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

Vasile PURDILĂ, Ștefan-Gheorghe PENTIUC
1,2Stefan cel Mare University of Suceava, Romania
1vasilep@eed.usv.ro, 2pentiuc@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 seem 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 calculates 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)**

1. node = {create a 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.   endif
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[][key] = {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>
<thead>
<tr>
<th>Dataset size (bytes)</th>
<th>SF - dataset size (1 + 595,296,000 bytes)</th>
<th>Execution time (minutes)</th>
<th>SF - execution time (1 + 40 minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>595,296,000</td>
<td>1</td>
<td>40</td>
<td>1.00</td>
</tr>
<tr>
<td>2,381,184,000</td>
<td>4</td>
<td>45</td>
<td>1.13</td>
</tr>
<tr>
<td>4,762,368,000</td>
<td>8</td>
<td>54</td>
<td>1.35</td>
</tr>
<tr>
<td>9,524,736,000</td>
<td>16</td>
<td>59</td>
<td>1.48</td>
</tr>
<tr>
<td>14,287,104,000</td>
<td>24</td>
<td>77</td>
<td>1.91</td>
</tr>
<tr>
<td>19,049,472,000</td>
<td>32</td>
<td>93</td>
<td>2.33</td>
</tr>
<tr>
<td>24,811,840,000</td>
<td>40</td>
<td>114</td>
<td>2.85</td>
</tr>
<tr>
<td>28,574,208,000</td>
<td>48</td>
<td>138</td>
<td>3.45</td>
</tr>
<tr>
<td>33,336,576,000</td>
<td>56</td>
<td>174</td>
<td>4.35</td>
</tr>
<tr>
<td>38,098,944,000</td>
<td>64</td>
<td>200</td>
<td>5.00</td>
</tr>
<tr>
<td>42,861,312,000</td>
<td>72</td>
<td>234</td>
<td>5.95</td>
</tr>
<tr>
<td>47,623,680,000</td>
<td>80</td>
<td>255</td>
<td>6.88</td>
</tr>
<tr>
<td>52,385,048,000</td>
<td>88</td>
<td>280</td>
<td>7.00</td>
</tr>
<tr>
<td>57,147,416,000</td>
<td>96</td>
<td>297</td>
<td>7.43</td>
</tr>
<tr>
<td>61,909,784,000</td>
<td>104</td>
<td>315</td>
<td>7.89</td>
</tr>
<tr>
<td>66,672,152,000</td>
<td>112</td>
<td>340</td>
<td>8.50</td>
</tr>
<tr>
<td>71,435,520,000</td>
<td>120</td>
<td>367</td>
<td>9.18</td>
</tr>
<tr>
<td>76,209,888,000</td>
<td>128</td>
<td>380</td>
<td>9.50</td>
</tr>
<tr>
<td>80,983,256,000</td>
<td>136</td>
<td>391</td>
<td>9.78</td>
</tr>
<tr>
<td>85,722,624,000</td>
<td>144</td>
<td>410</td>
<td>10.25</td>
</tr>
<tr>
<td>90,484,992,000</td>
<td>152</td>
<td>430</td>
<td>10.75</td>
</tr>
<tr>
<td>95,247,360,000</td>
<td>160</td>
<td>451</td>
<td>11.28</td>
</tr>
<tr>
<td>100,009,728,000</td>
<td>168</td>
<td>470</td>
<td>11.75</td>
</tr>
</tbody>
</table>

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

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


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 Pentiu 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.