Scalable Random Forests for Massive Data

Bingguo Li, Xiaojun Chen, Mark Junjie Li, Joshua Zhexue Huang, and Shengzhong Feng

Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
Shenzhen 518055, China
{bg.li,xj.chen,jj.li,zx.huang,sz.feng}@siat.ac.cn

Abstract. This paper proposes a scalable random forest algorithm SRF with MapReduce implementation. A breadth-first approach is used to grow decision trees for a random forest model. At each level of the trees, a pair of map and reduce functions split the nodes. A mapper is dispatched to a local machine to compute the local histograms of subspace features of the nodes from a data block. The local histograms are submitted to reducers to compute the global histograms from which the best split conditions of the nodes are calculated and sent to the controller on the master machine to update the random forest model. A random forest model is built with a sequence of map and reduce functions. Experiments on large synthetic data have shown that SRF is scalable to the number of trees and the number of examples. The SRF algorithm is able to build a random forest of 100 trees in a little more than 1 hour from 110 Gigabyte data with 1000 features and 10 million records.

Keywords: MapReduce, Random forests, Histogram.

1 Introduction

Data with millions of records and thousands of features present a big challenge to current data mining algorithms. On one hand, it is difficult to build classification models from such massive data with serial algorithms running on single machines. On the other hand, most classification algorithms are not capable of building accurate models from extremely high dimensional data with thousands of features. However, such high dimensional massive data exist in many application domains, such as text mining, bio-informatics and e-commerce.

Random forests [1] is an effective ensemble model for classifying high dimensional data [2]. A random forest consists of K decision trees, each grown from a data set randomly sampled from the training data with replacement. At each node of a decision tree, a subset of m features is randomly selected and the node is split according to the m features. Breiman [1] suggested $m = Log_2(M) + 1$ where M is the total number of features in data. For very high dimensional data, M is very big and m is much smaller than M. Therefore, decision trees in a random forest are grown from subspaces of features [3][4][5]. The random forest classifies data according to the majority votes of individual decision trees. Due to the computation of a large number of decision trees, it is extremely difficult to build random forest models from large data sets with millions of records on single servers.

MapReduce [6][7][8] is a simple programming model for distributed computing. It abstracts away many low-level details such as task scheduling and data management, and conceptualizes a computational process as a sequence of map and reduce functions. In a map phase, the same map function is dispatched to all computing nodes and executes the same set of operations in parallel. In the reduce phase, the reduce function merges results from different mapper tasks. If the data file is large, more data blocks are created and distributed on more computing nodes. The map and reduce functions are automatically dispatched to more computing nodes. Therefore, MapReduce model is scalable to large data.

To implement the random forest algorithm in MapReduce, a straightforward method is to build one decision tree from each data block. This assumes that each data block is one training data sampled from the large training data set. The decision tree algorithm is implemented in one map function so it is dispatched to all computing nodes to build all decision trees from the local data blocks. The majority voting is performed in the reduce function. Such implementation is adopted in Apache Mahout[1]. Two main drawbacks in this simple implementation are: 1) The data blocks are hard partitions of the large training data and can have biased distributions different from the training data. This problem results in weak and biased trees; 2) As the map function builds a decision tree recursively, it can cause memory leakage if the tree is large and complex.

In this paper, we propose a new scalable random forest algorithm (SRF) for constructing random forest models from massive data. The SRF algorithm takes advantages of MapReduce programming model to gain high scalability on large data. Instead of using the recursive process to grow trees, a breadth-first tree growing method is adopted. Starting from the root nodes, all trees grow on level basis. The nodes of all trees on each level split up into children nodes in one pair of map and reduce functions. A random forest model is built with a sequence of D pairs of map and reduce functions where D is the depth of the highest tree in the forest. To split nodes on each level, one pair of map and reduce functions are used. The histograms of features for each node are calculated first in the map function and all sets of histograms for the same node in a tree are merged into one set of global histograms in the reducer job. We employ the method in SPDT [9] to calculate the histograms from each local data block and merge them into the global histograms to split the node. The reducer also calculates the node split function such as information gain and determines the split conditions of data to generate the children nodes. At start, a reference table is generated to record the data sets for different trees that are randomly sampled from the training data with replacement (bagging).

We have conducted a series of experiments on both synthetic and real-life data sets and compared SRF with SPDT and random forests in Mahout. The result showed that SRF obtained higher accuracy than SPDT and accuracy of SRF was similar to the random forests in Mahout. To further compare SRF with Mahout, we

[1] http://mahout.apache.org

added noise to OCR data set. On this noise data, the accuracy of random forests in Mahout reduced to 51.9% while the accuracy of SRF was 78.6%. On a separate sparse data Real-sim[2], Mahout random forests was not able to produce a model and generated stack overflow error message due to memory leakage but SRF worked fine. We used a large synthetic data with more than one thousand attributes and millions of records, to test the scalability of SRF. With 30 computing nodes, SRF was able to build a random forest with 100 trees in a little more than 1 hour from a massive data set of 110 Gigabytes with 1000 features and 10 million records. This was indeed a significant result. SRF also demonstrated a linear property with respect to number of trees and number of examples.

The rest of the paper is organized as follows. Section 2 gives a brief review of the SPDT algorithm. In Section 3, we present the SRF algorithm in details. We present experiment results on real-life data and scalability tests in Section 4. The paper is concluded in Section 5.

2 Related Work

Building decision trees is the major function of building a random forest. Traditional decision algorithms [10][11] use a recursive process to create a decision tree from a training data set. These algorithms will have problems if the training data or the tree is too big to fit in the main memory. Scalable decision tree algorithms have been proposed to handle large data. Some take an approach to pre-sort the training data before building the decision tree, such as SLIQ [12], SPRINT [13] and ScalParC [14]. Others compute the histograms of attributes and split the training data according to the histograms, such as BOAT [15], CLOUDS [16], SPIES [17] and SPDT [9]. The later are more scalable as the tree growing process is no longer relevant to the size of training data after all histograms are created. The creation of histograms can be easily parallelized. Google also proposed PLANET [18] for regression trees based on MapReduce programming model. PLANET only supports sampling without replacement.

In this work, we use a breadth-first method to construct decision trees for a random forest. We select the streaming parallel decision tree algorithm SPDT recently developed at IBM as a framework to develop the breadth-first tree growing process. Figure 1 sketches the process of the SPDT algorithm. It runs in a distributed environment with one master node and several workers. Each worker stores $1/W$ percentage of the data where W is the number of workers. To grow a decision tree, the master node instructs workers to compute the local histograms of features from their local data blocks. After local histograms are complete, the workers send them to the master which merges them into the global histograms. The global histograms are then used to compute the conditions to split the nodes and grow the decision tree. After the nodes in the same level are split, the master node instructs the workers again to compute histograms for the newly generated children nodes. This process continues until no node needs further split.

[2] http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html

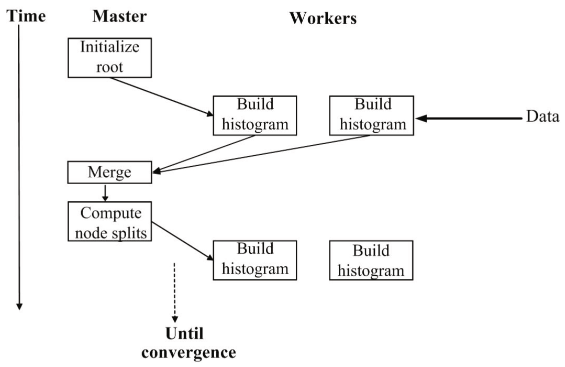


Fig. 1. The parallel framework of the SPDT algorithm

3 Scalable Random Forest Algorithm

In this section, we present a scalable random forest algorithm that uses SPDT node split method to grow decision trees in building a random forest model. The process of random forest building is implemented in the MapReduce programming model and runs on a distributed cloud computing platform.

3.1 Breadth-First Random Forest Construction

In a distributed environment, we create K decision trees for a random forest in parallel. Instead of the recursive process, we use the breadth-first method to create multiple decision trees. For example, to create three decision trees A, B, C in Figure 2, we first generate three root nodes. Then, we compute the best splits of the three root nodes in parallel and generate 6 children nodes, two from each root. We continue this process to generate grand children nodes, great grand children nodes, and etc. At each node, if the pre-defined stop conditions meet, the node is treated as a leaf node and no further split is necessary. After all leaf nodes are found, the tree growing process terminates and a random forest is obtained.

3.2 Scalable Random Forest Algorithm

With MapReduce programming model, we not only distribute map and reduce functions to create decision trees, but also partition the training data into data blocks and distribute them on different computing nodes. To create sample data sets for decision trees, we create an index table, called bagging table with K columns, each recording the examples that were selected in the sample data for a decision tree with the method of sampling with replacement. This bagging table is also distributed together with data blocks.

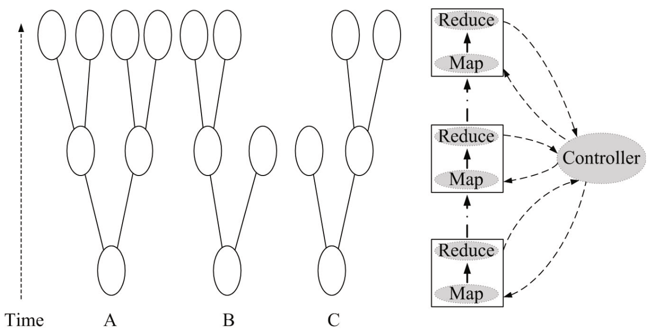


Fig. 2. Breadth-first method to grow decision trees for a random forest

The scalable random forest algorithm is composed of a sequence of map and reduce functions. Each mapper and reducer iteration creates one set of nodes in the same level of K trees, as shown in Figure 2. In the distributed cloud environment, each computing node stores a fixed number of data blocks. Each data block is accessed by one mapper dispatched by the controller that runs on the master node. The mapper is used to create the local histograms of subspace features for all nodes in the current level for all K trees. The split conditions of a node in a tree and the bagging table are used to select the objects that belong to the node. At a root node, no split condition exists and only the bagging table tells which objects in a data block are selected for the tree to be created.

After all local histograms are computed by all mapper jobs, they are sorted on the tree ids. The local histograms for the same tree are sent to the same reducer to compute the global histograms at each node. The reducer also calculates the best split from the global histograms and send the best split conditions back to the controller to update the random forest model. After a pair of map and reduce functions completed, each decision tree grows a new level of nodes, which are taken as the current level of nodes for the next pair of map and reduce functions. If a node satisfies the stop condition, it is marked as the leaf node.

3.3 Mapper, Reducer and Controller

The pseudocode of the mapper procedure is described in Algorithm 1. Each record of the data block is checked against the bagging table $\mathscr{T}$. If it belongs to the sample data for the tree, it is used to compute the local histograms of subspace features. After scanning all the training set,the local histograms for all decision trees are obtained.

At the end of the map phase, the mapper sends all local histograms in a set of (key, value set) pairs to reducers. The key is the id of a decision tree and the value is local histograms of nodes in that tree. In this way, we ensure all local histograms for the same decision tree are sent to one reducer.

Algorithm 1. Mapper Procedure

```
1: Input:
2: - $D^*$ : the training dataset;
3: - $\mathscr{M}$ : the random forests model, it contains the random forests constructed
   so far, nodes in the current level of all trees contain selection conditions of
   objects;
4: - $\mathscr{T}$ : the bagging table, it records the bagging indexes of each tree.
5:
6: Output: Generate histogram tuples hist-tup for nodes of all trees in the
   current level;
7: Method:
8: for each record $(x, y) \in D^*$ do
9:    for all decision tree $k \in \mathscr{M}$ do
10:       if tree $k$ has nodes to build AND the record is specified for tree $k$ in
          bagging table $\mathscr{T}$ then
11:          $hist\text{-}tup_k \leftarrow update(record, num)$ //update histograms for tree $k$
12:       end if
13:    end for
14: end for
15: for all decision tree $k \in \mathscr{M}$ do
16:    Output(treeId of $k$, $hist\text{-}tup_k$) // output pairs of tree id and histogram
       tuples
17: end for
```

The pseudocode of the reducer procedure is described in Algorithm 2. The reducer receives pairs of (key, value set) from different mappers. The key values are sorted ids of decision trees and the value sets are the local histograms of the trees from all mappers. The reducer merges the value sets of local histograms into global histograms with respect to the tree id and the node. The reducer computes the best split conditions from the global histograms for the node. If the split does not justify a split, the node is marked as a leaf node and no further split will occur. Otherwise, new children nodes are created under the node and the split conditions are recorded in node tuples for the children nodes. After all histograms are processed, the reducer sends the pairs of (id, *node-tuple*) to the controller. The controller adds the new nodes to the random forest model with the split conditions.

The pseudocode of the controller procedure is described in Algorithm 3. The controller starts with initialization of an empty model $\mathscr{M}$. Vector $\mathbb{N}$ is used to record the number of nodes to split in a level of each tree. $\mathbb{N}$ is initialized with each tree having one root node. $\mathbb{N} = [1, 1, 1]$ in the root level of Figure 2. Each iteration of a map and reduce pair generates a new $\mathbb{N}$ recording the number of nodes for each tree in the newly split level. For example, $\mathbb{N} = [2, 2, 2]$ in the level above the root and $\mathbb{N}$ becomes $[4, 2, 2]$ afterwards. $\mathbb{N} = [0, 0, 0]$ after the last level.

Algorithm 2. Reducer Procedure

```
1: Input:
2: - $k$ : the id of tree;
3: - $\mathbb{V}$ : the value set, receiving from different mappers.
4:
5: Output: Calculate the best split conditions at nodes for decision trees;
6: Method:
7: $hist\text{-}tup_k \leftarrow merge(\mathbb{V})$ // merge local histograms into global ones
8: for all built node $i$ in tree $k$ do
9:     The histogram of node $i$ is $hist\text{-}tup_k(i)$
10:    if $hist\text{-}tup_k(i)$ satisfies stop condition then
11:        $node\text{-}tup_i \leftarrow leaf$
12:    else
13:        $candidSplits \leftarrow Uniform(hist\text{-}tup_k(i))$ // compute the best split conditions
14:        $node\text{-}tup_i \leftarrow split$
15:    end if
16: end for
17: Output($k$, $node\text{-}tup$)
```

The loop of the controller checks whether $\mathbb{N}$ contains non-zero elements and terminates if all elements in $\mathbb{N}$ are zeros. Inside the loop, the controller configures a MapReduce job first and then dispatches it to the computing nodes to perform a pair of map and reduce functions on one level of nodes of all decision trees. The controller also passes the information of the nodes in the current level to each mapper for computing local histograms. The controller calculates the number of reducers for the MapReduce job according to the number of nodes in $\mathbb{N}$ to balance the load to reducers. After all reducers complete, parseOutput function processes the results from reducers and generates the new node information in *tree-tup*. The *tree-tup* data is used to update the random forest model $\mathscr{M}$ and computes a new $\mathbb{N}$ to record the number of new nodes generated in each tree.

4 Experiments

In this section we show the classification results of SRF on large complex data sets and comparisons of them with those by SPDT and Mahout. We also demonstrate the scalability of SRF to very large data sets. The results show the capability of SRF in building random forest models from extremely large data with 10 million records and 1000 features in less than 2 hours. Such models would be very difficult, if not impossible, to build via traditional random forest algorithms.

4.1 Data Sets

Two data sets were used in experiments. The first set was used to test the classification performance of SRF in accuracy and compare classification results

Algorithm 3. Controller Procedure

```
1: Input:
2: - $\mathscr{M}$ : the random forests model, $\mathscr{M} = \phi$;
3: - $\mathbb{N}$ : the vector, $\mathbb{N} = [1, 1, ...]$.
4:
5: Output: Construct a random forests model with MapReduce;
6: Method:
7: while $\mathbb{N}$ has non-zero elements do
8:     ConfigureMapReduce(job)
9:     Dispatch(job)
10:    $tree\text{-}tup = parseOutput(job)$
11:    $\mathbb{N} = updateForests(\mathscr{M}, tree\text{-}tup)$
12: end while
```

with those by SPDT and Mahout. Four real-life data sets were selected and these data sets were used in evaluating SPDT in [9]. The characteristics of these data sets are summarized in Table 1. They can be downloaded from UCI repository and Pascal Large Scale Learning Challenge[3].

Table 1. Real-life data sets used in accuracy experiments

Dataset	#Features	#Train Set	#Test Set	%Classes
Isolet	617	6,238	1,559	26
Multiple Features	649	2000	2000	10
Face Detection	900	1,000,000	100,000	2
OCR	1,156	1,000,000	100,000	2

The second set contained four synthetic data sets that were generated for scalability test of SRF. The four synthetic sets are listed in Table 2. The points in the same class in these data sets have Gaussian distributions. To generate separable classes in the synthetic data, we specified several central points with labels first and calculated the distance between a generated record and the central points. We set the class label of the record as the label of the central point that had the minimum distance to the record.

4.2 Experiment Settings

The experiment environment was composed of 30 machines, each having 8 cpus and 15 GB memory running CentOS Linux operating system. Hadoop was installed on these machines to form a MapReduce runtime environment. Each machine was configured with 8 mappers and 8 reducers. The size of data block was 64 MB by default.

[3] ftp://largescale.ml.tu-berlin.de/largescale

Table 2. Characteristics of synthetic sets for scalability evaluation

Dataset	#Features	#Train Set	#Classes	%Size of data(GB)
D1	1,000	10,000,000	5	110
D2	1,200	7,000,000	10	91.6
D3	1,400	5,000,000	15	76
D4	1,600	2,000,000	20	32.4

50 bins were used to build histograms. In performance experiment, a random forest had 100 decision trees and the size of subspace at each node was $log_2(M)+1$, where M was the number of features in training data.

In scalability experiment, we investigated the scalability of SRF with respect to four factors, i.e., the number of decision trees, the size of data, the size of data block and the number of machines. For the number of decision trees, we started with 1 single tree and added to 20 trees, then increased 20 more trees other times. For size of data, we started with 200,000 examples and increased the size of data with 200,000 more examples each time. For size of data block, we started with 16 MB of data block and double the size of data block each time. For the number of machines, we started with 15 machines, and increased 5 more machines each time.

4.3 Performance Results

Table 3 lists the classification results in term of accuracy of SPDT, Mahout and SRF on the four real-life data sets described in Table 1. We can see that the accuracies of SRF were higher than that of SPDT on all the four data sets. The most prominent result was from the OCR data set that had 1156 features and one million records. The increase of accuracy over SPDT was 19%. This result demonstrated that SRF could get better performance than SPDT in handling massive and high dimensional data, as SRF is a random forests algorithm, it can get better performance than decision tree algorithm (SPDT) in handling massive data.

Table 3. Accuracies of SPDT, Mahout and SRF on four real-life data sets

Dataset	#Acc.(%) of SPDT	#Acc.(%) of Mahout	#Acc.(%) of SRF
Isolet	77.42	92.6	92.8
Multiple Features	91.5	98.5	97
Face Detection	96.69	91.5	97.47
OCR	60.65	78.9	79.5

For Mahout and SRF, the accuracy of SRF was closely similar with Mahout random forests on Isolet and Multiple Features training data sets. The reason is that the size of these two training data sets was no more than 64 MB, Mahout and SRF constructed all decision trees based on one data block, thus these two algorithms degenerated into traditional random forests algorithms. On the other hand, SRF could obtained higher accuracy than Mahout random forests on massive data, such as Face Detection training set.

To illustrate the fact that random forests in Mahout builds all decision trees on the first data block when the number of blocks is larger than the number of decision trees, we added a block size of noise records, which was generated randomly for two labels, in front of the OCR training set. The accuracy of random forests in Mahout decreased rapidly from 78.9% to 51.9% while the accuracy of SRF decreased from 79.5% to 78.6% on the noised OCR training set. The reason is that, for Mahout, all decision trees were built on the first data block, which contained the added noise data. As a result, SRF outperforms Mahout in handling massive data.

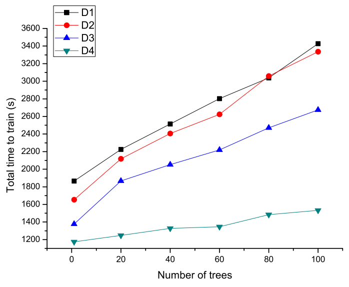
In addition, Mahout may lead to memory leakage problem while handling massive data. For example, Mahout random forests generated stack overflow error message when dealing with Real-sim training data set, which has 35,000 examples and 20,958 features. As random forests in Mahout built decision trees with depth-first mode, which may cause memory leakage problem.

4.4 Scalability

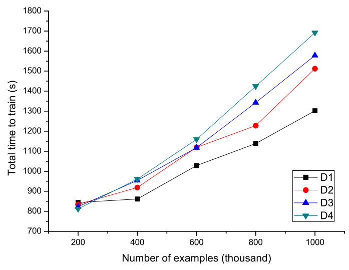
Figure 3 shows the scalability on four synthetic data sets with respect to the number of trees in random forests, the number of examples in data, the size of data block and the number of machines used. Figure 3(a) shows that the time used to build a random forest model increased linearly as the number of trees increased in the model. The run times for building one tree model for data sets D1-D4 are 1174s, 1377s, 1654s and 1866s respectively. The run time increases slowly as more trees are added to the model. For instance in data set D4, only less than 4 extra seconds were added to the total run time when each additional tree was added to the model. The larger the data set, more time it takes when more trees are added. However, the speed of time increase is very slow. This result demonstrates that SRF is scalable to the number of trees in the model.

Figure 3(b) shows the scalability of run time on four synthetic sets with respect to the number of examples in data. We can see a linear increase in time as more examples were added in building random forest models. When the number of examples was small, the differences of run times for the four data sets were very small. As more examples were involved, the run time increased but very slow. The large the data size, the larger the increase of run time. However, the speed of increase was not fast. This demonstrates that SRF is also scalable to the data size.

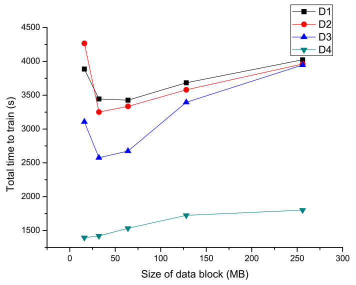
Figure 3(c) shows the change of run time over the change of the size of data block. The size of data block had impact on the run time of SRF. On the one hand, the smaller data block generates more mappers. However, if the number of mappers exceeds the mapper capacity of the system, the run time will increase rapidly. On the other hand, the larger data block generates heavy load mapper. From the chart, we can see that the proper size of data block is 32MB or 64MB for the data sets.



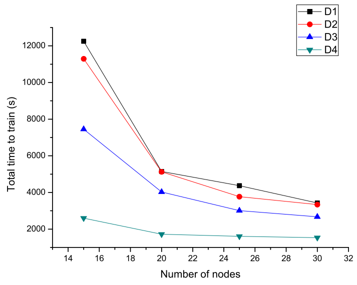
(a) Run time w.r.t. no. of trees.



(b) Run time w.r.t. no. of examples.



(c) Run time w.r.t. data block size.



(d) Run time w.r.t. no. of nodes.

Fig. 3. Scalability results on large synthetic data sets

Figure 3(d) shows the scalability of SRF with respect to the number of machines involved. We can see that the run time dropped rapidly as more machines added. This demonstrates that SRF is able to handle very large data by adding more machines.

5 Conclusions

We have presented the scalable random forest algorithm SRF and its implementation in MapReduce. In the algorithm, we adopted the breadth-first approach to build decision trees in a sequence of pairs of map and reduce functions to avoid the memory leakage in the depth-first recursive approach and make the algorithm more scalable. We have demonstrated the scalability of SRF with very large synthetic data sets and the results have shown SRF's ability in building random forest models from data with millions of records. Our future work is to further optimize SRF in the area of load balance to make it more efficient and scalable.

Acknowledgements. This research is supported in part by Shenzhen New Industry Development Fund under grant No.CXB201005250021A.

References

1. Breiman, L.: Random forests. Machine Learning 45(1), 5–32 (2001)
2. Banfield, R., Hall, L., Bowyer, K., Kegelmeyer, W.: A comparison of decision tree ensemble creation techniques. IEEE Transactions on Pattern Analysis and Machine Intelligence, 173–180 (2007)
3. Ho, T.: Random decision forests. In: Proceedings of the Third International Conference on Document Analysis and Recognition, vol. 1, pp. 278–282. IEEE (1995)
4. Ho, T.: C4.5 decision forests. In: Proceedings of Fourteenth International Conference on Pattern Recognition, vol. 1, pp. 545–549. IEEE (1998)
5. Ho, T.: The random subspace method for constructing decision forests. IEEE Transactions on Pattern Analysis and Machine Intelligence 20(8), 832–844 (1998)
6. White, T.: Hadoop: The definitive guide. Yahoo Press (2010)
7. Venner, J.: Pro Hadoop. Springer (2009)
8. Lam, C., Warren, J.: Hadoop in action (2010)
9. Ben-Haim, Y., Tom-Tov, E.: A streaming parallel decision tree algorithm. The Journal of Machine Learning Research 11, 849–872 (2010)
10. Breiman, L.: Classification and regression trees. Chapman & Hall/CRC (1984)
11. Quinlan, J.: C4.5: Programs for machine learning. Morgan Kaufmann (1993)
12. Mehta, M., Agrawal, R., Rissanen, J.: Sliq: A Fast Scalable Classifier for Data Mining. In: Apers, P.M.G., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 18–32. Springer, Heidelberg (1996)
13. Shafer, J., Agrawal, R., Mehta, M.: Sprint: A scalable parallel classifier for data mining. In: Proceedings of the International Conference on Very Large Data Bases, pp. 544–555. Citeseer (1996)
14. Joshi, M., Karypis, G., Kumar, V.: Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets. In: Proceedings of the First Merged International and Symposium on Parallel and Distributed Processing, Parallel Processing Symposium, IPPS/SPDP 1998, pp. 573–579. IEEE (1998)
15. Gehrke, J., Ganti, V., Ramakrishnan, R., Loh, W.: Boatoptimistic decision tree construction. In: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, pp. 169–180. ACM (1999)
16. AlSabti, K., Ranka, S., Singh, V.: Clouds: Classification for large or out-of-core datasets. In: Conference on Knowledge Discovery and Data Mining (1998)
17. Jin, R., Agrawal, G.: Communication and memory efficient parallel decision tree construction. In: 3rd SIAM International Conference on Data Mining, San Francisco, CA (2003)
18. Panda, B., Herbach, J., Basu, S., Bayardo, R.: Planet: massively parallel learning of tree ensembles with mapreduce. Proceedings of the VLDB Endowment 2(2), 1426–1437 (2009)