Ludwig-Maximilians-Universität München
Faculty for Mathematics, Computer Science and Statistics

Bachelor Thesis

Performance and usability comparison of supervised learning machines

Author: Niklas Klein

Advisor: Prof. Dr. Christian Heumann

8th September 2015

Declaration

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and bibliography.

Munich, 8th September 2015

Niklas Klein

Abstract

Classification and regression are crucial components in statistics. A feasible approach to solve such problems is supervised machine learning. Therefore, the main goal of this thesis is to compare those methods. We also peer into each techniques usability, that is, their individual difficulty of parameter tuning as well as potential capabilities (i.e. limitations with respect to usage). For this purpose, we apply a collection of six algorithms on a set of regression and classification problems. That is to say two differently tuned variations of the random forest and in particular a very recent proposed alteration, called synthetic random forest. Furthermore we employ SAMME, which is an extension of the AdaBoost for multiclass problems. For regression on the other hand, we utilize gradient boosting, a further generalization of boosting. Beside these tree ensembles, we implement support vector machines on both, classification and regression. While these are typical machine learning methods, we want to discover if the lasso, a procedure which has its roots in the statistics, is able to keep up with them. We also record the computational times of each method. The outcome is that each method shows advantages and drawbacks and strongly depend on the individual demands.

Namely, whether we are just interested in the results or do we also want to obtain an interpretable model. Computational time, which fluctuate very heavily, might also affect the decision-making process.

Contents

1 Introduction

Throughout the past few decades and accompanying with the the steady improvements of processing power, data science has become a crucial component in many fields. Authorities and ventures from basically all branches around the world collect data. Their natural intention is to extract information. Since the vast quantity makes it utterly impossible to do this by hand, automating such tasks has become a fundamental goal. Consider for example an insurance company. They would like to classify customers in high and low risk groups. Very likewise concerns affect the day-to-day routine in the banking sector. When it comes to the granting of loans, creditworthiness is essential and has to be determined somehow.

This work is dedicated to analyze supervised learning methods. Those are extremely versatile and rest upon the axiom to train algorithms with known examples (i.e. with training data). For our banking example, the training data could be the loan defaults of prior recipients as well as individual characteristics, such as the income or the family status. Based on that experience, the algorithm tries to detect patterns and draws conclusions. Subsequently, when the so called „training phase“ is over, one can finally use the corresponding model to predict.

A commonly applied technique for such problems are random forests. We will elucidate their functionality and utilize them on a set of regression and classification problems. In particular, we are going to consider a new variation, called synthetic random forests. Their demand is to dominate the classic algorithm in both, handling as well as its predictive capability.

Following up, we introduce an extension of the famous AdaBoost, which was awarded with the Gödel Prize in 2003 (an annual prize for outstanding work in the area of theoretical computer science). It is known that many real-time face detection technologies (for example surveillance cameras or the facebook face recognition feature) apply modifications of the AdaBoost. Hence we will dispose it on a set of classification tasks and look if it lives up its good reputation.

Furthermore, we expand our selection to gradient boosting. As it facilitates the use of different loss functions, we can regard it as a generalization of boosting. One of its real life implementations is for web page ranking (its an open secret that all major search engine companies such as Google or Yahoo apply it). Another interesting, yet completely different exertion was carried out in 2014. Participants of an competition at the CERN had to develop an algorithm to predict the behaviour of Higgs boson particles. One of the best performing algorithms was indeed a variation of gradient boosting.

Random forests and boosting in general too, are tree based methods. A completely different approach to solve regression and classification issues is provided by the support vector machine. In essence, SVMs try to solve an optimizing problem which maximizes the distance between a separating hyperplane and the observations. Beyond that, support

vector machines allow its operator to apply the „kernel-trick“. The idea is to map data into a higher dimension, wherefore partitioning of data points can be greatly simplified. In reality, their area of application appears to be endlessly large. Intrusion detection systems, such as computer firewalls, apply support vector machines to identify malicious activities. Also, image analysis algorithms to classify the facial expression of humans as well as text mining techniques often use variations of the SVM.
A frequent claim of statisticians is, that most machine learning methods lead to „black-box“ predictions. Thus, to draw better conclusions on cause and effect of the variables, we will also fit a classic regression variation called the lasso. Its main intention is to shrink down unnecessary variables. A penalty term, called the L_1 norm, serves as a constraint to execute the particular feature selection. Therefore, our goal is to find out how well it can compete with the introduced machine learning methods.
All these techniques still represent just a small fraction of supervised machine learning tasks. Nevertheless, one might already imagine how difficult the selection of a suitable method turns out in the end. So the main scope of this thesis is to find out how all these algorithms perform on a set of regression and classification problems. For this purpose, we will give in a first step an in-depth explanation, regarding the definition and functionality of all algorithms. Following up, we employ our methods on a set of regression problems. Afterwards, the procedure will repeated on a collection of classification tasks with a slightly modified set of methods. Finally, we draw conclusion and try to find out, if any method was able to stand out.

2 Supervised learning methods

This chapter is dedicated to provide a formal approach to all algorithms. At first we consider tree based ensemble methods whose origins lie more or less in computer science. Following up, we eye upon support vector machines before we finally head to the lasso, which has its roots in the classical statistics.

2.1 Introduction to random forests

Random forests are an ensemble learning method, primarily applied on classification and regression problems. They were first introduced by Leo Breiman (2001) to improve bagged trees. To fully understand the idea behind random forests, we have to take a step back and acquaint ourselves with Classification and Regression Trees, which have also been developed by Breiman et al. (1984).

The generic term CART (Classification and Regression Trees) tags a technique, which is applicable on classification or regression problems. Each CART consists of a root node, which represents the starting point for the problem at hand. Following up, the algorithm searches for the variable, to split that root node into exactly two, as heterogeneous as possible (= decision rule), child nodes. One may consider these first two child nodes as premature groups or classes. For every newly generated node, the entire process is repeated until a terminal node is found.

Consider the set of features V_1 to V_{60}. Each of them represents the level of energy for a sonar signal at different frequencies (i.e. 60 different frequencies). These energy levels result from the collision with objects in the ocean. Our ambition is to predict, whether the object is either a mine (M) or a rock (R) (Gorman & Sejnowski 1988)

Applying the CART algorithm in R (Therneau et al. 2014) and plotting its results with the rpart.plot (Milborrow 2015) as well as the rattle package (Williams et al. 2015) provides us the tree shown in figure 1.

All six terminal nodes indicate the probability for the two classes. Consider for example the node 8 (bottom left). Then, if the energy levels (in that order) of

$$V_{12} \geq 0.22,\ V_{51} \geq 0.012 \ \text{ and } \ V_9 \geq 0.11$$

we obtain a probability of 96% that the observation is a mine (M) (and trivially 4% a rock (R)). The third number (35%) illustrates the portion of observations used for that terminal node.

CART is a very simple algorithm and its main drawback is the vulnerability to (even minor) changes in the structure of the data. That means, the model might fit only for this particular data. Thereby, as we apply the model on other data, we may observe very high variance. Following up inevitably, the predictive performance will be poor (Hastie et al. 2009*a*). Once again Leo Breiman (1996) delivered a feasible approach of how one may lower the variance of statistical learning methods. Simplified, bootstrap aggregation

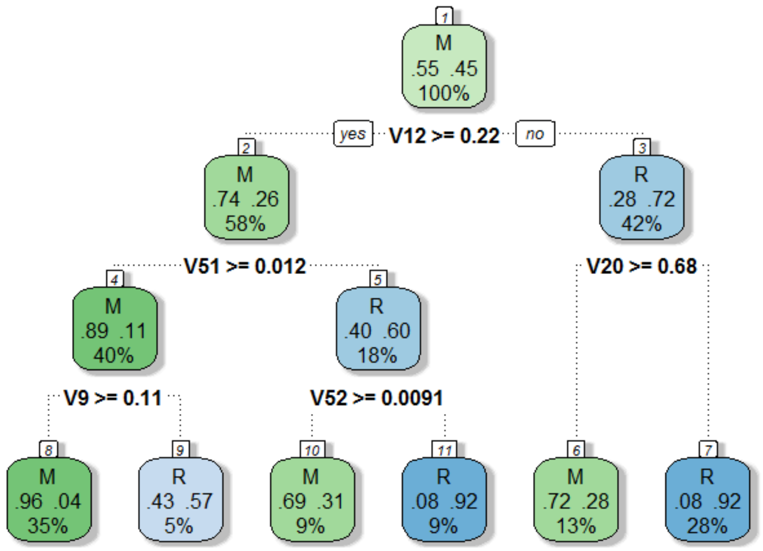

Figure 1: *CART* tree for the sonar data

(or in short bagging) means, that we only take a certain subset of our data and apply the desired method. In terms of trees, one has to draw B bootstrap samples from the training data (each with replacement) and fit a tree to each. To make a prediction for a regression problem, the results of all bootstrap trees are simply averaged. For classification, a majority vote over all B trees determines the class.

Given this randomization in the training data for each tree, we can greatly lower the variance (Hastie et al. 2009*a*) and are henceforth only one step away from random forests.

2.1.1 Classic random forests

Consider B identically distributed (but not independent) trees with positive pairwise correlation ρ. We can write the variance of their average as (Hastie et al. 2009*e*):

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \tag{1}$$

While the right term of equation 1 shrinks away as we increase the number of trees B, the „correlation term“ $\rho\sigma^2$ remains. Consequently, at some point we cannot reduce the variance significantly further by adding more trees into our ensemble.

In order to still obtain lower variance and thus better performance, one has to reduce the correlation between the trees. This effect can be accessed by only picking a random subset $m \leq p$ of the p variables at each node split.

Every single tree is now forced to use different predictors for splitting at all nodes.

Thereby, we now do not only construct bootstrap trees (i.e. trees with different training data), but also more de-correlated ones (Hastie et al. 2009*e*).

Virtually analogous to the bagging approach, the predictor of a random forest for regression is shown in equation 2. Under the assumption, that the bth random forest yielded

Algorithm 1: Random Forest (RF)

```
1. For b = 1 to B:
   (a) Draw a bootstrap sample $\boldsymbol{Z}^*$ of size $N$ from the training data
   (b) Grow a random forest Tree $T_b$ to the bootstrapped data, by
       recursively repeating the following steps for each terminal node
       of the tree, until the minimum node size $n_{min}$ is reached.
       i. Select $m$ variables at random from the $p$ variables.
       ii. Pick the best variable/split-point among the m.
       iii. Split the node into two child nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$
```

$$\hat{f}_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x) \tag{2}$$

$$\hat{C}_{rf}^B(x) = \text{majority vote } \{\hat{C}_b(x)\}_1^B \tag{3}$$

a class prediction $\hat{C}_b(x)$, the classification predictor is illustrated as a majority vote over all trees (equation 3). Algorithm 1 and its corresponding predictors (Hastie et al. 2009*e*) not only give us a brief overview on how random forests are being implemented, but also a first glimpse on their tuning parameters.

Number of trees B**:**

According to Hastie et al. (2009*e*), it is not possible to overfit random forests due to the quantity of trees grown. Therefore, as long as B is reasonably high, i.e. 500 or more, we can likely ignore that parameter. In addition we will hardly profit from growing more trees (see equation 1). To verify these claims graphically, we used 2/3 of the Sonar data to train and the remaining 1/3 to test. Figure 2 shows us the resulting error rates. As the test error and in particular the training error level off after approximately 500 trees, there is no evidence for overfitting. Secondly, we see that adding more trees into the ensemble hardly leads to a better result.

Node size:

The node size describes the minimum size of a terminal node (in terms of observations). Hence, a small value will lead to very bushy trees, which is very common for random forests. Segal (2004) claims, that tuning the node size has only little impact on performance gains. Unlike that, Ishwaran & Malley (2014) allege that an „optimal tuning of node size has the greatest potential to improve prediction performance“. However, we will scrutinize that issue in Chapter 3 ourselves.

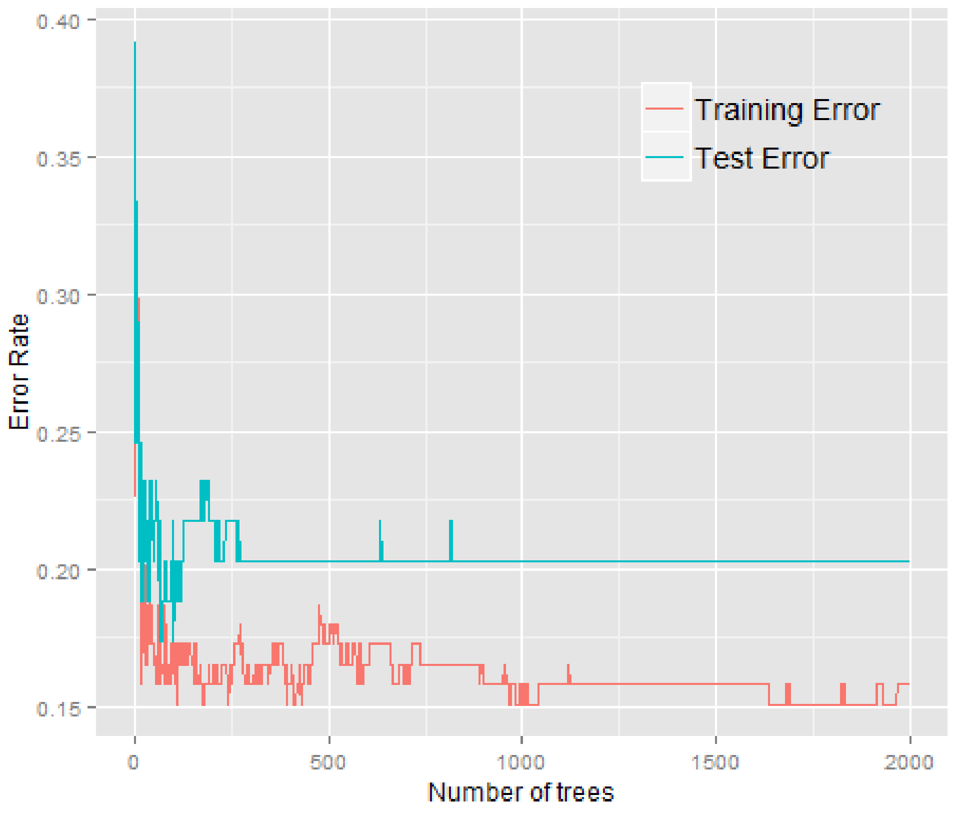

Figure 2: *Random Forest: Training Error and Test Error for 2000 trees* (Sonar)

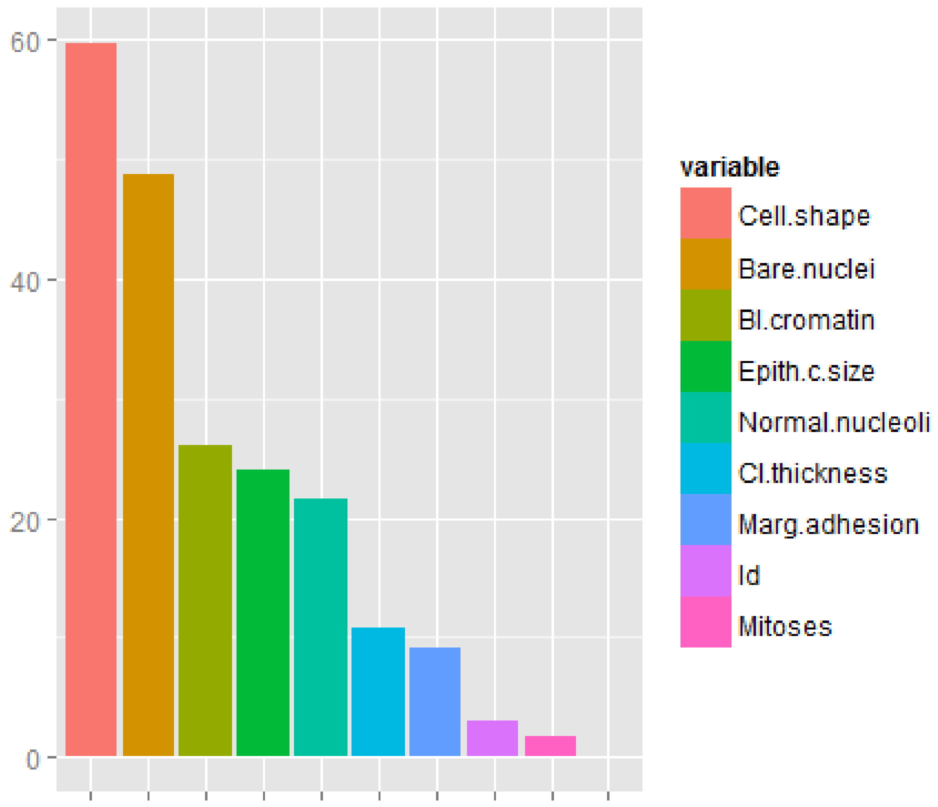

Figure 3: *Random Forest: Variable importance plot. The higher the mean decrease in Gini index (y-axis), the more important the variable.* (Breast Cancer)

Variables to consider for each node split m:
For the third parameter m (i.e. the „de-correlation parameter“), Breiman (2001) suggested $p/3$ for regression and $\sqrt{p}$ for classification problems.
Anyhow, there is a general consensus that tuning random forests is quite simple. Furthermore, they appear to be very robust with regard to overfitting (Hastie et al. 2009*e*).

Statisticians in general want to find the cause for an occurring effect. Unlike in linear regression, we have no classical parameters with standard errors. So, in order to still testify or measure the relevance of variables in random forests (i.e. for all tree based methods), we access the „variable importance“. Figure 3 shows the corresponding results for the Breast Cancer data (chapter 4.1.1). In classification, we record the mean decrease in Gini index (see Hastie et al. (2009*e*)). For a regression problem, we chart the total drop in RSS (residual sum of squares) resulting from the variable splits, averaged over all B trees. In both cases, a large value indicates an important variable. Anyhow, while we still cannot draw quantitative conclusions, random forests do at least not result in complete black box prediction.

2.1.2 Synthetic random forests

A slight, yet interesting modification to random forests has been proposed recently by Ishwaran & Malley (2014). In order to accomplish even less necessity for parameter tuning, a synthetic random forest (SRF) computes in a pre step new synthetic features and adds them into the original data set. That preparative procedure is followed up by the classic random forest.

```
Algorithm 2: Synthetic Random Forest (SRF)

1. Choose a set of node size values $\mathcal{N} = \{n_1, n_2, ..., n_D\}$.
2. Fit a random forest with $node\ size = n_j\ for\ j = 1, ..., D$. Use the same
   $B$ (number of trees) and $m$ (Variables to consider for each node split)
   values of each forest. Denote the resulting forests by $RF_1, ..., RF_D$.
3. Calculate the predicted value for each random forest $RF_j, j = 1, ..., D$.
   We call the predicted value the synthetic feature.
4. Fit a random forest for features both the newly created synthetic
   features and the original p features (using the same
   $B$ and $m$ value as before). We call this the synthetic RF.
```

Node size sequence $\mathcal{N}$:
Supplementary to the parameters we have already come to know in Chapter 2.1.1, we obtain an additional sequence for node size candidates $(n_1, n_2..., n_D)$. For a regression problem, such a synthetic feature would be the predicted value for each node size candidate. In classification, we obtain the predicted probability for each class. More general,

in multiclass problems with K classes, a K-dimensional feature. To avoid overfitting, synthetic features are constructed by out-of-bag predictions.

Consider the sonar data example. Applying algorithm 2 would result in D additional synthetic features. Thus, the new data set consists of 60 original, plus D new synthetic features. Following up, the classic random forest based on the new data will be computed.

2.2 Boosting

The basic assumption for every boosting approach goes back on the early work of Schapire (1990), where it was shown that combining a set of weak learners can generate a single strong one. Weak learners are relatively poor performing machines, that do only slightly better than random guessing (i.e. $P(error) < 0.5$). In case of boosting trees, such a weak learner could be a primitive stump (i.e. a tree with only two terminal nodes). Simplified, what boosting does is adding those weak learners sequentially into the ensemble. The crux is, that boosting modifies the structure or shape of the training data, every time such a weak learner is computed. By contrast, a random forests leaves the data untouched.
Similarly as it holds for bagging, boosting techniques can be applied on many other statistical methods beside trees as well. A crucial and important difference between bagging and boosting is their main ambition. Whereas bagging algorithms want to reduce the variance, boosting mainly aims on lowering the bias (Hastie 2003).

2.2.1 Multi-class AdaBoost

The first and probably most famous boosting algorithm is called AdaBoost by Freund & Schapire (1995). As the classic algorithm is more or less limited to binary problems (one could compute multiple two-class problems) it would not suffice our demands properly.
Hence, as we will study some multiclass problems later, we are going to employ a newer generalization of the AdaBoost.
First introduced by Zhu et al. (2006), stagewise additive modeling using a multi-class exponential loss function (or in short SAMME) applies just some small, yet crucial modifications to the original algorithm. In multiclass problems, it might occur that the error of a weak classifier $T^{(m)}$ is greater than 0.5. But in AdaBoost, the error of each classifier $T^{(m)}$ has to be smaller than 0.5. Elsewise, the $\alpha^{(m)}$ which is defined as

$$\alpha^{(m)} = log\frac{1 - err^{(m)}}{err^{(m)}} \tag{4}$$

and crucial for reweighting the data becomes negative. The data would then be reweighted into the wrong direction. An in-depth implementation and how Zhu et al. (2009) fixed that issue is described in algorithm 3.
In order to apply SAMME, we start off by fitting a simple classifier $T^{(1)}$ to the data (2a) and compute its error rate err^m (2b). Depending on that, a value $\alpha^{(1)}$ arises (2c). At this

step we can see the decisive difference to equation 4. The α is now computed as

$$\alpha^{(m)} = log\frac{1 - err^{(m)}}{err^{(m)}} + log(K - 1) \tag{5}$$

whereas the rear term solves the issue of negativity. Note that for a binary problem ($K = 2$), the algorithm reduces itself to the classic AdaBoost. Henceforward, α is being used in (2d) to compute new weights for the data (i.e. boost the data). Following up, the whole process is repeated while the next classifier $T^{(2)}$ will put more focus on „areas“ where a bad predictive performance was observed. Finally after M iterations, we combine all weak classifiers to a single „strong“ one.

Algorithm 3: SAMME

```
1. Initialize the observation weights $w_i = 1/n,\ i = 1, 2, ..., n$.
2. For $m = 1$ to $M$:
   (a) Fit a classifier $T^{(m)}(x)$ to the training data using weights $w_i$.
   (b) Compute
         $err^m = \sum_{i=1}^{n} w_i \mathbb{1}\Big(c_i \neq T^{(m)}(x_i)\Big) / \sum_{i=1}^{n} w_i.$
   (c) Compute
         $\alpha^{(m)} = log\frac{1-err^{(m)}}{err^{(m)}} + log(K - 1)$
   (d) Set
         $w_i \leftarrow w_i \cdot exp\Big(\alpha^{(m)} * \mathbb{1}(c_i \neq T^{(m)}(x_i)\Big),$
         for $i = 1, ..., n$
   (e) Re-normalize $w_i$
3. Output
         $C(x) = \arg\max_k \sum_{m=1}^{M} \alpha^{(m)} * \mathbb{1}\Big(T^{(m)}(x) = k\Big)$
```

Number of iterations M:
This step is very related to the number of trees B of a random forest. But unlike them, boosting can overfit because of adding to many classifiers $T^{(m)}$ into the ensemble. Bühlmann & Hothorn (2007) state that finding a good stopping iteration is the most important tuning parameter for the classic AdaBoost. Hastie (2003) claims that for noisy problems in particular, one has to carefully tune the stopping iteration, as overfitting is an imminent danger. We used the „mlbench“ package (Leisch & Dimitriadou 2010) to create such a noisy problem. As for the Sonar case, 2/3 of the data were used to train and hence, 1/3 to test. Indeed, figure 4 exhibits suspicious signs of overfitting, as the the

test error starts increasing at roughly 60 trees. Since this relation holds for all boosting algorithms, they seem to be more difficult to tune than random forests.

Tree depth d:
Suppose we would like to boost trees. Following up, one has to find a good value d for their maximum depth. This parameter is comparable to the node size of a random forest and therefore controls for the size of the tree. Hastie & Tibshirani (2014) argue that $d = 1$ (i.e. stumps) oftentimes work very well but one might also try other values, such as $d = 2, 4$ or 8. Moreover, deeper trees will generally lead to a lower bias, but accompanying, also a greater variance (i.e. variance-bias trade-off).

For the Sonar data, we boosted such stumps and obtained very good results (figure 5). Moreover, we observe a notable characteristic of the AdaBoost. The test error decreases even though the training error hits zero at approximately 30 trees. Schapire et al. (1998) show, that this behaviour results from the decision boundary, which can still be improved by further boosting iterations.
However, Breiman (1996) rated the classic AdaBoost when used on trees as the „best off-the-shelf classifier in the world“. We will try to find out, if that accounts as well for the generalized version (chapter 4).

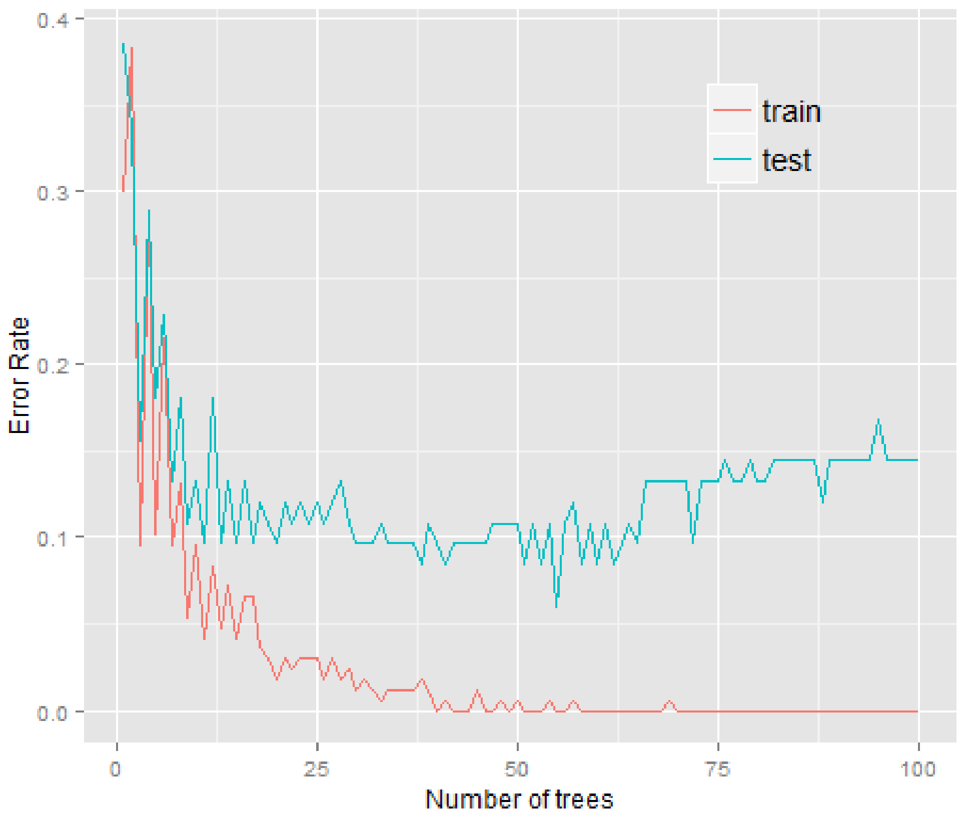

Figure 4: *SAMME starts overfitting after roughly 60 trees, as the test error begins to increase* (mlbench Smiley function with lots of noise)

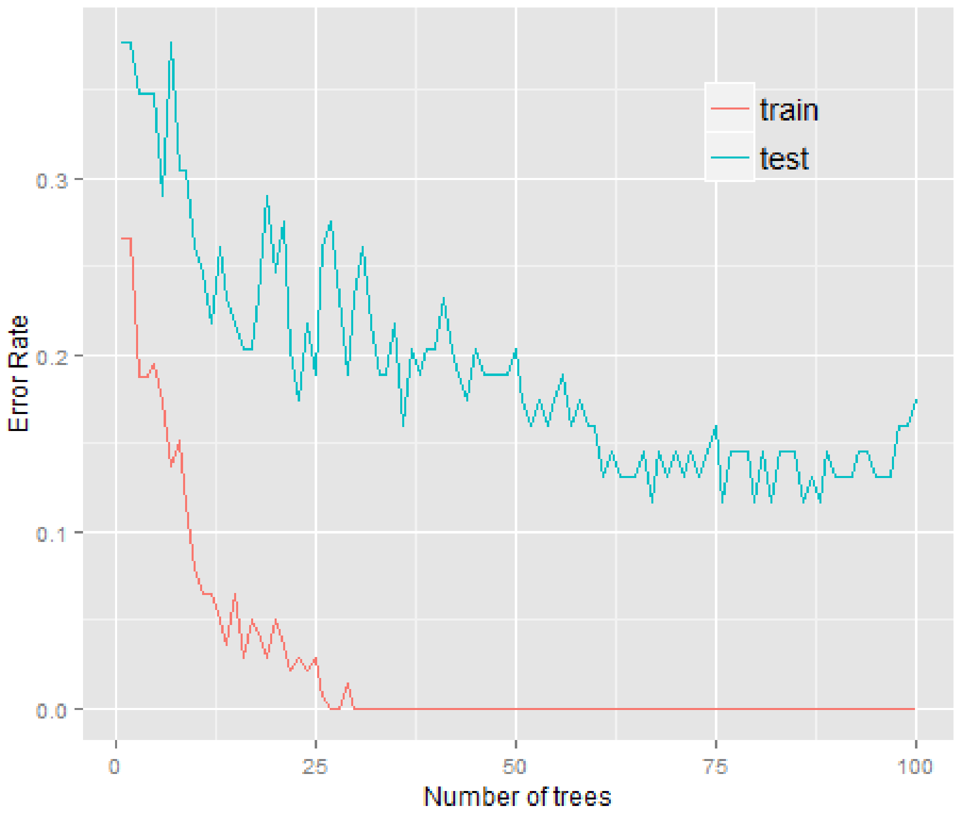

Figure 5: *SAMME: Training error and test error after 100 trees. Note that test error decrease even after training error hits zero.* (Sonar data)

2.2.2 Gradient boosting

As in the case of AdaBoost (or SAMME), gradient boosting (Friedman 2001) is an ensemble learning method, which adds many weak learners step-wisely into a prediction model. Furthermore, gradient boosting is a generalization of boosting, as it allows the use of an arbitrary loss function L. The goal is to minimize the expected value of that loss function:

$$F^* = \arg\min_f \mathbb{E}_{Y,X}[L(y, f(x^T))] \tag{6}$$

where y is the usual response variable, x^T a vector of features and F the prediction function. The major difference compared to AdaBoost can be understood very easily with an regression example. Consider a continuous response y and a single variable x. Applying gradient boosting on that problem, we could chose a stumps as our weak learner and fit it on the data. After that first step, AdaBoost would re-weight the data (see chapter 2.2.1) before the next learner is being computed. By contrast, in gradient boosting we repeatedly fit a weak learner on the error of the previous one. That is to say for regression the residuals (i.e. the negative gradients of the loss function: „squared residuals“ (Kneib & Hothorn (2008))). Figure 6 shows the procedure for the first three iterations. The left plot shows the original data points and tree 1 the corresponding fit. Tree 2 does now fit the residuals of tree 1. Thus, tree 3 fits the residuals of tree 2 and so on.

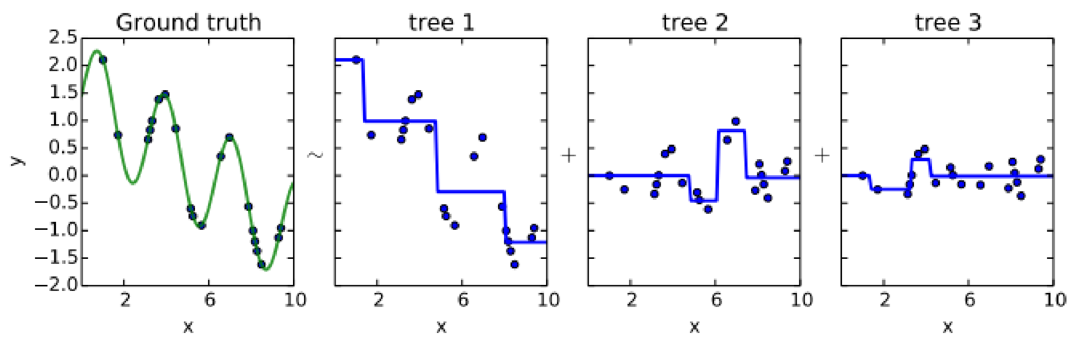

Figure 6: *Gradient boosting regression trees: tree 1 shows the fit on the original data, tree 2 the fit on the error (residuals) on tree 1 and so on (Prettenhofer 2014)*

Shrinkage parameter ν:

In addition to „number of iterations M" and „tree depth d" , in gradient boosting, we receive a third tuning parameter. Before a weak learner is added into the ensemble, we shrink it down by a parameter ν. That value is usually between 0.1 and 0.01 and slows down the rate of overfitting (Hastie et al. 2009*b*). On a side note: there are versions of the AdaBoost which also employ shrinkage.

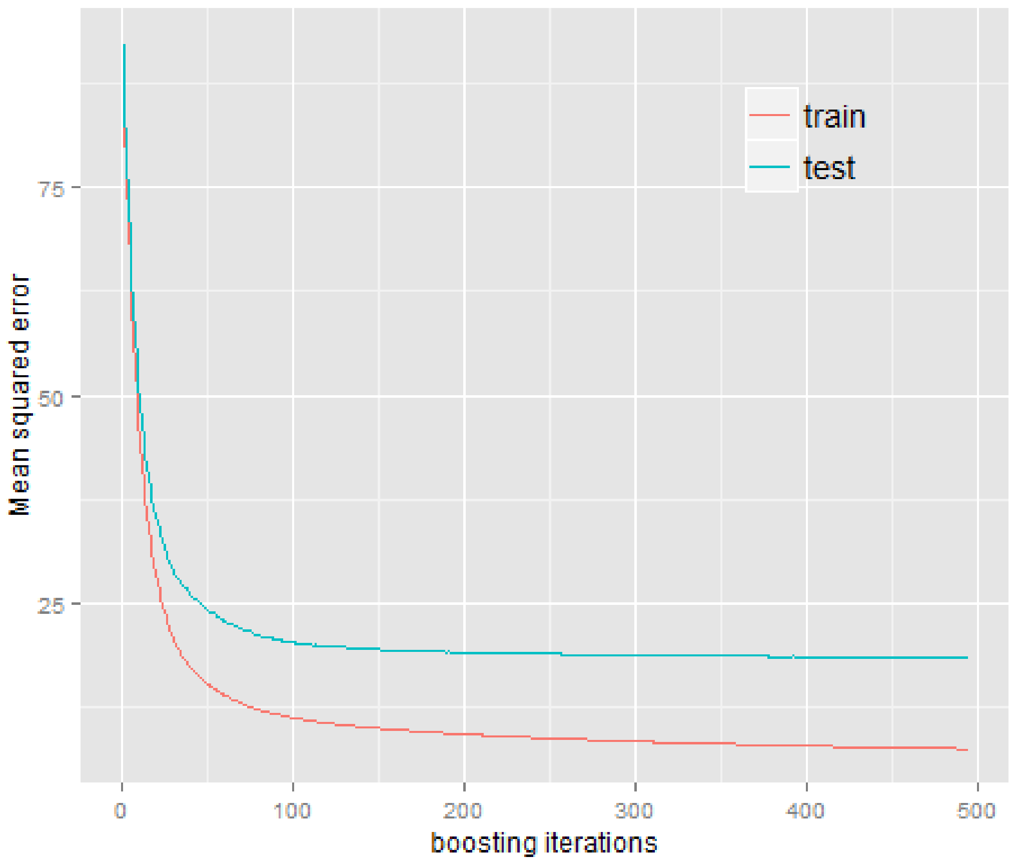

Figure 7: *Gradient boosting: Training error and test error after 500 trees. Even though the slope is very small, the test error is still improving (without overfitting) as a result of the shrinkage.* (Boston Housing data)

Figure 7 shows the training and test mean squared error results for one of the regression problems, we explore in chapter 3 (Boston Housing). We experience two characteristics, which can be explained by the shrinkage. That is to say the smoothness of the curve and the huge number of iterations compared to the AdaBoost (SAMME). Hofner et al. (2014) propose algorithm 4 to implement gradient boosting in a component-wise fashion. As a result from the additive structure in (2d) we can consider the final model as a generalized additive model (Hastie & Tibshirani 1986).

Algorithm 4: Component-wise gradient boosting

```
1. Initialize the function $\hat{F}^{(0)}$ and propose a set of $N$ weak learners
2. For $m = 1$ to $M$:
    (a) Compute the negative gradient $-\frac{\partial L}{\partial F}$ of the loss function and
        evaluate it at $\hat{F}^{(m-1)}(x_i^T), i = 1, ...n$.
        This yields the negative gradient vector:
        $u^{(m)} = \Big(u_i^{(m)}\Big)_{i=1,\ldots,n} = \Big(-\frac{\partial}{\partial F}L(y_i, \hat{F}^{(m-1)}(x_i^T))\Big)_{i=1,\ldots,n}$
    (b) Fit each of the $N$ weak learners to the negative gradient vector.
    (c) Select the best weak learner for $u^{(m)}$ according to the residual
        sum squares (RSS) criterion and set $\hat{u}^{(m)}$ equal to the fitted values
    (d) Update the estimation function:
        $\hat{F}^{(m)} = \hat{F}^{(m-1)} + \nu\hat{u}^{(m)}$, where $0 < \nu \leq 1$ is a shrinkage factor.
3. Iterate step 2 until the stopping iteration $m_{stop}$ is reached.
```

2.3 Support Vector Machine

Unlike random forests or boosting, support vector machines (Cortes & Vapnik 1995) are no ensemble learners. They act more like an optimization algorithm, solving a specific equation to assign classes.

2.3.1 Classification

In fact, the idea of separation by so-called hyperplanes is generalized by the SVM. That means, we can solve problems where no perfect linear partitioning is possible (i.e. overlapping classes). We want to illustrate the idea in figure 8. Consider a set of training data points $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, whereas the x_i represent the observations and y_i the associated classes (here visualized as red and green dots). To execute a classification between them, one has to compute a hyperplane (the blue line in the middle):

$$\{x : f(x) = x^T\beta + \beta_0 = 0\} \tag{7}$$

Furthermore, a goal is to obtain as much safety space as possible. This is crucial, since we assume the same distribution for the unknown data points (i.e. those we want to predict later) as for those we do know (the training data). Therefore, the demand is to maximize the space between the hyperplane and the margin (i.e. the dotted lines).

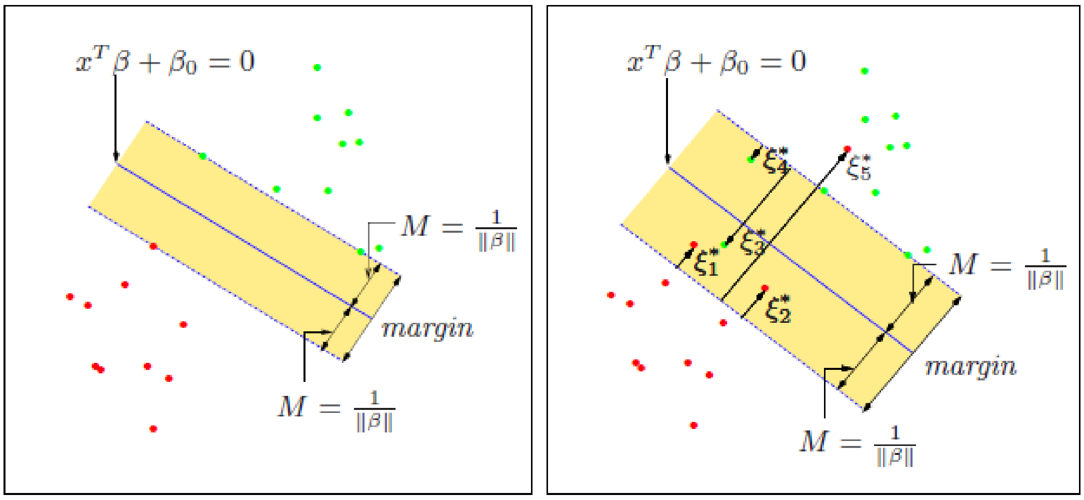

Figure 8: *Support vector classifier:* The left plot shows perfectly separable classes (red and green dots). The right one exhibits the overlapping case, whereas points labeled as ξ_j are those who are on the wrong side of their margin by a distance M. (Hastie et al. 2009*f*)

Following up, as we allow observations to overlap the margins, we have to penalize them in a certain manner. These are called slack variables and as we see later, generate the actual trade-off for our optimization problem. Assuming that the target classes are labeled with „−1" (red) and „+1", (green) we can write down the optimization problem

$$\min_{\beta,\beta_0} \frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N}\xi_i$$

$$\text{subject to} \quad \xi_i \geq 0 \quad \text{and} \quad y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i \; \forall i \tag{8}$$

Cost parameter C:
The cost parameter C is multiplied with each slack variables distance to their correct margin. Therefore, it controls for the size of it. Large values for C will lead to small margins (i.e. „strong" penalty for misclassification). Therefore, we might experience overfitting. Trivially, a small cost will result in a bigger margin and sometimes even in underfitting.

To solve equation 8 with its corresponding constraints, we have to apply the Lagrange function and use the Karush-Kuhn-Tucker conditions (see Hastie et al. (2009*f*)). The resulting solution for β

$$\beta = \sum_{i=1}^{N}\alpha_i y_i x_i \tag{9}$$

assumes non-zero coefficients α_i, for all observations i, where the constraints of equation

8 are exactly met. Hence, the β does only depend on those observations lying inside or on the edge of the margin. These observations are called the support vectors. On a side note: α depends on the cost parameter C and is obtained by minimizing the Lagrangian function. Consequently, to obtain now the missing β_0, one has to simply solve equation 7. Following up, we are finally able to classify observations according to the linear relationship:

$$G(x) = \text{sgn}[f(x)] = \text{sgn}[x^T\beta + \beta_0] \tag{10}$$

It turns out, that computing linear boundaries yields better results than nonlinear ones. Thus, if we experience complicated structure in the data, we have to map our observations into a higher dimension. That technique is oftentimes denoted as the „Kernel-trick". It can be shown, that the hyperplane for a specific set of input feature vectors $h(x_i)$, can be written as

$$\begin{aligned} f(x) &= h(x)^T\beta + \beta_0 \\ &= \sum_{i=1}^{N} \alpha_i y_i \langle h(x), h(x_i) \rangle + \beta_0 \end{aligned} \tag{11}$$

We see that $f(x)$ depends on the inner product of $h(x)$. This is mandatory to apply a Kernel function. Latter one is able to calculate inner products for higher dimensions, without actually mapping the data. A generalized notation of the Kernel function is shown in equation 12.

$$K(x,y) = \langle h(x), h(y) \rangle \tag{12}$$

Consider the strongly simplified example shown in figure 9. The three dots on the left show a one-dimensional problem. It is impossible to draw a linear decision boundary. The Kernel-trick allows us to compute the inner product for a higher dimension and hence, to execute a linear separation.

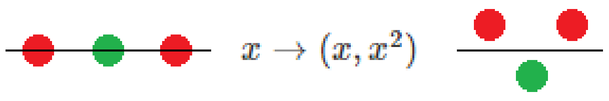

Figure 9: *Decision boundaries for red and green class: Left side: One dimension $\Rrightarrow$ no linear separation possible. Right side: mapped into two dimensions $\Rrightarrow$ linear separation easily attainable.*

A very popular kernel function is called the radial basis kernel (RBK). It solves the inner product in an infinite dimensional feature space.

$$K(x,y) = exp(-\gamma||x-y||^2) \tag{13}$$

RBK parameter γ:
Applying the RBK, a second tuning parameter for the SVM arises. Without going to deep into the mechanics of the radial basis kernel, the γ controls for the standard deviation of the Gaussian. Thus, a large value for γ means small variance around each observation and vice versa. That adjustability henceforthly takes influence on the smoothness of the decision boundary (Hastie et al. 2009*f*).

2.3.2 Regression

To apply a support vector machine on regression problems, one has to adjust a few of its properties. Consider the linear regression model: $f(x) = x^T\beta + \beta_0$. To estimate β, we have to minimize (Hastie et al. 2009*f*):

$$H(\beta, \beta_0) = \sum_{i=1}^{N} V(y_i - f(x_i)) + \frac{\lambda}{2}||\beta||^2 \tag{14}$$

where V acts as an so called „error measure“. A common variant for V is the „ϵ-incentive“ loss-function (equation 15).

$$V_\epsilon(r) = \begin{cases} 0 & \text{if } |r| < \epsilon \\ |r| - \epsilon, & else \end{cases} \tag{15}$$

Only data points lying in between the borders of the margin are being taken into consideration (see figure 10). This idea is almost identical to the support vectors in chapter 2.3.1 For regression, such points are those with small residuals. To solve equation 15, we apply almost the same procedure as in chapter 2.3.1. Thereby we can show, that the resulting solution has the form:

$$\begin{aligned} \hat{\beta} &= \sum_{i=1}^{N} (\hat{\alpha}_i^* - \hat{\alpha}_i) x_i, \\ \hat{f}(x) &= \sum_{i=1}^{N} (\hat{\alpha}_i^* - \hat{\alpha}_i) \langle x, x_i \rangle + \beta_0 \end{aligned} \tag{16}$$

Similarly as for the classification case: for the input values x, the solution depends only on their inner product (equation 16). Hence, we can apply kernel functions to access higher dimensions as well for regression problems.

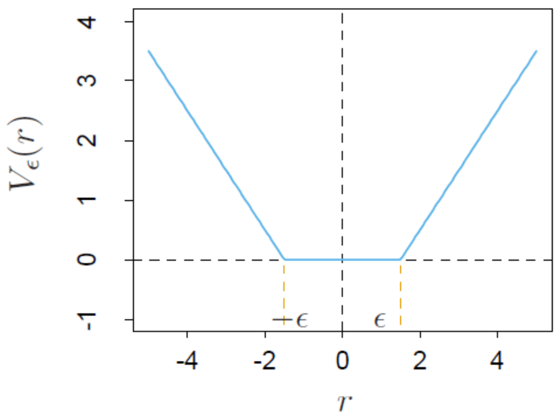

Figure 10: *ϵ-incentive error function: Errors of size $|r| < \epsilon$ will be ignored* (Hastie et al. 2009*f*)

2.4 Lasso

The last method we would like to acquaint ourselves is called least absolute shrinkage and selection operator (lasso). First introduced by Tibshirani (1994), the intention is to penalize the absolute size of regression coefficients by the use of an L_1 norm. For linear regression, its estimator is defined by (Hastie et al. 2009*b*):

$$\hat{\beta}^{lasso} = \underbrace{\arg\min_\beta \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2}_{\text{Residual Sum of Squares}}$$

$$\text{subject to } \lambda \underbrace{\sum_{j=1}^{p} |\beta_j|}_{L_1 \text{ norm}} \leq t \tag{17}$$

What happens when we compute equation 17 is that coefficients which are not important enough, will be shrunken down towards zero (or even to zero). Thus, one can regard the lasso as a feature selection algorithm. To better understand its strength and functionality, we consider a fictitious data set. Suppose a regression problem, with $n = 100$ observations and in relation, a relatively huge set of $p = 50$ features. Five of them are strongly correlated with the response, while the remaining 45 hardly matter.
Since λ controls the the L_1 norm, it is crucial to find a good value. Figure 11 displays cross validated values for a sequence of lambda values to minimize the mean squared error. On the other hand, figure 12 shows a curve for each variable. We see the path for their corresponding coefficients against the log lambda value. Those who are above the axis, indicate non-zero coefficients. Obviously, the five prominent upper curves are those from the strongly correlated variables. Furthermore, the vertical line was drawn to indicate the optimal log lambda value from the cross validation done in figure 11 and thus, represents the final model.

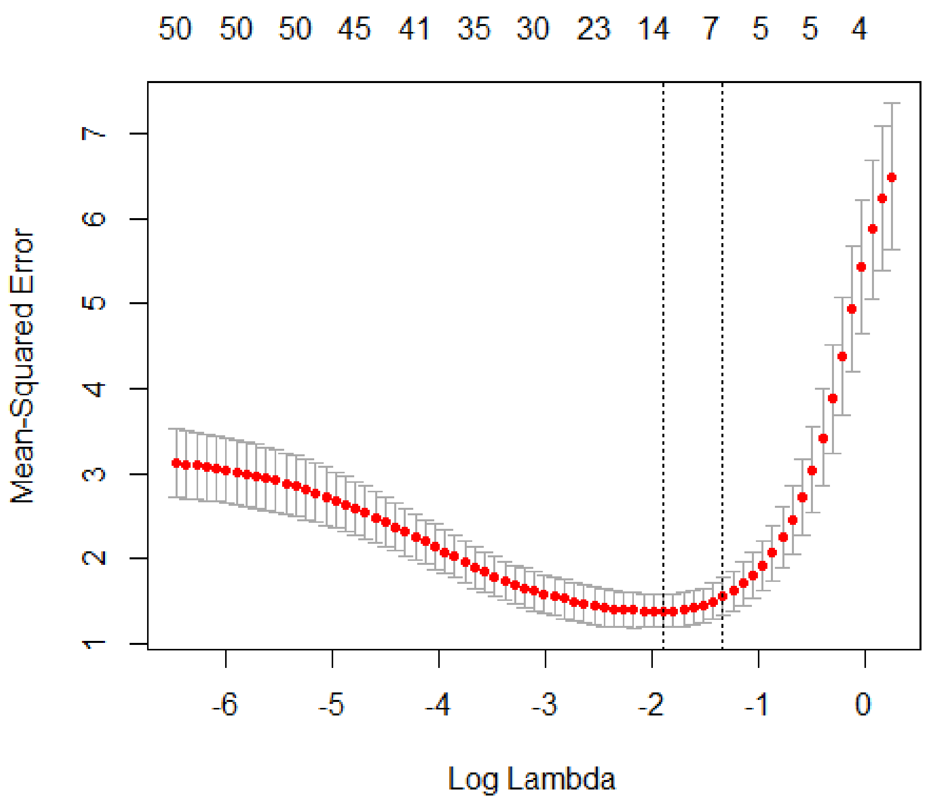

Figure 11: *Cross validated sequence of values for λ to the MSE. The left dotted line represents the best value for λ, where the MSE will be being minimized.*

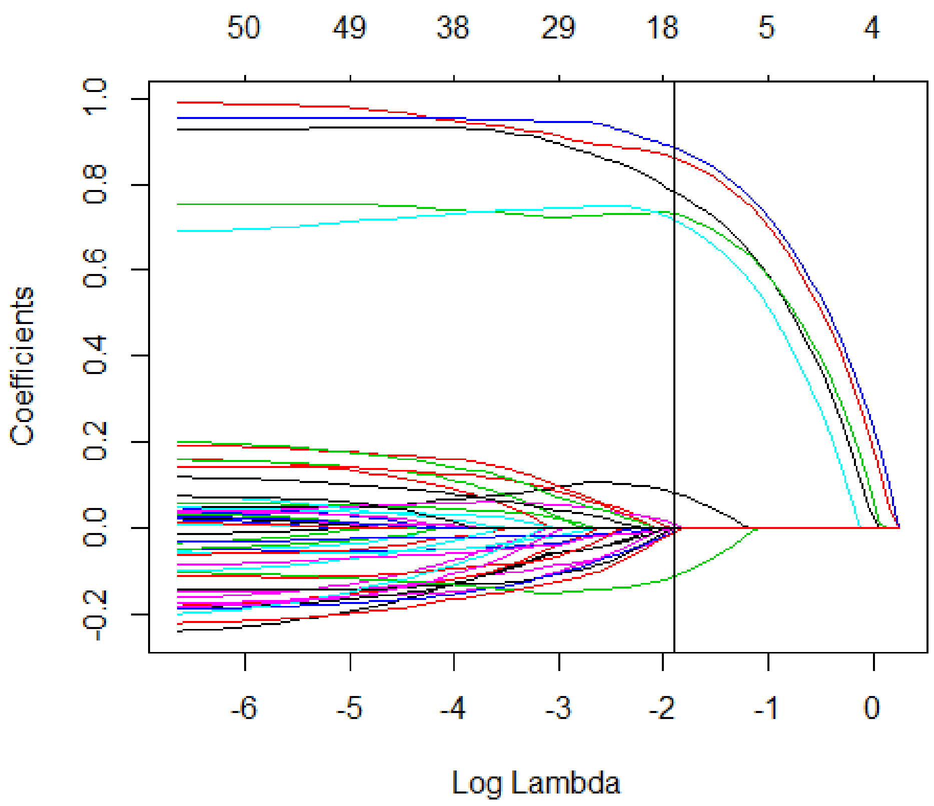

Figure 12: *Coefficient path against λ. Curves upon the x-axis indicate positive coefficients. The five prominent ones display the stronly correlated variables. The line was drawn to show the optimal λ from figure 11 and thus, the best fit.* (for all variables)

3 Regression problem analysis

This section is dedicated to study the performance of our methods on eight regression problems. Five of them origin to real data sets from former research projects. Hence, the remaining three ones arise from simulated data (we say synthetic). For better understanding, we will describe each data set and its purpose thoroughly. Afterwards, we examine boxplots with the results of 100 independent replications for all six algorithms that we used.
The performance measure shown there is the standardized mean squared error. It is defined as the classic MSE, divided by the variance of the response and multiplied by 100:

$$\text{Standardized } MSE = \frac{\mathbb{E}((y_i - f(x_i))^2)}{var(y)} \cdot 100$$

This adjustment allows better comparison of results across data sets.
To compute the MSE for real data, a 10-fold cross-validation was realized. This can be done by randomly splitting the data into 10 virtually equally sized folds (i.e. subsets). Following up, we use 9 folds to learn the algorithm and the spare one to test. We iterate the process until every fold was used for testing, i.e. 10 time (figure 13). While we apply that technique mainly to qualify the algorithms, it does also slow down the rate of overfitting (Hastie et al. 2009*d*).

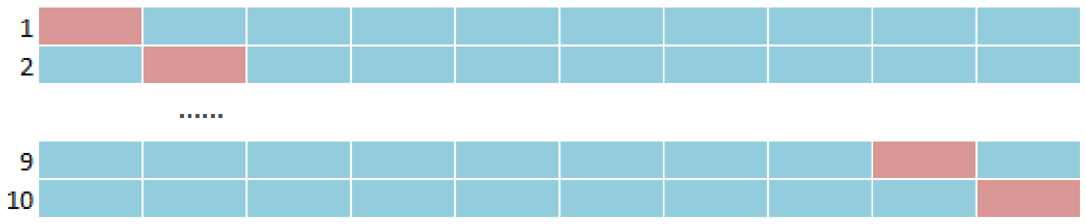

Figure 13: *10 fold cross validation.* Blue folds show the training data and red ones the test data

Finally, the corresponding (standardized) MSE values for each of the 100 replications emerge as the average over all ten folds. For synthetic data on the other hand, MSE values were computed by using an independent test set of size $n = 5000$.
In addition, we present a table with averaged MSE values over the 100 replications. The table does also contain mean values for the computational time of each algorithm. All computations were carried out non-parallelized on a 3.7 GHz CPU unit.

3.1 Implementation and execution

Before we finally eye upon the results, we will give an in-depth explanation on how the methods were implemented.
Synthetic random forest:
We used the „randomForestSRC“ package (Ishwaran & Kogalur 2015) to implement the

algorithm. The function was ran with default values for `mtry` ($p/3$) and `node size` (5). For `ntree` (the number of trees), a quantity of 500 were chosen (as well for all subsequent random forests). Moreover, the key-parameter `node size candidates` was defined as the sequence of $\mathcal{N} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 50, 100\}$. Thus, 14 synthetic features were added to each data set before the actual random forest computation was executed.

Random forest, tuned for the node size:
Here we applied the classic random forest function of the „randomForestSRC“ package with default values for `mtry`. In contrast, the value for `node size` was derived directly from the results of the synthetic random forest. The optimal value is defined as the `node size` of the sequence $\mathcal{N}$, that delivered the smallest out of bag error (Ishwaran & Malley 2014). As already described in chapter 2.1, bagging trees means that each tree is grown from bootstrap samples ($\approx 2/3$ of the training data). The trees are then internally tested on the unused (out of bag) data. Correspondingly, we obtain the out of bag error rate. Note that there is no function available to tune a random forest directly for its optimal node size value. Hence, one has to either compute a synthetic random forest first or multiple random forests and extract the belonging OOB error rates. Consequentially, the CPU time shown for this method are SRF computational time plus the classic random forest with optimal node size.

Random forest, tuned for the mtry:
Since „randomForestSRC“ itself features no ability for tuning the `mtry` parameter, we used a function of the „randomForest“ package (Liaw & Wiener 2014). Nevertheless, final computations were conducted with „randomForestSRC“.

Support vector machine:
To compute the SVM, we utilized the package „e1071“ (Meyer et al. 2015). We applied the radial basis kernel and the ϵ-incentive loss function. For tuning the parameters, a grid search was used. Candidates for the cost parameter `C` were set to {2, 4, 8, 16}. The kernel specific `gamma` value was optimized over a sequence reaching from its suggested value „one divided by the number of features“ (a) to a maximum of 2 (b) with `length.out = 10` (i.e. 10 iterations between (a) and (b) of equal length).

Gradient boosting:
Implementation of gradient boosting was accessed by the „mboost“ package (Hothorn et al. 2015). To obtain a generalized additive model, we applied the `gamboost` function. Furthermore we used stumps as weak learners and the default value for the `shrinkage` parameter (= 0.1). The optimal stopping iteration `mstop` was obtained by the `cvrisk` feature which minimizes the loss function. Note that we tuned one preparative iteration (i.e. 10 folds) with very high input for `mtry`. This was done to decrease CPU time for the upcoming 100 replications (we picked the maximum value).

Lasso:
Finally, the lasso computations were realized with the „glmnet“ package from Friedman et al. (2015). As we explore regression problems, the `family` was set to gaussian. To find

the best value for `lambda`, we employed the `cv.glmnet` function. The elastic-net penalty `alpha` was set to 1 (i.e. the lasso).

3.1.1 Airquality

Our first regression data set „*airquality*“ was taken directly from the „datasets“ package (R Core Team 2014). Its contents were gathered in 1973 by the New York State Department of Conservation and the National Weather Service. The purpose was to evaluate the air quality of New York from May to September 1973.
We will try to predict the mean of „*ozone*“ (inorganic molecule) in parts per billion. Our set of features consists of five variables, such as the average wind speed in miles per hour or the maximum daily temperature for 111 observations.
In figure 14 we see boxplots for each method with their corresponding MSE values.

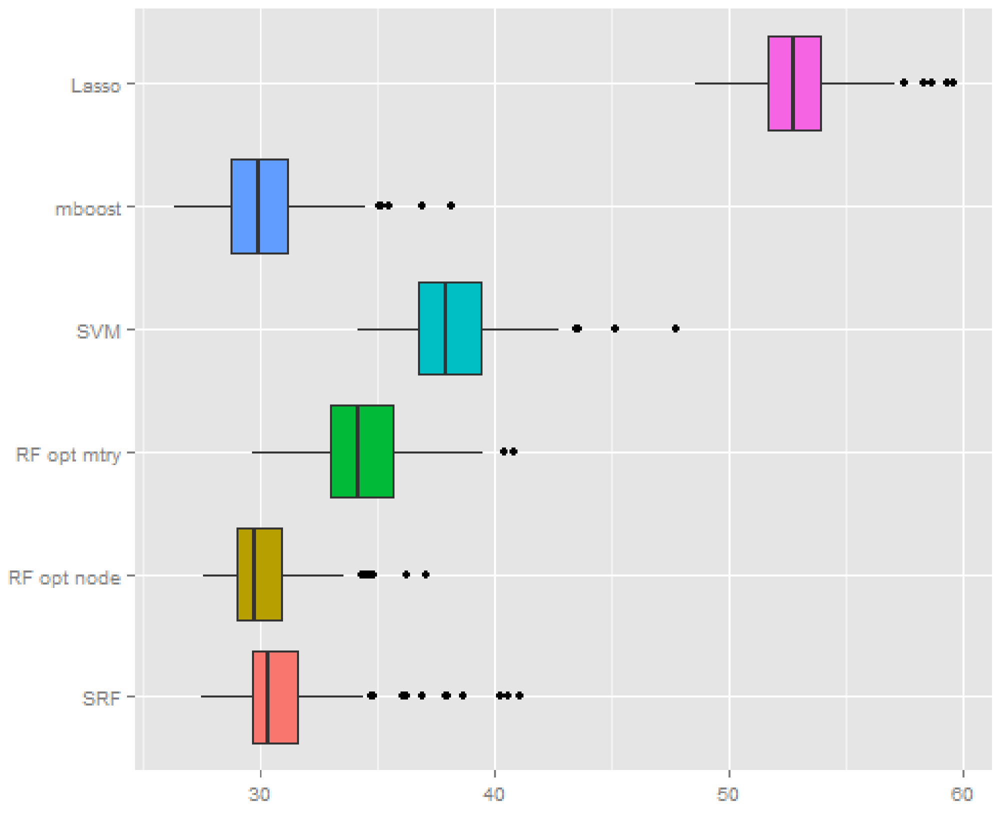

Figure 14: *Standardized and cross validated MSE values* ($\times$ *100) over 100 independent replications on the x-axis* (Airquality data)

While RF (opt node) and the SRF perform quite similar, the latter one shows greater fluctuations. That property is surprising, since the idea of random forests is to reduce the variance. Furthermore it seems that optimizing for node size leads to better results than mtry. That finding might be justified by the small amount of features (5). Thus, it is very likely that RF (opt node) uses the same mtry value. Gradient boosting (mboost) does perform very well, whereas the support vector machine is slightly chipped. The worst performance was accomplished by the lasso.

We may examine the mean MSE values as well as the mean CPU time in table 1. Apparently, the good performance of mboost comes along with the expense of notable greater computational time (74 seconds on average for one replication). By contrast, if we optimize the mtry value of a random forest we do need 2 seconds on average for one replication.

Table 1: mean MSE and mean CPU time in seconds (rounded) for the *Airquality* data, $n = 111$, $p = 5$

Method:	SRF	RF (node size)	RF (mtry)	SVM	mboost	Lasso
MSE	31.15	30.26	34.37	38.26	30.19	53.03
CPU time	13	15	2	31	74	1

3.1.2 BostonHousing

Now we want to predict the median value (in US$) of owner occupied homes in suburbs of Boston. The associated real data were obtained in a census in 1970. It is available in the „mlbench“ package (Leisch & Dimitriadou 2010) which allocated a couple of data sets from the Machine Learning Repository (Lichman 2013).
To execute, we have a total of 506 observations and 13 features. For example, we account for the crime rate per capita, the pupil-teacher ratio or the nitric oxide concentration in parts per 10 million. The results of the computations can be observed in figure 15.

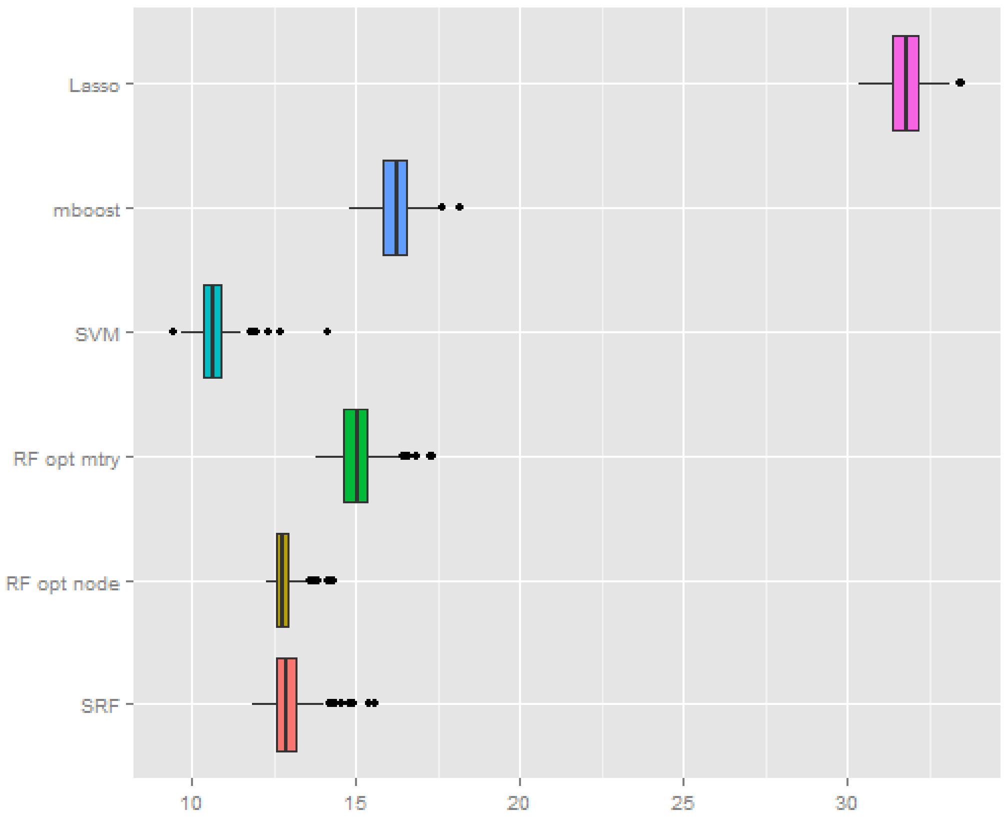

Figure 15: *Standardized and cross validated MSE values (× 100) over 100 independent replications on the x-axis* (Boston Housing data)

Again, the worst output was attained be the lasso. The synthetic random forest as well as the node size tuned forest are both showing good results. Best performance was yielded by the support vector machine, whereas gradient boosting and the random forest (mtry) are not really far behind. To review the mean MSE values and the mean CPU time, we take a look at table 2.

Table 2: mean MSE and mean CPU time in seconds (rounded) for the *Boston Housing* data, $n = 506$, $p = 13$

Method:	SRF	RF (node size)	RF (mtry)	SVM	mboost	Lasso
MSE	13.04	12.87	15.12	10.70	16.26	31.83
CPU time	112	130	14	230	348	1

It is not very surprising that the good performance of the SVM comes along with a greater computational time. The best time to performance ratio is accessed by the mtry optimized random forest. As for the airquality data, mboost CPU time appear to be disproportionate.

3.1.3 Crime

The Crime data was gathered between 1981 and 1987 in North Carolina. We access it by the use of the „Ecdat“ package (Croissant 2015). Our goal is to predict the crimes committed per person. Therefor, we employ a set of 23 features and 630 observations.

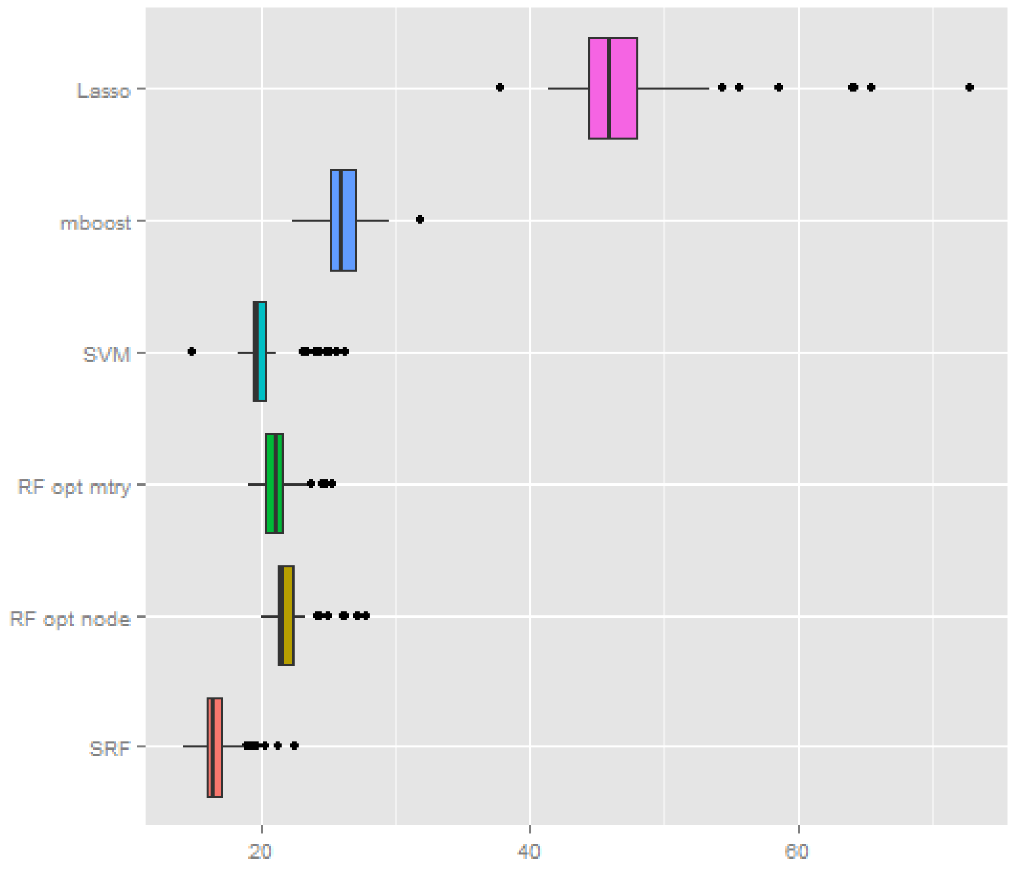

Figure 16: *Standardized and cross validated MSE values (× 100) over 100 independent replications on the x-axis* (Crime data)

Those are for example a county identifier, the police rate per capita, the population density, several income variables but also the proportion of various age groups.
Figure 16 represents the appropriate MSE values. The synthetic random forest shows the best performance. Only slightly behind, all remaining methods except the lasso. Latter one seems to struggle with the problem, as it shows huge variation plus bad performance.

Table 3: mean MSE and mean CPU time in seconds (rounded) for the *Crime* data, $n = 630$, $p = 23$

Method:	SRF	RF (node size)	RF (mtry)	SVM	mboost	Lasso
MSE	16.70	21.94	21.18	20.25	26.24	47.03
CPU time	229	276	42	467	1230	1

Anyhow, the Lasso scores with extremely fast CPU time (see table 3). On the other hand, gradient boosting needs almost 21 minutes for one replication.

3.1.4 Highway

Accessible in the „car“ package (Fox et al. 2015) and collected by of Carl Hoffstedt in Minnesota in 1973, our next data set is called „*highway*“.

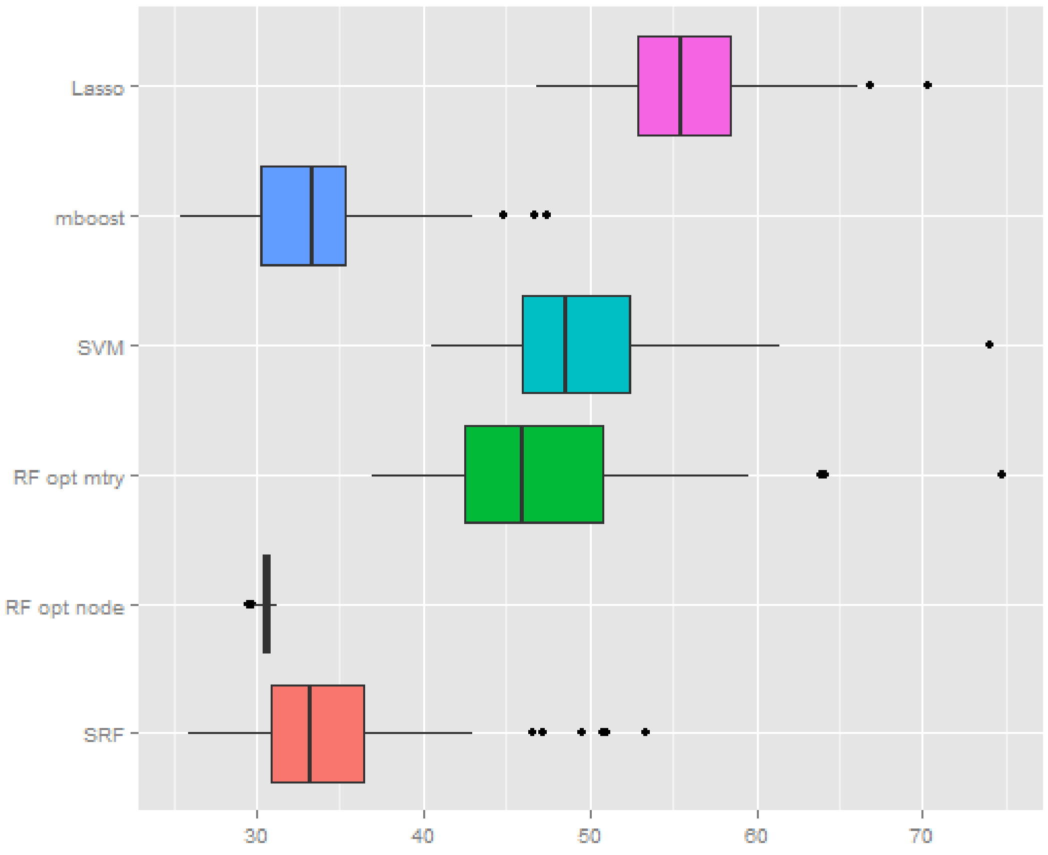

Figure 17: *Standardized and cross validated MSE values (× 100) over 100 independent replications on the x-axis* (Highway data)

The goal is to predict the accident rate per million vehicle miles. We have a total of

39 observations (sections of large highways) and 11 variables, like the speed limits, the average daily traffic count in thousands or the lane width in feet.
Applying all six methods to the data, yields the results shown in figure 17. What stands out immediately is the robustness of the node size optimized random forest. A feasible approach for explaining this characteristic, is according to Ishwaran & Malley (2014) that „node size acts as a type of bandwidth parameter that controls the level of smoothness of the RF predictor“. Second best performance was scored by SRF and mboost. In the middlefield we see the random forest optimized for the mtry value, persecuted by the SVM. Not so far behind this time but still the poorest outcome was achieved by the lasso.

Table 4: mean MSE and mean CPU time in seconds (rounded) for the *Highway* data, $n = 39$, $p = 11$

Method:	SRF	RF (node size)	RF (mtry)	SVM	mboost	Lasso
MSE	34.30	30.56	46.99	49.58	33.50	55.91
CPU time	5	6	1	31	82	1

Mean MSE values can be compared in table 4 where they back the findings from the boxplots. As usual, the gradient boosted model needs most time to compute.

3.1.5 Ozone

The last real data set we want to scrutinize was taken from the „mlbench“ package (Leisch & Dimitriadou 2010). Coming from a study on the atmospheric ozone concentration in Los Angeles, it has been made public domain by Breiman & Friedman (1985).
Our aim is to forecast the daily maximum one-hour-average ozone concentration.
To perform, we have a total of 203 observations plus 11 features. These might be for example the day of the week, the humidity or the pressure gradient in millimeter of mercury (mmHg).
Referring to our findings shown in figure 18 the SVM outperforms tree based methods as well as the lasso. As in the case before, the RF (node size) shows least scattering, whereas the SVM, mboost and RF (mtry) produce some outliers.
For another perspective one may examine table 5.

Table 5: mean MSE and mean CPU time in seconds (rounded) for the *Ozone* data, $n = 211$, $p = 11$

Method:	SRF	RF (node size)	RF (mtry)	SVM	mboost	Lasso
MSE	26.96	26.92	27.38	23.17	27.55	33.40
CPU time	126	138	8	92	110	1

Thus, it can be seen more explicitly, that the lasso is actually not that far away from random forests or the mboost. Furthermore, as for all data sets so far, its CPU time is by far the best. Another interesting point is the fact that mboost for the first time did not have the worst computational time.

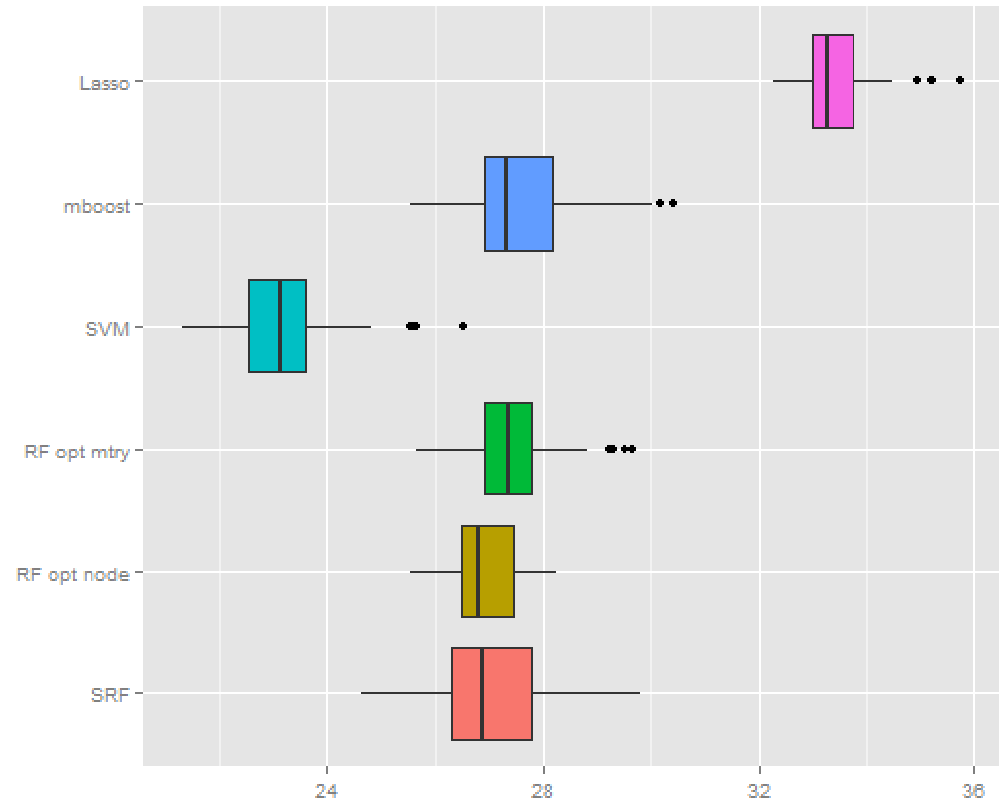

Figure 18: *Standardized and cross validated MSE values (× 100) over 100 independent replications on the x-axis* (Ozone data)

3.1.6 Peak

A nice facet of the „mlbench“ package (Leisch & Dimitriadou 2010) is the ability to create artificial data sets with arbitrary combinations of features and responses.
We use the peak function to create a regression problem y. Our x^T is uniformly distributed on a d-dimensional sphere with radius r for $r = 3u$ and $u \in [0, 1]$.

$$y = 25exp(0.5r^2) \tag{18}$$

For $n = 250$ and $d = 20$ (number of features) we obtain some kind of left-tailed bimodal density function that can be seen in figure 19.
As we apply the algorithms to the data, we obtain remarkable results from the SVM (figure 20). It is very likely that this results from the radial basis function kernel, which might fit very good on that problem (see chapter 2.3.1, equation 13). Second best performance was achieved by the SRF and right behind the mboost. A little further appear both random forest computations and then very chipped the lasso. On a side note one may discern that all algorithms seem to be very robust in this synthetic case and exhibit very little scattering.

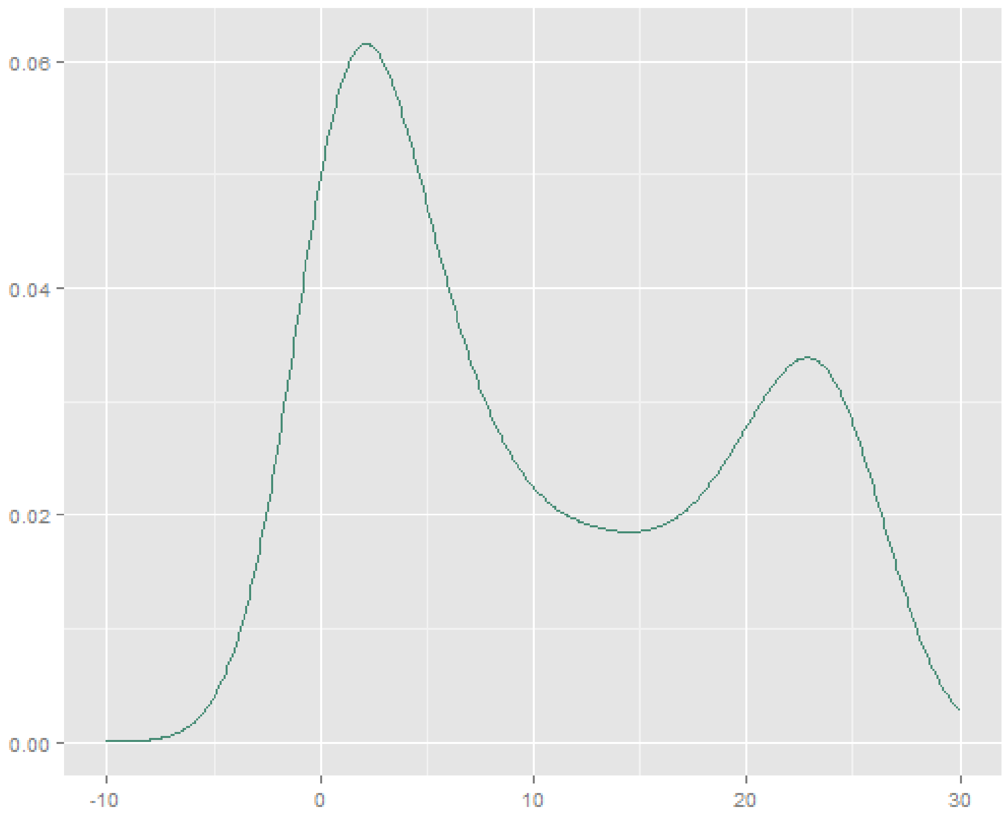

Figure 19: *Density for the mlbench peak function*

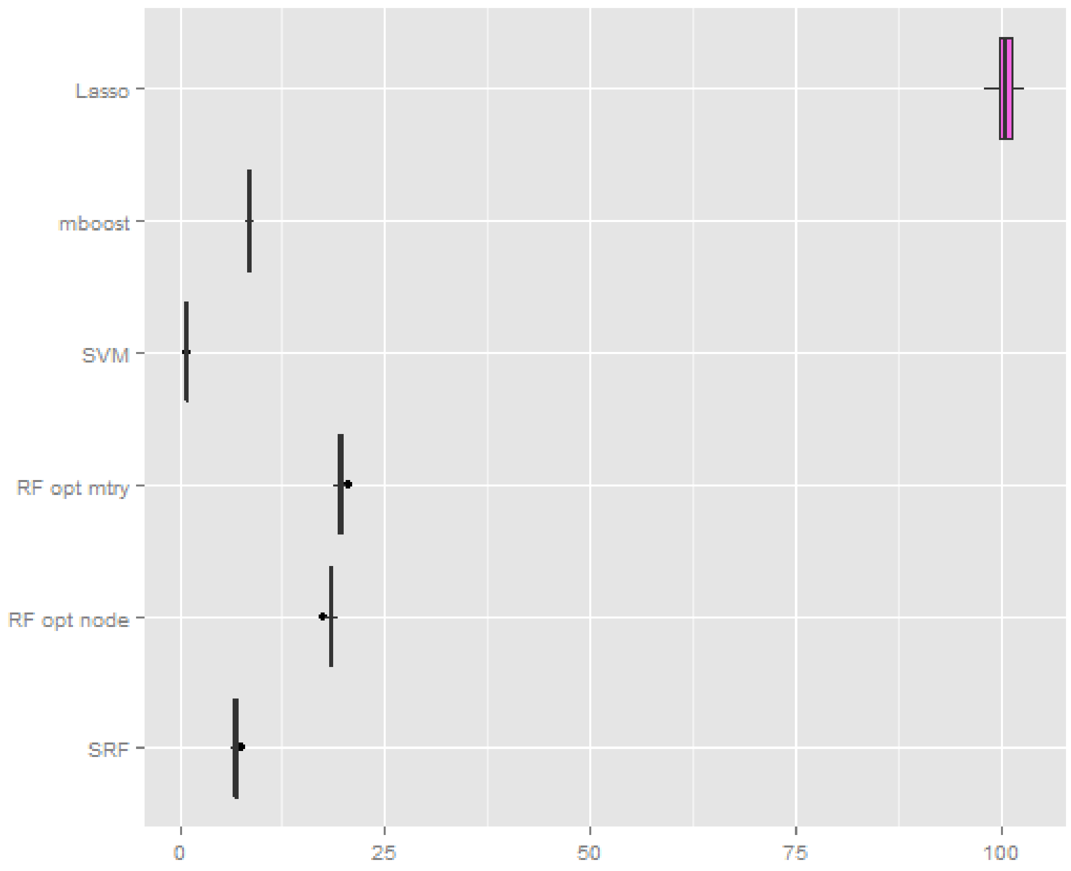

Figure 20: *Standardized test set MSE values (× 100) over 100 independent replications on the x-axis* (Peak data)

Furthermore, table 6 demonstrates, that the good performance from the SVM and the SRF does not come along with major CPU time. In particular, the SVM needs only one second for each replication. Again, it is very likely that this results from the radial basis kernel. The remaining random forests experience unusually high CPU time. Eclipsing the rest, mboost needs 379 seconds on average for one replication. Very fast CPU time, but unfortunately very bad performance too, was accomplished by the lasso.

Table 6: mean MSE and mean CPU time in seconds (rounded) for the *Peak* data, $n = 250$, $p = 20$

Method:	SRF	RF (node size)	RF (mtry)	SVM	mboost	Lasso
MSE	6.80	18.40	19.58	0.84	8.45	100.63
CPU time	16	89	69	1	379	0

3.1.7 Syn50

This section is dedicated to a regression problem with redundant variables (i.e. variables that have no influence on the response).
Therefore we created a Gaussian distributed data set with 50 features and 100 observations. We chose 5 of our predictors to be correlated with the response and hence, 45 pointless ones. The resulting prediction performance can be seen in figure 21.

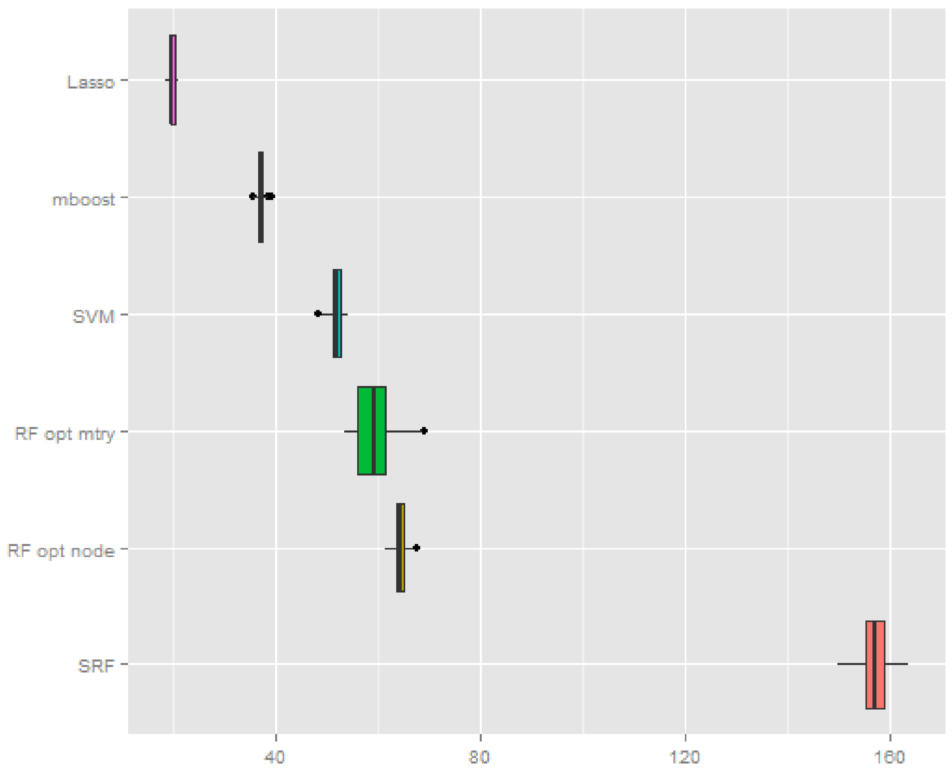

Figure 21: *Standardized and test set validated MSE values (× 100) over 100 independent replications on the x-axis* (Syn50 data)

For the first time, the synthetic random forest shows a really bad outcome. The reason for this phenomenon is pretty obvious. As described in chapter 2.1.2, the SRF computes new synthetic features using the underlying data. Employing the uncorrelated variables too, a bias will inevitably emerge. A second reason is caused by the feature selection procedure. At each node, $p/3$ variables are being considered for the next split. Given the proportion of 5 important to 45 redundant ones, the RF will very likely only pick redundant features. Hence, the prediction will be corrupted. That problem accounts too for the RF (mtry) and in particular for the RF (opt node).
Finally the lasso shows off, as its main intention to shrink down pointless variables and detect those who really matter, can be utilized. Second best performance yields gradient boosting which works in a likewise way as the lasso does. Slightly better performance than both RFs is accessed by the support vector machine.

Table 7: mean MSE and mean CPU time in seconds (rounded) for the $Syn50$ data, $n = 100$, $p_{total} = 50$, $p_{real} = 5$

Method:	SRF	RF (node size)	RF (mtry)	SVM	mboost	Lasso
MSE	157.15	64.34	59.61	51.85	37.23	19.74
CPU time	12	13	1	16	73	0

Scrutinizing table 7, we see in numbers how good the lasso actually performed. Beyond that, its CPU time is much better than any other algorithms.

3.1.8 Syn250

Now we would like to investigate, how our algorithms react, as we increase the number of pointless variables. While we hold $n = 100$ and $p_{true} = 5$ fix, we increase p_{total} to 250 (i.e. 245 redundant variables).
figure 22 shows the corresponding MSE values for 100 replications. Taking a first glimpse on the plot, one could assume that increasing p_{total} from 50 to 250 had no significant effect. In fact, the lasso as well as the mboost algorithm appear to be very robust. Moreover, the SVM and RFs (not the SRF) results became slightly worse.
But what happens to be particularly striking, is that the CPU time of the lasso hardly changed (table 8). A completely different outcome accounts for the mboost. If we quintuple the number of non-relevant variables, the computational time of mboost increases by a factor of roughly 33 from 73 seconds on average to 2431 for a single iteration.

Table 8: mean MSE and mean CPU time in seconds (rounded) for the $Syn250$ data, $n = 100$, $p_{total} = 250$, $p_{real} = 5$

Method:	SRF	RF (node size)	RF (mtry)	SVM	mboost	Lasso
MSE	123.58	84.57	79.36	76.26	38.31	18.81
CPU time	29	33	2	69	2431	0

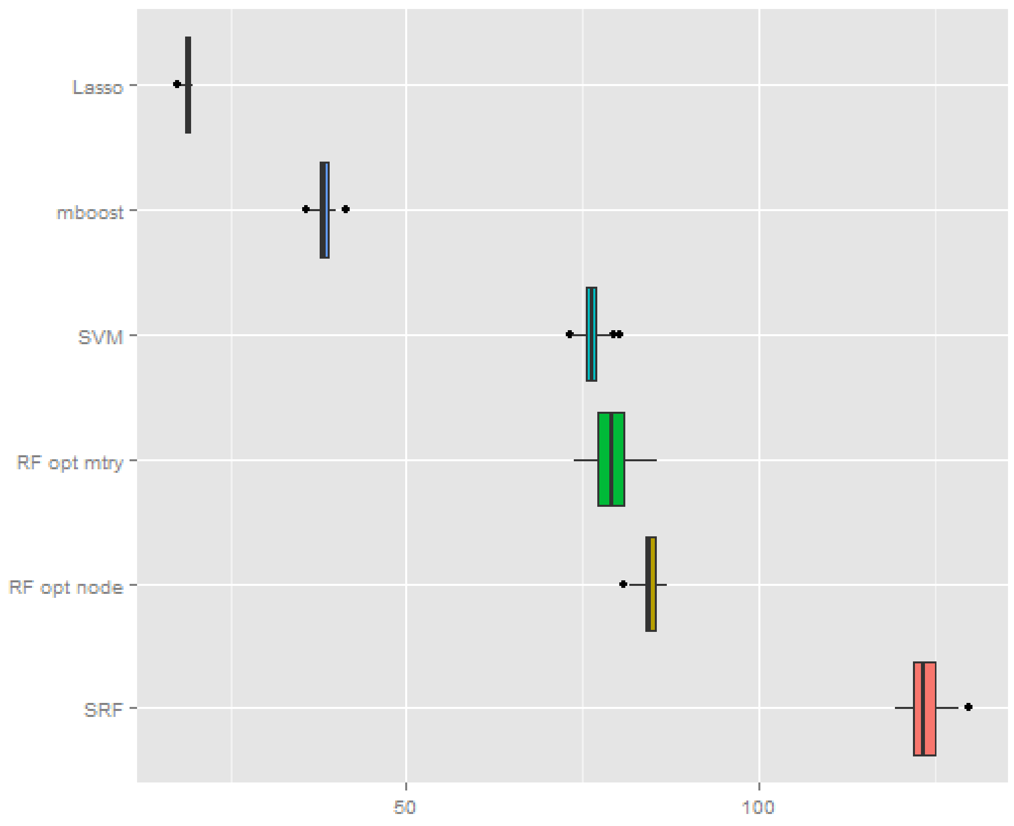

Figure 22: *Standardized and test validated set MSE values (× 100) over 100 independent replications on the x-axis* (Syn250 data)

4 Classification problem analysis

In the same manner as we did for chapter 3, we start off by specifying our procedure. We are going to analyze seven real and one synthetic classification problem. Four of them have a binary response, while the others range from four to fifteen classes. To evaluate the performance of our algorithms, we apply the Brier score as our main measurement. While we also look at the misclassification rate (i.e. the error), the Brier score is superior for data sets with unbalanced class sizes as it includes the probabilistic accuracy of the method at hand.

$$\text{Brier score} = \frac{1}{n}\sum_{i=1}^{n}(f_n - o_n)^2 \tag{19}$$

Where o_n displays the actual outcome (i.e. true/false). Consider a simple example. We have a data set with $n = 10.000$ observations, whereof 9.900 belong to class A and the remaining 100 to class B. Assume our algorithm classifies all 10.000 observations to class A. The corresponding error rate would be only 1% and thus, one might judge the algorithm erroneously as good. By contrast, the Brier score involves the probability for both classes of each observation. Suppose the estimated probability for class B of observation n_i is 80%. Then, the resulting Brier score is either

$$\text{Brier score}(n_i) = \begin{cases} (0.8-1)^2 = 0.04 & \text{if } n_i \in \{class\ B\} \\ (0.8-0)^2 = 0.64 & \text{if } n_i \in \{class\ A\} \end{cases}$$

Derived from the example, the best accessible Brier score is 0 and the worst 1. Note that for a binary problem, a Brier score of 0.25 means the algorithm estimated a probability of 50% for class A and 50% for class B. The result can be interpreted as „random guessing“ and is, although it seems to be far away from 1, already very bad.
All Brier score values shown in the upcoming boxplots were obtained from 50 independent replications, each time multiplied with 100.
As we did in the regression case for the MSE, we computed Brier score values by applying a 10-fold cross-validation for all real data sets. Analogously, an independent test set of size $n = 5000$ is used for the synthetic data.
In conclusion we inspect a table with mean values for the Brier score values as well as the misclassification rate of all 50 replications. Beyond that, computational time for all algorithms were recorded and attached to that table. As before, all computations were executed non-parallelized on a 3.7 GHz CPU unit.

4.1 Implementation and execution

This section is dedicated to the implementation and parameter selection of our methods. While some of them could be utilized in the same nature as for regression, we had to slightly alter a few of them. Only one algorithm has been replaced completely.

Synthetic random forest:
The very same values for all `parameters` such as described in chapter 3.1 were applied. Since the algorithm itself distinguishes between regression and classification, no major modifications were necessary. This means in particular for classification, a K-dimensional (K classes) feature has been computed for every `node size` candidate (i.e. the probability of each class label).

Random forest, tuned for the node size:
Similarly to SRF, no fundamental modifications were required in order to carry out the algorithm.

Random forest, tuned for the mtry:
Unsurprisingly, the random forest tuned for the `mtry` value did not need adjustments either.

Support vector machine:
To compute the SVM, we only changed the `length.out` step size for the tuning sequence of the radial basis functions `gamma` value from 10 to 5. This was due to the extremely high CPU time.

SAMME:
To employ the multi-class AdaBoost, specifically SAMME, we used the „adabag“ package from Alfaro et al. (2014). Unfortunately , „adabag“ itself features no function for parameter tuning. Therefore, to automatically obtain values for `mfinal` (number of trees added into the ensemble) and `tree depth`, we tuned in a stage-wise fashion.
In the first instance, we set `mfinal = 500` and boosted stumps for one replication (i.e. 10 times, once for each fold). Emerging from these computations, we analyzed the training errors. The conclusive value for `mfinal` was then subjected to the iteration where the the minimum training error was observed. That value appeared to be very small for nearly all data sets (i.e. between 20 and 50). Thus, to avoid underfitting, we established a minimum value for `mfinal = 100`.
After that, in consideration of the new `mfinal`, we tried all values $\in \{1, 2, 3, 4, 5, 6, 7, 8\}$ for the `tree depth`. Analogously, we evaluated training error results for each `tree depth` candidate. Obviously, we opted that value, whose exhibited the smallest training error for one replication (i.e. the mean over 10 folds).
Finally we started the SAMME computations with the previously selected values of `mfinal` and `tree depth`.

Lasso:
In order to apply the lasso on classification problems, we had to change the `family` to `multinomial`. Furthermore, some minor but crucial modifications in order to facilitate the computation of the Brier score were mandatory.

4.1.1 Breast Cancer

Our first classification problem originates from a clinical study. The goal was to gather characteristics of patients with tumors and subsequently predict their appropriate class. While one group had the hazard-free benignant mutation ($n_b = 444$), the others were affected by the harmful and cancerous malignant class ($n_m = 239$).
Such characteristics are several stampings of the patients cells (i.e. thickness, size and shape) or the bare nucleus on a scale from one to ten. The latter variable describes the degree of devoid of cytoplasm, which is commonly observed in cell degeneration. The study was carried out by Dr. Wolberg in Wisconsin (USA) between the years of 1989 and 1992. We took the data from the machine learning repository (Lichman 2013) and want to predict the target class label. For that, we have a total of 683 observations and 10 features available.
Figure 23 displays the resulting outcomes for our methods. Brier score values appear to be very low for all methods. Note that even SAMME which might look chipped on the first glimpse, does still deliver good results. However, it seems to be prone for the structure of the data as its values vary the most. A similar picture becomes apparent as we inspect the misclassification rate in figure 24.

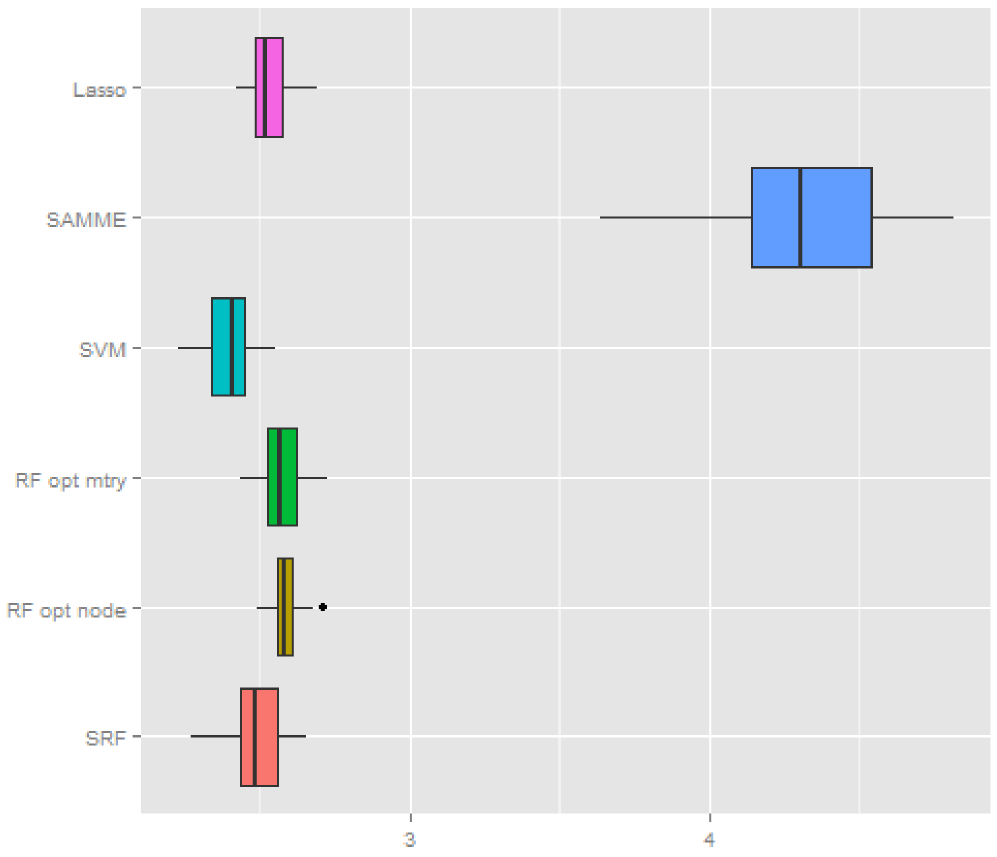

Figure 23: *Cross validated Brier Score values (× 100) over 50 independent replications on the x-axis* (Breast Cancer data)

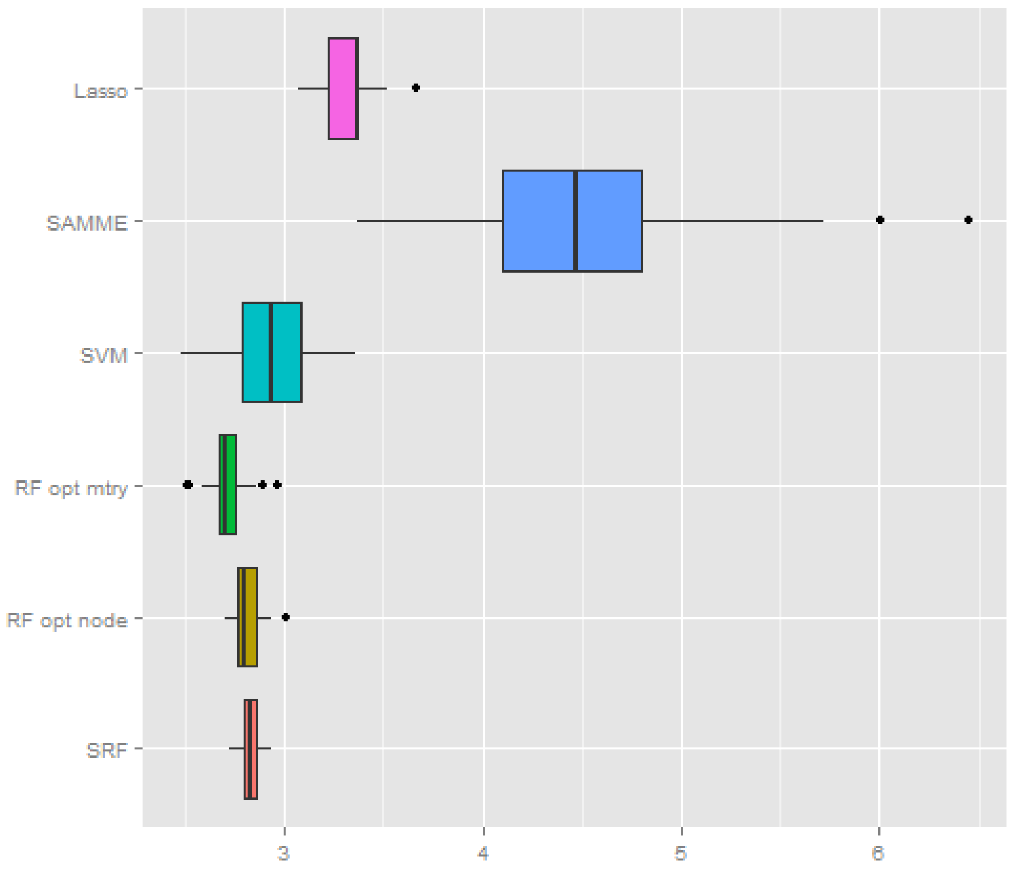

Figure 24: *Cross validated misclassification error rate (× 100) over 50 independent replications on the x-axis* (Breast Cancer data)

Yet, we can hardly elect a winner since some methods appear to perform equally well. Thus, we consider computational time as exhibited in table 9. The synthetic random forest (and thus the RF (node size)) requires an elusively high CPU time. By contrast, tuning a RF for the mtry value needs only a small fraction therefrom. Additionally, a slighty better result was obtained. Particularly striking, the lasso outperforms everything as it a lot faster while it still yields very good results.

Table 9: mean Brier Score, mean misclassification and mean CPU time in seconds (rounded) for the *Breast Cancer* data, $n = 683$, $p = 10$, $K = 2$

Method	SRF	RF (node size)	RF (mtry)	SVM	SAMME	Lasso
Brier Score	2.49	2.59	2.57	2.39	4.32	2.53
Misclassification	2.83	2.81	2.71	2.95	4.46	3.31
CPU time:	4784	4922	197	552	118	30

4.1.2 Colon

The next data set was used by Alon et al. (1999) to discover a potential relationship between the gene expression of people (in a broader sense: the genetic information of a gene) and colorectal cancer. In a microarray experiment (molecular biology), 62 tissue

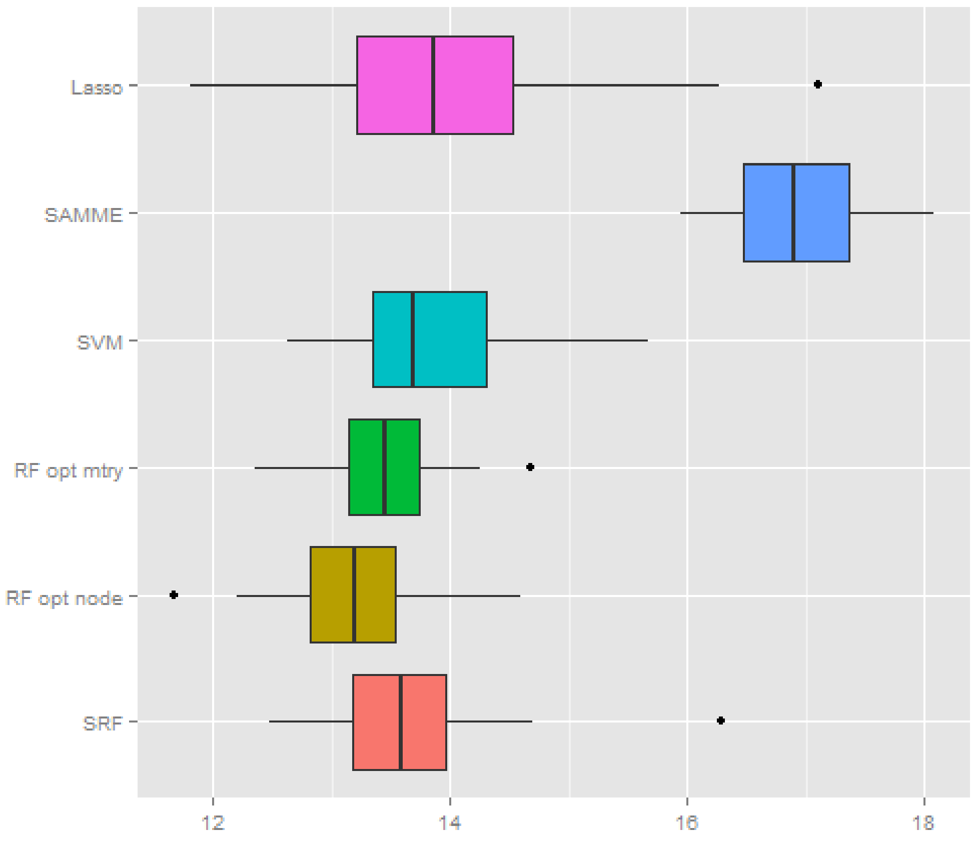

Figure 25: *Cross validated Brier Score values (× 100) over 50 independent replications on the x-axis* (Colon data)

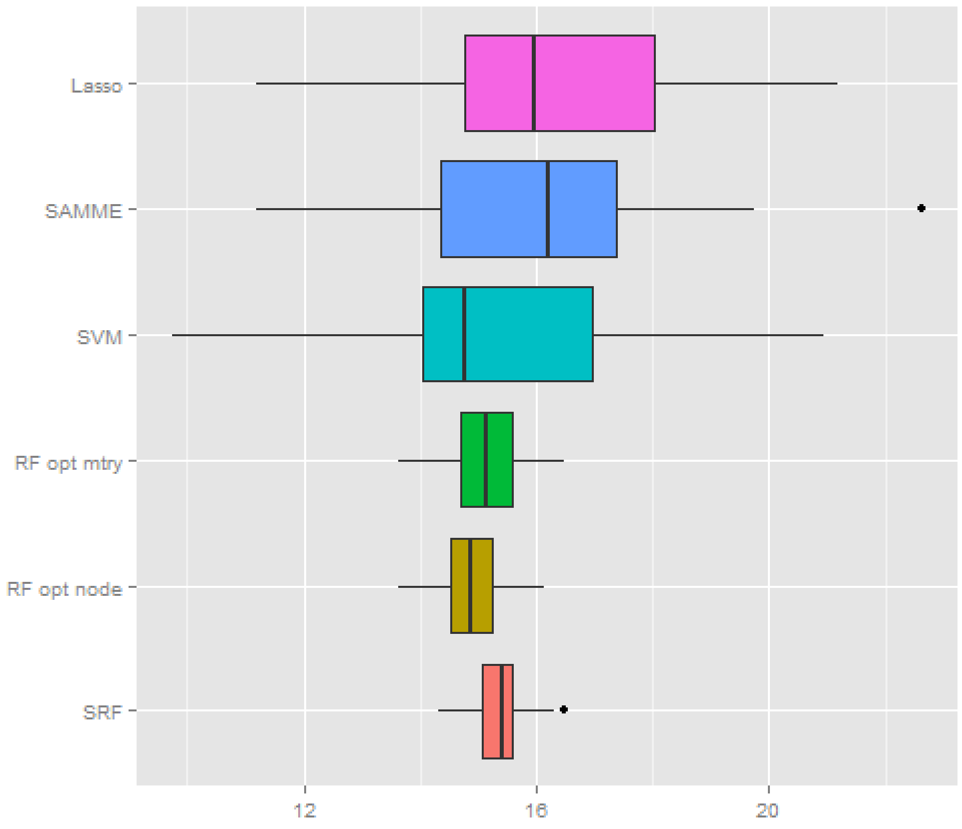

Figure 26: *Cross validated misclassification error rate (× 100) over 50 independent replications on the x-axis* (Colon data)

samples with 2000 gene characteristics each, were collected. 40 of the patients were diagnosed with cancerous tumors whereas the remaining 22 had healthy samples. Our goal is to predict the class label for the respective patient.
As we take a first look at figure 25, the pattern seems to be quite familiar. The Breast Cancer data showed a similar arrangement, with 5 methods performing in a likewise fashion, and slightly behind the SAMME. But unlike previously, SAMME together with RF (opt mtry) exhibit smallest variation. Overall, no algorithm seems to achieve an outstanding effort here. As we inspect the misclassification rate (figure 26), we see an interesting contrast between random forests and the remaining methods. The idea of lowering the variance becomes apparent as all three RF computations present only very little variability. Secondly, SAMME shows a better performance than we could have expected.
Table 10 reveals some really fascinating properties. It seem that the SVM had to struggle with the observation to feature ratio, as its mean CPU time for one replication almost exceeded one hour. Untouched from that proportion, the lasso does only need 13.41 seconds on average. Second best speed was accessed by the RF (mtry).

Table 10: mean Brier Score, mean misclassification and mean CPU time in seconds (rounded) for the *Colon* data, $n = 62$, $p = 2000$, $K = 2$

Method	SRF	RF (node size)	RF (mtry)	SVM	SAMME	Lasso
Brier Score	13.60	13.20	13.44	13.85	16.89	14.02
Misclassification	15.35	14.90	15.10	15.32	15.81	16.41
CPU time:	1508	1582	86	3184	1593	14

4.1.3 Glass

We will now analyze a data set called „*Glass*“ (Lichman 2013). Based on forensic investigations for evidence in criminal cases, we aim to identify the type of glass splinters according to their chemical composition.
That might be the proportion of Sodium, Magnesium or Aluminium. In total, nine features and 214 observations were collected by the Forensic Science Service of the United Kingdom. Our response variable divides into six groups, for example headlamps, vehicle or building windows.
Figure 27 portrays the observed Brier Score values and its corresponding ranges. Our findings suggest that best probabilistic results were obtained from all three random forest computations. Note that all six methods here deliver viable performance. Even though the lasso looks far behind, a Brier Score of 9 indicates a forecasting probability of 70% for the correct class (bearing in mind that there are six alternative classes).
To observe how classification ended up, we examine the equivalent error rates shown in figure 28. Interestingly SAMME catches up here and rarely even outperforms other methods. Responsible for that might be the extreme distribution within the classes.

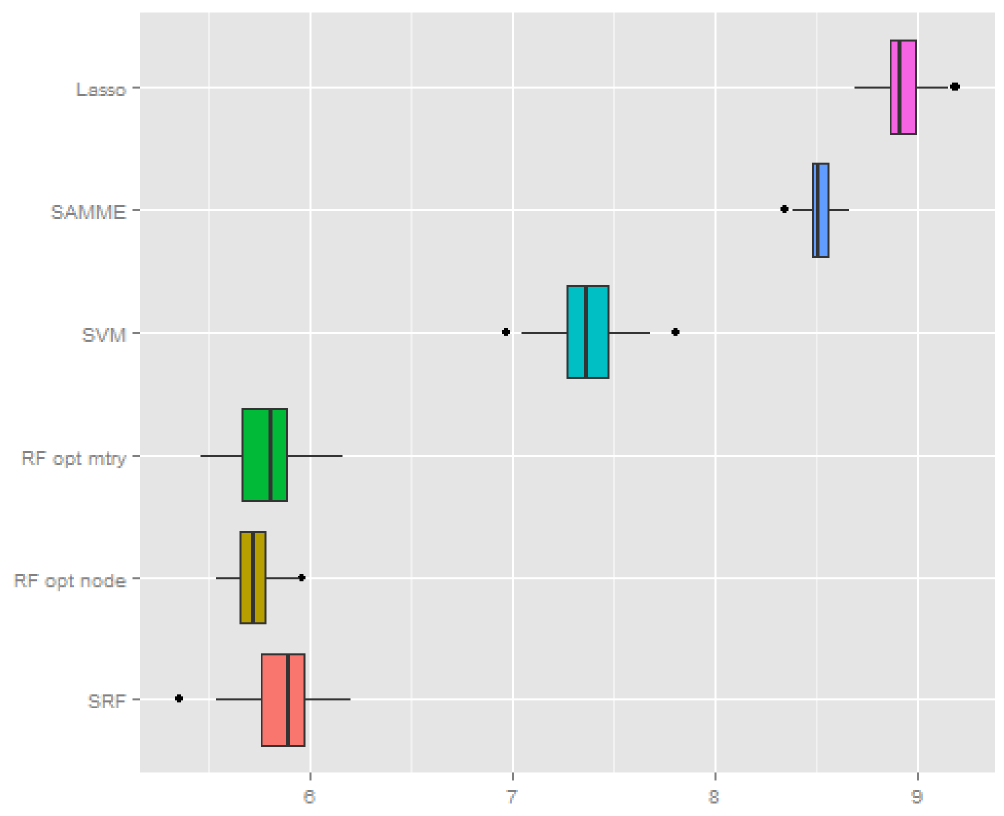

Figure 27: *Cross validated Brier Score values (× 100) over 50 independent replications on the x-axis* (Glass data)

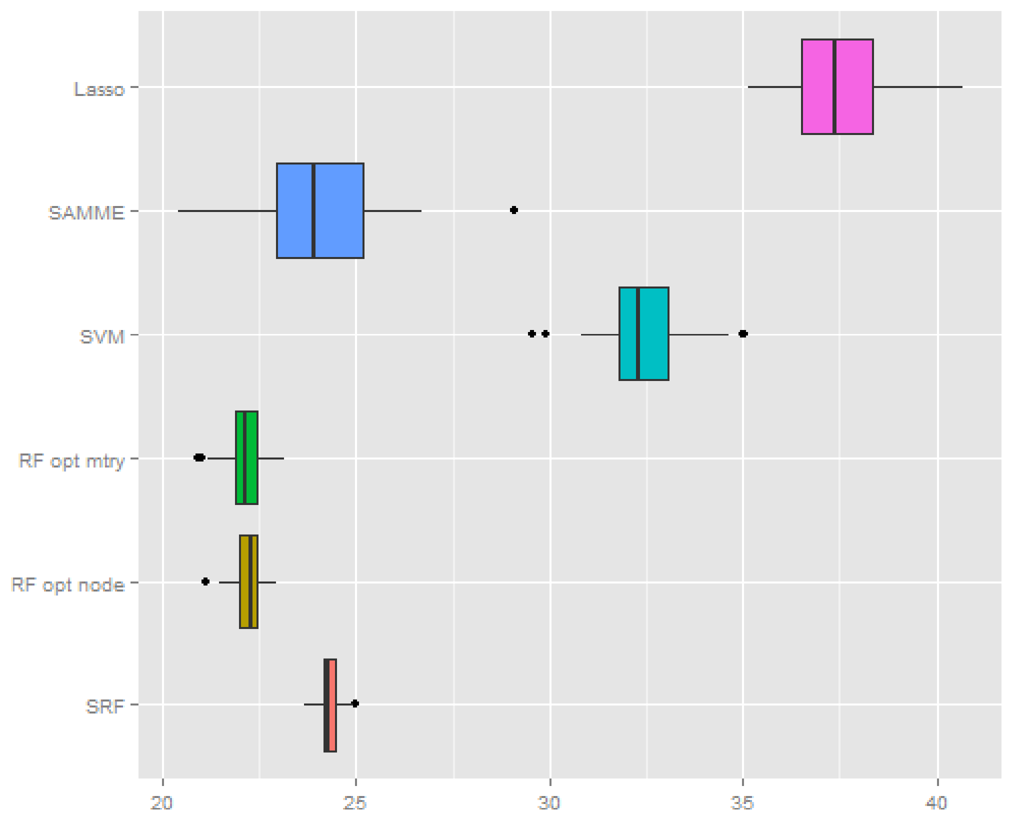

Figure 28: *Cross validated misclassification error rate (× 100) over 50 independent replications on the x-axis* (Glass data)

As we can see in table 11, the first two groups already utilize 70% of all observations. Thus, the misclassification rate has to be reviewed with caution.

Table 11: Total number of observations and cumulative proportion for all six classes (Glass)

Class	1	2	3	4	5	6
observations	70	76	17	13	9	29
cum. proportion	0.33	0.70	0.78	0.84	0.88	1

Generally, it seems that no method was able to manage the problem very well.
Finally to gain an insight of how the trade-off between mean performance and mean CPU time went off we eye on table 12. The unusual high computational time and moderate results of the lasso might be attributed to its lack of convergence for λ. Even after the algorithms internal maximum iteration ($i = 100.000$) no good solution was found. Parental might be the 10-fold cross-validation and respectively, the high number of classes conditioned on rather few observations. As we split the data into ten folds, it might have occurred that the training data did not have access to one of the rarely occurring classes (or insufficient access to draw good conclusion). Nevertheless, the fastest algorithm with second best performance was the RF optimized for the mtry value.

Table 12: mean Brier Score, mean misclassification and mean CPU time in seconds (rounded) for the *Glass* data, $n = 214$, $p = 9$, $K = 6$

Method	SRF	RF (node size)	RF (mtry)	SVM	SAMME	Lasso
Brier Score	5.88	5.73	5.79	7.38	8.51	8.93
Misclassification	24.26	22.23	22.12	32.39	23.97	37.53
CPU time:	34	38	4	35	112	69

4.1.4 Sonar

As already briefly presented in chapter 2, we are now going to exercise the Sonar data (Lichman 2013). The underlying contents were generated by Terry Sejnowski from the University of San Diego in collaboration with Paul Gorman from the Allied-Signal Aerospace Technology Center. They wanted to develop a system, that detects hazardous mines from world war II with sonar signals. In total, we have access to 208 observations. 111 of them are mines and the remaining 97 are rocks (i.e. pretty even groups). Moreover, we employ a huge set of 60 variables. Those are the sonar signals, differing from low to high frequency and their corresponding energy level from the collision with the objects. According to figure 29, the best Brier score was accomplished by the SVM. Leading random forest computation and second best overall, is the synthetic variant. The remaining two RFs deliver only moderate performance. Almost unusable results were yielded by SAMME and in particular the lasso.

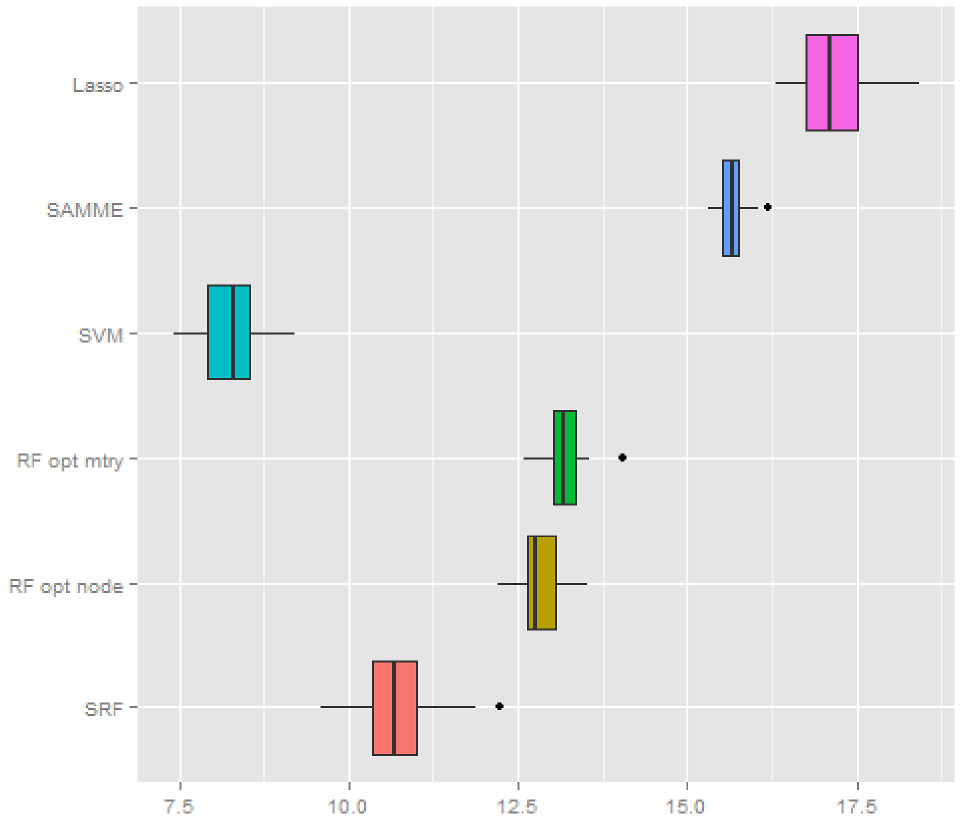

Figure 29: *Cross validated Brier Score values (× 100) over 50 independent replications on the x-axis* (Sonar data)

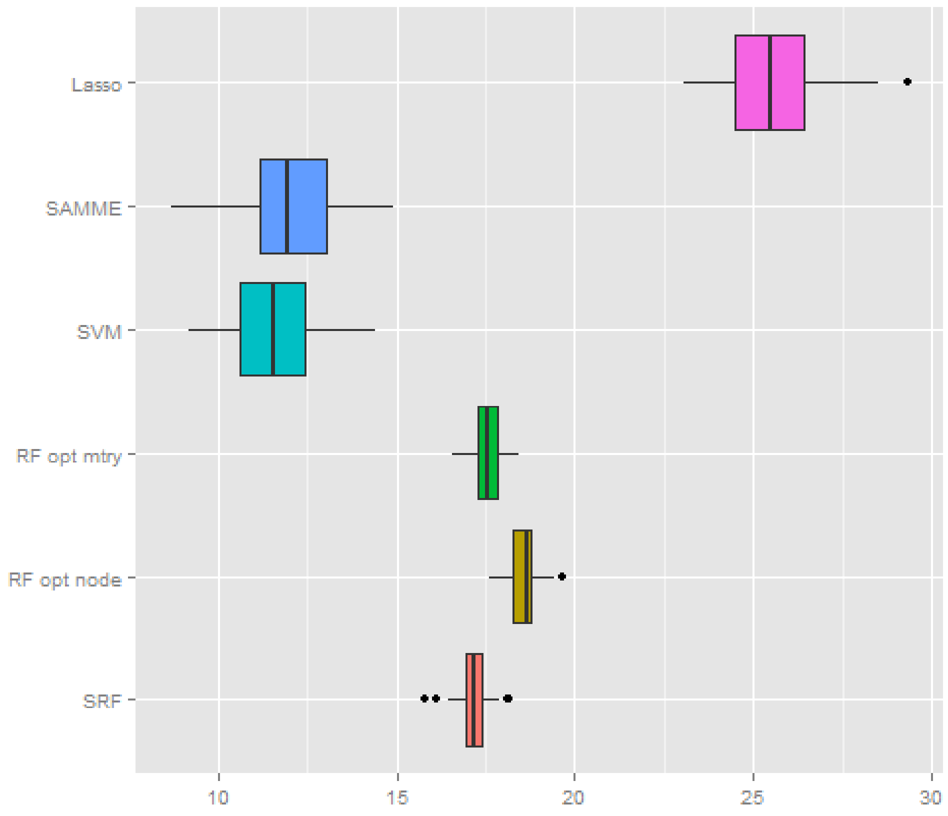

Figure 30: *Cross validated misclassification error rate (× 100) over 50 independent replications on the x-axis* (Sonar data)

Somehow, SAMME managed again to classify much better than we would have expected. In fact, it is second best and almost as good as the SVM (see figure 30). This is curious, because as mentioned before, the two classes are quite even. For each of the 50 replications, the algorithm yields zero error in 1-3 folds, but still high Brier score. Thus, there must be something else to explain SAMMEs special behaviour.

Table 13: mean Brier Score, mean misclassification and mean CPU time in seconds (rounded) for the *Sonar* data, $n = 208$, $p = 60$, $K = 2$

Method	SRF	RF (node size)	RF (mtry)	SVM	SAMME	Lasso
Brier Score	10.75	12.82	13.18	8.26	15.65	17.18
Misclassification	17.15	18.57	17.59	11.59	12.08	25.55
CPU time:	89	97	10	113	164	236

For the very first time, the lasso had the highest CPU time (see table 13). As in the Glass data before, the algorithm had troubles relating the convergence of lambda. It is very likely that there is a relation between the bad performance. Aside from that, we see that the SVM as it yields best results for Brier score and missclassification rate, does not need disproportionately high CPU time.

4.1.5 Soybean

With a quantity of 15, the Soybean data represents the problem with highest count of potential classes. While we took the data from the machine learning repository (Lichman 2013), it originates from a study in 1980.
The purpose was to develop an expert system, to automatically diagnose diseases of soybeans. Many of the associated predictors are binary and provide information about the intactness of leaves or the growing behaviour (i.e. good/bad). In total, we utilize for our computations 35 features and 562 observations. Class labels are very widespread and technical. Some are for example „herbicide injury“ or „bacterial-blight“. The class-distribution is very unbalanced and ranges from a maximum of 92 down to a minimum value of 20 observations.
Anyhow, contemplating figure 31, we observe an excellent performance done by all methods. Four of them even managed to achieve a Brier score smaller than 1 for every single replication. On a side note, while the SAMME result looks far behind, its Brier score is still very good.
A similar phenomenon as in the Sonar data right before illustrates itself as we inspect the classification error in figure 32. Once again, SAMME was able to perform a lot better than one might have expected by judging its Brier score. Nevertheless, best performance was scored by the lasso. Its low error rate (i.e. only 6.53% on average) comes along with unfamiliar high computational times. The RF (mtry) needs less than one-tenth of the lasso and yields almost same results (see Table 14).

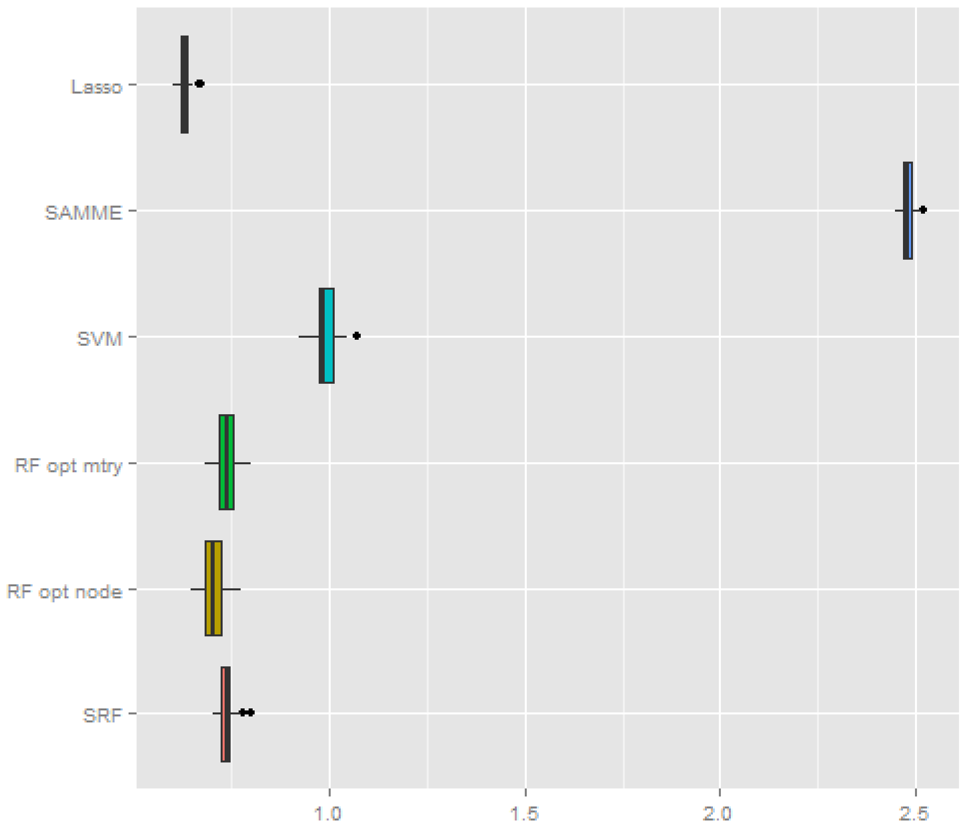

Figure 31: *Cross validated Brier Score values (× 100) over 50 independent replications on the x-axis* (Soybean data)

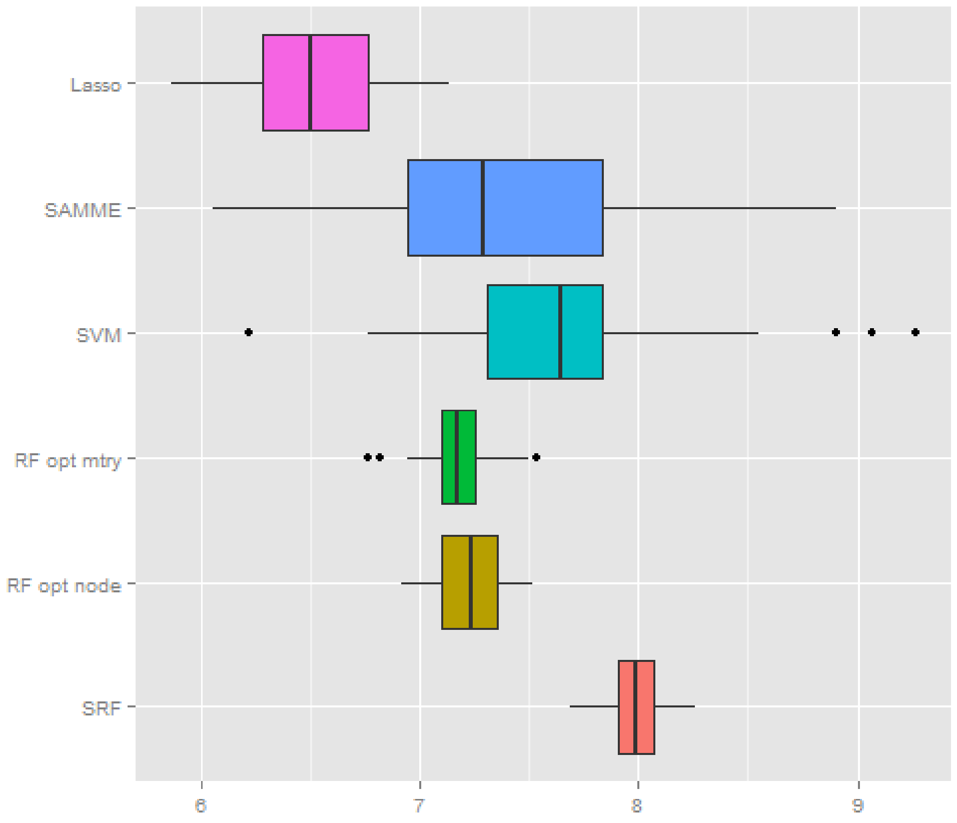

Figure 32: *Cross validated misclassification error rate (× 100) over 50 independent replications on the x-axis* (Soybean data)

Table 14: mean Brier Score, mean misclassification and mean CPU time in seconds (rounded) for the *Soybean* data, $n = 562$, $p = 35$, $K = 15$

Method	SRF	RF (node size)	RF (mtry)	SVM	SAMME	Lasso
Brier Score	0.74	0.70	0.74	0.99	2.48	0.63
Misclassification	7.98	7.23	7.17	7.65	7.41	6.53
CPU time:	250	278	30	291	2312	376

4.1.6 Spam

The famous „Spam“ data was gathered in 1998 by the Hewlett-Packard Labs. We utilized the „kernlab“ package (Karatzoglou et al. 2015), as it directly provides us with the data. Like the name already suggests, the task is to classify emails subjected to its contents either as spam or no spam. Overall, we have access to 57 features and respectable 4601 observations. 2788 of them are classified as spam and the remaining 1813 as no spam (i.e. a slightly patchy proportion).
Therefore, in case of its observations, „Spam“ is the largest data set we analyze. The first 48 variables represent words and their associated occurrence. These are for example „business“, „credit“, „money“ or „you“. Furthermore, variables that record the frequency of special characters such as „!“ or „$“ are part of the data.
Applying all six methods leads to the results shown in figure 33.

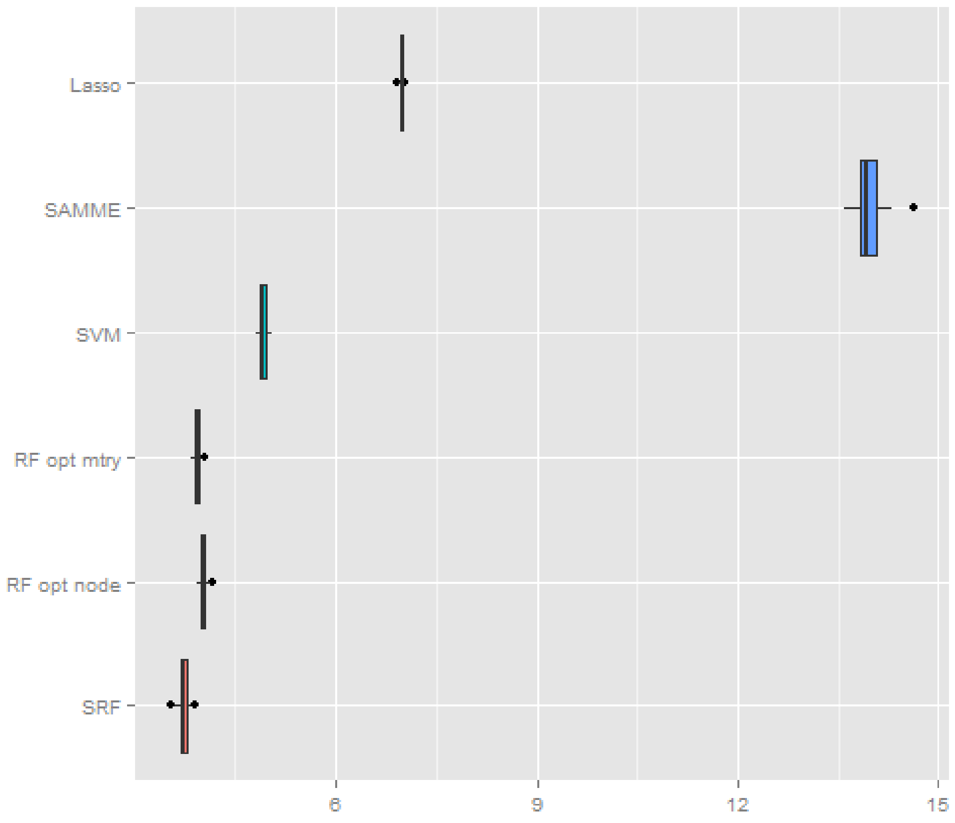

Figure 33: *Cross validated Brier Score values (× 100) over 50 independent replications on the x-axis* (Spam data)

Except for the SAMME, all algorithms produce very feasible Brier scores. Very notable are the small ranges for all techniques. Only SAMME experiences some small, but yet hardly worth mentioning, outliers.
The corresponding error rates show a very surprising finding (figure 34). SAMME does actually yield the best classification performance (even though its lead is very scarce).

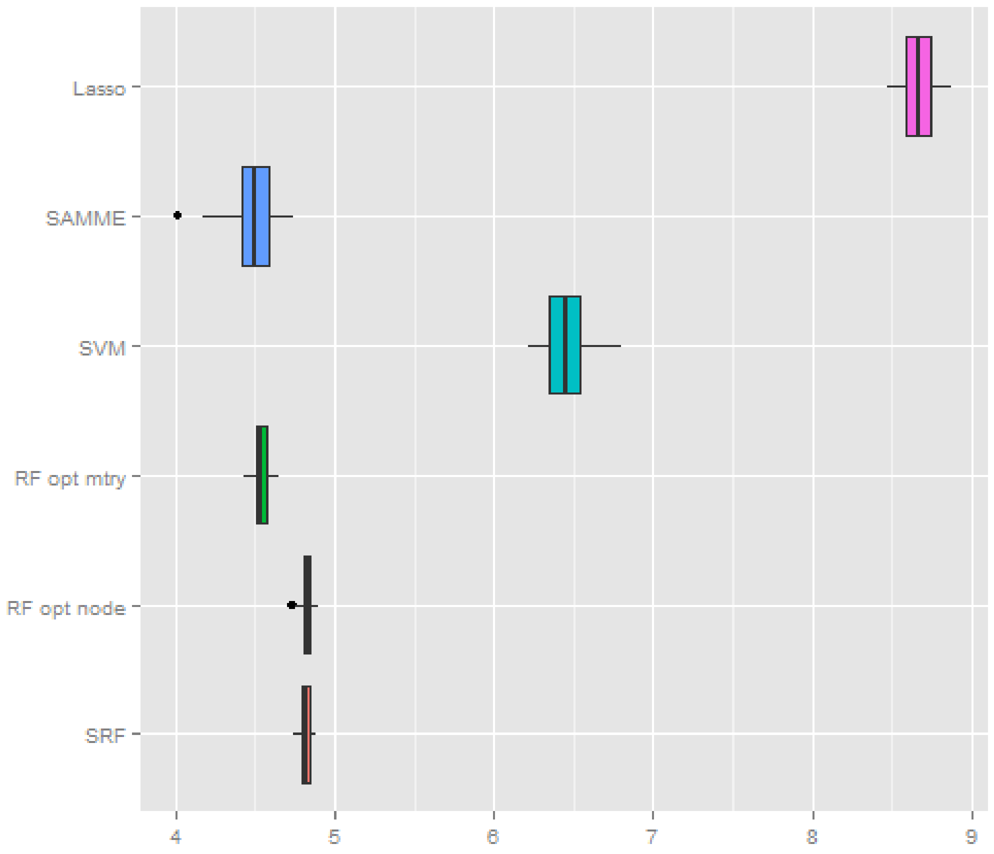

Figure 34: *Cross validated misclassification error rate (× 100) over 50 independent replications on the x-axis* (Spam data)

As we examine table 15, we see an tight contest between the random forests and the SAMME. Taking the computational times into account, the RF (mtry) performed best. Note that the SVM needed almost two hours for one replication. Second slowest performance was utilized by the lasso, which again had trouble finding convergence and hence iterated to its maximum value ($i = 100.000$).

Table 15: mean Brier Score, mean misclassification and mean CPU time in seconds (rounded) for the *Spam* data, $n = 4601$, $p = 57$, $K = 2$

Method	SRF	RF (node size)	RF (mtry)	SVM	SAMME	Lasso
Brier Score	3.73	4.03	3.92	4.92	13.95	6.99
Misclassification	4.82	4.83	4.54	6.46	4.50	8.66
CPU time:	2775	3195	554	7166	2666	4969

4.1.7 Vehicle

The last classification problem with real underlying data is called „Vehicle“. While the original data was generated at the Turing Institute (Glasgow) in 1986, it is also part of the „mlbench“ package (Leisch & Dimitriadou 2010). Its original purpose was to distinguish 3 dimensional objects within a 2 dimensional image. Four classes of detailed model cars, i.e. a double decker bus, a Cheverolet van, a Saab 9000 and an Opel Manta 400, were used for the experiment. Then, for each model car and a different camera angles, 120 images were taken, covering a full 360 degree rotation. All appending features were extracted by the HIPS (Hierarchical Image Processing System). Those are very subject-specific variables, like the circularity, the compactness, a hollow ratio or a skewness/kurtosis ratio about the major axis.
As we eye upon figure 35, the support vector machine demonstrates the best Brier score. Right behind the lasso and then very close the three random forest variations. Worst effort was yielded by the SAMME.
Our findings in figure 36 show a slightly altered circumstance. The SVM, the lasso and the synthetic random forest perform all three corresponding to their Brier score. For some reason, the remaining random forests (i.e. RF (opt mtry) and RF (opt node)) fell off. They do in fact perform even worse than the SAMME, whose Brier score was almost 50% higher.

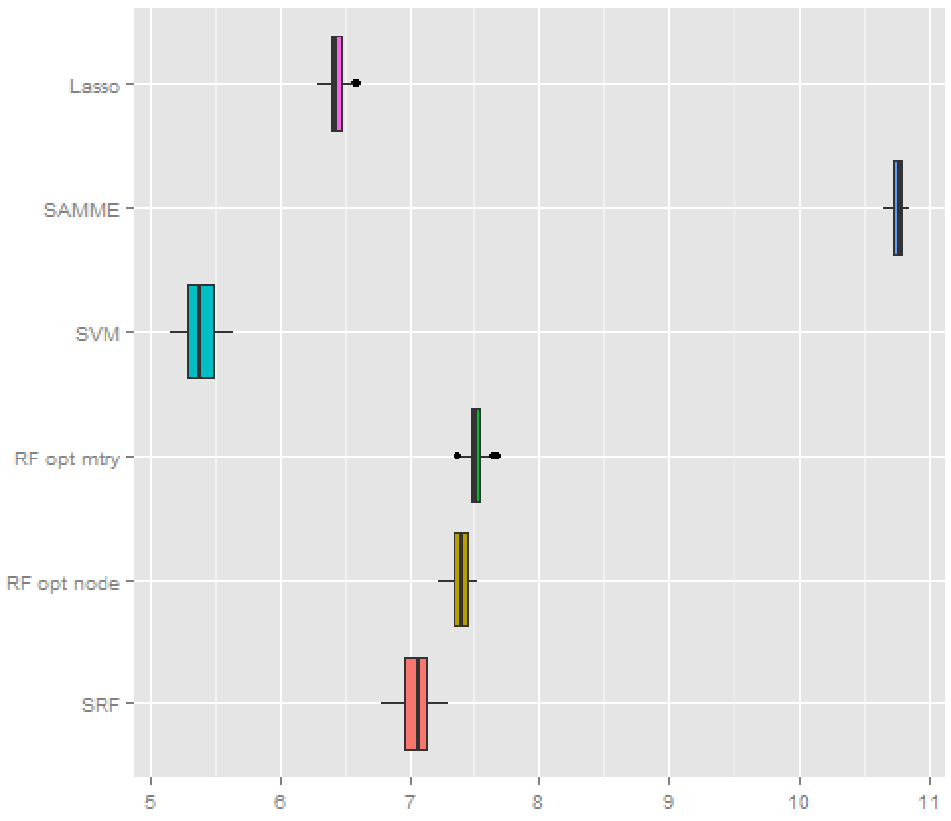

Figure 35: *Cross validated Brier Score values (× 100) over 50 independent replications on the x-axis* (Vehicle data)

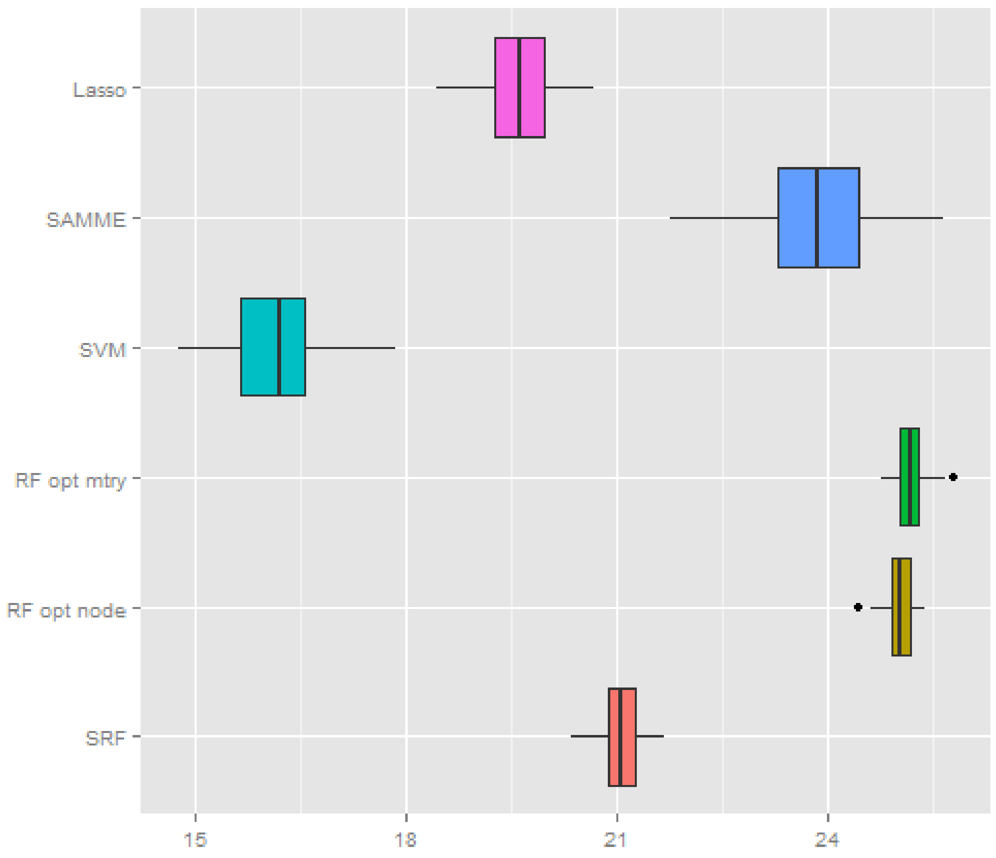

Figure 36: *Cross validated misclassification error rate (× 100) over 50 independent replications on the x-axis* (Vehicle data)

On average, or the support vector machine, more than five minutes were necessary to compute one replication. Slightly more time was required the lasso. Anyhow, it seems that no algorithm is able to deliver very good performance.

Table 16: mean Brier Score, mean misclassification and mean CPU time in seconds (rounded) for the $Vehicle$ data, $n = 846$, $p = 18$, $K = 4$

Method	SRF	RF (node size)	RF (mtry)	SVM	SAMME	Lasso
Brier Score	7.05	7.38	7.52	5.39	10.77	6.44
Misclassification	21.07	25.04	25.19	16.88	23.88	19.63
CPU time:	206	239	29	306	199	311

4.1.8 Shapes

Our final classification problem is synthetic and can be freely generated within the „mlbench“ (Leisch & Dimitriadou 2010) package.
Applying „Shape“, four structures (i.e. four classes) are projected into a two dimensional coordinate plane. In fact, those are a normally distributed object, a triangle, a square and a wave. Therefore, we have two features (abscissa and the ordinate). Furthermore, we decided to use 250 observations.
Plotting the corresponding outcome, we receive figure 37. Apparently, we observe no overlapping classes. Thus, a perfect allocation into the four groups seems to be achievable.

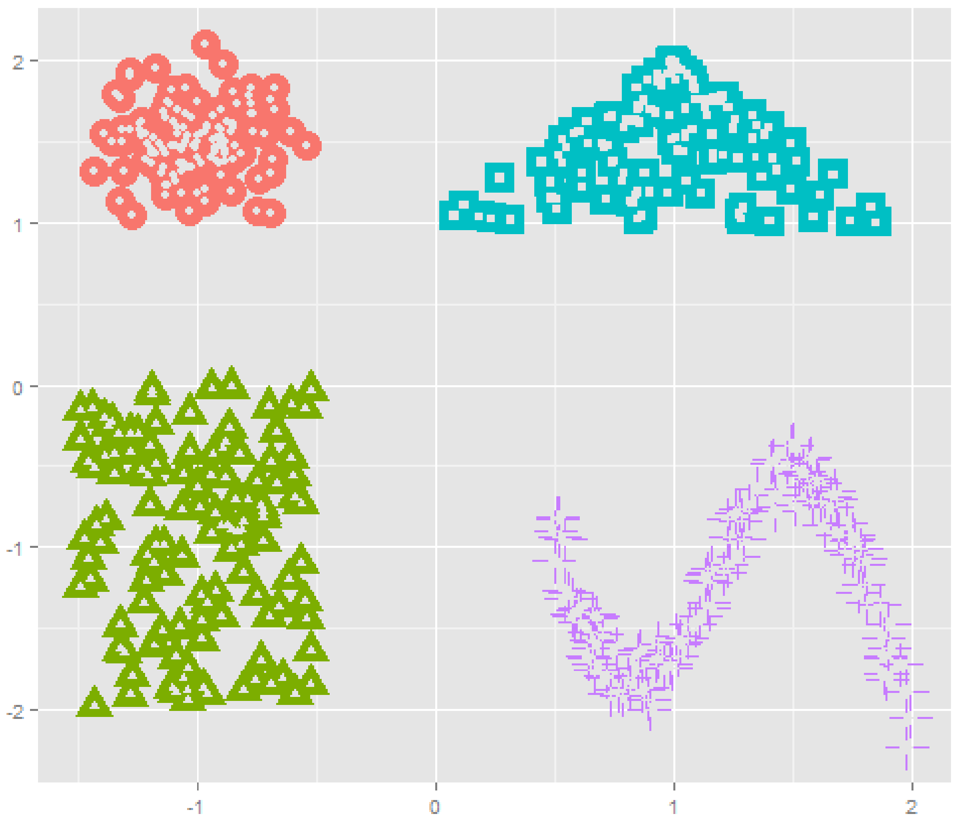

Figure 37: *Shapes: A a normally distributed object, a triangle, a square and a wave.*

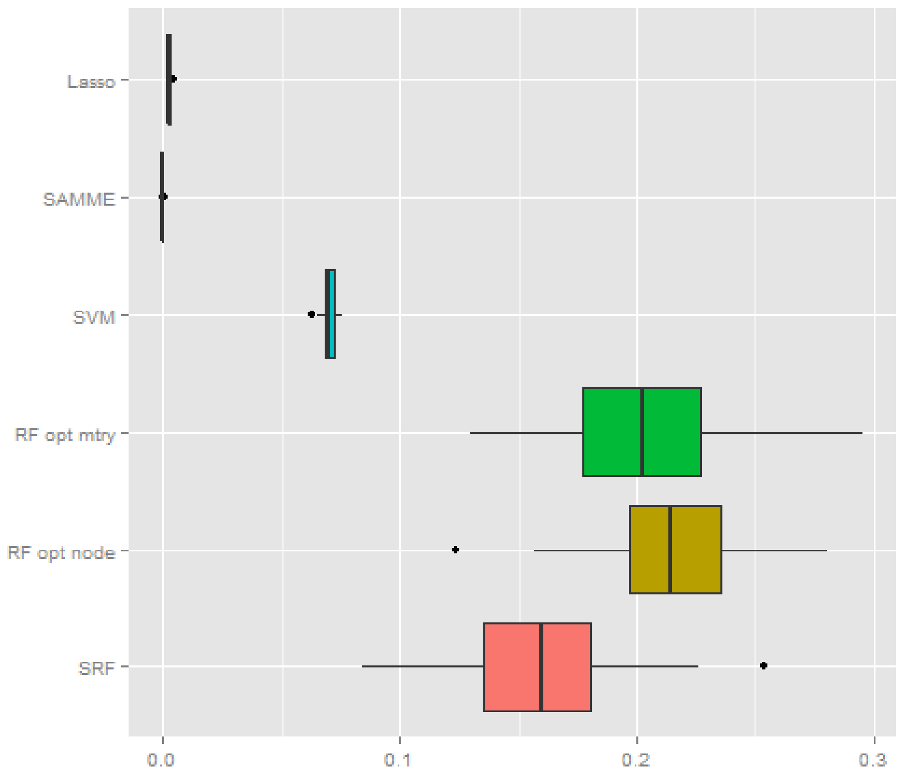

Figure 38: *Cross validated Brier Score values (× 100) over 50 independent replications on the x-axis* (Shape data)

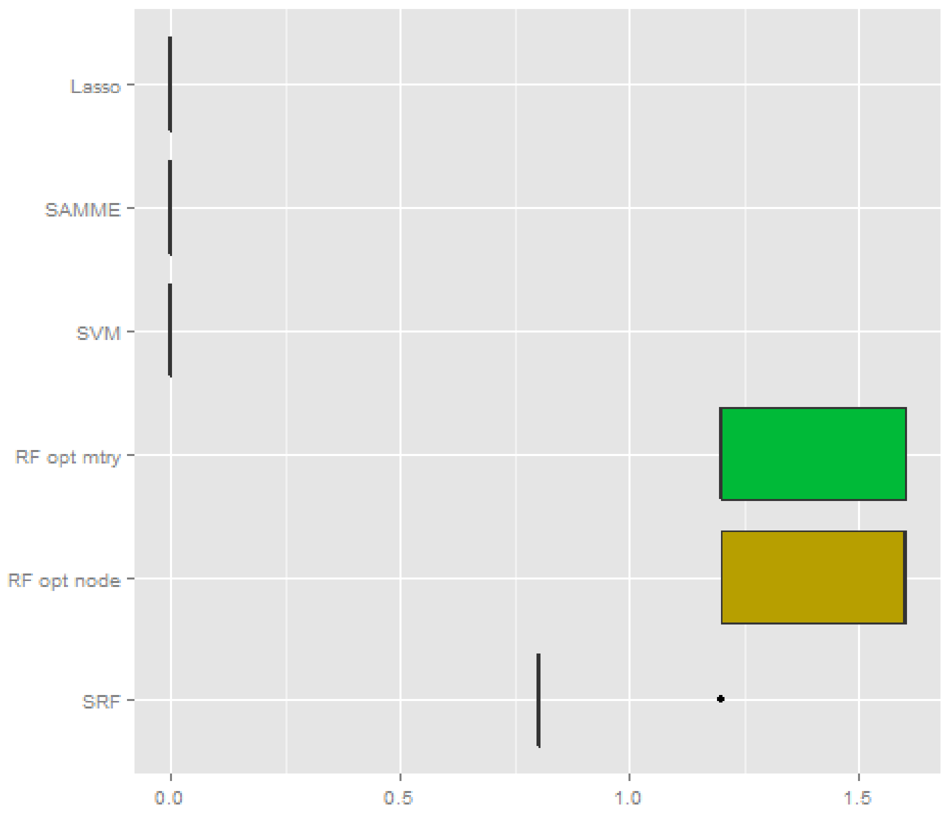

Figure 39: *Cross validated misclassification error rate (× 100) over 50 independent replications on the x-axis* (Shape data)

Figure 38 demonstrates excellent performance for all six methods. Random forests show very little struggle, but in this case we can consider it as complaining at a very high level, since no Brier score exceeds 0.3. For the lasso, the SAMME and the SVM, over all 50 replications, zero classification error rates were reported (see figure 39). SRF does slighty better than RF (opt node) and RF (opt mtry). SAMME requires most CPU time, while the RF (mtry) finishes one replication in only four seconds (see table 17).
Anyhow, for this problem, all algorithms deliver very convincing results.

Table 17: mean Brier Score, mean misclassification and mean CPU time in seconds (rounded) for the *Shapes* data, $n = 250$, $p = 2$, $K = 4$

Method	SRF	RF (node size)	RF (mtry)	SVM	SAMME	Lasso
Brier Score	0.16	0.21	0.20	0.07	0	0
Misclassification	0.81	1.43	1.39	0	0	0
CPU time:	34	38	4	35	112	69

5 Summary and Outlook

We have seen the results for eight regression and eight classification problems. From there on, an obvious advantage of random forests is their universal applicability. Our findings suggest, that optimizing the `node size` leads to slightly smoother estimates than the `mtry` value. The Highway data (chapter 3.1.4), a small set of only 39 observations, emphasises this claim very nicely. Thus, a further research interest might be to discover, if that behaviour holds especially for small data sets. On the hand, optimizing the `mtry` yields better results for problems with uncorrelated variables (see chapter 3.1.7 and 3.1.8). Admittedly, tuning the `node size` proved itself to be more difficult, as no R-package features a function for it. Therefore, as both tuning parameters show advantages concerning the structure of the data, we cannot elect an explicit winner. Synthetic forests could easily compete with primordial random forests. However, they were not superior to them. We found that data with uncorrelated variables completely corrupt their performance. Furthermore, in general they require much more computational time than the `mtry` optimized forest. Another drawback of synthetic forests is that they more or less disable the variable importance. Interpreting „predicted features“ is hardly possible (i.e. black box predictions). As a corollary of this, the question arises, if synthetic features can be applied on other methods. For instance, suppose a SVMs with different kernels or parameters. Each prediction could be used as such a synthetic feature. Summing up, the predictive performance of random forests was very convincing. Besides, tuning a random forest is not difficult, as one of their great benefits is their robustness against overfitting.
By contrast, SAMME was much more difficult to tune. Since its current R-implementation features no tuning ability itself, one has to carefully observe the process. The computation time varied from very low to extremely high. For the Breast Cancer problem (see Chapter 4.1.1), the algorithm needed on average only 118 seconds, compared to an SRF with 4784 seconds. However, in chapter 4.1.5 (Soybean), SAMME needed 2312 seconds on average, while the SRF finished one replication after only 250 seconds. While the actual classification showed persuasive results, the brier score was generally bad. Parental for this phenomenon might lie in the multiclass-functionality, which allows weak learners to exhibit error rates greater than 50%.
Gradient boosting, which was utilized only on regression problems, revealed itself to be nicely realisable. The tuning function of „mboost“ (Hofner et al. 2014) automatically determines the optimal stopping iteration, to avoid overfitting. While our implementation approach kept things simple, one could probably improve the performance by defining a greater set of weak learners (see algorithm 4 step 1). Those could be Markov random fields or radial basis functions, which require an in-depth knowledge. In addition, smaller values for the shrinkage parameter `mu` might also improve the results. Consequentially, that would greatly increase the CPU time, which brings us to the drawback of gradient boosting. For seven of the eight problems, the method required an extraordinary huge

computational time. For example, to handle the Syn250 problem (Chapter 3.1.8), one replication required on average 2431 seconds. In contrast, the even better performing lasso needed less than 1 second. Anyhow, unlike the random forests, the performance of gradient boosting for that problem was not negatively affected by the redundant variables. Beside the impairment of very high CPU time, one great advantage of the algorithm is, that we obtain prediction rules, that have similar interpretation potential like classic statistical models (i.e. no black box predictions).
Next, we would like to review the support vector machine. With respect to the theory behind it, tuning appears to be very difficult. In particular, when it comes to the choice of the right kernel, a lot of knowledge is necessary. Nevertheless, we were able to achieve predominantly good performance by applying the radial basis kernel. Tuning can then be accessed by utilizing a grid search over a sequence of values for the `cost` and corresponding `gamma` parameter. Unfortunately, that results in major computing time. Strikingly high values were observed for the Spam data (Chapter 4.1.6). On average, to complete one replication, 7166 seconds were mandatory. In comparison the RF (mtry) needed only 554 seconds, while it also yielded a better outcome (for both, Brier score and misclassification). Still notably four time in regression, and two times in classification, the SVM accomplished the best performance. Maybe the greatest disadvantage of a support vector machine, is the fact that predictions are completely black box and hence lack of any explanatory characteristic.
For our remaining method, the lasso, implementation revealed itself to be very easy. Functions of the „glmnet“ package (Friedman et al. 2015) automatically accessed the best value for the tuning parameter `lambda`. Eye-catching, we observed extremely fast computational time for regression (i.e. <2 seconds on average for one replication on all eight problems). However, for all five real data sets as well as the synthetic peak function, the lasso showed the worst performance. It seems that the lasso needs a special kind of data before it is able to reveal its potential. For that reason, we introduced the artificial data sets with redundant variables. Finally the lasso was able to show its strength, as it produced the best performance.
A much more varying outcome was exhibited in classification. For the Soybean problem (Chapter 4.1.5), the lasso yielded the best Brier score as well as the lowest classification error rate. These variations in the performance came along with changes in the computational time. It seems that, the higher the number of features in relation to the number of observations, the faster lasso will execute its computations. Consider for instance the Colon data (Chapter 4.1.2). While we have 2000 features for 62 observations, the lasso needed only 14 seconds for one replication. By contrast, the SVM with similar Brier score and classification error, required 3184 seconds on average for one replication. The opposite was observed for the Spam data, with many observations and low features, where one replication needed 4969 seconds on average.
Altogether, it might not be surprising that all techniques have their right to exist. They

all have their individual advantages as well as drawbacks. Deciding which method fits best on the problem at hand strongly depends on its composition and the corresponding intention. If someone wants to execute image mining, for example classifying faces to people, it obviously does not matter which variables accounts for the decision. But if the purpose is to find the cause of an effect, we obviously do not want to employ a support vector machine. While the accuracy of the prediction is in most cases fundamental, when dealing with high dimensional data, the computational time can become a serious issue. Thus, the lasso can be applied to detect important variables and shrink the redundant ones down towards zero. Another prospective approach is the combination of methods in a post-processing way. Sometimes, random forests and boosting compute very large ensembles of trees (i.e. thousands of trees). Many of them are very similar and hardly improve the predictor. So the lasso can be utilized to select a smaller subset of them before the final ensemble is constituted. Hastie et al. (2009*c*) show that this procedure can help oftentimes to improve the classifier.
In order to reproduce our results, we attached an USB-Stick with the applied code.

List of Figures

List of Tables

Bibliography

Alfaro, E., Gamez, M., Garcia, N. & Guo, L. (2014), *adabag: Applies Multiclass AdaBoost.M1, SAMME and Bagging*. R package version 4.0.
URL: *https://cran.r-project.org/web/packages/adabag/index.html*

Alon, U., Barkai, N., Notterman, D. A., Gish, K., Ybarra, S., Mack, D. & Levine, A. J. (1999), 'Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays', *Proceedings of the National Academy of Sciences* **96**(12), 6745–6750.
URL: *http://www.pnas.org/content/96/12/6745.abstract*

Breiman, L. (1996), 'Bagging predictors', *Mach. Learn.* **24**(2), 123–140.
URL: *http://dx.doi.org/10.1023/A:1018054314350*

Breiman, L. (2001), 'Random forests', *Machine Learning* **45**(1), 5–32.
URL: *http://dx.doi.org/10.1023/A%3A1010933404324*

Breiman, L. & Friedman, J. H. (1985), 'Estimating optimal transformations for multiple regression and correlation', *JASA* (80), 580–598.
URL: *https://users.soe.ucsc.edu/ draper/eBay-Google-2013-breiman-friedman-1985.pdf*

Breiman, L., Friedman, J., Stone, C. J. & Olshen, R. (1984), *Classification and Regression Trees*, The Wadsworth and Brooks-Cole statistics-probability series Wadsworth statistics/probability series, Taylor Francis.

Bühlmann, P. & Hothorn, T. (2007), 'Boosting algorithms: Regularization, prediction and model fitting', *Statist. Sci.* **22",**(4), 477–505.
URL: *http://dx.doi.org/10.1214/07-STS242*

Cortes, C. & Vapnik, V. (1995), 'Support-vector networks', *Machine Learning* **20**(3), 273–297.
URL: *http://dx.doi.org/10.1023/A%3A1022627411411*

Croissant, Y. (2015), *Ecdat: Data Sets for Econometrics*. R package version 0.2-9.
URL: *https://cran.r-project.org/web/packages/Ecdat/index.html*

Fox, J., Weisberg, S., Adler, D., Bates, D., Baud-Bovy, G., Ellison, S., Firth, D., Friendly, M., Gorjanc, G., Graves, S., Heiberger, R., Laboissiere, R., Monette, G., Murdoch, D., Nilsson, H., Ogle, D., Ripley, B., Venables, W., Zeileis, A. & R-Core (2015), *car: Companion to Applied Regression*. R package version 2.0-26.
URL: *https://cran.r-project.org/web/packages/car/index.html*

Freund, Y. & Schapire, R. (1995), A desicion-theoretic generalization of on-line learning and an application to boosting, *in* P. Vitányi, ed., 'Computational Learning Theory', Vol. 904 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 23–37.
URL: *http://dx.doi.org/10.1007/3-540-59119-2*$_1$66

Friedman, J. H. (2001), 'Greedy function approximation: A gradient boosting machine.', *Ann. Statist.* **29**(5), 1189–1232.
URL: *http://dx.doi.org/10.1214/aos/1013203451*

Friedman, J., Hastie, T., Simon, N. & Tibshirani, R. (2015), *glmnet: Lasso and Elastic-Net Regularized Generalized Linear Models*. R package version 2.0-2.
URL: *https://cran.r-project.org/web/packages/glmnet/index.html*

Gorman, R. P. & Sejnowski, T. J. (1988), 'Analysis of hidden units in a layered network trained to classify sonar targets'.
URL: *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.299.8959rep=rep1type=pdf*

Hastie, T. (2003), Boosting.
URL: *http://web.stanford.edu/ hastie/TALKS/boost*

Hastie, T. & Tibshirani, R. (1986), 'Generalized additive models', *Statist. Sci.* **1**(3), 297–310.
URL: *http://dx.doi.org/10.1214/ss/1177013604*

Hastie, T. & Tibshirani, R. (2014), Statistical learning.
URL: *lagunita.stanford.edu/courses/HumanitiesScience/StatLearning/Winter2014/about*

Hastie, T., Tibshirani, R. & Friedman, J. (2009*a*), Additive models, trees, and related methods, *in* 'The Elements of Statistical Learning', Springer Series in Statistics, Springer New York, pp. 295–336.
URL: *http://dx.doi.org/10.1007/978-0-387-84858-7*$_9$

Hastie, T., Tibshirani, R. & Friedman, J. (2009*b*), Boosting and additive trees, *in* 'The Elements of Statistical Learning', Springer Series in Statistics, Springer New York, pp. 337–387.
URL: *http://dx.doi.org/10.1007/978-0-387-84858-7*$_1$0

Hastie, T., Tibshirani, R. & Friedman, J. (2009*c*), Ensemble learning, *in* 'The Elements of Statistical Learning', Springer Series in Statistics, Springer New York, pp. 605–624.
URL: *http://dx.doi.org/10.1007/978-0-387-84858-7*$_1$6

Hastie, T., Tibshirani, R. & Friedman, J. (2009*d*), Model assessment and selection, *in* 'The Elements of Statistical Learning', Springer Series in Statistics, Springer New York, pp. 219–259.
URL: *http://dx.doi.org/10.1007/978-0-387-84858-7*$_7$

Hastie, T., Tibshirani, R. & Friedman, J. (2009*e*), Random forests, *in* 'The Elements of Statistical Learning', Springer Series in Statistics, Springer New York, pp. 587–604.
URL: *http://dx.doi.org/10.1007/978-0-387-84858-7*$_1$5

Hastie, T., Tibshirani, R. & Friedman, J. (2009*f*), Support vector machines and flexible discriminants, *in* 'The Elements of Statistical Learning', Springer Series in Statistics, Springer New York, pp. 417–458.
URL: *http://dx.doi.org/10.1007/978-0-387-84858-7*$_1$2

Hofner, B., Mayr, A., Robinzonov, N. & Schmid, M. (2014), 'Model-based boosting in r: a hands-on tutorial using the r package mboost', *Computational Statistics* **29**(1-2), 3–35.
URL: *http://dx.doi.org/10.1007/s00180-012-0382-5*

Hothorn, T., Buehlmann, P., Kneib, T., Schmid, M., Hofner, B., Sobotka, F. & Scheipl, F. (2015), *mboost: Model-Based Boosting.* R package version 2.5-0.
URL: *https://cran.r-project.org/web/packages/mboost/index.html*

Ishwaran, H. & Kogalur, U. B. (2015), *randomForestSRC: Random Forests for Survival, Regression and Classification.* R package version 1.6.1.
URL: *https://cran.r-project.org/web/packages/randomForestSRC/index.html*

Ishwaran, H. & Malley, J. (2014), 'Synthetic learning machines', *BioData Mining* **7**(1).
URL: *http://dx.doi.org/10.1186/s13040-014-0028-y*

Karatzoglou, A., Smola, A. & Hornik, K. (2015), *kernlab: Kernel-Based Machine Learning Lab.* R package version 0.9-22.
URL: *https://cran.r-project.org/web/packages/kernlab/index.html*

Kneib, T. & Hothorn, T. (2008), Componentwise boosting for generalised regression models.
URL: *https://www.r-project.org/conferences/useR-2008/slides/Kneib+Hothorn.pdf*

Leisch, F. & Dimitriadou, E. (2010), *mlbench: Machine Learning Benchmark Problems.* R package version 2.1-1.
URL: *https://cran.r-project.org/web/packages/mlbench/index.html*

Liaw, A. & Wiener, M. (2014), *randomForest: Breiman and Cutler's random forests for classification and regression.* R package version 4.6-10.
URL: *https://cran.r-project.org/web/packages/randomForest/index.html*

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

Meyer, D., Dimitriadou, E., Hornik, K., Weingessel, A., Leisch, F., Chang, C.-C. & Lin, C.-C. (2015), *e1071: Misc Functions of the Department of Statistics, Probability Theory*

Group. R package version 1.6-7.
URL: *https://cran.r-project.org/web/packages/e1071/index.html*

Milborrow, S. (2015), *rpart.plot: Plot rpart Models. An Enhanced Version of plot.rpart.* R package version 1.5.2.
URL: *http://CRAN.R-project.org/package=rpart.plot*

Prettenhofer, P. (2014), Gradient boosted regression trees.
URL: *https://docs.google.com/viewer?a=vpid=sitessrcid=ZGVmYXVsdGRvbWF pbnxwZXRlcnByZXR0ZW5ob2ZlcnxneDozNzE4NTUzMDE1NjcyZTM3*

R Core Team (2014), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria.
URL: *http://www.R-project.org/*

Schapire, R. E. (1990), 'The strength of weak learnability', *Machine Learning* **5**(2), 197–227.
URL: *http://dx.doi.org/10.1007/BF00116037*

Schapire, R. E., Freund, Y., Bartlett, P. & Lee, W. S. (1998), 'Boosting the margin: a new explanation for the effectiveness of voting methods', *Ann. Statist.* **26**(5), 1651–1686.
URL: *http://dx.doi.org/10.1214/aos/1024691352*

Segal, M. (2004), 'Machine learning benchmarks and random forest regression', *Machine Learning* .
URL: *http://escholarship.org/uc/item/35x3v9t4*

Therneau, T., Atkinson, B. & Ripley, B. (2014), *rpart: Recursive Partitioning and Regression Trees.* R package version 4.1-8.
URL: *http://CRAN.R-project.org/package=rpart*

Tibshirani, R. (1994), 'Regression shrinkage and selection via the lasso', *Journal of the Royal Statistical Society, Series B* **58**, 267–288.

Williams, G., Culp, M. V., Cox, E., Nolan, A., White, D., Medri, D., Waljee, A., & Ripley, B. (2015), *rattle: Graphical User Interface for Data Mining in R.* R package version 2.13.0.
URL: *https://cran.r-project.org/web/packages/rattle/*

Zhu, J., Zou, H., Rosset, S. & Hastie, T. (2006), Multi-class adaboost.
URL: *http://web.stanford.edu/ hastie/Papers/samme.pdf*

Zhu, J., Zou, H., Rosset, S. & Hastie, T. (2009), 'Multi-class adaboost', *Statistics and Its Interface* **2**(3).
URL: *http://web.stanford.edu/ hastie/Papers/SII-2-3-A8-Zhu.pdf*