A Study of Accelerated Bayesian Additive Regression Trees

by

Saar Yalov

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2019 by the
Graduate Supervisory Committee:

P. Richard Hahn, Chair
Robert McCulloch
Ming-Hung Kao

ARIZONA STATE UNIVERSITY

May 2019

ABSTRACT

Bayesian Additive Regression Trees (BART) is a non-parametric Bayesian model that often outperforms other popular predictive models in terms of out-of-sample error. This thesis studies a modified version of BART called Accelerated Bayesian Additive Regression Trees (XBART). The study consists of simulation and real data experiments comparing XBART to other leading algorithms, including BART. The results show that XBART maintains BART's predictive power while reducing its computation time. The thesis also describes the development of a Python package implementing XBART.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

Chapter 1

BACKGROUND OF STUDY

1.1 Introduction to Function Estimation

In many applications one would like to predict the value of an unobserved quantity based on some observed measurements. For example, one might like to predict the credit–worthiness of an individual based on that individual's financial records. Mathematically, a prediction problem of this type can be regarded as a function estimation problem, where the function of interest is the expected value of the response variable — probability of defaulting on a loan. We will denote the response variable of interest (loan default) by y_i where i denotes an individual, and we will let x_i denote a vector of attributes, or features, upon which we want to base our prediction. The relationship we want to learn can be defined as:

$$y_i = f(\mathrm{x}_i) + \epsilon_i$$

$$\mathbb{E}(\epsilon_i) = 0 \text{ and } \epsilon_i \text{ is not } \epsilon_i(\mathrm{x}_i)$$

.

Due to the generality of the functional form, there are many ways to estimate f. The success of a method varies greatly by the form of the underlying function f and the distribution of x_i's and ϵ_i.

This thesis assumes familiarity with supervised learning, as covered in Chapter 2.6 of Elements of Statistical Learning (Hastie, Tibshirani, and Friedman 2009), and will use vocabulary from that area without explicit definition. Examples of such

terms include: training data, test data, overfitting, underfitting, in–sample error, generalization error, and cross validation.

In supervised learning, the estimator is a mapping from our training data $\{\mathrm{x}_i, y_i\}$ into a class of functions $\mathcal{F}$. Each $f \in \mathcal{F}$ is a mappings from $\mathcal{X} \to \mathbb{R}$. The estimate is $\hat{f} \in \mathcal{F}$ – which given data x yields prediction $\hat{y_i} = \hat{f}(\mathrm{x}_i)$. To assess the performance of our estimate, we use loss function $L(y_i, \hat{y}_i)$. In particular, we check generalization error which uses unseen data $\{\tilde{\mathrm{x}}_i, \tilde{y}_i\}$ to evaluate $L(\tilde{y_i}, \hat{f}(\tilde{\mathrm{x}}_i))$. In the case where we know the true underlying function f (i.e simulation study), we can evaluate $L(f(\tilde{\mathrm{x}}_i), \hat{f}(\tilde{\mathrm{x}}_i))$.

To reflect the fact that we do not know the true underlying form of f (linear, polynomial, single index, etc.), modern statistics uses non–parametric methods to estimate f flexibly. Multivariate Adaptive Regression Splines (Friedman 1991) and Support Vector Machines (Cortes and Vapnik 1995) achieve flexibility by basis expansion. Neural networks achieve flexibility by introducing non–linearities. This thesis will focus on tree based methods. Tree based methods achieve flexibility by making recursive partitions on $\mathcal{X}$ using its covariates. Predictions are made using summary statistics from the final partitions.

The methods described above are expressive enough to represent complicated functions. This is both a blessing and curse. The blessing is that these methods can learn non–linear functions. The curse is that they can overfit to noise in the data and produce non–generalizable results. Therefore, when assessing the performance of an estimator it is helpful to frame the problem in terms of the Bias Variance tradeoff.

1.2 Bias Variance Tradeoff

In regression problems, our estimate $\hat{f}$ is a mapping from $\mathcal{X} \to \mathbb{R}$. Since $\hat{f}$'s co–domain is $\mathbb{R}$, we can measure its success by applying the Mean Squared Error (MSE) loss function to its prediction $\hat{f}(\mathrm{x})$:

$$\mathrm{MSE}_{Y|x} = \mathbb{E}\left[\left(Y - \hat{f}(\mathrm{x})\right)^2\right]$$

.

This term conveniently decomposes into:

$$\begin{aligned}\mathbb{E}\left[Y - \hat{f}(x)\right]^2 &= \left[f(x) - \mathbb{E}(\hat{f}(x))\right]^2 + \mathrm{Var}(\hat{f}(x)) + \mathrm{Var}(Y|x) \\ &= \mathrm{Bias}(\hat{f}(x))^2 + \mathrm{Var}(\hat{f}(x)) + \sigma^2\end{aligned}$$

.

This decomposition represents the dichotomy of accuracy and stability in prediction. One can get a very accurate but unstable predictions — low bias high variance. Such a model is said to overfit the data. Conversely, one can get a very stable but inaccurate prediction — low variance high bias. Such a model is said to underfit the data. The ideal compromise between bias and variance, by construction, is the (unknown) combination that minimize Mean Squared Error. The goal of Cross Validation is to find the right combination of these two using data (Hastie, Tibshirani, and Friedman 2009).

1.3 Regression Trees

The most well–known of decision tree models is Classification and Regression Trees — CART (Breiman et al. 1984). Breiman generalized the decision tree model to handle both classification and regression problems. The loss function used to find the optimal tree in the regression case, is

$$\text{SSE} = \sum_{i=1}^{N} (y_i - \hat{f}(\text{x}_i))^2$$

. Thus, given data partitions $P_i, P_2, ..., P_M$, the estimate is:

$$\hat{f}(\text{x}) = \sum_{m=1}^{M} \mu_m I(\text{x} \in P_m)$$

. For each m we want:

$$\arg\min_{\hat{\mu_m}} \left\{ \sum_{i=1}^{N} (y_i - \sum_{m=1}^{M} \hat{\mu}_m I(\text{x} \in P_m))^2 \right\} = \frac{1}{N_m} \sum_{\text{x}_i \in P_m} y_i$$

$$N_m = \{\# \text{ of observations in } P_m\}$$

.

To find partitions $P_1, P_2, ..., P_m$, CART recursively and greedly finds binary partition of the data on variable j and cutpoint c such that: $P_{left}(j, c) = X|X_j \leq c$ and $P_{right}(j, c) = X|X_j > c$ which minimize:

$$\arg\min_{j,c} \left\{ \sum_{\text{x}_i \in P_{left}(j,c)} (y_i - \mu_{left})^2 + \sum_{\text{x}_i \in P_{right}(j,c)} (y_i - \mu_{right})^2 \right\}$$

.

Then, given the fully grown tree, CART prunes the tree using cost-complexity pruning. That is, it finds the subtree T_α of the full tree T_{full} that minimizes:

$$C_\alpha(T) = \sum_{m=1}^{|T|} = \sum_{X_i \in P_m} (y_i - \mu_m)^2 + \alpha|T|$$

$$|T| = \{\#\text{ of Terminal Nodes}\}$$

.

Individual decision trees are high in variance — small perturbations in data can lead to big changes to the final model and prediction. Furthermore, due to their discrete fitting process, tree predictions are not smooth functions. In particular, they there are only 2^d possible values for any given tree where $d = \max$ depth. Thanks to the pruning above, d is kept to be fairly low.

1.4 Random Forests

In order to reduce the variance of individual trees, Brieman proposed Random Forest (Breiman 2001). Random Forest achieves this by using Bootstrap Aggregation (Bagging) and, at each node, only allowing a subset of features to be considered at each split.

Algorithm 1 Random Forest (Hastie, Tibshirani, and Friedman 2009)

procedure RANDOM FOREST
  for $b = 1$ to B **do**
    Draw boostrap sample $\boldsymbol{\chi}^*$ of size n
    Fit Decision Tree on $\boldsymbol{\chi}^*$
    Before each split, consider random `mtry` <= p variables
  $\hat{f}_{rf} = \frac{1}{B}\sum_{b=1}^{B} T_b$.

If each tree is identically distributed with some variance σ^2 and with positive pairwise correlation ρ, then:

$$\text{Var}(\hat{f}_{rf}) = \rho\sigma^2 + \sigma^2\frac{(1-\rho)}{B}$$

. Thus, the variance reduces as B increases and as $\rho \to 0$. By only considering `mtry`

out of p variables, we reduce the pairwise correlation of the trees. The assumptions above are not assumed — they are used to motivate the variance reduction given by `mtry` and Bagging. (Hastie, Tibshirani, and Friedman 2009)

By reducing the prediction variance, Random Forest avoids overfitting. In fact, it is common practice when using Random Forests to grow large unpruned trees (high variance trees) in comparison to individual CART trees. The variance reduction provides a protection net to overfitting while simultaneously allowing the trees to learn more intricate interactions in the data.

1.5 Gradient Boosted Trees

Adaptive Boosting (AdaBoost) (Freund and Schapire 1997) is an ensemble technique which iteratively fits weak learners — learners which perform slightly better than random. In Boosting, the weak learner is typically a decision tree. AdaBoost was originally made for classification problems but has since been generalized to Gradient Boosting (Friedman 2001) which works on arbitrary differentiable loss functions. Gradient Boosted Trees, sequentially refits decision trees to the residuals of the previous trees. By using the residual, the model forces the next tree to learn variations of the data not explained by previous trees. This process reduces the overall bias of prediction.

The individual trees are dependent on each other via the residual and are thus not easy to parallelize. However, the trees grown in Gradient Boosting are typically smaller than the ones grown in Random Forests. Furthermore, fast and scalable implementation of gradient boosting such as open source XGBoost (Chen and Guestrin 2016), Microsoft's open source LightGBM (Ke et al. 2017), and Yandex's open source

Algorithm 2 Gradient Boosted Trees (Hastie, Tibshirani, and Friedman 2009)

1: Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$
2: **for** $m = 1$ to M **do**
3: For $i = 1, 2, ..., N$ compute the psuedo residual:

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

4: Fit tree with r_{im} as target get partitions $P_{j,m}, j = 1, 2, .., J_m$
5: For $j = 1, 2, .., J_m$ compute:

$$\gamma_{jm} = argmin_\gamma \sum_{x_i \in P_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

6: Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in P_{jm})$
7: $\hat{f}_{boost}(x) = f_M(x)$.

Catboost (Prokhorenkova et al. 2018) have found ways to make other aspects of the computation parallelizable. Currently, these are the state of the art algorithms used to win many online machine learning competitions and to solve difficult business problems. Their popularity comes from their fast fitting and prediction times, great results on heterogenous data, and their relative transparency in comparison to black–box models like neural networks.

Boosting algorithms have inspired Bayesian Additive Regression Trees and Accelerated Bayesian Additive Regression Trees.

Chapter 2

THE BAYESIAN APPROACH

2.1 Bayesian CART

Bayesian CART (Chipman, George, and McCulloch 1998) presents a Bayesian approach to finding good CART models. It first specifies a prior for the tree structure and a prior on the terminal nodes conditional on the tree structure. Then, given the prior, it defines a Metroplis Hasting algorithm to sample trees from the posterior distribution.

2.1.1 Prior

The CART model is defined by $\Theta = \mu_1, ...\mu_M$ and the tree structure T as well as a residual variance parameter σ^2. We can define the prior compositionally:

$$p(\Theta, T) = p(\Theta|T)p(T)$$

.

The prior for the tree $p(T)$ is defined as random process which generates individual trees. Each tree output is considered a draw from $p(T)$.

The split prior finds the probability of a split. It is defined as $p_{\text{split}}(\eta, T) = \alpha(1 + d_\eta)^{-\beta}$ where d_η is the depth of node η. The values of α and β are modeling parameters which are analogous to the `max depth` modeling parameter seen in most decision tree implementations. These values can be set using cross validation.

Algorithm 3 Tree Prior Generation Process

1: Begin with a tree T with a single terminal node η
2: **procedure** GROW(η, T)
3: Split terminal node η with probability $p_{split}(\eta, T)$.
4: **if** SPLIT **then**
5: Assign rule ρ from $p_{rule}(\rho|\eta, T)$ which creates new tree T with new left and right nodes η_l and η_l.
6: GROW (η_l, t) and GROW (η_r, t)
7: **else**
8: **return**

Variables to split on and cut–points to split at are selected with probability p_{rule}. A common choice is to pick a predictor x_i uniformly at random, then, to pick a cut–point c uniformly at random from the realized values of x_i.

In practice we center and scale the data and use the following zero–centered priors:

$$\mu_j \mid T \overset{\text{iid}}{\sim} N(0, \tau)$$

$$\sigma^2 \sim IG(\nu/2, \nu\lambda/2)$$

.

2.1.2 Metropolis–Hasting Algorithm

The Metropolis–Hasting algorithm is used to generate draws from the tree posterior $T_0, T_1, T_2, ...$

$q(T, T^*)$ is defined by creating a new tree T^* from current tree T by randomly making one of these four local changes:

1) GROW: Randomly pick a terminal node. Grow to children by randomly picking splitting rule from p_{rule} .

Algorithm 4 MCMC

1: Generate candidate tree T^* via $q(T^i, T^*)$
2: Set $T^{i+1} = T^*$ with probability

$$\min\left\{\frac{q(T^*, T^i)}{q(T^i, T^*)}\frac{p(Y|X, T^*)p(T^*)}{p(Y|X, T^i)p(T^i)}, 1\right\}$$

3: Otherwise, set $T^{i+1} = T^i$

2) PRUNE: Randomly pick a parent node, and collapse its children — make it a terminal node.

3) CHANGE: Randomly pick an internal node and randomly change its splitting rule via p_{rule}.

4) SWAP: Randomly pick parent–child pair that are internal nodes and swap their splitting rules. If they have the same rule, swap with both children nodes.

Notice, GROW and PRUNE counter act each other where CHANGE and SWAP are their own counterparts. That is, if the chain is stuck in a local minima, the algorithm allows it to backtrack and explore other options. However, the Markov Chain tends to be sticky once a decent tree is found. Even with this problem, the trees grown from the Bayesian CART procedure outperform the trees grown in a typical CART procedure.

2.2 Bayesian Additive Regression Trees

Bayesian Additive Regression Trees — BART (Chipman, George, McCulloch, et al. 2010) defines an ensemble of Bayesian CART trees that is influenced from Gradient Boosting. In particular, the BART model is a sum of tree model which work by defining a prior for the trees and then drawing trees from posterior defined by a Bayesian Backfitting MCMC algorithm that regress on residual trees. This algorithm

mixes a lot better than Bayesian CART and, in most use cases, leads superior results in comparison to Gradient Boosting and Random Forests. The BART model is defined as:

$$Y = \sum_{j=1}^{m} g(x|T_j, M_j) + \epsilon; \epsilon \sim N(0, \sigma^2)$$

.

Each $g(x|T_j, M_j)$ is a single tree drawn from the Bayesian CART algorithm. T_j is the tree structure and $\mu_{1,j}, , ..., \mu_{b,j} = M_j$'s are the summary estimates for each terminal node.

2.2.1 Prior

The prior of BART is similar to the prior used for Bayesian Trees, but extended to multiple trees. It is used to regularize the trees to produce weak learners. This forces each tree to learn smaller aspects of the function at hand.

The regularization prior is defined as:

$$p((T_1, M_1), (T_2, M_2), ...(T_m, M_m, \sigma) = \Big[\prod_j p(T_j, M_j)\Big] p(\sigma)$$

.

Similarly to the formulation of Bayesian Tree, it is convenient to represent the prior as:

$$\Big[\prod_j p(M_j|T_j)p(T_j)\Big] p(\sigma) \quad \text{where} \quad p(M_j|T_j) = \prod_i p(\mu_{ij}|T_j)$$

.

The priors for each T_j, $p(T_j)$, and $M_j|T_j$, $p(M_j|T_j)$ are defined in the same way as in Bayesian CART.

2.2.2 A Bayesian Backfitting MCMC Algorithm

The Bayesian Backfitting MCMC algorithm defines a process to sample trees from

$$p((T_1, M_1), ..., (T_m, M_m), \sigma | y)$$

. It defines m trees and sequentially makes local changes to individual trees by using the residual given all other trees.

Define $T_{(j)}$ to be all trees in fit other than the T_j'th tree consisting of m-1 trees, define $M_{(j)}$ in the same fashion. To fit on the residual, the draws are defined by:

$$(T_j, M_j) | T_{(j)} M_{(j)}, \sigma, y$$

$$\sigma | T_1, ..., T_m, y$$

.

σ is drawn from the usual inverse gamma conjugate. To get the m draws of trees, we first define the residual in given sweep $k + 1$:

$$R_j^{(k+1)} \equiv \mathrm{y} - \sum_{j'<j} g(\mathbf{X}; T_{j'}, \mu_{j'})^{(k+1)} - \sum_{j'>j} g(\mathbf{X}; T_{j'}, \mu_{j'})^{(k)}$$

.

By doing so, we can more compactly represent the tree posterior as:

$$(T_j, M_j) | R_j, \sigma$$

.

Since M_j uses a conjugate prior, it is convenient to express the model as:

$$T_j | R_j, \sigma$$

$$M_j|T_j, R_j, \sigma$$

. Then, T_j is drawn via the Metropolis–Hasting Searching Algorithm defined for Bayesian CART. M_j is drawn using the typical Gaussian conjugate for each μ_{ij}.

This backfitting algorithm is ergotic and converges in distribution to the true posterior $p(f|y)$. To sample from the posterior, one can use posterior sample mean after burn–in $\hat{f}_1, \hat{f}_2, .., \hat{f}_k$:

$$\frac{1}{K}\sum_{k=1}^{K}\hat{f}_k^*(x)$$

.

The solution is not limited to the mean of the posterior distribution. As in all Bayesian analysis, any function of interest can be applied to the posterior. One can extract the median or credible intervals from the posterior. This is one of the great benefits of this algorithm in contrast with other traditional ensemble methods.

BART dominates Gradient Boosting, Random Forest, Neural Networks and many other machine learning algorithms in terms of performance across simulated and real data experiments. However, BART fits rather slowly and, by modern standards, is infeasible to fit on large data sets. One reason for this, is that in the Bayesian Backfitting MCMC BART makes local changes to each tree in each sweep. For each tree BART keeps the location of all samples in the tree. This means that BART must maintain at least m copies of the data at any given time. This computational burden is what inspired Accelerated Bayesian Additive Regression Trees (He, Yalov, and Hahn 2019).

2.3 Accelerated Bayesian Additive Regression Trees

Accelerated Bayesian Additive Regression Trees (XBART) present modifications to the existing BART algorithm which are more computationally efficient while still maintaining similar performance to BART. In particular they are, a grow–from–root strategy, using adaptive cut–points for likelihood evaluation, presorting the features, and using a sparse proposal distribution for features.

2.3.1 Bayesian Boosting — A Grow–From–Root Backfitting Strategy

Instead of making local changes to existing tree, XBART grows an entirely new tree for each T_j at iteration k. This removes the burden of keeping copies of the data without decreasing the performance of the model. However, this does require modifications to the Metropolis Hasting algorithm. The tree is grown recursively by evaluating the integrated likelihood criterion at all possible cut–points (see subsection 2) for each variable via:

$$\pi(v, c) = \frac{\exp\left(\ell(c, v)\right)\kappa(c)}{\sum_{v'=1}^{V}\sum_{c'=0}^{C}\exp\left(\ell(c', v')\right)\kappa(c')} \tag{2.1}$$

where

$$\begin{aligned}\ell(v, c) = &\frac{1}{2}\left\{\log\left(\frac{\sigma^2}{\sigma^2 + \tau n(\leq, v, c)}\right) + \frac{\tau}{\sigma^2(\sigma^2 + \tau n(\leq, v, c))}s(\leq, v, c)^2\right\} \\ &+ \frac{1}{2}\left\{\log\left(\frac{\sigma^2}{\sigma^2 + \tau n(>, v, c)}\right) + \frac{\tau}{\sigma^2(\sigma^2 + \tau n(>, v, c))}s(>, v, c)^2\right\}\end{aligned}$$

Here $n(\leq, v, c)$ is the number of observations in the current leaf node that have $x_v \leq c$ and $s(\leq, v, c)$ is the sum of the residual $r_l^{(k)}$ of those same observations; $n(>, v, c)$ and $s(>, v, c)$ are defined analogously. Also, $\kappa(c \neq 0) = 1$.

With the Bayesian Boosting strategy it is recommended to grow relatively deeper trees with fewer sweeps. The default choices are $\beta = 1.25$ and 40 MCMC iterations (sweeps). An important regularization parameter is τ is the prior variance of for each $\mu_1, \mu_2, ...\mu_b$ in M. The choice of τ is the apriori guess for the amount of variation of y explained by the data. Ideally, a choice for this value will be set by a domain expert. Otherwise, although not directly inline with the Bayesian approach, one can use Cross Validation to find a proper choice of τ.

2.3.2 Recursively Defined Cut–Points

When evaluating possible split values, the Bayes Rule calculation defined in (1) is time consuming. To expatiate the fitting process, in each possible split, only C possible cut–points are considered. At each step of the recursion, a cut–point c_j is given by $c_j = floor(\frac{n_b - 2}{C})$ where n_b is the number of nodes in the current partition. A default choice for C is $\max\{\sqrt{n}, 100\}$.

2.3.3 Index Pre–Sorting Features

Since the likelihood evaluation depends on partitioned cumulative sums (1):

$$s(\leq, v, c) = \sum_{h \leq c} r_{o_{vh}} \tag{2.2}$$

and

$$s(>, v, c) = \sum_{h=1}^{n} r_{lh} - s(\leq, v, c). \tag{2.3}$$

It is convenient to sort the features so that the cumulative sums can be computed via single sweeps of the data.

To do so, we define a matrix $\mathbf{O}$ with elements o_{vh} denoting the index of h^{th} largest observation of the x_v^{th} variable in the original data matrix. That way, we can sort any column of our data by a particular variable. This will be used to evaluate $s(\leq, v, c)$. Once a variable v and cut–point c are selected, the algorithm partitions $\mathbf{O}$ into two matrices $\mathbf{O}^{\leq}$ and $\mathbf{O}^{>}$. These matrices are populated by evaluating $x_j[x_v < c]$. By doing so, the order preserved for the next step of the recursion.

2.3.4 Sparse Proposal Distribution

Another modification, which is influenced by Random Forests, is to sample randomly and without replacement $m < P$ candidate variables at each split. This contrasts from BART that randomly picks only one variable to split on. The m variables selected are drawn with probability proportional to w. w is given a Dirichlet prior with hyper parameter $\bar{w} = 1$. This value is incremented after a variable has split. The split counts are then updated:

$$\bar{w} \leftarrow \bar{w} - \bar{w}_l^{(k-1)} + \bar{w}_l^{(k)} \tag{2.4}$$

where $\bar{w}_l^{(k)}$ denotes the length–V vector recording the number of splits on each variable in tree l at iteration k. The weight parameter is then sampled as $w \sim \text{Dirichlet}(\bar{w})$. Variables that improve the likelihood will be chosen more frequently than uninformative variables. This has two benefits. The first is that the algorithm is more likely to pick better candidate splits. The second is that the algorithm less frequently computes likelihood for unimportant variables.

2.3.5 Final Estimation

Similarly to BART, we can define an estimator as the mean of all sweeps. It is important to note that given the current formulation, the output of this model is not a true posterior distribution of the trees. However, the sum of all estimates past the burn–in yields good predictions. More formally, the default parameters are $K = 40$ sweeps and $I = 15$ burn–in rounds. The final estimator is given by:

$$\bar{X} = \frac{1}{K - I} \sum_{k>I}^{K} f^{(k)}(X)$$

.

Algorithm 5 Grow From Root

procedure GROW_FROM_ROOT(y, X, C, m, w)▷ Fit a tree using data y and X by recursion.
output A tree T_l and a vector of split counts w_l.
$N \leftarrow$ number of rows of y, x
Sample m variables use weight w as shown in section 2.3.4.
Select C cut–points as shown in section **??**.
Evaluate $C \times m + 1$ candidate cut–points and no-split option with equation (2.1).
Sample one cut–point proportional to equation (2.1).
if sample no-split option **then return**
else
$w_l[j] = w_l[j] + 1$, add count of selected split variable.
Split data to left and right node.
GROW_FROM_ROOT(y_{left},X_{left}, C, m, w)
GROW_FROM_ROOT(y_{right},X_{right}, C, m, w)

Algorithm 6 Accelerated Bayesian Additive Regression Trees (XBART)

procedure ABARTH($y, X, C, m, L, I, K, \alpha, \eta$) ▷ ($\alpha, \eta$ are prior parameter of σ^2)
output Samples of forest
  $V \leftarrow$ number of columns of X
  $N \leftarrow$ number of rows of X
  Initialize $r_l^{(0)} = y/L$.
  for k in 1 to K **do**
    for l in 1 to L **do**
      Calculate residual $r_l^{(k)}$ as shown in section 2.2.2.
      if $k < I$ **then**
        GROW_FROM_ROOT($r_l^{(k)}$,X, C, V, w) ▷ use all variables in burnin iterations
      else
        GROW_FROM_ROOT($r_l^{(k)}$,X, C, m, w)
      $\bar{w} \leftarrow \bar{w} - \bar{w}_l^{(k-1)} + \bar{w}_l^k$ ▷ update $\bar{w}$ with split counts of current tree
      $w \sim$ Dirichlet($\bar{w}$)
      $\sigma^2 \sim$ Inverse-Gamma($N + \alpha, r_l^{(k)t} r_l^{(k)} + \eta$)
  return

Chapter 3

EXPERIMENTS

3.1 Simulation Study

3.1.1 Data Generating Process

To demonstrate XBART's performance we estimate function evaluations with a hold–out set that is a quarter of the training sample size and judge accuracy according to root mean squared error (RMSE). We consider four different challenging functions, f, as defined in Table 1. In all cases, $x_j \overset{\text{iid}}{\sim} \text{N}(0,1)$ for $j = 1, \ldots, d = 30$. The data is generated according to

$$y_i = f(\text{x}_i) + \sigma\epsilon_i$$

for $\epsilon_i \overset{\text{iid}}{\sim} \text{N}(0,1)$. We consider $\sigma = \kappa \text{Var}(f)$ for $\kappa \in \{1, 10\}$.

Table 1. Four true f functions

Name	Function
Linear	$\text{x}^t\gamma;\ \gamma_j = -2 + \frac{4(j-1)}{d-1}$
Single index	$10\sqrt{a} + \sin(5a);\ a = \sum_{j=1}^{10}(x_j - \gamma_j)^2;\ \gamma_j = -1.5 + \frac{j-1}{3}.$
Trig + poly	$5\sin(3x_1) + 2x_2^2 + 3x_3x_4$
Max	$\max(x_1, x_2, x_3)$

3.1.2 Methods

We compare to leading machine learning algorithms: random forests, gradient boosting machines, neural networks, and BART MCMC. All implementations had an `R` interface and were the current fastest implementations to our knowledge: `r`anger (Wright and Ziegler 2015), `x`gboost (Chen and Guestrin 2016), and `k`eras (Chollet 2015),and `d`barts, respectively. For `r`anger and `d`barts we use the software defaults. For `k`eras we used a single strong architecture but varied epochs depending on the noise in the problem. For `x`gboost we consider two specifications, one using the software defaults and another determined by 5–fold cross–validated grid optimization (see Table 2); a reduced grid of parameter values was used at sample sizes $n > 10,000$.

Table 2. Hyperparameter Grid for XGBoost

Parameter name	$N = 10$K	$N > 10$K
`e`ta	$\{0.1, 0.3\}$	$\{0.1, 0.3\}$
`m`ax_depth	$\{4, 8, 12\}$	$\{4, 12\}$
`c`olsample_bytree	$\{0.7, 1\}$	$\{0.7, 1\}$
`m`in_child_weight	$\{1, 10, 15\}$	10
`s`ubsample	0.8	0.8
`g`amma	0.1	0.1

3.1.3 Computation

The simulations were computed on Ubuntu 18.04.1 LTS with Intel i7–8700K Hexa–core 3.7GHz 12MB Cache–64–bit processing , 4.3GHz overclocking speed.

The software used was `R` version 3.4.4 with `xgboost` 0.71.2, `dbarts` version 0.9.1 , `ranger` 0.10.1, and `keras` 2.2.0. The default hyperparameters for XGBoost are `eta` = 0.3, `c`olsample_bytree = 1,`m`in_child_weight = 1, and `max_depth` = 6. Ranger was

fit with `num.trees` = 500 and `mtry` = $5 \approx \sqrt{d}$. BART, with the package `dbarts`, was fit with the defaults of `ntrees` = 200, `alpha` = 0.95, `beta` = 2, with a burn–in of 5,000 samples (`nskip` = 5000) and 2,000 retrained posterior samples (`ndpost` = 2000). The default `dbarts` algorithm uses an evenly spaced grid of 100 cut–point candidates along the observed range of each variable (`numcuts` = 100, `usequants` = FALSE). For `keras` we build a network with two hidden layers (15 nodes each) using ReLU activation function, ℓ_1 regularization at 0.01, and with 50/20 epochs depending on the signal to noise ratio.

3.1.4 The Procedure

The development of the simulation study was a slow and iterative process. The code is composed of two main files, a simulation script and a helper function file. The helper functions file contains functions of the main repeated procedures in the code. The simulation script uses these functions in the simulation. This was done in order to organize the code to make it easy to iterate and debug.

For each algorithm there were two helper functions — a get hyperparameter function and a fit function. The hyperparameter function takes in data information such as number of observations and returns an R list of the parameters. The fit function takes the hyperparameter R list along with the data, it then fits the model and returns RMSE and time of the fit. This information was all stored in R DataFrame. After each round of the simulation, I would save the files with a specific naming convention. At the end of the simulations, the files were read to generate the final aggregations.

XGBoost tuned required an additional function, the tuning function. Initially we experimented with Bayesian Optimization to pick the hyperparameters. The process

was slow and didn't yield great results. We then switched to a simple grid search which performed well and was substantially faster. At the end of each optimization process the hyperparameters were saved in a JSON file for reference.

3.1.5 Results

The performance of the new XBART algorithm was even better than anticipated, showing superior speed and performance relative to all the considered alternatives on essentially every data generating processes. The full results, averaged across five Monte Carlo replications, are reported at the end of this chapter where XB, cv, xgb, NN are results of XBART and XGBoost with and without turning parameter by cross validation, and Neural Networks respectively. Neural networks perform as well as XBART in the low noise settings under the Max and Linear functions. To no surprise, neural networks outperform XBART under the linear function with low noise. Across all data generating processes and sample sizes, XBART was 31% more accurate than the cross–validated XGBoost method and typically faster. The XBART method was slower than the untuned default XGBoost method, but was 3.5 times more accurate. This pattern points to one of the main benefits of the proposed method, which is that it has excellent performance using the same hyperparameter settings across all data generating processes. Importantly, these default hyperparameter settings were decided on the basis of prior elicitation experiments using different true functions than were used in the reported simulations. While XGBoost is quite fast, the tuning processes is left to the user and can increase the total computational burden by orders of magnitude.

Random forests and traditional MCMC BART were prohibitively slow at larger

sample sizes. However, at $n = 10,000$ several notable patterns did emerge. First was that BART and XBART typically gave very similar results, as would be expected. BART performed slightly better in the low noise setting and quite a bit worse in the high noise setting (likely due to inadequate burn–in period). Similarly, random forests do well in higher noise settings, while XGBoost and neural networks performs better in lower noise settings.

3.2 BART Data Sets

To compare the performance on real data we recreated the data experiment used in the original BART paper. We used 40 datasets (Table 4) of varying sizes. For each dataset, 20 random train test splits were made, yielding a total of 800 experiments. In each experiment we fit XBART, BART, Random Forest, XGBoost with default feature, and a tuned XGBoost model with the same tuning parameters choices as in the simulation studies. Then, to fairly evaluate the results across all experiments, we use Relative RMSE (RRMSE) which is defined for experiment i and model $m \in \mathbf{M}$ a is defined as

$$\text{RRMSE}_{i,m} = \frac{\text{RMSE}_{i,m}}{\min\{RMSE_{i,j} | j \in \mathbf{M}\}}$$

To adjust for the smaller datasets, we made slight modifications to the XGBoost and XBART hyper–parameter. We forced $L = \max\{(\log n)^{\log\log n}, 100\}$ to allow more trees at each iteration — we set $L =$ nrounds for xgboost. Secondly, we set nsweeps = 200, max depth = 200, and burnin = 80 to allow for more flexible models.

Following BART, XBART performs the best in terms of RRMSE. Furthermore, the study reaffirms that for XGBoost to be competitive, it needs to be sufficiently

Table 3. Results

	BART	XBART	XGB_TUNED	RF	XGB
RRMSE	1.067	1.128	1.137	1.139	1.170
Average Time	16.930	5.680	6.207	0.385	0.117

tuned. As seen in the simulation studies, for larger datasets, the tuning becomes very computationally expensive.

Table 4. Data Sets

Name	n	Name	n	Name	n	Name	n	Name	n
Abalone	4177	Budget	1729	Diamond	308	Labor	2953	Rate	144
Ais	202	Cane	3775	Edu	1400	Laheart	200	Rice	171
Alcohol	2462	Cardio	375	Enroll	258	Medical	4406	Scenic	113
Amenity	3044	College	694	Fame	1318	Mpg	392	Servo	167
Attend	838	Cps	534	Fat	252	Mumps	1523	Smsa	141
Baseball	263	Cpu	209	Fishery	6806	Mussels	201	Strike	625
Baskball	96	Deer	654	Hatco	100	Ozone	330	Tecator	215
Boston	506	Diabetes	375	Insur	2182	Price	159	Tree	100

3.3 Elo Merchant Category Recommendation – Kaggle

As a way to test XBART's capabilities, I decided to participate in the Elo Kaggle competition. Elo is a Brazilian payment and credit company. Their goal with this competition, is to predict customer loyalty using historical transactional data. The best model is one which has the lowest RMSE on the private dataset — a holdout set which is revealed only after the competition ends.

The training data contained 201917 unique IDs while the holdout set contained 123623. For each unique id, there is a one to many relationship with transactional data. That is, each credit card has many transactions. To create features which can be used in machine learning models, the transactional information is aggregated via

summary statistics. For example, an important feature in this model was the number of months the card was active.

An important nuance in this competition was outliers. Although most of the loyalty scores was symmetrically distributed around 0, there were roughly 2207 values that were around -33. An outlier was defined to be any value less than -15.

To handle this, for each of the two sets of feature I built 3 sub–models. The first sub–model is a naiive model which was built on the full dataset using LGBM. The second sub–model is a LGBM classification model used to predict whether an observation is an outlier or not. The third sub–model is an XBART model that is used on all values that are not outliers. If an observation was sub–model two's top 25,000 predictions, it was given the score from sub–model 1, other wise it was given the score from sub–model 3.

For the first sub–model I used LGBM because XBART did not perform well on the highly unsymmetric data. This model was not particularly helpful or important in the final prediction since it only involved 20% of the observations. Furthermore, the winning models for the competition just used a simple point estimate rather than a model for all the outlying observations. The second sub–model used LGBM as well since it was a classification problem.

For each submodel above I used Backwards Feature selection to pick the best subset of features. To do so I fit an XGBoost baseline model, and recursively dropped the worst features. The set of features with the lowest performance error was picked.

To get a more robust model, I used two sets of features and developed a model, defined by the three sub–models, on each. The features sets are combinations of features taken from publicly available kernels. The final model is an average of the two model predictions. The XBART model has an RMSE of 3.61276 which scored

303 out 4,129 teams which is in the top 8% of models in the competition. This gave me my first bronze medal on Kaggle.

3.4 Simulation Result Tables

Table 5. Simulation Comparison with XGBoost and Keras Neural Networks

		RMSE				Seconds			
		Linear							
n	k	XB	CV	XGB	NN	XB	CV	XGB	NN
10K	1	1.74	2.63	3.23	1.39	20	64	<1	26
10K	10	5.07	8.04	21.25	7.39	16	61	<1	12
50K	1	1.04	1.99	2.56	0.66	180	142	4	28
50K	10	3.16	5.47	16.17	3.62	135	140	4	14
250K	1	0.67	1.50	2.00	0.28	1774	1399	55	40
250K	10	2.03	3.15	11.49	1.89	1228	1473	54	19
		Max							
n	k	XB	CV	XGB	NN	XB	CV	XGB	NN
10K	1	0.39	0.42	0.79	0.40	16	62	<1	30
10K	10	1.94	2.76	7.18	2.98	16	60	<1	15
50K	1	0.25	0.29	0.58	0.20	134	140	4	32
50K	10	1.22	1.85	5.49	1.63	133	139	4	16
250K	1	0.14	0.21	0.41	0.16	1188	1554	60	44
250K	10	0.75	1.05	3.85	0.85	1196	1485	54	22
		Single Index							
n	k	XB	CV	XGB	NN	XB	CV	XGB	NN
10K	1	2.27	2.65	3.65	2.76	17	61	<1	28
10K	10	7.13	10.61	28.68	9.43	16	61	<1	14
50K	1	1.54	1.61	2.81	1.93	153	141	4	31
50K	10	4.51	6.91	21.18	6.42	133	139	4	16
250K	1	1.14	1.18	2.16	1.67	1484	1424	55	41
250K	10	3.06	4.10	14.82	4.72	1214	1547	54	21
		Trig + Poly							
n	k	XB	CV	XGB	NN	XB	CV	XGB	NN
10K	1	1.31	2.08	2.70	3.96	17	61	<1	26
10K	10	4.94	7.16	17.97	8.20	16	61	<1	13
50K	1	0.74	1.29	1.67	3.33	147	141	4	29
50K	10	3.01	4.92	13.30	5.53	132	139	4	14
250K	1	0.45	0.82	1.11	2.56	1324	1474	59	41
250K	10	1.87	3.17	9.37	4.13	1216	1462	49	20

Table 6. $\kappa = 1$ and $n = 10K$

Function	Method	RMSE	Seconds
Linear	XBART	1.74	20
Linear	XGBoost Tuned	2.63	64
Linear	XGBoost Untuned	3.23	<1
Linear	Random Forest	3.56	6
Linear	BART	1.50	117
Linear	Neural Network	1.39	26
Trig + Poly	XBART	1.31	17
Trig + Poly	XGBoost Tuned	2.08	61
Trig + Poly	XGBoost Untuned	2.70	<1
Trig + Poly	Random Forest	3.04	6
Trig + Poly	BART	1.30	115
Trig + Poly	Neural Network	3.96	26
Max	XBART	0.39	16
Max	XGBoost Tuned	0.42	62
Max	XGBoost Untuned	0.79	<1
Max	Random Forest	0.41	6
Max	BART	0.44	114
Max	Neural Network	0.40	30
Single Index	XBART	2.27	17
Single Index	XGBoost Tuned	2.65	61
Single Index	XGBoost Untuned	3.65	<1
Single Index	Random Forest	3.45	6
Single Index	BART	2.03	116
Single Index	Neural Network	2.76	28

Table 7. $\kappa = 10$ and $n = 10K$

Function	Method	RMSE	Seconds
Linear	XBART	5.07	16
Linear	XGBoost Tuned	8.04	61
Linear	XGBoost Untuned	21.25	<1
Linear	Random Forest	6.52	6
Linear	BART	6.64	111
Linear	Neural Network	7.39	12
Trig + Poly	XBART	4.94	16
Trig + Poly	XGBoost Tuned	7.16	61
Trig + Poly	XGBoost Untuned	17.97	<1
Trig + Poly	Random Forest	6.34	7
Trig + Poly	BART	6.15	110
Trig + Poly	Neural Network	8.20	13
Max	XBART	1.94	16
Max	XGBoost Tuned	2.76	60
Max	XGBoost Untuned	7.18	<1
Max	Random Forest	2.30	6
Max	BART	2.46	111
Max	Neural Network	2.98	15
Single Index	XBART	7.13	16
Single Index	XGBoost Tuned	10.61	61
Single Index	XGBoost Untuned	28.68	<1
Single Index	Random Forest	8.99	6
Single Index	BART	8.69	111
Single Index	Neural Network	9.43	14

Chapter 4

XBART PYTHON PACKAGE

4.1 Creating Standalone C++ XBART

The original implementation of the XBART package was written using C++ with RCPP and RCPPArmadillo dependencies. In order to develop a python package, it was important remove all of the RCPP dependencies so that the source code is standalone C++ library. By doing so, users do not need to have R or the RCPP library installed to run the code. The main difficulties in this process were changing to a C++ random number generator, handling the input output stage, changing RCPP numeric matrix and Armadillo matrices to C++ standard library vectors or double arrays. A particularly important change was to create a standalone MCMC loop function that is independent of RCPP.

4.2 Simplified Wrapper and Interface Generator – SWIG

After removing the RCPP dependencies, I used SWIG to bridge between the C++ code base and python. SWIG (Beazley 1996) is an interface compiler that uses C/C++ header files along with interface files to wrap C/C++ code into various scripting languages. Here is a simple example to showcase how to use SWIG to wrap C++ code to python:

```
1 /* File: example.h */
```

```
double identity(double x);
```

Listing 4.1. Header File

```
/* File: example.cpp */
#include "example.h"

double identity(double x) {
  return x;
}
```

Listing 4.2. .cpp File

```
/* File: example.i */
%module example

%{
#define SWIG_FILE_WITH_INIT
#include "example.h" // Which Header files to look at
%}

double identity(double x); // Function to wrap into python
```

Listing 4.3. Swig Interface File

One can compile the code manually, but in python it is much easier to write a setup.py file which complies and takes care of a lot of the headaches.

```
# setup.py File
from distutils.core import setup, Extension

example_module = Extension('_example',# .so will be created during
    compilation
            # Specify all files to look at:
```

```
                            sources=['example_wrap.cxx', 'example.cpp'],
                            )
setup (name = 'example',
       version = '0.1',
       author      = "Saar Yalov",
       description = """SWIG example""",
       ext_modules = [example_module],
       py_modules = ["example"],
       )
```

Listing 4.4. setup.py

Then from the command line run:

```
swig -c++ -python -py3 example.i  ### Create wrappers using SWIG
python setup.py setup.py build_ext --inplace ### Create python package
```

Listing 4.5. Command Line

The previous script created the wrappers and a python package called example. We can now import it and use it in python in the following way:

```
>>> import example
>>> example.identity(3.0)
3.0
>>> example.identity(11)
11.0
```

Listing 4.6. show.py

Furthermore, SWIG provides a numpy interface which makes it easy to pass in and retrieve numpy arrays from python to cpp. The numpy data gets converted to a row major C++ double arrays. The C++ double arrays are then converted to

column major, since the algorithm works on column major arrays (RCPP and RCPP Armadillo are both column major double arrays under the hood).

4.3 Python Package

To handle the I/O between python and C++, I created a C++ class which handles all of the necessary conversion. The class keeps track of important information throughout fits and has the functionality to convert the numpy arrays inputs to fit into the standalone loop function. For example, one of the private member of this class is a matrix of trees. The fit function takes in a reference to the trees and populates them during the fitting process. By keeping these trees locally, I can call a separate predict function which traverses the trees in order to make prediction.

The Python XBART API is simple and is heavily influenced from Scikit Learn (Pedregosa et al. 2011) and XGBoost (Chen and Guestrin 2016). There are three main components — object instantiation, model fit, and model predict.

The object instancation takes in a dictionary of hyperparameters as an optional argument and returns an instance of class XBART. The fit function takes in a numpy array of data and response as well as integer specifying the number of categorical features. The predict function takes in a numpy array of data and returns an $n \times$ nsweep numpy array of predictions. Here is a use case example:

```
from xbart import XBART
import pandas as pd
# Read Data
train = pd.read_csv("train.csv")
train_target = pd.read_csv("train_target.csv")
test = pd.read_csv("test.csv")
```

```
test_target = pd.read_csv("test_target.csv")

# Build XBART model
xbart = XBART(num_trees = 100,num_sweeps =50,tau = 0.01) # Create Object
xbart.fit(train.values,train_target.values,p_cat = 5)
yhat_test = xbart.predict(test.values)

# Build XBART model using fit_predict
xbart_2 = XBART(num_trees = 100,num_sweeps =50,tau = 0.01) # Create
    Object
yhat_train=xbart_2.fit_predict(train.values,train_target.values,p_cat =
    5)
yhat_test = xbart_2.predict(test.values)
```

Chapter 5

DISCUSSION

The simulation study and real data experiments show that XBART outperforms leading machine learning models excluding BART in terms of holdout error. Furthermore, it is clear that XBART is faster than BART while still maintaining comparable holdout error performance. Although the benefits of XBART are clear, the reason for its predictive performance is not fully understood. This leads to many interesting questions:

1) Is the reason for XBART's superior performance rooted in BART's likelihood criteria? If this were the case, then fitting traditional boosting models with the BART spiting criteria would perform similarly to BART and XBART.

2) Is the reason for XBART's superior performance due to sampling rather than maximizing? If this were the case, then one can randomly sample from likelihood instead of maximizing in a boosting algorithm.

3) Is the reason for XBART's superior only because it builds many additive tree models and averages between? If this were the case, then bagging boosted trees will yield similar results to XBART.

There is still more work to be done to the XBART code base. In particular, the extension of XBART to the multinomial case is based on the BART multinomial extension (Murray 2017). Also, to better compete with the leading Gradient Boosting implementations, we need to add multi-threaded and GPU capabilities to XBART. Furthermore, we need to develop a predict function to the R version, improve some

issues with the categorical and continuous cut-point evaluation, and make both R and python packages more user friendly.

BIBLIOGRAPHY

Beazley, David M. 1996. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++." In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4,* 15–15. TCLTK'96. Monterey, California: USENIX Association. http://dl.acm.org/citation.cfm?id=1267498.1267513.

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

Breiman, Leo, Jerome Friedman, RA Olshen, and Charles J Stone. 1984. *Classification and regression trees.* Chapman / Hall/CRC.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A scalable tree boosting system." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 785–794. ACM.

Chipman, Hugh A, Edward I George, and Robert E McCulloch. 1998. "Bayesian CART model search." *Journal of the American Statistical Association* 93 (443): 935–948.

Chipman, Hugh A, Edward I George, Robert E McCulloch, et al. 2010. "BART: Bayesian additive regression trees." *The Annals of Applied Statistics* 4 (1): 266–298.

Chollet, François. 2015. *keras.*

Cortes, Corinna, and Vladimir Vapnik. 1995. "Support-Vector Networks." *Machine Learning* 20, no. 3 (September): 273–297. doi:10.1023/A:1022627411411.

Freund, Yoav, and Robert E. Schapire. 1997. *A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting.*

Friedman, Jerome H. 1991. "Multivariate Adaptive Regression Splines." *The Annals of Statistics* 19, no. 1 (March): 1–67. doi:10.1214/aos/1176347963.

———. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." *The Annals of Statistics* 29 (5): 1189–1232. http://www.jstor.org/stable/2699986.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning.* Springer.

He, Jingyu, Saar Yalov, and P. R. Hahn. 2019. *Accelerated Bayesian Additive Regression Trees.* https://arxiv.org/abs/1810.02215.

Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. "LightGBM: A highly efficient gradient boosting decision tree." In *Advances in Neural Information Processing Systems,* 3146–3154.

Murray, Jared S. 2017. "Log-linear Bayesian additive regression trees for categorical and count responses." *arXiv preprint arXiv:1701.01503.*

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research* 12:2825–2830.

Prokhorenkova, Liudmila, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. "CatBoost: unbiased boosting with categorical features." In *Advances in Neural Information Processing Systems 31,* edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, 6638–6648. Curran Associates, Inc. http://papers.nips.cc/paper/7898-catboost-unbiased-boosting-with-categorical-features.pdf.

Wright, Marvin N, and Andreas Ziegler. 2015. "ranger: A fast implementation of random forests for high dimensional data in C++ and R." *arXiv preprint arXiv:1508.04409.*