Ludwig-Maximilians-Universität München
Institut für Statistik
Ludwigstr. 33
80539 München

MSc Statistics

Master Thesis

Parallel Boosting

Ronert Obst

ronert.obst@gmail.com

Munich, November 1, 2013

Supervisors: Prof. Dr. Gerhard Tutz
Dr. Fabian Scheipl

I herewith declare that I have completed the present thesis independently making use only of the specified literature and aids. Sentences or parts of sentences quoted literally are marked as quotations; identification of other references with regard to the statement and scope of the work is quoted. The thesis in this form or in any other form has not been submitted to an examination body and has not been published.

....

Location, Date Signature

Abstract

Random forest is currently one of the prevalent algorithms for large datasets, since it easily scales by parallelization. Boosting is an iterative algorithm, and therefore harder to parallelize, but has a superior forecasting performance to random forests in many situations. Furthermore, boosting offers better interpretability. Current implementations of boosting do not scale to large datasets. I design and implement a more scalable version of gradient boosting using subsampling and parallelization techniques, both in a shared and distributed memory setting. To evaluate these new algorithms on predictive performance, variable selection capabilities and scalability, I run simulations and real data experiments. I make my implementations of these scalable gradient boosting algorithms available in two open source R-packages.

Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

1.1. Goal of this thesis

Boosting outperforms random forests in many applications, yet random forest is one of the prevalent methods in the machine learning community. One of the reasons for the prevalence of random forests is its scalability. Each tree in a random forest can be calculated independently, thus it is an "embarrassingly parallel" problem. Boosting on the other hand, is an iterative procedure, and therefore much harder to parallelize. I aim to make boosting as scalable—if not more scalable—as the random forests algorithm, while preserving its predictive performance and variable selection capabilities. I plan to do this in two settings:

1. Shared memory setting (one computer, multiple cores).

2. Distributed memory setting (a cluster of computers, each node possibly using multiple cores).

I achieve 1) by extending the `mboost` (Hothorn et al. 2012) R-package to fit the base learners in step 2 of algorithm 2 in parallel; implementing subsampling of the observations (with and without replacement); subsampling of the features (with and without replacement) and postprocessing using regularized regression. I achieve 2) by splitting the data into disjoint subsamples and distributing them over the cluster nodes. Alternatively, bootstrap samples can also be used. Each cluster node independently fits a boosting model (possibly using shared memory parallelization), which is then combined into an ensemble of ensembles. Optionally, the ensemble components can be postprocessed by (regularized) regression on their out-of-sample predictions to weigh them by their predictive performance. I implement the distributed memory algorithm I call ParBoost in a dedicated R-package of the same name (Obst 2013).

I demonstrate the predictive performance, scalability and variable selection capabilities of these variations of gradient boosting in a variety of simulations and real data experiments. To the best of my knowledge, `parboost` and `isleboost` are the first open source software packages to implement parallel boosting in a distributed and shared memory setting respectively.

1.2. Overview of this thesis

First, I will motivate boosting in section 1.3.1, followed by a short history of boosting in section 1.3.2 and an introduction to gradient boosting in section 1.3.3. In section 1.4, I talk about previous work on scaling up boosting. Section 2 presents the theory of ParBoost and ISLEBoost. I test the speedup of ParBoost and ISLEBoost in section 3.1. In section 3.2, I present the results of the simulations designed to examine the predictive performance of ParBoost and ISLEBoost,

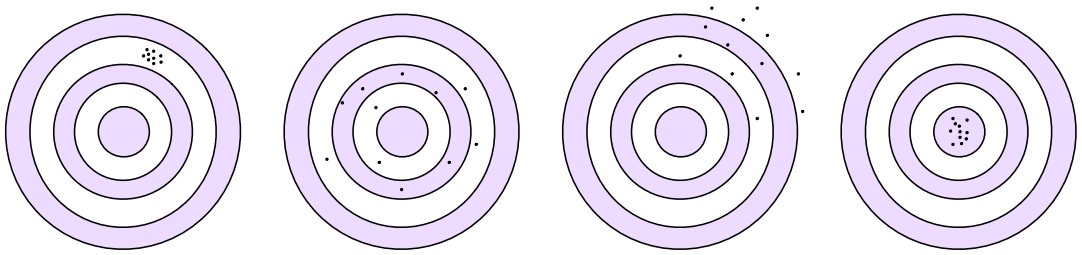
compared to ordinary gradient boosting and random forests. I discuss the simulation results for the variable selection capabilities of ParBoost and ISLEBoost, compared to ordinary gradient boosting, in section 3.3. Section 4 looks at the results of the real data experiments. Lastly, I discuss the implications of this thesis in section 5. Appendix A gives a short introduction to parallel and distributed computing and the electronic supplement to this thesis is explained in appendix B.

1.3. Boosting

1.3.1. Motivation: bias-variance tradeoff

Why use boosting over simpler models, such as generalized linear models (GLMs) or generalized additive models (GAMs)? In practice, the goal of predictive models is to generate accurate *out-of-sample* forecasts. To do this, the learned model must not only fit the training data well, but it also has to *generalize* well to new data. To achieve this for a given model, the main parameter we can vary is *model complexity*. Model complexity influences the bias and variance of a model and in turn, its expected *generalization error*. To explain this, I borrow an analogy from Moore, McCabe, and Craig (2010):

> *We can think of the true value of the population parameter as the bull's-eye on a target, and of the sample statistic as an arrow fired at the bull's-eye. Bias and variability [variance] describe what happens when an archer fires many arrows at the target. Bias means that the aim is off, and the arrows land consistently off the bull's-eye in the same direction. The sample values do not center about the population value. Large variability [variance] means that repeated shots are widely scattered on the target. Repeated samples do not give similar results but differ widely among themselves. Figure 1 shows this target illustration of the two types of error.*



High bias, low variance | Low bias, high variance | High bias, high variance | Low bias, low variance

FIGURE 1: Bias and variability in shooting arrows at a target. Bias means the archer systematically misses in the same direction. Variability [variance] means that the arrows are scattered. (Dartboard analogy, Moore, McCabe, and Craig (2010))

Increasing the complexity of the model increases the variance and reduces the bias. Decreasing the complexity reduces variance but increases bias. Since both a low variance and low bias are desirable, there is a *tradeoff* between the two (Geman, Bienenstock, and Doursat 1992). The difficulty of minimizing the expected generalization error lies in finding the sweet-spot of this bias-variance tradeoff[1], which is depicted in fig. 2.

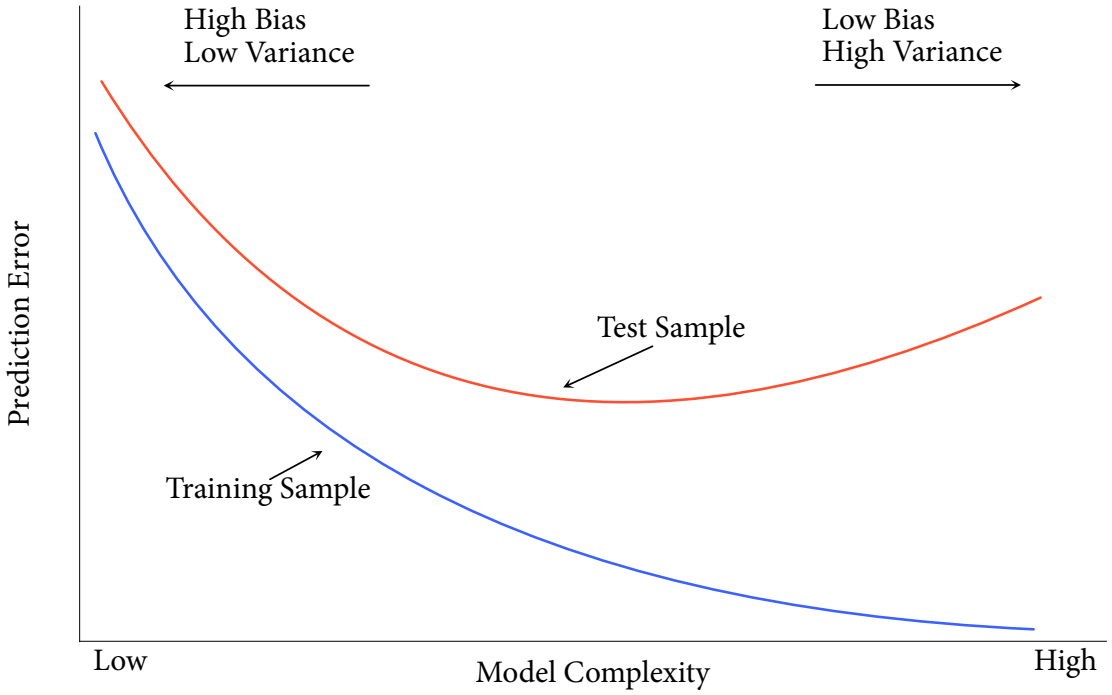


FIGURE 2: Test and training error as a function of model complexity (source: Hastie, Tibshirani, and J. H. Friedman 2009)

Boosting regulates model complexity in two ways:

1. By shrinking the coefficients towards 0 (regularization)

2. and by selecting a subset of variables determined by their predictive accuracy (variable selection).

By optimizing model complexity to minimize the expected generalization error, boosting can achieve better out-of-sample performance than GLMs and GAMs.

[1] See Geman, Bienenstock, and Doursat (1992) for a detailed and technical discussion of the bias-variance tradeoff.

1.3.2. History: AdaBoost

Robert E. Schapire and Yoav Freund—the inventors of the first boosting algorithm *AdaBoost*—describe their idea as follows (Schapire and Freund 2012):

> *[…] boosting, an approach to machine learning based on the idea of creating a highly accurate prediction rule by combining many relatively weak and inaccurate rules.*

This principle is best explained by an analogy to the show "Who Wants to be a Millionaire?" Whenever a contestant is unsure of an answer, he or she can use a "lifeline" to help answer the question. One of the lifelines is called "ask the audience," where the audience is polled for the answer, and then the results are displayed on the contestant's screen (e.g. 54 % of audience members chose answer a), 21 % chose b), and so on). This lifeline is known for its accuracy, even though individual audience members have probably not trained for the show and are, on average, no more intelligent than the general population. So polling a single random individual from the audience would be of little value to a contestant. Yet, aggregating all of the votes gives highly accurate answers. This phenomenon is called *the wisdom of the crowd*.

Boosting tries to make use of this phenomenon by combining "weak" *base learners* into an *ensemble*. Examples of base learners are:

- Componentwise linear least squares;
- B-Splines;
- Trees (or tree stumps).

A large number of these base learners are then aggregated into an ensemble, and in the case of a categorical response, a majority vote is conducted to determine the predicted class. Real-valued data is predicted by taking the average of all forecasts of the base learners. This corresponds to the audience voting (or giving an estimate in the case of real-valued data) in our analogy. Weak base learners are preferred to avoid overfitting. Picture a deep neural network as a base learner—it is clear how iteratively fitting this complex algorithm would lead to a woefully overfit model.

In each round of boosting, the algorithm fits all the base learners to the given data and chooses the one that minimizes the given loss criterion.

> *However, if the base learner is simply called repeatedly, always with the same set of training data, we cannot expect anything interesting to happen; instead, we expect the same, or nearly the same base classifier to be produced over and over again, so that little is gained over running the base learner just once. This shows, that the boosting algorithm, if it is to improve on the base learner, must in some way manipulate the data that feeds to it.*
>
> –Schapire and Freund (2012)

Boosting achieves this manipulation by using the residuals of the previous round as the response for the current round. In this way, observations that are hard to forecast get a larger and larger weight, until their forecasts become more accurate. Tukey (1977) almost discovered boosting three decades earlier: he proposed *Twicing*, which was essential L_2Boosting (table 1) for just two iterations. He did not pursue the idea further though.

Freund and Schapire (1996) called the first boosting algorithm *AdaBoost* (algorithm 1).

> *AdaBoost proceeds in rounds or iterative calls to the base learner. For choosing the training sets provided to the base learner on each round, AdaBoost maintains a distribution over the training examples. The distribution used on the m-th round is denoted D_m, and the weight it assigns to training example i is denoted by $D_m(i)$. Intuitively, this weight is a measure of the importance of correctly classifying example i on the current round. Initially, all weights are set equally, but on each round, the weights of incorrectly classified examples are increased so that, effectively, hard examples get successively higher weight, forcing the base learner to focus its attention on them.*
>
> – Schapire and Freund (2012)

Using $\alpha_m = \frac{1}{2}\ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$ (step 4, algorithm 1), AdaBoost gives larger weights to more accurate classifiers. A downside of AdaBoost is its exponential loss function (table 1), which can give outliers a disproportionate influence[2].

1.3.3. Gradient Boosting

Breiman (Breiman (1998) and Breiman (1999b)) was the first to realize that AdaBoost is a functional gradient descent algorithm. J. Friedman, Hastie, and Tibshirani (2000) then linked AdaBoost to additive basis expansion. J. H. Friedman (2001) and Bühlmann and B. Yu (2003) further

[2]The logistic loss would be more robust for example (table 1).

Algorithm 1: AdaBoost (Freund and Schapire 1996)

Given: $(x_1, y_1) \ldots (x_n, y_n)$ where $x_i \in X,\ y_i \in \{-1, +1\}$
Initialize weights $D_0(i) = 1/n$ for $i = 1, \ldots, n$
for m *in* $1 : m_{stop}$ **do**

1. Fit base learners to the weighted data using Distribution D.
2. Get weak hypothesis $f_m : X \to \{-1, +1\}$
3. Select f_m to minimize the weighted error:

$$\epsilon_m = \mathbb{P}\left[f_m(x_i) \neq y_i\right].$$

4. Choose $\alpha_m = \frac{1}{2}\ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$.
5. Update the weights for $i = 1, \ldots n$

$$\begin{aligned} D_{m+1}(i) &= \frac{D_m(i)}{Z_m} \times \begin{cases} e^{-\alpha_m} & f_m(x_i) = y_i \\ e^{\alpha_m} & f_m(x_i) \neq y_i \end{cases} \\ &= \frac{D_m(i)\exp\left(-\alpha_m y_i f_m(x_i)\right)}{Z_m} \end{aligned}$$

where Z_m is a normalization factor (chosen so that D_{m+1} will be a distribution).

end
Output the final hypothesis:

$$F(x) = \operatorname{sign}\left(\sum_{m=1}^{m_{\text{stop}}} \alpha_m f_m(x)\right).$$

generalized AdaBoost to a gradient descent algorithm that fits the base learners to the negative gradient of a variety of loss functions, resulting in an additive model—similar to a GAM—with variable selection and shrinkage. J. H. Friedman (2001) call this *stagewise, additive modeling* and Hastie, Tibshirani, and J. H. Friedman (2009) observe a close connection between boosting and the lasso. Algorithm 2 outlines the generic *componentwise gradient boosting* algorithm, and table 1 shows some typical loss functions for boosting.

Algorithm 2: Generic boosting (Bühlmann and Hothorn 2007)

Initialize: $F_0 \equiv 0$ or $F_0(\boldsymbol{x}) = \underset{c}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, c)$

for m *in* $1 : m_{stop}$ **do**

1. Calculate the residuals of the previous round $U_i = Y_i - F_{m-1}(X_i),\ i = 1, \ldots, n$
2. Fit base learners $f_m(\cdot)$ to working residuals U_i and pick the one that minimizes the loss function
3. Update the ensemble $F_m(\cdot) = F_{m-1}(\cdot) + \nu f_m(\cdot)$

end

TABLE 1: Various loss functions $\rho(y, f)$, population minimizers $f^*(x)$ and names of corresponding boosting algorithms; $p(x) = \mathbb{P}\left[Y = 1 \mid X = x\right]$ (Bühlmann and Hothorn 2007)

Range spaces	$\rho(y, f)$	$f^*(x)$	Algorithm
$y \in \{0, 1\}, f \in \mathbb{R}$	$\exp\left(-(2y-1)f\right)$	$\frac{1}{2}\log\frac{p(x)}{1-p(x)}$	AdaBoost
$y \in \{0, 1\}, f \in \mathbb{R}$	$\log_2\left(1 + e^{-2(2y-1)f}\right)$	$\frac{1}{2}\log\frac{p(x)}{1-p(x)}$	LogitBoost/BinomialBoosting
$y \in \mathbb{R}, f \in \mathbb{R}$	$\frac{1}{2}\left\lvert y - f\right\rvert^2$	$\mathbb{E}\left[Y \mid X = x\right]$	L_2Boosting

Usually, a fixed step-size[3] ν is chosen[4] ($\nu = 0.1$ or $\nu = 0.01$) and the model is optimized over the number of iterations m_{stop} to minimize the *generalization error* (fig. 2) using cross-validation or some information criterion[5]. Boosting is attractive because it performs regularization (due to $\nu < 1$), and at the same time, variable selection (due to early stopping). Efficient variable selection is especially important in high-dimensional situations ($p >> N$), when many variables are irrelevant. Underlying variable selection and regularization is the *bet on sparsity* principle:

> *Use a procedure that does well in sparse problems, since no procedure does well in dense problems.*
>
> – J. Friedman, Hastie, Rosset, et al. (2004)

[3] Also called *learning rate*.

[4] An alternative to a fixed step size ν, is to select ν using a line search for the greatest decrease in the loss function $\mathcal{L}(F)$.

[5] Estimation of the degrees of freedom in boosting remains an open question though, see Hastie (2007).

Variables remain unpenalized if the algorithm is initialized with them. E.g. boosting can be initialized with an unpenalized linear model and boosting then fits an additive nonlinear effect using componentwise smoothing splines as base learners.

1.4. Related work

Breiman (1999a) first introduced the idea of subsampling small "bites" from a large dataset, so they fit into memory. A predictor is fit on each of these bites, and then all of the predictors are "pasted" together into an ensemble. Fan, Stolfo, and Zhang (1999) propose two ways of scaling up boosting: 1) Subsampling of observations (without replacement) in each iteration of AdaBoost to speed up the algorithm and 2) Splitting the data into disjoint subsets over the observations and using a different subset in each iteration. C. Yu and Skillicorn (2001) partition the observations across available processors, and in each iteration of AdaBoost, the predictors are exchanged.

J. H. Friedman (2002) uses subsampling of the observations (without replacement) in each iteration of gradient boosting (step 2 of algorithm 2) and called it *stochastic gradient boosting*. "It is shown that both the approximation accuracy and execution speed of gradient boosting can be substantially improved by incorporating randomization into the procedure" (J. H. Friedman 2002). J. H. Friedman and Popescu (2003) observed that popular ensemble learning procedures—such as boosting—can be seen as high-dimensional numerical quadrature[6]. They called these methods *importance sampled learning ensembles* (ISLE). In this context, J. H. Friedman and Popescu (2003) proposed to *postprocess* ensembles by performing regularized regression on their components in conjunction with subsampling. Through simulations and real data experiments, they demonstrated that postprocessing can reduce the generalization error and variance of these learning ensembles.

Lazarevic and Obradovic (2002) develop a parallel version of AdaBoost for shared memory systems with a small number of processors. The authors also use a distributed version, where the data is split into disjoint subsets and distributed among cluster nodes. They find that:

> *Experimental results on several datasets indicate that the proposed boosting techniques can effectively achieve the same or even slightly better level of prediction accuracy than standard boosting when applied to centralized data, while the cost of learning and memory requirements are considerably lower.*
>
> – Lazarevic and Obradovic (2002)

But opposed to ParBoost, the classifiers are merged in each iteration, requiring a lot of communication and locking between the nodes.

Xie, Rojkova, and Pal (2009) bag boosting models to compete in the 2009 KDD[7] Cup. Pavlov,

[6] Integration.
[7] Knowledge discovery and data mining.

Gorodilov, and Brunk (2010) also present a combination of bagging and boosting: *BagBoo*. Through real data experiments, Pavlov, Gorodilov, and Brunk (2010) show that BagBoo has better predictive performance than bagging and is much more scalable than boosting.

> *We also emphasize that BagBoo is intrinsically scalable and parallelizable, allowing us to train order of half a million trees on 200 nodes in 2 hours CPU time and beat all of the competitors in the Internet Mathematics relevance competition sponsored by Yandex and be one of the top algorithms in both tracks of Yahoo ICML-2010 challenge.*
>
> – Pavlov, Gorodilov, and Brunk (2010)

BagBoo is similar to ParBoost in spirit, although it does not use postprocessing and it subsamples from observations *and* features without replacement instead of partitioning the data. Also, BagBoo only uses tree base learners and runs for short boosting spurts of 10–20 trees. They do not postprocess the ensemble. There is no publicly available software package that implements BagBoo. But Pavlov, Gorodilov, and Brunk (2010) show that combining boosting models trained on a subset of the data is a promising strategy to achieve the prediction accuracy of boosting, in combination with the scalability of bagging.

Palit (2012) develop a parallel boosting algorithm in the MapReduce (Dean and Ghemawat 2008) context for AdaBoost and LogitBoost for classification. They achieve parallelization in both time and space by splitting the data into subsets and fitting AdaBoost or LogitBoost on each subset. But their algorithms, AdaBoost.PL and LogitBoost.PL, are limited to classification and do not make use of the more general gradient boosting framework. AdaBoost.PL and LogitBoost.PL do not postprocess the final ensemble with (regularized) regression, and there is no publicly available implementation of their algorithms. BagBoo, AdaBoost.PL and LogitBoost.PL are much more scalable than traditional boosting, and Palit (2012) shows that they have a competetive predictive performance compared to their sequential versions.

Tyree et al. (2011) scale gradient boosted regression trees by constructing individual trees in parallel and Appel et al. (2013) speed up boosted decision trees by pruning underachieving features early. They train a model on a small subsample of the data first, and eliminate underachieving features. The authors prove that their algorithm produces identical trees as classical algorithms, and experimentally show that it is an order of magnitude faster for classification tasks. This approach is orthogonal to ParBoost and ISLEBoost, and can be used in conjunction with it for even greater speedup. Sapp, Laan, and Canny (2013) fit individual models on subsamples of the data and combine them into an ensemble, using a clever cross-validation scheme to weigh individual ensemble components. However, this scheme requires access to the whole data, and is thus not easily implemented in a distributed memory environment, where communication is the bottleneck.

2. Theory

2.1. Shared memory setting: ISLEBoost

J. H. Friedman and Popescu (2003) view boosting from the perspective of high-dimensional numerical quadrature, which helps to explain the effectiveness of stochastic subsampling and postprocessing (algorithm 4). Let the base learners $f(\boldsymbol{x};\gamma_m)$ be characterized by a parameter vector $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_M)$. "For example if the base learners are trees, then γ indexes the splitting variables, the split-points and the values in the terminal nodes (Hastie, Tibshirani, and J. H. Friedman 2009)." The learning problem can then be formulated as finding a high dimensional integral

$$F_M(\boldsymbol{x};\alpha) = \alpha_0 + \int \alpha(\boldsymbol{\gamma}) f(\boldsymbol{x};\boldsymbol{\gamma}_m)\, d\boldsymbol{\gamma}. \tag{1}$$

"Numerical quadrature amounts to finding a set of M evaluation points $\gamma_m \in \Gamma$ and corresponding weights α_m so that

$$F_M(\boldsymbol{x}) = \alpha_0 + \sum_{m=1}^{M} \alpha_m b(\boldsymbol{x};\gamma_m) \tag{2}$$

approximates $f(\boldsymbol{x})$ well over the domain of $\boldsymbol{x}$. Importance sampling amounts to sampling γ at random, but giving more weight to relevant regions of the space Γ (Hastie, Tibshirani, and J. H. Friedman 2009)."

Using only a single evaluation point $\boldsymbol{\gamma} \in \Gamma$ ($M = 1$), minimizing the prediction risk J amounts to

$$J(\boldsymbol{\gamma}) = \min_{\alpha_0,\alpha} \mathbb{E}\left[L(y, \alpha_0 + \alpha f(\boldsymbol{x},\boldsymbol{\gamma}))\right] \tag{3}$$

for some loss function L. Denote the optimal rule for ($M = 1$ iterations, eq. (2)) by

$$\boldsymbol{\gamma}^* = \operatorname*{argmin}_{\boldsymbol{\gamma}\in\Gamma} J(\boldsymbol{\gamma}). \tag{4}$$

This is akin to using a single decision tree or other base learner.

> *The assumption (hope) is that a collection of such evaluation points $\{\boldsymbol{\gamma}_m\}_1^M$, each with relatively small values of $J(\boldsymbol{\gamma}_m)$, will result in a integration rule with higher accuracy than with either a similar sized collection of points sampled with constant probability, or the best single point rule (eq. (4)). As evidenced by the success of bagging and random forests, this is often the case in practice.*
>
> – J. H. Friedman and Popescu (2003)

The goal is to sample $\boldsymbol{\gamma}$ such that points close to $\boldsymbol{\gamma}^*$ have a higher probability of being sampled than points further away. The corresponding distance measure is

$$d(\boldsymbol{\gamma},\boldsymbol{\gamma}^*) = J(\boldsymbol{\gamma}) - J(\boldsymbol{\gamma}^*). \tag{5}$$

The scale (width) of the density used for importance sampling (with a sampling probability density $r(\boldsymbol{\gamma})$) is then:

$$\sigma = \int_{\Gamma} d(\boldsymbol{\gamma}, \boldsymbol{\gamma}^*) r(\boldsymbol{\gamma})\, d\gamma. \tag{6}$$

Hastie, Tibshirani, and J. H. Friedman (2009) state that:

- *σ too narrow suggests too many of the $f(x; \boldsymbol{\gamma}_m)$ look alike, and similar to $f(\boldsymbol{x}; \boldsymbol{\gamma}^*)$;*
- *σ too wide implies a large spread in the $f(\boldsymbol{x}; \boldsymbol{\gamma}_m)$, but possibly consisting of many irrelevant cases.*

Figure 3 shows several potential sampling distributions $r(\gamma)$ characterized by a location and scale parameter. When choosing the width σ (eq. (6)), there is a tradeoff between minimizing

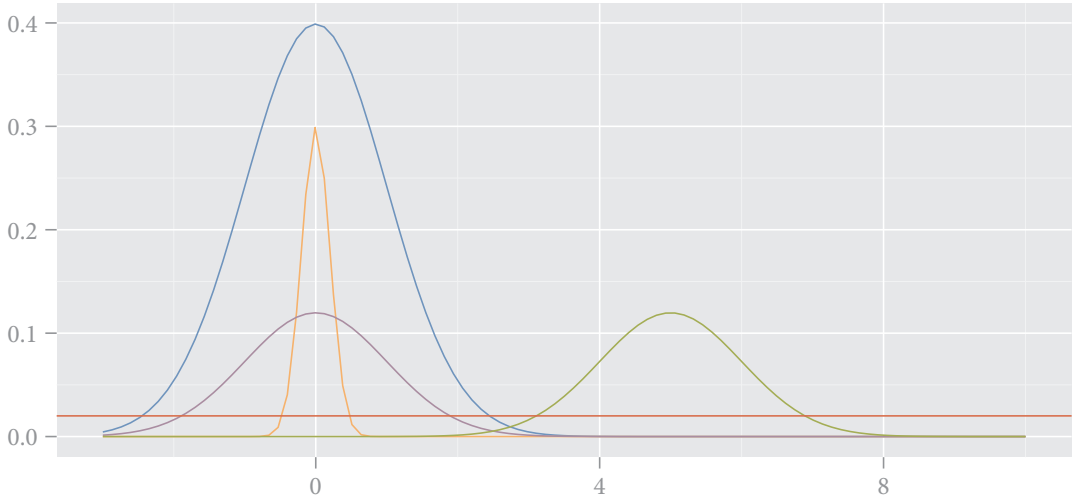


FIGURE 3: Potential sampling distributions $r(\gamma)$ characterized by a location and scale parameter. The blue density is the target $F(\gamma)$. The red proposal is almost constant with a very large scale, so few samples come from high probability regions of the target. The orange proposal has too narrow scale σ and thus samples few points in the outer regions of the target. The green proposal has the right scale, but the wrong location. The purple proposal has the right scale and location to efficiently integrate the target, since it covers the entire region of $F(\gamma)$ and has higher mass at high probability regions of the target $F(\gamma)$ (J. H. Friedman and Popescu 2003).

the individual bias of the base learners and the correlation between them (Breiman 2001). "A narrow sampling distribution (small σ (eq. (6))), produces base learners $\{f(\boldsymbol{x}; \boldsymbol{\gamma}_m)\}_1^M$ all of which having similar strength to the strongest one $f(\boldsymbol{x}; \boldsymbol{\gamma}^*)$, and thereby yielding similar highly correlated

predictions" (J. H. Friedman and Popescu 2003). This situation is akin to the narrow orange proposal in fig. 3. "A very broad $r(\boldsymbol{\gamma})$ (large σ) produces highly diverse base learners, most of which have larger values of $J(\boldsymbol{\gamma})$ (smaller strength) and less correlated predictions" (J. H. Friedman and Popescu 2003). This situation is analogous to the red proposal in fig. 3. The optimal width σ depends on the target function $F(\boldsymbol{x}^*)$.

J. H. Friedman and Popescu (2003) use *perturbation sampling* as a sampling strategy $r(\boldsymbol{\gamma})$ for importance sampling. Perturbation sampling works by perturbing the data in some way before fitting each base learner. The width σ is then determined by the amount of perturbation. Specifically, J. H. Friedman and Popescu (2003) use subsampling of the observations without replacement. The size of the subsample K, relative to the number of observations N, $\eta = K/N$, determines the width σ. "Reducing η increases the randomness, and hence the width σ. The parameter $\nu \in [0, 1]$ introduces *memory* into the randomization process; the larger ν, the more the procedure avoids $f(x; \gamma)$ similar to those found before" (Hastie, Tibshirani, and J. H. Friedman 2009).

Despite remarks from J. H. Friedman and Popescu (2003) and J. H. Friedman (2002) to the contrary, I found that in the context of boosting, setting $\eta < 1$ does not speed up the computation by N/K for most base learners, as one might initially expect (it does for non-sequential algorithms like bagging Breiman (1996)). Take piecewise linear base learners for example. The estimator for the coefficient vector is

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y}. \tag{7}$$

Setting $\eta = 1$,

$$(\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}' \tag{8}$$

stays constant over all boosting iterations for each base learner. Only the response vector $\boldsymbol{y}$ gets updated with the current working residuals. When setting $\eta < 1$, eq. (8) needs to be updated in every iteration, and may actually slow down computation. Updating the inverse $(\boldsymbol{X}'\boldsymbol{X})^{-1}$ using the Sherman-Morrison-Woodbury formula (Hager 1989), or using low rank updates for the Cholesky decomposition (Seeger 2007), is not efficient in this case either, since η is typically small (0.05–0.5). Denote the selected subsample in iteration m by S_m and the part of the data not in the subsample by $-S_m$. The product $\boldsymbol{X}_{-S_m} \cdot \boldsymbol{X}'_{-S_m}$ *excluding* the subsample of the observations has to be recalculated in every iteration for these updates, which is typically slower than calculating the inverse $(\boldsymbol{X}_{S_m}\boldsymbol{X}'_{S_m})^{-1}$ *including* the subsample when η is small. Additionally, out-of-sample predictions have to be computed in each iteration to calculate the working response, which further slows down computation.

J. H. Friedman and Popescu (2003) estimate the quadrature coefficients $\{\alpha_m\}_0^M$ with regularized regression on the response y using the selected base learners $\{f(\boldsymbol{x}; \alpha_m)\}_1^M$ as predictors:

$$\{\hat{\alpha}_m\}_0^M = \underset{\{\alpha_m\}_0^M}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, \alpha_0 + \sum_{m=1}^{M} \nu\alpha_m f(x_i; \gamma_m) \right) + \lambda \cdot \sum_{m=1}^{M} \operatorname{pen}(\alpha_m). \tag{9}$$

I included the Lasso (Tibshirani 1996), Ridge (Hoerl and Kennard 1970) and Elastic Net (Zou and Hastie 2005) penalty functions (pen, eq. (9)) in `isleboost` from the `glmnet` (J. Friedman, Hastie,

and Tibshirani 2010) package. The regularization parameter λ (eq. (9)) determines the degree of shrinkage and is optimized using cross-validation. ISLEBoost optimizes m_{stop} and λ sequentially to avoid a quadratic optimization procedure.

> *[…] the shrinkage implicit in eq. (9) has the important function of reducing bias of the coefficient estimates as well as variance. The importance sampling procedure selects predictors $f(\boldsymbol{x};\gamma_m)$ that have low empirical risk and thereby preferentially high values for their empirical partial regression coefficients $\hat{\alpha}_m$. If the multiple regression coefficients $\{\hat{\alpha}_m\}_0^M$ were estimated using data independent of that used to select ${\gamma_m}_1^M$, then using eq. (9) with $\lambda = 0$ would produce unbiased estimates (see Copas (1983)). However, using the same data for selection and estimation produces coefficient estimates that are biased towards high absolute values. The shrinkage […] helps compensate for this bias.*
>
> – J. H. Friedman and Popescu (2003)

Postprocessing gradient boosting is also a way to make the algorithm sparser, since it reduces the number of selected base learners. See for example Bühlmann and B. Yu (2006) and Bühlmann and Hothorn (2010) for other techniques to achieve more sparsity in boosting.

Algorithm 3: Generic ISLE ensemble generation (Hastie, Tibshirani, and J. H. Friedman 2009)

1. $F_0(\boldsymbol{x}) = \underset{c}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, c)$
2. For $m = 1$ to M do
   (a) $\gamma_m = \underset{\gamma}{\operatorname{argmin}} \sum_{i \in S_m(\eta)} L(y_i, F_{m-1}(x_i) + f(x_i;\gamma))$
   (b) $F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \nu f(\boldsymbol{x};\gamma_m)$
3. $\mathcal{T}_{\text{ISLE}} = \left\{ f(\boldsymbol{x};\gamma_1), f(\boldsymbol{x};\gamma_2), \ldots, f(\boldsymbol{x};\gamma_M) \right\}$.

I call the shared memory algorithm ISLEBoost, because I borrow most ideas from J. H. Friedman and Popescu (2003)). Specifically, ISLEBoost makes use of subsampling of observations and post-processing of the resulting ensemble. What sets ISLEBoost apart from the proposed modifications to gradient boosting in J. H. Friedman and Popescu (2003), is the ability to optionally subsample from features instead of observations, and improvements in computational efficiency: namely fitting individual base learners in parallel. It should be pointed out, that the postprocessing step for ISLEBoost (algorithm 4) uses the same distribution family that was used for boosting. This ensures that e.g. probability estimates for binomial responses are bounded by $[0, 1]$ and estimates for Poisson responses are counts. Postprocessing a boosting ensemble in this way has a somewhat similar structure to a single hidden layer, feed-forward neural network with a single output unit (see fig. 4).

Subsampling features instead of observations in every iteration also varies the width σ (eq. (6)), but it offers linear speedup, e.g. taking a 50 % subsample of features will speed up the algorithm

Algorithm 4: Generic ISLEBoost

Initialize: $F_0 \equiv 0$ or $F_0(\boldsymbol{x}) = \underset{c}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, c)$

for m *in* $1 : m_{stop}$ **do**

1. Optional: Subsample observations or features from complete data $d_m \subseteq D$ with or without replacement
2. Calculate residuals $U_i = Y_i - F_{m-1}(X_i) \; \forall \; X_i, Y_i \in d_m$
3. Fit base learners $f_m(\cdot)$ to working residuals U_i and pick the one that minimizes the loss function
4. Update model $F_m(\cdot) = F_{m-1}(\cdot) + \nu f_m(\cdot)$

end

Optional postprocessing: Regularized regression on ensemble components $f_m(\cdot)$ using an appropriate response function h:

$$F_m = h\left(\alpha_0 + \sum_{m=1}^{M} \nu \alpha_m f_m(\boldsymbol{x})\right)$$

$$\{\hat{\alpha}_m\}_0^M = \underset{\{\alpha_m\}_0^M}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, \alpha_0 + \sum_{m=1}^{M} \alpha_m \nu f(x_i; \gamma_m)\right) + \lambda \cdot \sum_{m=1}^{M} \operatorname{pen}(\alpha_m).$$

by approximately 100 %. Hsu et al. (2012) describe the tradeoff as "convergence time versus representation power." Subsampling features will learn restricted forms of linear predictors relative to what can be learned without subsampling (Hsu et al. 2012). The resulting predictor is somewhere between naïve Bayes and the unrestricted form, depending on the size of the subsample. The subsample of features can only account for feature correlation within the subsample, whereas naïve Bayes assumes complete independence. Postprocessing using regularized regression on the base learners can eliminate highly correlated features and alleviate some of the shortfall in representation power.

Subsampling features is vulnerable to datasets with only a few relevant features relative to the total number of features. For example, if only 5 % of the features are relevant, and 20 % of features are sampled in each iteration, it is easy to see that many selected base learners will be irrelevant too, crippling boosting's ability for variable selection. Postprocessing using regularized regression can help in this regard too, setting the weights of ineffective base learners to 0—but this is inefficient. If the problem is known to be sparse, it is best to avoid feature subsampling altogether. If the problem is dense though, feature subsampling can yield nearly identical results to ordinary boosting in a fraction of the time.

Using piecewise linear base learners for simplicity, gradient boosting has a computational complexity of

$$O\left(pn(M + \ln n)\right), \tag{10}$$

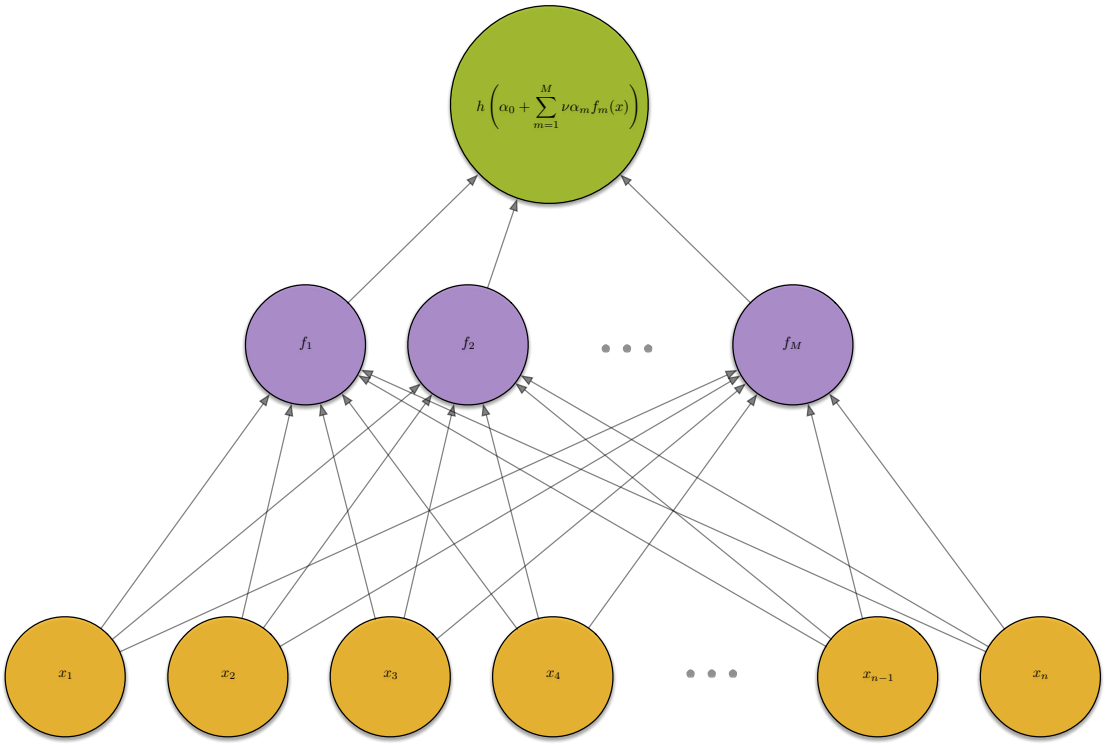


FIGURE 4: ISLEBoost network diagram with postprocessing. Note the similarity to a single hidden layer, feed-forward neural network with a single output unit.

with $O(pn)$ for finding the best regression function and $O(pn\ln n)$ for sorting the features (Palit 2012). Fitting the base learners on K cores reduces the computational complexity to

$$O\left(\frac{p}{K}n(M + pn\ln n)\right), \tag{11}$$

thus we should expect a linear speedup in the number of cores.

2.2. Distributed memory setting: ParBoost

The idea of ISLEBoost is to make boosting more scalable on a single computer with multiple cores, whereas ParBoost tries to scale boosting to a whole cluster of machines. The two algorithms can be combined of course: each node in the cluster may have multiple cores and can therefore use ISLEBoost to speed up the computation locally.

ParBoost works by partitioning the data into K samples. Boosting is then run on each $k \in K$ samples independently to minimize communication overhead, which is the bottleneck in a cluster setting. For smaller datasets, bootstrapping the original data and then fitting boosting models

Algorithm 5: ParBoost

Partition data into K subsamples or generate K bootstrap samples, $\{d_k \subseteq D\}_{k \in K}$
and send each sample d_k to a different cluster node.
for *k in K parallel* **do**
    Initialize: $F_0 \equiv 0$ or $F_0(\boldsymbol{x}) = \underset{c}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, c)$
    for *m in* $1 : m_{max}$ **do**
        1. Calculate residuals $U_i = Y_i - F_{k,(m-1)}(X_i) \; \forall \; X_i, Y_i \in d_k$
        2. Fit base learners $f_m(\cdot)$ to working residuals U_i in parallel and pick the one that minimizes the loss function
        3. Update model $F_{k,m}(\cdot) = F_{k,(m-1)}(\cdot) + \nu f_m(\cdot)$
    end
    Optimize m_{stop} using cross-validation.
end
Either

1. Create ensemble of ensembles $F = \sum_{k=1}^{K} F_k / K$ using equal weights of the components or
2. Postprocess by weighing the individual ensemble components with (penalized) regression on their predictions for the complete dataset D using an appropriate response function h:

$$F = h\left(\beta_0 + \sum_{k=1}^{K} \beta_k F_k\right)$$

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, \beta_0 + \sum_{k=1}^{K} \beta_k F_k(x_i)\right) + \lambda \cdot \sum_{k=1}^{K} \operatorname{pen}(\beta_k)$$

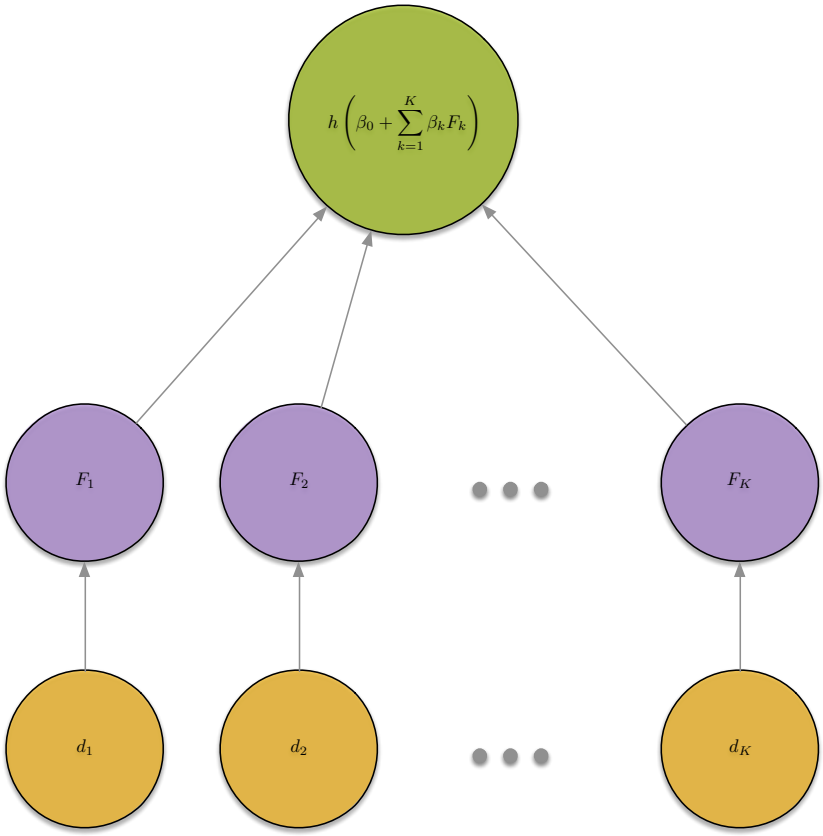


FIGURE 5: ParBoost network diagram with postprocessing.

on each bootstrap sample is an alternative. The resulting boosting models are combined in an ensemble of ensembles, either equally weighted or postprocessed using (regularized) regression (algorithm 5). For postprocessing, each of the k models generate predictions for the *entire* dataset D. Thus $N - N/K$ data points are "out-of-bag" for every model. These predictions are then used as features in the postprocessing step:

$$F = \beta_0 + \sum_{k=1}^{K} \beta_k F_k \tag{12}$$

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, \beta_0 + \sum_{k=1}^{K} \beta_k F_k(x_i) \right) + \lambda \cdot \sum_{k=1}^{K} \operatorname{pen}(\beta_k). \tag{13}$$

I have included GLMs, the Lasso, Ridge regression and the Elastic Net as postprocessing procedures in `parboost`. The higher the proportion of out-of-bag data points for each model in eq. (13), the less effective regularized regression theoretically becomes. This happens because more out-of-bag data points means using more data independent of estimating the models F_k is used for estimating the model weights $\boldsymbol{\beta}$, hence $\lambda = 0$ would produce less biased estimates (see Copas (1983)). Therefore, if the proportion of out-of-bag samples is high, the analyst should choose the unregularized GLM

procedure over the regularized variants.

ParBoost can easily be adapted to run in the MapReduce framework (Dean and Ghemawat 2008) on platforms such as Hadoop[8] (algorithm 6). The data would be automatically partitioned among the cluster nodes by HDFS. Each mapper fits an independent boosting model—possibly using parallelism from `isleboost`. These models are then postprocessed and aggregated in the reduce step.

Algorithm 6: ParBoost in the MapReduce framework

Partition data into K subsamples or generate K bootstrap samples, $\{d_k \subseteq D\}_{k \in K}$ and send each sample d_k to a different cluster node.

Map Fit a gradient boosting model on each mapper using the local subset $d_k \in D$ of the data:
for m *in* $1 : m_{max}$ **do**

1. Calculate residuals $U_i = Y_i - F_{k,(m-1)}(X_i) \; \forall \; X_i, Y_i \in d_k$
2. Fit base learners $f_m(\cdot)$ to residuals U_i in parallel and pick the one that minimizes the loss function
3. Update model $F_{k,m}(\cdot) = F_{k,(m-1)}(\cdot) + \nu f_m(\cdot)$

end

Optimize m_{stop} using cross-validation.

Reduce Postprocess the models:

1. Create ensemble of ensembles $F = \sum_{k=1}^{K} F_k / K$ or
2. Postprocess by weighing the individual ensemble components with (penalized) regression on their predictions for the complete dataset D:

$$F = \beta_0 + \sum_{k=1}^{K} \beta_k F_k$$

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, \beta_0 + \sum_{k=1}^{K} \beta_k F_k(x_i) \right) + \lambda \cdot \sum_{k=1}^{K} \operatorname{pen}(\beta_k)$$

3. Output final model F

Ignoring the postprocessing step and communication overhead—again for simplicity—ParBoost has a computational complexity of (see eq. (10))

$$O\left(\frac{pn}{K} \ln \frac{n}{K} + \frac{Mpn}{K} \right). \tag{14}$$

Figure 6 shows the computational complexity of ParBoost when varying the number of cluster nodes K and keeping the number of observations fixed at $n = 100000$ using $p = 100$ piecewise

[8] `http://hadoop.apache.org`

linear base learners and $M = 1000$ iterations. On small datasets, the overhead of setting up the cluster and communication between the nodes will outweigh any benefits in speed. On large datasets though, there is a clear benefit in scalability to using ParBoost over ordinary gradient boosting.

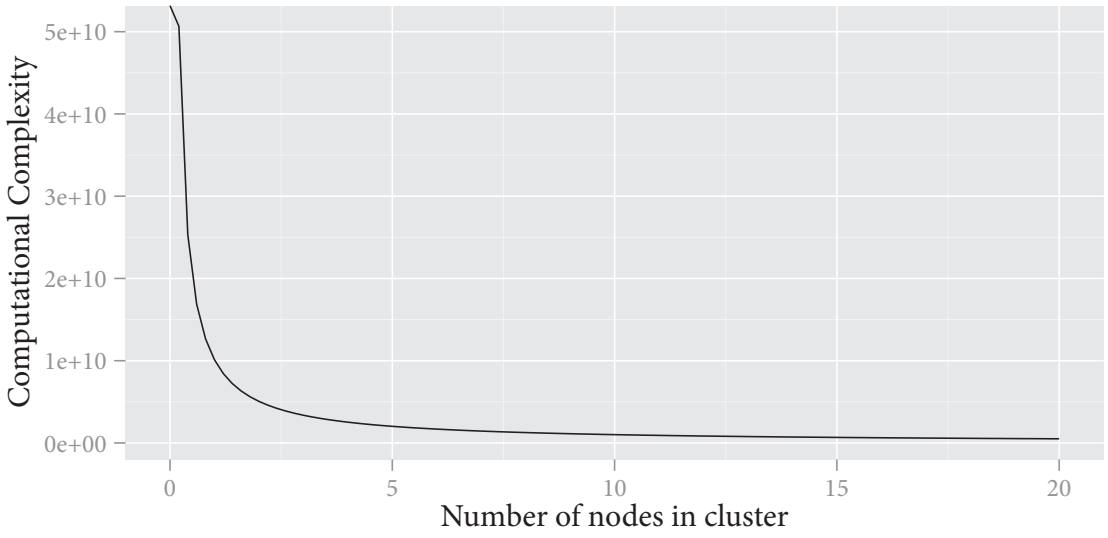


FIGURE 6: Computational complexity of ParBoost with fixed number of observations $n = 100000$, 100 piecewise linear base learners and 1000 iterations M while varying the number of cluster nodes K.

3. Simulation experiments

The purpose of ParBoost and ISLEBoost is to make Boosting scalable without sacrificing predictive performance. Furthermore, ParBoost should have a similar ability to select relevant base learners and discard irrelevant ones as ordinary Boosting. To test the effectiveness of ParBoost and ISLEBoost towards reaching these goals, I designed three simulation studies, which I will cover in the following three sections.

3.1. Speedup

To examine the scalability of ISLEBoost and ParBoost, I measure the *speedup* of each algorithm. "[Speedup] is defined as the ratio of the time taken to solve a problem on a single processing

element to the time required to solve the same problem on a parallel computer with p identical processing elements" (Grama et al. 2003). Denote speedup by

$$S = \frac{T_1}{T_p}, \tag{15}$$

with T_1 for the time taken on a single processing element and T_p for the time taken on p processing elements. In the shared memory framework of ISLEBoost, a processing element is a CPU Core, whereas in the distributed memory framework of ParBoost, a processing element is a cluster node.

3.1.1. ISLEBoost

For ISLEBoost, I simulated a dataset with 1,000,000 observations consisting of a normally distributed response and 80 normally distributed features. I measured the time to fit an ISLEBoost model on the simulated data using 2 and 4 CPU cores, relative to the time it took on a single CPU core[9]. No subsampling or postprocessing was used, so the speedup measured here is simply the result of fitting the base learners in parallel. Section 3.1.1 shows the speedup for ISLEBoost. The

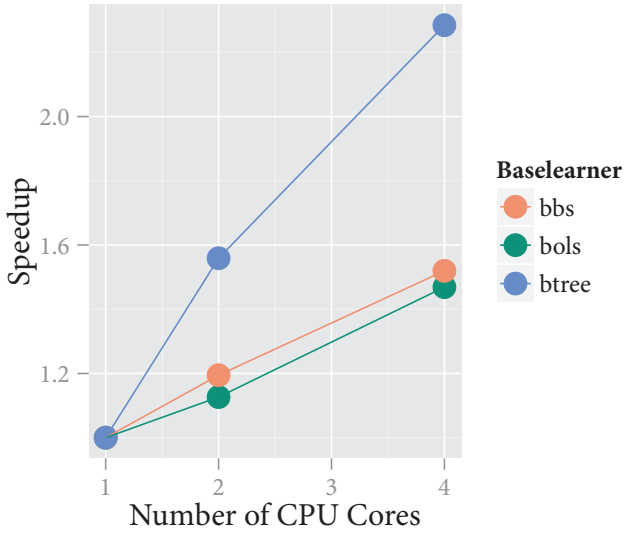


FIGURE 7: Speedup for ISLEBoost on simulated data with 1,000,000 observations and 80 features. The base learners are `bbs`=splines, `bols`=piecewise linear and `btree`=regression stumps.

speedup is fairly linear as expected (see eq. (11)), although going from one to two cores does not halve the computation time—clearly the overhead of the parallel computation is taking its toll.

[9]The simulation was run on an Amazon EC2 `cr1.8xlarge` instance (as of September 2013).

3.1.2. ParBoost

Going by the computational complexity of gradient boosting (eq. (10)) and that of ParBoost (eq. (14)), the theoretical speedup of ParBoost should be (Palit 2012):

$$\text{Speedup} = O\left(\frac{pn\ln n + Mpn}{\frac{pn}{K}\ln\frac{n}{K} + \frac{Mpn}{K}}\right) = O\left(K\frac{\ln n + M}{\ln\frac{n}{K} + M}\right). \tag{16}$$

Setting the number of nodes greater than one ($K > 1$), the fraction

$$\frac{\ln n + M}{\ln\frac{n}{K} + M}$$

in eq. (16) will be greater than 1. Thus the speedup for ParBoost should be greater than the number of nodes K.

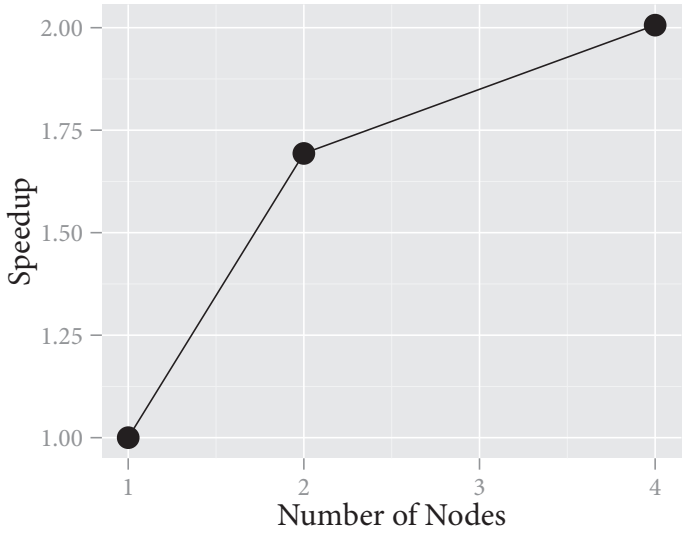


FIGURE 8: Speedup for ParBoost on simulated data with 500,000 observations and 80 features using piecewise linear base learners.

Figure 8 shows the simulation results for the speedup of ParBoost. The speedup is linear but not greater than 100 % for each node, as the computational complexity (eq. (16)) suggests—this is due to the overhead of setting up the cluster[10], reading the data on each node, and transmitting back the results. But as I demonstrate in section 4.2, ParBoost scales to 50 cluster nodes and beyond, making boosting feasible for very large datasets.

[10]The simulation was run on an Amazon EC2 `cr1.8xlarge` instance (as of September 2013).

3.2. Predictive performance

I ran the following simulation to compare the predictive performance of ParBoost (algorithm 5) and ISLEBoost (algorithm 4) to random forests (Breiman 2001), L_2Boosting (Bühlmann and B. Yu 2003) for regression and BinomialBoosting (Bühlmann and Hothorn 2007) for classification. I compare the MSE for regression and the misclassification rate for classification. To this end, I generate a random target function in each of the 10 simulation runs with $N = 20,000$ observations and $P = 100$ predictors. Of those 100 predictors, 25 are relevant and 75 are noise[11] variables. The predictors $\boldsymbol{X} \sim N(0, \boldsymbol{I}_{100})$ are i.i.d. normal with unit variance truncated at $[-2, 2]$. The target is generated as a function of the predictors

$$Y = F(\boldsymbol{\beta}, \boldsymbol{X}) + \epsilon \tag{17}$$

with

$$F(\boldsymbol{\beta}, \boldsymbol{X}) = \sum_{j=1}^{5} \beta_j X_j + \sum_{j=6}^{10} \beta_j X_j^2 + \sum_{j=11}^{15} \beta_j X_j^3 + \sum_{j=16}^{20} \beta_j \sin\left(X_j\right) + \sum_{j=21}^{25} \beta_j \frac{1}{1 + \exp\left(-X_j\right)}. \tag{18}$$

Each influence function $f_j \in F$ is centered and scaled to mean 0 and standard deviation 1. The coefficient vector $\boldsymbol{\beta}$ is sampled from the uniform distribution $U[-1, 1]^{25}$. Denote $F(\boldsymbol{\beta}, \boldsymbol{X})$ by η, then the standard deviation is

$$\text{sd}_\eta = \sqrt{\sum_{i=1}^{n} (\eta_i - \bar{\eta})^2 / n}, \tag{19}$$

and the signal-to-noise ratio is

$$\text{SNR} = \frac{n \cdot \text{sd}_\eta^2}{\sum_{i=1}^{n} \epsilon_i^2}. \tag{20}$$

Given a signal-to-noise ratio of 2 and η, I generated the target using

$$y_i \sim N\left(\eta_i, \frac{sd_\eta^2}{\text{SNR}}\right). \tag{21}$$

For the classification task, I split the target on the median of y to $\{0, 1\}$. I used half of the data for training and half of the data as a test set (i.e. 10,000 observations each) for every simulation run. To tune m_{stop}, I used 10-fold cross-validation on the training set for each of the 10 simulation runs.

The algorithms in this simulation are:

1. Random Forests from the `randomForest` (Liaw and Wiener 2002) R-package with 500 trees.

[11] They are completely unrelated to the response.

2. L_2Boosting for regression and BinomialBoosting for classification, both from the `mboost` (Hothorn et al. 2012) R-package with a maximum of 2000 iterations and a stepsize of $\nu = 0.1$. With:

   i) Spline base learners (cubic B-spline with 2$^{\text{nd}}$ order difference penalty, 20 knots and 1 degree of freedom) using the p-spline decomposition[12] (see Kneib, Hothorn, and Tutz (2009), Hofner et al. (2011)).

   ii) Tree stumps.

3. ISLEBoost from the `isleboost` R-package (appendix B.1) with identical settings to 2), but additionally using:

   i) Postprocessing with the lasso.

   ii) Feature subsampling ($\eta = 0.5$, without replacement) with and without lasso postprocessing.

   iii) Observation subsampling ($\eta = 0.1$, without replacement) with and without lasso postprocessing.

4. ParBoost from the `parboost` R-package (appendix B.2), splitting the data into 4 disjoint subsets, with a maximum of 2000 iterations and a stepsize of $\nu = 0.1$. The base learners and loss functions for each boosting model are identical to 2). Each ParBoost model was postprocessed (see section 2.2 for details) by:

   i) Simply taking the mean of the ensemble components.

   ii) Using a GLM

   iii) Fitting a Lasso.

Looking at the MSE ratios to the baseline L_2Boosting model for the boosting simulation with the real-valued response for the models using p-spline decomposition base learners in fig. 9, the MSE of the boosting models are all within 5 % of each other. ParBoost using GLM postprocessing does particularly well, with virtually identical MSEs to the baseline model. Keeping in mind that ParBoost is much more scalable than ordinary L_2Boosting, this result is very promising. ParBoost with no postprocessing at all (weighing each ensemble component equally) does slightly worse, whereas ParBoost with lasso postprocessing does a lot worse. Using lasso postprocessing on L_2Boosting (fig. 9, `isleboost.bbs.lasso`) slightly worsens predictive performance. ISLEBoost with feature or observation subsampling also does worse than the baseline L_2Boosting model and increases variance.

[12] Allows the algorithm to choose linear and smooth effects separately.

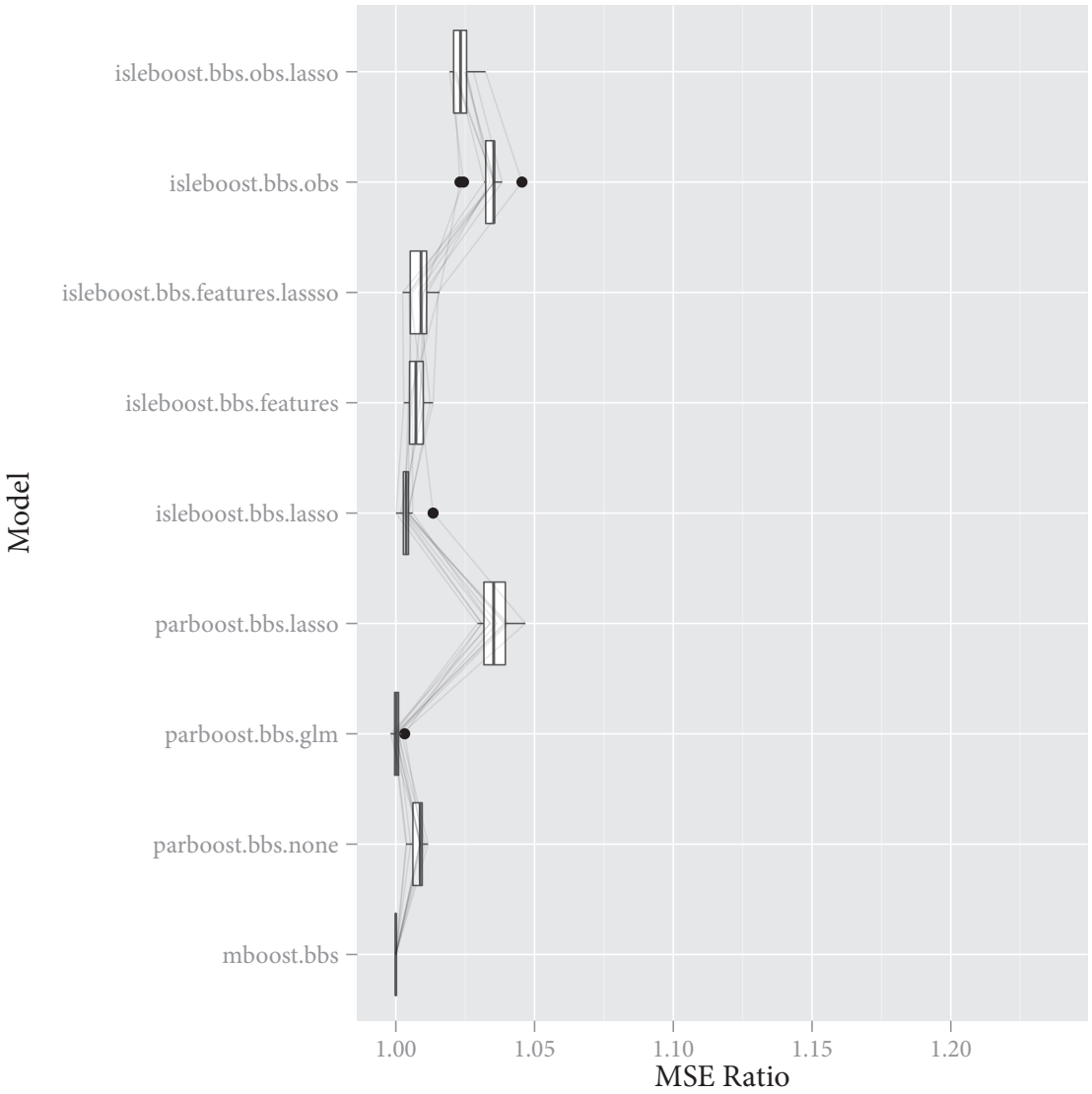


FIGURE 9: Predictive simulation (regression) with p-spline decomposition base learners: ratios of mean square error over 10 simulation runs to the baseline model for the real-valued response. Models from top to bottom: ISLEBoost using subsampling of the observations and lasso postprocessing; ISLEBoost using subsampling of the observations; ISLEBoost using feature subsampling and lasso postprocessing; ISLEBoost using feature subsampling; ISLEBoost using Lasso postprocessing; ParBoost using lasso postprocessing; ParBoost using GLM postprocessing; ParBoost without postprocessing; L_2Boosting (baseline). The light grey lines connect simulation runs.

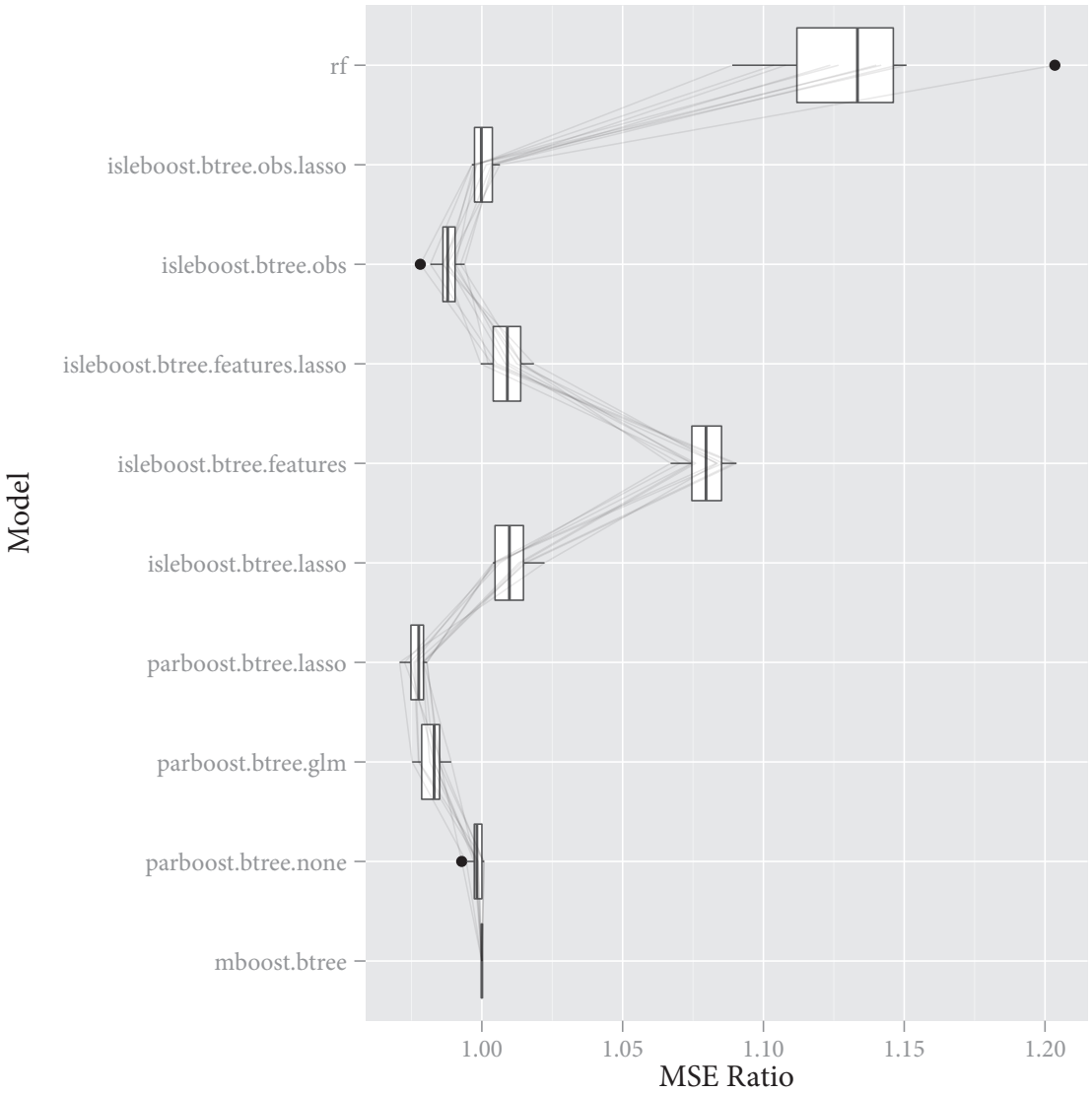


FIGURE 10: Predictive simulation (regression) with tree stump base learners: ratios of mean square error over 10 simulation runs to the baseline model for the real-valued response. Models from top to bottom: Random forest; ISLEBoost using subsampling of the observations and lasso postprocessing; ISLEBoost using subsampling of the observations; ISLEBoost using feature subsampling and lasso postprocessing; ISLEBoost using feature subsampling; ISLEBoost using lasso postprocessing; ParBoost using lasso postprocessing; ParBoost using GLM postprocessing; ParBoost without postprocessing; L_2Boosting (baseline). The light grey lines connect simulation runs.

The MSE ratios to the baseline L_2Boosting model for the simulation with the real-valued response for the boosting models using tree stump base learners (and the random forest) in fig. 10 show a different picture than the p-spline decomposition base learners in fig. 9. First of all, there are several models that do better than the baseline L_2Boosting model. All three ParBoost models show a superior performance to the baseline, and ParBoost using lasso postprocessing is actually the strongest model of all. Again, considering the superior scalability of ParBoost to ordinary gradient boosting, this is a very promising result for the algorithm. ISLEBoost using feature subsampling does worse than the baseline L_2Boosting model, but postprocessing improves performance—the opposite of the results using p-spline decomposition base learners.

Besides the strong results of ParBoost, the main difference to the p-spline decomposition base learners in fig. 9 is the strong showing of the ISLEBoost models using subsampling of the observations (especially using postprocessing). The gain in predictive performance using lasso postprocessing carries through to ParBoost, which does slightly better than the GLM postprocessing—as opposed to p-spline decomposition base learners, where GLM postprocessing was superior to lasso postprocessing. Figure 11 shows the MSEs for all the models in the predictive simulation with the real-valued response. The models using the p-spline decomposition base learners perform better than tree stump base learners for this particular target.

Figure 12 shows the ratios of the misclassification rate to the baseline BinomialBoosting model for the binary response using p-spline decomposition base learners. ISLEBoost using subsampling of the observations does slightly worse than the baseline, similar to the real-valued response (fig. 9). Whereas ParBoost using GLM postprocessing had virtually identical performance to the baseline for the real-valued response, it does worse for the classification problem. Figure 12 displays the ratios of the misclassification rate for the boosting models with tree base learners and the random forest to the baseline BinomialBoosting model. Random forest has a much higher misclassification rate than the boosting models. Subsampling observations also does badly—opposed to the real-valued response, where subsampling observations improved predictive performance (fig. 10). The ParBoost models using postprocessing perform better than the baseline (fig. 13), but ParBoost using no postprocessing does worse. This is in line with the findings for the real-valued response, where postprocessing generally seems to help models with tree base learners, but worsens performance for piecewise linear and spline base learners. Table 2 and table 3 show a summary of this simulation.

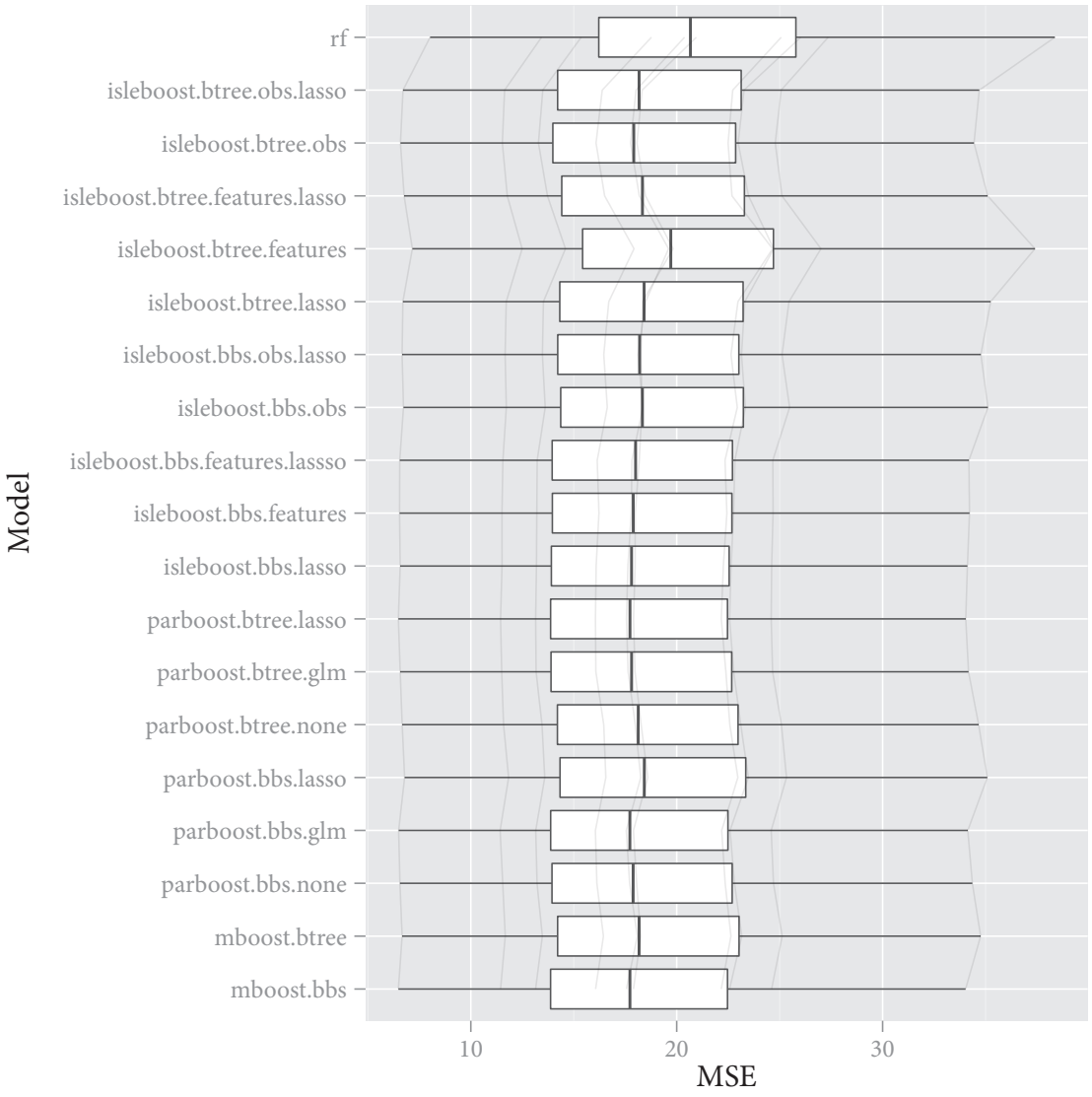


FIGURE 11: Predictive simulation (regression) mean square errors over 10 simulation runs. The light grey lines connect simulation runs.

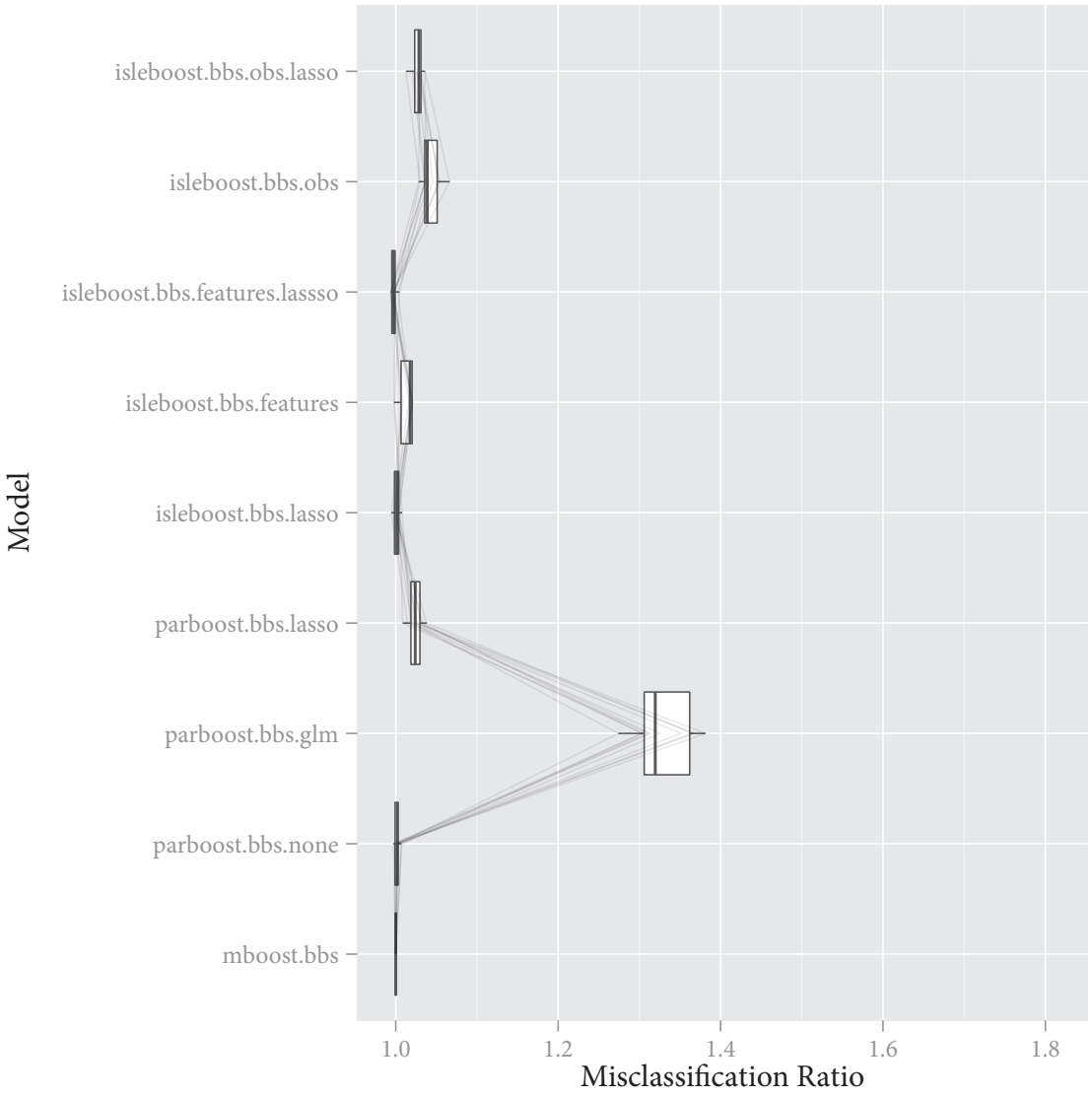


FIGURE 12: Predictive simulation (classification) with p-spline decomposition base learners: average ratio of misclassification rate over 10 simulation runs to the baseline model for the real-valued response. Models from top to bottom: ISLEBoost using subsampling of the observations and lasso postprocessing; ISLEBoost using subsampling of the observations; ISLEBoost using feature subsampling and lasso postprocessing; ISLEBoost using feature subsampling; ISLEBoost using lasso postprocessing; ParBoost using lasso postprocessing; ParBoost using GLM postprocessing; ParBoost without postprocessing; BinomialBoosting (baseline). The light grey lines connect simulation runs.

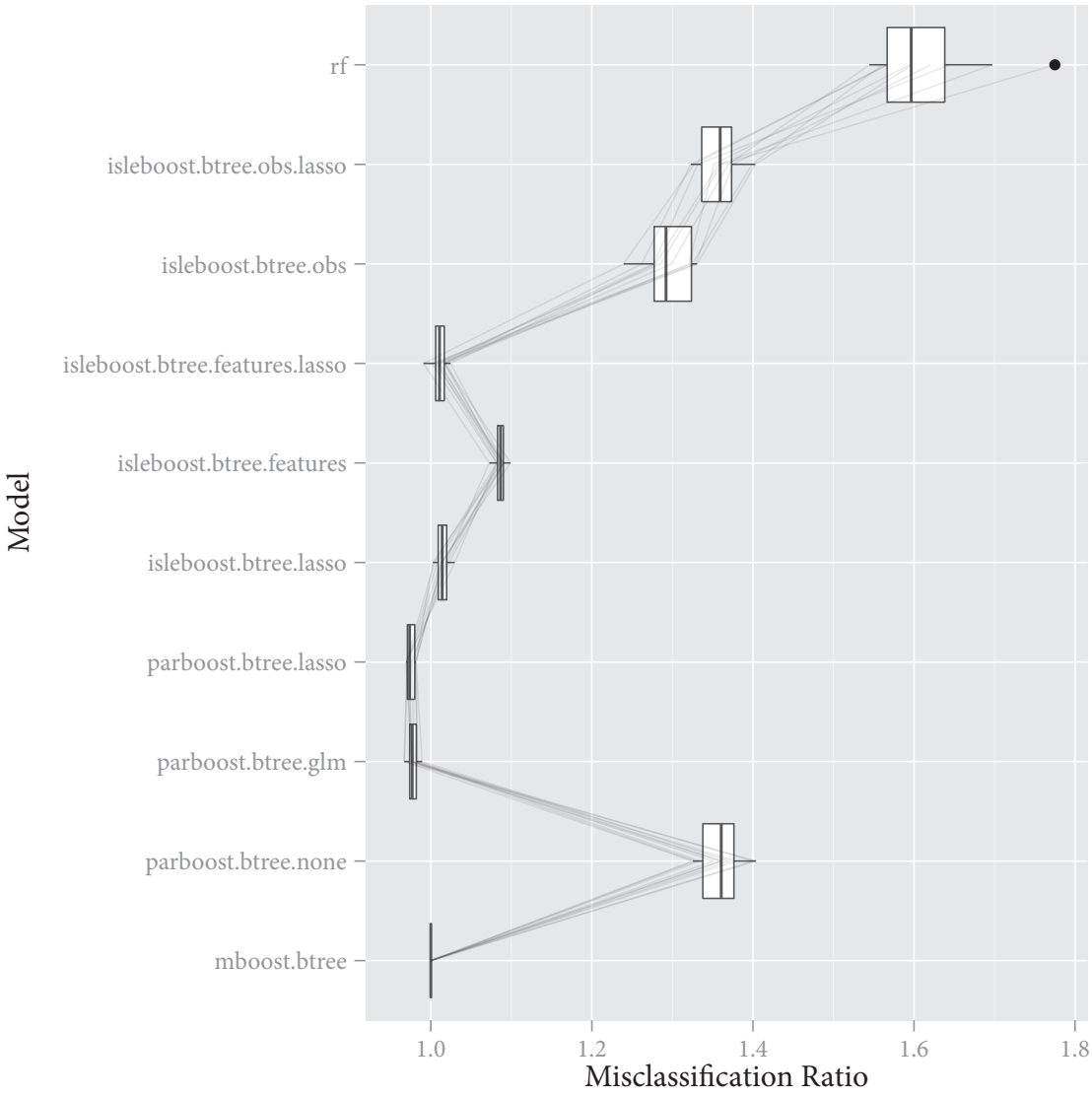


FIGURE 13: Predictive simulation (classification) with tree stump base learners: average ratio of misclassification rate over 10 simulation runs to the baseline model for the real-valued response. Models from top to bottom: ISLEBoost using subsampling of the observations and lasso postprocessing; ISLEBoost using subsampling of the observations; ISLEBoost using feature subsampling and lasso postprocessing; ISLEBoost using feature subsampling; ISLEBoost using lasso postprocessing; ParBoost using lasso postprocessing; ParBoost using GLM postprocessing; ParBoost without postprocessing; BinomialBoosting (baseline). The light grey lines connect simulation runs.

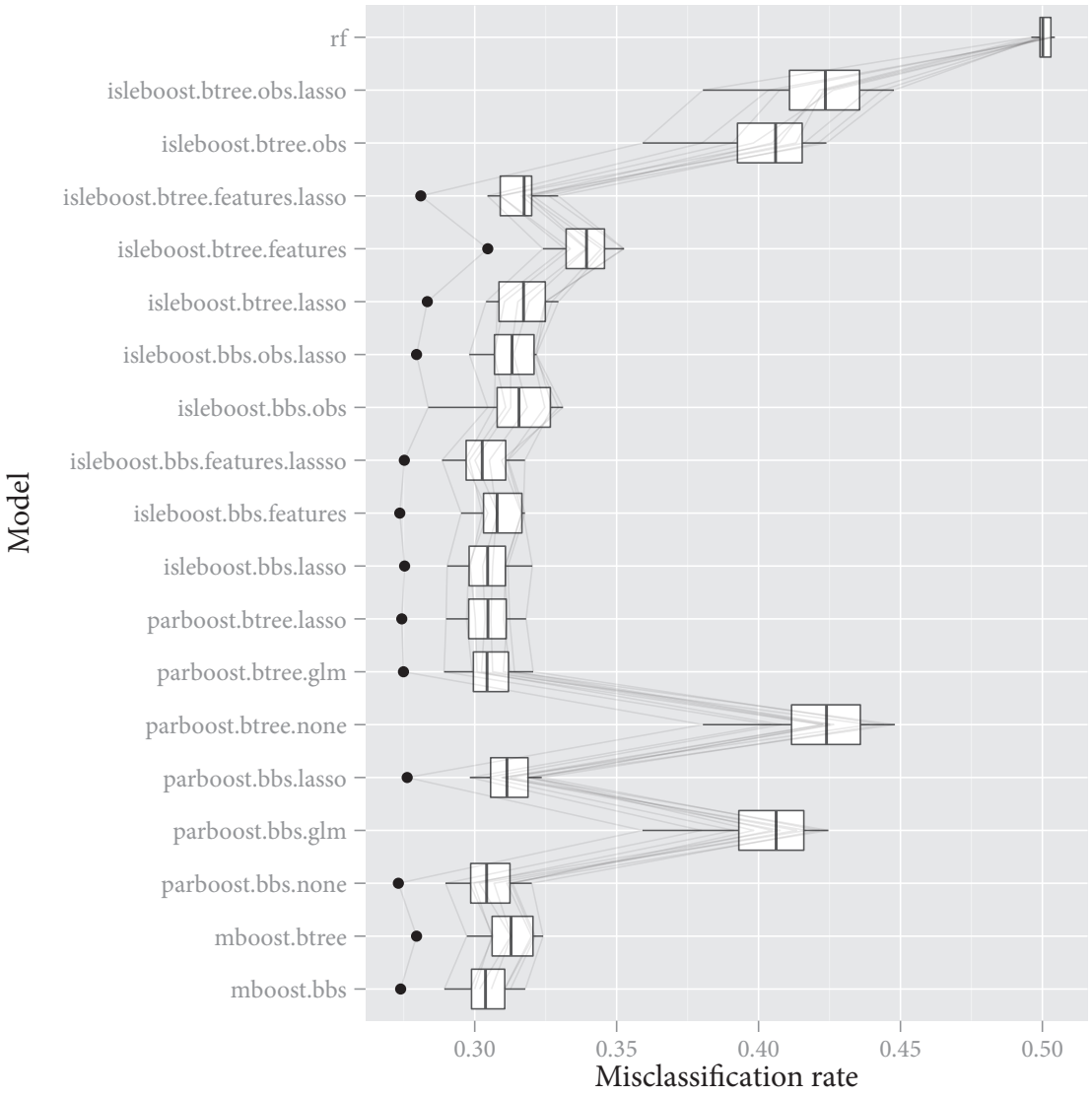


FIGURE 14: Predictive simulation (classification) misclassification rates over 10 simulation runs. The light grey lines connect simulation runs.

Table 2: MSEs for the predictive simulation (regression) over 10 simulation runs

Model	Base Learners	Mean	St. Dev.	Min	Max
L_2Boosting	P-Spline decomposition	18.596	7.735	6.482	34.031
L_2Boosting	Tree Stumpms	19.027	7.893	6.664	34.761
ParBoost	P-Spline decomposition	18.743	7.804	6.557	34.359
ParBoost with GLM postprocessing	P-Spline decomposition	18.608	7.760	6.495	34.139
Parboost with Lasso postprocessing	P-Spline decomposition	19.249	7.962	6.784	35.084
ParBoost	Tree Stumpms	18.987	7.873	6.667	34.664
ParBoost with GLM postprocessing	Tree Stumpms	18.691	7.769	6.569	34.173
ParBoost with Lasso postprocessing	Tree Stumpms	18.595	7.736	6.481	34.040
ISLEBoost with Lasso postprocessing	P-Spline decomposition	18.660	7.735	6.569	34.118
ISLEBoost subsampling features	P-Spline decomposition	18.736	7.777	6.538	34.222
ISLEBoost subsampling features with Lasso postprocessing	P-Spline decomposition	18.751	7.762	6.547	34.193
ISLEBoost subsampling observations	P-Spline decomposition	19.224	7.990	6.730	35.115
ISLEBoost subsampling observations with Lasso postprocessing	P-Spline decomposition	19.033	7.894	6.664	34.773
ISLEBoost with Lasso postprocessing	Tree Stumpms	19.246	8.025	6.700	35.247
ISLEBoost subsampling features	Tree Stumpms	20.537	8.483	7.160	37.400
ISLEBoost subsampling features with Lasso postprocessing	Tree Stumpms	19.181	7.938	6.760	35.101
ISLEBoost subsampling observations	Tree Stumpms	18.802	7.839	6.583	34.440
ISLEBoost subsampling observations with Lasso postprocessing	Tree Stumpms	19.033	7.882	6.707	34.694
Random Forest	Trees	21.375	8.468	8.020	38.366

TABLE 3: Misclassification rates for the predictive simulation (classification) over 10 simulation runs

Model	Base Learners	Mean	St. Dev.	Min	Max
BinomialBoosting	P-Spline decomposition	0.302	0.013	0.274	0.318
BinomialBoosting	Tree Stumpms	0.310	0.014	0.279	0.324
ParBoost	P-Spline decomposition	0.303	0.014	0.273	0.320
ParBoost with GLM postprocessing	P-Spline decomposition	0.402	0.020	0.359	0.425
Parboost with Lasso postprocessing	P-Spline decomposition	0.309	0.014	0.276	0.324
ParBoost	Tree Stumpms	0.422	0.020	0.380	0.448
ParBoost with GLM postprocessing	Tree Stumpms	0.303	0.013	0.275	0.321
ParBoost with Lasso postprocessing	Tree Stumpms	0.302	0.013	0.274	0.318
ISLEBoost with Lasso postprocessing	P-Spline decomposition	0.302	0.013	0.275	0.320
ISLEBoost subsampling features	P-Spline decomposition	0.306	0.014	0.274	0.318
ISLEBoost subsampling features with Lasso postprocessing	P-Spline decomposition	0.301	0.013	0.275	0.318
ISLEBoost subsampling observations	P-Spline decomposition	0.315	0.015	0.284	0.331
ISLEBoost subsampling observations with Lasso postprocessing	P-Spline decomposition	0.310	0.013	0.279	0.322
ISLEBoost with Lasso postprocessing	Tree Stumpms	0.315	0.014	0.283	0.330
ISLEBoost subsampling features	Tree Stumpms	0.337	0.014	0.305	0.353
ISLEBoost subsampling features with Lasso postprocessing	Tree Stumpms	0.313	0.014	0.281	0.329
ISLEBoost subsampling observations	Tree Stumpms	0.401	0.020	0.359	0.424
ISLEBoost subsampling observations with Lasso postprocessing	Tree Stumpms	0.422	0.020	0.380	0.448
Random Forest	Trees	0.500	0.003	0.496	0.504

3.3. Variable selection

This simulation experiment examines the ability of ParBoost and ISLEBoost to select relevant features and discard irrelevant ones in comparison to "standard" boosting. For each of the 25 simulation runs, I generated 40000 observations with 100 features, of which 25 are relevant and 75 are noise[13]. The variables are drawn from multivariate Gaussian truncated at $[-2, 2]$. There are 5 blocks of 5 variables each (to give a total of 25 relevant variables), with each block having a different compound symmetry covariance structure[14]: $\{0, 0.2, 0.4, 0.8, 0.99\}$. All variables have mean 0 and variance 1. The coefficient vector $\boldsymbol{\beta} \sim U[0, 1]^{25}$ is sampled from the uniform distribution. The target was then generated by eq. (21) with a signal-to-noise ratio of 2 and $\eta = \boldsymbol{X\beta}$. For the classification task, I split the target on the median of y to $\{0, 1\}$.

The algorithms are evaluated on the false positive and false negative rates of selected variables. To evaluate the predictive impact of variable selection, I also evaluate the ratios of the MSE for regression and the misclassification rate for classification, using standard boosting as the baseline model. As in section 3.2, I split the data in a training and a test set, each consisting of 20000 observations. 10-fold cross-validation was used to tune the algorithms on the training set for each of the 25 simulation runs.

The algorithms compared in this simulation are:

1. L_2Boosting for regression and BinomialBoosting for classification, both from the `mboost` R-package with a maximum of 2000 iterations and a stepsize of $\nu = 0.1$ using linear base learners.

2. ISLEBoost from the `isleboost` R-package (appendix B.1) with identical parameters to 1) using:

   i) Lasso postprocessing

   ii) Feature subsampling with $\eta = 0.5$ (no replacement)

   iii) Feature subsampling with lasso postprocessing and $\eta = 0.5$ (no replacement)

   iv) Subsampling of the observations with $\eta = 0.1$ (no replacement)

   v) Subsampling of the observations with lasso postprocessing and $\eta = 0.1$ (no replacement)

3. ParBoost from the `parboost` R-package (appendix B.2), splitting the data into 4 disjoint subsets. The rest of the boosting parameters are identical to 1). Each ParBoost model was postprocessed (see section 2.2 for details) by:

[13] They are completely unrelated to the response.
[14] Identical correlation between all predictors within a group.

i) Simply taking the mean of the ensemble components.

ii) Using a GLM.

iii) Fitting a Lasso.

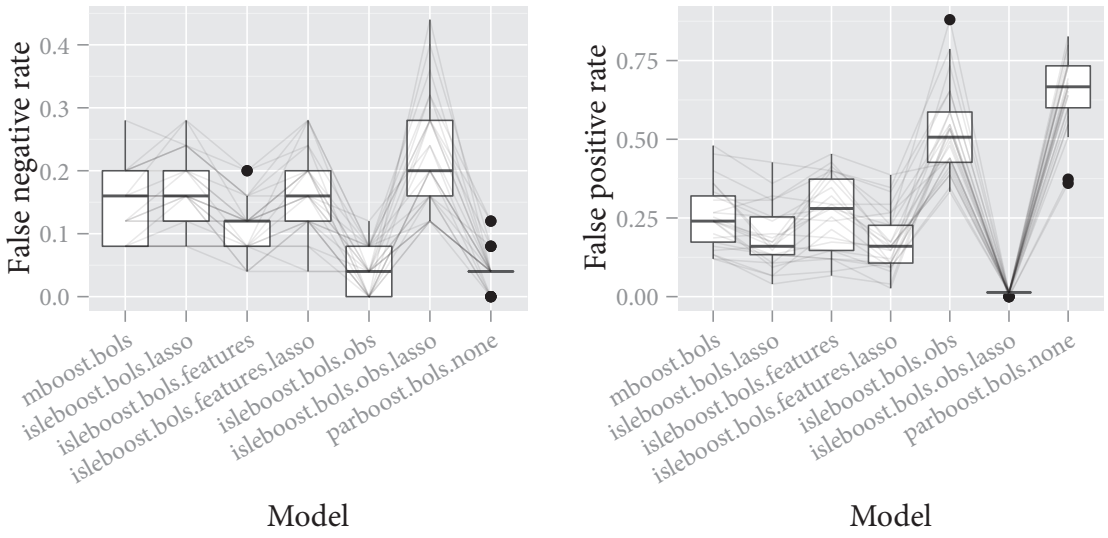


FIGURE 15: Variable selection simulation (regression): false negative and false positive rates for selecting the relevant coefficients over 25 simulation runs. The l^0 norm of the true model is 25. Models from left to right: L_2Boosting, ISLEBoost with lasso postprocessing; ISLEBoost using feature subsampling; ISLEBoost using feature subsampling and lasso postprocessing; ISLEBoost subsampling observations; ISLEBoost subsampling observations with lasso postprocessing; ParBoost without postprocessing. The light grey lines connect simulation runs.

Figure 15, table 4 and table 5 show the false negative and false positive rates for the real-valued response over the 25 simulation runs. As expected, the three models postprocessed by the lasso show the lowest average false positive rates by eliminating weak base learners in the postprocessing step. ISLEBoost with postprocessing but without subsampling, and ISLEBoost with feature subsampling and postprocessing, only show a slightly higher false negative rate than the baseline model. This makes them interesting alternatives to other sparse boosting techniques, such as SparseL_2Boost (Bühlmann and B. Yu 2006) or Twin Boosting (Bühlmann and Hothorn 2010).

ParBoost and ISLEBoost using subsampling of the observations without postprocessing have low false negative rates, but at the cost of high false positives. This suggests that these two methods have a harder time identifying relevant coefficients because of the subsampling of the observations. These two subsampling schemes of the observations are very different, but this simulation shows

that the effect on variable selection is similar. Subsampling features without postprocessing displays a somewhat higher false positive rate than L_2Boosting —but only slightly—and has a lower false negative rate. This is to be expected, since some iterations are bound to sample mostly irrelevant features in this rather sparse setting. The impact on variable selection here is small nevertheless, especially compared to the linear reduction in computing time due to the subsampling.

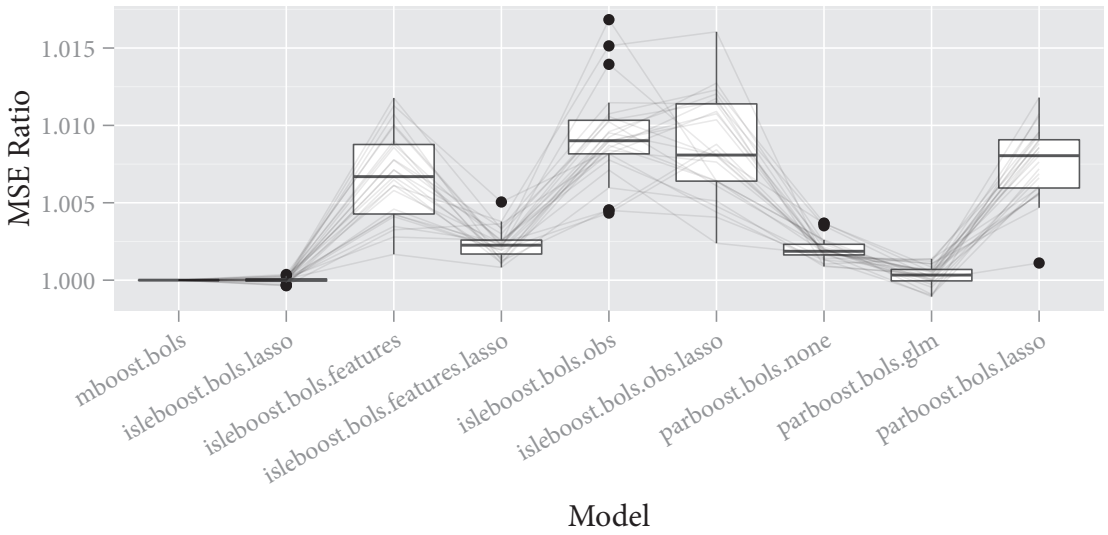


FIGURE 16: Variable selection simulation (regression): ratio of the mean square error using L_2Boosting as the baseline. 25 simulation runs. Models from left to right (all with piecewise linear base learners): L_2Boosting (baseline), ISLEBoost with lasso postprocessing; ISLEBoost using feature subsampling; ISLEBoost using feature subsampling and lasso postprocessing; ISLEBoost subsampling observations; ISLEBoost subsampling observations with lasso postprocessing; ParBoost without postprocessing; ParBoost using GLM Postprocessing; ParBoost using lasso postprocessing. The light grey lines connect simulation runs.

The MSE ratios to the baseline L_2Boosting model are shown in fig. 16. Note the scale of the y-axis: it only goes up to 1.015—so all models here show little difference in predictive performance to the baseline. That being said, ISLEBoost using lasso postprocessing has a nearly identical MSE to the baseline, whereas the ISLEBoost models using subsampling—with and without postprocessing—have higher MSEs. This supports the finding that postprocessing in ISLEBoost seems to harm piecewise linear base learners (see section 3.2). ParBoost using GLM postprocessing performs very well. The algorithms predictive performance displays very little variance and near identical performance to the baseline—yet it is much more scalable.

Figure 17, table 7 and table 8 show the false negative and false positive rates for the binary response over the 25 simulation runs. The rates of the models are nearly identical to the real valued response,

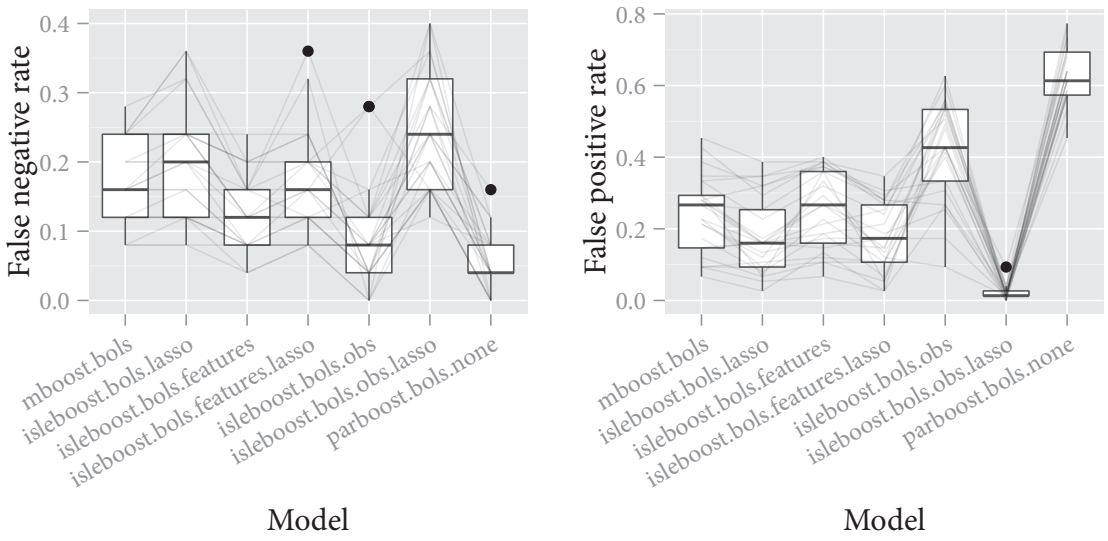


FIGURE 17: Variable selection simulation (classification): average false negative and false positive rates for selecting the relevant coefficients over 25 simulation runs. The l^0 norm of the true model is 25. Models from left to right: BinomialBoosting, ISLEBoost with lasso postprocessing; ISLEBoost using feature subsampling; ISLEBoost using feature subsampling and lasso postprocessing; ISLEBoost subsampling observations; ISLEBoost subsampling observations with lasso postprocessing; ParBoost without postprocessing. The light grey lines connect simulation runs.

so please refer to the analysis above. Looking at the ratio of the misclassification rates to the baseline BinomialBoosting model in fig. 18, it is obvious that ISLEBoost using subsampling of the observations and ParBoost using GLM postprocessing have a higher misclassification rate than the rest of the models—which have virtually identical misclassification rates. The result for ParBoost is similar to that for the predictive simulation for the binary response and p-spline decomposition base learners (see fig. 12). Table 6 and table 9 summarize the predictive performance of the various models in this simulation.

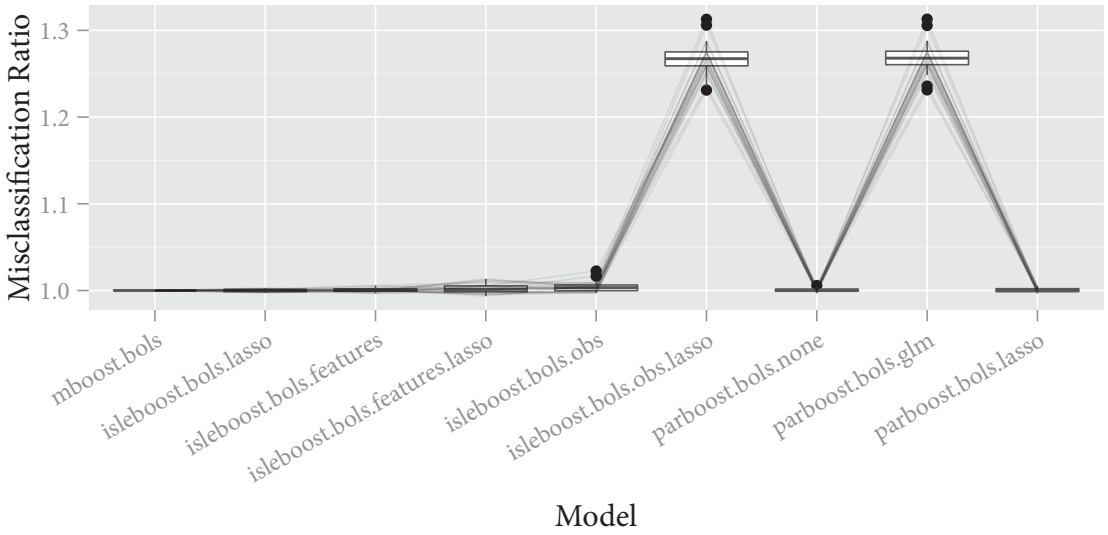


FIGURE 18: Variable selection simulation (classification): ratio of the misclassification rate using BinomialBoosting as the baseline. 25 simulation runs. Models from left to right (all with piecewise linear base learners): BinomialBoosting (baseline), ISLEBoost with lasso postprocessing; ISLEBoost using feature subsampling; ISLEBoost using feature subsampling and lasso postprocessing; ISLEBoost subsampling observations; ISLEBoost subsampling observations with lasso postprocessing; ParBoost without postprocessing; ParBoost using GLM Postprocessing; ParBoost using lasso postprocessing. The light grey lines connect simulation runs.

TABLE 4: False negative rates (regression) over 25 simulation runs

Model	Base Learners	Mean	St. Dev.	Min	Max
L_2Boosting	Piecewise linear	0.144	0.054	0.080	0.280
ParBoost	Piecewise linear	0.043	0.032	0.000	0.120
ParBoost with GLM postprocessing	Piecewise linear	0.043	0.032	0.000	0.120
Parboost with Lasso postprocessing	Piecewise linear	0.043	0.032	0.000	0.120
ISLEBoost with Lasso postprocessing	Piecewise linear	0.168	0.063	0.080	0.280
ISLEBoost subsampling features	Piecewise linear	0.110	0.045	0.040	0.200
ISLEBoost subsampling features with Lasso postprocessing	Piecewise linear	0.163	0.069	0.040	0.280
ISLEBoost subsampling observations	Piecewise linear	0.042	0.036	0.000	0.120
ISLEBoost subsampling observations with Lasso postprocessing	Piecewise linear	0.224	0.094	0.120	0.440

Table 5: False positive rates (regression) over 25 simulation runs

Model	Base Learners	Mean	St. Dev.	Min	Max
L_2Boosting	Piecewise linear	0.255	0.102	0.120	0.480
ParBoost	Piecewise linear	0.652	0.120	0.360	0.827
ParBoost with GLM postprocessing	Piecewise linear	0.652	0.120	0.360	0.827
Parboost with Lasso postprocessing	Piecewise linear	0.652	0.120	0.360	0.827
ISLEBoost with Lasso postprocessing	Piecewise linear	0.188	0.098	0.040	0.427
ISLEBoost subsampling features	Piecewise linear	0.262	0.120	0.067	0.453
ISLEBoost subsampling features with Lasso postprocessing	Piecewise linear	0.174	0.096	0.027	0.387
ISLEBoost subsampling observations	Piecewise linear	0.523	0.137	0.333	0.880
ISLEBoost subsampling observations with Lasso postprocessing	Piecewise linear	0.011	0.005	0.000	0.013

Table 6: MSEs for variable selection simulation over 25 simulation runs

Model	Base Learners	Mean	St. Dev.	Min	Max
L_2Boosting	Piecewise linear	90.397	48.008	16.884	173.826
ParBoost	Piecewise linear	90.574	48.093	16.912	174.176
ParBoost with GLM postprocessing	Piecewise linear	90.422	48.021	16.905	173.747
Parboost with Lasso postprocessing	Piecewise linear	91.080	48.369	17.046	174.975
ISLEBoost with Lasso postprocessing	Piecewise linear	90.397	48.006	16.885	173.819
ISLEBoost subsampling features	Piecewise linear	91.039	48.416	17.016	175.786
ISLEBoost subsampling features with Lasso postprocessing	Piecewise linear	90.616	48.150	16.942	174.704
ISLEBoost subsampling observations	Piecewise linear	91.218	48.423	17.078	175.284
ISLEBoost subsampling observations with Lasso postprocessing	Piecewise linear	91.126	48.311	17.077	175.152

TABLE 7: False negative rates (classification) over 25 simulation runs

Model	Base Learners	Mean	St. Dev.	Min	Max
BinomialBoosting	Piecewise linear	0.178	0.061	0.080	0.280
ParBoost	Piecewise linear	0.053	0.044	0.000	0.160
ParBoost with GLM postprocessing	Piecewise linear	0.053	0.044	0.000	0.160
Parboost with Lasso postprocessing	Piecewise linear	0.053	0.044	0.000	0.160
ISLEBoost with Lasso postprocessing	Piecewise linear	0.203	0.081	0.080	0.360
ISLEBoost subsampling features	Piecewise linear	0.131	0.056	0.040	0.240
ISLEBoost subsampling features with Lasso postprocessing	Piecewise linear	0.165	0.076	0.080	0.360
ISLEBoost subsampling observations	Piecewise linear	0.088	0.073	0.000	0.280
ISLEBoost subsampling observations with Lasso postprocessing	Piecewise linear	0.246	0.082	0.120	0.400

Table 8: False positive rates (classification) over 25 simulation runs

Model	Base Learners	Mean	St. Dev.	Min	Max
BinomialBoosting	Piecewise linear	0.242	0.107	0.067	0.453
ParBoost	Piecewise linear	0.629	0.080	0.453	0.773
ParBoost with GLM postprocessing	Piecewise linear	0.629	0.080	0.453	0.773
Parboost with Lasso postprocessing	Piecewise linear	0.629	0.080	0.453	0.773
ISLEBoost with Lasso postprocessing	Piecewise linear	0.179	0.104	0.027	0.387
ISLEBoost subsampling features	Piecewise linear	0.253	0.109	0.067	0.400
ISLEBoost subsampling features with Lasso postprocessing	Piecewise linear	0.175	0.096	0.027	0.347
ISLEBoost subsampling observations	Piecewise linear	0.429	0.138	0.093	0.627
ISLEBoost subsampling observations with Lasso postprocessing	Piecewise linear	0.020	0.019	0.000	0.093

Table 9: Misclassification rates for variable selection simulation over 25 simulation runs

Model	Base Learners	Mean	St. Dev.	Min	Max
BinomialBoosting	Piecewise linear	0.357	0.020	0.306	0.384
ParBoost	Piecewise linear	0.357	0.020	0.306	0.384
ParBoost with GLM postprocessing	Piecewise linear	0.453	0.025	0.385	0.489
Parboost with Lasso postprocessing	Piecewise linear	0.357	0.021	0.306	0.383
ISLEBoost with Lasso postprocessing	Piecewise linear	0.357	0.020	0.306	0.385
ISLEBoost subsampling features	Piecewise linear	0.357	0.021	0.306	0.384
ISLEBoost subsampling features with Lasso postprocessing	Piecewise linear	0.358	0.021	0.305	0.385
ISLEBoost subsampling observations	Piecewise linear	0.359	0.021	0.305	0.391
ISLEBoost subsampling observations with Lasso postprocessing	Piecewise linear	0.453	0.025	0.385	0.489

4. Real data experiments

4.1. UCI binary classification benchmarks

I use a collection of 21 binary classification problems from the UCI Machine Learning Repository (Newman et al. 1998)—previously analyzed in Scheipl (2011) and Eugster (2011)—to examine the predictive performance of ParBoost and ISLEBoost, compared to standard gradient boosting and random forests. See table 10 for a summary of the datasets. Following Scheipl (2011), I evaluate the predictive performance of the algorithms using 20-fold cross-validation on each dataset. Specifically, I compare the AUC (area under the receiver-operator-characteristic curve, see Fawcett (2004)) and misclassification rate on each dataset. The ROC curve plots true positives vs. false positives for binary classification models as its discrimination threshold is varied. The idea behind the AUC is to reduce this plot to a single numerical value representing expected performance, where higher AUC values signify better classification rules. "The AUC has an important statistical property: the AUC of a classifier is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance" (Fawcett 2004).

The algorithms in this simulation are:

1. Random Forests from the `randomForest` (Liaw and Wiener 2002) R-package with 500 trees.

2. BinomialBoosting from the `mboost` (Hothorn et al. 2012) R-package with a maximum of 1000 iterations and a stepsize of $\nu = 0.1$. With:

   i) Spline base learners (cubic B-spline with 2^{nd} order difference penalty, 10 knots and 1 degree of freedom) using the p-spline decomposition[15] (see Kneib, Hothorn, and Tutz (2009), Hofner et al. (2011)).

   ii) Tree stumps.

   iii) Trees with 6 terminal nodes to allow for interactions.

3. ISLEBoost from the `isleboost` R-package (appendix B.1) with identical settings to 2), but additionally using:

   i) Postprocessing with the lasso.

   ii) Feature subsampling ($\eta = 0.5$, without replacement) with and without lasso postprocessing.

[15] Allows the algorithm to choose linear and smooth effects separately.

iii) Observation subsampling ($\eta = 0.5$, without replacement) with and without lasso post-processing.

4. ParBoost from the `parboost` R-package (appendix B.2), using 10 bootstrap samples of the original data, with a maximum of 1000 iterations and a stepsize of $\nu = 0.1$. The base learners and loss functions for each boosting model are identical to 2). Each ParBoost model was postprocessed (see section 2.2 for details) by:

   i) Simply taking the mean of the ensemble components.

   ii) Using a GLM

   iii) Fitting a Lasso.

All algorithms were tuned using 10-fold cross-validation. I preprocessed the data identically to Scheipl (2011), using the following scheme: "All covariates with less than 6 unique values are coded as factor variables. All numeric covariates are scaled to the unit interval $[0, 1]$ first, followed by taking the logarithm of the covariate values (plus an offset of 0.1) if skewness is greater than 2 or taking the logarithm of 1.1 minus the covariate value if skewness is below -2. All numeric covariates (transformed or not) are then standardized to have mean 0 and standard deviation 1. All incomplete observations are removed" (Scheipl 2011).

Table 10: Summary of UCI binary classification datasets (Scheipl 2011)

Dataset	Observations	All Features	Categorical Features	Class Balance	p/N
BreastCancer	683	9	9	0.526	0.01
Cards	653	15	5	0.802	0.02
Circle	1200	2	0	0.974	0.00
Heart1	296	13	5	0.836	0.04
HouseVotes84	232	16	16	0.629	0.07
Ionosphere	351	33	0	0.560	0.09
PimaIndiansDiabetes	768	8	0	0.536	0.01
Sonar	208	60	0	0.874	0.29
Spirals	1200	2	0	1.000	0.00
chess	3196	36	1	0.915	0.01
credit	1000	24	9	0.429	0.02
hepatitis	80	19	10	0.260	0.24
liver	345	6	0	0.725	0.02
monks3	554	6	4	0.924	0.01
musk	476	166	0	0.770	0.35
promotergene	106	57	57	1.000	0.54
ringnorm	1200	20	0	1.000	0.02
threenorm	1200	20	0	1.000	0.02
tictactoe	958	9	9	0.530	0.01
titanic	2201	3	1	0.477	0.00
twonorm	1200	20	0	1.000	0.02

Figures 19 to 25 show the misclassification rates and figs. 26 to 32 the AUCs for the 21 datasets. To draw a more formal analysis from this experiment than simply plotting the results with a qualitative discussion, I make use of the framework of Eugster (2011) for the analysis of domain-based benchmark experiments. Eugster (2011) models benchmark experiments using linear mixed-effects models. The model is specified as

$$p_{mbk} = \kappa_k + \beta_m + \beta_{mk} + \beta_{mb} + \epsilon_{mbk} \tag{22}$$

with $m = 1, \ldots, M$ datasets, $b = 1, \ldots, B$ replications (cross-validation folds in this case) and $k = 1, \ldots K$ algorithms. The response p_{mbk} is the performance measure for dataset m, replication b and algorithm k.

> *κ_k represents the algorithms' mean performances, b_m the mean performances of the domain's datasets, β_{mk} the interactions between datasets and algorithms, β_{mb} the effect of the subsampling within the datasets, and ϵ_{mbk} the systematic error. The candidate algorithms' effect κ_k is modeled as fixed effect, the datasets' effect β_m as random effect (as the datasets can be seen as randomly drawn from the domain they define). Furthermore, β_{mk}, β_{mb} and ϵ_{mbk} are defined as random effects as well. The random effects follow $\beta_m \sim N(0, \sigma_1^2)$, $\beta_{mk} \sim N(0, \sigma_2^2)$, $\beta_{mb} \sim N(0, \sigma_3^2)$ and $\epsilon_{mbk} \sim N(0, \sigma^2)$. […] The model allows the following interpretation—of course conditional on the domain $\mathcal{D}$—for an algorithm a_k and a dataset $\mathcal{L}_m$: $\hat{\kappa}_k$ is the algorithm's performance difference from its mean performance conditional on the dataset.*
>
> – Eugster (2011)

Table 11 shows the fixed effects κ_k for the mixed effects model eq. (22) using the misclassification rate as the performance measure. The ParBoost variants have the worst performance—probably a result of the small sample sizes of the UCI binary classification domain. ISLEBoost subsampling features using p-spline decomposition base learners does best, in contrast to the predictive simulation (section 3.2), where it was not among the top performing models. The larger sampling width σ (section 2.1), and in turn lower false negative rate, of ISLEBoost with feature subsampling, compared to BinomialBoosting (fig. 17), works well for this domain. Random forest is also among the best algorithms for this domain (confirming the findings of Eugster (2011)). The estimated fixed effects for BinomialBoosting, ISLEBoost and random forest are very similar though—performance only starts to degrade with the ParBoost models, which are designed for larger datasets. Note that Lasso postprocessing without subsampling does not improve performance for these data—but this may be a result of the denseness of the UCI binary classification domain. Looking at the misclassification rates for the 21 datasets in figs. 19 to 25 confirms the good performance of random forests and ISLEBoost using feature subsampling, which show consistently good results with little variability.

Table 11: Estimated fixed effects (lower is better).

Model	Base Learners	Fixed Effect
ISLEBoost subsampling features	P-Spline Decomp.	0.092
ISLEBoost subsampling features	Stumps	0.100
Random Forest	Trees	0.100
ISLEBoost subsampling observations with Lasso postprocessing	P-Spline Decomp.	0.100
ISLEBoost subsampling features with Lasso postprocessing	Trees	0.104
ISLEBoost subsampling features with Lasso postprocessing	P-Spline Decomp.	0.104
ISLEBoost subsampling observations	P-Spline Decomp.	0.105
ISLEBoost subsampling observations with Lasso postprocessing	Trees	0.106
ISLEBoost subsampling observations	Stumps	0.106
BinomialBoosting	Trees	0.107
ISLEBoost subsampling features with Lasso postprocessing	Stumps	0.107
ISLEBoost with Lasso postprocessing	Trees	0.108
BinomialBoosting	Stumps	0.109
BinomialBoosting	P-Spline Decomp.	0.110
ISLEBoost subsampling observations	Trees	0.110
ISLEBoost subsampling features	Trees	0.111
ISLEBoost subsampling observations with Lasso postprocessing	Stumps	0.117
ISLEBoost with Lasso postprocessing	Stumps	0.132
ISLEBoost with Lasso postprocessing	P-Spline Decomp.	0.143
ParBoost	P-Spline Decomp.	0.150
ParBoost with GLM postprocessing	P-Spline Decomp.	0.150
ParBoost with Lasso postprocessing	P-Spline Decomp.	0.151
ParBoost	Trees	0.153
ParBoost with Lasso postprocessing	Trees	0.167
ParBoost	Stumps	0.169
ParBoost with GLM postprocessing	Stumps	0.196
ParBoost with GLM postprocessing	Trees	0.233
ParBoost with Lasso postprocessing	Stumps	0.256

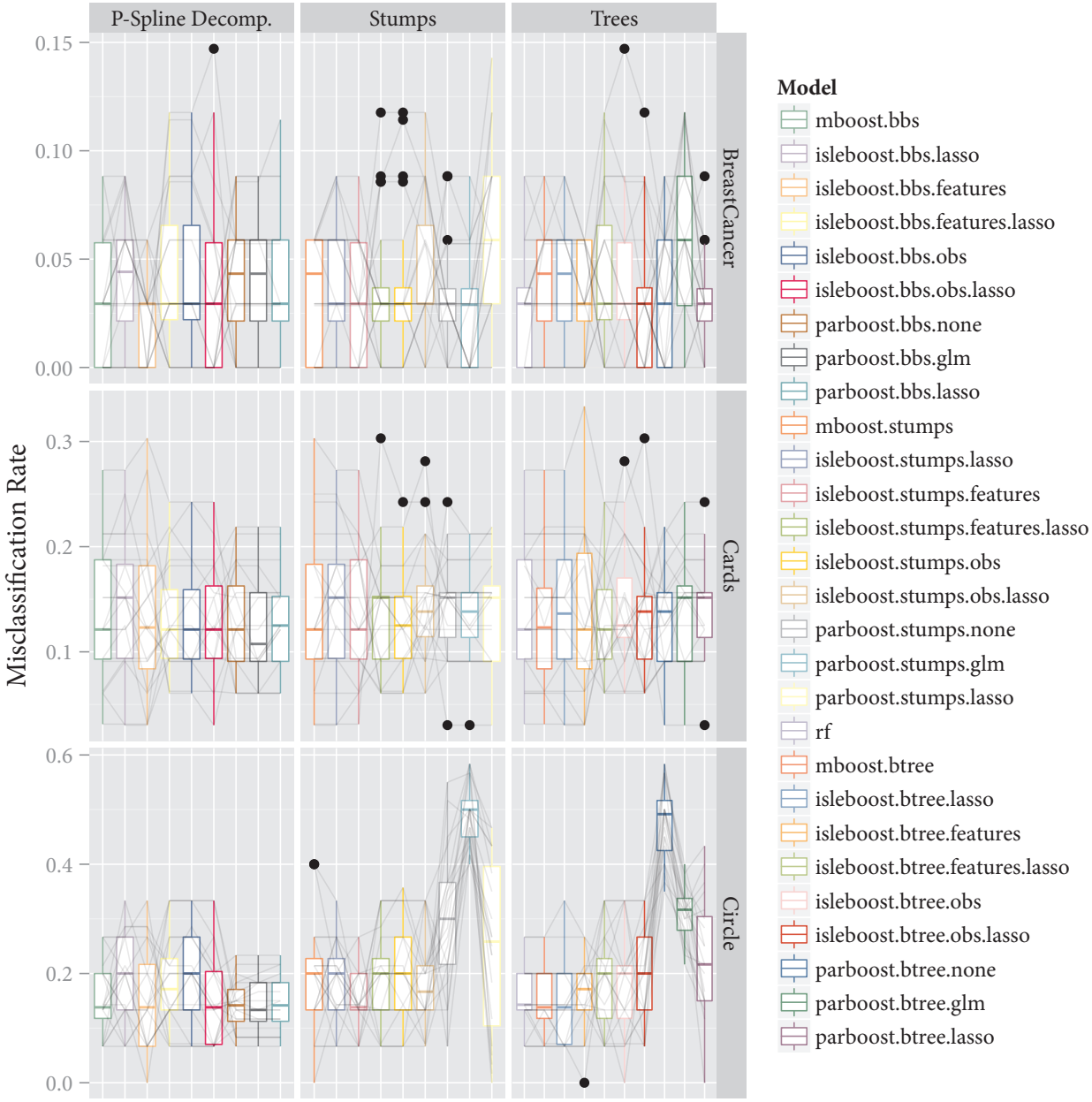


FIGURE 19: UCI binary classification benchmark experiments misclassification rate on test sets using 20-fold cross-validation. Plot 1/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

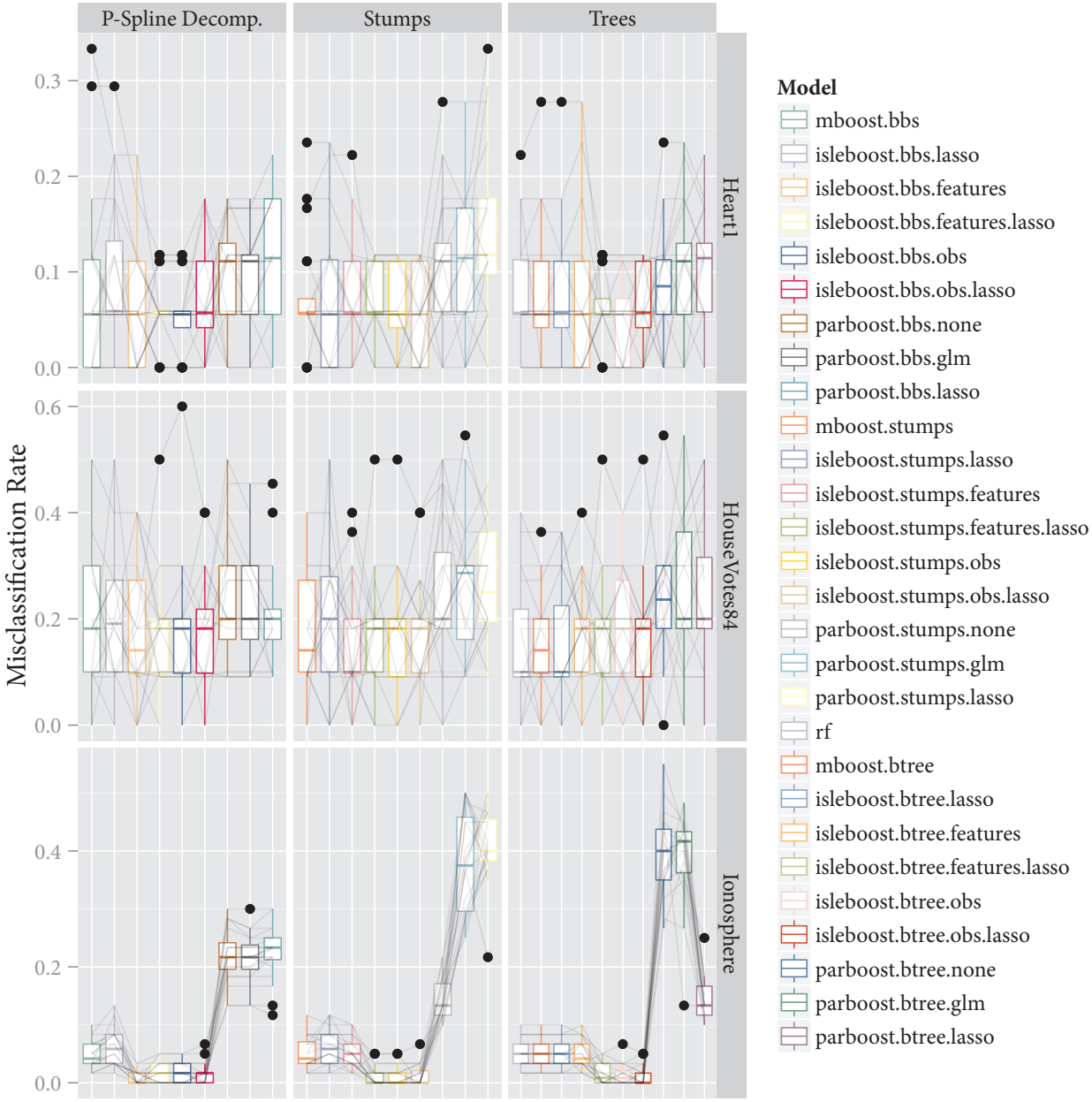


FIGURE 20: UCI binary classification benchmark experiments misclassification rate on test sets using 20-fold cross-validation. Plot 2/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

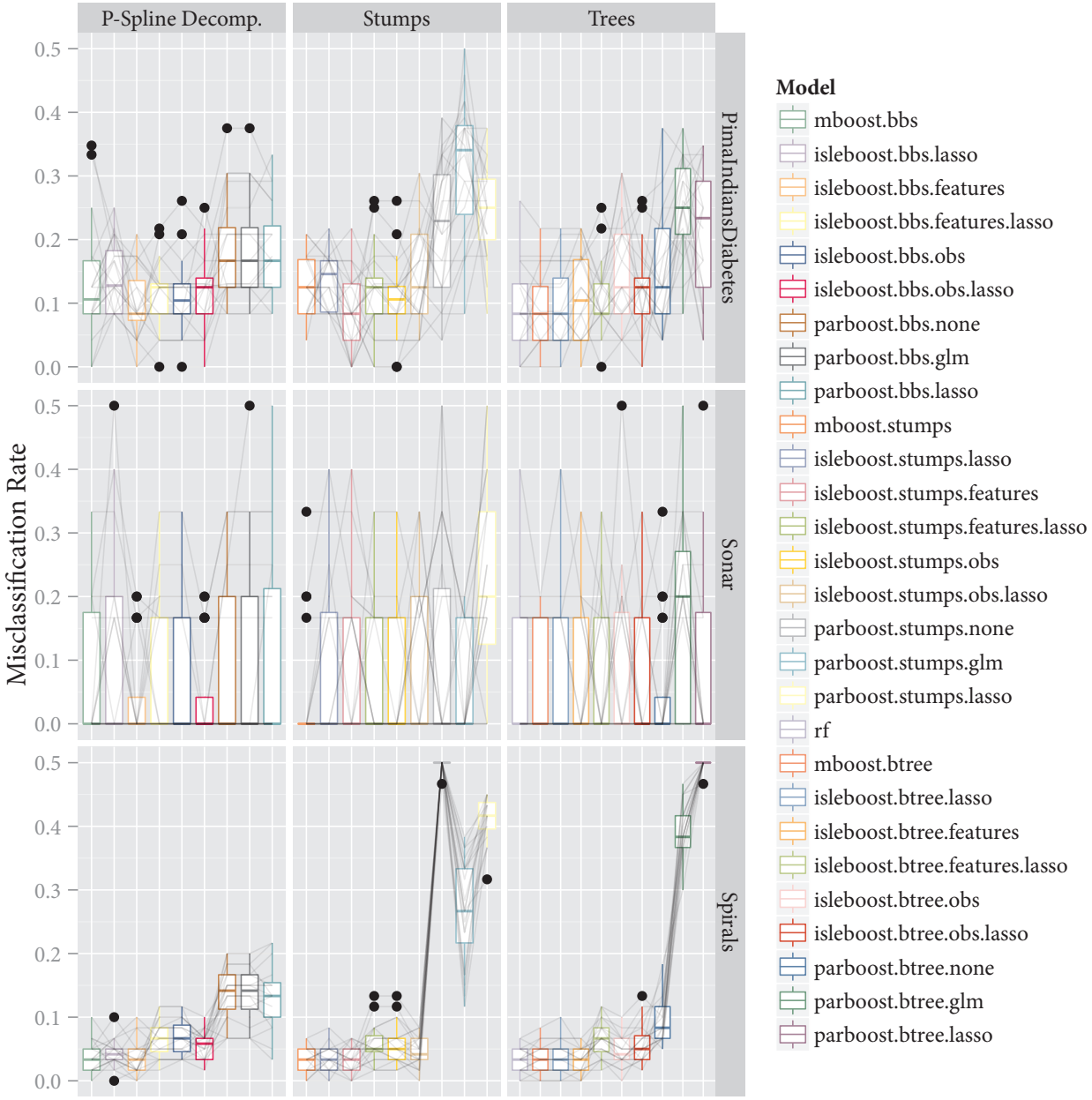


FIGURE 21: UCI binary classification benchmark experiments misclassification rate on test sets using 20-fold cross-validation. Plot 3/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

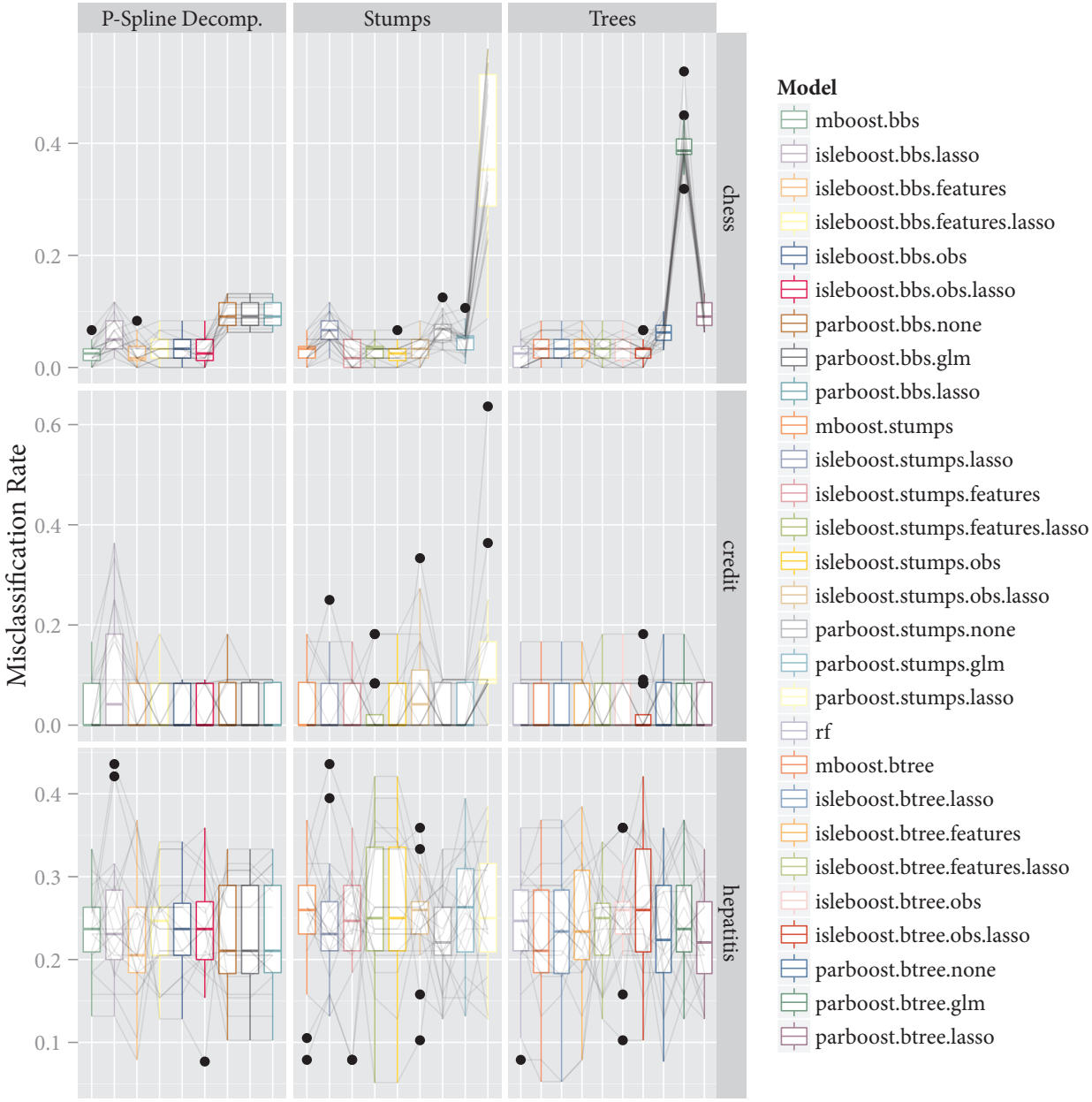


FIGURE 22: UCI binary classification benchmark experiments misclassification rate on test sets using 20-fold cross-validation. Plot 4/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

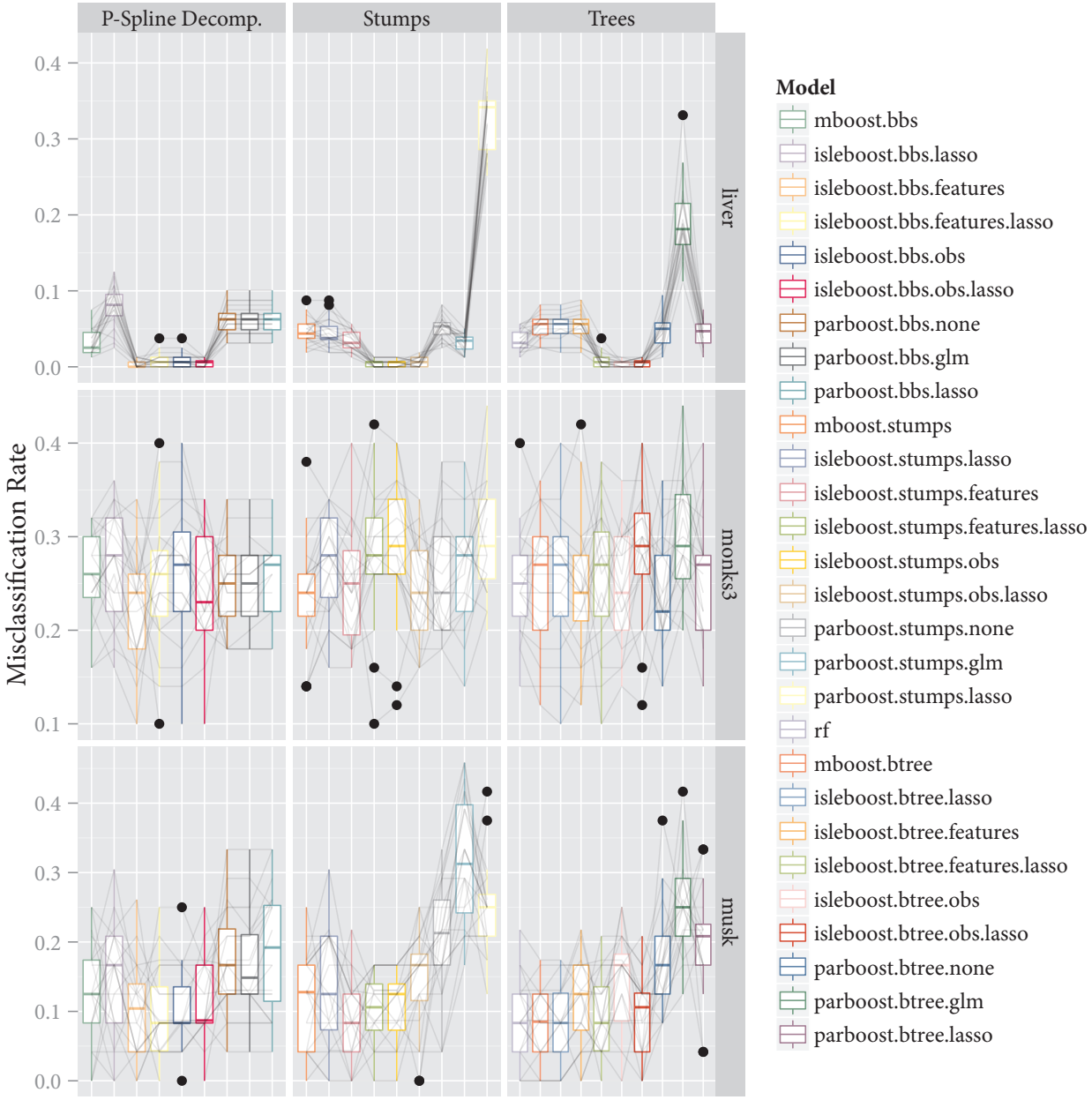


FIGURE 23: UCI binary classification benchmark experiments misclassification rate on test sets using 20-fold cross-validation. Plot 5/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

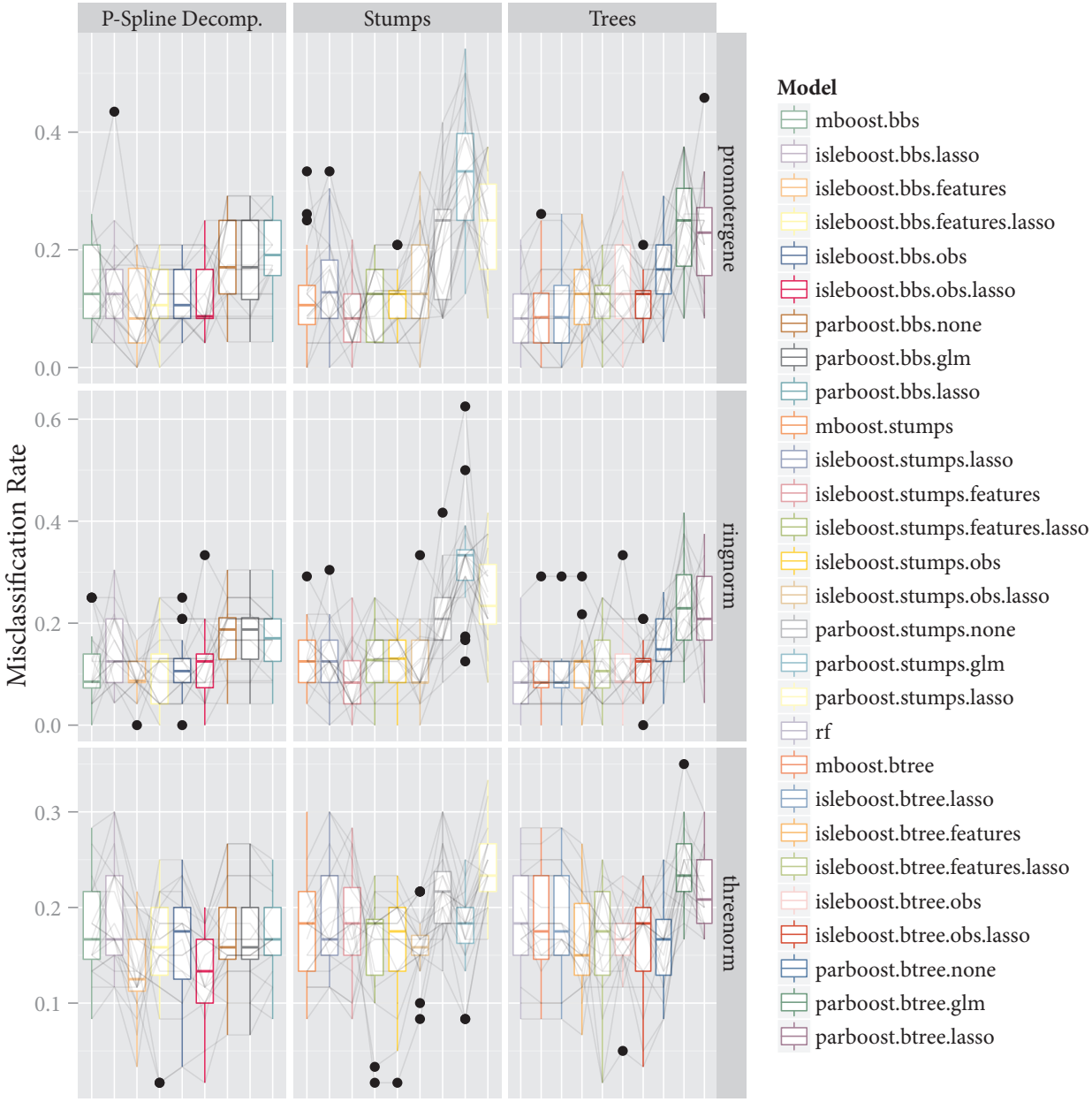


FIGURE 24: UCI binary classification benchmark experiments misclassification rate on test sets using 20-fold cross-validation. Plot 6/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

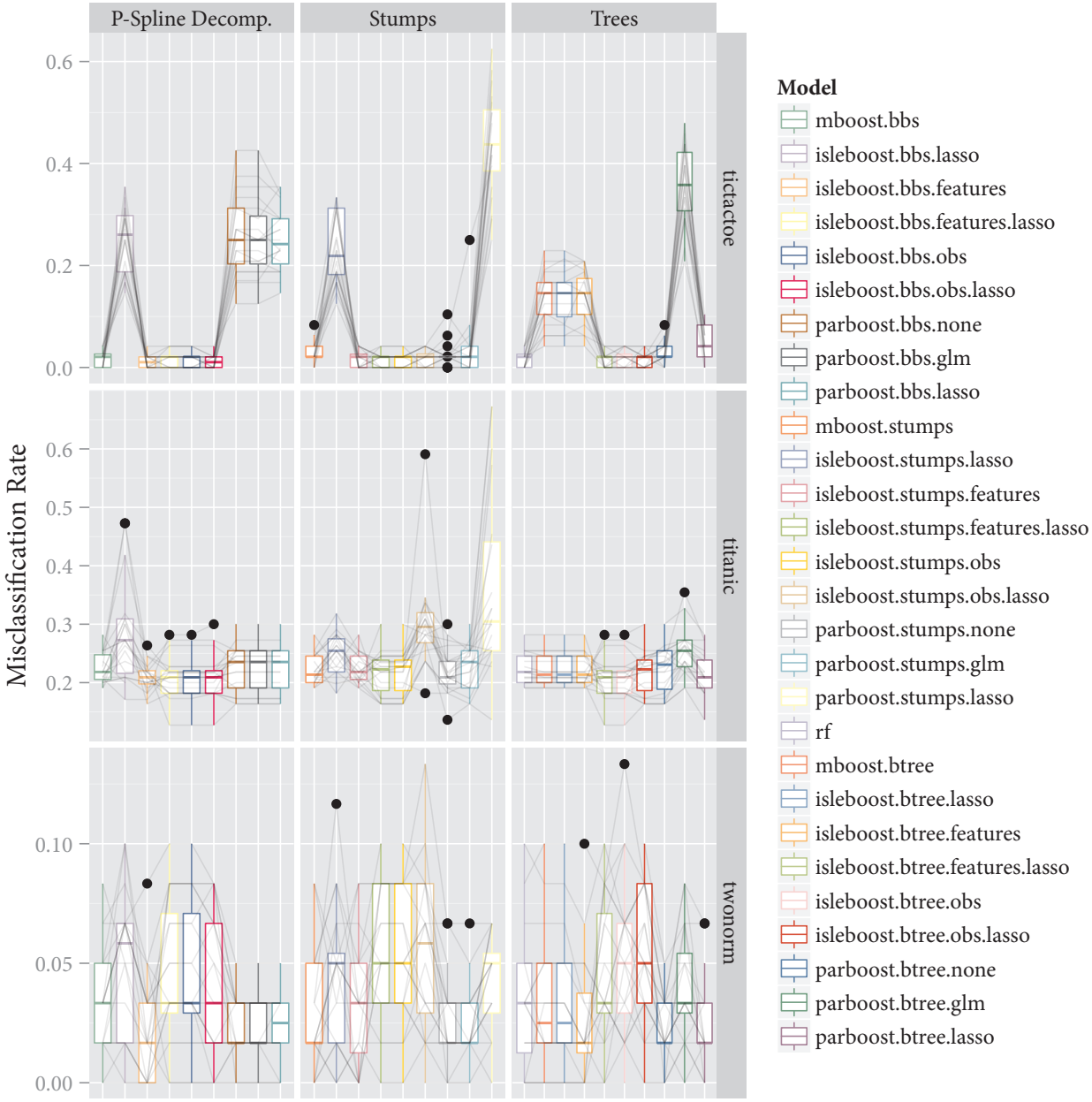


FIGURE 25: UCI binary classification benchmark experiments misclassification rate on test sets using 20-fold cross-validation. Plot 7/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

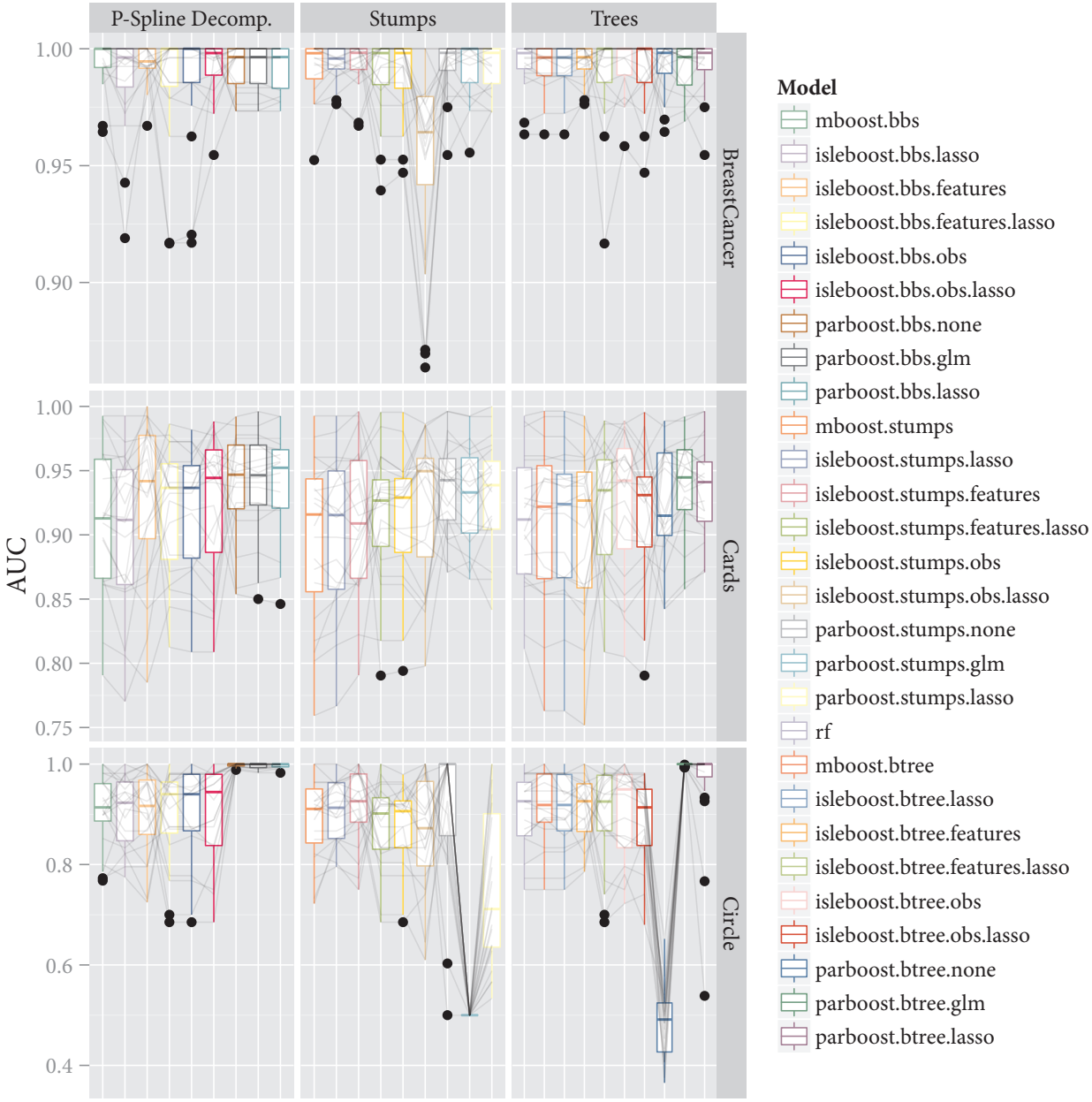


FIGURE 26: UCI binary classification benchmark experiments AUC on test sets using 20-fold cross-validation. Plot 1/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

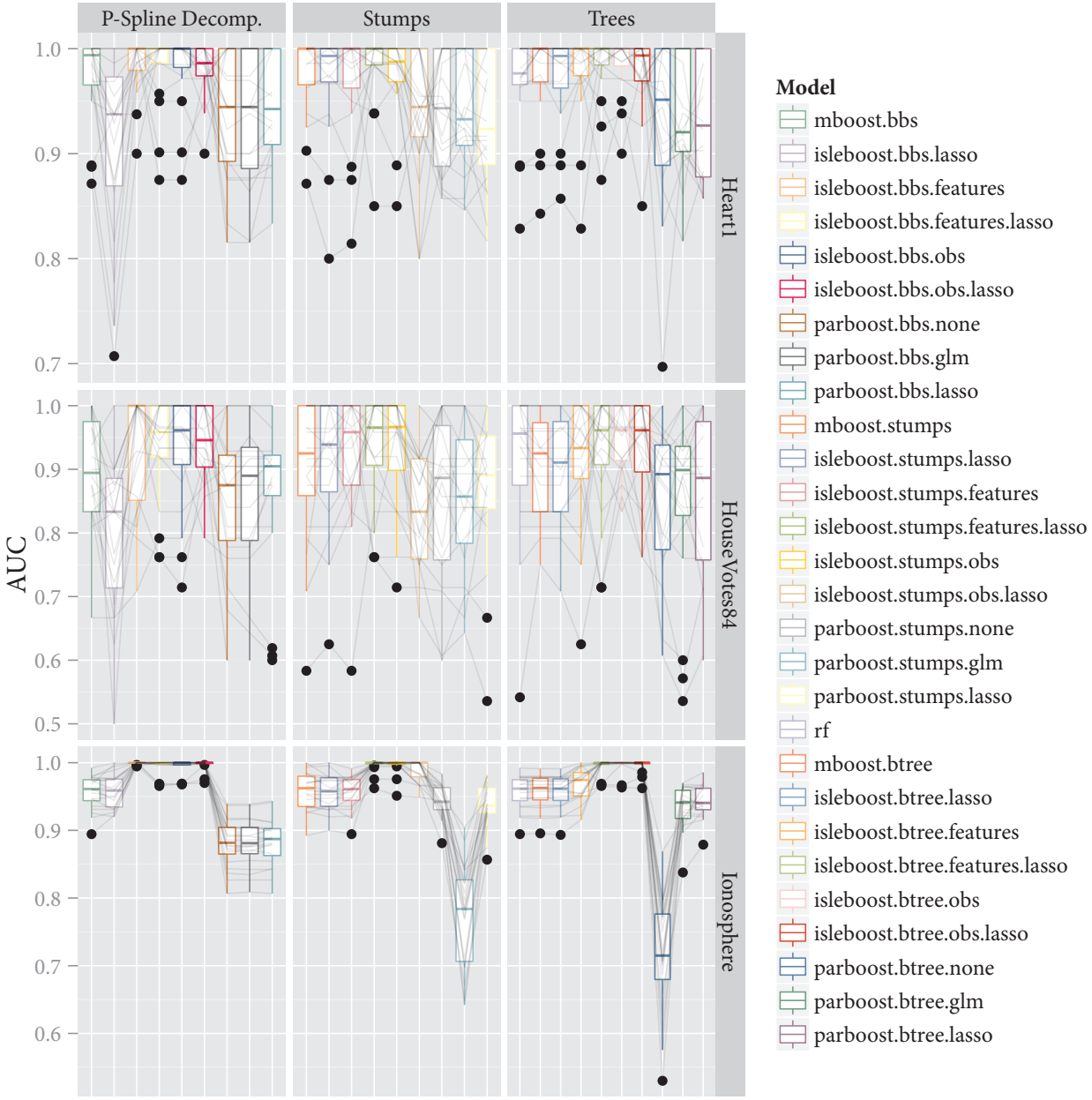


FIGURE 27: UCI binary classification benchmark experiments AUC on test sets using 20-fold cross-validation. Plot 2/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

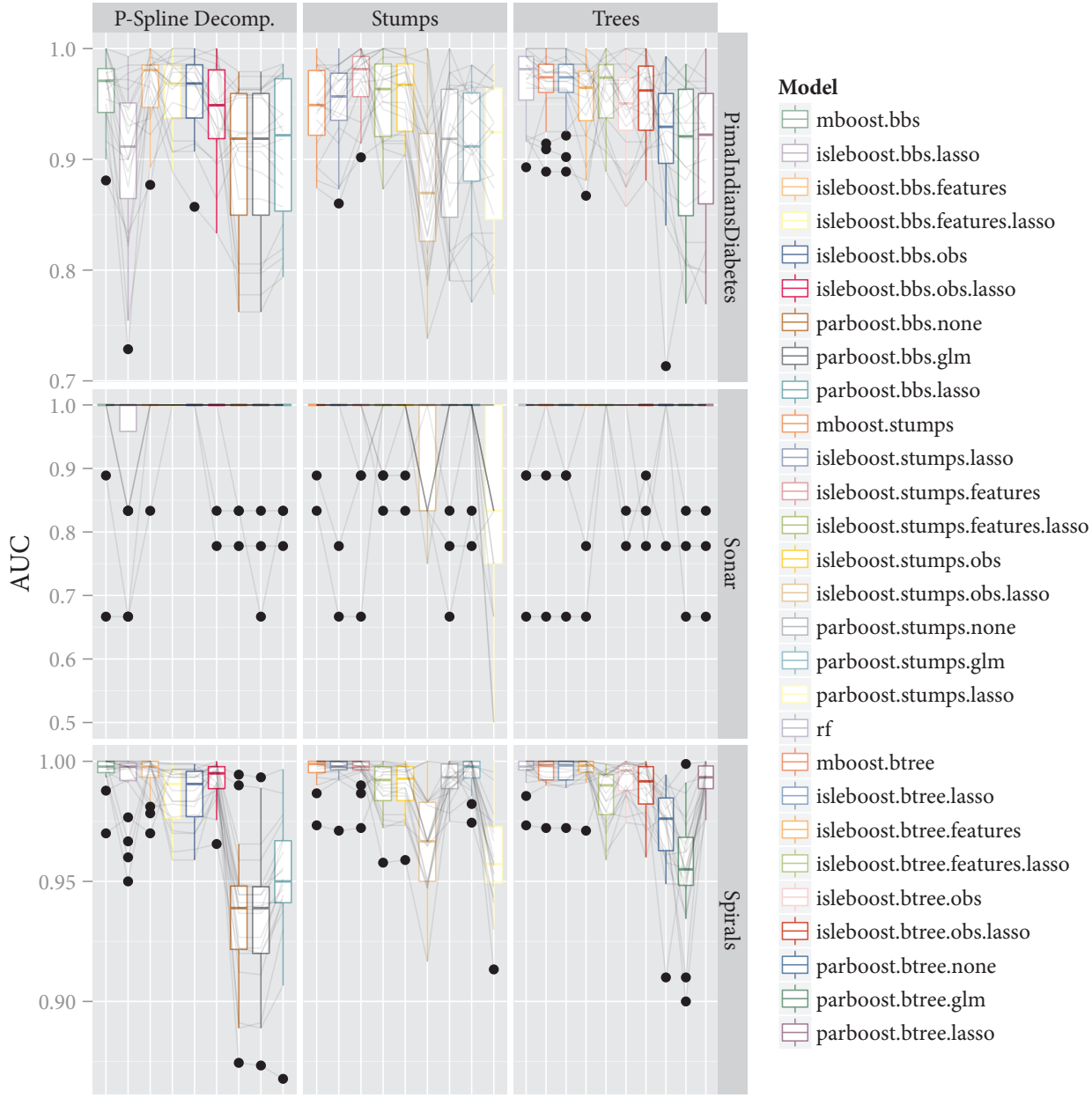


FIGURE 28: UCI binary classification benchmark experiments AUC on test sets using 20-fold cross-validation. Plot 3/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

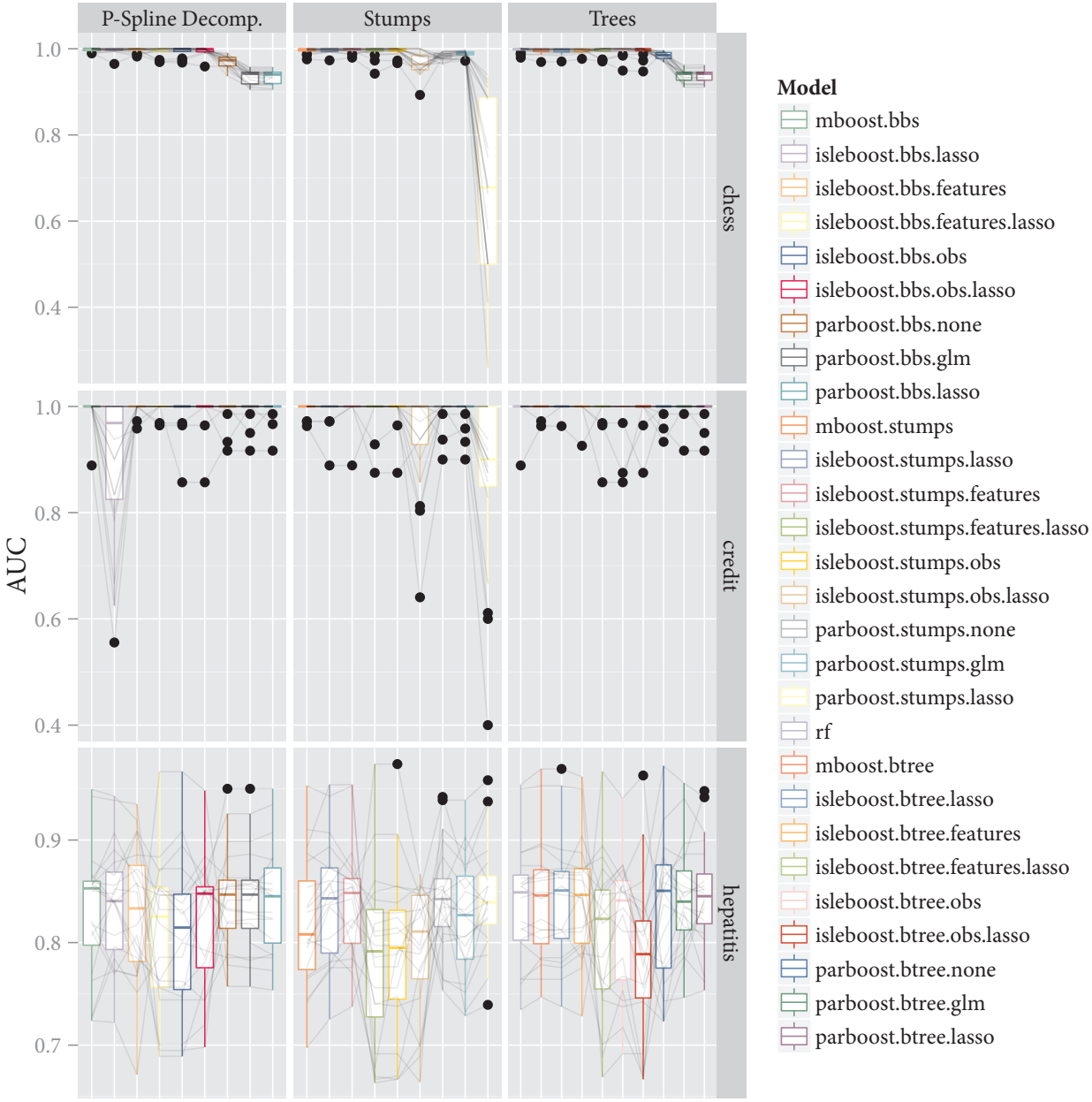


FIGURE 29: UCI binary classification benchmark experiments AUC on test sets using 20-fold cross-validation. Plot 4/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

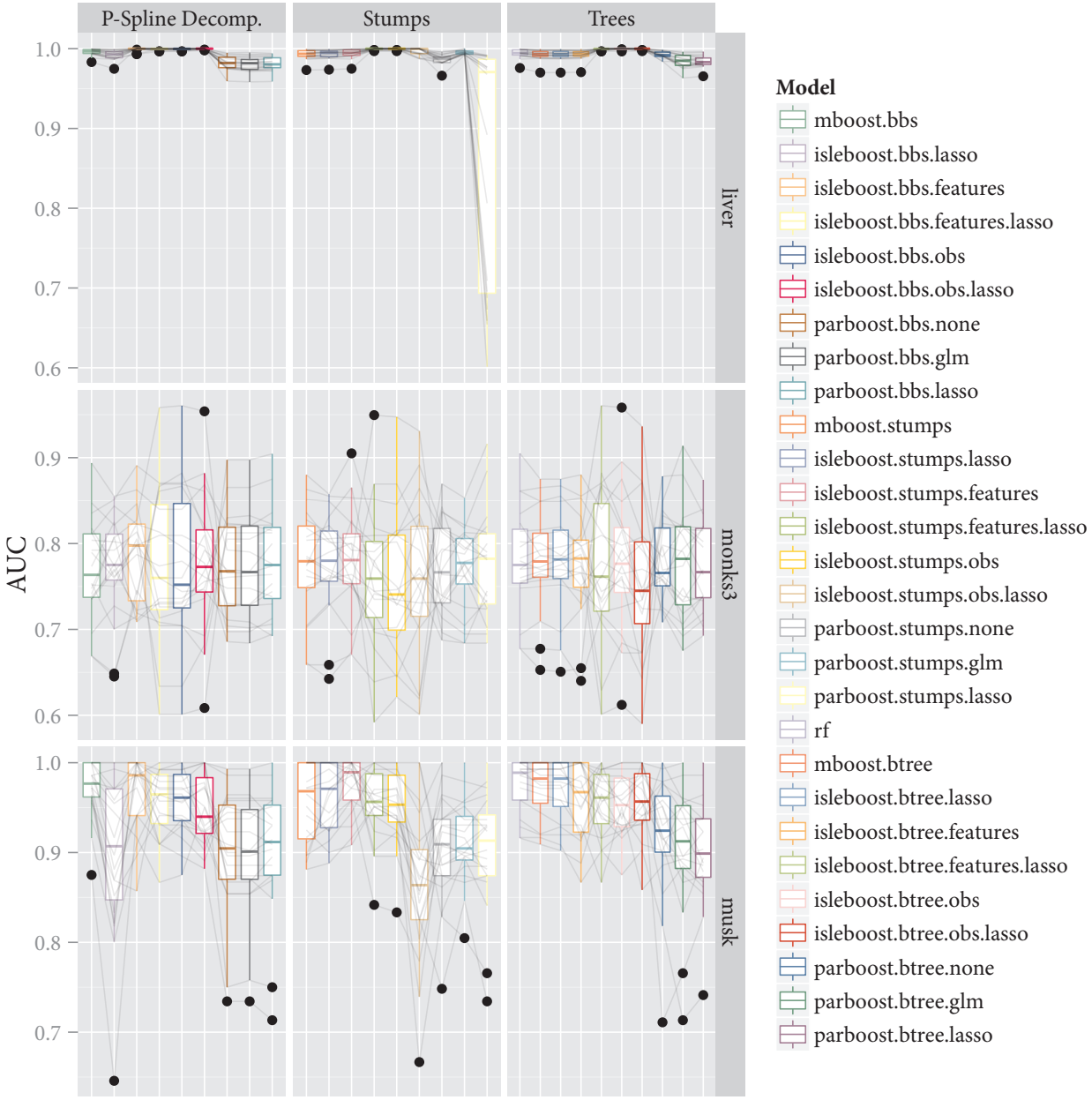


FIGURE 30: UCI binary classification benchmark experiments AUC on test sets using 20-fold cross-validation. Plot 5/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

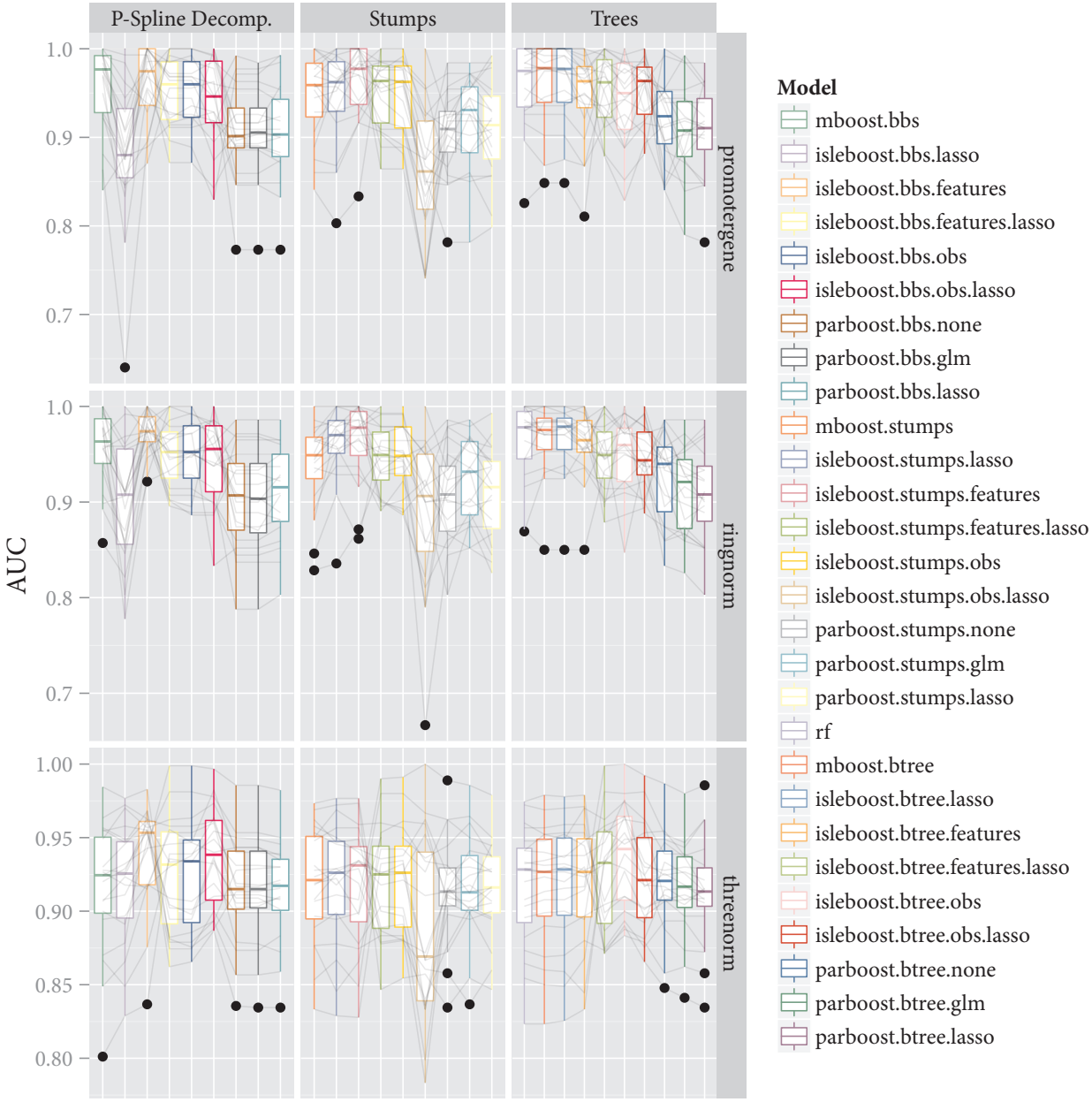


FIGURE 31: UCI binary classification benchmark experiments AUC on test sets using 20-fold cross-validation. Plot 6/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

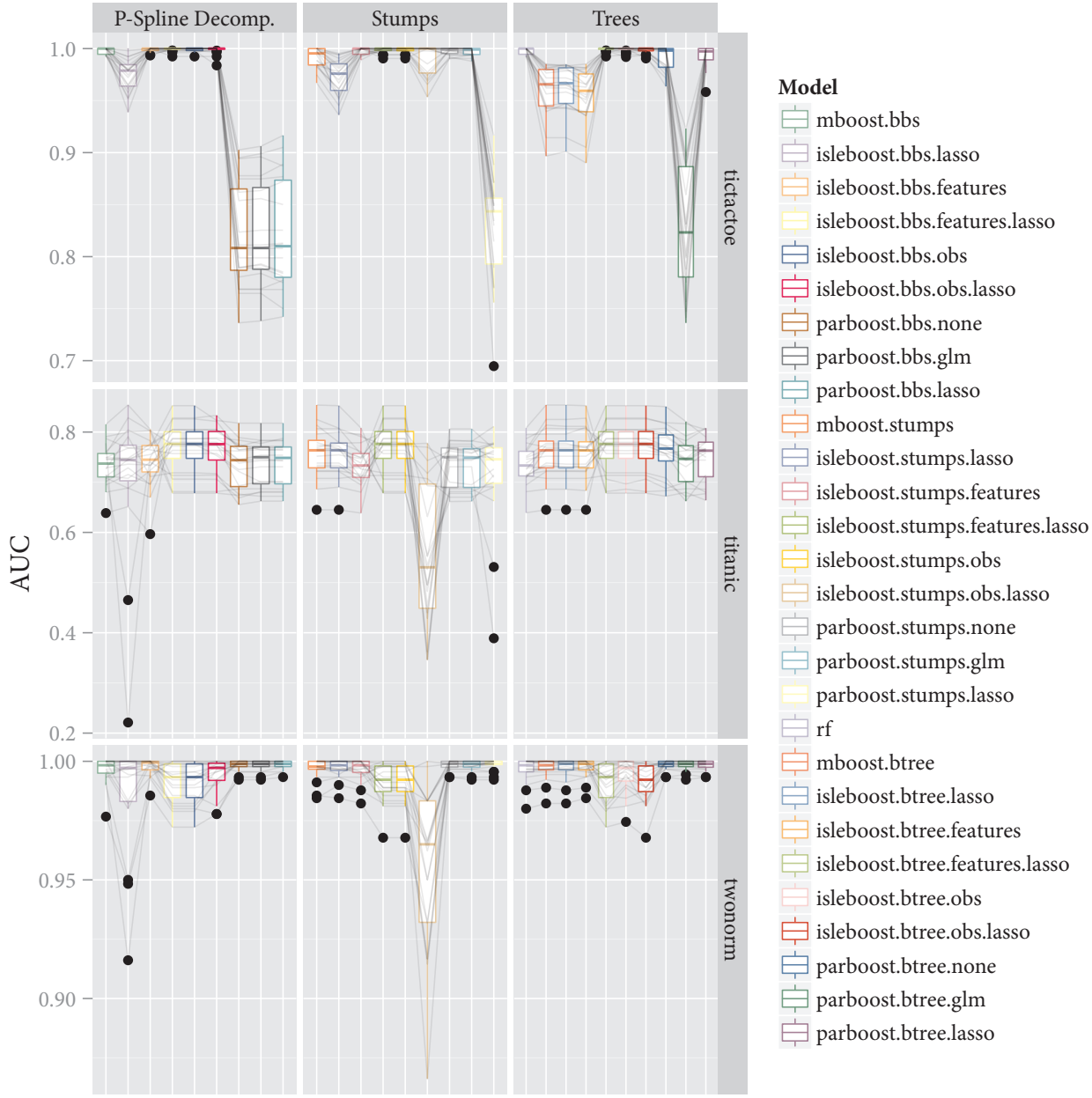


FIGURE 32: UCI binary classification benchmark experiments AUC on test sets using 20-fold cross-validation. Plot 7/7. `mboost` stands for BinomialBoosting, `isleboost` for ISLEBoost, `parboost` for ParBoost and `rf` for random forests. `bbs` are p-spline base learners, `stumps` are tree stump base learners and `trees` are tree base learners with 6-terminal nodes. `features` stands for feature subsampling, `obs` for subsampling of the observations and `lasso` for Lasso postprocessing. The light grey lines connect the folds.

4.2. Million Song Dataset

To demonstrate the scalability and predictive power of `parboost`, I wanted to find a large dataset with baseline models to compare the performance of `parboost` to. Since the `mboost` and `randomForest` packages do not scale to such large data, they could not serve as baseline models. The million song dataset (MSD, Bertin-Mahieux et al. (2011)) fits the requirement well. Each observation consists of a track from 1922–2011, and the task is to predict the year in which a song was released, based on its audio features. "Listeners often have particular affection for music from certain periods of their lives (such as high school), thus the predicted year could be a useful basis for recommendation. Furthermore, a successful model of the variation in music audio characteristics through the years could throw light on the long-term evolution of popular music" (Bertin-Mahieux et al. 2011). The MSD includes 515,576 tracks from 28,223 artists and is split into a training (463,715 observations) and test set (51,861). The features consist of averages and covariances of the timbre vectors of each song, giving a total of 90 features.

The two benchmark algorithms in Bertin-Mahieux et al. (2011) are k-nearest neighbors (k-NN) and Vowpal Wabbit (VW, Langford, Li, and Strehl (2007)), a state-of-the-art online learning algorithm. I trained ParBoost on a 50 node cluster, where each node[16] had 2 CPUs and 17.1 GB of RAM. ParBoost used spline base learners (cubic B-splines with 2^{nd} order difference penalty, 20 knots and 3 degrees of freedom) and ran for 2000 iterations. The training took 27 minutes. The mean absolute errors (MAE) for ParBoost and the baseline models are shown in table 12 (the data on the nearest neighbors models and VW from Bertin-Mahieux et al. (2011)). Even without further tuning (50 node clusters are expensive), ParBoost achieves comparable performance to the best performing algorithm VW.

TABLE 12: Mean absolute errors for the MSD year prediction

Algorithm	MAE
1-NN	9.81
50-NN	7.58
ParBoost	6.60
VW	6.14

5. Discussion

The goal of this thesis is to make gradient boosting more scalable. To this end, I implemented the ISLE boosting framework of J. H. Friedman and Popescu (2003) in an open source software package named `isleboost`—based on the `mboost` package—with the additional features of parallel

[16] The simulation was run on an Amazon EC2 cluster with `m2.xlarge` instances as worker nodes. The master node was a `cr1.8xlarge` instance (as of September 2013).

processing and feature subsampling (section 2.1). This software package is designed to work in a shared memory setting. For a distributed memory setting, I have designed a novel approach called ParBoost. I implemented ParBoost in an open source software package `parboost`, which is freely available on CRAN[17] (section 2.2).

Fitting the base learners in parallel using `isleboost` delivers linear speedup (section 3.1.1) in the number of cores used. Feature subsampling also delivers linear speedup, but subsampling of the observations does not result in any speedup—contrary to the findings of J. H. Friedman (2002) and J. H. Friedman and Popescu (2003) (section 2.1). Postprocessing produces sparser models that require less storage, it is thus computationally less expensive to generate predictions with them. This is potentially useful for real-time applications or embedded devices, which need to minimize storage and computational complexity.

The results of the predictive simulation (section 3.2) for ISLEBoost suggest that the impact of the various techniques (subsampling of the observations or features, with and without postprocessing) on predictive performance depends on the type of response (real-valued or binary) and on the type of base learner. None of the ISLEBoost models outperform the baseline boosting models (L_2Boosting and BinomialBoosting) in any meaningful way though. This is in contrast to the real data experiment of the UCI binary classification domain (section 4.1), where feature subsampling in particular outperforms BinomialBoosting. But this may be a result of the rather dense UCI domain. Overall, I cannot recommend any variant of ISLEBoost that robustly and reliably outperforms the baseline L_2Boosting and BinomialBoosting models in predictive performance, based on the simulation and real data experiments. Nevertheless, when the data is known to be dense, feature subsampling can provide linear speedup and possibly better predictive performance than gradient boosting without subsampling. The reasons for this are the increased sampling width σ (eq. (6)), and in turn the lower false negative rate.

In terms of variable selection (section 3.3), postprocessing reduces false positive rates, at the cost of higher false negative rates. Subsampling features or observations increases the false positive rates and reduces false negatives. This effect is much more pronounced for subsampling observations though, and thus postprocessing also has a larger impact on models that use subsampling of the observations rather than subsampling features—presumedly because of a larger sampling width σ (eq. (6)). Using the Lasso to postprocess a boosting ensemble (without subsampling) reduces the false positive rate, while keeping the false negative rate nearly identical to the same model without postprocessing (fig. 15). This makes postprocessing a viable alternative to other sparse boosting techniques such as SparseL_2Boost (Bühlmann and B. Yu 2006) or Twin Boosting (Bühlmann and Hothorn 2010).

ParBoost scales linearly in the number of cluster nodes (section 3.1.2), above a threshold of computational complexity that exceeds the overhead of setting up the cluster nodes. I have also shown that ParBoost scales to clusters with 50 nodes to process the million song dataset and generates competetive predictions for a test set (section 4.2). In the predictive simulation

[17] `http://cran.r-project.org/web/packages/parboost/`

(section 3.2), ParBoost generally does well, and sometimes even outperforms the baseline boosting models (see fig. 10 and fig. 13), but occasionally some configurations of the various postprocessing schemes produce worrisome outliers in predictive performance (see fig. 13 or fig. 18). I would have liked to compare ParBoost to the other algorithms using even more data and ensemble components to see whether these occasional outliers in performance are due to dataset size and the number of ensemble components. But the shared memory algorithms do not scale as much as ParBoost due to memory limits of single machines. Therefore, I am unable to compare them to ParBoost using larger datasets. Also, I have already exceeded my budget on the current simulations.

Based on the variable selection simulation (section 3.3), ParBoost does not produce very sparse models. This is mainly because it splits the data into disjoint subsamples, making it more difficult for each boosting model to find the relevant coefficients. Combining ParBoost with ISLEBoost using lasso postprocessing to achieve sparser models could be explored in future research. It is difficult to extract meaningful coefficients from ParBoost models because of their neural network like structure (see fig. 5), making them less useful for applications where interpretability is important. ParBoost using bootstrapping to create submodels—instead of splitting the data into disjoint subsets—for the relatively small datasets of the UCI binary classification domain did not do well. For this particular domain, "bagging boosting models" performed markedly worse than BinomialBoosting, ISLEBoost and random forests. ParBoost should be used for datasets that are too large for shared memory boosting and random forests to process, in situations where interpretability is not important. For these situations though, I have shown that ParBoost can deliver very competetive results in a reasonably short amount of time (section 4.2). On small datasets that fit into memory Random forests compared unfavorably to the boosting models in the simulations, but was among the top performers in the real data experiment for the 21 datasets of the UCI binary classification domain (section 4.1). J. H. Friedman and Popescu (2003) had a similar result, where random forests and bagging did worse than boosting for their simulations, but performed well in their real data experiments. Random forests remain difficult to beat in many real-world applications—yet we still have little theory on the algorithm. For example, it was only recently shown to be consistent by Biau (2012).

In future research, I would like to explore the similarities to a neural network structure of ISLEBoost and ParBoost further. In this context, I would also combine ParBoost and ISLEBoost in simulations and real-data experiments. ParBoost could also be implemented in a distributed computing system like Hadoop, since its data locality make it an ideal algorithm for such a distributed system. Furthermore, I would like to rewrite a majority of the `mboost` codebase in C/C++ or Fortran, which would speed up the algorithm.

A. A short introduction to parallel and distributed computing

Parallel computing is about running independent computations on multiple cores or machines simultaneously to shorten their running time. This thesis deals with two types of parallelism:

Shared memory Modern CPUs have multiple cores that can execute tasks in parallel which share access to the same memory—hence *shared* memory. To make use of these capabilities, the programmer has to decide on which level to parallelize his code. There is high-level parallelism, such as running cross-validation folds in parallel, and low-level parallelism, e.g. calculating an individual sum in parallel by dividing it in chunks. Depending on the task at hand, different granularities of parallelism can offer varying gains in efficiency. Using multiple cores in any given computation introduces overhead: tasks have to be distributed to each core and then collected again. Thus parallelizing any given computation is only sensible above some threshold where the computational load of the task is high enough to outweigh the overhead—depending on the programming language, hardware and level of parallelization used.

Distributed memory Shared memory parallelism is limited by the number of cores available in current CPUs. Distributed memory computing works by linking many computers together in a *cluster* over ethernet. Each node has its own memory—hence *distributed* memory. The advantage of distributed memory over shared memory is its scalability: a cluster can consist of hundreds of machines. The downside is, that network connections are comparatively slow: communication between the nodes introduces a huge overhead compared to shared memory parallelism. Therefore it is usually a good idea to only switch to distributed memory parallelism once the technical limits in scalability of the shared memory framework are reached.

B. Electronic supplement

B.1. `isleboost`

The `isleboost` R-package implements the ISLEBoost algorithm described in section 2.1. Since the `isleboost` package is a fork of the `mboost` package (Hothorn et al. 2012), I will only discuss the additional features of `isleboost`. `mboost` is the most comprehensive software package implementing gradient boosting: it comes with a large variety of loss functions and base learners. For the documentation for the basic `mboost` functionality, please see the help files included in `isleboost` or visit `http://cran.r-project.org/web/packages/mboost/`. `isleboost` adds the following features to `mboost`:

- Parallel computation of the base learners.

- Subsampling of features or observations.
- Postprocessing the boosting ensemble with regularized regression, using either a Lasso, Ridge or Elastic Net penalty.

All source files of `isleboost` can be found in the `isleboost` directory of the electronic supplement.

B.1.1. Parallel computation

There is a caveat with parallelizing any computation: it can actually *increase* computation time due to the overhead of the parallel processing (see appendix A). With that being said, `isleboost` can significantly speed up boosting on larger datasets (see section 3.1.1). On a UNIX type system, the multicore (`mclapply`) backend is the fastest. To fit a model in parallel, simply specify the backend you wish to use (`parallel = "mclapply"`) and the number of cores (`cores = 4`):

```
library(isleboost)
data("bodyfat")

ctrl <- boost_control(mstop = 1000, parallel = "mclapply", cores = 4)
model <- isleboost(formula = DEXfat ~ ., data = bodyfat, baselearner = "bbs")
```

On a Windows based system, the `fork` function that `mclapply` relies on is not supported. Thus a local cluster should be used by specifying `parallel = "parLapply"` and passing a cluster object to `boost_control`:

```
library(isleboost)
data("bodyfat")

cl <- makeCluster(4)
ctrl <- boost_control(mstop = 1000, parallel = "parLapply", cluster = cl)
model <- isleboost(formula = DEXfat ~ ., data = bodyfat, baselearner = "bbs")
```

The only difference to `mboost` thus far, is that the baselearners are fitted in parallel (step 2 in algorithm 2.

B.1.2. Subsampling

The `subsample` parameter specifies the proportion of observations or features the user wishes to subsample. The `type` parameter determines the type of subsampling (`"observations"` or `"features"`). Finally, the logical value `replace` regulates whether to sample with or without replacement (defaults to `FALSE`). So to subsample 10 % of the observations without replacement in each iteration, the user would specify:

```
library(isleboost)
data("bodyfat")

ctrl <- boost_control(mstop = 1000, subsample = 0.1, type = "observations")
model <- isleboost(formula = DEXfat ~ ., data = bodyfat, baselearner = "bbs")
```

To subsample 50 % of the features with replacement, the call to `isleboost` would look like this:

```
library(isleboost)
data("bodyfat")

ctrl <- boost_control(mstop = 1000, subsample = 0.5, type = "features", replace = TRUE)
model <- isleboost(formula = DEXfat ~ ., data = bodyfat, baselearner = "bbs")
```

A `seed` can also be passed to `boost_control` for reproducible results. The seed value gets incremented by 1 in each iteration.

B.1.3. Postprocessing

Postprocessing the boosting ensemble as described in section 2.1 can also be specified in `boost_control` with the `postprocess` parameter, which can take the values `"none"` (default), `"lasso"`, `"ridge"` and `"elasticnet"`. To fit a model using subsampling of the observations and lasso postprocessing, enter the following:

```
library(isleboost)
data("bodyfat")

ctrl <- boost_control(mstop = 1000, subsample = 0.1, type = "observations", postprocess = "lasso")
model <- isleboost(formula = DEXfat ~ ., data = bodyfat, baselearner = "btree")
```

The base learners will then be weighed by their lasso coefficients for increased sparsity.

B.2. `parboost`

The `parboost` package is available on CRAN[18] and implements the ParBoost algorithm from section 2.2. Its purpose is to run on a cluster of machines using a cluster backend, but execution on a single machine is supported. The individual boosting models are fit using the `mboost` (Hothorn et al. 2012) package. As an example, I will use the million song dataset from section 4.2.

[18] http://cran.r-project.org/web/packages/parboost/

```r
# Load libraries
library(parboost)
path <- "/Masterarbeit/real_data_experiments/msd/train.csv"

# Start cluster
cluster_names <- paste("node00", 1:9, sep="")
cluster_names <- c(cluster_names, paste("node0", 10:50, sep=""))
cluster <- makePSOCKcluster(names = cluster_names)

# Estimate model and time it
formula <- V1 ~ .
ctrl <- boost_control(mstop = 2000)
model <- parboost(cluster_object = cluster,
                  path_to_data = path, nsplits = 50,
                  formula = formula, baselearner = "bbs",
                  control = ctrl, postprocessing = "none")

# Generate testset predictions
path.test <- "/Masterarbeit/real_data_experiments/msd/testset.csv"
testset <- read.csv(path.test)
preds <- predict(model, newdata = testset[, -2])
stopCluster(cluster)
```

The `parboost` function takes a cluster object from the `parallel` package of base R. Simply pass the node hostnames or IPs to the `makePSOCKcluster` function, which can then be used to execute `parboost` on the cluster nodes. In appendix B.4, I explain how to set up a cluster on Amazon EC2[19] with StarCluster[20]. The `boost_control` function comes from the `mboost` package and controls the boosting parameters. It is more efficient to pass the path of the data to `parboost` instead of the data itself. This is because if all the cluster nodes have access to the data locally, it is faster if each node just reads the data from disk, instead of receiving it over the network from the master. You can pass a data frame instead of the path with the `data` argument though. The `nsplits` argument determines the number of disjoint subsets the data is split into.

Currently `parboost` supports spline (`bbs`), tree (`btree`) and piecewise linear (`bols`) baselearners. The postprocessing options (see section 2.2) are `none` for equal weights of the sub-models—the other four options (`glm`, `lasso`, `ridge`, `elasticnet`) for postprocessing perform (regularized) regression on the predictions of the sub-models to determine their weight. Note that the response functions of the postprocessing models change with the family of the underlying boosting model—e.g. a Poisson boosting family will result in a Poisson postprocessing family.

Another option that `parboost` offers is bootstrapping the original data instead of using disjoint subsamples. This can be used for creating more robust models on smaller datasets. Bootstrapping can be invoked with the `split_data = "bagging"` option like so (this time with GLM postprocessing and passing a data frame directly to `parboost`):

[19] http://aws.amazon.com/ec2/
[20] http://star.mit.edu/cluster/docs/latest/overview.html

```
data <- read.csv(path)
model <- parboost(split_data = "bagging", mc.cores = 4,
                  data = data, nsplits = 50,
                  formula = formula, baselearner = "bbs",
                  control = ctrl, postprocessing = "glm")
```

Here the `nsplits = 50` parameter determines the number of bootstrap samples instead of the number of disjoint subsets. Note the lack of a cluster object and the parameter `mc.cores = 4`. This tells `parboost` to execute locally using 4 CPU cores. If you need finer control on how the data gets read on the cluster nodes, you can pass a custom data import function and a preprocessing function. For example, if the file does not have a header and you wish to read the first 10 columns, you could do this:

```
model <- parboost(cluster_object = cluster,
                  path_to_data = path,
                  data_import_function = function(x) read.csv(x, header = FALSE)[, 1:10],
                  nsplits = 50,
                  formula = formula, baselearner = "bols",
                  control = ctrl, postprocessing = "lasso")
```

Each cluster node will then import the data according to the import function you specified. Additionally, you can preprocess the data, e.g. centering and scaling all the variables:

```
model <- parboost(cluster_object = cluster,
                  path_to_data = path,
                  data_import_function = function(x) read.csv(x, header = FALSE)[, 1:10],
                  preprocessing = function(x) scale(x),
                  nsplits = 50,
                  formula = formula, baselearner = "bols",
                  control = ctrl, postprocessing = "lasso")
```

By default, `parboost` will use 10-folds cross-validation to determine the optimal value of m_{stop}. You can change that behaviour like so:

```
model <- parboost(cluster_object = cluster,
                  path_to_data = path,
                  nsplits = 50,
                  formula = formula, baselearner = "btree",
                  control = ctrl, postprocessing = "elasticnet",
                  folds = 25, stepsize_mstop = 10)
```

Now `parboost` will use 25-fold cross-validation and a grid with a stepsize of 10 for optimizing m_{stop}. You can also completely turn off cross-validation by setting `cv = FALSE`. Several methods are available for `parboost` models:

- `print(model)` Prints information about the estimated model.
- `summary(model)` Prints a summary of the model.

- `predict(model)` Generates predictions for new data, if no new data is supplied, it returns the fitted values.

- `coef(model)` Displays the estimated coefficients for all the models (for base learners that have a notion of coefficients).

- `selected(model)` Displays the selected coefficients of all the models

If you need additional help, type `?parboost` in your R console. All of the source files for `parboost` are available in the `parboost` directory of the electronic supplement. They can also be found online at CRAN[21].

B.3. Simulations and real data experiments

All R-files for the simulations can be found in the `simulations` folder of the electronic supplement. The R-scripts I used to conduct the real data experiments, including the data itself, can be found in the `real_data_experiments` directory of the electronic supplement. I carried out all simulations and real data experiments on Amazon EC2 with R version 3.01 using the Amazon Linux AMI[22]. Except for the simulations making use of clusters, I used `cr1.8xlarge` instances. For the clusters, I used `m2.xlarge` instances (configurations as of September 2013).

B.4. Setting up StarCluster with R

> *StarCluster is a utility for creating and managing distributed computing clusters hosted on Amazon's Elastic Compute Cloud (EC2). StarCluster utilizes Amazon´s EC2 web service to create and destroy clusters of Linux virtual machines on demand.*
>
> – Riley (2013)

StarCluster provides a convenient way to quickly set up a cluster of machines to run `parboost` and other tools using a distributed memory framework. I give a brief introduction here to using StarCluster with R, so my simulation results can easily be reproduced.

Install StarCluster using

```
$ sudo easy_install StarCluster
```

[21] http://cran.r-project.org/web/packages/parboost/
[22] http://aws.amazon.com/amazon-linux-ami/

and then create a configuration file using

```
$ starcluster help
```

Add your AWS credentials to the configuration file and follow the instructions on http://star.mit.edu/cluster/docs/latest/quickstart.html.

Once you have StarCluster up and running, you need to install R on all the cluster nodes and any packages you require. Here is a shell script to automate the process:

```
#!/bin/zsh

starcluster put $1 starcluster.setup.zsh /home/starcluster.setup.zsh
starcluster put $1 Rpkgs.R /home/Rpkgs.R

numNodes=`starcluster listclusters | grep "Total nodes" | cut -d' ' -f3`
nodes=(`eval echo $(seq -f node%03g 1 $(($numNodes-1)))`)

for node in $nodes; do
    cmd="source /home/starcluster.setup.zsh >& /home/install.log.$node"
    starcluster sshmaster $1 "ssh $node $cmd" &
done
echo "Installation finished on Nodes"
starcluster sshmaster $1 $cmd &
echo "Installation finished on Master"
```

The script takes the name of your cluster as a parameter and pushes the two helper files to the cluster. It then runs the installation on the master and every node. It assumes you are running an Ubuntu Server based StarCluster AMI, which is the default. The first helper script, `starcluster.setup.zsh`, installs the basic software required:

```
#!/bin/zsh

echo "deb http://stat.ethz.ch/CRAN/bin/linux/ubuntu precise/" >> /etc/apt/sources.list
gpg --keyserver keyserver.ubuntu.com --recv-key E084DAB9
gpg -a --export E084DAB9 | sudo apt-key add -
apt-get update
apt-get install -y r-base r-base-dev
echo “DONE with Ubuntu package installation on $(hostname -s).”
R CMD BATCH --no-save /home/Rpkgs.R /home/install.Rpkgs.log
echo “DONE with R package installation on $(hostname -s).”
```

The second script, `Rpkgs.R`, is just a R script containing the packages you wish to install:

```
install.packages(c("randomForest", "caret", "mboost", "parboost", "plyr", "glmnet"),
 repos = "http://cran.cnr.berkeley.edu")
print(paste("DONE with R package installation on ", system("hostname -s", intern = TRUE), "."))
```

Once you have everything installed, you can `ssh` into your master node and start up R as usual:

```
$ starcluster sshmaster mycluster
$ R
```

Since StarCluster has set up all the networking nicely, you can use `parLapply` from the `parallel` package to run a task on your cluster without further configuration:

```
library("parallel")
cluster_names <- paste("node00", 1:9, sep="")
cluster_names <- c(cluster_names, "node010")
cluster <- makePSOCKcluster(names = cluster_names)
output <- parLapply(cluster, input, some_function)
stopCluster(cluster)
```

The above scripts for using StarCluster can be found in the `StarCluster` directory of the electronic supplement.

References

Appel, R. et al. (2013). 'Quickly Boosting Decision Trees-Pruning Underachieving Features Early'. In: *Journal of Machine Learning Research* 28.

Bertin-Mahieux, T. et al. (2011). 'The million song dataset'. In: *In Proceedings of the 12th International Society for Music Information Retrieval Conference ISMIR*.

Biau, G. (2012). 'Analysis of a random forests model'. In: *The Journal of Machine Learning Research*.

Breiman, L. (1996). 'Bagging predictors'. In: *Machine learning* 24.2, pp. 123–140.

— (1999a). 'Pasting bites together for prediction in large data sets and on-line'. In: *Machine learning* 36(1&2).

— (2001). 'Random forests'. In: *Machine learning* 45.1, pp. 5–32.

Breiman, L. (1998). 'Arcing classifier (with discussion and a rejoinder by the author)'. In: *The Annals of Statistics* 26.3, pp. 801–849.

— (1999b). 'Prediction games and arcing algorithms'. In: *Neural computation* 11.7, pp. 1493–1517.

Bühlmann, P. and B. Yu (2003). 'Boosting With the L 2 Loss'. In: *Journal of the American Statistical Association* 98.462, pp. 324–339.

— (2006). 'Sparse boosting'. In: *The Journal of Machine Learning Research* 7, pp. 1001–1024.

Bühlmann, P. and T. Hothorn (2007). 'Boosting Algorithms: Regularization, Prediction and Model Fitting'. In: *Statistical Science* 22.4, pp. 477–505.

— (2010). 'Twin boosting: improved feature selection and prediction'. In: *Statistics and computing* 20.2, pp. 119–138.

Copas, J. B. (1983). 'Regression, prediction and shrinkage'. In: *Journal of the Royal Statistical Society Series B*.

Dean, J. and S. Ghemawat (2008). 'MapReduce: simplified data processing on large clusters'. In: *Communications of the ACM* 51.1, pp. 107–113.

Eugster, M. J. A. (2011). 'Benchmark Experiments'. PhD thesis. Munich: Dr. Hut-Verlag.

Fan, W., S. J. Stolfo, and J. Zhang (1999). 'The application of AdaBoost for distributed, scalable and on-line learning'. In: *Proceedings of the fifth ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 362–366.

Fawcett, T. (2004). 'ROC graphs: Notes and practical considerations for researchers'. In: *Machine learning* 31, pp. 1–38.

Freund, Y. and R. E. Schapire (1996). 'Experiments with a new boosting algorithm'. In: *Machine Learning: Proc. Thirteenth International Conference*. Morgan Kauffman, pp. 148–156.

Friedman, J. H. (2002). 'Stochastic gradient boosting'. In: *Computational Statistics and Data Analysis* 38.4, pp. 367–378.

Friedman, J. H. and B. E. Popescu (2003). 'Importance sampled learning ensembles'. In: *Statistics, Stanford University, Tech. Report*.

Friedman, J., T. Hastie, and R. Tibshirani (2000). 'Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors)'. In: *The Annals of Statistics* 28.2, pp. 337–407.

Friedman, J., T. Hastie, and R. Tibshirani (2010). 'Regularization paths for generalized linear models via coordinate descent'. In: *Journal of Statistical Software* 33.1, p. 1.
Friedman, J. H. (2001). 'Greedy function approximation: a gradient boosting machine'. In: *The Annals of Statistics*, pp. 1189–1232.
Friedman, J., T. Hastie, S. Rosset, et al. (2004). 'Discussion of boosting papers'. In: *The Annals of Statistics* 32, pp. 102–107.
Geman, S., E. Bienenstock, and R. Doursat (1992). 'Neural networks and the bias/variance dilemma'. In: *Neural computation* 4.1, pp. 1–58.
Grama, A. et al. (2003). *Introduction to Parallel Computing*. Pearson Education.
Hager, W. W. (1989). 'Updating the inverse of a matrix'. In: *SIAM review* 31.2, pp. 221–239.
Hastie, T. (2007). 'Comment: Boosting Algorithms: Regularization, Prediction and Model Fitting'. In: *Statistical Science* 22.4, pp. 513–515.
Hastie, T., R. Tibshirani, and J. H. Friedman (2009). *The Elements of Statistical Learning*. Data Mining, Inference, and Prediction, Second Edition. Springer.
Hoerl, A. E. and R. W. Kennard (1970). 'Ridge regression: applications to nonorthogonal problems'. In: *Technometrics* 12.1, pp. 69–82.
Hofner, B. et al. (2011). 'A framework for unbiased model selection based on boosting'. In: *Journal of Computational and Graphical Statistics* 20.4, pp. 956–971.
Hothorn, T. et al. (2012). *Model-Based Boosting*. R package version 2.1-2. URL: http://CRAN.R-project.org/package=mboost.
Hsu, D. et al. (2012). 'Parallel Online Learning'. In: *Scaling Up Machine Learning*. Ed. by R. Bekkerman, M. Bilenko, and J. Langford. Cambridge University Press.
Kneib, T., T. Hothorn, and G. Tutz (2009). 'Variable selection and model choice in geoadditive regression models'. In: *Biometrics* 65, pp. 626–634.
Langford, J., L. Li, and A. L. Strehl (2007). *Vowpal Wabbit (fast online learning)*. URL: http://hunch.net/vw/ (visited on 09/30/2013).
Lazarevic, A. and Z. Obradovic (2002). 'Boosting algorithms for parallel and distributed learning'. In: *Distributed and Parallel Databases* 11.2, pp. 203–229.
Liaw, A. and M. Wiener (2002). 'Classification and Regression by randomForest'. In: *R News* 2.3, pp. 18–22. URL: http://CRAN.R-project.org/doc/Rnews/.
Moore, D., G. P. McCabe, and B. Craig (2010). *Introduction to the Practice of Statistics*. W. H. Freeman.
Newman, D. et al. (1998). *UCI Repository of machine learning databases*. URL: http://www.ics.uci.edu/~mlearn/MLRepository.html.
Obst, R. (2013). *Distributed Model-Based Boosting*. R package version 0.1.2. URL: http://CRAN.R-project.org/package=parboost.
Palit, I. (2012). *A Scalable and Parallel Boosting Framework*. Proquest, Umi Dissertation Publishing.
Pavlov, D. Y., A. Gorodilov, and C. A. Brunk (2010). 'BagBoo: a scalable hybrid bagging-the-boosting model'. In: *Proceedings of the 19th ACM international conference on information and knowledge management*.
Riley, J. (2013). *StarCluster*. URL: http://star.mit.edu/cluster/docs/latest/index.html.
Sapp, S., M. J. van der Laan, and J. Canny (2013). *Subsemble: An Ensemble Method for Combining Subset-Specific Algorithm Fits*. Tech. rep.

Schapire, R. E. and Y. Freund (2012). *Boosting*. Foundations and Algorithms. MIT Press.
Scheipl, F. (2011). 'Bayesian Regularization and Model Choice in Structured Additive Regression'. PhD thesis. Munich: Dr. Hut Verlag.
Seeger, M. (2007). 'Low rank updates for the Cholesky decomposition'. In: *University of California at Berkeley, Tech. Rep*.
Tibshirani, R. (1996). 'Regression shrinkage and selection via the lasso'. In: *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288.
Tukey, J. W. (1977). *Exploratory Data Analysis*. Addison-Wesley.
Tyree, S. et al. (2011). 'Parallel boosted regression trees for web search ranking'. In: *Proceedings of the th international conference on World wide web*, pp. 387–396.
Xie, J., V. Rojkova, and S. Pal (2009). 'A Combination of Boosting and Bagging for KDD Cup 2009-Fast Scoring on a Large Database.' In: *Journal of Machine Learning Research* 7, pp. 35–43.
Yu, C. and D. B. Skillicorn (2001). 'Parallelizing boosting and bagging'. In: *Queen's University, Kingston, Canada, Tech. Rep*.
Zou, H. and T. Hastie (2005). 'Regularization and variable selection via the elastic net'. In: *Journal of the Royal Statistical Society. Series B (Methodological)*.