Artificial Intelligence Review
https://doi.org/10.1007/s10462-020-09896-5

A comparative analysis of gradient boosting algorithms

Candice Bentéjac[1] · Anna Csörgő[2] · Gonzalo Martínez-Muñoz[3]

Abstract
The family of gradient boosting algorithms has been recently extended with several interesting proposals (i.e. XGBoost, LightGBM and CatBoost) that focus on both speed and accuracy. XGBoost is a scalable ensemble technique that has demonstrated to be a reliable and efficient machine learning challenge solver. LightGBM is an accurate model focused on providing extremely fast training performance using selective sampling of high gradient instances. CatBoost modifies the computation of gradients to avoid the prediction shift in order to improve the accuracy of the model. This work proposes a practical analysis of how these novel variants of gradient boosting work in terms of training speed, generalization performance and hyper-parameter setup. In addition, a comprehensive comparison between XGBoost, LightGBM, CatBoost, random forests and gradient boosting has been performed using carefully tuned models as well as using their default settings. The results of this comparison indicate that CatBoost obtains the best results in generalization accuracy and AUC in the studied datasets although the differences are small. LightGBM is the fastest of all methods but not the most accurate. Finally, XGBoost places second both in accuracy and in training speed. Finally an extensive analysis of the effect of hyper-parameter tuning in XGBoost, LightGBM and CatBoost is carried out using two novel proposed tools.

Keywords XGBoost · LightGBM · CatBoost · Gradient boosting · Random forest · Ensembles of classifiers

✉ Gonzalo Martínez-Muñoz
gonzalo.martinez@uam.es

Candice Bentéjac
candice.bentejac@u-bordeaux.fr; candice.bentejac@gmail.com

Anna Csörgő
csorgo.anna.erzsebet@hallgato.ppke.hu

[1] College of Science and Technology, University of Bordeaux, Bordeaux, France

[2] Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Budapest, Hungary

[3] Escuela Politéctica Superior, Universidad Autónoma de Madrid, Madrid, Spain

Published online: 24 August 2020

1 Introduction

As machine learning is becoming a critical part of the success of more and more applications—such as credit scoring (Xia et al. 2017), bioactive molecule prediction (Babajide Mustapha and Saeed 2016), solar and wind energy prediction (Torres-Barrán et al. 2017), oil price prediction (Gumus and Kiran 2017), classification of galactic unidentified sources and gravitational lensed quasars (Mirabal et al. 2016; Khramtsov et al. 2019), sentiment analysis (Valdivia et al. 2018), prediction of dementia using electronic health record data (Nori et al. 2019)—it is essential to find models that can deal efficiently with complex data, and with large amounts of it. With that perspective in mind, ensemble methods have been a very effective tool to improve the performance of multiple existing models (Breiman 2001; Friedman 2001; Yoav Freund 1999; Chen and Guestrin 2016). These methods mainly rely on randomization techniques, which consist in generating many diverse solutions to the problem at hand (Breiman 2001), or on adaptive emphasis procedures (e.g. boosting Yoav Freund 1999).

In fact, the above mentioned applications have in common that they all use ensemble methods and, in particular, they use one of the following very recent ensemble methods: eXtreme Gradient Boosting or XGBoost (Chen and Guestrin 2016), Light Gradient Boosting Machine (LightGBM) (Ke et al. 2017) or Categorical Boosting (CatBoost) (Prokhorenkova et al. 2018), with very competitive results. These methods, all based on gradient boosting, propose variants to the original algorithm to improve the training speed and generalization capability. One of these methods, XGBoost, has been consistently placing among the top contenders in Kaggle competitions (Chen and Guestrin 2016). The performance of CatBoost and LightGBM is promising specially in generalization accuracy for the former (Prokhorenkova et al. 2018) and in training speed for the latter (Ke et al. 2017). But these methods are not the only ones to achieve remarkable results over a wide range of problems. Random forest is also well known as one of the most accurate and as a fast learning method independently from the nature of the datasets, as shown by various recent comparative studies (Fernández-Delgado et al. 2014; Caruana and Niculescu-Mizil 2006; Rokach 2016).

This study follows the path of many other previous comparative analysis, such as Fernández-Delgado et al. (2014), Caruana and Niculescu-Mizil (2006), Dietterich (2000), with the intent of covering a gap related to gradient boosting and its more recent variants XGBoost, LightGBM and CatBoost. None of the previous comprehensive analysis included any machine learning algorithm of the gradient boosting family despite of their appealing properties. We have found only two comparative studies that included gradient boosting in their experiments (Zhang et al. 2017; Brown and Mues 2012). In Brown and Mues (2012), a specific analysis of different algorithms in five imbalance credit scoring datasets showed that random forest and gradient boosting performed among the best methods over a wide range of class imbalance percentages. In that study only the number of trees was optimized for gradient boosting. In a recent extensive experimental comparison (Zhang et al. 2017), closer to the present study, several algorithms are compared across multiple datasets. The results of the comparison shows gradient boosting and random forest as the best of the analysis. Although the analysis includes XGBoost library, it is analyzed as a plain gradient boosting algorithm without taking into consideration any of the specific features of XGBoost. The analysis only optimizes the learning rate and depth of the trees in a limited grid of 40 points. In this study we perform an extensive hyper-parameter analysis of XGBoost in addition to the other variants of gradient boosting (LightGMB

and CatBoost) focusing on the specific improvements over gradient boosting proposed in these methods. Specifically, the main improvements of the methods are: the incorporation of a complexity control term in the loss function for XGBoost, selective subsampling of high gradient instances in LightGBM and prevention of prediction shift in the computation of gradients in CatBoost.

The specific objectives of this study are, in the first place, to compare the performance of the three gradient boosting variants (XGBoost, LightGBM and CatBoost) with respect to the algorithm on which it is based (i.e. gradient boosting). The comparison is extended to random forest, which can be considered as a benchmark since many previous comparisons demonstrated its remarkable performance (Fernández-Delgado et al. 2014; Caruana and Niculescu-Mizil 2006; Zhang et al. 2017). The comparison is carried out in terms of accuracy, AUC and training speed. We believe this analysis can be very helpful to researchers and practitioners of various fields. Furthermore, a comprehensive analysis of the process of hyper-parameter setting in XGBoost, LightGBM and CatBoost is performed. For this, we propose two novel methodologies to analyze hyper-parameter tuning. In the first proposal, the variation of the average rank of a method using different hyper-parametrizations is measured when a hyper-parameter value of any given hyper-parameter is changed to any other value. The second proposal for hyper-parameter tuning analysis is a visual tool that analyzes how the performance of a method varies for a set of datasets when any given hyper-parameter is fixed to any given value, that is, when the hyper-parameter is not tuned.

The paper is organized as follows: Section 2 describes the methods of this study, emphasizing the different hyper-parameters that need to be tuned; Sect. 3 presents the results of the comparison; Finally, the conclusions are summarized in Sect. 4.

2 Methodology

2.1 Random forest

One of the most successful machine learning methods is random forest (Breiman 2001). Random forest is an ensemble of classifiers composed of decision trees that are generated using two different sources of randomization. First, each individual decision tree is trained on a random sample with replacement from the original data with the same size as the given training set. The generated bootstrap samples are expected to have approximately $\approx 37\%$ of duplicated instances. A second source of randomization applied in random forest is attribute sampling. For that, at each node split, a subset of the input variables is randomly selected to search for the best split. The value proposed by Breiman to be given to this hyper-parameter is $\lfloor \log_2(\#\text{features})+1 \rfloor$. For classification, the final prediction of the ensemble is given by majority voting.

Based on the Strong Law of Large Numbers, it can be proven that the generalization error for random forests converges to a limit as the number of trees in the forest becomes large (Breiman 2001). The implication of this demonstration is that the size of the ensemble is not a hyper-parameter that really needs to be tuned, as the generalization accuracy of random forest does not deteriorate on average when more classifiers are included into the ensemble. The largest the number of trees in the forest, the most probable the ensemble has converged to its asymptotic generalization error. Actually, one of the main advantages of random forest is that it is *almost* hyper-parameter-free or at least, the default hyper-parameter setting has a remarkable performance on average (Fernández-Delgado et al. 2014). The best two methods of that

comparative study are based on random forest, for which only the value of the number of random attributes that are selected at each split is tuned. The method that placed fifth (out of 179 methods) in the comparison was random forest using the default setting. This could also be seen as a drawback as it is difficult to further improve random forest by hyper-parameter tuning.

Anyhow, other hyper-parameters that may be tuned in random forest are those that control the depth of the decision trees. In general, decision trees in random forest are grown until all leaves are pure. However, this can lead to very large trees. For such cases, the growth of the tree can be limited by setting a maximum depth or by requiring a minimum number of instances per node before or after the split.

Among the set of hyper-parameters that can be tuned for random forest, we evaluate the following ones in this study:

- The number of features to consider when looking for the best split (`max_features`).
- The minimum number of samples (`min_samples_split`) required to split an internal node. This hyper-parameter limits the size of the trees but, in the worst case, the depth of the trees can be as large as $N -$ `min_samples_split`, with N the size of the training data.
- The minimum number of samples (`min_samples_leaf`) required to create a leaf node. The effect of this limit is different from the previous hyper-parameter, as it effectively removes split candidates that are on the limits of the data distribution in the parent node.
- The maximum depth of the tree (`max_depth`). This hyper-parameter limits the depth of the tree independently of the number of instances that are in each node.

2.2 Gradient boosting

Boosting algorithms combine weak learners, i.e. learners slightly better than random, into a strong learner in an iterative way (Yoav Freund 1999). Gradient boosting is a boosting-like algorithm for regression (Friedman 2001). Given a training dataset $D = \{\mathbf{x}_i, y_i\}_1^N$, the goal of gradient boosting is to find an approximation, $\hat{F}(\mathbf{x})$, of the function $F^*(\mathbf{x})$, which maps instances $\mathbf{x}$ to their output values y, by minimizing the expected value of a given loss function, $L(y, F(\mathbf{x}))$. Gradient boosting builds an additive approximation of $F^*(\mathbf{x})$ as a weighted sum of functions

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \rho_m h_m(\mathbf{x}), \tag{1}$$

where ρ_m is the weight of the m^{th} function, $h_m(\mathbf{x})$. These functions are the models of the ensemble (e.g. decision trees). The approximation is constructed iteratively. First, a constant approximation of $F^*(\mathbf{x})$ is obtained as

$$F_0(\mathbf{x}) = \arg\min_{\alpha} \sum_{i=1}^{N} L(y_i, \alpha). \tag{2}$$

Subsequent models are expected to minimize

$$(\rho_m, h_m(\mathbf{x})) = \arg\min_{\rho,h} \sum_{i=1}^{N} L(y_i, F_{m-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i)) \tag{3}$$

However, instead of solving the optimization problem directly, each h_m can be seen as a greedy step in a gradient descent optimization for F^*. For that, each model, h_m, is trained on a new dataset $D = \{\mathbf{x}_i, r_{mi}\}_{i=1}^{N}$, where the pseudo-residuals, r_{mi}, are calculated by

$$r_{mi} = \left[\frac{\partial L(y_i, F(\mathbf{x}))}{\partial F(\mathbf{x})}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})} \tag{4}$$

The value of ρ_m is subsequently computed by solving a line search optimization problem.

This algorithm can suffer from over-fitting if the iterative process is not properly regularized (Friedman 2001). For some loss functions (e.g. quadratic loss), if the model h_m fits the pseudo-residuals perfectly, then in the next iteration the pseudo-residuals become zero and the process terminates prematurely. To control the additive process of gradient boosting, several regularization hyper-parameters are considered. The natural way to regularize gradient boosting is to apply shrinkage to reduce each gradient decent step $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu\rho_m h_m(\mathbf{x})$ with $\nu = (0, 1.0]$. The value of ν is usually set to 0.1. In addition, further regularization can be achieved by limiting the complexity of the trained models. For the case of decision trees, we can limit the depth of the trees or the minimum number of instances necessary to split a node. Contrary to random forest, the default values for these hyper-parameters in gradient boosting are set to harshly limit the expressive power of the trees (e.g. the depth is generally limited to $\approx 3-5$). Finally, another family of hyper-parameters also included in the different versions of gradient boosting are those that randomize the base learners, which can further improve the generalization of the ensemble (Friedman 2002), such as random subsampling without replacement.

The attributes finally tested for gradient boosting are:

- The learning rate (`learning_rate`) or shrinkage ν.
- The maximum depth of the tree (`max_depth`): the same meaning as in the trees generated in random forest.
- The subsampling rate (`subsample`) for the size of the random samples. Contrary to random forest, this is generally carried out without replacement (Friedman 2002).
- The number of features to consider when looking for the best split (`max_features`): as in random forest.
- The minimum number of samples required to split an internal node (`min_samples_split`): as in random forest.

2.3 XGBoost

XGBoost (Chen and Guestrin 2016) is a decision tree ensemble based on gradient boosting designed to be highly scalable. Similarly to gradient boosting, XGBoost builds an additive expansion of the objective function by minimizing a loss function. Considering that XGBoost is focused only on decision trees as base classifiers, a variation of the loss function is used to control the complexity of the trees

$$L_{xgb} = \sum_{i=1}^{N} L(y_i, F(\mathbf{x}_i)) + \sum_{m=1}^{M} \Omega(h_m) \tag{5}$$

$$\Omega(h) = \gamma T + \frac{1}{2}\lambda \|w\|^2 \,, \tag{6}$$

where T is the number of leaves of the tree and w are the output scores of the leaves. This loss function can be integrated into the split criterion of decision trees leading to a pre-pruning strategy. Higher values of γ result in simpler trees. The value of γ controls the minimum loss reduction gain needed to split an internal node. An additional regularization hyper-parameter in XGBoost is shrinkage, which reduces the step size in the additive expansion. Finally, the complexity of the trees can also be limited using other strategies as the depth of the trees, etc. A secondary benefit of tree complexity reduction is that the models are trained faster and require less storage space. Randomization techniques are also implemented in XGBoost both to reduce overfitting and to increase training speed. The randomization techniques included in XGBoost are: random subsamples to train individual trees and column subsampling at tree and tree node levels. Furthermore, XGBoost can be extended to any user-defined loss function by defining a function that outputs the gradient and the hessian (second order gradient) and passing it through the "objective" hyper-parameter.

Moreover, XGBoost proposes a sparsity-aware algorithm for finding the best split. The sparsity of an attribute can be caused by the presence of many zero valued entries and/or missing values. XGBoost automatically removes these entries from the computation of the gain for split candidates. In addition, XGBoost trees learn the default child node in which instances with missing or null values are branched. Other interesting features of XGBoost include monotonic and feature interaction constraints. These features can be specially useful when domain specific information is known. Monotonic constraints force the output of XGBoost for regression to be monotonic (increasing or decreasing) with respect to any set of given input attributes. Feature interaction constraints limit the input attributes that can be combined in the paths from the root to any leaf node. Both constraints are implemented by limiting the set of candidate splits to be considered at each node.

In addition, XGBoost implements several methods to increment the training speed of decision trees not directly related to ensemble accuracy. Specifically, XGBoost focuses on reducing the computational complexity for finding the best split, which is the most time-consuming part of decision tree construction algorithms. Split finding algorithms usually enumerate all possible candidate splits and select the one with the highest gain. This requires performing a linear scan over each sorted attribute to find the best split for each node. To avoid sorting the data repeatedly in every node, XGBoost uses a specific compressed column based structure in which the data is stored pre-sorted. In this way, each attribute needs to be sorted only once. This column based storing structure allows to find the best split for each considered attributes in parallel. Furthermore, instead of scanning all possible candidate splits, XGBoost implements a method based on percentiles of the data where only a subset of candidate splits is tested and their gain is computed using aggregated statistics. This idea resembles the node level data subsampling that is already present in CART trees (Breiman et al. 1984).

The following hyper-parameters were tuned for XGBoost in this study:

- The learning rate (`learning_rate`) or shrinkage ν.
- The minimum loss reduction (`gamma`): The higher this value, the shallower the trees.
- The maximum depth of the tree (`max_depth`)
- The fraction of features to be evaluated at each split (`colsample_bylevel`).
- The subsampling rate (`subsample`): sampling is done without replacement.

2.4 LightGBM

LightGBM (Ke et al. 2017) is an extensive library that implements gradient boosting and proposes some variants. The implementation of gradient boosting in this library has been especially focused on creating a computationally efficient algorithm, which is based on the precomputation of histogram of features, as in XGBoost. The library also includes tens of learning hyper-parameters that allow this model to work in a wide variety of scenarios: The implementation works both in GPU and CPU, it can work as the basic gradient boosting and has many types of randomizations (column randomization, bootstrap subsampling, etc.).

LightGBM implementation also proposes new features (Ke et al. 2017), that mainly are: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). GOSS is a subsample technique used to create the training sets for building the base trees in the ensemble. As in AdaBoost, this technique aims at incrementing the importance of those instances with higher uncertainty in their classifications, which are identified as those instances with higher gradient. When the GOSS option is set, the training sets for the base learners are composed of the top fraction of the instances with the largest gradients (a) plus a random sample fraction (b) retrieved from the instances with the lowest gradients. To compensate for the change of the original distribution, the instances in the low gradient group are weighted up by $(1 - a)/b$ when computing the information gain. EFB technique bundles sparse features into a single feature. This can be done without losing any information when those features do not have non-zero values simultaneously. Both GOSS and EFB provide further training speed gains. EFB can be seen as a feature preprocessing technique and it will not be analyzed in this study as it is a technique that can be pre-applied to any dataset. In addition, for this library we will focus on the analysis of LightGBM with GOSS as the implementation of standard gradient boosting is already covered by Gradient Boosting above.

The following hyper-parameters were tuned for LightGBM GOSS mode (i.e. hyper-parameter `boosting_type` is set to 'goss') in this study:

- The learning rate (`learning_rate`).
- The maximum number of leaves of the tree. In LightGBM, the preferred method for controlling the complexity of the trees is to set a maximum number of leaves. This method gives more flexibility to the shape the trees can adopt (`num_leaves`).
- The fraction of instances a to be selected from the ones with the highest gradients (`top_rate`).
- The fraction of instances b to be sampled from the ones with the lowest gradients (`other_rate`).
- The fraction of features to be evaluated at each split (`feature_fraction_bynode`).

2.5 CatBoost

CatBoost (Prokhorenkova et al. 2018) is a gradient boosting library that aims at reducing the prediction shift that occurs during training. This distribution *shift* is the depart of $F(\mathbf{x_i})|\mathbf{x_i}$ with x_i being a training instance, with respect to $F(\mathbf{x})|\mathbf{x}$ for a test instance $\mathbf{x}$. This shift occurs because during training, gradient boosting is using the same instances for the

estimation of both the gradients and the models that minimize those gradients. The solution proposed in CatBoost (Prokhorenkova et al. 2018) is to estimate the gradients using a sequence of base models that do not include that instance in their training set. To do so, CatBoost first introduces a random permutation in the training instances. The idea in CatBoost (not its implementation) is to build $i = 1, \ldots, N$ base models per each of the M boosting iteration. The ith model of the mth iteration is trained on the first i instances of the permutation and is used to estimate the gradient of the $i + 1$ instance for the $(m + 1)$th boosting iteration. In order, to be independent of the initial random permutation this process is repeated using s different random permutations. Notwithstanding, the implementation of CatBoost is optimized such that a single model is build per iteration that handles all permutations and models. The base models are symmetric trees (or decision tables). These trees are grown by extending level-wise all leaf nodes using the same split condition. Another important feature of CatBoost is how it handles categorical features. In CatBoost categorical features are substituted by a numeric feature that measures the expected target value for each category. In order to avoid over-fitting to the training data, this numeric feature would ideally need to be computed using a different dataset. However, this is not generally possible. The procedure proposed in CatBoost to compute this new feature is similar to the one followed for building the models. That is, for a given random permutation of the instances, the information of instances $< i$ are used to compute the feature value of instance i. Then, several permutations are carried out and the obtained feature value for each instance is averaged. As with EFB in LightGBM, computing target statistics for categorical features is a preprocessing technique. Hence, it will not be taken into consideration in this study. Also, as LightBGM, CatBoost is an extensive library that includes many features as GPU training, standard boosting, and includes tens of hyper-parameters to adapt to many possible learning situations. This library also includes standard gradient boosting. However, in this analysis, we will focus in the implementation that prevents prediction shift (named Ordered CatBoost).

The following hyper-parameters were tuned for CatBoost with Ordered mode (i.e. hyper-parameter `boosting_type` is set to 'Ordered') in this study:

- The learning rate (`learning_rate`).
- The maximum depth of the tree (`depth`). Since symmetric trees, the ones preferably used in CatBoost, are complete trees, the value of this hyper-parameter has a different effect than similar hyper-parameters in other models. High depth values could lead to huge trees that could cause memory problems.
- Number of gradient steps to compute the value of the leaves (`leaf_estimation_iterations`).
- Regularization for the L2 coefficient (`l2_leaf_reg`).

3 Experimental results

In this section, an extensive comparative analysis of the efficiency of random forest, gradient boosting, XGBoost, GOSS LightGBM and Ordered CatBoost models is carried out. For the experiments, 28 different datasets coming from the UCI repository (Lichman 2013) were considered. These datasets come from different fields of application, and have different number of attributes, classes and instances. The characteristics of the analyzed datasets

Table 1 Characteristics of the studied datasets

Name	Inst.	Attrs.	Miss.	Class.
Australia	690	14	Yes	2
Banknote	1371	5	No	2
Breast Cancer	699	10	Yes	2
Dermatology	366	33	Yes	6
Diabetes	768	20	Yes	2
Echo	74	12	Yes	2
Ecoli	336	8	No	8
German	1000	20	No	2
Heart	270	13	No	2
Heart Cleveland	303	13	Yes	2
Hepatitis	155	19	Yes	2
Ionosphere	351	34	No	2
Iris	150	4	No	3
Liver	583	10	No	2
Magic04	19020	11	No	2
Parkinsons	197	23	No	2
Phishing	1353	10	No	3
Segment	2310	19	No	7
Sonar	208	60	No	2
Soybean	675	35	Yes	18
Spambase	4601	57	Yes	2
Teaching	151	5	No	3
Thyroid	215	21	No	3
Tic-Tac-Toe	958	9	No	2
Vehicle	946	18	No	4
Vowel	990	10	No	11
Waveform	5000	21	No	3
Wine	178	13	No	3

are shown in Table 1, which displays for each dataset its name, number of instances, number of attributes, if the dataset has missing values and number of classes. For this experiment, the implementation of `scikit-learn` (version 0.21.2) package (Pedregosa et al. 2011) was used for random forest and gradient boosting. For XGBoost, the `XGBoost` package[1] was used. LightGBM[2] and CatBoost[3] libraries were used for those methods. In addition, for the comparison, the different methods were analyzed tuning the hyper-parameters using a grid search as well as using the default hyper-parameters of the corresponding packages. All packages except CatBoost have a fixed set of default hyper-parameters. In CatBoost, the default behaviour includes an adaptive selection of some hyper-parameters, such as the learning rate, on the fly.

[1] https://github.com/dmlc/xgboost (version 0.6).
[2] https://github.com/microsoft/LightGBM/tree/master/python-package (version 2.3.0).
[3] https://catboost.ai/docs/concepts/python-installation.html (version 0.16).

Table 2 Default values and possible values for every hyper-parameter in the normal grid search for random forest, gradient boosting, XGBoost, LightGBM (goss) and CatBoost (Ordered)

Hyper-parameter	Default value	Grid search values
Random forest		
`max_depth`	Unlimited	5, 8, 10, unlimited
`min_samples_split`	2	2, 5, 10, 20
`min_samples_leaf`	1	1, 25, 50, 70
`max_features`	sqrt	log2, 0.25, sqrt, 1.0
Gradient boosting		
`learning_rate`	0.1	0.025, 0.05, 0.1, 0.2, 0.3
`max_depth`	3	2, 3, 5, 7, 10, unlimited
`min_samples_split`	2	2, 5, 10, 20
`max_features`	1.0	log2, sqrt, 0.25, 1.0
`subsample`	1	0.15, 0.5, 0.75, 1.0
XGBoost		
`learning_rate`	0.1	0.025, 0.05, 0.1, 0.2, 0.3
`gamma`	0	0, 0.1, 0.2, 0.3, 0.4, 1.0, 1.5, 2.0
`max_depth`	3	2, 3, 5, 7, 10, 100
`colsample_bylevel`	1	log2, sqrt, 0.25, 1.0
`subsample`	1	0.15, 0.5, 0.75, 1.0
LightGBM (boosting type *goss*)		
`learning_rate`	0.1	0.025, 0.05, 0.1, 0.2, 0.3
`num_leaves`	31	3, 7, 15, 31, 127, 1024
`top_rate`	0.2	0.2, 0.4, 0.6, 0.7
`other_rate`	0.1	0.05, 0.1, 0.3
`feature_fraction_bynode`	1	log2, sqrt, 0.25, 1.0
CatBoost (boosting type *Ordered*)		
`learning_rate`	Var.	0.025, 0.05, 0.1, 0.2, 0.3
`max_depth`	6	3, 6, 9
`leaf_estimation_iterations`	1 or 10	1, 10
`l2_leaf_reg`	3	1, 3, 6, 9

The comparison was carried out using stratified 10-fold cross-validation. For each dataset and partition of the data into train and test, the following procedure was carried out for the five tested methods: (1) The optimum hyper-parameters for each method were estimated with stratified 10-fold cross-validation within the training set using a grid search. A wide range of hyper-parameter values is explored in the grid search. For each of the five methods, these values are shown in Table 2. (2) The set of hyper-parameters of the grid search giving the best average prediction accuracy was used to train the corresponding ensemble using the whole training partition. In addition, for binary datasets with class unbalance higher than $\approx 65\%$ for the majority class, the hyper-parameters giving the best AUC were also used to build an ensemble on the whole training data; (3) Additionally, for XGBoost, GOSS LightGBM and Ordered CatBoost, ensembles were trained on the whole training set for all possible combination of hyper-parameters given in Table 2. This allows us, in combinations with step (1), to test for different alternative grids (more details down below); (4) The default sets of hyper-parameters for each method were additionally used to

train an ensemble of each type (Table 2 shows the default values for each hyper-parameter and ensemble type); (5) The generalization accuracy of the five ensembles selected in (2) and of the ensembles with default hyper-parametrization is estimated in the left out test set. The AUC is measured for the five models with highest AUC in train. (5) In addition, the accuracy of all the XGBoost, GOSS LightGBM and Ordered CatBoost ensembles trained in step (3) is computed using the test set.

All ensembles were composed of 200 decision trees. Note that the size of the ensemble is not a hyper-parameter that needs to be tuned in ensembles of classifiers (Friedman 2001; Breiman 2001). For random forest like ensembles, as more trees are combined into the ensemble, the generalization error tends to an asymptotic value (Breiman 2001). For gradient boosting like ensembles, the generalization performance can deteriorate with the number of elements in the ensemble especially for high learning rate values. However, this effect can not only be neutralized with the use of lower learning rates (or shrinkage) but reverted (Friedman 2001). The conclusion of Friedman (2001) is that the best option to regularize gradient boosting is to fix the number of models to the highest computationally feasible value and to tune the learning rate. Although XGBoost has its own mechanism to handle missing values, we decided not to use it in order to perform a fairer comparison with respect to random forest and gradient boosting, as the implementation of decision trees in `scikit-learn` does not handle missing values. Instead, we used a class provided by `scikit-learn` to impute missing values by the mean of the whole attribute. Similarly, EFB features and target statistic for categorical features were not used in LightGBM and CatBoost respectively.

3.1 Results

Table 3 displays the average accuracy and standard deviation (after the $\pm$ sign) for: XGBoost with default hyper-parameters (shown with label D. XGB), tuned XGBoost (as T. XGB in the table), random forest with default hyper-parametrization (with D. RF), tuned random forest (T. RF), default gradient boosting (D. GB), tuned gradient boosting (T. GB), tuned GOSS LightGBM (T. LGB goss), default GOSS LightGBM (D. LGB goss), tuned Ordered CatBoost (T. Cat ord) and default ordered CatBoost (D. Cat ord). The best accuracy for each dataset is highlighted using a bold font.

From Table 3, it can be observed that the method that obtains the best performance in relation to the number of datasets with the highest accuracy is tuned gradient boosting (in 7 out of 28 datasets). After that, the methods in order are: tuned XGBoost and default CatBoost, that achieves the best results in 6 datasets; tuned CatBoost and default LightGBM in 4; default gradient boosting, tuned and default random forest and tuned LightGBM in 3; and default XGBoost in 1. As it can be observed, the default performances of the five tested algorithms are quite different. Default random forest and default Ordered CatBoost are the methods that perform more evenly with respect to their tuned counterparts. Except for a few datasets, the differences in performance are very small for both configurations of random forest. In fact, the difference between the tuned and the default hyper-parametrizations of random forest is below 0.5% in 18 out of 28 datasets. For Catboost, the default parametrization works very well except for a few exceptions (*Teaching* and *Vowel*). This occurs because CatBoost has an inner tuning of the learning rate in its default mode. Default LightGBM obtains a good number of best results, but its performance is uneven across datasets. Although the difference in average rank is not statistically significant between Default and Tuned LightGBM, the

Table 3 Average accuracy and standard deviation for XGBoost, random forest, gradient boosting, LightGBM with GOSS and Ordered CatBoost, all using default and tuned hyper-parameter settings

Dataset	D. XGB	T. XGB	D. RF	T. RF	D. GB	T. GB	D. LGB goss	T. LGB goss	D. Cat ord	T. Cat ord
Australian	86.94% ± 2.73	87.53% ± 3.47	87.26% ± 3.83	86.08% ± 3.42	86.23% ± 3.41	86.38% ± 3.39	**87.54% ± 3.31**	86.81% ± 3.34	**87.54% ± 2.41**	86.96% ± 3.09
Banknote	99.64% ± 0.59	99.64% ± 0.49	99.34% ± 0.60	99.13% ± 0.71	99.71% ± 0.48	99.78% ± 0.33	99.71% ± 0.58	99.71% ± 0.48	**99.86% ± 0.43**	99.71% ± 0.48
Breast	96.28% ± 1.15	95.99% ± 1.68	**96.99% ± 1.36**	96.85% ± 1.26	96.71% ± 1.58	96.42% ± 1.61	95.71% ± 2.40	96.57% ± 1.95	**96.99% ± 2.08**	**96.99% ± 2.08**
Cleveland	81.14% ± 7.25	83.16% ± 7.94	82.46% ± 6.77	82.46% ± 8.03	83.78% ± 6.48	82.16% ± 8.35	81.48% ± 4.64	**84.50% ± 5.06**	84.13% ± 3.95	82.80% ± 5.16
Dermatology	96.74% ± 3.21	97.27% ± 3.24	97.27% ± 3.29	97.30% ± 3.24	96.47% ± 3.28	96.18% ± 2.81	96.47% ± 2.11	96.78% ± 2.24	97.53% ± 2.33	**98.08% ± 1.77**
Diabetes	75.65% ± 5.11	76.56% ± 4.50	76.69% ± 3.45	76.69% ± 4.89	**76.81% ± 4.43**	76.30% ± 3.74	74.22% ± 3.74	75.79% ± 2.82	76.70% ± 3.59	75.78% ± 2.80
Echo	94.46% ± 6.80	**98.75% ± 3.75**	**98.75% ± 3.75**	97.32% ± 5.37	97.32% ± 5.37	95.89% ± 6.29	95.71% ± 6.55	97.14% ± 5.71	98.57% ± 4.29	98.57% ± 4.29
Ecoli	86.87% ± 5.31	89.05% ± 4.12	89.07% ± 5.00	**89.11% ± 4.71**	87.25% ± 6.88	87.81% ± 4.77	83.35% ± 5.50	86.14% ± 6.92	88.57% ± 5.45	88.90% ± 6.27
German	**79.00% ± 4.22**	77.40% ± 4.13	76.40% ± 4.48	75.80% ± 4.17	76.70% ± 5.12	77.20% ± 3.99	77.00% ± 4.71	77.20% ± 4.64	77.30% ± 4.54	76.20% ± 4.69
Heart	79.26% ± 5.29	84.07% ± 5.98	83.33% ± 5.56	**84.44% ± 5.19**	81.85% ± 7.67	83.70% ± 5.29	81.48% ± 5.74	84.07% ± 4.07	83.33% ± 5.04	**84.44% ± 5.69**
Hepatitis	59.21% ± 8.28	**67.00% ± 6.56**	65.54% ± 12.35	61.83% ± 12.68	56.58% ± 11.90	64.96% ± 13.08	59.96% ± 5.83	64.46% ± 10.96	63.25% ± 7.92	65.79% ± 7.58
Ionosphere	92.56% ± 2.69	92.59% ± 3.17	93.44% ± 2.88	93.16% ± 2.91	**93.72% ± 2.51**	92.85% ± 3.02	91.76% ± 5.86	93.21% ± 4.55	93.18% ± 4.55	92.90% ± 4.73
Iris	92.67% ± 6.29	94.00% ± 4.67	94.67% ± 4.99	92.67% ± 6.29	94.67% ± 4.99	94.67% ± 4.99	92.67% ± 6.96	94.67% ± 4.99	**95.33% ± 4.27**	94.67% ± 4.99
Liver	68.65% ± 4.69	68.11% ± 6.12	67.76% ± 5.03	67.58% ± 4.07	69.33% ± 5.26	70.16% ± 4.42	**70.45% ± 6.17**	69.24% ± 5.98	68.99% ± 4.74	69.47% ± 3.51
Magic04	87.47% ± 0.57	88.63% ± 0.48	88.19% ± 0.42	88.18% ± 0.46	86.85% ± 0.37	**88.83% ± 0.45**	88.48% ± 0.56	88.42% ± 0.44	88.43% ± 0.62	88.76% ± 0.48
Newthyroid	95.80% ± 3.26	95.80% ± 3.26	**96.75% ± 2.94**	96.28% ± 3.48	**96.75% ± 2.94**	96.28% ± 3.48	96.21% ± 3.56	94.83% ± 4.50	96.26% ± 2.84	96.71% ± 3.03
Parkinsons	92.14% ± 5.72	90.14% ± 4.01	90.70% ± 5.60	90.70% ± 4.62	91.14% ± 6.00	91.20% ± 4.23	90.17% ± 5.13	92.64% ± 5.16	**92.72% ± 5.45**	92.22% ± 5.88
Phishing	89.65% ± 2.44	**91.13% ± 1.30**	88.63% ± 3.38	89.51% ± 2.42	90.62% ± 1.76	90.25% ± 2.19	90.02% ± 1.96	90.98% ± 2.45	89.87% ± 2.20	90.68% ± 1.96
Segment	98.48% ± 0.70	98.70% ± 0.77	98.01% ± 0.80	98.18% ± 0.67	97.75% ± 0.74	98.35% ± 0.64	**98.79% ± 0.79**	98.53% ± 0.62	96.97% ± 0.58	98.61% ± 0.61
Sonar	85.59% ± 6.44	86.97% ± 4.84	83.64% ± 3.87	85.59% ± 4.83	84.66% ± 4.61	**88.95% ± 4.32**	83.66% ± 7.09	86.97% ± 5.03	85.16% ± 7.16	83.61% ± 6.25
Soybean	94.80% ± 3.35	**95.22% ± 2.96**	94.06% ± 2.27	94.65% ± 2.57	93.63% ± 2.86	93.58% ± 3.45	93.92% ± 3.88	93.78% ± 2.40	94.56% ± 3.36	94.82% ± 2.70

Table 3 (continued)

Dataset	D. XGB	T. XGB	D. RF	T. RF	D. GB	T. GB	D. LGB goss	T. LGB goss	D. Cat ord	T. Cat ord
Spambase	95.17% ± 1.29	95.57% ± 1.28	95.46% ± 1.38	95.48% ± 1.26	94.59% ± 1.43	**96.11% ± 1.20**	95.65% ± 0.51	95.65% ± 0.93	95.46% ± 0.77	95.70% ± 0.59
Teaching	63.55% ± 8.00	64.26% ± 14.15	65.51% ± 8.95	63.41% ± 11.13	62.76% ± 7.38	**68.75% ± 14.65**	51.58% ± 12.81	61.68% ± 8.37	56.83% ± 6.68	63.77% ± 7.59
Tic-tac-toe	96.56% ± 1.63	**100.00% ± 0.00**	95.52% ± 1.68	95.62% ± 1.38	90.09% ± 2.77	**100.00% ± 0.00**	**100.00% ± 0.00**	**100.00% ± 0.00**	**100.00% ± 0.00**	**100.00% ± 0.00**
Vehicle	78.74% ± 3.29	77.18% ± 2.31	74.50% ± 3.55	74.13% ± 3.98	78.15% ± 1.89	77.79% ± 2.04	77.43% ± 3.50	**78.85% ± 2.39**	75.04% ± 2.17	77.54% ± 3.36
Vowel	92.63% ± 3.53	95.86% ± 2.14	97.47% ± 1.37	**97.78% ± 1.68**	93.23% ± 3.32	96.77% ± 2.01	95.25% ± 2.48	95.86% ± 2.91	89.09% ± 1.96	96.87% ± 2.14
Waveform	85.72% ± 1.05	85.72% ± 1.77	85.42% ± 1.73	85.62% ± 1.46	85.32% ± 0.85	**85.86% ± 1.18**	84.78% ± 1.41	85.84% ± 1.72	85.40% ± 1.93	85.74% ± 1.75
Wine	97.18% ± 2.83	**98.82% ± 2.37**	98.26% ± 2.66	98.26% ± 2.66	97.74% ± 2.78	**98.82% ± 2.37**	96.13% ± 5.41	97.18% ± 3.76	97.81% ± 3.66	97.21% ± 3.66

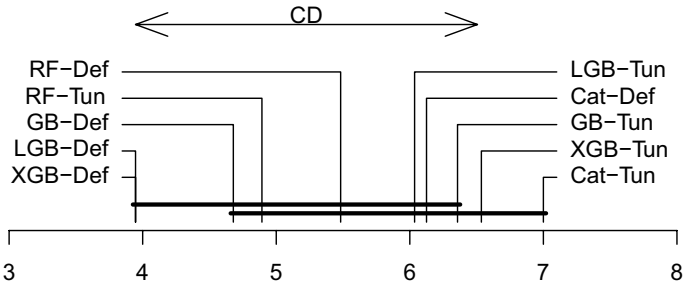

Fig. 1 Average ranks (a higher rank is better) for the tested methods across 28 datasets (Critical difference CD= 2.56)

latter is better than its default version in 75% of the datasets. More importantly when the default version is better, the differences in accuracy are small (1.7% in the most favorable case, *Liver*) but when the default is worse, the differences can be huge (as large as 17%, in *Teaching*). Default XGBoost and gradient boosting perform generally worse than their tuned versions. However, this is not always the case. This is especially evident in three cases for XGBoost (*German*, *Parkinson* and *Vehicle*), where the default hyper-parametrization for XGBoost achieves a better performance than the tuned XGBoost. These cases are a combination of two factors: noisy datasets and good default settings. The hyper-parameter estimation process, even though it is performed within-train cross-validation, may overfit the training set specially in noisy datasets. In fact, it has been shown that reusing the training data multiple times can lead to overfitting (Dwork et al. 2015). On the other hand, in these datasets, the default setting is one of the hyper-parametrizations that obtains the best results both in train and test (among the best $\approx 5\%$).

In order to summarize the figures shown in Table 3, we applied the methodology proposed in Demšar (2006). This methodology compares the performance of several models across multiple datasets. The comparison is carried out in terms of average rank of the performance of each method in the tested datasets. The results of this methodology are shown graphically in Fig. 1. In this plot, a higher rank indicates a better performance. Statistical differences among average ranks are determined using a Nemenyi test. There are no statistical significant differences in average ranks between methods that are connected with a horizontal solid line. The critical distance over which the differences are considered significant is shown in the plot for reference (CD = 2.56 for 10 methods, 28 datasets and p-value < 0.05).

From Fig. 1, it can be observed that there are not many statistically significant differences among the average ranks of the ten tested methods. The best methods in terms of average rank are first tuned Ordered CatBoost and then tuned XGBoost. These two methods present statistically significant differences with respect to default XGBoost and LightGBM. The next best methods are in order: tuned GB, default Ordered CatBoost, tuned GOSS LightGBM and default random forest.

In order to study the performance of these methods using other metrics, we obtained the average AUC for a selection of binary datasets with unbalanced classes. Datasets with a majority class above $\approx$65% were selected. The results are shown in Table 4. We only show the result for the best of each model, that is, the tuned models except for random forest, for which we use the default version. In the last row of the table the average rank is shown (higher is better). The results are similar to those of average accuracy

Table 4 Average AUC and standard deviation for unbalanced datasets

Dataset	T. XGB	D. RF	T. GB	T. LGB goss	T. Cat ord
Breast	99.09% ± 0.73	99.25% ± 0.60	99.25% ± 0.65	99.26% ± 0.55	**99.38% ± 0.60**
Diabetes	83.69% ± 4.83	82.59% ± 4.62	**84.22% ± 4.17**	83.28% ± 4.73	83.62% ± 3.97
Echo	**99.00% ± 3.00**	**99.00% ± 3.00**	**99.00% ± 3.00**	**99.00% ± 3.00**	**99.00% ± 3.00**
German	79.62% ± 7.27	**80.04% ± 6.48**	79.45% ± 6.67	79.01% ± 6.90	80.00% ± 6.19
Liver	71.44% ± 6.78	**74.55% ± 4.49**	72.40% ± 5.81	71.23% ± 6.07	72.57% ± 6.27
Parkinsons	97.42% ± 1.93	96.98% ± 2.52	96.49% ± 2.44	97.42% ± 3.01	**97.95% ± 1.51**
Tic-tac-toe	**100.00% ± 0.00**	99.87% ± 0.12	**100.00% ± 0.01**	**100.00% ± 0.01**	**100.00% ± 0.00**
Ave. rank	2.86	2.79	2.86	2.57	3.93

in the sense that the ordering of the methods is similar. The best method is Ordered CatBoost followed by XGBoost and Gradient Boosting, then random forest and finally LightGBM.

In Table 5, the average training execution time (in seconds) for the analyzed datasets is shown. For the methods using the default settings, the table shows the training time. For the methods that have been tuned using grid search, two numbers are shown separated with a '+' sign. The first number corresponds to the time spent in the within-train 10-fold cross-validated grid search. The second figure corresponds to the training time of each method, once the optimum hyper-parameters have been estimated. The last row of the table reports the average ratio of each execution time with respect to the execution time of XGBoost using the default setting. All experiments were run using an eight-core Intel® Core™ i7-4790K CPU@4.00GHz processor. The reported times are sequential times in order to have a real measure of the needed computational resources, even though the grid search was performed in parallel using all available cores. This comparison is fair independently of whether the learning algorithms include internal multi-thread optimizations or not. For instance random forest and XGBoost include multi-thread optimizations in their code to compute the splits in XGBoost and to train each single tree in random forest whereas gradient boosting does not. LightGBM and CatBoost include also multithreaded implementations for both CPU and GPU. Only single-thread CPU implementations were used. Notwithstanding, given that the grid search procedure is fully parallelizable, the within-training CPU parallelization optimizations of the different methods do not reduce the end-to-end time required to perform a grid search in a real setting.

As it can be observed from Table 5, finding the best hyper-parameters to tune the classifiers through the grid search is a rather costly process. In fact, the end-to-end training time of the tuned models is clearly dominated by the grid search process, which contributes with a percentage over 99.9% to the training time. Since the size of the grid is different for different classifiers (i.e. 3840, 256, 1920, 1440 and 120 for XGB, RF, GB, GOSS LGB and Ordered CatBoost respectively), the time dedicated to finding the best hyper-parameters is not directly comparable between classifiers. Anyhow, when measuring the training time of a model it is important to consider how complex is to find a reasonable set of hyper-parameters and not only the time needed to built the last model.

However, when it comes to fitting a single ensemble to the training data without taking into account the grid search time, GOSS LightGBM shows the fastest performance on average followed by XGBoost. The time necessary to train XGBoost given a set of hyper-parameters is about 1.6 times slower that training a GOSS LightGBM ensemble, 3.5 times

Table 5 Average execution time (in seconds) for training the tested methods (more details in the text)

Dataset	D. XGB	T. XGB	D. RF	T. RF	D. GB	T. GB	D. LGB goss	T. LGB goss	D. Cat ord	T. Cat ord
Australian	0.10	2775 + 0.08	0.32	766 + 0.30	0.12	6053 + 0.61	0.06	1181 + 0.05	4.78	8250 + 6.23
Banknote	0.10	2874 + 0.06	0.40	991 + 0.39	0.23	5410 + 0.15	0.10	1683 + 0.11	5.87	11771 + 3.34
Breast	0.06	1733 + 0.04	0.28	744 + 0.28	0.16	4071 + 0.35	0.05	780 + 0.05	3.54	3723 + 2.42
Cleveland	0.05	1357 + 0.02	0.28	700 + 0.26	0.10	3676 + 0.10	0.03	581 + 0.03	3.86	4941 + 2.75
Dermatology	0.46	11014 + 0.27	0.27	727 + 0.27	0.76	17840 + 0.93	0.15	3016 + 0.21	3.24	14315 + 14.11
Diabetes	0.08	2982 + 0.06	0.35	798 + 0.34	0.13	6612 + 0.50	0.06	1246 + 0.08	4.78	9978 + 12.51
Echo	0.02	532 + 0.01	0.25	671 + 0.25	0.05	1608 + 0.08	0.01	232 + 0.01	1.81	1896 + 1.15
Ecoli	0.17	4518 + 0.11	0.27	715 + 0.28	0.67	15448 + 1.25	0.09	2001 + 0.12	4.25	17992 + 14.28
German	0.17	5453 + 0.16	0.37	790 + 0.37	0.13	8589 + 0.48	0.08	1398 + 0.09	4.34	7837 + 6.94
Heart	0.05	1268 + 0.03	0.27	716 + 0.27	0.12	3579 + 0.10	0.03	537 + 0.03	2.93	4692 + 2.39
Hepatitis	0.04	1071 + 0.02	0.26	686 + 0.26	0.11	2756 + 0.11	0.02	371 + 0.02	2.86	3332 + 2.93
Ionosphere	0.14	2487 + 0.05	0.34	893 + 0.35	0.27	3982 + 0.22	0.07	1489 + 0.09	20.67	23103 + 11.95
Iris	0.04	1308 + 0.03	0.25	678 + 0.25	0.29	6555 + 0.39	0.02	633 + 0.05	1.08	3622 + 2.67
Liver	0.07	2308 + 0.05	0.35	804 + 0.31	0.16	5559 + 0.28	0.05	947 + 0.05	4.29	12397 + 6.68
Magic04	3.32	123764 + 7.86	11.39	11860 + 9.44	1.50	160831 + 48.07	1.09	27085 + 2.02	18.43	39787 + 64.67
New-thyroid	0.05	1728 + 0.04	0.26	686 + 0.26	0.30	7426 + 0.42	0.04	914 + 0.06	2.11	8524 + 9.50
Parkinsons	0.05	1169 + 0.03	0.27	697 + 0.28	0.10	2539 + 0.11	0.02	571 + 0.04	9.80	26291 + 11.25
Phishing	0.36	12464 + 0.75	0.33	816 + 0.37	0.80	31504 + 1.07	0.21	3990 + 0.19	1.29	4895 + 5.30
Segment	3.22	73309 + 2.21	0.77	1451 + 0.79	2.25	62647 + 2.71	2.89	39974 + 2.60	49.97	82075 + 128.78
Sonar	0.15	2442 + 0.05	0.31	828 + 0.33	0.32	3729 + 0.28	0.04	1136 + 0.11	23.10	34174 + 8.85
Soybean	3.73	104873 + 2.69	0.31	762 + 0.31	3.14	75660 + 5.51	0.97	16133 + 1.07	10.77	61748 + 41.22
Spambase	1.83	47412 + 1.46	1.40	2016 + 0.73	0.32	33088 + 4.79	1.27	19179 + 2.78	22.20	32840 + 59.34
Teaching	0.05	1879 + 0.06	0.26	676 + 0.26	0.35	8955 + 0.35	0.02	651 + 0.05	0.90	3217 + 2.94
Tic-tac-toe	0.08	2842 + 0.08	0.32	746 + 0.32	0.13	8184 + 0.45	0.05	965 + 0.07	3.18	2951 + 2.57
Vehicle	0.60	17217 + 0.38	0.43	843 + 0.42	0.70	24987 + 0.88	0.30	6425 + 0.34	8.47	33313 + 34.09

Table 5 (continued)

Dataset	D. XGB	T. XGB	D. RF	T. RF	D. GB	T. GB	D. LGB goss	T. LGB goss	D. Cat ord	T. Cat ord
Vowel	1.96	48726 + 1.24	0.57	977 + 0.50	2.31	61085 + 7.67	1.93	30567 + 2.25	48.99	83076 + 137.04
Waveform	3.34	111562 + 1.74	2.46	3268 + 1.58	1.66	96197 + 11.44	3.14	55333 + 1.10	32.15	59940 + 52.92
Wine	0.07	2104 + 0.05	0.26	700 + 0.26	0.28	7197 + 0.33	0.04	887 + 0.05	6.75	26097 + 22.14
Ave. ratio	1.0	29043.7 + 0.8	3.6	8953.4 + 3.5	2.4	69238.2 + 4.3	0.6	11922.6 + 0.8	52.4	106234.9 + 73.9

faster than training a random forest, 2.4-4.3 times faster than training a gradient boosting model and 52 times faster than training an Ordered CatBoost ensemble. However, the differences are uneven across datasets. Ordered CatBoost is clearly the slowest method taking the longest time to compute the tested grid in spite of being the smallest grid. Despite the inner optimizations implemented, the process for reducing the prediction shift is costly. However, Ordered CatBoost performs, relatively to other models, better the larger the dataset is. For instance, in *Magic04* (19020 instances, 10 attributes) the time needed for the computation of the XGBoost grid search is about 3 times slower than that for Ordered CatBoost and in *Sonar* (208 instances, 60 attributes) computing the Ordered CatBoost grid is about 14 slower than the time needed for XGBoost grid. The difference between XGBoost and Gradient Boosting can be observed in the time employed in the grid search by those methods. XGBoost takes less than half of the time to look for the best hyper-parameter setting than gradient boosting, despite the fact that its grid size is twice the size of the grid of gradient boosting. Finally, for some multiclass problems, as *Segment*, *Soybean* or *Vowel*, the execution time of XGBoost, gradient boosting, LightGBM and CatBoost deteriorates in relation to random forest since for multi-class problems, gradient boosting based models train one base tree per class and iteration.

3.2 Analysis of hyper-parametrization

In order to further analyze and understand the hyper-parametrization of XGBoost, GOSS LightGBM and Ordered CatBoost, we propose two novel analyses of the hyper-parametrizations. Firstly, we propose an analysis to measure how the model performance is affected when one hyper-parameter value is changed to another value. This analysis was carried out for XGBoost only. Secondly, we analyze the effect of having a hyper-parameter value fixed instead of doing a grid search over it on a method's performance.

These further experiments are carried out with two objectives for XGBoost. The first objective is to try to select a better default hyper-parametrization for XGBoost. The second is to explore alternative grids for XGBoost. For the first objective, we have analyzed, for each hyper-parameter, the relation among single value assignments. To do so, we have computed the average rank (in test error) across all datasets for the ensembles trained using all hyper-parameter configurations as given in Table 2 for XGBoost. Recall that we have analyzed 3840 possible hyper-parameter configurations. Then, for each given hyper-parameter, say *HParamX*, and hyper-parameter value, say *HParamX_valueA*, we compute the percentage of times that the average rank improves when *HParamX=HParamX_valueA* is changed to another value *HParamX_valueB* and no other hyper-parameter is modified. These results are shown in Tables 6, 7, 8, 9 and 10. The tables are to be read as follows: each cell of the table indicates the % of times the average rank improves when modifying the value on the corresponding top row by the value on the corresponding first column of

Table 6 Percentage of times the average rank of XGBoost improves when changing the `learning_rate` from the value in the top row to the value in the first column

%	0.025	0.05	0.1	0.2	0.3
0.025	0.0	5.1	15.9	**52.5**	**69.0**
0.05	**94.9**	0.0	**52.5**	**77.0**	**86.5**
0.1	**84.1**	47.5	0.0	**85.2**	**95.2**
0.2	47.5	22.9	14.8	0.0	**87.0**
0.3	31.0	13.5	4.8	13.0	0.0

Table 7 Percentage of times the average rank of XGBoost improves when changing the `gamma` from the value in the top row to the value in the first column

%	0.0	0.1	0.2	0.3	0.4	1.0	1.5	2.0
0	0.0	**51.7**	40.0	43.1	45.4	**69.2**	**85.2**	**93.8**
0.1	48.3	0.0	37.5	43.1	45.0	**66.9**	**83.8**	**93.3**
0.2	**60.0**	**62.5**	0.0	**51.0**	**57.9**	**76.3**	**90.2**	**96.7**
0.3	**56.9**	**56.9**	49.0	0.0	49.6	**73.5**	**91.5**	**96.7**
0.4	**54.6**	**55.0**	42.1	**50.4**	0.0	**74.8**	**91.5**	**95.8**
1	30.8	33.1	23.8	26.5	25.2	0.0	**80.8**	**94.2**
1.5	14.8	16.3	9.8	8.5	8.5	19.2	0.0	**82.7**
2	6.3	6.7	3.3	3.3	4.2	5.8	17.3	0.0

Table 8 Percentage of times the average rank of XGBoost improves when changing the `depth` from the value in the top row to the value in the first column

%	2	3	5	7	10	100
2	0.0	12.8	9.8	7.2	8.1	8.9
3	**87.2**	0.0	28.1	25.9	23.9	25.0
5	**90.2**	**71.9**	0.0	38.1	34.2	33.3
7	**92.8**	**74.1**	**61.9**	0.0	45.6	44.2
10	**91.9**	**76.1**	**65.8**	**54.4**	0.0	47.0
100	**91.1**	**75.0**	**66.7**	**55.8**	**53.0**	0.0

Table 9 Percentage of times the average rank of XGBoost improves when changing the `colsample_bylevel` from the value in the top row to the value in the first column

%	0.25	sqrt	log2	1.0
0.25	0.0	44.4	44.9	**67.2**
sqrt	**55.6**	0.0	**52.0**	**72.3**
log2	**55.1**	48.0	0.0	**70.9**
1	32.8	27.7	29.1	0.0

Table 10 Percentage of times the average rank of XGBoost improves when changing the `subsample` from the value in the top row to the value in the first column

%	0.25	0.5	0.75	1.0
0.25	0.0	5.3	7.8	22.4
0.5	**94.7**	0.0	30.5	47.7
0.75	**92.2**	**69.5**	0.0	**75.4**
1	**77.6**	**52.3**	24.6	0.0

the table. Values above 50% are highlighted in bold. For instance, as shown by Table 6, 94.9% of the times that the learning rate hyper-parameter goes from 0.025 to 0.05, while keeping the rest of the hyper-parameters fixed, the average rank across all the analyzed datasets improves.

From these tables, we can see what the most favorable hyper-parameter values in general are. In Table 6, it can be observed that the best values for the learning rate are intermediate values. Both 0.05 and 0.1 clearly improve the performance of XGBoost with respect

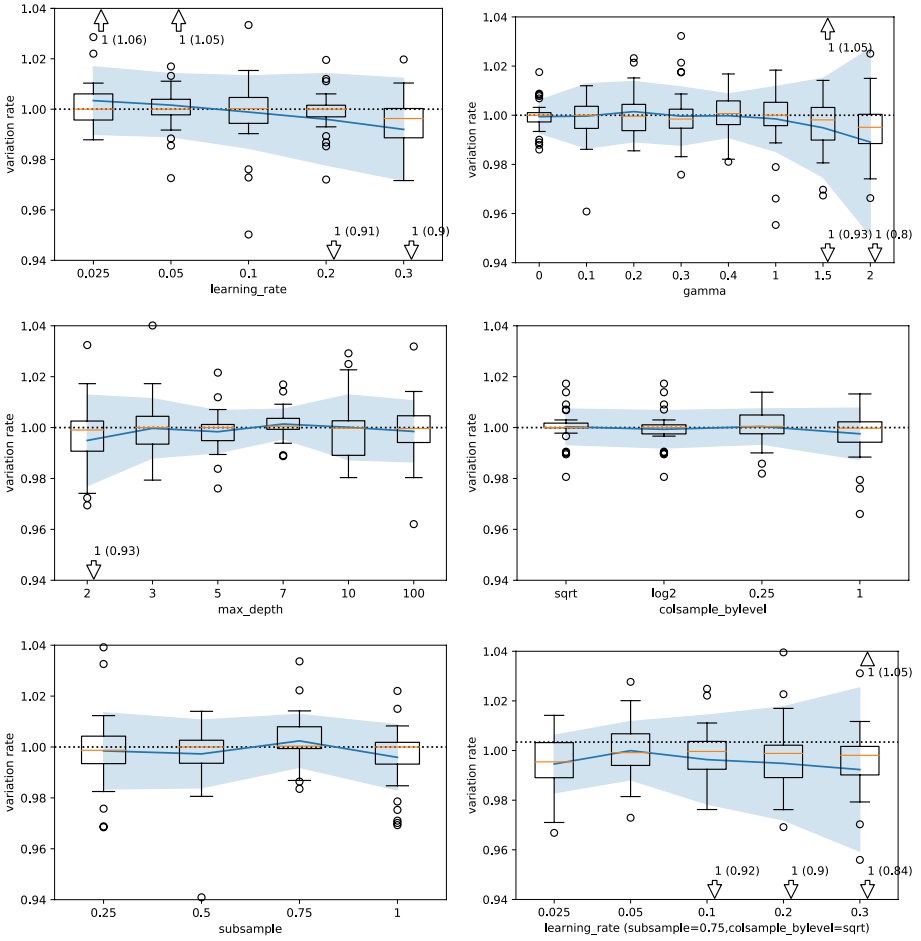

Fig. 2 Hyper-parametrization analysis of XGBoost. Each boxplot represents the performance ratio variation of the tested dataset if the grid search for the given hyper-parameter was to be held fixed at the given value

to the rest of the hyper-parameters on average. The best values for `gamma` (see Table 7) are also in the mid-range of the analyzed values. In this table, we can observe that the default gamma value, which is 0, is definitely not the best choice in general. A value of `gamma` $\in [0.2, 0.3]$ seems to be a reasonable choice. An interesting aspect related to the tree depth values, as shown in Table 8, is that the higher the depth, the better the performance on average. This is not necessarily in contradiction with the common use of shallow trees in gradient boosting algorithms since the depth hyper-parameter value is simply a maximum. Furthermore, the actual depth of the trees is also selected through the gamma hyper-parameter (that controls the complexity of the trees). Regarding the percentage of selected features when building the tree, it can be observed in Table 9 that values 0.25, sqrt and log2 perform very similarly on average. As for subsampling, the best value is 0.75 (see Table 10). In summary, we propose to use as the default XGBoost hyper-parameters: 0.05, 0.2, 100 (unlimited), sqrt and 0.75 for learning rate, gamma, depth, features and subsampling rate respectively.

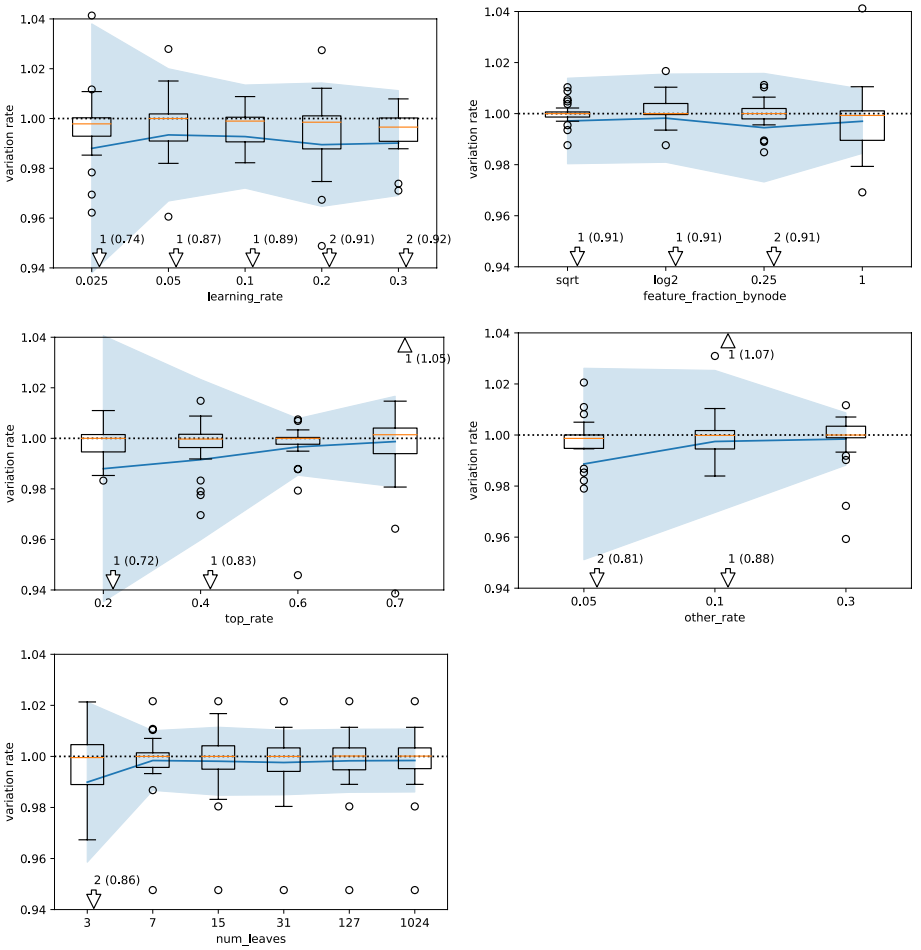

Fig. 3 Hyper-parametrization analysis of GOSS LightGBM. Each boxplot represents the performance ratio variation of the tested dataset if the grid search for the given hyper-parameter was to be held fixed at the given value

The second hyper-parametrization analysis consists in analyzing how the performance of the models vary if a given hyper-parameter *HParamX* is not optimized and its value is fixed to *HParamX=HParamX_valueA*. We propose a tool for visualizing the effect for a given set of tested datasets. To analyze this, we use the fact that we recorded for each train-test partition and for all possible grid combinations the within-train cross-validation error and the generalization test error. Hence, we can estimate the generalization error for any sub-grid within the tested hyper-parameter grid. The results of this experiments are shown in Fig. 2. This figure shows a plot for each tested hyper-parameter in XGBoost. For each plot, a boxplot is shown for each possible value assignment that indicates how the test performance ratio of the model on the tested datasets changes with respect to the use of the full grid (that is, with respect to the performance shown in Table 3 for tuned XGBoost), when the given hyper-parameter is not optimized and its value is set to the value shown in the x-axis. In addition to the boxplot, the mean ratio variation and deviation are shown with

a blue line and a blue shadowed area respectively. The reference performance for the full grid is shown with a dotted line. Note that the plots are all set to the range [0.94, 1.04] for easier comparison and that a few outlier points (datasets) may fall outside the plots. If this is the case an annotation arrow is shown with the number of outliers and their minimum value (for below limit points) or maximum value (for above limits points) between parenthesis. One would expect, for important hyper-parameter at least, that the performance would drop below 1.0 for most datasets. However, this is not always the case since not tuning one hyper-parameter can be compensated by setting another hyper-parameter to a different value. To analyze this, we show, in the last plot of Fig. 2, how the results change with respect to the full grid for learning_rate given that subsample is already fixed to 0.75 and colsample_bylevel to 'sqrt'. In this plot the dotted line corresponds to the reference average ratio when sumsample = 0.75 and colsample_bylevel = 'sqrt'.

These pictures provide a visual tool for analyzing the effect of the different hyper-parameters. From these pictures, we can extract conclusions similar to the previous analysis although they provide further insights about the importance of the different hyper-parameters as they also show the variability of the performance across datasets. From these plots we can observe that fixing a hyper-parameter to a certain value does not in general produce severe drops in performance with some exceptions for extreme values. For instance, if setting *gamma* = 2, for $\approx 3/4$ of datasets the performance drops. In some cases, as for *subsample* = 0.75 a slight improvement in performance can be observed. In addition, optimizing column subsampling does not seem to provide big gains and fixing its value either to 'sqrt' or 'log2' produces results similar to those of the full grid. Hence, fixing those hyper-parameters instead of optimizing them may seem a reasonable idea. In order to further analyze this idea, in the last plot of Fig. 2 the variation of learning_rate is shown given subsample = 0.75 and colsample_bylevel = 'sqrt'. From this plot, the importance of optimizing the learning rate becomes clearer: setting the learning rate to any of the studied values would produce a drop in performance in more than 75% of the datasets (except for 0.05 in which case the drop is in 19 out of 28 datasets, 68%).

The plots for the hyper-parametrization of GOSS LightGBM and Ordered CatBoost are shown in Figs. 3 and 4 respectively. From these plots, it can be observed that for both GOSS LightGBM and Ordered CatBoost the most important hyper-parameter is the learning rate since fixing it to any of the studied values would result in a drop in the overall performance. This is especially evident for GOSS LightGBM. In addition, for this method, there is a tendency to prefer higher values for hyper-parameter top_rate and other_rate. For these hyper-parameters the default values (top_rate = 0.1 and other_rate = 0.2) are suboptimal in the analyzed datasets. Ordered CatBoost seems to be quite stable independently of the hyper-parametrization, except for the learning_rate. Note that for l2_leaf_reg the best value for the studied datasets seems to be 1 instead of 3 as proposed in the default settings.

3.3 Selected hyper-parametrizations

In Table 11, we show the results for a selection of subgrids deduced from the previous hyper-parametrization analysis for the different libraries. First, Table 11 shows the average error for the proposed default hyper-parameters for XGBoost in column "D.P. XGB". For reference, the default hyper-parametrization is also shown in the first column (as D. XGB). In addition, we explore two different hyper-parameter grids for XGBoost. On one hand, we would like to analyze the differences between gradient boosting and

Table 11 Average accuracy and standard deviation of XGBoost with different configurations: default, proposed, tuned, no gamma tuning and no randomizations hyper-parameter tuning

Dataset	D. XGB	DP. XGB	T. XGB	T. NR XGB	T. NG XGB	T. LGB goss	T. NR LGB goss	T. Cat ord	T. Red Cat ord
Australian	86.94% ± 2.73	86.95% ± 3.41	87.53% ± 3.47	87.54% ± 3.00	86.95% ± 3.62	86.81% ± 3.34	86.08% ± 4.13	86.96% ± 3.09	**87.84% ± 3.36**
Banknote	99.64% ± 0.59	99.49% ± 0.57	99.64% ± 0.49	99.56% ± 0.58	99.71% ± 0.48	99.71% ± 0.48	99.71% ± 0.48	99.71% ± 0.48	**99.93% ± 0.22**
Breast	96.28% ± 1.15	96.13% ± 1.29	95.99% ± 1.68	**96.99% ± 1.50**	95.99% ± 1.55	96.57% ± 1.95	96.71% ± 2.13	**96.99% ± 2.08**	96.85% ± 2.30
Cleveland	81.14% ± 7.25	81.16% ± 5.26	83.16% ± 7.94	82.12% ± 6.69	83.09% ± 8.53	**84.50% ± 5.06**	82.15% ± 5.82	82.80% ± 5.16	83.45% ± 5.85
Dermatology	96.74% ± 3.21	97.83% ± 3.42	97.27% ± 3.24	98.39% ± 3.50	98.08% ± 3.29	96.78% ± 2.24	96.48% ± 2.97	98.08% ± 1.77	**98.64% ± 1.84**
Diabetes	75.65% ± 5.11	75.25% ± 3.95	76.56% ± 4.50	76.17% ± 4.43	77.08% ± 4.51	75.79% ± 2.82	**77.22% ± 2.60**	75.78% ± 2.80	76.82% ± 2.85
Echo	94.46% ± 6.80	**98.75% ± 3.75**	**98.75% ± 3.75**	**98.75% ± 3.75**	**98.75% ± 3.75**	97.14% ± 5.71	98.57% ± 4.29	98.57% ± 4.29	98.57% ± 4.29
Ecoli	86.87% ± 5.31	86.91% ± 6.38	89.05% ± 4.12	87.50% ± 5.88	87.81% ± 5.48	86.14% ± 6.92	86.76% ± 5.61	88.90% ± 6.27	**89.77% ± 6.26**
German	**79.00% ± 4.22**	76.00% ± 3.97	77.40% ± 4.13	77.30% ± 4.22	77.20% ± 3.28	77.20% ± 4.64	77.30% ± 4.29	76.20% ± 4.69	78.00% ± 3.85
Heart	79.26% ± 5.29	80.74% ± 6.58	84.07% ± 5.98	84.44% ± 6.15	**84.81% ± 4.81**	84.07% ± 4.07	84.07% ± 4.40	84.44% ± 5.69	84.44% ± 4.91
Hepatitis	59.21% ± 8.28	59.92% ± 7.37	67.00% ± 6.56	**67.50% ± 11.82**	66.92% ± 11.04	64.46% ± 10.96	62.46% ± 6.18	65.79% ± 7.58	63.21% ± 9.19
Ionosphere	92.56% ± 2.69	93.15% ± 3.26	92.59% ± 3.17	92.87% ± 3.43	91.99% ± 3.37	**93.21% ± 4.55**	93.16% ± 4.63	92.90% ± 4.73	**93.21% ± 5.46**
Iris	92.67% ± 6.29	94.67% ± 4.99	94.00% ± 4.67	**95.33% ± 4.27**	94.00% ± 5.54	94.67% ± 4.99	**95.33% ± 4.27**	94.67% ± 4.99	93.33% ± 5.16
Liver	68.65% ± 4.69	68.95% ± 6.00	68.11% ± 6.12	69.97% ± 6.13	69.31% ± 5.60	69.24% ± 5.98	68.75% ± 6.02	69.47% ± 3.51	**71.86% ± 3.47**
Magic04	87.47% ± 0.57	88.58% ± 0.51	88.63% ± 0.48	88.62% ± 0.31	88.46% ± 0.47	88.42% ± 0.44	88.41% ± 0.62	**88.76% ± 0.48**	88.52% ± 0.54
New-thyroid	95.80% ± 3.26	95.35% ± 3.58	95.80% ± 3.26	95.30% ± 3.01	94.85% ± 3.29	94.83% ± 4.50	94.85% ± 3.35	96.71% ± 3.03	**96.73% ± 2.14**
Parkinsons	92.14% ± 5.72	90.64% ± 4.82	90.14% ± 4.01	90.70% ± 3.37	90.14% ± 4.72	92.64% ± 5.16	**93.72% ± 4.82**	92.22% ± 5.88	93.33% ± 5.57
Phishing	89.65% ± 2.44	88.85% ± 2.78	**91.13% ± 1.30**	90.69% ± 1.99	90.03% ± 2.22	90.98% ± 2.45	90.46% ± 2.37	90.68% ± 1.96	90.75% ± 2.44
Segment	98.48% ± 0.70	98.57% ± 0.87	**98.70% ± 0.77**	98.66% ± 0.81	98.66% ± 0.79	98.53% ± 0.62	98.61% ± 0.67	98.61% ± 0.61	98.18% ± 0.69
Sonar	85.59% ± 6.44	**88.38% ± 8.85**	86.97% ± 4.84	87.50% ± 6.29	87.02% ± 7.45	86.97% ± 5.03	87.02% ± 5.73	83.61% ± 6.25	84.56% ± 6.52
Soybean	94.80% ± 3.35	94.67% ± 3.40	95.22% ± 2.96	94.65% ± 3.05	**95.53% ± 3.12**	93.78% ± 2.40	94.39% ± 2.45	94.82% ± 2.70	95.13% ± 2.83
Spambase	95.17% ± 1.29	95.76% ± 0.85	95.57% ± 1.28	95.65% ± 1.20	95.76% ± 1.27	95.65% ± 0.93	**95.81% ± 0.73**	95.70% ± 0.59	95.78% ± 0.92
Teaching	63.55% ± 8.00	66.21% ± 10.06	64.26% ± 14.15	**66.84% ± 10.86**	63.55% ± 11.86	61.68% ± 8.37	63.56% ± 9.02	63.77% ± 7.59	66.32% ± 5.83
Tic-tac-toe	96.56% ± 1.63	99.89% ± 0.32	**100.00% ± 0.00**	**100.00% ± 0.00**	**100.00% ± 0.00**	**100.00% ± 0.00**	**100.00% ± 0.00**	**100.00% ± 0.00**	**100.00% ± 0.00**
Vehicle	78.74% ± 3.29	77.20% ± 2.75	77.18% ± 2.31	**79.21% ± 2.80**	77.79% ± 2.83	78.85% ± 2.39	77.54% ± 2.37	77.54% ± 3.36	78.95% ± 2.82
Vowel	92.63% ± 3.53	94.65% ± 2.39	95.86% ± 2.14	94.95% ± 2.12	95.56% ± 2.31	95.86% ± 2.91	94.95% ± 3.81	96.87% ± 2.14	**97.17% ± 2.20**

Table 11 (continued)

Dataset	D. XGB	DP. XGB	T. XGB	T. NR XGB	T. NG XGB	T. LGB goss	T. NR LGB goss	T. Cat ord	T. Red Cat ord
Waveform	85.72% ± 1.05	85.38% ± 1.21	85.72% ± 1.77	85.78% ± 1.23	85.56% ± 1.80	**85.84% ± 1.72**	85.78% ± 1.71	85.74% ± 1.75	85.24% ± 1.65
Wine	97.18% ± 2.83	98.26% ± 2.66	**98.82% ± 2.37**	**98.82% ± 2.37**	**98.82% ± 2.37**	97.18% ± 3.76	97.74% ± 2.78	97.21% ± 3.66	97.74% ± 2.78

XGBoost in more details. There are little differences between both algorithms except that XGBoost is optimized for speed. The main difference, from a machine learning point of view, is that XGBoost incorporates into the loss function a hyper-parameter to explicitly control the complexity of the decision trees (i.e. gamma). In order to analyze whether this hyper-parameter provides any advantage in the classification performance of XGBoost, a grid with the same hyper-parameter values as the ones given by Table 2 is used except for gamma, which is always set to 0.0. On the other hand, we have observed that, in the case of random forest, tuning the randomization hyper-parameters is not very productive in general. As shown in Fig. 1, the average rank of default random forest is better than the rank of the forests for which the randomization hyper-parameters are tuned. Hence, the second proposed grid is to tune the optimization hyper-parameters of XGBoost (i.e. learning rate, gamma and depth) keeping the randomization hyper-parameters (i.e. random features and subsampling) fixed. The randomization hyper-parameters will be fixed to 0.75 for the subsampling ratio and to sqrt for the number of features as suggested by Tables 9 and 10 respectively and Fig. 2. The average generalization errors for XGBoost when using these two grids are shown in Table 11. Column "T. NG XGB" shows the results for the grid that does not tune gamma (i.e. gamma=0), and column "T. NR XGB" shows the results for the grid that does not tune the randomization hyper-parameters. For LightGBM, as observed from Fig. 3 for the studied datasets, the randomization hyper-parameters top_rate and other_rate seem to perform better for higher values, 0.7 and 0.3 respectively. Hence, a grid fixing those hyper-parameter is proposed (column "T. NR LGB" in the table). For CatBoost, we propose to fix l2_leaf_reg to 1 and leaf_estimation_iterations to 10 and to optimize the other hyper-parameters. The results of this reduced grid are shown in column "T. R Cat". The tested full grids for

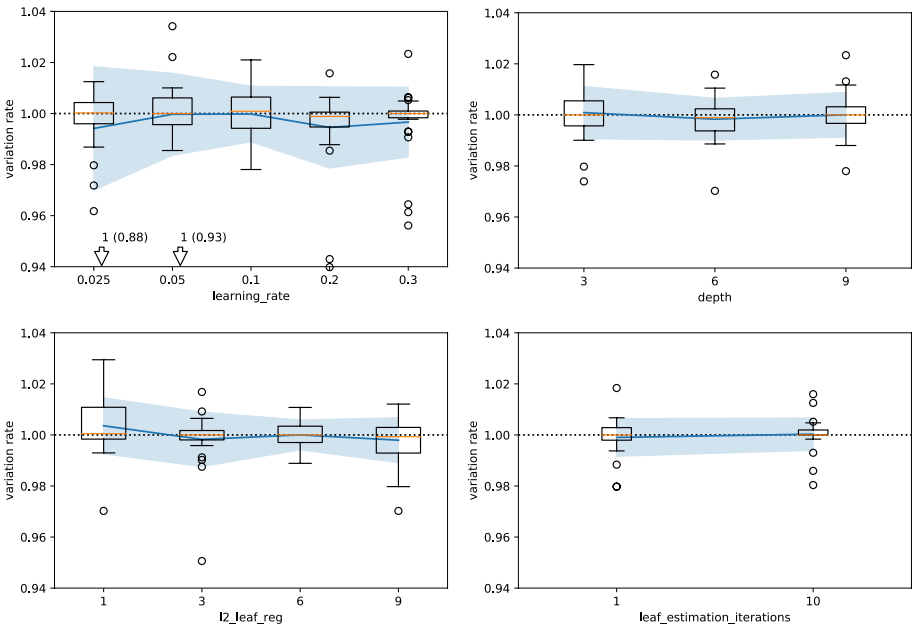

Fig. 4 Hyper-parametrization analysis of Ordered CatBoost. Each boxplot represents the performance ratio variation of the tested dataset if the grid search for the given hyper-parameter was to be held fixed at the given value

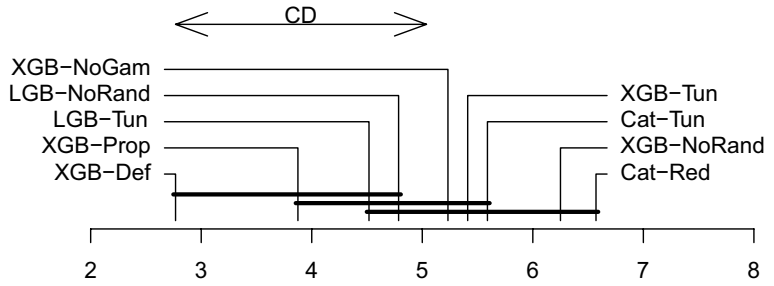

Fig. 5 Average ranks (higher rank is better) for different XGBoost configurations (Critical difference CD= 1.15)

XGBoost, GOSS LightGBM, and Ordered CatBoost are also shown in Table 11 for reference. The best average test error for each dataset is highlighted in boldface. Additionally, the average ranks for this table are shown in Fig. 5 following the methodology proposed in Demšar (2006). In this figure, differences among methods connected with a horizontal solid line are not statistically significant.

As it can be observed from Table 11, the proposed fixed hyper-parametrization for XGBoost (D.P. XGB) improves over the results of the default setting (D. XGB). With the proposed default setting, better results can be obtained in 17 out of 28 datasets with notable differences in datasets as *Echo*, *Sonar* or *Tic-tac-toe*. In addition, when the default setting is better than the proposed hyper-parameters, the differences are in general small, except for *German*, *Parkinson* and *Vehicle*. In spite of the improvements on average achieved by the proposed hyper-parameter setting, it seems clear that hyper-parameter optimization is necessary to further improve the performance of XGBoost and to adapt the model to the characteristics of each specific dataset.

The results shown in Table 11 and Fig. 5 are quite interesting. The number of best results are 9 and 8 for Ordered CatBoost with the reduced grid and for XGBoost for the grid without randomization hyper-parameter tuning. Both grids improve over their respective full tested grid search of Ordered CatBoost and XGBoost. For GOSS LightGBM the variations in performance are small between the two grids. The same occurs for XGBoost between the full grid and the grid in which gamma is not optimized. Similar observations can be made from the average ranks for the different tested grids shown in Fig. 5. From this plot the reduced grid for Ordered CatBoost obtains the best average rank closely followed by XGBoost with the grid without randomization hyper-parameter tuning. Then, the full grid version of these methods, no gamma optimization and LightGBM grids follow in the plot. One tendency that is observed from these results is that it seems that including a complexity term to control the size of the trees can have a small edge over not using it in XGBoost, although the differences are not statistically significant. A conclusion that may be clearer is that it seems unnecessary to tune the number of random features and the subsampling rate provided that those techniques are applied with reasonable values (in our case subsampling to 0.75 and feature sampling to sqrt).

Finally, the time required to perform the grid search and to train the single models is shown in Table 12 in the same manner as Table 5. As shown in the last row of this table for the tested settings, performing the grid search without tuning the randomization hyper-parameters for XGBoost and LightGBM is approximately 16 and 10 times faster the tuning the full grid on average. These results reinforce the idea that tuning the randomization hyper-parameter is unnecessary. For Ordered CatBoost, there is a

Table 12 Average execution time (in seconds) for training different variant of XGBoost, LightGBM GOSS and CatBoost Ordered (more details in the text)

Dataset	D. XGB	DP. XGB	T. XGB	T. NR XGB	T. NG XGB	T. LGB goss	T. NR LGB goss	T. Cat ord	T. Red Cat ord
Australian	0.10	0.09	2775 + 0.08	149 + 0.08	340 + 0.11	1181 + 0.05	124 + 0.10	8250 + 6.23	1313 + 10.36
Banknote	0.10	0.09	2874 + 0.06	377 + 0.06	340 + 0.06	1683 + 0.11	162 + 0.11	11771 + 3.34	1838 + 5.32
Breast	0.06	0.05	1733 + 0.04	108 + 0.05	212 + 0.04	780 + 0.05	72 + 0.06	3723 + 2.42	746 + 3.54
Cleveland	0.05	0.04	1357 + 0.02	78 + 0.03	165 + 0.02	581 + 0.03	60 + 0.03	4941 + 2.75	853 + 3.28
Dermatology	0.46	0.26	11014 + 0.27	572 + 0.27	1281 + 0.27	3016 + 0.21	332 + 0.27	14315 + 14.11	2498 + 10.13
Diabetes	0.08	0.10	2982 + 0.06	163 + 0.08	365 + 0.05	1246 + 0.08	131 + 0.07	9978 + 12.51	1637 + 9.85
Echo	0.02	0.01	532 + 0.01	33 + 0.01	65 + 0.01	232 + 0.01	25 + 0.02	1896 + 1.15	311 + 1.35
Ecoli	0.17	0.14	4518 + 0.11	284 + 0.13	538 + 0.11	2001 + 0.12	232 + 0.14	17992 + 14.28	2749 + 8.68
German	0.17	0.17	5453 + 0.16	251 + 0.13	670 + 0.19	1398 + 0.09	141 + 0.08	7837 + 6.94	1425 + 5.42
Heart	0.05	0.05	1268 + 0.03	74 + 0.03	155 + 0.02	537 + 0.03	55 + 0.03	4692 + 2.39	812 + 2.89
Hepatitis	0.04	0.03	1071 + 0.02	61 + 0.02	131 + 0.02	371 + 0.02	39 + 0.02	3332 + 2.93	533 + 5.14
Ionosphere	0.14	0.05	2487 + 0.05	112 + 0.04	289 + 0.04	1489 + 0.09	161 + 0.11	23103 + 11.95	2598 + 7.82
Iris	0.04	0.04	1308 + 0.03	171 + 0.04	154 + 0.03	633 + 0.05	65 + 0.05	3622 + 2.67	627 + 1.42
Liver	0.07	0.08	2308 + 0.05	129 + 0.05	283 + 0.06	947 + 0.05	101 + 0.07	12397 + 6.68	1976 + 6.87
Magic04	3.32	7.60	123764 + 7.86	6618 + 7.38	14974 + 8.11	27085 + 2.02	2638 + 1.71	39787 + 64.67	6415 + 79.63
New-thyroid	0.05	0.05	1728 + 0.04	112 + 0.05	198 + 0.04	914 + 0.06	91 + 0.06	8524 + 9.50	1306 + 6.08
Parkinsons	0.05	0.03	1169 + 0.03	64 + 0.02	138 + 0.03	571 + 0.04	63 + 0.04	26291 + 11.25	3525 + 16.79
Phishing	0.36	0.44	12464 + 0.75	746 + 0.40	1551 + 0.53	3990 + 0.19	408 + 0.18	4895 + 5.30	1032 + 7.77
Segment	3.22	1.88	73309 + 2.21	3878 + 1.48	8447 + 2.25	39974 + 2.60	4149 + 2.94	82075 + 128.78	10269 + 136.14
Sonar	0.15	0.05	2442 + 0.05	103 + 0.04	280 + 0.06	1136 + 0.11	133 + 0.12	34174 + 8.85	3838 + 19.21
Soybean	3.73	2.67	104873 + 2.69	5892 + 2.65	12730 + 3.49	16133 + 1.07	1712 + 1.07	61748 + 41.22	12295 + 101.14
Spambase	1.83	1.32	47412 + 1.46	1621 + 1.20	5788 + 1.54	19179 + 2.78	1994 + 3.88	32840 + 59.34	4210 + 49.01
Teaching	0.05	0.07	1879 + 0.06	124 + 0.06	228 + 0.05	651 + 0.05	71 + 0.05	3217 + 2.94	564 + 2.38
Tic-tac-toe	0.08	0.09	2842 + 0.08	171 + 0.08	347 + 0.09	965 + 0.07	96 + 0.08	2951 + 2.57	542 + 2.84
Vehicle	0.60	0.54	17217 + 0.38	899 + 0.28	1993 + 0.35	6425 + 0.34	698 + 0.37	33313 + 34.09	4604 + 24.17

Table 12 (continued)

Dataset	D. XGB	DP. XGB	T. XGB	T. NR XGB	T. NG XGB	T. LGB goss	T. NR LGB goss	T. Cat ord	T. Red Cat ord
Vowel	1.96	1.41	48726 + 1.24	2887 + 1.40	5577 + 1.24	30567 + 2.25	3183 + 2.10	83076 + 137.04	10685 + 125.93
Waveform	3.34	3.83	111562 + 1.74	5028 + 1.73	13073 + 2.15	55333 + 1.10	5792 + 1.98	59940 + 52.92	6622 + 55.10
Wine	0.07	0.05	2104 + 0.05	122 + 0.05	231 + 0.05	887 + 0.05	94 + 0.08	26097 + 22.14	3483 + 11.19
Ave. ratio	1.00	0.90	29043.7 + 0.79	1782.3 + 0.77	3472.0 + 0.80	11922.6 + 0.76	1244.3 + 0.88	106234.9 + 73.9	15676.1 + 73.1

reduction of approximately 7 times with respect to the full grid although the grid search time remains rather high.

4 Conclusion

In this study we present an empirical analysis of recent variants of gradient boosting: XGBoost, LightGBM and CatBoost, that have proven to be efficient and accurate models. Specifically, the performance of XGBoost, LightGBM and CatBoost in terms of training speed, AUC and accuracy is compared with the performance of gradient boosting and random forest under a wide range of classification tasks. The study focuses on specific characteristics of these methods with respect to gradient boosting: introduction of a complexity term in XGBoost, selective subsampling of high gradient instances in LightGBM and a computation of gradients that avoids prediction shift in CatBoost. In addition, the hyper-parameter tuning process of these methods is thoroughly analyzed.

The results of this study show that the most accurate classifier, in terms of average rank is CatBoost. Nevertheless, the differences with respect to XGBoost, gradient boosting, random forest (using the default hyper-parameters) and LightGBM are not statistically significant in terms of average ranks. XGBoost and LightGBM trained using the default hyper-parameters of the packages were the least successful methods (statistically worse than the tuned versions of XGBoost and CatBoost). The difference between the default settings of CatBoost and the tuned version are small. This is because the default version of CatBoost performs a internal adjustments of the learning rate. The performance of the different methods in terms of AUC is also consistent with their performance in terms of accuracy. In consequence, we conclude that a meticulous hyper-parameter search is necessary to create accurate models based on gradient boosting. This is not the case for random forest, whose generalization performance was slightly better on average when the default hyper-parameter values were used (those originally proposed by Breiman). In fact, tuning in XGBoost the randomization hyper-parameters subsampling rate and the number of features selected at each split was found to be unnecessary as long as some randomization is used. In our experiments, we fixed the values of the subsampling rate to 0.75 without replacement and the number of features to sqrt, reducing the size of the hyper-parameter grid search 16 fold and improving the average performance of XGBoost. For LightGBM, we found that, for the studied datasets, tuning the top and lower rate hyper-parameters produced results very similar to not tuning them. For CatBoost, we found that tuning the learning rate and the depth of the trees was sufficient to get the best results of the analysis.

Finally, from the experiments of this study, which are based on grid search hyper-parameter tuning using within-train 10-fold cross-validation, the tuning phase contributed to over 99.9% of the computational effort necessary to train any of the methods. However, the grid search time can be significantly reduced when the smaller proposed grids are used for XGBoost, LightGBM and CatBoost. The fastest algorithm was LightGBM. XGBoost has also very competitive speed results. Finally, Ordered CatBoost is much slower than the other methods in general. Anyhow, the time for selecting the model hyper-parameters has to be taken into consideration when choosing any given method and not only its speed for training the final model. In this sense, the default versions of random forest and Ordered CatBoost provide competitive results without the computational burden of hyper-parameter tuning.

Acknowledgement The authors acknowledge financial support from the European Regional Development Fund and from the Spanish Ministry of Economy, Industry, and Competitiveness-State Research Agency, project TIN2016-76406-P (AEI/FEDER, UE) and project PID2019-106827GB-I00 / AEI / 10.13039/501100011033. The authors thank the Centro de Computación Científica (CCC) at Universidad Autónoma de Madrid (UAM) for the use of their facilities.

References

Babajide Mustapha I, Saeed F (2016) Bioactive molecule prediction using extreme gradient boosting. Molecules 21(8):983

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984) Classification and regression trees. Chapman & Hall, New York

Brown I, Mues C (2012) An experimental comparison of classification algorithms for imbalanced credit scoring data sets. Expert Syst Appl 39(3):3446–3453

Caruana R, Niculescu-Mizil A (2006) An empirical comparison of supervised learning algorithms. In: Proceedings of the 23rd international conference on machine learning, ICML'06. ACM Press, New York, pp 161–168

Chen T, Guestrin C (2016) Xgboost: a scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, KDD'16. ACM, New York, pp 785–794

Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30

Dietterich TG (2000) An experimental comparison of three methods for constructing ensembles of decision trees: bagging, boosting, and randomization. Maxh Learn 40(2):139–157

Dwork C, Feldman V, Hardt M, Pitassi T, Reingold O, Roth A (2015) Generalization in adaptive data analysis and holdout reuse. Adv Neural Inf Process Syst 28:2350–2358

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

Friedman JH (2001) Greedy function approximation: a gradient boosting machine. Ann Stat 29(5):1189–1232

Friedman JH (2002) Stochastic gradient boosting. Comput Stat Data Anal 38(4):367–378 Nonlinear Methods and Data Mining

Gumus M, Kiran MS (2017) Crude oil price forecasting using xgboost. In: 2017 International conference on computer science and engineering (UBMK), pp 1100–1103

Ke G, Meng Q, Finley T, Wang T, Chen W, Ma W, Ye Q, Liu TY (2017) Lightgbm: a highly efficient gradient boosting decision tree. In: Guyon I, Luxburg UV, Bengio S, Wallach H, Fergus R, Vishwanathan S, Garnett R (eds) Advances in neural information processing systems, vol 30, pp 3146–3154

Khramtsov V, Sergeyev A, Spiniello C, Tortora C, Napolitano N, Agnello A, Getman F, De Jong J, Kuijken K, Radovich M, Shan H, Shulga V (2019) KiDS-SQuaD: II machine learning selection of bright extragalactic objects to search for new gravitationally lensed quasars. Astron Astrophys 632:A56

Lichman M (2013) UCI machine learning repository. http://archive.ics.uci.edu/ml

Mirabal N, Charles E, Ferrara EC, Gonthier PL, Harding AK, Sánchez-Conde MA, Thompson DJ (2016) 3FGL demographics outside the galactic plane using supervised machine learning: pulsar and dark matter subhalo interpretations. Astrophys J 825(1):69

Nori V, Hane C, Crown W, Au R, Burke W, Sanghavi D, Bleicher P (2019) Machine learning models to predict onset of dementia: a label learning approach. Alzheimer's Dementia Transl Res Clin Interven 5:918–925

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in python. J Mach Learn Res 12:2825–2830

Prokhorenkova L, Gusev G, Vorobev A, Dorogush AV, Gulin A (2018) Catboost: unbiased boosting with categorical features. In: Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R (eds) Advances in neural information processing systems, vol 31, pp 6638–6648

Rokach L (2016) Decision forest: twenty years of research. Inf Fusion 27:111–125

Torres-Barrán A, Alonso A, Dorronsoro JR (2017) Regression tree ensembles for wind energy and solar radiation prediction. Neurocomputing. https://doi.org/10.1016/j.neucom.2017.05.104

Valdivia A, Luzón MV, Cambria E, Herrera F (2018) Consensus vote models for detecting and filtering neutrality in sentiment analysis. Inf Fusion 44:126–135

Xia Y, Liu C, Li Y, Liu N (2017) A boosted decision tree approach using bayesian hyper-parameter optimization for credit scoring. Expert Syst Appl 78:225–241

Yoav Freund RES (1999) A short introduction to boosting. J Jpn Soc Artif Intell 14(5):771–780

Zhang C, Liu C, Zhang X, Almpanidis G (2017) An up-to-date comparison of state-of-the-art classification algorithms. Expert Syst Appl 82:128–150