Master's Thesis

Automatic Gradient Boosting

Stefan Coors

supervised by
Prof. Dr. Bernd Bischl

co-supervised by
Janek Thomas, M.Sc.

Ludwig Maximilians University Munich
Department of Statistics

March 13, 2018

Automatic Gradient Boosting

Abstract

The technological progress of the last decades, especially in the computational technology, allowed the storage and fast analysis of a continuously increasing number of datasets. This development is widely omnipresent leading to a situation where the job of a data scientist is "the sexiest job of the 21st century" (Harvard Business Review). However, well-qualified data scientist are not a dime a dozen. Instead, employees being not much familiar with data analysis are often called to do the job.
Automatic machine learning can help those persons to perform predictive modeling with high performing machine learning tools without having much experience. This is achieved by making those applications parameter-free, i.e. only the data is required as input. The rest is done automatically. Projects like Auto-WEKA or auto-sklearn aim to solve the *Combined Algorithm Selection and Hyperparameter optimization (CASH)* problem resulting in a huge optimization space.
However, for most real world applications, only few different learning algorithms are required to deliver superior performances. **autoxgboost** simplifies this idea one step further and the CASH problem to taking *Gradient Boosting* as a single learning algorithm in combination with intelligent model based hyperparameter tuning. It is based on the **XGBoost** R-Package and also supports categorical variables due to special inbuilt factor feature encoding.
After describing the main concepts of gradient boosting and the autoxgboost package, several benchmarks are done to improve the package. This includes multiclass threshold tuning as well as the factor encoding of categorical features. Thereafter, autoxgboost is compared to Auto-WEKA and auto-sklearn in a benchmark looking on the predictive performance. We find out that even though autoxgboost only uses one learner instead of a whole library, it provides comparable or even better performances on some datasets. However, when limitations of the computational resources are not an issue, auto-sklearn still provides superior performance.

Contents

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

2 Gradient Boosting **4**
2.1 Predictive modeling framework 4
2.2 Gradient-descent optimization 8
2.3 Boosting 9
2.4 Gradient Boosting algorithm 11
2.4.1 Regression 12
2.4.2 Classification 15
2.5 eXtreme Gradient Boosting 18
2.6 Comparison of different boosting implementations 20

3 The concept of automatic gradient boosting **22**
3.1 Factor feature encoding 22
3.1.1 Integer encoding 22
3.1.2 Dummy encoding 23
3.1.3 Impact encoding 24
3.1.3.1 Binary classification 24
3.1.3.2 Multiclass classification 26
3.1.3.3 Regression 27
3.1.3.4 Impact encoding of vtreat 28
3.2 Hyperparameter tuning with mlrMBO 28
3.2.1 SMBO algorithm 30
3.2.2 XGBoost learner with early stopping 33
3.2.3 Autoxgboost parameter set 33
3.3 Threshold tuning for classification tasks 35
3.3.1 Intelligent grid search for binary classification 37
3.3.2 Generalized Simulated Annealing for multiclass classification 39
3.4 The API of autoxgboost 42

4 Further Auto-ML projects **43**
4.1 Single-learner approaches 43
4.2 Multiple-learner approaches 44

5 Benchmark experiments for optimizing autoxgboost **46**
5.1 Factor encoding benchmarks 46

5.1.1 Benchmark configuration 47
5.1.2 Benchmark results 49
5.2 Threshold tuning benchmarks 57
5.2.1 Algorithm benchmark 57
5.2.1.1 Benchmark configuration 57
5.2.1.2 Benchmark results 59
5.2.2 Hyperparameter benchmark 61
5.2.2.1 Benchmark configuration 61
5.2.2.2 Benchmark results 62

6 Autoxgboost benchmarks 64
6.1 Benchmark configuration 64
6.2 Benchmark results 65

7 Conclusion 68

References 70

A Appendix vi
A.1 Additional figures of the factor encoding benchmark vi
A.2 Additional figures and tables of the GenSA tuning benchmark . . . xiv
A.3 Repositories containing the R-Code of this thesis xviii

List of Figures

1 Machine learning work-flow 1
2 Comparison between the classical statistical approach and machine learning. 4
3 Common loss functions for regression. 6
4 Common loss functions for classification. 7
5 Illustration of gradient descent optimization. 8
6 Differences of leaf wise and level wise tree construction. 21
7 General SMBO Framework 30
8 Optimization states of different iterations of an mlrMBO example . 32
9 ROC curve and corresponding measured values of a logistic regression classifier for german credit data depending on the threshold. 37
10 Performance comparison for theoretical and tuned thresholds on the credit costs and the mmce. 38
11 Benchmark results for KDD appetency dataset. 50
12 Benchmark results for Amazon dataset. 51
13 Benchmark results for cjs dataset. 52
14 Benchmark results for KDD98 dataset. 53
15 Average ranks of all methods on all datasets. 54
16 Average ranks of all methods on all datasets using ber and mse as performance measure. 55
17 Mean (left) and individual success-rate comparison between the benchmarked optimization functions. 60
18 Overview and detailed view of each GenSA parameter configuration performance. 62
19 Benchmark results for credit-g dataset. vi
20 Benchmark results for adult dataset. vii
21 Benchmark results for KDD churn dataset. viii
22 Benchmark results for KDD upselling dataset. ix
23 Benchmark results for bank-marketing dataset. x
24 Benchmark results for eucalyptus dataset. xi
25 Benchmark results for ozone dataset. xii
26 Benchmark results for nasa dataset. xiii
27 Average ranks of overall best GenSA parameter settings xiv
28 Average ranks of best GenSA parameter settings 3-class-datasets . xv
29 Average ranks of best GenSA parameter settings 4- and 5-class-datasets xvi
30 Average ranks of best GenSA parameter settings 8- and 10-class-datasets xvii

List of Tables

1 Scheme of integer encoding . . . 23
2 Scheme of creating dummy features . . . 24
3 Scheme of creating impact features (binary class) . . . 25
4 Example values of creating impact features from Table 3 . . . 26
5 Scheme of creating impact features (multiclass) . . . 26
6 Scheme of creating impact features (regression) . . . 28
7 XGBoost hyperparameters tuned by mlrMBO (Description taken from package manual). . . . 34
8 Cost matrix for credit data. . . . 38
9 Performances for different thresholds . . . 38
10 Arguments of the `autoxgboost()` function . . . 41
11 Arguments of the `AutoxgbResult` object . . . 42
12 Different encoding methods and boundary configurations used for the benchmark. . . . 47
13 Datasets used for the encoding benchmark and their properties. . . 48
14 Datasets used for the threshold tuning benchmark and their properties. 58
15 GenSA hyperparameters and ranges for the benchmark . . . 61
16 Best GenSA parameter configurations. . . . 63
17 Datasets used for the comparison benchmark of autoxgboost, Auto-WEKA and auto-sklearn. . . . 64
18 Autoxgboost benchmark results. . . . 66
19 Best overall GenSA parameter settings . . . xiv
20 Best GenSA parameter settings for datasets with 3 classes . . . xv
21 Best GenSA parameter settings for datasets with 4 and 5 classes . xvi
22 Best GenSA parameter settings for datasets with 8 and 10 classes . xvii

1. Introduction

Topics like *Big Data*, *Artificial Intelligence*, *Machine Learning* and *Advanced Analytics* are remaining ongoing topics for several years now. While data is gathered everywhere nowadays, much of its insight is still unused since data scientists are still rare. The needed know-how to analyze the data in a proper way is not present in many cases. This is a big problem for many companies of all sectors which are continuously looking for well-qualified personnel. Figure 1 shows the whole work-flow of a data scientist. It all starts with creating a plan which contains the main questions and goals of an analysis. After collecting the necessary datasets, one often needs to preprocess the data which could include imputing missing values or recoding one or more variables. The main data analysis then contains model building and evaluation/visualization. The illustration in Figure 1 is similar to the *Cross Industry Standard Process for Data Mining (CRSIP)* presentation by Shearer (2000), but more detailed. As on can see, several steps are connected and repeated until the final results are obtained.
While Planing needs to be done manually by the data scientist, the other steps and

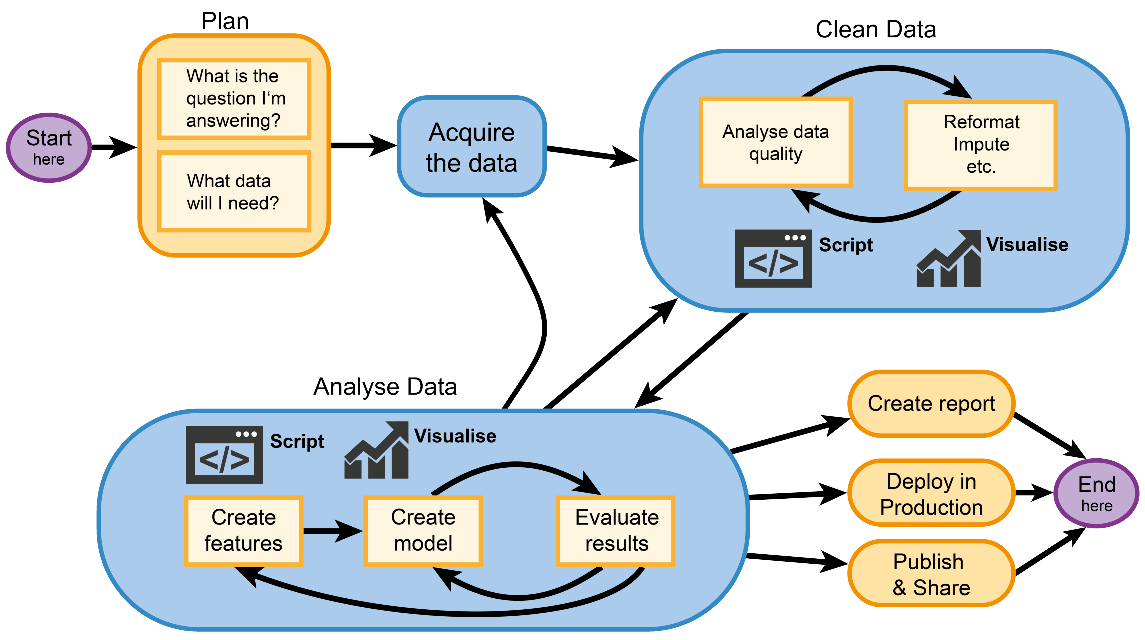


Figure 1: Machine learning work-flow: While some steps need to be done manually, especially the analytic part can be automated (inspired by Hortonworks Inc. (2015) The Data Science Work-flow).

especially the technical analysis of the data can be automated. Those analysis do more and more often include especially machine learning algorithms which require special skills by the data scientist. However, since many superior techniques are relatively new or only applicable due to technological and especially computational progress, the entry barrier for using such methods is not overcome too often. To counteract this dilemma, programers develop user-friendly software or graphical user interfaces (GUI) which are based on typical programing languages like *R* or *python* for applying machine learning tasks by end users being not coding-affine. One example of an R-based GUI which uses the **mlr** package by Bischl, Lang, Kotthoff, et al. (2016) is **shinyMlr** by Coors and Fendt (2017). It allows users to fully perform machine learning tasks without writing a single line of code. However, there are way more open source approaches for making machine learning easier to access for non-professional data scientists, but also professional commercial ones.
Global players like Google stepped into the market when introducing *AutoML*. This architecture allows it to make machine learning and especially neural nets a lot more accessible to people than before. This is happening by automating almost the whole machine learning work-flow. AutoML therefore uses neural networks itself to select the best suitable network for the problem including hyperparameter tuning as well as stacking together several models. However, the idea of automating machine learning processes is not new. Hall et al. (2009) introduced an automated work-frame for choosing from a broad variety of learning algorithms from their open source package **WEKA**. Hereby, *Auto-WEKA* (see Thornton et al. (2013)) simultaneously sets the optimal hyperparameters for the selected model using Bayesian optimization. All included algorithms support classification tasks, some also regression tasks.
Very similar to Auto-WEKA is **auto-sklearn** by Feurer et al. (2015), which is based on the scikit-learn toolkit for python and includes all of its learners.
The package introduced in this master's thesis simplifies the idea of automatic machine learning even further. It uses *Gradient Boosting*, which is an high-performance algorithm, if its hyperparameters are adequately tuned, and reduces the whole framework to constructing the optimal gradient boosting model. The whole package is R-based and uses **mlr** as a base in combination with the *eXtreme Gradient Boosting* (**XGBoost**) package by Chen and Guestrin (2016). Besides tuning the hyperparameters via Bayesian optimization using the **mlrMBO** package by Bischl, Richter, et al. (2017), factor feature transformation is performed as preprocessing step if necessary. Moreover, for classification tasks, threshold tuning for the trained models is

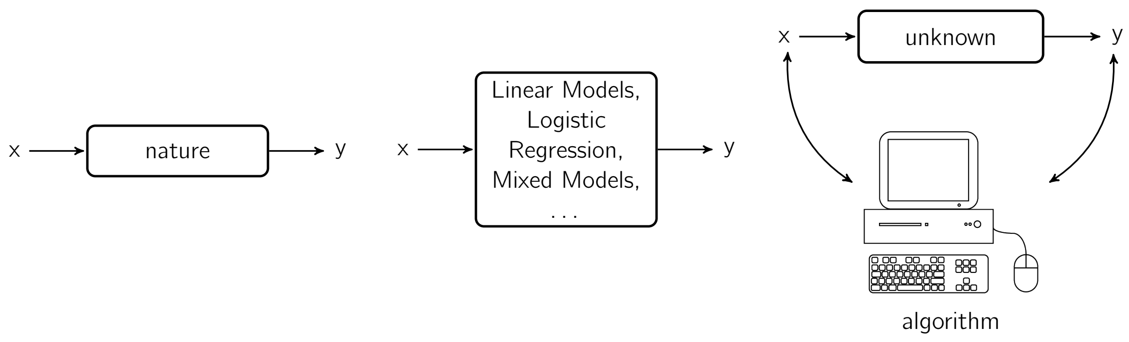
performed as a last optimization step via **mlr**. Due to its technical base, the automatic gradient boosting approach introduced in this thesis is called *autoxgboost*.
The thesis' content is structured as follows. The next section introduces gradient boosting in detail, since it is the basic learning concept of autoxgboost whose concept is described afterward in Section 3. It is compared to other Auto-ML projects in the subsequent Section 4. As a next step, since XGBoost cannot handle factor features, they need to be transformed. In order to do this properly, several different methods were benchmarked and therefore introduced in Section 3. The results can be seen in Section 5.1. While optimizing each part of autoxgboost, it became apparent that there was potential of improving the threshold tuning algorithm for multiclass classification tasks. Since this optimization part was used for all multiclass tasks, it needed to be optimized before finally comparing autoxgboost with other competitors. The underlying technique is described as part of Section 3 while the performed benchmark experiments for first finding the best optimizer and second its optimal hyperparameters, is described in Section 5.2.
After optimizing the technical base of the framework, autoxgboost is finally benchmarked against auto-WEKA and auto-sklearn in order to compare its performance for real world data in Section 6. After describing the benchmark settings, the performance results are illustrated in that section. We will see, that autoxgboost is able to compete with its competitors on some datasets.
Finally, the conclusion in Section 7 gives a summary of all topics and findings of this thesis and an outlook on further research. It provides ideas for further improvements of the automatic gradient boosting framework.
Note, that large work packages of this master's thesis consisted of creating R-code during the formation phase. Not only parts of the autoxgboost but also all performed benchmarks required a lot of effort which cannot be obviously seen in the written thesis.

2. Gradient Boosting

Gradient boosting, which was introduced by J. H. Friedman (2001), combines two different techniques, *Boosting* in combination with *gradient descent* which is also known as *steepest descent* method. To introduce and understand both concepts, some theoretical background needs to be discussed first. Hence, before concentrating directly on gradient boosting in this section, this technical framework will be explained in the next parts.

2.1. Predictive modeling framework

When introducing machine learning methods, a comparison with traditional approaches is obvious. Figure 2 shows the initial situation and compares the classical statistical approach with the machine learning one. Here, Figure 2a, which is inspired by illustrations of Breiman (2001), shows the relationship between input x and output y which is described by an unknown function of the nature which is also called *data generating process*, as seen in Figure 2b. Classical statistic approaches try to describe this relationship by a interpretable model. Those models are usually based on several assumptions which may or may not be supported by the data. If not, the model should be questioned and probably discarded.
In contrast, machine learning does not even try to model the relationship directly. Instead, it considers the connection as a black-box function which is approximated as good as possible by a learning algorithm using input x and output y (see Figure 2c).



(a) Initial situation (b) Classical statistics (c) Machine Learning

Figure 2: Comparison between the classical statistical approach and machine learning.

Thus, assumptions of e.g. several distributions within the data are not made and can therefore not be violated. Hence, machine learning methods are very good in imitating nature at the expense of interpretability. As a result, the data scientist always needs to check whether he requires this interpretability of the effects of single input variables and choose traditional statistical methods, or rather prefers a higher prediction performance resulting in the use of machine learning.
The typical predictive modeling setting contains a system with a d-dimensional random *response* vector $y \in \mathbb{R}^d$ and a set of *features* $x = \{x_1, \dots, x_n\}$. The features x are especially in statistics called *explanatory variables* or in the machine learning context *input*, while y is also called *output*. The goal of predictive modeling is then to use a *training dataset*, containing tuples (x_i, y_i) for $i = 1, \dots, n$, to approximate or estimate the unknown system by a prediction function $f()$, such that

$$f(x) = y \tag{1}$$

A good prediction is measured by a *loss function* $L(y, f(x))$ and its expected value, so so-called *risk*

$$\mathcal{R}(f(x)) = \mathbb{E}[L(y, f(x))] = \int L(y, f(x)) d\mathbb{P}_{xy}. \tag{2}$$

In this regard, the loss is determined by the point-wise deviance of the estimated model $\hat{f}(x)$ and the real data points y and simultaneously weights this distance. As a matter of principle, loss functions can be arbitrarily chosen. However, for regression, mostly used loss functions are the *quadratic loss* or *least-squares loss*:

$$L(y, f(x)) = (y - f(x))^2, \; -\frac{\partial L}{\partial f(x)} = 2(y - f(x)) \tag{3}$$

which is equivalent to a maximum-likelihood approach for normally distributed errors and therefore also sometimes called *Gaussian loss*. Moreover, it is differentiable. As one can see in Figure 3, the quadratic loss progressively weights points with higher distance to $\hat{f}(x)$. Hence it is not robust regarding outliers.
An alternative loss function is the *absolute loss*:

$$L(y, f(x)) = |y - f(x)|, \; -\frac{\partial L}{\partial f(x)} = sign(y - f(x)) \quad \text{for } x \neq 0.$$

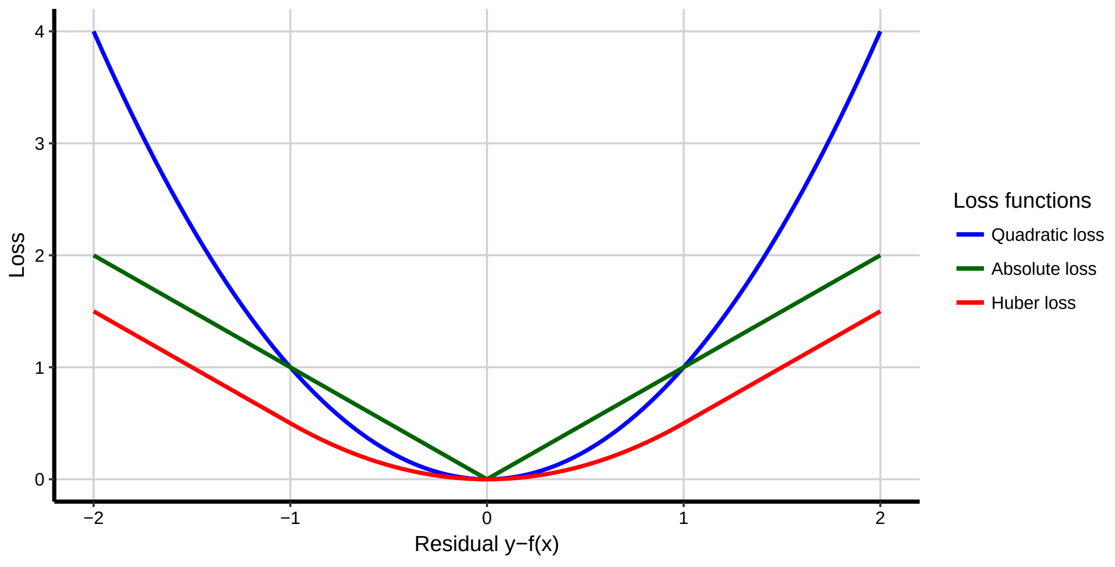


Figure 3: Common loss functions for regression.

It is equivalent to Laplace distributed errors, but in contrast to the quadratic loss not differentiable at 0 for $y = f$. Nevertheless, it is robust which can be seen in Figure 3.
The *Huber loss* by Huber (1964) combines the advantages of both loss functions above by being a combination. For an arbitrary $\delta > 0$ it is defines as

$$L(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \quad , hence$$

$$-\frac{\partial L}{\partial f(x)} = \begin{cases} y - f(x) & \text{for } |y - f(x)| \leq \delta \\ sign(y - f(x)) & \text{otherwise} \end{cases}$$

It is differentiable and robust since it is quadratic in an interval around 0 following by a linear continuation. Its graph is also illustrated in Figure 3.
When switching to binary classification tasks, other loss functions are usually used, since the ones for regression are not reasonable. Unfortunately, the *Zero-One loss* is not smooth and hence not suitable for optimization as seen in Figure 4. Instead, smooth and convex loss functions are typically used for classification of which some examples are also shown in Figure 4. A popular one is the *exponential loss* which is

defined as

$$L(y, f(x)) = \begin{cases} \exp(-yf(x)) & \text{for } y \in \{-1, +1\} \\ \exp(-(2y-1)f(x)) & \text{for } y \in \{0, 1\} \end{cases}$$

It is less robust to strongly misclassified observations compared to the following methods because of the exponential growth for negative values.
Moreover, the so-called *truncated Hinge loss*, which was introduced for support vector machines by Wu and Liu (2007), is suitable for classification tasks. The loss value is determined by

$$L(y, f(x)) = \max(0, 1 - yf(x)) = |1 - yf(x)|_+.$$

Due to linearity for negative values, it is more robust. The same yields for another possibility, which is to take the same loss function corresponding to the negative log-likelihood of logistic regression. It is called *binomial loss* or *cross-entropy loss*:

$$L(y, f(x)) = \begin{cases} \ln(1 + \exp(-2yf(x)) & \text{for } y \in \{-1, +1\} \\ -yf(x) + \ln(1 + \exp(f(x))) & \text{for } y \in \{0, 1\} \end{cases} \tag{4}$$

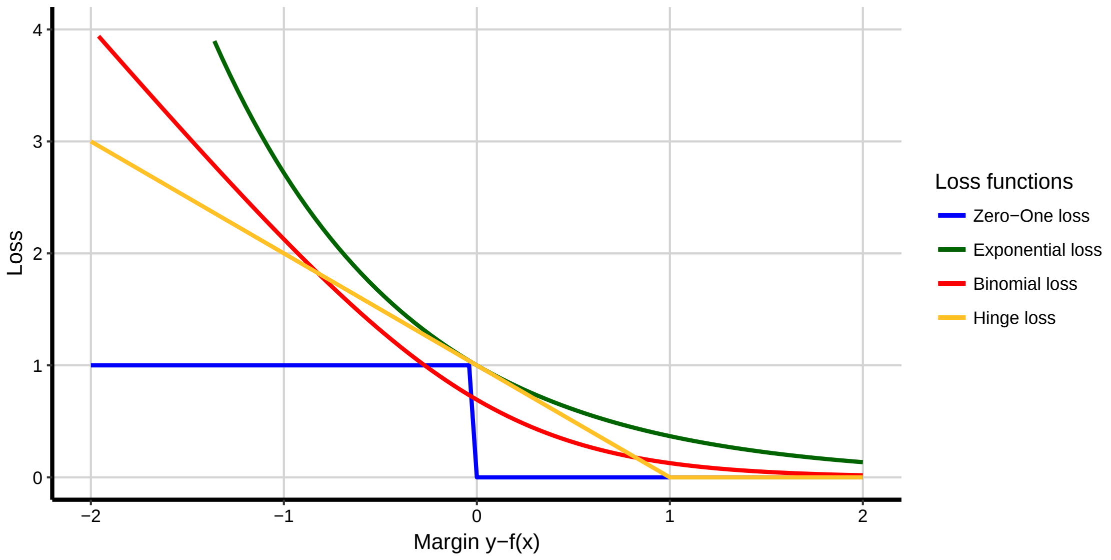


Figure 4: Common loss functions for classification.

For all maximum-likelihood models, a similar approach is possible resulting in the use of the negative log-likelihood as loss function. In general, a good loss function approximates the Zero-One loss sufficiently and has in addition appropriate properties. The binomial loss function can simply be extended for multiclass classification problems by assuming a multinomial model instead, i.e. the negative log-likelihood of the resulting multinomial logistic regression forms the loss function which is also known as *softmax* function. A variation of this function, which returns the predicted probabilities of each data point belonging to each class, is also used as objective function in autoxgboost for multiclass classification and called *softprob*.
Obviously, one is interested in finding an optimal estimate $\hat{f}(x)$ which minimizes the risk $\mathcal{R}(f(x))$ over the joint distribution of all tuples of the training set. Hence, with Eq. (2), $\hat{f}(x)$ is determined by

$$\begin{aligned}\hat{f}(x) &= \underset{f(x)}{\arg\min}\ \mathcal{R}(f(x)) = \underset{f(x)}{\arg\min}\ \mathbb{E}[L(y, f(x))] \\ &= \underset{f(x)}{\arg\min}\ \mathbb{E}_{x,y}\left[L(y, f(x))\right] \\ &= \underset{f(x)}{\arg\min}\ \mathbb{E}_{x}\left[\mathbb{E}_{y}(L(y, f(x)))|x\right].\end{aligned} \tag{5}$$

2.2. Gradient-descent optimization

When generally looking at optimization problems, one can separate between deterministic and stochastic methods. Deterministic approaches are usually faster than stochastic ones, however, the risk for being trapped in a local minima is significantly higher. Later in this thesis, some stochastic methods for hyperparameter- and threshold tuning will be introduced. Nevertheless, gradient-descent is a deterministic non-parametric iterative method for numerical function optimization which has often been proposed for minimizing the empirical risk (see e.g. Rumelhart et al. (1986)). We consider the situation of Eq. (5) with an arbitrary, differentiable and unrestricted target function $f(x)$. The *gradient* $\nabla f(x)$ can be understood as pointer which shows always in the direction of the *steepest ascent* of the func-

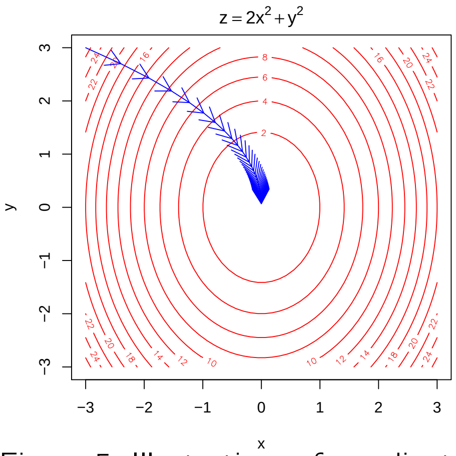


Figure 5: Illustration of gradient descent optimization.

tion. Likewise, $-\nabla f(x)$ points towards the *steepest descent* of $f(x)$. Hence, the gradient represents the tangent's slope of the function graph, which is similar to the derivative.
However, in contrast to the scalar-valued derivative, the gradient is vector valued containing the above described direction and depending on the underlying space, its representation differs. Applying the *nabla* operator ∇ on function f results in a unique vector field which is called gradient. Hence, for a function $f : \mathbb{R}^n \to \mathbb{R}$ we have

$$\nabla f(x) = \text{grad } f(x) = \left(\frac{\partial f(x)}{x_1}, \dots, \frac{\partial f(x)}{x_n} \right)^\top . \tag{6}$$

Then, as a first step for minimizing function $f(x)$, we select a starting point $x^{(0)}$ as initial guess. This point can be improved, i.e. we select the next point $x^{(1)}$ such that

$$x^{(1)} = x^{(0)} - \nu \nabla f(x^{(0)}),$$

so in general for iteration m,

$$x^{(m)} = x^{(m-1)} - \nu \nabla f(x^{(m-1)}) \quad \text{for } m = 1, \dots, M, \tag{7}$$

where ν controls the *step size* towards the steepest descent. In each iteration, the optimal ν is allowed to change. An obvious choice is the one which minimizes the objective function:

$$\nu^{(m)} = \underset{\nu>0}{\arg\min}\, f\left(x^{(m-1)} - \nu \nabla f(x^{(m)})\right) . \tag{8}$$

Eq. (8) is called *line search*. If the algorithm iteratively reaches an $x^{(m)} \in \mathbb{R}^n$ with $\nabla f\left(x^{(m)}\right) = 0 \in \mathbb{R}^n$, a local minima is reached. Figure 5 illustrates the whole procedure for a two dimensional function $f(x, y) = 2x^2 + y^2$. One can see how the step size and direction changes for each iteration.

2.3. Boosting

Boosting had its first appearance in form of a simple procedure in the PAC-learning framework of Valiant (1984) and Kearns and Vazirani (1994). It was developed by Robert E. Schapire et al. (1998) who were also the first who showed that the performance of a *weak learner* could always be improved by training additional ones. The only required property of such weak learners is to outperform random chance regarding predictive performance. Boosting is based on this concept which corresponds to *stagewise additive modeling*. Following J. Friedman et al. (2000) and J. H. Friedman (2001), the idea is to learn an additive model

$$f(x) = \sum_{m=1}^{M} f_m(x) = \sum_{m=1}^{M} \beta_m h(x, \theta_m). \tag{9}$$

Obviously, the goal is to minimize the empirical risk for which we obtain with Eq. (2)

$$\mathcal{R} = \sum_{i=1}^{n} L(y_i, f(x_i)) = \sum_{i=1}^{n} L\left(y_i, \sum_{m=1}^{M} \beta_m h(x_i, \theta_m)\right). \tag{10}$$

It clearly depends on the base learners $h(x, \theta_m)$ and especially their weights β_m and parameters θ_m. Hence, $\mathcal{R}$ has to be minimized regarding parameters $(\beta, \theta) = ((\beta_1, \theta_1), \ldots, (\beta_M, \theta_M))$ which, if tried directly, can be difficult depending on the chosen loss function L. Therefore, optimization can be done using a the iterative *"greedy" forward stagewise additive modeling* approach (see J. Friedman et al. (2000)). Thus for optimizing

$$(\beta^*, \theta^*) = \arg\min_{\beta,\theta} \sum_{i=1}^{n} L\left(y_i, \sum_{m=1}^{M} \beta_m h(x_i, \theta_m)\right), \tag{11}$$

we can use

$$(\beta_m^*, \theta_m^*) = \arg\min_{\beta,\theta} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \beta h(x_i, \theta)) \tag{12}$$

in order to get

$$f_m(x) = f_{m-1}(x) + \beta_m h(x_i, \theta_m) \tag{13}$$

with

$$f_{m-1}(x) = \sum_{j=1}^{m-1} \beta_j h(x_i, \theta_j). \tag{14}$$

Here, it is important to clarify that adding each component stagewise means that previous models are fixed and hence not readjusted. This strategy is called boosting in the machine learning context, but is also known as *matching pursuit* in signal processing (see Mallat and Zhang (1993)).
A typical *weak learner* $h(x, \theta)$, also called *base function*, are *tree stumps*, which are decision trees with only few split points. Those tree stumps bring several advantages of decision trees which include supporting categorical features and missing values or robustness regarding outliers. Moreover, training a tree is relatively fast compared to other algorithms. Additionally, a variable selection mechanism is conceptually included.
On the other hand, boosting is able to strongly improve the predictive predictive performance compared to training a single tree. However, interpretability is obviously lost by combining several trees. Those advantages are similar to the *random forest* approach which uses *bootstrap aggregation* to combine multiple decision trees for learning (see Breiman (1996)).
One of the first boosting algorithms was *AdaBoost* (Adaptive Boosting) by Freund and Robert E Schapire (1997).

2.4. Gradient Boosting algorithm

The gradient boosting algorithm combines the gradient descent method from Section 2.2 and boosting which was introduced in the previous Section 2.3. That means gradient boosting uses stagewise additive models for which the empirical risk $\mathcal{R}$ is minimized by gradient descent.
In detail, for the additive model of Eq. (9), we are looking for an optimal parameter combination (β_m^*, θ_m^*) like in Eq. (12), i.e. we want to determine the new additive component $\beta_m h(x_i, \theta_m)$ of Eq. (13) for iteration m. Here, Eq. (13), β_m is the step size of the gradient descent, in the previous section also denoted as ν. It is also called *learning rate* or *shrinkage parameter*. If $0 < \beta_m \ll 1$, only a small amount of the base learner is taken into account in the m-th iteration. This helps preventing the additive model from overfitting. Moreover, following Schmid and Hothorn (2008), far more important than the concrete choice of β_m is the fact that it is chosen small enough.

2.4.1. Regression

First, consider a non-parametric model, where predictions can arbitrarily made for every single observation x_i of the n observations of the training dataset (see Bischl (2017)). This results in n parameters $f(x_i)$, but no generalization on the whole space $\mathcal{X}$. Using the gradient descent we can obtain the gradient with Eq. (6) for a differentiable loss function L at a point x_j for each iteration by

$$\begin{aligned}\nabla\mathcal{R}|_{x_j} &= \frac{\partial\mathcal{R}}{\partial f(x_j)}\\ &= \frac{\partial\sum_{i=1}^{n} L(y_i, f(x_i))}{\partial f(x_j)}\\ &= \frac{\partial L(y_j, f(x_j))}{\partial f(x_j)} \qquad (15)\end{aligned}$$

Hence, the update for iteration m by gradient descent is

$$f_m(x_j) \leftarrow f_{m-1}(x_j) - \beta\frac{\partial\sum_{i=1}^{n} L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_j)} \qquad (16)$$

As a consequence, we can determine the directions of the steepest descent for each x_i and define these as *pseudo residuals* r_{im}

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}. \qquad (17)$$

Thus, the optimal weight β_m for iteration m can obtain by setting $r_{im} = h(x_i, \theta_m)$ in the line search of Eq. (12):

$$\beta_m = \arg\min_{\beta} \sum_{i=1}^{n} L\left(y_i, f_{m-1}(x_i) - \beta\left[\frac{\partial L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)}\right]\right) \qquad (18)$$

However, as stated above, this only holds for a single observation x_j of the training set. Hence, a generalization for all $x \in \mathcal{X}$ is necessary which can be achieved by approximating the negative gradient as good as possible by a regression model, previously called base function or weak learner in Section 2.3,

$$h(x, \theta_m) = -r_m = -\left[\frac{\partial L(y_j, f(x_j))}{\partial f(x_j)}\right] = -\left[\frac{\partial \sum_{i=1}^{n} L(y_i, f(x_i))}{\partial f(x_j)}\right] \tag{19}$$

Again, minimizing the risk

$$\mathcal{R}(h(x, \theta_m)) = L(h(x, \theta_m), r_m) \tag{20}$$

leads to

$$\theta_m = \arg\min_{\theta} \sum_{i=1}^{n} L(r_i m, h(x_i, \theta)) \tag{21}$$

which provides us our optimal parameter θ for Eq. (12). Finally, the new whole additive part containing the weak base learner $h(x, \theta_m)$ can be interpreted as component, which moves the model towards the highest decrease of loss, where β_m, determined by Eq. (18), expresses the step size for this move. When using the common least-squares loss function for regression problems, Eq. (21) reduces to

$$\theta_m = \arg\min_{\theta} \sum_{i=1}^{n} (r_i m - h(x_i, \theta))^2. \tag{22}$$

Basically every regression learner is able to be fitted via the quadratic loss. Additionally, solving is numerically efficient. Algorithm 1 combines all gradient boosting steps synoptically (see J. H. Friedman (2001) and Bischl (2017)).
As mentioned before, selecting decision trees as base learner has advantages which makes them also the preferred choice for autoxgboost. Following J. H. Friedman (2001), one can write such a tree as

$$h(x, b, R) = \sum_{j=1}^{J} b_j \mathbb{I}(x \in R_j), \tag{23}$$

where R_j are the disjoint regions, defined by the terminal nodes of the tree with the corresponding means γ_j which specify the boundaries of regions R_j. Putting Eq. (23)

Algorithm 1: Gradient Boosting Algorithm.

```
   Initialize: $f_0(x) = \arg\min_{\theta_0} \sum_{i=1}^{n} L(y_i, \theta_0)$
1  for $m = 1 \to M$ do
2      for all $i$ do
3          Calculate $r_{im} = -\left[\frac{\partial L(y, f(x))}{\partial f(x_i)}\right]_{f(x_i) = f_{m-1}(x_i)}$
4      end
5      Fit regression base learner to the pseudo-residuals $r_{im}$:
6          $\theta_m = \arg\min_{\theta} \sum_{i=1}^{n} (r_{im} - h(x_i, \theta))^2$
7      Find via line search:
8          $\beta_m = \arg\min_{\beta} \sum_{i=1}^{n} L(y_i, f_{m-1}(x) + \beta h(x, \theta_m))$
9      Update $f_m(x) = f_{m-1}(x) + \beta_m h(x, \theta_m)$
10 end
   Output: $\hat{f}(x) = f_M(x)$
```

into Eq. (13) leads to

$$f_m(x) = f_{m-1}(x) + \beta_m \sum_{j=1}^{J_m} b_{jm} \mathbb{I}(x \in R_{jm}) \tag{24}$$

which can be reduced to

$$f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} \mathbb{I}(x \in R_{jm}) \tag{25}$$

when setting $\gamma_j m = \beta_m b_{jm}$, where like before, β_m is determined by line search (see J. H. Friedman (2001)). Again, minimizing the loss function provides the optimal coefficients for γ_{jm} which is done by

$$\gamma_{jm} = \arg\min_{\gamma_j} \sum_{i}^{n} L\left(y_i, f_{m-1}(x_i) + \sum_{j=1}^{J_m} \gamma_{jm} \mathbb{I}(x \in R_{jm})\right). \tag{26}$$

Algorithm 2: Gradient Tree Boosting Algorithm.

```
Initialize: $f_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$
1  for $m = 1 \to M$ do
2      for all $i$ do
3          Calculate $r_{im} = -\left[\frac{\partial L(y, f(x))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)}$
4      end
5      Fit regression tree to the pseudo-residuals $r_{im}$ given terminal regions
       $R_{jm},\ j = 1, \dots, J_m$:
6      for $j = 1 \to J_m$ do
7          $\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$
8      end
9      Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} \mathbb{I}(x \in R_{jm})$
10 end
Output: $\hat{f}(x) = f_M(x)$
```

Due to the property that regions R_j are disjoint, this can be reduced to

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma). \tag{27}$$

Regarding a loss function L, this result is the optimal constant update given f_{m-1}. It can be determined individually and directly within each terminal region while the tree structure is still included. The above changes Algorithm 1 to Algorithm 2.

2.4.2. Classification

The general gradient boosting Algorithm 1 is obviously fully dependent of the loss function L. For the regression case, we chose the least-squares loss which was equivalent to the maximum-likelihood approach for normally distributed errors. For classification, we already saw suitable loss functions in Section 2.1 whose mathematical derivation we want to discuss a bit more in detail in this section. When restricting to binary classification first, the obvious difference to regression is the fact that our target variable does not contain continuous, but only two different class levels, e.g. $y \in \{0, 1\}$. If the model's output maps on real values, one can regard positive values as indication for class 1 and negative values for class 0 respectively. Hence, we get discrete predictions by $\mathbb{I}(f(x) > 0)$. Alternatively, we transform our model such that its function values lie in the interval $[0, 1]$, which can then be interpreted as probabilities for class 1. This can be achieved by applying the *logistic distribution function*

$$\text{logistic}(\eta) = \frac{\exp(\eta)}{1 + \exp(\eta)}, \tag{28}$$

where η is called *link function*, since it links the model with the prediction probabilities. Following Mehta and Patel (1995), Bischl (2017) and J. Friedman et al. (2000), denote

$$\pi_{i1} = \mathbb{P}(y_i = 1|x_{i1}, \ldots, x_{ik}) = \text{logistic}(\eta_i) = \frac{\exp(\eta_i)}{1 + \exp(\eta_i)}, \tag{29}$$

where

$$\eta_i = x_i^\top \beta_i, \quad x_i = x_{i1}, \ldots, x_{ik}.$$

Hence, probability π_{i1} is not directly modeled, but indirectly via the *logit* function

$$\eta_i = \text{logit}(y_i = 1|x_{i1}, \ldots, x_{ik}) = \ln \frac{\pi_{i1}}{1 - \pi_{i1}} = x_i^\top \beta_i. \tag{30}$$

When applying the maximum likelihood approach for the logistic regression model, one gets the log-likelihood

$$\begin{aligned} &\sum_{i=1}^{n} (y_i \ln \pi_{i1} + (1 - y_i) \ln(1 - \pi_{i1})) \\ =&\sum_{i=1}^{n} (y_i f(x_i) - \ln(1 + \exp(f(x_i)))) \end{aligned} \tag{31}$$

for $f(x_i) = x_i^\top \beta_i$. One can now define the negative log-likelihood of Eq. (31) as new loss function which we have already seen in Eq. (4) in Section 2.1. That is

$$L(y, f(x)) = -yf(x) + \ln(1 + \exp(f(x))), \text{ with } -\frac{\partial L}{\partial f(x)} = y - \pi_1(x),$$

where $\pi_1(x)$ an estimate for the posterior probability of class 1, that is $\hat{\mathbb{P}}(y = 1|x) = \text{logistic}(\hat{f}(x))$.

This can be generalized for the k-class classification problem, assuming a *multinomial* *model*. Therefore, with the maximum likelihood approach we obtain as loss function

$$L(y_k, f_k(x)) = -\sum_{k=1}^{K} y_k \ln \pi_k(x), \tag{32}$$

where $y_k = \mathbb{I}(y = k)$ for class k. Hence, the posterior probability of class k is given by

$$\pi_k(x) = \hat{\mathbb{P}}(y = k|x) = \frac{\exp(f_k(x))}{\sum_{j=1}^{J} \exp(f_j(x))} \tag{33}$$

which corresponds to the softmax function as mentioned in Section 2.1. This leads to the multiclass pseudo residuals $r_{ik,m}$ when substituting Eq. (33) into Eq. (32) and taking the first derivatives (see J. H. Friedman (2001):

$$r_{ik,m} = -\left[\frac{\partial L(y_{ik}, f_{k,m}(x_i))}{\partial f_{k,m}(x_i)}\right]_{f_{k,m}(x)=f_{k,m-1}(x)} = y_{ik} - \pi_{k,m-1}(x_i), \tag{34}$$

where $\pi_{k,m-1}(x_i)$ is derived from Eq. (33) for $f_{k,m-1}$. Summarizing the above, we see that K models (trees) are fitted in each iteration m to predict the pseudo residuals $r_{ik,m}$ for each class k. Each tree has J terminal nodes with regions $\{R_{1k,m} \ldots, R_{Jk,m}\}$ with corresponding updates

$$\gamma_{jk,m} = \arg\min_{\gamma_{jk}} \sum_{i=1}^{n} \sum_{k=1}^{K} \phi\left(y_{ik}, f_{k,m-1}(x_i) + \sum_{j=1}^{J} \gamma_{ik} \mathbb{I}(x_i \in R_{j,m})\right) \tag{35}$$

with $\phi(y_k, f_k(x)) = -y_k \ln \pi_k(x)$ from Eq. (32).
Following J. Friedman et al. (2000), this equation has no closed form solution. However, it can be approximated by a single *Newton-Raphson* step, which separates it into a single calculation for each terminal node, that is

$$\gamma_{jk,m} = \frac{K-1}{K} \frac{\sum_{x_i \in R_{jk,m}} r_{ik,m}}{\sum_{x_i \in R_{jk,m}} |r_{ik,m}|(1 - |r_{ik,m}|)} \tag{36}$$

which serves for the update

$$f_{k,m}(x) = f_{k,m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jk,m} \mathbb{I}(x \in R_{jk,m}). \tag{37}$$

Algorithm 3: K-class Classification Gradient Tree Boosting Algorithm.

```
   Initialize: $f_{k,0}(x) = 0, k = 1, \dots, K$
1  for $m = 1 \rightarrow M$ do
2      Set $\pi_k(x) = \frac{\exp(f_k(x))}{\sum_{j=1}^{J} \exp(f_j(x))}$ for $k = 1 \rightarrow K$ do
3          Calculate $r_{ik,m} = y_{ik} - \pi_{k,m-1}(x_i), i = 1, \dots, n$.
4          Fit regression tree to the pseudo-residuals $r_{ik,m}$ given terminal regions
            $R_{jk,m},\ j = 1, \dots, J_m$:
5          for $j = 1 \rightarrow J_m$ do
6              $\gamma_{jk,m} = \frac{K-1}{K} \frac{\sum_{x_i \in R_{jk,m}} r_{ik,m}}{\sum_{x_i \in R_{jk,m}} |r_{ik,m}|(1-|r_{ik,m}|)}$
7          end
8          Update $f_{k,m}(x) = f_{k,m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jk,m} \mathbb{I}(x \in R_{jk,m})$
9      end
10 end
   Output: $\hat{f}_k(x) = f_{k,M}(x)$
```

Finally, after M steps, $f_{k,M}(x)$ is returned as final model, leading to Algorithm 3.

2.5. eXtreme Gradient Boosting

XGBoost has several advantages which led to choose it for the autoxgboost package. A main reason is the strong connection between the developers of autoxgboost and the **mlr** package, since it uses the **mlr**'s framework in combination with its learner database. Therein, **XGBoost** is the leading gradient boosting implementation given its strong performance for most machine learning tasks compared to its competitor packages regarding both, speed and predictive quality. Another reason is that it natively supports parallel computing since a large part of its code back-end is written in `C++`. Consequently, XGBoost combines two very important advantages - compared to other gradient boosting implementations, it is fast by using a minimal amount of resources at the same time. Hence, the training of large datasets is not a problem for the algorithm. This made it also very popular for several machine learning competitions, e.g. on *Kaggle*, often XGBoost is used by the winning teams. Finally, XGBoost allows to use custom objective functions for model building which allows its use in a very flexible manner.
More in detail, the idea of Chen and Guestrin (2016) is to introduce regularization

of the objective function

$$L(y, f(x)) = \sum_{i=1}^{n} L(y_i, f(x_i)) + \sum_{m=1}^{M} \Omega(f_m(x)), \tag{38}$$

with

$$\Omega(f(x)) = \gamma |f(x)| + \frac{1}{2}\lambda \|\omega|^2 \tag{39}$$

where $|f(x)|$ is the number of leafs of tree $f(x)$ and ω is the weight vector containing each leaf's score. Hence, Ω can be understood as a combination of Lasso regularization of γ and ridge regularization of coefficient λ. It serves as a regularizer which penalizes the model complexity in order to avoid over-fitting.
Unfortunately, Eq. (38) cannot be optimized by traditional methods. Nevertheless, following J. Friedman et al. (2000), a second order Taylor approximation can be used for optimization, that is for boosting iteration m

$$L(y, f_m(x)) \approx \sum_{i=1}^{n} \left[L(y_i, f_{m-1}(x_i)) + g_i f_m(x_i) + \frac{1}{2} h_i f_m^2(x_i) \right] + \Omega(f_m(x)) \tag{40}$$

$$\propto \sum_{i=1}^{n} \left[g_i f_m(x_i) + \frac{1}{2} h_i f_m^2(x_i) \right] + \Omega(f_m) = \hat{L}(y, f_m(x)) \tag{41}$$

where $g_i = \frac{\partial L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)}$ and $g_i = \frac{\partial^2 L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}^2(x_i)}$. Define $I_j := \{i | q(x_i) = j\}$ as the set of all observations of leaf j, where $q(x_i)$ is the decision rule to classify observation x_i into leaf j. Using this definition and Eq. (39), Eq. (41) can be written as

$$\hat{L}(y, f_m(x)) = \sum_{i=1}^{n} \left[g_i f_m(x_i) + \frac{1}{2} h_i f_m^2(x_i) \right] + \gamma |f_m(x)| + \frac{1}{2}\lambda \sum_{j=1}^{|f_m(x)|} \omega_j^2 \tag{42}$$

$$= \sum_{i=1}^{|f_m(x)|} \left[\left(\sum_{i \in I_j} g_i \right) \omega_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) \omega_j^2 \right] + \gamma |f_m(x)|. \tag{43}$$

Instead of entropy or information gain, XGBoost uses an own scoring function which is similar to the impurity score for evaluating decision trees. Since it is usually impossible to determine all possible tree structures q, Chen and Guestrin (2016) use a greedy algorithm which iteratively adds branches to a tree starting by a single leaf. Thus, assume I_L and I_R as the instance sets of the left and right child nodes. Then, the split criteria is given by

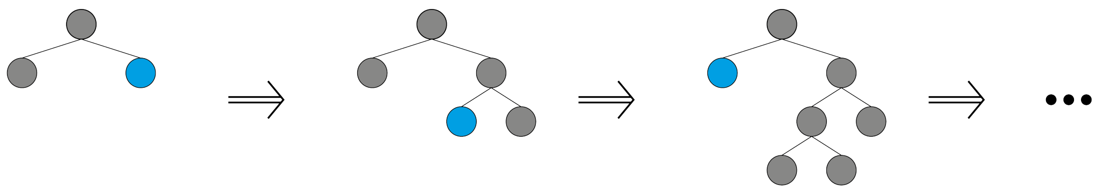
$$\frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i+\lambda}+\frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i+\lambda}-\frac{\left(\sum_{i\in\{I_L\cup I_R\}} g_i\right)^2}{\sum_{i\in\{I_L\cup I_R\}} h_i+\lambda}\right]-\gamma. \tag{44}$$

For an even more detailed look on the theoretical background of XGBoost and comparison benchmarks including e.g. **GBM** and **H2O-GBM**, the interested reader is referred to the paper of Chen and Guestrin (2016).

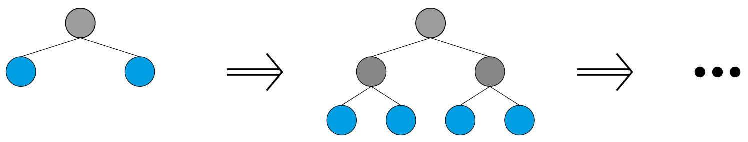
2.6. Comparison of different boosting implementations

There exist several different boosting algorithms for machine learning purposes. Starting historically, one of the first was the prominent *adaBoost* by Freund and Robert E Schapire (1997), Its principle is to change the sample distribution in each iteration by differently weighting the observations of the training dataset, i.e. it increases the weight of the "difficult" observation which were wrongly predicted in the previous iteration. Hence, the base learners concentrate more on those observations in contrast to the gradient boosting method, which trains the weak base learners on the remaining errors, i.e. the pseudo-residuals. However, also this technique transfers the focus to the difficult observations.

Another boosting algorithm is provided by the **mboost** package by Hofner et al. (2014). Its approach is a model-based boosting algorithm with support for (generalized) linear models (`glmboost()`), (generalized) additive models (`gamboost()`) or trees (`blackboost()`) as base learners. Unfortunately, it is very slow for large datasets. Other popular implementations of gradient boosting algorithms, which use trees as weak learners, are e.g. **GBM** by Ridgeway (2005), the **H2O-GBM** (see Kraljevic and team (2017)) , **lightGBM** by Ke et al. (2017), **CatBoost** (see Dorogush, Gulin, et al. (2017) and Dorogush, Ershov, et al. (2017)) and of course XGBoost, the base for autoxgboost. lightGBM and CatBoost are quite new and seem to deliver state-of-the-art performances. Moreover, lightGBM is designed to make tree gradient booster faster since it not checking all splits when creating new leaves but only look at some of them by sorting and binning them first. This idea by Ke et al. (2017) is called *histogram implementation*. Thus, the resulting trees grow leaf wise instead of level wise compared to other gradient boosting machines. The concept is shown in Figure 6. It shows that the presorted states are kept. For tree growing,



(a) Leaf wise tree growth.



(b) Level wise tree growth.

Figure 6: Differences of leaf wise and level wise tree construction.

the algorithm selects the leaf which provides the maximum delta loss. To conclude, lightGBM offers faster learning than other methods matched with state-of-the-art performance.
In contrast, speed is not the main argument for CatBoost. Instead, due to an intelligent factor encoding, it offers support for categorical features making special treatment obsolete before training. As a result, CatBoost frequently outperforms other gradient boosting algorithms like XGBoost and lightGBM regarding predictive performance. Moreover, the developers recently announced that they succeeded in improving the algorithm's speed significantly.
Comparing different gradient boosting implementations, especially the relatively new lightGBM and CatBoost algorithms look promising. However, implementations for both methods seem to be tricky including a non straightforward installation to make both algorithms working. As a result, the effort to switch to another gradient boosting algorithm might be too complex. Further research should be made to check the feasibility of a system switch and to compare speed and predictive performance leaving the door open for further improvements of our automated gradient boosting machine.

3. The concept of automatic gradient boosting

This section introduces the structure of the **autoxgboost** package which is based on the open source programming language R and available via Github (`https://github.com/ja-thomas/autoxgboost`). It contains several different parts which will be discussed in detail in the following parts of this master's thesis. As its name implies, autoxgboost uses the XGBoost package as implementation for gradient boosting. Hence, after the general approach of gradient boosting was introduced in the previous section, this one gives a closer look on those parts of the machine learning work-flow which are included and therefore automated by the package.

3.1. Factor feature encoding

After stating the main questions for an analysis, everything starts with data and preprocessing it. The latter includes all necessary steps to meet the requirements of the chosen learning algorithm. Within autoxgboost, this means to transform factor features as a first step, since XGBoost requires numeric input variables. Several different methods of doing this task exist. Hence, after starting with the first one, which is a simple transformation of the categories into integers, other techniques are introduces in the following parts of this section. This first mentioned *integer encoding* or *feature hashing* is described in the next one. Almost the same amount of complexity is reached by creating *dummy features* for all factor levels. This technique is described afterward, following by two unlike more complex variations of *impact encoding*.
Finding the best performing encoding method for autoxgboost required a benchmark experiment which compared all methods of Section 3.1. Its setting and results are shown in Section 5.1

3.1.1. Integer encoding

The simplest possibility of changing categorical variables to numeric ones is done by changing each factor level to an integer. The idea is straightforward and shown in Table 1, where a feature $X = \{x_1, \dots, x_n\}$ contains three factor levels a, b and c. The new feature X^* simply contains values 1 for a, 2 for b and 3 for c. This method, what we call *integer encoding*, is the simplest form of *feature hashing* for a single variable. In general, for feature hashing the original category x_i is transformed by a

$$\left.\begin{array}{c} X \\ \hline a \\ b \\ a \\ c \\ b \\ c \\ a \\ c \\ b \end{array}\right\} \quad \Longrightarrow \quad \left\{\begin{array}{c} X^* \\ \hline 1 \\ 2 \\ 1 \\ 3 \\ 2 \\ 3 \\ 1 \\ 3 \\ 2 \end{array}\right.$$

Table 1: Scheme of integer encoding

so-called *hash function* h in the way that

$$\begin{aligned} h: \quad X &\to \mathbb{R} \\ x_i &\mapsto h(x_i) \end{aligned}$$

In our case, we have $h(x_i) \in \mathbb{N}$. In general, $h(x_i)$, which is called *hash value* or simply *hash* could also be a vector resulting in more than a single new feature. However, this method has the advantage that not only one but several categorical features could be recoded together using a single hash function.

3.1.2. Dummy encoding

The concept of dummy encoding is to create one separate feature for each factor level. It is also sometimes called *one-hot encoding*. It is shown in Table 2, where a feature X is dummy encoded into features X_a^*, X_b^* and X_c^*. Hence, for each factor level of X a new feature is created, i.e. we get new variables X_a^*, X_b^* and X_c^* with $x_{j,i}^* = \mathbb{I}_{x_i=j}$ for observation i where $j \in \{a, b, c\}$ and $X_j^* = \{x_{j,1}^*, \dots, x_{j,n}^*\}$. Obviously, since all new features are linear dependent, one arbitrary dummy variable could be left out. So in general, for a factor feature X with n factor levels, one would end up with $n-1$ dummy encoded features when applying this "$1-n$"-method. The left-out one would then be regarded as "reference", since it equals 1, if all other dummy features are 0.

$$\left.\begin{array}{c} X \\ \hline a \\ b \\ a \\ c \\ b \\ c \\ a \\ c \\ b \end{array}\right\} \quad \Longrightarrow \quad \left\{\begin{array}{ccc} X^*_a & X^*_b & X^*_c \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right.$$

Table 2: Scheme of creating dummy features

3.1.3. Impact encoding

Another method of transforming factor features is called *impact encoding*. The name implies, that recoding depends on the impact each factor level has on the target variable of the machine learning task. This means that in contrast to dummy encoding, impact encoding can only be performed on a concrete training dataset. The first approach benchmarked in Section 5.1 was introduced by Micci-Barreca (2001).

3.1.3.1. Binary classification

For a binary target variable, which implies a binary classification context, i.e. $Y = \{y_1, \dots, y_n\}$, where $y_i \in \{0, 1\}$ for $i = 1, \dots, n$, it encodes each factor level of a feature X into a combination of the conditional probability for the positive class 1, given a factor level of X and the prior probability of class 1. As a result, we get a new impact feature $X^* = \{x_1^*, \dots, x_n^*\}$ whose entries x_i^* for $i = 1, \dots, n$ are defined as

$$\text{Impact}(x_i) = \lambda(n_{x_i}) \cdot P(Y = 1|X = x_i) + (1 - \lambda(n_{x_i})) \cdot P(Y = 1), \tag{45}$$

here n_{x_i} is the absolute number of observations for which feature X equals the factor level of x_i. For factor levels $l_1, \dots, l_m$ of X and $x_i = l_j$, $j \in \{1, \dots, m\}$, we get $n_{x_i} = n_{l_j}$. n_{l_j} is then called *cell size* of cell l_j, which contains all observations for which

X	Y
a	1
b	1
a	1
c	0
d	1
c	0
a	1
d	1
b	0

$$\Longrightarrow$$

X^*	Y
$\lambda(n_a) \cdot P(Y=1\|X=a) + (1-\lambda(n_a)) \cdot P(Y=1)$	1
$\lambda(n_b) \cdot P(Y=1\|X=b) + (1-\lambda(n_b)) \cdot P(Y=1)$	1
$\lambda(n_a) \cdot P(Y=1\|X=a) + (1-\lambda(n_a)) \cdot P(Y=1)$	1
$\lambda(n_c) \cdot P(Y=1\|X=c) + (1-\lambda(n_c)) \cdot P(Y=1)$	0
$\lambda(n_d) \cdot P(Y=1\|X=d) + (1-\lambda(n_d)) \cdot P(Y=1)$	1
$\lambda(n_c) \cdot P(Y=1\|X=c) + (1-\lambda(n_c)) \cdot P(Y=1)$	0
$\lambda(n_a) \cdot P(Y=1\|X=a) + (1-\lambda(n_a)) \cdot P(Y=1)$	1
$\lambda(n_d) \cdot P(Y=1\|X=d) + (1-\lambda(n_d)) \cdot P(Y=1)$	1
$\lambda(n_b) \cdot P(Y=1\|X=b) + (1-\lambda(n_b)) \cdot P(Y=1)$	0

Table 3: Scheme of creating impact features (binary class)

X has factor level l_j. Thus we can write Eq. (45) also for each x_i of cell l_j as

$$\text{Impact}(l_j) = \lambda(n_{l_j}) \cdot P(Y = 1|X = l_j) + (1 - \lambda(n_{l_j})) \cdot P(Y = 1). \tag{46}$$

Here, $\lambda(n_{l_j})$ controls the transition rate between the conditional probability of the positive class depending on the observations factor level l_j, i.e. $P(Y = 1|X = l_j)$, and the prior probability of class 1 of the target variable in the training dataset, $P(Y = 1)$. It is defined as an s-shaped function

$$\lambda(n_{l_j}) = \frac{1}{1 + \exp\left(-\frac{(n_{l_j} - p_{\text{trust}})}{p_{\text{slope}}}\right)} \tag{47}$$

and depends of besides cell size n_{l_j} of the trust parameter p_{trust} and the slope parameter p_{slope}. The trust parameter defines half of the minimal cell size, for which the impact value is almost completely shifted towards the conditional probability, if p_{slope} is sufficiently small at the same time. On the other hand, for $p_{\text{slope}} \to \infty$, $\lambda(n_{l_j})$ converges against 0.5. Hence, we get a fixed threshold in that case. Note that for $n_{l_j} = p_{\text{trust}}$, it also applies that $\lambda(n_{l_j}) = 0.5$. Table 3 shows an example, where impact encoding is performed on a training set with 9 observations. For a feature X with factor levels $l_j \in \{a, b, c, d\}$ and target variable Y with class levels $\{0, 1\}$, we get a new non-factor feature X^*, which consists of numeric values in $[0, 1]$ for each former

X	Y
a	1
b	1
a	1
c	0
d	1
c	0
a	1
d	0
b	0

$\Longrightarrow$

X^*	Y
$0.88 \cdot 1 + (1-0.88) \cdot \frac{1}{3}$	1
$0.73 \cdot 0.5 + (1-0.73) \cdot \frac{1}{3}$	1
$0.88 \cdot 1 + (1-0.88) \cdot \frac{1}{3}$	1
$0.73 \cdot 0 + (1-0.73) \cdot \frac{1}{3}$	0
$0.73 \cdot 1 + (1-0.73) \cdot \frac{1}{3}$	1
$0.73 \cdot 0 + (1-0.73) \cdot \frac{1}{3}$	0
$0.88 \cdot 1 + (1-0.88) \cdot \frac{1}{3}$	1
$0.5 \cdot 1 + (1-0.5) \cdot \frac{1}{3}$	0
$0.73 \cdot 0.5 + (1-0.73) \cdot \frac{1}{3}$	0

$\Longrightarrow$

X^*	Y
0.92	1
0.46	1
0.92	1
0.09	0
0.82	1
0.09	0
0.92	1
0.82	0
0.46	0

Table 4: Example values of creating impact features from Table 3

factor level of X. With $p_{\text{slope}} = p_{\text{trust}} = 1$, we obtain values

$$\begin{aligned}
&P(Y=1|X=a) = 1, && \lambda(n_a) = 0.88\\
&P(Y=1|X=b) = 0.5, && \lambda(n_b) = 0.73\\
&P(Y=1|X=c) = 0, && \lambda(n_c) = 0.73\\
&P(Y=1|X=d) = 1, && \lambda(n_d) = 0.73 \text{ and}\\
&P(Y=1) = \tfrac{1}{3}.
\end{aligned}$$

Hence, we get the final impact values of Table 4.

3.1.3.2. Multiclass classification

For multiclass classification tasks, a single new feature X^* for replacing the original variable X is not sufficient. Nevertheless, the multiclass extension for impact encoding is straightforward. Here, one new non-factor feature is created for each class of

X	Y
a	B
b	A
a	C
c	A
d	C
c	A
a	B
d	C
b	B

$\Longrightarrow$

X_A^*	X_B^*	X_C^*	Y
...	$\lambda(n_a) \cdot P(Y=B\|X=a) + (1-\lambda(n_a)) \cdot P(Y=B)$	...	B
...	$\lambda(n_b) \cdot P(Y=B\|X=b) + (1-\lambda(n_b)) \cdot P(Y=B)$	...	A
...	$\lambda(n_a) \cdot P(Y=B\|X=a) + (1-\lambda(n_a)) \cdot P(Y=B)$	...	C
...	$\lambda(n_c) \cdot P(Y=B\|X=c) + (1-\lambda(n_c)) \cdot P(Y=B)$	...	A
...	$\lambda(n_d) \cdot P(Y=B\|X=d) + (1-\lambda(n_d)) \cdot P(Y=B)$	...	C
...	$\lambda(n_c) \cdot P(Y=B\|X=c) + (1-\lambda(n_c)) \cdot P(Y=B)$	...	A
...	$\lambda(n_a) \cdot P(Y=B\|X=a) + (1-\lambda(n_a)) \cdot P(Y=B)$	...	B
...	$\lambda(n_d) \cdot P(Y=B\|X=d) + (1-\lambda(n_d)) \cdot P(Y=B)$	...	C
...	$\lambda(n_b) \cdot P(Y=B\|X=b) + (1-\lambda(n_b)) \cdot P(Y=B)$	...	B

Table 5: Scheme of creating impact features (multiclass)

the target variable. Each of them is determined by a ratio between the conditional probability of the related class depending on the factor level of the original feature X and the prior probability of that class. Again, this threshold is defined by function $\lambda(\cdot)$, which is equally calculated as in the binary setting. Hence, for any class c, we can calculate a new numeric value for a factor level l_j by

$$\text{Impact}(l_j) = \lambda(n_{l_j}) \cdot P(Y = c | X = l_j) + (1 - \lambda(n_{l_j})) \cdot P(Y = c). \qquad (48)$$

This scheme is shown in Table 5. It illustrates that for class levels $\{A, B, C\}$, a factor feature X with levels $\{a, b, c, d\}$ is replaced by the new numeric features X^*_A, X^*_B, and X^*_C. For X^*_B, the formulas for each observation are formulated in Table 5. Moreover, for X^*_A and X^*_C, only class B needs to be changed within the probabilities into the corresponding class A or C as seen in Eq. (48).
Like in dummy encoding, the resulting numeric features are linear dependent. Hence, one of them can be left out in the impact encoded dataset.

3.1.3.3. Regression

Impact encoding is not limited to classification, but also possible for regression tasks. Since there exist no probabilities in the regression case, a combination of the target's mean $\overline{Y}$ and the target's mean within each cell, i.e. $\overline{Y_{l_j}}$ for a factor level l_j for feature X is used instead to determine the transformed values. Remember, a cell for a factor level l_j was defined as the set of all observations, whose factor level of feature X is equal to l_j. As a results, we get a single numeric-valued transformed variable whose new impact values are calculated by

$$\text{Impact}(l_j) = \lambda(n_{l_j}) \cdot \overline{Y_{l_j}} + (1 - \lambda(n_{l_j})) \cdot \overline{Y}. \qquad (49)$$

Table 6 shows the resulting formulas similar to Table 3 and Table 5 for a regression example. Obviously, the necessary values for this example are

$$\begin{aligned}
&\overline{Y_a} = 11, \quad \lambda(n_a) = 0.88 \\
&\overline{Y_b} = 20, \quad \lambda(n_b) = 0.73 \\
&\overline{Y_c} = 31, \quad \lambda(n_c) = 0.73 \\
&\overline{Y_d} = 40, \quad \lambda(n_d) = 0.73 \text{ and} \\
&\overline{Y} = \tfrac{1}{3}.
\end{aligned}$$

X	Y
a	10
b	21
a	12
c	30
d	39
c	32
a	11
d	41
b	19

$$\Longrightarrow$$

X^*	Y
$\lambda(n_a) \cdot \overline{Y_a} + (1 - \lambda(n_a)) \cdot \overline{Y}$	10
$\lambda(n_b) \cdot \overline{Y_b} + (1 - \lambda(n_b)) \cdot \overline{Y}$	21
$\lambda(n_a) \cdot \overline{Y_a} + (1 - \lambda(n_a)) \cdot \overline{Y}$	12
$\lambda(n_c) \cdot \overline{Y_c} + (1 - \lambda(n_c)) \cdot \overline{Y}$	30
$\lambda(n_d) \cdot \overline{Y_d} + (1 - \lambda(n_d)) \cdot \overline{Y}$	39
$\lambda(n_c) \cdot \overline{Y_c} + (1 - \lambda(n_c)) \cdot \overline{Y}$	32
$\lambda(n_a) \cdot \overline{Y_a} + (1 - \lambda(n_a)) \cdot \overline{Y}$	11
$\lambda(n_d) \cdot \overline{Y_d} + (1 - \lambda(n_d)) \cdot \overline{Y}$	41
$\lambda(n_b) \cdot \overline{Y_b} + (1 - \lambda(n_b)) \cdot \overline{Y}$	19

Table 6: Scheme of creating impact features (regression)

Note, since the factor feature X is the same as in Table 3 and Table 5, the resulting values for λ are the same.

3.1.3.4. Impact encoding of vtreat

The second method of impact encoding, which was introduced by Mount and Zumel (2016), is an element of the **vtreat** package for R. However, while its application usually consists of several treatments for a single categorical variable, here, only a method very similar to the one of Section 3.1.3 was used. Both implementations only differ on the calculation of the ratio between the conditional and prior probabilities. Taking Eq. (46) and Eq. (49) for **vtreat**, both equations reduce to

$$\text{Impact}(x_{l_j}) = \text{logit}(P(Y = c | X = l_j)) - \text{logit}(P(Y = c)) \tag{50}$$

and

$$\text{Impact}(x_{l_j}) = \overline{Y_{l_j}} - \overline{Y}. \tag{51}$$

Obviously, this impact encoding variation forgoes the transition function λ.

3.2. Hyperparameter tuning with mlrMBO

Each machine learning model works best, when its hyperparameters are set to the optimal values. Since trying each possible parameter combination is not an option, since this approach is computational far too expensive, one needs an intelligent way of searching the learner's *hyperparameter space* for the best configuration. This

procedure is called *hyperparameter tuning* and can be understood as an optimization problem, where the models' performance is to be maximized given its hyperparameters. The hyperparameter space contains the *feasible sets* of all parameters, where each feasible set contains all possible values of a parameter.
As starting point for all tuning approaches, one needs to choose the hyperparameters, which should be tuned, and their value ranges. If the complete hyperparameter space is too large or resources for tuning are limited, the resulting *tuning parameter set* can also be a subset of it. Then, the optimal parameter setting can be found by tuning the model via various techniques.
One of the simplest methods is *grid search*, where the performance is measured and compared for each point of a predefined grid on the *tuning set*. Another method is the *random search* which randomly selects hyperparameter configurations for measuring and comparing the performance. All methods have in common that the best performing configuration during tuning (depending on the chosen performance measure) will be chosen for the final model.
More advanced tuning methods include a certain logic for choosing a parameter configuration within the parameter space. A popular tuning algorithm is **irace** by López-Ibáñez et al. (2016) which is based on the racing algorithm by Maron and Moore (1997). However, this method will not be discussed in detail in this master's thesis.
Another tuning method which uses *Sequential Model-Based Optimization (SMBO)*, also known as *Bayesian Optimization* is included in autoxgboost and called **mlrMBO** by Bischl, Richter, et al. (2017). This concept uses function approximations to construct an algorithm which efficiently solves our optimization problem.
There exist several variations of SMBO since its framework is modular. One Popular variation is the *Efficient Global Optimization algorithm (EGO)* by Jones et al. (1998) which was the first approach using *Gaussian processes (GP)* as so-called *surrogate models* to optimize box-constrained functions on a numeric-only input domain $\mathcal{X} \in \mathbb{R}^d$. Those surrogate models will be discussed in more detail in the next part of this section.
However, the underlying stochastic process approach for function approximations, which is called *Kriging*, was already introduced in the early 1960s and was ever since part of mathematical geology literature, optimization theory and statistics (see Matheron (1963), N. A. C. Cressie (2015) and N. Cressie (1990)).
Another common SMBO combination was proposed by Hutter et al. (2011) and is

called *SMAC* which stand for *Sequential Model-based Algorithm Configuration*. It extends the SMBO approach using random forests as surrogate model to be capable of also handling categorical parameters.

3.2.1. SMBO algorithm

No matter which variation of SMBO is used for tuning, each contains the same setup. Following Bischl, Richter, et al. (2017), we assume an arbitrary black-box function $f(x) : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X} \subset \mathbb{R}^d$ is an d-dimensional input space and $y \in \mathbb{R}$ denotes the deterministic function's output. This function is generally assumed to be expensive to evaluate. Hence, a predefined budget determines the maximum number of function evaluations.
In detail, each element $\mathcal{X}_i$ of $\mathcal{X}$, where $i = 1, \dots, d$, can either be a numeric interval, i.e. $\mathcal{X}_i \in \{[a, b] | a \leq b\, a, b \in \mathbb{R}\}$, or contains categorical factor levels, i.e. $\mathcal{X}_i \in \{\{l_{i,1}, \dots, l_{i,k}\} | k \in \mathbb{N}\}$. The goal is to find the input vector x^*, such that

$$x^* = \arg\min_{x \in \mathcal{X}} f(x).$$

As mentioned above, each function evaluation of f is usually expensive. Therefore, the SMBO framework introduces surrogate models $\hat{f}$ which approximate the expensive black-box function f while being cheaply evaluable. Those $\hat{f}$ are iteratively updated during the whole process which is illustrated in Figure 7 following Bischl, Richter, et al. (ibid.). It contains the following steps:

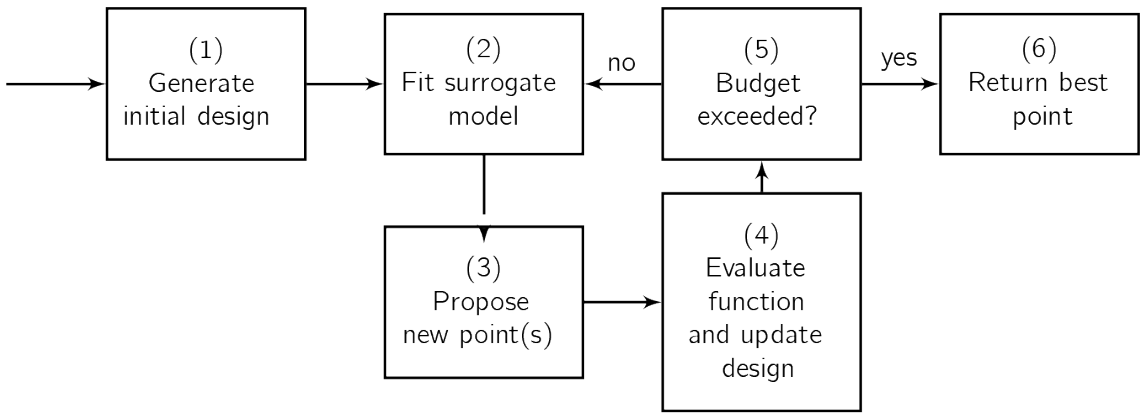


Figure 7: General SMBO Framework

(1) It all starts with generating an initial design of n_{init} points $(x^{(1)}, \dots, x^{(n_{\text{init}})}) \in \mathcal{X}$. It is important to choose a sufficiently large number ob initial points which should cover $\mathcal{X}$ good enough to ensure an adequate fit of $\hat{f}$. At these points, f get evaluated with $f(x^{(j)}) = y^{(j)}$, where $j = 1, \dots, n_{\text{init}}$. Then, the resulting pairs $(x^{(j)}, y^{(j)})$ serve as base of the initial surrogate model $\hat{f}$. As mentioned before, Kriging is a common and recommended choice for a surrogate model which delivers a state-of-the-art performance for numeric parameter spaces while random forests allow also categorical ones. Since XGBoost only contains numerical hyperparameters, autoxgboost relies on **mlrMBO**'s default, the Gaussian processes of Kriging. However, the package allows to use any regression learner as surrogate model which is included in the **mlr** package.

(2) As a next step, the surrogate model is fitted based on all pairs $(x^{(j)}, y^{(j)})$.

(3) The proposition of new m points $x^{(j+i)}$, where $i = 1, \dots, m$, is based on an *infill criterion* which is also known as *acquisition function*. Its purpose is to lead the optimization process towards the most promising points, which e.g. could offer the highest *expected improvement*, which is defined as

$$\text{EI}(x) := \text{E}(I(x)) = \text{E}(\max\{y_{\min} - Y(x), 0\}).$$

Here, $I(x)$ is a random variable, which defines the potential improvement at x for $y_{\min}$, the currently best observed function value. Furthermore, $Y(x)$ is a random variable, which is normally distributed within the Gaussian process context and expresses the posterior distribution of x with posterior mean $\hat{\mu}(x)$ and posterior standard deviation $\hat{s}^2(x)$. Thus, with $Y(x) \sim N(\hat{\mu}(x), \hat{s}^2(x))$, $\text{EI}(x)$ can be written as

$$\text{EI}(x) = (y_{\min} - \hat{\mu}(x))\Phi\left(\frac{y_{\min} - \hat{\mu}(x)}{\hat{s}(x)}\right) + \hat{s}(x)\phi\left(\frac{y_{\min} - \hat{\mu}(x)}{\hat{s}(x)}\right), \tag{52}$$

where ϕ is the density function and Φ the distribution function of the standard normal distribution.

(4) The proposed new points in (3) are evaluated by f in the next step resulting

in new tuples $(x^{(j+i)}, y^{(j+i)})$.

(5) Steps (2)-(4) are repeated until the budget is exhausted or other stop criteria take effect. Those might be limits set by the user, e.g. the maximum number of SMBO iterations, a certain time limit for the optimization or if a prespecified value for f is achieved. If at least on criterion is met, the tuning algorithm continues with step (6).

(6) Finally, the best point observed during optimization is usually returned as the final solution x^*. However, if the black-box function is noisy, an additional surrogate model fit to determine the optimal point is an option.

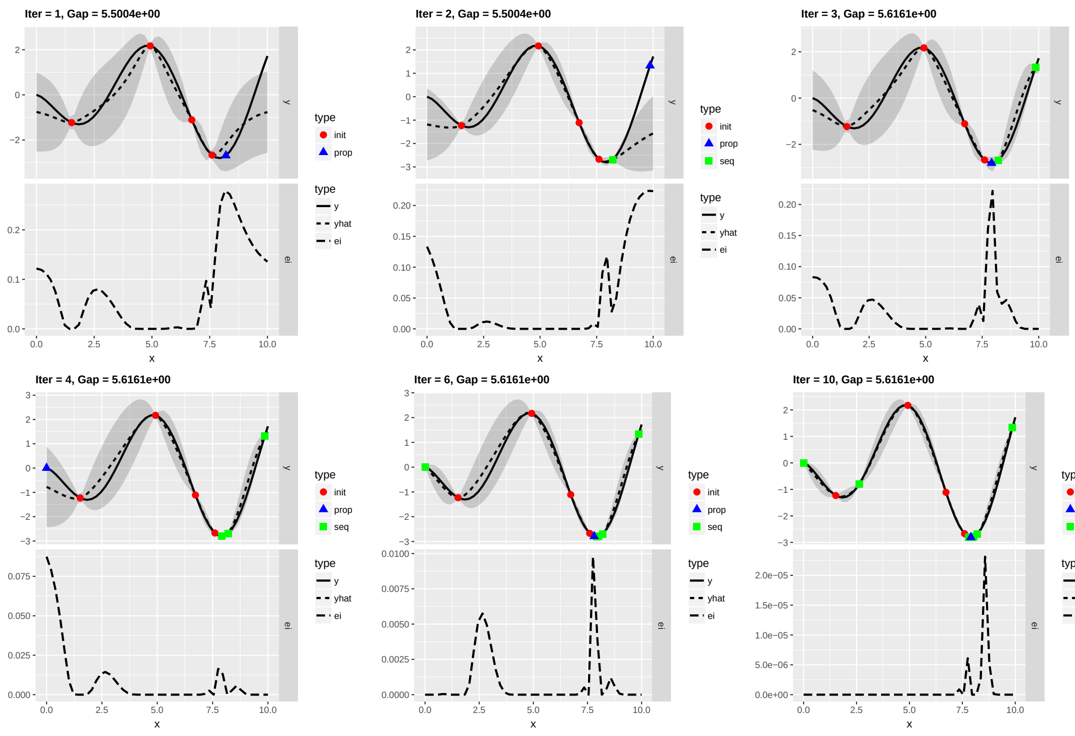


Figure 8: State at several iterations of an example mlrMBO run. Each plot consists of two parts, where the upper one shows the real function f (1-dimensional Alpine 2 function) as a solid line and its estimation $\hat{\mu}$ as a dotted line. The shaded area shows the uncertainty, the red circles are the initial design points while the sequential points are shown as green squares. The lower parts show the respective values for the expected improvements whose optimum is marked as a blue triangle and defines the next proposed point.

Six iterations of the SMBO optimization process for the *Alpine 2 Function* are shown in Figure 8. The 1-dimensional version is defined as

$$f(x) = \sqrt{x} \cdot \sin(x).$$

Each iteration, contains two plots. The upper one shows the original function f as solid line and its estimated function $\hat{\mu}$ as dotted line. Moreover, the upper plot illustrates the initial points as red circles, new proposed points as blue triangles and sequential points, i.e. previously proposed and hence evaluated points as green squares. Obviously, the first iteration does not have any sequential point yet and thus, it provides the first proposed point. This point is chosen by the largest expected improvement (EI) whose dashed graph can be seen in the lower plot of each iteration. Clearly, this graph is determined by the uncertainty level of the approximation function ($\hat{s}$) which is shown as the shaded area in each upper plot in combination with and the value of the approximation function $\hat{\mu}$. Higher uncertainty and a lower value for $\hat{\mu}$ increases the expected improvement as seen in Eq. (52). Figure 8 convincingly demonstrated the capability of SMBO, which is of course not limited to univariate objective functions. Therefore, it is a very suitable tuning method for autoxgboost.

3.2.2. XGBoost learner with early stopping

In Section 3.2 we saw that **mlrMBO** is used for hyperparameter tuning of autoxgboost. For each tuning iteration, an XGBoost learner is trained on the data. In contrast to the default settings of the algorithm, here, early stopping is enabled. This means that the training process in a single tuning iteration stops, if the performance does not improve for a certain number of boosting iterations on a specific validation set. This specific number is called `early stopping rounds` and set to 10 in autoxgboost. The validation set is determined by the `early stopping fraction` which is set to 4/5, i.e. this fraction is used for training while 1/5 of the data is used to measure the performance in each boosting iteration.
Using early stopping as stop criteria for training, one needs to ensure that enough iterations are performed in first place. Hence, the parameter `nrounds` is set to 10^6.

3.2.3. Autoxgboost parameter set

Tuning always requires parameters with a corresponding parameter space to search for the optimal values. The above mentioned early stopping rounds help to determine the `nrounds` argument of autoxgboost. The idea is to take the number rounds of the best performing model as value for the `nrounds` parameter. The other learner parameters are directly tuned and shown in Table 7 including the lower and upper

Autoxgboost parameterset				
Name	Lower	Upper	Trafo	Description
`eta`	0.01	0.2		controls the learning rate: scale the contribution of each tree by a factor of $0 <$ `eta` < 1 when it is added to the current approximation. Used to prevent overfitting by making the boosting process more conservative. Lower value for eta implies larger value for `nrounds`: low `eta` value means model more robust to overfitting but slower to compute. Default: 0.3
`gamma`	-7	6	2^x	minimum loss reduction required to make a further partition on a leaf node of the tree. the larger, the more conservative the algorithm will be.
`max_depth`	3	20		maximum depth of a tree. Default: 6
`colsample_bytree`	0.5	1		subsample ratio of columns when constructing each tree. Default: 1
`colsample_bylevel`	0.5	1		subsample ratio of columns for each split, in each level. Default: 1
`lambda`	-10	10	2^x	L2 regularization term on weights. Default: 0
`alpha`	-10	10	2^x	L1 regularization term on weights. (there is no L1 reg on bias because it is not important). Default: 0
`subsample`	0.5	1		subsample ratio of the training instance. Setting it to 0.5 means that xgboost randomly collected half of the data instances to grow trees and this will prevent overfitting. It makes computation shorter (because less data to analyze). It is advised to use this parameter with `eta` and increase `nround`. Default: 1

Table 7: XGBoost hyperparameters tuned by mlrMBO (Description taken from package manual).

bounds of the parameter space. Additionally, parameters `alpha`, `gamma` and `lambda` have a transformation function $f(x) = 2^x$. This ensures positive values with higher concentration on small values.

3.3. Threshold tuning for classification tasks

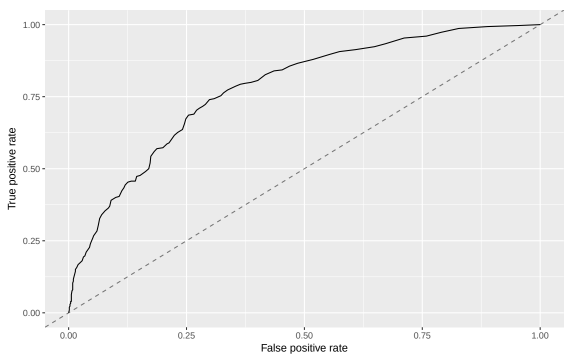
Spiegelhalter (1986) stated out, how important it is to consider prediction probabilities for classification tasks, which implies that looking at uncertain observations is key for good results. This is obviously not possible for regression tasks, since one does not get any probabilities from the model for numeric response values. Hence, when concentrating on classification problems, these decisions require reliable probabilities, which are not always guaranteed. Already J. Friedman et al. (2000) stated out that boosting is inclined to make poorly calibrated predictions. However, there exist two ways of dealing with this problem. The first one tries to "calibrate" the probabilities given by a model. This procedure is called *classifier calibration* and offers several solutions to overcome these deficient probabilities, e.g. *Logistic Correction*, by J. Friedman et al. (ibid.), *Platt Scaling* by Platt (1999) or *Isotonic Regression* (see Zadrozny and Elkan (2001) and Zadrozny and Elkan (2002)).
The other possibility to deal with unreliable probabilities is not to change the probabilities directly, but to adjust the resulting classification rules. This can be done by optimizing the classification probability threshold. Compared with hyperparameter tuning, for which different parameter settings are trained and compared afterward, the threshold tuning is done for predictions of an already trained model.
One could say that after receiving class probabilities by the machine learning model, one leaves the probabilistic part and switches to decision theory in order to apply non-probabilistic treatment for especially difficult observations, where the uncertainty level for one class or another is high.
For example, consider a binary classification model whose output provides probabilities for the positive class, let us say, class 1. Obviously, one classifies observations with probabilities close to 1 to this positive class 1 while observations with probabilities close to 0 are assigned to the other class, called e.g. class 0. Moreover, there is an area of high uncertainty around the probability of 0.5. While one intuitively would set a threshold at exactly 0.5, however, this might not be optimal for many datasets, since there exist situations where one reasonably doubts about the model predictions. As presented by Silver (2012), one key element is the signal - noise ratio within the data, which might be problematic for too small datasets and in other cases. One

possible case is highly unbalanced data, where machine learning models might fail or give non-trustworthy results. Consider data from a medical trial with 100 patients with a disease and 100.000 patients without that disease. A model which assigns all observations to the non-disease class has an accuracy of $99,9\%$. However, all diseased patients are classified incorrectly but might got a probability for having the disease of around 45% by the model. Hence, setting a threshold below this level might improve the overall performance and additionally, this would also corresponds to a decision boundary for a doctor who would run some additional tests for patients which have a probability significantly above zero.
While also measurement errors during the data collection process could push close observations towards one side of the threshold boundary, also cost-sensitive classification is one reason for tuning the threshold which labels the observations such that a certain cost function is minimized. A popular example for this task is a bank which wants to classify potential credit customers in two groups. One group, for which a credit can be granted, and another one, for which it should be denied. Obviously, the bank wants to avert giving credit to customers, which are not able to repay it. Hence, this error comes at a higher cost then refusing a credit to a financially strong customer. As a consequence, shifting uncertain observations towards the non-credit group allows to potentially minimize the costs.
While the examples above are of only binary nature, the concept can simply be extended to multiclass problems resulting in a threshold vector whose length equals the number of classes of the target variable.
But first, one should consider the general possibilities of implementing threshold tuning. One way, and that is the way how it is implemented in the **mlr** package and hence in autoxgboost, is to tune the threshold directly on given predictions without any further resampling strategy for evaluation. Here, the predictions depend on their origin. Technically, it would be possible to use predictions of the training dataset to tune the threshold, which is obviously not a good choice since it leads to overfitting. A better way of making predictions is based on a resampling strategy which is useful for model evaluation at the same time. Hence, if the model was trained by cross-validation or a fixed holdout split, all predictions, which are made on the specific test sets are used for threshold tuning. Then, the whole problem basically reduces to a black-box optimization problem of the performance function given those predictions. On the other hand, one could use an additional, autonomous resampling loop to find an even more adequate threshold based on another test set, which would however

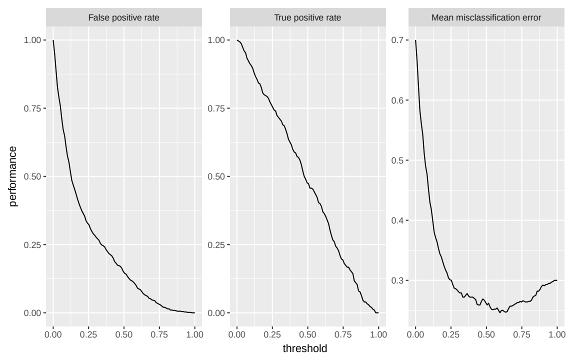
be the slower way of doing it. Obviously, there is still room for improvements and further research on this topic.
The next part of this section gives a more detailed example of the binary classification case and moreover introduces the technical details of the implemented threshold tuning within autoxgboost. Thereafter, a part showing the background for the multiclass case follows.

3.3.1. Intelligent grid search for binary classification

The binary case of threshold tuning is similar to a ROC-curve analysis. This ROC-curve illustrates the ratio between the *true positive rate (tpr)* and the *false positive rate (fpr)* of a classifier. Thus, the optimal threshold is the point, where the ratio between both measures is optimal regarding a cost function or the achieved prediction accuracy. A ROC-curve for a logistic regression model for the credit example from Section 3.3 is shown in Figure 9a, while Figure 9b shows the coresponding values for ROC measures, the true positive rate, and the false positive rate, as well as the *mean misclassification error (mmce)* depending on the threshold. The ROC-curve from Figure 9a looks quiet symmetric which indicates that the optimal threshold is close to 0.5. This impression is supported by Figure 9b, but additionally, if shows clearly that the minimum value for the mmce is at around 0.6. Hence, tuning the threshold could give additional prediction performance.
When looking at a cost matrix like in Table 8 for the same dataset, one can calculate



(a) ROC curve



(b) Values for ROC-measures and mmce

Figure 9: ROC curve and corresponding measured values of a logistic regression classifier for german credit data depending on the threshold.

	no credit	credit
fin. weak	0	10
fin. strong	1	0

Table 8: Cost matrix for credit data.

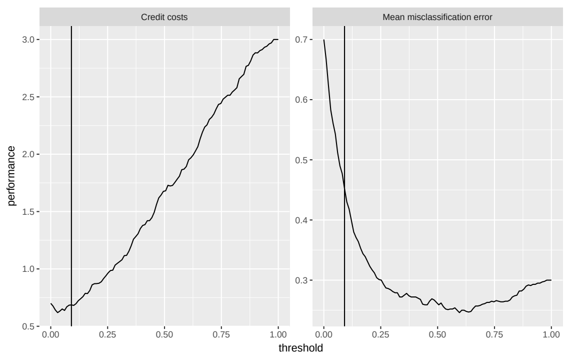
	Performance	
Threshold	credit costs	mmce
0.5 (default)	1.738	0.253
0.091 (theoretical)	0.631	0.442
0.0764 (tuned)	0.572	0.482

Table 9: Performances for different thresholds

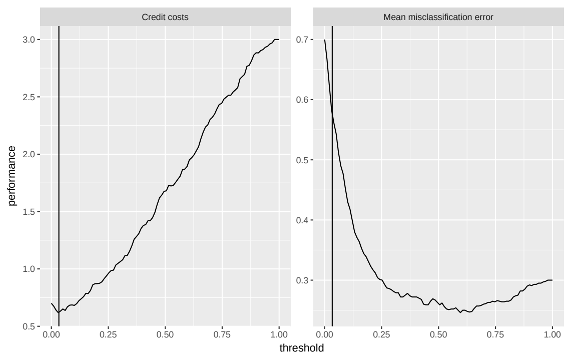
a theoretical threshold for the positive class 1 by

$$t^* = \frac{c(1,0) - c(0,0)}{c(1,0) - c(1,1) + c(0,1) - c(0,0)} = 0.091 \tag{53}$$

Table 9 compares for different thresholds the resulting performances when training a logistic regression learner applying 3-fold cross-validation on the data. While the default threshold at 0.5 has the best value for the mmce, it has clearly the weakest value for the credit costs compared to the theoretical and the tuned threshold values. Obviously, optimizing the threshold outperforms the default and is especially recommended for applications like this cost-sensitive classifications, which is done by using the predictions made on the test sets of each cross-validation iteration. Tuning gives extra performance compared to the theoretical value. Figure 10 shows the detailed performance for both variants and illustrates the results from Table 9.
Technically, within **mlr** the threshold is tuned by optimizing the performance on the predictions for a certain measure, depending on the threshold parameter. This is



(a) Theoretical threshold



(b) Tuned threshold

Figure 10: Performance comparison for theoretical and tuned thresholds on the credit costs and the mmce.

done by an algorithm, which could be simply described as an intelligent grid search. It is performed via the `optimizeSubInts()` function of the **BBmisc** package by Bischl, Lang, Bossek, et al. (2017). Internally, this function uses the base optimization function `optimize()` of the **stats** package, but prevent the algorithm being trapped within a local optima by dividing the parameter space into equally sized subintervals. Here in the binary case, the feasible space fo the threshold is $[0, 1] \subset \mathbb{R}$, which is by default divided into 20 smaller parts. On each of those, the `optimize()` function is then applied to optimize the performance function to obtain the optimal value for the threshold.
An R package, which is specialized for optimizing binary thresholds, which are called *cutpoints* there, is **cutpointr** by Thiele (2017). Several different methods for calculating cutpoints of which some are designed to be more robust than the simple empirical optimization of a metric are included in the package. Moreover, **cutpointr** is able to automatically bootstrap the variability of the optimal thresholds in order to return out-of-bag estimates of various performance metrics.

3.3.2. Generalized Simulated Annealing for multiclass classification

While the ROC-curve analysis is a well known topic, threshold tuning especially for multiclass problems is not a much discussed field in the literature yet and there exist still no silver bullet for doing it. For multiclass and multilabel classification tasks (the latter is not included in autoxgboost), another algorithm is used for threshold tuning by **mlr** than for the binary case, which has to be capable of optimizing multidimensional, potentially complex, non-convex and -smooth functions with a high number of local optima. To find an algorithm, which provides these properties by being simultaneously time efficient, a benchmark was done to compare different optimization functions. As one can see in Section 5.2, the *Generalized Simulated Annealing (GSA)* algorithm of the **GenSA** package by Yang Xiang et al. (2013) won this benchmark by providing the best performance and likewise being the fastest algorithm. Hence, the technical background of this optimization technique will be introduced in this section.
The concept of *Classical Simulated Annealing (CSA)* is unlike gradient descent not a deterministic but stochastic optimization method. Note, that since the objective function is a not known black-box function, there exist no derivatives, which makes e.g. gradient descent not applicable. CSA was proposed by Kirkpatrick et al. (1983) and has its origin in metallurgy. The algorithm was developed to simulate the an-

nealing process of molten metal, whose temperature is reduced until it reaches its crystalline state. At this state, the metal reaches its minimal thermodynamic energy level, which is a global minimum, when considering a function depending on the thermal energy. Therefore, in the simulated annealing algorithm, the whole process is simulated, where the metal's energy function is regarded as the objective function. During the cooling process, one or more artificial temperatures, which serve as stochastic thermal noise to escape local minima, are added and gradually annealed. At the end, the system should be inside the global optimum.
Additionally to CSA, Szu and Hartley (1987) proposed *Fast Simulated Annealing (FSA)* which cools faster. Moreover, following Tsallis and Stariolo (1996), CSA and FSA can be generalized to GSA. In general, the annealing process depends on the *visiting distribution* and the *acceptance probability*. One can understand both parts in the way that the first one proposes new points, which will be accepted (or declined) by the acceptance probability. In detail, the *visiting distribution* determines the trial jump distance $\triangle x(t)$ of $x(t)$ for an iteration t under artificial temperature $T_{q_\nu}(t)$, where q_ν controls the shape of the visiting distribution, that is, as proposed by Szu and Hartley (1987), a d-dimensional distorted Cauchy-Lorentz distribution and following Tsallis and Stariolo (1996) given by

$$g_{q_\nu}(\triangle x(t)) \propto \frac{[T_{q_\nu}(t)]^{-\frac{d}{3-q_\nu}}}{\left[1+(q_\nu-1)\frac{(\triangle x(t))^2}{[T_{q_\nu}(t)]^{\frac{2}{3-q_\nu}}}\right]^{\frac{1}{q_\nu-1}+\frac{d-1}{2}}}. \tag{54}$$

The resulting trial jump is accepted, if it is directed downwards to the minimum. Nevertheless, if it is uphill, it might still be accepted depending on the acceptance probability determined by a generalized Metropolis algorithm by Metropolis et al. (1953), given by

$$p_{q_a}(x_t \to x_{t+1}) = \min\{1, [1-(1-q_a)\beta\triangle E(x_t)]^{\frac{1}{1-q_a}}\}, \tag{55}$$

where $\beta \equiv 1/k_B T$ is a Lagrange parameter for the *Boltzmann constant* k_B and temperature T. $E(t)$ is the energy spectrum at iteration t and $\triangle E(x_t) \equiv E(x_{t+1}) - E(x_t)$. The acceptance probability is controlled by parameter q_a. When assuming that $E(x) \geq 0 \; \forall x$ and $q_a \geq 1$, $p_{q_a} \in [0,1]$ for uphill jumps, i.e. for $E(x_{t+1}) \geq E(x_t)$. Moreover, p_{q_a} is equal to 1 for downhill jumps.
From GSA, CSA and FSA can be obtained by setting the acceptance parameter

q_a to 1 in both cases and the visiting parameter $q_\nu = 1$ for CSA and $q_\nu = 2$ for FSA. The higher value for the latter makes cooling faster and hence the annealing algorithm to converge faster while increasing the chance of escaping local minima at the same time. Since GSA has a visiting parameter between 2 and 3, it has an even higher probability to find the global minimum than CSA and FSA. Moreover, higher negative values makes the algorithm more skeptical of uphill jumps and thus lead more often to their refusal.
The default values for the **GenSA** package are $q_\nu = 2.62$ and $q_a = -5$ (see Yang Xiang et al. (2013)). However, for the specific requirements of threshold tuning, an

`task`	A **mlr**-task object
`measure`	Performance measure to be optimized.
`control`	Control object for mbo. Specifies runtime behaviour. Default is to run for 160 mbo iterations or 1 hour, whatever happens first.
`par.set`	Parameter set. Default can be seen in Table 7.
`max.nrounds`	Maximum number of allowed XGBoost iterations. Default is 10^6.
`early.stopping.rounds`	After how many iterations without an improvement in the OOB error should be stopped? Default is `10`.
`early.stopping.fraction`	Should the best found model be fitted on the complete dataset? Default is `FALSE`.
`build.final.model`	What fraction of the data should be used for early stopping (i.e. as a validation set). Default is `4/5`.
`design.size`	Size of the initial design. Default is `15L`
`impact.encoding.boundary`	Defines the threshold on how factor variables are handled. Factors with more levels than the `impact.encoding.boundary` get impact encoded as introduced in Section 3.1.3 while factor variables with less or equal levels than the `impact.encoding.boundary` get dummy encoded. For `impact.encoding.boundary = 0L`, all factor variables get impact encoded while for `impact.encoding.boundary = Inf`, all of them get dummy encoded. Default is `10L`.
`mbo.learner`	Regression learner from mlr, which is used as a surrogate to model our fitness function like explained in Section 3.2
`nthread`	Number of cores to use. If `NULL` (default), xgboost will determine internally how many cores to use.
`tune.threshold`	Should thresholds be tuned as described in Section 3.3? Default is `TRUE`.

Table 10: Arguments of the `autoxgboost()` function

own benchmark was done to find the optimal default hyperparameters of the GenSA algorithm for our purpose whose results will be presented in Section 5.2.2.2.

3.4. The API of autoxgboost

The core function of autoxgboost is the homonymous function `autoxgboost()`. Its arguments are shown in Table 10. Obviously, the intention of autoxgboost implies that, except to input the data in form of a **mlr** task, no special action is required by the user for using the `autoxgboost` function. However, even though the default values were set in order to deliver state-of-the-art performances in most cases, the users are given fully flexibility of certain parameters. Starting with the measure used for optimization, the user can adapt autoxgboost to his budget restrictions for hyperparameter tuning via **mlrMBO**. This includes time- and tuning-iteration-limits. Related to this, the user can replace the default tuning parameter set from Table 7 and/or the surrogate learner by a customized one. Moreover, changes on the algorithm's early stopping mechanism as well as the factor encoding behaviour are possible. Finally, since XGBoost natively supports parallel computing, the user can set a specific number of logical cores used for training, if he does not want to use all available ones, which are automatically detected by autoxgboost.
The `autoxgboost()` function returns a `AutoxgbResult` object, which contains the fields shown in Table 11. Besides the optimal tuning result, which is also used to return the final tuned learner, a final model is returned, if the user wants to. Moreover, the measure used for tuning is returned.

`optim.result`	Optimization result object from tuning with **mlrMBO**
`final.learner`	Xgboost learner with best found hyper paramater configuration.
`final.model`	If `build.final.model=TRUE` in `autoxgboost` a **mlr** model build by the full dataset and `final.learner`.
`measure`	The measure used for optimization.

Table 11: Arguments of the `AutoxgbResult` object

4. Further Auto-ML projects

The introduction already told us that machine learning is an analyzing tool for the data scientist. Therefore, automating this step can obviously neither make any data scientist obsolete nor is it possible to automate his whole work-flow. Instead, Auto-ML projects exist to support them by considerably simplifying their workload and help for making decisions at least for the common predictive modeling tasks regression or classification. In order to do that, several open-source approaches were already proposed of which the most prominent ones are introduced in this section. When directly comparing with autoxgboost, especially methods which try to make single algorithms parameter-free (which is basically the main function of autoxgboost), are of special interest. Those “single-learner” approaches will be introduced in the next part of this section, followed by more general approaches, which include several learning algorithms into their optimization space.

4.1. Single-learner approaches

The simplest option for parameter free machine learning would be to choose a parameter free learner. While this sounds obvious, there are only few algorithms which belong to this group. Some of them have their origin in classical statistics like linear or logistic regression models. However, their performance is mostly poor compared to modern machine learning algorithms, which often have plenty of parameters in return. Hence, the better choice is to get those high-performance learners parameter-free. Within autoxgboost, this happens by automatic factor encoding and parameter tuning on a predefined parameter space.
Probst (2018) introduced the **tuneranger** package, which, as its name indicates, automatically tunes a random forest by the **ranger** package of Wright and Ziegler (2017). This tuning is done like autoxgboost by **mlrMBO**.
Furthermore, another algorithm approach is called *Parameter-free STOchastic Learning (PiSTOL)* and is proposed by Orabona and Pál (2016). It achieves model selection while training without parameter tuning or need for cross-validation. The concept is based on stochastic gradient descent and changes the step sizes data-dependent over time. Orabona and Pál (ibid.) support their concept with provable optimal convergence rates and compare their algorithm with a support vector machine in their paper. While achieving similar results, training happens up to 7 times faster due to no need for cross-validation.

Its focus on automatic deep learning for enterprises has *Driverless AI* by team (2017). Paired with a graphical user interface, it provides data exploration following by automatic feature engineering and hyperparameter tuning for best training performance. Moreover it comes with tools for model interpretation as e.g. K-LIME, variable importance and partial dependency plots. As a further surplus, it returns a Jupyter notebook containing the python code of the performed feature engineering and model building. Note, that Driverless AI is a commercially licensed product.
Directly comparable projects are rare at the moment or, like Driverless AI, they are not open source, but commercial projects. However, when reducing to methods, which simplify the hyperparameter tuning, independently of a prespecified learner, to a point, where it is parameter-free, at least another concept is worth mentioning. A *Robust Bayesian Optimization framework (RoBO)* by Klein et al. (2017) is one of those and uses Bayesian optimization for model tuning. It is python based and can be applied to several different learning algorithms like Gaussian processes, Bayesian neural networks or random forests.

4.2. Multiple-learner approaches

The concept of multiple-learner approaches is based on two important circumstances. First, no single machine learning method performs best on all datasets and second that machine learning algorithms mostly rely on hyperparameter optimization to perform well. Hence, the underlying problem can be described as *Combined Algorithm Selection and Hyperparameter optimization (CASH)*. Some open-source examples are discussed here. The advantage of this CASH concept is obvious. The user gets as final result the optimal learner with the optimal parameter settings for his problem. However, this is at the expense of a huge multi-dimensional optimization space, containing all available learners and their corresponding hyperparameters. As a consequence, a much higher budget is needed for optimization compared to single-learner approaches like autoxgboost.
Nevertheless, a prominent example is *Auto-WEKA* by Thornton et al. (2013) which was one of the first Auto-ML releases. To solve the CASH-problem, it searches for an optimal learning algorithms of the WEKA database with tuned parameters by Bayesian optimization.
The same tuning technique uses *auto-sklearn* by Feurer et al. (2015) which is python based and uses the **scikit-learn** library as learner database. It also supports parallel

computing using a shared file system.
Also the H2O-team by Kraljevic and team (2017) offers an Auto-ML interface called *H2O AutoML*, which includes automatic training of several models and model ensembles. Like the other Auto-ML solutions, its design only requires the user to provide the data and to define the response vector. There exist an API for R and python.
Another python-based Auto-ML tool is called *Tree-based Pipeline Optimization Tool (TPOT)* by Olson, Urbanowicz, et al. (2016) and serves as "your Data Science Assistant". It uses genetic programming instead of Bayesian optimization for parameter tuning on the same **scikit-learn** learner library as Auto-WEKA and auto-sklearn. TPOT's main advantage is that it generates standalone scitkit-learn python code of the optimal model which can be further processed for modifications or model inspections.
A further Auto-ML framework introduced in this section is called *pennAI*, which is developed by the same working group as TPOT, namely by Olson, Sipper, et al. (2017). It will launch in 2018 and tries to bring artificial intelligence (AI) to the mainstream. Moreover, it provides a user-friendly graphical user interface and is specialized for complex data in the biomedical and health care domains. The machine learning part consists of six different machine learning algorithms, again from the **scikit-learn** library, that is decision tree, k-nearest neighbors, support vector machine, random forest and gradient boosting for both regression and classification and additionally logistic regression for classification and the elastic net for regression. For model tuning, like in TPOT, genetic programming is used.
Using almost identical learning algorithms, *AutoCompete* is a framework mainly to participate machine learning competitions. Hereby, it tries to serve as an easy starting point for getting access to the given problem and creating first predictive models including tuning.

5. Benchmark experiments for optimizing autoxgboost

The previous parts of this master's thesis introduced the theory behind autoxgboost. Before comparing the performance of autoxgboost directly to those of Auto-WEKA and auto-sklearn in Section 6, major package optimizations were done first. The following section illustrates the results of three benchmark experiments which were done to improve two parts of the package:

(1) factor encoding and

(2) threshold tuning.

All benchmark experiments in this thesis were calculated on the *Leibnitz Rechenzentrum (LRZ)* using the R package **batchtools** by Lang et al. (2017) for parallel running of different calculations, so-called *jobs*, which need to be compatible with the *Slurm Workload Manager*. Hereby, jobs are determined by parametrization of problems and algorithms which then can be submitted to the backend, the *CooMUC2* Linux Cluster of the LRZ. To make cluster usage as efficient as possible, each job is managed by the previously mentioned scheduling system Slurm. That means that the user needs to assign the necessary resources to each job including the number of CPUs, memory and wall time. Then, the job is scheduled until Slurm allocates it to a free slot in the cluster. If the calculations of a job exceed those predefined resources of computation time or memory, it gets canceled immediately.

5.1. Factor encoding benchmarks

The first benchmark had two goals: first, the best performing encoding method should be found. Second, if the benchmark showed that it could be useful to apply a different method for factor features with a high compared to those with low cardinality, an optimal value for a boundary between both methods should be determined. Such a boundary could for example mean to dummy encode all factor variables whose number of factor levels are below this boundary and to impact encode the rest. Another possibility could be to apply impact encoding to all factor features while additionally dummy encode low cardinality features.
In order to get adequate results, the benchmark was performed with several different

settings on various datasets. The detailed benchmark configuration is discussed in the following.

5.1.1. Benchmark configuration

Since we were especially interested in benchmark results which are useful for the autoxgboost package, we reduced the experiments to only one learner, that is XG-Boost. Moreover, we enabled the early-stopping function, to act like the package's `xgboost.earlystop` learner described in Section 3.2.2. However, to concentrate on the different encoding methods, no further hyperparameter tuning was performed.
Nevertheless, 35 different encoding settings were tested as shown in Table 12 including integer, dummy and impact encoding. Here, as explained above, two different kinds of boundaries were benchmarked. The first one, the `impact.boundary` means, that only features with a number of factor levels above the boundary are impact encoded, while the rest of the factor features is dummy encoded. The `dummy.boundary` works the other way around. While all factors are impact encoded, all features, whose number of factor levels are below the boundary, are *additionally* dummy encoded. Note, that the same boundaries were benchmarked for both impact encoding variations. This includes the method of Micci-Barreca (2001), simply referred as "impact encoding" for this benchmark as well as the **vtreat** method of Mount and Zumel (2016), denoted by "vtreat".
When looking back to the introduction of impact encoding in Section 3.1.3, the new impact values for a certain factor level are a composition of conditional and prior probabilities of the class level of this specific observation. Its ratio is determined by the function λ(), which depends, besides the factor level's cell size, on two parameters, namely the slope and the trust parameter. Since in general, it is not clear,

Encoding configurations			
Encoding	simple	`impact.boundaries`	`dummy.boundary`
integer			
dummy			
impact	yes/no	1, 2, 3, 4, 5, 7, 10	3, 4, 5, 7, 10, 20
vtreat		1, 2, 3, 4, 5, 7, 10	

Table 12: Different encoding methods and boundary configurations used for the benchmark.

how to set their values, we set a fixed threshold $\lambda = 0.5$ between both probabilities (simple = no in Table 12). Moreover, we set $\lambda = 1$, resulting in determining the new impact values for a factor level for observation i, exclusively by the conditional probability of the target's class level of observation i, given the factor level. We denote this setting with *impact simple* (simple = yes in Table 12).

In order to make the benchmark meaningful, 12 datasets, including 3 regression, 7 binary and 2 multiclass classification problems, were chosen. All were taken from the OpenML platform. For this, the corresponding R-package **OpenML** by Casalicchio et al. (n.d.) was used. Table 13 shows the properties of each dataset. Note, that all regression datasets have -1 as number of class levels. For dataset selection, some major properties were key. Most important, each dataset needed to contain at least one factor variable with not less than 10 unique factor levels to test the impact boundaries. Datasets including high cardinality factor features were even more desirable to get insights in the general effects of impact encoding on those features and the resulting prediction performance.

Note, that the three KDD datasets *appetency*, *churn* and *upselling* containing the exact same features. They only differ on the target variable.

As performance measure, the multiclass Area Under the Curve (multiclass AUC) was used for classification tasks and the mean squared error (mse) for regression tasks. Together with XGBoost as the only learner, the jobs of this benchmark were hence determined by

$$\begin{aligned} & \#\{\texttt{Datasets}\} \times \#\{\texttt{Impact encoding methods}\} \\ = & \quad 12 \quad \times \quad 35 \quad = 420. \end{aligned}$$

Encoding benchmark datasets									
Name	Classes	Features	Instances	NAs	Numerics	Factors	Max fact. levels	Min fact. levels	Median fact. levels
credit-g	2	21	1000	0	7	13	10	2	4
adult	2	15	48842	6465	2	12	41	2	5.5
KDD_appetency	2	231	50000	8024152	192	38	15415	1	17.5
KDD_churn	2	231	50000	8024152	192	38	15415	1	17.5
KDD_upselling	2	231	50000	8024152	192	38	15415	1	17.5
bank-marketing	2	17	45211	0	7	9	12	2	3
Amazon	2	10	32769	0	0	9	7518	67	343
eucalyptus	5	20	736	448	14	5	27	8	14
cjs	6	35	2796	68100	32	2	57	10	33.5
ozone_level	−1	73	2536	0	1	72	1688	56	267.5
nasa_numeric	−1	24	93	0	4	20	14	2	4
KDD98	−1	479	191260	5587563	347	131	25847	2	8

Table 13: Datasets used for the encoding benchmark and their properties.

Each job was calculated on a single core, the wall time was not limited in the end, i.e. the wall time was increased if the initial set value of two hours was not enough. The memory usage was limited by 62GB, which was not enough for some datasets, for which all factor features were dummy encoded.
Within each job, a wrapped learner depending on the encoding method was created and afterward trained via repeated cross-validation with 10 folds and 10 repetitions. Hence, withing each job, 100 models where trained and evaluated on the corresponding cross-validation test sets using the *weighted average 1 vs. 1 multiclass AUC*, short multiclass AUC, for classification and mse for regression tasks. This was done to suppress stochastic effects, which could interfere with the results, if only few models were trained.

5.1.2. Benchmark results

The benchmark was evaluated by comparing the raw performances of each run for each dataset first. Several datasets did not provide a significant trend towards a single encoding method. The boxplots for those datasets are shown in the appendix in Figure 19 to Figure 26 in order of their appearance in Table 13. Also the results for the datasets KDD_churn and KDD_upselling can be found there, because of the similar look of the plots compared to the one of the KDD_appentency, which is showed in this section. Note, that for all KDD datasets, dummy encoding of the included high cardinality factor features required more than 62 GB memory which led to an error for all related jobs due to their resources limitations.
The remaining boxplots are shown in the following and offer clearer results. Starting with Figure 11, we see that for the appetency dataset integer encoding works very well and is significantly better than all other encoding methods. Moreover, since the number ob features is not changed compared to the original data, it is also the fastest method. All other encoding variations are not significantly different amongst themselves.
When looking at the amazon data in Figure 12, things are changed since dummy and integer encoding are clearly worse than all impact or mixed encoding methods. Between dummy and integer encoding and between all other encodings, it is however very close and not significantly different.
The results for the cjs datasets, shown in Figure 13, indicate a larger difference between integer and dummy encoding on the one side and the impact encodings on the other one. Moreover, it can be clearly seen that both methods have a much

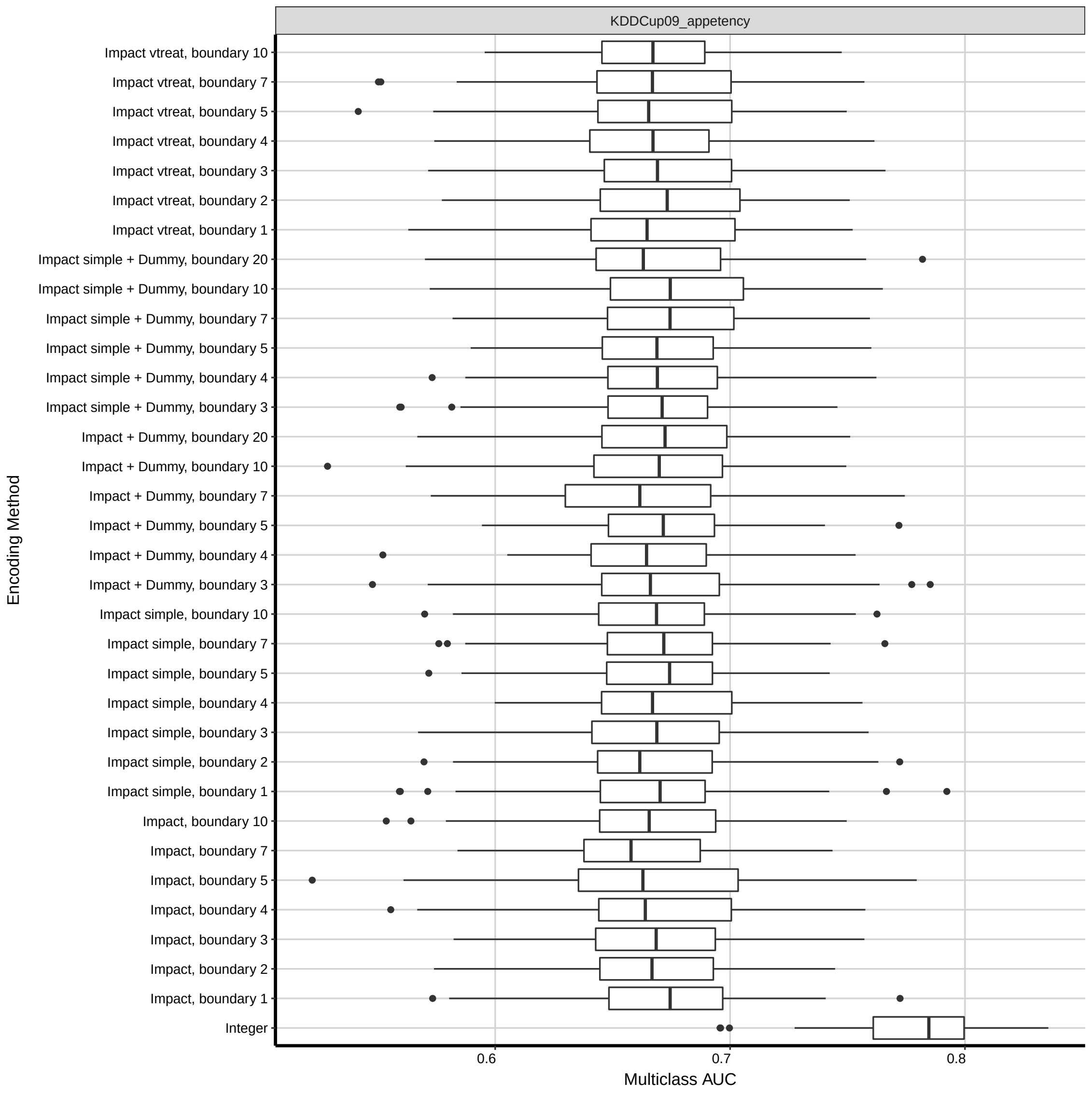


Figure 11: Benchmark results for KDD appetency dataset.

smaller deviation during the 100 runs. The multiclass AUCs for the impact encoding methods are widely spread between 0.6 and 0.9. In contrast, the performance for dummy and integer encoding lies for all runs close to 1 which means an enormous difference. However, this extreme result might be an indicator for a broken dataset which somehow includes the class levels within a factor feature in a hidden manner. This could explain why dummy and integer encoding work in such an impressive way. Nevertheless, similar results like for KDD_appetency are provided by the boxplots of the KDD98 regression task, illustrated in Figure 14. It shows, that integer encoding

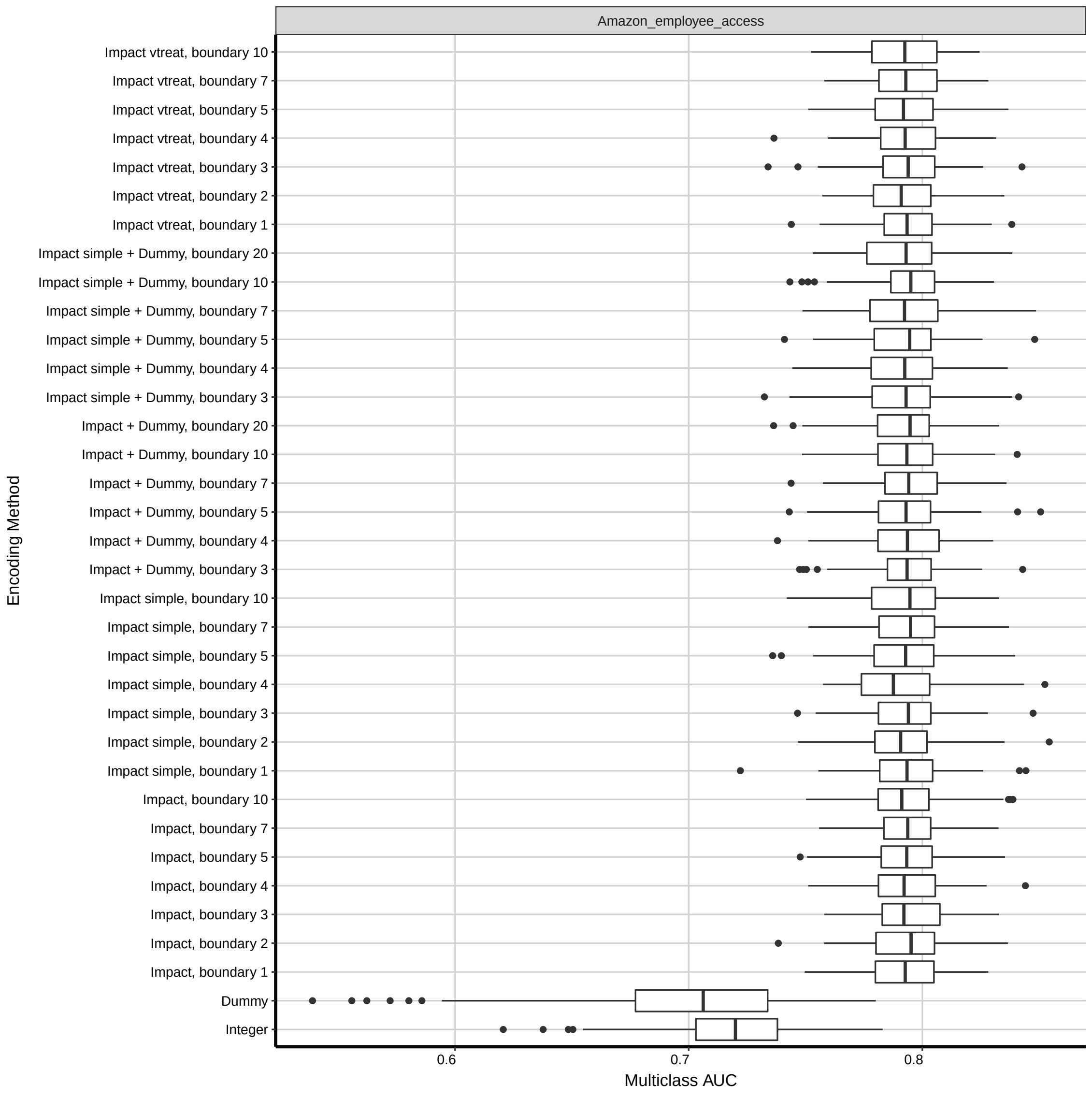


Figure 12: Benchmark results for Amazon dataset.

does not only work for classification tasks. Again, it is significantly better than impact encoding or a mixed variation of impact and dummy encoding. While also for this dataset, pure dummy encoding needed to much memory than allowed, all encoding methods except integer encoding are not significantly different among themselves. As a result, it is not clear, which encoding method is really the best one. Hence, Figure 15 shows the ranks of each encoding methods averaged over the single ranks achieved on all datasets. Note, that for datasets for which dummy encoding produced errors due to memory limitations, dummy encoding got automatically rank 35. This

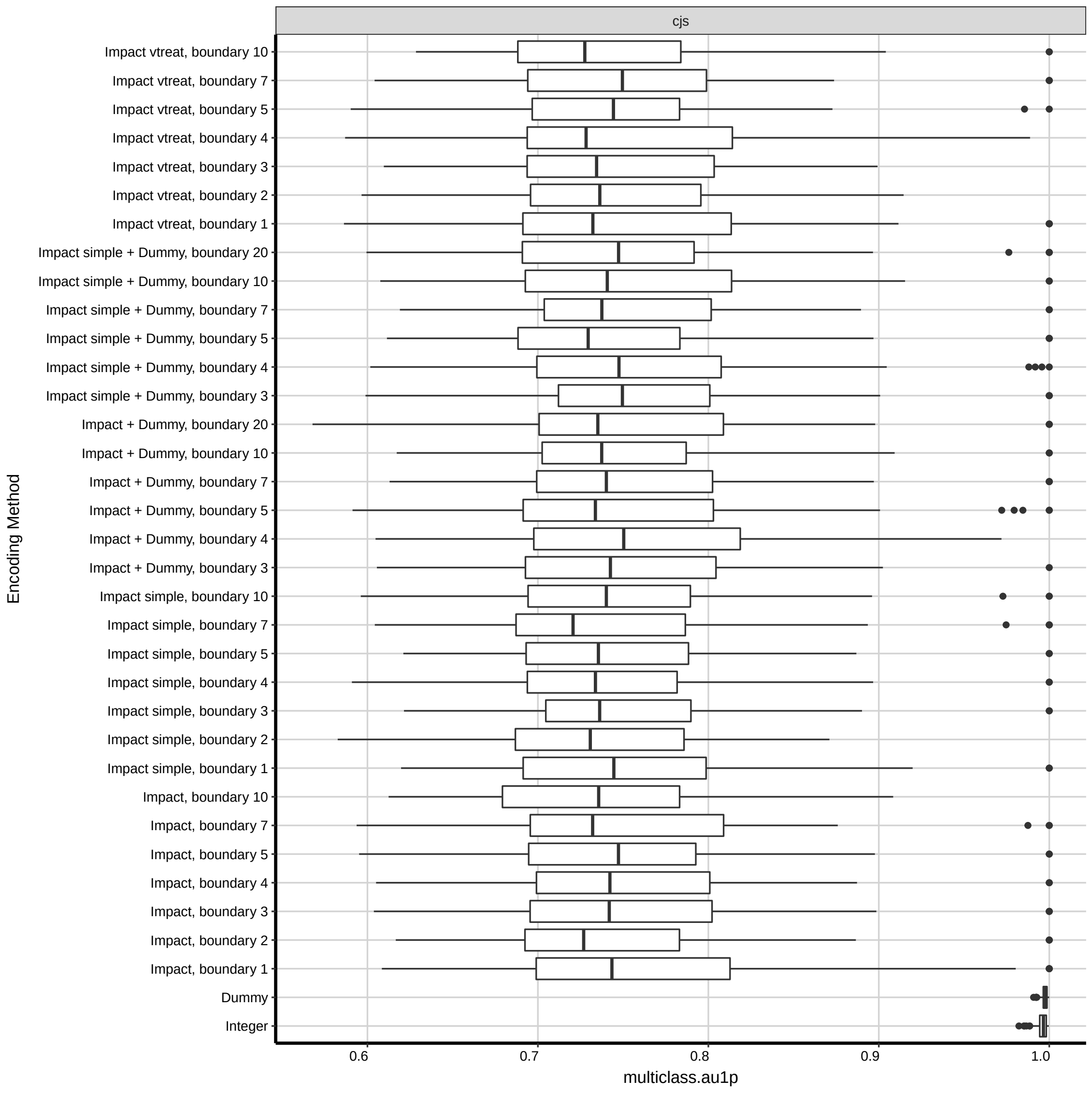


Figure 13: Benchmark results for cjs dataset.

explains, why its average rank is found in the rearmost midfield with an average rank of 20.21.

Figure 15 shows that integer encoding is the best performing encoding variation with an average rank of 11.25 in total for all datasets. Hence, because of all discussed results it is chosen to be implemented within autoxgboost for the following benchmark in Section 6.

When looking more precisely on Figure 15 and especially at impact encoding, one cannot see a clear pattern of the well performing methods. While, a dummy boundary

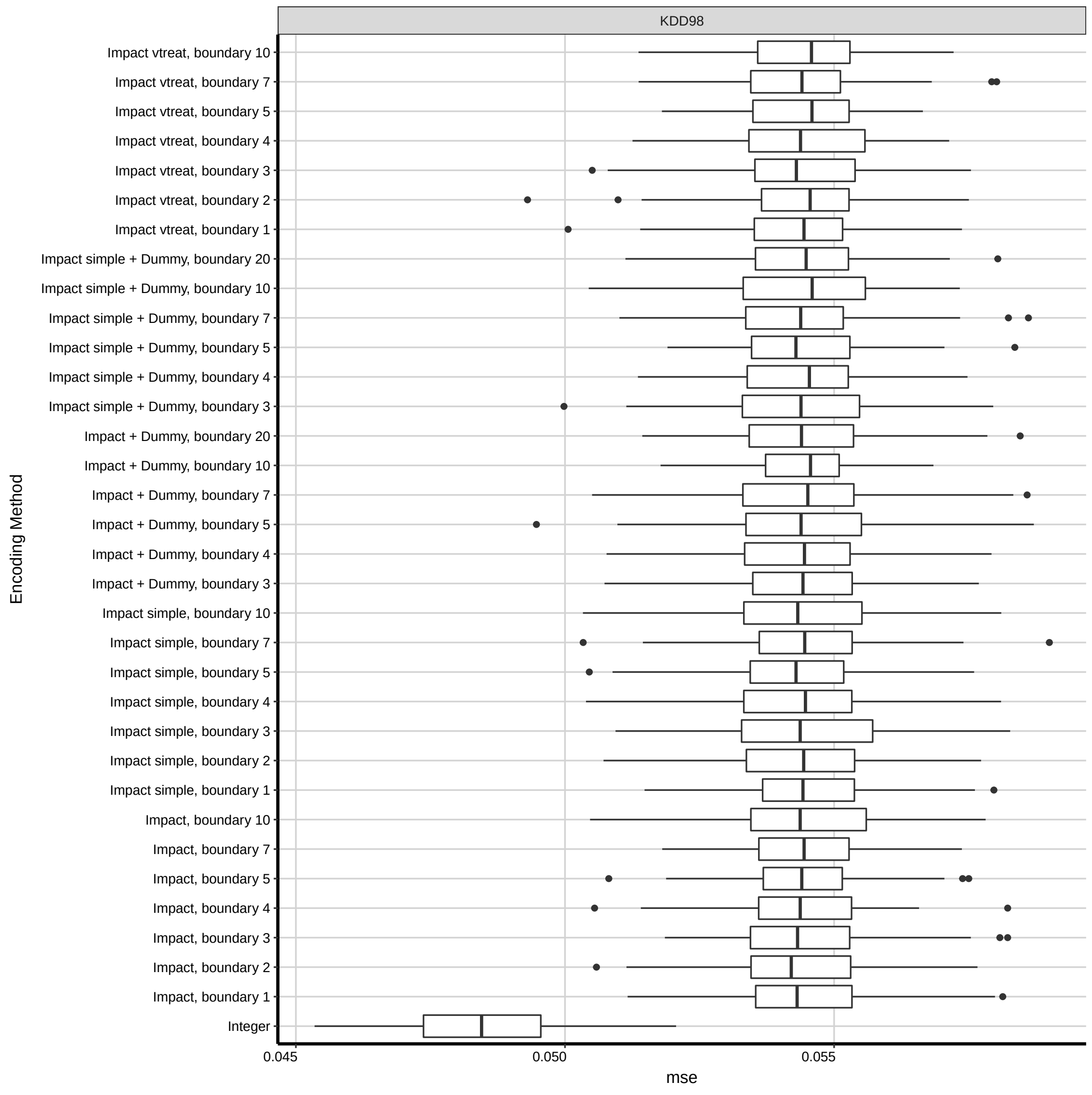


Figure 14: Benchmark results for KDD98 dataset.

of 4 performed best after integer encoding, it is followed by simple impact encoding with additional dummy encoding below a boundary of 7 factor levels.
With an average rank of 14.17,the best pure impact encoding method, i.e. impact boundary = 1, was the one proposed by Micci-Barreca (2001) for a value of 0.5 for rate λ, However, the method included in the **vtreat** package, proposed by Mount and Zumel (2016) with an average rank of 15.08 for a boundary of 1 is not far away In general, these results need to be taken with due care since the boxplots only indicated a significant difference between integer encoding (and dummy encoding

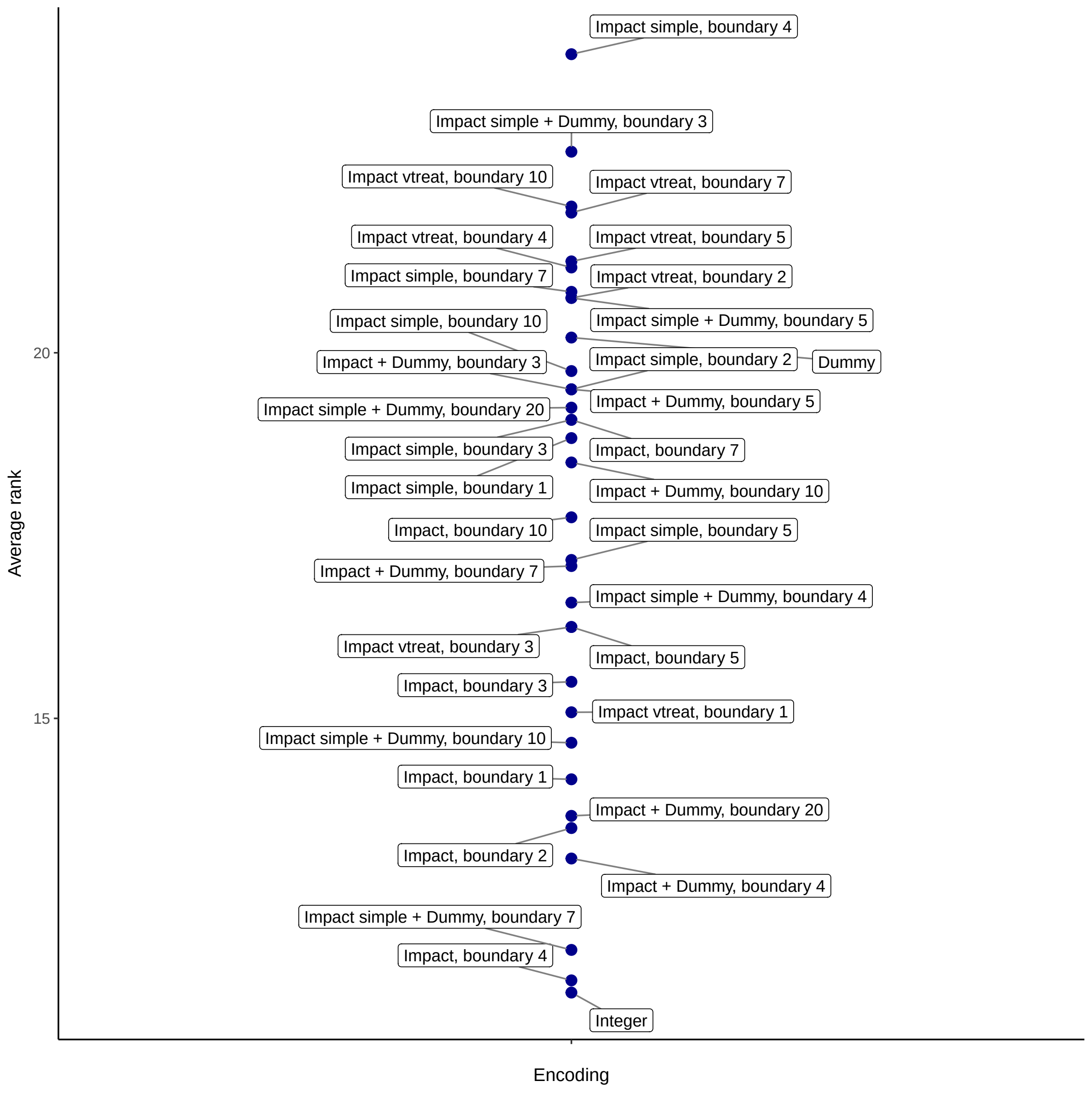


Figure 15: Average ranks of all methods on all datasets.

where applicable) and impact encoding for some datasets. Ranks are ordinal values which means that the difference between two values cannot be interpreted. Hence, they should not be overrated. Moreover, those results are only valid for the used XGBoost learner. For different learning algorithms, results may look different. This also yields for the chosen performance measure. When using the balanced error rate (ber) for the classification tasks, things changes completely. This is illustrated in Figure 16, where the pure vtreat impact encoding performs best with an average rank of 11.29. Moreover, integer encoding is only in the midfield with a rank of 17.00.

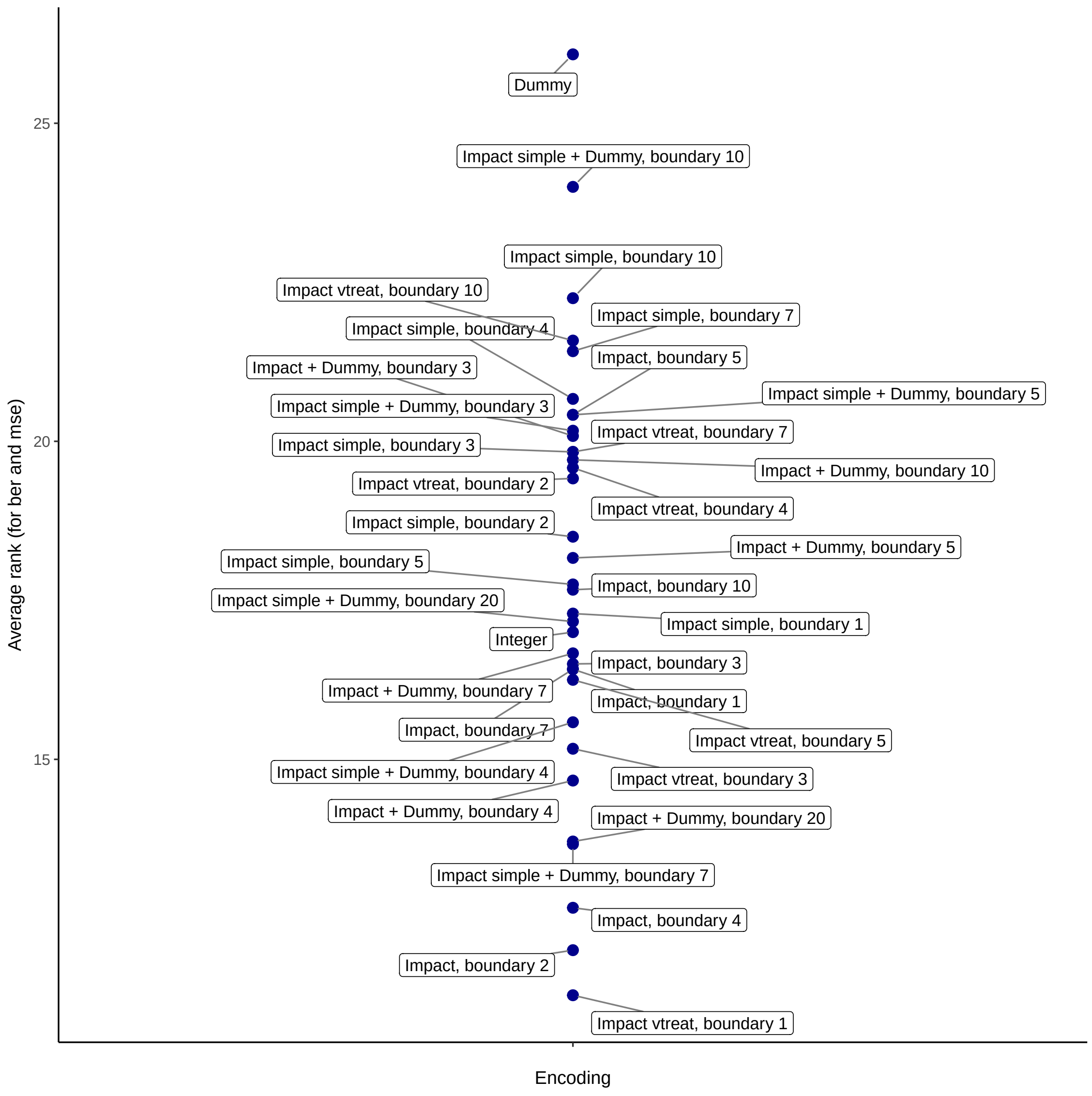


Figure 16: Average ranks of all methods on all datasets using ber and mse as performance measure.

Even worse performs dummy encoding by being last with a rank of 26.08. This strongly reveals the need for a more general benchmark as part of future research, including several learning algorithms, more datasets and different measures.

Furthermore, feature hashing might be worth a closer look as well. Its ability to combine multiple categorical features into only few new ones reveals much potential, especially for reducing the time for model training. A package, which should be considered for implementing further preprocessing operations is **mlrCPO** by Binder

(2018), where *CPO* stands for *Composable Preprocessing Operators*. The packages provides a wide range of preprocessing methods which can be attached as wrappers to **mlr** learner objects.

5.2. Threshold tuning benchmarks

When first setting up benchmarks for autoxgboost, we quickly recognized, that even though tuning the threshold improves the predictive performance of autoxgboost, it takes a lot of computational time. Hence, we tried to improve the tuning algorithm in the way that it might not only improve the speed of the whole procedure significantly, but also potentially increase the predictive performance even further. Since the optimizing process is not as time consuming for binary class problems, we concentrated on multiclass threshold tuning for improvements.

5.2.1. Algorithm benchmark

As a first step we generally had to improve the multiclass threshold tuning algorithm by finding a fast optimizer, delivering the best possible performance. The special requirements for such a black-box function optimizer were already discussed in Section 3.3. Hence, we chose different optimizing algorithms for the benchmark.
While this general approach was already introduced in Section 3.3, this section shows the results of two benchmarks, which led to choosing the **GenSA** package for multiclass threshold tuning and finding the algorithm's optimal hyperparameters.

5.2.1.1. Benchmark configuration

Both benchmarks used the same datasets whose properties are shown in Table 14. They were chosen from the OpenML database, based on several criteria. First of all, all datasets needed obviously to be multiclass classification problems with a number of unbalanced class levels between 3 and 20. Moreover, they should have a moderate number of features and observations without missing values.
To get general results which are not biased by the chosen learning algorithm, four different learners were chosen for the benchmark. Those are *rpart*, *randomForest*, *ksvm* and *naiveBayes* which are all included in **mlr**. For model evaluation, the mean misclassification error (mmce), balanced error rate (ber), kappa and weighted kappa (wkappa) were used as performance measures. For measures, which originally need to be maximized (kappa and wkappa), the achieved performance was multiplied by -1.
The benchmark algorithms were reduced to the optimization of the performance function. For this purpose, a learner was trained via holdout split on the training data and predicted on the test set. These predictions were then used to measure and improve

Threshold tuning benchmark datasets			
Name	Classes	Features	Instances
car	4	7	1 728
splice	3	62	3 190
analcatdata_authorship-a4	4	71	841
rmftsa_sleepdata	4	3	1 024
cardiotocography	10	36	2 126
volcanoes-a1	5	4	3 252
autoUniv-au6-750	8	41	750
autoUniv-au6-1000	8	41	1 000
cardiotocography	3	36	2 126
thyroid	3	22	3 772

Table 14: Datasets used for the threshold tuning benchmark and their properties.

the performance by optimizing the threshold, This means, that the interesting part of the benchmark was reduced to comparing how fast and well the different algorithms are able to improve the performance depending on the threshold. For that, the starting threshold for each algorithm was the vector $t_{\text{start}} = \underbrace{(1/k, \ldots, 1/k)}_{k-times}$, where k is the number of class levels of the datasets' target variable. Moreover, since the threshold values naturally sum up to 1, each algorithm was given the corresponding lower and upper value bounds as restrictions for the value space.
The first algorithms of the benchmark was the preliminary in **mlr** implemented *Covariance Matrix Adaptation Evolution Strategy (CMA-ES)* which was introduced by Hansen and Ostermeier (1996). The underlying evolution strategy approach is based on the principle of biological evolution, consisting of variation and selection. Variation happens in each iteration (generation) of the algorithm by stochastically drawing new candidate solutions from a multivariate normal distribution, generated by the current "parental" values. At those new candidate points, the objective function gets evaluated, and based on their performance, the best candidates become the next parents in the following iteration. Thus, the mean and the covariance matrix of the distribution are updated such that the likelihood of previously successful candidate solutions is maximized. Updating the latter is done by the method of *Covariance Matrix Adaption (CMA)*. Within **mlr**, the CMA-ES algorithm was implemented by the `cma_es()` function of the **cmaes** package by Trautmann et al. (2011).

Another CMA-ES implementation in R, which is also part of the benchmark, is given by the function `cmaes()` of the **cmaesr** package by Bossek (2016). This function also allows restarts.
A further stochastic algorithm, which was benchmarked, was the **GenSA** algorithm by Yang Xiang et al. (2013) which was already introduced in detail in section Section 3.3.2.
The last benchmarked algorithm was **hydroPSO** by Zambrano-Bigiarini and Rojas (2013), where PSO stands for *Particle Swarm Optimization*. It is implemented by the `hydroPSO()` function of the **hydroPSO** package by Zambrano-Bigiarini (2014). As a stochastic population-based algorithm, it shares similar properties with the evolutionary algorithm of CMA-ES. Its was discovered while investigating simulations of the social behavior of bird flocks and was introduced by Kennedy and Eberhart (1995). The method is frequently used due to its efficiency and flexibility. In contrast to CMA-ES, PSO does not include evolutionary parts for variability. Instead, it searches new solution candidates on the basis of individual and neighborhood-based velocity and position. The advantages of PSO lead to developing several different variations.
For this benchmark, a job was defined as dataset - learner - optimization algortihm - measure combination. In order to antagonize stochastic effects, each job was replicated ten times. Hence, the number of jobs was

$$\begin{aligned} & \#\{\texttt{Datasets}\} \times \#\{\texttt{Learners}\} \times \#\{\texttt{Optimizers}\} \\ \times\ & \#\{\texttt{Measures}\} \times \#\{\texttt{Replications}\} \\ =\ & 10 \times 4 \times 4 \times 4 \times 10 = 6400 \qquad . \end{aligned}$$

Each job was running on a single core machine of the LRZ Linux Cluster.

5.2.1.2. Benchmark results

Since the datasets were of moderate size, it was possible to set the maximal iterations for each algorithm to 20000. The idea behind this was to find out how fast each algorithm converges at some point and hence to get an overview over the *elbow point* of each optimizer. This point is the optimal ratio between the achieved performance and the time needed to reach it. Hence, an *optimal* optimization algorithm which would be likewise the fastest and best performing one would have its elbow point above and before the other algorithms, when looking at the evaluation measure.
In order to compare the results for different datasets, measures and learners, first,

the best value of each objective function was determined and regarded as global optimum. Afterward, three ϵ-neighborhoods, 10%, 5% and 1%, were defined around each optimum. Each experiment was then scored at each function evaluation, i.e. for a performance function value inside a neighborhood, it got 1 point inside the 10% neighborhood, 2 points inside the 5% and 3 points inside the 1% neighborhood. If no neighborhood was reached, no point was scored. The trace for each experiment was then used to calculated the *success rate* per algorithm, for an iteration *i*:

$$\begin{aligned}\text{sr}_i(\texttt{Algorithm}) &= \frac{\text{achieved points at iteration } i \text{ by the algorithm}}{\text{maximal possible points at iteration } i} \\ &= \frac{\sum_{\text{traces}} \text{points}(\text{i}|\texttt{Algorithm})}{3 \text{ points} \times \#\{\texttt{Datasets}\} \times \#\{\texttt{Learners}\} \times \#\{\texttt{Measures}\}}\end{aligned}$$

This success rate indicates how many percent of a problem an algorithm has solved at iteration i. Figure 17 shows the mean (left) and individual success-rates of all benchmarked algorithms.

Clearly, the GenSA algorithm outperformed the other methods when looking at the success rates. However, it was the last one which reaches its elbow point. Nevertheless, is offered the most potential for this kind of optimization problem.

When looking at the individual curves for the 10 replications, one can see that this result is even significant at the end. Surprisingly, the previously default of **mlr**, the cmaes algorithm performed significantly worst in the whole benchmark.

Having found the best performing algorithm for multiclass threshold tuning, we were

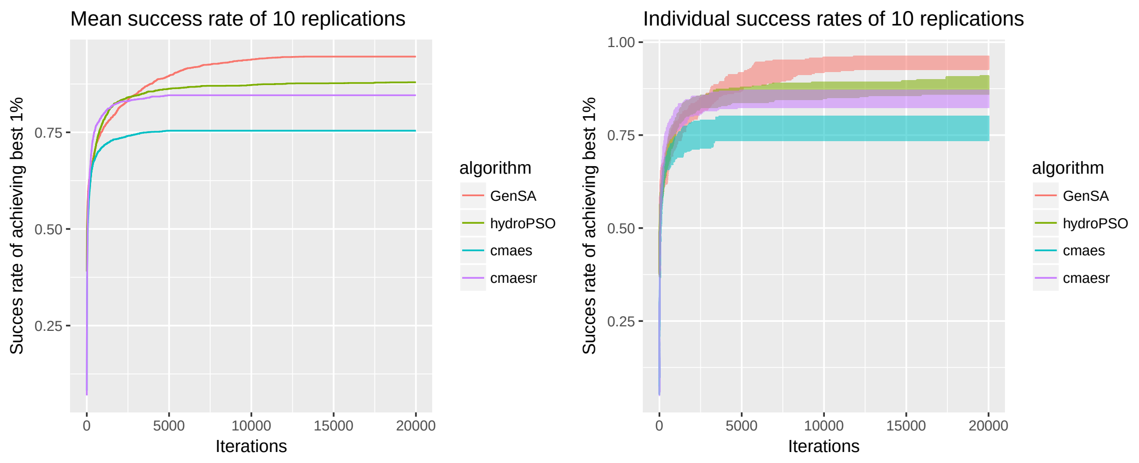


Figure 17: Mean (left) and individual success-rate comparison between the benchmarked optimization functions.

interested in further improving it by finding the optimal hyperparameters of function `GenSA()`. This was done in a second benchmark, which will be discussed in the next parts of this section.

5.2.2. Hyperparameter benchmark

While the GenSA algorithm already outperformed its competitors in the main benchmark, it might be further improved for the special optimization problem of threshold tuning when an optimal hyperparameter setting is used. Moreover, a default setting for **mlr**'s integrated threshold tuning should be found, which provides good results while being fast.

5.2.2.1. Benchmark configuration

One idea to find the optimal hyperparameters for the `GenSA` function was to tune it with iterated F-racing (irace) by López-Ibáñez et al. (2016), but in order to get a better insight in how each parameter works for the algorithm, the idea of making a benchmark experiment for a large parameter grid was pursued. The corresponding parameters and benchmarked values can be seen in Table 15. Each combination was applied to the GenSA function for tuning the threshold using the predictions of a 10-fold cross-validation on the same datasets, which were previously used for the main threshold tuning benchmark of Section 5.2.1. Remember, their properties were illustrated in Table 14.

GenSA Parameterset		
Name	values	Description
`smooth`	TRUE/FALSE	TRUE when the objective function is smooth, or differentiable almost everywhere in the region of par, FALSE otherwise.
`temperature`	$250, 1000, 5000, 10000$	Initial value for temperature.
`visiting.param`	$2.1, 2.3, 2.5, 2.7, 2.9$	Parameter for visiting distribution.
`acceptance.param`	$0, -3, -6, -9, -12, -15$	Parameter for acceptance distribution.
`simple.function`	TRUE/FALSE	FALSE means that the objective function is complicated with many local minima.

Table 15: GenSA hyperparameters and ranges for the benchmark

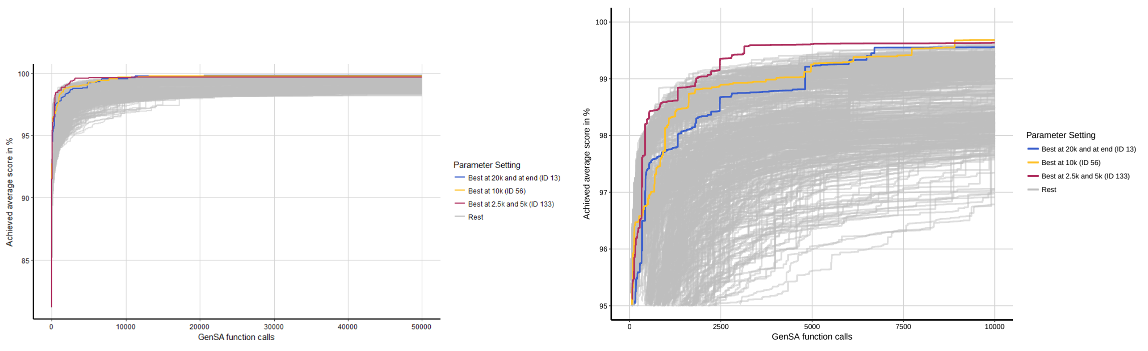
Additionally to the previous learners, XGBoost was used for model training. With the two evaluation measures mmce and ber, the jobs of this benchmark were given by

$$\begin{aligned} & \#\{\texttt{Datasets}\} \times \#\{\texttt{Learners}\} \\ \times\ & \#\{\texttt{Measures}\} \times \#\{\texttt{Parameter combinations}\} \\ =\ & 10 \times 5 \times 2 \times 480 = 48000. \end{aligned}$$

Each job was calculated on a single core machine without special wall time or memory limitations.
To get a good overview over the algorithm's progress during optimization, the maximal function calls for the GenSA algorithm were set to 30 000. Again, the aim of the benchmark is to find a parameter setting which provides the best trade-off between performance speed and quality.

5.2.2.2. Benchmark results

Within this benchmark experiment, each of the 480 parameter configurations was used on the 100 different dataset/learner/measure settings. For each of these settings, the best achieved value was determined and used as maximal score. Afterward, for each parameter configuration, the achieved score per iteration averaged on all 100 settings was calculated. These score traces of all 480 parameter configurations were finally compared. The result is illustrated in Figure 18. One can see in Figure 18a that the performance of all parameter configurations mostly improve up to the first 10000 function calls of the objective function. Therefore, this interesting



(a) Percentage of solved problems of each GenSA parameter configuration

(b) Detailed view on Figure 18a

Figure 18: Overview and detailed view of each GenSA parameter configuration performance.

Best GenSA parameter configurations (out of 480)						
ID	smooth	simple	temp	visit	accept	best at
133	FALSE	TRUE	250	2.5	-15	2.5k and 5k
56	FALSE	FALSE	1000	2.9	-12	10k
13	FALSE	FALSE	250	2.5	-15	20k to end

Table 16: Best GenSA parameter configurations.

sector is shown in more detail in Figure 18b. Three parameter configurations are highlighted in both plots whose details are shown in Table 16. The red one (ID 133) provides the best performance after 2 500 and 5 000 function calls, while the yellow one (ID 56) is the best at 10 000 and the blue one (ID 133) is the best performing configuration after 20 000 function calls all the way up to the last function evaluation. Clearly, parameter configuration 133 achieves its maximal performance already at around 3 000 function calls. Even though the other two configurations are able to pass it before 10 000 and 20 000 function calls, the difference in performance is only little. Hence, parameter configuration 13 was chosen as best trade-off between computational time and optimization performance and hence built into **mlr** as default parameter configuration.
Besides only looking on the scores, one can also look on the average rank of a parameter configuration per function call. This was done on Figure 27 to Figure 30, which can be found in the appendix, first for all datasets and then split into 3 dataset groups. The first contains all 3-class datasets, while the other figures show the ranks for the 4/5-class and 8/10 class datasets. Below each illustration, the parameter settings of the best performing configurations can be found in Table 19 to Table 22 in the appendix. The plots and tables support the previously found influence and connections of the different GenSA parameters. We can see that setting the `simple.function` argument to TRUE, the algorithm generally increases its performance fast resulting in low ranks. The speed is also improved by setting a big absolute negative value to the acceptance parameter, which reduces the acceptance of uphill jumps. In contrast, setting `simple.function` to FALSE increases the probability to get best overall performance, not being trapped in a local optimum. However, that comes at the expense of a long run-time.
Another indication about the structure of the threshold objective function is that almost all well performing parameter configurations have `smooth` set to FALSE.

6. Autoxgboost benchmarks

After optimizing autoxgboost, its performance is compared to the other Auto-ML solutions Auto-WEKA by Thornton et al. (2013) and auto-sklearn by Feurer et al. (2015),

6.1. Benchmark configuration

In order to assure the best possible comparability, we tested autoxgboost on the same datasets as the other two competitors used within their papers. This includes the same training- and test-data splits in combination with a *holdout* strategy and the same performance measure. The datasets are presented in Table 17. Moreover, like in the paper of Thornton et al. (2013), 25 runs were performed.
The parameter settings of autoxgboost were mostly set to the default values, which are known from Table 10. However, the tuning budget was increased to 150 **mlrMBO**-iterations and stopped after 10 hours, if tuning was not complete until then. For

Autoxgboost benchmark datasets					
Name	Factors	Numerics	Classes	Train instances	Test instances
Dexter	20 000	0	2	420	180
GermanCredit	13	7	2	700	300
Dorothea	100 000	0	2	805	345
Yeast	0	8	10	1 038	446
Amazon	10 000	0	49	1 050	450
Secom	0	591	2	1 096	471
Semeion	256	0	10	1 115	478
Car	6	0	4	1 209	519
Madelon	500	0	2	1 820	780
KR-vs-KP	37	0	2	2 237	959
Abalone	1	7	28	2 923	1 254
Wine Quality	0	11	11	3 425	1 469
Waveform	0	40	3	3 500	1 500
Gisette	5 000	0	2	4 900	2 100
Convex	0	784	2	8 000	50 000
Rot. MNIST + BI	0	784	10	12 000	50 000

Table 17: Datasets used for the comparison benchmark of autoxgboost, Auto-WEKA and auto-sklearn.

this, the XGBoost hyperparameters corresponded to the default ones from Table 7. Additionally to autoxgboost, the featureless learner of **mlr** was trained as a baseline. Furthermore, a random forest of the **randomForest** package and a decision tree from the **rpart** were trained with their default hyperparameters set.
Also this benchmark was performed on the LRZ Linux Cluster. Given the benchmark configuration described above, 1600 jobs were created, determined by

$$\begin{aligned} & \#\{\texttt{Datasets}\} \times \#\{\texttt{Learners}\} \times \#\{\texttt{Repetitions}\} \\ = \; & 16 \times 4 \times 25 = 1600. \end{aligned}$$

The tree, random Forest and featureless learner were all trained on a single-core machine with a maximum memory of 62GB. No time limit was set for those learners. In contrast, due to the native support for parallelization, autoxgboost was running on the parallel cluster on 28 cores also using 62GB of memory. Nevertheless, the overall time budget was limited to 12 hours.
As seen in Table 17, the chosen datasets are very different regarding the number of numeric and factor features, but also when looking at the number of target class levels and the train and test dataset sizes. Hence, the datasets chosen by Thornton et al. (2013) should serve as an adequate heterogeneous base to test autoxgboost's performance on different situations.

6.2. Benchmark results

As mentioned before, each job was repeated 25 times. Afterward, 100 000 bootstrap samples were drawn of those runs of which 4 random runs were selected for each bootstrap sample. Then, the one of these four runs with the best performance was saved. Finally, the median of those 100 000 mean misclassification error values is returned and presented in Table 18. The bold numbers in each row indicates the best performing algorithm for the specific dataset. Training of the decision tree and the random forest failed on the datasets Dexter and Dorothea. Hence, no comparison values for those combinations are available in Table 18.
Obviously, the featureless learner is not a real competitor but an indicator that all other learners work as they should, i.e. they should significantly outperform this baseline learner. As we can see easily in Table 18, only for the datasets Dorthea and Secom, the featureless learner achieves surprising results by providing better per-

Autoxgboost benchmark results						
Dataset	baseline	rpart	randomForest	autoxgboost	Auto-WEKA	auto-sklearn
Dexter	52,78	*Error*	*Error*	12.22	7.22	**5.56**
GermanCredit	32.67	29.67	**26.33**	27.67	28.33	27.00
Dorothea	6.09	*Error*	*Error*	**5.22**	6.38	5.51
Yeast	68.99	42.25	**37.30**	38.88	40.45	40.67
Amazon	99.33	74.22	22.00	26.22	37.56	**16.00**
Secom	**7.87**	10.43	8.30	**7.87**	**7.87**	**7.87**
Semeion	92.45	36.06	6.92	8.38	**5.03**	5.24
Car	29,15	5.98	1.54	1.16	0.58	**0.39**
Madelon	50.26	21.41	25.64	16.54	21.15	**12.44**
KR-vs-KP	48.96	2.92	1.77	1.67	**0.31**	0.42
Abalone	84.04	75.98	75.26	73.75	**73.02**	73.50
Wine Quality	55.68	48.40	**32.68**	33.70	33.70	33.76
Waveform	68.80	28.33	15.73	15.40	**14.40**	14.93
Gisette	50.71	7.33	2.57	2.48	2.24	**1.62**
Convex	0.50	48.80	23.43	22.74	22.05	**17.53**
Rot. MNIST + BI	88.88	78.19	53.43	47.09	55.84	**46.92**

Table 18: Benchmark results are median percent error across 100 000 bootstrap samples (out of 25 runs) simulating 4 parallel runs. Bold numbers indicate best performing algorithms. Best Auto-ML results are represented by blue numbers

formances than other learning algorithms. As a consequence, the results on those datasets should be questioned and the datasets further investigated.
When concentrating on the other learners, and especially on the rpart at first, one can see that it is significantly outperformed on almost all datasets by all automatic machine learning algorithms and the random forest. Only on the Madelon dataset, it performs at least better than the random Forest, but still clearly worse than autoxgboost and auto-sklearn.
The biggest surprise of this benchmark is the performance of the random Forest. While this learner is known for its strong performance without any parameter tuning, it provided the best performance of all methods on three datasets. Moreover, the performance difference is generally not very large on most datasets.
When looking only on the automatic machine learning algorithms autoxgboost, Auto-WEKA and auto-sklearn, one can concentrate on the blue numbers in Table 18. On half of the 16 datasets, auto-sklearn provides the best results. So does Auto-WEKA on four and autoxgboost on two datasets. On one of the two remaining datasets,

Secom, all Auto-ML algorithms deliver the same performance, while autoxgboost and Auto-WEKA slightly perform better than auto-sklearn on the wine quality dataset. Consequently, auto-sklearn has been identified as superior Auto-ML method beneath all benchmarked ones in this thesis. Given the fact, that Auto-WEKA and auto-sklearn are both CASH methods, whose optimization includes learner selection and hyperparameter tuning, they can be directly compared regarding their time requirements and computational complexity. As a result, there is every indication that auto-sklearn should be favored over Auto-WEKA.
In contrast, the problem space of autoxgboost is much smaller due to its restriction on a single gradient boosting learner. Hence, results should require significant less resources than a CASH algorithm. When looking again at Table 18, autoxgboost provides similar results than its two competitors on more than half of all datasets. Hence, it's right to exist is based on the case where e.g. resources are limited. On the other hand, when resources are not an issue for the data scientist, auto-sklearn should yet be preferably used.

7. Conclusion

When automating a specific process or work-flow, one needs to handle all possible potential sources of error. For automating gradient boosting with XGBoost, this means to encode factor variables as a first step. Different methods for doing this task where introduced in Section 3.1 and compared by a benchmark experiment in Section 5.1. Surprisingly, none of the more complex impact encoding variations, but simple feature hashing delivered not only the best performance, but also was the fastest method and was hence chosen for autoxgboost. However, this benchmark must be understood as a starting point for further research. This includes more and better datasets which are probably not that much unbalanced as e.g. the KDD datasets or the Amazon one. Moreover, more suitable measures should be considered for evaluating the results, since we saw that different measures might lead to different results.

As a second part, multiclass threshold tuning was improved within the **mlr** package. The starting point was a benchmark in Section 5.2.1 which compared different optimization algorithms which can handle potentially complex multimodal functions. With Generalized Simulating Annealing of the **GenSA** package winning this benchmark regarding speed and optimization quality, a second benchmark in order to determine its optimal hyperparameters was performed in Section 5.2.2. Interestingly, the parameter indicates, that the black-box threshold functions are of a rather simple, but non smooth structure.

After optimizing autoxgboost, it was compared in a last benchmark experiment in Section 6 with its competitors Auto-WEKA and auto-sklearn. These final results showed that while autoxgboost is outperformed on most datasets by auto-sklearn, it is able to compete with both on some datasets, providing state-of-the-art performance with only one learning algorithm instead using a whole library of learners. This last property makes autoxgboost much faster to perform, since the optimization space of autoxgboost is much smaller than the Combined Algorithm Selection and Hyperparameter optimization (CASH) problem, which Auto-WEKA and auto-sklearn are trying to solve.

When comparing XGBoost with other gradient boosting implementations in Section 2.6, we saw that especially lightGBM and CatBoost are two algorithms which are worth a look. Due to their superior predictive performance they might be able to replace XGBoost for our automatic gradient boosting package, but only, if this does not come at the expense of computational speed. While lightGBM already

claims to meet both demands, CatBoost comes with advanced categorical feature handling which frequently outperforms XGBoost and lightGBM. However, installation and implementation of both gradient boosting machines seem do not seem to be that easy. Further research on this topic including real world data benchmark experiments to compare the R-implementation of all three learning algorithms should be part of future work in order to improve our automatic gradient boosting package even more. This, together with an improved factor encoding could provide this extra performance needed to fully compete with auto-sklearn.
The benchmark experiments in this master's thesis indicated that the reliability on their results is not always guaranteed easily. Especially the choice of the datasets and performance measures requires great care, since little changes might have large effects. As a consequence, an even wider diversification of datasets and measures, but also learning algorithms should be considered for further benchmark experiment based research.

References

[1] Martin Binder. *mlrCPO: Composable Preprocessing Operators for mlr*. R package. 2018. URL: `https://github.com/mlr-org/mlrCPO`.

[2] Bernd Bischl. *Fortgeschrittene Computerintensive Methoden/Predictive Modeling*. Lecture Slides. Department of Statistics - LMU Munich, Apr. 2017. Chap. 7: Gradient Boosting.

[3] Bernd Bischl, Michel Lang, Jakob Bossek, Daniel Horn, Jakob Richter, and Dirk Surmann. *BBmisc: Miscellaneous Helper Functions for B. Bischl*. R package version 1.11. 2017. URL: `https://CRAN.R-project.org/package=BBmisc`.

[4] Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. "mlr: Machine Learning in R." In: *Journal of Machine Learning Research* 17.170 (2016), pp. 1–5. URL: `http://jmlr.org/papers/v17/15-066.html`.

[5] 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*. 2017. URL: `http://arxiv.org/abs/1703.03373`.

[6] Jakob Bossek. *cmaesr: Covariance Matrix Adaptation Evolution Strategy with optional restarts (IPOP-CMA-ES)*. R package version 1.0.3. 2016. URL: `https://CRAN.R-project.org/package=cmaesr`.

[7] Leo Breiman. "Bagging Predictors." In: *Mach. Learn.* 24.2 (Aug. 1996), pp. 123–140. ISSN: 0885-6125. DOI: `10.1023/A:1018054314350`. URL: `http://dx.doi.org/10.1023/A:1018054314350`.

[8] Leo Breiman. "Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author)." In: *Statist. Sci.* 16.3 (Aug. 2001), pp. 199–231. DOI: `10.1214/ss/1009213726`. URL: `https://doi.org/10.1214/ss/1009213726`.

[9] Giuseppe Casalicchio, Bernd Bischl, Dominik Kirchhoff, Michel Lang, Benjamin Hofner, Jakob Bossek, Pascal Kerschke, and Joaquin Vanschoren. *OpenML: Exploring Machine Learning Better, Together*. R package version 1.1. URL: `https://github.com/openml/openml-r`.

[10] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System." In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: `10.1145/2939672.2939785`. URL: `http://doi.acm.org/10.1145/2939672.2939785`.

[11] Stefan Coors and Florian Fendt. *shinyMlr: A graphical user interface for machine learning in R.* R package version 1.0. 2017. URL: https://github.com/mlr-org/shinyMlr.

[12] Noel Cressie. "The origins of kriging." In: *Mathematical Geology* 22.3 (Apr. 1990), pp. 239–252. ISSN: 1573-8868. DOI: 10.1007/BF00889887. URL: https://doi.org/10.1007/BF00889887.

[13] Noel A. C. Cressie. "Geostatistics." In: *Statistics for Spatial Data.* John Wiley & Sons, Inc., 2015, pp. 27–104. ISBN: 9781119115151. DOI: 10.1002/9781119115151.ch2. URL: http://dx.doi.org/10.1002/9781119115151.ch2.

[14] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. "CatBoost: gradient boosting with categorical features support." In: (2017). URL: http://learningsys.org/nips17/assets/papers/paper_11.pdf.

[15] Anna Veronika Dorogush, Andrey Gulin, Gleb Gusev, Nikita Kazeev, Liudmila Ostroumova Prokhorenkova, and Aleksandr Vorobev. "Fighting biases with dynamic boosting." In: *CoRR* abs/1706.09516 (2017). arXiv: 1706.09516. URL: http://arxiv.org/abs/1706.09516.

[16] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. "Efficient and Robust Automated Machine Learning." In: *Advances in Neural Information Processing Systems 28.* Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., 2015, pp. 2962–2970. URL: http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf.

[17] Yoav Freund and Robert E Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting." In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.1997.1504. URL: http://www.sciencedirect.com/science/article/pii/S002200009791504X.

[18] Jerome H. Friedman. "Greedy function approximation: A gradient boosting machine." In: *Ann. Statist.* 29.5 (Oct. 2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451. URL: https://doi.org/10.1214/aos/1013203451.

[19] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. "Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors)." In: *Ann. Statist.* 28.2 (Apr. 2000), pp. 337–407. DOI: 10.1214/aos/1016218223. URL: https://doi.org/10.1214/aos/1016218223.

[20] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. "The WEKA Data Mining Software: An Update." In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 10–18. ISSN: 1931-0145. DOI: 10.1145/1656274.1656278. URL: http://doi.acm.org/10.1145/1656274.1656278.

[21] Nikolaus Hansen and Andreas Ostermeier. "Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation." In: (June 1996), pp. 312–317.

[22] Benjamin Hofner, Andreas Mayr, Nikolay Robinzonov, and Matthias Schmid. "Model-based boosting in R: a hands-on tutorial using the R package mboost." In: *Computational Statistics* 29.1 (Feb. 2014), pp. 3–35. ISSN: 1613-9658. DOI: 10.1007/s00180-012-0382-5. URL: https://doi.org/10.1007/s00180-012-0382-5.

[23] Peter J. Huber. "Robust Estimation of a Location Parameter." In: *Ann. Math. Statist.* 35.1 (Mar. 1964), pp. 73–101. DOI: 10.1214/aoms/1177703732. URL: https://doi.org/10.1214/aoms/1177703732.

[24] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential Model-Based Optimization for General Algorithm Configuration." In: *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*. Ed. by Carlos A. Coello Coello. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 507–523. ISBN: 978-3-642-25566-3. DOI: 10.1007/978-3-642-25566-3_40. URL: https://doi.org/10.1007/978-3-642-25566-3_40.

[25] Donald R. Jones, Matthias Schonlau, and William J. Welch. "Efficient Global Optimization of Expensive Black-Box Functions." In: *Journal of Global Optimization* 13.4 (Dec. 1998), pp. 455–492. ISSN: 1573-2916. DOI: 10.1023/A:1008306431147. URL: https://doi.org/10.1023/A:1008306431147.

[26] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 3149–3157. URL: http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf.

[27] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. Cambridge, MA, USA: MIT Press, 1994. ISBN: 0-262-11193-4.

[28] James Kennedy and Russell Eberhart. *Particle swarm optimization*. 1995.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing." In: *Science* 220.4598 (1983), pp. 671–680. ISSN: 0036-8075. DOI: 10.1126/science.220.4598.671. eprint: http://science.sciencemag.org/content/220/4598/671.full.pdf. URL: http://science.sciencemag.org/content/220/4598/671.

[30] A. Klein, S. Falkner, N. Mansur, and F. Hutter. "RoBO: A Flexible and Robust Bayesian Optimization Framework in Python." In: *NIPS 2017 Bayesian Optimization Workshop*. Dec. 2017.

[31] Tom Kraljevic and The H2O.ai team. *R Interface for H2O*. R package version 3.16.0.2. 2017. URL: https://cran.r-project.org/web/packages/h2o/h2o.pdf.

[32] Michel Lang, Bernd Bischl, and Dirk Surmann. "batchtools: Tools for R to work on batch systems." In: *The Journal of Open Source Software* 2.10 (Feb. 2017). DOI: 10.21105/joss.00135. URL: https://doi.org/10.21105/joss.00135.

[33] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. "The irace package: Iterated Racing for Automatic Algorithm Configuration." In: *Operations Research Perspectives* 3 (2016), pp. 43–58. DOI: 10.1016/j.orp.2016.09.002.

[34] S. G. Mallat and Zhifeng Zhang. "Matching pursuits with time-frequency dictionaries." In: *IEEE Transactions on Signal Processing* 41.12 (Dec. 1993), pp. 3397–3415. ISSN: 1053-587X. DOI: 10.1109/78.258082.

[35] Oded Maron and Andrew W. Moore. "The Racing Algorithm: Model Selection for Lazy Learners." In: *Lazy Learning*. Ed. by David W. Aha. Dordrecht: Springer Netherlands, 1997, pp. 193–225. ISBN: 978-94-017-2053-3. DOI: 10.1007/978-94-017-2053-3_8. URL: https://doi.org/10.1007/978-94-017-2053-3_8.

[36] Georges Matheron. "Principles of geostatistics." In: *Economic Geology* 58.8 (1963), p. 1246. DOI: 10.2113/gsecongeo.58.8.1246. eprint: /gsw/content_public/journal/economicgeology/58/8/10.2113_gsecongeo.58.8.1246/4/1246.pdf. URL: +%20http://dx.doi.org/10.2113/gsecongeo.58.8.1246.

[37] Cyrus R. Mehta and Nitin R. Patel. "Exact logistic regression: Theory and examples." In: *Statistics in Medicine* 14.19 (1995), pp. 2143–2160. ISSN: 1097-0258. DOI: 10.1002/sim.4780141908. URL: http://dx.doi.org/10.1002/sim.4780141908.

[38] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. "Equation of State Calculations by Fast Computing Machines." In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092. DOI: `10.1063/1.1699114`. eprint: `https://doi.org/10.1063/1.1699114`. URL: `https://doi.org/10.1063/1.1699114`.

[39] Daniele Micci-Barreca. "A Preprocessing Scheme for High-cardinality Categorical Attributes in Classification and Prediction Problems." In: *SIGKDD Explor. Newsl.* 3.1 (July 2001), pp. 27–32. ISSN: 1931-0145. DOI: `10.1145/507533.507538`. URL: `http://doi.acm.org/10.1145/507533.507538`.

[40] John Mount and Nina Zumel. *vtreat: a data.frame Processor for Predictive Modeling*. 2016. URL: `https://arxiv.org/abs/1611.09477`.

[41] Randal S. Olson, Moshe Sipper, William La Cava, Sharon Tartarone, Steven Vitale, Weixuan Fu, John H. Holmes, and Jason H. Moore. "A System for Accessible Artificial Intelligence." In: *CoRR* abs/1705.00594 (2017). arXiv: `1705.00594`. URL: `http://arxiv.org/abs/1705.00594`.

[42] Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. "Automating Biomedical Data Science Through Tree-Based Pipeline Optimization." In: *Applications of Evolutionary Computation*. Ed. by Giovanni Squillero and Paolo Burelli. Cham: Springer International Publishing, 2016, pp. 123–137. ISBN: 978-3-319-31204-0.

[43] Francesco Orabona and Dávid Pál. "From Coin Betting to Parameter-Free Online Learning." In: *CoRR* abs/1602.04128 (2016). arXiv: `1602.04128`. URL: `http://arxiv.org/abs/1602.04128`.

[44] John C. Platt. "Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods." In: *ADVANCES IN LARGE MARGIN CLASSIFIERS*. MIT Press, 1999, pp. 61–74.

[45] Philipp Probst. *tuneRanger: A package for tuning random forests*. R package. 2018. URL: `https://github.com/PhilippPro/tuneRanger`.

[46] Greg Ridgeway. "Generalized Boosted Models: A Guide to the GBM Package." In: 1 (Jan. 2005), pp. 1–12.

[47] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1." In: ed. by David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press, 1986. Chap. Learning Internal Representations by Error Propagation, pp. 318–362. ISBN: 0-262-68053-X. URL: `http://dl.acm.org/citation.cfm?id=104279.104293`.

[48] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. “Boosting the margin: a new explanation for the effectiveness of voting methods.” In: *Ann. Statist.* 26.5 (Oct. 1998), pp. 1651–1686. DOI: 10.1214/aos/1024691352. URL: https://doi.org/10.1214/aos/1024691352.

[49] Matthias Schmid and Torsten Hothorn. “Boosting additive models using component-wise P-Splines.” In: *Computational Statistics & Data Analysis* 53.2 (2008), pp. 298–311. ISSN: 0167-9473. DOI: https://doi.org/10.1016/j.csda.2008.09.009. URL: http://www.sciencedirect.com/science/article/pii/S0167947308004416.

[50] C Shearer. “The CRISP-DM model: the new blueprint for data mining.” In: 5 (Jan. 2000), pp. 13–22.

[51] Nate Silver. *The Signal and the Noise: Why So Many Predictions Fail - But Some Don’t*. The Penguin Press, 2012, p. 544. ISBN: 978-1594204111.

[52] D. J. Spiegelhalter. “Probabilistic prediction in patient management and clinical trials.” In: *Statistics in Medicine* 5.5 (1986), pp. 421–433. ISSN: 1097-0258. DOI: 10.1002/sim.4780050506. URL: http://dx.doi.org/10.1002/sim.4780050506.

[53] Harold Szu and Ralph Hartley. “Fast simulated annealing.” In: *Physics Letters A* 122.3 (1987), pp. 157–162. ISSN: 0375-9601. DOI: https://doi.org/10.1016/0375-9601(87)90796-1. URL: http://www.sciencedirect.com/science/article/pii/0375960187907961.

[54] The H2O.ai team. *Driverless AI: Automatic Machine Learning for Enterprise*. 2017. URL: https://www.h2o.ai/driverless-ai/.

[55] Christian Thiele. *cutpointr: An R package for determining and validating optimal cutpoints in binary classification*. R package version 0.7.0. 2017. URL: https://github.com/Thie1e/cutpointr.

[56] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. “Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms.” In: *Proc. of KDD-2013*. 2013, pp. 847–855.

[57] Heike Trautmann, Olaf Mersmann, and David Arnu. *cmaes: Covariance Matrix Adapting Evolutionary Strategy*. R package version 1.0-11. 2011. URL: https://CRAN.R-project.org/package=cmaes.

[58] Constantino Tsallis and Daniel A. Stariolo. “Generalized simulated annealing.” In: *Physica A: Statistical Mechanics and its Applications* 233.1 (1996), pp. 395–406. ISSN: 0378-4371. DOI: https://doi.org/10.1016/S0378-4371(96)00271-3. URL: http://www.sciencedirect.com/science/article/pii/S0378437196002713.

[59] L. G. Valiant. “A Theory of the Learnable.” In: *Commun. ACM* 27.11 (Nov. 1984), pp. 1134–1142. ISSN: 0001-0782. DOI: 10.1145/1968.1972. URL: http://doi.acm.org/10.1145/1968.1972.

[60] Marvin Wright and Andreas Ziegler. “ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R.” In: *Journal of Statistical Software, Articles* 77.1 (2017), pp. 1–17. ISSN: 1548-7660. DOI: 10.18637/jss.v077.i01. URL: https://www.jstatsoft.org/v077/i01.

[61] Yichao Wu and Yufeng Liu. “Robust Truncated Hinge Loss Support Vector Machines.” In: *Journal of the American Statistical Association* 102.479 (2007), pp. 974–983. ISSN: 01621459. URL: http://www.jstor.org/stable/27639939.

[62] Yang Xiang, Sylvain Gubian, Brian Suomela, and Julia Hoeng. “Generalized Simulated Annealing for Efficient Global Optimization: the GenSA Package for R.” In: *The R Journal Volume 5/1, June 2013* (2013). URL: https://journal.r-project.org/archive/2013/RJ-2013-002/index.html.

[63] Bianca Zadrozny and Charles Elkan. “Obtaining Calibrated Probability Estimates from Decision Trees and Naive Bayesian Classifiers.” In: *Proceedings of the Eighteenth International Conference on Machine Learning*. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 609–616. ISBN: 1-55860-778-1. URL: http://dl.acm.org/citation.cfm?id=645530.655658.

[64] Bianca Zadrozny and Charles Elkan. “Transforming Classifier Scores into Accurate Multiclass Probability Estimates.” In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’02. Edmonton, Alberta, Canada: ACM, 2002, pp. 694–699. ISBN: 1-58113-567-X. DOI: 10.1145/775047.775151. URL: http://doi.acm.org/10.1145/775047.775151.

[65] Mauricio Zambrano-Bigiarini. *hydroPSO: Particle Swarm Optimisation, with focus on Environmental Model*. R package version 0.3.4. 2014. URL: https://CRAN.R-project.org/package=hydroPSO.

[66] Mauricio Zambrano-Bigiarini and Rodrigo Rojas. “A model-independent Particle Swarm Optimisation software for model calibration.” In: *Environmental Modelling & Software* 43 (2013), pp. 5–25. ISSN: 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2013.01.004. URL: http://www.sciencedirect.com/science/article/pii/S1364815213000133.

A. Appendix

A.1. Additional figures of the factor encoding benchmark

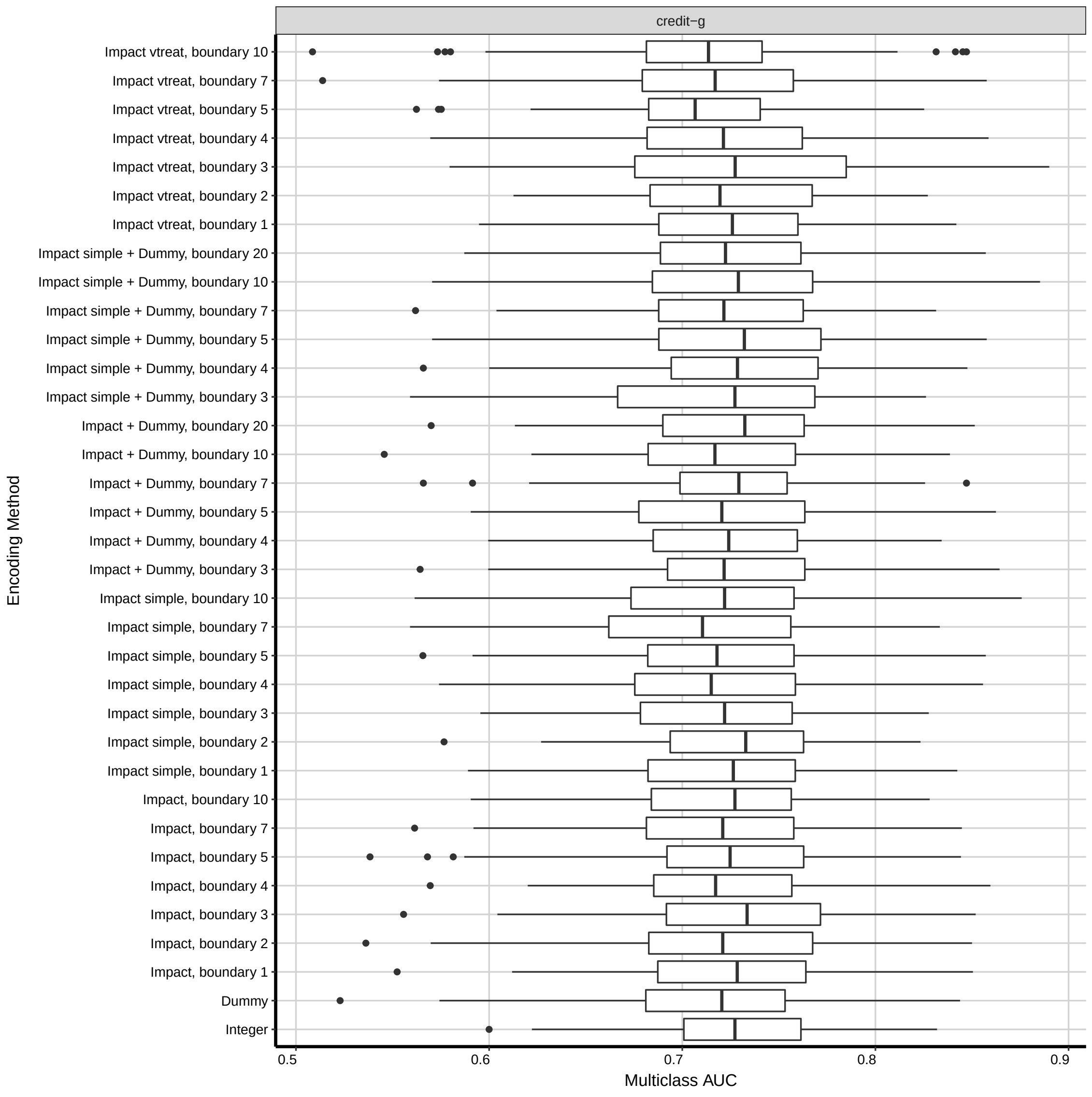


Figure 19: Benchmark results for credit-g dataset.

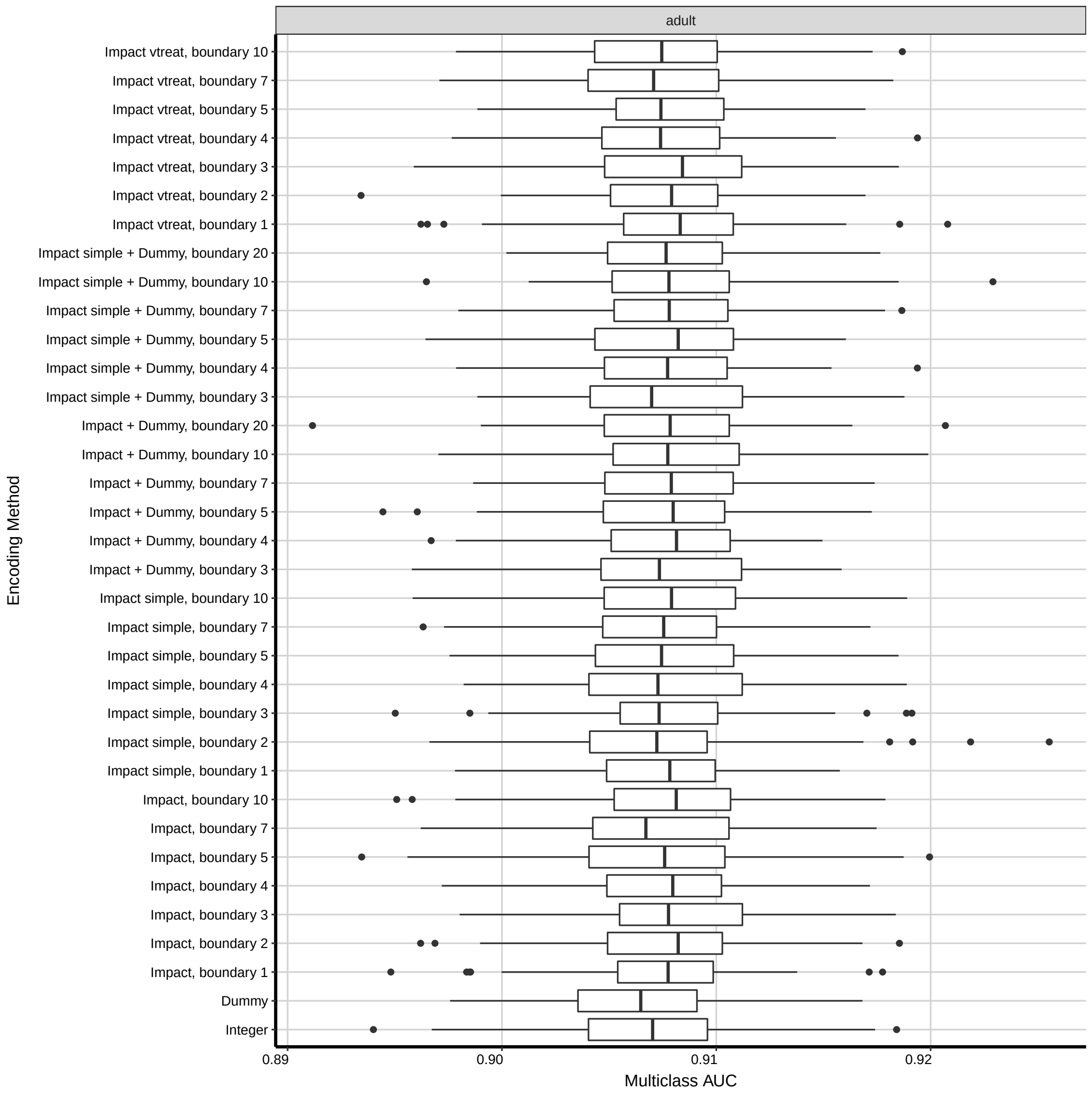


Figure 20: Benchmark results for adult dataset.

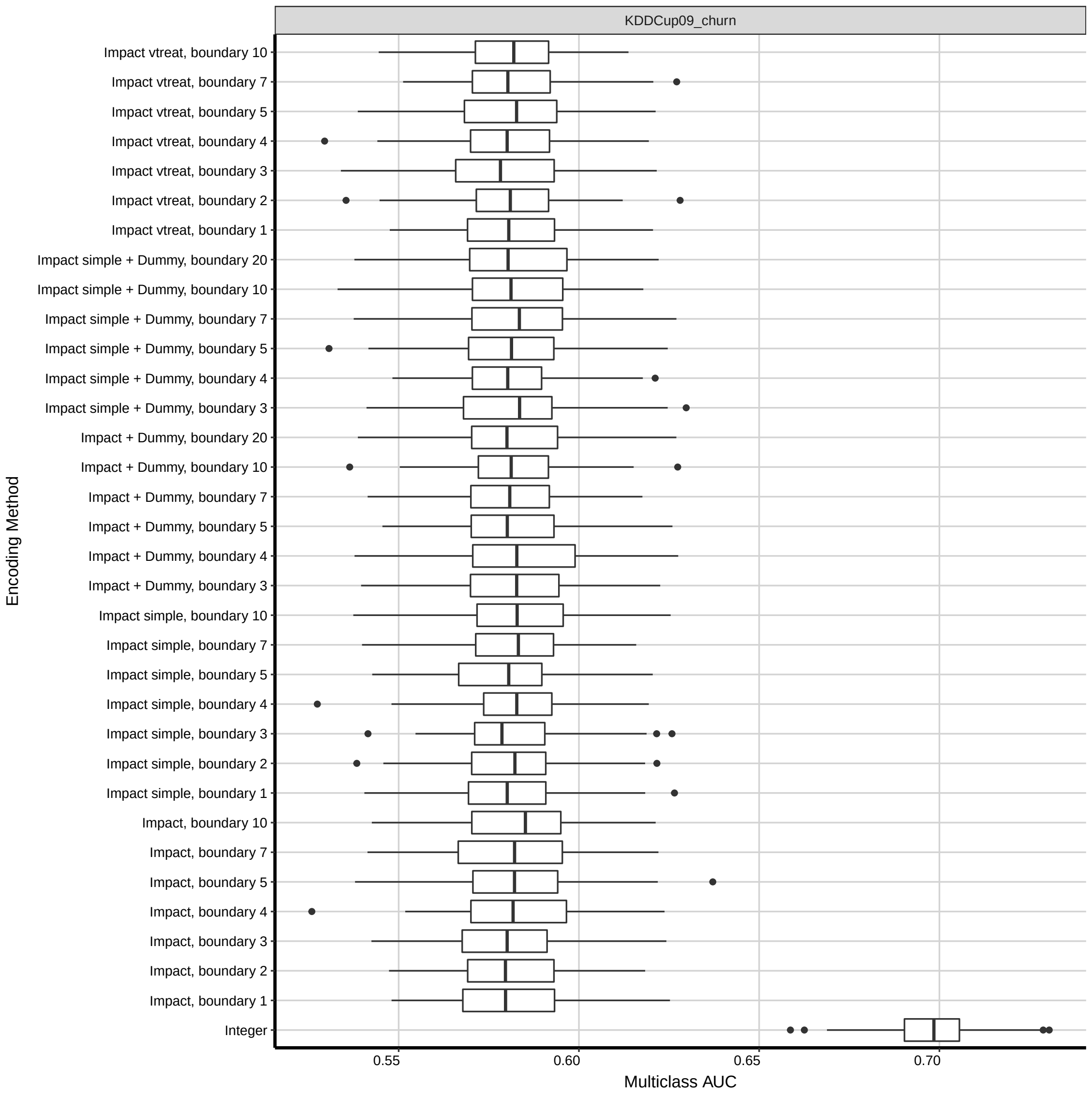


Figure 21: Benchmark results for KDD churn dataset.

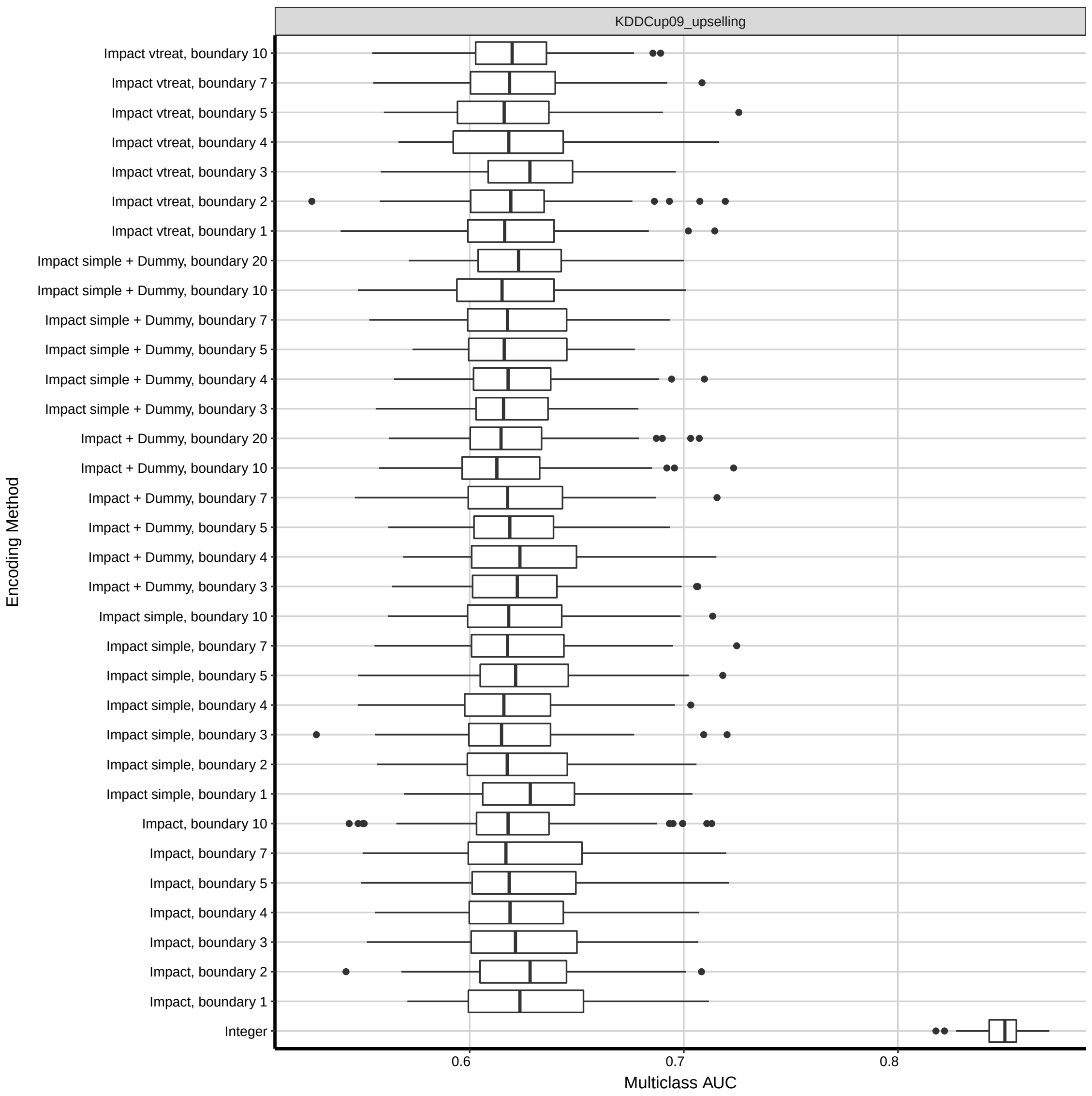


Figure 22: Benchmark results for KDD upselling dataset.

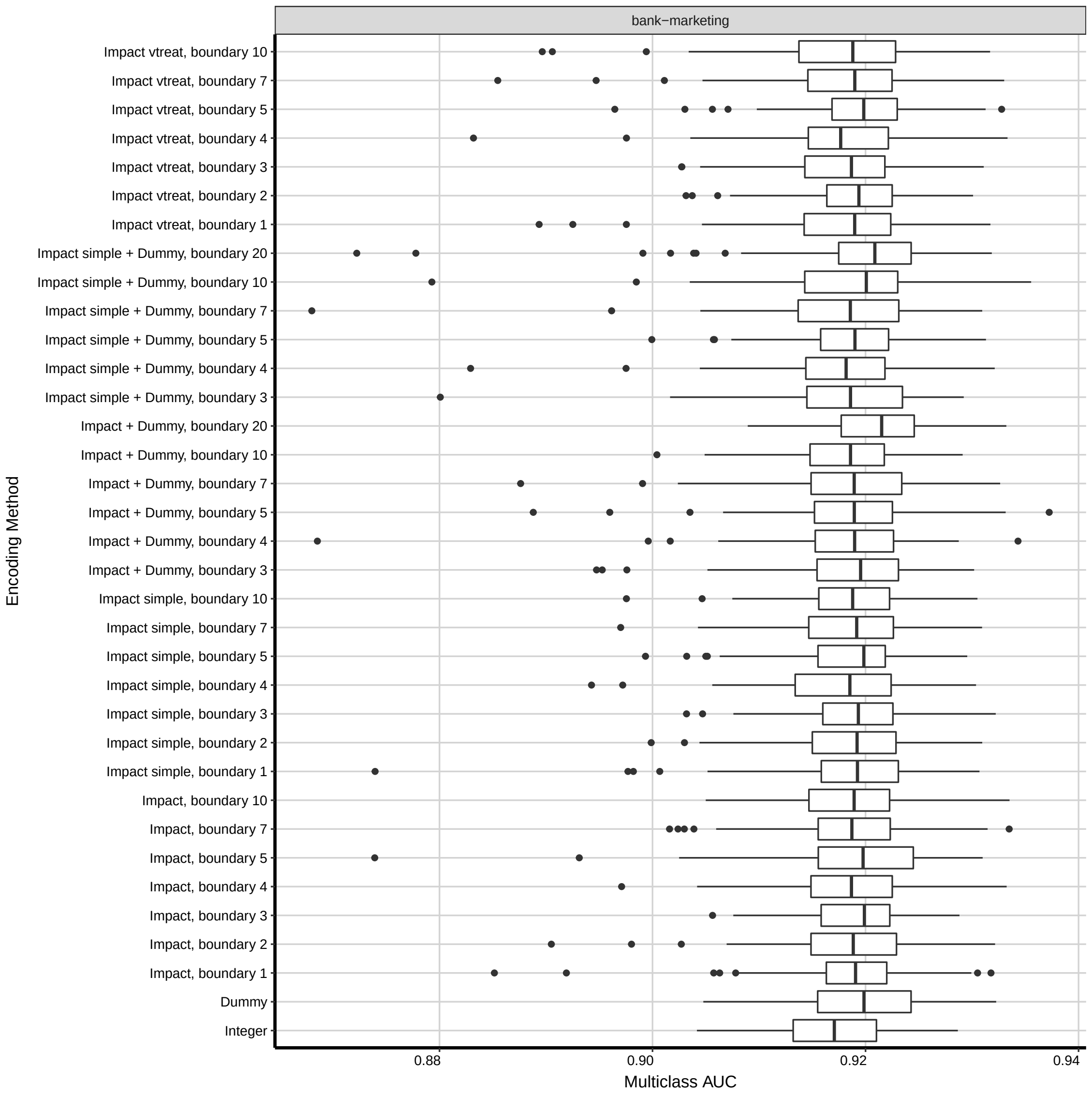


Figure 23: Benchmark results for bank-marketing dataset.

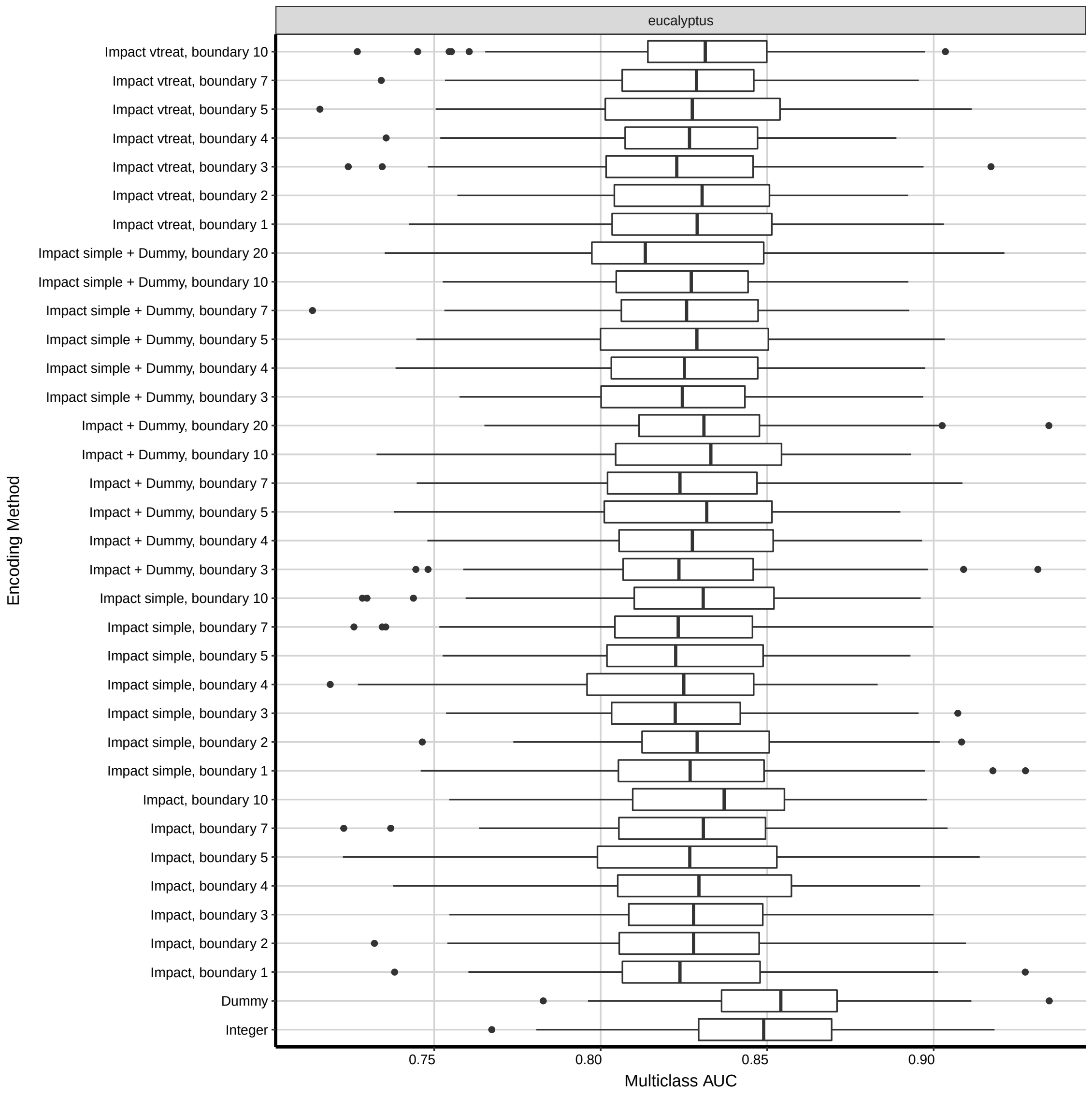


Figure 24: Benchmark results for eucalyptus dataset.

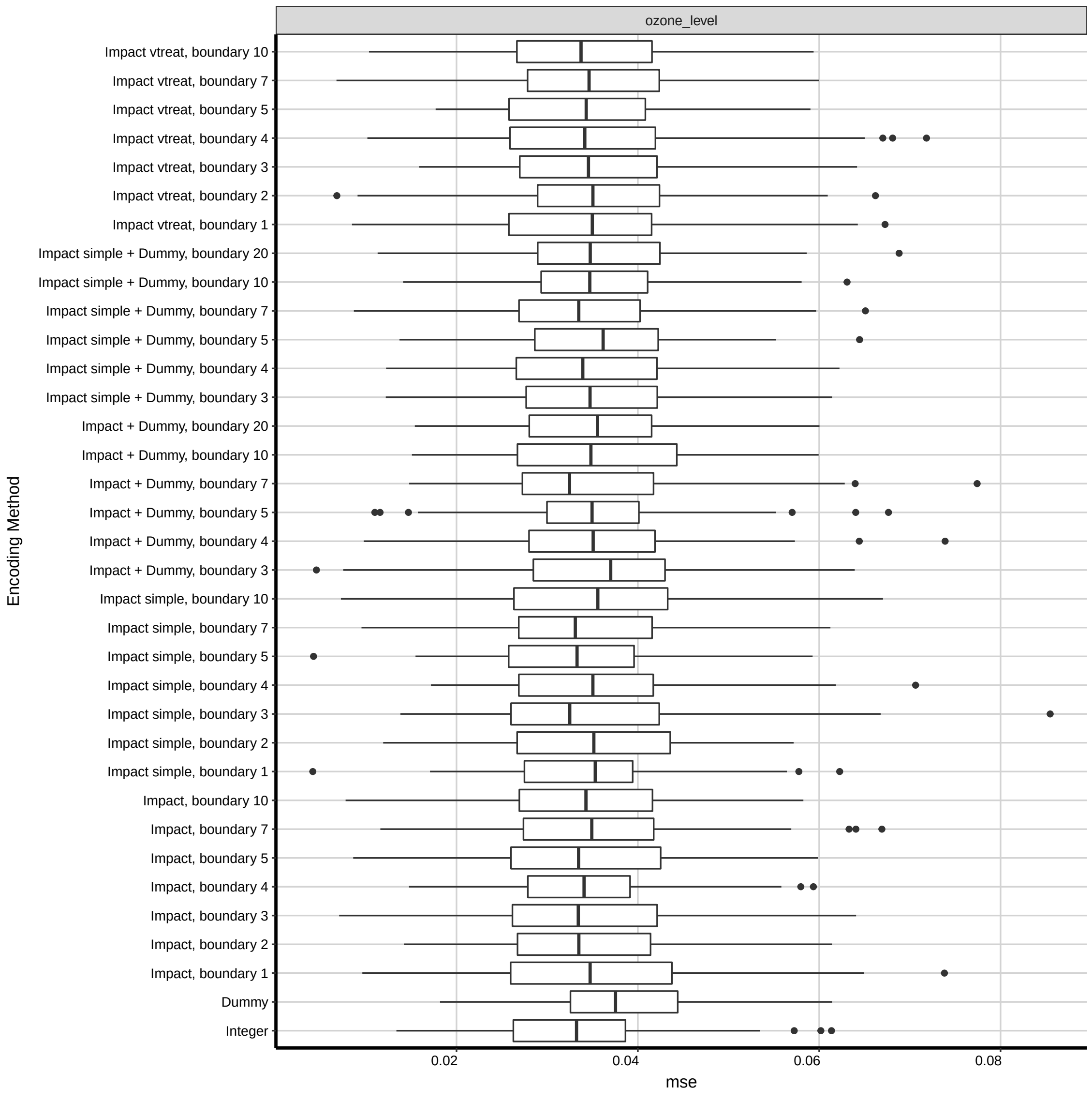


Figure 25: Benchmark results for ozone dataset.

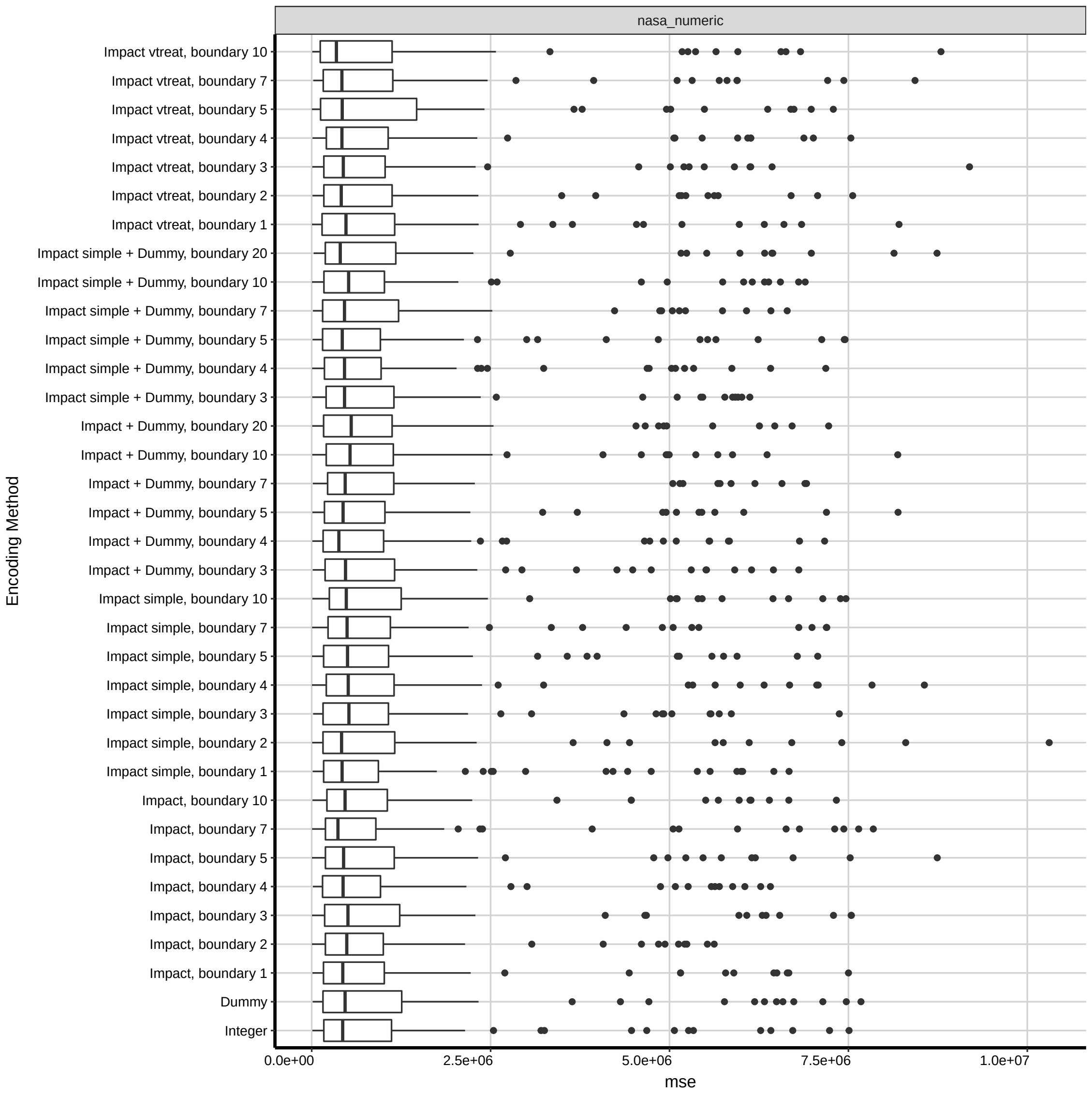


Figure 26: Benchmark results for nasa dataset.

A.2. Additional figures and tables of the GenSA tuning benchmark

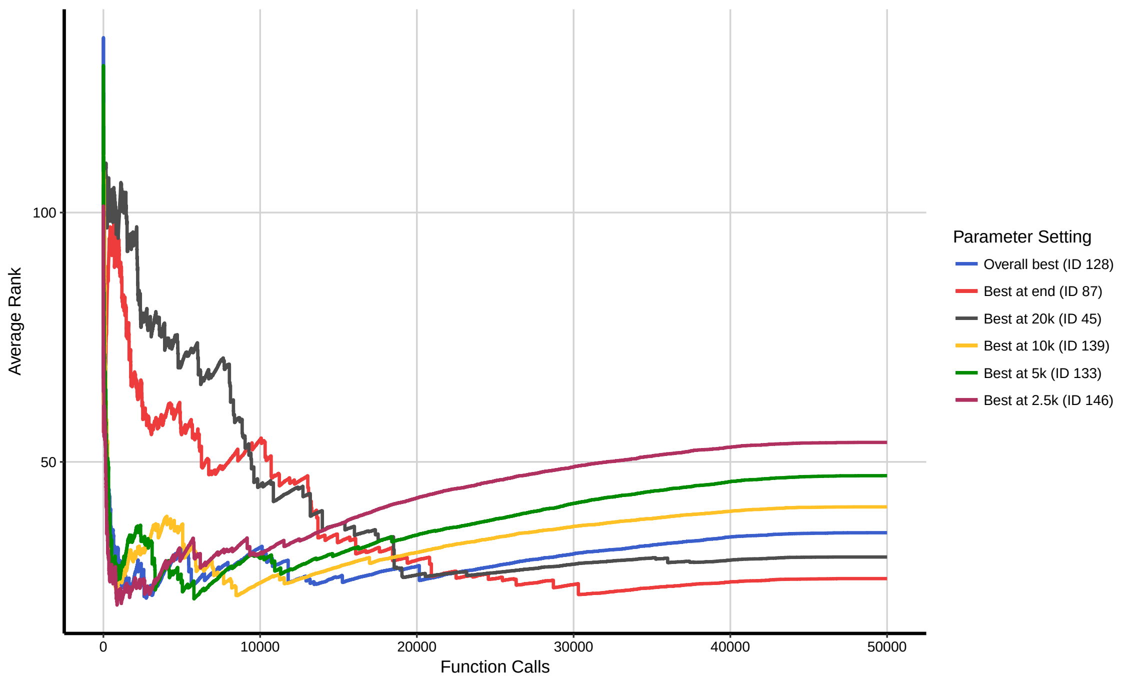


Figure 27: Average ranks of overall best GenSA parameter settings

Best GenSA parameter configurations (out of 480)						
ID	smooth	simple	temp	visit	accept	best at
128	FALSE	TRUE	250	2.3	-12	Overall best
87	FALSE	FALSE	5000	2.9	-9	End
45	FALSE	FALSE	1000	2.5	-9	20k
139	FALSE	TRUE	250	2.7	-15	10k
133	FALSE	TRUE	250	2.5	-15	5k
146	FALSE	TRUE	250	2.9	-12	2.5k

Table 19: Best overall GenSA parameter settings

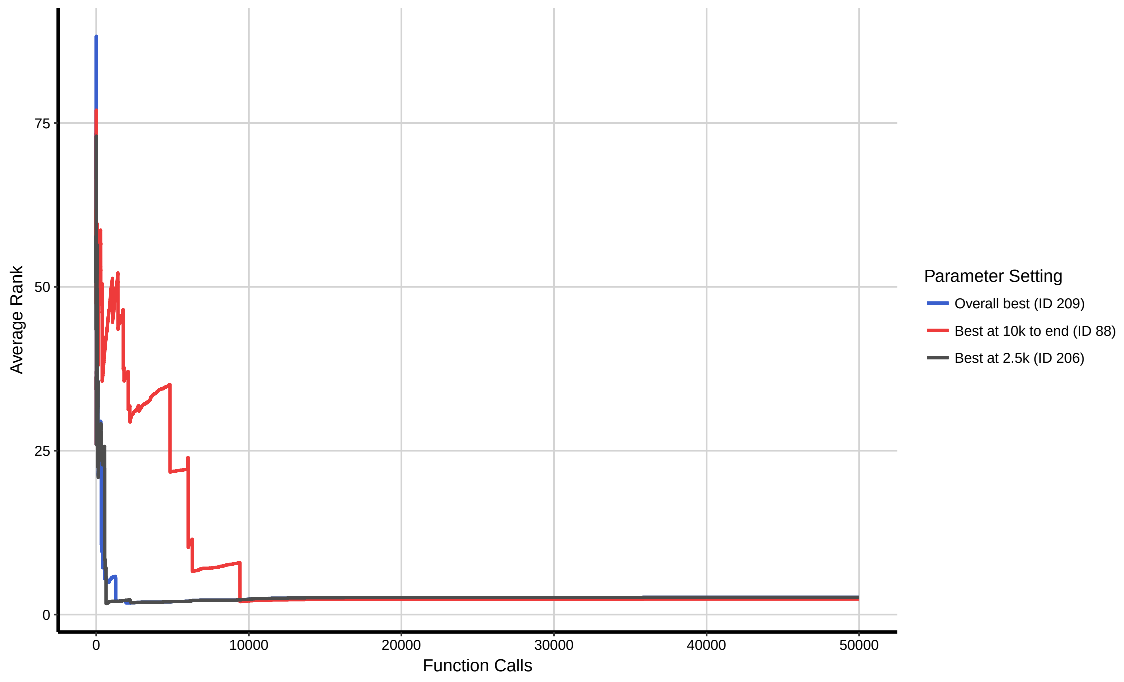


Figure 28: Average ranks of best GenSA parameter settings 3-class-datasets

Best GenSA parameter configurations (out of 480)						
ID	smooth	simple	temp	visit	accept	best at
209	FALSE	TRUE	5000	2.9	-3	Overall
88	FALSE	FALSE	5000	2.9	-6	10k to end
206	FALSE	TRUE	5000	2.9	-12	2.5k

Table 20: Best GenSA parameter settings for datasets with 3 classes

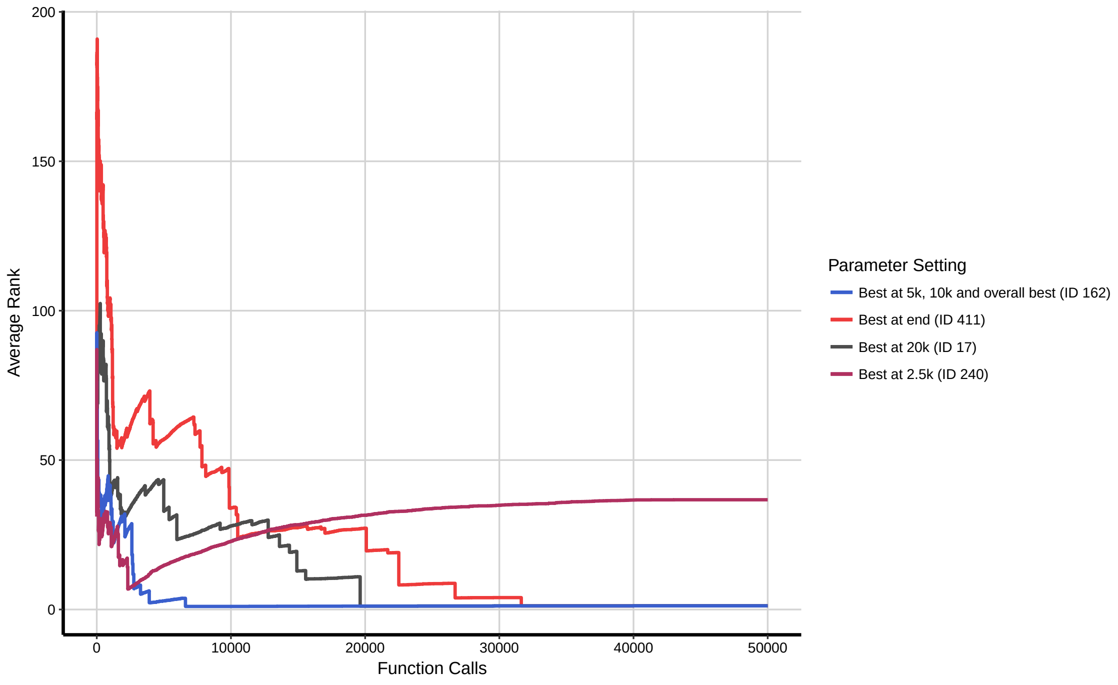


Figure 29: Average ranks of best GenSA parameter settings 4- and 5-class-datasets

Best GenSA parameter configurations (out of 480)						
ID	smooth	simple	temp	visit	accept	best at
411	TRUE	TRUE	1000	2.7	-9	5k, 10k and overall
17	FALSE	FALSE	250	2.5	-3	End
162	FALSE	TRUE	1000	2.3	0	20k
240	FALSE	TRUE	10000	2.9	0	2.5k

Table 21: Best GenSA parameter settings for datasets with 4 and 5 classes

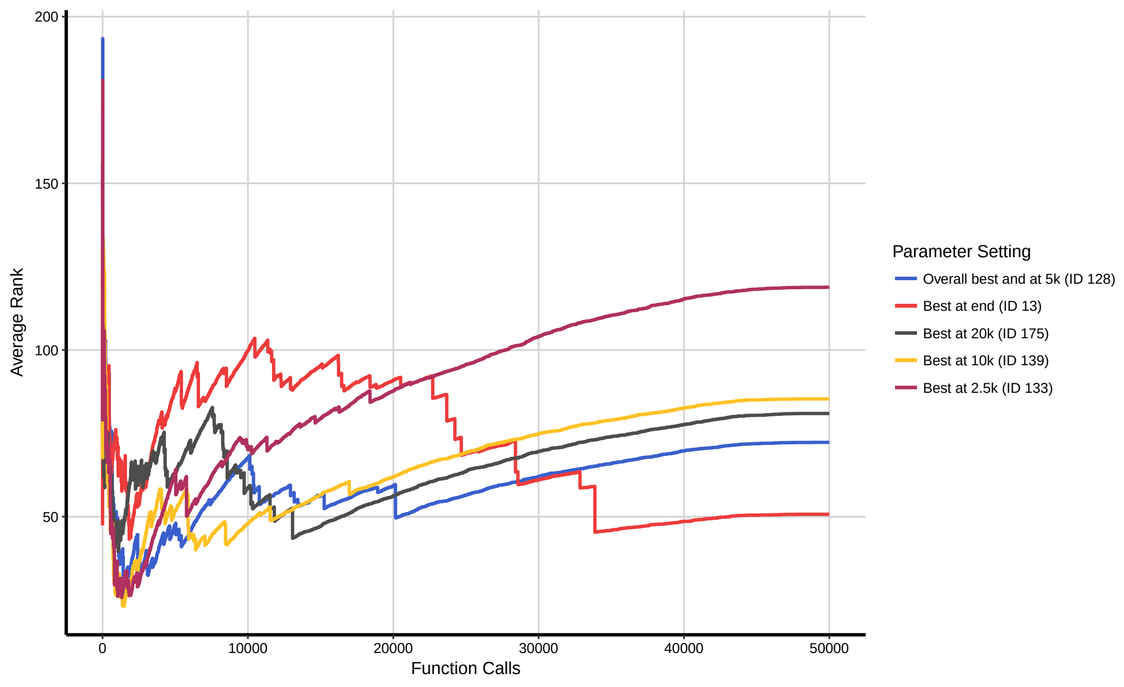


Figure 30: Average ranks of best GenSA parameter settings 8- and 10-class-datasets

Best GenSA parameter configurations (out of 480)						
ID	smooth	simple	temp	visit	accept	best at
128	FALSE	TRUE	250	2.3	-12	5k and overall
13	FALSE	FALSE	250	2.5	-15	End
175	FALSE	TRUE	1000	2.9	-15	20k
139	FALSE	TRUE	250	2.7	-15	10k
133	FALSE	TRUE	250	2.5	-15	2.5k

Table 22: Best GenSA parameter settings for datasets with 8 and 10 classes

A.3. Repositories containing the R-Code of this thesis

autoxgboost
`https://github.com/ja-thomas/autoxgboost`

mlr
`https://github.com/mlr-org/mlr`

autoxgboost benchmark
`https://github.com/ja-thomas/autoxgb_benchmark`

Threshold tuning benchmark
`https://github.com/berndbischl/tune_threshold_benchmark`

GenSA parameter optimization benchmark
`https://github.com/Coorsaa/GenSA_tuning_benchmark`

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Munich, March 13, 2018

signature