

A Comparative Analysis of XGBoost

Candice Bentéjac^a Anna Csörgő^b
Gonzalo Martínez-Muñoz^c

^aCollege of Science and Technology, University of Bordeaux, France

^bPázmány Péter Catholic University, Faculty of Information
Technology and Bionics, Hungary

^cEscuela Politécnica Superior, Universidad Autónoma de Madrid,
Spain

Abstract

XGBoost is a scalable ensemble technique based on gradient boosting that has demonstrated to be a reliable and efficient machine learning challenge solver. This work proposes a practical analysis of how this novel technique works in terms of training speed, generalization performance and parameter setup. In addition, a comprehensive comparison between XGBoost, random forests and gradient boosting has been performed using carefully tuned models as well as using the default settings. The results of this comparison may indicate that XGBoost is not necessarily the best choice under all circumstances. Finally an extensive analysis of XGBoost parametrization tuning process is carried out.

keywords: XGBoost, gradient boosting, random forest, ensembles of classifiers

1 Introduction

As machine learning is becoming a critical part of the success of more and more applications — such as credit scoring [18], bioactive molecule prediction [1], solar and wind energy prediction [16], oil price prediction [11], classification of galactic unidentified sources [13], sentiment analysis [17]— 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 [3, 9, 19, 5]. These methods mainly rely on randomization techniques, which consist in generating many diverse solutions to the problem at hand [3], or on adaptive emphasis procedures (e.g. boosting [19]).

In fact, the above mentioned applications have in common that they all use ensemble methods and, in particular, a recent ensemble method called eXtreme Gradient Boosting or XGBoost [5] with very competitive results. This method,

based on gradient boosting [9], has been consistently placing among the top contenders in Kaggle competitions [5]. But XGBoost is not the only one 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 [8, 4, 15].

This study follows the path of many other previous comparative analysis, such as [8, 4, 15], with the intent of covering a gap related to gradient boosting and its more recent variant XGBoost. None of the previous comprehensive analysis included any machine learning algorithm of the gradient boosting *family* despite of their appealing properties. The specific objectives of this study are, in the first place, to compare XGBoost's performance with respect to the algorithm on which it is based (i.e. gradient boosting). Secondly, the comparison is extended to random forest, which can be considered as a benchmark since many previous comparisons demonstrated its remarkable performance [8, 4]. The comparison is carried out in terms of accuracy and training speed. Finally, a comprehensive analysis of the process of parameter setting in XGBoost is performed. We believe this analysis can be very helpful to researchers of various fields to be able to tune XGBoost more effectively.

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

2 Methodology

2.1 Random forest

One of the most successful machine learning methods is random forest [3]. 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 parameter is $\lfloor \log_2(\#features)+1 \rfloor$. 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 [3]. The implication of this demonstration is that the size of the ensemble is not a 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* parameter-free or at least, the default parameter setting has a remarkable performance on average [8]. 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 parameter tuning.

Anyhow, other 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 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 parameter limits the size of the trees but, in the worst case, the depth of the trees can be as large as $N - \text{min_samples_split}$, 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 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 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 [19]. Gradient boosting is a boosting-like algorithm for regression [9]. 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

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

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

$$F_0(\mathbf{x}) = \underset{\alpha}{\operatorname{argmin}} \sum_{i=1}^N L(y_i, \alpha) . \quad (2)$$

Subsequent models are expected to minimize

$$(\rho_m, h_m(\mathbf{x})) = \underset{\rho, h}{\operatorname{argmin}} \sum_{i=1}^N L(y_i, F_{m-1}(\mathbf{x}_i) + \rho h(\mathbf{x}_i)) \quad (3)$$

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

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

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 [9]. 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 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 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 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 [10], 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 [10].

- 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 [5] 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

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

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

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 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.

Furthermore, randomization techniques are also implemented in XGBoost both to reduce overfitting and to increment training speed. The randomization techniques included in XGBoost are: random subsamples to train individual trees and column subsampling at tree and tree node levels.

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 [2].

Moreover, a sparsity-aware algorithm is used in XGBoost to effectively remove missing values from the computation of the loss gain of split candidates.

The following 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.

3 Experimental results

In this section, an extensive comparative analysis of the efficiency of random forest, gradient boosting and XGBoost models is carried out. For the experiments, 28 different datasets coming from the UCI repository [12] 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 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` package [14] was used for random forest and gradient boosting. For XGBoost, the `XGBoost` package ¹ was used. In addition, for the comparison, XGBoost, random forest and gradient boosting were analyzed tuning the parameters using a grid search as well as using the default parameters of the corresponding packages.

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 random forest, XGBoost and gradient boosting: (i) the optimum parameters for each method were estimated with stratified 10-fold cross-validation within the training set using a grid search. A wide range of parameter values is explored in the grid search. For each of the three methods, these values are shown in Table 2. (ii) The best set of parameters extracted from the grid search was used to train the corresponding ensemble using the whole training partition; (iii) Additionally, for XGBoost, an ensemble was trained on the whole training set for each possible combination of parameters given in Table 2. This allows us, in combination with step (i), to test for different grids (more details down below); (iv) The default sets of parameters for each method were additionally used to train an ensemble of each type (Table 2 shows the default values for each parameter and ensemble type); (v) The generalization accuracy of the three ensembles selected in (i) and of the three ensembles with

¹<https://github.com/dmlc/xgboost>

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	75	Yes	5
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

Table 1: Characteristics of the studied datasets

Random forest		
Parameter	Default value	Grid search values
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		
Parameter	Default value	Grid search values
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		
Parameter	Default value	Grid search values
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

Table 2: Default values and possible values for every parameter in the normal grid search for random forest, gradient boosting and XGBoost

default parametrization is estimated in the left out test set. (v) In addition, the accuracy of all the XGBoost ensembles trained in step (iii) is also computed using the test set.

All ensembles were composed of 200 decision trees. Note that the size of the ensemble is not a parameter that needs to be tuned in ensembles of classifiers [9, 3]. For random forest like ensembles, as more trees are combined into the ensemble the generalization error tends to an asymptotic value [3]. For gradient boosting like ensembles, the generalization performance can deteriorate with the number of elements in the ensemble especially for high learning rates values. However, this effect can not only be neutralized with the use of lower learning rates (or shrinkage) but reverted [9]. The conclusion of [9] 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.

3.1 Results

Table 3 displays the average accuracy and standard deviation (after the \pm sign) for: XGBoost with default parameters (shown with label D. XGB), tuned XGBoost (as T. XGB in the table), random forest with default parametrization (with D. RF), tuned random forest (T. RF), default gradient boosting (D. GB) and tuned gradient boosting (T. GB). The best accuracy for each dataset is highlighted using a yellow background.

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 10 out of 28 datasets). After that, the methods in order are: tuned XGBoost, that achieves the best results in 8 datasets; default gradient boosting in 5; tuned and default random forest in 4; and default XGBoost in 3. As it can be observed, the default performance of the three tested algorithms is quite different. Default random forest is the method that performs more evenly with respect to its tuned counterpart. Except for a few datasets, the differences in performance are very small in random forest. In fact, the difference between the tuned and the default parametrizations of random forest is below 0.5% in 18 out of 28 datasets. 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 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 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 [7]. On the other hand, in these datasets, the default setting is one of the 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 [6]. 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 Figure 1. In this plot, a higher rank indicates a better accuracy. **Statistical differences among average ranks are determined using a Nemenyi test.** There are no statistical significant differences in average ranks between methods connected with a horizontal solid line. The critical distance over which the differences are considered significant is shown in the plot for reference ($CD = 1.42$ for 6 methods, 28 datasets and $p\text{-value} < 0.05$).

From Figure 1, it can be observed that there are not any statistically significant differences among the average ranks of the six tested methods. The best method in terms of average rank is tuned XGBoost, followed by gradient boosting and random forest with default settings.

In Table 4, the average training execution time (in seconds) for the analyzed datasets is shown. For the methods using the default settings, the table shows

Dataset	D. XGB	T. XGB	D. RF	T. RF	D. GB	T. GB
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
New-thyroid	95.80%±3.26	95.80%±3.26	96.75%±2.94	96.28%±3.48	96.75%±2.94	96.28%±3.48
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
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
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
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
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
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
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
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
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
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
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
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

Table 3: Average accuracy and standard deviation for random forest, gradient boosting and XGBoost, both using default and tuned parameter settings

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 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. Notwithstanding, given that the grid search procedure is fully parallelizable, these optimizations 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 4, finding the best 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 and 1920 for XGB, RF and GB respectively), the time dedicated to finding the best parameters is not directly comparable between classifiers.

However, when it comes to fitting a single ensemble to the training data without taking into account the grid search time, XGBoost clearly shows the fastest performance on average. The time necessary to train XGBoost given a set of parameters is about 3.5 times faster than training a random forest and 2.4-4.3 times faster than training a gradient boosting model. This last difference can be observed in the time employed in the grid search by XGBoost and gradient boosting. XGBoost takes less than half of the time to look for the best 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* or *Soybean*, the execution time of XGBoost and gradient boosting deteriorates in relation to random forest.

3.2 Analysis of XGBoost parametrization

In order to further analyze and understand the parametrization of XGBoost, we carry out further experiments with two objectives. The first objective is to try to select a better default parametrization for XGBoost. The second is to explore alternative grids for XGBoost. For the first objective, we have analyzed, for each 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 parameter configurations as given in Table 2 for XGBoost. Recall that we have analyzed 3840 possible parameter

Dataset	D. XGB	T. XGB	D. RF	T. RF	D. GB	T. GB
Australian	0.10	2775 + 0.08	0.32	766 + 0.30	0.12	6053 + 0.61
Banknote	0.10	2874 + 0.06	0.40	991 + 0.39	0.23	5410 + 0.15
Breast	0.06	1733 + 0.04	0.28	744 + 0.28	0.16	4071 + 0.35
Cleveland	0.05	1357 + 0.02	0.28	700 + 0.26	0.10	3676 + 0.10
Dermatology	0.46	11014 + 0.27	0.27	727 + 0.27	0.76	17840 + 0.93
Diabetes	0.08	2982 + 0.06	0.35	798 + 0.34	0.13	6612 + 0.50
Echo	0.02	532 + 0.01	0.25	671 + 0.25	0.05	1608 + 0.08
Ecoli	0.17	4518 + 0.11	0.27	715 + 0.28	0.67	15448 + 1.25
German	0.17	5453 + 0.16	0.37	790 + 0.37	0.13	8589 + 0.48
Heart	0.05	1268 + 0.03	0.27	716 + 0.27	0.12	3579 + 0.10
Hepatitis	0.04	1071 + 0.02	0.26	686 + 0.26	0.11	2756 + 0.11
Ionosphere	0.14	2487 + 0.05	0.34	893 + 0.35	0.27	3982 + 0.22
Iris	0.04	1308 + 0.03	0.25	678 + 0.25	0.29	6555 + 0.39
Liver	0.07	2308 + 0.05	0.35	804 + 0.31	0.16	5559 + 0.28
Magic04	3.32	123764 + 7.86	11.39	11860 + 9.44	1.50	160831 + 48.07
New-thyroid	0.05	1728 + 0.04	0.26	686 + 0.26	0.30	7426 + 0.42
Parkinsons	0.05	1169 + 0.03	0.27	697 + 0.28	0.10	2539 + 0.11
Phishing	0.36	12464 + 0.75	0.33	816 + 0.37	0.80	31504 + 1.07
Segment	3.22	73309 + 2.21	0.77	1451 + 0.79	2.25	62647 + 2.71
Sonar	0.15	2442 + 0.05	0.31	828 + 0.33	0.32	3729 + 0.28
Soybean	3.73	104873 + 2.69	0.31	762 + 0.31	3.14	75660 + 5.51
Spambase	1.83	47412 + 1.46	1.40	2016 + 0.73	0.32	33088 + 4.79
Teaching	0.05	1879 + 0.06	0.26	676 + 0.26	0.35	8955 + 0.35
Tic-tac-toe	0.08	2842 + 0.08	0.32	746 + 0.32	0.13	8184 + 0.45
Vehicle	0.60	17217 + 0.38	0.43	843 + 0.42	0.70	24987 + 0.88
Vowel	1.96	48726 + 1.24	0.57	977 + 0.50	2.31	61085 + 7.67
Waveform	3.34	111562 + 1.74	2.46	3268 + 1.58	1.66	96197 + 11.44
Wine	0.07	2104 + 0.05	0.26	700 + 0.26	0.28	7197 + 0.33
Ave. ratio	1.0	29043.7 + 0.8	3.6	8953.4 + 3.5	2.4	69238.2 + 4.3

Table 4: Average execution time (in seconds) for training XGBoost, random forest and gradient boosting (more details in the text)

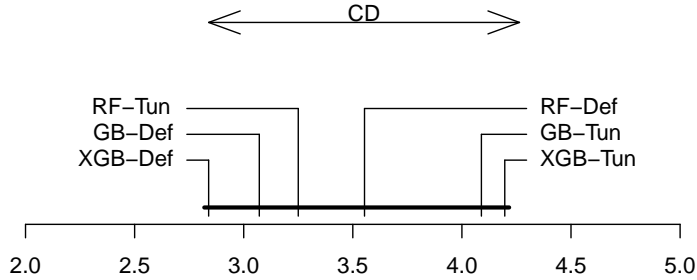


Figure 1: Average ranks (a higher rank is better) for the tested methods across 28 datasets (Critical difference CD= 1.42)

%	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 5: Percentage of times the average rank of XGBoost improves when changing the `learning_rate` from the value in the first column to the value in the first row

configurations. Then, for each given parameter, say *ParamX*, and parameter value, say *ParamX_valueA*, we compute the percentage of times that the average rank improves when *ParamX=ParamX_valueA* is changed to another value *ParamX_valueB* and no other parameter is modified. These results are shown in Tables 5 to 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 first column by the value on the corresponding top row of the table. Values above 50% are highlighted with a yellow background. For instance, as shown by Table 5, 94.9% of the times that the learning rate parameter goes from 0.05 to 0.025, while keeping the rest of the parameters fixed, the average rank across all the analyzed datasets improves.

From these tables, we can see what the most favorable parameter values in general are. In Table 5, 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 to the rest of the 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 parameter value is simply a maximum. Furthermore, the actual depth of the trees is also selected through the gamma 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 parameters: 0.05, 0.2, 100 (unlimited), sqrt and 0.75 for learning rate, gamma, depth, features and subsampling rate respectively.

In Table 11, the average error for the proposed parameters is shown in column “Prop. Def.”. For reference, the default parametrization is also shown in the first column. As it can be observed, the proposed fixed parametrization improves over the results of the default setting. With the proposed default setting, better results can be obtained in 17 out of 28 datasets with notable differences

%	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 6: Percentage of times the average rank of XGBoost improves when changing the `gamma` from the value in the first column to the value in the first row

Table 7:

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

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

%	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

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

in datasets as *Echo*, *Sonar* or *Tic-tac-toe*. In addition, when the default setting is better than the proposed parameters, the differences are in general small, except for *German*, *Parkinson* and *Vehicle*.

In spite of the improvements on average achieved by the proposed parameter setting, it seems clear that parameter optimization is necessary to further improve the performance of XGBoost and to adapt the model to the characteristics of each specific dataset. In this context, we carry out an experiment to explore two different parameter grids. On one hand, we would like to analyze the differences between gradient boosting and XGBoost in more detail. 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 parameter to explicitly control the complexity of the decision trees (i.e. γ). In order to analyze whether this parameter provides any advantage in the classification performance of XGBoost, a grid with the same parameter values as the ones given by Table 2 is used except for γ , which is always set to 0.0. On the other hand, we have observed that, in the case of random forest, tuning the randomization parameters is not very productive in general. As shown in Figure 1, the average rank of default random forest is better than the rank of the forests for which the randomization parameters are tuned. Hence, the second proposed grid is to tune the optimization parameters of XGBoost (i.e. learning rate, γ and depth) keeping the randomization parameters (i.e. random features and subsampling) fixed. The randomization parameters will be fixed to 0.75 for the subsampling ratio and to \sqrt{p} for the number of features as suggested by Tables 10 and 9 respectively. The average generalization errors for XGBoost when using these two grids are shown in Table 11. Column “No γ ” shows the results for the grid that does not tune γ (i.e. $\gamma=0$), and column “No rand tun.” shows the results for the grid that does not tune the randomization parameters. The best average test error for each dataset is highlighted with a different background color. Additionally, the average ranks for this table are shown in Figure 2 following the methodology proposed in [6]. In this figure, differences among methods connected with a horizontal solid line are not statistically significant.

The results shown in Table 11 and Figure 2 are quite interesting. The number of best results are 11, 9 and 8 when the grid without randomization parameter tuning, the full grid and the grid without γ tuning are applied respectively. Similarly, the average ranks for these three methods, shown in Figure 2, is favorable to the grid without randomization parameter tuning, then for the full grid and finally the grid without γ tuning. 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, 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{p}).

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 4. As shown in the

Dataset	Default	Prop. Def.	Tuned	No rand tun.	No gamma
Australian	86.94%±2.73	88.11%±3.08	87.53%±3.47	87.54%±3.00	86.95%±3.62
Banknote	99.64%±0.59	99.42%±0.54	99.64%±0.49	99.56%±0.58	99.71%±0.48
Breast	96.28%±1.15	96.28%±1.15	95.99%±1.68	96.99%±1.50	95.99%±1.55
Cleveland	81.14%±7.25	82.78%±6.24	83.16%±7.94	82.12%±6.69	83.09%±8.53
Dermatology	96.74%±3.21	97.27%±3.24	97.27%±3.24	98.39%±3.50	98.08%±3.29
Diabetes	75.65%±5.11	75.91%±4.30	76.56%±4.50	76.17%±4.43	77.08%±4.51
Echo	94.46%±6.80	98.75%±3.75	98.75%±3.75	98.75%±3.75	98.75%±3.75
Ecoli	86.87%±5.31	87.17%±6.50	89.05%±4.12	87.50%±5.88	87.81%±5.48
German	79.00%±4.22	75.90%±4.53	77.40%±4.13	77.30%±4.22	77.20%±3.28
Heart	79.26%±5.29	81.85%±5.84	84.07%±5.98	84.44%±6.15	84.81%±4.81
Hepatitis	59.21%±8.28	59.83%±9.75	67.00%±6.56	67.50%±11.82	66.92%±11.04
Ionosphere	92.56%±2.69	93.72%±2.51	92.59%±3.17	92.87%±3.43	91.99%±3.37
Iris	92.67%±6.29	95.33%±4.27	94.00%±4.67	95.33%±4.27	94.00%±5.54
Liver	68.65%±4.69	68.98%±5.90	68.11%±6.12	69.97%±6.13	69.31%±5.60
Magic04	87.47%±0.57	88.66%±0.45	88.63%±0.48	88.62%±0.31	88.46%±0.47
New-thyroid	95.80%±3.26	95.35%±3.58	95.80%±3.26	95.30%±3.01	94.85%±3.29
Parkinsons	92.14%±5.72	91.70%±4.35	90.14%±4.01	90.70%±3.37	90.14%±4.72
Phishing	89.65%±2.44	89.07%±2.30	91.13%±1.30	90.69%±1.99	90.03%±2.22
Segment	98.48%±0.70	98.40%±0.87	98.70%±0.77	98.66%±0.81	98.66%±0.79
Sonar	85.59%±6.44	87.52%±8.17	86.97%±4.84	87.50%±6.29	87.02%±7.45
Soybean	94.80%±3.35	94.65%±3.28	95.22%±2.96	94.65%±3.05	95.53%±3.12
Spambase	95.17%±1.29	95.67%±1.06	95.57%±1.28	95.65%±1.20	95.76%±1.27
Teaching	63.55%±8.00	64.21%±9.54	64.26%±14.15	66.84%±10.86	63.55%±11.86
Tic-tac-toe	96.56%±1.63	99.58%±0.51	100.00%±0.00	100.00%±0.00	100.00%±0.00
Vehicle	78.74%±3.29	76.95%±2.69	77.18%±2.31	79.21%±2.80	77.79%±2.83
Vowel	92.63%±3.53	94.34%±2.76	95.86%±2.14	94.95%±2.12	95.56%±2.31
Waveform	85.72%±1.05	85.54%±1.37	85.72%±1.77	85.78%±1.23	85.56%±1.80
Wine	97.18%±2.83	98.82%±2.37	98.82%±2.37	98.82%±2.37	98.82%±2.37

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

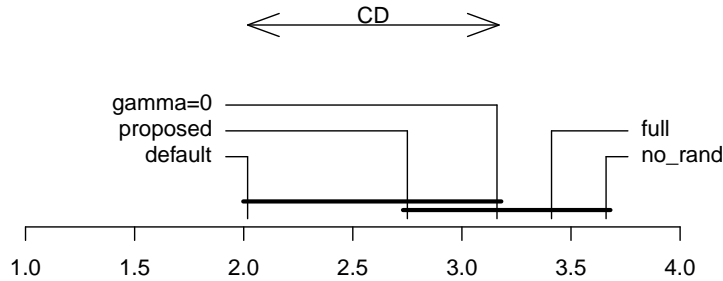


Figure 2: Average ranks (higher rank is better) for different XGBoost configurations (Critical difference $CD=1.15$)

last row of this table for the tested settings, performing the grid search without tuning the randomization parameters is over 16 times faster than tuning the full grid on average. These results reinforce the fact that tuning the randomization parameter is unnecessary.

4 Conclusion

In this study we present an empirical analysis of XGBoost, a method based on gradient boosting that has proven to be an efficient challenge solver. Specifically, the performance of XGBoost in terms of training speed and accuracy is compared with the performance of gradient boosting and random forest under a wide range of classification tasks. In addition, the parameter tuning process of XGBoost is thoroughly analyzed.

The results of this study show that the most accurate classifier, in terms of the number of problems with the best performance in the problems investigated, was gradient boosting. Nevertheless, the differences with respect to XGBoost and to random forest using the default parameters are not statistically significant in terms of average ranks. We observed that XGBoost and gradient boosting trained using the default parameters of the packages were the least successful methods. In consequence, we conclude that a meticulous 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 parameter values were used (those originally proposed by Breiman). In fact, tuning in XGBoost the randomization 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

Dataset	Default	Proposed	Tuned	No gamma	No rand tun.
Australian	0.10	0.09	2775 + 0.08	149 + 0.08	340 + 0.11
Banknote	0.10	0.11	2874 + 0.06	377 + 0.06	340 + 0.06
Breast	0.06	0.06	1733 + 0.04	108 + 0.05	212 + 0.04
Cleveland	0.05	0.04	1357 + 0.02	78 + 0.03	165 + 0.02
Dermatology	0.46	0.28	11014 + 0.27	572 + 0.27	1281 + 0.27
Diabetes	0.08	0.11	2982 + 0.06	163 + 0.08	365 + 0.05
Echo	0.02	0.01	532 + 0.01	33 + 0.01	65 + 0.01
Ecoli	0.17	0.15	4518 + 0.11	284 + 0.13	538 + 0.11
German	0.17	0.19	5453 + 0.16	251 + 0.13	670 + 0.19
Heart	0.05	0.05	1268 + 0.03	74 + 0.03	155 + 0.02
Hepatitis	0.04	0.03	1071 + 0.02	61 + 0.02	131 + 0.02
Ionosphere	0.14	0.06	2487 + 0.05	112 + 0.04	289 + 0.04
Iris	0.04	0.04	1308 + 0.03	171 + 0.04	154 + 0.03
Liver	0.07	0.09	2308 + 0.05	129 + 0.05	283 + 0.06
Magic04	3.32	8.07	123764 + 7.86	6618 + 7.38	14974 + 8.11
New-thyroid	0.05	0.05	1728 + 0.04	112 + 0.05	198 + 0.04
Parkinsons	0.05	0.03	1169 + 0.03	64 + 0.02	138 + 0.03
Phishing	0.36	0.47	12464 + 0.75	746 + 0.40	1551 + 0.53
Segment	3.22	2.13	73309 + 2.21	3878 + 1.48	8447 + 2.25
Sonar	0.15	0.05	2442 + 0.05	103 + 0.04	280 + 0.06
Soybean	3.73	2.80	104873 + 2.69	5892 + 2.65	12730 + 3.49
Spambase	1.83	1.44	47412 + 1.46	1621 + 1.20	5788 + 1.54
Teaching	0.05	0.07	1879 + 0.06	124 + 0.06	228 + 0.05
Tic-tac-toe	0.08	0.10	2842 + 0.08	171 + 0.08	347 + 0.09
Vehicle	0.60	0.59	17217 + 0.38	899 + 0.28	1993 + 0.35
Vowel	1.96	1.60	48726 + 1.24	2887 + 1.40	5577 + 1.24
Waveform	3.34	4.13	111562 + 1.74	5028 + 1.73	13073 + 2.15
Wine	0.07	0.06	2104 + 0.05	122 + 0.05	231 + 0.05
Ave. ratio	1.0	1.0	29043.7 + 0.8	1782.3 + 0.8	3472.0 + 0.8

Table 12: Average execution time (in seconds) for training XGBoost, random forest and gradient boosting (more details in the text)

fixed the values of the subsampling rate to 0.75 without replacement and the number of features to sqrt, reducing the size of the parameter grid search 16 fold and improving the average performance of XGBoost.

Finally, from the experiments of this study, which are based on grid search parameter tuning using within-train 10-fold cross-validation, the tuning phase contributed to over 99.9% of the computational effort necessary to train gradient boosting or XGBoost. The grid search time can however be dramatically reduced when the smaller proposed grid is used for XGBoost.

These results are not necessarily in contradiction with the top performances obtained by XGBoost in Kaggle competitions. The best contender in such competitions is the single model that achieves the best performance even if it is only for a slight margin. XGBoost allows for a fine parameter tuning using a computationally efficient algorithm. This is not as feasible with random forest (as small gains are obtained, if at all, with parameter tuning) or with gradient boosting, which requires longer computational times.

Acknowledgements

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)

References

- [1] Ismail Babajide Mustapha and Faisal Saeed. Bioactive molecule prediction using extreme gradient boosting. *Molecules*, 21(8), 2016.
- [2] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman & Hall, New York, 1984.
- [3] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [4] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 161–168, New York, NY, USA, 2006. ACM Press.
- [5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.
- [6] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

- [7] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toni Pitassi, Omer Reingold, and Aaron Roth. Generalization in adaptive data analysis and hold-out reuse. In *Advances in Neural Information Processing Systems 28*, pages 2350–2358. 2015.
- [8] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, 15:3133–3181, 2014.
- [9] Jerome H. Friedman. Greedy function approximation: a Gradient Boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001.
- [10] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367 – 378, 2002. Nonlinear Methods and Data Mining.
- [11] M. Gumus and M. S. Kiran. Crude oil price forecasting using xgboost. In *2017 International Conference on Computer Science and Engineering (UBMK)*, pages 1100–1103, Oct 2017.
- [12] M. Lichman. UCI machine learning repository, 2013.
- [13] N. Mirabal, E. Charles, E. C. Ferrara, P. L. Gonthier, A. K. Harding, M. A. Sanchez-Conde, and D. J. Thompson. 3fgl demographics outside the galactic plane using supervised machine learning: Pulsar and dark matter subhalo interpretations. *The Astrophysical Journal*, 825(1):69, 2016.
- [14] 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, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [15] Lior Rokach. Decision forest: Twenty years of research. *Information Fusion*, 27:111 – 125, 2016.
- [16] Alberto Torres-Barrán, Álvaro Alonso, and José R. Dorronsoro. Regression tree ensembles for wind energy and solar radiation prediction. *Neurocomputing (2017)*, 2017.
- [17] Ana Valdivia, M. Victoria Luzn, Erik Cambria, and Francisco Herrera. Consensus vote models for detecting and filtering neutrality in sentiment analysis. *Information Fusion*, 44:126 – 135, 2018.
- [18] Yufei Xia, Chuanzhe Liu, YuYing Li, and Nana Liu. A boosted decision tree approach using bayesian hyper-parameter optimization for credit scoring. *Expert Systems with Applications*, 78:225 – 241, 2017.
- [19] Robert E. Schapire Yoav Freund. A Short Introduction to Boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771 – 780, 1999.