Distributed forests for MapReduce-based machine learning

Ryoji Wakayama†, Ryuei Murata†, Akisato Kimura‡,
Takayoshi Yamashita†, Yuji Yamauchi†, Hironobu Fujiyoshi†
† Chubu University, Japan. ‡ NTT Communication Science Laboratories, Japan.
akisato@ieee.org, hf@cs.chubu.ac.jp

Abstract

This paper proposes a novel method for training random forests with big data on MapReduce clusters. Random forests are well suited for parallel distributed systems, since they are composed of multiple decision trees and every decision tree can be independently trained by ensemble learning methods. However, naive implementation of random forests on distributed systems easily overfits the training data, yielding poor classification performances. This is because each cluster node can have access to only a small fraction of the training data. The proposed method tackles this problem by introducing the following three steps. (1) "Shared forests" are built in advance on the master node and shared with all the cluster nodes. (2) With the help of transfer learning, the shared forests are adapted to the training data placed on each cluster node. (3) The adapted forests on every cluster node are returned to the master node, and irrelevant trees yielding poor classification performances are removed to form the final forests. Experimental results show that our proposed method for MapReduce clusters can quickly learn random forests without any sacrifice of classification performance.

1. Introduction

Statistical machine learning has become one of the most important techniques in the field of computer vision and pattern recognition, and it covers a broad range of fundamental tasks such as classification, regression and clustering. In statistical machine learning, the amount of training data is one of the most significant factors in terms of achieving better performances. The emergence of the internet and the widespread use of social networking services make it easy to obtain huge numbers of training samples, which implies that we can easily obtain predictors with a lot of better performances than ever before if we have much time for training. However, the increase in training data causes an increase in computational costs incurred for training.

Parallel distributed processing is a promising approach for solving this problem. Large training tasks can often be divided into smaller sub-tasks that are then solved simultaneously with multi-core processors or computer clusters. In particular, MapReduce [4] is suitable for parallelizable tasks across huge data sets with a large number of computers (nodes). A MapReduce program is composed of two procedures: a Map procedure that performs a parallelized sub-task on every worker node, and a Reduce procedure that receives all the results of the Map procedures and performs a summary operation on a master node. Provided that each Map operation is independent of all others, all the Map operations can be performed in parallel. In addition, MapReduce can take advantage of the locality of data, and process them on or near the storages to reduce the distance over which it must be transmitted.

This paper proposes a novel method for statistical machine learning with big data on MapReduce clusters. In particular, we focus on the use of random forests [2], a class of multi-class classifiers. Random forests are well suited to parallel distributed processing, since it is composed of multiple decision trees and every decision tree can be indepen-

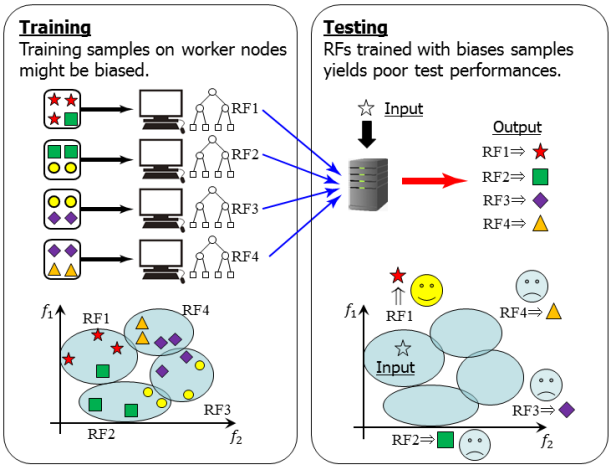

Figure 1. Drawback of naive MapReduce implementation of random forest. It easily overfits the training data, since each worker node can have access to only a small part of the data.

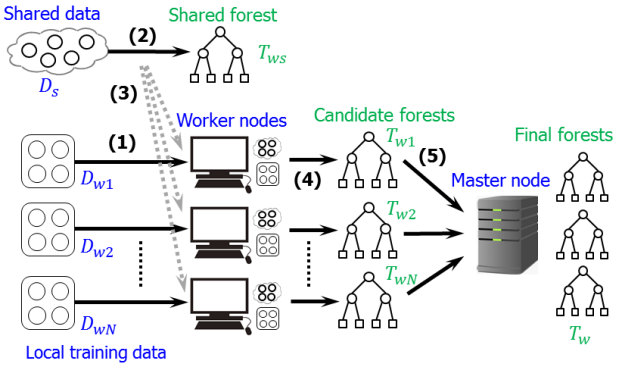

Figure 2. Map operation on distributed file systems.

dently trained. Several previous studies [7, 6, 1] have already reported parallel implementations of random forests and great improvements in terms of computational costs. However, those implementations have a crucial drawback: the resulting random forests easily overfits the training data, since each worker node on a MapReduce cluster can have access to only a small fraction of the training data.

The proposed method named *Distributed Forest* enables us to accelerate the training of random forests with the help of MapReduce clusters while maintaining the classification performances, by introducing the following three novel components. (1) We introduce "shared forests" built in advance on the master node and shared with all the worker nodes. (2) With the help of transfer learning [9], the shared forests are appropriately adapted to the local training data distributed on each worker node, even though there is only a small amount of highly biased local training data. (3) The adapted forests on every worker node are returned to the master node, and trees responsible for poor classification performances are eliminated to form the final forests.

2. Distributed forest on distributed file systems

This section describes the framework of our proposed method Distributed Forest. We first consider an application scenario for distributed file systems, where all the training data $\mathcal{D}_w$ have already been splitted into local training data $\{\mathcal{D}_{w1}, \mathcal{D}_{w2}, \ldots, \mathcal{D}_{wN}\}$ and distributed to worker nodes $1, 2, \ldots, N$. Therefore, each worker node i has access to its local training data $\mathcal{D}_{wi}$ for training. This is a typical setting in the MapReduce framework, and it can take advantage of the locality of data to reduce the cost of data transactions between master and worker nodes. Meanwhile, local training data might be biased and the degree of biases cannot be controlled, which often causes overfitting of the final random forests.

We assume that another type of training data $\mathcal{D}_s$ named *shared data* and a random forest $\mathcal{T}_s$ named *a shared forest* built from the shared data are shared with all the worker nodes prior to Map operations. The shared forest can be built on the master node in advance only just once. The shared data can be regarded as an auxiliary domain, and meanwhile every local training data can be viewed as a target domain, in the context of transfer learning. Therefore, any transfer learning methods for random forests can be applicable as a Map operation (See Section 6 for the detail of transfer learning), and the shared forests are appropriately adapted to the local training data $\mathcal{D}_{wi}$ on each worker node i, even though there is only a small amount of highly biased local training data. Random forests $\mathcal{T}_{wi}$ obtained on all the worker nodes $i = 1, \ldots, N$ are transferred to the master node for further Reduce processings (cf. Section 5).

Figure 2 shows an outline of the Map operation of the proposed Distributed Forest in this scenario, and Algorithm 1 shows the algorithm.

Algorithm 1 Map operation on distributed file systems

Require: Local training data $\mathcal{D}_w = \cup_{i=1}^{N} \mathcal{D}_{wi}$, shared data $\mathcal{D}_s$.

(Preprocessing)

1. Each local data $\mathcal{D}_{wi}$ is placed on the worker node i.
2. Build a shared forest $\mathcal{T}_s$ from the shared data $\mathcal{D}_s$ on the master node in advance.

(Main operations)

3. Distribute the shared data $\mathcal{D}_s$ and shared forest $\mathcal{T}_s$ to all the worker nodes.
4. Build a local forest $\mathcal{T}_{wi}$ by Transfer Forest, from the local data $\mathcal{D}_{wi}$, shared data $\mathcal{D}_s$ and shared forest $\mathcal{T}_s$.
5. Return the local forest $\mathcal{T}_{wi}$ to the master node.

Ensure: Local forests $\mathcal{T}_w = \cup_{i=1}^{N} \mathcal{T}_{wi}$.

3. Distributed forest on a single architecture

The Map operation proposed in the previous section can be easily extended to a rather traditional scenario where all the nodes (including the master node) are packed in a single computer in which all the training data are stored. In this scenario, we can assign local training data $\mathcal{D}_{wi}$ to every worker node i so that all the local training data have almost the same sample distribution. However, there is still a risk of overfitting, since each worker node i has access to only a small number of training samples especially when utilizing a lot of worker nodes (i.e. N is large) to accelerate the training process.

We again employ the framework of transfer learning as with distributed file systems. In this scenario, we do not have to prepare additional training data as shared data, and instead we can assign a part of the training data for it. However, we again assume that shared data and a shared forest are shared with all the worker nodes, for simplicity.

Figure 3 shows an outline of the Map operation in this

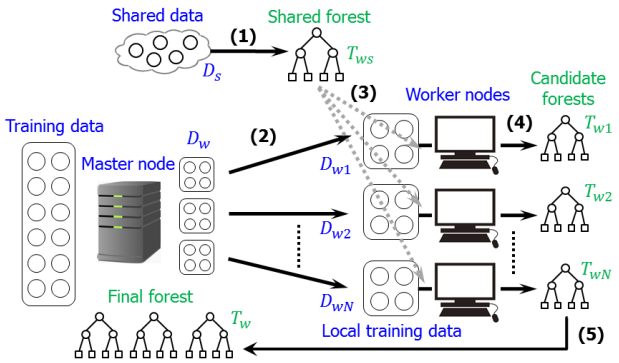

Figure 3. Map operation on many-core architectures.

Algorithm 2 Map operation on many-core architectures.

Require: Training data $\mathcal{D}_w$, shared data $\mathcal{D}_s$.

(Preprocessing)

1. Build a shared forest $\mathcal{T}_s$ from the shared data $\mathcal{D}_s$ on the master node in advance.

(Main operations)

2. Split the training data $\mathcal{D}_w$ into local data $\{\mathcal{D}_{w1}, \mathcal{D}_{w2}, \ldots, \mathcal{D}_{wN}\}$, and assign each $\mathcal{D}_{wi}$ to the corresponding worker node i.
3. Distribute the shared data $\mathcal{D}_s$ and shared forest $\mathcal{T}_s$.
4. Build a local forest $\mathcal{T}_{wi}$ by using Transfer Forest, from the local data $\mathcal{D}_{wi}$, shared data $\mathcal{D}_s$ and shared forest $\mathcal{T}_s$.
5. Return the local forest $\mathcal{T}_{wi}$ to the master node.

Ensure: Local forests $\mathcal{T}_w = \cup_{i=1}^{n} \mathcal{T}_{wi}$.

scenario, and Algorithm 2 shows the algorithm.

4. Distributed forest for unlabeled local data

The Map operation proposed in the previous section can be applied to the case in which no class labels are provided for local training samples. This is commonly observed when we obtain local training data on each worker node in a stream fashion. We cannot directly exploit those unlabeled data for training. However, remembering that the shared forest has already been distributed to every worker node, we can employ those unlabeled data for training by making the full use of this shared forest to estimate class labels of the local training samples. This strategy is called self-training in the context of semi-supervised learning [3]. This self-training strategy assumes that class predictions by the shared forest are expected to be correct, and thus it might not work well for highly biased local training data. However, this strategy is still useful in practice especially for single architectures.

5. Reduce operation

The purpose of the Reduce operation is to eliminate undesirable decision trees from the forests obtained from the worker nodes. We want to remove as many decision trees in a forest as possible since the number of decision trees directly influences the computational costs in the testing stage, meanwhile blindly removing decision trees sacrifices the classification performance. Although a method for removing irrelevant leafs and nodes of random forests that do not contribute to the improvement of classification performances has already been proposed [8], the way of selecting irrelevant trees is still unknown, to the best of our knowledge.

Algorithm 3 Reduce operation

Require: A set of random forests $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_i, \ldots, \}$, each forest $\mathcal{T}_i$ is obtained from the worker node i.

1. Integrate all the forests into a single forest $\mathcal{T}$.
2. Compute a score for each decision tree t in the forest $\mathcal{T}$ with Eq. (1).
3. Remove decision trees from $\mathcal{T}$ in the ascending order of scores until the number of trees in the forest $\mathcal{T}$ reaches a specified number.

Ensure: The final forest $\mathcal{T}$

We propose a method for eliminating irrelevant trees based on posterior distributions. In the testing stage, an input sample is assigned to a class that yields the maximum posterior probability, and the posterior probability of a class is the sum of the posterior probabilities computed from the decision trees in the forest. This implies that decision trees yielding higher posterior probabilities for incorrect class labels should be eliminated. Based on this intuition, we compute a score for each decision tree t as follows:

$$Score(t) = -\sum_{s \in \mathcal{S}^I} \max_{c \neq c(s)} p(c|s;t), \tag{1}$$

where $\mathcal{S}^I$ is a set of samples for this score computing, $c(s)$ is the ground-truth label of a sample s, and $p(c|s;t)$ is a posterior probability of a class c for a given sample s and tree t.

Algorithm 3 summarizes the algorithm of the Reduce operation. This Reduce operation can be utilized both for distributed file systems and single architectures, however, the details of the computing scores Eq. (1) are different from each other, especially in terms of how to select a set $\mathcal{S}^I$ for score computing. With distributed file systems, we have to prepare additional samples that are different from the shared and local training data. Meanwhile on single architectures, we can utilize out-of-bag (OOB) samples of all the worker nodes for this purpose.

6. Transfer learning of random forests

Our proposed method Distributed Forest can utilize any types of transfer learning methods for random forests. However in this paper, we particularly exploit Transfer Forest

[9] based on covariate shift loss. During the training process of Transfer Forest, auxiliary samples are iteratively reweighted by evaluating distances from the target domain using the covariate loss between the target sample and auxiliary distribution. The covariate loss can be estimated from the classification output of two forests, that is, the auxiliary forest that has already been constructed with auxiliary samples and the target forest that will be constructed with target and auxiliary samples.

The training process of Transfer Forest consists of the following four steps.

Step 1: Generating subsets We randomly select samples from both auxiliary and target samples to form a subset of samples, where the number of samples selected from each (i.e. auxiliary and target samples) is equal.

Step 2: Building a decision tree We train a decision tree from the subset using covariate losses $\Lambda = \{\lambda_k\}_{k, s_k \in \mathcal{D}_s}$ computed at Step 3 for all the samples. More specifically, every sample s_k is weighted by the corresponding covariate loss λ_k.

Step 3: Updating covariate losses We include the decision tree trained in Step 2 as a candidate for the target forest $\mathcal{T}_w$. A covariate loss λ_k of an auxiliary sample $s_k \in \mathcal{D}_s$ is updated using the target $\mathcal{T}_w$ and auxiliary forests $\mathcal{T}_s$ as follows:

$$\lambda_k = \frac{1 + \exp\{p(c(s_k)|s_k; \mathcal{T}_s)\}}{1 + \exp\{p(c(s_k)|s_k; \mathcal{T}_w)\}}, \qquad (2)$$

All the covariate losses of target samples are set to 1.

Step 4: Forming the final forest Steps 1-3 are repeated until a large number of candidates $\mathcal{T}_s$ for the target forest are obtained. The latter half of the candidates is selected as the final target forest.

7. Experiments

We utilized Letter Recognition dataset [5] distributed on UCI Machine Learning repository, to evaluate our proposed method Distributed Forest. This dataset consists of 20,000 samples with 16 dimensions and 26 classes. In this experiment, we used 6,666 samples as test data, and the rest (13,334 samples) as training data. From among all the training data, we used 2,000 or 4,000 samples as shared data, and the rest (11,334 or 9,334 samples) as local training data. We used the same parameters of decision trees for all the methods, namely the number of trees was 50, and the maximum depth of trees was 15. The platform used in this evaluation is as follows: Intel(R) Xeon(R) CPU E5-2637 v2 (3.50GHz) ×2, 32GB RAM, Windows Server(R) 7 Professional, Microsoft Visual C++ 2012.

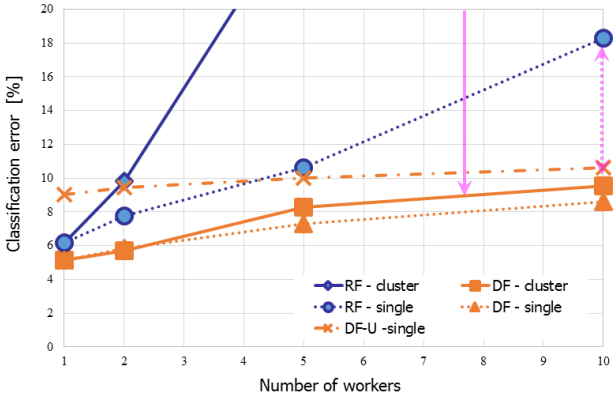

Figure 4. Trend of classification error [%]

Table 1. Classification error (shared = 4,000 samples) [%]

	cluster		single		
# workers	RF	DF	RF	DF	DF-U
1	6.18	5.14	6.19	5.14	9.03
2	9.82	5.71	7.78	5.83	9.45
5	26.4	8.28	10.6	7.29	10.0
10	67.7	9.53	18.3	8.59	10.6

First, we compared the proposed method Distributed Forest (cf. DF) with a naive implementation of random forest (cf. RF) training for parallel processing in terms of classification accuracy. We simulated two application scenarios: distributed file systems (cf. cluster) where the distribution of local training data on each worker node were heavily biased and very different from each other, and single architectures (cf. single) where the distribution of local training data were all almost the same. For the single architecture scenario, we also examined the performance of Distributed Forest for unlabeled data (DF-U) described in Section 4, where all the class labels of local training data were hidden.

Table 1 and Figure 4 show the classification errors. Those results indicate that a naive implementation considerably degraded classification performances, especially in the scenario of distributed file systems. On the other hand, our proposed Distributed Forest achieved low classification errors for both scenarios, which was comparable to that without any parallelizations. The results also indicate that Distributed Forest for unlabeled data was also very close to (fully supervised) random forest even though all the local training data were unlabeled.

Next, we examined the effectiveness of our Reduce operation. The purpose of the Reduce operation is to reduce the number of decision trees while maintaining the classification performance. As described in Section 5, random forests consisting of more decision trees tend to yield better performances in general. In this experiment, we removed

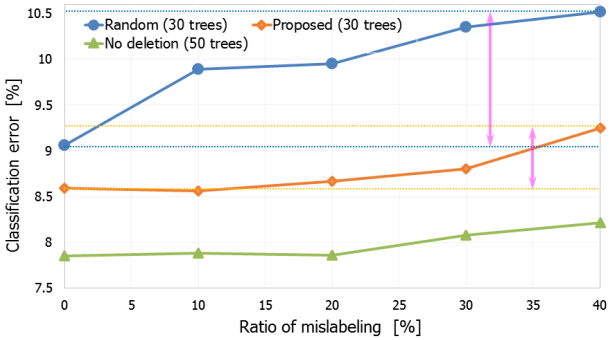

Figure 5. Effects of Reduce operation

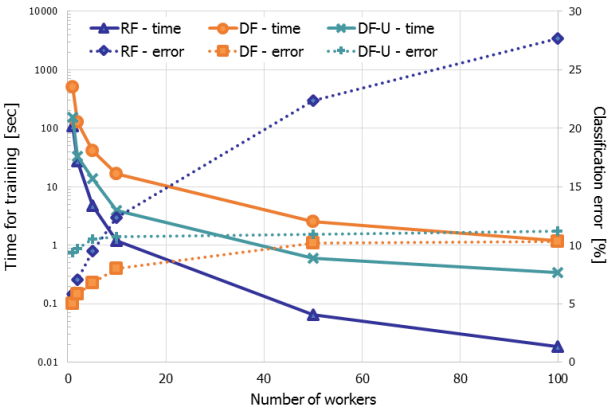

Figure 6. Simulated computational costs for training

20 trees from candidates obtained from 10 worker nodes. To make it clear the effectiveness of the proposed method, we intentionally injected mislabeled samples (i.e. samples with incorrect class labels) into the local training data on several worker nodes. By doing this, irrelevant decision trees were often generated on those worker nodes.

We compared our proposed method ($50 - 20 = 30$ trees) with random eliminations (30 trees, cf. random) as a simple baseline and no eliminations (50 trees) in terms of classification errors. Figure 5 shows the result, where the horizontal axis is the ratio of worker nodes on which mislabeled samples were injected. This result indicates that our proposed Reduce operation effectively removed irrelevant trees while maintaining the classification performance and it was much better than random eliminations.

Finally, we compared the proposed Distributed Forest with a naive implementation of random forests in terms of (simulated) computational costs. In this experiment, the number of decision trees was set to 100, and the number of worker nodes varied from 1 to 100.

Figure 6 shows the computational costs (seconds in CPU time) and classification errors against the number of worker nodes. The results indicate that the proposed method can accelerate the training process as the number of worker nodes increases while maintaining classification performances. The results also indicate that our method for unlabeled data greatly reduced the computational cost compared with that for fully labeled data.

8. Conclusion

This paper proposed a novel method named Distributed Forest for training random forests on parallel (distributed) architectures. Our Distributed Forest was built on the MapReduce framework to accelerate the training process of random forests while maintaining the classification performances with the benefit of transfer learning and posterior-based tree elimination. We proposed algorithms of Distributed Forest for two possible application scenarios, distributed file systems and many-core single architectures. Future work will includes more sophisticated implementations of Distributed Forest built on real MapReduce architectures, experiments with much larger datasets, and extensions to other types of machine learning tasks such as regression and density estimation.

References

[1] J. Assuncao, P. Fernandes, L. Lopes, and S. Normey. Distributed stochastic aware random forests - Efficient data mining for big data. In *Proc. IEEE International Conference on Big Data (BigData)*, pages 425–426, 2013. 2

[2] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, Oct. 1996. 1

[3] O. Chapelle, B. Schoelkopf, and A. Zien, editors. *Semi-Supervised Learning*. The MIT Press, 1 edition, 1 2010. 3

[4] J. Dean and S. Chemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 1

[5] P. M. Frey and J. Slate, David. Letter recognition using Holland-style adaptive classifiers. *Machine Learning*, 6(1):161–182, 1991. 4

[6] J. Han, Y. Liu, and X. Sun. A scalable random forest algorithm based on MapReduce. In *Proc. IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 849–852, 2013. 2

[7] B. Li, Z. Chen, M. J. Li, J. Z. Huang, and S. Feng. Scalable random forests for massive data. In *Proc. Pasific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 135–146, 2012. 2

[8] S. Ren, X. Cao, Y. Wei, and J. Sun. Global refinement of random forest. In *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015. 3

[9] S. Tsuchiya, R. Yumiba, Y. Yamauchi, T. Yamashita, and H. Fujiyoshi. Transfer forest based on covariate shift. In *Proc. IAPR Asian Conference on Patten Recognition (ACPR)*, 2015. 2, 4