Cognitive Computation (2019) 11:294–316
https://doi.org/10.1007/s12559-018-9615-4

Mining Big Data with Random Forests

Alessandro Lulli[1] · **Luca Oneto**[1] · **Davide Anguita**[1]

Received: 23 June 2018 / Accepted: 12 November 2018 / Published online: 3 January 2019

Abstract

In the current big data era, naive implementations of well-known learning algorithms cannot efficiently and effectively deal with large datasets. Random forests (RFs) are a popular ensemble-based method for classification. RFs have been shown to be effective in many different real-world classification problems and are commonly considered one of the best learning algorithms in this context. In this paper, we develop an RF implementation called ReForeSt, which, unlike the currently available solutions, can distribute data on available machines in two different ways to optimize the computational and memory requirements of RF with arbitrarily large datasets ranging from millions of samples to millions of features. A recently proposed improved RF formulation called random rotation ensembles can be used in conjunction with model selection to automatically tune the RF hyperparameters. We perform an extensive experimental evaluation on a wide range of large datasets and several environments with different numbers of machines and numbers of cores per machine. Results demonstrate that ReForeSt, in comparison to other state-of-the-art alternatives such as MLlib, is less computationally intensive, more memory efficient, and more effective.

Keywords Random forest · Random rotation ensembles · Model selection · Big data · Apache Spark · Memory efficiency · Computational efficiency · Open source software

Introduction

In the current big data era, 2.5 EB of data is generated each day by individuals, media, industries, scientific research, connected devices [1, 41, 43, 45], and sensors; moreover, 90% of the available data have been generated in the past 2 years [64]. Only a small amount of these data are exploited to derive new actionable information, since datasets are often imperfect, complex, and large [25]. Hence, it is extremely important to develop tools that can overcome such limitations [12, 33, 34, 38, 70]. Developing efficient and effective implementations of algorithms to extract new actionable information is thus of paramount importance. One of the most common solutions is to exploit the most effective learning strategies and adapt them to distribute computation to multiple machines and obtain efficiency and scalability [38].

Random forests (RF) of tree classifiers were proposed by Breiman [11] in 2001. They are considered state-of-the-art learning algorithms for classification purposes since they have shown to be one of the most effective tools in this context [18, 58]. From a cognitive point of view, RFs implement the wisdom of crowds principle, namely the aggregation of information in groups, resulting in decisions that are often better than could have been made by any single member of the group [19, 31, 61]. The main requirements behind this principle, which yields better results, is that there should be significant differences or diversity among the models. There are many examples of the use of this principle in literature, especially in cognitive computation research [10, 13, 26, 30, 31, 39, 47, 61, 65]. For these reasons, RFs combine bagging and random subset feature selection. In bagging [71], each tree is independently constructed using a bootstrap sample of the dataset. RFs add an additional layer of randomness to bagging. In addition to constructing each tree using a different bootstrap sample

✉ Luca Oneto
luca.oneto@unige.it

Alessandro Lulli
alessandro.lulli@dibris.unige.it

Davide Anguita
davide.anguita@unige.it

[1] DIBRIS Department, University of Genoa, Via Opera Pia 13, I-16145, Genoa, Italy

of the data, RFs change how the classification trees are constructed. In standard trees, each node cut is chosen by searching for the best among all features. In RFs, instead, each node cut is chosen by searching for the best among a subset of features randomly chosen at that node. Eventually, the mode among the trees composing the forest is taken as the prediction. RFs have been recently further improved in [9] with random rotation ensembles (RRE), which purport to avoid the initial bootstrapping and subset feature selection at each node in constructing the trees and replace it with a random rotation of the numerical feature space before learning each tree in the forest. The purpose of this new way of adding randomness is to improve the accuracy of the individual predictors while still preserving the diversity of the full ensemble. A common misconception about RFs is that the algorithm is a hyperparameter-free learning algorithm [5, 7, 46]. Indeed, there are several hyperparameters that characterize the performance of the final model, such as the number of trees, the depth of each tree, and the number of predictors exploited in each subset during the growth of each tree. For these reasons, to achieve satisfactory generalization performance, a model selection (MS) procedure is needed to select an optimal set of hyperparameters [2, 28]. Another common misconception about RFs is that they cannot overfit. Indeed, just mentioning the no free lunch theorem [63] would be enough to support our statement. In the literature, there are works that clearly show that, in particular situations, RFs can overfit [20, 54], but in practical situations, RFs show quite satisfying levels of accuracy [58]. Correct tuning of the hyperparameters of RFs during MS can reduce this issue but not completely remove it.

Unfortunately, RFs is a computationally intensive algorithm that needs to be reengineered to be applied to arbitrarily large datasets. In this context, the most common solution is to distribute the computation among multiple machines [38, 42, 64]. The data is partitioned on the available machines, and each machine executes the same program on its portion of data. Among the many approaches to performing this kind of computation (e.g., MPI [49] and MapReduce [16]), Apache Spark [69] is one of the most appealing since it enables reliable and fast iterative in-memory computation. Moreover, it provides a large set of operations, which allow the development of complex algorithms.

In this paper, we extend ReForeSt [35, 36], an Apache Spark–based distributed, scalable implementation of the RF learning algorithm that targets fast and memory efficient computation. ReForeSt shows better memory and computational efficiency than the de-facto standard MLlib [38] without reducing the accuracy of the final model [36]. In this work, we extended ReForeSt both by reengineering the original version and by including new features, thus obtaining a library able to outperform state-of-the-art alternatives in terms of both performance and features. In particular:

- The computational and memory requirements of ReForeSt are now optimized. It has lower memory requirements than its state-of-the-art competitors and automatically arranges computation according to the available memory;
- ReForeSt can distribute the data in two different manners and automatically select the best one based on the problem under consideration;
- ReForeSt can deal with arbitrarily large datasets, ranging from millions of samples to millions of features, whereas alternative implementations fail to provide a result;
- ReForeSt implements both the original RF formulation and the novel RRE;
- ReForeSt implements an efficient MS procedure for the whole set of hyperparameters. It allows us to introduce many optimizations during the learning of the trees to complete the computation in less time than does checking all the possible combinations of hyperparameters individually.

ReForeSt is publicly available at GitHub[1] to permit easy adoption of the library.

The rest of the paper is organized as follows. The "Related Work" section lists and compares the most important related work for our proposal. In the "Preliminaries" section, we report the state-of-the-art concepts and results that we exploited in the "ReForeSt" section to present our proposal. In the "Experimental Evaluation" section, we report results on a wide range of large datasets, showing the effectiveness of our proposal. The "Conclusions" section concludes the paper.

Related Work

Planet [48] was one of the first RF implementations for distributed environments. It was developed by a team at Google based on their in-house MapReduce [16] disk-based implementation. According to MapReduce, programmers need to provide specific code segments for two functions, Map and Reduce. The Map function is applied to the input and emits a list of intermediate key-value pairs, while the Reduce function aggregates the values according to the keys. In Planet, instead of serially building each node of each tree in the RF, they build a queue of nodes to be computed and then process a subset of nodes in each iteration. Each subset of nodes becomes a MapReduce task and all the machines contribute to collecting the information

[1] https://github.com/alessandrolulli/reforest

required to detect the best cut. A large part of this work is devoted to reducing the cost of setting up each MapReduce task, which are costly operations since data must be loaded and saved on the disk each time.

MLlib [38] can be considered the de facto standard for the use of the RF learning algorithm in a distributed environment. It has been developed in Apache Spark [69], a framework that addresses the limitations of MapReduce disk-based implementations by enabling in-memory data processing. MLlib is inspired by Planet in terms of the way nodes are processed and how information is collected. The initial data is divided into several partitions, which generally are larger than the number of machines. Each machine maintains a subset of the partitions. Therefore, to construct the forest, the necessary information must be collected inside a partition. Then, the latter is aggregated locally in the machine, and, finally, all the information kept in each machine is aggregated to complete the computation. The main downside is that, for each node, an amount of memory proportional to the number of partitions handled by the machine is allocated.

ReForeSt [36] improves MLlib by avoiding the use of multiple data structures as requested by the partitioning-based distributed computation: each machine maintains only one data structure to collect information from the data stored in it. In contrast to Planet and MLlib, we process the nodes in a breadth-first manner instead of using a queue; this allows ReForeSt to grow all the trees in parallel and reduce the number of data scans.

Other distributed RF implementations exist and exhibit some drawbacks in comparison to ReForeSt. Chung [15] presents an optimization of MLlib, called Sequoia Forest, that switches from distributed to local computation when the size of the data relative to a subtree is below a threshold, but no source code is available. In ReForeSt, we introduce a similar optimization as that in Sequoia Forest, but the threshold is automatically tuned based on the problem under examination and the size of the dataset. Moreover, unlike Sequoia Forest, we cluster the elements of the dataset to reduce element copies among the machines and the memory consumption of each machine during the local computations.

In [21] and [59], it is proposed that we build one RF in each machine based on a subsample of data that is stored on the machine. In addition, in [21], several optimizations and simplifications are proposed that can be adopted when working with big data. Those that are feasible in a distributed scenario have also been considered in MLlib, with which we perform an extensive experimental comparison. On the other hand, in [59], it is also proposed to keep just the trees that show the best accuracy. In both methods, the limitation in comparison to ReForeSt is that the RF is not built on the whole set of data and, moreover, in [59], many trees are discarded, which results in a waste of computation time; in addition, the source code is not publicly available.

Chen et al. [14] propose vertically partitioning the data. The training dataset is split into several feature subsets, and each subset is allocated to a different distributed data structure. Unfortunately, this approach requires a large shuffling phase at the beginning, and the comparisons are performed against a very old implementation of MLlib; the source code is not available.

Note that none of the abovementioned libraries implement an efficient MS strategy, even if the resulting RF model generalization performance can depend strongly on the RF hyperparameter settings [46].

Preliminaries

Let us consider the multiclass classification problem [55] in which we have an input space $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_{n_f}$, consisting of n_f features, and an output space $\mathcal{Y}$. $\mathcal{X}_i$ can be a categorical feature space (the values of the features belong to a finite unsorted set) or a numerical-valued feature space (the values of the features belong to a possibly infinite sorted set). The output space, instead, is an n_c-valued space. The goal is to estimate the unknown rule that maps an item $X \in \mathcal{X}$ to an item $Y \in \mathcal{Y}$. Note that, in general, the rule can be non-deterministic [55] and some values of X may be missing [17]. In this case, if the missing value is in a categorical feature, an additional category for missing values is introduced for that feature. If, instead, the missing value is associated with a numerical feature, as suggested in [17], the missing value is replaced with the mean value of that feature and an additional logical feature is introduced to indicate if the value of that feature is missing or not for a particular sample. A set of labeled samples $\mathcal{D} = \{(X_1, Y_1), \cdots, (X_d, Y_d)\}$ is available for learning. A learning algorithm $\mathscr{A}_h$, characterized by its hyperparameters h, maps $\mathcal{D}$ to a function $f = \mathscr{A}_h(\mathcal{D})$. The error of f in approximating the unknown mapping rule is measured with reference to a loss function ℓ. Since we are dealing with classification problems, we choose the loss function that counts the number of misclassified samples; we have $\ell(f(X), Y) = [f(X) \neq Y]$, in which the Iverson bracket notation is exploited. The expected error of f is called the generalization error [55] and is defined as

$$L(f) = \mathbb{E}\{\ell(f(X), Y)\}. \tag{1}$$

Since the distribution of the samples is unknown, $L(f)$ cannot be computed, but we can compute its empirical estimator, or the empirical error, which is defined as

$$\widehat{L}(f, \mathcal{T}) = \frac{1}{|\mathcal{T}|} \sum_{(X,Y)\in\mathcal{T}} \ell(f(X), Y), \tag{2}$$

where $\mathcal{T} = \{(X_1, Y_1), \cdots, (X_t, Y_t)\}$ must be a different set from $\mathcal{D}$, which has been used to build f to ensure that the estimator of the model quality is unbiased [2]. To help readers, the nomenclature used in this paper is listed in Table 1.

Random Forests

RFs are a state-of-the-art powerful learning algorithm, first developed in [11]. Before describing RFs in detail, we have to recall the definition and construction of a binary decision tree (DT) [51]. A binary DT is a flowchart-like structure in which each internal node represents a test of a feature, each branch represents the outcome of the test, and each leaf node represents a class label. A path from the root to a leaf represents a classification rule. A graphical representation of a DT is reported in Fig. 1. A DT is built with a recursive schema until it reaches its desired depth n_d. Each node of the DT, starting from the root node, is built by choosing the attribute and the cut that most effectively splits the set of samples into two subsets based on the information gain.

Table 1 Nomenclature for the"Preliminaries" section

Symbol	Description
$\mathcal{X}$	Input space
$\mathcal{Y}$	Output space
$\mathcal{X}_i$	ith component of the feature space
n_f	Number of features
X	Item in the input space
Y	Item in the output space
n_c	Number of classes
$\mathcal{D}$	Dataset of labeled samples
d	Cardinality of $\mathcal{D}$
$\mathcal{V}$	Validation set
$\mathcal{T}$	Test set
$\mathscr{A}_h$	Learning Algorithm
h	Hyperparameters of $\mathscr{A}_h$
f	Function learned by $\mathscr{A}_h$
ℓ	Loss function
L	Generalization error
$\widehat{L}$	Empirical error
n_t	Number of trees in an RF
n_b	Number of bootstrap samples in an RF
n_v	Number of features to subsample in an RF
n_d	Maximum depth of a DT or of each tree in an RF

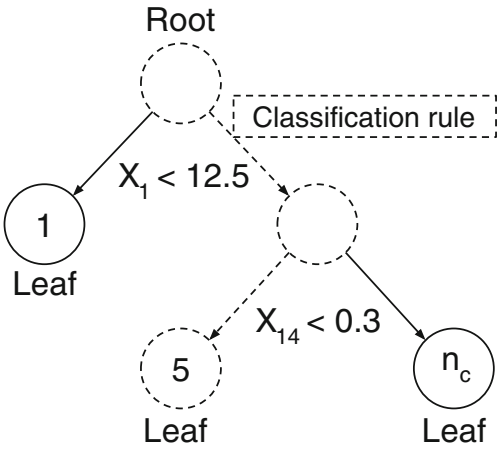


Fig. 1 DT example

More formally, for numerical features, the node test of a feature j (briefly called cut or split) can be defined as:

$$X_j < c. \tag{3}$$

Based on this cut, it is possible to split the original data $\mathcal{D}$ into two subsets $\mathcal{D}^{\text{left}}$ and $\mathcal{D}^{\text{right}}$ such that

$$\mathcal{D}^{\text{left}} = \{(X, Y) \in \mathcal{D} | X_j < c\}, \quad \mathcal{D}^{\text{right}} = \mathcal{D} \setminus \mathcal{D}^{\text{left}}. \tag{4}$$

Then, we can define the information gain of Eq. 3, namely the reduction in entropy (or confusion) induced by the cut:

$$G(j, c) = H(\mathcal{D}) - \frac{|\mathcal{D}^{\text{left}}|}{|\mathcal{D}|} H(\mathcal{D}^{\text{left}}) - \frac{|\mathcal{D}^{\text{right}}|}{|\mathcal{D}|} H(\mathcal{D}^{\text{right}}), \tag{5}$$

Here, H is the entropy of the set of samples:

$$H(\mathcal{D}) = \sum_{i\in\{\text{Distinct values of } Y\in\mathcal{D}\}} \frac{|\mathcal{D}^i|}{|\mathcal{D}|} \log_{n_c}\left(\frac{|\mathcal{D}^i|}{|\mathcal{D}|}\right) \tag{6}$$

$$\mathcal{D}^i = \{(X, Y) \in \mathcal{D} | Y = i\}. \tag{7}$$

Consequently, the best cut is defined as

$$(j^*, c^*) = \arg \max_{j\in\{1,\cdots,n_f\}, c\in\mathcal{C}_j} G(j, c) \tag{8}$$

where $\mathcal{C}_j$ are the possible cuts for the feature X_j. These cuts can be found, for numerical features, by sorting all the values of the feature X_j and by using as cuts all the mean values between two consecutive sorted values ($n - 1$ cuts). The extension to categorical features can be found in [51] and is not reported since it is out of the scope of this presentation. Pseudocode for the DT learning and forward phases can be found in Algorithm 1 by setting $n_v = n_f$. An

example of the application of a DT to the artificial datasetinf Fig. 2a, defined in Section 2.3 of [23], is reported in Fig. 2d.

Algorithm 1 DT.

```
/* Learning phase                                                    */
Input: D, n_v, and n_d
Output: A tree T
function T = DT_learn(D, n_v, n_d);
if n_d = 0 ∨ |D| = 1 then
    T.l = mode({Y ∈ D}) ;
else
    Split D in D^left and D^right based on the best cut (j*, c*)
    (see Eq. 8) chosen over n_v ≤ n_f features randomly
    sampled without repetition from all n_f features ;
    T.cut = (j*, c*);
    T.T^left = DT_learn(D^left, n_v, n_d − 1);
    T.T^right = DT_learn(D^right, n_v, n_d − 1);

/* Forward phase                                                     */
Input: X and a tree T
Output: The class Y
while true do
    if exists(T.l) then
        Y = T.l;
    else
        Based on T.cut: T = T.T^left or T.T^right;
```

Given the definition of DT, we can now briefly describe the learning phase of each of the n_t trees composing the RF. From $\mathcal{D}$, n_b samples are taken with replacement, and $\mathcal{D}'$ is built. A tree is constructed with $\mathcal{D}'$, but the best split is chosen among a subset of n_v predictors over the possible n_f predictors randomly chosen at each node. The tree is grown until its depth reaches the maximum value of n_d or until all the samples in $\mathcal{D}'$ are correctly classified. During the classification phase of a previously unseen X, each tree classifies X into a class Y; the final classification is the mode of all the answers of each tree in the RF. If $n_b = n$, $n_v = \sqrt{n_f}$, and $n_d = \infty$ we obtain the original RF formulation [11], where n_t is usually chosen as a tradeoff between accuracy and efficiency [11, 24, 46]. Pseudocode for the RF learning and forward phases can be found in Algorithm 2.

Algorithm 2 RF (see functions in Algorithm 1).

```
/* Learning phase                                                    */
Input: D, n_t, n_b, n_v, and n_d
Output: A set of trees {t_1, ··· , t_{n_t}}
for i = 1 to n_t do
    D' sample with replacement n_b sample from D;
    t_i = DT_learn(D', n_v, n_d);

/* Forward phase                                                     */
Input: X, n_t, and a set of trees {t_1, ··· , t_{n_t}}
Output: Y
for i = 1 to n_t do
    Y_i = t_i(X);
Y = mode({Y_1, ··· , Y_{n_t}}) ;
```

An example of the application of the RF with $n_v = 1$, $n_t \in \{1, 100\}$, for the same artificial dataset exploited for DT, is reported in Fig. 2a and b.

Random Rotations

RFs have recently been improved in [9] with RREs. RREs propose to randomly rotate the numerical feature space before learning each tree of the forest instead of performing the bootstrapping and subset feature selection of RF. The idea is to overcome the weakness of decision trees whose decision boundaries are axis-aligned while the real boundaries may not be.

The rotation is implemented with a rotation matrix Θ defined in [9], which, instead of rotating each angle in spherical coordinates, uniformly samples over all feasible rotations. Note that, since rotations can be sensitive to scale in general and to outliers in particular, the RF developed in [9] requires scaling the numerical feature space. As suggested by the results in [9], each feature in the range [0, 1] should be scaled. The pseudocode of the RRE learning and forward phases can be found in Algorithm 3.

Algorithm 3 RRE (see Algorithm 1).

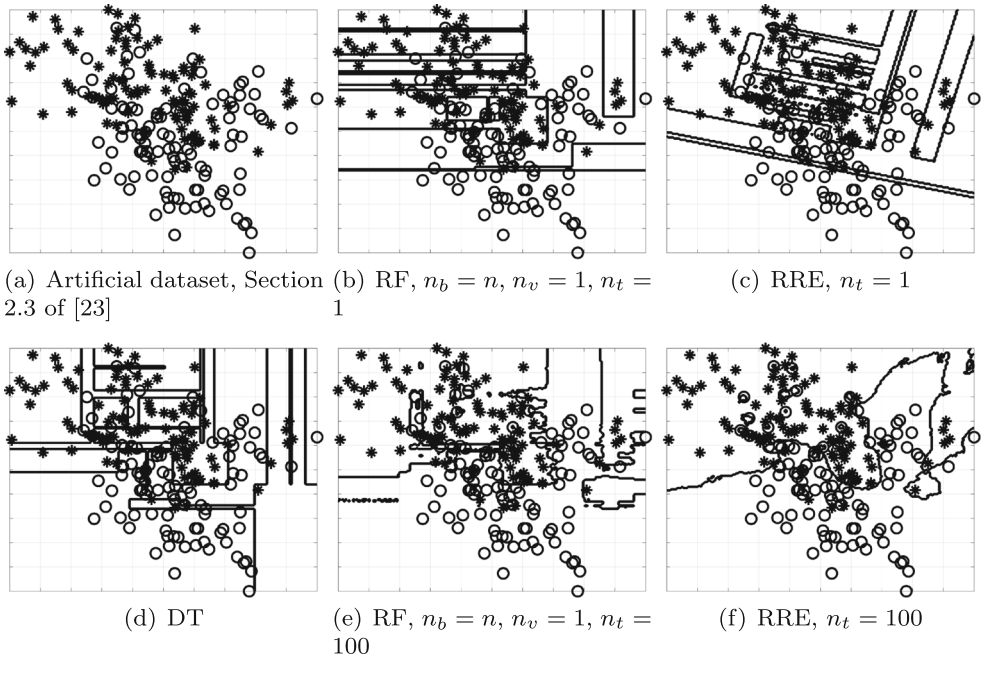
```
/* Learning phase                                                    */
Input: D, n_t, and n_d
Output: A set of trees {t_1, ··· , t_{n_t}}
for i = 1 to n_t do
    t_i.Θ = random rotation matrix defined in [9];
    D' = rotate the numerical feature space D based on t_i.Θ;
    t_i.T = DT_learn(D', n_f, n_d);

/* Forward phase                                                     */
Input: X, n_t
Output: Y
for i = 1 to n_t do
    X' = rotate X based on t_i.Θ;
    Y_i = t_i.T(X');
Y = mode({Y_1, ··· , Y_{n_t}}) ;
```

An example of the application of RREs with $n_t \in \{1, 100\}$, for the same artificial dataset exploited for DT, is reported in Fig. 2b and e. Note that the RREs ensure a smoother boundary with better generalization performance.

Model Selection

To tune the different hyperparameters of the RF in a data-dependent manner, many alternatives exist [2], from the classic resampling methods, such as the holdout, the cross validation, and the bootstrap [3, 29], to more theoretically grounded alternatives, such as Redemacher complexity or algorithmic stability-based techniques [40]. Since we are dealing with big data problems, the only computationally feasible alternative is to use the holdout (HO) method. HO



(a) Artificial dataset, Section 2.3 of [23]
(b) RF, $n_b = n$, $n_v = 1$, $n_t = 1$
(c) RRE, $n_t = 1$
(d) DT
(e) RF, $n_b = n$, $n_v = 1$, $n_t = 100$
(f) RRE, $n_t = 100$

Fig. 2 Artificial dataset to better understand the differences among DT, RF, and RRE

relies on the following idea: from the original dataset $\mathcal{D}$, a relatively small subset of the data is kept apart and exploited as a validation set $\mathcal{V}$. More formally,

$$\mathcal{V} \subset \mathcal{D}, \quad \mathcal{D} = \mathcal{D} \setminus \mathcal{V}. \tag{9}$$

Then, to select the best configuration of hyperparameters h in the set of possibilities, $\mathcal{H} = \{h_1, h_2, \cdots\}$ for the RF or, in other words, to perform the MS phase, the following procedure needs to be applied:

$$h^* : \arg\min_{h \in \mathcal{H}} \widehat{L}(\mathscr{A}_h(\mathcal{D}), \mathcal{V}). \tag{10}$$

Since the data in $\mathcal{D}$ are independent of the data in $\mathcal{V}$, the idea is that h^* should be the set of hyperparameters that allows us to achieve a small error on that data set that is independent of the dataset exploited to learn the model. The procedure is depicted in Fig. 3.

Apache Spark

Apache Spark [68, 69] is an efficient and scalable distributed data processing framework. Because of its flexibility, usability, and API richness, it has quickly become popular in the field of big data analytics [27, 53]. Machine learning, data mining and computational science are just a few areas where Apache Spark is employed. Apache Spark relies on resilient distributed dataset (RDD) abstraction. An RDD is a fault-tolerant immutable data structure comprising partitions distributed over several machines that can be operated in parallel. Recently, new data structures

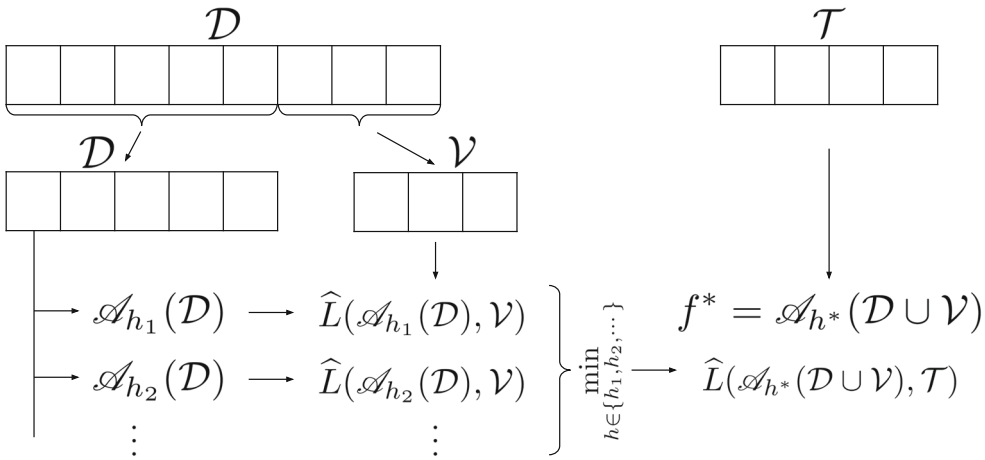


Fig. 3 MS phase. $\mathcal{D}$ is the dataset of labeled samples, $\mathcal{V}$ the validation set, and $\mathcal{T}$ the test set. $\mathcal{D} \cap \mathcal{V} = \varnothing$ and $\mathcal{V} \cap \mathcal{T} = \varnothing$

(e.g. Dataframe and Datasets) have been introduced in Apache Spark, since they have some advantages in many applications of data science; however, for our purposes, the API is still not rich and mature enough (e.g., reduction by key and many other fundamental functions are still missing).

Note that, in Apache Spark, it is possible to choose among different partitioning schemas for the RDD. This choice may have a strong impact on the computational requirements [6]. Nevertheless, for our particular problem, it is extremely difficult to think of particular partition schemas that could offer performance improvements over standard hash-based partitioning. In fact, in RF, the only effective way to parallelize the computation, as we will see later in the "ReForeSt" section, is to build the nodes of trees composing the forest in parallel. Because of the different sources of randomness, there is no way, before constructing the forest, to assume that a subset of the data will contribute to just a subset of the nodes. Therefore, the only reasonable solution is to randomly split the data and the node computation among the machines.

Note also that, in Apache Spark, distributed computation is achieved by transforming an RDD into another RDD or by creating an RDD from a data structure stored on a non-volatile, and often distributed, storage device. Non-volatile distributed storage can be implemented with different technologies (e.g., HDFS FS [62], Hbase [22], and Hive [57]). The choice of this technology may impact the reading performance. However, ReForeSt, as we will see later in the "ReForeSt" section, reads the dataset from the non-volatile distributed storage just once at the beginning of the execution, after which the non-volatile distributed storage is not accessed anymore. In fact, if the data stored on the non-volatile distributed storage device is read and transformed into an RDD that does not fit into the volatile memory (RAM), Apache Spark handles the overbooking by exploiting the non-volatile memory (HDD or SDD) of the machine rather than the non-volatile distributed storage. For this reason, the performance of ReForeSt does not depend on the non-volatile distributed storage technology in use.

Finally, note that even if the RDDs can be spilled to the disk, the tasks to be executed on them may require additional data structures to be stored in the volatile memory, which can result in overbooking and crash the program since the Java Virtual Machine (JVM) has a memory limit. As we will see later in the "ReForeSt" section, ReForeSt (unlike for example, MLlib) always pre-computes the amount of memory needed to perform a task and, if needed, splits the task into sub-tasks to avoid overbooking the volatile memory.

Algorithm 4 ReForeSt (see functions in Algorithms 5 and 6).

```
   /* Learning phase                                                              */
   Input: D, n_t, n_b, n_w, n_v and n_d;        /* D is an RDD, each machine has a subset of samples */
   Output: A set of trees {t_1, ..., t_{n_t}} and C
1  (D^{n_w}, C) = Rt_{data-preparation}(D, n_b, n_w) ;   /* D^{n_w} is an RDD, each machine has a subset of samples */
2  {t_1, ..., t_{n_t}} = Rt_{tree-generation}(D^{n_w}, n_t, n_w, n_c, n_v, n_d);

   /* Forward phase                                                               */
   Input: X, n_t, a set of trees {t_1, ..., t_{n_t}}, and C
   Output: Y
3  X' = binned version of X based on C ;                          /* Every machine has X' */
4  for i = 1 to n_t do in parallel          /* Every machine will compute a subset of the trees */
5  |  Y_i = t_i(X');
6  Y = mode({Y_1, ..., Y_{n_t}}) ;
```

Algorithm 5 ReForeSt *data preparation* phase.

```
   Input: D, n_b, n_w;                    /* D is an RDD and each machine has a subset of samples */
   Output: D^{n_w} and C
1  function (D^{n_w}, C) = Rt_{data-preparation}(D, n_b, n_w);
2  D' = sample(D);                        /* D' is an RDD and each machine has a subset of the features */
3  C = find-bins(n_w, D') ;  /* each machine will process the subset of features stored in its RDD's
   partitions */
4  D^{n_w} = convert-to-working-data(D, C, n_b) ; /* each machine will process the subset of D stored in its
   RDD's partitions */
```

ReForeSt

ReForeSt is a distributed and scalable RF implementation that includes several distinctive features that distinguish it from state-of-the-art alternatives. ReForeSt optimizes memory consumption and computational time and automatically arranges the computation according to the available memory. ReForeSt can distribute the data in two different manners and automatically selects the best one based on the problem under consideration. ReForeSt can deal with an arbitrarily large number of samples, an arbitrarily large number of features, and an arbitrarily large number of categories for categorical features. ReForeSt implements both the original RF formulation and the novel RRE. Finally, ReForeSt implements a parallel and incremental MS procedure for the whole set of hyperparameters that lets the learning process stop for the sets of hyperparameters showing low accuracy. This approach permits the computation to be completed in less time than does checking all the possible combinations of hyperparameters individually.

To improve the readability of the paper, analogously to what we have done in the "Preliminaries" section, we will describe the details of our implementation with the help of illustrations and pseudocode. Moreover, the additional nomenclature that we will exploit in this section, with respect to the "Preliminaries" section, is listed in Table 2. An initial high-level description of ReForeSt is presented in Algorithm 4. The ReForeSt forward phase is equivalent to that of the classic RF algorithm, except that the prediction of all the trees in the forest is performed in parallel and we need to discretize the points to classify them in accordance with the discretization made during the training phase (see the "Data Preparation" section). The ReForeSt learning phase, instead, consists of two phases: the *data preparation* phase and the *tree generation* phase. The first phase, presented in the "Data Preparation" section, oversees the loading of the raw data and produces the *working data*, a statically allocated collection of elements representing the original samples. The second phase, presented in the "Tree Generation" section, grows all the trees of the forest iteratively and in parallel based on the *working data*. This second phase can be performed following two different approaches to distribute the data on the available machines: (i) a *fully distributed* approach in which the dataset is partitioned among the machines ("Fully Distributed Computation" section), and (ii) a *local computation* approach in which each machine collects the subset of the dataset needed to complete the computation of a subtree locally ("Local Computation" section). Two approaches are needed since they are optimal in different situations. When we have to learn a node based on a set of data that does not fit into the volatile memory, we have to use the *fully distributed* approach. When, instead, we have to learn a node and all its sub-nodes or leaves, based on a reasonably small set of data (namely, a set of data that fits into the volatile memory of a single machine), *local computation* is a more efficient choice. ReForeSt, beyond implementing both approaches, also automatically detects when it is more efficient to switch from one to the other approach by computing how much memory is required to perform the *local computation* approach. The extension of ReForeSt to RRE is presented in the "ReForeSt RRE" section, while the implemented MS procedure is described in the "Model Selection in ReForeSt" section. The "Computational Complexity" section describes the computational complexity of ReForeSt with respect to the classic RF algorithms.

Table 2 Nomenclature for the"ReForeSt" section, see also Table 1

Symbol	Description
$\mathcal{D}^{n_w}$	*working data*
n_w	Number of bins for the *working data*
$\mathcal{C}$	Bins for the *binning* phase
B	The vector containing the *bagging* of an element
N_m	Number of machines
N_c^i	Number of cores for the ith machine
Γ_j	ReForeSt *Collecting matrix* for the ith machine
Γ	ReForeSt *Collecting RDD*

Data Preparation

The *data preparation* phase (see Algorithm 5) oversees the building of the *working data* $\mathcal{D}^{n_w}$, a collection of elements stored in an RDD that allows us to represent the original dataset $\mathcal{D}$ and the result of the bagging procedure in a more compact way. The *working data* will be exploited later during the *tree generation* phase (see Algorithm 4). An example of the *data preparation* phase is depicted in Fig. 4. To build the *working data* $\mathcal{D}^{n_w}$ from the original dataset $\mathcal{D}$, we need to perform *binning* and *bagging*.

During the *binning* phase, the domain of each feature is discretized into n_w configurable bins. For each feature, we search for n_w bins (see function *find-bins* A5L3, namely Line 3 of Algorithm 5). To find the bins, we make use of a subsample of the original dataset $\mathcal{D}' \subseteq \mathcal{D}$ in which $|\mathcal{D}'| \approx 10^4$ (see the function *sample* in A5L2 and the column "Subsample of Dataset" in Fig. 4). We used $\mathcal{D}'$ instead of $\mathcal{D}$ to speed up the computation. All these operations are parallelized over the N_m available machines and exploit all the cores of each machine. In particular, each machine processes a subset of the n_f features. We provide two strategies to find the bins: *equi-depth* and *random*. In the *equi-depth* strategy, each bin is constructed to have approximately the same number of samples $|\mathcal{D}'|/_{n_w}$. The

random strategy instead searches for $X_i^{\min}$ and $X_i^{\max}$, which are, respectively, the minimum and maximum values of the ith feature, and randomly divides the space $[X_i^{\min}, X_i^{\max}]$ into n_w bins. The *random* strategy is alternative to the *equi-width* strategy. We chose a random strategy because, in the *equi-width* strategy, outliers may cause most of the data to concentrate in a few bins [37, 52], thus reducing the accuracy of the algorithm, whereas introducing randomness can alleviate this downside. Moreover, since using the *random* strategy is not a computationally intensive task, it is suitable for high-dimensional datasets. The *binning* phase allows us to reduce the memory requirements in comparison to directly storing $\mathcal{D}$ in an RDD because each value can be stored in its smallest integer representation instead of a larger integer representation or floating point number.

After the *binning* phase, we have to perform *bagging* (in Fig. 4; the input for bagging is the column "Binned Data" and the output is the column "Working Data"). Bagging is the process of resampling with replacement from $\mathcal{D}$ another dataset of cardinality n_b for each tree in the forest that we have to build. This procedure would require storing a different dataset of size n_b for each tree, for a total of $n_b n_t$ elements of size n_f, since each element has n_f features. Since this procedure consumes a great deal of memory, we adopt a different approach. We keep track of the number of times that a sample has been extracted from $\mathcal{D}$ during the bagging phase of the ith tree in the element B_i of the vector B. This approach permits us to store the original dataset $\mathcal{D}$ once (and partitioned on all the machines) and one vector B for each sample, resulting in storing d elements of size $n_f + n_t$; this is much fewer than in the previous approach.

The result of the *binning* and *bagging* phases is the transformation of an element $(X, Y) \in \mathcal{D}$ into an element $(X^{n_w}, Y, B) \in \mathcal{D}$, where the vector X^{n_w} is the discretized version of X, Y remains unchanged, and the vector $B \in \mathbb{N}^{n_t}$ contains in B_i the number of times the element X has been sampled in the *bagging* process of the ith tree. These operations are done in parallel in the *convert-to-working-data* function (see A5L4). In particular, each machine processes a subset of the samples in $\mathcal{D}$ and the results are stored in a distributed manner in an RDD (see the last two columns of Fig. 4).

Tree Generation

The goal of tree generation is to grow the forest while choosing the best cut for each node among all the possible cuts. ReForeSt can perform this action in two different manners: (i) a *fully distributed* computation when the *working dataset* is too large to be stored in only one machine and (ii) a *local computation* when the *working data* relative to a single node can be stored on one machine, and each machine finishes the computation of a different subtree locally. In the *fully distributed* computation, the *working data* is partitioned over all the available machines. Each machine collects a contribution to computing the information gain (defined in the "Random Forests" section) for its partition of data in parallel. Subsequently, the data relative to the same node are aggregated, and the best cut is identified. On the other hand, in the *local computation*, each machine maintains a subset of the *working data* that contains enough information to complete the computation of a subtree on a single machine.

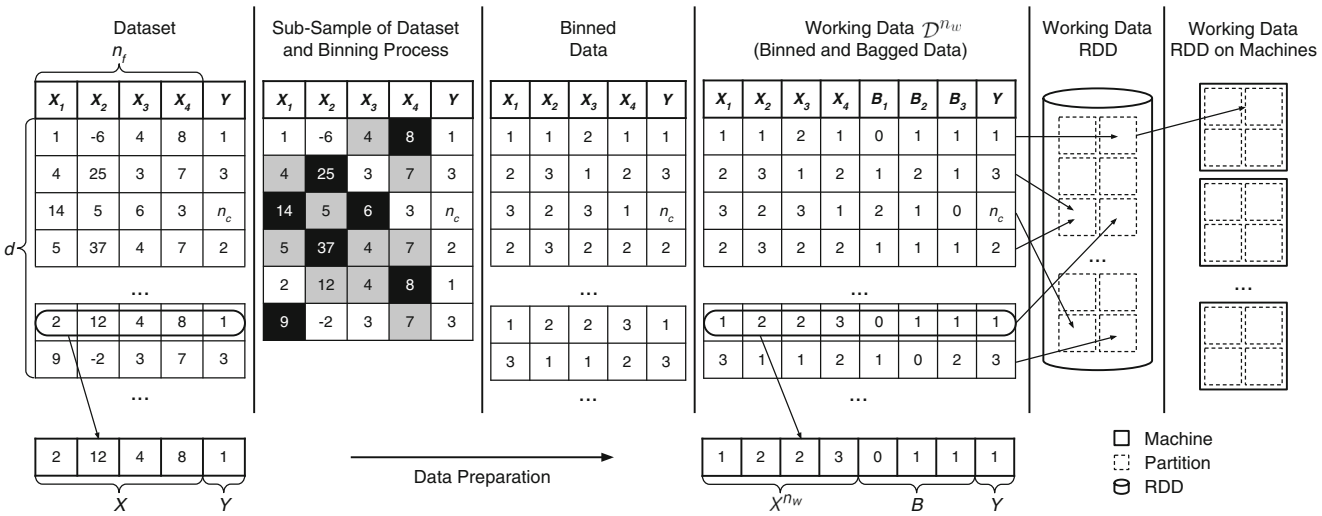


Fig. 4 Example of *data preparation* phase. We let $n_w = 3$ and suppose we have the following bins for each feature: $\mathcal{X}_1 : ([1, 2], [3, 5], [6, 14])$, $\mathcal{X}_2 : ([-6, -2], [-1, 12], [13, 37])$, $\mathcal{X}_3 : ([3, 3], [4, 4], [5, 6])$, $\mathcal{X}_4 : ([3, 3], [4, 7], [8, 8])$

Fully Distributed Computation

Algorithm 6 ReForeSt *tree generation* phase via *fully distributed* computation.

```
/* Tree generation function                                                    */
Input: D^{n_w}, n_t, n_w, n_c, n_v, and n_d ;   /* D is an RDD and each machine has a subset of samples */
Output: A set of trees {t_1, ..., t_{n_t}}
1 function {t_1, ..., t_{n_t}} = Rt_{tree-generation}(D^{n_w}, n_t, n_w, n_c, n_v, n_d);
2 {t_1, ..., t_{n_t}} = initialize-empty-trees(n_t);
3 for i = 0, ..., (n_d − 1) do
4 |   Rt_{tree-generation-iteration}({t_1, ..., t_{n_t}}, D_n^{n_w}, i, n_t, n_w, n_c, n_v);

/* Rt_{tree-generation-iteration} function                                      */
Input: A set of trees {t_1, ..., t_{n_t}}, D^{n_w}, i, n_t, n_w, n_c, n_v ;   /* D is an RDD and each machine has a subset of
       samples */
Output: A set of trees {t_1, ..., t_{n_t}}
5 function {t_1, ..., t_{n_t}} = Rt_{tree-generation-iteration}({t_1, ..., t_{n_t}}, D^{n_w}, i, n_t, n_w, n_c, n_v);
// Local Information Collection: each machine j is running in parallel for the following
   steps
6 Γ_j = instantiate a matrix N^{(2^i n_t)×(n_w n_c n_v)} ;   /* Γ_j is the collecting matrix stored in the volatile
memory of the jth machine */
7 for (X^{n_w}, Y, B) ∈ (D^{n_w})^j do   /* (D^{n_w})^j is the subset of D^{n_w} that resides on the jth machine */
8 |   for t = 1 to n_t do
9 |   |   node = get-node(t_t, X^{n_w});
10 |  |   if is-not-leaf(node) then
11 |  |   |   F = set of n_v indices of subset features randomly sampled without repetition from all the n_f features;
12 |  |   |   for f ∈ F do
13 |  |   |   |   p = get-column-idx(t, node, f, X^{n_w}, Y);   r = get-row-idx(node);   Γ_{j,r,p} += B_t ;
// Distributed Information Aggregation
14 Barrier Synchronization ;
15 Γ = merge-by-node(Γ_1, ..., Γ_{N_m}) ;   //* Γ is a matrix stored in an RDD, each machine has a subset
of its rows and
// Trees Update   aggregates in these rows the corresponding rows of all the Γ_j with
   j ∈ {1, cdots, N_m}  */
16 for r = 1 to 2^i n_t do in parallel   /* Each machine handles a subset of the 2^i n_t nodes */
17 |   node = get-node-from-row-idxΓ(r);   bestCut = find-best-cut(Γ_{r,{1,...,n_w n_c n_v}});   grow-forest({t_1, ..., t_{n_t}}, node, bestCut)
   |   ;
```

The *tree generation* phase via *fully distributed* computation builds the trees of the forest iteratively in a breadth-first manner (see A6L1). Our main idea is to share a data structure, stored in the volatile memory and called the *collecting matrix*, among the partitions of the same machine to perform efficient learning of the trees. The *collecting matrix* is exploited to collect the necessary information to select the best cut based on the information gain. Γ_j is the matrix that resides on the jth machine. This choice is one of the most fundamental differences between our approach and MLlib, in which this information is collected in a similar data structure but each partition replicates that structure, resulting in considerable memory requirements and computational inefficiencies. Subsequently, we distributively aggregate all the Γ_j in the collecting RDD Γ and finally detect the best cut for each processed node.

In each iteration i, all the nodes at the ith level of each tree are processed. In detail, each iteration $i \in \{0, \cdots, n_d - 1\}$ is divided into three steps: (i) *local information collection*, (ii) *distributed information aggregation*, and (iii) *tree updating*. We make use of the Figs. 5 and 6 to describe, respectively, the local information collection and distributed information aggregation.

Local Information Collection Figure 5 describes this step. At a given iteration i, each machine works autonomously, and the idea is to collect information to detect the best cut for each node at level i. This information will subsequently be distributively aggregated. The stored information permits us to count, for each possible cut of a specific node, the number of elements that will fall in the left or right child if the cut is selected. Finally, this information is exploited to detect the best cut for each node.

Delving into the details, the jth machine collects information from the *collecting matrix* Γ_j, which is instantiated at the beginning of the iteration (see A6L6 and is represented by the cube in Fig. 5). The purpose of $\Gamma_j \in \mathbb{N}^{2^i \times n_t \times n_w \times n_c \times n_v}$ is to store the information on each of the samples from the portion of the *working data* stored on the jth machine to compute the information gain of all the possible cuts. Each cut is made relative to one of the n_w discretized values, one feature of the subset of features of cardinality n_v, and one node at the ith level of each n_t tree (of which there are $n_t 2^i$). For each cut, we need to collect information for each of the n_c classes. Γ_j is flattened from a five-dimensional matrix to a two-dimensional matrix $\Gamma_j \in \mathbb{N}^{2^i n_t \times n_w n_c n_v}$ for performance reasons. Each Γ_j

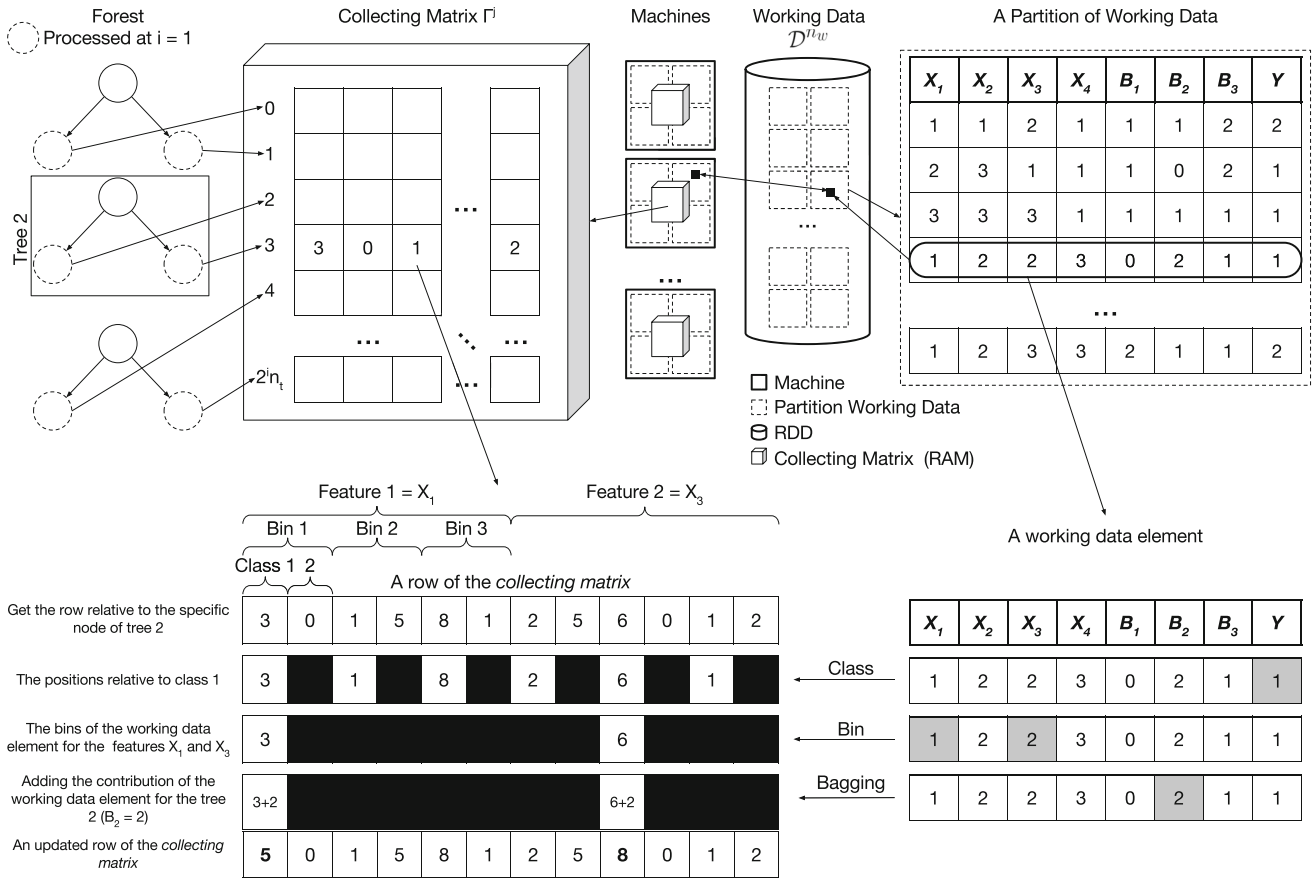


Fig. 5 Example of *local information collection* step. We consider $n_c = 2$, $n_w = 3$, and $n_v = 2$

contains one row for each of the processed nodes at level i, represented by the forest in the left of Fig. 5. The details of one row are depicted at the bottom of Fig. 5, where the number of columns depends on n_c, n_w, and n_v.

The *local information collection* phase is completely parallel since no communication among the machines is required; it proceeds as follows. Given a working data element $(X^{n_w}, Y, B) \in (\mathcal{D}^{n_w})^j$, where $(\mathcal{D}^{n_w})^j$ is the subset of $\mathcal{D}^{n_w}$ that resides on the jth machine, and a tree $t \in \{1, \cdots, n_t\}$, we have to find the node in tree t to which (X^{n_w}, Y, B) contributes (see A6L9) and the corresponding row in Γ_j. Next, for each feature $f \in \mathcal{F}$ ($\mathcal{F}$ is a set of n_v indices randomly sampled without replacement from $\{1, \cdots, n_f\}$), the proper element of the row is found (see A6L13). The column of Γ_j is computed as $n_w n_c f + n_c X_f^{n_w} + Y$. Finally, B_t is added in the aforementioned position of Γ_j (see A6L13). Note that each working data element (X^{n_w}, Y, B) adds information to just one node in each of the trees t with its weight B_t (see A6L8). This is because, at the ith iteration, all the nodes at the ith depth of all the trees are processed, and then one element can be contained in only one node in each of the trees.

These operations are described in the bottom of Fig. 5, where we selected a *working data* element and show how this element contributes to updating a specific row of Γ_j. In particular, given the selected *working data* element and tree 2, we navigate the tree from the root to detect the node on level i at which element is stopping. The information about the selected node is stored in row 3 of Γ_j (the machine maintains a mapping between the nodes at level i and the indices in Γ_j). The columns to be updated are chosen by checking the class of the element and the discretized values for each of the features in the subset of features. Finally, B_2 represents the bagging for tree 2 of the element added to the selected columns.

Distributed Information Aggregation Figure 6 reports an example that describes the operations performed during the *distributed information aggregation* phase. The idea is to start from all the Γ_j and aggregate the information in a unique matrix Γ. At the end of this step, Γ contains the aggregated information computed by all the machines in the previous step and each machine maintains a subset of it.

During this phase, the *collecting matrices* Γ_j with $j \in \{1, \cdots, N_m\}$ (each machine stores one of these matrices in its volatile memory) are aggregated in the *collecting RDD* Γ. Figure 6 describes, in particular, two machines and their respective Γ_j.

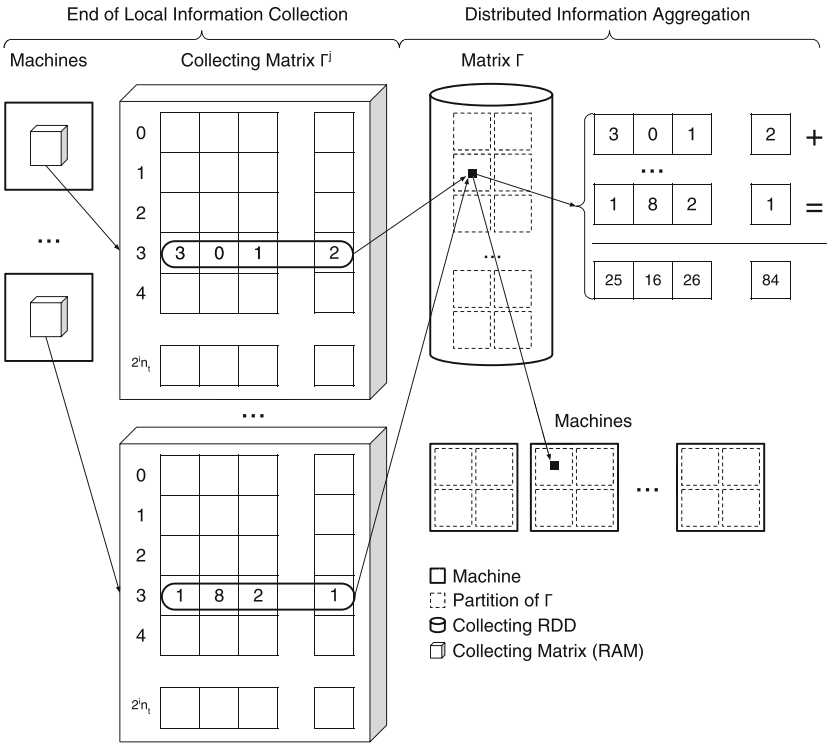


Fig. 6 Example of *distributed information aggregation* step

For each node $l \in \{1, \cdots, 2^i n_t\}$, the row relative to l is collected from each *collecting matrix* and aggregated by the machine $l \% N_m$ (see A6L15). In this way, the information of approximately $2^i n_t / N_m$ nodes is stored in each machine and the Γ_j can be destroyed. The result is the *collecting RDD* Γ is partitioned on the available machines. For instance, in Fig. 6, all the machines send row 3 of their Γ_j to the same machine, which can perform the aggregation (the sum in the right of the figure) and store the aggregated value in the *collecting RDD*.

Tree Updating In this last step, each machine, having complete knowledge of the nodes stored in it, searches for the best cut of each node. The best cut is chosen in such a way as to maximize the information gain. Finally, the n_t trees in the forest are updated based on these cuts (see A6L17).

Local Computation

The *local computation* is an alternative to the *fully distributed* one. The idea behind this approach is to complete the computation serially when the size of the *working data* relative to a subtree fits the memory of one machine. In this way, ReForeSt can complete the construction of a subtree without requiring any communication among machines.

To decide whether *local computation* can be activated, ReForeSt has to determine whether the volatile memory of the machine is large enough to store the subset of *working data* needed to complete the construction of a particular subtree using the standard serial algorithm reported in Algorithm 1. If the memory is not enough, it performs another iteration of the *fully distributed* computation reported in Algorithm 6. In contrast, if the memory is enough, the computation is switched to *local computation* and:

- each machine takes and removes one subtree from the sets of subtrees that can be serially completed;
- the subset of *working data* needed for completing the particular subtree is collected by the machine, which oversees its completion;
- each machine proceeds independently in learning the subtree with the Algorithm 1;
- this procedure continues until all the subtrees that can be serially completed have been constructed.

Figure 7 shows how ReForeSt decides when it is possible to switch from the *fully distributed* approach to the *local computation* approach.

ReForeSt RRE

ReForeSt also implements RRE targeting, as in ReForeSt RF, in the case of low memory and few computational requirements. Since each rotation generates different *working data*, we iteratively process the rotations one at the time. Unlike in the RRE described in the "Random Rotations" section, we allow the user to reuse the same rotation in the construction of multiple trees to reduce the computational

requirements. n_r rotations (the default value is n_t, as in the standard RRE) and approximately n_t/n_r trees are built using the same rotation (see A7L3).

The ReForeSt RRE learning phase proceeds as follows:

1. first, we generate in parallel the rotation matrices Θ_i with $i \in \{1, \cdots, n_r\}$, following the definition provided in [9], and distribute them on each machine;
2. we scale the numerical feature space as described in the "Random Rotations" section;
3. we proceed iteratively: for each rotation matrix, we rotate the numerical feature space and perform *data preparation* (see the "Data Preparation" and "Tree Generation" sections) phases, creating a subset of the trees of the forest.

For the ReForeSt RRE forward phase, with respect to the original ReForeSt RF forward phase (see Algorithm 4) where we had to perform the discretization phase just once, we must rotate and discretize the point to classify it in a different way based on the rotation associated with the particular tree.

Pseudocode for the learning and forward phases of ReForeSt RRE is presented in Algorithm 7.

Algorithm 7 ReForeSt RRE (see functions in Algorithms 5 and 6).

```
/* Learning Phase                                                              */
Input: D, n_t, n_b, n_w, n_v, n_d and n_r;   /* D is an RDD and each machine has a subset of samples */
Output: A set of trees {t_1, ..., t_{n_t}}, {C_1, ..., C_{n_r}}, and {Θ_1 ... Θ_{n_r}}
Compute {Θ_1 ... Θ_{n_r}} (see [9]) ;   /* All machines store the matrices and each machine computes a
subset of them */
S = scale(D) ;                          /* S is an RDD and each machine has a subset of samples */
for i = 0 to (n_r − 1) do
    R = rotate the numerical feature space S based on Θ_i ; /* R is an RDD and each machine has a subset of
    samples */
    (D^{n_w}, C_i) = Rt_{data-preparation}(R, n_b, n_w) ;   /* D^{n_w} is an RDD and each machine has a subset of
    samples */
    {t_{(i−1)n_t/n_r+1}, ..., t_{i n_t/n_r}} = Rt_{tree-generation}(D^{n_w}, n_t, n_w, n_c, n_v, n_d);

/* Forward phase                                                               */
Input: X, n_t, n_r a set of trees {t_1, ..., t_{n_t}}, {C_1, ..., C_{n_r}}, and {Θ_1 ... Θ_{n_r}}
Output: Y
for i = 0 to (n_r − 1) do
    X' = rotated and binned version of X based on Θ_i and C_i ;   /* Each machine has X and X' */
    for i = (i − 1)n_t/n_r + 1 to i n_t/n_r do in parallel   /* Each machine computes a subset of the trees */
        Y_i = t_i(X');
Y = mode({Y_1, ..., Y_{n_t}}) ;
```

Model Selection in ReForeSt

In this section, we describe how the MS procedure has been implemented in ReForeSt RF. ReForeSt performs MS for four of the RF hyperparameters: the number of bins n_w, the number of features to subsample in each node n_v, the maximum depth n_d, and the number of trees n_t. Note that the other hyperparameters are set to their default values ($n_b = d$ and $n_r = n_t$) since they are not as influential, according to some recent work in the field [44, 46].

A user of ReForeSt must specify a set of values for each of the abovementioned parameters, including $\mathcal{P}^{n_w}$, $\mathcal{P}^{n_v}$, $\mathcal{P}^{n_d}$, and $\mathcal{P}^{n_t}$.

Instead of implementing the naive MS procedure described in the "Model Selection" section, in which an RF must be built for each combination of the hyperparameters, our ideas are

- to construct concurrently the RFs corresponding to all combinations of $(n_w, n_v) \in \mathcal{P}^{n_w} \times \mathcal{P}^{n_v}$ and to discard the combinations of hyperparameters $(n_w, n_v) \in \mathcal{P}^{n_w} \times \mathcal{P}^{n_v}$ as soon as the error of the corresponding RF on $\mathcal{V}$ is too high;
- to incrementally construct all the RFs corresponding first to all $n_d \in \mathcal{P}^{n_d}$ and then to all $n_t \in \mathcal{P}^{n_t}$, starting from their smallest values, and to stop the search early when no improvement is achieved.

Basically, we follow a greedy approach instead of searching for the best combination in $\mathcal{P}^{n_w} \times \mathcal{P}^{n_v} \times \mathcal{P}^{n_d} \times \mathcal{P}^{n_t}$.

More details on ReForeSt RF with MS are as follows.

First, analogously to the ReForeSt RF, the *data preparation* is performed (see Algorithm 5) by creating the *working data* $\mathcal{D}^{n_w}$, where the number of bins is set to the maximum value in $\mathcal{P}^{n_w}$. Note that, in ReForeSt RF with MS, we maintain only one *working dataset*, analogously to the case in which no MS is performed. This choice targets the optimization of the memory requirements and avoids storing one copy of the *working data* for each $n_w \in \mathcal{P}^{n_w}$. Also, to work with values $n_w < \max(\mathcal{P}^{n_w})$, we introduce the *shrunk* function for the processing of forests with $n_w < \max(\mathcal{P}^{n_w})$. The *shrunk* function reduces the number of bins from $\max(\mathcal{P}^{n_w})$ to a desired value n_w, evenly distributing the $\max(\mathcal{P}^{n_w})$ bins into the n_w bins every time ReForeSt accesses a binned value inside $\mathcal{D}^{\max(\mathcal{P}^{n_w})}$.

Then, we initialize an empty forest for each combination of the hyperparameters $\mathcal{P}^{n_w} \times \mathcal{P}^{n_v} \times \mathcal{P}^{n_d} \times \mathcal{P}^{n_t}$.

At this point, we fix $n_t = \min(\mathcal{P}^{n_t})$ and search for the best possible value of n_d in a greedy way. In particular, we start iterating from the smallest $n_d \in \mathcal{P}^{n_d}$ and proceed to the largest $n_d \in \mathcal{P}^{n_d}$. Fixing n_d and n_t, we concurrently grow all the RFs corresponding to all the combinations of $(n_w, n_v) \in \mathcal{P}^{n_w} \times \mathcal{P}^{n_v}$. The combination of (n_w, n_v) that performs ε_{MS} worst on $\mathcal{V}$ (where $\varepsilon_{MS} \in [0, 1]$ is some user-defined parameter) is discarded from the set of possible combinations. We stop the iterative search of n_d when increasing n_d does not bring a benefit of ε_{MS} to the error of the best RF found up to that iteration; we choose n_d, the last tested value, as the best value. The hypothesis behind this greedy approach is that the accuracy of an RF should increase and then decrease as n_d increases [11, 46].

Finally, since we have found the local best value for n_d, we repeat the same procedure for $n_t \in \mathcal{P}^{n_t}$. The hypothesis behind this additional greedy approach is that the accuracy of an RF should always increase as n_t increases [11, 46].

Pseudocode for the learning phase of ReForeSt RF with MS is presented in Algorithm 8.

Algorithm 8 ReForeSt RF with MS (see the functions in Algorithms 5 and 6).

```
/* Learning phase                                                              */
Input: 𝒟, 𝒱, 𝒫^{n_t}, ε_MS, n_b, 𝒫^{n_w}, 𝒫^{n_v} and 𝒫^{n_d}
;                                  /* 𝒟 is an RDD and each machine has a subset of samples */
Output: A set of trees {t_1, ··· , t_{n_t}}, 𝒞
(𝒟^{max(𝒫^{n_w})}, 𝒞) = Rt_{data-preparation}(𝒟, n_b, max(𝒫_i)) ;   /* 𝒟^{max(𝒫^{n_w})} is an RDD and each machine has a subset
of samples */
{F_1, ··· , F_{|𝒫^{n_w}|·|𝒫^{n_v}|·|𝒫^{n_d}|·|𝒫^{n_t}|}} = initialize-empty-forest();
currentError = ∞;
for n_d ∈ ascending-sort(𝒫_d) do
    for (n_v, n_w) ∈ 𝒫^{n_v} × 𝒫^{n_w} do in parallel
        i = get-forest-idx(min(𝒫^{n_t}), n_w, n_c, n_v, n_d);
        F_i = Rt_{tree-generation}(shrunk(𝒟^{max(𝒫^{n_w})}), min(𝒫^{n_t}), n_w, n_c, n_v, n_d);
        if error − currentError > ε_MS AND (currentError ISNOT ∞) then
            𝒫^{n_v} = 𝒫^{n_v} \ n_v;   𝒫^{n_w} = 𝒫^{n_w} \ n_w ;
    previousError = currentError;
    currentError = smallest error among the errors of the different forests in {F_1, ··· , F_{|𝒫^{n_w}|×|𝒫_{n_v}|×|𝒫^{n_d}|×|𝒫^{n_t}|}} over 𝒱;
    if |previousError − currentError| < ε_MS then
        Exit ;
for n_t ∈ ascending-sort(𝒫^{n_t} \ min(𝒫^{n_t})) do
    for (n_v, n_w) ∈ 𝒫^{n_v} × 𝒫^{n_w} do in parallel
        i = get-forest-idx(n_t, n_w, n_c, n_v, n_d);
        F_i = Rt_{tree-generation}(shrunk(𝒟^{max(𝒫^{n_w})}), n_t, n_w, n_c, n_v, n_d);
        error = errors of F_i over 𝒱;
        if error − currentError > ε_MS then
            𝒫^{n_v} = 𝒫^{n_v} \ n_v;   𝒫^{n_w} = 𝒫^{n_w} \ n_w ;
    previousError = currentError;
    (currentError, index) = smallest error and index among the errors of the different forests in {F_1, ··· , F_{|𝒫^{n_w}|×|𝒫_{n_v}|×|𝒫^{n_d}|×|𝒫^{n_t}|}}
    over 𝒱;
    if |previousError − currentError| < ε_MS then
        Exit ;
{t_1, ··· , t_{n_t}} = F_index;

/* For the forward phase, refer to that of Algorithm 4                        */
```

The extension of ReForeSt RRE to MS is straightforward since it is based on the same principles described in this section.

Computational Complexity

In this section, we discuss the complexity of learning an RF with respect to ReForeSt.

We start by discussing the memory complexity. For the classic RF (see Algorithm 2), it is necessary to store d samples with n_f features, plus a structure that contains n_t trees with a maximum depth of n_d. Consequently, the memory requirement of RF is $O(d{\cdot}n_f)$ for storing the data and $O(n_t{\cdot}2^{n_d})$ for storing the trees. In ReForeSt (see Algorithm 4), we have the same data structures, but they are sampled and spread among the N_m available machines. Moreover, ReForeSt requires the storage of the jth *collecting matrix* $\Gamma_j \in \mathbb{N}$ in the volatile memory of each machine. In addition, the *collecting RDD* Γ, which is the same size as Γ_j, is spread among the N_m available machines. Consequently, in ReForeSt, the memory requirements of storing Γ and Γ_j are $O(2^{n_d}{\cdot}n_t{\cdot}n_w{\cdot}n_c{\cdot}n_v)$.

At this point, we can discuss computational complexity. The computational complexity of classic RF (see Algorithm 2) is dominated by the sorting of each of the n_f features, resulting in a complexity of $O(n_f{\cdot}d{\cdot}log(d))$. Then, we

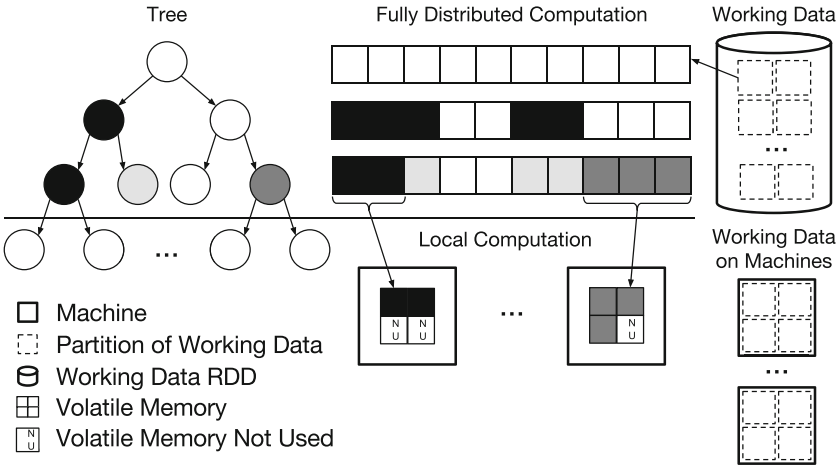


Fig. 7 Activation of the *local computation* approach

must find the best cuts for all the nodes in the tree; the computational complexity of this procedure is $O(2^{n_d} \cdot n_t \cdot d$. The forward phase requires visiting n_t trees of maximum depth n_d, resulting in a computational complexity of $O(n_t \cdot n_d)$. In ReForeSt (see Algorithm 4), we do not sort the features; rather, we discretize them, and this requires scanning the dataset, resulting in a computational complexity of $O(d \cdot n_f)$. Next, in the *fully distributed* phase for each of the n_d levels of the trees, we have to scan all the samples in the dataset and update all the trees, resulting in a complexity of $O(n_d \cdot d \cdot n_t)$. Finally, we have to select the best cut for each node, resulting in a computational complexity of $2^{n_d} \cdot n_t \cdot n_w)$.

Note that ReForeSt, with respect to the classic RF algorithm, requires a bit more memory but has lower computational requirements, especially when d and n_f are large.

The computational complexities of ReForeSt RRE are the same as those of ReForeSt RF. The computational complexity of MS, in the worst case, is equivalent to that of testing all the combinations of the hyperparameters, but, as we will see in the "Experimental Evaluation" section, our implementation allows us to reduce the actual computational time, thanks to the computational optimization described in the "Model Selection in ReForeSt" section.

Experimental Evaluation

In this section, we evaluate the performance of ReForeSt with an extensive experimental campaign conducted on a series of real-world datasets with different characteristics. We measure the ReForeSt computational performances, memory requirements, and accuracies of the selected models of ReForeSt RF, ReForeSt RRE, and ReForeSt with MS by comparing them with the state-of-the-art library MLlib[2] [38], an open source implementation of RF available within the Apache Spark framework. ReForeSt is publicly available on GitHub[3] to speed up its adoption.

To let the reader replicate the experiments, we will describe in detail the hardware, software, and datasets exploited during the experiments, along with their configurations.

We run these experiments on the Google Cloud Platform[4] (GCP) and automatically deploy clusters of virtual machines (VMs) that make use of Linux shell scripts. Each VM runs Debian 8.7 and is equipped with Hadoop 2.7.3 and Apache Spark 2.1.1. In all the configurations, we have a master node and, to evaluate scalability, different numbers of worker machines $N_m = \{4, 8, 16\}$ equipped with different numbers of cores $N_c = \{4, 8, 16\}$ (the ith machine has N_c^i cores, and all the machines have the same number of cores $N_c^i = \cdots = N_c^{N_m} = N_c$). For this purpose, we use the *n1-standard-4*, *n1-standard-8*, and *n1-standard-16* machine types from the GCP with approximately 15, 30, and 60 GB of RAM, respectively, and 500 GB of SSD disk space in the region *us-central1-a* of the GCP. For each combination of parameters, we run the experiments 30 times. The hyperparameters of the algorithms and the library settings exploited in the experiments are reported when discussing each of the reported tables and figures. The datasets exploited in this paper are reported in Table 3; for each of them, the number of elements in the dataset d, the number of features n_f, the number of classes n_c, a brief description, and a reference for retrieving it are reported. Moreover, 60% of each dataset is used as $\mathcal{D}$, 20% as $\mathcal{V}$, and the remaining 20% as $\mathcal{T}$. Datasets are stored in HDFS FS with the default block size of 64 MB. For Apache Spark, instead, we fix the number of RDD partitions to its default value, namely the number of HDFS FS blocks, in both ReForeSt and MLlib to allow a fair comparison. Note that, in this work, we do not report

[2] https://spark.apache.org/mllib

[3] https://github.com/alessandrolulli/reforest

[4] https://cloud.google.com

Table 3 Dataset exploited in the paper along with d, the number of items; n_f, the number of features; and n_c, the number of classes

Real Name	Name	Ref.	d	n_f	n_c	Size (GB)	Task
covertype	Covertype	[8]	$5.8 \cdot 10^5$	55	3	0.5	Predicting forest cover type from cartographic variables only.
mfeat-karhunen	Karhunen	[50]	10^6	64	10	0.8	Dataset of Karhunen features for handwritten numerals ("0"–"9") extracted from a collection of Dutch utility maps.
SUSY	Susy	[4]	$5 \cdot 10^6$	18	2	3	Distinguish between a signal process that produces supersymmetric particles and a background process that does not.
kdd2010 raw version bridge to algebra	Kdd	[66]	$1.9 \cdot 10^7$	$3 \cdot 10^7$	2	5	Predict student performance on mathematical problems from logs of students in a interacting with intelligent tutoring systems.
HIGGS	Higgs	[4]	$11 \cdot 10^6$	28	2	8.4	Distinguish between a signal process that produces Higgs bosons and a background process that does not.
Epsilon	Epsilon	[67]	$5 \cdot 10^5$	$2 \cdot 10^3$	2	11	This is an artificial data set for the Pascal large scale learning challenge in 2008.
-	Infimnist	[32]	$14 \cdot 10^6$	784	10	20	Identify digits in images representing numbers $\in [0, 9]$.
webspam tri-gram	Webspam	[60]	$3.5 \cdot 10^5$	$1.7 \cdot 10^7$	2	24	Identify whether a web page is spam, or a page created to manipulate search engines and deceive Web users.
Dna	Dna	[56]	$5 \cdot 10^7$	200	2	54	Identify translation initiation sites in nucleotide sequences.

the impact of block size and number of partitions on the results, since these parameters change the absolute values but not the relative performances when comparing different algorithms [6].

The rest of this section is organized as follows. The "Memory Requirements" section presents the results on the memory requirements of ReForeSt versus those of MLlib. The "Computational Requirements and Scalability" section presents a comparison between ReForeSt and MLlib in terms of computational requirements and scalability. Finally, the "Accuracies of the Learned RF Models" section compares the accuracies of the learned RF models implemented in MLlib against ReForeSt RF, ReForeSt RRE, and ReForeSt when the MS function is activated (Fig. 8).

Memory Requirements

In this first set of experiments, we evaluated the memory consumption of ReForeSt and MLlib. For this purpose, we collect the Java Virtual Machine (JVM) memory usage in each second of computation, cleaning the memory with the JVM Garbage Collector before measurement to avoid artifacts in the results. Figure 9 depicts the average memory consumption of ReForeSt and MLlib for the Higgs dataset with different cluster environments ($N_m = 8$ and $N_c = \{4, 8, 16\}$) and with $n_b = 32$ and $n_d = 10$. The x-axis of Fig. 9 represents the normalized computational time with respect to the total time to better compare ReForeSt and MLlib memory usage, while the y-axis reports the memory usage in gigabits (Gb). Figure 9 shows the memory usage choices due to the allocation of the matrices exploited by ReForeSt and MLlib to collect information at each iteration of tree creation. From Fig. 9, it is possible to observe that:

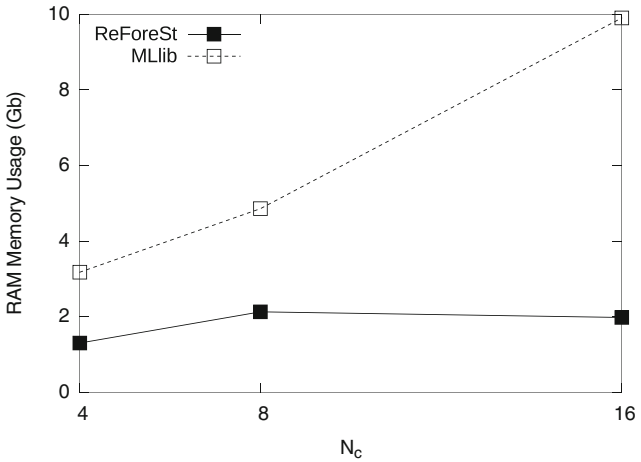


Fig. 8 Memory usage comparison of ReForeSt and MLlib over the Higgs dataset with $N_m = 8$, $N_c = \{4, 8, 16\}$, $n_t = 100$, $n_w = 32$, and $n_d = 10$

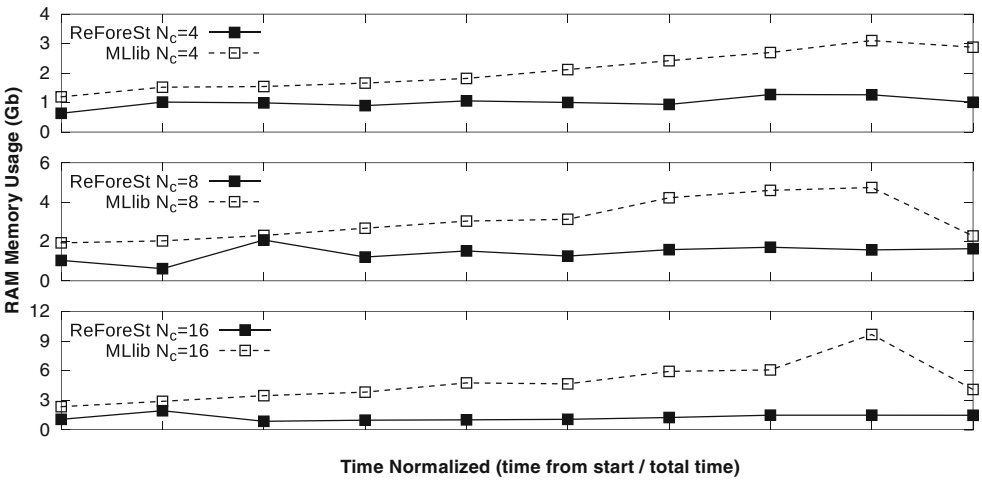


Fig. 9 Memory usage of ReForeSt and MLlib over the Higgs dataset with $N_m = 8$, $N_c = \{4, 8, 16\}$, $n_t = 100$, $n_w = 32$ and $n_d = 10$

- ReForeSt always requires less memory than does MLlib to perform the computation;
- ReForeSt exhibits similar memory consumption for the whole execution, whereas MLlib shows a constant increase;
- the MLlib memory usage increases linearly with the number of cores, whereas the ReForeSt memory usage does not depend on the number of cores (by design).

The last observation is particularly evident when we analyze Fig. 8, where the maximum memory consumption is depicted on the y-axis for different cluster environments $N_c = \{4, 8, 16\}$. ReForeSt has always a maximum memory consumption lower than that of MLlib, and its value is constant over all configurations. Results over the other datasets are not reported since they present similar behavior.

Computational Requirements and Scalability

In this section, we study the computational requirements and scalability of ReForeSt and MLlib with respect to several parameters:

- the number of trees $n_t \in \{100, 200, 400\}$;
- the number of trees $N_m \in \{4, 8, 16\}$;
- the number of cores in each machine $N_c \in \{4, 8, 16\}$.

Tables 4, 5, and 6 report the computational time of ReForeSt RF with t_R, MLlib with t_M, and $\mathsf{D} = \mathsf{t}_M / \mathsf{t}_R$ for all the datasets described in Table 3. Tables 4, 5 and 6 depict the scalability with respect to n_t, N_c and N_m, respectively. In all the tables, n_b and n_d have been set to the default MLlib values ($n_b = 32$ and $n_d = 10$); the standard deviation is not reported because of space constraints. However, changing n_b and n_d does not substantially change the results, and the standard deviation of the results is always less than 5% of the reported values.

Based on the results in Tables 4, 5, and 6, it is possible to observe that:

- ReForeSt is much more efficient than MLlib; in fact, in all the experiments, ReForeSt is faster than MLlib, and on average, ReForeSt is 2 times faster than MLlib;
- ReForeSt scales better with n_t than does MLlib. For instance, on the Epsilon dataset, MLlib requires 2.38, 3.36, 4.9 as much time as ReForeSt for $n_t = 100, 200, 400$, respectively;
- ReForeSt scales better with both N_c and N_m; in fact, D increases when N_c and/or N_m increases;
- ReForeSt may exhibit a super-linear scale-up in several cases, for instance, in Table 6 for the Kdd dataset or in Table 5 for the Infimnist dataset. In fact, ReForeSt automatically adapts to the environment, switching between the two different data partitioning approaches presented in the "ReForeSt" section. The more machines we have, the sooner the *local computation* can start, and then, the faster the results can be computed;
- ReForeSt can complete the computation on the Kdd and Webspam datasets, which have $n_f > 10^7$, in a reasonable amount of time, whereas MLlib returns JVM errors;
- when working with categorical data using the Dna dataset, ReForeSt is still faster than MLlib.

To derive some more insight on the results of Tables 4, 5, and 6, we will make use of some figures.

Figure 10 presents a subset of the results in Table 5 and depicts the speedup metric S. This metric has been computed by dividing the actual time by the time required to compute the same dataset in a baseline scenario with $N_c = 4$. From Fig. 10, it is possible to observe that:

- both ReForeSt and MLlib scale better when the dataset is larger, for example, compare the result on a smaller

Table 4 Comparison between the computational times in seconds of MLlib and ReForeSt with $N_m = 8$, $N_c = 8$ and $n_t = \{100, 200, 400\}$. "∗" indicates a JVM out-of-memory error, ">" indicates that the configuration was not completed in 3000 s, and "−" indicates that the value could not be computed

N_c	Covertype			Karhunen			Susy			Kdd			Higgs			Epsilon			Infimnist			Webspam			Dna		
	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D
100	*17*	30	1.76	*24*	81	3.38	*68*	110	1.62	*647*	*	–	*158*	233	1.47	*66*	157	2.38	*513*	659	1.28	*71*	*	–	*845*	1201	1.42
200	*20*	44	2.2	*33*	146	4.42	*105*	212	2.02	*1233*	*	–	*268*	459	1.71	*76*	255	3.36	*823*	1177	1.42	*68*	*	–	*1464*	>	–
400	*26*	76	2.92	*47*	277	5.89	*198*	425	2.15	*2351*	*	–	*458*	952	2.08	*96*	470	4.9	*1403*	2258	1.61	*70*	*	–	*2842*	>	–

The result in italic is the best result

Table 5 Comparison between the computational times in seconds of MLlib and ReForeSt with $N_m = 8$, $n_t = 200$ and $N_c = \{4, 8, 16\}$. "∗" indicates a JVM out-of-memory error, ">" indicates that the configuration was not completed in 3000 s, and "−" indicates that the value could not be computed

N_c	Covertype			Karhunen			Susy			Kdd			Higgs			Epsilon			Infimnist			Webspam			Dna		
	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D
4	*28*	57	2.04	*53*	184	3.47	*191*	371	1.94	*2051*	*	–	*462*	755	1.63	*132*	313	2.37	*1638*	*	–	*127*	*	–	*2828*	*	–
8	*20*	44	2.2	*33*	146	4.42	*105*	212	2.02	*1233*	*	–	*268*	459	1.71	*76*	255	3.36	823	1177	1.43	*68*	*	–	*1464*	>	–
16	*19*	50	2.63	*25*	157	6.28	*77*	187	2.43	645	*	–	*157*	333	2.12	*56*	201	3.59	*538*	976	1.81	*54*	*	–	*1063*	1610	1.51

The result in italic is the best result

Table 6 Comparison between the computational times in seconds of MLlib and ReForeSt with $n_t = 200$, $N_c = 8$ and $N_m = \{4, 8, 16\}$. "*" indicates a JVM out-of-memory error, ">" indicates that the configuration was not completed in 3000 s, and "–" indicates that the value could be computed

N_m	Covertype			Karhunen			Susy			Kdd			Higgs			Epsilon			Infimnist			Webspam			Dna		
	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D	t_R	t_M	D
4	*25*	55	2.2	*47*	186	3.96	*177*	371	2.1	*2136*	*	–	*432*	812	1.88	*131*	313	2.39	*1463*	*	–	*134*	*	–	*2795*	*	–
8	*20*	44	2.2	*33*	146	4.42	*105*	212	2.02	*1233*	*	–	*268*	459	1.71	*76*	255	3.36	*823*	1177	1.43	*68*	*	–	*1464*	>	–
16	*18*	45	2.5	*27*	143	5.3	*82*	187	2.28	*580*	*	–	*151*	269	1.78	*50*	201	4.02	*592*	976	1.65	*41*	*	–	*963*	>	–

The result in italic is the best result

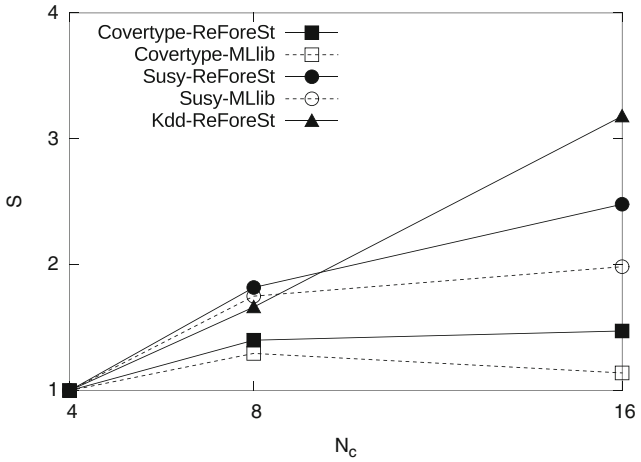


Fig. 10 Speedup when evaluating the scalability with $N_c = \{4, 8, 16\}$. The speedup is the difference between the actual computational time and the time to compute with $N_c = 4$

dataset (e.g., Covertype) to that on a larger dataset (e.g., Susy);
- ReForeSt scales better than does MLlib; in all cases, the speedup of ReForeSt is better than that of MLlib;
- the bad scalability results for the Covertype dataset occur because the time to learn the forest is negligible compared to the dataset reading time. In fact, on larger datasets such as Kdd, the speedup is larger.

Finally, Fig. 11 depicts D as N_m varies. From these last figures, it is possible to observe that:

- in general, D when $N_m = 16$ is larger than in the case when $N_m = 4$, which indicates that ReForeSt scales better than MLlib as the number of machines increases;
- we have $\mathsf{D} < 2$ only for the Higgs dataset; on the other datasets, we always obtain $\mathsf{D} > 2$. On average, ReForeSt is 2 times faster than MLlib.

We do not compare the forward phase times because ReForeSt and MLlib have identical forward phases. Nevertheless, the forward phase computational requirements of an RF are negligible ($\approx$ milliseconds) and scale fully linearly with N_m and the number of cores N_c.

Accuracies of the Learned RF Models

In this set of experiments, we will compare the accuracies of the learned RF models implemented in MLlib against those learned via ReForeSt RF and ReForeSt RRE. Moreover, we will show the benefits of using ReForeSt MS, both in terms of accuracy when the hyperparameters are set to their default values and also in terms of computational time when implementing MS in a naive way by simply checking all possible hyperparameter combinations (AC).

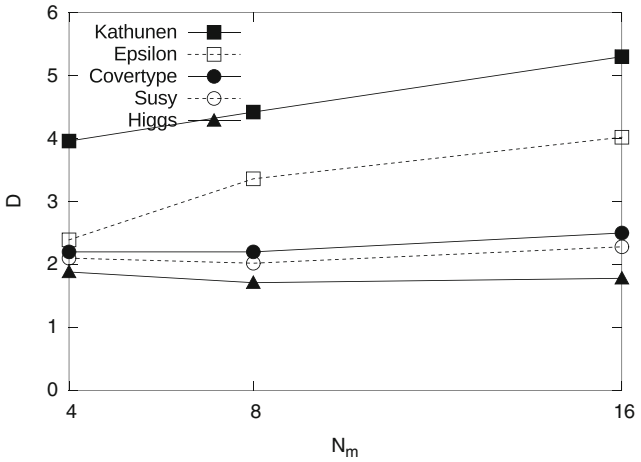


Fig. 11 The D values when evaluating the scalability with $N_m = \{4, 8, 16\}$

Regarding the first issue, Table 7 reports the empirical error for the test set $\widehat{L}$ and the computational times t for ReForeSt RF, ReForeSt RRE, and MLlib. For the sake of completeness, we also report on a comparison with a deep neural network implemented in Keras[5] running on Tensorflow[6]. In the network, we exploited rectified linear units. The numbers of layers and units have been optimized for each dataset, and gradient descend has been exploited as an optimizer. Based on the results of Table 7, it is possible to observe that:

- the ReForeSt RF and MLlib errors are statistically equivalent, as expected;
- ReForeSt RRE may, in some cases, outperform ReForeSt RF, as expected, based on the results of [9] (see the Covertype and Karhunen datasets);
- the computational times of ReForeSt RRE and MLlib are similar, while ReForeSt RF is the fastest method;
- the accuracy of DL is sometimes higher than that of RF, but the computational time needed to train the model is always much higher.

For the effectiveness and computational requirements of ReForeSt MS, the results are reported in Table 8. We search for the best hyperparameters by specifying the following sets of values: $\mathcal{P}^{n_t} = \{50, 100, 200\}$, $\mathcal{P}^{n_d} = \{6, 8, 10, 12, 14\}$, $\mathcal{P}^{n_w} = \{16, 24, 32, 40, 48\}$, $\mathcal{P}^{n_v} = n_f^{\{0.4,0.5,0.6,0.7\}}$. Table 8 reports, as in Table 7, $\widehat{L}$ and t for ReForeSt RF in three different cases:

- when the parameters are set to their default values $n_t = 100$, $n_d = 10$, $n_w = 32$, and $n_v = n_f^{0.5}$ (ReForeSt NoMS)

[5] https://keras.io/

[6] https://www.tensorflow.org/

Table 7 Comparison between accuracy and computational time in seconds of MLlib, ReForeSt, random rotations with ReForeSt with $n_t = 200$ and deep learning (DL)

	Covertype				Karhunen				Susy				Higgs			
	ReForeSt		MLlib	DL	ReForeSt		MLlib	DL	ReForeSt		MLlib	DL	ReForeSt		MLlib	DL
	RF	RRE			RF	RRE			RF	RRE			RF	RRE		
$\widehat{L}$	0.74	0.77	0.74	*0.8*	0.93	*0.96*	0.93	0.94	*0.79*	0.73	*0.79*	*0.79*	0.71	0.58	0.71	*0.78*
t	29	68	44	180	*65*	124	146	369	*122*	174	212	505	*310*	485	463	1207

The result in italic is the best result

Table 8 Comparison between ReForeSt and ReForeSt when performing MS

	Covertype			Karhunen			Susy			Higgs			Epsilon		
	NoMS	AC	MS	NoMS	AC	MS	NoMS	AC	MS	NoMS	AC	MS	NoMS	AC	MS
$\widehat{L}$	0.74	*0.85*	*0.85*	0.93	0.96	*0.97*	0.78	*0.80*	*0.80*	0.71	*0.73*	*0.73*	0.76	*0.78*	0.77
t	20	919	*413*	33	2918	*514*	105	3394	2229	268	9150	*4939*	76	2425	*1113*

The result in italic is the best result

– when we select the best hyperparameters by checking all possible combinations $n_t, n_d, n_w, n_v \in \mathcal{P}^{n_t} \times \mathcal{P}^{n_d} \times \mathcal{P}^{n_w} \times \mathcal{P}^{n_v}$ with the naive algorithm of Eq. 10 (ReForeSt AC);
– when the ReForeSt MS proposal of Algorithm 8 is exploited (ReForeSt MS).

Based on the results of Table 8, we can observe that:

– ReForeSt MS is generally faster than ReForeSt AC while achieving the same accuracy; for instance, in the Higgs dataset, ReForeSt MS completes the execution in 54% of the time needed by ReForeSt AC;
– ReForeSt MS can remarkably improve the accuracy of the final RF model in the execution with the default ReForeSt NoMS.

For the sake of completeness, we report in Table 9 the computational time t, the scalability S of ReForeSt NoMS (t_{NoMS} and S_{NoMS}), the ReForeSt AC (t_{AC} and S_{AC}), the ReForeSt MS (t_{MS}, S_{MS}), and $D_{MS} = t_{AC}/t_{MS}$ for the Epsilon dataset. Based on the results shown in Table 9, we observe that:

– ReForeSt MS scales better than does ReForeSt AC (see columns S_{MS} and S_{AC}). For instance, when $N_m = 8$, S_{MS} is equal to 1.8, 3.36, 4.49 and S_{AC} is equal to 1.57, 2.56, 3.67 for $N_c \in \{4, 8, 16\}$, respectively;
– ReForeSt MS scales better than does ReForeSt NoMS; in all scenarios, the S_{MS} values are larger than the corresponding S_{NoMS} values;
– in all configurations, ReForeSt MS is around 2× faster than ReForeSt AC (see columns D_{MS}).

In conclusion, ReForeSt NoMS is fast but inaccurate, ReForeSt AC is slow but accurate, and ReForeSt MS is much faster than but otherwise comparable to ReForeSt AC.

Table 9 Scalability evaluation of ReForeSt MS over the Epsilon dataset. S_R, S_C, and S_{MS} show, respectively, the scalabilities of ReForeSt, ReForeSt with all the configured hyperparameters, and ReForeSt MS with respect to the executions with $N_m = 4$ and $N_c = 4$

N_m	N_c	t_R	t_C	t_{MS}	S_R	S_C	S_{MS}	D_{MS}
4	4	238	6199	*3738*	1	1	1	1.66
4	8	139	3809	*2031*	1.71	1.63	1.84	1.88
4	16	106	2614	*1312*	2.25	2.37	2.85	1.99
8	4	145	3947	*2077*	1.64	1.57	1.8	1.9
8	8	90	2425	*1113*	2.64	2.56	3.36	2.18
8	16	69	1689	*832*	3.45	3.67	4.49	2.03
16	4	98	2692	*1171*	2.43	2.3	3.19	2.3
16	8	70	1737	*755*	3.4	3.57	4.95	2.3
16	16	61	1434	*647*	3.9	4.32	5.78	2.22

The result in italic is the best result

Conclusions

This paper presents ReForeSt, an Apache Spark–based fully distributed implementation of random forests, which is one of the best learning algorithms in the context of classification. Our implementation allows us to unlock the potential of random forests for the analysis of large datasets commonly available in the current big data era.

Unlike currently available solutions, our implementation supports different data partitioning approaches for optimizing the computational and memory requirements of RFs; arbitrarily large datasets ranging from millions of samples to millions of features; a recently proposed, improved RF formulation called random rotation ensembles; and model selection for auto-tuning the RF hyperparameters. ReForeSt is a good alternative to the de-facto standard MLlib and compensates for its shortcomings. Results for a wide range of large datasets analyzed on the Google Cloud Platform confirm that ReForeSt outperforms MLlib in terms of computational time while providing additional functionalities, such as multiple data partitioning and computation strategies, support for random rotations and model selection, and greater accuracy.

Future goals include the refinement of ReForeSt to reduce computational time and the development of a platform for RF regression.

Compliance with Ethical Standards

Conflict of Interest The authors declare that they have no conflict of interest.

Ethical Approval This article does not contain any studies with human participants or animals performed by any of the authors.

Informed Consent Not applicable.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

1. Abdullah A, Hussain A, Khan IH. Introduction: dealing with big data-lessons from cognitive computing. Cogn Comput. 2015;7(6):635–636.
2. Anguita D, Ghio A, Oneto L, Ridella S. In-sample and out-of-sample model selection and error estimation for support vector machines. IEEE Trans Neural Netw Learn Syst. 2012;23:1390–1406.
3. Arlot S, Celisse A. A survey of cross-validation procedures for model selection. Stat Survey. 2010;4:40–79.
4. Baldi P, Sadowski P, Whiteson D. Searching for exotic particles in high-energy physics with deep learning. Nat Commun. 2014;5(4308):1–9.
5. Bernard S, Heutte L, Adam S. Influence of hyperparameters on random forest accuracy. In: MCS. pp. 171–180; 2009.
6. Bertolucci M, Carlini E, Dazzi P, Lulli A, Ricci L. Static and dynamic big data partitioning on apache spark. In: PARCO. pp. 489–498; 2015.
7. Biau G. Analysis of a random forests model. J Mach Learn Res. 2012;13:1063–1095.
8. Blackard J, Dean D. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. Comput Electron Agric. 1999;24(3):131–151.
9. Blaser R, Fryzlewicz P. Random rotation ensembles. J Mach Learn Res. 2015;2:1–15.
10. Bosse T, Duell R, Memon ZA, Treur J, van der Wal CN. Agent-based modeling of emotion contagion in groups. Cogn Comput. 2015;7(1):111–136.
11. Breiman L. Random forests. Mach Learn. 2001;45(1):5–32.
12. Cambria E, Chattopadhyay A, Linn E, Mandal B, White B. Storages are not forever. Cogn Comput. 2017;9(5):646–658.
13. Cao L, Sun F, Liu X, Huang W, Kotagiri R, Li H. End-to-end convnet for tactile recognition using residual orthogonal tiling and pyramid convolution ensemble. Cogn Comput. 2018;10(5):1–19.
14. Chen J, Li K, Tang Z, Bilal K, Yu S, Weng C, Li K. A parallel random forest algorithm for big data in a spark cloud computing environment. IEEE Trans Parallel Distributed Syst. 2017;28(4):919–933.
15. Chung S. Sequoia forest : random forest of humongous trees. In: Spark summit; 2014.
16. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–113.
17. Donders ART, van der Heijden GJMG, Stijnen T, Moons KGM. Review: a gentle introduction to imputation of missing values. J Clin Epidemiol. 2006;59(10):1087–1091.
18. Fernández-Delgado M, Cernadas E, Barro S, Amorim D. Do we need hundreds of classifiers to solve real world classification problems? J Mach Learn Res. 2014;15(1):3133–3181.
19. Galton F. Vox populi (the wisdom of crowds). Nature. 1907;75(7):450–451.
20. Gashler M, Giraud-Carrier C, Martinez T. Decision tree ensemble: small heterogeneous is better than large homogeneous. In: International conference on machine learning and applications; 2008.
21. Genuer R, Poggi J, Tuleau-Malot C, Villa-Vialaneix N. Random forests for big data. In: arXiv:1511.08327; 2015.
22. George L. HBAse: the definitive guide: random access to your planet-size data. Sebastopol: O'Reilly Media, Inc; 2011.
23. Hastie T, Tibshirani R, Friedman J. The elements of statistical learning: data mining, inference, and prediction. Berlin: Springer; 2009.
24. Hernández-Lobato D, Martínez-muñoz G, Suárez A. How large should ensembles of classifiers be? Pattern Recogn. 2013;46(5):1323–1336.
25. Hilbert M. Big data for development: a review of promises and challenges. Dev Policy Rev. 2016;34(1):135–174.
26. Jin XB, Xie GS, Huang K, Hussain A. Accelerating infinite ensemble of clustering by pivot features. Cogn Comput. 2018;1–9. https://link.springer.com/article/10.1007/s12559-018-9583-8.
27. Karau H, Konwinski A, Wendell P, Zaharia M. Learning spark: lightning-fast big data analysis. Sebastopol: O'Reilly Media Inc; 2015.
28. Khan FH, Qamar U, Bashir S. Multi-objective model selection (moms)-based semi-supervised framework for sentiment analysis. Cogn Comput. 2016;8(4):614–628.

29. Kleiner A, Talwalkar A, Sarkar P, Jordan MI. A scalable bootstrap for massive data. J R Stat Soc Ser B Stat Methodol. 2014;76(4):795–816.
30. Li Y, Zhu E, Zhu X, Yin J, Zhao J. Counting pedestrian with mixed features and extreme learning machine. Cogn Comput. 2014;6(3):462–476.
31. Liu N, Sakamoto JT, Cao J, Koh ZX, Ho AFW, Lin Z, Ong MEH. Ensemble-based risk scoring with extreme learning machine for prediction of adverse cardiac events. Cogn Comput. 2017;9(4):545–554.
32. Loosli G, Canu S, Bottou L. Training invariant support vector machines using selective sampling. In: Large scale kernel machines; 2007.
33. Lulli A, Carlini E, Dazzi P, Lucchese C, Ricci L. Fast connected components computation in large graphs by vertex pruning. IEEE Trans Parallel Distributed Syst. 2017;28(3):760–773.
34. Lulli A, Debatty T, Dell'Amico M, Michiardi P, Ricci L. Scalable k-nn based text clustering. In: IEEE International conference on big data. pp. 958–963; 2015.
35. Lulli A, Oneto L, Anguita D. Crack random forest for arbitrary large datasets. In: IEEE International conference on big data (IEEE BIG DATA); 2017.
36. Lulli A, Oneto L, Anguita D. Reforest: random forests in apache spark. In: International conference on artificial neural networks; 2017.
37. Manjusha KK, Sankaranarayanan K, Seena P. Prediction of different dermatological conditions using naive bayesian classification. Int J Adv Res Comput Sci Softw Eng. 2014;4.
38. Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai DB, Amde M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A. Mllib: machine learning in apache spark. J Mach Learn Res. 2016;17(1):1235–1241.
39. Ofek N, Poria S, Rokach L, Cambria E, Hussain A, Shabtai A. Unsupervised commonsense knowledge enrichment for domain-specific sentiment analysis. Cogn Comput. 2016;8(3):467–477.
40. Oneto L. Model selection and error estimation without the agonizing pain. WIREs DMKD. 2018;, pp (In–Press).
41. Oneto L, Bisio F, Cambria E, Anguita D. Statistical learning theory and elm for big social data analysis. IEEE Comput Intell Mag. 2016;11(3):45–55.
42. Oneto L, Bisio F, Cambria E, Anguita D. Semi-supervised learning for affective common-sense reasoning. Cogn Comput. 2017;9(1):18–42.
43. Oneto L, Bisio F, Cambria E, Anguita D. Slt-based elm for big social data analysis. Cogn Comput. 2017;9(2):259–274.
44. Oneto L, Coraddu A, Sanetti P, Karpenko O, Cipollini F, Cleophas T, Anguita D. Marine safety and data analytics: Vessel crash stop maneuvering performance prediction. In: International conference on artificial neural networks; 2017.
45. Oneto L, Fumeo E, Clerico C, Canepa R, Papa F, Dambra C, Mazzino N, Davide A. Train delay prediction systems: a big data analytics perspective. Big Data Research. 2017, pp (in–press).
46. Orlandi I, Oneto L, Anguita D. Random forests model selection. In: European symposium on artificial neural networks, computational intelligence and machine learning; 2016.
47. Ortín S, Pesquera L. Reservoir computing with an ensemble of time-delay reservoirs. Cogn Comput. 2017;9(3):327–336.
48. Panda B, Herbach J, Basu S, Bayardo R. Planet: massively parallel learning of tree ensembles with mapreduce. In: International conference on very large data bases; 2009.
49. Reyes-Ortiz JL, Oneto L, Anguita D. Big data analytics in the cloud: spark on hadoop vs mpi/openmp on beowulf. Procedia Comput Sci. 2015;53:121–130.
50. Rijn J. BNG(mfeat-karhunen) - OpenML Repository. https://www.openml.org/d/252. 2014.
51. Rokach L, Maimon O. Data mining with decision trees: theory and applications world scientific. 2008.
52. Rotem D, Stockinger K, Wu K. Optimizing candidate check costs for bitmap indices. In: Proceedings of the 14th ACM international conference on Information and knowledge management. pp 648–655; 2005.
53. Ryza S. Advanced analytics with spark: patterns for learning from data at scale. Sebastopol: O'Reilly Media Inc; 2017.
54. Segal MR. Machine learning benchmarks and random forest regression. In: UCSF: center For bioinformatics and molecular biostatistics; 2004.
55. Shalev-Shwartz S, Ben-David S. Understanding machine learning: from theory to algorithms. Cambridge: Cambridge University Press; 2014.
56. Sonnenburg S, Franc V, Yom-Tov E, Sebag M. Pascal large scale learning challenge. In: International conference on machine learning; 2008.
57. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: a warehousing solution over a map-reduce framework. Proc VLDB Endowment. 2009;2(2):1626–1629.
58. Wainberg M, Alipanahi B, Frey BJ. Are random forests truly the best classifiers? J Mach Learn Res. 2016;17(1):3837–3841.
59. Wakayama R, Murata R, Kimura A, Yamashita T, Yamauchi Y, Fujiyoshi H. Distributed forests for mapreduce-based machine learning. In: IAPR Asian conference on pattern recognition; 2015.
60. Wang D, Irani D, Pu C. Evolutionary study of web spam: Webb spam corpus 2011 versus webb spam corpus 2006. In: International conference on collaborative computing: networking, Applications and Worksharing; 2012.
61. Wen G, Hou Z, Li H, Li D, Jiang L, Xun E. Ensemble of deep neural networks with probability-based fusion for facial expression recognition. Cogn Comput. 2017;9(5):597–610.
62. White T. Hadoop: The definitive guide. Sebastopol: O'Reilly Media Inc; 2012.
63. Wolpert DH. The lack of a priori distinctions between learning algorithms. Neural Comput. 1996;8(7):1341–1390.
64. Wu X, Zhu X, Wu G, Ding W. Data mining with big data. IEEE Trans Knowl Data Eng. 2014;26(1):97–107.
65. Yang B, Zhang T, Zhang Y, Liu W, Wang J, Duan K. Removal of electrooculogram artifacts from electroencephalogram using canonical correlation analysis with ensemble empirical mode decomposition. Cogn Comput. 2017;9(5):626–633.
66. Yu H, Hsieh C, Chang K, Lin C. Large linear classification when data cannot fit in memory. ACM Trans Knowl Discovery Data. 2012;5(4):23.
67. Yuan G, Ho C, Lin C. An improved glmnet for l1-regularized logistic regression. J Mach Learn Res. 2012;13:1999–2030.
68. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on networked systems design and implementation. pp. 2–2; 2012.
69. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. HotCloud. 2010; 10(10–10):1–9.
70. Zhang S, Huang K, Zhang R, Hussain A. Learning from few samples with memory network. Cogn Comput. 2018;10(1):15–22.
71. Zhou ZH. Ensemble methods: foundations and algorithms. Boca Raton: CRC Press; 2012.