# ReForeSt: Random Forests in Apache Spark

Alessandro Lulli, Luca Oneto, and Davide Anguita

DIBRIS - University of Genoa, Via Opera Pia 13, Genova, I-16145, Italy
alessandro.lulli@dibris.unige.it,{luca.oneto,davide.anguita}@unige.it

**Abstract.** Random Forests (RF) of tree classifiers are a popular ensemble method for classification. RF are usually preferred with respect to other classification techniques because of their limited hyperparameter sensitivity, high numerical robustness, native capacity of dealing with numerical and categorical features, and effectiveness in many real world classification problems. In this work we present ReForeSt, a Random Forests Apache Spark implementation which is easier to tune, faster, and less memory consuming with respect to MLlib, the de facto standard Apache Spark machine learning library. We perform an extensive comparison between ReForeSt and MLlib by taking advantage of the Google Cloud Platform[1]. In particular, we test ReForeSt and MLlib with different library settings, on different real world datasets, and with a different number of machines equipped with different number of cores. Results confirm that ReForeSt outperforms MLlib in all the above mentioned aspects. ReForeSt is made publicly available via GitHub[2].

**Keywords:** Random Forests, Apache Spark, Open Source Software

## 1 Introduction

It is well known that combining the output of several classifiers results in a much better performance than using any one of them alone [11]. In [4] Breiman proposed the Random Forests (RF) of tree classifiers, one of the state-of-the-art learning algorithm for classification which has shown to be one of the most effective tool in this context [9, 16]. RF combine bagging to random subset feature selection. In bagging, each tree is independently constructed using a bootstrap sample of the dataset [8]. RF add an additional layer of randomness to bagging. In addition to constructing each tree using a different bootstrap sample of the data, RF change how the classification trees are constructed. In standard trees, each node is split using the best division among all variables. In RF, each node is split using the best among a subset of predictors randomly chosen at that node. Eventually, a simple majority vote is taken for prediction.

The challenge today is that the size of data is constantly increasing making infeasible to analyze it with classic learning algorithms or their naive implementations [14]. Data is often distributed, since it is collected and stored by distributed platforms like the Hadoop-based ones [7]. Apache Spark [19] is currently gaining momentum for distributed processing because it enables fast iterative in-memory computation with respect to classical disk-based MapReduce

[1] https://cloud.google.com
[2] https://github.com/alessandrolulli/reforest

jobs. MLlib [14] is the de facto standard for the use of RF learning algorithm in a distributed environment. It is built-in in Spark, implemented by the very same authors. They follow a common approach to distributed computation. The initial data is divided into a number of partitions, which is in general larger with respect to the number of machines. Each machine maintains a subset of the partitions. Therefore, in order to construct the forest (i) the needed information must be collected inside a partition, (ii) it is aggregated locally in the machine, and (iii) all the machines' information is aggregated in order to complete the computation. The main downside is that for each node an amount of memory proportional with the number of partitions handled by the machine is allocated.

In addition to the already mentioned MLlib [14] other RF implementations exist, but they show analogous drawbacks. For instance, Chung [6] presents an optimization of MLlib that switches from distributed to local computation when the size of the data relative to a sub-tree is below a threshold, but no source code is available. In [10] it is proposed to send each chunk of data to an independent job and then aggregate the information of each chunk. However, this approach requires the transmission of the entire subsamples to all the machines in the cluster. Wakayama et al. [17] construct in each machine a candidate random forest. Such forests are then aggregated and just the trees which show the best accuracy are kept. The major drawback is that many trees are discarded resulting in wasting computation time. Chen et al. [5] propose to vertically partition the data. The training dataset is split into several feature subsets and then 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 an old implementation of MLlib.

In this paper we present ReForeSt, a distributed, scalable implementation of the RF learning algorithm which targets fast and memory efficient processing. ReForeSt main contributions are manifold: (i) it provides a novel approach for the RF implementation in a distributed environment targeting an in-memory efficient processing, (ii) it is faster and more memory efficient with respect to the de facto standard MLlib, (iii) the level of parallelism is self-configuring, and (iv) its source code is made publicly available. With respect to current approaches we provide several benefits. With respect to MLlib we avoid the use of multiple data structures requested by the partitioning-based distributed computation, since each machine maintains only one data structure to collect the information of the data stored in it. Contrarily to [14] we grow all the trees in parallel in order to reduce the number of scans of the data. With respect to [17] and [5, 10] we avoid to generate trees that are not useful for the final result and to perform too many communications in each iteration respectively. Finally, we do not occupy memory with repetitions of the same data as opposed to [5].

## 2 ReForeSt: Random Forests in Apache Spark

Let us recall the multi-class classification problem [3] where a set of labeled samples $\mathcal{D}_n = \{(X_1, Y_1), \cdots, (X_n, Y_n)\}$ drawn according to an unknown probability distribution $\mu$ over $\mathcal{X} \times \mathcal{Y}$ are available and where $X \in \mathcal{X} = \{\mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_d\}$ and $Y \in \mathcal{Y} = \{1, 2, \cdots, c\}$. A learning algorithm $\mathscr{A}$ maps $\mathcal{D}_n$ into a function belonging to a possibly unknown set of functions $f \in \mathcal{F}$ according to some criteria $\mathscr{A} : \mathcal{D}_n \rightarrow \mathcal{F}$. The error of $f$ in approximating $\mu$ is measured with reference to

a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. Since we are dealing with classification problems we choose the loss function which counts the number of misclassified samples $\ell(f(X), Y) = [f(X) \neq Y]$, where the Iverson bracket notation is exploited. The expected error of $f$ in representing $\mu$ is called generalization error [15] and it is defined as $L(f) = \mathbb{E}_{(X,Y)}\ell(f(X), Y)$. Since $\mu$ is unknown $L(f)$ cannot be computed, but we can compute its empirical estimator, the empirical error, defined as $\widehat{L}(f) = {}^1\!/\!{}_n \sum_{i=1}^{n} \ell(f(\widehat{X}_i), \widehat{Y}_i)$ where $\mathcal{T}_n = \{(\widehat{X}_1, \widehat{Y}_1), \cdots, (\widehat{X}_n, \widehat{Y}_n)\}$ must be a different set with respect to $\mathcal{D}_n$ which has been used to build $f$ in order to ensure that the estimator of the quality of the model is unbiased [1].

In our case $\mathscr{A}$ are RF. We briefly describe the learning phase of each of the $n_t$ trees composing the RF. From $\mathcal{D}_n$, $\lfloor bn \rfloor$ samples are sampled with replacement and $\mathcal{D}'_{\lfloor bn \rfloor}$ is built. A tree is constructed with $\mathcal{D}'_{\lfloor bn \rfloor}$ but the best split is chosen among a subset of $n_v$ predictors over the possible $d$ predictors randomly chosen at each node. The tree is grown until its depth reaches the maximum value of $n_d$ or all the samples in $\mathcal{D}_n$ are correctly classified. During the classification phase of a previously unseen $X \in \mathcal{X}$, each tree classifies $X$ in a class $Y_{i \in \{1, \cdots, n_t\}} \in \mathcal{Y}$, and then the final classification is the majority vote of all the answers of each tree of the RF. If $b = 1$, $n_v = \sqrt{n}$, and $n_d = \infty$ we get the original RF formulation [4] where $n_t$ is usually chosen to tradeoff accuracy and efficiency [12].

ReForeSt is a distributed, scalable algorithm for RF computation targeting fast and memory efficient processing. Our main idea is to create only one data structure, called *matrix*, on each machine. This is used for local data aggregation and to concurrently aggregate information to compute the best cuts. The goal is to reduce the memory requirements and the computational time with respect to alternative approaches. For example, MLlib uses one data structure for each partition in order to collect the information resulting in an inefficient memory management.

Our proposal (see Algorithm 1) counts two main phases. The first phase is called *data preparation*. The output of the first phase is the *working data*, a statically allocated collection of items. Each item is an optimized representation of the original data coupled with how much it contributes to each of the trees. The second phase performs the *tree generation*. The working data is iteratively processed to grow each tree of the forest.

### 2.1 Data Preparation

Starting from the raw data $\mathcal{D}_n$ we build the *working data*. Such working data is kept in memory for the entire duration of the second phase statically. For performance reasons, similarly to [14, 6], the domain of each feature is discretized. We call this operation *binning* and the number of bin $n_b$ is configurable. For each feature, we search for the $n_b$ values for splitting the domain (see L:3, namely Line 3 of Algorithm 1) using a sample $\mathcal{D}''_s \subset \mathcal{D}_n$ where $s \approx 10^4$. Than, each bin is constructed to have approximately the same number of samples $\mathcal{D}''_s / n_b$. Finally, each sample $(X_j, Y_j) \in \mathcal{D}_n$ is converted in a working data item $(X_j^{n_b}, Y_j, B_j)$ which belongs to the dataset of converted data items $\mathcal{D}_n^{n_b} = \{(X_j^{n_b}, Y_j, B_j) : j \in \{1, \cdots, n\}\}$. $X_j^{n_b}$ is the discretized version of $X_j$, while $B_j \in \mathbb{N}^{n_t}$ is a vector which contains in $B_{j,i}$ the contribution of $(X_j, Y_j) \in \mathcal{D}_n$ to the $i$-th tree built based on $\mathcal{D}'_{\lfloor bn \rfloor}$.

**Algorithm 1:** ReForeSt pseudo-code.

**Input:** $\mathcal{D}_n$, $n_t$, $b$, $n_b$, $n_v$ and $n_d$
**Output:** A set of tree $\{T_1, \cdots, T_{n_t}\}$
`// Data Preparation`
1 $\mathcal{D}_n \leftarrow loadData()$; `/* Parallelized over the` $n$ `samples */`
2 $\mathcal{D}''_s \leftarrow sample(\mathcal{D}_n)$; `/* Parallelized over the` $d$ `features */`
3 $\mathcal{S} \leftarrow findSplit(n_b, \mathcal{D}''_s)$; `/* Parallelized over the` $d$ `features */`
4 $\mathcal{D}^{n_b}_n \leftarrow convertInWorkingData(\mathcal{D}_n, \mathcal{S}, b)$; `/* Parallelized over the` $n$ `samples */`
`// Tree Generation`
5 $\{T_1, \cdots, T_{n_t}\} \leftarrow initializeEmptyTrees(n_t)$;
6 **for** $i \leftarrow 0, \cdots, (n_d - 1)$ **do**
    `// Local Information Collection`
7     **for** $j \leftarrow 1, \cdots, N_m$ **do in parallel**
8         $M^j \leftarrow$ instantiate a matrix $\mathbb{N}^{2^i n_t \times n_b c n_v}$ ;
9         **for** $(X^{n_b}, Y, B) \in (\mathcal{D}^{n_b}_n)^j$ **do** `/*` $(\mathcal{D}^{n_b}_n)^j \subset \mathcal{D}^{n_b}_n$ `which resides on machine` $j$ `*/`
10             **for** $t \leftarrow 1, \cdots, n_t$ **do**
11                 $node \leftarrow getNode(t, X^{n_b})$;
12                 **if** $\neg isLeaf(node)$ **then**
13                     **for** $f \leftarrow \mathcal{R}_{n_v}$ **do**
14                         $p \leftarrow getColumnInM(t, node, f, X^{n_b}, Y)$;
15                         $r \leftarrow getRowOfMFromNode(node)$;
16                         $M^j_{r,p} \leftarrow M^j_{r,p} + B_t$;
    `// Distributed Information Aggregation`
17     $M \leftarrow mergeByNode(\forall M^j,\ j \in \{1, \cdots, N_m\})$;
    `// Trees Update`
18     **for** $r \leftarrow 1, \cdots, 2^i n_t$ **do in parallel** `/* Every machine handles a subset of the nodes */`
19         $node \leftarrow getNodeFromRowOfM(r)$;
20         $bestCut \leftarrow findBestCut(M_{r,\{1,\cdots,n_b c n_v\}})$;
21         $\{T_1, \cdots, T_{n_t}\}.growTreeBasedOnNode(node, bestCut)$;

## 2.2 Tree Generation

The *tree generation* phase proceeds breadth-first and each tree is computed in parallel. An iteration is divided in three steps: local information collection, distributed information aggregation, and trees update. At each iteration $i \in \{0, \cdots, n_d - 1\}$ all the nodes at the $i$-th level are computed by each machine $j \in \{1, \cdots, N_m\}$ in parallel. In particular, we exploit the matrix $M^j \in \mathbb{N}^{2^i \times n_t \times n_b \times c \times n_v}$ which resides on the $j$-th machine. $M^j$ contains the contributions, needed to select the best split based on the information gain criteria, to the different $n_v$ randomly selected subset of the $d$ original features and to the $c$ classes of the subset of items $\mathcal{D}^{n_b}_n$ handled by the $j$-th machine to the $2^i$ nodes at $i$-th level of the $n_t$ trees of the forest. Note that $M^j$ is flattened from a five-dimensional matrix to a two-dimensional matrix $M^j \in \mathbb{N}^{2^i n_t \times n_b c n_v}$ for performance reasons. Then all the matrices are aggregated in order to collect the information needed to find the best cuts and update the trees of the forest. If $M^j$ does not fit in the memory of the machine, since $M^j$ can become very large, the iterations are automatically divided in many sub-iterations based on the available memory and the number of nodes processed at the $i$-th iteration.

**Local Information Collection** This step does not require any communication between the machines. It operates on the working data saved in each machine $j$ and collects the information in $M^j$ which is instantiated at the beginning of the iteration. All the partitions are processed concurrently by the machine which stores them. In the following we describe how an item $(X^{n_b}, Y, B) \in \mathcal{D}^{n_b}_n$ contributes to $M^j$ for one tree $t \in \{1, \cdots, n_t\}$ with its weight $B_t$. Each $(X^{n_b}, Y, B)$ is stored in a Spark partition and it contributes to all the trees in

the forest as follows. First we have to recall that $(X^{n_b}, Y, B)$ can contribute to only one node per tree since we are processing all the nodes at a particular depth of all the trees. Then, given $(X^{n_b}, Y, B)$ and the $t$-th tree we can navigate it until we reach the right node where $(X^{n_b}, Y, B)$ contributes (see L:11). For each feature $f \in \mathcal{R}_{n_v}$, where $\mathcal{R}_{n_v}$ is a set of $n_v$ indexes randomly sampled without replacement from $\{1, \cdots, d\}$, the proper element of $M^j$ to be updated is found (see L:14). The row of $M^j$ is the specified node index. The column of $M^j$ is computed as $n_b c f + c X_f^{n_b} + Y$. $B_t$ is added to the aforementioned position of $M^j$ (see L:16). At the end of this step, each machine has one matrix populated with the contributions of each item of the working data stored in it.

**Distributed Information Aggregation** In this step the information stored in each $M^j$ is aggregated as follows. The rows of each $M^j$ are shuffled in the machines. In particular, the rows belonging to the same node are collected in a same machine thanks to the Spark hashing function. At the end of this step, each machine stores a subset of the nodes processed at the particular iteration $i$. The number of nodes in each machine is approximately $2^i n_t / N_m$ thanks to the hashing function that distributes the nodes to the machines uniformly. The matrix instantiated at the beginning of the iteration is freed during the shuffling phase.

**Trees Update** In the last step of each iteration each machine, having the complete knowledge about the nodes stored in it, searches the best cuts (see L:20). The best cut is chosen in such a way to maximize the information gain in the sub-tree. All the computed cuts are then exploited to update the $n_t$ trees in the forest (see L:21).

## 3 Experimental Evaluation

In this section ReForeSt and MLlib [14] are compared through a comprehensive set of experiments. We run the experiments on the Google Cloud Platform (GCP) making use of Linux shell scripts for automatically deploying the clusters of virtual machines (VMs). Each VM runs Debian 8.7 and is equipped with Hadoop 2.7.3 and Apache Spark 2.1.0. To evaluate the ReForeSt and MLlib scalability we tested them on different cluster configurations. In particular, one master node is deployed and a different number of worker machines $N_m \in \{4, 8, 16\}$ equipped with different number of cores $N_c \in \{4, 8, 16\}$ is handled by the master. For this purpose, we used the n1-standard-4, n1-standard-8, and n1-standard-16 machine types from GCP with approximately 15, 30, and 60 GB of RAM respectively and 500 GB of SSD disk space. For every combination of parameters we run the experiments 10 times. Different datasets have been exploited to conduct the experiments: *Susy*, *Epsilon*, *Higgs*, and *Infimnist*. Their descriptions are reported in Table 1. The 70% of each dataset is used as $\mathcal{D}_n$ whereas the remaining 30% as $\mathcal{T}_m$.

**Table 1.** Dataset Exploited in the Paper.

| Name | Ref. | n | d | c | Size (GB) |
|---|---|---|---|---|---|
| Susy | [2] | $5 \cdot 10^6$ | 18 | 2 | 3 |
| Epsilon | [18] | $5 \cdot 10^5$ | $2 \cdot 10^3$ | 2 | 11 |
| Higgs | [2] | $11 \cdot 10^6$ | 28 | 2 | 8.4 |
| Infimnist | [13] | $14 \cdot 10^6$ | 784 | 10 | 20 |

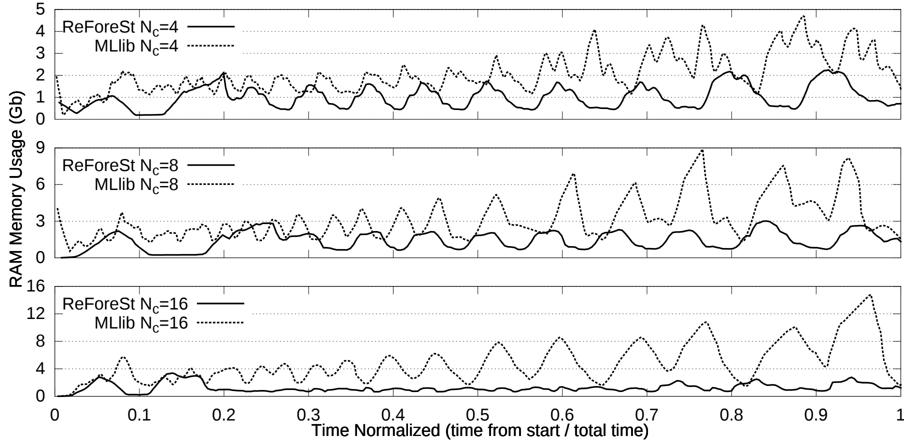


**Fig. 1.** Memory usage of MLlib and ReForeSt over the Higgs dataset with $N_m = 4$, $N_c = \{4, 8, 16\}$, $n_t = 100$, $n_b = 32$, $n_d = 10$, $b = 1$.

Figure 1 depicts the average memory consumption of ReForeSt and MLlib for the Higgs dataset. We collect the memory used by the Java Virtual Machine (JVM) in each second of computation for the environment with $N_m = 4$, $N_c = \{4, 8\}$, $n_b = 32$, $n_d = 10$ and $b = 1$. The JVM Garbage Collector has been invoked periodically in order to avoid artifacts in the results. On the x-axis we report the normalized computational time with respect to the total time in order to better compare the ReForeSt and MLlib memory usage. Figure 1 shows the memory usage picks due to the allocation of the matrices used to collect the information at each iteration of ReForeSt and MLlib. From Figure 1 it is possible to observe that:

- ReForeSt requires always less memory with respect to MLlib to perform the computation. In particular it requires always less than 3GB of RAM while MLlib requires 16GB of RAM;
- the MLlib memory usage linearly increases with the number of cores, whereas the ReForeSt memory usage does not depend on the number of cores.

Results over the other datasets present a similar behavior.

Table 2 reports a series of metrics for the complete sets of experiments run over the different cluster architectures and datasets and by changing $n_t$. These metrics are: the computation time of ReForeSt and MLlib in seconds, respectively $t_R$ and $t_M$, the speed-up of ReForeSt and MLlib with respect to the base scenario $N_m = 4$ and $N_c = 4$, respectively $S_R$ and $S_M$, and $\Delta = {}^{t_M}/_{t_R}$. $n_b$ and $n_d$ have been fixed to the default MLlib values ($n_b = 32$ and $n_d = 10$) and standard deviation is not reported because of space constraints. However changing $n_b$ and $n_d$ does not substantially change the outcomes and the standard deviation of the results is always less than 5%. Based on Table 2 it is possible to observe that:

- from a computational point of view, ReForeSt is much more efficient than MLlib. We obtain a speed-up with respect to MLlib of at least $\Delta = 1.48$ with a maximum of $\Delta = 3.05$. On average, ReForeSt is two times faster than MLlib;

**Table 2.** Comparison between the computational time of MLlib and ReForeSt. (∗) JVM out of memory error, (-) value cannot be computed because of the JVM error.

| $N_m$ | $N_c$ | $n_t$ | Susy (3GB) | | | | | Higgs (8GB) | | | | | Epsilon (11GB) | | | | | Infimnist (20GB) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $t_R$ | $t_M$ | $S_R$ | $S_M$ | $\Delta$ | $t_R$ | $t_M$ | $S_R$ | $S_M$ | $\Delta$ | $t_R$ | $t_M$ | $S_R$ | $S_M$ | $\Delta$ | $t_R$ | $t_M$ | $S_R$ | $S_M$ | $\Delta$ |
| 4 | 4 | 100 | 178 | 287 | 1 | 1 | 1.61 | 399 | 685 | 1 | 1 | 1.72 | 197 | 348 | 1 | 1 | 1.77 | 1064 | ∗ | 1 | 1 | - |
| | | 200 | 319 | 570 | 1 | 1 | 1.79 | 756 | 1306 | 1 | 1 | 1.73 | 264 | 543 | 1 | 1 | 2.06 | 1847 | ∗ | 1 | 1 | - |
| | | 400 | 665 | 1163 | 1 | 1 | 1.75 | 1598 | ∗ | 1 | 1 | - | 400 | 959 | 1 | 1 | 2.4 | 3635 | ∗ | 1 | 1 | - |
| 4 | 8 | 100 | 104 | 173 | 1.7 | 1.7 | 1.66 | 223 | 400 | **1.8** | 1.7 | 1.79 | 123 | 192 | 1.6 | **1.8** | 1.56 | 581 | 1106 | 1.8 | - | 1.9 |
| | | 200 | 175 | 357 | **1.8** | 1.6 | 2.04 | 429 | 788 | **1.8** | 1.7 | 1.84 | 155 | 304 | 1.7 | **1.8** | 1.96 | 1013 | 2095 | 1.8 | - | 2.07 |
| | | 400 | 370 | 688 | **1.8** | 1.7 | 1.86 | 903 | 1662 | **1.8** | - | 1.84 | 246 | 543 | 1.6 | **1.8** | 2.21 | 1935 | ∗ | 1.9 | - | - |
| 4 | 16 | 100 | 78 | 138 | **2.3** | 2.1 | 1.77 | 142 | 275 | **2.8** | 2.5 | 1.94 | 94 | 172 | **2.1** | 2 | 1.83 | 380 | 799 | 2.8 | 2.8 | 2.1 |
| | | 200 | 118 | 280 | **2.7** | 2 | 2.37 | 617 | **3** | 2.1 | 2.47 | 654 | 107 | 285 | **2.5** | 1.9 | 2.66 | 250 | 1460 | 2.8 | 2.9 | 2.23 |
| | | 400 | 226 | 608 | **2.9** | 1.9 | 2.69 | 521 | 1256 | **3.1** | 2.6 | 2.41 | 172 | 519 | **2.3** | 1.8 | 3.02 | 1284 | 2842 | 2.8 | - | 2.21 |
| 8 | 4 | 100 | 107 | 160 | 1.7 | **1.8** | 1.5 | 238 | 408 | 1.7 | 1.7 | 1.71 | 122 | 192 | 1.6 | **1.8** | 1.57 | 659 | 1078 | 1.6 | 2.1 | 1.64 |
| | | 200 | 182 | 317 | 1.8 | 1.8 | 1.74 | 437 | 752 | 1.7 | 1.7 | 1.72 | 168 | 303 | 1.6 | **1.8** | 1.8 | 1040 | ∗ | 1.8 | - | - |
| | | 400 | 365 | 619 | 1.8 | **1.9** | 1.7 | 930 | 1530 | 1.7 | 2.2 | 1.65 | 257 | 526 | 1.6 | **1.8** | 2.05 | 1912 | ∗ | 1.9 | - | - |
| 8 | 8 | 100 | 65 | 97 | 2.7 | **3** | 1.49 | 133 | 239 | **3** | 2.9 | 1.8 | 78 | 151 | **2.5** | 2.3 | 1.94 | 354 | 639 | 3 | 3.5 | 1.81 |
| | | 200 | 106 | 213 | **3** | 2.7 | 2.01 | 249 | 452 | **3** | 2.9 | 1.82 | 100 | 256 | **2.6** | 2.1 | 2.56 | 586 | 1170 | 3.2 | 3.6 | 2 |
| | | 400 | 210 | 410 | **3.2** | 2.8 | 1.95 | 478 | 897 | 3.3 | 3.7 | 1.88 | 162 | 458 | **2.5** | 2.1 | 2.83 | 1164 | 2235 | 3.1 | 5.1 | 1.92 |
| 8 | 16 | 100 | 54 | 80 | 3.3 | **3.6** | 1.48 | 78 | 168 | **5.1** | 4.1 | 2.15 | 66 | 128 | **3** | 2.7 | 1.94 | 234 | 513 | **4.5** | 4.3 | 2.19 |
| | | 200 | 72 | 162 | **4.4** | 3.5 | 2.25 | 140 | 328 | **5.4** | 4 | 2.34 | 78 | 194 | **3.4** | 2.8 | 2.49 | 392 | 963 | **4.7** | 4.4 | 2.46 |
| | | 400 | 139 | 375 | **4.8** | 3.1 | 2.7 | 275 | 722 | **5.8** | 4.6 | 2.63 | 129 | 371 | **3.1** | 2.6 | 2.88 | 776 | 1927 | 4.7 | 5.9 | 2.48 |
| 16 | 4 | 100 | 67 | 102 | 2.7 | **2.8** | 1.52 | 135 | 222 | 3 | **3.1** | 1.64 | 83 | 162 | **2.4** | 2.1 | 1.95 | 364 | 629 | 2.9 | 3.5 | 1.73 |
| | | 200 | 109 | 193 | 2.9 | **3** | 1.77 | 252 | 423 | 3 | **3.1** | 1.68 | 109 | 265 | **2.4** | 2 | 2.43 | 610 | 1138 | 3 | 3.7 | 1.87 |
| | | 400 | 223 | 381 | 3 | **3.1** | 1.71 | 512 | 860 | 3.1 | 3.9 | 1.68 | 174 | 488 | **2.3** | 2 | 2.8 | 1167 | 2177 | 3.1 | 5.2 | 1.87 |
| 16 | 8 | 100 | 47 | 80 | **3.8** | 3.6 | 1.7 | 74 | 135 | **5.4** | 5.1 | 1.82 | 57 | 115 | **3.5** | 3 | 2.02 | 216 | 435 | 4.9 | 5.1 | 2.01 |
| | | 200 | 79 | 175 | **4** | 3.3 | 2.22 | 150 | 263 | 5 | 5 | 1.75 | 73 | 185 | **3.6** | 2.9 | 2.53 | 381 | 823 | 4.8 | 5.1 | 2.16 |
| | | 400 | 130 | 270 | **5.1** | 4.3 | 2.08 | 261 | 519 | 6.1 | 6.4 | 1.99 | 118 | 342 | **3.4** | 2.8 | 2.9 | 686 | 1477 | 5.3 | 7.7 | 2.15 |
| 16 | 16 | 100 | 44 | 64 | 4 | **4.5** | 1.45 | 50 | 101 | **8** | 6.8 | 2.02 | 53 | 110 | **3.7** | 3.2 | 2.08 | 132 | 339 | **8.1** | 6.5 | 2.57 |
| | | 200 | 50 | 127 | **6.4** | 4.5 | 2.54 | 89 | 212 | **8.5** | 6.2 | 2.38 | 62 | 179 | **4.3** | 3 | 2.89 | 230 | 636 | **8** | 6.6 | 2.77 |
| | | 400 | 102 | 250 | **6.5** | 4.7 | 2.45 | 172 | 441 | **9.3** | 7.5 | 2.56 | 104 | 317 | **3.8** | 3 | 3.05 | 455 | 1266 | 8 | 9 | 2.78 |

- ReForeSt scales better with $n_t$ with respect to MLlib as one can observe trough the values of $\Delta$. For instance, on the Epsilon dataset when $N_m = 16$ and $N_c = 16$ we obtain $\Delta = 2.08$ with $n_t = 100$ and $\Delta = 3.05$ with $n_t = 400$;
- ReForeSt scales better with $N_c$. This effect is easier to observe when the size of the dataset is larger. For instance, in the Infimnist dataset with $N_m = 16$ and $n_t = 400$ we obtain $\Delta = 1.87, 2.15, 2.78$ respectively for $N_c = 4, 8, 16$;
- ReForeSt exhibits comparable or better speed-up, when $N_m$ or $N_c$ are increased, with respect to MLlib;
- ReForeSt requires considerable less memory with respect to MLlib; In fact MLlib is not able to finish the computation on several clusters because of memory constraints. For instance, with $N_m = 4$ and $N_c = 4$ we got the following error "There is insufficient memory for the Java Runtime Environment to continue" since MLlib fails to provide a solution for the Infimnist dataset and for Higgs with $n_t = 400$.

Finally, we do not include in the table the accuracies since the differences between ReForeSt and MLlib are not statistically relevant. For instance, for the Infimnist dataset with $n_t = 400$, with ReForeSt we obtain an error of $\widehat{L}_R(f) = 0.082 \pm .001$ while with MLlib we obtain an error of $\widehat{L}_M(f) = 0.083 \pm .001$. With the Higgs dataset the errors are $\widehat{L}_R(f) = 0.029 \pm .001$ and $\widehat{L}_M(f) = 0.028 \pm .001$.

## 4 Conclusion

In this work we developed ReForeSt, an Apache Spark implementation of the RF learning algorithm that we made publicly available through GitHub. ReForeSt is easier to tune, faster, and less memory consuming with respect to MLlib, the de-

facto standard Apache Spark machine learning library. An extensive comparison between ReForeSt and MLlib performed by taking advantage of the GCP and different big data problems confirms the quality of the proposal.

As future works, we plan to further develop ReForeSt by introducing the possibility to conclude the construction of the sub-tree on a single machine when the cardinality of the data relative to a sub-tree is below a certain threshold similarly to [6] and we will take care to develop an ad-hoc computationally inexpensive model selection strategy for the purpose of automatically tuning the hyperparameters of the RF over the available data.

## References


1. Anguita, D., Ghio, A., Oneto, L., Ridella, S.: In-sample and out-of-sample model selection and error estimation for support vector machines. IEEE Transactions on Neural Networks and Learning Systems 23(9), 1390–1406 (2012)
2. Baldi, P., Sadowski, P., Whiteson, D.: Searching for exotic particles in high-energy physics with deep learning. Nature communications 5(4308), 1–9 (2014)
3. Bishop, B.M.: Neural networks for pattern recognition. Oxford university press (1995)
4. Breiman, L.: Random forests. Machine learning 45(1), 5–32 (2001)
5. Chen, J., et al.: A parallel random forest algorithm for big data in a spark cloud computing environment. IEEE Transactions on Parallel and Distributed Systems pp. (in–press) (2016)
6. Chung, S.: Sequoia forest : Random forest of humongous trees. In: Spark Summit (2014)
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
8. Efron, B.: Bootstrap methods: Another look at the jackknife. The Annals of Statistics 7(1), 1–26 (1979)
9. Fernández-Delgado, M., Cernadas, E., Barro, S., Amorim, D.: Do we need hundreds of classifiers to solve real world classification problems? JMLR 15(1), 3133–3181 (2014)
10. Genuer, R., Poggi, J., Tuleau-Malot, C., Villa-Vialaneix, N.: Random forests for big data. In: arXiv preprint arXiv:1511.08327 (2015)
11. Germain, P., Lacasse, A., Laviolette, A. ahd Marchand, M., F., R.J.: Risk bounds for the majority vote: From a pac-bayesian analysis to a learning algorithm. JMLR 16(4), 787–860 (2015)
12. Hernández-Lobato, D., Martínez-Muñoz, G., Suárez, A.: How large should ensembles of classifiers be? Pattern Recognition 46(5), 1323–1336 (2013)
13. Loosli, G., Canu, S., Bottou, L.: Training invariant support vector machines using selective sampling. In: Large Scale Kernel Machines (2007)
14. Meng, X., et al.: Mllib: Machine learning in apache spark. Journal of Machine Learning Research 17(34), 1–7 (2016)
15. Vapnik, V.N.: Statistical learning theory. Wiley New York (1998)
16. Wainberg, M., Alipanahi, B., Frey, B.J.: Are random forests truly the best classifiers? Journal of Machine Learning Research 17(110), 1–5 (2016)
17. Wakayama, R., et al.: Distributed forests for mapreduce-based machine learning. In: Asian Conference on Pattern Recognition (2015)
18. Yuan, G., Ho, C., Lin, C.: An improved glmnet for l1-regularized logistic regression. Journal of Machine Learning Research 13, 1999–2030 (2012)
19. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. HotCloud 10(10-10), 1–9 (2010)