Hyperparameters and Tuning Strategies for Random Forest

by Philipp Probst, Marvin Wright and Anne-Laure Boulesteix

February 27, 2019

Abstract

The random forest algorithm (RF) has several hyperparameters that have to be set by the user, e.g., the number of observations drawn randomly for each tree and whether they are drawn with or without replacement, the number of variables drawn randomly for each split, the splitting rule, the minimum number of samples that a node must contain and the number of trees. In this paper, we first provide a literature review on the parameters' influence on the prediction performance and on variable importance measures.

It is well known that in most cases RF works reasonably well with the default values of the hyperparameters specified in software packages. Nevertheless, tuning the hyperparameters can improve the performance of RF. In the second part of this paper, after a brief overview of tuning strategies we demonstrate the application of one of the most established tuning strategies, model-based optimization (MBO). To make it easier to use, we provide the **tuneRanger** R package that tunes RF with MBO automatically. In a benchmark study on several datasets, we compare the prediction performance and runtime of **tuneRanger** with other tuning implementations in R and RF with default hyperparameters.

1 Introduction

The random forest algorithm (RF) first introduced by Breiman (2001) has now grown to a standard non-parametric classification and regression tool for constructing prediction rules based on various types of predictor variables without making any prior assumption on the form of their association with the response variable. RF has been the topic of several reviews in the last few years including our own review (Boulesteix et al., 2012b) and others (Criminisi et al., 2012; Ziegler and König, 2014; Biau and Scornet, 2016; Belgiu and Drăguţ, 2016). RF involves several hyperparameters controlling the structure of each individual tree (e.g., the minimal size *nodesize* a node should have to be split) and the structure and size of the forest (e.g., the number of trees) as well as its level of randomness (e.g., the number *mtry* of variables considered as candidate splitting variables at each split or the sampling scheme used to generate the datasets on which the trees are built). The impact of these hyperparameters has been studied in a number of papers. However, results on this impact are often focused on single hyperparameters and provided as by-product of studies devoted to other topics (e.g., a new variant of RF) and thus are difficult to find for readers without profound knowledge of the literature. Clear guidance is missing and the choice of adequate values for the parameters remains a challenge in practice.

It is important to note that RF may be used in practice for two different purposes. In some RF applications, the focus is on the construction of a classification or regression rule with good accuracy that is intended to be used as a prediction tool on future data. In this case, the objective is to derive a rule with high prediction performance—where performance can be defined in different ways depending on the context, the simplest approach being to consider the classification error rate in the case of classification and the mean squared error in the case of regression. In other RF applications, however, the goal is not to derive a classification or regression rule but to investigate the relevance of the candidate predictor variables for the prediction problem at hand or, in other words, to assess their respective contribution to the prediction of the response variable. See the discussion by Shmueli et al. (2010) on the difference between "predicting" and "explaining". These two objectives have to be kept in mind when investigating the effect of parameters. Note, however, that there might be overlap of these two objectives: For example one might use a variable selection procedure based on variable importance measures to obtain a well performing prediction rules using RF.

Note that most hyperparameters are so-called "tuning parameters", in the sense that their values have to be optimized carefully—because the optimal values are dependent on the dataset at hand. Optimality here refers to a certain performance measure that have to be chosen beforehand. An important concept related to parameter tuning is overfitting: parameter values corresponding to complex rules tend to *overfit* the training data, i.e. to yield prediction rules that are too specific to the training data—and perform very well for this data but probably worse for independent data. The selection of such sub-optimal parameter values can be partly avoided by using a test dataset or cross-validation procedures for tuning. In the case of random forest the out-of-bag observations can also be used.

We will see that for random forest not all but most presented parameters are tuning parameters. Furthermore, note that the distinction between hyperparameters and algorithm variants is blurred. For example, the splitting rule may be considered as a (categorical) hyperparameter, but also as defining distinct variants of the RF algorithm. Some arbitrariness is unavoidable when distinguishing hyperparameters from variants of RF. In the present paper, considered hyperparameters are the number of candidate variables considered at each split (commonly denoted as *mtry*), the hyperparameters specifying the sampling scheme (the *replace* argument and the sample size), the minimal node size and related parameters, the number of trees, and the splitting rule.

This paper addresses specifically the problem of the choice of parameters of the random forest algorithm from two different perspectives. Its first part presents a review of the literature on the choice of the various parameters of RF, while the second part presents different tuning strategies and software packages for obtaining optimal hyperparameter values which are finally compared in a benchmark study.

2 Literature Review

In the first section of this literature review, we focus on the influence of the hyperparameters on the prediction performance, e.g., the error rate or the area under the ROC Curve (AUC), and the runtime of random forest, while literature dealing specifically with the influence on the variable importance is reviewed in the second section. In Table 1 the different hyperparameters with description and typical default values are displayed.

Hyperparameter	Description	Typical default values
mtry	Number of drawn candidate variables in each split	$\sqrt{p}$, $p/3$ for regression
sample size	Number of observations that are drawn for each tree	n
replacement	Draw observations with or without replacement	TRUE (with replacement)
node size	Minimum number of observations in a terminal node	1 for classification, 5 for regression
number of trees	Number of trees in the forest	500, 1000
splitting rule	Splitting criteria in the nodes	Gini impurity, p-value, random

Table 1: Overview of the different hyperparameter of random forest and typical default values. n is the number of observations and p is the number of variables in the dataset.

2.1 Influence on performance

As outlined in Breiman (2001), *"[t]he randomness used in tree construction has to aim for low correlation ρ while maintaining reasonable strength"*. In other words, an optimal compromise between low correlation and reasonable strength of the trees has to be found. This can be controlled by the parameters *mtry*, sample size and node size which will be presented in Section 2.1.1, 2.1.2 and 2.1.3, respectively. Section 2.1.4 handles the number of trees, while Section 2.1.5 is devoted to the splitting criterion.

2.1.1 Number of randomly drawn candidate variables (*mtry*)

One of the central hyperparameters of RF is *mtry*, as denoted in most RF packages, which is defined as the number of randomly drawn candidate variables out of which each split is selected when growing a tree. Lower values of *mtry* lead to more different, less correlated trees, yielding better stability when aggregating. Forests constructed with a low *mtry* also tend to better exploit variables with moderate effect on the response variable, that would be masked by variables with strong effect if those had been candidates for splitting. However, lower values of *mtry* also lead to trees that

perform on average worse, since they are built based on suboptimal variables (that were selected out of a small set of randomly drawn candidates): possibly non-important variables are chosen. We have to deal with a trade-off between stability and accuracy of the single trees.

As default value in several software packages *mtry* is set to $\sqrt{p}$ for classification and $p/3$ for regression with p being the number of predictor variables. In their paper on the influence of hyperparameters on the accuracy of RF, Bernard et al. (2009) conclude that $mtry = \sqrt{p}$ is a reasonable value, but can sometimes be improved. They especially outline that the real number of relevant predictor variables highly influences the optimal *mtry*. If there are many relevant predictor variables, *mtry* should be set small because then not only the strongest influential variables are chosen in the splits but also less influential variables, which can provide small but relevant performance gains. These less influential variables might, for example, be useful for the prediction of a small group of observations that stronger variables fail to predict correctly. If *mtry* is large, however, these less influential variables might not have the chance to contribute to prediction because stronger variables are preferably selected for splitting and thus "mask" the smaller effects. On the other hand, if there are only a few relevant variables out of many, which is the case in many genetic datasets, *mtry* should be set high, so that the algorithm can find the relevant variables (Goldstein et al., 2011). A large *mtry* ensures that there is (with high probability) at least one strong variable in the set of *mtry* candidate variables.

Further empirical results are provided by Genuer et al. (2008). In their low dimensional classification problems $mtry = \sqrt{p}$ is convenient regarding the error rate. For low dimensional regression problems, in their examples $\sqrt{p}$ performs better than $p/3$ regarding the mean squared error. For high dimensional data they observe lower error rates for higher *mtry* values for both classification and regression, corroborating Goldstein et al. (2011).

Computation time decreases approximately linearly with lower *mtry* values (Wright and Ziegler, 2017), since most of RF's computing time is devoted to the selection of the split variables.

2.1.2 Sampling scheme: sample size and replacement

The sample size parameter determines how many observations are drawn for the training of each tree. It has a similar effect as the *mtry* parameter. Decreasing the sample size leads to more diverse trees and thereby lower correlation between the trees, which has a positive effect on the prediction accuracy when aggregating the trees. However, the accuracy of the single trees decreases, since fewer observations are used for training. Hence, similarly to the *mtry* parameter, the choice of the sample size can be seen as a trade-off between stability and accuracy of the trees. Martínez-Muñoz and Suárez (2010) carried out an empirical analysis of the dependence of the performance on the sample size. They concluded that the optimal value is problem dependent and can be estimated with the out-of-bag predictions. In most datasets they observed better performances when sampling less observations than the standard choice (which is to sample as many observations with replacement as the number of observations in the dataset). Setting it to lower values reduces the runtime.

Moreover, Martínez-Muñoz and Suárez (2010) claim that there is no substantial performance difference between sampling with replacement or without replacement when the sample size parameter is set optimally. However, both theoretical (Janitza et al., 2016) and empirical results (Strobl et al., 2007) show that sampling with replacement may induce a slight variable selection bias when categorical variables with varying number of categories are considered. In these specific cases, performance may be impaired by sampling *with* replacement, even if this impairment could not be observed by Martínez-Muñoz and Suárez (2010) when considering averages over datasets of different types.

2.1.3 Node size

The *nodesize* parameter specifies the minimum number of observations in a terminal node. Setting it lower leads to trees with a larger depth which means that more splits are performed until the terminal nodes. In several standard software packages the default value is 1 for classification and 5 for regression. It is believed to generally provide good results (Díaz-Uriarte and De Andres, 2006; Goldstein et al., 2011) but performance can potentially be improved by tuning it (Lin and Jeon, 2006). In particular, Segal (2004) showed an example where increasing the number of noise variables leads to a higher optimal node size.

Our own preliminary experiments suggest that the computation time decreases approximately exponentially with increasing node size. In our experience, especially in large sample datasets it may be helpful to set this parameter to

a value higher than the default as it decreases the runtime substantially, often without substantial loss of prediction performance (Segal, 2004).

Note that other hyperparameters than the node size may be considered to control the size of the trees. For example, the R package **party** (Hothorn et al., 2006) allows to set the minimal size, *minbucket*, that child nodes should have for the split to be performed. The hyperparameters *nodesize* and *minbucket* are obviously related, since the size of all parent nodes equals at least twice the value of *minbucket*. However, setting *minbucket* to a certain value does not in general lead to the same trees as setting *nodesize* to double this value. To explain this, let us consider a node of size $n = 10$ and a candidate categorical predictor variable taking value 1 for $n_1 = 9$ of the $n = 10$ observations of the node, and value 0 for the remaining observation. If we set *nodesize* to 10 and do not put any restriction on *minbucket* (i.e., set it to 1), our candidate variable can be selected for splitting. If, however, we proceed the other way around and set *minbucket* to 5 while not putting any restriction on *nodesize* (i.e., while setting it to 2), our candidate variable cannot be selected, because it would produce a—too small—child node of size 1. On the one hand, one may argue that splits with two large enough child nodes are preferable—an argument in favor of setting *minbucket* to a value larger than one. On the other hand, this may yield a selection bias in the case of categorical variables, as demonstrated through this simple example and also discussed in Boulesteix et al. (2012a) in the context of genetic data.

Furthermore, in the R package **randomForest** (Liaw and Wiener, 2002), it is possible to specify *maxnodes*, the maximum number of terminal nodes that trees in the forest can have, while the R package **party** allows to specify the strongly related hyperparameter *maxdepth*, the maximal depth of the trees, which is the maximum number of splits until the terminal node.

2.1.4 Number of trees

The number of trees in a forest is a parameter that is not tunable in the classical sense but should be set sufficiently high (Díaz-Uriarte and De Andres, 2006; Oshiro et al., 2012; Probst and Boulesteix, 2017). Out-of-bag error curves (slightly) increasing with the number of trees are occasionally observed for certain error measures (see Probst and Boulesteix, 2017, for an empirical study based on a large number of datasets). According to measures based on the mean quadratic loss such as the mean squared error (in case of regression) or the Brier score (in case of classification), however, more trees are always better, as theoretically proved by Probst and Boulesteix (2017).

The convergence rate, and thus the number of trees needed to obtain optimal performance, depends on the dataset's properties. Using a large number of datasets, Oshiro et al. (2012) and Probst and Boulesteix (2017) show empirically that the biggest performance gain can often be achieved when growing the first 100 trees. The convergence behaviour can be investigated by inspecting the out-of-bag curves showing the performance for a growing number of trees. Probst and Boulesteix (2017) argue that the error rate is not the optimal measure for that purpose because, by considering a prediction as either true or false, one ignores much of the information output by the RF and focuses too much on observations that are close to the prediction boundary. Instead, they recommend the use of other measures based on the predicted class probabilities such as the Brier score or the logarithmic loss, as implemented in the R package **OOBCurve** (Probst, 2017).

Note that the convergence rate of RF does not only depend on the considered dataset's characteristics but possibly also on hyperparameters. Lower sample size (see Section 2.1.2), higher node size values (see Section 2.1.3) and smaller *mtry* values (see Section 2.1.1) lead to less correlated trees. These trees are more different from each other and are expected to provide more different predictions. Therefore, we suppose that more trees are needed to get clear predictions for each observation which leads to a higher number of trees for obtaining convergence.

The computation time increases linearly with the number of trees. As trees are trained independently from each other they can be trained in parallel on several CPU cores which is implemented in software packages such as **ranger** (Wright and Ziegler, 2017).

2.1.5 Splitting rule

The splitting rule is not a classical hyperparameter as it can be seen as one of the core properties characterizing the RF. However, it can in a large sense also be considered as a categorical hyperparameter. The default splitting rule of Breiman's original RF (Breiman, 2001) consists of selecting, out of all splits of the (randomly selected *mtry*) candidate variables, the split that minimizes the Gini impurity (in the case of classification) and the weighted variance (in case of regression). This method favors the selection of variables with many possible splits (e.g., continuous variables or categorical variables with many categories) over variables with few splits (the extreme case being binary variables, which have only one possible split) due to multiple testing mechanisms (Strobl et al., 2007).

Conditional inference forests (CIF) introduced by Hothorn et al. (2006) and implemented in the R package **party** and in the newer package **partykit** (Hothorn and Zeileis, 2015) allow to avoid this variable selection bias by selecting the variable with the smallest p-value in a global test (i.e., *without* assessing all possible splits successively) in a first step, and selecting the best split from the selected variable in a second step by maximizing a linear test statistic. Note that the global test to be used in the first step depends on the scale of the predictor variables and response variable. Hothorn et al. (2006) suggest several variants. A computationally fast alternative using p-value approximations for maximally selected rank statistics is proposed by Wright et al. (2017). This variant is available in the **ranger** package for regression and survival outcomes.

When tests are performed for split selection, it may only make sense to split if the p-values fall below a certain threshold, which should then be considered as a hyperparameter. In the R package **party** the hyperparameter *mincriterion* represents one minus the p-value threshold and in **ranger** the hyperparameter *alpha* is the p-value threshold.

To increase computational efficiency, splitting rules can be randomized (Geurts et al., 2006). To this end, only a randomly selected subset of possible splitting values are considered for a variable. The size of these subsets is specified by the hyperparameter *numRandomCuts* in the **extraTrees** package and by *num.random.splits* in **ranger**. If this value is set to 1, this variant is called *extremely randomized trees* (Geurts et al., 2006). In addition to runtime reduction, randomized splitting might also be used to add a further component of randomness to the trees, similar to *mtry* and the sample size.

Until now, none of the existing splitting rules could be proven as superior to the others in general regarding the performance. For example, the splitting rule based on the decrease of Gini impurity implemented in Breiman's original version is affected by a serious variable selection bias as outlined above. However, if one considers datasets in which variables with many categories are more informative than variables with less categories, this variable selection bias—even if it is in principle a flaw of the approach—may accidentally lead to improved accuracy. Hence, depending on the dataset and its properties one or the other method may be better. Appropriate benchmark studies based on simulation or real data have to be designed, to evaluate in which situations which splitting rule performs better.

Regarding runtime, extremely randomized trees are the fastest as the cutpoints are drawn completely randomly, followed by the classical random forest, while for conditional inference forests the runtime is the largest.

2.2 Influence on variable importance

The RF variable importance (Breiman, 2001) is a measure reflecting the importance of a variable in a RF prediction rule. While effect sizes and p-values of the Wald test or likelihood ratio tests are often used to assess the importance of variables in case of logistic or linear regression, the RF variable importance measure can also automatically capture non-linear and interaction effects without specifying these a priori (Wright et al., 2016) and is also applicable when more variables than observations are available. Several variants of the variable importance exist, including the Gini variable importance measure and the permutation variable importance measure (Breiman, 2001; Strobl et al., 2007). In this section we focus on the latter, since the former has been shown to be strongly biased. The Gini variable importance measure assigns higher importance values to variables with more categories or continuous variables (Strobl et al., 2007) and to categorical variables with equally sized categories (Boulesteix et al., 2012a) even if all variables are independent of the response variable.

Many of the effects of the hyperparameters described in the previous section 2.1 are expected to also have an effect on the variable importance. However, specific research on this influence is still in its infancy. Most extensive is probably the research about the influence of the number of trees. In contrast, the literature is very scarce as far as the sample size and node size are concerned.

2.2.1 Number of trees

More trees are generally required for stable variable importance estimates (Genuer et al., 2010; Goldstein et al., 2011), than for the simple prediction purpose. Lunetta et al. (2004) performed simulations with more noisy variables than truly associated covariates and concluded that multiple thousands of trees must be trained in order to get stable estimates of the variable importance. The more trees are trained, the more stable the predictions should be for the variable importance. In order to assess the stability one could train several random forests with a fixed number of trees and check whether the ranking of the variables by importance are different between the forests.

2.2.2 *mtry*, splitting rule and node size

Genuer et al. (2010) examine the influence of the parameter *mtry* on the variable importance. They conclude that increasing the *mtry* value leads to much higher magnitudes of the variable importances. As already outlined in Section 2.1.5 the random forest standard splitting rule is biased when predictor variables vary in their scale. This has also a substantial impact on the variable importance (Strobl et al., 2007). In the case of the Gini variable importance, predictor variables with many categories or numerical values receive on average a higher variable importance than binary variables if both variables have no influence on the outcome variable. The permutation variable importance remains unbiased in these cases, but there is a higher variance of the variable importance for variables with many categories. This could not be observed for conditional inference forests combined with subsampling (without replacement) as sampling procedure and therefore Strobl et al. (2007) recommend to use this method for getting reliable variable importance measures.

Grömping (2009) compared the influence of *mtry* and the node size on the variable importance of the standard random forest and of the conditional inference forest. Higher *mtry* values lead to lower variable importance of weak regressors. The values of the variable importance from the standard random forest were far less dependent on *mtry* than the ones from the conditional inference forests. This was due to the much larger size (i.e., number of splits until the terminal node) of individual trees in the standard random forest. Decreasing the tree size (for example by setting a higher node size value) while setting *mtry* to a small value leads to more equal values of the variable importances of all variables, because there was less chance that relevant variables were chosen in the splitting procedures.

3 Tuning random forest

Tuning is the task of finding optimal hyperparameters for a learning algorithm for a considered dataset. In supervised learning (e.g., regression and classification), optimality may refer to different performance measures (e.g., the error rate or the AUC) and to the runtime which can highly depend on hyperparameters in some cases as outlined in Section 2. In this paper we mainly focus on the optimality regarding performance measures.

Even if the choice of adequate values of hyperparameters has been partially investigated in a number of studies as reviewed in Section 2, unsurprisingly the literature provides general trends rather than clear-cut guidance. In practice, users of RF are often unsure whether alternative values of tuning parameters may improve performance compared to default values. Default values are given by software packages or can be calculated by using previous datasets (Probst et al., 2018). In the following section we will review literature about the "tunability" of random forest. "Tunability" is defined as the amount of performance gain compared to default values that can be achieved by tuning one hyperparameter ("tunability" of the hyperparameter) or all hyperparameters ("tunability" of the algorithm); see Probst et al. (2018) for more details. Afterwards, evaluation strategies, evaluation measures and tuning search strategies are presented. Then we review software implementations of tuning of RF in the programming language R and finally show the results of a large scale benchmark study comparing the different implementations.

3.1 Tunability of random forest

Random forest is an algorithm which is known to provide good results in the default settings (Fernández-Delgado et al., 2014). Probst et al. (2018) measure the "tunability" of algorithms and hyperparameters of algorithms and conclude that random forest is far less tunable than other algorithms such as support vector machines. Nevertheless, a small performance gain (e.g., an average increase of the AUC of 0.010 based on the 38 considered datasets) can be achieved via tuning compared to the default software package hyperparameter values. This average performance gain, although moderate, can be an important improvement in some cases, when, for example, each wrongly classified observation implies high costs. Moreover, for some datasets it is much higher than 0.01 (e.g., around 0.03).

As outlined in Section 2, all considered hyperparameters might have an effect on the performance of RF. It is not completely clear, however, which of them should routinely be tuned in practice. Beyond the special case of RF, Probst et al. (2018) suggest a general framework to assess the tunability of different hyperparameters of different algorithms (including RF) and illustrate their approach through an application to 38 datasets. In their study, tuning the parameter *mtry* provides the biggest average improvement of the AUC (0.006), followed by the sample size (0.004), while the node size had only a small effect (0.001). Changing the *replace* parameter from drawing with replacement to drawing without replacement also had a small positive effect (0.002). Similar results were observed in the work of van Rijn and Hutter (2018). As outlined in Section 2.1.4, the number of trees cannot be seen as tuning parameter: higher values are generally preferable to smaller values with respect to performance. If the performance of RF with default values of the hyperparameters can be improved by choosing other values, the next question is how this choice should be performed.

3.2 Evaluation strategies and evaluation measures

A typical strategy to evaluate the performance of an algorithm with different values of the hyperparameters in the context of tuning is k-fold cross-validation. The number k of folds is usually chosen between 2 and 10. Averaging the results of several repetitions of the whole cross-validation procedure provides more reliable results as the variance of the estimation is reduced (Seibold et al., 2018).

In RF (or in general when bagging is used) another strategy is possible, namely using the out-of-bag observations to evaluate the trained algorithm. Generally, the results of this strategy are reliable (Breiman, 1996), i.e., approximate the performance of the RF on independent data reasonably well. A bias can be observed in special data situations (see Janitza and Hornung, 2018, and references therein), for example in very small datasets with $n < 20$, when there are many predictor variables and balanced classes. Since these problems are specific to particular and relatively rare situations and tuning based on out-of-bag-predictions has a much smaller runtime than procedures such as k-fold cross-validation (which is especially important in big datasets), we recommend the out-of-bag approach for tuning as an appropriate procedure for most datasets.

The evaluation measure is a measure that is dependent on the learning problem. In classification, two of the most commonly considered measures are the classification error rate and the Area Under the Curve (AUC). Two other common measures that are based on probabilities are the Brier score and the logarithmic loss. An overview of evaluation measures for classification is given in Ferri et al. (2009).

3.3 Tuning search strategies

Search strategies differ in the way the candidate hyperparameter values (i.e., the values that have to be evaluated with respect to their out-of-bag performance) are chosen. Some strategies specify all the candidate hyperparameter values from the beginning, for example, random search and grid search presented in the following subsections. In contrast, other more sophisticated methods such as F-Race (Birattari et al., 2010), general simulated annealing (Bohachevsky et al., 1986) or sequential model-based optimization (SMBO) (Jones et al., 1998; Hutter et al., 2011) iteratively use the results of the different already evaluated hyperparameter values and choose future hyperparameters considering these results. The latter procedure, SMBO, is introduced at the end of this section and used in two of the software implementations that are presented in the sections 3.4 and 3.5.

Grid search and random search

One of the simplest strategies is grid search, in which all possible combinations of given discrete parameter spaces are evaluated. Continuous parameters have to be discretized beforehand. Another approach is random search, in which hyperparameter values are drawn randomly (e.g., from a uniform distribution) from a specified hyperparameter space. Bergstra and Bengio (2012) show that for neural networks random search is more efficient in searching good hyperparameter specifications than grid search.

Sequential model-based optimization

Sequential model-based optimization (SMBO) is a very successful tuning strategy that iteratively tries to find the best hyperparameter settings based on evaluations of hyperparameters that were done beforehand. SMBO is grounded in the "black-box function optimization" literature (Jones et al., 1998) and achieves state-of-the-art performance for solving a number of optimization problems (Hutter et al., 2011). We shortly describe the SMBO algorithm implemented in the R package **mlrMBO** (Bischl et al., 2017), which is also used in the R package **tuneRanger** (Probst, 2018) described in Section 3.5. It consists of the following steps:

1. Specify an evaluation measure (e.g., the AUC in the case of classification or the mean squared error in the case of regression), also sometimes denoted as "target outcome" in the literature, an evaluation strategy (e.g., 5-fold cross-validation) and a constrained hyperparameter space on which the tuning should be executed.
2. Create an initial design, i.e., draw random points from the hyperparameter space and evaluate them (i.e., evaluate the chosen evaluation measure using the chosen evaluation strategy).
3. Based on the results obtained from the previous step following steps are iteratively repeated:
   (a) Fit a regression model (also called surrogate model, e.g., kriging (Jones et al., 1998) or RF) based on all already evaluated design points with the evaluation measure as the dependent variable and the hyperparameters as predictor variables.
   (b) Propose a new point for evaluation on the hyperparameter space based on an infill criterion. This criterion is based on the surrogate model and proposes points that have good expected outcome values and lie in regions of the hyperparameter space where not many points were evaluated yet.
   (c) Evaluate the point and add it to the already existing design points.

3.4 Existing software implementations

Several packages already implement such automatic tuning procedures for RF. We shortly describe the three most common ones in R:

- **mlrHyperopt** (Richter, 2017) uses sequential model-based optimization as implemented in the R package **mlrMBO**. It has predefined tuning parameters and tuning spaces for many supervised learning algorithms that are part of the **mlr** package. Only one line of code is needed to perform the tuning of these algorithms with the package. In case of **ranger**, the default parameters that are tuned are *mtry* (between 1 and the number of variables p) and the node size (from 1 to 10), with 25 iteration steps (step number 3 in the previous subsection) and no initial design. In **mlrHyperopt**, the standard evaluation strategy in each iteration step of the tuning procedure is 10-fold cross-validation and the evaluation measure is the mean missclassification error. The parameters, the evaluation strategy and measure can be changed by the user. As it is intended as a platform for sharing tuning parameters and spaces, users can use their own tuning parameters and spaces and upload them to the webservice of the package.
- **caret** (Kuhn, 2008) is a set of functions that attempts to streamline the process for creating predictive models. When executing **ranger** via **caret** it automatically performs a grid search of *mtry* over the whole *mtry* parameter space. By default, the algorithm evaluates 3 points in the parameter space (smallest and biggest possible mtry and the mean of these two values) with 25 bootstrap iterations as evaluation strategy. The algorithm finally chooses the value with the lowest error rate in case of classification and the lowest mean squared error in case of regression.

- `tuneRF` from the **randomForest** package implements an automatic tuning procedure for the *mtry* parameter. First, it calculates the out-of-bag error with the default *mtry* value (square root of the number of variables p for classification and $p/3$ for regression). Second, it tries out a new smaller value of *mtry* (default is to deflate *mtry* by the factor 2). If it provides a better out-of-bag error rate (relative improvement of at least 0.05) the algorithm continues trying out smaller *mtry* values in the same way. After that, the algorithm tries out larger values than the default of *mtry* until there is no more improvement, analogously to the second step. Finally the algorithm returns the model with the best *mtry* value.

3.5 The tuneRanger package

As a by-product of our literature review on tuning for RF, we created a package, **tuneRanger** (Probst, 2018), for automatic tuning of RF based on the package **ranger** through a single line of code, implementing all features that we identified as useful in other packages, and intended for users who are not very familiar with tuning strategies. The package **tuneRanger** is mainly based on the R packages **ranger** (Wright and Ziegler, 2017), **mlrMBO** (Bischl et al., 2017) and **mlr** (Bischl et al., 2016). The main function `tuneRanger` of the package works internally as follows:

- Sequential model-based optimization (SMBO, see Section 3.3) is used as tuning strategy with 30 evaluated random points for the initial design and 70 iterative steps in the optimization procedure. The number of steps for the initial design and in the optimization procedure are parameters that can be changed by the user, although the default settings 30 resp. 70 provide good results in our experiments.
- As a default, the function simultaneously tunes the three parameters *mtry*, sample size and node size. *mtry* values are sampled from the space $[0, p]$ with p being the number of predictor variables, while sample size values are sampled from $[0.2 \cdot n, 0.9 \cdot n]$ with n being the number of observations. Node size values are sampled with higher probability (in the initial design) for smaller values by sampling x from $[0, 1]$ and transforming the value by the formula $[(n \cdot 0.2)^x]$. The tuned parameters can be changed by the user by changing the argument `tune.parameters`, if, for example, only the *mtry* value should be tuned or if additional parameters, such as the sampling strategy (sampling with or without resampling) or the handling of unordered factors (see Hastie et al. (2001), chapter 9.2.4 or the help of the `ranger` package for more details), should be included in the tuning process.
- Out-of-bag predictions are used for evaluation, which makes it much faster than other packages that use evaluation strategies such as cross-validation.
- Classification as well as regression is supported.
- The default measure that is optimized is the Brier score for classification, which yields a finer evaluation than the commonly used error rate (Probst and Boulesteix, 2017), and the mean squared error for regression. It can be changed to any of the 50 measures currently implemented in the R package **mlr** and documented in the online tutorial (Schiffner et al., 2016): `http://mlr-org.github.io/mlr-tutorial/devel/html/measures/index.html`
- The final recommended hyperparameter setting is calculated by taking the best 5 percent of all SMBO iterations regarding the chosen performance measure and then calculating the average of each hyperparameter of these iterations , which is rounded in case of *mtry* and node size.

Installation and execution

In the following one can see a typical example of the execution of the algorithm. The dataset *monks-problem-1* is taken from OpenML. Execution time can be estimated beforehand with the function `estimateTimeTuneRanger` which trains a random forest with default values, multiplies the training time by the number of iterations and adds 50 for the training and prediction time of surrogate models. The function `tuneRanger` then executes the tuning algorithm:

```
library(tuneRanger)
library(mlr)
library(OpenML)
monk_data_1 = getOMLDataSet(333)$data
monk.task = makeClassifTask(data = monk_data_1, target = "class")
```

```r
# Estimate runtime
estimateTimeTuneRanger(monk.task)
# Approximated time for tuning: 1M 13S
set.seed(123)
# Tuning
res = tuneRanger(monk.task, measure = list(multiclass.brier), num.trees = 1000,
  num.threads = 2, iters = 70, iters.warmup = 30)
res
# Recommended parameter settings:
#   mtry min.node.size sample.fraction
# 1    6             2       0.8988154
# Results:
#   multiclass.brier exec.time
# 1      0.006925637    0.2938
#
# Ranger Model with the new tuned hyperparameters
res$model
# Model for learner.id=classif.ranger; learner.class=classif.ranger
# Trained on: task.id = monk_data; obs = 556; features = 6
# Hyperparameters: num.threads=2,verbose=FALSE,respect.unordered.factors=order,mtry=6,min.node.size=2,
# sample.fraction=0.899,num.trees=1e+03,replace=FALSE
```

We also performed a benchmark with 5 times repeated 5-fold cross-validation to compare it with a standard random forest with 1000 trees trained with `ranger` on this dataset. As can be seen below, we achieved an improvement of 0.014 in the error rate and 0.120 in the Brier score.

```r
  # little benchmark
  lrn = makeLearner("classif.ranger", num.trees = 1000, predict.type = "prob")
  lrn2 = makeLearner("classif.tuneRanger", num.threads = 1, predict.type = "prob")
  set.seed(354)
  rdesc = makeResampleDesc("RepCV", reps = 5, folds = 5)
  bmr = benchmark(list(lrn, lrn2), monk.task, rdesc, measures = list(mmce, multiclass.brier))
  bmr
  # Result
  #    task.id          learner.id mmce.test.mean multiclass.brier.test.mean
  # 1 monk_data     classif.ranger     0.01511905                  0.1347917
  # 2 monk_data classif.tuneRanger     0.00144144                  0.0148708
```

Further parameters

In the main function `tuneRanger` there are several parameters that can be changed. The first argument is the task, that has to be created via the **mlr** functions `makeClassifTask` or `makeRegrTask`. The argument `measure` has to be a list of the chosen measures that should be optimized, possible measures can be found with `listMeasures` or in the online tutorial of **mlr**. The argument `num.trees` is the number of trees that are trained, `num.threads` is the number of cpu threads that should be used by **ranger**, `iters` specifies the number of iterations and `iters.warmup` the number of warm-up steps for the initial design. The argument `tune.parameters` can be used to specify manually a list of the tuned parameters. The final recommended hyperparameter setting (average of the best 5 percent of the iterations) is used to train a RF model, which can be accessed via the list element `model` in the final outcome.

3.6 Benchmark study

We now compare our new R package **tuneRanger** with different software implementations with tuning procedures for random forest regarding performance and execution time in a benchmark study on 39 datasets.

3.6.1 Compared algorithms

We compare different algorithms in our benchmark study:

- Our package **tuneRanger** is used with its default settings (30 warm-up steps for the initial design, 70 iterations, tuning of the parameters *mtry*, node size and sample size, sampling without replacement) that were set before executing the benchmark experiments. The only parameter of the function that is varied is the performance measure that has to be optimized. We do not only consider the default performance measure Brier score (***tuneRangerBrier***) but also examine the versions that optimize the mean missclassification error (MMCE) (***tuneRangerMMCE***), the AUC (***tuneRangerAUC***) and the logarithmic loss (***tuneRangerLogloss***). Moreover, to examine if only tuning *mtry* is enough we run the same algorithms with only tuning the *mtry* parameter.
- The three tuning implementations of the R packages ***mlrHyperopt***, ***caret*** and ***tuneRF*** that are described in Section 3.4 are executed with their default setting. We did not include ***mlrHyperopt*** with other performance measures because we expect similar performance as with ***tuneRanger*** but very long runtimes.
- The standard RF algorithm as implemented in the package **ranger** with default settings and without tuning is used as reference, to see the improvement to the default algorithm. We use **ranger** instead of the **randomForest** package, because it is faster due to the possibility of parallelization on several cores (Wright and Ziegler, 2017).

For each method, 2000 trees are used to train the random forest. Whenever possible (for **tuneRanger**, **mlrHyperopt**, **caret** and the default **ranger** algorithm) we use parallelization with 10 CPU cores with the help of the **ranger** package (trees are grown in parallel).

3.6.2 Datasets, runtime and evaluation strategy

The benchmark study is conducted on datasets from OpenML (Vanschoren et al., 2013). We use the **OpenML100** benchmarking suite (Bischl et al., 2017) and download it via the **OpenML** R package (Casalicchio et al., 2017). For classification we only use datasets that have a binary target and no missing values, which leads to a collection of 39 datasets. More details about these datasets such as the number of observations and variables can be found in Bischl et al. (2017). We classify the datasets into small and big by using the `estimateTimeTuneRanger` function of **tuneRanger** with 10 cores. If the estimated runtime is less than 10 minutes, the dataset is classified as small, otherwise it is classified as big. This results in 26 small datasets, 13 big datasets.

For the small datasets we perform a 5-fold cross-validation and repeat it 10 times and for the big we just perform a 5-fold cross-validation. We compare the algorithms by the average of the evaluation measures mean missclassification error (MMCE), AUC, Brier score and logarithmic loss. Definitions and a good comparison between measures for classification can be found in Ferri et al. (2009). In case of error messages of the tuning algorithms (on 2 datasets for ***mlrHyperopt***, on 4 datasets for ***caret*** and on 3 datasets for ***tuneRF***), the worst result of the other algorithms are assigned to these algorithms, if it fails in more than 20 % of the cross-validation iterations, otherwise we impute missing values by the average of the rest of the results as proposed by Bischl et al. (2013).

3.6.3 Results

First, to identify possible outliers, we display the boxplots of the differences between the performances of the algorithms and the ***ranger default***. Afterwards average results and ranks are calculated and analyzed. We denote the compared algorithms as ***tuneRangerMMCE***, ***tuneRangerAUC***, ***tuneRangerBrier***, ***tuneRangerLogloss***, ***hyperopt***, ***caret***, ***tuneRF*** and ***ranger default***.

Outliers and boxplots of differences

In Figure 1 on the left side, the boxplots of the differences between the performances of the algorithms and the ***ranger default*** with outliers are depicted.

We can see that there are two datasets, for which the performance of default random forest is very bad: the error rate is by around 0.15 higher than for all the other algorithms (other evaluation measures behave similarly). For these two datasets it is essential to tune *mtry*, which is done by all tuning algorithms. The first dataset (called *monks-problems-2* in OpenML) is an artificial dataset which has 6 categorical predictor variables. If two of them take the value 2 the binary outcome is 1, otherwise 0. Setting *mtry* to the default 2 (or even worse to the value 1) leads to wrongly partitioned trees: the dependence structure between the categorical variables cannot be detected perfectly as sometimes the wrong variables are used for the splitting (since one of the two variables that were randomly chosen in the splitting procedure has to be used for the considered split). On the other hand, setting *mtry* to 6 leads to nearly perfect predictions as the dependence structure can be identified perfectly. The second dataset with bad performance of the default random forest is called *madelon* and has 20 informative and 480 non-informative predictor variables. The default *mtry* value of 22 is much too low, because too many non-informative predictor variables are chosen in the splits. The tuning algorithms choose a higher *mtry* value and get much better performances on average. For ***tuneRF*** we have 3 outliers regarding the logarithmic loss. This tuning algorithm tends to yield clear-cut predictions with probabilities near 0 and 1 for these datasets, which lead to the bad performance regarding the logarithmic loss in these cases.

Average results and ranks

The average results for the 39 datasets can be seen in Table 2 and the average ranks in Table 3. The ranks are given for each measure and each dataset separately from 1 (best) to 8 (worst) and then averaged over the datasets. Moreover, on the right side of Figure 1 we can see the boxplots of the performance differences to the ***ranger default*** without the outliers, which give an impression about the distribution of the performance differences.

	MMCE	AUC	Brier score	Logarithmic Loss	Training Runtime
tuneRangerMMCE	0.0923	0.9191	0.1357	0.2367	903.8218
tuneRangerAUC	0.0925	0.9199	0.1371	0.2450	823.4048
tuneRangerBrier	0.0932	0.9190	0.1325	0.2298	967.2051
tuneRangerLogloss	0.0936	0.9187	0.1330	0.2314	887.8342
mlrHyperopt	0.0934	0.9197	0.1383	0.2364	2713.2438
caret	0.0972	0.9190	0.1439	0.2423	1216.2770
tuneRF	0.0942	0.9174	0.1448	0.2929	862.9917
ranger default	0.1054	0.9128	0.1604	0.2733	3.8607

Table 2: Average performance results of the different algorithms.

	Error rate	AUC	Brier score	Logarithmic Loss	Training Runtime
tuneRangerMMCE	4.19	4.53	4.41	4.54	5.23
tuneRangerAUC	3.77	2.56	4.42	4.22	4.63
tuneRangerBrier	3.13	3.91	1.85	2.69	5.44
tuneRangerLogloss	3.97	4.04	2.64	2.23	5.00
mlrHyperopt	4.37	4.68	4.74	4.90	7.59
caret	5.50	5.24	6.08	5.51	4.36
tuneRF	4.90	5.08	5.44	6.23	2.76
ranger default	6.17	5.96	6.42	5.68	1.00

Table 3: Average rank results of the different algorithms.

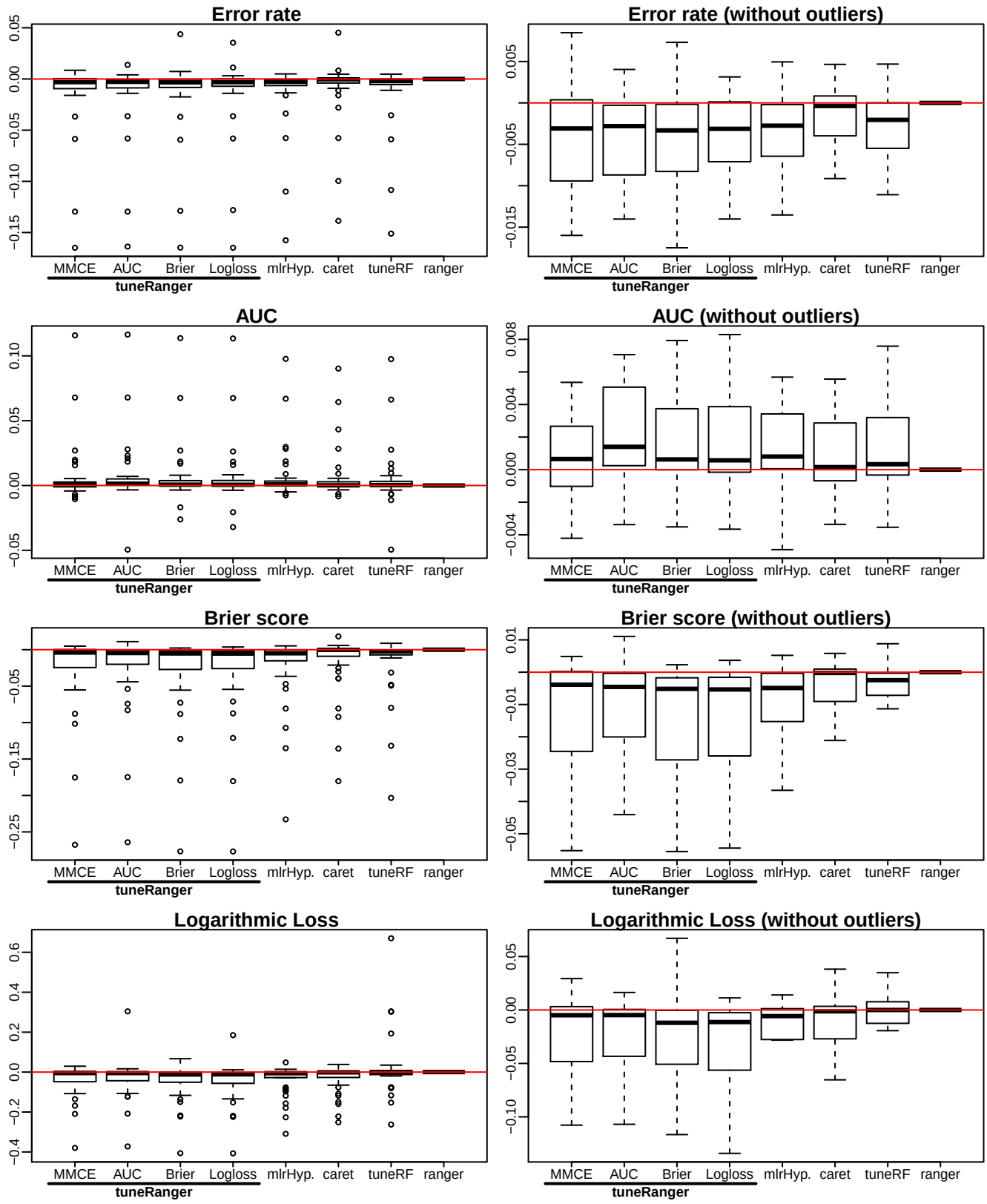

Figure 1: Boxplots of performance differences to ***ranger default***. On the left side the boxplots with outliers are depicted and on the right side the same plots without outliers. For the error rate, the Brier score and the logarithmic loss, low values are better, while for the AUC, high values are preferable. If the tuned measure equals the evaluation measure the boxplot is displayed in grey.

We see that the differences are small, although on average all algorithms perform better than the ***ranger default***. The best algorithm is on average by around 0.013 better regarding the error rate (MMCE) and by 0.007 better in terms of the AUC. These small differences are not very surprising as the results by Probst et al. (2018) suggest that random forest is one of the machine learning algorithms that are less tunable.

On average the ***tuneRanger*** methods outperform ***ranger*** with default settings for all measures. Also tuning the specific measure does on average always provide the best results among all algorithms among the ***tuneRanger*** algorithms. It is only partly true if we look at the ranks: ***tuneRangerBrier*** has the best average rank for the error rate, not ***tuneRangerMMCE***. ***caret*** and ***tuneRF*** are on average better than ***ranger default*** (with the exception of the logarithmic loss), but are clearly outperformed by most of the ***tuneRanger*** methods for most of the measures. ***mlrHyperopt*** is quite competitive and achieves comparable performance to the ***tuneRanger*** algorithms. This is not surprising as it also uses **mlrMBO** for tuning like **tuneRanger**. Its main disadvantage is the runtime. It uses 25 iterations in the SMBO procedure compared to 100 in our case, which makes it a bit faster for smaller datasets. But for bigger datasets it takes longer as, unlike **tuneRanger**, it does not use the out-of-bag method for internal evaluation but 10-fold cross-validation, which takes around 10 times longer.

Figure 2 displays the average runtime in seconds for the different algorithms and different datasets. The datasets are ordered by the runtime of the ***tuneRangerMMCE*** algorithm. For most of the datasets, ***tuneRF*** is the fastest tuning algorithm, although there is one dataset for which it takes longer than all the other datasetes. The runtime of ***mlrHyperopt*** is similar to the runtime of ***tuneRanger*** for smaller datasets, but when runtime increases it gets worse and worse compared to the ***tuneRanger*** algorithms. For this reason we claim that **tuneRanger** is preferable especially for bigger datasets, when runtime also plays a more important role.

To examine if tuning only *mtry* could provide comparable results to tuning the parameters *mtry*, node size and sample size all together we run the ***tuneRanger*** algorithms with only tuning *mtry*. The results show that tuning the node size and sample size provides on average a valuable improvement. On average the error rate (MMCE) improves by 0.004, the AUC by 0.002, the Brier score by 0.010 and the logarithmic loss by 0.014 when tuning all three parameters.

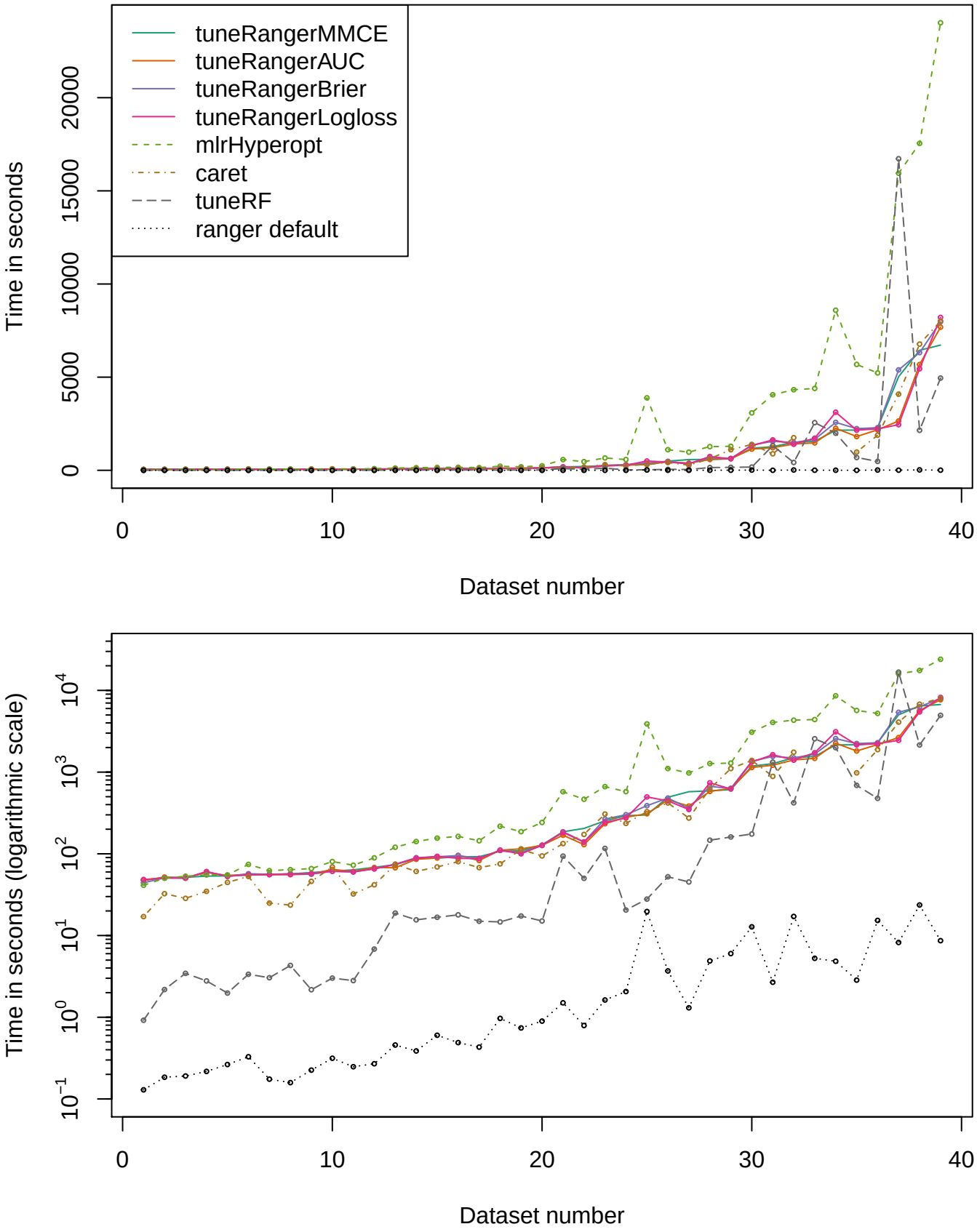

Figure 2: Average runtime of the different algorithms on different datasets (upper plot: unscaled, lower plot: logarithmic scale). The datasets are ordered by the average runtime of the ***tuneRangerMMCE*** algorithm.

4 Conclusion and Discussion

The RF algorithm has several hyperparameters that may influence its performance. The number of trees should be set high: the higher the number of trees, the better the results in terms of performance and precision of variable importances. However, the improvement obtained by adding trees diminishes as more and more trees are added. The hyperparameters *mtry*, sample size and node size are the parameters that control the randomness of the RF. They should be set to achieve a reasonable strength of the single trees without too much correlation between the trees (bias-variance trade-off). Out of these parameters, *mtry* is most influential both according to the literature and in our own experiments. The best value of *mtry* depends on the number of variables that are related to the outcome. Sample size and node size have a minor influence on the performance but are worth tuning in many cases as we also showed empirically in our benchmark experiment. As far as the splitting rule is concerned, there exist several alternatives to the standard RF splitting rule, for example those used in conditional inference forests (Hothorn et al., 2006) or extremely randomized trees (Geurts et al., 2006).

The literature on RF cruelly lacks systematic large-scale comparison studies on the different variants and values of hyperparameters. It is especially scarce as far as the impact on variable importance measures is concerned. This is all the more regrettable given that a large part of the data analysts using RF pay at least as much attention to the output variable importances as to the output prediction rule. Beyond the special case of RF, literature on computational methods tends to generally focus on the development of new methods as opposed to comparison studies investigating existing methods. As discussed in Boulesteix et al. (2018), computational journals often require the development of novel methods as a prequisit for publication. Comparison studies presented in papers introducing new methods are often biased in favor of these new methods—as a result of the publication bias and publication pressure. As a result of this situation, *neutral* comparison studies as defined by Boulesteix et al. (2017) (i.e., focusing on the comparison of existing methods rather than aiming at demonstrating the superiority of a new one, and conducted by authors who are as a group approximately equally competent on all considered methods) are important but rare.

The literature review presented in this paper, which is to some extent disappointing in the sense that clear guidance is missing, leads us to make a plea for more studies investigating and comparing the behaviors and performances of RF variants and hyperparameter choices. Such studies are, in our opinion, at least as important as the development of further variants that would even increase the need for comparisons.

In the second part of the paper, different tuning methods for random forest are compared in a benchmark study. The results and previous studies show that tuning random forest can improve the performance although the effect of tuning is much smaller than for other machine learning algorithms such as support vector machines (Mantovani et al., 2015). Out of existing tuning algorithms, we suggest to use sequential model-based optimization (SMBO) to tune the parameters *mtry*, sample size and node size simultanously. Moreover, the out-of-bag predictions can be used for tuning. This approach is faster than, for example, cross-validation. The whole procedure is implemented in the package **tuneRanger**. This package allows users to choose the specific measure that should be minimized (e.g., the AUC in case of classification). In our benchmark study it achieved on average better performances than the standard random forest and other software that implement tuning for random forest, while the fast **tuneRF** function from the package **randomForest** can be recommended if computational speed is an issue.

References

Belgiu, M. and Drăguţ, L. (2016) Random forest in remote sensing: A review of applications and future directions. *ISPRS Journal of Photogrammetry and Remote Sensing*, **114**, 24–31.

Bergstra, J. and Bengio, Y. (2012) Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, **13**, 281–305.

Bernard, S., Heutte, L. and Adam, S. (2009) Influence of hyperparameters on random forest accuracy. In *MCS*, vol. 5519 of *Lecture Notes in Computer Science*, 171–180. Springer.

Biau, G. and Scornet, E. (2016) A random forest guided tour. *Test*, **25**, 197–227.

Birattari, M., Yuan, Z., Balaprakash, P. and Stützle, T. (2010) F-Race and iterated F-Race: An overview. In *Experimental methods for the analysis of optimization algorithms*, 311–336. Springer.

Bischl, B., Casalicchio, G., Feurer, M., Hutter, F., Lang, M., Mantovani, R. G., van Rijn, J. N. and Vanschoren, J. (2017) OpenML benchmarking suites and the OpenML100. *ArXiv preprint arXiv:1708.03731*. URL: `https://arxiv.org/abs/1708.03731`.

Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., Studerus, E., Casalicchio, G. and Jones, Z. M. (2016) mlr: Machine learning in R. *Journal of Machine Learning Research*, **17**, 1–5.

Bischl, B., Richter, J., Bossek, J., Horn, D., Thomas, J. and Lang, M. (2017) mlrMBO: A modular framework for model-based optimization of expensive black-box functions. *ArXiv preprint arXiv:1703.03373*. URL: `https://arxiv.org/abs/1703.03373`.

Bischl, B., Schiffner, J. and Weihs, C. (2013) Benchmarking local classification methods. *Computational Statistics*, **28**, 2599–2619.

Bohachevsky, I. O., Johnson, M. E. and Stein, M. L. (1986) Generalized simulated annealing for function optimization. *Technometrics*, **28**, 209–217.

Boulesteix, A.-L., Bender, A., Lorenzo Bermejo, J. and Strobl, C. (2012a) Random forest gini importance favours snps with large minor allele frequency: impact, sources and recommendations. *Briefings in Bioinformatics*, **13**, 292–304.

Boulesteix, A.-L., Binder, H., Abrahamowicz, M. and Sauerbrei, W. (2018) On the necessity and design of studies comparing statistical methods. *Biometrical Journal*, **60**, 216–218.

Boulesteix, A.-L., Janitza, S., Kruppa, J. and König, I. R. (2012b) Overview of random forest methodology and practical guidance with emphasis on computational biology and bioinformatics. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, **2**, 493–507.

Boulesteix, A.-L., Wilson, R. and Hapfelmeier, A. (2017) Towards evidence-based computational statistics: lessons from clinical research on the role and design of real-data benchmark studies. *BMC Medical Research Methodology*, **17**, 138.

Breiman, L. (1996) Out-of-bag estimation. *Tech. rep.*, UC Berkeley, Department of Statistics.

— (2001) Random forests. *Machine Learning*, **45**, 5–32.

Casalicchio, G., Bossek, J., Lang, M., Kirchhoff, D., Kerschke, P., Hofner, B., Seibold, H., Vanschoren, J. and Bischl, B. (2017) OpenML: An R package to connect to the machine learning platform OpenML. *Computational Statistics*, **32**, 1–15.

Criminisi, A., Shotton, J., Konukoglu, E. et al. (2012) Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends® in Computer Graphics and Vision*, **7**, 81–227.

Díaz-Uriarte, R. and De Andres, S. A. (2006) Gene selection and classification of microarray data using random forest. *BMC Bioinformatics*, **7**, 3.

Fernández-Delgado, M., Cernadas, E., Barro, S. and Amorim, D. (2014) Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, **15**, 3133–3181.

Ferri, C., Hernández-Orallo, J. and Modroiu, R. (2009) An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, **30**, 27–38.

Genuer, R., Poggi, J.-M. and Tuleau, C. (2008) Random forests: Some methodological insights. *ArXiv preprint arXiv:0811.3619*. URL: `https://arxiv.org/abs/0811.3619`.

Genuer, R., Poggi, J.-M. and Tuleau-Malot, C. (2010) Variable selection using random forests. *Pattern Recognition Letters*, **31**, 2225–2236.

Geurts, P., Ernst, D. and Wehenkel, L. (2006) Extremely randomized trees. *Machine Learning*, **63**, 3–42.

Goldstein, B. A., Polley, E. C. and Briggs, F. (2011) Random forests for genetic association studies. *Statistical Applications in Genetics and Molecular Biology*, **10**.

Grömping, U. (2009) Variable importance assessment in regression: linear regression versus random forest. *The American Statistician*, **63**, 308–319.

Hastie, T., Tibshirani, R. and Friedman, J. (2001) *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc.

Hothorn, T., Hornik, K. and Zeileis, A. (2006) Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, **15**, 651–674.

Hothorn, T. and Zeileis, A. (2015) partykit: A modular toolkit for recursive partytioning in R. *Journal of Machine Learning Research*, **16**, 3905–3909.

Hutter, F., Hoos, H. H. and Leyton-Brown, K. (2011) *Sequential model-based optimization for general algorithm configuration*, 507–523. Berlin, Heidelberg: Springer Berlin Heidelberg.

Janitza, S., Binder, H. and Boulesteix, A.-L. (2016) Pitfalls of hypothesis tests and model selection on bootstrap samples: causes and consequences in biometrical applications. *Biometrical Journal*, **58**, 447–473.

Janitza, S. and Hornung, R. (2018) On the overestimation of random forest's out-of-bag error. *PLOS ONE*, **13**, e0201904.

Jones, D. R., Schonlau, M. and Welch, W. J. (1998) Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, **13**, 455–492.

Kuhn, M. (2008) Building predictive models in R using the caret package. *Journal of Statistical Software*, **28**, 1–26.

Liaw, A. and Wiener, M. (2002) Classification and regression by randomForest. *R News*, **2**, 18–22.

Lin, Y. and Jeon, Y. (2006) Random forests and adaptive nearest neighbors. *Journal of the American Statistical Association*, **101**, 578–590.

Lunetta, K. L., Hayward, L. B., Segal, J. and Van Eerdewegh, P. (2004) Screening large-scale association study data: exploiting interactions using random forests. *BMC Genetics*, **5**, 32.

Mantovani, R. G., Rossi, A. L., Vanschoren, J., Bischl, B. and Carvalho, A. C. (2015) To tune or not to tune: recommending when to adjust svm hyper-parameters via meta-learning. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, 1–8. IEEE.

Martínez-Muñoz, G. and Suárez, A. (2010) Out-of-bag estimation of the optimal sample size in bagging. *Pattern Recognition*, **43**, 143–152.

Oshiro, T. M., Perez, P. S. and Baranauskas, J. A. (2012) How many trees in a random forest? In *Machine Learning and Data Mining in Pattern Recognition: 8th International Conference, MLDM 2012, Berlin, Germany, July 13-20, 2012, Proceedings*, vol. 7376, 154. Springer.

Probst, P. (2017) *OOBCurve: Out of Bag Learning Curve*. R package version 0.2.

— (2018) *tuneRanger: Tune random forest of the 'ranger' package*. R package version 0.1.

Probst, P., Bischl, B. and Boulesteix, A.-L. (2018) Tunability: Importance of hyperparameters of machine learning algorithms. *ArXiv preprint arXiv:1802.09596*. URL: `https://arxiv.org/abs/1802.09596`.

Probst, P. and Boulesteix, A.-L. (2017) To tune or not to tune the number of trees in random forest? *ArXiv preprint arXiv:1705.05654*. URL: `https://arxiv.org/abs/1705.05654`.

Richter, J. (2017) *mlrHyperopt: Easy hyperparameter optimization with mlr and mlrMBO*. R package version 0.0.1.

van Rijn, J. N. and Hutter, F. (2018) Hyperparameter importance across datasets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2367–2376. ACM.

Schiffner, J., Bischl, B., Lang, M., Richter, J., Jones, Z. M., Probst, P., Pfisterer, F., Gallo, M., Kirchhoff, D., Kühn, T., Thomas, J. and Kotthoff, L. (2016) mlr tutorial. *ArXiv preprint arXiv:1609.06146*. URL: `https://arxiv.org/abs/1609.06146`.

Segal, M. R. (2004) Machine learning benchmarks and random forest regression. *Center for Bioinformatics & Molecular Biostatistics*.

Seibold, H., Bernau, C., Boulesteix, A.-L. and De Bin, R. (2018) On the choice and influence of the number of boosting steps for high-dimensional linear cox-models. *Computational Statistics*, **33**, 1195–1215.

Shmueli, G. et al. (2010) To explain or to predict? *Statistical Science*, **25**, 289–310.

Strobl, C., Boulesteix, A.-L., Zeileis, A. and Hothorn, T. (2007) Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics*, **8**, 25.

Vanschoren, J., van Rijn, J. N., Bischl, B. and Torgo, L. (2013) OpenML: Networked science in machine learning. *SIGKDD Explorations*, **15**, 49–60.

Wright, M. N., Dankowski, T. and Ziegler, A. (2017) Unbiased split variable selection for random survival forests using maximally selected rank statistics. *Statistics in Medicine*, **36**, 1272–1284.

Wright, M. N. and Ziegler, A. (2017) ranger: A fast implementation of random forests for high dimensional data in C++ and R. *Journal of Statistical Software*, **77**, 1–17.

Wright, M. N., Ziegler, A. and König, I. R. (2016) Do little interactions get lost in dark random forests? *BMC Bioinformatics*, **17**, 145.

Ziegler, A. and König, I. R. (2014) Mining data with random forests: current options for real-world applications. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, **4**, 55–63.