Hyperparameters, Tuning and Meta-Learning for Random Forest and Other Machine Learning Algorithms

Philipp Probst

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

München, 02.05.2019

Erstgutachter: Prof. Dr. Anne-Laure Boulesteix
Zweitgutachter: Prof. Dr. Bernd Bischl
Drittgutachter: Prof. Dr. Pierre Geurts
Tag der mündlichen Prüfung: 25.07.2019

Summary

In this cumulative dissertation thesis, I examine the influence of hyperparameters on machine learning algorithms, with a special focus on random forest. It mainly consists of three papers that were written in the last three years.

The first paper (Probst and Boulesteix, 2018) examines the influence of the number of trees on the performance of a random forest. In general it is believed that the number of trees should be set higher to achieve better performance. However, we show some real data examples in which the expectation of measures such as accuracy and AUC (partially) decrease with growing numbers of trees. We prove theoretically why this can happen and argue that this only happens in very special data situations. For other measures such as the Brier score, the logarithmic loss or the mean squared error, we show that this cannot happen. In a benchmark study based on 306 classification and regression datasets, we illustrate the extent of this unexpected behaviour. We observe that, on average, most of the improvement regarding performance can be achieved while growing the first 100 trees. We use our new `OOBCurve` R package (Probst, 2017a) for the analysis, which can be used to examine performances for a growing number of trees of a random forest based on the out-of-bag observations.

The second paper (Probst et al., 2019b) is a more general work. Firstly we review literature about the influence of hyperparameters on random forest. The different hyperparameters considered are the number of variables drawn at each split, the sampling scheme for drawing observations for each tree, the minimum number of observations in a node that a tree is allowed to have, the number of trees and the splitting rule. Their influence is examined regarding performance, runtime and variable importance. In the second part of the paper different tuning strategies for obtaining optimal hyperparameters are presented. A new software package in R is introduced, `tuneRanger`. It executes the tuning strategy sequential model-based optimization based on the out-of-bag observations. The hyperparameters and ranges for tuning are chosen automatically. In a benchmark study this implementation is compared with other different implementations that execute tuning for random forest.

The third paper (Probst et al., 2019a) is even more general and presents a general framework for examining the tunability of hyperparameters of machine learning algorithms. It first defines the concept of defaults properly and proposes definitions for measuring the tunability of the whole algorithm, of single hyperparameters and of combinations of hyperparameters. To apply these definitions to a collection of 38 binary classification datasets, a random bot is created, which generated in total around 5 million experiment runs of 6 algorithms with different hyperparameters. The details of this bot are described in an extra paper (Kühn et al., 2018), co-authored by myself, that is also included in this dissertation. The results of this bot are used to estimate the tunability of these 6 algorithms and their specific hyperparameters. Furthermore, ranges for parameter tuning of these algorithms are proposed.

Zusammenfassung

In dieser kumulativen Dissertation untersuche ich den Einfluss von Hyperparametern auf Machine Learning Algorithmen, mit besonderem Fokus auf Random Forest. Sie besteht hauptsächlich aus drei Artikeln, die in den letzten drei Jahren verfasst wurden.

Der erste Artikel (Probst and Boulesteix, 2018) untersucht den Einfluss der Anzahl der Bäume auf die Performance von Random Forest. Im Allgemeinen wird angenommen, dass die Anzahl der Bäume höher eingestellt werden sollte, um eine bessere Performance zu erzielen. Wir zeigen jedoch einige reale Datenbeispiele, in denen der Erwartungswert von Maßen wie Accuracy und AUC (teilweise) mit zunehmender Anzahl von Bäumen abnimmt. Wir beweisen theoretisch, warum dies möglich ist, und argumentieren, dass dies nur bei sehr speziellen Datensituationen vorkommt. Für andere Maße wie den Brier-Score, den Logarithmic Loss oder den mittleren quadratischen Fehler zeigen wir, dass dies nicht möglich ist. In einer Benchmark-Studie, die auf 306 Klassifizierungs- und Regressionsdatensätzen basiert, wird das Ausmaß dieses unerwarteten Verhaltens veranschaulicht. Wir stellen fest, dass im Durchschnitt die meisten Verbesserungen bezüglich der Performance beim Trainieren der ersten 100 Bäume erzielt werden können. Wir verwenden unser neues `OOBCurve` R-Paket (Probst, 2017a) für die Analyse, mit dem die Performance eines Random Forests für eine wachsende Anzahl von Bäumen anhand der Out-of-Bag-Beobachtungen untersucht werden kann.

Der zweite Artikel (Probst et al., 2019b) ist eine allgemeinere Arbeit. Zunächst fassen wir die bestehende Literatur über den Einfluss von Hyperparametern auf Random Forest zusammen. Die verschiedenen betrachteten Hyperparameter sind die Anzahl der Variablen, die bei jedem Split untersucht werden, das Stichprobenschema zum Ziehen von Beobachtungen für jeden Baum, die Mindestanzahl von Beobachtungen in einem Knoten, die ein Baum haben muss, die Anzahl der Bäume und die Regel zum Splitten. Ihr Einfluss auf Performance, Laufzeit und Variable Importance wird geprüft. Im zweiten Teil des Artikels werden verschiedene Tuningstrategien zur Ermittlung optimaler Hyperparameter vorgestellt. Ein neues Softwarepaket in R wird eingeführt, `tuneRanger`. Basierend auf den Out-of-Bag-Beobachtungen führt es das Tuning mittels sequentieller modellbasierter Optimierung durch. Die Hyperparameter und Räume für das Tuning werden automatisch ausgewählt. In einer Benchmark-Studie wird diese Implementierung mit anderen Implementierungen für das Tunen von Random Forest verglichen.

Der dritte Artikel (Probst et al., 2019a) ist noch allgemeiner und erläutert einen allgemeines Framework für die Messung der Tunebarkeit von Hyperparametern von Machine Learning Algorithmen. Zunächst wird das Konzept der Defaults erläutert und Definitionen zur Messung der Tunebarkeit des gesamten Algorithmus, einzelner Hyperparameter und von Kombinationen von Hyperparametern vorgeschlagen. Diese Definitionen werden auf eine Sammlung von 38 binären Klassifizierungsdatensätzen angewendet, anhand eines Bots, der insgesamt rund 5 Millionen Experimente von 6 Algorithmen mit unterschiedlichen Hyperparametern generiert. Details dieses Bots werden in einem von mir als Koautor verfassten Zusatzartikel (Kühn et al., 2018) beschrieben, der ebenfalls in dieser Dissertation enthalten ist. Die Ergebnisse dieses Bots werden zur Abschätzung der Tunebarkeit dieser 6 Algorithmen und ihrer einzelnen Hyperparameter verwendet. Außerdem werden Räume zum Tuning dieser Algorithmen vorgeschlagen.

Acknowledgments

Thanks for discussions and cooperations to Roman Hornung, Silke Janitza, Simon Klau, Raphael Couronné, Riccardo de Bin, Anne-Laure Boulesteix and all the other people that I met at the IBE. Moreover, thanks to Bernd Bischl, Giuseppe Casalicchio, Janek Thomas, Florian Pfisterer, Daniel Kühn, Quay Au, Jakob Richter, Michel Lang, Marvin Wright, Joaquin Vanschoren, Jan van Rijn and Andreas Müller.

Contents

Preface

The human mind is limited in capturing complex patterns. We can easily reason that when X happens Y will happen, or it will happen with a certain probability and save this information in our mind. Slightly more complex patterns are also possible for us to imagine, for example when X and Y happen, then Z will happen. The more complex the connections are, the less easily we are able to envisage the relationships. From a certain point on, it is better to write connections on a paper, because we are not able to memorize everything in our mind and because it is easier to share it with other people. Connections and information can get too complex to write them on paper, which is one of the reasons for the invention of the computer. While connections and information written on paper could still be captured by the human mind, it can be difficult or infeasible to capture them from a computer. In this case simplifications have to be done, so that the complex patterns are filtered and only the most important information is shown to us.

The division of the statistical world into two parts described in the famous article *The two cultures* by Breiman et al. (2001) is strongly related to this problem. In the classical statistical world, relations between variables were designed by hand. The design could be determined by prior knowledge or by using visualization techniques. Growing computational power and the internet have provided more and more data, data storing and computational capacities. This has lead to the development of new algorithmic methods nowadays called machine learning which can handle this big amount of data and can find the complex patterns within the data by themselves with only limited amount of user input. These methods can provide high predictive power but can be more difficult to interpret than models for which the design was created by hand. Interpretation methods, such as partial dependence plots (Friedman, 2001) and individual conditional expectation (ICE) (Goldstein et al., 2015), try to tackle this problem and to make the machine learning algorithms more interpretable.

The machine learning techniques are only used by part of the statistical community but are increasing in popularity, which can, for example, be seen in the download statistics of popular machine learning packages in R (Csardi, 2015). In some cases simple and hand-crafted models such as linear regression can be good enough to model simple relationships between variables. Moreover, the interpretability tools and statistical tests for these simple models are more developed although there are new developments for this in the machine learning community (see, for example, Casalicchio et al., 2018; Molnar, 2019). In many other cases machine learning methods will provide better results and can handle more complex data patterns, for example, in case of non-linear relationships, for datasets with more variables than observations or for image data. Some statisticians still hesitate to use the machine learning methods. One reason for this is, that the methods were partly developed by scientists that are more related to computer science than to statistics. Statistics deals with the collection, organization, analysis, interpretation and presentation of data (Dodge, 2006). Machine learning methods have exactly the same purpose, with a focus on computational methods and automatization. In my opinion, an open minded statistician should be open to any method that is available and use the one that is the most suitable for his or her purpose. The unification of these two fields is nowadays called data science.

The topics of this thesis are based on a principle of machine learning - to automate parts of the data analysis process. A special focus lies on using data for this automatization although in our first paper (Probst and Boulesteix, 2018) we also analyse theoretical properties of the number of trees in a random forest.

Chapter 1

Statistical Learning Methods

In this chapter, the fundamental statistical learning methods that are used in this dissertation are introduced. Statistical learning is defined here as the process of learning from data (Hastie et al., 2001). Statistical learning is used here as a term that emphasizes the statistical side of machine learning although both terms are used interchangeably.

In the following, we will firstly introduce simple measures to summarize variables. Then machine learning algorithms are presented that can detect more complex patterns between the variables. Afterwards model assessment techniques for these algorithms are presented as well as tuning strategies for the hyperparameters. The concepts of automated machine learning and meta-learning are presented in more detail as they are not well known in the community and play an important role in most of the topics discussed in this dissertation.

1.1 Simple Statistics for Summarizing

Statistics started with collecting data. Single data points could be captured by the human mind, larger amounts could not, so adequate techniques had to be applied to make some sense of piles of data. In this section, I will shortly present some of the most classical summary statistics. They can be applied on data that consists of several observations and one or several variables.

1.1.1 Statistical Approach

Univariate methods can be used to summarize properties of the single variables of the collected data. The basic statistic for continuous data is the mean. More robust methods that are not so sensitive against outliers are the trimmed mean, the median, the lower and upper quartiles or other quantiles. For nominal data we cannot calculate these statistics, but we can calculate the absolute and relative frequency of categories and the mode, which is the most frequent category. For ordinal data additionally the median and quantiles can be calculated.

The next natural step is to examine associations between the variables. One naturally starts with examining associations between two variables. The techniques here again differ between continuous, ordinal and nominal data, but are based on the same principles. The linear relationship between two continuous variables can be measured by the Pearson correlation coefficient. More robust against outliers are the rank correlation coefficients Spearman's rho and alternatively Kendall's tau which can also be applied on ordinal variables. These measures can only capture linear (Pearson) or monotonic (Spearman/Kendall) dependencies. To measure other relationships, one can use the distance correlation (Székely et al., 2007) or the information theoretical approach described in Section 1.1.3. For two nominal variables the contingency coefficient can be calculated. For a nominal and a continuous variable we can calculate coefficients such as the point biserial-correlation or a logistic regression. Other approaches are the one-way ANOVA and the Kruskal-Wallis H test statistic.

For measuring the association of more than two variables more sophisticated methods have to be used. If one of the variables is of special interest one can use methods of supervised learning (see Section 1.2), otherwise methods from unsupervised learning such as clustering or principal component analysis can be used to investigate the relationship between the variables.

1.1.2 Visualization Tools

The continuous variables can be visualized in boxplots, histograms or empirical distribution functions. Individual nominal variables can be adequately visualised by barplots or pie charts. For two continuous variables a scatterplot can be used. The association can be further visualized by a representing line, such as a linear regression line, splines or other curves, such as the lowess (Cleveland, 1979). If we have at least one nominal variable, some of the univariate plots mentioned above can be used and the categories can be distinguished by putting the correspondent plots side by side or by using colors or different shapes. For a representation of three or more variables, techniques such as 3D-plots, scatter plots with colors and facetting can be used. With more and more dimensions it will be infeasible to plot the exact value in a way, such that the reader can capture the values of all the variables for each observation. Then reduction methods such as principal component analysis, clustering tools or self-organizing maps can be used.

1.1.3 Information Theory Approach

Another approach for obtaining information on variables is the information theory approach established by Shannon (1948). In the information theoretic approach, no distribution assumption is made beforehand and also no assumption about the functional relationship between variables. In the following, we will assume discrete distribution of variables, although these measures can be analogously calculated for continuous variables. For a single discrete variable X on the sample space $\mathbb{X}$, one can calculate the entropy that measures the amount of uncertainty of a variable when only its distribution is known:

$$H(X) = \mathbb{E}_X[-\log(X)] = -\sum_{x\in\mathbb{X}} p(x)\log p(x). \tag{1.1}$$

For combinations of variables, several measures can be calculated. The joint entropy of two discrete variables X and Y is defined as:

$$H(X,Y) = \mathbb{E}_{X,Y}[-\log p(x,y)] = -\sum_{x\in\mathbb{X},y\in\mathbb{Y}} p(x,y)\log p(x,y). \tag{1.2}$$

The conditional entropy measures the entropy of one variable given another variable,

$$H(X|Y) = \mathbb{E}_Y[H(X|y)] = -\sum_{y\in Y} p(y)\sum_{x\in X} p(x|y)\log p(x|y) = -\sum_{x\in\mathbb{X},y\in\mathbb{Y}} p(x,y)\log p(x|y), \tag{1.3}$$

which can also be written as $H(X|Y) = H(X,Y) - H(Y)$. Strongly related to this is the mutual information, that measures the information that can be obtained about one random variable when another one is given:

$$I(X;Y) = \mathbb{E}_{X,Y}\left[-\log\frac{p(x,y)}{p(x)\,p(y)}\right] = \sum_{x\in\mathbb{X},y\in\mathbb{Y}} p(x,y)\log\frac{p(x,y)}{p(x)\,p(y)}. \tag{1.4}$$

It can be interpreted as the difference of the conditional entropy and the entropy, and it is symmetrical in the two variables:

$$I(X;Y) = H(X) - H(X|Y) = H(X) + H(Y) - H(X,Y) = I(Y;X). \tag{1.5}$$

Simon and Tibshirani (2014) showed in a comparison study that the mutual information has less power than the distance correlation to detect associations in several data situations. A disadvantage of the distance correlation is the low speed of calculation in case of many observations, so faster calculation methods were proposed recently by Chaudhuri and Hu (2019).

The Kullback-Leibler divergence is another measure that can be used to compare two distributions. Given a true probability distribution and another arbitrary probability distribution, it measures the unnecessary surprise introduced when using the arbitrary probability distribution $q(x)$ instead of the true probability distribution $p(x)$:

$$D_{\mathrm{KL}}(p(X)\|q(X)) = \sum_{x\in X} -p(x)\log q(x) - \sum_{x\in X} -p(x)\log p(x) = \sum_{x\in X} p(x)\log\frac{p(x)}{q(x)}. \tag{1.6}$$

It is not symmetrical in the two variables.

Currently also tests are developed for information theoretical measures, for example, for the mutual information (Berrett and Samworth, 2017).

1.2 Supervised Learning

Supervised learning is defined as the machine learning task of using some input variables to predict one or several outputs (Hastie et al., 2001). Other denotations for the inputs are predictors, independent variables, covariates or features and responses or dependent variables for the outputs. I will use the terms interchangeably as done in Hastie et al. (2001).

Depending on the outputs, there are different denominations for the different tasks. If we have a continuous output, the task is typically denoted as regression, for a nominal output it is called classification. In the case of several quantitative outputs, it is called multivariate regression, in the case of several qualitative outputs, multilabel classification. If it is a mix of both it is called multitarget prediction. In contrast, in unsupervised learning there is not a known target output. Throughout this dissertation I will focus on methods for supervised learning coming from the area of machine learning. In the following I will shortly describe the best known ones.

Algorithms for Supervised Learning

Simple statistical methods such as the linear model need, in general, user input and adjustments to perform well. For example, the functional relationship between two variables has to be specified beforehand properly, which can only be done by using domain knowledge or graphical analysis. Moreover, it is important to exclude non-important variables beforehand. A linear model is, for example, also not usable if there are more predictors than observations. In contrast, advanced machine learning methods are meant to learn patterns from data automatically, without needing a lot of user input. They usually do not rely on any fixed distribution assumption beforehand. In the following I will shortly present some of the most well known machine learning techniques for supervised learning. They all can be used for regression or classification. For a more indepth introduction into the methods I refer to common books of machine learning (for example, Hastie et al., 2001).

A decision tree, is built by taking the observations and recursively performing the split of one of the input variables that provides the best split for getting homogenous groups regarding the outcome variable. Homogenity is usually quantified by using measures such as the Gini impurity, in the case of classification, or the residual sum of squares, in the case of regression. This leads to trees that split the observations into homogenous groups regarding the outcome variable while taking into account dependencies between the other variables. See Breiman et al. (1984) for more details about decision trees.

Usually these single trees do not provide very good predictions, because they overfit the given data. An intelligent way to improve this is to introduce some randomness into the single trees to get distinct trees and to aggregate these trees. A well-established technique for this is random forest (Breiman, 2001) for which many random trees are generated and aggregated. The randomness is introduced in two ways. Firstly, not all observations, but only a sample is used to construct each tree. Secondly, not all variables but only a random subset are considered as split candidates for each split. The prediction for an observation is done by obtaining a prediction of each tree and finally aggregating the predictions through majority voting (for classification) or by taking the average of all predictions (for regression). The strength of the aggregation principle can be explained by the bias-variance trade-off (Louppe, 2014). The variance of the prediction of a single tree is reduced significantly by aggregating random trees while the bias of the single trees is ideally not much higher than the bias of a standard decision tree.

Another successful technique is gradient boosting (Freund and Schapire, 1996; Friedman, 2001), which tries to improve base learners in a different way. After having trained a base learner, e.g. a decision tree, the algorithm evaluates how well each observation can be predicted with the base learner. For learning the next base learner the observations that were badly predicted obtain a bigger weight. This is repeated in an iterative way by always giving a higher weight for observations that perform bad on the current set of base learners.

Another flexible and simple technique, which is neither directly related to trees nor to linear models, is k-nearest neighbors (Altman, 1992). To make a prediction for a new observation it just calculates the distance (e.g., the euclidean distance or some other Minkowski distance) of the input variables of the new observation to the input variables of the observations in the training set. Then the k nearest points according to this distance are taken and used for the prediction, for example by taking the majority vote or the (weighted) average of these points. This technique does not rely on any distribution assumption and can take into account very local and complicated relationships between the variables. If k is chosen too small, it is prone to overfitting, if k is too big it cannot identify local structures. Therefore, k should be chosen and tuned carefully, for more information about tuning see chapter 1.4.1.

A more complex machine learning method for supervised learning is the support vector machine (SVM) (Cortes and Vapnik, 1995). It divides data by a so called hyperplane. Roughly speaking, the hyperplane is specified so that the observations of different classes are lying on different sides of the hyperplane and the distance of the hyperplane to the nearest observations is maximized. For given data, the division of the observations of the different classes by the hyperplane is not always possible. Then so called slack variables can be introduced to weaken this restriction. Moreover, the input variables can be mapped into high-dimensional feature spaces by a so called kernel function in order to make the separation of observations of different classes easier. SVMs are also applicable to regression analysis by reformulating the problem for the regression case (Drucker et al., 1997).

Another popular method is elastic net (Zou and Hastie, 2005), which can be seen as an automic variable selection method combined with a linear model. It is a mix of the lasso and ridge regression. In lasso and ridge regression, a linear model is fitted with a penalty term that penalizes the size of the coefficients which leads to the shrinkage of the coefficients. The penalization function is the sum of the absolute values of the coefficients in case of lasso and the sum of the quadratic values of the coefficients for the ridge regression. The penalty coefficient in this penalization function has to be tuned carefully to obtain good performance. If set to zero we obtain a simple linear regression for continuous output, and a multinomial logistic regression for nominal output. Elastic net combines the two methods by using a weighted sum of these penalization functions, the weighting factor is a hyperparameter. The optimization problem of elastic net can be transformed to a support vector machine problem (Zhou et al., 2015), which shows that these two algorithms are closely related.

1.3 Model Assessment in Supervised Learning

Models for supervised learning can be evaluated in different ways. Usually we need two components: one or several performance measures and a resampling strategy.

1.3.1 Performance Measures

Performance measures aim at summarizing the appropriateness of a model for a specified task by quantifying the quality of the model. Usually true values are compared to predicted values by using an appropriate measure. There are different performance measures for different supervised learning tasks and for different purposes.

For regression, the most common measure is the mean squared error. A possibility to scale this measure between 0 and 1 is the R-squared, which divides the mean squared error by the mean squared error that would be achieved by simply taking the mean. Other measures that are based on ranks instead of absolute values, and thus are more robust, are Spearman's rho and Kendall's tau. They compare the predicted and the true response by applying these rank correlation measures. For classification, the most common measure is the mean missclassification error. A measure that takes into account the probabilities of a classifier is the area under the ROC curve (AUC), for which also multiclass variants exist (Ferri et al., 2009). Other measures that take into account probabilities are the Brier score or the logarithmic loss (Ferri et al., 2009). Another thing that can be measured while running the algorithm is the execution time for training and predicting observations.

In the course of this PhD project, several of the above mentioned measures were implemented in the `mlr` R package (Bischl et al., 2016) with the help of other contributors of `mlr`. The measures were also extracted to an external R package called `measures` (Probst, 2018b), which make them easier to use for non-`mlr` users and without relying on `mlr` and other packages. For example, in the package `varImp` (Probst, 2018c) the variable importance of conditional random forests of the `party` R package (Hothorn et al., 2006) can be calculated for any measure that is available in `measures`. The use of measures other than the standard measure accuracy for assessing the variable importance (VIMP) was proposed by Janitza et al. (2013), who showed that AUC-based permutation VIMP is more robust against class imbalance than accuracy-based permutation VIMP. The use of other measures may yield advantages in other specific situations.

A further topic that was adressed in the course of this PhD project was the implementation of survival measures in `mlr`. Survival analysis in `mlr` was implemented by Michel Lang in his dissertation (Lang, 2015). Additionally to the already existing concordance index (`cindex` in `mlr`), Uno's concordance index (`cindex.uno`) (Uno et al., 2011), Uno's estimator of cumulative/dynamic AUC for right-censored time-to-event data (`iauc`) (Uno et al., 2007) and the integrated Brier score (`ibrier`) (Mogensen et al., 2012) were added to the `mlr` package in cooperation with Michel Lang and Moritz Herrmann. For the integrated Brier score, probability predictions for given time points have to be provided and therefore `mlr` was modified to provide probability predictions for survival models.

1.3.2 Resampling Strategy

The simplest approach for calculating the performance of a model is to train the model on a dataset and then compare the predictions of this model with the real response by using a performance measure. The problem with this technique is that it does not take into account the overfitting on the trained data. A perfect classifier in this sense would just predict the true values, but that does not guarantee that the predictions on new data would be good. To avoid this bias due to overfitting, resampling strategies are used. A very common resampling strategy is k-fold cross-validation. The data is divided into k folds, and iteratively the model is trained on all of the folds except of one, which is used for evaluation. So the evaluation happens on a part of the data, that was not used for training and overfitting does not influence the evaluation. This procedure is repeated k times and a summary measure such as the average can be calculated.

The k-fold cross-validation should be repeated especially for small datasets to diminish the variance of the estimates of the performance measures that originates from the random splitting into k folds (Kim, 2009; Bischl et al., 2012).

1.3.3 Graphical Analysis

Instead of reducing the predictive power to one or several numbers, it is also possible to use graphical tools to evaluate the appropriateness of a model. In classical linear regression, one could inspect the residuals that should be normally distributed by using a QQ-Plot. Moreover, homoskedasticity can be examined by plotting the fitted values and the residuals in a scatterplot. Such tools are usually not used for more complex methods (for example, tree-based methods), as the models are quite flexible and no fixed distribution assumption is made beforehand.

The performance results of the used resampling strategy of different learners can be compared with graphical tools such as boxplots. Furthermore, calibration plots can be used in classification to evaluate if the predicted probabilities match the true relative frequency of observations in a certain probability interval. Receiver operating characteristic (ROC) curves can be used in binary classification to compare the true positive rate and the false positive rate combinations that are possible when varying the probability threshold for classifying the observations as positive or negative.

1.3.4 Runtime

Runtime of machine learning algorithms is a factor that is not extensively discussed in the literature but plays a more and more important role in the era of growing datasets. The runtime for training and predicting is a random variable that can be measured by a resampling strategy. It of course depends also on external factors such as the computing power. A good algorithm should be trained in as short time as possible. Also the predictions should be provided as fast as possible by the algorithm.

There are different possibilities to optimize the runtime. Some algorithms have the possibility to run in parallel on several CPU cores or on other distributed systems, for example on servers. A typical example is the random forest, for which the trees are independent of each other and can be trained in parellel as done in the `ranger` package (Wright, 2016). Of course, there are other more sophisticated examples of runtime optimization, such as the implementation of the `xgboost` package (Chen and Guestrin, 2016). A general possibility to reduce the runtime is to train the algorithm only on a small part of the data. Then iteratively the amount of data can be increased while plotting a so-called learning curve, that shows the performance of the algorithm (calculated for example on the out-of-bag data) for different sample sizes. With the help of this curve it is possible to observe, if bigger amount of data leads to an improvement of the algorithm. A stopping criteria can be defined with the help of this curve, for example by stopping the iterative process of feeding the algorithm with more data when a certain convergence criteria has been reached.

Another nice option for users is an input parameter for the algorithms to restrict the time before starting the algorithm. Intelligent systems have to be built to take this extra parameter into account.

1.3.5 Interpretability

Interpretability of a model can be hardly measured quantitatively but is an important aspect for assessing a model. Often users are not only interested in receiving a well performing model but also in a model that is interpretable, for example, by being able to tell that one variable influences another variable in the model, how strong the influence is and in which functional relationship the variables are related to each other.

In linear models, for example, statistical tests can provide clear-cut guidance if a variable has a significant influence on another and how strong this influence is. The coefficients in these models are easily interpretable. Or in a simple tree, the tree's decisions are easy to understand and to grasp for the humand mind. For other more complex methods the interpretation is harder, although, for example, specialized tools exist, such as the variable importance for random forest. The problem is getting more attention recently and model agnostic tools as described for example in Molnar (2019) are getting more attention.

1.3.6 Runtime, Performance and Interpretability

The three main parts that should be used to choose and evaluate a machine learning algorithm is the triumvirate of runtime, performance and interpretability. Depending on the user, the importance weight of each of these three factors is different.

For some users runtime is very important because results have to be obtained fast or because the computational resources are more restricted. Some users want to optimize the prediction performance to the last decimal, for example because the model is used in a business environment and better performance can provide immediate economical benefits or can save lives in a medical context. For other users the interpretability of the models is very important, because, for example, the underlying causal relationship is of interest and the obtained knowledge is not immediately used for a specific target, but for writing a publication. This contributes to the scientific community and the obtained knowledge can be used by other researchers in an iterative way. Or it can be applied in practice by developing guidelines or writing this knowledge in books, so that this knowledge is transmitted in a more automated way. The pipeline in this case is much longer although the final target can sometimes also be a good prediction in a less automated way.

Between the three targets runtime, performance and interpretability there is a trade off. Usually more complex methods lead to higher runtime while providing better performance. Well performing models can possibly better represent the underlying structure of the data and hence provide better insights than worse performing models. On the other hand, as described in Section 1.3.5, simpler models can possibly be easier to interpret. A possible solution for this dilemma is to use several different models for the different purposes.

1.4 Hyperparameters

Hyperparameters are parameters that have to be set before executing a machine learning algorithm, as opposed to normal parameters of an algorithm that are not fixed before execution but optimized while training the algorithm. For all of the machine learning methods described in Section 1.2 there are hyperparameters that have to be set beforehand. Some examples are the number of variables that are regarded in each split in a random forest, the number of boosting steps in gradient boosting, the number k in k-nearest neighbors, the kernel in support vector machines and the weighting of lasso and ridge regression in elastic net.

1.4.1 Defaults and Hyperparameter Tuning

Usually, there are default hyperparameters given in the software packages. For a given problem, they possibly provide good results, if they are set appropriately. Most of the time, tuning the hyperparameters, that means finding an optimal value for them, can provide better performance than using the default value. For some of the algorithms, for example for the support vector machine, tuning is important and setting their hyperparameters to optimal values can provide big performance gains. The tunability of the different algorithms and hyperparameters is defined and discussed in more detail in Probst et al. (2019a). Also, the hyperparameter space on which the tuning should be executed is an important topic in this paper.

1.4.2 Tuning Strategies

For tuning hyperparameters, different strategies can be applied. Ideally, the used strategy should find the best possible hyperparameter values and it should find them in the shortest possible amount of time. Usually, an evaluation method is used to compare different hyperparameters. The most common one is cross-validation. For evaluating the whole algorithm, including the tuning procedure, a nested cross-validation has to be performed.

A simple tuning strategy that can be used is the grid search. For each hyperparameter a finite amount of possible values have to be defined and all possible hyperparameter combinations are evaluated. Another simple strategy is the random search. The hyperparameters are drawn randomly from a given hyperparameter space, for example, by using the uniform distribution. For neural networks Bergstra and Bengio (2012) show that random search is more efficient in finding good hyperparameter values than grid search. Other designs, such as the latin hypercube (Park, 1994) or sobol sequences (Sobol, 1976) can be used that determine all hyperparameter specifications in advance that should be evaluated.

Other more sophisticated approaches determine the hyperparameter specifications iteratively. A typical example for this is bayesian optimization (Hutter et al., 2011; Snoek et al., 2012; Bischl et al., 2017), also called model-based optimization. In bayesian optimization a surrogate model is trained with the performances of already run hyperparameters as output and the hyperparameters as input. With the help of this surrogate model new hyperparameter specifications are proposed that fulfill two requirements: they should provide good results according to the trained surrogate model and they should lie in regions of the hyperparameter space that are still unexplored. In practice, these two goals are combined by an infill criterion, also called acquisition function. For a given hyperparameter the mean and the standard deviation of the performance can be estimated via the surrogate model and combined, for example with weighting factors in the acquisition function (Bischl et al., 2017). The next hyperparameter specification is chosen as the value which provides the best infill criterion, which means optimizing the acquisition function. The estimations of the surrogate model are cheap and simple, so search methods such as branch and bound (Jones et al., 1998) or focus search (Bischl et al., 2017) can be used for the optimization.

Other common procedures are based on gradient-based optimization techniques that calculate the gradient of the hyperparameters and then use gradient descent search methods to find the optimal value. Examples can be seen in Larsen et al. (1996) for neural networks, in Chapelle et al. (2002) for support vector machines and in Foo et al. (2008) for logistic regression.

Claesen and Moor (2015) mention some other strategies, that include swarm algorithms such as particle swarm optimization (Lin et al., 2008; Meissner et al., 2006), evolutionary algorithms such as genetic algorithms (Tsai et al., 2006), simulated annealing (Xavier-de Souza et al., 2010) and racing algorithms (Birattari et al., 2010).

The tuning algorithm can also be tailored especially for a specific method, as presented in Chen et al. (2017) for SVM or in Probst et al. (2019b) for random forest.

1.5 Automatic Machine Learning

Automatic machine learning is the automation of the whole machine learning process from getting data, to obtaining the desired results.

1.5.1 Input for Automatic Machine Learning

Typical automatic machine learning algorithms nowadays take at least two inputs: data and the target of the data scientist. These two inputs cannot be obtained automatically, although default targets (e.g., certain performance measures for a certain task) can be given by the algorithm. The data may need to be transformed to be in the correct format for the algorithm. In several implementations it is also possible to restrict the runtime beforehand.

Target Definition

As described in Section 1.3 different targets can be achieved with different models. In automatic machine learning the main target is usually to get a good performance, measured by an appropriate performance measure that can be chosen by the user. As the performance measure can be measured quantitatively the optimization of this target is usually easier than, for example, to optimize the interpretability. The relationship between the targets is described in more detail in Section 1.3.6.

Interplay between Data Generation and Target Definition

Data can be generated before formulating the target or afterwards. We visualize the two different paths in Figure 1.1. When the target is obtained while examining the data, it is called exploratory data analysis (Tukey, 1977). Certain steps of this analysis can also be automatized, for example, standardized visualizations and summaries that can be well received by the human brain and can lead to new ideas. Of course, issues such as data dredging have to be considered here (Smith and Ebrahim, 2002). Standardized tools for data exploration such as the R package `DataExplorer` (Cui, 2018) facilitate the analysis.

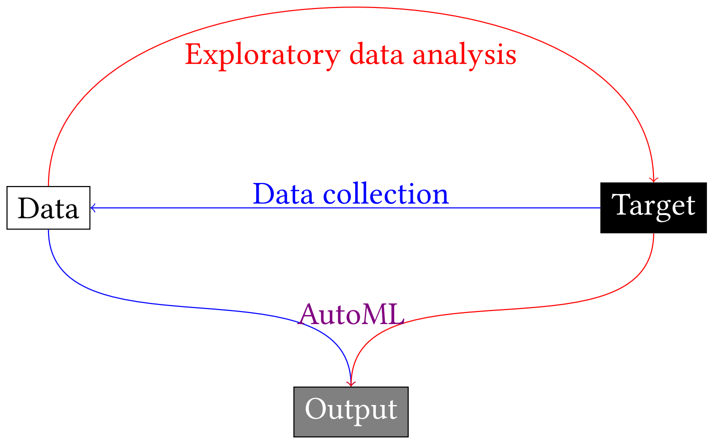

Figure 1.1: Overview of process of automatic machine learning: the red path is chosen when data is given first, the blue path when the target is given first.

Starting the Algorithm

After having obtained the data and the target, the automatic machine learning algorithm can be started. It finally should provide results and information about whether the target can be reached and how well. All the mentioned steps are not fixed - several loops through the process could be necessary to satisfy the needs of the user.

1.5.2 Typical Steps, Implementations and Ideal Design

Typical steps that are included in the automatic machine learning process are data-preprocessing steps such as imputation of missing values, normalization, feature creation and feature selection and transformation of specific variable classes (for example, from multiclass to binary vectors). Then the main steps are the application, evaluation and tuning of different machine learning algorithms, choosing the best of these algorithms and building an ensemble of these algorithms, for example to get the best prediction or to get a good and simple model that is easily interpretable.

As the whole process is usually very time consuming, automatic machine learning algorithms can include parameters for restricting the time beforehand and a solution should be provided in time. Moreover, it is desirable that the algorithm could be restarted building up on the current solution.

Some already existing implementations are `Auto-Weka` (Thornton et al., 2013), `autosklearn` (Feurer et al., 2015), `H2O AutoML` (The H2O.ai team, 2017) and `TPOT` (Olson et al., 2016).

How to ideally design such an algorithm? An ideal implementation works well for the problem at hand. The no free lunch theorem (Wolpert, 2002) states that any two optimization algorithms are equivalent when their performance is averaged across all possible problems. But the problems that usually arise in machine learning applications are not completely randomly chosen, but follow certain patterns. So certain algorithms will usually (e.g., on average) work better than others (Vanschoren, 2018). Therefore, a reasonable option for designing new algorithms is based on results on other datasets which is known as meta-learning and which is described in detail in the following section.

1.6 Meta-Learning

Meta-Learning is an important step in the automation of machine learning. There are different possible definitions and approaches for meta-learning (Brazdil et al., 2008; Lemke et al., 2015; Vanschoren, 2018).

1.6.1 Definitions

The first definition for meta-learning was given by Brazdil et al. (2008):

> “
> Metalearning is the study of principled methods that exploit meta-knowledge to obtain efficient models and solutions by adapting machine learning and data mining processes.
> ”

Another more recent definition from Lemke et al. (2015) emphasizes, that information of the current data can also be included:

> “
> 1. A meta-learning system must include a learning subsystem, which adapts with experience.
> 2. Experience is gained by exploiting metaknowledge extracted
>   (a) ...in a previous learning episode on a single dataset, and/or
>   (b) ...from different domains or problems.
>
> ”

The definition in 2a) contains also approaches such as boosting (Freund and Schapire, 1995), stacked-generalization also called stacking (Wolpert, 1992) or cascade generalization (Gama and Brazdil, 2000) while, for example, bagging (Breiman, 1996) does not fullfill point 2, as it does not learn from previous runs (Lemke et al., 2015). In this thesis I will mainly focus on approaches as defined in 2b), which means including information of algorithm runs on other datasets for the current dataset.

Another definition of meta-learning is given by Vanschoren (2018) in his overview paper:

> “
> Meta-Learning, or *learning to learn*, is the science of systematically observing how different machine learning approaches perform on a wide range of learning tasks, and then learning from this experience, or *meta-data*, to learn new tasks much faster than otherwise possible.
> ”

He emphasizes the application of *machine learning approaches* on *tasks*, the use of *meta-data* and the *speed* as an important factor in this process.

In principle, meta-learning can be applied to anything that the algorithm consists of. Every decision that is taken within an algorithm can be parametrized and, on each of these parameters, meta-learning can be applied. For example, the decision of which algorithm to choose, hyperparameters of the algorithm that have to be set before executing the algorithm or parameters that are usually optimized in the training process of the algorithm, can be set via meta-learning. This can even include the decision if the parameter has to be optimized at all. In the following, I will describe some distinctions that can be made to distinguish different meta-learning methods.

1.6.2 Distinction of Meta-Learning Methods

In his overview paper Vanschoren (2018) differentiates between different types of meta-learning, depending on which information from the current task (e.g., a regression task on a specific dataset) is used. The same distinction is made by Luo (2016). In the following, I will give a brief overview of this differentiation. The focus lies on the setting of the hyperparameters and on the definition of meta-learning that was given by Lemke et al. (2015) in 2b), that means using information of other tasks for the current task.

1.6.3 Task Independent Methods

The simplest meta-learning technique is to transfer knowledge from other tasks without using information of the current task. For example, we can make task independent recommendations for hyperparameters. This can be one so-called *default hyperparameter setting* (Probst et al., 2019a) or also several and possibly ranked fixed hyperparameter settings (Wistuba et al., 2015; Pfisterer et al., 2018) that could be sequentially evaluated.

A more complex transfer of information is called *configuration space design* by Vanschoren (2018), which includes information about previous model evaluations on other datasets but does not use any information from the dataset at hand. For example, these model evaluations can be used to construct a fast and good performing tuning algorithm for a given algorithm, by, for example, specifying which hyperparameters should be tuned and on what hyperparameter space should be searched for the ideal solutions (van Rijn and Hutter, 2017; Probst et al., 2019a; Weerts et al., 2018) possibly also by transforming and reparametrizing hyperparameters (Probst et al., 2019a).

1.6.4 Task Dependent Methods

When information about the current dataset is given, more sophisticated methods could be used to combine them with the information of other datasets. There are two different kinds of information of the current dataset that can be used - either just information about model evaluations for specific hyperparameters on the current dataset - or so-called *task properties.*

Using Information from Model Evaluations

Using information from model evaluations of the current dataset can be done in different ways. The different methods presented here are described in more detail in Vanschoren (2018). *Relative landmarks* are the performance differences of model evaluations on the current and on different datasets (Fürnkranz and Petrak, 2001). In *active testing* these differences are used to iteratively choose a new hyperparameter in each round which performs best on the most similar datasets (Leite et al., 2012). Instead of using only differences, surrogate models can be used to measure similarity and choose new hyperparameter settings, for an overview of different methods, see Vanschoren (2018). In multi-task learning, a joint task representation is learned by combining the results of the single tasks or surrogate models into one big model that is used to predict the accuracy of hyperparameter settings on the new dataset (Vanschoren, 2018). Other techniques described in Vanschoren (2018) include multi-armed bandits and Thompson sampling. Learning curves describe the development of the performance after each iteration step in the learning process. The curve on a new task can be predicted by finding similar learning curves on other tasks. For more details on the literature see Vanschoren (2018). Iterative tuning strategies as discussed in Section 1.4.2 also fall into this category.

Using Task Properties

Another common approach when trying to transfer knowledge of other datasets is using task properties, also called *meta-features*. Typical *meta-features* are the number of observations n and the number of features p or the number of classes in a classification task. Many more *meta-features* can be constructed, see Rivolli et al. (2018) for a detailed overview of meta-features.

With the help of these meta-features, one can measure the similarity to other tasks and propose promising hyperparameter settings according to this. A very simple technique, that is used in many software packages, is using simple functions depending on meta-features for setting the hyperparameters. Prominent examples are the settings $\sqrt{p}$ for *mtry* (number of variables regarded in each split) in random forest or $1/p$ for *gamma* in support vector machines. These functions are usually created by hand by observing for example performance curves of hyperparameter settings on different datasets. A more sophisticated approach for getting simple symbolic functions automatically via meta-learning is described in van Rijn et al. (2018).

Meta-models are more sophisticated models that take meta-features as input and recommend hyperparameters that are expected to provide good performance. These models can be trained by using previously obtained performances of different hyperparameters on other datasets. They can predict the performance and provide a task dependent ranking of promising hyperparameter settings. More literature regarding this topic is given by Vanschoren (2018).

1.6.5 Overview

To have an overview, in Table 1.1 selected articles that describe meta-learning techniques for supervised machine learning techniques are classified by their application purpose (meta-learning for simple hyperparameter setting or for tuning). Other articles that are more complex and also have different purposes are described in Vanschoren (2018) and Brazdil and Giraud-Carrier (2018).

Hyperparameter Setting		
Article	Method	Task dependent
Probst et al. (2019a)	Defaults calculation	no
Weerts et al. (2018)	Defaults calculation, symbolic defaults	no/yes
Pfisterer et al. (2018)	Ranked list of defaults	no
van Rijn et al. (2018)	Symbolic defaults	yes
Tuning		
Article	Method	Task dependent
van Rijn and Hutter (2017)	Tuning importance, tuning priors	no
Probst et al. (2019a)	Tuning importance, tuning space	no
Weerts et al. (2018)	Tuning importance	no
Wistuba et al. (2015)	List of defaults for warm start tuning	yes
Feurer et al. (2015)	List of defaults for warm start tuning	yes

Table 1.1: Selected articles regarding meta-learning for machine learning algorithms ordered by topics.

1.7 Calibrating and Choosing an Algorithmic System

There is a multitude of methods for setting hyperparameters, tuning, automatic machine learning and meta-learning as described above. Which of these methods should be used? The three main parts that should be used to evaluate an approach is the triumvirate of speed, performance and interpretability as described in Section 1.3. Following Occam's razor simpler models are generally prefered over more complex models, so a complex pipeline has to show substantial advantage over simpler approaches. In addition more complex systems will generally take more time than simpler ones.

Every step that is used in an algorithmic pipeline has to be chosen carefully. The applied methods in the pipeline, should work on a broad range of problems, such that a new upcoming problem should be solved adequately. The optimization of the general pipeline structure can be done by task independent meta-learning that means that the chosen steps should work well on many datasets and they should be robust against *outlier* datasets. New approaches have to prove their usefulness and should be compared in fair comparison studies in many different data situations. The usual presentation of the superiority of a method over another by using just a handful of datasets is no longer sufficient today, unless clear superiority can be shown theoretically. Open science platforms such as OpenML (Vanschoren et al., 2013) facilitate the access to a plentitude of datasets and standardized benchmarking suites such as the OpenML100 (Bischl et al., 2017) and PMLB (Olson et al., 2017) facilitate the selection of datasets.

The pipeline could also incorporate task dependent meta-learning. For example, the selection of the performed steps may be based on meta-features. Or the tuning algorithm (which is part of the AutoML pipeline) could be warm-started by using hyperparameter settings that worked well on similar datasets as proposed by Feurer et al. (2015). For simple problems a simple model can be good enough without complex pipeline steps. For more complex datasets with a lot of observations and variables that are possibly also grouped, with influential and non-influential variables, with linear and non-linear relationships more sophisticated approaches are necessary.

In general, meta-learning can be applied to all steps of a machine learning pipeline. Task independent meta-learning techniques are easier to implement as they do not require information of the dataset at hand. A standardized automatic machine learning approach can serve as fast solution or also as benchmark that can be compared to a manual approach. Ideally the performed steps in the algorithmic pipeline should be clearly visible, reproducible and also changeable, for example via hyperparameters.

1.8 Hyperparameter Setting in Practice

For the purpose of setting the hyperparameters ideally (for example, regarding the performance or the runtime) for a certain task there are different ways to use the techniques described above in practice:

- The most subtle one is probably the information that a user has saved in his mind by using an algorithm several times on other datasets with different hyperparameters and applying this knowledge on a new dataset by, for example, trying out certain hyperparameter settings that worked well in the past.
- Alternatively, the user can search for publications that have analysed the behaviour of the algorithm for different hyperparameter settings. In our publication Probst and Boulesteix (2018) (Chapter A), for example, we investigate in detail the influence of the number of trees on a random forest. Moreover, in the publication Probst et al. (2019b) (Chapter A) we wrote a review for all random forest hyperparameters where we summarize the information that we could find in available literature.
- This knowledge transmission can be automatized in a certain way:
  - A software maintainer can fix one hyperparameter setting as default in his software package or provide a list of possible hyperparameter settings.
  - Specific automatic machine learning algorithms that could include different steps, such as data preprocessing, tuning, ensemble methods and meta-learning could be implemented in software packages. Ideally the steps should be choosable and changeable by the user.

1.9 Structure of this thesis

In this cumulative dissertation thesis, I examine the influence of hyperparameters on machine learning algorithms, with a special focus on random forest. It mainly consists of three papers that were written in the last three years.

The first paper (Probst and Boulesteix, 2018) in Section A.1 in the Appendix examines the influence of the number of trees on the performance of a random forest. In general it is believed that the number of trees should be set higher to achieve better performance. However, we show some real data examples in which the expectation of measures such as accuracy and AUC (partially) decrease with growing numbers of trees. We prove theoretically why this can happen and argue that this only happens in very special data situations. For other measures such as the Brier score, the logarithmic loss or the mean squared error, we show that this cannot happen. In a benchmark study based on 306 classification and regression datasets, we illustrate the extent of this unexpected behaviour. We observe that, on average, most of the improvement regarding performance can be achieved while growing the first 100 trees. We use our new `OOBCurve` R package (Probst, 2017a) for the analysis, which can be used to examine performances for a growing number of trees of a random forest based on the out-of-bag observations.

The second paper (Probst et al., 2019b) in Section A.2 is a more general work. Firstly we review literature about the influence of hyperparameters on random forest. The different hyperparameters considered are the number of variables drawn at each split, the sampling scheme for drawing observations for each tree, the minimum number of observations in a node that a tree is allowed to have, the number of trees and the splitting rule. Their influence is examined regarding performance, runtime and variable importance. In the second part of the paper different tuning strategies for obtaining optimal hyperparameters are presented. A new software package in R is introduced, `tuneRanger`. It executes the tuning strategy sequential model-based optimization based on the out-of-bag observation. The hyperparameters and ranges for tuning are chosen automatically. In a benchmark study this implementation is compared with other different implementations that execute tuning for random forest.

The third paper (Probst et al., 2019a) in Section A.3 is even more general and presents a general framework for examining the tunability of hyperparameters of machine learning algorithms. It first defines the concept of defaults properly and proposes definitions for measuring the tunability of the whole algorithm, of single hyperparameters and of combinations of hyperparameters. For applying these definitions to a collection of 38 binary classification datasets, a random bot is created. It generated in total around 5 million experiment runs of 6 algorithms with different hyperparameters. The details of this bot are described in an extra paper (Kühn et al., 2018), co-authored by me, that is also included in this dissertation in Section A.4. The results of this bot are used to estimate the tunability of these 6 algorithms and their specific hyperparameters. Furthermore, ranges for parameter tuning of these algorithms are proposed.

1.10 Additional topics and work

There were several different additional topics that I worked on during my doctoral studies. At the beginning we finished a paper about the implementation of multilabel classification in `mlr` (Probst et al., 2017) that I started while writing the master thesis. Moreover, I co-authored several articles, including an empirical comparison of the performance of random forest with logistic regression (Couronné et al., 2018) and an article about the possibilities of making prediction rules applicable for readers (Boulesteix et al., 2018). I also contributed to the `mlr` tutorial (Schiffner et al., 2016) and created the

R package `quantregRanger` (Probst, 2017b) for quantile regression with the `ranger` R package (Wright, 2016) which now is available in an improved version in the `ranger` package itself. I also had several consulting projects with medical researchers at IBE, for example a project about asthma features (Matthes et al., 2018) and some other projects that are not finished at the present time. Last but not least, I supervised several bachelor and master theses, including the master thesis of Eva-Maria Müntefering about boosted random forests, a master thesis about the stability of the random forest variable importance by Thomas Klein-Heßling, a bachelor thesis about the influence of hyperparameters on support vector machines by Frederik Ludwigs and a bachelor thesis about a benchmarking suite of regression datasets by Merlin Raabe which we are planning to publish in future. Moreover, I supervised the openly available master thesis of Myriam Hatz about the influence of *mtry* in random forest (Hatz, 2018) and the master thesis of Moritz Herrmann which consisted of a large-scale benchmark experiment of prediction methods for survival using multi-omics data (Herrmann, 2019) which we also plan to publish in future.

Chapter 2

Concluding remarks and further steps

The topic of this thesis was the influence of hyperparameters on machine learning algorithms and how to improve their setting via meta-learning. The first paper treated a very specific hyperparameter of random forest (number of trees) (Probst and Boulesteix, 2018), while the second paper gave a general overview of the literature of hyperparameters and their influence in random forest with a benchmark for several tuning implementations in R (Probst et al., 2019b). The third paper was more general and examined the tunability of 6 different machine learning algorithms based on experiments on 38 datasets (Probst et al., 2019a). The description of these experiments, for which the results are openly available, were published in an extra paper of Kühn et al. (2018).

The first three papers describe different approaches for examining hyperparameters. One can make conclusions about how they theoretically can be set ideally or one can observe the empirical performance and behaviour of these hyperparameters on several datasets and then make general conclusions about them. In my opinion, both sides are important. For example, one can make theoretical considerations about the ideal kernel in support vector machines (e.g. should be able to create flexible decision boundaries), but without applying this kernel to real data one cannot really be sure if this kernel is good. So the interplay between theory and application is very important for the creation and evaluation of new algorithms and hyperparameters.

To standardize, automatize and speed-up the benchmarking of different algorithms and their hyperparameters it is useful to have standardized tools and benchmarking datasets. A first step into this direction is the OpenML data sharing platform (Vanschoren et al., 2013). Furthermore, dataset collections such as the OpenML100 classification datasets (Bischl et al., 2017) make it easier for users to get and choose datasets for their benchmarking. Other collections, for example for regression tasks and survival analysis, should be made available in the future.

Moreover, the machine learning benchmarking framework that is already well established for regression and classification could be expanded to other learning tasks. It is not yet completely clear, especially for non-expert users, which resampling methods and performance measures should be preferably used for tasks such as ordinal regression, time series, survival, multi-target prediction and clustering. Software solutions for these tasks are still in their infancy.

Probably, the most discussed machine learning technique nowadays is neural networks. Due to their flexibility they are applicable on a very broad range of problems. Their approximation capabilities have already been shown by Hornik (1991). As setting the hyperparameters for neural networks is very important and computation is usually very intensive, meta-learning can play a crucial role. Popular papers (Santoro et al., 2016; Finn et al., 2017) show the importance and actuality of this topic.

Another idea we partly already worked on is the creation of a set of multiple defaults for an algorithm that perform well on several datasets (Pfisterer et al., 2018). Moreover, software tools that provide automatic and specific tuning of algorithms such as `tuneRanger` (Probst, 2018a), `autoxgboost` (Thomas et al., 2017) and `liquidSVM` (Steinwart and Thomann, 2017) can be improved and developed for further algorithms. This is part of the process of automatizing the whole learning process which also consists of data preprocessing steps and final ensemble aggregation methods such as stacking. Software implementations such as `H2O AutoML` (The H2O.ai team, 2017) and `autosklearn` (Feurer et al., 2015) are first steps of putting the process of training, evaluating and ensemble building into one algorithm, so that users do not need to think and program the single processing steps.

Computing time plays a crucial role here, especially for large datasets. Good predictions should be made available in a reasonable time and new software tools should ideally provide an option to restrict the runtime of an algorithms before starting the algorithm. Further investigations should be done to optimize the process of improving the performance of an algorithm in the shortest possible time. One possibility is to compare learning curves across algorithms. Meta-Learning can also be applied here (Vanschoren, 2018).

An idea to combine timing issues and tuning especially for big datasets would be to start running an algorithm with small samples of the data (regarding observations and input variables) and different hyperparameter settings and evaluate the results on the out-of-bag data (data that was not used for training). Then iteratively the amount of data that is put into the algorithm is increased and the algorithm is run with hyperparameter settings that were successful in previous runs of the algorithm. A learning curve could show the amount of performance gain that is achieved with growing amount of data. A stopping criterion either externally as algorithm input or internally using the learning curve could stop the algorithm and finally ensemble techniques could be applied to combine the trained algorithms that were run so far.

Another idea for future work is the automatization of the exploratory data analysis. Here it is important to find an ideal solution for the human-machine interaction. The process of finding interesting patterns in the data should be facilitated by providing user interfaces that are also suitable for non-expert users and visualizations should be easy to create.

Bibliography

N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

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

T. B. Berrett and R. J. Samworth. Nonparametric independence testing via mutual information. *arXiv preprint arXiv:1711.06642*, 2017.

M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-Race and iterated F-Race: An overview. In *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer, 2010.

B. Bischl, O. Mersmann, H. Trautmann, and C. Weihs. Resampling methods for meta-model validation with recommendations for evolutionary computation. *Evolutionary Computation*, 20(2):249–275, 2012.

B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones. mlr: Machine learning in R. *Journal of Machine Learning Research*, 17 (170):1–5, 2016. R package version 2.9.

B. Bischl, G. Casalicchio, M. Feurer, F. Hutter, M. Lang, R. G. Mantovani, J. N. van Rijn, and J. Vanschoren. OpenML benchmarking suites and the OpenML100. *ArXiv preprint arXiv:1708.03731*, Aug. 2017.

B. Bischl, J. Richter, J. Bossek, D. Horn, J. Thomas, and M. Lang. mlrMBO: A modular framework for model-based optimization of expensive black-box functions. *ArXiv preprint arXiv:1703.03373*, 2017.

A.-L. Boulesteix, S. Janitza, R. Hornung, P. Probst, H. Busen, and A. Hapfelmeier. Making complex prediction rules applicable for readers: Current practice in random forest literature and recommendations. *Biometrical Journal*, pages 1–15, 2018.

P. Brazdil and C. Giraud-Carrier. Metalearning and algorithm selection: progress, state of the art and introduction to the 2018 special issue. *Machine Learning*, 107(1):1–14, 2018. ISSN 1573-0565.

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

L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

L. Breiman et al. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science*, 16(3):199–231, 2001.

G. Casalicchio, C. Molnar, and B. Bischl. Visualizing the feature importance for black box models. *arXiv preprint arXiv:1804.06620*, 2018.

O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1):131–159, Jan 2002. ISSN 1573-0565.

A. Chaudhuri and W. Hu. A fast algorithm for computing distance correlation. *Computational Statistics & Data Analysis*, 2019.

G. Chen, W. Florero-Salinas, and D. Li. Simple, fast and accurate hyper-parameter tuning in gaussian-kernel svm. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 348–355. IEEE, 2017.

T. Chen and C. 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.

M. Claesen and B. D. Moor. Hyperparameter search in machine learning. *MIC 2015: The XI Metaheuristics International Conference*, 2015.

W. S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, 1979.

C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

R. Couronné, P. Probst, and A.-L. Boulesteix. Random forest versus logistic regression: a large-scale benchmark experiment. *BMC Bioinformatics*, 19(1):270, 2018.

G. Csardi. *cranlogs: Download Logs from the 'RStudio' 'CRAN' Mirror*, 2015. R package version 2.1.0.

B. Cui. *DataExplorer: Data Explorer*, 2018. R package version 0.7.0.

Y. Dodge. *The Oxford Dictionary of Statistical Terms*. Oxford University Press on Demand, 2006.

H. Drucker, C. J. Burges, L. Kaufman, A. J. Smola, and V. Vapnik. Support vector regression machines. In *Advances in Neural Information Processing Systems*, pages 155–161, 1997.

C. Ferri, J. Hernández-Orallo, and R. Modroiu. An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, 30(1):27–38, 2009.

M. Feurer, J. T. Springenberg, and F. Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1128–1135. AAAI Press, 2015.

C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR.org, 2017.

C.-s. Foo, C. B. Do, and A. Y. Ng. Efficient multiple hyperparameter learning for log-linear models. In *Advances in Neural Information Processing Systems*, pages 377–384, 2008.

Y. Freund and R. E. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European Conference on Computational Learning Theory*, pages 23–37. Springer, 1995.

Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*, ICML'96, pages 148–156, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.

J. Fürnkranz and J. Petrak. An evaluation of landmarking variants. In *Working Notes of the ECML/PKDD 2000 Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning*, pages 57–68, 2001.

J. Gama and P. Brazdil. Cascade generalization. *Machine Learning*, 41(3):315–343, 2000.

A. Goldstein, A. Kapelner, J. Bleich, and E. Pitkin. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *Journal of Computational and Graphical Statistics*, 24(1):44–65, 2015.

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

M. Hatz. *Der Einfluss von mtry auf Random Forests*. PhD thesis, 2018. URL `http://nbn-resolving.de/urn/resolver.pl?urn=nbn:de:bvb:19-epub-59094-4`.

M. Herrmann. *Large-scale benchmark study of prediction methods using multi-omics data*. PhD thesis, 2019. URL `http://nbn-resolving.de/urn/resolver.pl?urn=nbn:de:bvb:19-epub-60505-4`.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

T. Hothorn, K. Hornik, and A. Zeileis. Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, 15(3):651–674, 2006.

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

S. Janitza, C. Strobl, and A.-L. Boulesteix. An AUC-based permutation variable importance measure for random forests. *BMC Bioinformatics*, 14(1):119, 2013.

D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

J.-H. Kim. Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. *Computational Statistics and Data Analysis*, 53(11):3735 – 3745, 2009. ISSN 0167-9473.

D. Kühn, P. Probst, J. Thomas, and B. Bischl. Automatic exploration of machine learning experiments on OpenML. *ArXiv preprint arXiv:1806.10961*, 2018.

D. Kühn, P. Probst, J. Thomas, and B. Bischl. OpenML R bot benchmark data (final subset). 2018. URL `https://figshare.com/articles/OpenML_R_Bot_Benchmark_Data_final_subset_/5882230`.

M. Lang. *Automatische Modellselektion in der Überlebenszeitanalyse*. PhD thesis, TU Dortmund, 2015.

J. Larsen, L. K. Hansen, C. Svarer, and M. Ohlsson. Design and regularization of neural networks: the optimal use of a validation set. In *Neural Networks for Signal Processing [1996] VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop*, pages 62–71. IEEE, 1996.

R. Leite, P. Brazdil, and J. Vanschoren. Selecting classification algorithms with active testing. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 117–131. Springer, 2012.

C. Lemke, M. Budka, and B. Gabrys. Metalearning: a survey of trends and technologies. *Artificial Intelligence Review*, 44(1):117–130, 2015.

S.-W. Lin, K.-C. Ying, S.-C. Chen, and Z.-J. Lee. Particle swarm optimization for parameter determination and feature selection of support vector machines. *Expert Systems with Applications*, 35(4):1817–1824, 2008.

G. Louppe. Understanding random forests: From theory to practice. *arXiv preprint arXiv:1407.7502*, 2014.

G. Luo. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 5(1):1–16, 2016.

S. Matthes, J. Stadler, J. Barton, G. Leuschner, D. Munker, P. Arnold, H. Villena-Hermoza, M. Frankenberger, P. Probst, A. Koch, et al. Asthma features in severe COPD: Identifying treatable traits. *Respiratory Medicine*, 145:89–94, 2018.

M. Meissner, M. Schmuker, and G. Schneider. Optimized particle swarm optimization (opso) and its application to artificial neural network training. *BMC Bioinformatics*, 7 (1):125, 2006.

U. B. Mogensen, H. Ishwaran, and T. A. Gerds. Evaluating random forests for survival analysis using prediction error curves. *Journal of Statistical Software*, 50(11):1, 2012.

C. Molnar. *Interpretable Machine Learning*. 2019. `https://christophm.github.io/interpretable-ml-book/`.

R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4206-3.

R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, and J. H. Moore. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData mining*, 10(1):36, 2017.

J.-S. Park. Optimal latin-hypercube designs for computer experiments. *Journal of Statistical Planning and Inference*, 39(1):95–111, 1994.

F. Pfisterer, J. N. van Rijn, P. Probst, A. Müller, and B. Bischl. Learning multiple defaults for machine learning algorithms. *arXiv preprint arXiv:1811.09409*, 2018.

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

P. Probst. *quantregRanger: Quantile Regression Forests for 'ranger'*, 2017b. R package version 1.0.

P. Probst. *tuneRanger: Tune random forest of the 'ranger' package*, 2018a. R package version 0.4.

P. Probst. *measures: Performance Measures for Statistical Learning*, 2018b. R package version 0.2.

P. Probst. *varImp: RF Variable Importance for Arbitrary Measures*, 2018c. R package version 0.2.

P. Probst and A.-L. Boulesteix. To tune or not to tune the number of trees in random forest. *Journal of Machine Learning Research*, 18(181):1–18, 2018.

P. Probst, Q. Au, G. Casalicchio, C. Stachl, and B. Bischl. Multilabel classification with R package mlr. *The R Journal*, 9(1):352–369, 2017.

P. Probst, A.-L. Boulesteix, and B. Bischl. Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, 20(53):1–32, 2019a.

P. Probst, M. N. Wright, and A.-L. Boulesteix. Hyperparameters and tuning strategies for random forest. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(3):e1301, 2019b.

A. Rivolli, L. P. Garcia, C. Soares, J. Vanschoren, and A. C. de Carvalho. Towards reproducible empirical research in meta-learning. *arXiv preprint arXiv:1808.10406*, 2018.

A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap. Meta-learning with memory-augmented neural networks. In *International Conference on Machine Learning*, pages 1842–1850, 2016.

J. Schiffner, B. Bischl, M. Lang, J. Richter, Z. M. Jones, P. Probst, F. Pfisterer, M. Gallo, D. Kirchhoff, T. Kühn, et al. mlr tutorial. *arXiv preprint arXiv:1609.06146*, 2016.

C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

N. Simon and R. Tibshirani. Comment on "detecting novel associations in large data sets" by reshef et al, science dec 16, 2011. *arXiv preprint arXiv:1401.7645*, 2014.

G. D. Smith and S. Ebrahim. Data dredging, bias, or confounding: They can all get you into the BMJ and the Friday papers, 2002.

J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

I. M. Sobol. Uniformly distributed sequences with an additional uniform property. *USSR Computational Mathematics and Mathematical Physics*, 16(5):236–242, 1976.

I. Steinwart and P. Thomann. liquidSVM: A fast and versatile svm package. *arXiv preprint arXiv:1702.06899*, 2017.

G. J. Székely, M. L. Rizzo, N. K. Bakirov, et al. Measuring and testing dependence by correlation of distances. *The Annals of Statistics*, 35(6):2769–2794, 2007.

The H2O.ai team. *h2o: R Interface for H2O*, 2017. R package version 3.16.0.2.

J. Thomas, S. Coors, and B. Bischl. Automatic gradient boosting. 2017.

C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 847–855. ACM, 2013.

J.-T. Tsai, J.-H. Chou, and T.-K. Liu. Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *IEEE Transactions on Neural Networks*, 17(1):69–80, 2006.

J. W. Tukey. Exploratory data analysis. *Reading, Ma 231*, 32, 1977.

H. Uno, T. Cai, L. Tian, and L. Wei. Evaluating prediction rules for t-year survivors with censored regression models. *Journal of the American Statistical Association*, 102(478): 527–537, 2007.

H. Uno, T. Cai, M. J. Pencina, R. B. D'Agostino, and L. Wei. On the c-statistics for evaluating overall adequacy of risk prediction procedures with censored survival data. *Statistics in Medicine*, 30(10):1105–1117, 2011.

J. N. van Rijn and F. Hutter. Hyperparameter importance across datasets. *ArXiv preprint arXiv:1710.04725*, 2017.

J. N. van Rijn, F. Pfisterer, J. Thomas, A. Muller, B. Bischl, and J. Vanschoren. Meta learning for defaults: Symbolic defaults. In *Neural Information Processing Workshop on Meta-Learning*, 2018.

J. Vanschoren. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*, 2018.

J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. OpenML: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

H. Weerts, M. Meuller, and J. Vanschoren. Importance of tuning hyperparameters of machine learning algorithms. Technical report, TU Eindhoven, 2018.

M. Wistuba, N. Schilling, and L. Schmidt-Thieme. Learning hyperparameter optimization initializations. In *Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on*, pages 1–10. IEEE, 2015.

D. H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.

D. H. Wolpert. The supervised learning no-free-lunch theorems. In *Soft Computing and Industry*, pages 25–42. Springer, 2002.

M. N. Wright. *ranger: A Fast Implementation of Random Forests*, 2016. R package version 0.6.0.

S. Xavier-de Souza, J. A. Suykens, J. Vandewalle, and D. Bollé. Coupled simulated annealing. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 40 (2):320–335, 2010.

Q. Zhou, W. Chen, S. Song, J. Gardner, K. Weinberger, and Y. Chen. A reduction of the elastic net to support vector machines with an application to GPU computing, 2015.

H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

Appendix A

Publications for the cumulative dissertation

To Tune or Not to Tune the Number of Trees in Random Forest

This chapter is a reprint of:

Probst, P. and Boulesteix, A.-L. (2018) To tune or not to tune the number of trees in random forest. Journal of Machine Learning Research 18 (181), 1-18.

Status:

Published.

Copyright:

Journal of Machine Learning Research, 2018.

Author contributions:

Philipp Probst first performed experiments and observed different patterns of the performance dependent on the number of trees. The theoretical part was developed together with Anne-Laure Boulesteix. Philipp Probst performed the benchmark analysis and prepared a first draft of the manuscript. Anne-Laure Boulesteix added valuable input and revised the document.

Acknowledgements:

We would like to thank Alexander Dürre for useful comments on the approximation of the logarithmic loss and Jenny Lee for language editing.

Hyperparameters and Tuning Strategies for Random Forest

This chapter is a reprint of:

Probst, P., Wright, M. and Boulesteix, A.-L. (2019): Hyperparameters and Tuning Strategies for Random Forest.

Status:

Published.

Copyright:

WIREs Data Mining Knowledge Discovery, 2019.

Author contributions:

Philipp Probst conducted the literature search and prepared a first draft of the manuscript. He developed the R package `tuneRanger` and realized the benchmark experiments. Anne-Laure Boulesteix and Marvin Wright revised the article and added valuable input (Anne-Laure Boulesteix especially in the theoretical section, Marvin Wright especially on the practical section). Marvin Wright revised the R package `tuneRanger`.

Acknowledgements:

Simon Klau helped testing the `tuneRanger` package. Thanks to Jenny Lee for language editing.

Tunability: Importance of Hyperparameters of Machine Learning Algorithms

This chapter is a reprint of:

Probst, P., Bischl, B. and Boulesteix, A.-L. (2019): Tunability: Importance of hyperparameters of machine learning algorithms.

Status:

Published.

Copyright:

Author contributions:

Methods and study design were developed by Philipp Probst in cooperation with Bernd Bischl and Anne-Laure Boulesteix. Philipp Probst then performed the analysis and prepared a first draft of the manuscript. Bernd Bischl and Anne-Laure Boulesteix finally revised the article.

Acknowledgements:

We would like to thank Joaquin Vanschoren for support regarding the OpenML platform and Andreas Müller, Jan van Rijn, Janek Thomas and Florian Pfisterer for reviewing and useful comments. Thanks to Jenny Lee for language editing.

Automatic Exploration of Machine Learning Experiments on OpenML

This chapter is a reprint of:

Kühn, D., Probst, P., Thomas, J. and Bischl, B. (2018): Automatic Exploration of Machine Learning Experiments on OpenML. *arXiv preprint arXiv:1806.10961.*

Status:

Arxiv Paper.

Author contributions:

The first code of the bot was created by Janek Thomas. Afterwards Daniel Kühn and Philipp Probst revised and improved it. The experiments were run and supervised on computer clusters by Daniel Kühn and Philipp Probst. Philipp Probst prepared a first draft of the manuscript. Daniel Kühn and Bernd Bischl added valuable input and revised it. Janek Thomas finally revised the document. Daniel Kühn and Philipp Probst are both first authors.

Acknowledgements:

We would like to thank Joaquin Vanschoren for support regarding the OpenML platform.

Journal of Machine Learning Research 18 (2018) 1-18 Submitted 5/17; Revised 2/18; Published 4/18

To Tune or Not to Tune the Number of Trees in Random Forest

Philipp Probst PROBST@IBE.MED.UNI-MUENCHEN.DE
Institut für medizinische Informationsverarbeitung, Biometrie und Epidemiologie
Marchioninistr. 15, 81377 München

Anne-Laure Boulesteix BOULESTEIX@IBE.MED.UNI-MUENCHEN.DE
Institut für medizinische Informationsverarbeitung, Biometrie und Epidemiologie
Marchioninistr. 15, 81377 München

Editor: Isabelle Guyon

Abstract

The number of trees T in the random forest (RF) algorithm for supervised learning has to be set by the user. It is unclear whether T should simply be set to the largest computationally manageable value or whether a smaller T may be sufficient or in some cases even better. While the principle underlying bagging is that more trees are better, in practice the classification error rate sometimes reaches a minimum before increasing again for increasing number of trees. The goal of this paper is four-fold: (i) providing theoretical results showing that the expected error rate may be a non-monotonous function of the number of trees and explaining under which circumstances this happens; (ii) providing theoretical results showing that such non-monotonous patterns cannot be observed for other performance measures such as the Brier score and the logarithmic loss (for classification) and the mean squared error (for regression); (iii) illustrating the extent of the problem through an application to a large number (n = 306) of datasets from the public database OpenML; (iv) finally arguing in favor of setting T to a computationally feasible large number as long as classical error measures based on average loss are considered.

Keywords: Random forest, number of trees, bagging, out-of-bag, error rate

1. Introduction

The random forest (RF) algorithm for classification and regression, which is based on the aggregation of a large number T of decision trees, was first described in its entirety by Breiman (2001). T is one of several important parameters which have to be carefully chosen by the user. Some of these parameters are *tuning parameters* in the sense that both too high and too low parameter values yield sub-optimal performances; see Segal (2004) for an early study on the effect of such parameters. It is unclear, however, whether the number of trees T should simply be set to the largest computationally manageable value or whether a smaller T may be sufficient or in some cases even better, in which case T should ideally be tuned carefully. This question is relevant to any user of RF and has been the topic of much informal discussion in the scientific community, but has to our knowledge never been addressed systematically from a theoretical and empirical point of view.

Breiman (2001) provides proofs of convergence for the generalization error in the case of classification random forest for growing number of trees. This means that the error rate for a given test or training dataset converges to a certain value. Moreover, Breiman (2001) proves that there exists an upper bound for the generalization error. Similarly he proves the convergence of the mean squared generalization error for regression random forests and also provides an upper bound. However, these results do not answer the question of whether the number of trees is a tuning parameter or should be set as high as computationally feasible, although convergence properties may at first view be seen as an argument in favor of a high number of trees. Breiman (1996a) and Friedman (1997) note that bagging and aggregation methods can make good predictors better but poor predictors can be transformed into worse. Hastie et al. (2001) show in a simple example that for a single observation that is incorrectly classified (in the binary case), bagging can worsen the expected missclassification rate. In Section 3.1 we will further analyse this issue and examine the outcome of aggregating performances for several observations.

Since each tree is trained individually and without knowledge of previously trained trees, however, the risk of overfitting when adding more trees discussed by Friedman (2001) in the case of boosting is not relevant here.

The number of trees is sometimes considered as a tuning parameter in current literature (Raghu et al., 2015); see also Barman et al. (2014) for a study in which different random seeds are tested to obtain better forests—a strategy implicitly assuming that a random forest with few trees may be better than a random forest with many trees. The R package `RFmarkerDetector` (Palla and Armano, 2016) even provides a function, 'tuneNTREE', to tune the number of trees. Of note, the question of whether a smaller number of trees may be better has often been discussed in online forums (see Supplementary File 1 for a non-exhaustive list of links) and seems to remain a confusing issue to date, especially for beginners.

A related but different question is whether a smaller number of trees is *sufficient* (as opposed to "better") in the sense that more trees do not improve accuracy. This question is examined, for example, in the very early study by Latinne et al. (2001) or by Hernández-Lobato et al. (2013). Another important contribution to that question is the study by Oshiro et al. (2012), which compared the performance in terms of the Area Under the ROC Curve (AUC) of random forests with different numbers of trees on 29 datasets. Their main conclusion is that the performance of the forest does not always substantially improve as the number of trees grows and after having trained a certain number of trees (in their case 128) the AUC performance gain obtained by adding more trees is minimal. The study of Oshiro et al. (2012) provides important empirical support for the existence of a "plateau", but does not directly address the question of whether a smaller number of trees may be substantially better and does not investigate this issue from a theoretical perspective, thus making the conclusions dependent on the 29 examined datasets.

In this context, the goal of our paper is four-fold: (i) providing theoretical results showing that, in the case of binary classification, the expected error rate may be a non-monotonous function of the number of trees and explaining under which circumstances this happens; (ii) providing theoretical results showing that such non-monotonous patterns cannot be observed for other performance measures such as the Brier score and the logarithmic loss (for classification) and the mean squared error (for regression); (iii) illustrating the extent of the problem through an application to a large number (n = 306) of datasets from the public database OpenML; (iv) finally arguing in favor of setting it to a computationally feasible

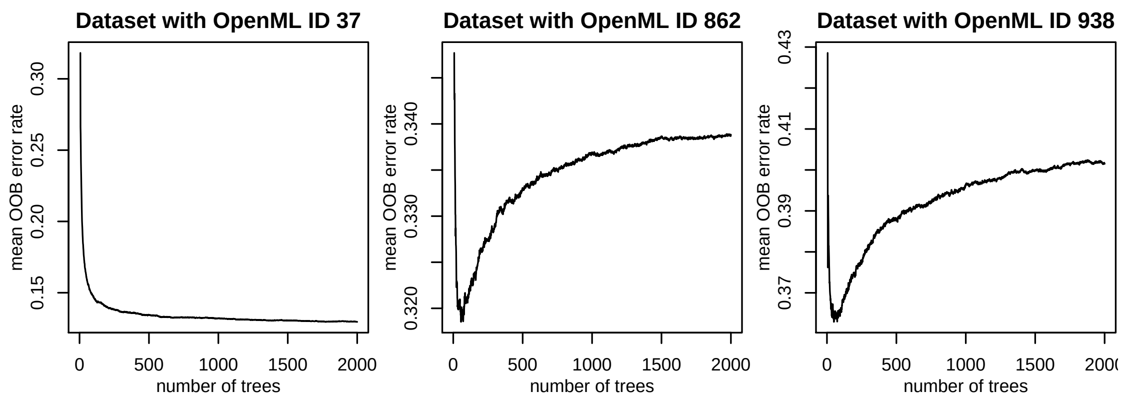

Figure 1: Mean OOB error rate curves for OpenML datasets with IDs 37, 862 and 938. The curves are averaged over 1000 independent runs of random forest.

large number as long as classical error measures based on average loss are considered. Furthermore, we introduce our new R package `OOBCurve`, which can be used to examine the convergence of various performance measures.

To set the scene, we first address this issue empirically by looking at the curve depicting the out-of-bag (OOB) error rate (see Section 2 for a definition of the OOB error) for different number of trees (also called OOB error rate curve) for various datasets from the OpenML database (Vanschoren et al., 2013). To obtain more stable results and better estimations for the expected error rate we repeat this procedure 1000 times for each dataset and average the results.

For most datasets we observe monotonously decreasing curves with growing number of trees as in the left panel of Figure 1, while others yield strange non-monotonous patterns, for example the curves of the datasets with the OpenML ID 862 and 938, which are also depicted in Figure 1. The initial error rate drops steeply before starting to increase after a certain number of trees before finally reaching a plateau.

At first view, such non-monotonous patterns are a clear argument in favor of tuning T. We claim, however, that it is important to understand why and in which circumstances such patterns happen in order to decide whether or not T should be tuned in general. In Section 3, we address this issue from a theoretical point of view, by formulating the expected error rate as a function of the probabilities ε_i of correct classification by a single tree for each observation i of the training dataset, for $i = 1, \dots, n$ (with n denoting the size of the training dataset). This theoretical view provides a clear explanation of the non-monotonous error rate curve patterns in the case of classification. With a similar approach, we show that such non-monotonous patterns cannot be obtained with the Brier score or the logarithmic loss as performance measures, which are based on probability estimations and also not for the mean squared error in the case of regression. Only for the AUC we can see non-monotonous curves as well.

The rest of this paper is structured as follows. Section 2 gives a brief introduction into random forest and performance estimation. Theoretical results are presented in Section 3, while the results of a large empirical study based on 306 datasets from the public database OpenML are reported in Section 4. More precisely, we empirically validate our theoretical model for the error as a function of the number of trees as well as our statements regarding the properties of datasets yielding non-monotonous patterns. We finally argue in Section 5 that there is no inconvenience—except additional computational cost—in adding trees to a random forest and that T should thus not be seen as a tuning parameter as long as classical performance measures based on the average loss are considered.

2. Background: Random Forest and Measures of Performance

In this section we introduce the random forest method, the general notation and some well known performance measures.

2.1 Random Forest

The random forest (RF) is an ensemble learning technique consisting of the aggregation of a large number T of decision trees, resulting in a reduction of variance compared to the single decision trees. In this paper we consider the original version of RF first described by Breiman (2001), while acknowledging that other variants exist, for example RF based on conditional inference trees (Hothorn et al., 2006) which address the problem of variable selection bias investigated by Strobl et al. (2007). Our considerations are however generalizable to many of the available RF variants and other methods that use randomization techniques.

A prediction is obtained for a new observation by aggregating the predictions made by the T single trees. In the case of regression RF, the most straightforward and common procedure consists of averaging the prediction of the single trees, while majority voting is usually applied to aggregate classification trees. This means that the new observation is assigned to the class that was most often predicted by the T trees.

While RF can be used for various types of response variables including censored survival times or (as empirically investigated in Section 4) multicategorical variables, in this paper we mainly focus on the two most common cases, binary classification and regression.

2.2 General Notations

From now on, we consider a fixed training dataset D consisting of n observations, which is used to derive prediction rules by applying the RF algorithm with a number T of trees. Ideally, the performance of these prediction rules is estimated based on an independent test dataset, denoted as D_{test}, consisting of n_{test} test observations.

Considering the ith observation from the test dataset ($i = 1, \dots, n_{test}$), we denote its true response as y_i, which can be either a numeric value (in the case of regression) or the binary label 0 vs. 1 (in the case of binary classification). The predicted value output by tree t (with $t = 1, \dots, T$) is denoted as $\hat{y}_{it}$, while $\hat{y}_i$ stands for the predicted value output by the whole random forest. Note that, in the case of regression, $\hat{y}_i$ is usually obtained by averaging as

$$\hat{y}_i = \frac{1}{T}\sum_{t=1}^{T} \hat{y}_{it}.$$

In the case of classification, $\hat{y}_i$ is usually obtained by majority voting. For binary classification, it is equivalent to computing the same average as for regression, which now takes the form

$$\hat{p}_i = \frac{1}{T}\sum_{t=1}^{T} I(\hat{y}_{it} = 1)$$

and is denoted as $\hat{p}_i$ (standing for probability), and finally deriving $\hat{y}_i$ as

$$\hat{y}_i = \begin{cases} 1 & \text{if } \hat{p}_i > 0.5, \\ 0 & \text{otherwise.} \end{cases}$$

2.3 Measures of Perfomance for Binary Classification and Regression

In regression as well as in classification, the performance of a RF for observation i is usually quantified through a so-called loss function measuring the discrepancy between the true response y_i and the predicted response $\hat{y}_i$ or, in the case of binary classification, between y_i and $\hat{p}_i$. For both regression and binary classification, the classical and most straightforward measure is defined for observation i as

$$e_i \;=\; (y_i - \hat{y}_i)^2 \;=\; L(y_i, \hat{y}_i),$$

with $L(.,.)$ standing for the loss function $L(x, y) = (x - y)^2$. In the case of regression this is simply the squared error. Another common loss function in the regression case is the absolute loss $L(x, y) = |x - y|$. For binary classification both measures simplify to $e_i = 0$ if observation i is classified correctly by the RF, $e_i = 1$ otherwise, which we will simply denote as *error* from now on. One can also consider the performance of single trees, that means the discrepancy between y_i and $\hat{y}_{it}$. We define e_{it} as

$$e_{it} = L(y_i, \hat{y}_{it}) = (y_i - \hat{y}_{it})^2$$

and the mean error—a quantity we need to derive our theoretical results on the dependence of performance measures on the number of tree T—as

$$\varepsilon_i = E(e_{it}),$$

where the expectation is taken over the possible trees conditionally on D. The term ε_i can be interpreted as the difficulty to predict y_i with single trees. In the case of binary classification, we have $(y_i - \hat{y}_{it})^2 = |y_i - \hat{y}_{it}|$ and ε_i can be simply estimated as $|y_i - \hat{p}_i|$ from a RF with a large number of trees.

In the case of binary classification, it is also common to quantify performance through the use of the Brier score, which has the form

$$b_i \;=\; (y_i - \hat{p}_i)^2 = L(y_i, \hat{p}_i)$$

or of the logarithmic loss

$$l_i \;=\; -(y_i \ln(\hat{p}_i) + (1-y_i)\ln(1-\hat{p}_i)).$$

Both of them are based on $\hat{p}_i$ rather than $\hat{y}_i$, and can thus be only defined for the whole RF and not for single trees.

The area under the ROC curve (AUC) cannot be expressed in terms of single observations, as it takes into account all observations at once by ranking the $\hat{p}_i$-values. It can be interpreted as the probability that the classifier ranks a randomly chosen observation with $y_i = 1$ higher than a randomly chosen observation with $y_i = 0$. The larger the AUC, the better the discrimination between the two classes. The (empirical) AUC is defined as

$$\text{AUC} = \frac{\sum_{i=1}^{n_1}\sum_{j=1}^{n_2} S(\hat{p}_i^{\star}, \hat{p}_j^{\star\star})}{n_1 n_2},$$

where $\hat{p}_1^{\star}, ..., \hat{p}_{n_1}^{\star}$ are probability estimations for the n_1 observations with $y_i = 1$, $\hat{p}_1^{\star\star}, ..., \hat{p}_{n_2}^{\star\star}$ are probability estimations for the n_2 observations with $y_i = 0$ and $S(.,.)$ is defined as $S(p,q) = 0$ if $p < q$, $S(p,q) = 0.5$ if $p = q$ and $S(p,q) = 1$ if $p > q$. The AUC can also be interpreted as the Mann-Whitney U-Statistic divided by the product of n_1 and n_2.

2.4 Measures for Multiclass Classification

The measures defined in the previous section can be extended to the multiclass classification case. Let K denote the number of classes ($K > 2$). The response y_i takes values in $\{1, ..., K\}$. The error for observation i is then defined as

$$e_i = I(y_i \neq \hat{y}_i).$$

We denote the estimated probability of class k for observation i as

$$\hat{p}_{ik} = \frac{1}{T}\sum_{t=1}^{T} I(\hat{y}_{it} = k).$$

The logarithmic loss is then defined as

$$l_i = \sum_{k=1}^{K} -I(y_i = k)\log(\hat{p}_{ik})$$

and the generalized Brier score is defined as

$$b_i = \sum_{k=1}^{K} (\hat{p}_{ik} - I(y_i = k))^2,$$

which in the binary case is twice the value of the definition that was used in the previous section. Following Hand and Till (2001), the AUC can also be generalized to the multiclass case as

$$\text{AUC} = \frac{1}{K(K-1)}\sum_{j=1}^{K}\sum_{\substack{k=1\\k\neq j}}^{K} \text{AUC}(j,k),$$

where $\text{AUC}(j,k)$ is the AUC between class k and j, see also Ferri et al. (2009) for more details. It is equivalent to the definition given in Section 2.3 in the binary classification case.

2.5 Test Dataset Error vs. Out-of-Bag Error

In the cases where a test dataset D_{test} is available, performance can be assessed by averaging the chosen performance measure (as described in the previous paragraphs) over the n_{test} observations. For example the classical error rate (for binary classification) and the mean squared error (for regression) are computed as

$$\frac{1}{n_{test}} \sum_{i=1}^{n_{test}} L(y_i, \hat{y}_i),$$

with $L(x,y) = (x-y)^2$, while the mean absolute error (for regression) is obtained by defining $L(.,.)$ as $L(x,y) = |x-y|$. Note that, in the context of regression, Rousseeuw (1984) proposes to consider the median $med\,(L(y_1, \hat{y}_1), ..., L(y_{n_{test}}, \hat{y}_{n_{test}}))$, instead of averaging, which results in the median squared error for the loss function $L(x,y) = (x-y)^2$ and in the median absolute error for the loss function $L(x,y) = |x-y|$. These measures are more robust against outliers and contamination (Rousseeuw, 1984).

An alternative to the use of a test dataset is the out-of-bag error which is calculated by using the out-of-bag (OOB) estimations of the training observations. OOB predictions are calculated by predicting the class, the probability (in the classification case) or the real value (in the regression case) for each training observation i (for $i = 1, \ldots, n$) by using only the trees for which this observation was not included in the bootstrap sample (i.e., it was not used to construct the tree). Note that these predictions are obtained based on a subset of trees—including on average $T \times 0.368$ trees. These predictions are ultimately compared to the true values by calculating performance measures (see Sections 2.3, 2.4 and 2.5).

3. Theoretical Results

In this section we compute the expected performance—according to the error, the Brier score and the logarithmic loss outlined in Section 2.3—of a binary classification or regression RF consisting of T trees as estimated based on the n_{test} test observations, while considering the training dataset as fixed. For the AUC we prove that it can be a non-monotonous function in T. The case of other measures (mean absolute error, median of squared error and median of absolute error for regression) and multiclass classification is much more more complex to investigate from a theoretical point of view. It will be examined empirically in Section 4.

In this section we are concerned with *expected* performances, where expectation is taken over the sets of T trees. Our goal is to study the monotonicity of the expected errors with respect to T. The number T of trees is considered a parameter of the RF and now mentioned in parentheses everytime we refer to the whole forest.

3.1 Error Rate (Binary Classification)

We first show that for single observations the expected error rate curve can be increasing and then show exemplified how this can influence the shape of the average curve of several observations. The observation that bagging can worsen the expected error rate of a single observation was already done by Hastie et al. (2001), Breiman (1996a) and Friedman (1997). In this section we provide a general formula explaining this observation, and then extend our theoretical considerations to further performance measures in the following sections.

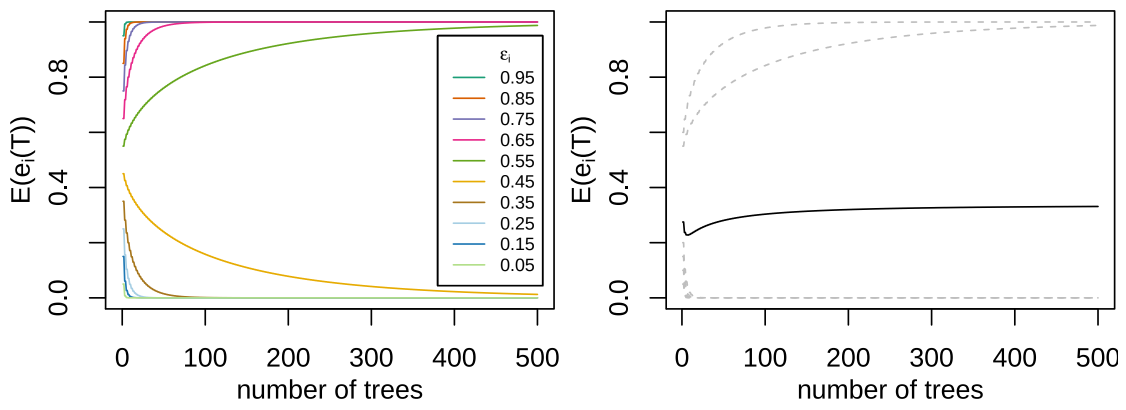

Figure 2: Left: Expected error rate curves for different ε_i values. Right: Plot of the average curve (black) of the curves with $\varepsilon_1 = 0.05$, $\varepsilon_2 = 0.1$, $\varepsilon_3 = 0.15$, $\varepsilon_4 = 0.2$, $\varepsilon_5 = 0.55$ and $\varepsilon_6 = 0.6$ (depicted in grey and dotted)

3.1.1 Theoretical Considerations

Let us first consider the classical error rate $e_i(T)$ for observation i with a RF including T trees and derive its expectation, conditionally on the training set D,

$$E(e_i(T)) \quad = \quad E\left(I\left(\frac{1}{T}\sum_{t=1}^{T} e_{it} > 0.5\right)\right) = P\left(\sum_{t=1}^{T} e_{it} > 0.5 \cdot T\right).$$

We note that e_{it} is a binary variable with $E(e_{it}) = \varepsilon_i$. Given a fixed training dataset D and observation i, the e_{it}, $t = 1, ..., T$ are mutually independent. It follows that the sum $X_i = \sum_t^T e_{it}$ follows the binomial distribution $B(T, \varepsilon_i)$. It is immediate that the contribution of observation i to the expected error rate, $P(X_i > 0.5 \cdot T)$, is an increasing function in T for $\varepsilon_i > 0.5$ and a decreasing function in T for $\varepsilon_i < 0.5$.

Note that so far we ignored the case where $\sum_{t=1}^{T} e_{it} = 0.5 \cdot T$, which may happen when T is even. In this case, the standard implementation in R (`randomForest`) assigns the observation randomly to one of the two classes. This implies that $0.5 \cdot P(\sum_{t=1}^{T} e_{it} = 0.5 \cdot T)$ has to be added to the above term, which does not affect our considerations on the ε_i's role.

3.1.2 Impact on Error Rate Curves

The error rate curve for observation i is defined as the curve described by the function $e_i : T \to \mathbb{R}$. The expectation $E(e_i(T))$ of the error rate curve for observation i with the mentioned adjustment in the case of an even number of trees can be seen in the left plot of Figure 2 for different values of ε_i. Very high and very low values of ε_i lead to rapid convergence, while for ε_i-values close to 0.5 more trees are needed to reach the plateau. The error rate curve obtained for a test dataset consists of the average of the error rate curves of the single observations. Of course, if trees are good classifiers we should have $\varepsilon_i < 0.5$ for most observations. In many cases, observations with $\varepsilon_i > 0.5$ will be compensated by observations with $\varepsilon_i < 0.5$ in such a way that the expected error rate curve is monotonously decreasing. This is typically the case if there are many observations with $\varepsilon_i \approx 0$ and a few with $\varepsilon_i \approx 1$. However, if there are many observations with $\varepsilon_i \approx 0$ and a few observations with $\varepsilon_i \geq 0.5$ that are close to 0.5, the expected error rate curve initially falls down quickly because of the observation with $\varepsilon_i \approx 0$ and then grows again slowly as the number of trees increases because of the observations with $\varepsilon_i \geq 0.5$ close to 0.5. In the right plot of Figure 2 we can see (black solid line) the mean of the expected error rate curves for $\varepsilon_1 = 0.05$, $\varepsilon_2 = 0.1$, $\varepsilon_3 = 0.15$, $\varepsilon_4 = 0.2$, $\varepsilon_5 = 0.55$ and $\varepsilon_6 = 0.6$ (displayed as gray dashed lines) and can see exactly the non-monotonous pattern that we expected: due to the ε_i's 0.55 and 0.6 the average curve increases again after reaching a minimum. In Section 4 we will see that the two example datasets whose non-monotonous out-of-bag error rate curves are depicted in the introduction have a similar distribution of ε_i.

We see that the convergence rate of the error rate curve is only dependent on the distribution of the ε_i's of the observations. Hence, the convergence rate of the error rate curve is not directly dependent on the number of observations n or the number of features, but these characteristics could influence the empirical distribution of the ε_i's and hence possibly the convergence rate as outlined in Section 4.4.1.

3.2 Brier Score (Binary Classification) and Squared Error (Regression)

We now turn to the Brier score and compute the expected Brier score contribution of observation i for a RF including T trees, conditional on the training set D. We obtain

$$\begin{aligned} E(b_i(T)) &= E((y_i - \hat{p}_i(T))^2) = E\left(\left(y_i - \frac{1}{T}\sum_{t=1}^{T}\hat{y}_{it}\right)^2\right) \\ &= E\left(\left(\frac{1}{T}\sum_{t=1}^{T}(y_i - \hat{y}_{it})\right)^2\right) = E\left(\left(\frac{1}{T}\sum_{t=1}^{T}e_{it}\right)^2\right). \end{aligned}$$

From $E(Z^2) = E(Z)^2 + Var(Z)$ with $Z = \frac{1}{T}\sum_{t=1}^{T} e_{it}$ it follows:

$$E(b_i(T)) = E(e_{it})^2 + \frac{Var(e_{it})}{T},$$

which is obviously a strictly monotonous decreasing function of T. This also holds for the average over the observations of the test dataset. In the case of binary classification, we have $e_{it} \sim \mathcal{B}(1, \varepsilon_i)$, yielding $E(e_{it}) = \varepsilon_i$ and $Var(e_{it}) = \varepsilon_i(1-\varepsilon_i)$, thus allowing the formulation of $E(b_i(T))$ as $E(b_i(T)) = \varepsilon_i^2 + \frac{\varepsilon_i(1-\varepsilon_i)}{T}$. Note that the formula $E(b_i(T)) = E(e_{it})^2 + Var(e_{it})/T$ is also valid for the squared error in the regression case, except that in this case we would write $\hat{y}_i$ instead of $\hat{p}_i$ in the first line.

3.3 Logarithmic Loss (Binary Classification)

As outlined in Section 2.3, another usual performance measure based on the discrepancy between y_i and $\hat{p}_i$ is the logarithmic loss $l_i(T) = -(y_i \ln(\hat{p}_i(T)) + (1 - y_i)\ln(1 - \hat{p}_i(T)))$. Noticing that $\hat{p}_i(T) = 1 - \frac{1}{T}\sum_{t=1}^{T} e_{it}$ for $y_i = 1$ and $\hat{p}_i(T) = \frac{1}{T}\sum_{t=1}^{T} e_{it}$ for $y_i = 0$, it can be in both cases $y_i = 0$ and $y_i = 1$ reformulated as

$$l_i(T) = -\ln\left(1 - \frac{1}{T}\sum_{t=1}^{T} e_{it}\right).$$

In the following we ensure that the term inside the logarithm is never zero by adding a very small value a to $1 - \frac{1}{T}\sum_{t=1}^{T} e_{it}$. The logarithmic loss $l_i(T)$ is then always defined and its expectation exists. This is similar to the solution adopted in the `mlr` package, where 10^{-15} is added in case that the inner term of the logarithm equals zero.

With $Z := 1 - \frac{1}{T}\sum_{t=1}^{T} e_{it} + a$, we can use the Taylor expansion,

$$\begin{aligned} E\left[f(Z)\right] =& E\left[f(\mu_Z + (Z - \mu_Z))\right] \\ \approx& E\left[f(\mu_Z) + f'(\mu_Z)\left(Z - \mu_Z\right) + \frac{1}{2}f''(\mu_Z)\left(Z - \mu_Z\right)^2\right] \\ =& f(\mu_Z) + \frac{f''(\mu_Z)}{2} \cdot Var(Z) = f(E(Z)) + \frac{f''(E(Z))}{2} \cdot Var(Z) \end{aligned}$$

where μ_Z stands for $E(Z)$ and $f(.)$ as $f(.) = -\ln(.)$. We have $Var(Z) = \frac{\varepsilon_i(1-\varepsilon_i)}{T}$, $E(Z) = 1 - \varepsilon_i + a$, $f(E(Z)) = -\ln(1 - \varepsilon_i + a)$ and $f''(E(Z)) = (1 - \varepsilon_i + a)^{-2}$, finally yielding

$$E(l_i(T)) \approx -\ln(1 - \varepsilon_i + a) + \frac{\varepsilon_i(1 - \varepsilon_i)}{2T(1 - \varepsilon_i + a)^2},$$

which is obviously a decreasing function of T. The Taylor approximation gets better and better for increasing T, since the variance of $l_i(T)$ decreases with increasing T and thus $l_i(T)$ tends to get closer to its expectancy.

3.4 Area Under the ROC Curve (AUC) (Classification)

For the AUC, considerations such as those we made for the error rate, the Brier score and the logarithmic loss are impossible, since the AUC is not the sum of individual contributions of the observations. It is however relatively easy to see that the expected AUC is not always an increasing function of the number T of trees. For example, think of the trivial example of a test dataset consisting of two observations with responses y_1 resp. y_2 and $E(\hat{p}_1(T)) = 0.4$ resp. $E(\hat{p}_2(T)) = 0.6$. If $y_1 = 0$ and $y_2 = 1$, the expected AUC curve increases monotonously with T, as the probability of a correct ordering according to the calculated scores $\hat{p}_1(T)$ and $\hat{p}_2(T)$ increases. However, if $y_1 = 1$ and $y_2 = 0$, we obtain a monotonously decreasing function, as the probability of a wrong ordering gets higher with increasing number of trees. It is easy to imagine that for different combinations of $E(\hat{p}_i(T))$, one can obtain increasing curves, decreasing curves or non-monotonous curves.

3.5 Adapting the Models to the OOB Error

The "OOB estimator" of the performance outlined in Section 2.5 is commonly considered as an acceptable proxy of the performance estimator obtained through the use of an independent test dataset or through resampling-techniques such as cross-validation (Breiman, 1996b) for a random forest including $T \times 0.368$ trees. Compared to these techniques, the

OOB estimator has the major advantage that it neither necessitates to fit additional random forests (which is advantageous in terms of computational resources) nor to reduce the size of the dataset through data splitting. For these reasons, we will consider OOB performance estimators in our empirical study.

However, if we consider the OOB error instead of the test error from an independent dataset, the formulas given in the previous subsections are not directly applicable. After having trained T trees, for making an OOB estimation for an observation we can only use the trees for which the observation was out-of-bag. If we take a simple bootstrap sample from the n training observation when bagging we have *on average* only $T \cdot (1 - \frac{1}{n})^n \approx T \cdot \exp(-1) \approx T \cdot 0.368$ trees for predicting the considered observation. This means that we would have to replace T by $T \cdot \exp(-1)$ in the above formulas and that the formulas are no longer exact because $T \cdot \exp(-1)$ is only an average. Nonetheless it is still a good approximation as confirmed in our benchmark experiments.

4. Empirical Results

This section shows a large-scale empirical study based on 193 classification tasks and 113 regression tasks from the public database OpenML (Vanschoren et al., 2013). The datasets are downloaded with the help of the `OpenML` R package (Casalicchio et al., 2017). The goals of this study are to (i) give an order of magnitude of the frequency of non-monotonous patterns of the error rate curve in real data settings; (ii) empirically confirm our statement that observations with ε_i greater than (but close to) 0.5 are responsible for non-monotonous patterns; (iii) analyse the results for other classification measures, the multiclass classification and several regression measures; (iv) analyse the convergence rate of the OOB curves.

4.1 Selection of Datasets

To select the datasets to be included in our study we define a set of candidate datasets—in our case the datasets available from the OpenML platform (Vanschoren et al., 2013)—and a set of inclusion criteria as recommended in Boulesteix et al. (2017). In particular, we do not select datasets with respect to the results they yield, thus warranting representativity.

Our inclusion criteria are as follows: (i) the dataset has predefined tasks in OpenML (see Vanschoren et al., 2013, for details on the OpenML nomenclature); (ii) it includes less than 1000 observations; (iii) it includes less than 1000 features. The two latter criteria aim at keeping the computation time feasible.

Cleaning procedures such as the deletion of duplicated datasets (whole datasets that appear twice in the OpenML database) are also applied to obtain a decent collection of datasets. No further modification of the tasks and datasets were done.

This procedure yields a total of 193 classification tasks and 113 regression tasks.

From the 193 classification tasks, 149 are binary classification tasks and 44 multiclass classification tasks.

The tasks contained easy, medium and difficult tasks - for binary classification tasks the mean (out-of-bag) AUC of a random forest with 2000 trees was 0.841, the minimum 0.502, the first quartile 0.732, the median 0.870, the third quartile 0.962 and the maximum 1. Similarly the regression tasks contained easy and difficult tasks with a mean R^2 of 0.559.

4.2 Study Design

For each dataset we run the RF algorithm with $T = 2000$ trees 1000 times successively with different seeds using the R package `randomForest` (Liaw and Wiener, 2002) with the default parameter settings. We choose 2000 trees because in a preliminary study on a subset of the datasets we could observe that convergence of the OOB curves was reached within these 2000 trees. Note that all reported results regarding the performance gain and convergence are made with the out-of-bag predictions. As for these predictions on average only $\exp(-1) \cdot T$ of the T trees are used, the convergence of independent test data is faster by the factor 2.7. For the classification tasks we calculate the OOB curves for the error rate, the balanced error rate, the (multiclass) Brier score, the logarithmic loss and the (multiclass) AUC using our new package `OOBCurve`, see details in the next section.

For the regression tasks we calculate the OOB curves using the mean squared error, the mean absolute error, the median squared error and the median of absolute error as performance measures. We parallelize the computations using the R package `batchtools` (version 0.9.0) (Lang et al., 2017). For each measure and each dataset, the final curve is obtained by simply averaging over the 1000 runs of RF. We plot each of them in three files separetely for binary classification, multiclass classification and regression. In the plots the x-axis starts at $T = 11$ since overall performance estimates are only defined if each observation was out-of-bag in at least one of the T trees, which is not always the case in practice for $T < 10$. We plot the curves only until $T = 500$, as no interesting patterns can be observed after this number of trees (data not shown). The graphics, the R-codes and the results of our experiment can be found on `https://github.com/PhilippPro/tuneNtree`.

4.3 The R Package `OOBCurve`

The calculation of out-of-bag estimates for different performance measures is implemented in our new R package `OOBCurve`. More precisely, it takes a random forest constructed with the R package `randomForest` (Liaw and Wiener, 2002) or `ranger` (Wright, 2016) as input and can calculate the OOB curve for any measure that is available from the `mlr` package (Bischl et al., 2016). The `OOBCurve` package is available on CRAN R package repository and also on Github (`https://github.com/PhilippPro/OOBCurve`). It is also possible to calculate OOB curves of other hyperparameters of RF such as `mtry` with this package.

4.4 Results for Binary Classification

The average gain in performance in the out-of-bag performance for 2000 trees instead of 11 trees is -0.0324 for the error rate, -0.0683 for the brier score, -2.383 for the logarithmic loss and 0.0553 for the AUC. In the following we will concentrate on the visual analysis of the graphs and are especially interested in the results of the error rate.

4.4.1 Overall Results for the OOB Error Rate Curves

We observe in the graphs of the OOB error rate curves that for most datasets the curve is quickly decreasing until it converges to a dataset-specific plateau value. In 16 cases which make approximately 10% of the datasets, however, the curve grows again after reaching its lowest value, leading to a value at 2000 trees that is by at least 0.005 bigger than the lowest value of the OOB error rate curve for $T \in [10, 250]$. This happens mainly for smaller datasets, where a few observations can have a high impact on the error curve. Of these 16 cases 15 belong to the smaller half of the datasets—ordered by the number of observations multiplied with the number of features. The mean increase of these 16 datasets was 0.020 (median: 0.012). The difference in mean and median is mainly caused by one outlier where the increase was around 0.117.

4.4.2 Datasets with Non-Monotonous OOB Error Rate Curve

We now examine in more detail the datasets yielding non-monotonous patterns. In particular, the histograms of the estimates $\hat{\varepsilon}_i = |y_i - \hat{p}_i|$ of the observation-specific errors ε_i are of interest, since our theoretical results prove that the distribution of the ε_i determines the form of the expected error rate curve. To get these histograms we compute the estimates $\hat{\varepsilon}_i$ of the observation-specific errors ε_i (as defined in Section 2.3) from a RF with a big number $T = 100000$: the more trees, the more accurate the estimates of ε_i.

The histograms for the exemplary datasets considered in the introduction (see Figure 1) are displayed in Figure 3. A typical histogram for an OOB curve with monotonously decreasing error rate curve is displayed in the left panel. The heights of the bins of this histogram of the $\hat{\varepsilon}_i$ are monotonously decreasing from 0 to 1.

The histograms for the non-monotonous error rate curves from the introduction can be seen in the middle (OpenML ID 862) and right (OpenML ID 938) panels of Figure 3. In both cases we see that a non-negligible proportion of observations have ε_i larger than but close to 0.5. This is in agreement with our theoretical results. With growing number of trees the chance that these observations are incorrectly classified increases, while the chance for observations with $\varepsilon_i \approx 0$ is already very low—and thus almost constant. Intuitively we expect such shapes of histograms for datasets with few observations—where by chance the shape of the histogram of the $\hat{\varepsilon}_i$ could look like in our two examples. For bigger datasets we expect smoother shapes of the histogram, yielding strictly decreasing error rate curves.

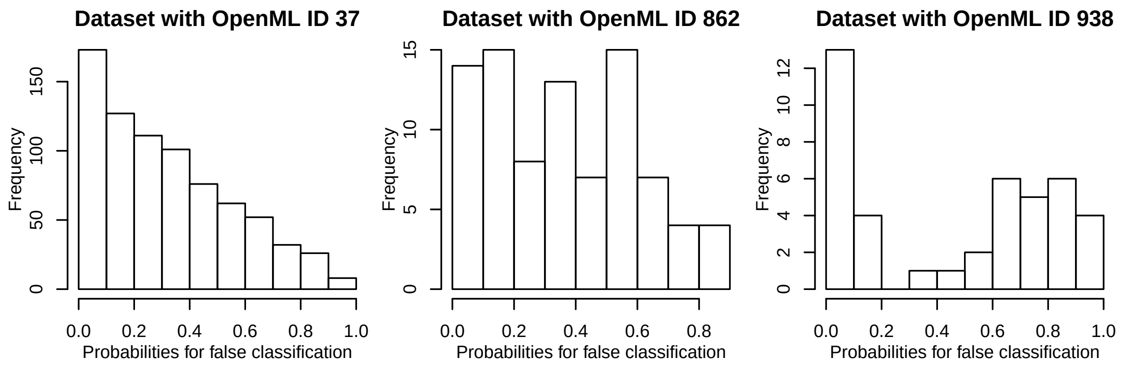

Figure 3: Histograms of the estimates of ε_i $(i = 1, \ldots, n)$ from random forests with 100000 trees for dataset with IDs 36, 862 and 938

	error rate		Brier score		logarithmic loss		AUC	
error rate	**1.00**	(1.00)	**0.28**	(0.44)	**0.27**	(0.45)	**-0.18**	(-0.43)
Brier score	**0.72**	(0.86)	**1.00**	(1.00)	**0.96**	(0.98)	**-0.63**	(-0.87)
logarithmic loss	**0.65**	(0.84)	**0.93**	(0.95)	**1.00**	(1.00)	**-0.63**	(-0.87)
AUC	**-0.64**	(-0.85)	**-0.84**	(-0.95)	**-0.81**	(-0.92)	**1.00**	(1.00)

Table 1: Linear (bottom-left) and rank (top-right) correlation results for binary classification datasets and for multiclass classification (in brackets)

4.4.3 Other Measures

For the Brier score and the logarithmic loss we observe, as expected, monotonically decreasing curves for all datasets. The expected AUC curve usually appears as a growing function in T. In a few datasets such as the third binary classification example (OpenML ID 905), however, it falls after reaching a maximum.

To assess the similarity between the different curves, we calculate the Bravais-Pearson linear correlation and Kendall's τ rank correlation between the values of the OOB curves of the different performance measures and average these correlation matrices over all datasets. Note that we do not perform any correlation tests, since the assumption of independent identically distributed observations required by these tests is not fulfilled: our correlation analyses are meant to be explorative. The results can be seen in Table 1. The Brier score and logarithmic loss have the highest correlation. They are also more correlated to the AUC than to the error rate, which has the lowest correlation to all other measures.

4.5 Results for Multiclass Classification

The average gain in out-of-bag performance for 2000 trees instead of 11 trees is -0.0753 for the error rate, -0.1282 for the brier score, -5.3486 for the logarithmic loss and 0.0723 for the AUC. These values are higher than the ones from binary classification. However, the visual observations we made for the binary classification also hold for the multiclass classification. For 5 of the 44 datasets the minimum error rate for $T \in [11; 250]$ is lower by more than 0.005 than the error rate for $T = 2000$. In contrast to the binary classification case, 3 of these 5 datasets belong to the bigger half of the datasets. The results for the correlation are quite similar, although the correlation (see Table 1) is in general slightly higher than in the binary case.

4.6 Results for Regression

The average performance gain regarding the out-of-bag performance of the R^2 for 2000 trees compared to 11 trees is 0.1249. In the OOB curves for regression we can observe the monotonously decreasing pattern expected from theory in the case of the most widely used mean squared error (mse). The mean absolute error (mae) is also strictly decreasing for all the datasets considered in our study.

For the median squared error (medse) and the median absolute error (medae), we get a performance gain by using 2000 trees instead of 10 in most but not all cases (around 80% of the datasets). In many cases (around 50%) the minimum value for $T \in [11; 250]$ is smaller than the value for $T = 2000$ which means that growing more trees is rather disadvantagous in these cases in terms of medse and medae. This could be explained by the fact that each tree in a random forest tries to minimize the squared error in the splits and therefore adding more trees to the forest will improve the mean squared error but not necessarily measures that use the median. More specifically, one could imagine that the additional trees focus on the reduction of the error for outlying observations at the price of an increase of the median error. In a simulated dataset (linear model with 200 observations, 5 relevant features and 5 non-relevant features drawn from a multivariate normal distribution) we could observe this pattern (data not shown). Without outlier all expected curves are strictly decreasing. When adding an outlier (changing the outcome of one observation to a very big value) the expected curves of mse and mae are still strictly decreasing, while the expected curves of medse and medae show are increasing for higher T. The curves of the measures which take the mean of the losses of all observations have a high linear and rank correlation (> 0.88), as well as the curves of the measures which take the median of the losses (> 0.97). Correlation between these two groups of measures are lower, around 0.5 for the linear correlation coefficient and around 0.2 for the rank correlation coefficient.

4.7 Convergence

It is clearly visible from the out-of-bag curves (`https://github.com/PhilippPro/tuneNtree/tree/master/graphics`) that increasing the number of trees yields a substantial performance gain in most of the cases, but the biggest performance gain in the out-of-bag curves can be seen while growing the first 250 trees. Setting the number of trees from 10 to 250 in the binary classification case provides an average decrease of 0.0306 of the error rate and an increase of 0.0521 of the AUC. On the other hand, using 2000 trees instead of 250 does not yield a big performance gain, the average error rate improvement is only 0.0018 (AUC: 0.0032). The improvement in the multiclass case is bigger with an average improvement of the error rate of 0.0739 (AUC: 0.0665) from 10 trees to 250 and an average improvement of 0.0039 (AUC: 0.0057) for using 2000 trees instead of 250. For regression we have an improvement of 0.1210 of the R^2 within the first 250 trees and an improvement of 0.0039 for using 2000 trees instead of 250. These results are concordant with a comment by Breiman (1996a) (Section 6.2) who notes that fewer bootstrap replicates are necessary when the outcome is numerical and more are required for an increasing number of classes.

5. Conclusions and Extensions

In this section we draw conclusions of the given results and discuss possible extensions.

5.1 Assessment of the Convergence

For the assessment of the convergence in the classification case we generally recommend using measures other than the error rate, such as AUC, the Brier score or the logarithmic loss for which the OOB curves are much more similar as we have seen in our correlation analysis. Their convergence rate is not so dependent on observations with ε_i close to 0.5 (in the binary classification case), and they give an indication of the general stability of the probability estimations of all observations. This can be especially important if the threshold for classification is not set a priori to 0.5. The new `OOBCurve` R package is a tool to examine the rate of convergence of the trained RF with any measure that is available in the `mlr` R package. It is important to remember that for the calculation of the OOB error curve at T only $\exp(-1) \cdot T$ trees are used. Thus, as far as future independent data is concerned, the convergence of the performances is by $\exp(1) \approx 2.7$ faster than observed from our OOB curves. Having this in mind, our observations (see Section 4.7) are in agreement with the results of Oshiro et al. (2012), who conclude that after growing 128 trees no big gain in the AUC performance could be achieved by growing more trees.

5.2 Why More Trees Are Better

Non-monotonous expected error rate curves observed in the case of binary classification might be seen as an argument in favour of tuning the number T of trees. Our results, however, suggest that tuning is not recommendable in the case of classification. Firstly, non-monotonous patterns are observed only with some performance measures such as the error rate and the AUC in case of classification. Measures such as the Brier score or the logarithmic loss, which are based on probabilities rather than on the predicted class and can thus be seen as more refined, do not yield non-monotonous patterns, as theoretically proved in Section 3 and empirically observed based on a very large number of datasets in Section 4. Secondly, non-monotonous patterns in the expected error rate curves are the result of a particular rare combination of ε_i's in the training data. Especially if the training dataset is small, the chance is high that the distribution of the ε_i will be different for independent test data, for example values of ε_i close to but larger than 0.5 may not be present. In this case, the expected error rate curve for this independent future dataset would not be non-monotonous, and a large T is better. Thirdly, even in the case of non-monotonic expected error rate curves, the minimal error rate value is usually only slightly smaller than the value at convergence (see Section 4.4.1). We argue that this very small gain - which, as outlined above, is relevant only for future observations with $\varepsilon_i > 0.5$ - probably does not compensate the advantage of using more trees in terms of other performance measures or in terms of the precision of the variable importance measures, which are very commonly used in practice.

In the case of regression, our theoretical results show that the expected out-of-bag mse curve is monotonously decreasing. For the mean absolute error the empirical results suggest the same. In terms of the less common measures *median* squared error and *median* absolute error (as opposed to *mean* losses), however, performance may get worse with increasing number of trees. More research is needed.

5.3 Extensions

Note that our theoretical results are not only valid for random forest but generalizable to any ensemble method that uses a randomization technique, since the fact that the base learners are trees and the specific randomization procedure (for example bagging) do not play any role in our proofs. Our theoretical results could possibly be extended to the multiclass case, as supported by our results obtained with 44 multiclass datasets.

Although we claim that increasing the number of trees cannot harm noticeably as far as measures based on average loss are considered, our empirical results show that for most of the examined datasets, the biggest performance gain is achieved when training the first

100 trees. However, the rate of convergence may be influenced by other hyperparameters of the RF. For example lower sample size while taking bootstrap samples for each tree, bigger constraints on the tree depth or more variables lead to less correlated trees and hence more trees are needed to reach convergence.

One could also think of an automatic break criterion which stops the training automatically according to the convergence of the OOB curves. For example, training could be stopped if the last T_{last} trees did not improve performance by more than Δ, where T_{last} and Δ are parameters that should be fixed by the user as a compromise between performance and computation time. Note that, if variable importances are computed, it may be recommended to also consider their convergence. This issue also requires more research.

Acknowledgments

We would like to thank Alexander Dürre for useful comments on the approximation of the logarithmic loss and Jenny Lee for language editing.

References

Ranjan Kumar Barman, Sudipto Saha, and Santasabuj Das. Prediction of interactions between viral and host proteins using supervised machine learning methods. *PLOS ONE*, 9(11):1–10, 2014.

Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016. R package version 2.9.

Anne-Laure Boulesteix, Rory Wilson, and Alexander Hapfelmeier. Towards evidence-based computational statistics: lessons from clinical research on the role and design of real-data benchmark studies. *BMC Medical Research Methodology*, 17(1):138, 2017.

Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996a.

Leo Breiman. Out-of-bag estimation. *Technical report, Statistics Department, University of California 1996*, 1996b.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

Giuseppe Casalicchio, Jakob Bossek, Michel Lang, Dominik Kirchhoff, Pascal Kerschke, Benjamin Hofner, Heidi Seibold, Joaquin Vanschoren, and Bernd Bischl. OpenML: An R package to connect to the machine learning platform OpenML. *Computational Statistics*, 32(3):1–15, 2017.

César Ferri, José Hernández-Orallo, and R Modroiu. An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, 30(1):27–38, 2009.

Jerome H Friedman. On bias, variance, 0/1-loss, and the curse-of-dimensionality. *Data Mining and Knowledge Discovery*, 1(1):55–77, 1997.

Jerome H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.

David J Hand and Robert J Till. A simple generalisation of the area under the ROC curve for multiple class classification problems. *Machine Learning*, 45(2):171–186, 2001.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

Daniel Hernández-Lobato, Gonzalo Martínez-Muñoz, and Alberto Suárez. How large should ensembles of classifiers be? *Pattern Recognition*, 46(5):1323–1336, 2013.

Torsten Hothorn, Kurt Hornik, and Achim Zeileis. Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, 15 (3):651–674, 2006.

Michel Lang, Bernd Bischl, and Dirk Surmann. batchtools: Tools for R to work on batch systems. *The Journal of Open Source Software*, 2(10), 2017.

Patrice Latinne, Olivier Debeir, and Christine Decaestecker. Limiting the number of trees in random forests. In *International Workshop on Multiple Classifier Systems*, pages 178–187. Springer, 2001.

Andy Liaw and Matthew Wiener. Classification and regression by randomForest. *R News*, 2(3):18–22, 2002. R package version 4.6-12.

Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas. How many trees in a random forest? In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 154–168. Springer, 2012.

Piergiorgio Palla and Giuliano Armano. *RFmarkerDetector: Multivariate Analysis of Metabolomics Data using Random Forests*, 2016. R package version 1.0.1.

Arvind Raghu, Praveen Devarsetty, Peiris David, Tarassenko Lionel, and Clifford Gari. Implications of cardiovascular disease risk assessment using the who/ish risk prediction charts in rural india. *PLOS ONE*, 10(8):1–13, 2015.

Peter J. Rousseeuw. Least median of squares regression. *Journal of the American Statistical Association*, 79(388):871–880, 1984.

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

Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis, and Torsten Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics*, 8(1):25, 2007.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

Marvin N. Wright. *ranger: A Fast Implementation of Random Forests*, 2016. R package version 0.6.0.

Received: 28 February 2018 | Revised: 21 November 2018 | Accepted: 12 December 2018
DOI: 10.1002/widm.1301

WILEY WIREs DATA MINING AND KNOWLEDGE DISCOVERY

ADVANCED REVIEW

Hyperparameters and tuning strategies for random forest

Philipp Probst[1] | Marvin N. Wright[2] | Anne-Laure Boulesteix[1]

[1]Institute for Medical Information Processing, Biometry und Epidemiology, Ludwig-Maximilians-Universität München, Munich, Germany

[2]Leibniz Institute for Prevention Research and Epidemiology – BIPS, Bremen, Germany

Correspondence
Philipp Probst, Institute for Medical Information Processing, Biometry und Epidemiology, Ludwig-Maximilians-Universität München, Marchioninistr. 15, 81377 Munich, Germany.
Email: probst@ibe.med.uni-muenchen.de

Funding information
Deutsche Forschungsgemeinschaft, Grant/Award Number: BO3139/2-3BO3139/6-1; Bundesministerium für Bildung und Forschung, Grant/Award Number: 01IS18036A

The random forest (RF) algorithm has several hyperparameters that have to be set by the user, for example, 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 presenting 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.

This article is categorized under:
Algorithmic Development > Biological Data Mining
Algorithmic Development > Statistics
Algorithmic Development > Hierarchies and Trees
Technologies > Machine Learning

KEYWORDS
ensemble, literature review, out-of-bag, performance evaluation, ranger, sequential model-based optimization, tuning parameter

1 | INTRODUCTION

The random forest (RF) algorithm first introduced by Breiman (2001) has now grown to a standard nonparametric 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, Janitza, Kruppa, & König, 2012) and others (Belgiu & Drăguţ, 2016; Biau & Scornet, 2016; Criminisi, Shotton, & Konukoglu, 2012; Ziegler & König, 2014). 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 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 (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 has 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, that is, 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 suboptimal 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, for example, the error rate or the area under the 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.

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.

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

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, 1,000
Splitting rule	Splitting criteria in the nodes	Gini impurity, p value, random

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 nonimportant 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, Heutte, and Adam (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, Polley, & Briggs, 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, Poggi, and Tuleau (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 & 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, Binder, & Boulesteix, 2016) and empirical results (Strobl, Boulesteix, Zeileis, & Hothorn, 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 & De Andres, 2006; Goldstein et al., 2011) but performance can potentially be improved by tuning it (Lin & 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 one 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, Hornik, & Zeileis, 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 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, Bender, Lorenzo Bermejo, and Strobl (2012) in the context of genetic data.

Furthermore, in the R package **randomForest** (Liaw & 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 & De Andres, 2006; Oshiro, Perez, & Baranauskas, 2012; Probst & Boulesteix, 2017; Scornet, 2018). Out-of-bag error curves (slightly) increasing with the number of trees are occasionally observed for certain error measures (see Probst & 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 behavior 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 & 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 (CIFs) introduced by Hothorn et al. (2006) and implemented in the R package **party** and in the newer package **partykit** (Hothorn & 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, Dankowski, and Ziegler (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, Ernst, & Wehenkel, 2006). To this end, only a randomly selected subset of possible splitting values is 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, in general 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 CIFs 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 nonlinear and interaction effects without specifying these a priori (Wright, Ziegler, & König, 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, Bender, et al., 2012) 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, Poggi, & Tuleau-Malot, 2010; Goldstein et al., 2011), than for the simple prediction purpose. Lunetta, Hayward, Segal, and Van Eerdewegh (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 CIFs 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 CIF. 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 CIFs. 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, Bischl, & Boulesteix, 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 with 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. Afterward, 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, Cernadas, Barro, & Amorim, 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, Bernau, Boulesteix, & De Bin, 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), that is, approximate the performance of the RF on independent data reasonably well. A bias can be observed in special data situations (see Janitza &

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 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, Hernández-Orallo, and Modroiu (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, Yuan, Balaprakash, & Stützle, 2010), general simulated annealing (Bohachevsky, Johnson, & Stein, 1986) or sequential model-based optimization (SMBO) (Hutter, Hoos, & Leyton-Brown, 2011; Jones, Schonlau, & Welch, 1998) 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.

3.4 | 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.

3.5 | 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, Richter, 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., fivefold cross validation) and a constrained hyperparameter space on which the tuning should be executed.
2. Create an initial design, that is, 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, for example, 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.

3.6 | 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 SMBO 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 (MMCE). 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 three 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.

3.7 | 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 & Ziegler, 2017), **mlrMBO** (Bischl, Richter, et al., 2017) and **mlr** (Bischl et al., 2016). The main function `tuneRanger` of the package works internally as follows:

- 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, Tibshirani, and Friedman (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 & 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): https://mlr.mlr-org.com/articles/tutorial/measures.html.
- The final recommended hyperparameter setting is calculated by taking the best 5% 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.

3.8 | 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 (Vanschoren, van Rijn, Bischl, & Torgo, 2013). 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)$data
monk.task = makeClassifTask(data = monk_data_1, target = “class”)
# Estimate runtime
estimateTimeTuneRanger(monk.task)
# Approximated time for tuning: 1M 13S
set.seed(123)
# Tuning
res = tuneRanger(monk.task, measure = list(multiclass.brier), num.trees = 1000,
num.threads = 2, iters = 70, iters.warmup = 30)
res
# Recommended parameter settings:
# mtry min.node.size sample.fraction
# 1 6 2 0.8988154
# Results:
# multiclass.brier exec.time
# 1 0.006925637 0.2938
#
# Ranger Model with the new tuned hyperparameters
res$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 five times repeated fivefold cross validation to compare it with a standard random forest with 1,000 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.

```
# little benchmark
lrn = makeLearner(“classif.ranger”, num.trees = 1,000, 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
```

3.9 | 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% of the iterations) is used to train a RF model, which can be accessed via the list element `model` in the final outcome.

3.10 | 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.

3.10.1 | 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 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 & 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).

3.10.2 | 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, Casalicchio, 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, Casalicchio, 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 min, 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 fivefold cross validation and repeat it 10 times and for the big we just perform a fivefold cross validation. We compare the algorithms by the average of the evaluation measures 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 two datasets for ***mlrHyperopt***, on four datasets for ***caret***, and on three 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, Schiffner, and Weihs (2013).

3.10.3 | Results

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

3.11 | 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 six 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

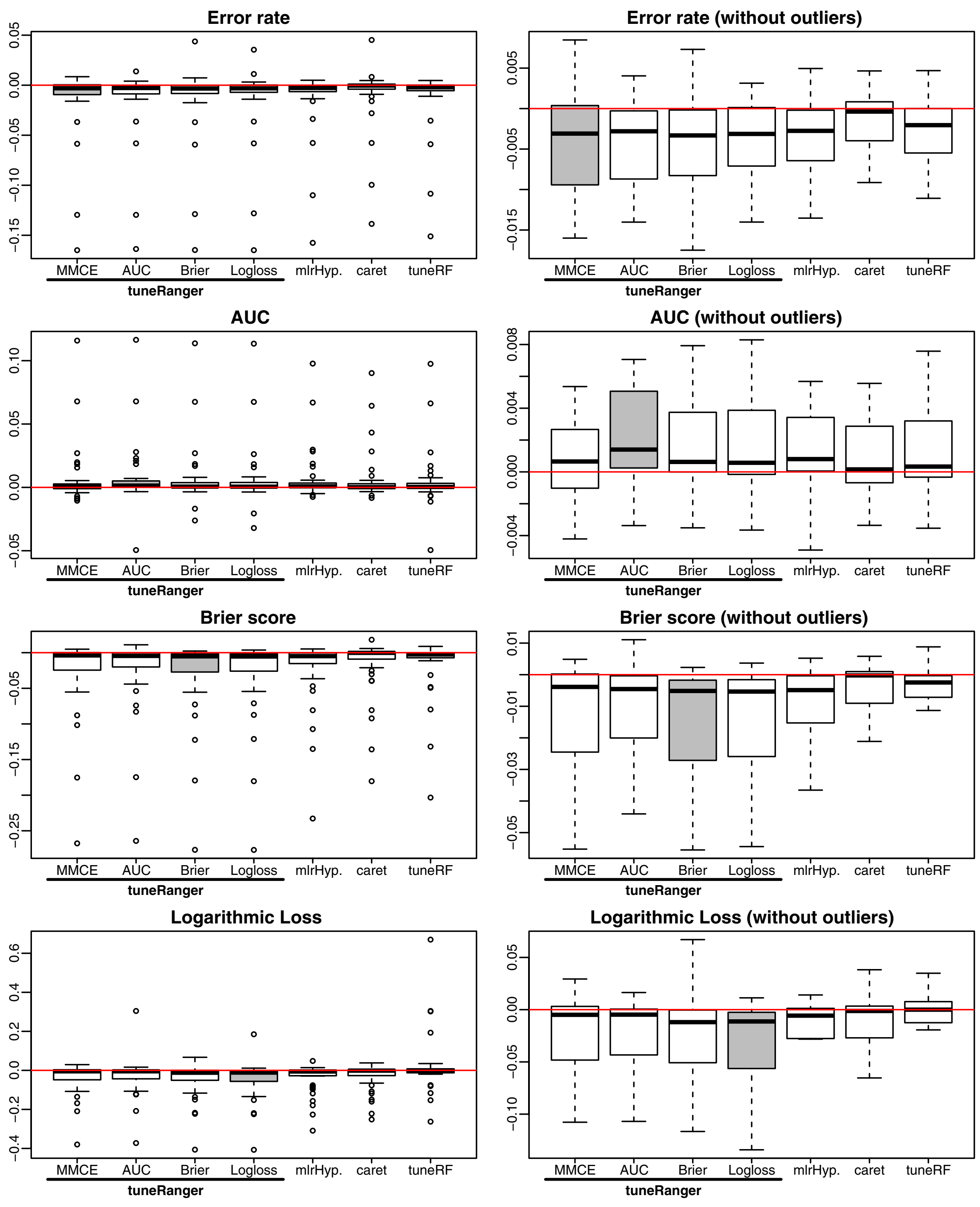

FIGURE 1 Boxplots of performance differences to ***ranger default***. On the left side the boxplots with outliers are depicted and on the right side the same plots without outliers. For the error rate, the brier score and the logarithmic loss, low values are better, while for the area under the curve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

(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 noninformative predictor variables. The default *mtry* value of 22 is much too low, because too many noninformative 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 three 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.

3.12 | 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.

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,

TABLE 2 Average performance results of the different algorithms

	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	2,713.2438
caret	0.0972	0.9190	0.1439	0.2423	1,216.2770
tuneRF	0.0942	0.9174	0.1448	0.2929	862.9917
Ranger default	0.1054	0.9128	0.1604	0.2733	3.8607

Note. Training runtime in seconds. AUC, area under the curve; MMCE, mean missclassification error.

TABLE 3 Average rank 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

Note. Training runtime in seconds. AUC, area under the curve; MMCE, mean missclassification error.

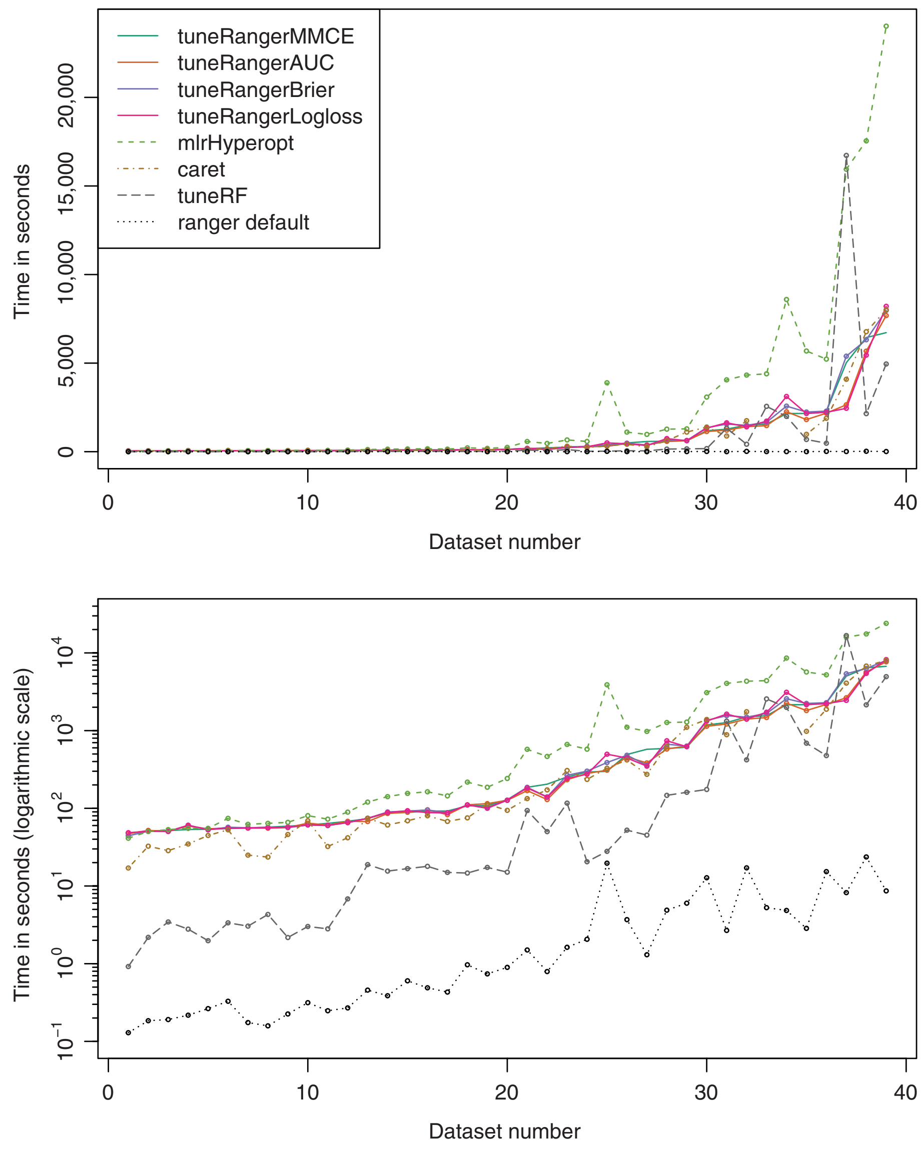

FIGURE 2 Average runtime of the different algorithms on different datasets (upper plot: Unscaled, lower plot: Logarithmic scale). The datasets are ordered by the average runtime of the ***tuneRangerMMCE*** algorithm

although there is one dataset for which it takes longer than all the other datasets. 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.

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 the 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 CIFs (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, Binder, Abrahamowicz, and Sauerbrei (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, Wilson, and Hapfelmeier (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 as the very recent article by Scornet (2018). 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, Rossi, Vanschoren, Bischl, & Carvalho, 2015). Out of existing tuning algorithms, we suggest to use 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 R 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.

ACKNOWLEDGMENT

This article is funded by German Research Foundation (DFG), grants BO3139/6-1 and BO3139/2-3 to A.L.B and by Bundesministerium für Bildung und Forschung, Grant/Award Number: 01IS18036A" to A.L.B.

CONFLICT OF INTEREST

The authors have declared no conflicts of interest for this article.

REFERENCES

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

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

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

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

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

Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., Studerus, E., … 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., & Lang, M. (2017). mlrMBO: A modular framework for model-based optimization of expensive black-box functions. *ArXiv preprint arXiv:1703.03373.* Retreived from https://arxiv.org/abs/1703.03373

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

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

Boulesteix, A.-L., Bender, A., Lorenzo Bermejo, J., & Strobl, C. (2012). 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., & 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., & König, I. R. (2012). Overview of random forest methodology and practical guidance with emphasis on computational biology and bioinformatics. *WIREs: Data Mining and Knowledge Discovery*, *2*, 493–507.

Boulesteix, A.-L., Wilson, R., & 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.* Technical Report, UC Berkeley, Department of Statistics.

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

Casalicchio, G., Bossek, J., Lang, M., Kirchhoff, D., Kerschke, P., Hofner, B., … 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. (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., & 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., & 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., & Modroiu, R. (2009). An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, *30*, 27–38.

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

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

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

Goldstein, B. A., Polley, E. C., & Briggs, F. (2011). Random forests for genetic association studies. *Statistical Applications in Genetics and Molecular Biology*, *10*, 32.

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

Hastie, T., Tibshirani, R., & Friedman, J. (2001). *The Elements of Statistical Learning. Springer series in statistics*. New York, NY: Springer New York Inc.

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

Hothorn, T., & 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 and Heidelberg, Germany: Springer Berlin Heidelberg.

Janitza, S., Binder, H., & 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., & Hornung, R. (2018). On the overestimation of random forest's out-of-bag error. *PLoS One*, *13*, e0201904.

Jones, D. R., Schonlau, M., & 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., & Wiener, M. (2002). Classification and regression by randomForest. *R News*, *2*, 18–22.

Lin, Y., & 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., & 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., & 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* (pp. 1–8). IEEE.

Martínez-Muñoz, G., & 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., & 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*.

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

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

Probst, P., & Boulesteix, A.-L. (2017). To tune or not to tune the number of trees in a random forest? *Journal of Machine Learning Research*, *18*, 1–18.

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

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

Scornet, E. (2018). Tuning parameters in random forests. *ESAIM: Proceedings And Surveys*, *60*, 144–162.

Segal, M. R. (2004). Machine learning benchmarks and random forest regression. *UCSF: Center for Bioinformatics and Molecular Biostatistics*. Retrieved from https://escholarship.org/uc/item/35x3v9t4

Seibold, H., Bernau, C., Boulesteix, A.-L., & 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. (2010). To explain or to predict? *Statistical Science*, *25*, 289–310.

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

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

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

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

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

How to cite this article: Probst P, Wright MN, Boulesteix A-L. Hyperparameters and tuning strategies for random forest. *WIREs Data Mining Knowl Discov*. 2019;e1301. https://doi.org/10.1002/widm.1301

Journal of Machine Learning Research 18 (2018) 1-32 Submitted 7/18; Revised 2/19; Published 3/19

Tunability: Importance of Hyperparameters of Machine Learning Algorithms

Philipp Probst PROBST@IBE.MED.UNI-MUENCHEN.DE
Institute for Medical Information Processing, Biometry and Epidemiology, LMU Munich
Marchioninistr. 15, 81377 München, Germany

Anne-Laure Boulesteix BOULESTEIX@IBE.MED.UNI-MUENCHEN.DE
Institute for Medical Information Processing, Biometry and Epidemiology, LMU Munich
Marchioninistr. 15, 81377 München, Germany

Bernd Bischl BERND.BISCHL@STAT.UNI-MUENCHEN
Department of Statistics, LMU Munich
Ludwigstraße 33, 80539 München, Germany

Editor: Ryan Adams

Abstract

Modern supervised machine learning algorithms involve hyperparameters that have to be set before running them. Options for setting hyperparameters are default values from the software package, manual configuration by the user or configuring them for optimal predictive performance by a tuning procedure. The goal of this paper is two-fold. Firstly, we formalize the problem of tuning from a statistical point of view, define data-based defaults and suggest general measures quantifying the tunability of hyperparameters of algorithms. Secondly, we conduct a large-scale benchmarking study based on 38 datasets from the OpenML platform and six common machine learning algorithms. We apply our measures to assess the tunability of their parameters. Our results yield default values for hyperparameters and enable users to decide whether it is worth conducting a possibly time consuming tuning strategy, to focus on the most important hyperparameters and to choose adequate hyperparameter spaces for tuning.

Keywords: machine learning, supervised learning, classification, hyperparameters, tuning, meta-learning

1. Introduction

Machine learning (ML) algorithms such as gradient boosting, random forest and neural networks for regression and classification involve a number of *hyperparameters* that have to be set before running them. In contrast to direct, first-level model parameters, which are determined during training, these second-level *tuning parameters* often have to be carefully optimized to achieve maximal performance. A related problem exists in many other algorithmic areas, e.g., control parameters in evolutionary algorithms (Eiben and Smit, 2011).

In order to select an appropriate hyperparameter *configuration* for a specific dataset at hand, users of ML algorithms can resort to default values of hyperparameters that are specified in implementing software packages or manually configure them, for example, based on recommendations from the literature, experience or trial-and-error.

Alternatively, one can use hyperparameter *tuning strategies*, which are data-dependent, second-level optimization procedures (Guyon et al., 2010), which try to minimize the expected generalization error of the inducing algorithm over a hyperparameter search space of considered candidate configurations, usually by evaluating predictions on an independent test set, or by running a resampling scheme such as cross-validation (Bischl et al., 2012). For a recent overview of tuning strategies, see, e.g., Luo (2016). These search strategies range from simple grid or random search (Bergstra and Bengio, 2012) to more complex, iterative procedures such as Bayesian optimization (Hutter et al., 2011; Snoek et al., 2012; Bischl et al., 2017b) or iterated F-racing (Birattari et al., 2010; Lang et al., 2017).

In addition to selecting an efficient tuning strategy, the set of tunable hyperparameters and their corresponding ranges, scales and potential prior distributions for subsequent sampling have to be determined by the user. Some hyperparameters might be safely set to default values, if they work well across many different scenarios. Wrong decisions in these areas can inhibit either the quality of the resulting model or at the very least the efficiency and fast convergence of the tuning procedure. This creates a burden for:

1. ML users—Which hyperparameters should be tuned and in which ranges?

2. Designers of ML algorithms—How do I define robust defaults?

We argue that many users, especially if they do not have years of practical experience in the field, here often rely on heuristics or spurious knowledge. It should also be noted that designers of fully automated tuning frameworks face at least very similar problems. It is not clear how these questions should be addressed in a data-dependent, automated, optimal and objective manner. In other words, the scientific community not only misses answers to these questions for many algorithms but also a systematic framework, methods and criteria, which are required to answer these questions.

With the present paper we aim at filling this gap and formalize the problem of parameter tuning from a statistical point of view, in order to simplify the tuning process for less experienced users and to optimize decision making for more advanced processes.

After presenting related literature in Section 2, we define theoretical measures for assessing the impact of tuning in Section 3. For this purpose we (i) define the concept of *default hyperparameters*, (ii) suggest measures for quantifiying the tunability of the whole algorithm and specific hyperparameters based on the differences between the performance of default hyperparameters and the performance of the hyperparameters when this hyperparameter is set to an optimal value. Then we (iii) address the tunability of hyperparameter combinations and joint gains, (iv) provide theoretical definitions for an appropriate hyperparameter space on which tuning should be executed and (v) propose procedures to estimate these quantities based on the results of a benchmark study with random hyperparameter configurations with the help of surrogate models. In sections 4 and 5 we illustrate these concepts and methods through an application. For this purpose we use benchmark results of six machine learning algorithms with different hyperparameters which were evaluated on 38 datasets from the OpenML platform. Finally, in the last Section 6 we conclude and discuss the results.

2. Related Literature

To the best of our knowledge, only a limited amount of articles address the problem of tunability and generation of tuning search spaces. Bergstra and Bengio (2012) compute the relevance of the hyperparameters of neural networks and conclude that some are important on all datasets, while others are only important on some datasets. Their conclusion is primarily visual and used as an argument for why random search works better than grid search when tuning neural networks.

A specific study for decision trees was conducted by Mantovani et al. (2016) who apply standard tuning techniques to decision trees on 102 datasets and calculate differences of accuracy between the tuned algorithm and the algorithm with default hyperparameter settings.

A different approach is proposed by Hutter et al. (2013), which aims at identifying the most important hyperparameters via forward selection. In the same vein, Fawcett and Hoos (2016) present an *ablation analysis* technique, which aims at identifying the hyperparameters that contribute the most to improved performance after tuning. For each of the considered hyperparameters, they compute the performance gain that can be achieved by changing its value from the initial value to the value specified in the target configuration which was determined by the tuning strategy. This procedure is iterated in a greedy forward search.

A more general framework for measuring the importance of single hyperparameters is presented by Hutter et al. (2014). After having used a tuning strategy such as sequential model-based optimization, a functional ANOVA approach is used for measuring the importance of hyperparameters.

These works concentrate on the importance of hyperparameters on single datasets, mainly to retrospectively explain what happened during an already concluded tuning process. Our main focus is the generalization across multiple datasets in order to facilitate better general understanding of hyperparameter effects and better decision making for future experiments. In a recent paper van Rijn and Hutter (2017) pose very similar questions to ours to assess the importance of hyperparameters across datasets. We compare it to our approach in Section 6.

Our framework is based on using surrogate models, also sometimes called empirical performance models, which allow estimating the performance of arbitrary hyperparameter configurations based on a limited number of prior experiments. The idea of surrogate models is far from new (Audet et al., 2000), as it constitutes the central idea of Bayesian optimization for hyperparameter search but is also used, for example, in Biedenkapp et al. (2017) for increasing the speed of an ablation analysis and by Eggensperger et al. (2018) for speeding up the benchmarking of tuning strategies.

3. Methods for Estimation of Defaults, Tunability and Ranges

In this section we introduce theoretical definitions for defaults, tunability and tuning ranges, then describe how to estimate them and finally discuss the topic of reparametrization.

3.1. General Notation

Consider a target variable Y, a feature vector X, and an unknown joint distribution P on (X, Y), from which we have sampled a dataset $\mathcal{T}$ of n observations. A machine learning (ML) algorithm now learns the functional relationship between X and Y by producing a prediction model $\hat{f}(X, \theta)$, controlled by the k-dimensional hyperparameter configuration $\theta = (\theta_1, ..., \theta_k)$ from the hyperparameter search space $\Theta = \Theta_1 \times ... \times \Theta_k$. In order to measure prediction performance pointwise between the true label Y and its prediction $\hat{f}(X, \theta)$, we define a loss function $L(Y, \hat{f}(X, \theta))$. We are naturally interested in estimating the expected risk of the inducing algorithm, w.r.t. θ on new data, also sampled from $\mathcal{P}$: $R(\theta) = E(L(Y, \hat{f}(X, \theta))|\mathcal{P})$. This mapping encodes, given a certain data distribution, a certain learning algorithm and a certain performance measure, the numerical quality for any hyperparameter configuration θ. Given m different datasets (or data distributions) $\mathcal{P}_1, ..., \mathcal{P}_m$, we arrive at m hyperparameter risk mappings

$$R^{(j)}(\theta) := E(L(Y, \hat{f}(X, \theta))|\mathcal{P}_j), \qquad j = 1, ..., m. \tag{1}$$

For now, we assume all $R^{(j)}(\theta)$ to be known, and show how to estimate them in Section 3.7.

3.2. Optimal Configuration per Dataset and Optimal Defaults

We first define the best hyperparameter configuration for dataset j as

$$\theta^{(j)\star} := \arg\min_{\theta \in \Theta} R^{(j)}(\theta). \tag{2}$$

Defaults settings are supposed to work well across many different datasets and are usually provided by software packages, in an often ad hoc or heuristic manner. We propose to define an optimal default configuration, based on an extensive number of empirical experiments on m different benchmark datasets, by

$$\theta^{\star} := \arg\min_{\theta \in \Theta} g(R^{(1)}(\theta), ..., R^{(m)}(\theta)). \tag{3}$$

Here, g is a summary function that has to be specified. Selecting the mean (or median as a more robust candidate) would imply minimizing the average (or median) risk over all datasets.

The measures $R^{(j)}(\theta)$ could potentially be scaled appropriately beforehand in order to make them more commensurable between datasets, e.g., one could scale all $R^{(j)}(\theta)$ to $[0, 1]$ by subtracting the result of a very simple baseline like a featureless dummy predictor and dividing this difference by the absolute difference between the risk of the best possible result (as an approximation of the Bayes error) and the result of the very simple baseline predictor. Or one could produce a statistical z-score by subtracting the mean and dividing by the standard deviation from all experimental results on the same dataset (Feurer et al., 2018).

The appropriateness of the scaling highly depends on the performance measure that is used. One could, for example, argue that the AUC does not have to be scaled by using the probabilistic interpretation of the AUC. Given a randomly chosen observation x belonging to class 1, and a randomly chosen observation x' belonging to class 0, the AUC is the probability that the evaluated classification algorithm will assign a higher score to x than to x'. Thus, an improvement from 0.5 to 0.6 on one dataset could be seen as equally important to an improvement from 0.8 to 0.9 on another dataset. On the other hand, averaging the mean squared error on several datasets does not make a lot of sense, as the scale of the outcome of different regression problems can be very different. Then scaling or using another measure such as the R^2 is reasonable. As our main risk measure is the AUC, we do not use any scaling.[1]

3.3. Measuring Overall Tunability of a ML Algorithm

A general measure of the tunability of an algorithm per dataset can then be computed based on the difference between the risk of an overall reference configuration (e.g., either the software defaults or definition (3)) and the risk of the best possible configuration on that dataset:

$$d^{(j)} := R^{(j)}(\theta^\star) - R^{(j)}(\theta^{(j)\star}), \text{ for } j = 1, ..., m. \tag{4}$$

For each algorithm, this gives rise to an empirical distribution of performance differences over datasets, which might be directly visualized or summarized to an aggregated tunability measure d by using mean, median or quantiles.

3.4. Measuring Tunability of a Specific Hyperparameter

The best hyperparameter value for one parameter i on dataset j, when all other parameters are set to defaults from $\theta^\star := (\theta_1^\star, ..., \theta_k^\star)$, is denoted by

$$\theta_i^{(j)\star} := \underset{\theta \in \Theta, \theta_l = \theta_l^\star \forall l \neq i}{\arg\min} R^{(j)}(\theta). \tag{5}$$

A natural measure for tunability of the i-th parameter on dataset j is then the difference in risk between the above and our default reference configuration:

$$d_i^{(j)} := R^{(j)}(\theta^\star) - R^{(j)}(\theta_i^{(j)\star}), \text{ for } j = 1, ..., m, i = 1, ..., k. \tag{6}$$

Furthermore, we define $d_i^{(j),\mathrm{rel}} = \frac{d_i^{(j)}}{d^{(j)}}$ as the fraction of performance gain, when we only tune parameter i compared to tuning the complete algorithm, on dataset j. Again, one can calculate the mean, the median or quantiles of these two differences over the n datasets, to get a notion of the overall tunability d_i of this parameter.

3.5. Tunability of Hyperparamater Combinations and Joint Gains

Let us now consider two hyperparameters indexed as i_1 and i_2. To measure the tunability with respect to these two parameters, we define

$$\theta_{i_1,i_2}^{(j)\star} := \underset{\theta \in \Theta, \theta_l = \theta_l^\star \forall l \notin \{i_1, i_2\}}{\arg\min} R^{(j)}(\theta), \tag{7}$$

1. We also tried out normalization (z-score) and got qualitatively similar results to the non-normalized results presented in Section 5.

i.e., the θ-vector containing the default values for all hyperparameters other than i_1 and i_2, and the optimal combination of values for the i_1-th and i_2-th components of θ.

Analogously to the previous section, we can now define the tunability of the set (i_1, i_2) as the gain over the reference default on dataset j as

$$d^{(j)}_{i_1,i_2} := R^{(j)}(\theta^*) - R^{(j)}(\theta^{(j)\star}_{i_1,i_2}). \tag{8}$$

The joint gain which can be expected when tuning not only one of the two hyperparameters individually, but both of them jointly, on a dataset j, can be expressed by

$$g^{(j)}_{i_1,i_2} := \min\{(R^{(j)}(\theta^{(j)\star}_{i_1})), (R^{(j)}(\theta^{(j)\star}_{i_2}))\} - R^{(j)}(\theta^{(j)\star}_{i_1,i_2}). \tag{9}$$

Furthermore, one could be interested in whether this joint gain could simply be reached by tuning both parameters i_1 and i_2 in a univariate fashion sequentially, either in the order $i_1 \to i_2$ or $i_2 \to i_1$, and what order would be preferable. For this purpose one could compare the risk of the hyperparameter value that results when tuning them together $R^{(j)}(\theta^{(j)\star}_{i_1,i_2})$ with the risks of the hyperparameter values that are obtained when tuning them sequentially, that means $R^{(j)}(\theta^{(j)\star}_{i_1\to i_2})$ or $R^{(j)}(\theta^{(j)\star}_{i_2\to i_1})$, which is done for example in Waldron et al. (2011).

Again, all these measures should be summarized across datasets, resulting in d_{i_1,i_2} and g_{i_1,i_2}. Of course, these approaches can be further generalized by considering combinations of more than two parameters.

3.6. Optimal Hyperparameter Ranges for Tuning

A reasonable hyperparameter space $\Theta^\star$ for tuning should include the optimal configuration $\theta^{(j)\star}$ for dataset j with high probability. We denote the p-quantile of the distribution of one parameter regarding the best hyperparameters on each dataset $(\theta^{(1)\star})_i, ..., (\theta^{(m)\star})_i$ as $q_{i,p}$. The hyperparameter tuning space can then be defined as

$$\Theta^\star := \{\theta \in \Theta | \forall i \in \{1, ..., k\} : \theta_i \geq q_{i,p_1} \wedge \theta_i \leq q_{i,p_2}\}, \tag{10}$$

with p_1 and p_2 being quantiles which can be set for example to the 5 % quantile and the 95 % quantile. This avoids focusing too much on outlier datasets and makes the definition of the space independent from the number of datasets.

The definition above is only valid for numerical hyperparameters. In case of categorical variables one could use similar rules, for example only including hyperparameter values that were at least once or in at least 10 % of the datasets the best possible hyperparameter setting.

3.7. Practical Estimation

In order to practically apply the previously defined concepts, two remaining issues need to be addressed: a) We need to discuss how to obtain $R^{(j)}(\theta)$; and b) in (2) and (3) a multivariate optimization problem (the minimization) needs to be solved.[2]

2. All other previous optimization problems are univariate or two-dimensional and can simply be addressed by simple techniques such as a fine grid search.

For a) we estimate $R^{(j)}(\theta)$ by using surrogate models $\hat{R}^{(j)}(\theta)$, and replace the original quantity by its estimator in all previous formulas. Surrogate models for each dataset j are based on a meta dataset. This is created by evaluating a large number of configurations of the respective ML method. The surrogate regression model then learns to map a hyperparameter configuration to estimated performance. For b) we solve the optimization problem—now cheap to evaluate, because of the surrogate models—through black-box optimization.

3.8. Reparametrization

All tunability measures mentioned above can possibly depend on and be influenced by a reparametrization of hyperparameters. For example, in the case of the elastic net the parameters λ and α could be reparametrized as $\lambda_1 = \alpha\lambda$ and $\lambda_2 = (1-\alpha)\lambda$. Formally, such a reparametrization could be defined as a (bijective) function $\phi : \Theta \to \tilde{\Theta}$, such that $\phi(\theta)$ maps an original configuration θ to a new representation $\tilde{\theta} = \phi(\theta)$ from $\tilde{\Theta}$, in a one-to-one manner. Then defaults (calculated by the approach in Section 3.2) are naturally transformed via $\tilde{\theta}^\star = \phi(\theta^\star)$ into the new space $\tilde{\Theta}$, but will stay logically the same. Moreover, the general tunability of the algorithm does (obviously) not change. Depending on the parameters that are involved in the reparametrization, the tunability of the parameters can change. If, for example, only one parameter is involved, all tunabilities remain the same. If two or more parameters are involved, the single tunabilities of the parameters could change but the tunability of the set of the transformed parameters remains the same.

One might define a reparametrization as ideal (in the sense of simplified tuning) if the tunability is concentrated on one (or only few) hyperparameter(s), so that only this parameter has to be optimized and all remaining hyperparameters can remain at their (optimal) default values, reducing a multivariate optimization problem to a 1-dimensional or at least lower dimensional one. Using the definition above, this would imply that the joint gain of the new parameter(s) is (close to) 0. For example, imagine that the optimal hyperparameter values per dataset of two hyperparameters θ_1 and θ_2 lie on the line of equation $\theta_1 = \theta_2$. A useful reparametrization would then be $\tilde{\theta}_1 = \theta_1 + \theta_2$ and the orthogonal $\tilde{\theta}_2 = \theta_1 - \theta_2$. It would then only be necessary to tune $\tilde{\theta}_1$, while $\tilde{\theta}_2$ would be set to the default value 0.

A more general formulation is possible if we use the definition of 3.4. We could, for example, search for a bijective and invertible function $\phi^\star(.)$, across a certain parameterized function space, such that the mean tunability is concentrated on and therefore maximal for the first parameter and minimal for the other parameters, i.e.:

$$\phi^\star := \arg\min_{\phi \in \Phi} \frac{1}{m} \sum_{j=1}^{m} \min_{\tilde{\theta} \in \tilde{\Theta}, \tilde{\theta}_l = \tilde{\theta}_l^\star \forall l \neq 1} R^{(j)}\left(\phi^{-1}\left(\tilde{\theta}\right)\right). \tag{11}$$

We could select a restricted function space for ϕ, e.g., restrict ourselves to the space of all linear (invertible) transformations $\{\phi : \mathbb{R}^k \to \mathbb{R}^k | \phi(x) = Ax, A \in \mathbb{R}^{k \times k}, \det(A) \neq 0\}$. If concentrating the whole tunability on only one parameter is not possible, we could try a similar approach by concentrating it on a combination of two hyperparameters.

Note that such a reparametrization is not always helpful. For example, imagine we have two binary parameters and transform them such that (i) one of them has 4 levels that correspond to all possible combinations of these two parameters and (ii) the other parameter is set to a fixed constant. This reparametrization would not be useful: all the tunability is contained in the first parameter, but there is no real advantage, as still four evaluations have to be executed in the tuning process to get the best hyperparameter combination.

Finally, note that it can also be useful to reparametrize a single hyperparameter for the purpose of tuning. Imagine, for example, that most of the optimal parameters on the different datasets are rather small and only a few are large. A transformation of this parameter such as a log-transformation may then be useful. This is very similar to using prior probabilities for tuning (based on results on previous datasets) which could be seen as a useful alternative to a reparametrization and which is already proposed in van Rijn and Hutter (2017).

4. Experimental Setup

In this section we give an overview about the experimental setup that is used for obtaining surrogate models, tunability measures and tuning spaces.

4.1. Datasets from the OpenML Platform

Recently, the OpenML project (Vanschoren et al., 2013) has been created as a flexible online platform that allows ML scientists to share their data, corresponding tasks and results of different ML algorithms. We use a specific subset of carefully curated classification datasets from the OpenML platform called *OpenML100* (Bischl et al., 2017a). For our study we only use the 38 binary classification tasks that do not contain any missing values.

4.2. ML Algorithms

The algorithms considered in this paper are common methods for supervised learning. We examine elastic net (`glmnet` R package), decision tree (`rpart`), k-nearest neighbors (`kknn`), support vector machine (`svm`), random forest (`ranger`) and gradient boosting (`xgboost`). For more details about the used software packages see Kühn et al. (2018b). An overview of their considered hyperparameters is displayed in Table 1, including respective data types, box-constraints and a potential transformation function.

In the case of `xgboost`, the underlying package only supports numerical features, so we opted for a dummy feature encoding for categorical features, which is performed internally by the underlying packages for `svm` and `glmnet`.

Some hyperparameters of the algorithms are dependent on others. We take into account these dependencies and, for example, only sample a value for `gamma` for the support vector machine if the radial kernel was sampled beforehand.

4.3. Performance estimation

Several measures are regarded throughout this paper, either for evaluating our considered classification models that should be tuned, or for evaluating our surrogate regression models. As no optimal measure exists, we will compare several of them. In the classification case, we consider AUC, accuracy and Brier score. In the case of surrogate regression, we consider R^2, which is directly proportional to the regular mean squared error but scaled to [0,1] and explains the gain over a constant model estimating the overall mean of all data points. We also compute Kendall's tau as a ranking based measure for regression.

Algorithm	Hyperparameter	Type	Lower	Upper	Trafo
glmnet					
(Elastic net)	alpha	numeric	0	1	-
	lambda	numeric	-10	10	2^x
rpart					
(Decision tree)	cp	numeric	0	1	-
	maxdepth	integer	1	30	-
	minbucket	integer	1	60	-
	minsplit	integer	1	60	-
kknn	-	-			
(k-nearest neighbor)	k	integer	1	30	-
svm					
(Support vector machine)	kernel	discrete	-	-	-
	cost	numeric	-10	10	2^x
	gamma	numeric	-10	10	2^x
	degree	integer	2	5	-
ranger					
(Random forest)	num.trees	integer	1	2000	-
	replace	logical	-	-	-
	sample.fraction	numeric	0.1	1	-
	mtry	numeric	0	1	$x \cdot p$
	respect.unordered.factors	logical	-	-	-
	min.node.size	numeric	0	1	n^x
xgboost					
(Gradient boosting)	nrounds	integer	1	5000	-
	eta	numeric	-10	0	2^x
	subsample	numeric	0.1	1	-
	booster	discrete	-	-	-
	max_depth	integer	1	15	-
	min_child_weight	numeric	0	7	2^x
	colsample_bytree	numeric	0	1	-
	colsample_bylevel	numeric	0	1	-
	lambda	numeric	-10	10	2^x
	alpha	numeric	-10	10	2^x

Table 1: Hyperparameters of the algorithms. p refers to the number of variables and n to the number of observations. The columns *Lower* and *Upper* indicate the regions from which samples of these hyperparameters are drawn. The transformation function in the trafo column, if any, indicates how the values are transformed according to this function. The exponential transformation is applied to obtain more candidate values in regions with smaller hyperparameters because for these hyperparameters the performance differences between smaller values are potentially bigger than for bigger values. The `mtry` value in `ranger` that is drawn from $[0, 1]$ is transformed for each dataset separately. After having chosen the dataset, the value is multiplied by the number of variables and afterwards rounded up. Similarly, for the `min.node.size` the value x is transformed by the formula $[n^x]$ with n being the number of observations of the dataset, to obtain a positive integer values with higher probability for smaller values (the value is finally rounded to obtain integer values).

The performance estimation for the different hyperparameter experiments is computed through 10-fold cross-validation. For the comparison of surrogate models 10 times repeated 10-fold cross-validation is used.

4.4. Random Bot sampling strategy for meta data

To reliably estimate our surrogate models we need enough evaluated configurations per classifier and dataset. We sample these points from independent uniform distributions where the respective support for each parameter is displayed in Table 1. Here, *uniform* refers to the untransformed scale, so we sample uniformly from the interval [*Lower*, *Upper*] of Table 1.

In order to properly facilitate the automatic computation of a large database of hyperparameter experiments, we implemented a so called OpenML bot. In an embarrassingly parallel manner it chooses in each iteration a random dataset, a random classification algorithm, samples a random configuration and evaluates it via cross-validation. A subset of 500000 experiments for each algorithm and all datasets are used for our analysis here.[3] More technical details regarding the random bot, its setup and results can be obtained in Kühn et al. (2018b), furthermore, for simple and permanent access the results of the bot are stored in a figshare repository (Kühn et al., 2018a).

4.5. Optimizing Surrogates to Obtain Optimal Defaults

Random search is also used for our black-box optimization problems in Section 3.7. For the estimation of the defaults for each algorithm we randomly sample 100000 points in the hyperparameter space as defined in Table 1 and determine the configuration with the minimal average risk. The same strategy with 100000 random points is used to obtain the best hyperparameter setting on each dataset that is needed for the estimation of the tunability of an algorithm. For the estimation of the tunability of single hyperparameters we also use 100000 random points for each parameter, while for the tunability of combination of hyperparameters we only use 10000 random points to reduce runtime as this should be enough to cover 2-dimensional hyperparameter spaces.

Of course one has to be careful with overfitting here, as our optimal defaults are chosen with the help of the same datasets that are used to determine the performance. Therefore, we also evaluate our approach via a "10-fold cross-validation across datasets". Here, we repeatedly calculate the optimal defaults based on 90% "training datasets" and evaluate the package defaults and our optimal defaults—the latter induced from the training datasets—on the surrogate models of the remaining 10% "test datasets", and compare their difference in performance.

4.6. The Problem of Hyperparameter Dependency

Some parameters are dependent on other superordinate hyperparameters and are only relevant if the parameter value of this superordinate parameter was set to a specific value. For example `gamma` in `svm` only makes sense if the `kernel` was set to "radial" or `degree` only makes sense if the kernel was set to "polynomial". Some of these subordinate parameters might be invalid/inactive in the reference default configuration, rendering it impossible to

3. Only 30 experiments are used for each dataset for `kknn`, because we only consider the parameter k.

univariately tune them in order to compute their tunability score. In such a case we set the superordinate parameter to a value which makes the subordinate parameter active, compute the optimal defaults for the rest of the parameters and compute the tunability score for the subordinate parameter with these defaults.

4.7. Software Details

All our experiments are executed in R and are run through a combination of custom code from our random bot (Kühn et al., 2018b), the `OpenML` R package (Casalicchio et al., 2017), `mlr` (Bischl et al., 2016) and `batchtools` (Lang et al., 2017) for parallelization. All results are uploaded to the OpenML platform and there publicly available for further analysis. `mlr` is also used to compare and fit all surrogate regression models. The fully reproducible R code for all computations and analyses of our paper can be found on the github page: `https://github.com/PhilippPro/tunability`. We also provide an interactive shiny app under `https://philipppro.shinyapps.io/tunability/`, which displays all results of the following section in a potentially more convenient, interactive fashion and which can simply be accessed through a web browser.

5. Results and Discussion

We calculate all results for AUC, accuracy and Brier score but mainly discuss AUC results here. Tables and figures for the other measures can be accessed in the appendix and in our interactive shiny application.

5.1. Surrogate Models

We compare different possible regression models as candidates for our surrogate models: the linear model (`lm`), a simple decision tree (`rpart`), k nearest-neighbors (`kknn`) and random forest (`ranger`)[4] All algorithms are run with their default settings. We calculate 10 times repeated 10-fold cross-validated regression performance measures R^2 and Kendall's tau per dataset, and average these across all datasets.[5] Results for AUC are displayed in Figure 1. A good overall performance is achieved by `ranger` with qualitatively similar results for other classification performance measures (see Appendix). In the following we use random forest as surrogate model because it performs reasonably well and is already an established algorithm for surrogate models in the literature (Eggensperger et al., 2014; Hutter et al., 2013).

5.2. Optimal Defaults and Tunability

Table 2 displays our mean tunability results for the algorithms as defined in formula (4) w.r.t. package defaults (`Tun.P` column) and our optimal defaults (`Tun.O`). The distribution of the tunability values of the optimal defaults can be seen in Figure 2 in the modified

4. We also tried `cubist` (Kuhn et al., 2016), which provided good results but the algorithm had some technical problems for some combinations of datasets and algorithms. We did not include gaussian process which is one of the standard algorithms for surrogate models as it cannot handle categorical variables.
5. In case of `kknn` four datasets did not provide results for one of the surrogate models and were not used.

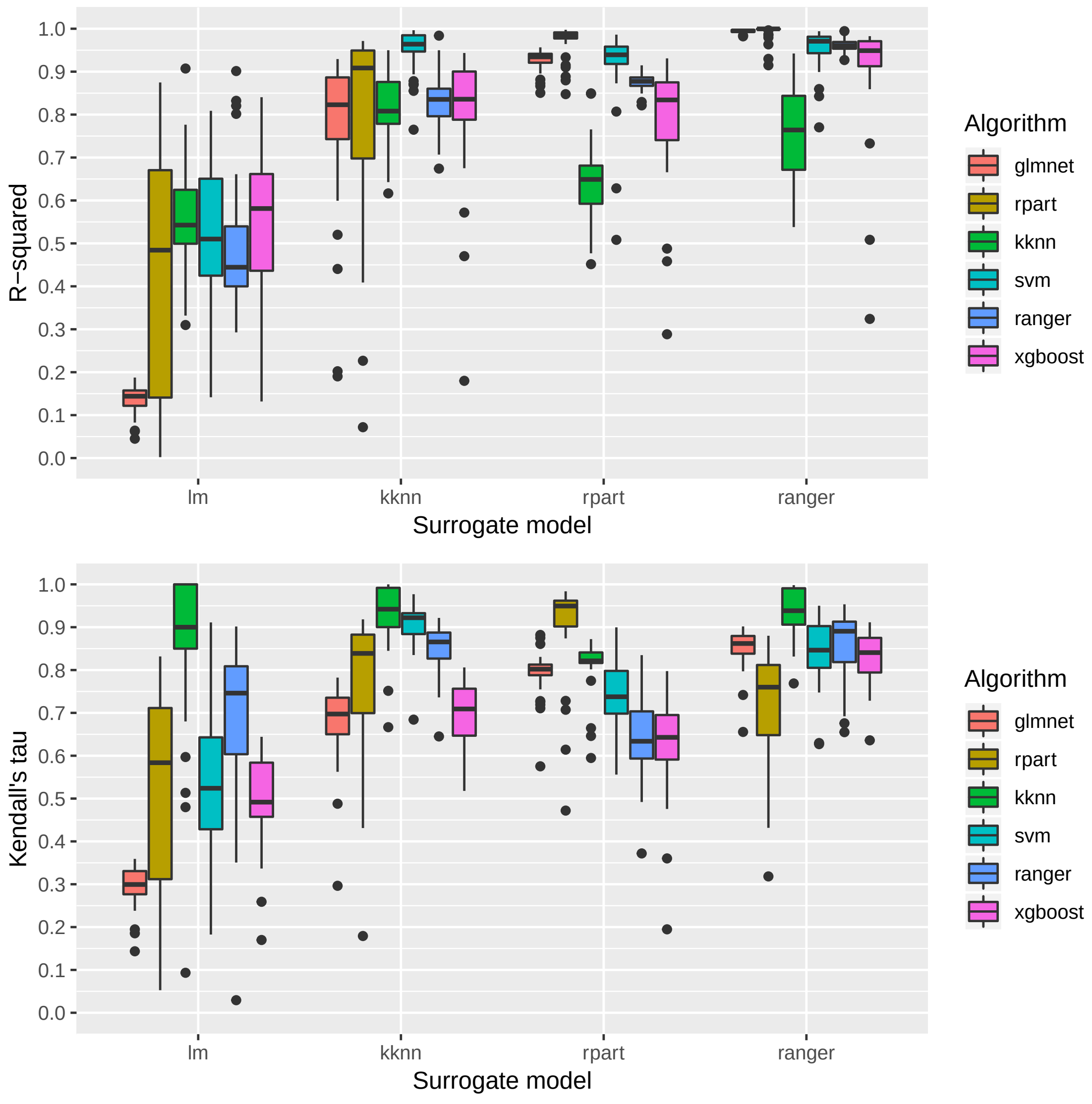

Figure 1: Average performances over the datasets of different surrogate models (target: AUC) for different algorithms (that were presented in 4.2). For an easier comparison of the surrogate models the same graph with exchanged x-axis and legend is available in the appendix in Figure 5.

Algorithm	Tun.P	Tun.O	Tun.O-CV	Improv	Impr-CV
glmnet	0.069 ± 0.019	0.024 ± 0.013	0.037 ± 0.015	0.045 ± 0.015	0.032 ± 0.015
rpart	0.038 ± 0.006	0.012 ± 0.004	0.016 ± 0.004	0.025 ± 0.006	0.022 ± 0.006
kknn	0.031 ± 0.006	0.006 ± 0.004	0.006 ± 0.004	0.025 ± 0.008	0.025 ± 0.008
svm	0.056 ± 0.011	0.042 ± 0.007	0.048 ± 0.008	0.014 ± 0.005	0.008 ± 0.007
ranger	0.010 ± 0.003	0.006 ± 0.001	0.007 ± 0.001	0.004 ± 0.003	0.003 ± 0.003
xgboost	0.043 ± 0.006	0.014 ± 0.006	0.017 ± 0.007	0.029 ± 0.003	0.026 ± 0.003

Table 2: Mean tunability (regarding AUC) with the package defaults (Tun.P) and the optimal defaults (Tun.O) as reference, cross-validated tunability (Tun.O-CV), average improvement (Improv) and cross-validated average improvement (Impr-CV) obtained by using optimal defaults compared to package defaults. The (cross-validated) improvement can be calculated by the (rounded) difference between Tun.P and Tun.O (Tun.O-CV). Standard error of the mean (SEM) is given behind the "±"-sign.

boxplots. Table 2 also displays the average improvement per algorithm when moving from package defaults to optimal defaults (`Improv`), which was positive overall. This also holds for `svm` and `ranger` although the package defaults are data dependent, which we currently cannot model (`gamma` $= 1/p$ for `svm` and `mtry` $= \sqrt{p}$ for `ranger`). As our optimal defaults are calculated based on all datasets, there is a risk of overfitting. So we perform a 5-fold cross-validation on dataset level, always calculating optimal defaults on $\frac{4}{5}$ of datasets and evaluating them on $\frac{1}{5}$ of the datasets. The results in the column `Impr-CV` in Table 2 show that the improvement compared to the package defaults is less pronounced but still positive for all algorithms.

From now on, when discussing tunability, we will only do this w.r.t. our optimal defaults.

Clearly, some algorithms such as `glmnet` and `svm` are much more tunable than the others, while `ranger` is the algorithm with the smallest tunability, which is in line with common knowledge in the web community. In the boxplots in Figure 2 for each ML algorithm, some values that are much bigger than the others are visible, which indicates that tuning has a much higher impact on some specific datasets.

5.3. Tunability of Specific Hyperparameters

In Table 3 the mean tunability (regarding the AUC) of single hyperparameters as defined in Equation (6) in Section 3.4 can be seen. Moreover, in Figure 3 the distributions of the tunability values of the hyperparameters are depicted in boxplots, which makes it possible to detect outliers and to examine skewness. The same results for the Brier score and accuracy can be found in the appendix. In the following analysis of our results, we will refer to tunability only with respect to optimal defaults.

For `glmnet` `lambda` seems to be more tunable than `alpha` regarding the AUC, especially for two datasets tuning seems to be crucial. For accuracy we observe the same pattern, while for Brier score `alpha` seems to be more tunable than `lambda` (see Figure 11 and Figure 13 in the appendix). We could not find any recommendation in the literature for

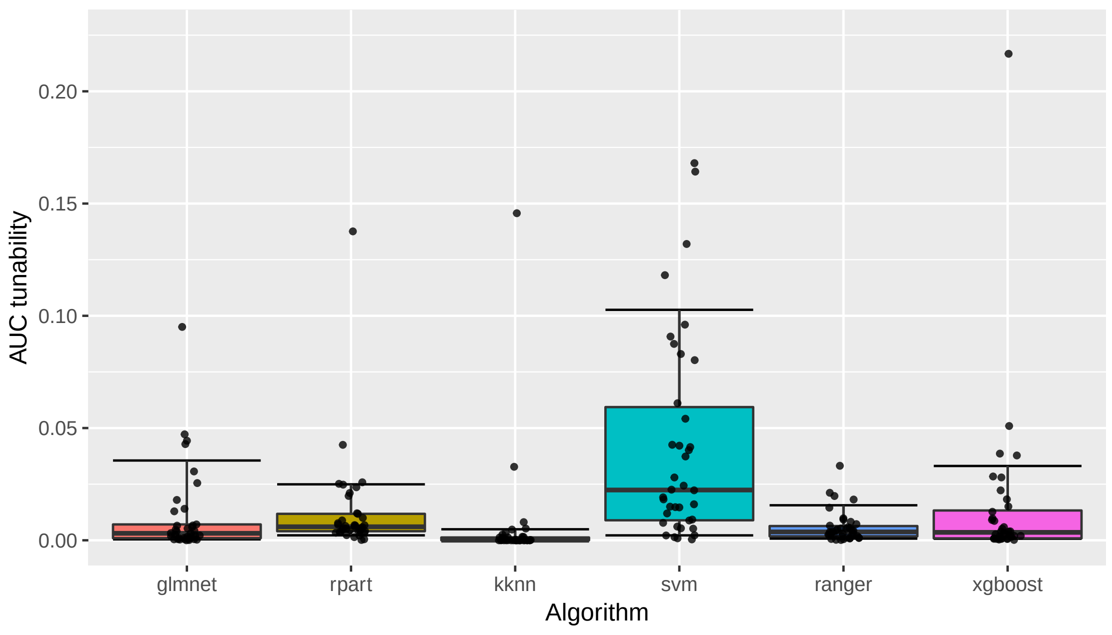

Figure 2: Boxplots of the tunabilities (AUC) of the different algorithms with respect to optimal defaults. The upper and lower whiskers (upper and lower line of the boxplot rectangle) are in our case defined as the 0.1 and 0.9 quantiles of the tunability scores. The 0.9 quantile indicates how much performance improvement can be expected on at least 10% of datasets. One outlier of `glmnet` (value 0.5) is not shown.

these two parameters. In `rpart` the `minbucket` and `minsplit` parameters seem to be the most important ones for tuning which is in line with the analysis of Mantovani et al. (2018). `k` in the `kknn` algorithm is very tunable w.r.t. package defaults, but not regarding optimal defaults. Note that the optimal default is 30 which is at the boundary of possible values, so possibly bigger values can provide further improvements. A classical suggestion in the literature (Lall and Sharma, 1996) is to use $\sqrt{n}$ as default value. This is in line with our results, as the number of observations is bigger than 900 in most of our datasets.

In `svm` the biggest gain in performance can be achieved by tuning the `kernel`, `gamma` or `degree`, while the `cost` parameter does not seem to be very tunable. To the best of our knowledge, this has not been noted in the literature yet. In `ranger` `mtry` is the most tunable parameter which is already common knowledge and is implemented in software packages such as `caret` (Kuhn, 2008). For `xgboost` there are two parameters that are quite tunable: `eta` and `booster`. The tunability of `booster` is highly influenced by an outlier as can be seen in Figure 3. The 5-fold cross-validated results can be seen in Table 10 of the appendix: they are quite similar to the non cross-validated results and for all parameters slightly higher.

Parameter	Def.P	Def.O	Tun.P	Tun.O	$q_{0.05}$	$q_{0.95}$
glmnet			0.069	0.024		
alpha	1	0.403	0.038	0.006	0.009	0.981
lambda	0	0.004	0.034	0.021	0.001	0.147
rpart			0.038	0.012		
cp	0.01	0	0.025	0.002	0	0.008
maxdepth	30	21	0.004	0.002	12.1	27
minbucket	7	12	0.005	0.006	3.85	41.6
minsplit	20	24	0.004	0.004	5	49.15
kknn			0.031	0.006		
k	7	30	0.031	0.006	9.95	30
svm			0.056	0.042		
kernel	radial	radial	0.030	0.024		
cost	1	682.478	0.016	0.006	0.002	920.582
gamma	$1/p$	0.005	0.030	0.022	0.003	18.195
degree	3	3	0.008	0.014	2	4
ranger			0.010	0.006		
num.trees	500	983	0.001	0.001	206.35	1740.15
replace	TRUE	FALSE	0.002	0.001		
sample.fraction	1	0.703	0.004	0.002	0.323	0.974
mtry	$\sqrt{p}$	$p \cdot 0.257$	0.006	0.003	0.035	0.692
respect.unordered.factors	TRUE	FALSE	0.000	0.000		
min.node.size	1	1	0.001	0.001	0.007	0.513
xgboost			0.043	0.014		
nrounds	500	4168	0.004	0.002	920.7	4550.95
eta	0.3	0.018	0.006	0.005	0.002	0.355
subsample	1	0.839	0.004	0.002	0.545	0.958
booster	gbtree	gbtree	0.015	0.008		
max_depth	6	13	0.001	0.001	5.6	14
min_child_weight	1	2.06	0.008	0.002	1.295	6.984
colsample_bytree	1	0.752	0.006	0.001	0.419	0.864
colsample_bylevel	1	0.585	0.008	0.001	0.335	0.886
lambda	1	0.982	0.003	0.002	0.008	29.755
alpha	1	1.113	0.003	0.002	0.002	6.105

Table 3: Defaults (package defaults (Def.P) and optimal defaults (Def.O)), tunability of the hyperparameters with the package defaults (Tun.P) and our optimal defaults (Tun.O) as reference and tuning space quantiles ($q_{0.05}$ and $q_{0.95}$) for different parameters of the algorithms.

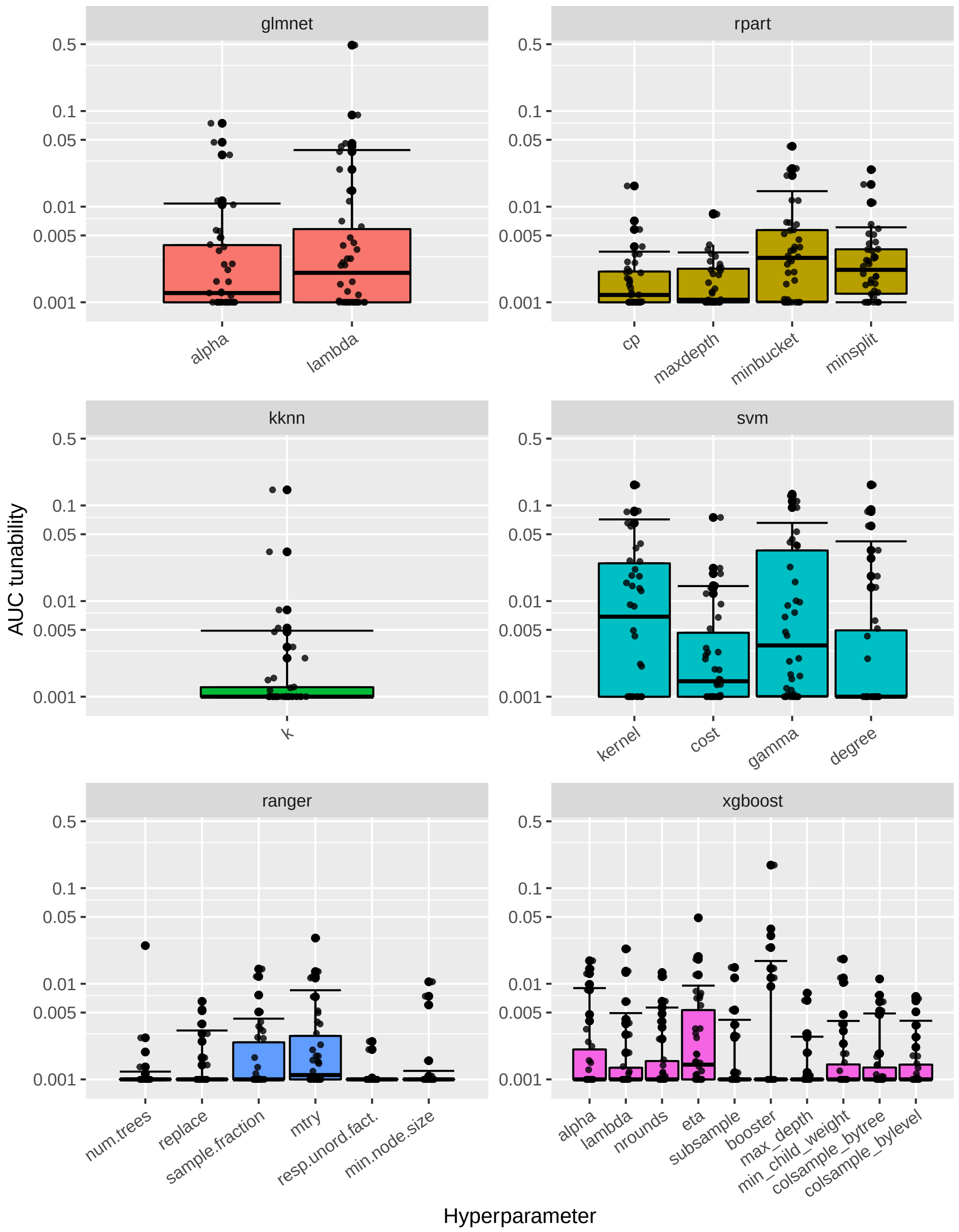

Figure 3: Boxplots of the tunabilities of the hyperparameters of the different algorithms with respect to optimal defaults. The y-axis is on a logarithmic scale. All values below 10^{-3} were set to 10^{-3} to be able to display them. Same definition of whiskers as in Figure 2.

	cp	maxdepth	minbucket	minsplit
cp	0.002	0.003	0.006	0.004
maxdepth		0.002	0.007	0.005
minbucket			0.006	0.011
minsplit				0.004

Table 4: Tunability d_{i_1,i_2} of hyperparameters of `rpart`, diagonal shows tunability of the single hyperparameters.

	maxdepth	minbucket	minsplit
cp	0.0007	0.0005	0.0004
maxdepth		0.0014	0.0019
minbucket			0.0055

Table 5: Joint gain g_{i_1,i_2} of tuning two hyperparameters instead of the most important in `rpart`.

5.4. Tunability of Hyperparameter Combinations and Joint Gains

As an example, Table 4 displays the average tunability d_{i_1,i_2} of all 2-way hyperparameter combinations for `rpart`. Obviously, the increased flexibility in tuning a 2-way combination enables larger improvements when compared with the tunability of one of the respective individual parameters. In Table 5 the joint gain of tuning two hyperparameters g_{i_1,i_2} instead of only the best as defined in Section 3.5 can be seen. The parameters `minsplit` and `minbucket` have the biggest joint effect, which is not very surprising, as they are closely related: `minsplit` is the minimum number of observations that must exist in a node in order for a split to be attempted and `minbucket` the minimum number of observations in any terminal leaf node. If a higher value of `minsplit` than the default performs better on a dataset it is possibly not enough to set it higher without also increasing `minbucket`, so the strong relationship is quite clear. Again, further figures for other algorithms are available through the shiny app. Another remarkable example is the combination of `sample.fraction` and `min.node.size` in `ranger`: the joint gain is very low and tuning `sample.fraction` only seems to be enough, which is concordant to the results of Scornet (2018). Moreover, in `xgboost` the joint gain of `nrounds` and `eta` is relatively low, which is not surprising, as these parameters are highly connected with each other (when setting `nrounds` higher, `eta` should be set lower and vice versa).

5.5. Hyperparameter Space for Tuning

The hyperparameter space for tuning, as defined in Equation (10) in Section 3.6 and based on the 0.05 and 0.95 quantiles, is displayed in Table 3. All optimal defaults are contained in this hyperparameter space while some of the package defaults are not.

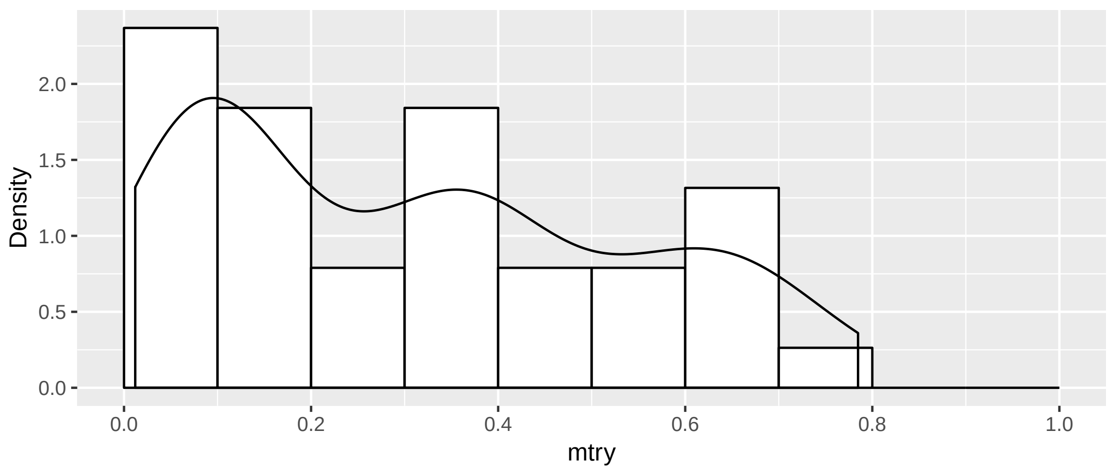

Figure 4: Density and histogram of best parameter values for `mtry` of random forest over all considered datasets.

As an example, Figure 4 displays the full histogram of the best values of `mtry` of the random forest over all datasets. Note that for quite a few datasets much higher values than the package defaults seem advantageous.

6. Conclusion and Discussion

Our paper provides concise and intuitive definitions for optimal defaults of ML algorithms and the impact of tuning them either jointly, tuning individual parameters or combinations, all based on the general concept of surrogate empirical performance models. Tunability values as defined in our framework are easily and directly interpretable as *how much performance can be gained by tuning this hyperparameter?*. This allows direct comparability of the tunability values across different algorithms.

In an extensive OpenML benchmark, we computed optimal defaults for elastic net, decision tree, k-nearest neighbors, SVM, random forest and xgboost and quantified their tunability and the tunability of their individual parameters. This—to the best of our knowledge—has never been provided before in such a principled manner. Our results are often in line with common knowledge from literature and our method itself now allows an analogous analysis for other or more complex methods.

Our framework is based on the concept of default hyperparameter values, which can be seen both as an advantage (default values are a valuable output of the approach) and as an inconvenience (the determination of the default values is an additional analysis step and needed as a reference point for most of our measures).

We now compare our method with van Rijn and Hutter (2017). In contrast to us, they apply the functional ANOVA framework from Hutter et al. (2014) on a surrogate random forest to assess the importance of hyperparameters regarding empirical performance of a support vector machine, random forest and adaboost, which results in numerical importance scores for individual hyperparameters. Their numerical scores are - in our opinion - less directly interpretable, but they do not rely on defaults as a reference point, which one might see as an advantage. They also propose a method for calculating hyperparameter priors, combine it with the tuning procedure hyperband, and assess the performance of this new tuning procedure. In contrast, we define and calculate ranges for all hyperparameters. Setting ranges for the tuning space can be seen as a special case of a prior distribution - the uniform distribution on the specified hyperparameter space. Regarding the experimental setup, we compute more hyperparameter runs (around 2.5 million vs. 250000), but consider only the 38 binary classification datasets of OpenML100 while van Rijn and Hutter (2017) use all the 100 datasets which also contain multiclass datasets. We evaluate the performance of different surrogate models by 10 times repeated 10-fold cross-validation to choose an appropriate model and to assure that it performs reasonably well.

Our study has some limitations that could be addressed in the future: a) We only considered binary classification, where we tried to include a wider variety of datasets from different domains. In principle this is not a restriction as our methods can easily be applied to multiclass classification, regression, survival analysis or even algorithms not from machine learning whose empirical performance is reliably measurable on a problem instance. b) Uniform random sampling of hyperparameters might not scale enough for very high dimensional spaces, and a smarter sequential technique might be in order here, see Bossek et al. (2015) for an potential approach of sampling across problem instances to learn optimal mappings from problem characteristics to algorithm configurations. c) We currently are learning static defaults, which cannot depend on dataset characteristics (like number of features, or further statistical measures). Doing so might improve performance results of optimal defaults considerably, but would require a more complicated approach. A recent paper regarding this topic was published by van Rijn et al. (2018). d) Our approach still needs initial ranges to be set, in order to run our sampling procedure. Only based on these wider ranges we can then compute more precise, closer ranges.

Acknowledgments

We would like to thank Joaquin Vanschoren for support regarding the OpenML platform and Andreas Müller, Jan van Rijn, Janek Thomas and Florian Pfisterer for reviewing and useful comments. Thanks to Jenny Lee for language editing. This work has been partially funded by grant BO3139/2-3 to ALB from the German Research Foundation and by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IS18036A. The authors of this work take full responsibilities for its content.

Appendix A. Additional Graphs and Tables

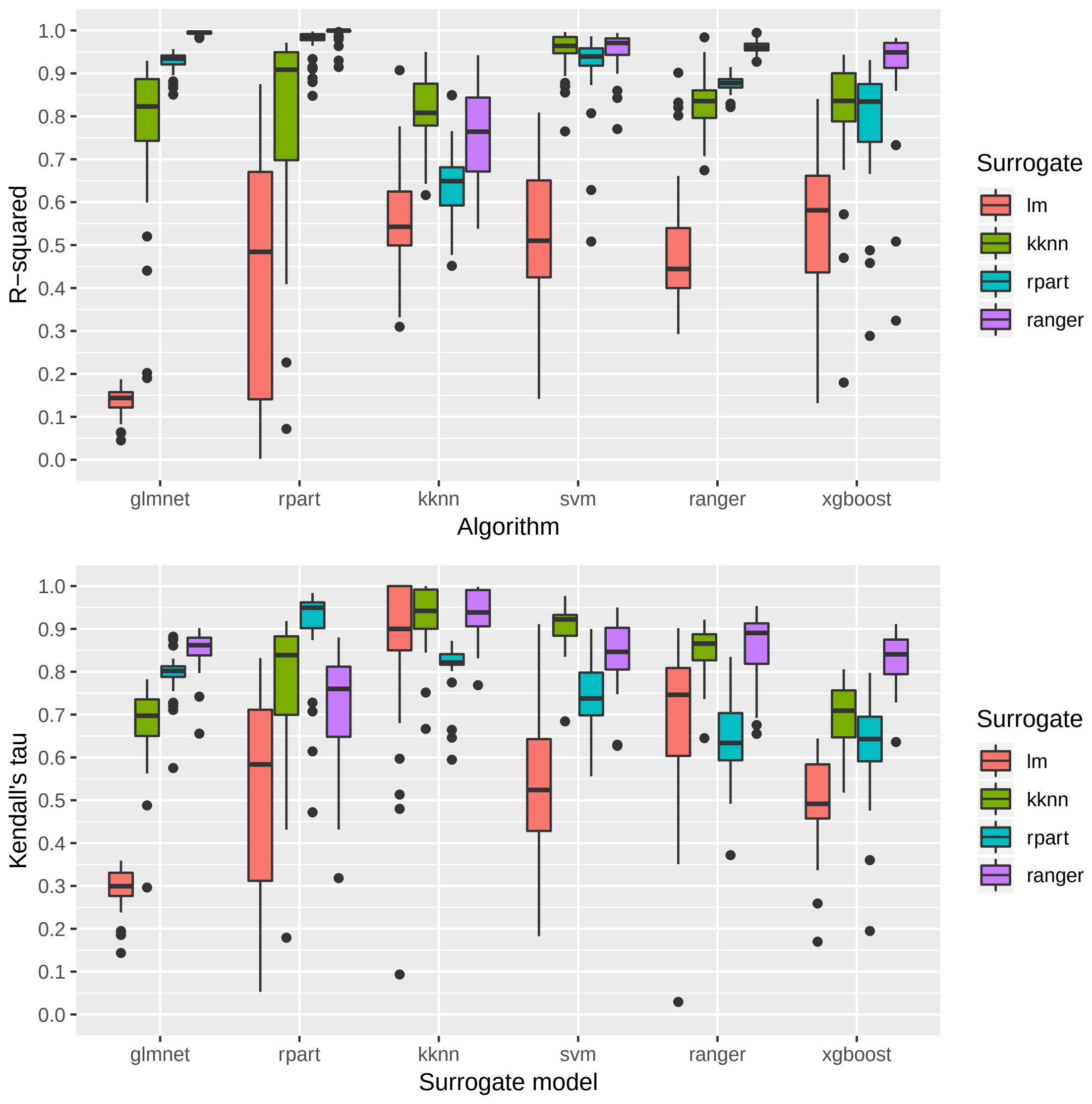

Figure 5: Same as Figure 1 but with exchanged x-axis and legend. Average performances over the datasets of different surrogate models (target: AUC) for different algorithms (that were presented in 4.2).

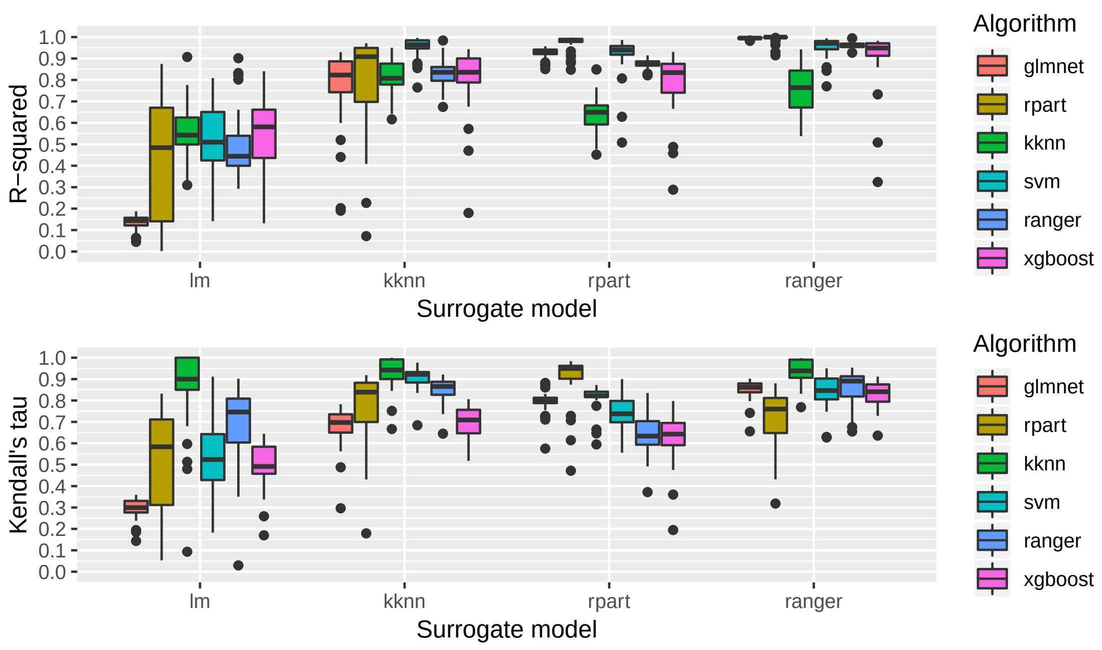

Figure 6: Surrogate model comparison as in Figure 1 but with accuracy as target measure.

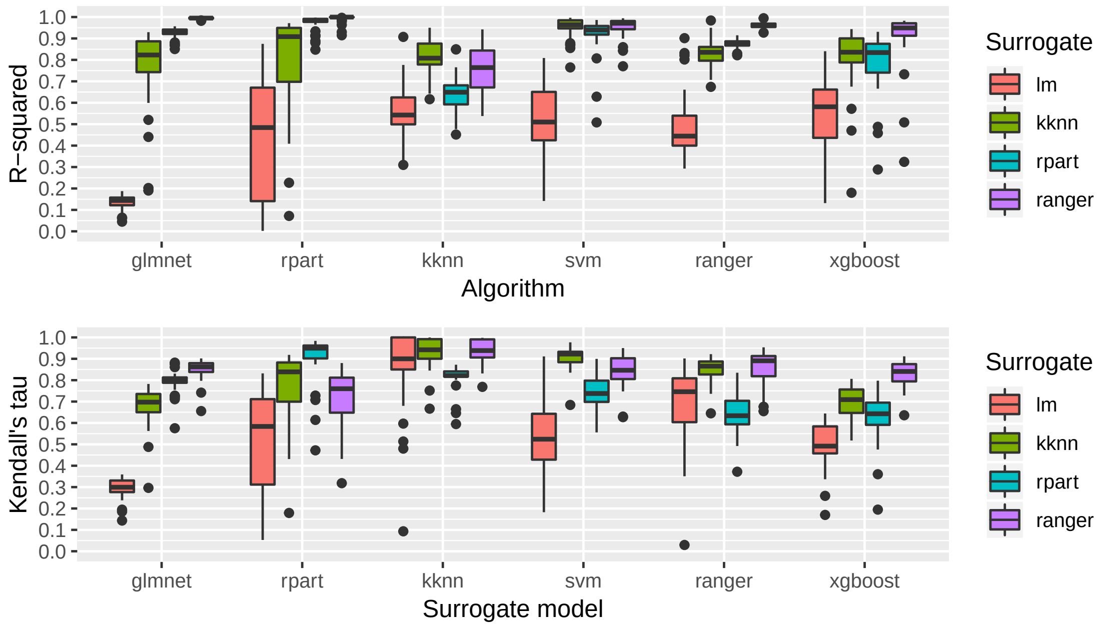

Figure 7: Surrogate model comparison as in Figure 6 (target: accuracy) but with exchanged x-axis and legend.

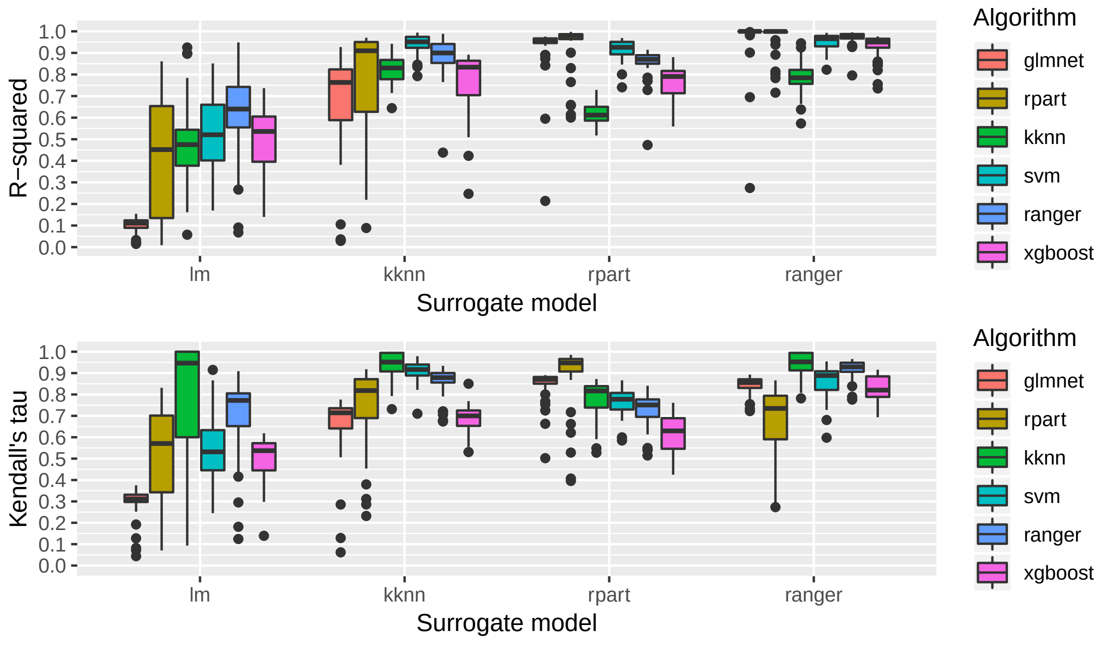

Figure 8: Surrogate model comparison as in Figure 1 but with Brier score as target measure.

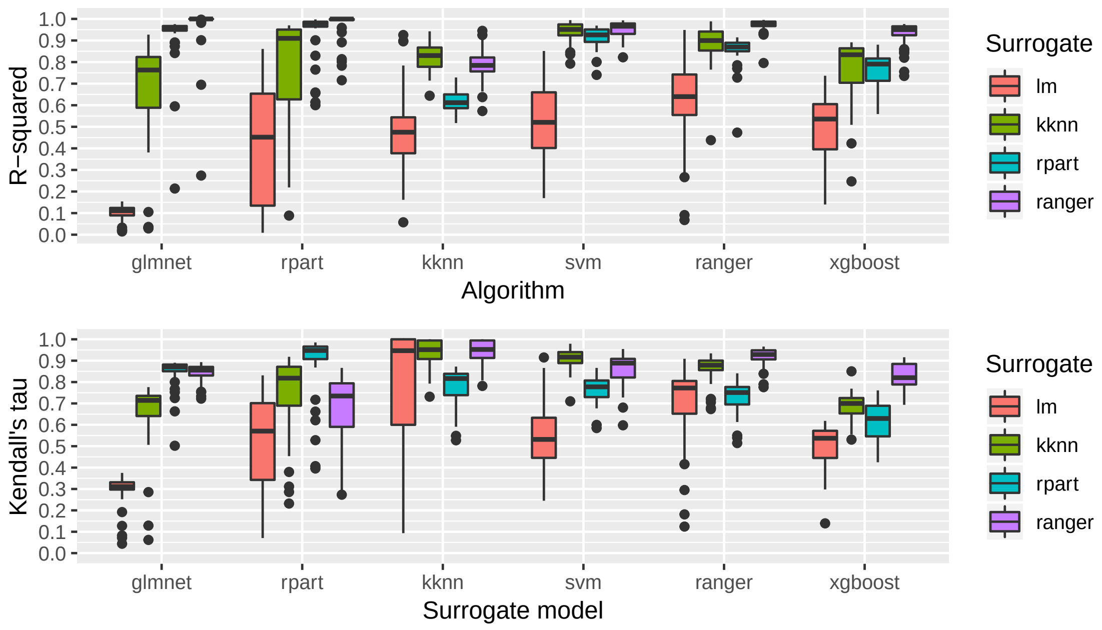

Figure 9: Surrogate model comparison as in Figure 8 but with exchanged x-axis and legend.

Algorithm	Tun.P	Tun.O	Tun.O-CV	Improv	Impr-CV
glmnet	0.042 ± 0.020	0.019 ± 0.010	0.042 ± 0.018	0.023 ± 0.021	0.001 ± 0.013
rpart	0.020 ± 0.004	0.012 ± 0.002	0.014 ± 0.004	0.008 ± 0.003	0.005 ± 0.002
kknn	0.021 ± 0.006	0.008 ± 0.002	0.010 ± 0.004	0.013 ± 0.005	0.010 ± 0.006
svm	0.041 ± 0.009	0.030 ± 0.008	0.041 ± 0.012	0.011 ± 0.004	-0.001 ± 0.011
ranger	0.016 ± 0.004	0.007 ± 0.001	0.009 ± 0.002	0.009 ± 0.004	0.006 ± 0.004
xgboost	0.034 ± 0.005	0.011 ± 0.004	0.012 ± 0.004	0.023 ± 0.004	0.022 ± 0.004

Table 6: Mean tunability as in Table 2, but calculated for the accuracy. Overall tunability (regarding accuracy) with the package defaults (Tun.P) and the optimal defaults (Tun.O) as reference points, cross-validated tunability (Tun.O-CV), average improvement (Improv) and cross-validated average improvement (Impr-CV) obtained by using optimal defaults compared to package defaults. The (cross-validated) improvement can be calculated by the (rounded) difference between Tun.P and Tun.O (Tun.O-CV). Standard error of the mean (SEM) is given behind the "±"-sign.

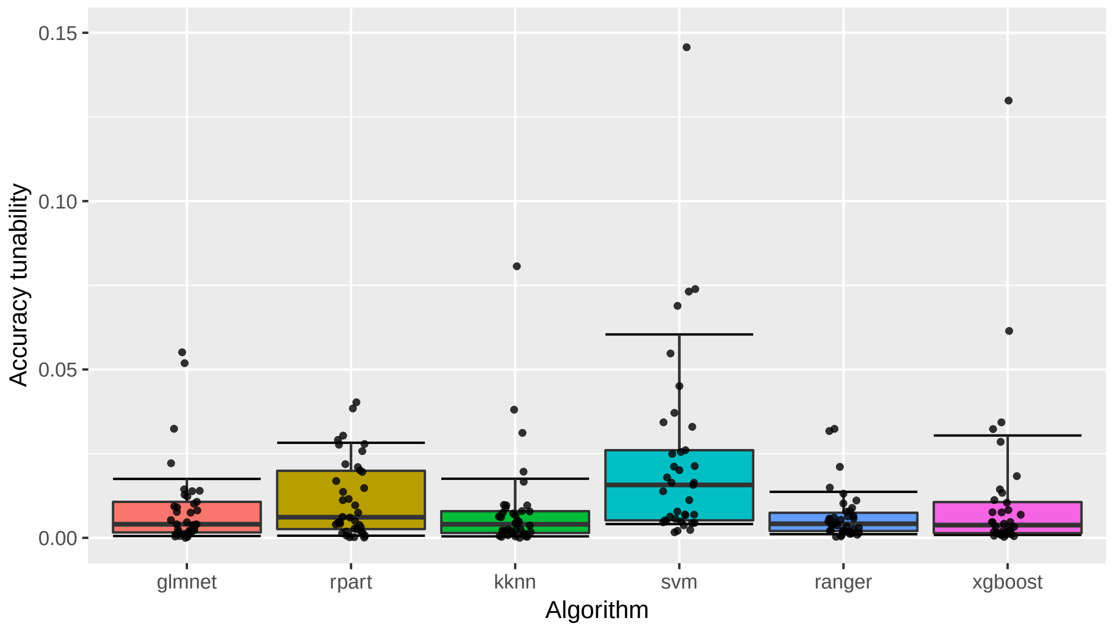

Figure 10: Boxplots of the tunabilities (accuracy) of the different algorithms with respect to optimal defaults.

Parameter	Def.P	Def.O	Tun.P	Tun.O	$q_{0.05}$	$q_{0.95}$
glmnet			0.042	0.019		
alpha	1	0.252	0.022	0.010	0.015	0.979
lambda	0	0.005	0.029	0.017	0.001	0.223
rpart			0.020	0.012		
cp	0.01	0.002	0.013	0.008	0	0.528
maxdepth	30	19	0.004	0.004	10	28
minbucket	7	5	0.005	0.006	1.85	43.15
minsplit	20	13	0.002	0.003	6.7	47.6
kknn			0.021	0.008		
k	7	14	0.021	0.008	2	30
svm			0.041	0.030		
kernel	radial	radial	0.019	0.018		
cost	1	936.982	0.019	0.003	0.025	943.704
gamma	$1/p$	0.002	0.024	0.020	0.007	276.02
degree	3	3	0.005	0.014	2	4
ranger			0.016	0.007		
num.trees	500	162	0.001	0.001	203.5	1908.25
replace	TRUE	FALSE	0.004	0.001		
sample.fraction	1	0.76	0.003	0.003	0.257	0.971
mtry	$\sqrt{p}$	$p \cdot 0.432$	0.010	0.003	0.081	0.867
respect.unordered.factors	TRUE	TRUE	0.001	0.000		
min.node.size	1	1	0.001	0.002	0.009	0.453
xgboost			0.034	0.011		
nrounds	500	3342	0.004	0.002	1360	4847.15
eta	0.3	0.031	0.005	0.005	0.002	0.445
subsample	1	0.89	0.003	0.002	0.555	0.964
booster	gbtree	gbtree	0.008	0.005		
max_depth	6	14	0.001	0.001	3	13
min_child_weight	1	1.264	0.009	0.002	1.061	7.502
colsample_bytree	1	0.712	0.005	0.001	0.334	0.887
colsample_bylevel	1	0.827	0.006	0.001	0.348	0.857
lambda	1	2.224	0.002	0.002	0.004	5.837
alpha	1	0.021	0.003	0.002	0.003	2.904

Table 7: Tunability measures for single hyperparameters and tuning spaces as in Table 3, but calculated for the accuracy. Defaults (package defaults (Def.P) and own optimal defaults (Def.O)), tunability of the hyperparameters with the package defaults (Tun.P) and our optimal defaults (Tun.O) as reference and tuning space quantiles ($q_{0.05}$ and $q_{0.95}$) for different parameters of the algorithms.

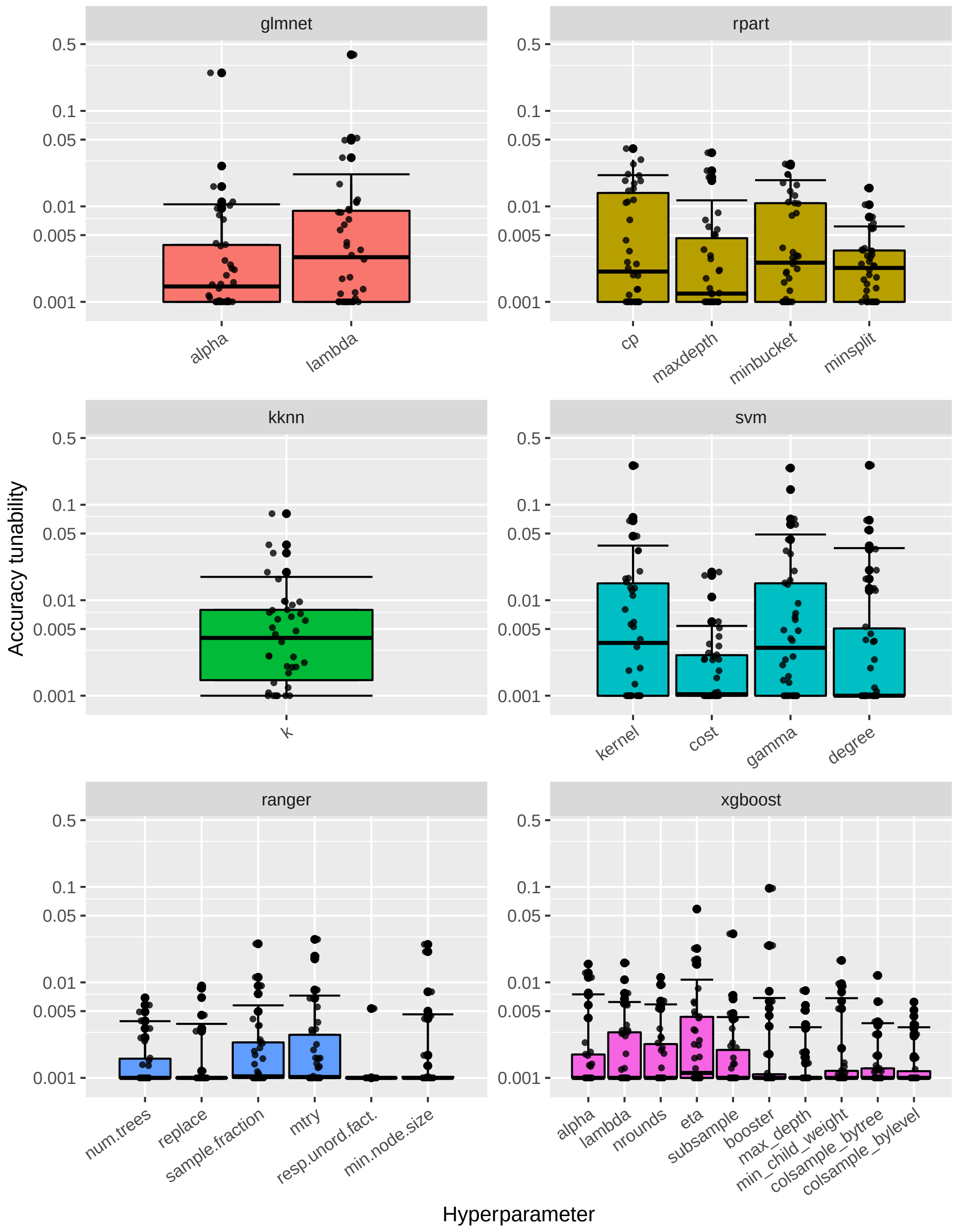

Figure 11: Boxplots of the tunabilities (accuracy) of the hyperparameters of the different algorithms with respect to optimal defaults. The y-axis is on a logarithmic scale. All values below 10^{-3} were set to 10^{-3} to be able to display them. Same definition of whiskers as in Figure 2.

Algorithm	Tun.P	Tun.O	Tun.O-CV	Improv	Impr-CV
glmnet	0.022 ± 0.007	0.010 ± 0.004	0.020 ± 0.014	0.011 ± 0.006	0.001 ± 0.012
rpart	0.015 ± 0.002	0.009 ± 0.002	0.011 ± 0.003	0.006 ± 0.002	0.004 ± 0.002
kknn	0.012 ± 0.003	0.003 ± 0.001	0.003 ± 0.001	0.009 ± 0.003	0.009 ± 0.003
svm	0.026 ± 0.005	0.018 ± 0.004	0.023 ± 0.006	0.008 ± 0.003	0.003 ± 0.005
ranger	0.015 ± 0.004	0.005 ± 0.001	0.006 ± 0.001	0.010 ± 0.004	0.009 ± 0.004
xgboost	0.027 ± 0.003	0.009 ± 0.002	0.011 ± 0.003	0.018 ± 0.002	0.016 ± 0.002

Table 8: Mean tunability as in Table 2, but calculated for the Brier score. Overall tunability (regarding Brier score) with the package defaults (Tun.P) and the optimal defaults (Tun.O) as reference points, cross-validated tunability (Tun.O-CV), average improvement (Improv) and cross-validated average improvement (Impr-CV) obtained by using optimal defaults compared to package defaults. The (cross-validated) improvement can be calculated by the (rounded) difference between Tun.P and Tun.O (Tun.O-CV). Standard error of the mean (SEM) is given behind the "±"-sign.

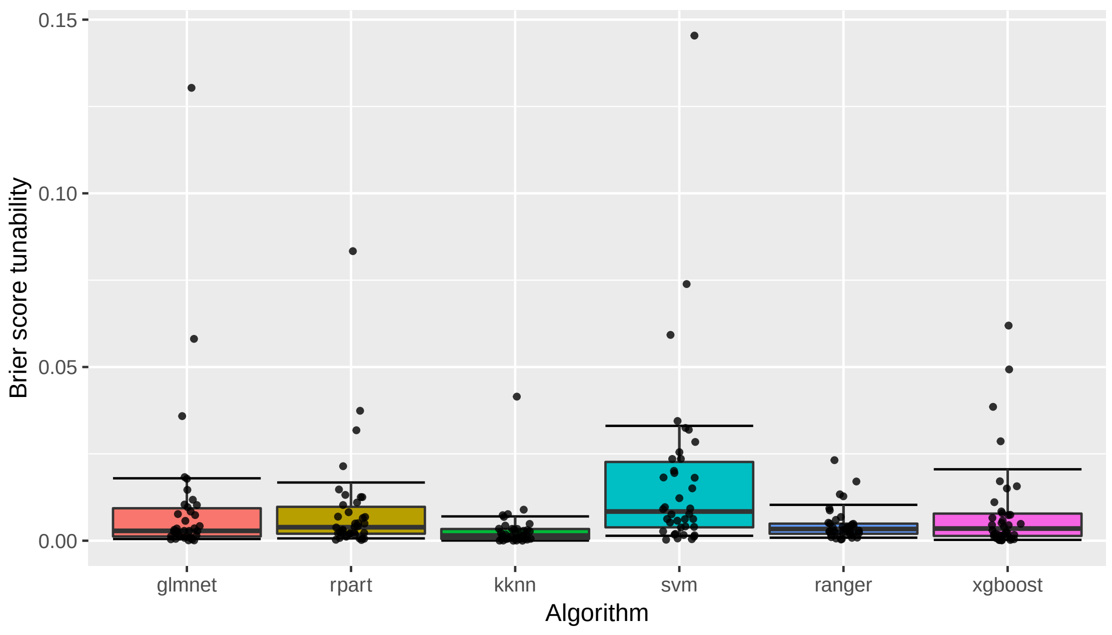

Figure 12: Boxplots of the tunabilities (Brier score) of the different algorithms with respect to optimal defaults.

Parameter	Def.P	Def.O	Tun.P	Tun.O	$q_{0.05}$	$q_{0.95}$
glmnet			0.022	0.010		
alpha	1	0.997	0.009	0.005	0.003	0.974
lambda	0	0.004	0.014	0.007	0.001	0.051
rpart			0.015	0.009		
cp	0.01	0.001	0.009	0.003	0	0.035
maxdepth	30	13	0.002	0.002	9	27.15
minbucket	7	12	0.004	0.006	1	44.1
minsplit	20	18	0.002	0.002	7	49.15
kknn			0.012	0.003		
k	7	19	0.012	0.003	4.85	30
svm			0.026	0.018		
kernel	radial	radial	0.013	0.011		
cost	1	950.787	0.012	0.002	0.002	963.81
gamma	$1/p$	0.005	0.015	0.012	0.001	4.759
degree	3	3	0.003	0.009	2	4
ranger			0.015	0.005		
num.trees	500	198	0.001	0.001	187.85	1568.25
replace	TRUE	FALSE	0.002	0.001		
sample.fraction	1	0.667	0.002	0.003	0.317	0.964
mtry	$\sqrt{p}$	$p \cdot 0.666$	0.010	0.002	0.072	0.954
respect.unordered.factors	TRUE	TRUE	0.000	0.000		
min.node.size	1	1	0.001	0.001	0.008	0.394
xgboost			0.027	0.009		
nrounds	500	2563	0.004	0.002	2018.55	4780.05
eta	0.3	0.052	0.004	0.005	0.003	0.436
subsample	1	0.873	0.002	0.002	0.447	0.951
booster	gbtree	gbtree	0.009	0.004		
max_depth	6	11	0.001	0.001	2.6	13
min_child_weight	1	1.75	0.007	0.002	1.277	5.115
colsample_bytree	1	0.713	0.004	0.002	0.354	0.922
colsample_bylevel	1	0.638	0.004	0.001	0.363	0.916
lambda	1	0.101	0.002	0.003	0.006	28.032
alpha	1	0.894	0.003	0.004	0.003	2.68

Table 9: Tunability measures for single hyperparameters and tuning spaces as in Table 3, but calculated for the Brier score. Defaults (package defaults (Def.P) and own optimal defaults (Def.O)), tunability of the hyperparameters with the package defaults (Tun.P) and our optimal defaults (Tun.O) as reference and tuning space quantiles ($q_{0.05}$ and $q_{0.95}$) for different parameters of the algorithms.

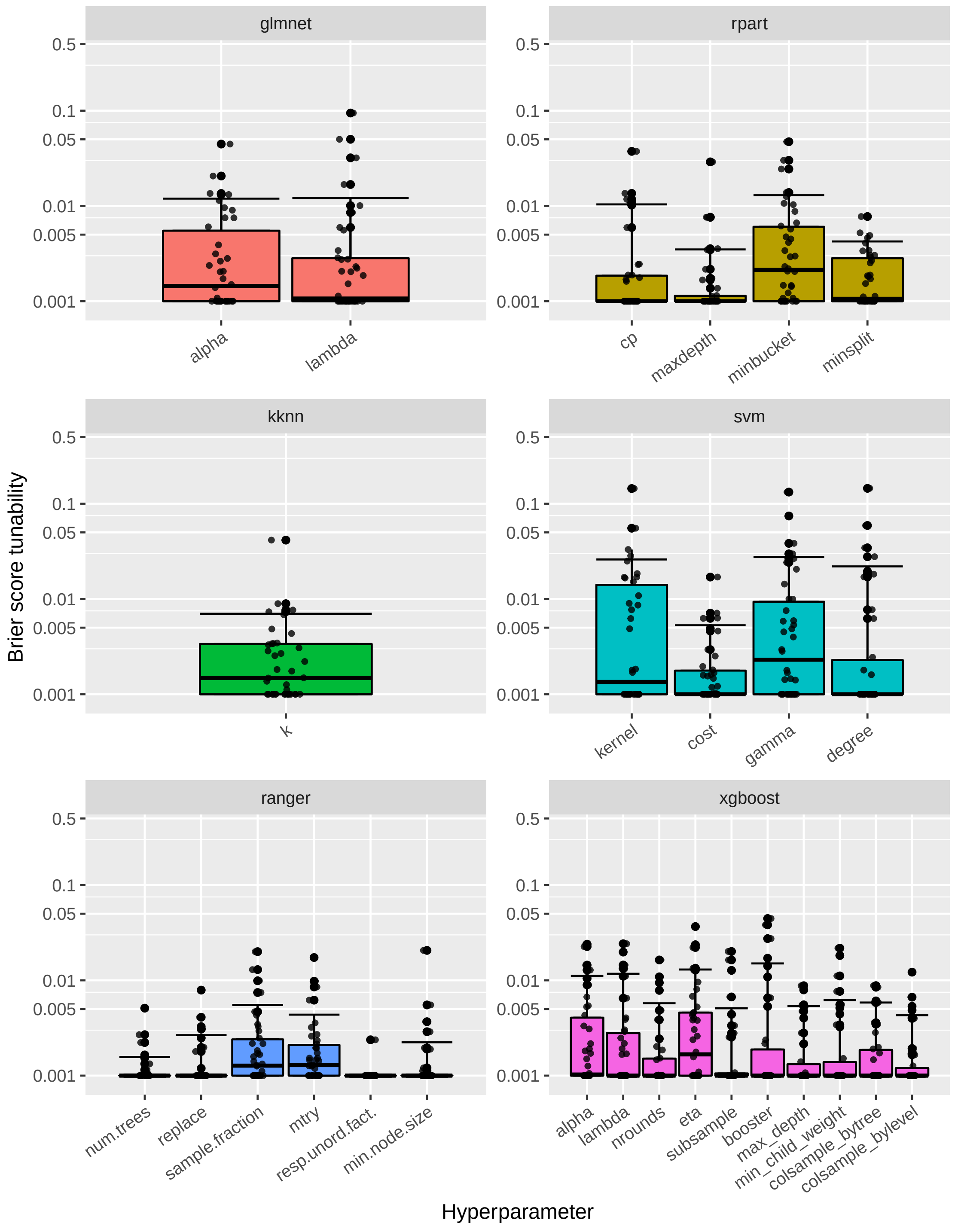

Figure 13: Boxplots of the tunabilities (Brier score) of the hyperparameters of the different algorithms with respect to optimal defaults. The y-axis is on a logarithmic scale. All values below 10^{-3} were set to 10^{-3} to be able to display them. Same definition of whiskers as in Figure 2.

Measure	**AUC**		**Accuracy**		**Brier score**	
Parameter	Tun.O	Tun.O-CV	Tun.O	Tun.O-CV	Tun.O	Tun.O-CV
glmnet	0.024	0.037	0.019	0.042	0.010	0.020
alpha	0.006	0.006	0.010	0.026	0.005	0.015
lambda	0.021	0.034	0.017	0.039	0.007	0.018
rpart	0.012	0.016	0.012	0.014	0.009	0.011
cp	0.002	0.002	0.008	0.008	0.003	0.005
maxdepth	0.002	0.002	0.004	0.004	0.002	0.003
minbucket	0.006	0.009	0.006	0.007	0.006	0.006
minsplit	0.004	0.004	0.003	0.003	0.002	0.003
kknn	0.006	0.006	0.008	0.010	0.003	0.003
k	0.006	0.006	0.008	0.010	0.003	0.003
svm	0.042	0.048	0.030	0.041	0.018	0.023
kernel	0.024	0.030	0.018	0.031	0.011	0.016
cost	0.006	0.006	0.003	0.003	0.002	0.002
gamma	0.022	0.028	0.020	0.031	0.012	0.016
degree	0.014	0.020	0.014	0.027	0.009	0.014
ranger	0.006	0.007	0.007	0.009	0.005	0.006
num.trees	0.001	0.002	0.001	0.003	0.001	0.001
replace	0.001	0.002	0.001	0.002	0.001	0.001
sample.fraction	0.002	0.002	0.003	0.003	0.003	0.003
mtry	0.003	0.004	0.003	0.005	0.002	0.003
respect.unordered.factors	0.000	0.000	0.000	0.001	0.000	0.000
min.node.size	0.001	0.001	0.002	0.002	0.001	0.001
xgboost	0.014	0.017	0.011	0.012	0.009	0.011
nrounds	0.002	0.002	0.002	0.003	0.002	0.002
eta	0.005	0.006	0.005	0.006	0.005	0.006
subsample	0.002	0.002	0.002	0.002	0.002	0.002
booster	0.008	0.008	0.005	0.005	0.004	0.004
max_depth	0.001	0.001	0.001	0.001	0.001	0.001
min_child_weight	0.002	0.003	0.002	0.002	0.002	0.003
colsample_bytree	0.001	0.002	0.001	0.001	0.002	0.002
colsample_bylevel	0.001	0.001	0.001	0.001	0.001	0.002
lambda	0.002	0.003	0.002	0.003	0.003	0.004
alpha	0.002	0.004	0.002	0.003	0.004	0.004

Table 10: Tunability with optimal defaults as reference without (Tun.O) and with (Tun.O-CV) cross-validation for AUC, accuracy and Brier score

References

Charles Audet, John E. Dennis, Douglas Moore, Andrew Booker, and Paul Frank. A surrogate-model-based method for constrained optimization. In *8th Symposium on Multidisciplinary Analysis and Optimization*, page 4891, 2000.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

André Biedenkapp, Marius Thomas Lindauer, Katharina Eggensperger, Frank Hutter, Chris Fawcett, and Holger H Hoos. Efficient parameter importance analysis via ablation with surrogates. In *AAAI*, pages 773–779, 2017.

Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-Race and iterated F-Race: An overview. In *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer, 2010.

Bernd Bischl, Olaf Mersmann, Heike Trautmann, and Claus Weihs. Resampling methods for meta-model validation with recommendations for evolutionary computation. *Evolutionary Computation*, 20(2):249–275, 2012.

Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016.

Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. OpenML benchmarking suites and the OpenML100. *ArXiv preprint arXiv:1708.03731*, 2017a. URL `https://arxiv.org/abs/1708.03731`.

Bernd Bischl, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. mlrMBO: A modular framework for model-based optimization of expensive black-box functions. *ArXiv preprint arXiv:1703.03373*, 2017b. URL `https://arxiv.org/abs/1703.03373`.

Jakob Bossek, Bernd Bischl, Tobias Wagner, and Günter Rudolph. Learning feature-parameter mappings for parameter tuning via the profile expected improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1319–1326. ACM, 2015.

Giuseppe Casalicchio, Jakob Bossek, Michel Lang, Dominik Kirchhoff, Pascal Kerschke, Benjamin Hofner, Heidi Seibold, Joaquin Vanschoren, and Bernd Bischl. OpenML: An R package to connect to the machine learning platform OpenML. *Computational Statistics*, 32(3):1–15, 2017.

Katharina Eggensperger, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Surrogate benchmarks for hyperparameter optimization. In *Proceedings of the 2014 International Conference on Meta-learning and Algorithm Selection-Volume 1201*, pages 24–31. CEUR-WS.org, 2014.

Katharina Eggensperger, Marius Lindauer, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Efficient benchmarking of algorithm configurators via model-based surrogates. *Machine Learning*, pages 1–27, 2018.

Agoston E Eiben and Selmar K Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

Chris Fawcett and Holger H Hoos. Analysing differences between algorithm configurations through ablation. *Journal of Heuristics*, 22(4):431–458, 2016.

Matthias Feurer, Benjamin Letham, and Eytan Bakshy. Scalable meta-learning for bayesian optimization. *arXiv preprint 1802.02219*, 2018. URL `https://arxiv.org/abs/1802.02219`.

Isabelle Guyon, Amir Saffari, Gideon Dror, and Gavin Cawley. Model selection: Beyond the bayesian/frequentist divide. *Journal of Machine Learning Research*, 11(Jan):61–87, 2010.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Identifying key algorithm parameters and instance features using forward selection. In *International Conference on Learning and Intelligent Optimization*, pages 364–381. Springer, 2013.

Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 754–762, 2014.

Max Kuhn. Building predictive models in R using the caret package. *Journal of statistical software*, 28(5):1–26, 2008.

Max Kuhn, Steve Weston, Chris Keefer, and Nathan Coulter. *Cubist: Rule- and instance-based regression modeling*, 2016. R package version 0.0.19.

Daniel Kühn, Philipp Probst, Janek Thomas, and Bernd Bischl. OpenML R bot benchmark data (final subset), 2018a. URL `https://figshare.com/articles/OpenML_R_Bot_Benchmark_Data_final_subset_/5882230/2`.

Daniel Kühn, Philipp Probst, Janek Thomas, and Bernd Bischl. Automatic Exploration of Machine Learning Experiments on OpenML. *ArXiv preprint arXiv:1806.10961*, 2018b. URL `https://arxiv.org/abs/1806.10961`.

Upmanu Lall and Ashish Sharma. A nearest neighbor bootstrap for resampling hydrologic time series. *Water Resources Research*, 32(3):679–693, 1996.

Michel Lang, Bernd Bischl, and Dirk Surmann. batchtools: Tools for R to work on batch systems. *The Journal of Open Source Software*, 2(10), 2017.

Gang Luo. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 5(1):1–16, 2016.

Rafael G. Mantovani, Tomáš Horváth, Ricardo Cerri, Andre C.P.L.F. Carvalho, and Joaquin Vanschoren. Hyper-parameter tuning of a decision tree induction algorithm. In *Brazilian Conference on Intelligent Systems (BRACIS 2016)*, 2016.

Rafael Gomes Mantovani, Tomáš Horváth, Ricardo Cerri, Sylvio Barbon Junior, Joaquin Vanschoren, André Carlos Ponce de de Carvalho, and Leon Ferreira. An empirical study on hyperparameter tuning of decision trees. *ArXiv preprint arXiv:1812.02207*, 2018. URL `https://arxiv.org/abs/1812.02207`.

Erwan Scornet. Tuning parameters in random forests. *ESAIM: Procs*, 60:144–162, 2018.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Jan N. van Rijn and Frank Hutter. Hyperparameter importance across datasets. *ArXiv preprint arXiv:1710.04725*, 2017. URL `https://arxiv.org/abs/1710.04725`.

Jan N van Rijn, Florian Pfisterer, Janek Thomas, Andreas Muller, Bernd Bischl, and J Vanschoren. Meta learning for defaults: Symbolic defaults. In *Neural Information Processing Workshop on Meta-Learning*, 2018.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

Levi Waldron, Melania Pintilie, Ming-Sound Tsao, Frances A Shepherd, Curtis Huttenhower, and Igor Jurisica. Optimized application of penalized regression methods to diverse genomic data. *Bioinformatics*, 27(24):3399–3406, 2011.

Automatic Exploration of Machine Learning Experiments on OpenML

Daniel Kühn*[a], Philipp Probst*[a], Janek Thomas[a], Bernd Bischl[a]

[a] *Ludwig-Maximilians-Universität München, Germany*

Abstract

Understanding the influence of hyperparameters on the performance of a machine learning algorithm is an important scientific topic in itself and can help to improve automatic hyperparameter tuning procedures. Unfortunately, experimental meta data for this purpose is still rare. This paper presents a large, free and open dataset addressing this problem, containing results on 38 OpenML data sets, six different machine learning algorithms and many different hyperparameter configurations. Results where generated by an automated random sampling strategy, termed the *OpenML Random Bot*. Each algorithm was cross-validated up to 20.000 times per dataset with different hyperparameters settings, resulting in a meta dataset of around 2.5 million experiments overall.

1. Introduction

When applying machine learning algorithms on real world datasets, users have to choose from a large selection of different algorithms with many of them offering a set of hyperparameters to control algorithmic performance. Although sometimes default values exist, there is no agreed upon principle for their definition (but see our recent work in in (Probst et al., 2018) for a potential approach). Automatic tuning of such parameters is a possible solution (Claesen and Moor, 2015), but comes with a considerable computational burden.

Meta-learning tries to decrease this cost (Feurer et al., 2015), by reusing information of previous runs of the algorithm on similar datasets, which obviously requires access to such prior empirical results. With this paper we provide a freely accessible meta dataset that contains around 2.5 million runs of six different machine learning algorithms on 38 classification datasets.

Large, freely available datasets like Imagenet (Deng et al., 2009) are important for the progress of machine learning, we hope to support developments in the area of meta-learning and benchmarking, meta-learning and hyperparameter tuning with our work here.

While similar meta-datasets have been created in the past, we were not able to access them by the links provided in their respective papers: Smith et al. (2014) provides a repository with Weka-based machine learning experiments on 72 data sets, 9 machine learning algorithms, 10 hyperparameter settings for each algorithm, and several meta-features of each data set. Reif (2012) created a meta-dataset based on machine learning experiments on 83 datasets, 6 classification algorithms, and 49 meta features.

In this paper, we describe our experimental setup, specify how our meta-dataset is created by running random machine learning experiments through the OpenML platform (Vanschoren et al., 2013) and explain how to access our results.

Email addresses: `daniel.kuehn.87@gmail.com` (Daniel Kühn*), `philipp_probst@gmx.de` (Philipp Probst*), `janek.thomas@stat.uni-muenchen.de` (Janek Thomas), `bernd_bischl@gmx.net` (Bernd Bischl)

2. Considered ML data sets, algorithms and hyperparameters

To create the meta dataset, six supervised machine learning algorithms are run on 38 classification tasks. For each algorithm the available hyperparameters are explored in a predefined range (see Table 1). Some of these hyperparameters are transformed by the function found in column *trafo* of Table 1 to allow non-uniform sampling, a usual procedure in tuning.

algorithm	hyperparameter	type	lower	upper	trafo
glmnet	alpha	numeric	0	1	-
	lambda	numeric	-10	10	2^x
rpart	cp	numeric	0	1	-
	maxdepth	integer	1	30	-
	minbucket	integer	1	60	-
	minsplit	integer	1	60	-
kknn	k	integer	1	30	-
svm	kernel	discrete	-	-	-
	cost	numeric	-10	10	2^x
	gamma	numeric	-10	10	2^x
	degree	integer	2	5	-
ranger	num.trees	integer	1	2000	-
	replace	logical	-	-	-
	sample.fraction	numeric	0	1	-
	mtry	numeric	0	1	$x \cdot p$
	respect.unordered.factors	logical	-	-	-
	min.node.size	numeric	0	1	n^x
xgboost	nrounds	integer	1	5000	-
	eta	numeric	-10	0	2^x
	subsample	numeric	0	1	-
	booster	discrete	-	-	-
	max_depth	integer	1	15	-
	min_child_weight	numeric	0	7	2^x
	colsample_bytree	numeric	0	1	-
	colsample_bylevel	numeric	0	1	-
	lambda	numeric	-10	10	2^x
	alpha	numeric	-10	10	2^x

Table 1: Hyperparameters of the algorithms. p refers to the number of variables and n to the number of observations. The used algorithms are `glmnet` (Friedman et al., 2010), `rpart` (Therneau and Atkinson, 2018), `kknn` (Schliep and Hechenbichler, 2016), `svm` (Meyer et al., 2017), `ranger` (Wright and Ziegler, 2017) and `xgboost` (Chen and Guestrin, 2016).

These algorithms are run on a subset of the OpenML100 benchmark suite (Bischl et al., 2017), which consists of 100 classification datasets, carefully curated from the thousands of datasets available on OpenML (Vanschoren et al., 2013). We only include datasets without missing data and with a binary outcome resulting in 38 datasets. The datasets and their respective characteristics can be found in Table 2.

Data_id	Task_id	Name	n	p	majPerc	numFeat	catFeat
3	3	kr-vs-kp	3196	37	0.52	0	37
31	31	credit-g	1000	21	0.70	7	14
37	37	diabetes	768	9	0.65	8	1
44	43	spambase	4601	58	0.61	57	1
50	49	tic-tac-toe	958	10	0.65	0	10
151	219	electricity	45312	9	0.58	7	2
312	3485	scene	2407	300	0.82	294	6
333	3492	monks-problems-1	556	7	0.50	0	7
334	3493	monks-problems-2	601	7	0.66	0	7
335	3494	monks-problems-3	554	7	0.52	0	7
1036	3889	sylva_agnostic	14395	217	0.94	216	1
1038	3891	gina_agnostic	3468	971	0.51	970	1
1043	3896	ada_agnostic	4562	49	0.75	48	1
1046	3899	mozilla4	15545	6	0.67	5	1
1049	3902	pc4	1458	38	0.88	37	1
1050	3903	pc3	1563	38	0.90	37	1
1063	3913	kc2	522	22	0.80	21	1
1067	3917	kc1	2109	22	0.85	21	1
1068	3918	pc1	1109	22	0.93	21	1
1120	3954	MagicTelescope	19020	12	0.65	11	1
1461	14965	bank-marketing	45211	17	0.88	7	10
1462	10093	banknote-authentication	1372	5	0.56	4	1
1464	10101	blood-transfusion-service-center	748	5	0.76	4	1
1467	9980	climate-model-simulation-crashes	540	21	0.91	20	1
1471	9983	eeg-eye-state	14980	15	0.55	14	1
1479	9970	hill-valley	1212	101	0.50	100	1
1480	9971	ilpd	583	11	0.71	9	2
1485	9976	madelon	2600	501	0.50	500	1
1486	9977	nomao	34465	119	0.71	89	30
1487	9978	ozone-level-8hr	2534	73	0.94	72	1
1489	9952	phoneme	5404	6	0.71	5	1
1494	9957	qsar-biodeg	1055	42	0.66	41	1
1504	9967	steel-plates-fault	1941	34	0.65	33	1
1510	9946	wdbc	569	31	0.63	30	1
1570	9914	wilt	4839	6	0.95	5	1
4134	14966	Bioresponse	3751	1777	0.54	1776	1
4534	34537	PhishingWebsites	11055	31	0.56	0	31

Table 2: Included datasets and respective characteristics. n are the number of observations, p the number of features, *maj.class* the percentage of observations in the largest class, *numFeat* the number of numeric features and *catFeat* the number of categorical features.

3. Random Experimentation Bot

To conduct a large number of experiments a bot was implemented to automatically plan and execute runs, following the paradigm of random search. The bot iteratively executes these steps:

1. Randomly sample a task T (with an associated data set) from Table 2.
2. Randomly sample one ML algorithm A.
3. Randomly sample a hyperparameter setting θ of algorithm A, uniformly from the ranges specified in Table 1, then transform, if a transformation function is given.
4. Obtain task T (and dataset) from OpenML and store it locally.
5. Evaluate algorithm A with configuration θ on task T, with associated 10-fold cross-validation from OpenML.
6. Upload run results to OpenML, including hyperparameter configuration and time measurements.
7. OpenML now calculates various performance metrics for the uploaded cross-validated predictions.
8. The OpenML-ID of the bot (2702) and the tag `mlrRandomBot` is used for identification.

A clear advantage of random sampling is that all bot runs are completely independent of each other, making all experiments embarrassingly parallel. Furthermore, more experiments can easily and conveniently added later on, without introducing any kind of bias into the sampling method.

The bot is developed open source in R and can be found on GitHub[1]. The bot is based on the R packages `mlr` (Bischl et al., 2016) and `OpenML` (Casalicchio et al., 2017) and written in modular form such that it can be extended with new sampling strategies for hyperparameters, algorithms and datasets in the future. Parallelization was performed with R package `batchtools` (Lang et al., 2017).

After more than 6 million benchmark experiments the results of the bot are downloaded from OpenML. For each of the algorithms 500000 experiments are used to obtain the final dataset. The experiments are chosen by the following procedure: For each algorithm, a threshold B is set (see below) and, if the number of results for a dataset exceeds B, we draw randomly B of the results obtained for this algorithm and this dataset. The threshold value B is chosen for each algorithm separately to exactly obtain in total 500000 results for each algorithm.

For `kknn` we only execute 30 experiments per dataset because this number of experiments is high enough to cover the hyperparameter space (that only consists of the parameter k for $k \in \{1, ..., 30\}$) appropriately, resulting in 1140 experiments. All in all this results in around 2.5 million experiments.

The distribution of the runs on the datasets and algorithms is displayed in Table 3.

Data_id	Task_id	glmnet	rpart	kknn	svm	ranger	xgboost	Total
3	3	15547	14633	30	19644	15139	16867	81860
31	31	15547	14633	30	19644	15139	16867	81860
37	37	15546	14633	30	15985	15139	16866	78199
44	43	15547	14633	30	19644	15139	16867	81860
50	49	15547	14633	30	19644	15139	16866	81859
151	219	15547	14632	30	2384	12517	16866	61976
312	3485	6613	13455	30	18740	12985	15886	67709
333	3492	15546	14632	30	19644	15139	16867	81858
334	3493	15547	14633	30	19644	14492	16867	81213
335	3494	15547	14633	30	15123	15139	10002	70474
1036	3889	14937	14633	30	2338	7397	2581	41916
1038	3891	15547	5151	30	5716	4827	1370	32641
1043	3896	6466	14633	30	10121	3788	16867	51905
1046	3899	15547	14633	30	5422	8842	11812	56286
1049	3902	7423	14632	30	12064	15139	4453	53741
1050	3903	15547	14633	30	19644	11357	13758	74969
1063	3913	15547	14633	30	19644	7914	16866	74634
1067	3917	15546	14632	30	10229	7386	16866	64689
1068	3918	15546	14633	30	13893	8173	16866	69141
1120	3954	15531	7477	30	3908	9760	8143	44849
1461	14965	6970	14073	30	2678	14323	2215	40289
1462	10093	8955	14633	30	6320	15139	16867	61944
1464	10101	15547	14632	30	19644	15139	16867	81859
1467	9980	15547	14633	30	4441	15139	16866	66656
1471	9983	15547	14633	30	9725	13523	16866	70324
1479	9970	15546	14633	30	19644	15140	16867	81860
1480	9971	15024	14633	30	19644	15139	16254	80724
1485	9976	8247	10923	30	10334	15139	9237	53910
1486	9977	3866	11389	30	1490	15139	5813	37727
1487	9978	15547	6005	30	19644	15139	11194	67559
1489	9952	15547	14633	30	17298	15139	16867	79514
1494	9957	15547	14632	30	19644	15139	16867	81859
1504	9967	15547	14633	30	19644	15140	16867	81861
1510	9946	15547	14633	30	19644	15139	16867	81860
1570	9914	15546	14632	30	19644	15139	16867	81858
4134	14966	1493	3947	30	560	14516	2222	22768
4534	34537	2801	3231	30	2476	15139	947	24624
Total	257661	486995	485368	1110	485549	484860	486953	2430835

Table 3: Number of experiments for each combination of dataset and algorithm.

[1]`https://github.com/ja-thomas/OMLbots`

4. Access to the results

The results of the benchmark can be accessed in different ways:

- The easiest way to access them is to go to the figshare repository (Kühn et al., 2018) and to download the `.csv` files. For each algorithm there is a csv file that contains a row for each algorithm run with the columns `Data_id`, the hyperparameter settings, the performance measures (auc, accuracy and brier score), the runtime, the scimark reference runtime and some characteristics of the dataset such as the number of features or the number of observations.

- Alternatively the code for the extraction of the data from the nightly database snapshot of OpenML can be found here: `https://github.com/ja-thomas/OMLbots/blob/master/snapshot_database/database_extraction.R`. With this script all results that were created by the random bot (OpenML-ID 2702) are downloaded and the final dataset is created. (Warning: As the OpenML database is updated daily, changes can occur.)

5. Discussion and potential usage of the results

The presented data can be used to study the effect and influence of hyperparameter setting on performance in various ways. Possible applications are:

- Obtaining defaults for ML algorithm that work well across many datasets (Probst et al., 2018);

- Measuring the importance of hyperparameters, to investigate which should be tuned (see van Rijn and Hutter, 2017; Probst et al., 2018);

- Obtaining ranges or priors of tuning parameters to focus on important regions of the search space (see van Rijn and Hutter, 2017; Probst et al., 2018);

- Meta-Learning;

- Investigating, debugging and improving the robustness of algorithms.

Possible weaknesses of the approach, which we would like to address in the future, are:

- For each ML algorithm, a set of considered hyperparameters and their initial ranges has to be provided. It would be much more convenient if the bot could handle the set of all technical hyperparameters, with infinite ranges.

- Smarter, sequential sampling might be required to scale to high-dimensional hyperparameter spaces. But note that we not only care about optimal configurations but much rather would like to learn as much as possible about the considered parameter space, including areas of bad performance. So simply switching to Bayesian optimization or related search techniques might not be appropriate.

References

B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones. mlr: Machine learning in R. *Journal of Machine Learning Research*, 17 (170):1–5, 2016.

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

G. Casalicchio, J. Bossek, M. Lang, D. Kirchhoff, P. Kerschke, B. Hofner, H. Seibold, J. Vanschoren, and B. Bischl. OpenML: An R package to connect to the machine learning platform OpenML. *Computational Statistics*, 32(3):1–15, 2017.

T. Chen and C. 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.

M. Claesen and B. D. Moor. Hyperparameter search in machine learning. *MIC 2015: The XI Metaheuristics International Conference*, 2015.

J. Deng, W. Dong, R. Socher, L. jia Li, K. Li, and L. Fei-fei. Imagenet: A large-scale hierarchical image database. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

M. Feurer, J. T. Springenberg, and F. Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 1128–1135. AAAI Press, 2015.

J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

D. Kühn, P. Probst, J. Thomas, and B. Bischl. OpenML R bot benchmark data (final subset). 2018. URL `https://figshare.com/articles/OpenML_R_Bot_Benchmark_Data_final_subset_/5882230`.

M. Lang, B. Bischl, and D. Surmann. batchtools: Tools for R to work on batch systems. *The Journal of Open Source Software*, 2(10), 2017.

D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, and F. L. h. *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*, 2017. URL `https://CRAN.R-project.org/package=e1071`. R package version 1.6-8.

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

M. Reif. A comprehensive dataset for evaluating approaches of various meta-learning tasks. In *ICPRAM*, 2012.

K. Schliep and K. Hechenbichler. *kknn: Weighted k-Nearest Neighbors*, 2016. URL `https://CRAN.R-project.org/package=kknn`. R package version 1.3.1.

M. R. Smith, A. White, C. Giraud-Carrier, and T. Martinez. An easy to use repository for comparing and improving machine learning algorithm usage. In *Meta-Learning and algorithm selection workshop at ECAI 2014*, page 41, 2014.

T. Therneau and B. Atkinson. *rpart: Recursive Partitioning and Regression Trees*, 2018. URL `https://CRAN.R-project.org/package=rpart`. R package version 4.1-12.

J. N. van Rijn and F. Hutter. Hyperparameter importance across datasets. *ArXiv preprint arXiv:1710.04725*, 2017. URL `https://arxiv.org/abs/1710.04725`.

J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

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

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12. Juli 2011, § 8 Abs. 2 Pkt. 5)

Hiermit erkläre ich an Eides statt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

München, den 02.05.2019

Philipp Probst