAPREDUCE JOB

Here the discussion is when should we launch a job and what should it be calculating. This is a very pertinent question since launching a job has considerable setup times (overheads). If we launch too many jobs, we will lose a lot of time setting up each one, but if we launch too few jobs we will lose

flexibility in the system, because each job will have to perform much more work.

Although we haven't yet explained how, assume that we have a MR job *J*, that iterates through the entire dataset and is able to compute the attribute that best splits the dataset under a given node.

Also imagine a dataset, $\mathbb{S}$, with various attributes, one being Blood Pressure. The execution of job *J* ascertained that for the dataset $\mathbb{S}$ the Blood Pressure is the best attribute for the root of the decision tree, and we are now faced with the choice of how to compute the best attribute for each of its child nodes, as shown in Figure 1.

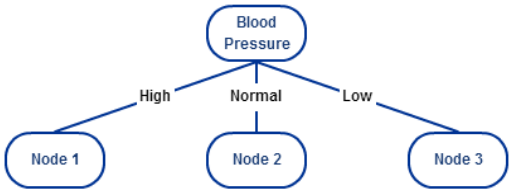

Figure 1 - An incomplete decision tree.

We could do this by launching the job for each of the child nodes or we could launch the job for the three child nodes simultaneously, that is, this single job would compute the best attribute for each node of an entire level of the decision tree. Although not obvious let us demonstrate that the best option is to launch a single job.

We know that each job must iterate the entire data set. So, in the job relative to the node 1, we would have to iterate through the instances where the Blood Pressure is Normal and Low, only to find that these instances do not fall within Node 1. This means all the work needed to traverse the decision tree for each of these instances would be wasted. And because we are launching three jobs we would be wasting even more time because of the setup time of each job, as well as the unnecessary iteration of the entire data set thrice.

If we instead launch a single job for every node, we will have the guarantee that every time we traverse the decision tree the example will always fall within a node we are interested, and therefore we will not waste any computation. With this approach we are trading computation for storage, but since the dataset is assumed to be big we will clearly reap a greater benefit with this strategy.

7 THE MRID4 ALGORITHM

MRID4 stands for *MapReduce Iterative Dichotomizer 4*. The number 4 was chosen because our algorithm also supports continuous attributes as proposed in the C4.5 algorithm [6] but it is not a MR implementation of the C4.5 algorithm. In other words, it sits between ID3 and C4.5 algorithms.

The problem of implementing ID3 in MapReduce comes down to: how to compute the IG for each attribute. So before explaining how the algorithm works we will first introduce the concept we call Class Counts which overcomes the very restrictive Equation 1, and allows an easy parallelization of the IG computation.

7.1 CLASS COUNTS

To explain what the class counts are, we need to introduce some formal definitions first.

Let $\mathbb{S}$ be a set of instances with cardinality $|\mathbb{S}|$, and $attr(\mathbb{S})$ the set of attributes describing the instances. For an attribute $a \in attr(\mathbb{S})$, $vals(a)$ represents the set of its possible values.

A training set $\mathbb{T}$ is a set of instances of the form $(\mathbb{x}, y) = (x_1, x_2, \ldots, x_n, y)$, where $x_i \in vals(a_i)$ is the value of the *i*-th attribute and $y \in \mathbb{C}$, with $\mathbb{C}$ the set of possible values for the class label. Additionally, consider that $\mathbb{T}_{x_i=v}$ is a subset of $\mathbb{T}$, where the *i*-th attribute has value $v \in vals(a_i)$, and $\mathbb{T}_{y=c}$ a subset of $\mathbb{T}$ where the class label has value $c \in \mathbb{C}$.

Given the usual equation for entropy (Equation 3).

$$H(\mathbb{S}) = -\sum_{c \in \mathbb{C}} \frac{|\mathbb{S}_{y=c}|}{|\mathbb{S}|} log_2 \frac{|\mathbb{S}_{y=c}|}{|\mathbb{S}|}$$

Equation 2 – Entropy of a set.

We can manipulate Equation 2 it into a more manageable representation (Equation 3).

$$H(\mathbb{S}) = \frac{|\mathbb{S}| \log|\mathbb{S}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{y=c}| \log|\mathbb{S}_{y=c}|}{|\mathbb{S}| \log 2}$$

Equation 3 – A more manageable representation for the entropy for a set.

Using this representation, we can write the IG in a more manageable form (Equation 4).

$$IG(\mathbb{S}, a) = H(\mathbb{S}) - \sum_{v \in vals(a)} \frac{|\mathbb{S}_{x_a=v}|}{|\mathbb{S}|} H(\mathbb{S}_{x_a=v})$$

$$= H(\mathbb{S}) - \frac{1}{|\mathbb{S}| \log 2} \sum_{v \in vals(a)} \left(|\mathbb{S}_{x_a=v}| \log|\mathbb{S}_{x_a=v}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{x_a=v \wedge y=c}| \log|\mathbb{S}_{x_a=v \wedge y=c}| \right)$$

Equation 4 – A more manageable representation for the information gain of a set for a given attribute.

The objective is to be able to compute the attribute with the highest IG in a given dataset, which means that the values $H(\mathbb{S})$ and $|\mathbb{S}|$ in Equation 4 are always constant. Since we are not interested in knowing the actual value of the IG, but rather the attribute with the highest IG we can transform this maximization problem into a minimization (Equation 5).

$$\max_{a \in attr(\mathbb{S})} IG(\mathbb{S}, a)$$

$$\propto \min_{a \in attr(\mathbb{S})} \sum_{v \in vals(a)} \left(|\mathbb{S}_{x_a=v}| \log|\mathbb{S}_{x_a=v}| - \sum_{c \in \mathbb{C}} |\mathbb{S}_{x_a=v \wedge y=c}| \log|\mathbb{S}_{x_a=v \wedge y=c}| \right)$$

Equation 5 – Simplified formula to calculate the attribute with maximum information gain.

Notice that since we are iterating through all the possible values for the attribute a, the expression $\sum_{v \in vals(a)} |\mathbb{S}_{x_a=v}|$ will always be equal to $|\mathbb{S}|$, similarly for a given a and v, the expression $\sum_{c \in \mathbb{C}} |\mathbb{S}_{x_a=v \wedge y=c}|$ will always be equal to $|\mathbb{S}_{x_a=v}|$. This means that if we have $|\mathbb{S}_{x_a=v \wedge y=c}|$ for each a, v and c we can compute (Equation 5) and therefore compute the attribute with the highest information gain.

The Class Counts are the counts $|\mathbb{S}_{x_a=v \wedge y=c}|$ for each a, v and c in the set $\mathbb{S}$. This can easily be visualized as a tri-dimensional associative array. The first dimension are the attributes (each a), the second dimension are the attribute values (each v) of the attribute, the third dimension are the class label values (each c) and the stored values correspond to the counts $|\mathbb{S}_{x_a=v \wedge y=c}|$.

7.2 How MRID4 Operates

There will exist a machine, the Central node, which will: schedule the execution of MR jobs; assign unique IDs to the decision tree nodes that will be processed in the next job; and read the outcome of each scheduled job into the decision tree, that is, it will maintain the model learned so far. The Central node will launch a job for each level in the tree.

Each MapReduce job will operate as follows: each mapper will calculate the partial[1] class counts, for each of the nodes in the current level of the decision tree. To perform this computation the mapper will need to have access to the decision tree computed so far, which will be available through the Distributed Cache. The output of the mappers will have as key the IDs of the decision tree nodes being calculated in the job, and as values the partial class counts for those nodes.

Each reducer will read the partial class counts and compute Equation 5, in order to find the attribute with the highest IG. The output of the reducers will have as key the node ID and as value the tuple $(Attribute, Final\ Class\ Counts)$. The $Attribute$ field will allow the Central node to know what attribute was chosen for the node. The $Final\ Class\ Counts$ field corresponds to the final[2] class counts for the chosen attribute and will be used to compute whether we are at a leaf node or if there are still child nodes that need to be computed.

[1] Partial in the sense that the computed class counts are only relative to the mapper split.

[2] Final in the sense that these class counts are relative to the entire dataset.

$$Map: \langle \_, Instance \rangle \rightarrow \langle NodeID, Partial\ Class\ Counts \rangle$$

$$Reduce: \langle NodeID, list\ of\ Partial\ Class\ Counts \rangle \rightarrow \langle NodeID, \langle Attribute, Final\ Class\ Counts \rangle \rangle$$

Equation 6 –Overview of the MapReduce job launched by the Central node.

Angle brackets represent key-value pairs and underscore means we don't care about the key.

To better explain how the final class counts can be used to determine whether the computed node has any child nodes. Suppose that the Central node encounters for a given attribute a the following final class counts:

	Yes	No
High	5	1
Normal	2	4
Low	3	3

Where Yes and No are the class label values (each c) and the High, Normal and Low are the attribute values (each v).

In this example 83% (5/6) of the instances with the value High for the attribute a have the class label Yes. If this percentage were to be 100% we clearly would be at a leaf node where the class label is Yes (the stop condition number 1 from ID3). In the real case, we cannot expect this percentage to reach 100% so we must define a threshold according to our needs. In our algorithm this value is, by default, 90%.

7.3 CONTINUOUS ATTRIBUTES

The major difference from the discrete attributes is the way the split points are considered. With discrete attributes every value of the attribute will correspond to a split point. So for example, a discrete attribute with 3 values will split the data set into three sub sets. With a continuous attribute, if we were to employ the same tactic the resulting number of sub sets would be enormous. So, we turned to the proposal by Quinlan for the C4.5 algorithm.

His proposal is as follows: for a continuous attribute being considered, its list of values, in a set $\mathbb{S}$, are sorted to give ordered values $v_1, v_2, \dots, v_n$; every pair of adjacent values suggests a potential threshold $\theta = \frac{v_i + v_{i+1}}{2}$ and the corresponding partitions $\mathbb{S}_{x_a \leq \theta}$ and $\mathbb{S}_{x_a > \theta}$; the threshold for which the resulting partitions have the best information gain is selected.

With discrete attributes the mappers know all the possible values for each of the discrete attributes. They leverage this fact by using a fixed sized data structure to send the class counts to the reducers. Using the same approach for continuous attributes would result in a lot of waste, since all the possible values for a continuous attribute will not be present in each split. So to resolve this issue, the mappers will send a dynamic structure to the reducers; this structure will only contain the values of the continuous attribute that are present in the mapper split. In other words, only the reduce will have access to the complete list of the values for a continuous attribute.

8 DATASET

Our work is being done under the educare project. And, as such we are required to follow the project main goals. This lead us to use an educational dataset, constructed using the data within the project's data warehouse.

8.1 DATASET CHARACTERIZATION

The dataset we used for testing had 1181 records, where each record corresponds to a unique student. Besides the class label attribute the dataset has: two attributes per subject in the degree of Information Systems and Computer Engineering at IST; the attribute number of approved subjects; the attribute total number of enrollments and the attribute total number of enrollments when the student reproved. Which amounts to 64 attributes.

The two attributes per subject correspond to the grade the student obtained at that subject, and the number of times the student enrolled at that subject.

The dataset has manually annotated using a binary class label classifying whether the student had quit the degree or not. The annotation resulted in 486 (41%) records having a quit class label

8.2 REPLICATION

Since the dataset is too small to be considered big data we replicated it. The replication has performed multiple times in order to achieve a range of different sizes: ~5 megabytes, ~50 megabytes, ~500 megabytes, ~5 gigabytes and ~50 gigabytes. And two replication schemes were used:

1. By dataset: we copied the dataset n times and appended each copy sequentially
2. By record: we copied each record n times and appended each one sequentially.

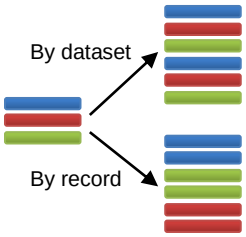

Figure 2 – Visual representation of the replication.

Since we are copying each record the same amount of times, the output of the algorithm will always be the same. We chose to replicate it in two different ways to show that the algorithm can handle unbalanced datasets and that this fact does not contribute, in a appreciable way, to its performance.

9 EXPERIMENTAL RESULTS

The experimental results were obtained by running MRID4 in the RNL cluster, with a fixed minimum support of 90%. For each dataset size and replication scheme, the algorithm was ran 5 times. The charts we represent use the average of those 5 times.

9.1 RUN TIME

The run time is the time it took to execute the algorithm from start to finish. This includes all the MR jobs, one for each level of the tree.

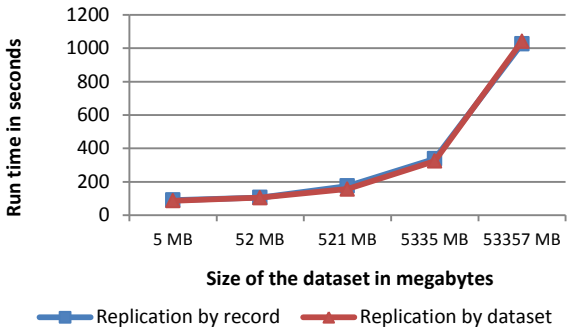

Figure 3 – Chart of the run times of the algorithm

Figure 3 shows the run times of the algorithm, as the size of the dataset increases, as well as, the run times obtained with the two different replication methods.

We can see that for sizes up to ~500 megabytes the run times are almost the same. This is coherent with the general opinion that for jobs where the dataset is smaller than 1 gigabyte, Hadoop introduces more overhead than it actually helps to perform the intended task. It is also visible that the replication method had almost no impact on the run time of the algorithm.

9.2 SCALABILITY

Scalability can be measured by how much more time the system takes to process a larger data set.

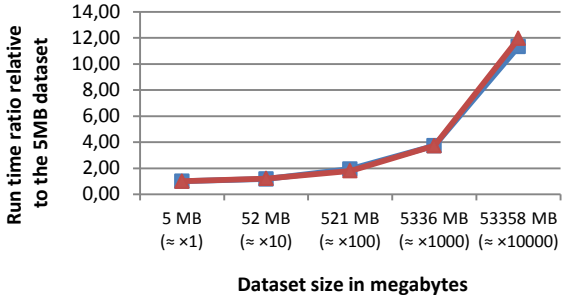

Figure 4 – The run time ratio relative to the 5MB dataset, as the dataset size increases.

Figure 4 represents the scalability of the algorithm by showing the ratio between the run time for a given dataset size and the run time for the dataset size of 5 MB. So as an example, for the dataset with 5336 MB, which is roughly 1000 times bigger than the dataset with 5 MB, the algorithm only took

3.73 times longer to run. This represents a very good scalability, which means that the algorithm with be able to run in a reasonable amount of time for very large datasets.

10 CONCLUSION

In recent years there have been many applications of MapReduce to implement data mining and machine learning algorithms. Many of these implementations have had great success. However in the area of classification not much work has been done. And the focus of the work that has been done seems to be in regression trees.

In this work we introduced an easy to understand notation and approach to implement and calculate metrics such as information gain or information gain ratio and implemented a classifier, capable of building classification trees. This classifier is based on the ID3 algorithm and is built using MapReduce. We showed that although the programming model already handles many of the challenges that come with a distributed and parallel computation, there is still a need to ponder carefully on some of the aspects that arise in this environment.

We showed, through experimental results, that our algorithm able to scale to datasets with very high amounts of data.

REFERENCES

[1] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Y. Yu, G. Bradski, Y. A. Ng and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," *Advances in Neural Information Processing Systems 19,* pp. 281-288, 2006.

[2] G. Wu, H. Li, X. Hu, Y. Bi, J. Zhang and X. Wu, "MReC4.5: C4.5 Ensemble Classification with MapReduce," *ChinaGrid Annual Conference,* vol. 4, pp. 249-255, 2009.

[3] B. Panda, J. S. Herbach, S. Basu and J. R. Bayardo, "PLANET: massively parallel learning of tree ensembles with MapReduce," *Proceedings of the VLDB Endowment,* vol. 2, no. 2, pp. 1426-1437, 2009.

[4] W. Yin, Y. Simmhan e V. K. Prasana, “Scalable regression tree learning on Hadoop using OpenPlanet,” *Proceedings of third international workshop on MapReduce and its Applications Date (MapReduce '12),* pp. 57-64, 2012.

[5] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning,* vol. 1, no. 1, pp. 81-106, 1986.

[6] J. R. Quinlan, C4.5: programs for machine learning, San Francisco, CA: Morgan Kaufmann Publishers Inc., 1993.