Information Fusion 64 (2020) 205–237

Contents lists available at ScienceDirect

Information Fusion

journal homepage: www.elsevier.com/locate/inffus

Full length article

A practical tutorial on bagging and boosting based ensembles for machine learning: Algorithms, software tools, performance study, practical perspectives and opportunities

Sergio González [a,*], Salvador García [a], Javier Del Ser [b,c], Lior Rokach [d], Francisco Herrera [a]

[a] Andalusian Research Institute on Data Science and Computational Intelligence (DaSCI), University of Granada, 18071 Granada, Spain
[b] TECNALIA, Basque Research and Technology Alliance (BRTA), 48160 Derio, Spain
[c] University of the Basque Country (UPV/EHU), 48013 Bilbao, Spain
[d] Ben-Gurion University of the Negev, P.O.B. 653, Beer-Sheva 8410501, Israel

ARTICLE INFO

Keywords:
Decision trees
Ensemble learning
Classification
Machine learning
Software

ABSTRACT

Ensembles, especially ensembles of decision trees, are one of the most popular and successful techniques in machine learning. Recently, the number of ensemble-based proposals has grown steadily. Therefore, it is necessary to identify which are the appropriate algorithms for a certain problem. In this paper, we aim to help practitioners to choose the best ensemble technique according to their problem characteristics and their workflow. To do so, we revise the most renowned bagging and boosting algorithms and their software tools. These ensembles are described in detail within their variants and improvements available in the literature. Their online-available software tools are reviewed attending to the implemented versions and features. They are categorized according to their supported programming languages and computing paradigms. The performance of 14 different bagging and boosting based ensembles, including XGBoost, LightGBM and Random Forest, is empirically analyzed in terms of predictive capability and efficiency. This comparison is done under the same software environment with 76 different classification tasks. Their predictive capabilities are evaluated with a wide variety of scenarios, such as standard multi-class problems, scenarios with categorical features and big size data. The efficiency of these methods is analyzed with considerably large data-sets. Several practical perspectives and opportunities are also exposed for ensemble learning.

1. Introduction

An ensemble is the fusion of several trained models whose combined prediction aimed to improve the predictive performance of a single model [1–4]. Thanks to this combination of diverse predictions, ensembles have resulted in very powerful predictive techniques with great capacities for generalization without neglecting more local or specific knowledge [1,2]. Their good performance has established themselves as one of the best and most influential machine learning methods [5,6]. They have been very successful in many machine learning challenges for real-life applications [7,8] and common winners of machine learning competitions [9].

Plenty of ensemble-based proposals are among the *state-of-the-art* methods for the majority of supervised learning tasks [1,6] (classification, regression, and ranking), and they have also succeeded at other learning tasks, such as preprocessing problems [10–12], data stream learning [13], anomaly detection [14,15].

In recent years, a large amount of ensemble algorithms has been proposed with a variety of built-in prediction fusions, different base learners and various mechanisms for diversity promotion over several applied problems [1,16]. However, bagging [17] and boosting [18] techniques have drawn greater attention to machine learning researchers [1,4].

With the growing popularity and proliferation of new proposals, studies explaining and empirically comparing the best performing ensembles are needed. During these past years, several overviews and studies have been published involving ensemble learning. Rokach [19] presents a tutorial that gives practical knowledge about the ensemble methodology and some classification ensemble-based techniques. An extensive experimental study [6] with 179 different classifiers reports that the Random Forest based ensemble family is the best classifier, and that boosting ensembles are among the best. Two years later, an article [20] questions this conclusion due to a biased experimental

* Corresponding author.
E-mail addresses: sergiogvz@decsai.ugr.es (S. González), salvagl@decsai.ugr.es (S. García), javier.delser@tecnalia.com (J. Del Ser), liorrk@bgu.ac.il (L. Rokach), herrera@decsai.ugr.es (F. Herrera).

https://doi.org/10.1016/j.inffus.2020.07.007
Received 23 May 2020; Received in revised form 19 July 2020; Accepted 20 July 2020
Available online 28 July 2020

study and no significant difference with other methods. Some popular decision forest methods have been reviewed [3]. Ren et al. [16] review the recent developments of ensemble classification and regression. The success of AdaBoost and Random Forest is connected to the behavior of their interpolating classifiers [21]. The authors also suggest training boosting methods in the same way as random forests. General ensemble learning is well reviewed by Sagi et al. [22]. Among the existing books of ensemble learning [1,2,4,15], the most recent book [4] is worth to be remarked for the description of the *state-of-the-art* of ensemble learning.

Some of these overviews lack the most recent ensembles, detailed algorithmic descriptions or extensive empirical comparisons. In addition, none of them assesses the online available software for ensemble techniques, which would help machine learning practitioners design their research and implement their production workflow faster and better suited to the needs of the problem at hand.

In this paper, we aim to fill this gap and also to go ahead with different technical discussions on ensembles. The main five contributions and objectives achieved in our study are the following ones:

- To present an overview of the most recent and iconic bagging and boosting based algorithms. We focus on 14 ensemble algorithms, 11 of which are chosen due to their popularity in the literature and their inclusion in well-known machine learning software packages. In addition, we would like to highlight three more ensemble variants, which are considerably different from their original approaches. Among these 14 ensemble algorithms, there are two traditional boosting methods: AdaBoost [23,24] and LogitBoost [25]; three forest-like ensembles: Random Forest [26], Extra-Trees [27] and Random Patches [28]; three Rotation feature space algorithms: Rotation Forest [29], Random Rotation Forest [30] and Random Rotation Extra-Trees [30]; and six gradient boosting machines: traditional GBM [31], XGBoost [32], XGBoost with DART [33], LightGBM [34], LightGBM with GOSS [34] and CatBoost [35,36]. We consider Random Rotation Forest and Random Rotation Extra-Trees as variants of Random Rotation Ensembles, and XGBoost with DART and LightGBM with GOSS as variants of XGBoost and LightGBM, respectively.
- To introduce the software tools available online of the presented bagging and boosting algorithms. We prioritize trustworthy packages with a certain positive impact on the machine learning field and coded with popular programming languages: Python, R, Scala, and Java. Each software is analyzed by highlighting the implemented ensemble versions, their general features, and other special supports, such as categorical feature support. Software packages are also categorized by the supported computing paradigms: Sequential CPU computing, Parallel CPU computing, Parallel GPU computing, and Distributed computing.
- To perform a practical study on the performance of the 14 ensembles in terms of predictive capability and efficiency, getting some lessons learned from the empirical analysis. For this purpose, 76 different standard classification data-sets are used in the experimental framework, covering a wide variety of scenarios, such as multi-class problems, large data, and scenarios with categorical features. Seeking a fair comparison, the same family of base learners is used for all methods (Scikit-learn's CART-based decision tree [3,37,38]) and the software packages are coded with the same programming language and under the same machine learning framework (Python's Scikit-learn [39,40]). Furthermore, we choose for each ensemble learning algorithm the default hyperparameter values set by the corresponding software package for classification tasks. Our purpose to proceed in this way is to provide a general analysis of the performance that one could expect from the off-the-shelf implementation of the different ensembles under analysis.
  Nevertheless, in line with recent critical studies on the convenience of hyperparameter tuning [41], we complement our general analysis with a closer look at the performance gains that emerge from the optimization of hyperparameters.
- To discuss some perspectives on ensemble learning: ensemble diversity, ensembles for data preprocessing, ensemble fusion, multi-class decomposition techniques, and a short discussion on the relevance of hyperparameter tuning for ensembles.
- To discuss some opportunities and potential future fields for ensembles in machine learning focused on the design of ensembles for Big Data, the design of explainable artificial intelligence approaches for ensembles, the study of the robustness and stability of ensembles, and meta-learning techniques for ensemble learning.

The paper is organized as follows. In Section 2 we describe in detail the reviewed ensemble methods with their recent improvements. Section 3 reviews the online-available software packages according to their different features. The experimental framework used in the empirical studies is presented in Section 4. In Section 5, the performance of the featured ensembles is empirically analyzed on classification data-sets according to different behaviors: binary and multi-class learning, learning with categorical features and learning performance and efficiency with big sized data-sets. Section 6 exposes all the lessons learned from the revision done in the paper. In Section 7, we present several practical perspectives for ensemble learning. Section 8 outlines a few opportunities for ensemble techniques. Finally, the study ends in Section 9, which summarizes the main conclusions derived from this work. Training and test runtimes for standard-size data-sets are included in Appendix A, whereas Appendix B collects the detailed results of the experiments related to hyperparameter tuning. Given the large number of acronyms used in the paper, we have added a list of abbreviations in Appendix C.

2. Algorithmic description of bagging and boosting based ensembles

In this section we describe the aforementioned set of 11 bagging and boosting ensembles and the 3 ensemble variants considered in our study. Through Sections 2.2 to 2.12, we delve into their main algorithmic procedures, and explain their limitations in detail. Additionally, their latest improvements and variants designed to deal with such limitations are also described. Some background knowledge and notations are first needed before describing each of the ensembles under consideration. This is the purpose of the opening Section 2.1, which follows below.

2.1. Background and notations

The ensembles discussed in this section fit into the following categorization [1,2,4]:

- **Independent base learner training or bagging-like ensembles** [17] train their base learners independently from each other, and they use data transformations to promote diversity into the predictions of the model. These techniques usually use bootstrapping, i.e. data resampling with replacement. Random Forest [26,42], Random Patches [28] and Extra-Trees [27] independently train their base learners and therefore, belong to this category.
- **Iterative/dependent base learner training or boosting algorithms** [18] learn from the errors of previous iterations by increasing the importance of those wrongly predicted training instances in future iterations. AdaBoost [23,24] or Gradient Boosting Machine [31] are examples of boosting ensembles.

Fig. 1 represents graphically the workflows of bagging and boosting based algorithms. Further to this categorization, another group called label switching could be considered. Label switching methods [11, 43,44] exchange the labels of a small number of randomly selected instances as a promotion of diversity within the ensemble. Although they could be interesting for the study, we have not found any software

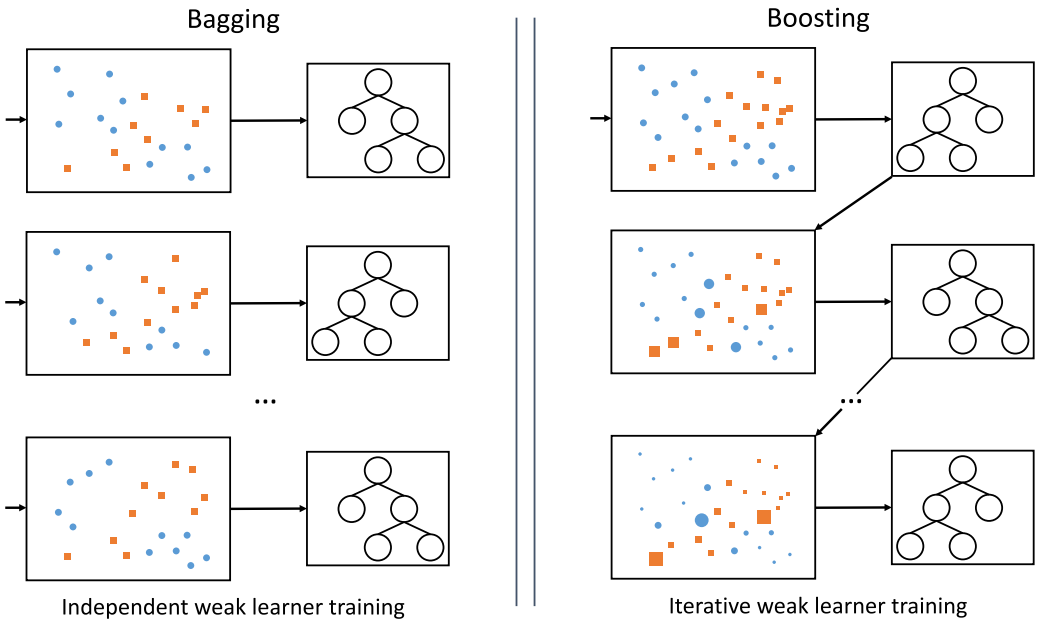

Fig. 1. Representation of the workflows of Bagging and Boosting strategies.

that includes their main proposals. Thus, it falls away from the scope on reviewing ensembles with a high presence in known software tools.

These ensemble techniques are mainly used in supervised learning tasks, such as classification, regression, and ranking. These learning tasks aim to learn a model $C(x)$ from a training set $D = \{(x_i, y_i)\}_{i=1}^{N}$, where x_i are the input variables and y_i the target variable or outputs, to predict future values y by providing the inputs x.

Generally, we use the following math notation to refer to some common algorithmic aspect of the ensembles. N and L stand for the number of instances and classes of the data-set. T is the set of base learners of the ensemble. t marks each of the iterations. Therefore, T_t means the estimator of iteration t. ϵ denotes the error made by previous estimators. We use w as the weights given to training instances by boosting methods.

2.2. AdaBoost: Adaptive Boosting

AdaBoost is one of the classic and most emblematic boosting methods. Many variants of AdaBoost have been developed with different characteristics and purposes [23,24,45,46]. In each iteration of every AdaBoost version, a base learner is trained with the weights of the training examples updated according to the performance of the previous iterations. The variants of the AdaBoost method are determined by how these weights are updated and which instances are affected.

The original AdaBoost algorithm [23] was initially designed for binary classification problems with base classifiers predicting the probability of a target class. In this method, the weight w_i of each x_i instance is adjusted proportionally to its probability of being correctly predicted and indirectly proportional to the error ϵ_t of the classifier T_t. In addition, each classifier decision on the final prediction of a new example is also weighted in relation to its accuracy during the training phase.

Along with this method, a multi-class variant, called AdaBoost.M1, was proposed in [23]. Algorithm 1 shows the pseudo-code of Adaboost.M1. In this version, only the weights of the correctly predicted instances are decreased, as shown in Line 8. This decrease is still made in relation to the error made by the base learner ϵ_t (Line 5). The predictions of each classifier are still weighted by its accuracy, as seen in Line 6 and Line 15.

In order to perform properly, both approaches assume that the error ϵ_t of each base learner is less than 0.5. Otherwise, the correctly predicted examples are the ones that are boosted, instead of the misclassified ones. Furthermore, the contribution of each classifier to the final decision becomes negative ($\alpha_t < 0$). The upper limitation of ϵ_t to 0.5 is possible, which entails keeping the weights unchanged.

The restriction $\epsilon_t < 0.5$ is hard to achieve in multi-class scenarios. For this reason, different multi-class versions of AdaBoost have been developed in an attempt to alleviate this problem [24,45–47]. Although not solving the problem, AdaBoost.M2 algorithm [23] is able to incorporate the probabilities given by base learners for all classes in the manipulation of weights. AdaBoost.MH [45] approach combines AdaBoost and a strategy of binary class decomposition with minimization of Hamming error. MultiBoost [46] is a combination of AdaBoost and Wagging that incorporates a random reset of the weights when $\epsilon_t = 0$ or $\epsilon_t > 0.5$. Stagewise Additive Modeling — SAMME and Real SAMME (SAMME.R) extensions of AdaBoost [24] generalize the exponential loss for multi-class case by summing to the calculation of α_t (Line 5) the term $\log(L-1)$, where L is the number of classes. As a consequence, the accuracy of base learners only has to be better than random guessing, i.e. $(1-\epsilon_t) > 1/L$, instead of $1/2$, so that α_t is positive.

Algorithm 1: AdaBoost.M1 algorithm.

1: **function** AdaBoost($D = \{(x_i, y_i)\}_{i=1}^{N}$ - training set, $nEstimators$ - number of learners)
2: **initialize:** $T[1..nEstimators]$, $w_i = 1/N$
3: **for** t in 1, .., $nEstimators$ **do**
4: $T_t = Fit\_Estimator(D, w)$
5: $\epsilon_t = \sum_{x_i \in D | T_t(x_i) \neq y_i} w_i / \sum_{x_i} w_i$
6: $\alpha_t = \log \frac{1-\epsilon_t}{\epsilon_t}$
7: **for** x_i in D **do**
8: **if** $T_t(x_i) = y_i$ **then**
9: $w_i = w_i \cdot \frac{\epsilon_t}{1-\epsilon_t}$
10: **end if**
11: **end for**
12: $Normalize(w)$
13: **end for**
14: **output:**
15: $C(x) = \arg\max_l \sum_{t \in [1,nEstimators] | T_t(x)=l} \alpha_t$
16: **end function**

2.3. LogitBoost

LogitBoost uses a boosting scheme to fit an additive logistic regression [25]. This boosting approach can be seen as a greedy stage-wise procedure to optimize maximum-likelihood within a generalization of logistic regression. Logistic regression estimates the class probabilities $p_l(x_i)$ of instances x_i as:

$$p_l(x_i) = \frac{e^{F_l(x_i)}}{\sum_{j=1}^{L} e^{F_j(x_i)}} \quad (1)$$

where $F_l(x_i)$ is a regression function of class l.

Algorithm 2 explains with pseudo-code the fitting procedure of LogitBoost. Initially, no assumption is made about the class probabilities of the training samples. That is, all instances have the same probability for each class: $1/L$, where L is the number of classes. For each iteration t, a regression model $f_{t,l}$ is individually trained for each class, as shown in Lines 4 to 15.

First, the weight $w_{i,l}$ and response $z_{i,l}$ of each sample x_i are updated according their probabilities $p_l(x_i)$ for each class l. The weights $w_{i,l}$ are computed proportionally to the uncertainty of belonging to the class l with the highest value of $w_{i,l}$ in $p_l(x_i) = 0.5$. Values $z_{i,l}$ are set in relation to the discrepancies between the belongings to the class l and the estimated probabilities p_l, as represented in Lines 7 to 11. Then, the regression function $f_{t,l}$ of each class l is fitted with the training samples $(x_i, z_{i,l})$ and their weights $w_{i,l}$. Regression trees are the standard base learner in LogitBoost. Once all $f_{t,l}$ models have been trained, they are included in the additive model F_l of their corresponding class l (See Line 17).

Unlike AdaBoost, the class probabilities used to update the weights are estimated with the complete ensemble built so far, and not just with the last trained estimator. The final class of an instance x is the class l with the highest value for $F_l(x)$.

Algorithm 2: LogitBoost algorithm.

1: **function** LogitBoost($D = \{(x_i, y_i)\}_{i=1}^{N}$ - training set, $nEstimators$ - number of learners)
2: **initialize:** $w_i = 1/N$, $p_l(x_i) = 1/L$
3: **for** t in [1,$nEstimators$] **do**
4: **for** l in [1, L] **do**
5: **for** x_i in D **do**
6: $w_{i,l} = p_l(x_i)(1 - p_l(x_i))$
7: **if** $y_i = l$ **then**
8: $z_{i,l} = 1/p_l(x_i)$
9: **else**
10: $z_{i,l} = -1/(1 - p_l(x_i))$
11: **end if**
12: **end for**
13: // Fit $f_{t,l}$ by a weighted regression of $z_{i,l}$ to x_i with $w_{i,l}$
14: $f_{t,l}(x) = Fit\_Weighted\_Regression(D, z_l, w_l)$
15: **end for**
16: $f_{t,l}(x) = \frac{L-1}{L}(f_{t,l}(x) - \frac{1}{L}\sum_{j=1}^{L} f_{j,l}(x))$
17: $F_l(x) = F_{l-1}(x) + f_{t,l}(x)$
18: // Update probabilities $p_l(x_i)$
19: $p_l(x_i) = e^{F_l(x_i)} / \sum_{j=1}^{L} e^{F_j(x_i)}$
20: **end for**
21: **output:**
22: $C(x) = \arg\max_l F_l(x),\ l = 1, ..., L$
23: **end function**

LogitBoost has been well received both by industry and the scientific community, but to a lesser extent compared to other algorithms such as AdaBoost or Gradient Boosting. In recent years, two new versions of LogitBoost were developed [48], Robust LogitBoost and ABC-LogitBoost, seeking greater numerical stability in their implementations. Robust LogitBoost adapts the formulation of the regression tree and the split criterion to logistic regression [48]. ABC-LogistBoost [48] formulates the boosting algorithm using a base class, which is chosen in relation to training loss of each iteration. Additionally, AOSO- and AOSA-LogitBoost [49,50] were designed as Adaptive One-vs-One (AOSO) and One-vs-All (AOSA) alternatives of LogitBoost for multi-class classification.

2.4. Random Forest

Random Forest (RF) [26,42] is possibly one of the most famous ensembles, probably because of its effectiveness and simplicity. RF is considered in the top 10 best classifiers of the literature [6].

Algorithm 3 presents a brief pseudo-code of the Random Forest method. Random Forest is the flagship method of decision tree Bagging. As such, RF promotes diversity among the different trees of the forest through bootstrap sampling. That is, each decision tree is trained with a different training set resulting from sampling with replacement of the original data-set (Line 4).

However, RF includes an additional boost of diversity during the growth of each tree. While selecting the best cut for tree branching, only a subset of m features are considered (Line 6). Typically, the number of considered features m is recommended to be the square root of the total set of variables. Additionally, the trees are grown to the maximum possible depth without pruning. The final prediction of a sample is obtained by majority voting among the decisions taken by the different trees, as shown in Line 20.

Algorithm 3: Random Forest algorithm.

1: **function** Random Forest($D = \{(x_i, y_i)\}_{i=1}^{N}$ - training set, $nEstimators$ - number of trees built, m - maximum number of features to consider as possible split)
2: **initialize:** $T[1..nEstimators]$
3: **for** t in [1,$nEstimators$] **do**
4: // Random selection of N samples with replacement
5: $D_t = Bootstrap\_Sampler(D)$
6: // Build considering a subset of only m features
7: $T_t = Fit\_Tree(D_t, m, pruning = False, maxDepth = None)${
8: **function** Fit_Tree($\hat{D}, m$)
9: **for** $!Stop\_Split()$ **do**
10: $a_1, ..., a_m = Pick\_Random\_Attributes(m)$
11: **for** a_j in $a_1, ..., a_m$ **do**
12: $s_j = Pick\_Best\_Split(\hat{D}, a_j)$
13: **end for**
14: $a, s = \arg\max_j Score(\hat{D}, a_j, s_j)$
15: $Split\_Tree(\hat{D}, a, s)$
16: **end for**
17: **end function**
18: }
19: **end for**
20: **output:** // Output as majority voting
21: $C(x) = \arg\max_l \sum_{t\in[1,nEstimators]}[T_t(x) = l]$
22: **end function**

2.5. Gradient Boosting Machine

Together with RF, Gradient Boosting Machine (GBM) [31] is one of the most reputed methods of the past few years. Recently, this method has inspired the design of multiple remarkable ensembles that we review afterward.

Gradient Boosting Machine is a stage-wise additive model, in which its iterations can be seen as the steepest descent minimization regarded a given loss function. The predictive function is estimated through numerical optimization in the function space. GBM is designed to use both lineal regression or decision tree as its base learner, but practitioners commonly choose Gradient Boosting Decision Trees alternative (GBDT). Algorithm 4 presents the general learning procedure of Gradient Boosting in pseudo-code.

GBM starts with a constant initial guess $F_0(x)$ for the output of all the training samples. As seen in line 2, this constant is set to minimize the loss function for all known samples. At the beginning of each iteration, the errors or, more precisely, the pseudo-residuals r_i of the training samples x_i are computed with relation to the previously estimated additive function. For the first boosting step, these are calculated with the initial prediction. As shown in line 7, these residuals r_i are the results of the partial derivative of the loss function with respect to function $F_{t-1}(x)$. This derivative is the gradient that GB is named after.

Then, a regression tree is fitted with samples x_i and the residuals r_i as their outputs. That is, it intends to predict the errors made (pseudo-residuals) regarding the previous predictions. In original GBM, trees are grown to a maximum depth fixed by the user. Once the tree is built, the terminal leaves are named and their responses are calculated to minimize the loss function for all the samples in each leaf (See line 12). These outputs are summed into the additive model weighted by a factor ν, called learning rate. The learning rate ν scales the contribution from new trees, which allows the method to slowly and gradually improve its performance. This technique helps to deal with low bias and high variance.

GBM keeps fitting trees until reaching the maximum number of estimators. The output for new measurements x is predicted by the initial guess and the sum of all scaled outputs from the trees in the ensemble.

Algorithm 4: Gradient Boosting Machine algorithm.

1: **function** GBM($D = \{(x_i, y_i)\}_{i=1}^{N}$ - training set, $nEstimators$ - number of learners, $\mathcal{L}(y, F(x))$ - loss function, ν - learning rate)
2: **initialize:** $F_0(x) = \text{argmin}_\gamma \sum_{i=1}^{N} \mathcal{L}(y_i, \gamma)$
3: **for** t in [1,$nEstimators$] **do**
4: // Compute pseudo-residual $r_{i,m}$ for each sample x_i
5: **for** i in [1,N] **do**
6: $r_i = -\left[\frac{\partial \mathcal{L}(y, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F(x)_{t-1}}$
7: **end for**
8: $T_t = Fit\_Regression\_Tree(D, r)$
9: **for** i in [1,J] **do**
10: $R_j = Name\_Leaves(T_t)$
11: // Compute output γ_j of each leaf j
12: $\gamma_j = \text{argmin}_\gamma \sum_{x_i \in R_j} \mathcal{L}(y_i, F_{t-1}(x_i) + \gamma)$
13: **end for**
14: $F_t(x) = F_{t-1}(x) + \nu \sum_{j=1}^{J} \gamma_j I(x \in R_j)$
15: **end for**
16: **output:** $F(x)$
17: **end function**

One of the strengths of GBM is its adaptability to any regression or classification problem described with a derivable loss function. For classification, GBM estimates an additive logistic regression function. Therefore, negative binomial log-likelihood or multinomial loss is usually used as the loss function for binary or multi-class classification, respectively.

For multi-class problems, GBM approximates an additive function $F_l(x)$ for each class l guided by the following loss function $\mathcal{L}$:

$$\mathcal{L}(\{y_l, F_l(x)\}_1^L) = -\sum_{l=1}^{L} \log y_l p_l(x) \tag{2}$$

where L is the number of classes, y_l takes the value 1 when sample x belongs to class l or 0, otherwise and $p_l(x)$ is the probability of x for the class l. This probability $p_l(x)$ is estimated by the method as follows:

$$p_l(x) = \frac{e^{F_l(x)}}{\sum_{j=1}^{L} e^{F_j(x)}} \tag{3}$$

With Eqs. (2) and (3) and the first derivative needed, the calculation of the pseudo-residuals is straightforward as the differences of the observed and the estimated probabilities:

$$r_{i,l} = -\left[\frac{\partial \mathcal{L}(\{y_l, F_l(x)\}_1^L)}{\partial F_l(x_i)}\right]_{\{F_l(x)=F(x)_{l,t-1}\}_1^L} = y_{i,l} - p_{l,t-1}(x_i) \tag{4}$$

Then, the regression trees are trained with the pseudo-residuals of probabilities. Due to its complexity and transformation of scales, the calculation of the leaves outputs is estimated with the following expression:

$$\gamma_{i,l} = \frac{L-1}{L} \frac{\sum_{x_i \in R_{j,l}} r_{i,l}}{\sum_{x_i \in R_{j,l}} |r_{i,l}|(1 - |r_{i,l}|)} \tag{5}$$

The results are summed to the corresponding additive function $F_l(x)$. After all iterations, the estimated functions can be used through expression (3) to predict the probabilities of new observations for all classes.

Later on, Stochastic Gradient Boosting (SGB) [51] was proven to improve the accuracy, execution speed, and ensemble diversity by including randomization during the training. This stochastic version randomly sub-samples the training set without replacement at each iteration of the ensemble.

2.6. Rotation Forest

Rotation Forest [29] (RotF) is another ensemble in which the training of the different classifiers is completely independent. Although it includes bagging mechanisms, Rotation Forest is mainly known for generating variability in the ensemble and, at the same time, promoting individual precision thanks to a feature extraction process.

To avoid the same result in the feature extraction, this is done with K disjoint subsets of variables and sampling of the training set. The feature extraction is originally done by Principal Component Analysis (PCA) [29]. Additionally, other feature engineering techniques have been considered with great performances: Nonparametric Discriminate Analysis (NDA) [52], Sparse Random Projections (SRP) and Independent Component Analysis (ICA) [53]. The ensemble uses decision trees due to their sensitivity to feature rotations.

Algorithm 5: Rotation Forest algorithm.

1: **function** ROTATION FOREST($D = \{(x_i, y_i)\}_{i=1}^{N}$ - training set, $nEstimators$ - number of trees built, K - number of features subsets)
2: **initialize:** $T[1..nEstimators]$
3: **for** t in [1,$nEstimators$] **do**
4: // Prepare rotation matrix R_t
5: $F_{t,1}, ..., F_{t,K} = Split\_Features(F, K)$
6: **for** j in [1,K] **do**
7: $D_{t,j} = Filter\_Features(D, F_{t,j})$
8: // Randomly select a subset of classes
9: $D_{t,j} = Randomly\_select\_Classes(D_{t,j})$
10: // Bootstrap sampling of size 75% of $D_{t,j}$
11: $D_{t,j} = Bootstrap\_Sampler(D_{t,j}, 75\%)$
12: $a_{t,j}^{(1)}, a_{t,j}^{(2)} ..., a_{t,j}^{(M_j)} = PCA(D_{t,j})$
13: **end for**
14: // Arrange coefficients in matrix R_t as in Eq. (6)
15: // Rearrange R_t to match the correct order of F
16: $R_t^a = Construct\_Rotation\_Matrix()$
17: $T_t = Fit\_Tree(DR_t^a)$
18: **end for**
19: **output:** // Output as greatest probability
20: $C(x) = \arg\max_l \sum_{t \in [1,nEstimators]} \frac{T_{t,l}(xR_t^a)}{nEstimators}, l = 1, ..., L$
21: **end function**

Algorithm 5 introduces with pseudo-code the data transformations to train each classifier T_t of the Rotation Forest method. First, the set of features F is randomly split into K disjointed subsets, as stated in Line

4 Then, for each $F_{t,j}$ feature subset, a subset of classes are randomly selected and 75% of the instances is chosen by bootstrap sampling to conform the $D_{t,j}$ data-set, as summarized in Lines 6 to 11. Feature extraction with PCA is performed on each resultant set.

Once all the coefficients of the feature subsets are obtained, they are arranged in the rotation matrix R_t as shown in the following expression:

$$R_t = \begin{bmatrix} a_{t,1}^{(1)}, a_{t,1}^{(2)}..., a_{t,1}^{(M_1)}, & [0] & ... & [0] \\ [0] & a_{t,2}^{(1)}, a_{t,2}^{(2)}..., a_{t,2}^{(M_2)}, & ... & [0] \\ \vdots & \vdots & \ddots & \vdots \\ [0] & [0] & ... & a_{t,K}^{(1)}, a_{t,K}^{(2)}..., a_{t,K}^{(M_K)} \end{bmatrix} \tag{6}$$

This matrix is reorganized to match its columns with the original features of the data-set. The decision tree is trained with the set obtained from the product DR_t^a. The final label for an example x is given as the class with the greatest probability.

2.7. Extremely randomized trees

Extremely randomized trees or Extra-Trees (ET) [27] is a tree-based randomization ensemble such as RF [26] and Rotation Forest [29]. In this case, an extremely random selection of the splitting cut-point of each tree is used to promote diversity and reduce the variance of the final ensemble. Algorithm 6 gives a pseudo-code of the Extra-Trees method and its randomization procedure.

Similar to RF, a random subset of only m features are considered as possible candidates in each splitting procedure, as seen in Line 7. However, in contrast to RF, only one possible cut-point is randomly chosen for each of these variables (Line 9). Then, the best cut is selected among those randomly drawn in terms of the measure of impurity to branch the tree (Line 11).

This extremely random splitting procedure is the only random resource of the ensemble. However, it is sufficient to further reduce the variance of the model in exchange for slightly increasing the bias. Therefore, trees are recommended to be trained with the original and complete data-set to reduce bias. Additionally, trees are grown to the maximum extent possible. A final prediction of the ensemble is given by the majority voting of the predictions of the different trees.

Algorithm 6: Extra-Trees algorithm.

1: **function** EXTRA-TREES($D = \{(x_i, y_i)\}_{i=1}^N$ - training set, $nEstimators$ - number of trees built, m - maximum number of features to consider as possible split)
2: **initialize:** $T[1..nEstimators]$
3: **for** t in [1,$nEstimators$] **do**
4: $T_t = Fit\_Extra\_Tree(\hat{D}, m, pruning = False)$){
5: **function** FIT_EXTRA_TREE(D, m)
6: **for** !$Stop\_Split()$ **do**
7: $a_1, ..., a_m = Pick\_Random\_Attributes(m)$
8: **for** a_j in $a_1, ..., a_m$ **do**
9: $s_j = Pick\_Random\_Split(\hat{D}, a_j)$
10: **end for**
11: $a, s = \arg\max_j Score(\hat{D}, a_j, s_j)$
12: $Split\_Tree(\hat{D}, a, s)$
13: **end for**
14: **end function**
15: }
16: **end for**
17: **output:** // Output as majority voting
18: $C(x) = \arg\max_l \sum_{t\in[1,nEstimators]}[T_t(x) = l]$
19: **end function**

2.8. Random Patches

Despite its similarity to methods such as RF [26], Random Subspace or Pasting, Random Patches [28] deserve to be studied in this paper.

Random Patches (RP) are considered as a generalization of Pasting [54] and Random Subspace [55] and as a combination of both. RP introduces diversity into the base learners by training them with different data patches, i.e. subsets of $p_s * N$ samples and $p_f * M$ randomly drawn from the original data-set. p_s and p_f are parameters to control the percentage of samples and features, respectively, chosen from the original amounts of N instances and M variables. RP predicts new examples by the vote of all independently trained estimators. Algorithm 7 presents in pseudo-code the algorithmic procedure explained above.

Algorithm 7: Random Patches algorithm.

1: **function** RANDOM PATCHES($D = \{(x_i, y_i)\}_{i=1}^N$ - training set, $nEstimators$ - number of trees built, p_s - percentage of samples considered in each iteration, p_f - percentage of feature considered in each iteration)
2: **initialize:** $T[1..nEstimators]$
3: **for** t in [1,$nEstimators$] **do**
4: // Uniformly random selection of $p_s * N$ samples and $p_f * M$ features
5: $D_t = Sampler(D, p_s * N, p_f * M)$
6: // Fit with the data patch D_t
7: $T_t = Fit\_Estimator(D_t)$
8: **end for**
9: **output:** // Output as majority voting
10: $C(x) = \arg\max_l \sum_{t\in[1,nEstimators]}[T_t(x) = l]$
11: **end function**

Although RP shares some similarities with RF, they substantially differ from each other. First, RP is not necessarily subject to decision trees as a base learner, even though standard decision trees or extra trees are recommended. For each base learner, the subset of features is globally selected prior to its training. Finally, the samples of each training subset are randomly drawn without replacement.

2.9. Random Rotation Ensembles

Random Rotation (RR) Ensembles [30] are designed to introduce extra diversity into well-known ensembles without heavily decreasing the accuracy of each classifier. This is done by applying random rotation matrices to the feature space. RR ensembles are built on forest-based methods to benefit from the sensitivity of decision trees to rotations in the feature space.

Algorithm 8 shows the construction and the use of the random rotation matrices in Random Rotation Ensembles. These rotation matrices R should be real-valued $m \times m$ orthogonal square matrix $R^T = R^{-1}$ with unit determinant $|R| = 1$, where m is the number of features. In order to randomly generate them, RR ensemble uses a indirect procedure based on Householder QR decomposition [56], as described in Lines 5 to 11. First, m^2 numbers are independently and randomly drawn to configure a $m \times m$ matrix A as shown in Line 6. Then, A matrix is factorized by Householder QR procedure in the form $A = QR$, where R is an upper triangular matrix and Q is an orthogonal square matrix. However, $|Q|$ may have a positive or negative unit. In the case of being negative (Line. 10), the sign on the first column of Q can be flipped, obtaining a proper rotation matrix.

Once the rotation matrix is obtained, the RR ensemble applies this rotation to the original data-set D and then, trains a decision tree with the rotated data-set in the same way as the combined ensemble. RR ensemble is recommended in combination with RF and ET [30].

Although RR and RotF share the incorporation of rotations into an ensemble, their purposes and their extractions are completely different.

Algorithm 8: Random Rotation Ensemble algorithm.

function RANDOM ROTATION ENSEMBLE($D = \{(x_i, y_i)\}_{i=1}^{N}$ - training set, $nEstimators$ - number of trees built, $typeEnsemble$ - type of ensemble)
  initialize: $T[1..nEstimators]$
  $D' = Variable\_Scaling(D)$
  for t in [1,$nEstimators$] **do**
    // Prepare rotation matrix R_t
    $A = Random\_Matrix(m \times m)$
    $Q, R = Householder\_Decomposition(A)$
    $R_t = Q * Diag(Sign(Diag(R)))$
    if $Det(R_t) < 0$ **then**
      $R_t[\,,0] = -R_t[\,,0]$
    end if
    // Train the tree T_t as rotation matrix R_t
    $T_t = Fit\_Tree\_As(typeEnsemble, DR_t)$
  end for
  output: // Output as majority voting
  $C(x) = \arg\max_l \sum_{t \in [1,nEstimators]} [T_t(x) = l]$
end function

RotF performs several feature extraction procedures, originally by PCA, with partial information of the data-set. Thus, a different full feature set is reconstructed for each classifier of the ensemble. This procedure is its only source of diversity in the ensemble, which is guided by the feature extraction process. In contrast, the rotations of RR ensembles are completely random to provide an extra diversity preserving the individual accuracy to an existing ensemble.

2.10. XGBoost: eXtreme Gradient Boosting

XGBoost — eXtreme Gradient Boosting [32] is definitely the most famous gradient boosting based method and probably one of the most successful machine learning algorithms. XGBoost is designed as a highly scalable and accurate tree boosting system. To do so, the following features are incorporated into traditional gradient boosting:

- Re-formulation of the objective function to incorporate a regularization term.
- Approximate split finding through weighted quantile sketch.
- Sparsity-aware split function.
- Parallel tree learning with cache-aware column blocks.
- Out-of-core computation.

Most XGBoost enhancements focus on taking full advantage of computing and memory capacities, in order to speed up the learning process to the maximum. However, XGBoost also includes important adaptations to reduce the over-fitting and to extend its use to all types of problems.

The main feature against over-fitting is its regularized model formalization. Additionally, it has other regularization techniques: shrinkage and instances subsampling; which SGB already includes them; and column subsampling, similar to RF.

XGBoost objective function includes a regularization term Ω, that controls the complexity of the model. This addition allows to learn simple and predictive models and find a good bias–variance trade-off. This objective function is defined as follows:

$$obj = \sum_i \mathcal{L}(y_i, F(x_i)) + \sum_t \Omega(f_t), \text{ where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_j^T w_j^2 \tag{7}$$

with T denoting the number of leaves of the tree f, and w the score of a leaf j of the tree f. Here tree $f(x)$ is defined as $f(x) = w_{q(x)}$, where q is the tree structure that maps a sample x to its corresponding leaf. The regularization parameter λ is similar to Ridge regression L2 regularization [57], which reduce variance by making the prediction less sensitive to training data. The parameter γ works as a threshold of the score function improvement to continue splitting the tree.

As an additive model, XGBoost is greedily trained by the minimization of the objective function through the addition of a new tree:

$$obj_t = \sum_i \mathcal{L}(y_i, F_{t-1}(x_i) + f_t(x_i)) + \Omega(f_t) \tag{8}$$

Taylor's second-order approximation is used to facilitate the optimization of different loss functions. Thus, XGBoost supports customized loss functions. Then, all constant terms are removed. The expression is rewritten in terms of the score w of the tree leaves. And it is expanded with Ω resulting in the following:

$$obj_t' = \sum_j^T \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2 \right] + \gamma T \tag{9}$$

where $g_i = \partial_{F_{t-1}(x_i)} \mathcal{L}(y, F_{t-1}(x_i))$ and $h_i = \partial^2_{F_{t-1}(x_i)} \mathcal{L}(y, F_{t-1}(x_i))$ are the first and second order gradient values on the loss function. Here $I_j = \{i | q(x_i) = j\}$ represents the samples assigned to leaf j.

From this formulation, the optimal score w_j^* of leaf j for a specific tree q can be computed as:

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{10}$$

The splitting criterion also derives from (9) as the difference of the sum of the impurity scores of the left I_L and right I_R candidate leaves and the impurity score of the original leaf without further split ($I = I_L \cup I_R$).

$$obj_{\text{split}} = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma \tag{11}$$

This evaluation function is similar to the traditional impurity score of a decision tree, except for the inclusion of the regularization parameters and its suitability to a wide variety of objective functions. The splitting procedure itself has also been modified to speed up the procedure and make it efficient for big data-sets and distributed environments. Additionally to the traditional greedy algorithm for finding the best-split candidate, an approximated split finding algorithm is included. This procedure first proposes splitting candidates for each feature through a weighted quantile sketch. Then, the gradient values are aggregated for each group and a final split point is found according to the aggregated values. The splitting candidates can be proposed globally or locally after each split.

The splitting rule is also aware of the sparsity in the data-sets and treats the different sparsity types in a unified way, such as missing values or sparsity due to feature encoding. The splitting procedure defines a default branch for the missing patterns in each split. This default direction is learned from the data by scoring the two choices of the binary split.

In pursuit of greater efficiency, XGBoost stores the data in in-memory blocks in a compressed sparse column format, where the columns are sorted by the feature values. Thus, the split finding can be done by linearly scanning each pre-sorted column. The split finding is also performed collectively for all leaves, scanning only once in the collection of the gradient statistics. This process can be parallelized, leading to a parallel algorithm for split finding.

However, this split finding technique causes access to non-continuous memory in the use of gradient values, which will lead to a cache miss in data-sets with many instances. XGBoost alleviates the problem by pre-buffering the required values and processing them in batches.

To make the model really scalable, XGBoost saves part of the data-set in blocks when it does not fit in the main memory. Out-of-core computation techniques, such as block compression or block sharding

into several disks, are used to speed up the disk IO computation runtime.

In this section, the explanation is mainly focused on the tree booster version of XGBoost and its features. However, the XGBoost algorithm can also be configured with linear regression estimators or as a DART booster. DART [33] designs dropout mechanisms into the Gradient Tree Boosting method to mitigate over-specialization in the last learned trees (over-fitting). These dropouts are introduced in two parts of the algorithm. First, the gradient values used for the next tree learning are computed considering just a random subset of the ensemble. Then, the new tree is scaled to fit the same order of magnitude of the unconsidered trees.

XGBoost is also designed taking into consideration some common data problems or singular supervised learning problems. As mentioned before, it is capable to handle missing values. Imbalanced classification [58] can be dealt with in-built sample weighting. Additionally, two custom loss functions [59], weighted cross-entropy function and focal loss, were proposed specifically for binary imbalanced classification. XGBoost can also enforce monotonic constraints [60] on the features by including inherited leaf coefficient limits.

2.11. LightGBM: Light Gradient Boosting Machine

Light Gradient Boosting Machine or LightGBM [34] is a gradient boosting decision tree ensemble with several algorithmic adaptations to reduce the data dimension, apart from the inclusion of system improvements to speed up the learning process. Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling are the most innovative and relevant features of LightGBM, which intends to reduce the number of samples and features, respectively.

GOSS is an under-sampling technique guided by the gradients of the training set. The samples with greater gradients (in absolute value) are usually the most undertrained and contribute more to the learning. Therefore, GOSS keeps a fixed a% of samples with the greatest gradients and randomly samples from the rest until reaching $b\% * N$. In order to maintain the data distribution, GOSS scales samples with small gradients by $\frac{1-a}{b}$. This technique effectively reduces the number of instances and hence, the learning time. The diversity of the decision trees also increases, which may improve the overall performance.

EFB is a mechanism for dealing with and taking advantage of high-dimensional, highly sparse feature spaces, which are common in real-life problems. In these scenarios, features rarely have significant values (nonzero) simultaneously. Therefore, these almost exclusive features can be safely bundled, in order to reduce the data dimension. EFB greedily computes the sub-optimal bundles of features with a tolerated degree c of collisions as in the graph coloring problem [61]. Then, offsets are added to the feature values, so they can be identified when they merge.

The split finding of LightGBM is done with a histogram-based algorithm [62,63] to reduce runtime and memory consumption during the tree growth. The continuous values of each feature are discretized into bins, which are used to build the different feature histograms. The best split points are obtained from these feature histograms. The histograms need to be updated just for one leaf (the one with the smallest amount of data) per split since the others are computed by the subtraction with its parent. The histogram-based algorithm is further optimized for sparse data by ignoring zero feature values and recording them in tables.

LightGBM uses leaf-wise or best-first tree learning [64]. That is, the leaf with the greatest loss function improvement is first expanded. With a fixed number for leaves, this approach usually obtains more accurate predictions. However, it could lead to over-fitting, which must be controlled by setting a maximum depth for the tree.

The decision tree of LightGBM supports categorical features without the use of feature encoding, such as one-hot encoding. The split finding procedure computes the optimal split of the categorical features into 2 subsets. This is done by sorting the feature histogram according to the accumulated loss values and then, fetching the best split in the resulting histogram.

In addition, LightGBM includes several optimizations in network communication and parallel/distributed learning to further speed up and scale the learning procedure. LightGBM has built-in collective communication algorithms that improve network communications during distributed learning. And, it provides three different parallel learning algorithms: feature, data and voting parallel. Its feature parallel algorithm finds the best split of each feature in independent machines. The data-parallel algorithm aims to parallelize the construction of the histograms by building local histograms and merging them. Parallel Voting Decision Trees (PV-Tree) [63] further reduces the communication cost. PV-Tree distributes the data in different machines. Then, each worker locally votes for its best k attributes. The votes are globally aggregated and the $2k$ top attributes are selected. Finally, the full histograms of these attributes are collected to find the best split.

As XGBoost, LightGBM is configurable with DART boosting algorithm [33] and supports monotonicity constraints.

2.12. CatBoost: Gradient Boosting with Categorical features

CatBoost — Unbiased Gradient Boosting with categorical features [35,36] is the most recent GBM-based algorithm with relevance in the literature. The most important qualities of CatBoost are the categorical features support and the novel ordered boosting scheme avoiding prediction shift.

CatBoost supports categorical features with different solutions. These procedures are optimized to be applied during the tree splitting instead than in a preprocessing stage. For features with a small number of categories, CatBoost incorporates them with one-hot encoding [65]. CatBoost can also transform categorical features to numerical with the number of appearances of the categories. For a more complex solution, the categories are substituted with their average target. To prevent over-fitting, the average for a sample $x_{\sigma_i,k}$ is computed with the target values of the samples before $x_{\sigma_i,k}$ in a random permutation $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_N)$ of the data-set, as seen in the following equation:

$$x_{\sigma_i,k} = \frac{\sum_{j=1}^{i-1}[x_{\sigma_i,k} = x_{\sigma_j,k}]y_{\sigma_j} + a * P}{\sum_{j=1}^{i-1}[x_{\sigma_i,k} = x_{\sigma_j,k}] + a} \tag{12}$$

where $[x_{\sigma_i,k} = x_{\sigma_j,k}]$ takes the value 1 when the condition is satisfied, P is a prior value and the parameter a weights the prior. P is set to the average for the whole data in case of a regression task or the prior probability, for classification.

These features transformations could lead to information loss of the interactions between categorical features. Therefore, CatBoost considers greedy combinations of previous combinations or features in the current state of the tree with the rest of categorical features.

Gradient Boosting methods usually suffer from over-fitting and prediction shift, since gradients are computed for the same samples used to build the model. To prevent this, CatBoost has an additional configurable boosting scheme based on the same ordering principle applied to categorical features.

This ordered boosting performs random permutation of the data-set and builds N different supporting models $(M_1, \ldots, M_N)$, where N is the number of examples. The model M_i is trained with the ith first instances of the given permutation. Therefore, the residual of the ith samples is estimated using the model M_{i-i}, which does not include this sample in its training. The tree structures are shared for all models to reduce the complexity of the algorithm.

CatBoost works with oblivious trees, also called decision tables, which use the same splitting criterion for an entire level of the tree [66]. These trees are symmetric, balanced, less prone to over-fitting and faster to learn and during the prediction stage.

Additionally, CatBoost handles missing values. For categorical features, missing values are treated as an individual category. During

the splitting procedure, the instances with missing values of a certain numerical feature are separated in their leaf.

Similar to LightGBM, Histogram-based algorithms can be used to speed up the split finding. In contrast to XGBoost and LightGBM, CatBoost cannot be configurable with a DART booster.

3. Software tools for ensembles in machine learning

This section presents the online-available software packages that implement the previously revised ensemble algorithms. For each software package, we highlight the implemented version of the ensembles, its general and special features. The reviewed software packages are also categorized by the supported computing paradigms:

- Sequential CPU computing (SC): The software runs in a single thread and core of the CPU.
- Parallel CPU computing (PC): The program runs in multiple CPU cores with multi-threading. Memory is shared by the different processes.
- Parallel GPU computing (GC): The ensemble training is accelerated with the use of GPU. Software with GPU support relies on GPU programming frameworks, such as CUDA or OpenCL.
- Distributed computing (DC): The software runs in multiple computers without shared memory. Communications are done through network. These software tools are implemented with libraries or frameworks, such as MPI, SPARK or Hadoop.

3.1. AdaBoost software

Due to its popularity, AdaBoost can be found in almost all general-purpose programming languages and on various software platforms. However, their implementations differ greatly from each other due to the myriad of variants of the original algorithm. Table 1 gathers all the available software concerning the AdaBoost algorithm organized by the computing paradigm in use.

Scikit-Learn machine learning package written in Python [39,40] includes an implementation of AdaBoost [67]. This is based on the multi-class extensions of AdaBoost, SAMME, and SAMME.R, excluding AdaBoost.MH variant. By default, the base estimator used is a Decision Stump, i.e. a decision tree of one depth. However, any Scikit-learn compatible classifier with samples weighting support can be used.

R users can find two different packages compatible with Caret environment [68]: fastAdaboost [69] and adabag [70,71]. fastAdaboost implements AdaBoost.M1 and AdaBoost-SAMME.R algorithms in C++ back-end code, which allows faster executions. The base learner used is a self-coded decision tree without parameter tuning. Despite the multi-class variants of Adaboost implemented, this software only supports binary data-sets. The CRAN package adabag includes AdaBoost.M1 and AdaBoost-SAMME for binary and multi-class classification tasks. Recursive Partitioning And Regression Tree — rpart [72], a CART [73] based decision tree, is used as base estimator. This provides greater flexibility for the control of the tree parameters.

Smile — Statistical Machine Intelligence and Learning Engine offers its implementation of multi-class AdaBoost.SAMME coded in Java and fully compatible with Scala code [75]. Smile CART is used as a base classifier.

Apart from Smile, Java software for AdaBoost can also be found in the WEKA software [78]. AdaBoost.M1 [76] is implemented to be compatible with any base learner of WEKA, even if Decision Stump is used by default.

Given the sequence nature of AdaBoost, its parallel or distributed execution is difficult to achieve. Therefore, this kind of software is very scarce. A distributed version of AdaBoost.MH can be found for Apache Spark [79] distributed computing platform coded in Java [77]. Additionally, there is a parallel version of AdaBoost implemented with Open MP in C++ and Python [74].

Table 1
Summary of available software for the AdaBoost algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
Scikit-learn AdaBoost, 2019	[67]	✓	✗	✗	✗
Parallel-AdaBoost, 2018	[74]	✗	✓	✗	✗
R					
fastAdaboost, 2016	[69]	✓	✗	✗	✗
adabag, 2018	[71]	✓	✗	✗	✗
Scala					
Smile AdaBoost, 2019	[75]	✓	✗	✗	✗
Java					
Smile AdaBoost, 2019	[75]	✓	✗	✗	✗
Weka AdaBoost, 2017	[76]	✓	✗	✗	✗
SparkBoost, 2015	[77]	✗	✗	✗	✓

Table 2
Summary of available software for the LogitBoost algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
LogitBoost in Python, 2018	[80]	✓	✗	✗	✗
R					
caTools, 2019	[81]	✓	✗	✗	✗
Rweka LogitBoost, 2019	[82]	✓	✓	✗	✗
Scala					
No software found					
Java					
Weka LogitBoost, 2018	[83]	✓	✓	✗	✗

3.2. LogitBoost software

LogitBoost does not have as much presence as GBM or AdaBoost in the software associated with machine learning. Table 2 shows its presence in different main-purpose languages usually employed by data analysts.

LogitBoost has a Python implementation for both binary and multi-class classification built on top of Scikit-learn [80]. This LogitBoost class can be configured with any regressor as its base estimator. By default, Scikit-learn decision trees with 1 level of depth are used. A minimum weight threshold can be given as a percentile of the sample weight distribution. The samples with weights below this threshold are not considered for the following iterations. A maximum limit of the estimator outputs is supported as a technique for the numerical stability of the model. Weighted bootstrap and shrinkage regularization are also supported.

The package *caTools* in CRAN offers a simple function to fit a LogitBoost model with decision stumps as base learners in R programming language [81]. It does not provide any other parameter than the number of iterations.

Java users can rely on Weka implementation of LogitBoost for binary and multi-class classification [83]. By default, Weka LogitBoost uses decision stumps as base estimation, but any other Weka regressor is supported. Its Java class has a multitude of parameters, including those mentioned above in the Python package. Additionally, estimated prior probabilities can be set instead of uniform ones. The possibility of multi-threading should be highlighted. These functionalities can be accessed in R by using the RWeka interface of LogitBoost [82,84].

Regarding the multi-class variants of LogitBoost, only AOSO-LogitBoost [49] is available in C++ code with interfaces in Matlab [85], which supports multi-threaded learning.

Table 3
Summary of available software for the Random Forest algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
Scikit-learn Random Forest, 2019	[87]	✓	✓	✗	✗
H2O4GPU, 2019	[89]	✗	✗	✓	✗
XGBoost library, 2019	[90]	✗	✗	✓	✗
RAPIDS cuML, 2019	[91]	✗	✗	✓	✗
Apache Spark MLlib, 2019	[92]	✗	✗	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓
R					
randomForest, 2018	[88]	✓	✓	✗	✗
H2O4GPU, 2019	[89]	✗	✗	✓	✗
XGBoost library, 2019	[90]	✗	✗	✓	✗
Apache Spark MLlib, 2019	[92]	✗	✗	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓
Scala					
Smile Random Forest, 2019	[75]	✓	✗	✗	✗
Apache Spark MLlib, 2019	[92]	✗	✗	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓
Java					
Smile Random Forest, 2019	[75]	✓	✗	✗	✗
Weka Random Forest, 2017	[94]	✓	✓	✗	✗
Apache Spark MLlib, 2019	[92]	✗	✗	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓

Although in the specialized literature there are proposals of LogitBoost for distributed computing [86], we have not been able to find any available software of this kind.

3.3. Random Forest software

Similar to AdaBoost, RF is found in all data science oriented programming languages and platforms. Due to the simplicity and homogeneity of versions, its software is more consistent and abundant. Table 3 shows a large amount of software that involves the Random Forest method.

Scikit-Learn library [39] has a class of Random Forest classifier coded in Python [87]. This class presents numerous parameters with which to adjust both the ensemble and the decision tree used. By default, each tree is trained with bootstrap, a subset of variables equal to the square root of the total and without maximum depth. However, all these parameters can be changed. In addition, it is possible to modify the split criteria from Gini impurity (by default) to information gain, to incorporate class balancing by class weights or to compute the out-of-bag error.

Among all the packages in R featuring RF, the randomForest package [88] is recommended due to its concretion and completeness. This RF software is an R port of the original Fortran code of Breiman and Cutler [26]. Similar to Scikit-Learn RF, randomForest package by default matches the original method, but users can tune all its parameters. Among these, it is worth mentioning the possibility of carrying out a stratified sampling, which is very useful for unbalanced scenarios. Only the Gini index can be used as an impurity measure.

Scala users have at their disposal the Java class of RF method fully compatible with Scala in the Smile software [75]. The possibility of selecting the criterion of impurity and of reducing the size of the ensemble through ensemble pruning should be highlighted.

Weka also includes the RF algorithm in its Java software [94]. It includes functionalities similar to those mentioned in the previous software, except for class weighting and the selection of different tree branching criteria.

All these software allow extracting the relevance of the different variables in the classification task.

Since each iteration of the RF is independent of each other, its parallel code version is trivial. Therefore, the majority of the software mentioned above, Scikit-Learn RF [87], randomRorest [88] package in R and Weka RF [94] offer the possibility to run in parallel. The user can control the number of execution jobs with a method parameter.

H2O collection of GPU solvers (H204GPU) [89] features a RF method under the CUDA GPU environment with Python and R APIs. XGBoost library [90] can be used to train a standalone RF with GPUs in CLI, Python and R. RAPIDS cuML [91] also has a GPU-accelerated RF with a histogram-based algorithm for split finding in a Python Scikit-learn interface. Additionally, a scalable GPU implementation of RF can also be found written in C++ within CUDA environment [95,96].

Apache Spark [79] has its own distributed RF [92] in their official machine learning library MLlib. It uses the distributed implementation of the decision tree existing in Spark. Finally, another distributed variant of RF can be found in H20 3 implemented on top of the Map/Reduce framework [93].

3.4. Gradient Boosting Machine software

GBM is undoubtedly one of the ensembles with the largest representation of software. However, many of the most famous implementations are variants considered different algorithms such as XGBoost [32], CatBoost [35,36] or LightGBM [34]. Table 4 lists those implementations closest to the original GBM and SGB methods.

Scikit-learn library has two different versions of Gradient Boosted Trees: a standalone method [97] and a Histogram-based method [98]. The first one includes a variety of different configurable functionalities into an original GBDT scheme to prevent over-fitting: instance subsampling as in SGB, a limited number of features considered as split candidates and early-stopping. Additionally, the data can be presorted to speed up the training and custom or Scikit-learn estimator can be used to calculate the priors. Histogram-based Gradient Boosting algorithm is designed for big data-sets, in which their feature real values will be pre-binned into discrete values. This reduces the number of possible splitting points and thus, the time and memory needed to fit the model.

The R package *gbm — Generalized Boosted Regression Models* [99] implements extensions of additive boosting models such as gradient boosting machine in C++ with a R front-end. It includes around 10 different loss functions for regression, classification and ranking problems. Monotonicity constraints [60] and SGB row subsampling are supported. However, the *gbm* package development was discontinued and users are now redirected to the new *gbm3* package [100]. This software is faster and parallel, but not completely developed. Furthermore, it has not been updated for a few years.

In the Scala programming language, a simple GBM operator with few parameters can be found in Smile library [75]. The learning and subsampling rates, the number of iterations and leaves in each tree are the only configurable aspects of this implementation.

Weka includes their own GBM Java class with the name of *AdditiveRegression* [102]. By default, it uses a decision stump as its base learner. This implementation is only able to minimize absolute and squared errors. Therefore, it does not support classification nor ranking problems. Smile implementation is rather recommended for Java users [75].

Given its additive stepwise learning, GBM is difficult to run in parallel. However, scalable and faster GBM has always drawn a lot of interest, leading to several parallel codes for large amounts of data.

ThunderGBM [101,103] is a GPU-based implementation for CUDA environment with a Python Scikit-learn-like interface. This software supports learning in multiple GPUs for regression, classification and ranking problems. The split finding procedure can be configured between an exact or a histogram-based algorithm. Several regularization techniques can be added, such as column subsampling and an L2 regularization term.

Table 4
Summary of available software for Gradient Boosting algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
Scikit-learn Gradient Boosting, 2019	[97]	✓	✗	✗	✗
Scikit-learn Hist. Gradient Boosting, 2019	[98]	✓	✗	✗	✗
ThunderGBM, 2019	[101]	✗	✗	✓	✗
Apache Spark MLlib, 2019	[92]	✗	✗	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓
R					
gbm, 2019	[99]	✓	✗	✗	✗
gbm3, 2017	[100]	✓	✓	✗	✗
Apache Spark MLlib, 2019	[92]	✗	✗	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓
Scala					
Smile Gradient Boosting, 2019	[75]	✓	✗	✗	✗
Apache Spark MLlib, 2019	[92]	✗	✗	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓
Java					
Smile Gradient Boosting, 2019	[75]	✓	✗	✗	✗
Weka Gradient Boosting, 2018	[102]	✓	✗	✗	✗
Apache Spark MLlib, 2019	[92]	✗	✗	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓

Table 5
Summary of available software for Rotation Forest algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
Rotation Forest in Python, 2019	[104]	✓	✓	✗	✗
extbfR					
rotationForest, 2017	[105]	✓	✗	✗	✗
Rweka Rotation Forest, 2019	[82]	✓	✓	✗	✗
Scala					
No software found					
Java					
Weka Rotation Forest, 2018	[106]	✓	✓	✗	✗

H20 3 [93] includes a distributed Gradient Boosting Machine with Python, R, Scala, and R APIs. The model is sequentially built with trees built in parallel. Among its features, categorical feature support, histogram-based splitting, learning rate annealing and highly configurable column and row sampling are the most remarkable.

Spark MLlib [92] also implements a standard GBM with Spark decision trees as its base learner. Only regression with squared or absolute errors and binary classification can be addressed. This implementation just offers the standard parameters of the GBM algorithm.

3.5. Rotation Forest software

Software featuring the Rotation Forest algorithm can be found in Python, R and Java programming languages. Table 5 gathers all the software found for Rotation Forest.

Python users can benefit from a Rotation Forest implementation built on the Random Forest class of Scikit-learn [104]. This software extends the Scikit-learn decision tree and Random Forest classifiers to add rotation feature extraction procedures. As a handicap, it lacks proper documentation of its use and the tuning of the method parameters. Therefore, its Rotation Forest class has the same parameters and features as these Scikit-learn classifiers. About the parameters intrinsic to the RotF algorithm, the number of feature subsets and solver of the Scikit-learn PCA method can be configured.

Table 6
Summary of available software for Extra-Trees algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
Scikit-learn Extra-Trees, 2019	[107]	✓	✓	✗	✗
R					
extraTrees, 2016	[108]	✓	✓	✗	✗
Scala					
No software found					
Java					
Weka Extra-Trees, 2018	[109]	✓	✓	✗	✗

A CRAN package named *rotationForest* brings this ensemble to R environment [105]. Rpart [72] is used as its base learner, which opens up the opportunity to adjust the parameters of the base classifier. The number of variable subsets and trees are the parameters of the ensemble. This package is limited to binary classification tasks. For multi-class classification, Rotation Forest implementation of Weka can be executed in R thanks to RWeka [82,84]. RWeka provides R interfaces for potentially all Weka Java algorithms.

Weka Rotation Forest [106] is the most complete implementation. Apart from the common parameters, the feature extraction technique and the base classifier can be changed from the default PCA and J48 tree of Weka. The size of the bootstrap sampling and the minimum and the maximum number of features per group are configurable.

The parallel training of Rotation Forest is straightforward to code. Both Python and Weka implementations of Rotation Forest supports parallel learning in CPU. RWeka is needed to access this feature in R, since *rotationForest* package does not provide it. No open or private software is found for GPU or distributed training of Rotation Forest.

3.6. Extra-Trees software

Extra-Trees is present in almost all Machine Learning software libraries. Table 6 summarizes the availability of ET software in the most common programming languages for machine learning.

Scikit-learn python library includes the ET algorithm with its decision tree adaptation as a base learner [107]. Since RF and ET share structures and the same base learner in this software, their parameters are the same. By default, samples bootstrap and the square root of the total amount of features are used to train each tree. It is worth mentioning the possibility of growing trees in the best-first fashion by limiting their extension by the maximum number of leaves. Like many methods in Scikit-learn, ET supports class weighting to deal with class unbalancing but does not support categorical features.

R users have at their disposal the package named *extraTrees* [108], which is an R wrapper for a Java implementation of the ET algorithm. It implements its decision tree that chooses the split points in relation to the Gini impurity index. As a parameter, users can choose the number of random cuts proposed for each feature. By default, a cut is proposed just as the original method was designed. Additionally, this package supports sample weighting, rows under-sampling, and multi-task learning

Apart from *extraTrees*, Weka also includes a Java Extra-Trees implementation [109]. Weka implements a single Extreme Randomized Tree that can be used to form an ensemble through a metaclass, called RandomCommittee. This software only includes the basic parameters of the original ET method.

As with other methods whose learners can be trained independently, all the libraries mentioned above include the possibility of parallel training. ET barely lacks software in distributed computing or GPU computing paradigms, except for a C++ implementation under the CUDA GPU environment [95,96].

Table 7
Summary of available software for Random Patches algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
Scikit-learn Random Patches, 2019	[110]	✓	✓	✗	✗
R, Scala & Java					
No software found					

Table 8
Summary of available software for Random Rotation Ensemble algorithms. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
Random Rotation Ensembles in Python, 2019	[111]	✓	✓	✗	✗
R, Scala & Java					
No software found					

3.7. Random Patches software

In terms of software presence, RP has probably gone unnoticed due to its similarity to Bagging and Random Subspace. Although Bagging and Random Subspace are present in most programming languages, RP lacks its own software in almost all of them. Table 7 shows the few software packages related to the Random Patches method.

As an exception, Scikit-learn includes RP through a bagging-based ensemble meta-classifier [110]. By adjusting its parameters, the ensemble can be trained with different bagging-based schemes: traditional Bagging, Pasting, Random Subspace or Random Patches. Any classifier compatible with Scikit-learn can be used as a base learner. In addition, Scikit-learn RP can be run in parallel with multiple CPU jobs.

3.8. Random Rotation Ensembles software

Random Rotation Ensembles are not as well known as the other ensembles described in this article, which is reflected in the lack of software that includes them. Table 8 shows the only available implementation of these ensembles.

This Python package [111] is fully compatible with Scikit-learn. Random Rotation Extra-Trees and Random Forest classes extend their respective Scikit-learn classes to introduce the rotation of space before the training of each tree. Thus, these new methods inherit all their features and parameters. Among them, the parameter tuning of the Scikit-learn decision tree used as their base learner, the choice of bootstrap and the maximum number of considered features and the parallel training are the most remarkable.

3.9. XGBoost software

XGBoost library is an open-source project with a huge community and well-known sponsors, such as NVIDIA, InAccel, and AWS. Its large community of users and developers has contributed to the expansion of the software to all kinds of platforms and programming languages. Table 9 gathers all platforms and languages supported by official and third-party libraries.

The official XGBoost library [90] has APIs for Python, R, Scala, Java and Julia programming languages with similar features in sequential and parallel CPU environments.

Python package supports the main data format typically used by Python users: NumPy array, SciPy sparse array and Pandas data frame, among others. The learning API is available in its self-implemented API and in a Scikit-Learn wrapper interface, which both include all key parameters to control the additional features of XGBoost. Additionally, Plotting API allows the users to understand better the model learned through plots of the fitted trees and importance and Callback API performs the early stopping.

Table 9
Summary of available software for XGBoost algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
XGBoost library, 2019	[90]	✓	✓	✓	✓
H2O4GPU, 2019	[89]	✗	✗	✓	✗
H2O 3, 2019	[93]	✗	✗	✗	✓
R					
XGBoost library, 2019	[90]	✓	✓	✓	✗
H2O4GPU, 2019	[89]	✗	✗	✓	✗
H2O 3, 2019	[93]	✗	✗	✗	✓
sparkxgb, 2019	[112]	✗	✗	✗	✓
Scala					
XGBoost library, 2019	[90]	✓	✓	✗	✓
H2O 3, 2019	[93]	✗	✗	✗	✓
Java					
XGBoost library, 2019	[90]	✓	✓	✗	✗
H2O 3, 2019	[93]	✗	✗	✗	✓

The R package of XGBoost supports R dense and sparse matrix and its own implementation DMatrix, which is recommended. The training can be done in parallel on a single machine with OpenMP. Similar to the Python package, callbacks and very complete plot functions, including SHapley Additive exPlanation — SHAP plots [113], are available. The package is also compatible with Caret [68].

XGBoost JVM package includes Java and Scala APIs. The input data is managed by its own DMatrix, which supports the LibSVM text format, the sparse matrix in CSC and CSR formats and dense matrix using a row-major layout. The JVM package has the same core functionalities as Python and R packages.

As mentioned along the XGBoost section, parallel tree learning is one of XGBoost's important improvements. Therefore, all these 3 packages enable parallel CPU computing. Additionally, the XGBoost library has CUDA-GPU and major distributed environments support. GPU accelerated XGBoost [114] is implemented with a fast histogram algorithm in Python and R packages [90] and H2O4GPU [89]. Multi-GPU learning is supported through Dask.

Besides Dask API, the official XGBoost library [90] has Scala API for Spark and Flink and Python wrapper for PySpark. XGBoost is also included in the H20 3 library of distributed algorithms [93]. Sparkxgb [112] is a XGBoost implementation in SparklyR, a R interface for Spark.

3.10. LightGBM software

LightGBM framework [115] is an open-source project of Microsoft corporation under MIT license. This library has two popular programming language APIs, GPU training and Spark support. Table 10 references the LightGBM available software classified by computation paradigms.

Python is one of the programming languages supported by the LightGBM framework. As XGBoost Python API, Input data can be passed in plenty of Python standard formats: NumPy and SciPy sparse array, Pandas data frame, CSV files, and H2O data table. The models can be trained with a self-designed or a Scikit-learn API. Callbacks and Plots are supported by their respective APIs.

LightGBM R API offers similar functionalities of the Python API, such as early-stopping, saving/loading models, predictive analysis through feature importance and plotting, etc. Additionally, LightGBM also has an official C API [115] and external Julia API [117].

Table 10
Summary of available software for LightGBM algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
LightGBM library, 2019	[115]	✓	✓	✓	
MMLSpark, 2019	[116]	✗	✗	✗	✓
R					
LightGBM library, 2019	[115]	✓	✓	✓	✗
MMLSpark, 2019	[116]	✗	✗	✗	✓
Scala					
MMLSpark, 2019	[116]	✗	✗	✗	✓
Java					
MMLSpark, 2019	[116]	✗	✗	✗	✓

Table 11
Summary of available software for CatBoost algorithm. SC: Sequential CPU computing, PC: Parallel CPU computing, GP: GPU computing, DC: Distributed computing.

Software tool	Ref.	SC	PC	GC	DC
Python					
CatBoost library, 2019	[119]	✓	✓	✓	✗
R					
CatBoost library, 2019	[119]	✓	✓	✓	✗
Scala & Java					
No software found					

Both Python and R APIs, support parallel learning in CPU and GPU. As explained before, the LightGBM library provides 3 different parallel algorithms: data-parallel recommended for a high number of samples, feature parallel for high dimensional data and voting parallel for high dimensional data with a big amount of samples. GPU LightGBM implementation [118] works with CUDA/NVIDIA, AMD and Intel GPU environments. GPU algorithm accelerates the building process of the feature histograms.

Microsoft Machine Learning for Apache Spark — MMLSpark [116] has a LightGBM implementation available on Spark, PySpark and SparklyR.

3.11. CatBoost software

CatBoost is available as an open-source library of the company Yandex LLC [119]. CatBoost library is not as popular as XGBoost or LightGBM libraries and lacks some software supports. Table 11 gathers the programming language and computation paradigms supported by the CatBoost project.

CatBoost is implemented in C++ with Python and R packages. Python API has a similar style to the Scikit-learn interface. Both packages have the same features and support sequential or parallel in CPU or GPU. GPU parallel learning can be done in single or multiple GPUs with NVIDIA drivers. Once models are trained, CatBoost library provides the possibility to export them to other language environments, such as Swift CoreML, C/C++, Java, .NET, among others.

Besides the training package, CatBoost offers a package for model analysis, a data visualization package, and CatBoost viewer. The model analysis explains the model with functions, such as SHAP values, feature importance, and interactions. The data visualization package provides plots for training statistics during and after the learning procedure. CatBoost viewer brings these charts to an interactive user interface accessible from a browser.

4. Experimental setup

In this section, we present the experimental framework used to compare the ensembles reviewed in this paper. Section 4.1 introduces the data-sets involved in the experiments. We list the algorithms and their hyper-parameters in Section 4.2. Section 4.3 present the evaluation metrics and statistical tests used.

4.1. Data-sets

Although most of the featured ensembles can be used for both classification and regression, even for ranking, the experiments are focused on classification tasks. 76 different data-sets from the KEEL software [120] have been considered, of which 70 data-sets are small or medium size, and 6 are large data-sets. These big data-sets have a considerable big number of instances and characteristics to significantly increase computation time. In this framework, the threshold is around 250 K samples. Two separate studies are proposed according to the size of the data-sets.

Table 12 shows the 76 data-sets with a brief description of the number of instances (Ins.), attributes (At.), classes (Cl.) and imbalance ratio (IR). IR is computed as the ratio of the numbers of instances of the majority class and the minority class. An identification number (Id.) is used to refer each data-set in following tables. These data-sets can be found at the official website of the KEEL software.[1]

As can be seen in this table, there is a great diversity in this experimental framework, with binary and multi-class problems, with different numbers of classes and features and some class imbalance problems [58]. We consider a data-set to be considerably imbalanced when its IR is higher than 2. These are marked with † in Table 12.

The presence of categorical variables in many of the data-sets is also noteworthy. These are marked with * in Table 12. The majority of the implementations do not handle categorical variables except for LightGBM and CatBoost. Therefore, these are preprocessed with ordinal encoding for those without categorical support. This technique replaces categories by integers given the order of appearance. This sub-optimal mechanism could highlight the difference from those models that treat them in more sophisticated ways. In addition, other techniques such as one-hot encoding are discouraged for tree models.

4.2. Algorithms and hyper-parameters

This empirical comparison consists of 14 different algorithms from the 11 previously described ensembles and variants of Random Rotation Ensembles, XGBoost and LightGBM. From Random Rotation Ensembles, RR Forest and RR Extra-Trees have been executed. XGBoost GBDT and DART boosters have also been considered. LightGBM has been separately configured with standard GBDT and with GOSS. CatBoost automatically selects the most appropriate boosting scheme (plain or ordered GBDT) for a given problem, therefore its two variants were not executed separately.

For a more homogeneous and fair comparison of the software, a common programming environment and language have been prioritized: Python with implementations based on Scikit-learn API. Additionally, most of these implementations share the same base learner, Scikit-learn decision tree or its extensions. The exceptions are XGBoost, LightGBM, and CatBoost, which implement their trees.

Table 13 lists the algorithms and their software implementations, as well as their configuration hyper-parameters. The hyper-parameters are kept by default except for the number of trees and their maximum depth, which are tuned to the best accuracy for 5-fold cross-validation of each data-set. From 10 to 100 trees with steps of 10 are tested for the number of estimators and the maximum depth is chosen among 1, 3, 5 and 10. For large data-sets, the number of trees is set at 100 and their depth at 5, to compare the training and test runtimes with the same number of iterations.

[1] https://sci2s.ugr.es/keel/category.php?cat=clas

Table 12
Description of the classification data-sets under consideration. The marks * and † indicate data-sets with categorical features and imbalanced data-sets, respectively.

Id.	Data-set	Ins.	At.	Cl.	IR	Id.	Data-set	Ins.	At.	Cl.	IR
#1	*abalone*†*	4,174	8	28	689.00	#36	*marketing*†	6,876	13	9	2.49
#2	*adult*†*	45,222	14	2	3.03	#37	*monk-2*	432	6	2	1.12
#3	*appendicitis*†	106	7	2	4.05	#38	*movement_libras*	360	90	15	1.00
#4	*australian**	690	14	2	1.25	#39	*mushroom**	5,644	22	2	1.62
#5	*automobile*†*	150	25	6	16.00	#40	*newthyroid*†	215	5	3	5.00
#6	*balance*†	625	4	3	5.88	#41	*nursery*†*	12,690	8	5	2160.00
#7	*banana*	5,300	2	2	1.23	#42	*optdigits*	5,620	64	10	1.03
#8	*bands*	365	19	2	1.70	#43	*page-blocks*†	5,472	10	5	175.46
#9	*breast*†*	277	9	2	2.42	#44	*penbased*	10,992	16	10	1.08
#10	*bupa*	345	6	2	1.38	#45	*phoneme*†	5,404	5	2	2.41
#11	*car*†*	1,728	6	4	18.62	#46	*pima*	768	8	2	1.87
#12	*chess**	3,196	36	2	1.09	#47	*ring*	7,400	20	2	1.02
#13	*cleveland* †	297	13	5	12.31	#48	*saheart**	462	9	2	1.89
#14	*coil2000*†	9,822	85	2	15.76	#49	*satimage*†	6,435	36	7	2.45
#15	*connect-4*†*	67,557	42	3	6.90	#50	*segment*	2,310	19	7	1.00
#16	*contraceptive*	1,473	9	3	1.89	#51	*shuttle*†	58,000	9	7	4558.60
#17	*crx**	653	15	2	1.21	#52	*sonar*	208	60	2	1.14
#18	*dermatology*†	358	34	6	5.55	#53	*spambase*	4,597	57	2	1.54
#19	*ecoli*†	336	7	8	71.50	#54	*spectfheart*†	267	44	2	3.85
#20	*fars*†*	100,968	29	8	4679.56	#55	*splice*†*	3,190	60	3	2.16
#21	*flare*†*	1,066	11	6	7.70	#56	*tae*	151	5	3	1.06
#22	*german*†*	1,000	20	2	2.33	#57	*texture*	5,500	40	11	1.00
#23	*glass*†	214	9	7	8.44	#58	*thyroid*†	7,200	21	3	40.16
#24	*haberman*†	306	3	2	2.78	#59	*tic-tac-toe**	958	9	2	1.89
#25	*hayes-roth*†	160	4	3	2.10	#60	*titanic*†	2,201	3	2	2.10
#26	*heart*	270	13	2	1.25	#61	*twonorm*	7,400	20	2	1.00
#27	*housevotes**	232	16	2	1.15	#62	*vehicle*	846	18	4	1.10
#28	*ionosphere*	351	33	2	1.79	#63	*vowel*	990	13	11	1.00
#29	*iris*	150	4	3	1.00	#64	*wdbc*	569	30	2	1.68
#30	*kr-vs-k*†*	28,056	6	17	168.63	#65	*wine*	178	13	3	1.48
#31	*led7digit*	500	7	10	1.54	#66	*winequality-red*†	1,599	11	11	68.10
#32	*letter*	20,000	16	26	1.11	#67	*winequality-white*†	4,898	11	11	439.60
#33	*lymphography*†*	148	18	4	40.50	#68	*wisconsin*	683	9	2	1.86
#34	*magic*	19,020	10	2	1.84	#69	*yeast*†	1,484	8	10	92.60
#35	*mammographic*	830	5	2	1.06	#70	*zoo*†*	101	16	7	10.25
#71	*skin*†	245,057	3	2	3.82	#74	*poker*†	1,025,010	10	10	64212
#72	*kddcup*†*	494,020	41	23	140395	#75	*ECBDL14*†*	2,063,187	631	2	2.00
#73	*ht-sensor*	928,991	11	3	1.25	#76	*watch-acc*	3,540,962	20	7	1.50

With this practical study, we aim to compare the overall performance of the ensembles' software tools. Therefore, an extensive hyper-parameter tuning is beyond the scope of this paper, particularly given the large number of methods and data sets in this macro-experimentation. Since hyper-parameters tuning is crucial for GBM-based algorithms, we explore its impact on their performance with a smaller experimental setup in Section 7.5.

The model learning has been limited to a single sequential process for small and medium data-sets since the overhead time of thread creation is greater than the computational time. On the other hand, for large data-sets, each software has the responsibility to select the correct number of processes.

4.3. Evaluation metrics and statistical tests

To evaluate the quality of the ensembles and their software, 4 different metrics are obtained from the 5-fold stratified cross-validation executions for each data-set:

- Accuracy: Ratio of correctly classified samples with respect to the total amount of instances.
- Cohen's kappa score: Level of agreement between two classifiers or a classifier and the actual values. It is defined as $k = (p_o - p_e)/(1 - p_e)$, where p_o is the accuracy and p_e, the probability of agreement by chance. Values close to 1 indicate a high level of agreement and values around 0 or below mean random guessing. It is a useful metric for multi-class problems.
- Training runtime: Seconds needed to learn the model. Fast training is essential to perform an exhaustive and agile parameter tuning.
- Test runtime: Time measured in seconds to predict the entire test set.

Accuracy and Cohen's kappa score are very common metrics for evaluating general classification performance. Training and test runtimes measure the efficiency of the models.

Statistical tests [121] are used to validate the outcomes of the experimental analysis. In particular, Wilcoxon Signed Rank test ascertains the significance of pairwise comparisons. The resulting p-values of each comparison are presented graphically in a matrix with a black-orange color gradient, where pure black means the rejection of the hypothesis of equivalence with $\alpha = 0.05$. The algorithms are also arranged in the axes from left to right and from bottom to top by the median of the positions in the ranking of each data-set.

Although Wilcoxon Signed Rank test is discouraged to determine the best method among multiples, we have decided to use it instead of a statistical test for multiple comparisons because the latter does not yield significant differences among the ensembles given the large size of the experiment — 14 algorithms × 70 data-sets. Besides, we want to graphically show the difference between pairs of algorithms, and highlight those that have a better performance in pairwise comparisons. In this regard, we also note that recently Bayesian statistical tests [122] have emerged as an alternative to non-parametric tests. However, their output is not convenient for its visualization in a comparative table comprising multiple models. This, along with the rationale given above, motivates the adoption of the Wilcoxon test for the table depicting all pairwise comparisons.

Table 13
Hyper-parameters for the compared ensembles.

Algorithm	Software	Hyper-parameters
Common parameters (tuned)	Python	n_trees = [10,...,100], max_depth=[1, 3, 5, 10]
AdaBoost (AdaB) [23,24]	[67]	base_estimator=sklearn.DT, algorithm=SAMME.R,
LogitBoost (LB) [25]	[80]	base_estimator=sklearn.DT, weight_trim_quantile=0.05, max_response=4.0, learning_rate=0.1, bootstrap=False
Random Forest (RF) [26]	[87]	max_features=sqrt(n_features)
Gradient Boosting Machine (GBM) [31]	[97]	loss=deviance, learning_rate=0.1, subsample=1.0
Rotation Forest (RotF) [29]	[104]	n_features_per_subset=3
Extra-Trees (ET) [27]	[107]	max_features=sqrt(n_features)
Random Patches (RP) [28]	[110]	base_estimator=sklearn.DT, max_samples=0.66, max_features=0.66, bootstrap=False, bootstrap_features=False
Random Rotation Ensemble [30]	[111]	
Random Rotation Forest (RRF)		ensemble=RF, max_features=sqrt(n_features)
Random Rotation Extra-Trees (RRET)		ensemble=ET, max_features=sqrt(n_features)
XGBoost [32]	[90]	obj=binary:logistic/multi:softmax, default parameters
XGB-GBDT		booster=gbtree
XGB-DART		booster=dart, rate_drop=0.1, skip_drop=0.5
LightGBM [34]	[115]	obj=binary/multiclass, default parameters
LightGBM-GBDT (LGB-GBDT)		boosting_type=gbdt
LightGBM-GOSS (LGB-GOSS)		boosting_type=goss
CatBoost (CatB) [35,36]	[119]	obj=multiclass, default parameters

5. Experimental study on ensembles performance: small and medium size, categorical variables and large data

In this Section, we empirically compare the different algorithms and their software implementations in terms of predictive capability and efficiency. Their behaviors are analyzed over different scenarios. In Section 5.1, the predictive capabilities are tested with small- and medium-sized classification problems. The methods are compared in scenarios with categorical variables in Section 5.2. Section 5.3 presents ensembles' performance and efficiency with large problems.

5.1. Scenario I: Performance analysis over standard classification problems

The standard accuracy is one of the measures used to evaluate the predictive capability of algorithms. Table 14 gathers the results in terms of accuracy obtained in small and medium problems. The bold-face font is used to highlight the best result of each data-set. Fig. 2 represents the p-values of Wilcoxon Signed Rank tests for the accuracy results achieved in small and medium classification problems. The algorithms are ranked in the axes from left to right and bottom to top. Pure black cells represent rejections of the hypothesis of equivalence.

Given the results shown in Table 14 and Fig. 2, we can draw the following findings:

- LB obtains the best result on average closely followed by XGB-GDBT and GBM with a very small difference. However, it is pushed down to the fourth position in the ranking of Fig. 2.
- XGB-GDBT is ranked first followed by LGB-GBDT, RotF and LB in Fig. 2. It achieves significant better accuracy results than XGB.DART, ET, AdaB, CatB, RP, RRF, and RRET.
- Random Rotation Ensembles (RRF and RRET) obtain the worst accuracy performance with a large margin and statistical significance. These methods are the only ones with an average accuracy below 0.8.
- Boosting methods are usually rated ahead of ensembles with independent base classifier learning, with exception of AdaB and CatB. CatB performs significantly worse than the majority of boosting algorithms in terms of accuracy as shown in Fig. 2.
- Among bagging-like and rotation-based algorithms, RotF has the best performance in terms of accuracy supported by the statistical tests. It is also the method with the greater number of cases with the best accuracy: a total of 13 data-sets.
- Concerning algorithmic adaptations, ET and RF obtain significantly better accuracy results that their random rotation adaptations, RRET and RRF. The GBDT version of XGB is considered significantly superior to XGB-DART. However, there are no statistical differences between LGB-GBDT and LGB-GOSS.

Next, the predictive capability of the ensembles is analyzed in terms of a balance good prediction for all classes measured by Cohen's kappa coefficient. Table 15 records Cohen's kappa results achieved on small and medium problems. Bold-face font highlights the best result among the different ensembles for each data-set. Fig. 3 shows the p-values of Wilcoxon Signed Rank tests for Cohen's kappa scores obtained in small and medium-size data-sets. From Table 15 and Fig. 3, we can extract the following statements:

- In terms of Cohen's kappa coefficient, GBM obtains the best average results, followed by XGB-GDBT and AdaB. However, GBM ranks fifth.
- XGB-GDBT is ranked first according to the kappa results, closely followed by LGB-GBDT and LGB-GOSS. Its superior performance is supported by the statistical tests when compared to XGB-DART, RF, ET, CatB, RP, RRF and RRET.
- RRF and RRET remain the methods with the worst performance. The difference between the average kappa score of RR ensembles and the rest is around 0.1, which is considerably large. In Fig. 3, they are also rated at the last positions with significant differences.
- As shown in Fig. 3, boosting methods mainly outperform bagging-based methods with exception of RotF and CatB. RotF is significantly the best bagging-based ensemble. And, it ranks higher than a few boosting methods, such as XGB-DART, AdaB, and CatB. CatB is the worst boosting method in terms of kappa score with statistical support.
- Similarly to the accuracy results, ET and RF are significantly better that their random rotation versions, RRET and RRT. And, there are no significant differences among the gradient boosting methods, except CatB.
- For imbalanced problems, the methods experience a drop in performance in terms of both Accuracy and Cohen's Kappa score. The average Accuracy and Kappa score obtained for imbalanced sets is considerably lower. In addition, some ensembles report extremely low kappa scores close to or equal to 0 for highly imbalanced sets. These algorithms are not able to learn from the minority classes and, therefore, they are systematically misclassified. These data-sets require a special treatment during the learning process, such as preprocessing or cost-sensitive weighting [10,123]. In particular, the combination of sampling techniques and ensembles are great solutions in these scenarios [124].

Appendix A includes the efficiency results over the 70 standard-size problems measured with the training and test runtimes.

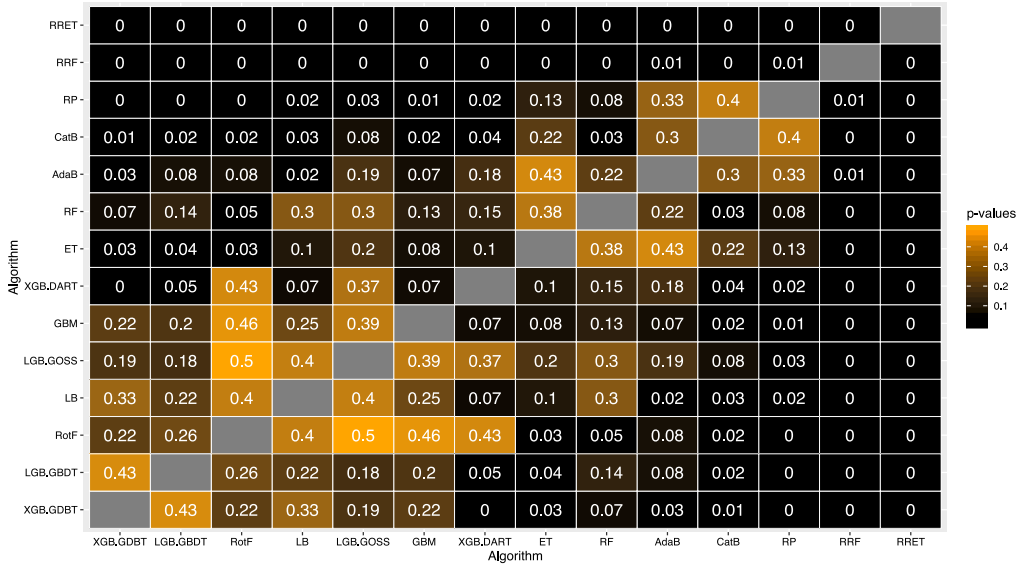

Fig. 2. P-values of Wilcoxon Signed Rank test for accuracy performance.

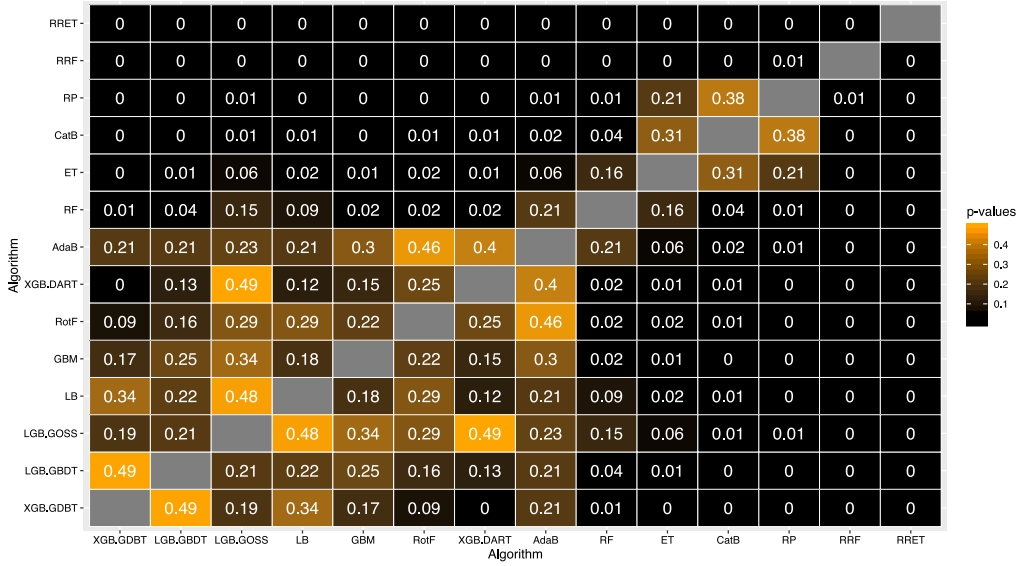

Fig. 3. P-values of Wilcoxon Signed Rank test for Cohen's kappa scores.

5.2. Scenario II: Performance analysis over problems with categorical features

Next, we analyze the predictive behavior of bagging and boosting based algorithms on data with categorical features. These data-sets are highlighted with the symbol * in all tables. The analysis of their performance is done in terms of accuracy and Cohen's kappa. Figs. 4 and 5 represent the statistical test outcomes for accuracy and kappa performance on only data-set with categorical features. With these test, we aim to analyze the significance of the use of methods with categorical feature support for problems with this type of inputs.

From the results marked with * in Tables 14 and 15 and the statistical tests in Figs. 4 and 5, we can extract the following conclusions:

- At first glance on Tables 14 and 15, algorithms with categorical support do not seem to outperform either the accuracy or the kappa score of the other ensembles. Their average performance represented with *Avg** are not significantly higher. LGB-GBDT, LGB-GOSS, and CatB achieve superior accuracy results only in 4 of the 21 data-sets with categorical inputs. In terms of kappa score, they are better in only 6 of these particular data-sets marked with *.
- GBM obtains the best on-average accuracy and kappa scores for these scenarios. RotF performs a bit worse, due to the limited benefit of space rotations with categorical variables.
- Figs. 4 and 5 indicate no significant differences between the algorithms with categorical input support (LGB-GBDT, LGB-GOSS, and CatB) and the rest of ensembles.
- Among the ensembles with this support, LGB-GBDT is the best with the overall second-best performance in Figs. 4 and 5. LGB-GOSS maintains good performance. And, CatB is one of the worst methods, surpassing only the Random Rotation ensembles in 5.

5.3. Scenario III: Ensembles' performance and efficiency with large problems

In this subsection, we analyze the results obtained for the big-sized data-sets in terms of precision and time consumption. Since some of the problems are very large, no parameter tuning is performed and those

Table 14
Accuracy results for small and medium data-sets.

	AdaB	LB	RF	GBM	RotF	ET	RP	RRF	RRET	XGB GDBT	XGB DART	LGB GBDT	LGB GOSS	CatB
#1†*	0.2423	0.26	0.2646	0.2676	**0.2702**	0.2695	0.2667	0.2695	0.269	0.2662	0.2688	0.2669	0.2676	0.2625
#2†*	0.8657	0.8683	0.855	0.8698	0.8604	0.8367	0.8625	0.8368	0.8079	0.8692	0.868	**0.8701**	0.8683	0.8585
#3†	0.8771	0.887	0.887	0.887	0.8775	0.8775	0.8775	0.8779	0.887	0.887	0.887	0.8584	0.8022	**0.8965**
*#4**	0.8551	0.8652	0.868	0.8652	0.8725	**0.8739**	0.8681	0.871	0.8725	0.8695	0.8667	0.8638	0.8652	0.8623
#5†*	0.8772	0.8646	0.8697	0.8766	0.8968	**0.9059**	0.8834	0.7866	0.7957	0.8891	0.8904	0.8644	0.7072	0.66
#6	0.7724	0.8047	0.7232	0.7968	0.7996	0.7454	0.7394	**0.8799**	0.8671	0.7744	0.7555	0.7663	0.8243	0.7649
#7	0.8917	0.9	0.9009	0.9015	0.8845	0.9023	0.7234	0.8983	0.9009	0.9028	0.9028	0.9028	0.9026	**0.903**
#8	0.6247	0.6219	0.6411	0.6274	0.6301	0.6438	0.6301	**0.6575**	0.6438	0.6301	0.6137	0.6301	0.6301	0.6356
#9†*	0.7328	0.762	0.7549	0.7584	0.7584	**0.7802**	0.762	0.7763	0.7766	0.762	0.762	0.7623	0.7767	0.7693
#10	0.7333	0.7246	0.742	0.7304	**0.7478**	0.7043	0.7159	0.7391	0.7101	0.7304	0.742	0.7333	**0.7478**	0.7391
#11†*	**0.8213**	0.8155	0.8172	0.8143	0.8091	0.7669	0.7396	0.7547	0.7257	0.7843	0.7854	0.7918	0.7894	0.7981
*#12**	0.97	0.984	0.9387	**0.9844**	0.9775	0.9449	0.9381	0.8752	0.8417	0.9803	0.98	0.9772	0.9746	0.9459
#13†	0.5824	0.5958	0.5957	0.6128	0.5894	0.6197	0.5958	0.5962	0.5894	0.623	0.6095	**0.6263**	0.6096	0.5995
#14†	0.94	0.9403	0.9406	0.9405	0.9403	0.9406	0.9403	0.9403	0.9403	0.9405	**0.9407**	0.9406	0.9404	0.9403
#15†*	**0.6997**	0.6617	0.6583	0.6628	0.6583	0.6584	0.6583	0.6583	0.6583	0.6583	0.6583	0.6897	0.6901	0.6604
#16	0.5567	0.5682	0.5627	**0.5723**	0.5716	0.5498	0.5567	0.4705	0.4684	0.5703	0.5696	0.5703	0.5621	0.5587
*#17**	0.8517	0.8642	0.8624	0.8642	0.8642	0.8627	**0.87**	0.8624	0.8639	0.8642	0.8642	0.8642	0.8656	0.8642
#18†	0.9552	0.9612	0.975	0.961	0.9667	**0.986**	0.9721	0.8128	0.8155	0.964	0.9641	0.9664	0.9636	0.9776
#19†	0.8534	0.8306	0.8511	0.8361	0.8508	0.8415	**0.8628**	0.8268	0.8243	0.8396	0.8396	0.8304	0.8184	0.8333
#20†*	0.6846	0.7416	**0.7797**	0.7648	0.7645	0.7731	0.7794	0.7472	0.7098	0.7728	0.7668	0.7682	0.7723	0.7756
#21†*	0.7186	0.7589	0.7533	**0.7627**	0.7496	0.7457	0.7533	0.7243	0.7196	0.7589	0.7571	0.7561	0.7571	0.7429
#22†*	0.765	0.775	0.765	**0.782**	0.779	0.762	0.774	0.764	0.75	0.781	**0.782**	0.78	0.765	0.749
#23†	0.7994	0.7887	0.8081	0.7574	0.7615	0.7673	**0.8129**	0.7336	0.7292	0.7804	0.7801	0.7947	0.7263	0.7719
#24†	0.7321	0.7518	0.7387	0.7485	0.7388	0.7353	0.7419	0.732	0.7353	0.7485	0.7485	0.7518	0.7549	**0.7583**
#25†	**0.8626**	0.8443	0.8566	0.8443	0.8503	0.8445	0.8503	0.6	0.5688	0.8322	0.8318	0.6809	0.5758	0.8187
#26	0.8333	0.8444	0.8593	0.8481	0.8444	0.863	0.837	0.8481	0.8519	0.8556	0.8556	0.8556	0.8519	**0.8667**
*#27**	0.9653	0.9696	0.9654	0.9696	0.9696	**0.9739**	0.9696	0.9524	0.9526	0.9696	0.9696	0.9696	0.9696	0.9696
#28	0.9175	0.9145	0.9288	0.9174	0.9316	0.9458	0.9232	**0.9572**	0.9429	0.9174	0.9145	0.9145	0.9118	0.9288
#29	0.9667	0.9667	0.9667	0.9667	0.9667	0.9667	0.9667	**0.9733**	0.9667	0.9667	0.9667	0.9667	0.94	0.9667
#30†*	0.2155	0.2498	0.1994	0.2531	0.2103	0.2098	0.1916	0.213	0.2158	0.2077	0.2269	0.2135	0.2066	**0.2554**
#31	0.7179	**0.7659**	0.7364	0.7617	0.7441	0.7381	0.7441	0.7377	0.7338	0.7638	0.7574	0.7617	0.7597	0.744
#32	0.9412	0.9596	0.8679	0.952	0.87	0.8676	0.8631	0.3114	0.3114	0.9551	0.9408	0.9577	**0.9627**	0.8855
#33†*	0.8256	0.8396	**0.8485**	0.8383	0.7992	0.8394	0.8401	0.8148	0.8256	0.8265	0.8053	0.8454	0.7929	0.7932
#34	0.8725	0.8817	0.8684	0.8815	0.8723	0.8375	0.8701	0.8521	0.8195	**0.8847**	0.8805	0.8846	0.881	0.8652
#35	0.8289	**0.8481**	0.8421	0.8445	0.8385	0.8229	0.8397	0.8385	0.8433	0.8433	0.8469	0.8445	0.8433	0.8445
#36†	0.3314	0.3381	0.3466	0.3408	**0.3501**	0.3413	0.348	0.2952	0.2915	0.3445	0.3441	0.3434	0.346	0.3491
#37	**1.0**	**1.0**	0.9907	**1.0**	**1.0**	0.9908	0.9791	0.9722	0.9722	**1.0**	**1.0**	**1.0**	**1.0**	0.9954
#38	0.652	0.6933	0.7247	0.6107	**0.824**	0.7567	0.6973	0.806	0.8107	0.6667	0.67	0.694	0.692	0.7087
*#39**	0.9377	0.9377	0.9327	0.935	0.9391	0.9417	0.9377	0.9386	0.9391	0.9355	0.9355	0.9812	0.9812	**0.9887**
#40†	0.9581	0.9628	0.9581	0.9581	**0.9674**	**0.9674**	0.9628	0.9628	0.9628	0.9581	0.9535	0.9581	0.9535	0.9628
#41†*	0.9453	**0.9622**	0.8221	0.9497	0.878	0.8394	0.826	0.7837	0.7692	0.9272	0.9119	0.9002	0.9065	0.7793
#42	**0.9797**	0.9735	0.9765	0.9713	0.971	0.9758	0.969	0.7692	0.7599	0.9722	0.9692	0.9751	0.9792	0.9573
#43†	0.9666	0.9687	0.9636	0.9656	0.9656	0.9556	0.9645	0.9598	0.9587	0.968	0.9669	**0.9693**	**0.9693**	0.9647
#44	**0.9934**	0.9909	0.9855	0.989	0.9911	0.9773	0.9859	0.6367	0.6371	0.9882	0.9849	0.9889	0.9898	0.9759
#45†	**0.9123**	0.9077	0.9017	0.9071	0.9047	0.8577	0.8893	0.8884	0.8588	0.9054	0.9047	0.9008	0.8982	0.8731
#46	0.7525	0.759	0.7617	0.7669	0.7499	**0.7734**	0.759	0.7722	0.7669	0.7669	0.7629	0.7669	0.7695	0.759
#47	0.9676	0.9635	0.938	0.9628	0.9523	0.9615	0.9385	0.9418	0.9505	0.9635	0.9615	0.9646	**0.9703**	0.9505
*#48**	0.7122	0.7533	0.736	0.7576	0.7359	0.7317	0.736	0.7317	0.7338	0.7532	0.7511	**0.7597**	0.7359	0.7382
#49†	0.9012	0.9001	0.8934	0.8943	0.8984	0.8844	0.89	0.5275	0.5277	0.9009	0.8974	0.901	**0.9038**	0.8724
#50	**0.9866**	0.9818	0.9766	0.9792	0.9792	0.9706	0.9723	0.961	0.9494	0.9801	0.9775	0.9848	0.9848	0.968
#51†	**0.9999**	**0.9999**	**0.9999**	0.9998	**0.9999**	0.997	**0.9999**	0.8533	0.8533	**0.9999**	0.9998	0.9977	0.9977	0.9996
#52	0.7422	0.7369	**0.7608**	0.6984	0.722	0.7415	0.7129	0.7068	0.6987	0.7372	0.7178	0.7267	0.7367	0.737
#53	0.9367	0.9439	0.9371	0.9432	0.9408	0.8908	0.9384	0.9345	0.9221	0.9434	0.9439	**0.9484**	0.9474	0.9332
#54†	**0.8315**	0.824	0.8126	0.8165	0.8164	0.8199	0.8125	0.809	0.809	0.824	0.8163	0.8164	0.8277	0.8088
#55†*	0.9439	0.9599	0.9489	**0.9624**	0.9505	0.9455	0.9593	0.684	0.6132	0.9596	0.9574	0.9586	0.9596	0.9351
#56	0.7163	0.6887	0.7101	0.703	**0.7301**	0.717	0.6901	0.489	0.489	0.6961	0.6894	0.5468	0.4941	0.5735
#57	**0.988**	0.9835	0.962	0.9815	0.9858	0.9685	0.9578	0.9765	0.9744	0.9805	0.9749	0.9849	0.9873	0.9438
#58†	0.9965	**0.9968**	0.9961	0.9965	0.9965	0.9392	0.9961	0.9764	0.965	0.9967	0.9965	**0.9968**	**0.9968**	0.9949
*#59**	**0.9979**	0.9677	0.8832	0.9667	0.8989	0.8801	0.8133	0.7372	0.7215	0.9521	0.9375	0.9404	0.9092	0.7257
#60†	**0.7905**	**0.7905**	**0.7905**	**0.7905**	**0.7905**	**0.7905**	0.761	0.7878	**0.7905**	**0.7905**	**0.7905**	0.7851	0.7892	0.7824
#61	0.9715	0.9693	0.9709	0.9693	**0.977**	0.9731	0.9665	0.9766	0.9753	0.9699	0.9701	0.9695	0.9713	0.9654
#62	0.7826	0.7814	0.746	0.7755	**0.7861**	0.7578	0.7458	0.4753	0.4754	0.7791	0.7766	0.7789	0.7765	0.7341
#63	0.6263	0.5545	0.6131	0.5727	0.6121	**0.6384**	0.5899	0.5303	0.5091	0.5616	0.5465	0.5838	0.5949	0.5768
#64	0.9755	0.9615	0.9668	0.9667	0.9668	0.9614	0.9703	0.9737	**0.9772**	0.9685	0.9668	0.9703	**0.9772**	0.9579
#65	0.9175	0.9496	0.9722	0.9555	0.9614	**0.9887**	0.9778	0.9836	0.9833	0.9496	0.9388	0.9665	0.9663	0.9779
#66†	0.5704	0.5835	0.5899	0.5848	0.5948	0.5924	0.5929	0.5917	0.5855	0.5898	0.5879	0.5892	0.5911	**0.5986**
#67†	0.5184	0.5237	0.5268	0.53	0.5331	0.5294	0.5282	**0.539**	0.5266	0.5345	0.5306	0.5372	0.5337	0.5378
#68	0.9634	0.962	0.9678	0.9634	**0.978**	0.9707	0.9678	0.9415	0.9415	0.9634	0.9664	0.962	0.9634	0.9678
#69†	0.5612	0.5923	0.6051	0.6031	**0.6071**	0.5849	0.5943	0.5904	0.5977	0.6031	0.6004	0.591	0.5942	0.6057
#70†*	0.9613	0.9387	0.9719	0.9482	0.9486	0.9704	0.9704	**0.9799**	0.9704	0.9704	0.9704	0.9508	0.7478	0.9599
Avg:	0.8162	**0.8212**	0.8162	0.8205	0.8203	0.8141	0.8089	0.7668	0.76	0.821	0.8181	0.8173	0.8074	0.8061
Avg:*	0.7899	0.8000	0.7855	**0.8025**	0.7900	0.7863	0.7809	0.7506	0.7396	0.7980	0.7960	0.7988	0.7766	0.7664
Avg†:	0.7718	0.7786	0.7734	**0.7795**	0.7752	0.7692	0.7714	0.7307	0.7226	0.7782	0.7759	0.7720	0.7550	0.7631

Table 15
Cohen's kappa scores for small and medium data-sets.

	AdaB	LB	RF	GBM	RotF	ET	RP	RRF	RRET	XGB GDBT	XGB DART	LGB GBDT	LGB GOSS	CatB
#1†*	0.1372	0.1563	0.1553	0.1679	0.1625	0.1603	0.1637	0.154	0.1598	0.1622	0.1648	0.1649	**0.1696**	0.1493
#2†*	0.624	0.6217	0.5642	0.6284	0.599	0.4932	0.5909	0.5059	0.3326	0.6283	0.6237	**0.6303**	0.627	0.587
#3†	0.5756	0.6138	0.6007	0.6007	0.5796	0.5781	0.591	0.5403	0.5586	0.6138	0.6138	0.4871	0.0765	**0.6268**
*#4**	0.7052	0.7298	0.7325	0.7301	0.7424	**0.745**	0.7316	0.7272	0.7246	0.736	0.7328	0.727	0.7272	0.7241
#5†*	0.8387	0.8222	0.8291	0.8384	0.8662	**0.8774**	0.8472	0.7188	0.7358	0.8551	0.8564	0.8216	0.6065	0.538
#6†	0.6326	0.6379	0.533	0.6234	0.6406	0.5279	0.517	0.7567	**0.7624**	0.5817	0.5468	0.5666	0.6744	0.5686
#7	0.7801	0.7971	0.7987	0.8	0.7661	0.8008	0.4302	0.7927	0.7987	**0.8028**	0.8027	0.8027	0.8024	0.8027
#8	**0.1854**	−0.0048	0.1087	0.1325	0.0038	0.1364	0.1644	0.163	0.1398	0.1348	0.1333	0.0	0.0	0.0369
#9†*	0.3208	0.2897	0.2996	0.2748	0.2941	0.3672	0.2766	0.333	0.3115	0.2897	0.2897	0.3183	**0.3703**	0.3152
#10	0.4464	0.4256	0.4505	0.4348	0.4759	0.3636	0.3754	0.4142	0.4021	0.4435	0.4691	0.4527	**0.476**	0.4379
#11†*	**0.6578**	0.63	0.5771	0.6369	0.6092	0.4722	0.2245	0.3681	0.2437	0.5503	0.5522	0.5572	0.5562	0.5862
*#12**	0.9397	0.968	0.8765	**0.9686**	0.9548	0.8894	0.8752	0.7332	0.6871	0.9604	0.9598	0.9542	0.9491	0.8911
#13†	0.342	0.313	0.3035	0.3381	0.2833	**0.3655**	0.2888	0.3332	0.2844	0.3564	0.3217	0.3622	0.3411	0.273
#14†	0.0142	0.0	**0.0212**	0.0064	0.0	0.0095	0.0	0.0	0.0	0.0094	0.0157	0.0125	0.0032	0.0
#15†*	0.2404	0.0379	0.0	0.1875	0.0	0.0007	0.0	0.0002	0.0006	0.0	0.0	0.2364	**0.242**	0.1129
#16	0.3066	0.3263	0.317	0.3322	0.3303	0.2956	0.2953	0.1655	0.1348	0.3326	0.3287	**0.3339**	0.313	0.3066
*#17**	0.7037	0.7353	0.7239	0.7353	0.7353	0.7319	**0.7388**	0.7344	0.7232	0.7353	0.7353	0.7353	0.7334	0.7353
#18†	0.9439	0.9514	0.9686	0.9513	0.9584	**0.9825**	0.965	0.7706	0.7601	0.9549	0.9551	0.9578	0.9542	0.9719
#19†	0.7957	0.7608	0.7913	0.7703	0.7922	0.7779	**0.8075**	0.7577	0.7406	0.7774	0.7778	0.7605	0.7454	0.7667
#20†*	0.5773	0.6534	**0.6984**	0.6772	0.6765	0.6886	0.698	0.6519	0.5878	0.6885	0.6796	0.6822	0.6878	0.692
#21†*	0.6397	0.6885	0.6803	**0.6934**	0.6761	0.6695	0.6807	0.6507	0.6393	0.6885	0.6858	0.6848	0.6869	0.6662
#22†*	0.3825	0.4062	0.3773	0.4411	0.4285	0.3546	0.3843	0.3476	0.2448	0.4414	**0.4433**	0.4401	0.3992	0.2816
#23†	0.7205	0.7094	0.7354	0.6659	0.6706	0.676	**0.7419**	0.6163	0.6225	0.6972	0.6973	0.7161	0.6181	0.6791
#24†	0.2021	0.2111	0.1046	0.216	0.2305	0.0	0.0474	0.0386	0.0	0.2061	0.2098	0.2265	**0.2903**	0.2361
#25†	**0.7847**	0.7561	0.7752	0.7561	0.7663	0.7563	0.7654	0.3895	0.3205	0.737	0.7364	0.4982	0.2931	0.7164
#26	0.6634	0.6816	0.7112	0.6901	0.6825	0.7203	0.6652	0.7053	0.6899	0.7049	0.7046	0.7048	0.6997	**0.7281**
*#27**	0.9305	0.9392	0.9308	0.9392	0.9392	**0.9479**	0.9392	0.8964	0.8964	0.9392	0.9392	0.9392	0.9392	0.9392
#28	0.8157	0.8021	0.8411	0.8105	0.8486	0.8793	0.8321	**0.8915**	0.8859	0.8102	0.8016	0.8007	0.8024	0.8376
#29	0.95	0.95	0.95	0.95	0.95	0.95	0.95	**0.96**	**0.96**	0.95	0.95	0.95	0.91	0.95
#30†*	0.1243	0.1685	0.0847	**0.1725**	0.0743	0.087	0.0459	0.0674	0.0804	0.1042	0.1264	0.1243	0.1159	0.1294
#31	0.686	**0.7397**	0.7066	0.735	0.7153	0.7086	0.7152	0.7038	0.7123	0.7373	0.7302	0.735	0.7327	0.7152
#32	0.9389	0.9579	0.8626	0.9501	0.8648	0.8623	0.8576	0.2839	0.2837	0.9533	0.9384	0.956	**0.9612**	0.8809
#33†*	0.6709	0.6952	**0.7079**	0.6897	0.6128	0.6901	0.6876	0.6447	0.6566	0.6671	0.6259	0.7003	0.5827	0.5922
#34	0.7104	0.7334	0.6982	0.7333	0.7075	0.6163	0.6982	0.6562	0.5653	**0.7401**	0.7301	**0.7401**	0.7321	0.6898
#35	0.6567	**0.695**	0.6831	0.6877	0.676	0.6466	0.6784	0.6538	0.6694	0.6853	0.6929	0.6877	0.6858	0.6878
#36†	0.2256	0.2341	0.2316	0.2398	0.2381	0.2338	0.2353	0.1671	0.1561	0.2393	0.2372	0.239	**0.2422**	0.2389
#37	**1.0**	**1.0**	0.9814	**1.0**	**1.0**	0.9816	0.9583	0.9445	0.9445	**1.0**	**1.0**	**1.0**	**1.0**	0.9908
#38	0.6271	0.6714	0.705	0.5829	**0.8114**	0.7393	0.6757	0.7807	0.7993	0.6429	0.6464	0.6721	0.67	0.6879
*#39**	0.8573	0.8573	0.8435	0.8502	0.86	0.8645	0.8573	0.8627	0.8604	0.8519	0.8519	0.9585	0.9585	**0.9763**
#40†	0.9045	0.9132	0.8994	0.9035	0.9224	0.9224	0.9132	**0.9244**	0.9115	0.8978	0.8934	0.8995	0.8887	0.911
#41†*	0.9198	**0.9448**	0.7371	0.9268	0.8212	0.7624	0.7423	0.675	0.6637	0.8935	0.8712	0.854	0.8633	0.6768
#42	**0.9774**	0.9705	0.9739	0.9681	0.9678	0.9731	0.9656	0.7465	0.733	0.9691	0.9658	0.9723	0.9769	0.9525
#43†	0.8141	0.8287	0.7997	0.8076	0.8014	0.7262	0.804	0.7783	0.756	0.8243	0.8215	0.8293	**0.8301**	0.7986
#44	**0.9926**	0.9899	0.9839	0.9878	0.9901	0.9747	0.9843	0.5968	0.5981	0.9869	0.9832	0.9877	0.9887	0.9732
#45†	**0.7865**	0.7762	0.7619	0.7751	0.7696	0.6426	0.7278	0.7284	0.6491	0.7704	0.769	0.7596	0.7533	0.6895
#46	0.4381	0.4305	0.4595	0.4467	0.4101	0.4648	0.4226	**0.48**	0.4643	0.4695	0.4404	0.4705	0.472	0.4389
#47	0.9351	0.927	0.8759	0.9256	0.9046	0.9229	0.877	0.8827	0.9075	0.927	0.9229	0.9292	**0.9405**	0.901
*#48**	0.3519	0.4209	0.3535	0.4341	0.3718	0.3488	0.3465	0.3729	0.313	0.4265	0.4149	**0.4375**	0.3808	0.3644
#49†	0.8775	0.8764	0.8677	0.8692	0.874	0.8562	0.8635	0.3943	0.3934	0.8772	0.8729	0.8774	**0.8809**	0.8416
#50	**0.9843**	0.9788	0.9727	0.9758	0.9758	0.9657	0.9677	0.9535	0.9434	0.9768	0.9737	0.9823	0.9823	0.9626
#51†	**0.9998**	**0.9998**	**0.9998**	0.9996	0.9997	0.9915	0.9997	0.456	0.4558	0.9997	0.9995	0.9934	0.9934	0.9988
#52	0.4872	0.4744	**0.5095**	0.3855	0.4343	0.466	0.4174	0.4172	0.3512	0.4719	0.4315	0.454	0.4734	0.4625
#53	0.8671	0.8819	0.867	0.8804	0.8753	0.7623	0.8701	0.8673	0.8384	0.8809	0.882	**0.8916**	0.8894	0.8592
#54†	0.4097	0.3781	0.2778	0.3292	0.3535	0.267	0.2646	0.2129	0.1565	0.3631	0.3682	0.3311	**0.4571**	0.1415
#55†*	0.9092	0.9351	0.9169	**0.9391**	0.9202	0.9105	0.9339	0.4065	0.2257	0.9346	0.9311	0.9331	0.9346	0.8961
#56	0.5752	0.534	0.5667	0.5552	**0.5958**	0.5761	0.5359	0.236	0.2367	0.5449	0.5346	0.3215	0.2421	0.3611
#57	**0.9868**	0.9818	0.9582	0.9796	0.9844	0.9654	0.9536	0.9742	0.9726	0.9786	0.9724	0.9834	0.986	0.9382
#58†	0.9753	0.9773	0.9727	0.9754	0.9754	0.3018	0.972	0.8134	0.6884	0.9765	0.9757	0.9775	**0.9776**	0.9643
*#59**	**0.9954**	0.9291	0.7411	0.9282	0.7725	0.7367	0.5683	0.3702	0.2707	0.8974	0.8647	0.8689	0.8047	0.3771
#60†	0.4326	0.4326	0.4326	0.4326	0.4326	0.4326	0.3382	0.432	0.4326	0.4326	0.4326	0.4273	0.4302	**0.4557**
#61	0.943	0.9386	0.9419	0.9386	**0.954**	0.9462	0.933	0.9538	0.9516	0.9397	0.9403	0.9389	0.9427	0.9308
#62	0.7101	0.7084	0.6613	0.7006	**0.7147**	0.6772	0.6612	0.3058	0.3057	0.7055	0.7022	0.7053	0.702	0.6457
#63	0.5889	0.51	0.5744	0.53	0.5733	**0.6022**	0.5489	0.4667	0.4578	0.5178	0.5011	0.5422	0.5544	0.5344
#64	0.9473	0.9175	0.9286	0.9286	0.929	0.9169	0.9362	0.9471	0.9395	0.9319	0.9287	0.9364	**0.9511**	0.9096
#65	0.8761	0.924	0.9579	0.9328	0.9418	**0.9829**	0.9664	0.9746	**0.9829**	0.924	0.9081	0.9492	0.9492	0.9668
#66†	0.2968	0.3271	0.3141	0.3251	0.3303	0.3204	0.3236	0.3307	0.3171	0.3255	0.3245	0.3214	**0.3315**	0.3236
#67†	0.2208	0.2321	0.2377	0.2392	0.2333	0.2194	0.2375	0.2444	0.2182	0.2543	**0.2709**	0.2598	0.2466	0.2452
#68	0.9197	0.9159	0.9292	0.9195	**0.9523**	0.936	0.9293	0.8665	0.8651	0.9196	0.9259	0.9158	0.9191	0.9295
#69†	0.4257	0.4677	0.4839	0.4818	**0.4849**	0.4515	0.4661	0.4663	0.4632	0.4836	0.4798	0.4656	0.4699	0.4842
#70†*	0.9493	0.9185	**0.9634**	0.9308	0.9314	0.9612	0.9611	0.9611	0.9612	0.9614	0.9614	0.9352	0.6505	0.9469
Avg:	0.6693	0.6686	0.6554	**0.6713**	0.6631	0.6376	0.6347	0.5731	0.55	0.6696	0.6653	0.6641	0.6463	0.6375
Avg:*	0.6417	0.6451	0.6092	**0.6567**	0.6213	0.6076	0.5854	0.5325	0.4914	0.6387	0.6338	0.6525	0.6184	0.5799
Avg†:	0.5821	0.5819	0.5630	**0.5860**	0.5717	0.5295	0.5459	0.4753	0.4426	0.5784	0.5752	0.5729	0.5426	0.5457

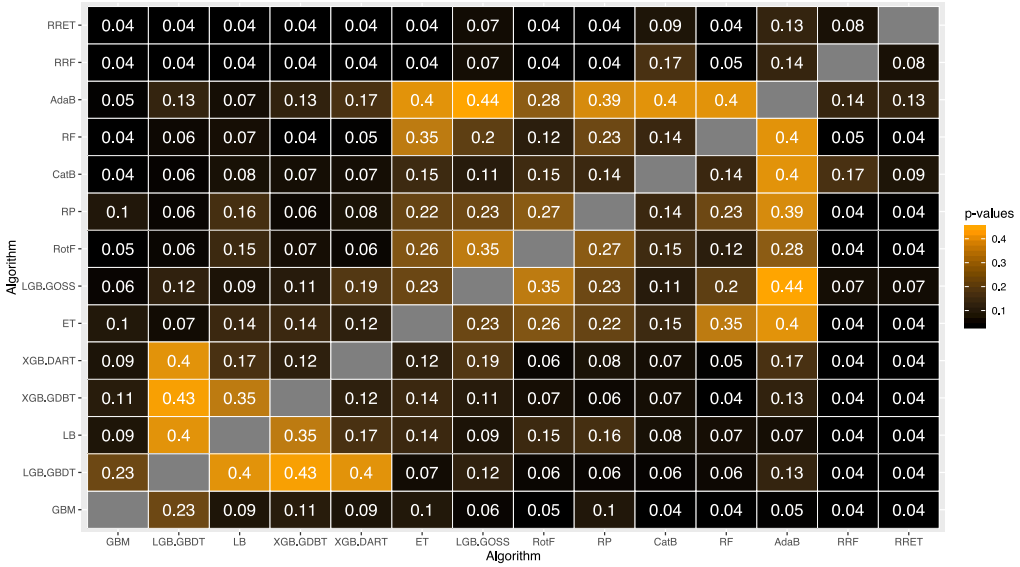

Fig. 4. P-values of Wilcoxon Signed Rank test for accuracy performance on data-sets with categorical features.

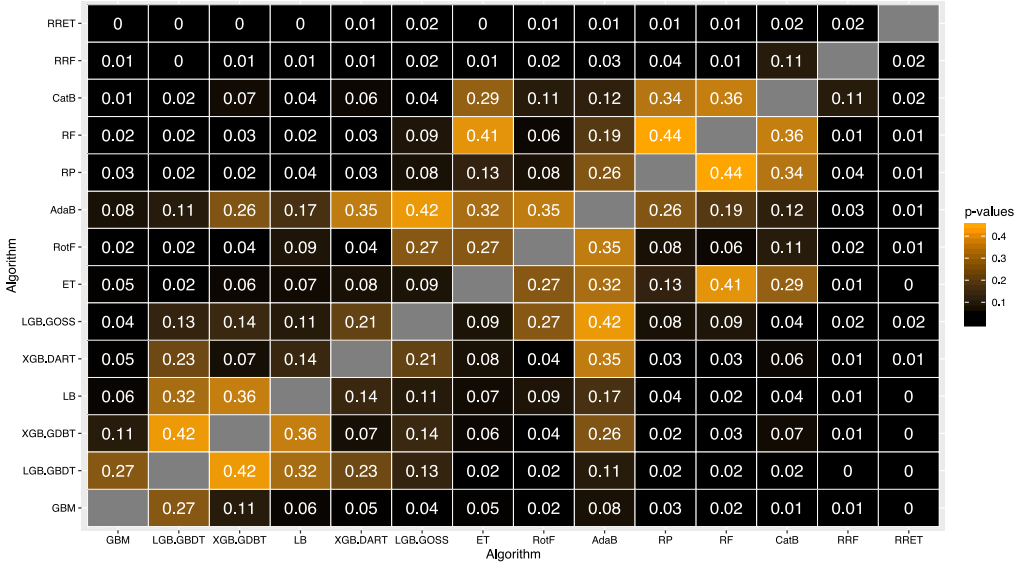

Fig. 5. P-values of Wilcoxon Signed Rank test for kappa scores on data-sets with categorical features.

algorithms with parallel computation support are executed in multiple cores.

Table 16 presents the results in terms of accuracy achieved by the selected ensembles for large problems. As before, bold-face font highlights the best result of each data-set. From these results, we can conclude the following:

- GBM is more accurate than all the other ensembles as shown in Table 16. It obtains the highest accuracy on average closely followed by LB and XGB. Additionally, GBM is the best ensemble in 4 out of the 6 big sized data-sets. Only in *skin* and *ECBDL14*, GBM is outperformed by AdaB.
- In general, the accuracy performance of all boosting type algorithms is very similar, close to each other. CatB distances itself a bit from the rest boosting algorithms with the worst accuracy among them. XGB ensembles are the second-most accurate regarding gradient boosting ensembles. LGB methods follow them in this order.
- Independent base classifier learning ensembles under-perform in terms of accuracy when compared with boosting-based algorithms. These algorithms might need more than 100 decision trees to converge into results similar to boosting. RotF and RF are the most accurate bagging-based ensembles. RRF and RRET obtain the worst results in terms of accuracy among all ensembles.

Table 17 shows Cohen's kappa scores of the featured ensembles for large problems. The general conclusions of the kappa performance are very similar to those results in terms of accuracy:

- GBM is still the best ensemble in terms of kappa coefficient, on average and for 4 different sets.
- RRF and RRET stand further out as the ensembles with the worst performance in terms of the kappa coefficient, especially for *skin*, *poker* and *ECBDL14* data-sets.
- Regarding the bagging-based algorithms, RotF obtains the highest kappa scores, but it is still faraway from the results of boosting methods. And, CatB is also the worst among the boosting methods.
- The differences between the best and worst results are even more pronounced with Cohen's kappa scores. The overall poor performance for *poker* and *ECBDL14* sets is clearly reflected in very low kappa

Table 16
Accuracy results for large data-sets.

	skin†	*kddcup**†	*ht-sensor*	*poker*†	*watch-acc*	*ECBDL14**†	*Avg:*
AdaB	**0.9995**	0.9522	0.907	0.4775	0.8126	**0.7825**	0.8219
LB	0.9983	0.9578	0.93	0.6143	0.9917	0.7739	0.8777
RF	0.9885	0.9521	0.6877	0.5359	0.6955	0.7256	0.7642
GBM	0.9991	**0.9587**	**0.9551**	**0.6883**	**0.994**	0.7753	**0.8951**
RotF	0.9931	0.9507	0.688	0.5333	0.7386	0.7415	0.7742
ET	0.9457	0.9452	0.6026	0.5071	0.7021	0.6756	0.7297
RP	0.8483	0.9493	0.6764	0.5202	0.7049	0.7444	0.7406
RRF	0.8006	0.9509	0.6834	0.5336	0.682	0.6668	0.7196
RRET	0.8006	0.9454	0.6405	0.5282	0.6305	0.6667	0.702
XGB-GDBT	0.9985	0.9577	0.8852	0.6389	0.9455	0.7745	0.8667
XGB-DART	0.9985	0.9577	0.8838	0.6385	0.9457	0.7713	0.8659
LGB-GBDT	0.9988	0.8541	0.8838	0.6405	0.9495	0.7767	0.8506
LGB-GOSS	0.9987	0.8141	0.8832	0.6409	0.95	0.7763	0.8439
CatB	0.9869	0.9546	0.6887	0.6066	0.7631	0.7569	0.7928

scores. Some of such scores are around 0, which implies random guessed predictions.

Table 18 records the seconds spent in the training phase of the ensembles with big data-sets. The symbol ‡ indicates no support for multi-thread parallel computation. As marked by ‡, RotF and RP were trained with a single job for *ECBDL14* set, because their parallel executions fail due to excessive memory consumption. From the training runtimes in this table, we extract the following statements:

- As shown in Table 18, ET has the on-average fastest training runtime and the fastest training runtime in 2 of the data-sets with the greatest number of instances. The second on-average fastest ensemble LGB-GOSS obtains the shortest runtimes in *skin*, *ht-sensor* and *ECBDL14* sets. And, RF has the fastest training with the data-set with the greatest number of classes *kddcup*.
- The on-average slowest training phase is the RotF training phase manly due to its memory consumption problems and the single-core execution of *ECBDL14*. However, it definitely has the largest training runtime among all bagging-based ensemble, even if *ECBDL14* is excluded from the comparison. The PCA feature extractions performed in each iteration are the main reason for its high training runtime and memory consumption, especially when there are so many features as in *ECBDL14*.
- The most accurate ensemble GBM has the slowest training with these big data-sets among all boosting ensembles. When compared to other gradient boosting methods, their training runtime differences are overwhelmingly huge. The second slowest GB-based ensemble XGB-DART is on average 23 times faster than GBM. Both LGB-GBDT and LGB-GOSS has really impressive, low training runtimes, even compared to much simpler algorithms such as RF and ET. The training runtimes of CatB lay in between of XGB and LBG runtimes.
- These low training runtimes of XGB, LGB and CatB make intensive tuning of their numerous parameters possible, which may allow them to catch up with GBM prediction capabilities and possibly outperform any other ensemble.

Table 19 shows the time in seconds dedicated to predicting the test partitions of the large size data-sets. From these results, we draw the following conclusions:

- LGB-GOSS and LGB-GBDT are the fastest and second-fastest ensemble in processing the test partitions with an amazing difference when compared to others. XGB, ET, and RF are also very fast.
- The on-average slowest ensembles in their test phases are RP, RRF, RRET, and RotF, in this order. This is mainly due to their poor results obtained for *ECBDL14*. For rotation space-based algorithms (RRF, RRET, and Rot), the input transformation takes a big chunk of time during the prediction phase, especially when there are so many features. Additionally, RP does not seem to be well optimized for high dimensional data-sets such as *ECBDL14*.
- Among boosting ensembles, LB has the slowest test phase. And, CatB has the greatest testing runtime regarding the gradient boosting machines, even though it is the overall fastest for *poker* and *watch-acc* testing.

6. Lessons learned

Below, we expose all the lessons learned from the revision of the featured ensembles and their software tools, in accordance with the results of the conducted comparison study:

- As stated before [125], simple methods usually obtain almost the same or even better results than more sophisticated ones. A good example is the much worse performance of RRF and RRET compared to the original RF and ET. The more advanced GBM-based methods, such as CatB, LGB-GOSS, and XGB-DART, do not outperform the predictive capabilities of standard GBM, LGB-GBDT, and XGB-GBDT. Furthermore, there is no significant difference between a few of these more complex ensembles and other simpler methods, such as RF or ET.
- Contrary to the statement in [6], RF does not prove to be the best ensemble in our experimental studies. AdaB also does not exhibit as high a level of performance as indicated by other studies [6,21]. The majority of GBM-based ensembles and RotF are usually ranked better than these two methods, with or without statistical ratification. This might be explained by the fact that GBM based methods have not been examined in [6], mainly because their modern and useful implementations (e.g. XGBoost) have just started to emerge.
- The predictive performance of XGB-GBDT is ranked as the best ensembles in most scenarios, but its superiority is statistically significant for only a few comparisons. For large data-sets, XGB-GBDT performs similar to GBM, which has the best performance, but with a much faster training phase. In addition, the XGB software package has the greatest availability and the greatest applicability to concrete problems of machine learning. Its large community is constantly providing utilities and documentation helpful for potential users.
- In our empirical studies, RotF is one of the best bagging-like methods and significantly superior to RF in multiple cases, although RotF has much less recognition.
- The methods with categorical feature support have not proven to be better than the rest in the scenarios with categorical inputs of our experimental framework. Ordinal encoding has not harmed the predictive capability of the ensembles in comparison to the more advanced techniques of LGB and CatB. Perhaps for that reason, the vast majority of software does not support this type of input. If categorical features are essential for the problem knowledge, LGB is recommended due to its overall good performance.
- LGB is an impressively fast method to train in both sequential and parallel paradigms for standard and big data-sets. It is comparable to really fast and simple methods, such as ET. This allows a faster

Table 17
Cohen's kappa results for large data-sets.

	skin†	*kddcup*†*	*ht-sensor*	*poker*†	*watch-acc*	*ECBDL14*†*	*Avg:*
AdaB	**0.9986**	0.9241	0.8602	0.0941	0.7807	**0.4916**	0.6915
LB	0.9949	0.931	0.8947	0.2627	0.9903	0.4609	0.7558
RF	0.9656	0.9198	0.5321	0.0792	0.6436	0.2586	0.5665
GBM	0.9974	**0.9348**	**0.9324**	**0.4119**	**0.993**	0.4655	**0.7892**
RotF	0.9791	0.9173	0.5316	0.0718	0.6944	0.3877	0.597
ET	0.8224	0.9098	0.4073	0.0138	0.6503	0.037	0.4734
RP	0.386	0.9144	0.5156	0.0437	0.6556	0.3804	0.4826
RRF	0.1134	0.9219	0.5257	0.0713	0.6268	0.0004	0.3766
RRET	0.1134	0.9124	0.4612	0.0594	0.5654	0.0000	0.3519
XGB-GDBT	0.9954	0.9334	0.8276	0.3141	0.9362	0.4639	0.7451
XGB-DART	0.9954	0.9334	0.8254	0.3133	0.9364	0.4548	0.7431
LGB-GBDT	0.9964	0.7575	0.8256	0.3193	0.9409	0.4672	0.7178
LGB-GOSS	0.9961	0.7238	0.8245	0.326	0.9415	0.4666	0.7131
CatB	0.9606	0.9236	0.5323	0.2464	0.7228	0.4043	0.6317

Table 18
Training runtimes measured in seconds for large data-sets.

	skin†	*kddcup*†*	*ht-sensor*	*poker*†	*watch-acc*	*ECBDL14*†*	*Avg:*
AdaB‡	35.3804	213.3364	478.9176	185.9128	1225.9183	23868.4214	4334.6478
LB‡	12.854	2023.1935	1336.0385	1465.8477	7537.307	23230.5104	5934.2918
RF	1.0536	**3.9642**	7.202	6.9467	25.3584	69.2577	18.9638
GBM‡	10.2448	1917.6497	602.3654	1988.9792	7179.2572	35849.3312	7924.6379
RotF	3.9587	41.9409	50.3434	35.4985	152.5229	69259.188‡	11590.5754
ET	0.5644	4.3084	2.9245	**5.4038**	**17.2237**	59.6703	**15.0159**
RP	1.039	5.6681	14.4696	7.9083	36.8189	10084.0606‡	1691.6608
RRF	1.058	10.713	11.5993	9.5366	50.2278	4241.1853	720.72
RRET	1.0041	9.3412	5.6159	7.2867	24.1414	2862.9395	485.0548
XGB-GDBT	2.5296	102.5739	72.8956	80.5319	605.6499	459.4339	220.6025
XGB-DART	5.5174	241.5984	122.1177	218.2411	881.5627	534.814	333.9752
LGB-GBDT	0.6012	15.6323	3.3805	45.456	45.8715	58.6994	28.2735
LGB-GOSS	**0.516**	16.267	**2.8211**	25.7159	27.4412	**37.4945**	18.3759
CatB	6.3118	297.3267	32.8895	69.8049	178.3804	484.4987	178.202

Table 19
Test runtimes measured in seconds for large data-sets.

	skin†	*kddcup*†*	*ht-sensor*	*poker*†	*watch-acc*	*ECBDL14*†*	*Avg:*
AdaB	0.583	8.4193	2.7293	8.4683	24.037	18.5257	10.4604
LB	0.1563	22.8974	2.1884	10.1133	22.6065	55.2674	18.8715
RF	0.1108	1.035	0.3488	0.8623	2.1343	1.7903	1.0469
GBM	0.0962	4.2899	0.6904	5.0504	10.4724	4.69	4.2149
RotF	0.3955	1.4164	0.8741	1.2025	2.4545	150.5292	26.1453
ET	0.1101	0.9981	0.366	0.9081	1.9052	1.6141	0.9836
RP	0.1756	1.4449	0.5474	1.172	3.5784	256.1519	43.845
RRF	0.4191	2.3695	1.064	1.8943	4.7679	224.4544	39.1616
RRET	0.4713	2.146	1.0836	1.7898	5.3136	190.6765	33.5801
XGB-GDBT	**0.0179**	0.6644	0.2231	0.6102	1.5628	2.8429	0.9869
XGB-DART	0.0234	0.7282	0.2378	0.6982	1.6583	2.3544	0.9501
LGB-GBDT	0.0198	0.2744	**0.1746**	0.5847	1.4306	0.728	0.5354
LGB-GOSS	0.0198	**0.2733**	0.1747	0.6185	1.4368	**0.5104**	**0.5056**
CatB	0.0533	0.6444	0.387	**0.4459**	**1.2596**	38.7862	6.9294

and more agile parameter tuning, which is critical for such a GBM-based method with plenty of parameters. Moreover, the quality and community of the LGB open source project are comparable to those of the XGB package.

- Software optimization, usability, and availability should be used as additional quality measurements of any algorithm, including ensembles [4]. These software features of XGB and LGB are some of the main reasons for their success. However, good quality software, such as optimized implementations of existing algorithms, is often less valued than new proposals, or not considered as sufficient scientific advances. Also, many researchers do not tend to publish the code of their new proposals, which can cause a loss of interest/popularity and lead to notable differences in subsequent implementations done by third parties. Recently, Extreme Usability has become popular in software engineering, which is based on incorporating users into early stages of software development in pursuit of high usability [126]. However, scientific software development is far from adapting these software engineering methods.

7. Practical perspectives

We now explain some practical perspectives for ensemble learning:

- First, Section 7.1 develops the concept of ensemble diversity, and discusses some related techniques, such as ensemble pruning, label switching or the dynamic selection of learners.
- Next, Section 7.2 stresses on the relevance of ensembles for data preprocessing tasks, particularly for noise filtering, imputation of missing values and data reduction.
- Then, Section 7.3 explains several strategies for ensemble fusion, including weighting aggregations, meta-learners, and combinations of multiple learning systems.
- Error correcting output codes are introduced in Section 7.4 as an alternative multi-class decomposition approach for multi-class classification tasks.
- Finally, we elaborate on the relevance of hyper-parameter tuning in Section 7.5, giving empirical evidences of this claim, and calling

for efficient AutoML means to automate the process of configuring ensembles, particularly in problems of large dimensionality.

7.1. Ensemble diversity and selection

The diversity among the different base learners is one of the main reason of the effectiveness of ensemble techniques [1,4]. A multitude of diversity promotion techniques can be found in the literature [4], which are categorized as algorithm-level, data-level and hybrid techniques [1]. Algorithm-level techniques directly use different base learners, differently tune their parameters or introduce randomness in the base algorithms to achieve different outputs. Data-level algorithms perform small modifications to the training data-set to promote different responses from same-type base learners.

Some diversity promotion techniques have been already explained along the reviewed ensembles. There are certain techniques worth to highlight:

- Randomness incorporation in the algorithmic processes of base learners, such as random selection of features and cuts in RF [26] and ET [27], is a key element of diversity promotion.
- Resampling, both bootstrapping [17] and weight-based [23], is one of the most popular techniques [18,26–28].
- Ensembles based on feature space perturbations are also very common, including methods based on feature subspace (Random Subspace and RP [28]) and feature-space rotations (RotF [29] and RRE [30]).
- Manipulations of the target attribute are a very interesting and not-highly explored approaches of diversity promotion. Label switching algorithms [11,43,44] change the labels of randomly selected sample as a technique of diversity promotion.
- Multi-class decomposition [127,128], such as Error-Correcting Output Codes (ECOC) [129], can be also considered as diversity promotion techniques [4]. We will further elaborate on ECOC in Section 7.4.

Another measure to control ensemble diversity is ensemble pruning or selection of learners [130,131]. An excessive number of estimators could spoil the ensemble diversity and usually leads to over-fitting. Ensemble pruning or selection [130,131] intends to choose the best built learners in order to promote the diversity and precision of the ensemble. Base estimators are evaluated and selected with help of a selection criterion or function [130,132], based on concepts of competence [133,134] and diversity [135]. This selection can be performed statically or dynamically:

- Statically pruning selects a fixed group of learners in order to reduce complexity while increasing or maintaining the precision [130].
- Dynamic selection [131,132] chooses different subsets of estimators for different testing samples without discarding any models. Each learner covers its competence area of the feature space [131].

7.2. Ensembles for data preprocessing

As stated several times along the paper, ensemble techniques have been very useful for data preprocessing tasks. Bagging and boosting methods can be used by themselves for some preprocessing problems thank to their algorithmic natures. For example, decision forests can be used as implicit feature selectors with the ranking of the features given for each decision tree of the ensemble [3]. The weights of boosting algorithms can be an indicative of core instances, easy- or difficult-to-learn instances and even, noisy samples.

Additionally, a multitude of ensemble-based techniques have been developed especially for data preprocessing. Traditionally, noise filters have relied on ensemble mechanisms. Ensemble Filter [136], Cross-Validated Committees Filter [137] and Iterative-Partitioning Filter [138] are well-known ensemble-based noise filtering algorithms. Novel ensemble-based filters have recently been published [12,139].

Concerning missing values, ensemble methods are popular kernel-based imputation techniques [140]. Random Forest [141], Decision forest [142] and Bagging [140] are some examples. Other imputation approaches have been designed with ensemble concepts [143,144].

Ensembles are also used for data reduction. Several ensemble-based approaches have been designed for feature selection [145–147]. Boosting algorithms have adapted for instance selection [148].

A special mention should be made of ensembles used in singular data preprocessing problems such as class imbalance classification [124].

7.3. Ensemble fusion

As explained in previous sections, bagging and boosting algorithms with the most software representation commonly aggregate their predictions with simple yet effective fusion methods, such as weighted and majority voting. However, there are much more fusion techniques with higher complexity.

For example, several weighting aggregations establish the weights of the classifiers in proportions to its performance [149]. Other fusion methods combine probability vectors returned by classifiers with concepts of probability distribution [150], Bayesian or Naive Bayes algorithms [151]. Also noteworthy fusion models are Vogging [152], the use of order statistics [153] or logarithmic opinion pool [154]. Recently, new classifier fusions have been designed based on Contextual Reliability Evaluation [155], Evidential Reasoning [156] and on scalable robust probabilistic aggregations [157].

Another important segment within the ensemble fusion is the meta learners, which learn from the base classifiers in order to improve the final outcomes. Stacking [158] is the most popular meta-learning algorithm. Stacking meta-learner attempts to infer the reliability of the base learners by learning from the predicted labels or probabilities of each classifier as input and the real classes as output. Stacking is generally used to combine heterogeneous models learnt with the same data. There are many approaches to design Stacking systems with multiple learners [159–162]. Beside Stacking, Arbiter and Combiner trees [163,164] and Grading classifiers [165] are some other relevant meta-learners.

Lastly, the combination of ensembles with other techniques to promote a greater heterogeneous diversity is highly relevant to achieve great success in real applications. For example, the fusion of ensembles and deep learning has proved to be very accurate in the MNIST problem [166]. Data fusion and the combinations of multiple learning systems has also resulted beneficial for human activity and health monitoring [167].

7.4. Multi-class decomposition and Error-Correcting Output Codes

Several classification algorithms are designed to learn only from binary class data-sets. However, multi-class classification problems are really common in real-life applications. To enable these binary classifiers, the multi-class problem is decomposed into several binary classification tasks. Then, each binary problem is solved with a binary classifier and the solutions are combined with an ensemble [127,128]. This technique has been attributed with a reduction in computational complexity and the possibility of parallel learning [128,168,169].

The decomposition is usually represented with a code-matrix [127], where each row represents the codeword of each class given by the desired outputs of the binary classifiers (columns of the matrix). The predicted class of a new sample is given by the minimum distance to each class codeword.

There are two main groups of performing code-matrix decomposition [170]. The first group seeks, with a fixed code matrix, an optimal ensemble of binary classifiers. And, the second type simultaneously optimizes the sets of classifiers and the code matrix.

Within the first scheme, One-vs-One and One-vs-All are the simplest decomposition techniques [128]. One-vs-One decomposition builds a binary classifier for each pair of classes. One-vs-All decomposes the multi-class problem into as many binary problems as number of classes, where each class is learned against the rest.

As a more complex decomposition, ECOC [129] encode classes to codewords, where the bits represent the expected responses of the ensemble classifiers. Normally, ECOC techniques include additional redundant bits in order to correct possible classification errors. There are several coding strategies in the literature without a clear winner [127, 169].

In second group, Data-driven error correcting output coding (DECOC) [171] takes problem properties into consideration in order to optimize both the number of classifiers, their combinations and the codewords in the code matrix. In DECOC strategies, the different binary classifiers are selected by measuring their separability quality. Additionally, other approaches combine boosting scheme with ECOC concepts to optimize the code-matrix and the base learners in the ensemble [172].

7.5. Hyperparameter tuning in ensembles

Finally, we discuss about the configuration of hyperparameters in ensembles. In short, hyper-parameter tuning techniques pursue to automate and perform efficiently the search for the hyperparameters' values that make the model at hand perform best for a certain learning task. An exhaustive search over a grid of possible values can be a simple yet practical solution for discrete search spaces comprising a few hyper-parameters. However, in more complex hyper-parameter tuning scenarios other algorithmic approaches have been reported in the literature, including wrapper methods that rely on population-based algorithms [173], model-based (e.g. Bayesian) optimization [174,175], or other assorted choices such as iterated F-race [176]. The growing body of contributions dealing with increasing levels of sophistication in the automated selection of model properties have lately sparkled the so-called field of Automated Machine Learning (AutoML [177,178]), which spans the domain in which the search is done towards the optimal selection of the learning algorithm itself.

When it comes to ensemble learning, hyperparameter tuning is often overlooked in practice. Instead, hyperparameters are set to recommended values found in related works, established as per the experience of the scientist, or performed manually by checking a reduced number of settings in a trial-and-error fashion. The main reason behind this usual practice can be found in the concept of *tunability* [179], i.e. the performance gain that can be obtained for a given Machine Learning algorithm by tuning its hyperparameter values instead of relying on its default configuration as per the software package in use. The results discussed in Section 5 evince that the default configuration of the ensembles under study lead to good performance scores, suggesting narrow improvement gaps achievable via hyperparameter tuning. Recent works on hyper-parameter tuning with specific ensemble learning algorithms are also aligned with this intuition, as in e.g. [180] for random forest ensembles. However, this question remains uninformed with empirical evidence, motivating us to complement this work with further experiments to shed light on this matter.

For this purpose, we have designed a small experiment comprising a subset of the ensembles under consideration, namely, those based on gradient boosting: GBM, XGB GDBT, XGB DART, LGB GDBT, LGB GOSS and CatB. Likewise, we filter out the 40 data-sets that are smaller in size from the previous set of experiments, over which we measure the accuracy and Cohen's Kappa of such ensemble learning algorithms *with* and *without* hyper-parameter tuning. In order to avoid any bias in our conclusions with respect to the stochastic nature of most wrapper methods existing in the literature, we instead resort to an exhaustive grid search over the hyperparameter values of each ensemble listed in Table 20. Those hyperparameters not shown in this table are set to the default values imposed by the software package implementing the corresponding ensemble learning algorithm. Performance scores are averaged over 5 stratified folds for every data-set, which are ensured to be the same partitions for the model with default parameters and the model with hyper-parameter tuning.

Table 20
Grid search values explored by the hyperparameter tuning of every gradient boosting based ensemble learning algorithm.

Ensemble algorithm	Hyperparameter	Explored range of values
All	*n_estimators*	[10,30,50,70,90,110]
	max_depth	[1,3,5,10]
	learning_rate	[0.01,0.05,0.1]
GBM, XGB GDBT, XGB DART, LGB GDBT, LGB GOSS	*subsample*	[0.1,0.3,0.5,1.0]
GBM	*max_features*	[0.3,0.5,0.7,1.0]
XGB GDBT, XGB DART, LGB GDBT, LGB GOSS	*colsample_bytree*	[0.3,0.5,0.7,1.0]
XGB GDBT, XGB DART, LGB GDBT, LGB GOSS, CatB	*reg_lambda*	[0., 1e−2, 0.1, 1]
CatB	*colsample_bylevel*	[0.1,0.5,1.0]

The results from this study are aggregated and summarized in Tables 21 (accuracy) and 22 (Cohen's Kappa), whereas detailed outcomes obtained for every data-set are collected in Appendix B. These tables depict the mean score for each ensemble learning algorithm averaged over all the data-sets (i) with the default configuration and (ii) with hyper-parameter tuning. The P-values rendered by a Wilcoxon Signed Rank test applied to the performance scores are also included to account for the statistical relevance of the performance gaps reported between both counterparts. It is straightforward to observe that hyper-parameter tuning *does* yield performance gains in terms of both performance scores, with statistical relevance for $\alpha = 0.05$. We note, however, that performance differences are not high, and vary among different learners, with concurs with the consensus on the low tunability of ensembles noted above. This small performance improvement could not compensate for the computational resources needed for hyper-parameter tuning in practical settings, specially in scenarios dealing with large data-sets.

On a closing note, we advocate for deeper studies on the notion of tunability for ensembles, elaborating on the relative impact of each hyperparameter in the generalization performance of the ensemble model. It is only by advancing over this crucial aspect when hyperparameter tuning approaches will be of practical use in realistic settings, specially towards addressing Big Data problems. In these cases, long learning runtimes can outweigh the narrow performance gaps that appear to be achievable by such tools. To avoid this issue, hyperparameter tuning tools must be improved in two directions:

(i) They must focus on those hyperparameters that, when optimized, imprint larger performance gains in the ensemble algorithm at hand; and
(ii) They must be designed for tasks in which the evaluation of the generalization performance for a given hyperparameter configuration is computationally expensive, thus requiring the adoption of modern computation paradigms (e.g. Big Data).

We foresee a long road ahead in these directions.

8. Opportunities

This section outlines some opportunities for the ensembles in machine learning. First, we highlight the lack of ensembles designed for Big Data problems. Then, we present the concept of Explainable Artificial Intelligence and the role of the ensembles in this field. Finally, robustness and stability are stressed out as relevant features of ensemble techniques.

Table 21
Average accuracy scores of gradient-boosting ensembles with and without parameter tuning for a subset of the considered small and medium data-sets.

	GBM	XGB GDBT	XGB DART	LGB GDBT	LGB GOSS	CatB
Default configuration	0.777	0.779	0.779	0.776	0.773	0.786
Hyper-parameter tuning	0.801	0.797	0.796	0.785	0.780	0.801
Wilcoxon *P*-value	1.73e−6	5.60e−6	3.79e−6	2.70e−5	4.01e−5	1.23e−5

Table 22
Average Cohen's kappa scores of gradient-boosting ensembles with and without parameter tuning for a subset of the considered small and medium data-sets.

	GBM	XGB GDBT	XGB DART	LGB GDBT	LGB GOSS	CatB
Default configuration values	0.580	0.588	0.588	0.583	0.566	0.594
Hyper-parameter tuning	0.618	0.611	0.618	0.594	0.576	0.608
Wilcoxon *P*-value	3.79e−6	2.70e−5	1.23e−5	5.96e−5	5.83e−5	4.01e−5

8.1. Ensembles for Big Data problems

The vast volume of information and its exponential rate of generation have brought new computational and learning challenges, which are collectively referred to as Big Data problems. These problems cannot be tackled with traditional machine learning techniques, due to the big amount of data, their complexity and variety, and/or the velocity of generation [123]. Therefore, new hardware, new algorithms and software optimizations are needed to processing Big Data. One of the most widespread practices for this purpose is the design of algorithms tailored for distributed computing architectures.

Ensembles can be an important asset in Big Data, thanks to their high predictive capabilities with a good balance between specification and generalization, and the possibility of parallel training of their base learners with partial information. However, only a few ensembles are adapted to tackle these problems. As shown in Section 3, XGBoost [32,90], LightGBM [34,116], GBM [31,181], AdaBoost [23,77] and RF [26,92] are the only ensembles with software implementations for Big Data. Therefore, new distributed ensemble algorithms are needed to address Big Data problems by harnessing the inherent parallelization and computational efficiency of these learning models.

8.2. Explainable Artificial Intelligence (XAI) and ensembles

Another road ahead for ensemble learning aims at the explainability of their captured knowledge and prediction process for different audiences, disregarding their prior expertise in Machine Learning. Although most ensembles are built on top of weak tree learners that are explainable on their own, their combination as per different ensemble strategies gives rise to black-box models that require additional techniques for eliciting understandable information about their internals. Explainable Artificial Intelligence (XAI) is the recently forged term to refer to all methods devised for explaining, to a given audience, how decisions are produced by models and systems with Artificial Intelligence at their core, specially machine learning algorithms [182,183]. Explainability is particularly relevant for certain application domains where human intervention and decision making is particularly critical, such as medicine, law or defense. Causability is another important concept connected to explainability for these application domains, particularly for medicine [184]. Causability refers to the quality measurements of the explanations produced by the learning models.

Similarly to other non-transparent learners (e.g. Deep Learning models), ensembles currently undergo a vibrant momentum in regards to literature proposing different means to explain their predictions, in a richer fashion than the information produced by the ensemble by itself (e.g. feature importance). The simplification of already trained ensembles is among the schemes that have been more extensively applied, for many years now [185,186] and more recently [187]. However, most of the activity noted to date focus on bagging and boosting ensembles, with a scarcity of proposals dealing with more sophisticated strategies for ensemble learning and fusion, such as stacking [188,189] or multiple classifier systems [190].

The transformation of decision forests to yield single trees that ensure explainability without any penalty in performance is another interesting research line to pursue in regards to the explainability of tree ensembles. The work in [191] is a recent example illustrating the increased explainability of such a transformation. The algorithm proposed in this work constructs a set of rule conjunctions that represent the original decision forest to be explained. Once organized hierarchically, the resulting tree not only preserves the performance of the decision forest, but also describes the prediction process in a human understandable fashion. As noted in this work, the extrapolation of this approach to ensembles of dependent learners (e.g. boosting) is among the most promising research lines to follow in the near future.

Finally, attention must be paid to the comparison of the quality of explanations given by XAI techniques specifically designed for tree ensembles, to those produced by model-agnostic methods. Such a comparison study should consider the challenges beneath the evaluation of explanations themselves [192], embracing advances in user mental models and measures to quantify the user satisfaction, trust and reliance, as well as the explanation usefulness, sufficiency and understandability [193].

8.3. Robustness and stability of ensembles

Robustness is the property of a learning method for dealing with imperfections in the data, such as noise or missing values. Noisy data is really common in real-life problems. Ensemble algorithms have different levels of noise tolerance. Some traditional boosting methods, such as AdaBoost, are sensitive to noise [1], which has encouraged the design of robust boosting proposals [194–196]. Newer boosting models such as XGBoost turn out slightly more noise-tolerant [197]. Other ensembles have been used as noise filters [10,12,123]. Concerning missing values, XGBoost and CatBoost can handle sparse data as mentioned in Section 3.

The stability of a learning technique is the property of obtaining the same predictions for a given test set when training with different subsets of the same problem. The instability of predictions due to slight data changes raises doubts about the reliability and validity of the ensemble model. This property has not been studied for previous and newer ensemble algorithms. New proposals could exploit the strengths and improvements of the already existing ensembles in regards to robustness and stability. We advocate for alternative approaches to be explored for achieving these very desirable properties.

8.4. Meta-learning for ensembles

Finally, another research direction attracting a great deal of attention lately is *algorithm selection*, which refers to those methods aimed at identifying which learning algorithms perform well when addressing a given task on a certain dataset [198]. Among the different approaches that can be followed for this purpose, we pause at *meta-learning*, which focuses on modeling the similarities and dependencies between datasets/tasks so as to select, given a new dataset/task, the learning algorithm that potentially performs best among a portfolio of choices [199–201]. By modeling the similarities between datasets/tasks, meta-learning algorithm selection procedures avoid acknowledged downsides of AutoML tools, which require notably higher computational resources for evaluating candidate algorithms over the dataset/task under target.

In Section 7.5 we have remarked and supported with empirical evidence that hyperparameter tuning makes a difference in ensemble learning. Recent promising results [202] suggest that a step further should be taken towards efficient meta-learning schemes suited to automatically select ensemble learning algorithms. In particular, we foresee an exciting research opportunity in the design of meta-learning techniques capable of jointly automating selection and hyperparameter configuration of ensemble learning algorithms. Furthermore, efficient methods to model and quantify the similarity between datasets/tasks are still sought, given the noted complexity of meta-feature extraction techniques used so far in related studies [203,204]. All in all, it is our belief that meta-learning will play a major role in the research landscape of ensemble learning in years to come.

9. Concluding remarks

In this work we have revisited 14 recent and recognized bagging and boosting based ensembles for supervised leaning tasks and their software packages. These algorithms and their recent versions have been explained in detail. We have described the different online-available software tools featuring these ensembles in terms of their supported features and the programming environments. Different experimental scenarios have been analyzed and discussed to identify the strengths and limitations of the methods in terms of predictive performance and computational efficiency. Over the course of these experimental scenarios, we have found out that:

- XGBoost has resulted in the best ensemble according to predictive performance for multiple scenarios, but without critical differences.
- LightGBM has an impressive fast learning process for both standard and big data-sets in sequential and parallel computing paradigms.
- Among bagging-like ensembles, RotF is remarkable due to its good prediction capabilities. RF has showed greater efficiency while obtaining similar results as RotF.
- We have also highlighted the importance of good quality software for ensembles in machine learning. XGBoost and LightGBM packages are good examples of the well-optimized and designed software, which has contributed positively to their popularity.

Furthermore, we have outlined some practical perspectives and opportunities for ensemble learning. Specifically, we envision an exciting future beyond the proposal of new ensemble strategies. The larger scales at which data is produced nowadays call for further efforts towards extrapolating different ensemble variants to the Big Data paradigm. The parallelization and efficiency of ensembles seem to be a perfect match for coping with learning tasks of unprecedented complexity. We have also highlighted the need for hyperparameter tuning tools specifically tailored for optimizing ensemble learning algorithms when applied to tasks of large dimensionality. Another direction identified in our prospects stems from the contextual restrictions imposed by certain application domains, which dictate that ensemble models do not only need to perform well, but must be also interpretable for the user of its decisions. Studies delving on understanding and improving the robustness and stability of ensembles are also needed, so that new ensemble strategies can be developed for imperfect data. Finally, we have also discussed on the opportunity of meta-learning algorithms for the automated selection of ensembles, along with challenges and open issues related to this research line.

All these directions underscore our firm belief that research maturity in ensemble learning should be complemented by further studies focused on their practicality, so that the potential of ensemble learning leads to valuable decisions in real-world application contexts. For this to occur, we advocate for future research towards high-quality software frameworks, explainability tools and Big Data implementations. Accomplishments in these research lines will make the case for the continued success of ensemble learning.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Spanish National Research Project TIN2017-89517-P, and by a research scholarship (FPU) given to the author Sergio González by the Spanish Ministry of Education, Culture and Sports. Javier Del Ser would also like to thank the Basque Government for its funding support through the ELKARTEK and EMAITEK programs, as well as the Consolidated Research Group MATHMODE (IT1294-19) granted by the Department of Education of this institution.

Appendix A. Efficiency results over standard-size problems

The computational efficiency of the different ensembles over small- and medium-size data-sets is presented as follows. Table A.1 shows the time spent in the training phase of the ensembles with small and medium data-sets. These runtimes correspond to sequential training, as the data-sets are not big enough to benefit from parallel training. Table A.2 records the runtime dedicated to predict the test partitions of the small and medium data-sets.

Appendix B. Performance results of gradient boosting ensembles with and without hyperparameter tuning

In the following tables, detailed average accuracy (Table B.1) and Cohen's Kappa scores (Table B.2) are reported for gradient-based ensembles (GBM, XGBOOST, LGM, CatBoost) over a subset of 40 data-sets considered in our study. For each ensemble and data-set, two average scores are given: the one rendered by the ensemble with the default configuration of the software in use (column labeled as *Default*), and that provided by the ensemble with optimized hyper-parameter values as per an exhaustive search over the grid of possible values given in Table B.2 (column labeled as *Opt*). Further details on this analysis are given in Section 7.5.

Appendix C. Acronyms

ABC Adaptive Base Class

AdaB AdaBoost — Adaptive Boosting

AOSA Adaptive One-vs-All

AOSO Adaptive One-vs-One

API Application Programming Interface

Table A.1
Training time results measured in seconds for small and medium data-sets.

	AdaB	LB	RF	GBM	RotF	ET	RP	RRF	RRET	XGB GDBT	XGB DART	LGB GBDT	LGB GOSS	CatB
#1†*	1.402	3.9671	**0.0554**	3.2568	0.1959	0.1285	0.5847	0.3169	0.1813	0.6456	1.4958	0.7062	0.9455	36.0279
#2†*	3.6493	5.7959	1.4859	4.6725	2.7821	**0.1316**	2.2216	3.6865	1.4516	4.7396	14.442	0.5253	0.6284	51.0839
#3†	0.0378	0.0317	0.0125	0.0183	0.0458	0.0117	0.0109	0.0375	0.0276	**0.0074**	0.0079	0.0084	0.0076	0.0443
*#4**	0.1768	0.0281	0.0963	0.0391	0.4356	0.0417	0.1619	0.0562	0.0743	0.0413	0.0626	0.0252	**0.0228**	3.4788
#5†*	0.058	0.1876	0.0817	0.1158	0.2432	0.0339	0.0418	0.1255	0.096	0.1153	0.199	0.0852	**0.0237**	3.2699
#6†	0.0588	0.3527	0.0856	0.1235	0.0274	0.0427	**0.0162**	0.0742	0.0652	0.0263	0.1594	0.0436	0.0434	0.2109
#7	0.0496	0.3873	0.304	0.1125	0.0677	0.0836	0.1099	0.1034	0.0991	0.1661	0.6668	**0.0436**	0.0741	10.7203
#8	0.0126	0.0229	0.0258	0.0386	0.1505	0.0644	0.0331	0.0882	0.0708	0.0239	0.0456	**0.0116**	0.0119	0.0319
#9†*	0.0187	0.0201	0.0231	0.0113	0.045	0.0578	0.0693	0.0286	0.037	**0.0071**	0.0078	0.0121	0.0169	0.132
#10	0.1067	0.1067	0.0604	0.026	0.137	0.0907	0.0439	0.0871	0.062	0.025	0.0567	0.0321	**0.0108**	0.2564
#11†*	0.1852	0.6332	0.1104	0.6814	0.1462	0.0542	0.052	0.191	**0.0122**	0.2146	0.7406	0.0698	0.1346	5.6531
*#12**	0.0725	0.3547	0.078	0.6353	0.2264	**0.0245**	0.0713	1.4117	0.4147	0.4835	0.9893	0.1514	0.1428	2.4349
#13†	0.0868	0.3543	0.0245	0.1141	0.1282	0.0638	0.0436	0.0223	**0.0188**	0.0419	0.1123	0.0346	0.0578	1.621
#14†	0.1161	0.1308	0.571	0.3805	0.6906	0.1473	**0.066**	0.2184	0.1507	0.6373	0.955	0.1552	0.1528	0.3624
#15†*	8.0706	10.7316	**0.1219**	18.926	1.6766	0.1914	0.2802	16.0069	0.2794	0.7732	1.7346	1.9242	1.8237	110.5447
#16	0.0345	0.1109	0.0541	0.096	0.2665	0.1106	0.1335	0.0592	**0.0214**	0.0266	0.0743	0.0721	0.0547	0.7599
*#17**	0.0916	0.0248	0.0582	**0.0042**	0.049	0.013	0.1755	0.0987	0.0182	0.0096	0.0117	0.0133	0.0367	0.0287
#18†	0.0412	0.2228	0.0512	0.0418	0.1534	0.0417	0.0425	0.2028	0.1113	**0.0229**	0.042	0.0276	0.04	4.6896
#19†	0.1237	0.3618	0.0666	0.3379	0.1976	0.0633	0.1184	0.0977	0.0552	**0.0407**	0.0649	0.0501	0.0594	4.8893
#20†*	16.5132	24.1318	2.4217	7.8318	3.0169	**0.6696**	0.6835	27.8153	4.5955	18.6143	98.8483	4.4473	2.9725	290.1041
#21†*	0.0312	0.4519	**0.0244**	0.2507	0.1771	0.0677	0.1018	0.0983	0.0614	0.0763	0.2962	0.0516	0.0673	1.2876
#22†*	0.0978	0.1531	**0.0203**	0.1105	1.1953	0.038	0.1057	0.3494	0.1552	0.1666	0.4261	0.113	0.0564	5.9459
#23†	0.1266	0.73	0.1061	0.368	0.1965	**0.0385**	0.0724	0.0753	0.0689	0.1148	0.2204	0.0993	0.0653	25.2296
#24†	0.0855	0.0629	0.0607	0.0261	0.0925	0.0118	0.033	0.0156	0.014	**0.0061**	0.0211	0.0121	0.0146	0.0769
#25†	0.0454	0.327	0.0132	0.118	0.0277	0.0283	0.077	**0.008**	0.0658	0.0224	0.0457	0.0374	0.0099	0.0779
#26	**0.0097**	0.0442	0.0577	0.0234	0.2887	0.013	0.087	0.0489	0.056	0.0166	0.0295	0.0218	0.0231	0.1577
*#27**	0.0436	0.0201	0.0232	**0.0065**	0.0392	0.0119	0.0158	0.0465	0.0397	0.0071	0.0081	0.0087	0.0084	0.0144
#28	0.2623	0.1062	0.1242	0.0428	0.7687	0.0609	0.0997	0.1908	0.083	**0.0266**	0.0437	0.0358	0.057	0.3203
#29	0.0259	0.1068	0.014	0.0422	0.0385	0.0484	0.0495	0.0237	0.0325	0.0162	0.0229	0.0402	**0.0098**	0.0245
#30†*	6.4894	38.4735	0.1615	136.0944	1.3464	**0.1185**	0.2619	0.2732	0.2464	4.8154	73.4857	12.5356	9.8852	37.3753
#31	0.0678	0.7641	**0.0328**	0.2553	0.0675	0.0696	0.0858	0.0511	0.0724	0.0659	0.3369	0.078	0.0433	0.474
#32	8.8578	98.9515	1.0205	86.8789	17.5466	**0.6804**	2.3159	0.8894	1.0069	52.6411	270.3499	17.5038	12.5393	203.585
#33†*	0.0683	0.1463	0.0422	0.1226	0.0401	0.0299	0.0157	0.0777	0.0682	0.0586	0.1077	0.0532	**0.0145**	0.1163
#34	15.358	12.8454	2.2998	10.1114	9.8967	**0.3513**	5.6946	3.5399	0.3871	3.607	10.6595	0.7968	0.5746	42.9841
#35	0.0189	0.0654	0.0569	0.0191	0.1449	0.0316	0.0276	0.0405	**0.0121**	0.0153	0.0254	0.013	0.0141	0.2503
#36†	0.1856	4.1102	0.1609	3.8415	1.3193	0.1535	**0.0864**	0.1093	0.242	1.2422	3.5722	0.5971	0.3618	5.203
#37	0.0165	0.0504	0.0226	0.022	0.0159	0.022	0.0572	**0.0089**	0.0424	0.0128	0.0167	0.0128	0.0141	0.1132
#38	0.7088	4.7712	0.1654	1.906	2.4291	**0.0985**	0.333	0.642	0.812	0.9915	1.3062	2.0845	0.5212	1267.7446
*#39**	0.3728	0.0972	0.0506	0.066	0.3496	0.0779	0.0602	0.3048	0.0515	0.0799	0.1149	0.0488	**0.0463**	2.9907
#40†	0.0313	0.1968	0.0354	0.0438	0.1955	0.0276	0.0459	0.0493	**0.0095**	0.0253	0.0676	0.027	0.0222	2.5456
#41†*	1.4177	3.3196	0.1635	3.9759	0.2541	0.2922	**0.0541**	1.6642	0.3278	2.6466	9.5925	1.0277	0.685	116.7301
#42	4.4743	21.6732	0.5152	10.6806	18.2899	**0.3119**	2.1033	3.7262	0.9381	11.0156	28.4838	4.546	2.5624	51.7896
#43†	2.1046	2.5118	0.1672	0.3483	1.9858	0.1566	0.5292	0.9559	**0.1045**	0.6417	1.902	0.2086	0.1792	68.0168
#44	4.7006	24.1655	0.6637	13.1565	17.0956	**0.3996**	2.057	0.7732	0.4536	6.2837	30.648	5.1204	3.4352	131.036
#45†	1.633	0.94	0.362	1.6564	1.398	**0.0803**	0.4897	0.446	0.1715	0.3537	0.918	0.234	0.1867	13.5465
#46	0.3147	0.0637	0.0706	0.0259	0.1048	0.0592	0.1109	0.0351	0.0616	0.0311	0.135	0.0293	**0.0232**	2.9645
#47	1.2871	4.7968	1.285	1.7652	10.4723	**0.2467**	3.9269	2.1242	0.5065	2.003	4.0173	0.8134	0.5155	62.2473
*#48**	0.0317	0.11	0.026	0.036	0.1085	0.0472	0.0201	0.0579	0.0376	0.0251	0.0593	0.0239	**0.0153**	0.4631
#49†	4.4557	26.3015	0.7124	9.1704	17.0441	0.2615	2.3104	0.944	**0.2387**	6.9271	19.1515	2.4994	1.6473	100.7943
#50	1.3369	5.2682	0.1481	2.3025	1.8194	**0.0652**	0.5515	0.4313	0.1624	1.243	3.3619	2.1319	1.1472	167.6257
#51†	4.3883	16.5709	0.9896	12.2668	15.6946	1.0701	1.8871	1.825	1.2696	7.3574	32.1297	**0.3146**	0.4419	89.1258
#52	0.0687	0.2742	0.0574	**0.011**	0.1001	0.0481	0.0636	0.0482	0.3223	0.0671	0.0748	0.0675	0.0389	0.0568
#53	2.5204	3.4321	**0.1559**	0.6779	13.5925	0.1955	1.6129	0.7464	0.6659	0.8117	1.5966	0.5916	0.3639	118.5586
#54†	0.1343	0.0759	0.1017	0.0641	0.9512	0.0523	0.1016	0.1985	0.1803	**0.0219**	0.0262	0.0335	0.0305	0.3138
#55†*	2.1968	3.4209	0.2716	1.855	8.6694	0.2138	0.5337	0.9541	**0.1449**	1.4791	3.3494	0.3198	0.212	164.2367
#56	0.0745	0.0831	0.0493	0.1897	0.0569	0.0296	0.0633	0.0584	0.0166	0.0553	0.1172	0.042	**0.0104**	0.31
#57	10.2725	31.5176	0.9042	14.4321	13.982	**0.2009**	1.3434	1.8703	0.4547	8.9158	20.7705	13.8478	7.4452	673.921
#58†	**0.0096**	0.5236	0.1862	0.7343	1.8152	0.2344	0.2384	0.8773	0.191	0.6911	1.7812	0.3165	0.1659	14.0925
*#59**	0.1142	0.1651	0.0926	0.1641	0.1989	0.0469	0.076	0.1271	0.04	0.0773	0.2394	0.0593	**0.0288**	0.7676
#60†	0.0559	0.0227	0.064	0.0256	0.0248	0.0143	0.0999	0.0102	**0.0093**	0.0134	0.0158	0.0278	0.0387	0.0381
#61	8.443	4.9144	1.2147	2.1345	5.3163	**0.2381**	2.5611	0.8859	0.5204	2.2522	4.911	0.411	0.5992	3.7371
#62	0.4041	1.2825	0.0522	0.2423	0.415	**0.0391**	0.143	0.0889	0.086	0.3149	0.9081	0.1532	0.1176	1.1676
#63	0.5487	3.4553	0.1719	1.5838	0.49	0.1069	0.2611	0.1868	**0.0859**	0.906	2.5677	0.9352	0.269	171.0835
#64	0.2615	0.3269	0.0876	0.1045	0.705	0.0455	0.0559	0.1207	0.1069	**0.0368**	0.0454	0.0579	0.0733	0.801
#65	0.0625	0.1508	0.0702	0.0699	0.0646	0.0347	0.0653	0.0639	0.031	**0.0235**	0.0356	0.0265	0.0328	8.1513
#66†	0.153	0.714	0.0669	0.1697	0.339	**0.0649**	0.0947	0.1608	0.0962	0.0844	0.1647	0.071	0.0687	0.9503
#67†	1.6181	1.7964	0.4104	1.4235	2.478	**0.0985**	0.7299	1.274	0.2867	0.9134	4.5964	0.3635	0.2068	4.7414
#68	0.0216	0.0792	0.054	0.0608	0.1139	0.0192	0.0924	0.0098	**0.009**	0.0186	0.0254	0.0202	0.0209	0.0697
#69†	0.4726	1.4208	0.1618	0.6244	0.1211	**0.0474**	0.1116	0.0681	0.1082	0.3235	1.9355	0.1559	0.1263	15.1883
#70†*	**0.0051**	0.1549	0.0227	0.0462	0.038	0.012	0.0164	0.0497	0.0375	0.0271	0.0426	0.0795	0.0152	0.1497
Avg:	1.677	5.2852	0.2797	5.1116	2.5799	**0.1254**	0.527	1.1193	0.2719	2.089	9.3639	1.102	0.7473	58.6505

Table A.2
Test runtime results measured in seconds for small and medium data-sets.

	AdaB	LB	RF	GBM	RotF	ET	RP	RRF	RRET	XGB GDBT	XGB DART	LGB GBDT	LGB GOSS	CatB
#1†*	0.0642	0.0701	0.0048	0.0088	**0.0027**	0.02	0.0195	0.053	0.0769	0.0217	0.1338	0.0417	0.0569	0.0067
#2†*	0.0573	0.0473	0.0667	0.0188	0.0156	**0.0085**	0.0551	0.1024	0.2159	0.0616	0.0733	0.0564	0.0641	0.0342
#3†	0.0037	0.0018	0.0017	**0.0002**	0.002	0.0018	0.0014	0.0086	0.0068	0.001	0.0006	0.0006	0.0007	0.0045
*#4**	0.0087	0.0012	0.0078	**0.0003**	0.0081	0.0048	0.0098	0.0103	0.0171	0.0006	0.0006	0.001	0.0009	0.0036
#5†*	0.0042	0.0077	0.0078	**0.0004**	0.0042	0.0039	0.0032	0.0223	0.0196	0.0007	0.0014	0.0012	0.0007	0.0051
#6†	0.005	0.0186	0.0081	**0.0004**	0.0016	0.0049	0.0019	0.0155	0.0144	0.0007	0.0009	0.0015	0.0013	0.0026
#7	0.0027	0.0078	0.016	**0.0013**	0.0019	0.0107	0.0072	0.0119	0.0273	0.0048	0.0051	0.0034	0.0053	0.0063
#8	0.0012	0.0013	0.0027	**0.0002**	0.0034	0.0072	0.0025	0.0161	0.0157	0.0006	0.0006	0.0007	0.0007	0.0026
#9†*	0.002	0.0012	0.0027	**0.0003**	0.0021	0.0065	0.0066	0.0057	0.0087	0.0005	0.0006	0.0007	0.0007	0.0033
#10	0.0077	0.0051	0.0057	**0.0003**	0.0056	0.0103	0.0039	0.0161	0.0139	0.0007	0.0006	0.0011	0.0005	0.0047
#11†*	0.0122	0.0273	0.0103	**0.0028**	0.0055	0.0061	0.0055	0.0218	0.0031	0.0043	0.0029	0.0032	0.0092	0.0046
*#12**	0.0023	0.0104	0.0071	**0.0016**	0.0029	0.0022	0.0058	0.059	0.0593	0.0027	0.0032	0.0059	0.0066	0.0095
#13†	0.0087	0.0212	0.0027	**0.0004**	0.0035	0.0073	0.0037	0.0045	0.0047	0.0008	0.0011	0.0008	0.0013	0.0033
#14†	0.0042	0.0038	0.0226	**0.0016**	0.0062	0.0077	0.0057	0.0309	0.0316	0.0032	0.0106	0.0052	0.0054	0.0271
#15†*	0.1867	0.2798	**0.0068**	0.0464	0.0188	0.0089	0.02	0.3779	0.0644	0.013	0.035	0.1753	0.1735	0.1386
#16	0.0037	0.0046	0.0048	**0.0008**	0.0073	0.0113	0.0116	0.0115	0.0057	0.0011	0.0022	0.0031	0.0025	0.0039
*#17**	0.004	0.0013	0.0061	**0.0002**	0.0017	0.0019	0.0114	0.0175	0.0048	0.0006	0.0006	0.0008	0.0011	0.0041
#18†	0.0035	0.0119	0.0054	**0.0004**	0.0024	0.0048	0.0039	0.0304	0.0201	**0.0004**	0.0008	0.0008	0.001	0.0049
#19†	0.0097	0.0194	0.0066	**0.0008**	0.0068	0.0079	0.0108	0.0183	0.0127	**0.0008**	0.0011	0.0013	0.0014	0.0042
#20†*	0.416	0.5359	0.1486	0.0367	0.0325	0.0408	**0.0295**	0.756	0.6654	0.281	0.3803	0.2693	0.2312	0.1595
#21†*	0.0023	0.0242	0.0029	**0.0008**	0.0063	0.0084	0.0097	0.0174	0.0139	0.0016	0.0036	0.0022	0.003	0.005
#22†*	0.0043	0.0056	0.0019	**0.0006**	0.0138	0.0038	0.0057	0.0336	0.0288	0.0017	0.0018	0.0033	0.0021	0.0061
#23†	0.0096	0.0334	0.0098	**0.0008**	0.0065	0.0046	0.006	0.0143	0.0154	0.0013	0.0019	0.0016	0.0012	0.0044
#24†	0.0079	0.0034	0.0063	**0.0003**	0.0055	0.0017	0.0035	0.0039	0.0037	**0.0003**	0.0005	0.0006	0.0007	0.0042
#25†	0.0048	0.018	0.0017	**0.0003**	0.0017	0.0036	0.0079	0.0022	0.0151	0.0006	0.0007	0.0008	0.0006	0.0041
#26	0.0011	0.0024	0.0061	**0.0003**	0.0078	0.0019	0.0079	0.0094	0.0125	0.0006	0.0005	0.0007	0.0007	0.005
*#27**	0.0043	0.0012	0.0027	**0.0003**	0.0014	0.0018	0.0019	0.009	0.0087	0.0005	0.0006	0.0007	0.0006	0.0058
#28	0.0098	0.0042	0.0089	**0.0003**	0.0088	0.0066	0.0037	0.0235	0.0146	0.0005	0.0005	0.0007	0.001	0.0044
#29	0.0026	0.0055	0.0018	**0.0003**	0.0022	0.0057	0.0047	0.0054	0.0077	0.0005	0.0006	0.0008	0.0006	0.0043
#30†*	0.3843	0.708	0.0262	0.3594	0.106	0.0261	0.0537	0.0292	0.1285	0.3165	0.5449	1.8602	1.8895	**0.0194**
#31	0.0085	0.0434	0.0039	**0.0006**	0.0032	0.0084	0.0099	0.0112	0.0175	0.0021	0.0024	0.0022	0.0014	0.0049
#32	0.3883	0.9859	0.0734	0.2506	0.1368	0.0817	0.0965	0.1504	0.33	1.1635	1.5269	1.8723	1.4892	**0.0206**
#33†*	0.0064	0.0076	0.0044	**0.0003**	0.0013	0.0036	0.0019	0.0149	0.0142	0.0007	0.0006	0.0011	0.0006	0.0043
#34	0.0831	0.0341	0.0311	0.0175	0.0354	0.0291	0.0469	0.1042	0.0602	0.037	0.0403	0.0511	0.036	**0.0129**
#35	0.0022	0.0036	0.006	**0.0003**	0.0067	0.0039	0.0028	0.0094	0.0035	0.0006	0.0007	0.0008	0.0008	0.0041
#36†	0.0235	0.1133	0.0164	0.0167	0.0255	0.0158	**0.0069**	0.0201	0.0697	0.0311	0.0886	0.0458	0.0322	0.0084
#37	0.002	0.0028	0.0025	**0.0003**	0.0011	0.0027	0.0056	0.0023	0.01	0.0005	0.0005	0.0007	0.0007	0.0042
#38	0.006	0.0655	0.008	**0.0012**	0.0113	0.009	0.0052	0.0359	0.047	0.002	0.0109	0.0062	0.0032	0.0061
*#39**	0.0141	0.0027	0.0042	**0.0004**	0.006	0.007	0.0039	0.0162	0.01	0.0007	0.0011	0.0024	0.002	0.0112
#40†	0.0034	0.0109	0.004	**0.0003**	0.0092	0.0033	0.0043	0.0104	0.0029	0.0007	0.0009	0.0008	0.0007	0.0043
#41†*	0.0532	0.073	0.0154	0.0193	0.0056	0.0307	**0.0037**	0.0896	0.0748	0.0421	0.0609	0.1012	0.1027	0.0137
#42	0.0659	0.1918	0.0214	0.0228	0.0509	0.0202	0.0378	0.1034	0.1038	0.0744	0.1312	0.1467	0.1216	**0.0178**
#43†	0.0384	0.0286	0.005	**0.0016**	0.0147	0.0164	0.0095	0.0419	0.023	0.009	0.0081	0.0071	0.008	0.0076
#44	0.11	0.1914	0.0235	0.0417	0.0604	0.0363	0.0403	0.1138	0.0889	0.1342	0.2348	0.296	0.2474	**0.0132**
#45†	0.0312	0.0088	0.0124	0.0054	0.0135	0.0095	0.0134	0.0321	0.0398	**0.0052**	0.0098	0.0114	0.0094	0.0063
#46	0.0141	0.0029	0.0056	**0.0003**	0.0031	0.0065	0.0089	0.0066	0.0142	0.0006	0.0006	0.0012	0.0009	0.0047
#47	0.0264	0.0145	0.0188	**0.0033**	0.0272	0.0205	0.0245	0.0761	0.0835	0.0113	0.0141	0.0119	0.0122	0.0099
*#48**	0.003	0.0051	0.0027	**0.0003**	0.0036	0.0056	0.0021	0.0111	0.0091	0.0006	0.0006	0.0009	0.0007	0.0028
#49†	0.0469	0.1185	0.0203	0.0148	0.0445	0.0191	0.0292	0.0805	0.0422	0.0611	0.0899	0.0796	0.0749	**0.0121**
#50	0.0245	0.0617	0.0063	**0.0057**	0.0124	0.0074	0.0112	0.0351	0.0353	0.0112	0.019	0.029	0.0241	0.0066
#51†	0.2837	0.1832	0.0456	0.0472	0.1155	0.1075	0.0678	0.3934	0.258	0.1202	0.113	0.0244	**0.0243**	0.0307
#52	0.004	0.0067	0.0055	**0.0003**	0.0013	0.0054	0.0047	0.0109	0.0346	0.0005	0.0006	0.0007	0.0008	0.003
#53	0.0244	0.0187	0.0062	**0.0013**	0.0324	0.0144	0.0241	0.0306	0.0799	0.0034	0.0062	0.007	0.0059	0.0138
#54†	0.0052	0.0023	0.0082	**0.0003**	0.0093	0.0053	0.0061	0.0243	0.0284	0.0005	0.0005	0.0008	0.0009	0.0052
#55†*	0.0278	0.0385	0.014	**0.004**	0.0272	0.0129	0.0125	0.0363	0.017	0.0086	0.0106	0.01	0.0093	0.0145
#56	0.0074	0.0041	0.0048	**0.0005**	0.0028	0.0036	0.0057	0.0131	0.0043	0.0009	0.0008	0.0009	0.0006	0.0043
#57	0.0634	0.1417	0.0159	0.0167	0.0295	0.017	**0.0085**	0.0693	0.0794	0.0577	0.0706	0.1408	0.1156	0.0121
#58†	**0.0009**	0.0113	0.0132	0.0023	0.0166	0.0217	0.0128	0.0347	0.0362	0.0064	0.0081	0.0227	0.007	0.0092
*#59**	0.0078	0.0072	0.0081	**0.0007**	0.0059	0.0045	0.0066	0.0146	0.0086	0.0014	0.0011	0.0022	0.0014	0.0041
#60†	0.0024	0.0013	0.0068	**0.0004**	0.0022	0.0021	0.0097	0.0028	0.0028	0.0007	0.0007	0.0016	0.0017	0.0037
#61	0.0343	0.0158	0.0188	**0.0039**	0.0174	0.0201	0.0158	0.0482	0.0872	0.0122	0.0149	0.0087	0.0108	0.0096
#62	0.0119	0.029	0.0034	**0.0008**	0.0043	0.0042	0.0069	0.0186	0.0188	0.0035	0.0022	0.0038	0.0036	0.0034
#63	0.014	0.0701	0.0099	**0.0031**	0.0056	0.0107	0.0088	0.0211	0.0186	0.0115	0.0144	0.0114	0.0052	0.0053
#64	0.007	0.007	0.0056	**0.0003**	0.007	0.0047	0.0023	0.0173	0.0215	0.0005	0.0007	0.0009	0.0011	0.0043
#65	0.0051	0.0075	0.0068	**0.0003**	0.0019	0.0041	0.0054	0.014	0.0076	0.0005	0.0005	0.0007	0.0008	0.0024
#66†	0.0044	0.0177	0.0047	**0.0009**	0.0047	0.007	0.0046	0.0199	0.0214	0.0015	0.0068	0.0024	0.0024	0.0031
#67†	0.0374	0.0493	0.0176	**0.0037**	0.0299	0.0099	0.018	0.0785	0.0612	0.0216	0.0198	0.0257	0.015	0.0063
#68	0.0021	0.004	0.0051	**0.0004**	0.0045	0.0026	0.008	0.0026	0.0026	0.0006	0.0006	0.0009	0.0009	0.0025
#69†	0.022	0.0683	0.0133	**0.0015**	0.0025	0.0067	0.0081	0.0105	0.0259	0.0104	0.0078	0.0078	0.0068	0.0039
#70†*	0.0007	0.0086	0.0027	**0.0003**	0.0013	0.0018	0.0018	0.0105	0.0085	0.0006	0.0007	0.0011	0.0006	0.0027
Avg:	0.0394	0.0649	0.0131	0.014	0.0156	0.0121	0.0132	0.0511	0.0491	0.0368	0.0533	0.077	0.0693	**0.0116**

Table B.1
Average accuracy scores of gradient-boosting ensembles with and without parameter tuning for a subset of the considered small- and medium-size data-sets.

	GBM		XGB GDBT		XGB DART		LGB GDBT		LGB GOSS		CatB	
	Default	Opt.	Default	Opt.	Default	Opt.	Default	Opt.	Default	Opt.	Default	Opt.
#3†	0.84	0.859	0.849	0.849	0.849	0.868	0.84	0.84	0.802	0.802	0.849	0.868
*#4**	0.851	0.864	0.841	0.857	0.841	0.87	0.835	0.87	0.852	0.852	0.858	0.861
#6†	0.738	0.861	0.77	0.858	0.77	0.832	0.741	0.762	0.816	0.819	0.794	0.83
#7	0.896	0.899	0.89	0.899	0.89	0.898	0.898	0.898	0.899	0.899	0.898	0.902
#8	0.584	0.625	0.57	0.592	0.57	0.597	0.584	0.6	0.581	0.589	0.578	0.589
*#9**†	0.697	0.729	0.676	0.726	0.676	0.729	0.733	0.751	0.758	0.758	0.74	0.762
#10	0.701	0.701	0.704	0.71	0.704	0.707	0.707	0.707	0.696	0.713	0.725	0.725
*#11**†	0.784	0.804	0.79	0.79	0.79	0.79	0.788	0.788	0.781	0.781	0.742	0.756
#13†	0.559	0.602	0.579	0.579	0.579	0.593	0.586	0.592	0.586	0.586	0.579	0.589
#16	0.561	0.561	0.518	0.562	0.518	0.553	0.534	0.557	0.528	0.553	0.523	0.557
*#17**	0.847	0.847	0.856	0.856	0.856	0.856	0.842	0.842	0.858	0.858	0.847	0.847
#18†	0.958	0.975	0.969	0.972	0.969	0.969	0.964	0.964	0.947	0.958	0.978	0.978
*#21**†	0.736	0.744	0.724	0.751	0.724	0.748	0.721	0.748	0.734	0.747	0.717	0.742
*#22**†	0.772	0.772	0.769	0.774	0.769	0.772	0.758	0.77	0.753	0.763	0.766	0.775
#23†	0.724	0.78	0.78	0.78	0.78	0.78	0.771	0.771	0.687	0.711	0.794	0.794
#24†	0.693	0.739	0.693	0.726	0.693	0.745	0.706	0.732	0.726	0.726	0.667	0.742
#25†	0.831	0.838	0.8	0.838	0.8	0.831	0.669	0.669	0.556	0.575	0.806	0.819
#26	0.789	0.856	0.789	0.841	0.789	0.848	0.796	0.804	0.833	0.841	0.815	0.848
*#27**	0.944	0.961	0.944	0.957	0.944	0.957	0.961	0.97	0.97	0.97	0.957	0.957
#28	0.9	0.912	0.915	0.915	0.915	0.915	0.903	0.903	0.898	0.903	0.909	0.918
#29	0.967	0.967	0.96	0.96	0.96	0.96	0.967	0.967	0.933	0.94	0.967	0.967
#31	0.712	0.748	0.714	0.74	0.714	0.74	0.724	0.762	0.736	0.752	0.708	0.744
#35	0.819	0.833	0.802	0.831	0.802	0.831	0.806	0.827	0.829	0.831	0.81	0.839
#36†	0.335	0.343	0.318	0.341	0.318	0.341	0.335	0.345	0.323	0.341	0.321	0.349
#37	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
#40†	0.949	0.949	0.953	0.953	0.953	0.953	0.94	0.958	0.944	0.953	0.963	0.963
#45†	0.86	0.907	0.893	0.901	0.893	0.901	0.898	0.898	0.893	0.893	0.902	0.904
#46	0.746	0.764	0.729	0.754	0.729	0.743	0.742	0.749	0.749	0.749	0.742	0.756
*#48**	0.716	0.749	0.677	0.717	0.677	0.727	0.693	0.712	0.695	0.721	0.684	0.719
#52	0.664	0.717	0.693	0.735	0.693	0.693	0.731	0.731	0.741	0.741	0.707	0.707
#54†	0.801	0.801	0.816	0.816	0.816	0.816	0.805	0.805	0.79	0.816	0.794	0.816
#56	0.707	0.707	0.687	0.687	0.687	0.687	0.555	0.555	0.509	0.509	0.693	0.693
*#59**	0.879	0.964	0.962	0.962	0.962	0.962	0.937	0.941	0.903	0.917	0.94	0.94
#60†	0.791	0.791	0.788	0.788	0.788	0.791	0.784	0.784	0.788	0.788	0.788	0.788
#62	0.771	0.772	0.758	0.782	0.758	0.773	0.772	0.772	0.78	0.78	0.78	0.78
#63	0.574	0.61	0.559	0.579	0.559	0.585	0.581	0.581	0.602	0.602	0.653	0.653
#64	0.963	0.963	0.972	0.972	0.972	0.974	0.968	0.968	0.972	0.974	0.972	0.972
#65	0.899	0.977	0.933	0.972	0.933	0.955	0.966	0.966	0.972	0.972	0.955	0.977
#66†	0.575	0.586	0.547	0.577	0.547	0.575	0.554	0.582	0.551	0.568	0.546	0.595
#68	0.962	0.962	0.959	0.965	0.959	0.965	0.959	0.962	0.958	0.959	0.963	0.965

CART Classification And Regression Tree

CatB CatBoost — Unbiased Gradient Boosting with categorical features

CRAN Comprehensive R Archive Network

CSC Compressed Sparse Column

CSR Compressed Sparse Row

CUDA Compute Unified Device Architecture

DART Dropouts meet Multiple Additive Regression Trees

DC Distributed computing

DECOC Data-driven Error-Correcting Output Codes

ECOC Error-Correcting Output Codes

EFB Exclusive Feature Bundling

ET Extremely randomized trees or Extra-Trees

GBDT Gradient Boosting Decision Trees

GBM Gradient Boosting Machine

GC Parallel GPU computing

GOSS Gradient-based One-Side Sampling

ICA Independent Component Analysis

JVM Java Virtual Machine

LB LogitBoost

LGB LightGBM — Light Gradient Boosting Machine

ML Machine Learning

MLlib Spark Machine Learning library

NDA Nonparametric Discriminate Analysis

PC Parallel CPU computing

PCA Pricipal Component Analysis

PV-tree Parallel Voting Decision Trees

RF Random Forest

RotF Rotation Forest

RP Random Patches

Table B.2
Average Cohen's kappa scores of gradient-boosting ensembles with and without parameter tuning for a subset of the considered small- and medium-size data-sets.

	GBM		XGB GDBT		XGB DART		LGB GDBT		LGB GOSS		CatB	
	Default	Opt.	Default	Opt.	Default	Opt.	Default	Opt.	Default	Opt.	Default	Opt.
#3†	0.472	0.5	0.494	0.494	0.494	0.536	0.475	0.475	0.076	0.076	0.489	0.49
*#4**	0.698	0.725	0.677	0.708	0.677	0.735	0.665	0.734	0.7	0.7	0.713	0.718
#6†	0.545	0.742	0.606	0.741	0.606	0.693	0.558	0.565	0.667	0.672	0.651	0.692
#7	0.788	0.795	0.777	0.795	0.777	0.794	0.793	0.793	0.795	0.795	0.793	0.801
#8	0.092	0.132	0.08	0.08	0.08	0.08	0.123	0.123	0.103	0.103	0.072	0.072
*#9**†	0.191	0.257	0.188	0.23	0.188	0.254	0.265	0.288	0.324	0.324	0.295	0.295
#10	0.382	0.382	0.393	0.393	0.393	0.393	0.4	0.4	0.367	0.403	0.433	0.433
*#11**†	0.547	0.598	0.567	0.567	0.567	0.567	0.564	0.577	0.552	0.552	0.461	0.504
#13†	0.297	0.302	0.332	0.332	0.332	0.332	0.322	0.322	0.327	0.327	0.303	0.303
#16	0.318	0.318	0.251	0.315	0.251	0.299	0.277	0.305	0.263	0.296	0.259	0.303
*#17**	0.697	0.697	0.713	0.713	0.713	0.713	0.685	0.685	0.717	0.717	0.696	0.696
#18†	0.948	0.968	0.961	0.965	0.961	0.961	0.954	0.954	0.933	0.947	0.972	0.972
*#21**†	0.662	0.669	0.647	0.68	0.647	0.674	0.643	0.674	0.658	0.673	0.637	0.667
*#22**†	0.421	0.421	0.42	0.42	0.42	0.42	0.377	0.401	0.37	0.39	0.383	0.384
#23†	0.62	0.692	0.698	0.698	0.698	0.698	0.682	0.682	0.552	0.586	0.714	0.714
#24†	0.144	0.188	0.164	0.164	0.164	0.228	0.171	0.198	0.189	0.189	0.131	0.149
#25†	0.735	0.745	0.683	0.745	0.683	0.736	0.48	0.48	0.259	0.289	0.696	0.715
#26	0.571	0.706	0.574	0.676	0.574	0.69	0.588	0.6	0.661	0.676	0.623	0.69
*#27**	0.888	0.922	0.888	0.914	0.888	0.914	0.922	0.939	0.939	0.939	0.913	0.913
#28	0.775	0.802	0.809	0.809	0.809	0.809	0.783	0.783	0.771	0.783	0.794	0.815
#29	0.95	0.95	0.94	0.94	0.94	0.94	0.95	0.95	0.9	0.91	0.95	0.95
#31	0.68	0.72	0.682	0.71	0.682	0.711	0.693	0.735	0.706	0.724	0.675	0.715
#35	0.638	0.664	0.603	0.662	0.603	0.662	0.611	0.652	0.657	0.662	0.618	0.676
#36†	0.231	0.236	0.217	0.232	0.217	0.233	0.234	0.238	0.224	0.232	0.22	0.24
#37	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
#40†	0.881	0.881	0.893	0.893	0.893	0.893	0.855	0.9	0.869	0.89	0.914	0.914
#45†	0.656	0.773	0.739	0.759	0.739	0.759	0.754	0.754	0.742	0.742	0.761	0.767
#46	0.427	0.452	0.402	0.428	0.402	0.402	0.419	0.422	0.44	0.44	0.423	0.437
*#48**	0.355	0.408	0.27	0.319	0.27	0.353	0.294	0.31	0.294	0.316	0.272	0.316
#52	0.322	0.43	0.384	0.463	0.384	0.384	0.462	0.462	0.481	0.481	0.41	0.41
#54†	0.323	0.323	0.367	0.367	0.367	0.367	0.378	0.378	0.279	0.318	0.259	0.267
#56	0.56	0.56	0.53	0.53	0.53	0.53	0.333	0.333	0.264	0.264	0.54	0.54
*#59**	0.735	0.922	0.919	0.919	0.919	0.919	0.862	0.869	0.791	0.819	0.87	0.87
#60†	0.433	0.436	0.428	0.432	0.428	0.433	0.425	0.429	0.428	0.428	0.428	0.428
#62	0.694	0.696	0.677	0.71	0.677	0.697	0.696	0.696	0.707	0.707	0.707	0.707
#63	0.531	0.571	0.514	0.537	0.514	0.543	0.539	0.539	0.562	0.562	0.618	0.618
#64	0.921	0.921	0.94	0.94	0.94	0.943	0.933	0.933	0.94	0.943	0.94	0.94
#65	0.849	0.966	0.899	0.958	0.899	0.932	0.949	0.949	0.958	0.958	0.933	0.966
#66†	0.319	0.319	0.272	0.292	0.272	0.286	0.279	0.301	0.275	0.285	0.267	0.324
#68	0.917	0.917	0.91	0.923	0.91	0.923	0.91	0.916	0.906	0.909	0.92	0.923

RR Random Rotation

RRET Random Rotation Extra-Trees

RRF Random Rotation Forest

SAMME Stagewise Additive Modeling

SC Sequential CPU computing

SGB Stochastic Gradient Boosting

SHAP SHapley Additive exPlanation

Smile Statistical Machine Intelligence and Learning Engine

SRP Sparse Random Projections

XAI Explainable Artificial Intelligence

XGB XGBoost — eXtreme Gradient Boosting

References

[1] Z.-H. Zhou, Ensemble Methods: Foundations and Algorithms, Chapman and Hall/CRC, 2012.

[2] C. Zhang, Y. Ma, Ensemble Machine Learning: Methods and Applications, Springer, 2012.

[3] L. Rokach, Decision forest: Twenty years of research, Inf. Fusion 27 (2016) 111–125.

[4] L. Rokach, Ensemble Learning: Pattern Classification Using Ensemble Methods, World Scientific, 2019.

[5] X. Wu, V. Kumar, J.R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G.J. McLachlan, A. Ng, B. Liu, S.Y. Philip, et al., Top 10 algorithms in data mining, Knowl. Inf. Syst. 14 (1) (2008) 1–37.

[6] M. Fernández-Delgado, E. Cernadas, S. Barro, D. Amorim, Do we need hundreds of classifiers to solve real world classification problems?, J. Mach. Learn. Res. 15 (1) (2014) 3133–3181.

[7] D.T. Ahneman, J.G. Estrada, S. Lin, S.D. Dreher, A.G. Doyle, Predicting reaction performance in C–N cross-coupling using machine learning, Science 360 (6385) (2018) 186–190.

[8] K. Lee, H.-o. Jeong, S. Lee, W.-K. Jeong, CPEM: Accurate cancer type classification based on somatic alterations using an ensemble of a random forest and a deep neural network, Sci. Rep. 9 (1) (2019) 1–9.

[9] I. Triguero, S. del Río, V. López, J. Bacardit, J.M. Benítez, F. Herrera, ROSEFW-RF: the winner algorithm for the ECBDL'14 big data competition: an extremely imbalanced big data bioinformatics problem, Knowl.-Based Syst. 87 (2015) 69–79.

[10] S. García, J. Luengo, F. Herrera, Data Preprocessing in Data Mining, Springer, 2015.

[11] S. Gónzalez, S. García, M. Lázaro, A.R. Figueiras-Vidal, F. Herrera, Class switching according to nearest enemy distance for learning from highly imbalanced data-sets, Pattern Recognit. 70 (2017) 12–24.

[12] D. García-Gil, F. Luque-Sánchez, J. Luengo, S. García, F. Herrera, From big to smart data: Iterative ensemble filter for noise filtering in big data classification, Int. J. Intell. Syst. (2019).

[13] B. Krawczyk, L.L. Minku, J. Gama, J. Stefanowski, M. Woźniak, Ensemble learning for data stream analysis: A survey, Inf. Fusion 37 (2017) 132–156.

[14] K.G. Mehrotra, C.K. Mohan, H. Huang, Anomaly Detection Principles and Algorithms, Springer, 2017.
[15] C.C. Aggarwal, S. Sathe, Outlier Ensembles: An Introduction, Springer, 2017.
[16] Y. Ren, L. Zhang, P.N. Suganthan, Ensemble classification and regression-recent developments, applications and future directions, IEEE Comput. Intell. Mag. 11 (1) (2016) 41–53.
[17] L. Breiman, Bagging predictors, Mach. Learn. 24 (2) (1996) 123–140.
[18] R.E. Schapire, Y. Freund, Boosting: Foundations and algorithms, Kybernetes (2013).
[19] L. Rokach, Ensemble-based classifiers, Artif. Intell. Rev. 33 (1–2) (2010) 1–39.
[20] M. Wainberg, B. Alipanahi, B.J. Frey, Are random forests truly the best classifiers?, J. Mach. Learn. Res. 17 (1) (2016) 3837–3841.
[21] A.J. Wyner, M. Olson, J. Bleich, D. Mease, Explaining the success of adaboost and random forests as interpolating classifiers, J. Mach. Learn. Res. 18 (1) (2017) 1558–1590.
[22] O. Sagi, L. Rokach, Ensemble learning: A survey, Wiley Interdiscipl. Rev.: Data Min. Knowl. Discov. 8 (4) (2018) e1249.
[23] Y. Freund, R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, J. Comput. Syst. Sci. 55 (1) (1997) 119–139.
[24] T. Hastie, S. Rosset, J. Zhu, H. Zou, Multi-class adaboost, Stat. Interface 2 (3) (2009) 349–360.
[25] J. Friedman, T. Hastie, R. Tibshirani, et al., Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors), Ann. Statist. 28 (2) (2000) 337–407.
[26] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32.
[27] P. Geurts, D. Ernst, L. Wehenkel, Extremely randomized trees, Mach. Learn. 63 (1) (2006) 3–42.
[28] G. Louppe, P. Geurts, Ensembles on random patches, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2012, pp. 346–361.
[29] J.J. Rodriguez, L.I. Kuncheva, C.J. Alonso, Rotation forest: A new classifier ensemble method, IEEE Trans. Pattern Anal. Mach. Intell. 28 (10) (2006) 1619–1630.
[30] R. Blaser, P. Fryzlewicz, Random rotation ensembles, J. Mach. Learn. Res. 17 (1) (2016) 126–151.
[31] J.H. Friedman, Greedy function approximation: a gradient boosting machine, Ann. Statist. (2001) 1189–1232.
[32] T. Chen, C. Guestrin, Xgboost: A scalable tree boosting system, in: Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 785–794.
[33] K.V. Rashmi, R. Gilad-Bachrach, DART: Dropouts meet multiple additive regression trees, in: AISTATS, 2015, pp. 489–497.
[34] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, LightGBM: A highly efficient gradient boosting decision tree, in: Advances in Neural Information Processing Systems, 2017, pp. 3146–3154.
[35] A.V. Dorogush, V. Ershov, A. Gulin, CatBoost: gradient boosting with categorical features support, 2018, arXiv preprint arXiv:1810.11363.
[36] L. Prokhorenkova, G. Gusev, A. Vorobev, A.V. Dorogush, A. Gulin, CatBoost: unbiased boosting with categorical features, in: Advances in Neural Information Processing Systems, 2018, pp. 6638–6648.
[37] L. Rokach, O. Maimon, Top-down induction of decision trees classifiers-a survey, IEEE Trans. Syst. Man Cybern. C 35 (4) (2005) 476–487.
[38] L. Rokach, O.Z. Maimon, Data Mining with Decision Trees: Theory and Applications, vol. 69, World scientific, 2008.
[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[40] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, G. Varoquaux, API design for machine learning software: experiences from the scikit-learn project, in: ECML PKDD Workshop: Languages for Data Mining and Machine Learning, 2013, pp. 108–122.
[41] A. Bagnall, G.C. Cawley, On the use of default parameter settings in the empirical evaluation of classification algorithms, 2017, arXiv:1703.06777.
[42] G. Louppe, Understanding random forests: From theory to practice, 2014, arXiv preprint arXiv:1407.7502.
[43] L. Breiman, Randomizing outputs to increase prediction accuracy, Mach. Learn. 40 (3) (2000) 229–242.
[44] G. Martínez-Muñoz, A. Suárez, Switching class labels to generate classification ensembles, Pattern Recognit. 38 (10) (2005) 1483–1494.
[45] R.E. Schapire, Y. Singer, Improved boosting algorithms using confidence-rated predictions, Mach. Learn. 37 (3) (1999) 297–336.
[46] G.I. Webb, Multiboosting: A technique for combining boosting and wagging, Mach. Learn. 40 (2) (2000) 159–196.
[47] I. Mukherjee, R.E. Schapire, A theory of multiclass boosting, J. Mach. Learn. Res. 14 (Feb) (2013) 437–497.
[48] P. Li, Robust logitboost and adaptive base class (abc) logitboost, 2012, arXiv preprint arXiv:1203.3491.
[49] P. Sun, M.D. Reid, J. Zhou, Aoso-logitboost: Adaptive one-vs-one logitboost for multi-class problem, 2011, arXiv preprint arXiv:1110.3907.
[50] K. Wu, AOSA-Logistboost: Adaptive one-vs-all logistboost for multi-class classification problems, in: 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), IEEE, 2015, pp. 654–662.
[51] J.H. Friedman, Stochastic gradient boosting, Comput. Stat. Data Anal. 38 (4) (2002) 367–378.
[52] L.I. Kuncheva, J.J. Rodríguez, An experimental study on rotation forest ensembles, in: International Workshop on Multiple Classifier Systems, Springer, 2007, pp. 459–468.
[53] K.W. De Bock, D. Van den Poel, An empirical evaluation of rotation-based ensemble classifiers for customer churn prediction, Expert Syst. Appl. 38 (10) (2011) 12293–12301.
[54] L. Breiman, Pasting small votes for classification in large databases and on-line, Mach. Learn. 36 (1–2) (1999) 85–103.
[55] I. Barandiaran, The random subspace method for constructing decision forests, IEEE Trans. Pattern Anal. Mach. Intell. 20 (8) (1998).
[56] A.S. Householder, Unitary triangularization of a nonsymmetric matrix, J. ACM 5 (4) (1958) 339–342.
[57] A.E. Hoerl, R.W. Kennard, Ridge regression: Biased estimation for nonorthogonal problems, Technometrics 12 (1) (1970) 55–67.
[58] A. Fernández, S. García, M. Galar, R.C. Prati, B. Krawczyk, F. Herrera, Learning from Imbalanced Data Sets, Springer, 2018.
[59] C. Wang, C. Deng, S. Wang, Imbalance-XGBoost: Leveraging weighted and focal losses for binary label-imbalanced classification with XGBoost, 2019, arXiv preprint arXiv:1908.01672.
[60] J.-R. Cano, P.A. Gutiérrez, B. Krawczyk, M. Woźniak, S. García, Monotonic classification: An overview on algorithms, performance measures and data sets, Neurocomputing 341 (2019) 168–182.
[61] T.R. Jensen, B. Toft, Graph Coloring Problems, vol. 39, John Wiley & Sons, 2011.
[62] S. Ranka, V. Singh, CLOUDS: A decision tree classifier for large datasets, in: Proceedings of the 4th Knowledge Discovery and Data Mining Conference, vol. 2, 1998, pp. 2–8.
[63] Q. Meng, G. Ke, T. Wang, W. Chen, Q. Ye, Z.-M. Ma, T. Liu, A communication-efficient parallel algorithm for decision tree, in: Advances in Neural Information Processing Systems, 2016, pp. 1279–1287.
[64] H. Shi, Best-First Decision Tree Learning (Ph.D. thesis), The University of Waikato, 2007.
[65] D. Micci-Barreca, A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems, ACM SIGKDD Explor. Newsl. 3 (1) (2001) 27–32.
[66] Y. Lou, M. Obukhov, BDT: Gradient boosted decision tables for high accuracy and scoring efficiency, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2017, pp. 1893–1901.
[67] Scikit-learn developers, Scikit-learn AdaBoost classifier, 2019, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html. (Accessed 2019-04-02).
[68] M. Kuhn, Caret: Classification and regression training. Package official website, 2019, http://topepo.github.io/caret/index.html. (Accessed 2019-04-02).
[69] S. Chatterjee, fastAdaboost: a fast implementation of Adaboost, 2016, https://cran.r-project.org/package=fastAdaboost. (Accessed 2019-04-02).
[70] E. Alfaro, M. Gámez, N. García, Adabag: An R package for classification with boosting and bagging, J. Stat. Softw. 54 (2) (2013) 1–35, http://www.jstatsoft.org/v54/i02/.
[71] E. Alfaro, M. Gamez, N. Garcia, adabag: Applies Multiclass AdaBoost.M1, SAMME and Bagging, 2018, https://CRAN.R-project.org/package=adabag. (Accessed 2019-04-02).
[72] T. Therneau, B. Atkinson, B. Ripley, Rpart: Recursive partitioning and regression trees, 2018, https://CRAN.R-project.org/package=rpart. (Accessed 2019-04-02).
[73] L. Breiman, Classification and Regression Trees, Routledge, 2017.
[74] V. Jawa, P. Mahajan, Parallelizing AdaBoost on multi core machines using Open MP in C++, 2018, https://github.com/praateekmahajan/parallel-adaboost. (Accessed 2019-04-02).
[75] H. Li, Smile - Statistical machine intelligence and learning engine classification website, 2019, https://haifengl.github.io/smile/classification.html. (Accessed 2019-09-07).
[76] Weka developers, Weka adaboost classifier, 2017, http://weka.sourceforge.net/doc.dev/weka/classifiers/meta/AdaBoostM1.html. (Accessed 2019-04-02).
[77] T. Fagni, Apache Spark AdaBoost classifier, 2015, https://spark-packages.org/package/tizfa/sparkboost. (Accessed 2019-04-02).
[78] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, ACM SIGKDD Explor. Newslett. 11 (1) (2009) 10–18.

[79] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al., Mllib: Machine learning in Apache Spark, J. Mach. Learn. Res. 17 (1) (2016) 1235–1241.

[80] A. Mavrin, Python implementation of logitboost classification algorithm, 2018, https://logitboost.readthedocs.io. (Accessed 2019-09-03).

[81] J. Tuszynski, caTools: R package for LogitBoost algorithm, 2019, https://CRAN.R-project.org/package=caTools. (Accessed 2019-09-03).

[82] K. Hornik, C. Buchta, A. Zeileis, Rweka CRAN package, 2019, https://CRAN.R-project.org/package=RWeka. (Accessed 2019-09-07).

[83] Weka developers, Weka LogitBoost classifier, 2018, http://weka.sourceforge.net/doc.dev/weka/classifiers/meta/LogitBoost.html. (Accessed 2019-09-03).

[84] K. Hornik, C. Buchta, A. Zeileis, Open-source machine learning: R meets Weka, Comput. Stat. 24 (2) (2009) 225–232.

[85] P. Sun, AOSO-Logitboost official github repository, 2015, https://github.com/pengsun/AOSOLogitBoost. (Accessed 2019-09-03).

[86] I. Palit, C.K. Reddy, Scalable and parallel boosting with mapreduce, IEEE Trans. Knowl. Data Eng. 24 (10) (2011) 1904–1916.

[87] Scikit-learn developers, Scikit-learn random forest classifier, 2019, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. (Accessed 2019-04-02).

[88] A. Liaw, M. Wiener, Randomforest: R package of Random Forest, 2018, https://CRAN.R-project.org/package=randomForest. (Accessed 2019-04-02).

[89] N. Gill, E. LeDell, Y. Tang, H2O4GPU: Machine Learning with GPUs in R adn Python, 2019, https://github.com/h2oai/h2o4gpu. (Accessed 2019-04-02).

[90] XGBoost developers, XGBoost Library, 2019, https://xgboost.readthedocs.io/. (Accessed 2019-09-07).

[91] RAPIDS Developers, RAPIDS Cuml - GPU machine learning algorithms, 2019, https://github.com/rapidsai/cuml. (Accessed 2019-09-07).

[92] Apache Spark, APache Spark MLlib ensembles API, 2019, https://spark.apache.org/docs/2.1.0/mllib-ensembles.html. (Accessed 2019-04-02).

[93] H2O.ai, H2O 3: Distributed, fast, and scalable machine learning software, 2019, http://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html (Accessed 2019-04-02).

[94] Weka developers, Weka Random Forest classifier, 2017, http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/RandomForest.html. (Accessed 2019-04-02).

[95] K. Jansson, H. Sundell, H. Boström, GpuRF and gpuERT: efficient and scalable GPU algorithms for decision tree ensembles, in: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, IEEE, 2014, pp. 1612–1621.

[96] K. Jansson, Github repository of gpuRF and gpuERT, 2017, https://github.com/KarlJansson/GPU_tree-ensembles. (Accessed 2019-04-10).

[97] Scikit-learn developers, Scikit-learn Gradient Boosting classifier, 2019, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html. (Accessed 2019-09-06).

[98] Scikit-learn developers, Scikit-learn histogram-based gradient boosting classifier, 2019, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html. (Accessed 2019-09-06).

[99] B. Greenwell, B. Boehmke, J. Cunningham, P. Metcalfe, R package gbm: Generalized boosted regression models, 2019, https://CRAN.R-project.org/package=gbm. (Accessed 2019-04-06).

[100] GBM3 Developers, R Package gbm3: Generalized boosted regression models, 2017, https://github.com/gbm-developers/gbm3. (Accessed 2019-04-06).

[101] Z. Wen, J. Shi, B. He, Q. Li, J. Chen, ThunderGBM: Fast GBDTs and random forests on GPUs, 2019, https://github.com/Xtra-Computing/thundergbm. (Accessed 2019-09-07).

[102] Weka developers, Weka gradient boosting model, 2018, http://weka.sourceforge.net/doc.dev/weka/classifiers/meta/AdditiveRegression.html (Accessed 2019-09-07).

[103] Z. Wen, B. He, R. Kotagiri, S. Lu, J. Shi, Efficient gradient boosted decision tree training on GPUs, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018, pp. 234–243.

[104] J.D. Loyal, A. Maiti, Python implementation of Rotation Forest algorithm, 2019, https://github.com/digital-idiot/RotationForest. (Accessed 2019-09-04).

[105] M. Ballings, D.V. den Poel, rotationForest: Fit and deploy rotation forest models in R, 2017, https://CRAN.R-project.org/package=rotationForest. (Accessed 2019-09-04).

[106] Weka developers, Weka Rotation Forest classifier, 2018, http://weka.sourceforge.net/doc.packages/rotationForest/weka/classifiers/meta/RotationForest.html (Accessed 2019-09-03).

[107] Scikit-learn developers, Scikit-learn Extra-Trees classifier, 2019, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html. (Accessed 2019-09-03).

[108] J. Simm, I.M. de Abril, extraTrees: Extremely Randomized Trees (ExtraTrees) method for classification and regression, 2016, https://CRAN.R-project.org/package=extraTrees. (Accessed 2019-04-03).

[109] Weka developers, Weka Extra-Trees classifier, 2018, http://weka.sourceforge.net/doc.packages/extraTrees/weka/classifiers/trees/ExtraTree.html (Accessed 2019-09-03).

[110] Scikit-learn developers, Scikit-learn Random Patches classifier, 2019, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html (Accessed 2019-04-10).

[111] T. Madl, Scikit-learn compatible implementations of the random rotation ensembles, 2019, https://github.com/tmadl/sklearn-random-rotation-ensembles (Accessed 2019-09-04).

[112] K. Kuo, Sparkxgb: R interface for XGBoost on apache spark, 2019, https://CRAN.R-project.org/package=sparkxgb. (Accessed 2019-04-11).

[113] M.T. Ribeiro, S. Singh, C. Guestrin, Why should i trust you?: Explaining the predictions of any classifier, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 1135–1144.

[114] R. Mitchell, E. Frank, Accelerating the XGBoost algorithm using GPU computing, PeerJ Comput. Sci. 3 (2017) e127.

[115] Microsoft LightGBM developers, LightGBM Library, 2019, https://lightgbm.readthedocs.io/en/latest/index.html. (Accessed 2019-09-14).

[116] Microsoft Azure developers, MMLSpark: Microsoft Machine Learning for Apache Spark, 2019, https://github.com/Azure/mmlspark. (Accessed 2019-09-14).

[117] A. van Mossel, Julia interface for Microsoft's LightGBM, 2017, https://github.com/Allardvm/LightGBM.jl. (Accessed 2019-09-14).

[118] H. Zhang, S. Si, C.-J. Hsieh, GPU-Acceleration for large-scale tree boosting, 2017, arXiv preprint arXiv:1706.08359.

[119] Yandex developers, CatBoost Library, 2019, https://catboost.ai/docs/. (Accessed 2019-09-14).

[120] I. Triguero, S. González, J.M. Moyano, S. García López, J. Alcalá Fernández, J. Luengo Martín, A. Fernández Hilario, J. Díaz, L. Sánchez, F. Herrera, et al., KEEL 3.0: an open source software for multi-stage analysis in data mining, Int. J. Comput. Intell. Syst. 10 (2017) 1238–1249.

[121] S. García, A. Fernández, J. Luengo, F. Herrera, Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power, Inform. Sci. 180 (10) (2010) 2044–2064.

[122] J. Carrasco, S. García, M. Rueda, S. Das, F. Herrera, Recent trends in the use of statistical tests for comparing swarm and evolutionary computing algorithms: Practical guidelines and a critical review, Swarm Evol. Comput. 54 (2020) 100665.

[123] J. Luengo, D. García-Gil, S. Ramírez-Gallego, S. García, F. Herrera, Big Data Preprocessing. Enabling Smart Data, Springer International Publishing, 2020.

[124] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, F. Herrera, A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches, IEEE Trans. Syst. Man Cybern. C 42 (4) (2012) 463–484.

[125] D.J. Hand, Classifier technology and the illusion of progress, Stat. Sci. (2006) 1–14.

[126] A. Holzinger, M. Errath, G. Searle, B. Thurnher, W. Slany, From extreme programming and usability engineering to extreme usability in software engineering education (xp+ ue/spl rarr/xu), in: 29th Annual International Computer Software and Applications Conference (COMPSAC'05), vol. 2, IEEE, 2005, pp. 169–172.

[127] E.L. Allwein, R.E. Schapire, Y. Singer, Reducing multiclass to binary: A unifying approach for margin classifiers, J. Mach. Learn. Res. 1 (Dec) (2000) 113–141.

[128] M. Galar, A. Fernández, E. Barrenechea, H. Bustince, F. Herrera, An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes, Pattern Recognit. 44 (8) (2011) 1761–1776.

[129] T.G. Dietterich, G. Bakiri, Solving multiclass learning problems via error-correcting output codes, J. Artificial Intelligence 2 (1994) 263–286.

[130] G. Tsoumakas, I. Partalas, I. Vlahavas, A taxonomy and short review of ensemble selection, in: Workshop on Supervised and Unsupervised Ensemble Methods and their Applications, 2008, pp. 1–6.

[131] R.M. Cruz, R. Sabourin, G.D. Cavalcanti, Dynamic classifier selection: Recent advances and perspectives, Inf. Fusion 41 (2018) 195–216.

[132] A.S. Britto Jr., R. Sabourin, L.E. Oliveira, Dynamic selection of classifiers - a comprehensive review, Pattern Recognit. 47 (11) (2014) 3665–3680.

[133] T. Woloszynski, M. Kurzynski, P. Podsiadlo, G.W. Stachowiak, A measure of competence based on random classification for dynamic ensemble selection, Inf. Fusion 13 (3) (2012) 207–213.

[134] T.T. Nguyen, A.V. Luong, M.T. Dang, A.W.-C. Liew, J. McCall, Ensemble selection based on classifier prediction confidence, Pattern Recognit. 100 (2020) 107104.

[135] H. Guo, H. Liu, R. Li, C. Wu, Y. Guo, M. Xu, Margin & diversity based ordering ensemble pruning, Neurocomputing 275 (2018) 237–246.

[136] C.E. Brodley, M.A. Friedl, Identifying mislabeled training data, J. Artif. Intell. Res. 11 (1999) 131–167.

[137] S. Verbaeten, A. Van Assche, Ensemble methods for noise elimination in classification problems, in: International Workshop on Multiple Classifier Systems, Springer, 2003, pp. 317–325.

[138] T.M. Khoshgoftaar, P. Rebours, Improving software quality prediction by noise filtering techniques, J. Comput. Sci. Tech. 22 (3) (2007) 387–396.

[139] J. Luengo, S.-O. Shim, S. Alshomrani, A. Altalhi, F. Herrera, CNC-NOS: Class noise cleaning by ensemble filtering and noise scoring, Knowl.-Based Syst. 140 (2018) 27–49.

[140] X. Lu, J. Si, L. Pan, Y. Zhao, Imputation of missing data using ensemble algorithms, in: 2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), vol. 2, IEEE, 2011, pp. 1312–1315.

[141] T. Ishioka, Imputation of missing values for unsupervised data using the proximity in random forests, in: Nternational Conference on Mobile, Hybrid, and on-Line Learning. Nice, 2013, pp. 30–36.

[142] M.G. Rahman, M.Z. Islam, Missing value imputation using decision trees and decision forests by splitting and merging records: Two novel techniques, Knowl.-Based Syst. 53 (2013) 51–65.

[143] B. Zhu, C. He, P. Liatsis, A robust missing value imputation method for noisy data, Appl. Intell. 36 (1) (2012) 61–74.

[144] M.M. Jenghara, H. Ebrahimpour-Komleh, V. Rezaie, S. Nejatian, H. Parvin, S.K.S. Yusof, Imputing missing value through ensemble concept based on statistical measures, Knowl. Inf. Syst. 56 (1) (2018) 123–139.

[145] H. Elghazel, A. Aussem, Unsupervised feature selection with ensemble learning, Mach. Learn. 98 (1–2) (2015) 157–180.

[146] A.K. Das, S. Das, A. Ghosh, Ensemble feature selection using bi-objective genetic algorithm, Knowl.-Based Syst. 123 (2017) 116–127.

[147] V. Bolón-Canedo, A. Alonso-Betanzos, Ensembles for feature selection: A review and future trends, Inf. Fusion 52 (2019) 1–12.

[148] N. García-Pedrajas, A. De Haro-García, Boosting instance selection algorithms, Knowl.-Based Syst. 67 (2014) 342–360.

[149] F. Moreno-Seco, J.M. Inesta, P.J.P. De León, L. Micó, Comparison of classifier fusion methods for classification in pattern recognition tasks, in: Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR), Springer, 2006, pp. 705–713.

[150] P. Clark, R. Boswell, Rule induction with cn2: Some recent improvements, in: European Working Session on Learning, Springer, 1991, pp. 151–163.

[151] W.L. Buntine, A Theory of Learning Classification Rules (Ph.D. thesis), University of Technology, Sydney, 1990.

[152] P. Derbeko, R. El-Yaniv, R. Meir, Variance optimized bagging, in: European Conference on Machine Learning, Springer, 2002, pp. 60–72.

[153] K. Tumer, J. Ghosh, Robust Order Statistics Based Ensembles for Distributed Data Mining, Technical Report, Texas University at Austin, 2001.

[154] J.V. Hansen, Combining Predictors: meta Machine Learning Methods and Bias/variance & Ambiguity Decompositions (Ph.D. thesis), Aarhus University, Computer Science Department, 2000.

[155] Z. Liu, Q. Pan, J. Dezert, J.-W. Han, Y. He, Classifier fusion with contextual reliability evaluation, IEEE Trans. Cybern. 48 (5) (2017) 1605–1618.

[156] Z.-G. Liu, Q. Pan, J. Dezert, A. Martin, Combination of classifiers with optimal weight based on evidential reasoning, IEEE Trans. Fuzzy Syst. 26 (3) (2017) 1217–1230.

[157] M. Albardan, J. Klein, O. Colot, SPOCC: Scalable possibilistic classifier combination-toward robust aggregation of classifiers, Expert Syst. Appl. (2020) 113332.

[158] D.H. Wolpert, Stacked generalization, Neural Netw. 5 (2) (1992) 241–259.

[159] D. Wolpert, W.G. Macready, Combining Stacking with Bagging to Improve a Learning Algorithm, Santa Fe Institute, Technical Report, 30, 1996.

[160] A.K. Seewald, How to make stacking better and faster while also taking care of an unknown weakness, in: Proceedings of the Nineteenth International Conference on Machine Learning, 2002, pp. 554–561.

[161] S. Džeroski, B. Ženko, Is combining classifiers with stacking better than selecting the best one?, Mach. Learn. 54 (3) (2004) 255–273.

[162] C.F. Kurz, W. Maier, C. Rink, A greedy stacking algorithm for model ensembling and domain weighting, BMC Res. Notes 13 (1) (2020) 1–6.

[163] P.K. Chan, S.J. Stolfo, et al., Toward parallel and distributed learning by meta-learning, in: AAAI Workshop in Knowledge Discovery in Databases, 1993, pp. 227–240.

[164] P.K. Chan, S.J. Stolfo, On the accuracy of meta-learning for scalable data mining, J. Intell. Inf. Syst. 8 (1) (1997) 5–28.

[165] A.K. Seewald, J. Fürnkranz, An evaluation of grading classifiers, in: International Symposium on Intelligent Data Analysis, Springer, 2001, pp. 115–124.

[166] S. Tabik, R. Alvear-Sandoval, M. Ruiz, J. Sancho-Gómez, A. Figueiras-Vidal, F. Herrera, A tutorial on ensembles and deep learning fusion with MNIST as guiding thread: A complex heterogeneous fusion scheme reaching 10 digits error, 2020, arXiv preprint arXiv:2001.11486.

[167] H.F. Nweke, Y.W. Teh, G. Mujtaba, M.A. Al-Garadi, Data fusion and multiple classifier systems for human activity detection and health monitoring: Review and open research directions, Inf. Fusion 46 (2019) 147–170.

[168] F. Masulli, G. Valentini, Effectiveness of error correcting output codes in multiclass learning problems, in: International Workshop on Multiple Classifier Systems, Springer, 2000, pp. 107–116.

[169] T. Windeatt, R. Ghaderi, Coding and decoding strategies for multi-class learning problems, Inf. Fusion 4 (1) (2003) 11–21.

[170] K. Crammer, Y. Singer, On the learnability and design of output codes for multiclass problems, Mach. Learn. 47 (2–3) (2002) 201–233.

[171] J. Zhou, H. Peng, C.Y. Suen, Data-driven decomposition for multi-class classification, Pattern Recognit. 41 (1) (2008) 67–76.

[172] M. Saberian, N. Vasconcelos, Multiclass boosting: Margins, codewords, losses, and algorithms, J. Mach. Learn. Res. 20 (137) (2019) 1–68.

[173] M. Jaderberg, V. Dalibard, S. Osindero, W.M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, K. Kavukcuoglu, Population based training of neural networks, 2017, arXiv:1711.09846.

[174] F. Hutter, H.H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: International Conference on Learning and Intelligent Optimization, Springer, 2011, pp. 507–523.

[175] J. Snoek, H. Larochelle, R.P. Adams, Practical bayesian optimization of machine learning algorithms, in: Advances in Neural Information Processing Systems, 2012, pp. 2951–2959.

[176] M. Birattari, Z. Yuan, P. Balaprakash, T. Stützle, F-race and iterated F-race: An overview, in: Experimental Methods for the Analysis of Optimization Algorithms, Springer, 2010, pp. 311–336.

[177] F. Hutter, L. Kotthoff, J. Vanschoren, Automated Machine Learning, Springer, 2019.

[178] M.-A. Zöller, M.F. Huber, Benchmark and survey of automated machine learning frameworks, 2019, arXiv:1904.12054.

[179] P. Probst, B. Bischl, A.-L. Boulesteix, Tunability: Importance of hyperparameters of machine learning algorithms, 2018, arXiv:1802.09596.

[180] P. Probst, M.N. Wright, A.-L. Boulesteix, Hyperparameters and tuning strategies for random forest, Wiley Interdiscipl. Rev.: Data Min. Knowl. Discov. 9 (3) (2019) e1301.

[181] Apache Spark, Apache Spark MLlib gradient boosting classifier, 2019, https://spark.apache.org/docs/2.1.0/mllib-ensembles.html. (Accessed 2019-09-06).

[182] D. Gunning, Explainable artificial intelligence (XAI), Defense Adv. Res. Projects Agency (DARPA), nd Web 2 (2017).

[183] A.B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. García, S. Gil-López, D. Molina, R. Benjamins, F. Herrera, Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI, Inf. Fusion 58 (2020) 82–115.

[184] A. Holzinger, G. Langs, H. Denk, K. Zatloukal, H. Müller, Causability and explainability of artificial intelligence in medicine, Wiley Interdiscipl. Rev. Data Min. Knowl. Discov. 9 (4) (2019) e1312.

[185] A. Van Assche, H. Blockeel, Seeing the forest through the trees: Learning a comprehensible model from an ensemble, in: European Conference on Machine Learning, Springer, 2007, pp. 418–429.

[186] Y. Akiba, S. Kaneda, H. Almuallim, Turning majority voting classifiers into a single decision tree, in: Proceedings Tenth IEEE International Conference on Tools with Artificial Intelligence (Cat. No. 98CH36294), IEEE, 1998, pp. 224–230.

[187] G. Vandewiele, O. Janssens, F. Ongenae, F. De Turck, S. Van Hoecke, GENESIM: genetic extraction of a single, interpretable model, in: NIPS2016, the 30th Conference on Neural Information Processing Systems, 2016, pp. 1–6.

[188] Y. Wang, D. Wang, N. Geng, Y. Wang, Y. Yin, Y. Jin, Stacking-based ensemble learning of decision trees for interpretable prostate cancer detection, Appl. Soft Comput. 77 (2019) 188–204.

[189] N.F. Rajani, R. Mooney, Stacking with auxiliary features for visual question answering, in: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), 2018, pp. 2217–2226.

[190] H. Chen, S. Lundberg, S.-I. Lee, Explaining models by propagating shapley values of local components, 2019, arXiv:1911.11888.

[191] O. Sagi, L. Rokach, Explainable decision forest: Transforming a decision forest into an interpretable tree, Inf. Fusion 61 (2020) 124–138.

[192] R.R. Hoffman, S.T. Mueller, G. Klein, J. Litman, Metrics for explainable AI: Challenges and prospects, 2018, arXiv:1812.04608.

[193] S. Mohseni, N. Zarei, E.D. Ragan, A multidisciplinary survey and framework for design and evaluation of explainable AI systems, 2018, arXiv:1811.11839.

[194] Y. Freund, A more robust boosting algorithm, 2009, arXiv preprint arXiv:0905.2138.

[195] Q. Miao, Y. Cao, G. Xia, M. Gong, J. Liu, J. Song, Rboost: label noise-robust boosting algorithm based on a nonconvex loss function and the numerically stable base learners, IEEE Trans. Neural Netw. Learn. Syst. 27 (11) (2015) 2216–2228.

[196] B. Sun, S. Chen, J. Wang, H. Chen, A robust multi-class adaboost algorithm for mislabeled noisy data, Knowl.-Based Syst. 102 (2016) 87–102.

[197] A. Gómez-Ríos, J. Luengo, F. Herrera, A study on the noise label influence in boosting algorithms: adaboost, GBM and xgboost, in: International Conference on Hybrid Artificial Intelligence Systems, Springer, 2017, pp. 268–280.

[198] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning, in: Advances in Neural Information Processing Systems, 2015, pp. 2962–2970.

[199] Y. Peng, P.A. Flach, C. Soares, P. Brazdil, Improved dataset characterisation for meta-learning, in: International Conference on Discovery Science, Springer, 2002, pp. 141–152.

[200] P. Brazdil, C.G. Carrier, C. Soares, R. Vilalta, Metalearning: Applications to Data Mining, Springer Science & Business Media, 2008.

[201] J. Vanschoren, Meta-learning: A survey, 2018, arXiv preprint arXiv:1810.03548.

[202] N. Cohen-Shapira, L. Rokach, B. Shapira, G. Katz, R. Vainshtein, AutoGRD: Model recommendation through graphical dataset representation, in: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, 2019, pp. 821–830.

[203] R. Vainshtein, A. Greenstein-Messica, G. Katz, B. Shapira, L. Rokach, A hybrid approach for automatic model recommendation, in: Proceedings of the 27th ACM International Conference on Information and Knowledge Management, 2018, pp. 1623–1626.

[204] G. Katz, E.C.R. Shin, D. Song, Explorekit: Automatic feature generation and selection, in: 2016 IEEE 16th International Conference on Data Mining (ICDM), IEEE, 2016, pp. 979–984.