UNIVERSITY OF BERGEN

Faculty of Mathematics and Natural Sciences

Master's Thesis

Statistics

Stochastic Automatic Gradient Tree Boosting

Author:

Eirik Lund Rikstad

Supervisors:

Prof. Hans J. Skaug

Berent Å. S. Lunde

November 2021

Abstract

In this thesis we introduce stochastic sampling to the automatic machine learning framework called Automatic Gradient Tree Boosting (AGTBoost) (Lunde & Kleppe, 2020). This framework is based on a method for automating Gradient Tree Boosting through the introduction of an information criterion for reduction in generalization error (Lunde, Kleppe, & Skaug, 2020). We show that by uniformly drawing a sample of the training observations at each boosting step with a constant sample rate, we can improve the performance of AGTBoost with the cost of making it less automatic. The problem here is that the sampling rates need to be manually selected, and therefore we also introduce the concept of dynamic sampling in the gradient boosting setting, where the sampling rate is selected at each boosting step dependent on a pre-defined sampling scheme. We show that this decrease the variance in performance over different datasets, reducing the need for manual tuning, while still producing strong results relative with the non-dynamic sampling.

Contents

1 Introduction **4**

2 Supervised Learning **9**

2.1 Learning Model and Model selection 11

2.2 Optimism 13

2.3 Estimating the optimism 15

3 Tree Based Methods **19**

3.1 Classification and Regression Trees 23

3.1.1 Regression Trees 23

3.1.2 Classification Trees 24

3.2 Regularization 25

3.3 Ups and downs with trees 26

4 Gradient Boosting **29**

4.1 Ensemble Methods 29

4.2 Boosting 30

4.3 Gradient Descent 34

4.4 Gradient Tree Boosting 36

4.5 Extreme Gradient Boosting 38

4.6 Regularization in Gradient Tree Boosting 41

4.6.1 Learning Rate 41

4.6.2 Ensemble complexity 42

5 Automatic Gradient Tree Boosting **44**

5.1 Global Subset 46

5.2 Information Criterion 48

6 Stochastic Automatic Gradient Tree Boosting **53**
6.1 Sampling in Boosting 53
6.2 Creating the algorithm 54
6.3 Sampling induced variance 57
6.4 Dynamic Sampling 60
6.5 Sampling schemes 61
6.5.1 Random 61
6.5.2 Half 62
6.5.3 Pulse 62
6.5.4 Switch 62
6.6 Implementation 63

7 Experimental design **67**
7.1 Real Data 67
7.2 Simulated Data 68

8 Experimental results **72**
8.1 One rate subsampling 72
8.2 Sampling schemes 75

9 Discussion **85**

References **88**

1 Introduction

The exploitation of big data, with hype terms like machine learning and artificial intelligence has been viral for many years, and while this is typically described as a field where one enables computers to learn without explicit programming, any machine learning practitioner can tell that the development of effective and accurate machine learning pipelines is a tedious endeavor, and more or less requires expert knowledge. Further, with more experience one can always take things one step further, and even though one can aim at perfect, one always has to settle for good enough.

Analyzing a new data set, one both has to select the right algorithm or algorithms and optimize this by tuning the given parameters, and hopefully find some local optimum where the algorithm performs at an acceptable level. Even with expert knowledge, one can not know which algorithm is favorable for a given task, and by all means one has to carefully tune the one selected and find a decent parameter setting, which in most cases even after tuning only will be sub-optimal. This is a time demanding process, and the need for special skills and manual programming different from one algorithm to another creates a tall entry barrier for those who want to utilize the range of possibilities of machine learning. In particular this is true for deep learning, arguably the most powerful and versatile method, where one must chose the right network architecture, parameters, training procedure and regularization methods, but to a large degree this is also the case with conventional machine learning methods.

With automated machine learning, one tries to overcome these problems and create algorithms or entire pipelines, including data cleaning and preprocessing, algorithm selection and algorithm tuning, that are more or less auto-

matic, hopefully providing better models than those created and optimized by hand.

Thornton et al. (2013) created one of the first and most prominent fully automated machine learning pipeline with AUTO-WEKA based on the WEKA framework Hall (2009), where the both algorithm and parameters settings are selected automatically. After that, Feurer et. al (2015) created auto-sklearn, a similar method based on the scikit-learn framework (Pedregosa et al., 2011) for python. In later years many big tech companies have also joined the race with giants like Microsoft, Amazon and Google providing their own cloud based tools for automatic machine learning where parts of the work-flow is automated, making machine learning and in particular neural networks more accessible to more or less anyone. Common for these frameworks are that they to some extend automates both the algorithm selection and parameter tuning. For the individual algorithm, frameworks exists where the automation part extends to parameter tuning only, or where the need for preset parameters are eliminated so that the algorithm itself becomes automatic.

One powerful machine learning algorithm is Gradient Boosting, independently introduced by Friedman (1999) and Mason et al. (1999). The technique was motivated as being a gradient descent method in function space, capable of fitting generic non-parametric predictive models. Gradient boosting has been particularly successful when applied to trees, in which case it fits additive tree models. Friedman devised a special enhancement for this case, known as MART (Multiple Additive Regression Trees), or GBRT (Gradient Boosted Regression Trees). More recently, new implementations of Gradient Tree Boosting has come to stage and quickly gained popularity, such as XG-Boost (Chen & Guestrin, 2016), LightGBM (Ke et al., 2017) and catboost (Prokhorenkova, Gusev, Vorobev, Dorogush, & Gulin, 2017). These has em-

pirically proven itself to be highly effective for a vast array of classification and regression problems, and has become vastly popular, in particular for machine learning competitions. However, with large flexibility of the underlying algorithm, these implementations comes with a large range of parameters available for tuning, and requires expert knowledge to be well optimized.

Stefan Coors et al. (2018) created a automated version of XGBoost, namely AutoXGBoost, based also on Bayesian optimization, which performed decent against other automated machine learning architectures, without ever reaching top shelf.

All of the methods H2O, AUTO-Weka, auto-sklearn and AutoXGboost exploits Bayesian optimization, a procedure where different combinations of algorithms and parameters setting are sequentially evaluated in a guided manner such that it finds a minimum in relatively few iterations (Snoek, Larochelle, & Adams, 2012). Compared to a manual random or gridded search, Bayesian optimization may find better minimums in substantially shorter time, but one cannot avoid the need to train and evaluate a range of different models.

Lunde et al. (2020) took another approach to automating gradient tree boosting, by introducing an information criterion for the reduction in generalization loss caused by splitting a node in a tree in the boosting procedure, that serves as a substitute for the most vital hyperparameters. This makes it optimizable in one run without manual tuning which compared to the aforementioned methods means that one can drastically reduce the time spent in optimizing the model. However, one of the drawbacks with AGTBoost is that some important features present in the previous named GTB implementations such as the possibility of row and column subsampling, L1 and L2 regularization are not yet implemented (Lunde & Kleppe, 2020).

In this thesis we will try and improve on the current implementation of

AGTBoost by developing a method for automatic stochastic row subsampling in the AGTBoost setting and implement and test this. We will first review the theory needed to understand how and why AGTBoost works as well as it does and how we can combine this with the possibility of row subsampling, before we test our implementation against the current implementation of AGTBoost as well as XGBoost and the automatic implementation of XGBoost, AutoXG-Boost.

2 Supervised Learning

In all statistical or machine learning, the goal is to learn some pattern from a data set that also generalize to the full population of which this data set is a sample. In general, the underlying distribution of the population is unknown, and we want to extract information about this distribution from the data set we have at hand, so that we can infer something about the distribution or new data points from the same population, that thus share the same distribution. Formally, we can say that given a data set $D = (\mathbf{X}, \mathbf{Y})$, where $\mathbf{X}$ is an $R^{n \cdot m}$-matrix containing the m explanatory variables of the n data points, and $\mathbf{Y}$ is an $R^{n \cdot 1}$-vector containing the response values from the same data points, consisting of n data points (x_i, y_i), where x_i is an m-vector, and y_i is a single value, we assume that there exist some inherent relationship between $\mathbf{X}$ and $\mathbf{Y}$, and more specifically between each x_i and y_i,

$$y_i = f(x_i) + \epsilon \tag{1}$$

that is true for the entire population of which D is a sample. The matrix $\mathbf{X}$ is also referred to the design matrix, and is a set of covariates, that are also called predictors or independent variables, where the vector $\mathbf{Y}$ on the other hand are referred to as the dependent variable. The term ϵ represents a random error independent of $\mathbf{X}$, with mean 0, and is an aggregation term for everything affecting $\mathbf{Y}$ that is not encapsulated in $\mathbf{X}$. The function f represents the systematic part of the relationship between y_i and x_i, and is the pattern we seek to recognize. Our goal is to find f or an approximation of f, $\hat{f}$, that fulfills the purpose of inferring information about the distribution of each (x_i, y_i) in a best possible way. What information wee seek and what best

possible means and how this is measured is dependent on the specific problem at hand, the user and the area of usage.

In supervised learning we are, as the name suggest, in some sense supervising the learning process, by having access to the true response values $\mathbf{Y}$ of D. We use this to form $\hat{f}$ in a way such that the fitted values $\hat{\mathbf{Y}} = \hat{f}(\mathbf{X})$ resembles $\mathbf{Y}$. In general, we reduce the task of finding f to finding the $\hat{f}$ that minimizes the expected error between the fitted values $\hat{\mathbf{Y}}$ and the true response values, $\mathbf{Y}$, for a chosen error measuring function known as a *loss function.* This is a function that maps any values for one or more variables into a single measure of loss or error. When having estimated $\hat{\mathbf{Y}}$, we feed these and $\mathbf{Y}$ into the chosen loss function, which outputs a single comparable loss value.

So if we let $x_i \in R^m$ be an unseen data point with m features, and $y_i \in R$ be the corresponding response value, the goal of supervised learning is to find a function $\hat{f}$ that minimizes the expected error for the estimated value $\hat{y}_i$ given the true response value

$$\hat{f} = \arg\min_f E_{x_i,y_i}[l(y_i, f(x_i))], \tag{2}$$

measured by some loss function $l(.,.)$ over the joint distribution of all (x, y). The true relation f is of course typically unknown, as this is what we seek to approximate, so in practice (2) is reduced to finding the empirical error over the training set, the set of data used for training,

$$\hat{f} = \arg\min_f \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_i)). \tag{3}$$

The goal however, is still to find a function that generalizes to the full data population, specifically unseen data, and that minimize the error over

that data. When on contrary using the empirical error over the same data for training and evaluation, we expect that the error over unseen data, the generalization error, will be higher than the error over the seen data, the training error. Due to the training loss being an optimistic estimate of the generalization error, this gap is called *optimism*, and because of the optimism, minimizing the training loss is not equivalent with optimizing the model for generalization. Further, the function $\hat{f}$ discussed in this section is normally an optimized function from a pre-selected model class and learning algorithm with a range of inherent constraints and limitation, so having optimized $\hat{f}$, how do one know that our model is good generally speaking?

2.1 Learning Model and Model selection

To solve a learning problem, we want to find the solution to Equation (3). In theory, f can take any functional form, and to search trough the entire function space is both conceptually and computationally infeasible. This means that in order find a solution, we need to restrict the set of possible functions to a subset of the full hypothesis space. The restriction is done by defining of which model class f belongs, meaning that we assume some general form or structure of f. Now, having restricted f to a predefined model class, the learning problem is effectively turned into a feasible optimization problem, where we now in general minimize the empirical error over the training set given the predefined model class. We must also define a learning algorithm suited for solving this optimization problem. This learning algorithm takes as input the dataset D and outputs the fitted model $\hat{f}$, and is the machine aspect of the machine learning problem. If we let $f = f(\cdot;\theta)$ where the function f of a predefined model class are specified by a finite set of parameters θ that we

optimize, what we do specifically is finding

$$\hat{\theta} = \arg\min_{\theta} l(\mathbf{Y}, f(\mathbf{X}; \theta)). \tag{4}$$

But optimizing one algorithm is rarely sufficient to achieve a satisfying model, or in any case knowing to have achieved a satisfying model, and as we know that this optimization is done over training data, even that one algorithm is rarely optimized for generalization purposes which is the real target. There are also no one model class that performs best or even sufficient for all datasets, so one wants to optimize a range of different models and compare the lot to select the best one. A standard way of going about this issue is to hold out a subset of the data available as a validation set, and evaluate the models on this validation data which is unseen at training time. As this can be largely impacted by the way the validation set is sampled, it is common to do *cross validation*, where validation and training sets are sampled from the data multiple times, each time optimizing the models on the training data and evaluating on validation data. The evaluation are then averaged over the multiple different optimizations before the best algorithm for the current data are selected and trained on the full data set.

Another obvious way to find the best model in regards to generalization is to estimate the generalization error directly by estimating the optimism and adding in to the training error. Several methods exists for this purpose, of which two of the most known are Akaike Information Criterion (Akaike, 1998) and Schwarz Information Criterion or Bayesian Information Criterion (Schwarz, 1978). In the next section we will discuss these methods after an elaboration on the term optimism.

2.2 Optimism

Still considering a the design matrix $\mathbf{X}$ and the corresponding response vector $\mathbf{Y}$, and letting $f(x;\theta)$ be the function for modeling $\mathbf{Y}$ given $\mathbf{X}$ with a set of parameters θ, with $\hat{\theta} = \hat{\theta}(D_n)$ being the parameters learned from the training data D_n, we draw a new data point (x^0, y^0) with the same distribution as each $(x_i, y_i) \in D_n$. Then we have

$$Err_D = E_{x^0,y^0}[l(y^0, f(x^0;\hat{\theta}))] \tag{5}$$

is the generalization error, the expected loss value for the true response y^0 and the predicted value from the fitted model, $f(x^0;\hat{\theta})$ for the given loss function. In accordance with Lunde et al. (2020) we use

$$Err = E_{\hat{\theta}} E_{x^0,y^0}[l(y^0, f(x^0;\hat{\theta}))] \tag{6}$$

As a measure of this generalization error. Fitting the function $f(x_i;\theta)$ to the training data means we are adapting θ to achieve the best possible fit for our model over that exact data as in Equation (3). This means that the training loss,

$$\overline{err} = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f(x_i;\hat{\theta})), \tag{7}$$

the averaged loss over this same data used for training, will be overly optimistic as an estimate of the generalization error, and thus the name optimism. We define here this optimism as

$$op = Err - \overline{err}. \tag{8}$$

Now, what is interesting for us is the expected optimism over the training set which we denote

$$C = E_y(op) \tag{9}$$

The definition of the optimism gives us the relation

$$Err = \overline{err} + C \tag{10}$$

So if one can estimate the optimism, one can use it to evaluate a model directly through the training loss by adding the optimism to get an estimate of the generalization error. This is also the main contribution of Lunde et al. (2020) which forms the basis for Automatic Gradient Tree Boosting, and will be elaborated in Section 2.3 and Section 5.2. By this approach, the supervised learning problem is changed from minimizing the empirical loss over the training data to minimizing the sum of the training loss and the estimated optimism.

One can show that the expected optimism over the training set in general can be expressed by two times the covariance between y_i and $\hat{y}_i$ averaged over the training set,

$$C = \frac{2}{n}\sum_{i=1}^{n} Cov(\hat{y}_i, y_i) \tag{11}$$

(Hastie, Tibshirani, & Friedman, 2009, p. 229). This implies that the more each y_i, the true response value of the training data, impact the predicted values $\hat{y}_i$, the larger the optimism, as $cov(y_i, \hat{y}_i)$ will be larger the more y_i affects $\hat{y}_i$. In the next subsection we will take a look at how this optimism can be estimated.

2.3 Estimating the optimism

As mentioned previously, one can deal with the optimism of the training error by estimating the generalization error as a sum of the training error and some estimate of the optimism. One usual way of doing this is to estimate the optimism by some complexity term indicating how complex the function f is. The idea behind this is that the more complex a model is, the more flexible it will become and the stronger it will adapt to the training data. We know from Equation (11) that the optimism is driven by how much the training data impacts the predicted values, and therefore also by the complexity of f. It is not obvious however how to define or measure the complexity and flexibility of a model.

In terms of learning models we separate between parametric models, where we assume the functional form of f and the parameter set is fixed, and non-parametric functions where we do not make such assumptions, and where the parameter set can grow with the size and properties of the training data, and thus the models are free to learn in principle any functional form from the training data. When dealing with parametric models, the straight forward way to measure complexity is by counting the number of parameters.

When $\hat{y}_i$ is obtained by a linear fit with d coefficients from d inputs or basis functions, the right hand side of Equation (11) simplifies (Hastie et al., 2009, p 229), for instance for the model of type $Y = f(X) + \epsilon$, in which case

$$\sum_{i=1}^{N} Cov(\hat{y}_i, y_i) = d\sigma_\epsilon^2. \tag{12}$$

Now, combining Equation (12) with (11) and (10) we get an estimate of the generalization error

$$\hat{Err} = \overline{err} + 2 \cdot \frac{d}{N}\sigma_\epsilon^2 \tag{13}$$

that increase with increase in inputs or basis functions and decrease with increasing number of training observations. The AIC which is similar to this, but formally operates with the log-likelihood of the training data as opposed to the training loss $\overline{err}$, exploits the relationship

$$-2 \cdot E[\log Pr_{\hat{\eta}}(Y)] \approx -\frac{2}{N} \cdot E(\sum_{i=1}^{N} \log Pr_{\hat{\eta}}(y_i)) + 2 \cdot \frac{d}{N} \tag{14}$$

which holds asymptotically as $N \rightarrow \infty$ (Hastie et al., 2009, p 230). Here, $logPr_\eta(Y)$ is the log-likelihood of Y under the distribution η and $\hat{\eta}$ is the maximum likelihood estimate of η. The right hand term of this is the AIC, and when using this to compare models trained on the same data, the N term is simply a scaling term, so one chose the model where

$$AIC = -2 \cdot E(\sum_{i=1}^{N} \log Pr_{\hat{\eta}}(y_i)) + 2 \cdot 2 \tag{15}$$

is minimized. When comparing models of the same model class with different parameter tuning and noise variance $\hat{\sigma_\epsilon}$, Hastie et al. (2009, p 231) defines

$$AIC(a) = \overline{err}(\theta) + 2 \cdot \frac{d(\theta)}{N}\hat{\sigma_\epsilon} \tag{16}$$

where θ is the tuning of the parameter set. We select the model $f(\cdot;\hat{\theta})$ with the tuning $\hat{\theta}$ that minimize the AIC.

However, Equation (12) only holds for linear models where the parameters are not chosen adaptively, meaning that AIC can not be utilized for non-parametric model classes. Here, one rather has to indirectly estimate the effective number of parameters. In the case of AGTBoost, we will see how this is done in Section 5.2, but before that we need to understand the base learner, the tree part, of AGTBoost.

3 Tree Based Methods

We will in this section describe tree methods, which is the base learner class of gradient tree boosting. In particular we will discuss the classification and regression trees, CART, as this simple three method variant is used as the base learner of XGBoost and is similar to the base learner of AGTBoost. Tree algorithms work by segmenting the predictor space, that is the set of possible values for x_i, in non-overlapping regions $R_1, R_2, ...R_j$, and then making predictions or inference based on the distribution of the data in each region. An example of a decision tree segmentation can be seen in Figure 3.1.

The set of splitting rules that defines the segmentation can be visualized as a tree, as can be seen in Figure 3.2. At the very top, we have the so called *root node*. All data are passed to this node and evaluated in regards to its attributes, and a splitting rule is decided and split made. Each outcome of the split results in a branch springing out of the node. Each of these branches then leads to another node, which can be either a new *decision node* or a *terminal node* or *leaf node*. A decision node is simply a node where a split decision is made, thus a decision node also has two or more *child nodes*, of which the decision node is the *parent node*. Say the node is a decision node, then the data in that leaf, the observations in the part of the predictor space corresponding with the branch leading to that node, are yet again split based on another rule. There will then be new branches based on that last splitting rule, and so on until every branch leads to a leaf node. At the leaf or terminal node, the data in the region which possesses all the attributes corresponding to the decision rules forming the branches that lead up to that leaf node are assigned to that region, and prediction or inference are made based on the distribution of all the data in that region. Predictions springing from a leaf

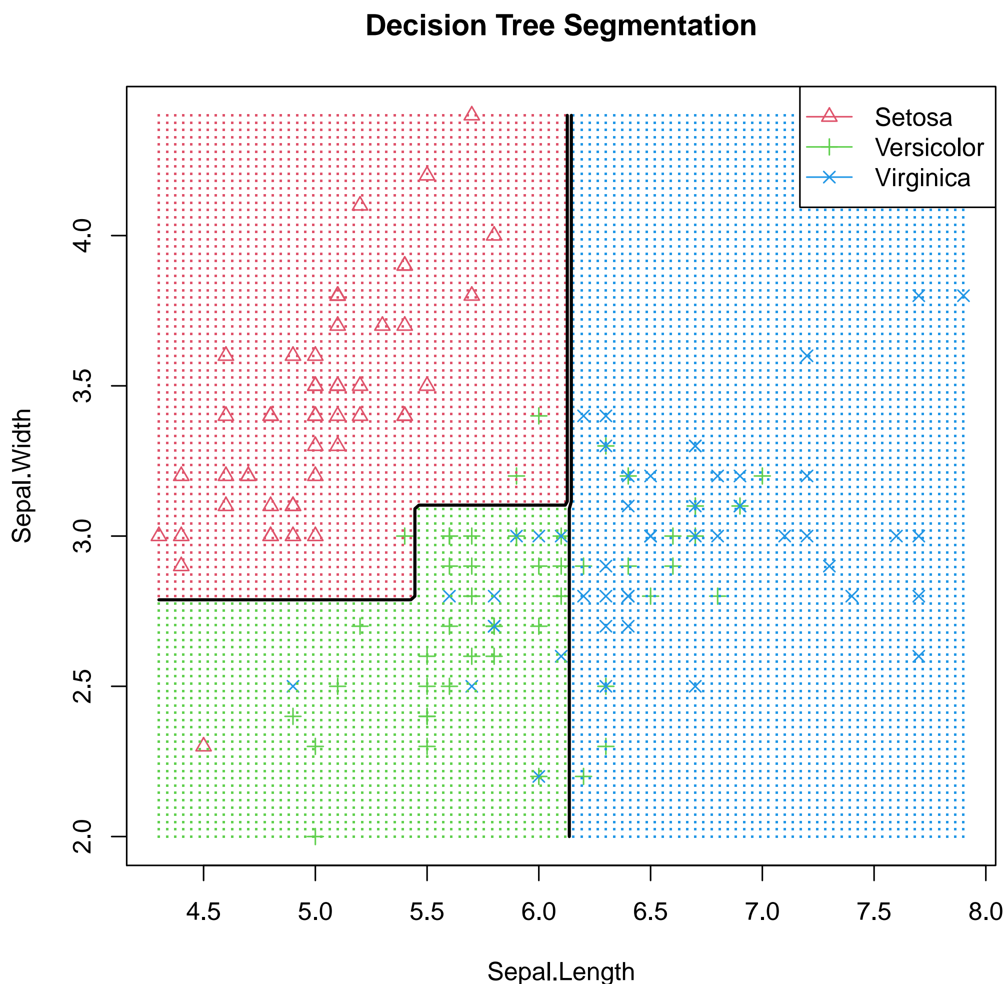

Figure 3.1: Segmentation of predictor space produced by a decision tree trained on the Iris dataset, with Sepal Length and Sepal Width as explanatory variables and Species as response variable. The coloring of each region indicates what prediction is made on data points with the corresponding length and width values, and each points indicates an observation in the training data. The decision tree is made with the R-Package rpart() (Terry Therneau, 2019)

node is typically simple such as a constant, a single class or probability. For regression, the constant prediction is often the mean of the data contained in that region, and for classification the prediction can typically be the majority class or in case of binary classification the proportions of the classes.

Since considering every partitioning of the predictor space is infeasible, in fact learning a tree model is NP-Complete (Hyafil & Rivest, 1976), the splitting is in practice done sequentially in a greedy manner, the same way as the tree is built up. Starting in a root node where the full dataset is evaluated, we make the best possible split in that point. Considering all predictor variables and all possible values of each variable, we find the point that optimize the target function from the prediction made given that split. In each child node we do the same, but now only considering the data passed to that specific node. The splitting is continued recursively as long as it is still possible to further improve the target value, and no stopping criteria is reached. If the target function is a loss function over the training instances it will, if we assume no identical x_i's, always be possible to improve this until all leaves consist a single observation and all predictions are identical with the response values. Since the decision tree is so flexible, being able to perfectly fit a training set, it is also prone to overfitting, and thus one may specify some stopping criteria such as a maximum number of leaf nodes or maximum depth of a tree or a minimum number of observations in a leaf node. Other regularization methods are also possible. After the splitting, we have what we call the *decision tree*. Based on this tree and more specifically the data partitioning rules, we make a prediction for all observations that falls in the same leaf or subset, based on the distribution of the training data in that subset.

Three methods can be utilized for both regression and classification purposes.

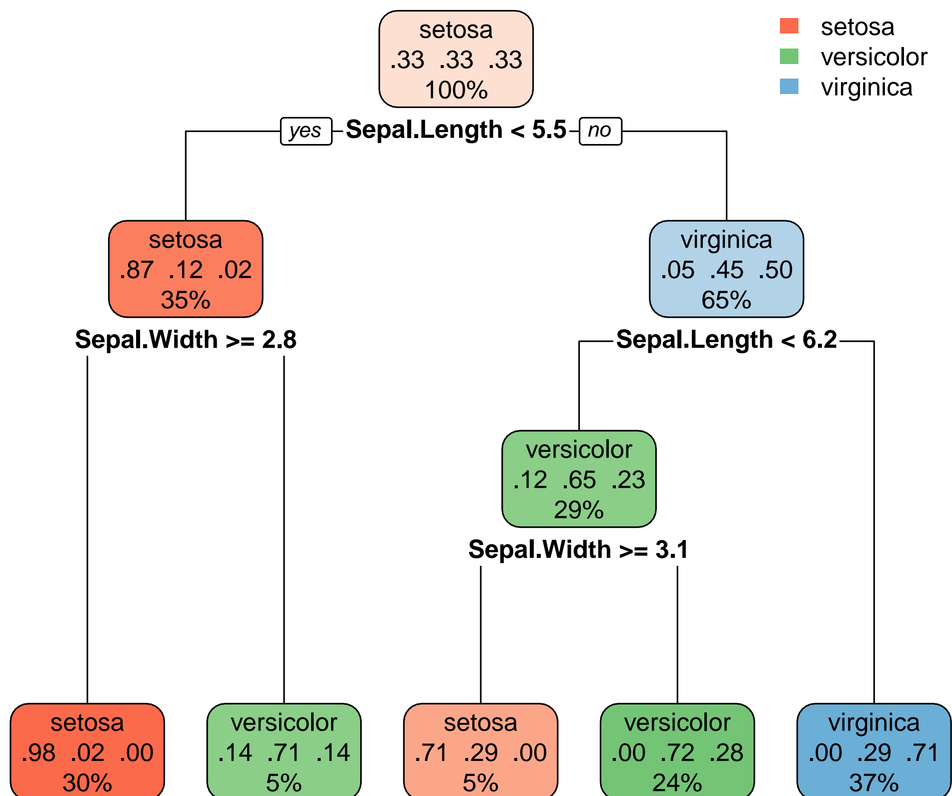

Figure 3.2: The same segmentation as in Figure 3.1, here depicted with the splits as a tree. The decision tree is trained on the Iris dataset, with Sepal Length and Width as explanatory variables and Species as response variable. The coloring of each leaf indicates what the majority class in each leaf is, and thus which class the decision tree would predict for a new observation in that leaf. The decision tree is produced with rpart() (Terry Therneau, 2019), and the plot is made with R-Package rpart.plot() (Milborrow, 2021)

3.1 Classification and Regression Trees

One specific algorithm for creating a decision tree is the Classification and regression Trees (CART) proposed by Breiman et al. (1984). Here, all decision nodes are binary meaning that there are always two branches connected to the node, and the tree splits the data recursively in two halves. Each split is made to minimize a measure of impurity for the two child nodes, but the measure varies depending on which type of regression or classification we are doing.

3.1.1 Regression Trees

In regression problems, the response variable is as we know of a quantitative nature, meaning it takes on numerical values. The task is to predict values as close to the true values as possible. For regression trees in the CART framework, this translates into using residual sum of squares to measure the impurity of a node. The idea is to split the predictor space into B non-overlapping regions, $R_1, R_2, ..., R_M$ to minimize the residual sum of squares(RSS)

$$\sum_{m=1}^{m} \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2 \tag{17}$$

over all leaf nodes. Here, $\hat{y}_{R_m}$ is the constant prediction made for the observations that fall in region R_m. Since we have to take a greedy recursive approach starting from the top node, in contrast to optimizing (17) directly, we evaluate at every node all possible split points and select the split where the two child nodes has the lowest possible RSS, without considering possible splits further down the tree. If we denote the split point at feature j by s_j, when splitting node t containing the set of instances I_t we seek to find the

(j, s_j) that minimize

$$\sum_{i:x_i \in I_L(j,s_j)} (y_i - \hat{y}_{I_l})^2 + \sum_{i:x_i \in I_r(j,s_j)} (y_i - \hat{y}_{I_R})^2. \tag{18}$$

Here, $I_L(j, s_j) = \{i \in I_t : x_{ij} \geq s_j\}$ and $I_R(j, s_j) = \{i \in I_t : x_{ij} > s_j\}$ are the instances from node t falling in the left and right child node respectively. All splits made from node t and below are only relevant for the instances falling into node t, so we have to only consider I_t when evaluating splits.

This process is done recursively until no reduction in RSS is possible. As noted above this is only true when each leaf node contains one training instance, so normally one constraints the tree with a stopping criterion. When reaching the stopping criterion, the recursive splitting is stopped and we make a constant prediction, c_m for each region m,

$$\hat{f}(X) = \sum_{m=1}^{M} c_m I(x \in R_m) \tag{19}$$

which minimize the RSS of the predicted and the response values of the training observations falling in that region.

3.1.2 Classification Trees

A classification tree is much the same as a regression tree. Of course in the classification setting we are interested in our tree predicting a class rather than a value, so the classification tree is made to predict to which class an instance belongs. There are different ways to measure the impurity of a classification tree, and Breiman et al. (1984, p 103) suggest two in the CART framework.

First is the cross entropy, where one select regions to minimize

$$-\sum_{m=1}^{M}\sum_{k=1}^{K} p_{mk} \log p_{mk} \tag{20}$$

Where $\hat{p}_{mk}$ is the fraction of training instances of class k in region m. Later they also suggested minimizing the *Gini index*

$$\sum_{m=1}^{M}\sum_{k=1}^{K} p_{mk}(1 - p_{mk}). \tag{21}$$

Both of these favors regions with class fractions close to 1 and 0, which translates into homogeneous regions in terms of classes represented. Of course, just as the regression tree, we are not actually minimizing these measures, but instead locally minimizing the cross entropy or Gini index in the two child nodes resulting from a split. The resulting prediction of a classification tree is simply $\arg\max_k \hat{p}mk$, which is the majority class of the leaf node. However, we are often interested in the probability of an instance belonging to a specific class, but it is technically trivial to instead return the fractions p_{mk} which approach the probability as n goes to infinity.

3.2 Regularization

As we have noticed, decision trees are prone to over-fitting, and as long as all training instances can be separated in terms of its features, the CART algorithm presented will always result in a perfect fit to the training data in case no stopping criterion is present. A stopping criterion can be of different forms, but the goal is always to reduce the number of splits while at the same time keeping the most important. There are at least tree usual stopping criteria

that sets a absolute limit on the tree. One is to set a maximum depth on the tree, such that there are a maximal number of subsequent splits. Another is to set a maximum number of leaf nodes, which obviously will create deeper but narrower trees. A third is to set a minimum number of instances in each leaf node, such that no split will be made to create nodes with a smaller instance set that the minimum.

All three of these criteria limits the structure directly in some manner, and does not consider the actual reduction in impurity caused by a split. One other stopping criterion or rather splitting criterion that does consider this is a minimum reduction in impurity to execute a split. The problem with this is that it is possible for a split that causes only a slight reduction to enable later splits that causes large reductions. Therefore, another possibility is to first build a large tree without restrictions, and than after the tree is built, consider each split and the subtree resulting from that split to see if the split causes a reduction in impurity dominant to the negative effect of a more complex tree. If that is not the case, one removes the subtree from the tree and continue the process for all subtrees. This process is called *pruning*.

3.3 Ups and downs with trees

Trees are conceptually very simple, they are easily comprehensible and the decision making down to the splitting criteria is something that may resemble human decision making in many cases and that easily can be visualized and interpreted in a practical manner. However, a tree does not have the same predictive accuracy as other simple regression and classification approaches such as linear and Logistic Regression, K-Nearest Neighbors, Support Vector Machines, Neural Networks and more. Second, decision trees have high varia-

tion, meaning that changes in data used for training, however small, can have big impact on a single decision tree. In Section 4 we will explore *boosting*, a method that deals with both of these limitations.

4 Gradient Boosting

In this section we will discuss *ensemble* methods, and in particular boosting and a variant of this called *gradient boosting*. Ensemble methods are methods to combine outputs from multiple learners with the hope that the "committee" of learners are more powerful than each of the individual learners. Considering trees in particular, two of the downsides are as noted low predictive power and high variance. Combining multiple trees into an ensemble can possibly deal with both of these downsides (James, Witten, Hastie, & Tibshirani, 2013, p. 316).

4.1 Ensemble Methods

The general idea of ensemble methods is to train multiple *base learners* and combining the weighted outputs into one output. This can be described as an additive expansion of the form

$$\hat{f}(x) = \sum_{m=1}^{M} c_m T_m(x) \tag{22}$$

where $T_m(x)$ represents a base learner and c_m some constant that represents the weight we assign to the output of each base learner. The c_m can be different for each learner, but a natural approach is to simply average the output leaving $c_m = \frac{1}{M}$. The problem of (3) then turns into finding the combination of base learners which output minimize the empirical training loss. Optimizing (3)

Ensemble methods perform extremely well in a variety of problem domains, have desirable statistical properties, and scale well computationally (Seni & Elder, 2010), and trees are some of the most used *base learners*. Finding the combination of learners that optimize the ensemble is infeasible even if we

restrict the individual learners, so we need a strategy to find a good ensemble. One of the possible strategies for building an ensemble is called *boosting*, which will be discussed in the next subsection.

4.2 Boosting

One way of building an ensemble is by a method called *boosting*. The basic approach is to sequentially train a set of learners, with each learner considering the output of the ensemble of previously trained learners, adding the current learner to the ensemble before training the next one. Boosting can in theory be applied with any base learner, with trees being very popular. The main idea of boosting is the same as that of ensemble methods in general, to combine the output of multiple base learners into one ensemble, efficiently making a strong learner of weak ones. If we let $F(x_i; \theta)$ be a function mapping x_i to y_i that we want to approximate, Friedman et al. (2002) show that the boosting procedure fits an additive model of the form

$$F^{(K)}(x_i; \hat{\theta}) = \sum_{k=1}^{K} \beta_k f_k(x_i; \hat{\theta}_k) \tag{23}$$

Where $f_k(x_i; \hat{\theta}_k)$ is the individual *base learner*. Each base learner is a function of x_i with a given structure optimized with the set of parameters $\hat{\theta}_k$, at boosting step k, and β_k is the expansion coefficient at step k weighting the k'th base learner.

Boosting is a forward stage-wise additive technique, meaning it essentially consists of creating multiple learners in a consecutive order, all fit by, in general, minimizing a loss function over the training data set, where each learner takes into consideration both the output of the previous learners and the re-

sponse values $\mathbf{Y}$ (Hastie et al., 2009, page. 342). In this way, the ensemble of weak learners are boosted into a strong learner which is the full ensemble. Our goal of boosting is to minimize the generalization error for the final ensemble by minimizing the training loss value for some loss function. To solve this optimally, each weak learner f_i should be fitted considering all other learners in the ensemble, both former and future. As this problem in general is intractable, boosting is generally done in a sequential, greedy manner. In particular f_0 is trained in conventional manner to considering $(\mathbf{X}, \mathbf{Y})$, and f_k is trained considering $(\mathbf{X}, \mathbf{Y})$ and the ensemble consisting of the $k-1$ first learners. So if we suppose we have an ensemble model with trees, f^{k-1}, where $k-1$ trees already have been selected, the theoretical objective of f_k, the k'th and next tree in boosting reduces to

$$\hat{f}_k(x) = \arg\min_{f_k} E_{x,y}[l(y, f^{(k-1))}(x) + \beta_k f_k(x))]. \tag{24}$$

Combining all these weak base learners in the end gives an ensemble of weak decision rules, combined into one more complex. The learner f_k may in principle be any function, but in practice this is often a simple, weak learner.

In general, having a learner $f_k(X;\theta)$, the full boosting procedure is

Algorithm 1 Boosting

1. Initialize $f_0(X)$
2. For b = 1 to B:
   (a) Compute $\arg\min_\theta l(y, f^{k-1}(X) + \beta_k f_k(X;\theta))$
   (b) Set $f^k(x) = f^{k-1} + \beta_k f_k$

Boosting emerged first when Shapire (1990) constructed the theoretical

idea of boosting. This initial contribution to boosting then opened the doors for maybe the most influential boosting Algorithm, the Adaptive Boosting algorithm for classification, AdaBoost (Freund, Schapire, et al., 1996). It is adaptive in the way that it adaptively reweights the training data with a distribution D based on the previous classifier's classifications, or in particular misclassifications. AdaBoost then uses the weights of the data points to construct a new classifier and determine the size of its vote based on the misclassification rate. AdaBoost typically uses small classification tree or simply tree stumps as base-learners and at the time dramatically outperformed the single tree and other single base-learners (Ridgeway, 1999), (Meir & Rätsch, 2003), as well as performing well, and more often than not better, than other ensemble methods (Breiman, 1996b), (Dietterich, 2000). The AdaBoost algorithm is a simple boosting algorithm that very much resembles those of the most popular boosting implementations of today.

Schapire et al. (1998) gave an upper bound for the generalization error of an voting classifier such as AdaBoost, and proved that this does not depend on the number of classifiers combined, but on the distribution of the confidence on of the classification results from the classifiers, which is called the "margin", together with the size of the training data as well as the complexity of the base learners. Since AdaBoost produce a good margin distribution, they claimed this could explain the success of the method. Breiman (1997) gave the mathematical explanation, and showed that what AdaBoost is doing is optimizing this margin distribution. He claimed however that this could not be the full or true explanation, as he was able to produce an algorithm with a better margin distribution which performed worse. Reyzin and Schapire (2006) later found that the poorer performance by AdaBoost could be explained by the higher complexity of the base learners in Breiman's algorithm.

Algorithm 2 AdaBoost (Freund et al., 1996)

Input:

- design matrix $\mathbf{X} \in R^{N \cdot M}$ and response vector $\mathbf{Y} \in R^{N \cdot 1}$
- weak learning algorithm $f(\mathbf{X}; \theta)$
- number of iterations K

Initialize $D_1(i) = 1/n$ for all i **Do for** $k = 1, 2, ..., K$:

1. fit f to $(\mathbf{X}, \mathbf{Y})$ given D_k
2. get back hypothesis $h_k : \hat{y} = \hat{f}(X; \hat{\theta})$
3. calculate the error of h_k: $\eta_k = \sum_{i:h_t(x_i) \neq y_i} D_k(i)$. If $\eta_k > 1/2$, then set K = k-1 and abort loop.
4. set $\beta_k = \eta_k/(1 - \eta_k)$.
5. Update distribution D_k: $D_{k+1}(i) = \frac{D_k(i)}{Z_k} \cdot \begin{cases} \beta_k & \text{if } h_k(x_i) = y_i \\ 1 & \text{otherwise} \end{cases}$

Output the final hypothesis $h_{fin}(x) = \arg_{y \in Y} \max \sum_{k:h_k(x)=y} \log \frac{1}{\beta_t}$

The findings of Breiman, that AdaBoost success was due to the optimization of an objective function, namely the margins, led to Friedman (1999) and Mason et al. (1999) independently developed a generalization of this objective function optimizing boosting, Gradient Tree Boosting. To understand why it is called "Gradient" Tree Boosting, we will first explain the concept of *Gradient Descent*, before exploring how this is used in a boosting setting.

4.3 Gradient Descent

To solve Equation (24), we need some optimization method. Gradient descent is a method where the idea is to find the direction of the gradient of some function g you want to minimize, and then take a step in that direction. If we let $g = g(\mathbf{X}; \theta)$ be an objective function of $\mathbf{X}$ with a set of parameters θ that we want to minimize. The idea is then to calculate the gradient of g with respect to θ, $\partial_\theta g(\mathbf{X}; \theta)$, and then updating the parameters θ in the negative direction of the gradient. We then recalculate the gradient with respect to the updated parameters and take another step and so on. As the gradient $\partial_\theta g(\mathbf{X}; \theta)$ points in the direction of the steepest ascent, the direction of steepest descent is given by $-\partial_\theta g(\mathbf{X}; \theta)$. Moving in this direction will eventually cause the gradient in that same direction to be non-negative, meaning one can no longer improve the loss by moving on that line. At that point, one should again find the gradient at the new point and do a new minimization in the new steepest descending direction. Usually, one however does not want to move all the way to the minimum at each line of descent, as any movement from the starting point may change the direction of the gradient. The standard procedure is to move in the direction of the gradient only by some small step size λ that we call the *learning rate*, thus we have formally $\theta = \theta - \lambda \cdot \partial_\theta g(\mathbf{X}; \theta)$.

To motivate the need for stochastic sampling in Gradient Tree Boosting, we will conceptualize Gradient Descent as finding a route from an arbitrary point to the, at least locally, lowest point on a height map. Starting out on the given point, we move in the direction where the height decreases most rapidly, we do this for a given step length or until we reach the minimum along that line of direction, before we reassess the slope in all directions and find a new steepest descending one. If we now imagine this height map in a multidimensional setting, having as many dimensions as our design matrix $\mathbf{X}$, we can think of the value of the loss function over unseen data in a given point on the multidimensional map as the true height of the terrain. The height curves of the map correspond to the value levels of the objective function, and the distance between the curves indicates the gradient. One must not confuse the height indicated by the height curves with the real height of the terrain the map represents, as any selection of data from a population will have a distribution not identical to the entire population. By altering the selection of training data, the height curves are somewhat altered, but as the training data comes from the same distribution as unseen data, the map to some degree still represent the true curvatures of the terrain. However, the smaller the size of the training data, the more it will differ from the true population.

Now, imagine that you have moved to some valley, a local minima in the objective function sense, where the height increases in all directions from the point. However, there is a even lower valley next to it, separated by a hill between. If we use the full training set when calculating the gradient, we will never reach this lower valley, but if we select only a subset of the training set, the height map changes, and the smaller the subset, the larger the change. Now, if we are lucky, we select a subset such that the hill between the two valleys disappears on the map, and there is a route with a decreasing height

into the lower valley.

This strategy is known as Mini-Batch Gradient Descent or Stochastic Gradient Descent, as the selection of the subset is done in a stochastic manner. This strategy induces some randomness into the Gradient Descent, and in that way has a regularizing effect on the Gradient Descent (Wilson & Martinez, 2003).

4.4 Gradient Tree Boosting

We now try to unite the idea of gradient descent with the one of boosting, and focus on boosting with CARTs as defined in Section 3.1 as base learners. There are many reasons why trees and in particular CARTs are well suited for gradient boosting purposes (Hastie et al., 2009, p 351). First, it is very flexible in terms of complexity, which range from the very simple tree stump model with only one split and to a complete fit of the training data. Traditionally, this means that the user will have to pre-define the allowed complexity in terms of the regularization parameters from Section 3.2. The upside is that this still leaves room for tuning complexity in accordance with the data in question. Further, we will see later that it is possible to automate the selection of complexity for the individual tree which means that we leave space for the algorithm to learn an optimized complexity. Furthermore, trees naturally incorporate both continuous and categorical variables including missing values, and it is not sensitive to transformations of the data such as scaling. Feature selection is done inherently, which mean it is insensitive to irrelevant features. Trees are also natural good handlers of outliers, as the outliers has little effect on the splitting procedure and only local effect on the leaf weights. It turns out also that the expected reduction in training loss from the additional tree

in an ensemble can be approximated by something that is very much related to the weights of the three, which we will explore further in Section 4.5.

In gradient boosting, we are not doing gradient descent in parameter space as in Section 4.3, but rather in function space. Decision trees partitions the predictor space into T regions, $R_1, R_2, ..., R_T$ represented by a set of leaf nodes L, assigning the same constant value w_t for all instances present in a particular region. With regression trees as base learners, we have that boosting fits an additive model of the form

$$f^K(x) = \sum_{k=1}^{K} f_k(x) = \sum_{k=1}^{K} \sum_{t=1}^{T_k} w_{tk} I(x \in R_{tk}) \tag{25}$$

(Friedman, 1999). Where f^K is the ensemble of K trees which is to be seen as the current approximation of f in our model $y_i = f(x_i) + \epsilon$ at stage K. The indexes t and k indicates that we are in the t-th leaf and k'th tree. At each step of the boosting procedure, a k'th tree are fit to the loss function given the approximation of f consisting of the first $k-1$ trees, $F^{k-1}(x)$ such that

$$\hat{f}_k(x) = \arg\min_{f_k} E_{x,y}[l(y, f^{(k-1))}(x) + f_k(x))]. \tag{26}$$

If we denote the current approximation of f at any point in boosting as $F(x)$, which is the ensemble consisting of the already fitted trees, we can view every additional boosting step as a step in minimizing

$$\phi((F(x))) = E_y(l(y, F(x))|x). \tag{27}$$

From gradient descent we know that the direction of steepest descent in this

optimization problem is

$$g(x) = \frac{\partial \phi F(x)}{\partial F(x)}. \tag{28}$$

Now, the idea of gradient boosting is that having the ensemble $f^{k-1}(x) = \sum_{m=1}^{k-1} f_m(x)$, then analogous to gradient descent, let

$$f_k(x) = -p_k g_k(x) = -p_k \frac{\partial \phi f^{k-1}(x)}{\partial F^{k-1}(x)} \tag{29}$$

Where p_k is the step size. $g_k(x)$ is here the unconstrained gradient, but the problem with using this is that it can only be evaluated at each x_i and that it does not generalize to other points. Therefore, we rather want to constrain the step direction to be in the form of a CART, and thus find the CART that is closest to $g_k(x)$. A solution to this is to do this in the squared error sense, and find $f_k(x)$ such that

$$\theta_k = \arg\min_{\theta_k} \sum_{i=1}^{N} (-g_k(x_i) - f(x_i; \theta_k)) \tag{30}$$

This is the same objective as for fitting a single tree for least squared loss, with y_i substituted by $-g_k(x_i)$. This is exactly what Friedman (1999) does, and the full algorithm for gradient boosting is then

4.5 Extreme Gradient Boosting

One of the most favored gradient tree boosting methods of the last couple of years have been the one created by Chen and Guestrin (2016), Extreme Gradient Boosting, or XGBoost. In this section we will present the XGBoost algorithm as this acts as a basis for parts of the Stochastic Automatic Gradient Tree Boosting.

Algorithm 3 Gradient Tree Boosting

1. Initialize $f_0(x) = \arg\min_{f(x_i)} \sum_{i=1}^{N} L(y_i, f(x_i))$
2. for k = 1 to K:
   - For $i = 1, 2, ..., N$ compute

$$r_i k = -\left[\frac{\partial l(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{k-1}} \tag{31}$$

   - Fit a regression tree to the targets r_{im} giving terminal regions $R_{jk}, j = 1, 2, ..., J_k$.
   - For $j = 1, 2, ..., J_k$ compute

$$\gamma_{jk} = \arg\min \gamma \sum_{x_i \in R_{jk}} l(y_i, f_{k-1}(x_i) + \gamma) \tag{32}$$

   - Update $f_k(x) = f_{m-1}(x) + \sum_{j=1}^{J_k} \gamma_{jk} I(x \in R_{jk})$
3. Output $f(\hat{x}) = f^K(x)$

In XGBoost, Chen and Guestrin implements for CART a approximation-based boosting strategy originally proposed by Friedman et al. (2002). We still want to minimize (24), but we want to approximate this with the training data empirical counterpart, and do a second order taylor expansion around $\hat{y} = f^{k-1}(x)$ to gain analytical tractability

$$\hat{l}(y, \hat{y} + f_k(x)) = l(y, \hat{y}) + g(y, \hat{y}) f_k(x) + \frac{1}{2} h(y, \hat{y}) f_k^2(x). \tag{33}$$

Here, $g(y, \hat{y}) = \frac{\partial}{\partial \hat{y}} l(y, \hat{y})$ and $h(y, \hat{y}) = \frac{\partial^2}{\partial (\hat{y})^2} l(y, \hat{y})$ are first and second order gradient statistics on the loss function. Now, (24), the theoretical objective to be optimized at the k'th boosting iteration is replaced with

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^{n} l(y_i, \hat{y}_i^{k-1} + f_k(x_i)) &\approx \frac{1}{n} \sum_{i=1}^{n} \left(l(y_i, \hat{y}_i^{k-1}) + g_{ik} f_k(x_i) + \frac{1}{2} h_{ik} f_k(x_i)^2 \right) \\ &= \frac{1}{n} \sum_{i=1}^{n} l(y_i, \hat{y}_i^{k-1}) + \frac{1}{n} \sum_{t \in L_k} (\sum_{i \in I_{tk}} g_{ik} w_{tk} + \frac{1}{2} h_{ik} w_{tk}^2) =: l_k(q_k, w_k) \end{aligned} \tag{34}$$

where

$$g_{ik} = g(y_i, f^{k-1}(x_i)) and h_{ik} = h(y_i, f^{k-1}(x_i)). \tag{35}$$

For a given feature mapping q_k the weight estimates $\hat{w}_k$ minimizing $l_k(q_k, w_k)$ are given by

$$\hat{w}_t k = -\frac{G_{tk}}{H_{tk}}, G_{tk} = \sum_{i \in I_t k} g_{ik}, H_{tk} = \sum_{i \in I_{tk}} h_{ik} \tag{36}$$

where I_{tk} is the instance set of leaf t. Using the weights (36), the improvement in training loss is given by

$$l_k(q_k, \hat{w}) - \frac{1}{n} \sum_{i=1}^{n} l(y, \hat{y}_i^{k-1}) = -\frac{1}{2n} \sum_{t=1}^{T_k} \frac{G_{tk}^2}{H_{tk}} \tag{37}$$

Now, as mentioned in Section 3, finding the optimal tree structure would mean considering every possible tree structure, which leads to combinatorial explosion, and the splitting is therefore done in a greedy fashion (Chen & Guestrin, 2016): In the above procedure, $R_t(j, s_j)$ is the difference in training

---

Algorithm 4 Extreme Gradient Boosting

---

1. Predict a constant value over all features, $\hat{w} = -\sum_{i=1}^{n} \frac{g_{ik}}{h_{ik}}$
2. For each leaf node t: For every feature j, compute the training loss reduction

$$R_t(j, s_j) = \frac{1}{2n} \left(\frac{(\sum_{i \in I_{L(j,s_j)}} g_{ik})^2}{\sum_{i \in I_{L(j,s_j)}} h_{ik}} + \frac{(\sum_{i \in I_{R(j,s_j)}} g_{ik})^2}{\sum_{i \in I_{R(j,s_j)}} h_{ik}} - \frac{(\sum_{i \in I_{tk}} g_{ik})^2}{\sum_{i \in I_{tk}} h_{ik}} \right) \tag{38}$$

   Where $I_L(j, s_j) = i \in I_t k : x_{ij} =< s_j$ and $I_R(j, s_j) = i \in I_t k : x_{ij} > s_j$. Select the split, s and s_j where (38) is maximized creating two new leaves from t.
3. Repeat step 2 until some tree complexity criterion is reached.

---

loss reduction (37) between a three where t is a leaf node and the same tree, where t is a split node, with two child nodes L(t) and R(t).

4.6 Regularization in Gradient Tree Boosting

4.6.1 Learning Rate

One way to prevent overfitting of the individual base learner to cause overfitting of the ensemble is the learning rate λ. The learning rate in boosting is analogous to the step size in gradient descent. The best way to do gradient descent would be to continuously evaluating the gradient and moving in the direction of steepest descent by letting the step size go to 0, but one also has

to take into account the increased computational demands caused by a small learning rate. With a learning rate close to 0, the impact of each base learner is negligible, and the learning process takes a relatively longer time, and with a learning rate of 1, there is no scaling. Reduced learning rate is however associated with better performance, at the cost of increased computational complexity.

4.6.2 Ensemble complexity

In the algorithms above, we have assumed an ensemble of K trees. This K has to be set by the user, and is the last way of regularization that will be presented here is to regularize the ensemble complexity. The most usual way of doing so is to set a max limit on the number of boosting iterations. This reduces the complexity of the ensemble at the same time as reducing the computational cost. The optimal K will be dependent on the data in question, so a usual way to select it is by cross-validation.

5 Automatic Gradient Tree Boosting

Lunde, Kleppe and Skaug (2020) proposed a information theoretic approach to automatically tune the complexity of a GTB ensemble. More specifically, they introduce an upper bound for the reduction in generalization-loss from adding another split to the single tree. This approximation is utilized for model selection without hyperparameter tuning, which decrease the need for user input, and relative with cross validated XGBoost drastically reduce time spent on selecting a model with tuned parameters (Lunde et al., 2020).

They recognize that for all nodes in the single tree, the evaluation of whether to make a new split or stop the tree splitting at a node is independent of all other splits and nodes in the tree, and identical, with the only difference being a different subset of the training data. Thus one only has to consider the data passed to the node in question, and the split or no-split decision can be made independently.

We know that in XGBoost, the decision of whether or not to split a node is made based on a minimization of a regularized objective function evaluated on the training data. A node is split at the point where it reduces the objective function maximally, and the splitting process is continued as long as there can be made a reduction in this objective function and until some complexity threshold is reached. The goal of the process is of course to minimize the generalization loss, for which we know this objective function, which includes the training loss, to be an optimistic estimate.

With the information criterion approach of Lunde et al. (2020), one rather finds an approximation of the optimism of the training loss reduction from a node relative to the generalization loss reduction. One can then correct for optimism in the training loss reduction, and find an approximation of the gen-

eralization loss reduction, which in order can be used for decisions on when and how to split a node and when to stop boosting iterations. Specifically, Lunde et al. finds an upper bound of the optimism. The algorithm implementing these results are called Automatic Gradient Tree Boosting or AGTBoost.

As said, the goal idea behind AGTBoost is to find approximations for the generalization-loss counterparts of the measures of reduction in training loss R_t and $R_t(j)$, where

$$R_t(j) = \max_{\mathbf{s_j}} R_t(j, s_j), j = 1, ..., m \tag{39}$$

is the reduction in training loss conditional on feature j and

$$R_t = \max_{\mathbf{j \in (1, ..., m)}} R_t(j) \tag{40}$$

is the unconditional reduction in training loss. The generalization-loss counterparts are denoted $R_t^0(j)$ and R_t^0 respectively. This should further be done in a way that excludes the need for manual tuning of hyperparameters.

If we recognize that the split/no-split decision at every node is identical, except for different subsets of the training data being passed to each node, we notice that at every node, we need to consider the two options of no-split, leaving the node a root tree, consisting of a single node, and the do-split, making it a stump tree, with two leaf nodes. Denoting the optimism of the root model C_{root} and the stump model C_{stump}, we have that

$$E_{y^0,x^0}[R^0(j)] = E_{x,y}[R(j)] + C_{root} - C_{stump}. \tag{41}$$

And under the assumption that the features split over are independent of the response y, Lunde et al. shows that the three terms on the right hand side

of (41) may be estimated efficiently for each candidate split and allow for correction of the training loss reduction for optimism, this will be elaborated in Section 5.2.

In turn, this allows for some changes in the gradient tree boosting algorithm. First, instead of K iterations of boosting, where K is selected as a hyperparameter set before training, AGTBoost stops boosting at the moment when

$$\tilde{R}_{\delta} = \delta(2-\delta)R_1 + \delta(C_{root,1} - \tilde{C}_{stump,1}) > 0. \tag{42}$$

in words the moment when there is no reduction in generalization-loss from splitting the top node of a new tree.

Second, the estimated generalization-loss also forms a natural stopping criteria for node splitting in the individual tree. Instead of splitting up until some regularization criterion is reached, the splitting of the single node at level t is stopped when

$$\tilde{R}_t^0 < 0. \tag{43}$$

Meaning that when (43) is true for all current leaf nodes, the tree building for the individual tree is complete.

Combined, these two criteria for stopping of node splitting and tree boosting removes the need of several hyperparameters that else would need to be manually tuned, and we get the vanilla algorithm of AGTBoost.

5.1 Global Subset

Later, Lunde and Kleppe (2020) introduce a slight change to Algorithm 5. Instead of continuing the splitting process of the single tree until no more splits can be made with an reduction in generalization-loss, one can for each

Algorithm 5 Automatic Gradient Tree Boosting Algorithm (Lunde et al., 2020)

Input: - A training set $D_n = (x_i, y_i)_{i=1}^n$ - A differentiable loss function $l(.,.)$ - A learning rate $\delta \in (0, 1]$

1. Initialize model with a constant value: $f^0(x) = \arg_\eta \min \sum_{i=1}^n l(y_i, \eta)$
2. **while** the inequality (42) evaluate to **false**
   (a) Compute derivatives (35) given D_n
   (b) Determine the structure q_k by iteratively selecting the binary split that maximizes (38) until the inequality (43) evaluates to **true** for all leaf nodes t
   (c) Determine leaf weights (36), given q_k
   (d) Scale the tree with the learning rate $f^{(k)}(x) = \delta w_{qk}(x)$
   (e) Update the model Set $f^k(x) = f^{k-1} + f_k$

   end while
3. Output the model: Return $f^{(k)}$

candidate split evaluate the expected generalization loss of splitting the root node of a next tree with that of making the candidate split. If the expected reduction is larger for the root split, one stops the splitting process, and move on to the next tree. The effects of this implementation is that earlier trees seems to grow much shallower, leaving space for later trees to learn (Lunde & Kleppe, 2020).

5.2 Information Criterion

The main solution for automating the Gradient Tree Boosting in Lunde et al. (2020) is the introduction of an information criterion as an approximation of the optimism in Gradient Tree Boosting. As seen, by using an information criterion for approximating optimism, one can add this to the training loss in the training procedure to estimate the generalization loss directly and utilize this for model selection. What Lunde et al. does is directly compare the approximated generalization loss before and after any candidate split in the tree building process to find the generalization loss reduction and evaluate whether there is any gain from splitting any leaf node of the current tree or if the splitting should stop at that point. This also facilitates the identification of when to stop adding new trees to the ensemble, as there is no gain in splitting the root node in the next tree, and the boosting is thus stopped.

The optimism of the model f, C, is by definition

$$C \approx E[l(y^0, f(x^0; \hat{\theta}))] - E[l(y_1, f(x_1; \hat{\theta}))]. \tag{44}$$

Here, (x_1, y_1) is an instance of the training data and (x_0, y_0) is an unseen data point drawn from the underlying distribution of the training data. In general,

this optimism can be approximated by

$$\tilde{C} = \mathbf{tr}(E(\nabla_\theta^2 l(t, f(x;\theta_0))]Cov(\hat{\theta})) \approx C \tag{45}$$

where θ_0 is the set of parameters that minimize (2) and the approximation $\hat{\theta}$ is not at the boundary of parameter space (Anderson & Burnham, 2004, Eqn. 7.32), (Lunde & Kleppe, 2020).

Now, as mentioned in Section 5, in tree building we only need to consider the decision of whether to split or not split the single node given only the data passed to that node (Lunde et al., 2020), as this decision is independent on all other splits made. If (41), the expected reduction in generalization loss caused from making the split is positive, we split the node. To estimate (41), we use Equation (37) to find the expected reduction in training loss when splitting on feature j. We then need to find C_{root}, and since there by definition are no split points in C_{root}, this can be estimated by (45) over the subset of data contained in the particular leaf.

However, estimating C_{stump} is not so straight forward. Given a split feature j and a splitting point s_j this could be calculated (45) as the two resulting child nodes can be treated as two root nodes with the respective proportion of the training data according to the split s_j. the problem is that the optimization over which feature j and which point s_j to split at also induce some optimism into the model, and thus Equation (45) can not handle this directly.

The solution proposed by (Lunde et al., 2020) is to first consider the optimism of the stump model $C_{stump}(j)$ evaluated on a single fixed feature j, where the j-th column of the training data design matrix $x_j = x_{\cdot,j}$ is assumed to be independent of the response y. Then letting $u_i = i/n$ where $f(\cdot;\hat{w}_l(u_i), \hat{w}_r(u_i))$, $i = 1, ..., n-1$ is the tree stump with observations $x_{1:i,j}$ in the left node and

$x_{i:n,j}$ in the right node, the distribution of the difference in training and generalization as a function of i converges to what can be expressed by the joint distribution of $\tilde{C}_{root}(1 + S(\tau(u))$ as

$$n[E_{y^0,x^0}[\hat{l}(y^0, f(x^0; \hat{w}_l(u_i), \hat{w}_r(u_i))) - \hat{l}(\mathbf{y}, f(\mathbf{x}; \hat{w}_l(u_i), \hat{w}_r(u_i)))] \xrightarrow[n\to\infty]{D} n\tilde{C}_{root}(1+S(\tau(u))). \tag{46}$$

Here, $S(\tau(u))$ is a stochastic process with dynamics given by

$$dS(\tau) = 2(1 - S(\tau))dt + 2\sqrt{2S(\tau)}dW(\tau) \tag{47}$$

where $W(\tau)$ is a Wiener process. The time τ follows $\tau = \frac{1}{2}\log\frac{u(1-e)}{e(1-u)}$ where $u = \min[1-\epsilon, \max(\epsilon, \frac{i}{n})]$, $0 < \epsilon << 1$. We notice that $S(\tau(u))$ is a function of the splitting point of the feature j. If we suppose that $x.j$ contains $a + 1$ distinct values, there are a different possible split points for this feature, we can solve this by taking the expected maximum of $[\tilde{C}_{root}(1 + S_{\tau k})]_{k=1}^{a}$.

Now we have an upwards biased approximation to $C_{stump}(j)$,

$$\tilde{C}_{stump}(j) = \tilde{C}_{root}(1 + E[\max_{1\leq k\leq a} S(\tau_k)]) \tag{48}$$

that converges to $C_{stump}(j)$ as $n \to \infty$ (Lunde et al., 2020).

If we then let

$$B(\mathbf{X}) = max_{\mathbf{1=<j=<m}} B_j(\mathbf{x}_{.j}), B_j(x_{.j}) = max_{\mathbf{1<=k<=a}} S_j(\tau_k). \tag{49}$$

and take the expectation of the maximum over both j and k, we get an upwards biased approximation of C_{stump} optimized over multiple features. $B(\mathbf{X})$ is here defined by a specification of the Cox-Ingersoll-Ross process (Cox, Ingersoll, & Ross, 1985). This is then approximated using the Gumbel distribution

(Gombay & Horvath, 1990) (Lunde & Kleppe, 2020). To make this generally applicable to the node t to which the fraction of data passed is π_t, we let

$$\tilde{C}_{stump} = \pi_t \tilde{C}_{root}(1 + E[B(\mathbf{x})). \tag{50}$$

By the assumption that the B_js are independent, C_root is shown to be calculable using the sandwich estimator (Huber et al., 1967) and the empirical Hessian matrix (Lunde et al., 2020).

6 Stochastic Automatic Gradient Tree Boosting

AGTBoost as described by (Lunde & Kleppe, 2020) has some limitations compared to the most popular implementations of GTB. The one that is explored further in this thesis is the lack of subsampling, which is an important technique that has a regularizing effect and is known to improve performance (Friedman, 2002).

6.1 Sampling in Boosting

One problem of the single decision tree is that it suffers from high variance (James et al., 2013, p. 316). This implies that when fitting trees on different parts of the same data, the produced decision trees can differ substantially based on the observations selected for training. Breiman (1996a) introduced a concept that reduces this problem, namely bagging or bootstrap aggregation. This concept is similar to boosting in the way that is aggregates multiple learners in order to make a prediction. For each fitted learner, a new bootstrap sample is drawn from the training data, injecting randomness into the procedure. This reduces the variance in the final learner, which now is an ensemble, and improves the performance. Friedman (2002) took concept of subsampling the training data into boosting, specifically by drawing a subsample of size $\alpha \in (0, 1]$ without replacement at each iteration of the boosting procedure. This is referred to as Stochastic Gradient Boosting. Friedman indicated that stochastic gradient boosting may outperform deterministic gradient boosting, where $\alpha = 1$, depending on the specific problem and the sample size α. A lower sample size increases variance in each individual learner, but this also

reduce the correlation between the learners which may cause an averaging effect, and in that way lower the variance of the full ensemble. In addition to this, subsampling in boosting is also helpful in the way that it reduces computational cost at each boosting iteration. The most popular implementations of Gradient Decision Tree Boosting today also allows for subsampling between tree building.

6.2 Creating the algorithm

The main focus of this thesis is the introduction of stochastic (sub)sampling in the boosting process of AGTBoost. As seen in (Friedman, 2002), row and column subsampling in each boosting iteration, training the individual trees on more or less varying data, can reduce the variance and increase the predictive quality of the ensemble. We seek to introduce row sub-sampling in the AGTBoost procedure. In doing so, we seek to find an alternative to Algorithm 5. On the superficial level, the stochastic automatic gradient tree boosting algorithm is similar to Algorithm 5 with some obvious changes.

We sample a subset D_ζ from which we calculate a initial prediction, and for each tree that are built in the boosting process, we sample a new subset from which we built the trees. When evaluating (42), we evaluate $\tilde{R}_{delta}$ the reduction in generalization loss for the entire training set. However, the evaluation of optimism for each tree is done considering only the data points in the subset D_ζ. We know from Section 2.2 that optimism is an averaged measure, and by considering the optimism for the subset only, we over-estimate the optimism. This is easily seen considering that there are no optimism in the expected training loss reduction for the data points not in the subset selected for training that specific tree, and thus if we included the optimism from these

Algorithm 6 Stochastic Automatic Gradient Tree Boosting

Input:

- A training set $D_n = (x_i, y_i)_{i=1}^n$
- A differentiable loss function $l(.,.)$
- A learning rate $\delta \in (0, 1]$
- A sampling rate $s \in (0, 1]$

1. Sample a subset $D_\zeta^0 \subseteq D_n$ with $\zeta = s \cdot n$ observations
2. Initialize model with a constant value: $f^0(x) = \arg_\eta \min \sum_{i=1}^{\zeta} l(y_i, \eta)$
3. **while** the inequality (42) evaluate to **false**
   (a) Sample a new subset $D_\zeta^{(k)} \subseteq D_n$
   (b) Compute derivatives (35) given D_ζ
   (c) Determine the structure q_k by iteratively selecting the binary split that maximizes (38) until the inequality (43) evaluates to **true** for all leaf nodes t
   (d) Determine leaf weights (36), given q_k
   (e) Scale the tree with the learning rate $f^{(k)}(x) = \delta w_{qk}(x)$
   (f) Update the model Set $f^k(x) = f^{k-1} + f_k$

   end while
4. Output the model: Return $f^{(k)}$

points in the averaged measure of the optimism, it would decrease. Considering (11), we see that the optimism of the expected reduction in training loss for the full training data is the optimism of the subset scaled by the sampling rate. If we denote the optimism of the expected reduction in training loss from the root and the stump model C^{ζ}_{root} and C^{ζ}_{root} respectively, we now seek the point in the boosting process where

$$\tilde{R}_{\delta} = \delta(2-\delta)R_1 + s\delta(C^{\zeta}_{root,1} - \tilde{C}^{\zeta}_{stump,1}) > 0. \tag{51}$$

There are however two obvious problems with this approach. First, when using a subsample of the training data for training the tree, we must consider the possibility that we select a subsample in which there is no possible reduction in generalization loss while this is still possible for the entire training set. Second, we note that it is possible that even if there is no possible reduction in generalization loss from the tree model $f_k(x)$ trained on the subsample D^{ζ}_n, it may exist a possible reduction in generalization loss from the tree model $f_{k+1(x)}$ given $f_k(x)$. To cope with both of these problems, we do not want to use (51) directly to identify the stopping point of the boosting procedure. Ideally, we want to consider (51) for every possible subset and every possible combination of subsets of D_n. With decreasing sampling rate and increasing data size, this quickly becomes infeasible. We therefore introduce a function $P(s)$ which decides how many trees we allow to train without a reduction in the estimated generalization loss for the ensemble before we stop the boosting procedure.

In clear opposition to the goal of AGTBoost, namely to have a gradient tree boosting algorithm free of tuning necessity, we now have two parameters to tune, the sampling rate s and the function $P(s,n)$. The function $P(s,n)$

is the easiest to tune considering the natural stopping point (51). The only purpose of this function is to reduce computation time and complexity, and to ensure the best possible tree ensemble, this should be set as high as possible to maximize predictive power of the ensemble, and as low as possible to minimize training time. With decreasing sampling rate, the trees will have a lower covariance, and thus it is natural to have $P(s, n)$ decreasing in s, and with a value of 0 in $s = 1$. P(s) The optimal sampling rate S however, is dependent on all the number of observations and variables in the data, how the response variable is distributed and other properties. Thus, this must be tuned in a much more careful way.

6.3 Sampling induced variance

One issue with stochastic compared to deterministic AGTBoost is that in stochastic AGTBoost, when we sample a subsample of the training data at random at each boosting iteration, we also randomize the tree built at each iteration. This means that training multiple stochastic AGTBoost models on the same training data is unlikely to produce the exact same ensemble. The lower the sampling rate, the more different each ensemble will be to another. This means that we can train multiple models on the same data having output models with different predictive power. When evaluating methods, we also have to take into consideration the variance of each method. Even though a method is on average better than another, the average worse may be prefered if the average best don't produce as consistent as the worse.

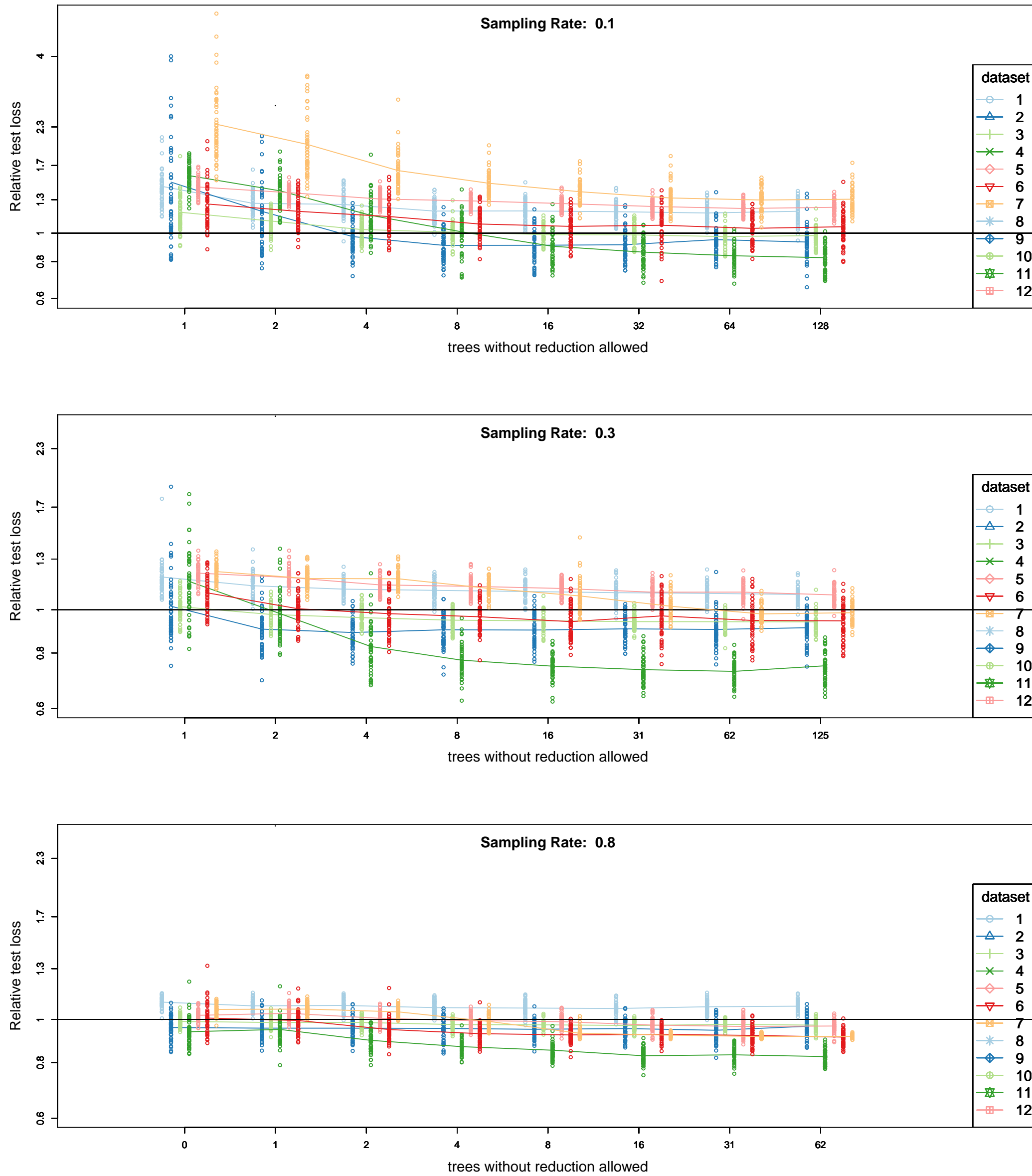

Figure 6.1: Mean squared error for each individual training of stochastic AGTBoost with a constant sampling rate, with different maximal number of boosting iterations with no reduction of generalization loss allowed. For each of the datasets in Table 3, 50 models for each maximum no reduction are fitted on the exact same train/test split, thus all variance coming from the nature of the subsampling in training. The lines indicates the relative averaged mean squared error over the 50 models for each dataset. This is plotted for three different sampling rates, and it's clear that a higher maximum number of no reduction trees gives better mean accuracy and lower variance, and that this effect is decreasing with increasing sampling rate.

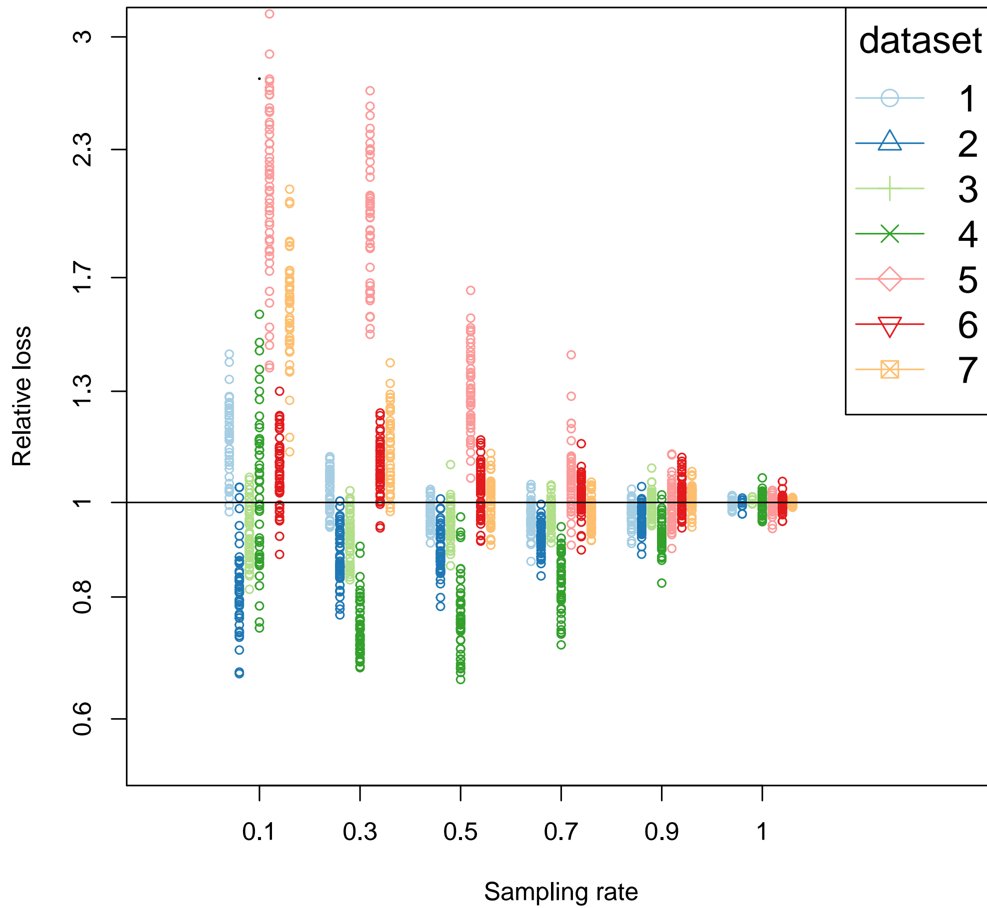

Figure 6.2: Mean squared error for each individual training of stochastic AGTBoost with a constant sampling rate, relative to the average mean squared error of deterministic AGTBoost trained on the same data with the same train, test split. For each of the datasets, the 50 models are trained at each sampling rate with the exact same train, test split. All variation in performance for each datasets thus comes from the algorithm. The datasets used are the 7 first datasets from Table 3.

6.4 Dynamic Sampling

There is no obvious way for a particular dataset to decide the best sampling rate without manual tuning. A normal sampling fraction is 1/2 (Hastie et al., 2009, p. 365), but for many datasets it would be better with a smaller fraction and for other a higher fraction. One other possibility is to use a dynamic sampling rate. At first, this technique was proposed as a way of tuning the sampling rate, with increasing sampling rate until the sample was good enough, or representable, of the entire dataset (Provost, Jensen, & Oates, 1999), (John & Langley, 1996). Provost et al. proposes a general progressive sampling method where the sample sizes are given by a sampling schedule

$$S = n_0, n_1, n_2, n3, ..., n_k \tag{52}$$

where each n_i is the number of samples used for training at the i'th stage of training. Provost et al. defines n_i as $n_i = n_0 \cdot a^i$, so that the sampling rate changes with a rate of a at each stage, where John and Langley defines n_i as $n_i = n_0 + (i \cdot n_\delta)$ which Provost et al. calls arithmetic sampling. Lazarevic et al. (2001) and Sadid et al. (2004) takes this into the boosting setting, with something they call progressive boosting. Here, the sampling rate is changed according to the sampling schedule in (52) at each iteration of the boosting process. Inspired by this approach, we will in this thesis implement the aforementioned progressive sampling techniques and some modified version of these into the automatic gradient tree boosting. We will also propose two other progressive techniques that we will implement. When implementing progressive sampling into the AGTB setting, we have to take a couple of things into consideration. First, progressive boosting as proposed updates the sample size in each boosting iteration. This means that with in certain cases

where $a < 1$ or n_δ, there will be implied a maximum number of iterations, as the sampling rate reaches 0.

As an alternative approach, we introduce another sampling schedule where we do boosting with a given sampling rate until we no longer are capable of improving our ensemble with the given sampling rate. At that point, we update our sampling rate in an arithmetic or geometric fashion, and we keep doing this for a given range of values. We also introduce two other approaches with a random sampling rate where the sampling rate is randomly selected in the allowed range. In the first approach, we randomly select a sampling rate at each boosting iteration. For the other, we randomly select a sampling rate, and continue boosting with this sampling rate until we no longer are able to improve the ensemble. At that point, we randomly select a new sampling rate and continue boosting. When we have tried a given number of sampling rates without improvement, we stop the boosting process.

6.5 Sampling schemes

6.5.1 Random

The first sampling scheme we test is a random sampling scheme, where the sampling rate is drawn uniformly at each boosting iteration. Since the sampling rate is random, we allow for a relative high number of boosting iterations without improvement, and the in this case namely 30. Letting the sampling rate be random, we allow both for high sampling rates where we utilize all data to find the best additional tree to the ensemble, at the same time allowing for low sampling rates, which maximize the stochastic nature which we hope to be useful.

6.5.2 Half

The second sampling scheme we test is the Half-scheme. Here, we start out at a relatively high sampling rate, close to or equal to one. When we are no longer able to improve our ensemble, we half the sampling rate and continue training, until we get to some predefined lower limit. The intuition behind this scheme is to have a high sampling rate when we are far away from any minima in the loss terrain, and lower and lower as we for each iteration are more likely to get stuck in some local minimal, and want bigger jumps to find the best directions. We denote this sampling scheme with Half(S) where $S \in (0, 1]$ is the starting sample rate.

6.5.3 Pulse

The third sampling scheme we test is the Pulse-scheme, where we start at a sampling rate close to 1, and then decrease the sampling rate whenever we are not able to improve the ensemble. If we however are able to improve the ensemble at a lower sampling rate than 1, we reset the sampling rate to 1 and start over. When we are not able to improve the ensemble for all sampling rates down to a pre-defined lower limit, we stop the boosting process. By this sample scheme, we allow for the same decreasing sampling rate strategy of the Half scheme, but when improvement is made, we again try to find further improvement at a high sampling rate, where we know the convergence to be faster.

6.5.4 Switch

The last sampling scheme we test is a Switch-scheme. Here, we have two sampling rates, in this case one high and one low, and when we are not able to

improve the ensemble for one of them, we simply switch to the other. When we are not able to improve the ensemble for any of the two, we stop the boosting process. The idea behind this scheme is to train at a relatively high sampling rate, but if for some reason we are not able to continue improvement, we use a lower sampling rate to increase chances of getting out of local minima. In contrast to the Pulse scheme however, we do not use multiple decreasing sampling rates, which increases the time spent on training.

All of the Sampling schemes, except from the random one, comes from an idea of initiating the boosting with a higher sampling rate and reducing it as the generalization loss is reduced. The intuition behind this comes from the idea of gradient descent, and the thought is that as the boosting progress, the likelihood of finding a local minima increases, and the reduction in sampling rate increases randomness and thus the chance of escaping these minima. As one can see from Figure 6.3, most of the boosting iterations of all sample schemes are done with the highest sampling rate, but continues with lower sampling rates when the improvement stops on the higher one.

6.6 Implementation

The Stochastic Automatic Gradient Tree Boosting is implemented as part of the gbt.train() function in the R-Package AGTBoost (Lunde, 2020), and is available from `http://github.com/eirikstad/agtboost`. The implementation has introduced several new arguments for the function. Most important it allows for input of a sampling rate or a list of sampling rates through the argument *sample_rate*. These sampling rates will be iterated through at training time. A parameter *step_type* is also introduced, and specifies how the iteration should be carried out. For step type value *"std"* it will per-

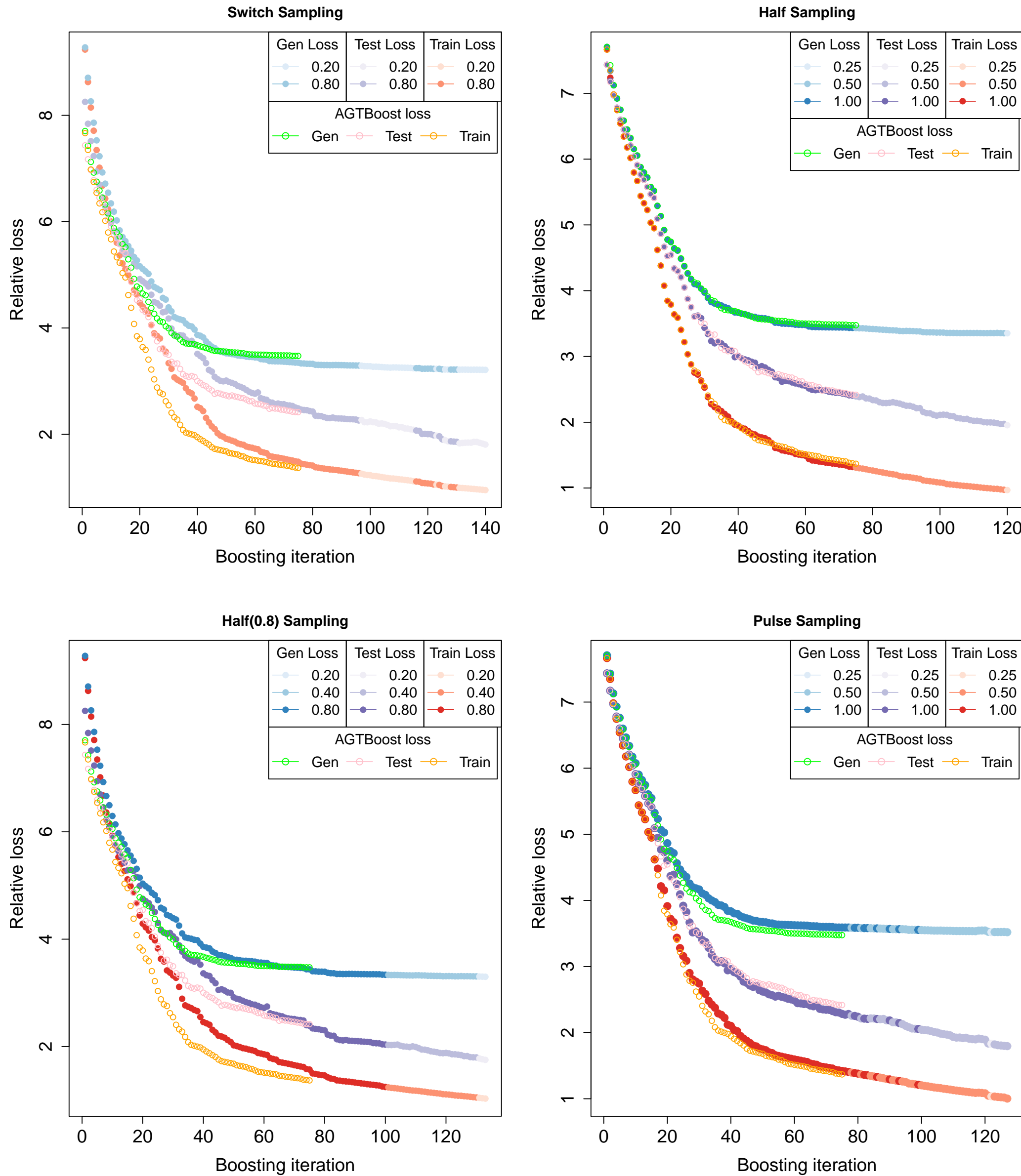

Figure 6.3: Train, test and estimated generalization loss at each boosting iteration for ensembles of each of the sampling schemes Switch, Half, Half(0.8) and Pulse, plotted next to standard AGTBoost. The brightness of the points indicates which sampling rate is used building the tree at the given boosting iteration.

Table 1: Sampling Arguments of gbt.train()

Argument	**Value**	**Function**	**Default**
sample_rate	vector of or single float $\in [0, 1)$	rate of row subsampling	1
step_type	"std", "pulse" or "repeat"	specifies how the sample rates are iterated through	"std"
max_max_no_red	positive int	specifies how many iterations are maximally allowed without improvement	10

Table 1: New arguments introduced to gbt.train() to facilitate sampling and the different sampling schemes tested.

Sampling Scheme	**sample_rate**	**step_type**
Random	999	"std"
Pulse	seq(1,0.1,by=-0.1)	"pulse"
Half	(1,0.5,0.25,0.125)	"std"
Half08	(0.8,0.4,0.2,0.1)	"std"
Switch	(1,0.1)	"repeat"

Table 2: Argument settings of the different sampling schemes tested.

form boosting at each sampling rate as long as the estimated generalization loss improves, and stop at the last sampling rate. For step type value *"pulse"*, the boosting will start over at the first sampling rate listed every time there is an improvement at another sampling rate. For step type value *"repeat"*, the boosting will start over at the first sampling rate after the last sampling rate as long as there was an improvement in the last round. The last parameter utilized in this paper is *max_max_no_red*, which specifies how many boosting iterations allowed with no reduction in generalization loss. For a given sampling rate, the maximal number of iterations is given by $max_max_no_red - max_max_no_red \cdot sample_rate^3$. This formula has the property that it sets an absolute maximum for number of iterations, while at the same time giving increasingly higher number of iterations for smaller sampling rates. In table 2, the argument settings of the sample schemes compared in this thesis are listed.

7 Experimental design

We now have a way of training AGTBoost models with subsampling with Stochastic AGTBoost. However, the sampling rate still needs to be manually tuned, as there is no one sampling rate that suits all datasets. We have introduced some ideas of sampling schemes that hopefully can eliminate the absolute need of manually tuning the sampling rate. In the next sections, we will test a handful of these sampling schemes on real and simulated datasets, to identify if any of those can be used as a default sampling scheme which can improve the AGTBoosting for all or at least a range of different datasets. All results are relative to the results of deterministic AGTBoost as of the R-package aGTBoost (Lunde, 2020), and in some cases we compare the results to the ones of XGBoost (Chen & Guestrin, 2016) and autoXGBoost (Thomas et al., 2018), with the two latter being restricted in terms of functionality not yet implemented in AGTBoost to get comparable results. This means that column subsampling are set to 1, and regularization methods such as L1- and L2-regularization are set to 0, such that it matches the current implementation of AGTBoost. We will do testing on a range of real and generated data sets as seen in Section 7.1 and Section 7.2.

7.1 Real Data

Minding the evaluability of our testing, we use datasets from Introduction to Statistical Learning (James et al., 2013) and Elements of Statistical Learning (Hastie et al., 2009), the same as seen in (Lunde et al., 2020). We also include the Higgs Dataset from (Baldi, Sadowski, & Whiteson, 2014). In total we have 7 datasets with continuous response variable, and 6 with a binary response variable. These are summarized in Table 3.

Name	dimensions	response type	train - test	Source Package
Boston	506 x 13	continous	50 - 50	MASS
Ozone	111 x 3	continous	50 - 50	EOSL
Auto	392 x 310	continous	50 - 50	ISLR
Carseats	400 x 11	continous	70 - 30	ISLR
College	777 x 17	continous	70 - 30	ISLR
Hitters	263 x 19	continous	70 - 30	ISLR
Wage	3000 x 25	continous	70 - 30	ISLR
Caravan	5822 x 85	binary	70 - 30	ISLR
Default	10000 x 3	binary	70 - 30	ISLR
OJ	1070 x 17	binary	70 - 30	ISLR
Smarket	1250 x 6	binary	70 - 30	ISLR
Weekly	1089 x 6	binary	70 - 30	ISLR
HIGGS	12500 x 30	binary	50 - 50	

Table 3: All datasets from the books (Hastie et al., 2009) and (James et al., 2013), their number of observations and covariates, the type of the response variable, the train and test split proportion and the source. The dimensions are after using the R function model.matrix, which performs one-hot encoding and removes rows containing NA values.

7.2 Simulated Data

In addition to these datasets, we will also generate some datasets with the specified properties as seen in Table 4. By generating data, we can adjust the size of the data, the number of explanatory variables, the number of those explanatory variables that are dependent on the response variable, and also add effects such as interaction effects between variables and adjust how each variable on its own affects the response variable. We will try and generate a wide range of datasets with different properties so that the model testing is done on datasets as diverse as possible. We separate the datasets by 6 parameters. Number of rows or observations in the dataset, the number of columns or covariates, the number of dependent variables affecting the response variables, whether there should be interaction effects between the dependent variables

Set	Dimensions	Dependent variables	Interactions	Polynomial degree
1	200 x 100	100	FALSE	0
2	200 x 100	100	TRUE	3
3	200 x 10	5	FALSE	0
4	200 x 10	5	TRUE	3
5	200 x 100	1	FALSE	0
6	1000 x 100	100	FALSE	0
7	1000 x 100	100	TRUE	3
8	1000 x 100	10	FALSE	0
9	1000 x 100	10	TRUE	3
10	1000 x 100	1	FALSE	0
11	200 x 250	250	FALSE	0
12	200 x 250	250	TRUE	3

Table 4: All simulated datasets, their number of observations and columns, the number of columns affecting the response variable, boolean indicating whether there are interaction effects between the dependent variables and the polynomial degree of the variables

and if there should be a polynomial dependency between the dependent variables and the response variable. When generating the datasets, we calculate a mean of the response variables to be a function of the dependent variables, and then draw the response variables from a normal distribution with the given mean and a standard deviation given by the standard deviation of the means times a factor of 0.3. One can see the full set of artificial datasets in Table 4

In each dataset, the explanatory variables are generated from a normal distribution with mean 0 and standard deviation 100, and with the given number of rows and column. Then, the first a columns, a being the number of dependent variables for the given dataset, are summed, divided by their standard deviation and multiplied by 100 and added to the response mean. If interaction effects are specified, the first column is multiplied with each of the other dependent variables. The products are summed, divided by their standard deviation and multiplied by 100 and added to the response mean. If

the polynomial are specified to larger than 1, each of the dependent variables are multiplied with themselves as many times as specified, and this is added to the response mean in the same way as earlier. The response variables are then generated from a normal distribution with the the response mean and standard deviation as specified.

8 Experimental results

In this section, we apply our model with some selected sampling schemes to the aforementioned data, and see how they perform compared to standard deterministic aGTBoost, XGBoost and autoXGBoost.

8.1 One rate subsampling

The first stochastic AGTBoost algorithm we test is the one-rate subsampling algorithm. Here, a sampling rate is selected pre training, and all trees are trained on a subsample of the training set with size corresponding to the selected sampling rate. For this thesis, we train models for samples rates between 0.1 and 1, with an increment of 0.1, and compare each sampling rate for each dataset. There is no reason to believe strongly that the optimal constant sampling rate for any dataset lies in this set of sampling rates, but it is a wide range spanning from a low sampling rate of 0.1 and the deterministic model with a sampling rate of 1. Thus, this set is suited to investigate the effect of increasing and decreasing the sampling rate for the given dataset. Further, there is no reason to believe that the same sample size will be efficient for all types of dataset, thus the approach with a constant sampling rate will typically favor tuning of the sampling rate before training a final model. This is also the reason why we have introduced sampling schemes in this thesis.

As we suspected, we observe from Figure 8.2 and Figure 8.1 that the most beneficial sampling rate varies over both the real and the simulated datasets. For some dataset, the best sampling rate seems to be around 0.2, while some datasets need substantially higher sampling rates to even beat deterministic AGTBoost. The smallest sampling rate tested, 0.1 is in no case the best performing. We notice a trend over most datasets, that there seems to be

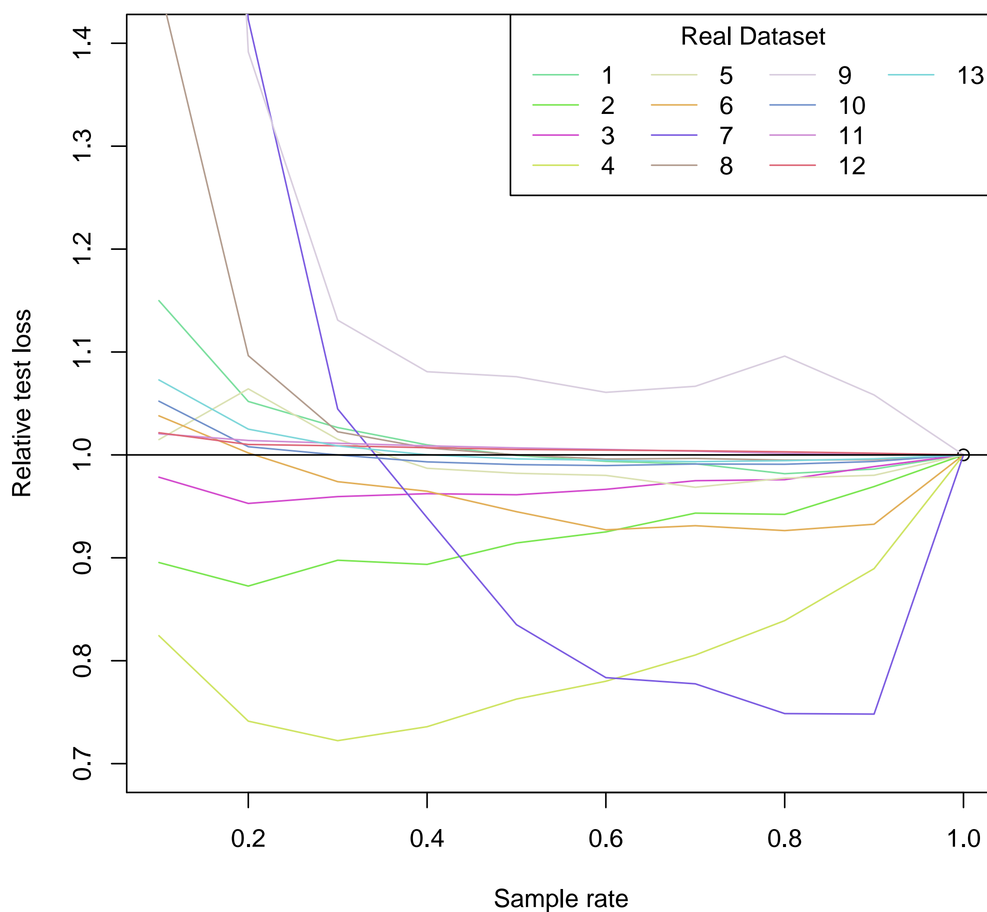

Figure 8.1: Average test loss for AGTBoost trained with constant sampling rates relative to deterministic AGTBoost trained on the same datasets with the same train/test splits. The datasets and loss functions are in accordance with Table 3. For each dataset, we train 100 models on different splits of the datasets for each of the sampling rates 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 and 1 and the test loss is averaged over the different splits. The black line at $y = 1$ represents the performance of deterministic AGTBoost.

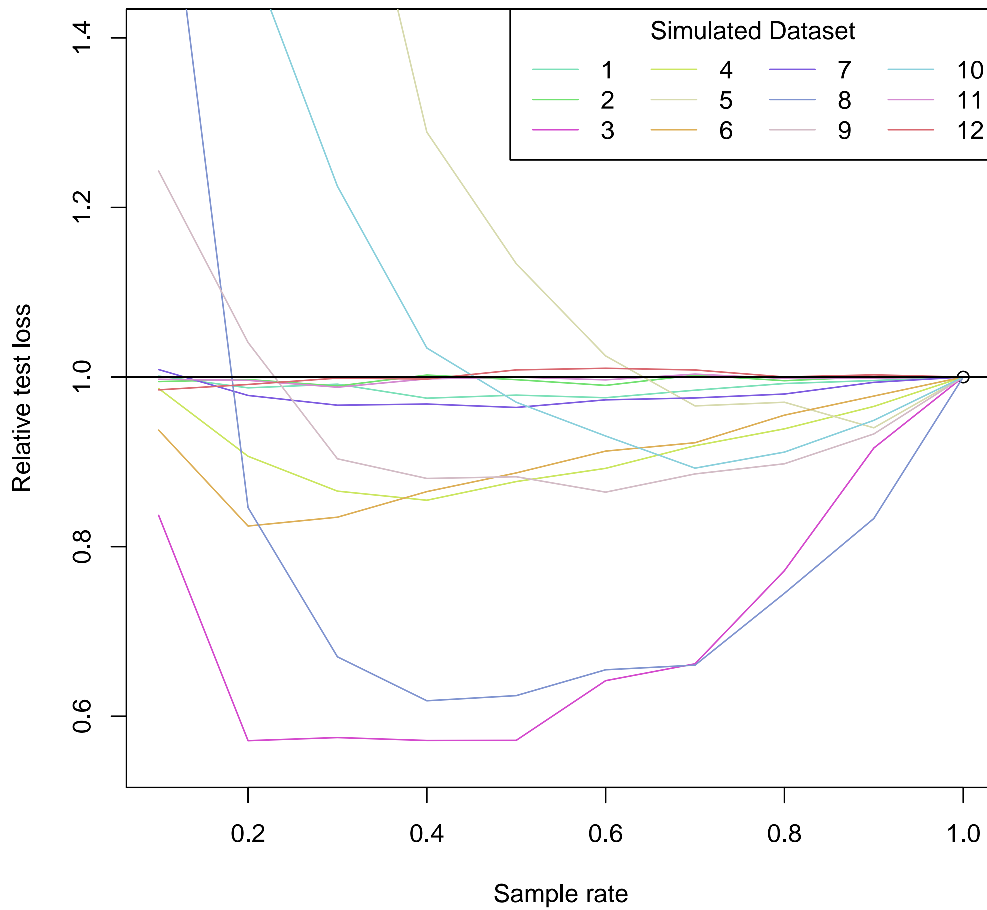

Figure 8.2: Comparison of sampling schemes for simulated datasets corresponding to Figure 8.1. Datasets and loss functions can be seen in Table 4

a drastic improvement in increasing sampling rate until some optimal point, before the performance decrease at a slow rate as it approaches 1.

8.2 Sampling schemes

In the next section, we will present results from models trained with the different sampling schemes presented in Section 6.5. Common for these are that we define some rule that decides at each boosting iteration the sampling rate used for that particular iteration, and that this sampling rate varies over the iterations. The motivation behind introducing sampling schemes is what we know from the last section, that the optimal constant sampling rate varies over different dataset. We therefore hope by varying the sampling rate over the boosting iteration that we can find a scheme that performs well compared to a constant sampling rate over a range of datasets, and in this way eliminate the need for manual tuning of the sampling rate. We will present the sampling schemes in an aggregated manner, so for details of any particular sampling scheme we refer to Section 6.5.

We test the sampling schemes in the same way as we did for constant sampling rates, by training 100 models for each scheme on each of the real and simulated datasets and comparing the test loss to that of deterministic AGTBoost.

Testing the sampling schemes on real data, we find that apart from the random scheme, there are on average only small differences between the schemes in terms of prediction accuracy. However, differences between the sampling schemes for the single data set vary more. It is hard to recognize any sampling scheme that overall outperforms the others, but we notice that all of the non-random sampling schemes performs on par or better than deterministic

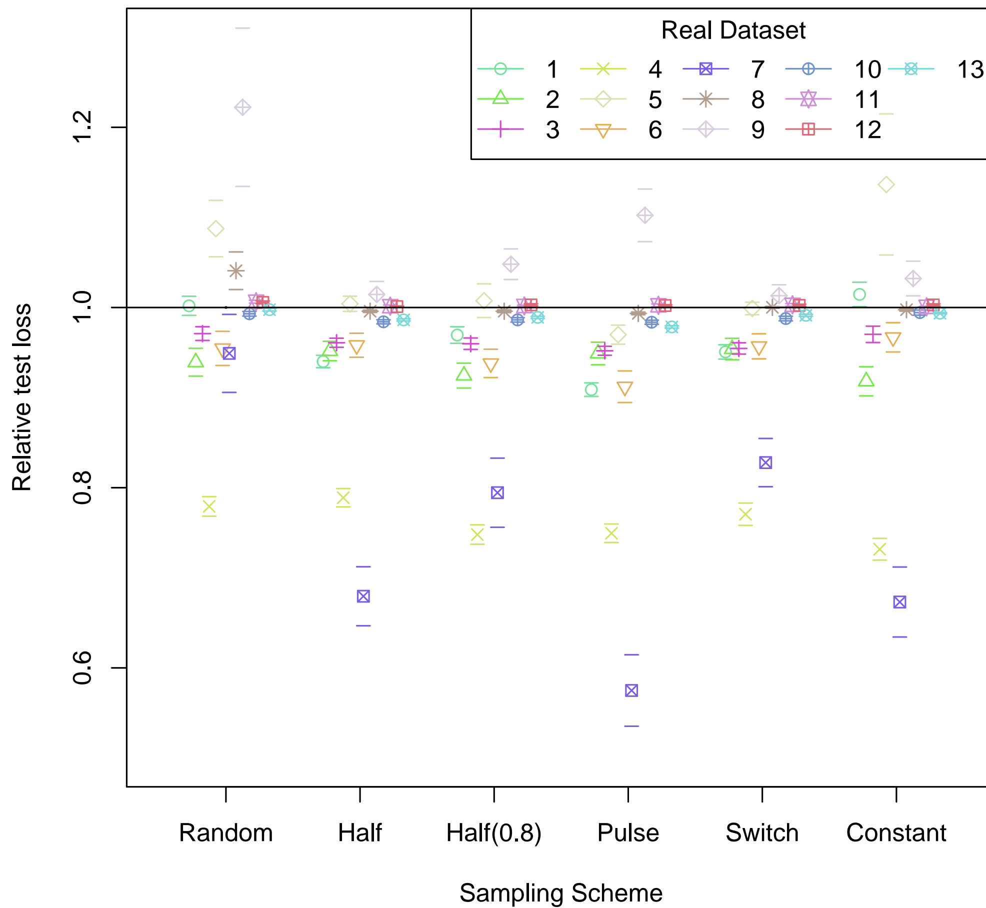

Figure 8.3: Comparison of sampling schemes over real data. Averaged test loss over the 13 datasets from Table 3 relative to AGTBoost for models trained with sampling schemes. The constant scheme represent the best constant sampling rate for each dataset, selected for each split by cross validation over the training set. For each dataset and sampling scheme combination, 100 stochastic models are trained on 100 different train/test splits, and compared with a deterministic model trained on the same splits. The mean test loss is averaged over the 100 training iterations, with a 95% mean confidence interval marked by same color lines.

AGTBoost for the vast majority of datasets. In the case of real datasets, it seems that tuning a constant sampling rate is favorable to using any of the specified sampling schemes in terms of performance, even though for set 1 and 5 it is possible to dramatically increase performance by using a sampling scheme instead of a constant sampling rate. We notice that for set 7, which is a large data where we know it to be beneficial with a sampling rate around 0.9, the schemes Half 0.8, and switch seems to do substantially worse than the constant sampling rate and the other sampling schemes.

Inspecting the results from the simulated data, we find that both Pulse and Switch tends to do marginally better than the other sampling schemes and the constant sampling rate, and may based on this seem like a preferred choice for a sampling scheme. Interestingly, on the simulated data, not starting the Half scheme with a deterministic sampling rate of 1 seems to be beneficial over all but set 7, where there are only a small advantage of the Half sampling scheme compared to Half(08). This result is not however consistent over the real data sets, where if some, the advantage is the other way around. We notice that in all cases, all sampling schemes apart from Random performs on par or better than the deterministic AGTBoost, as well as the constant sampling rate.

We are not however only interested in the average performance, but also the consistency over multiple trainings as noted in Section 6.3. In Figure 8.5, we have plotted the relative test loss of 50 models of each sampling scheme trained on the same split of the data sets in Table 3. From this we can observe the variance of the performance of all of our sampling schemes induced by the sampling procedure itself. Also here the results of a selected best constant sampling rate is plotted. The best constant sampling rate is selected by cross validation over the training sets for real data, and by separate simulated datasets for the simulated data. We have from Figure 8.3 that the best

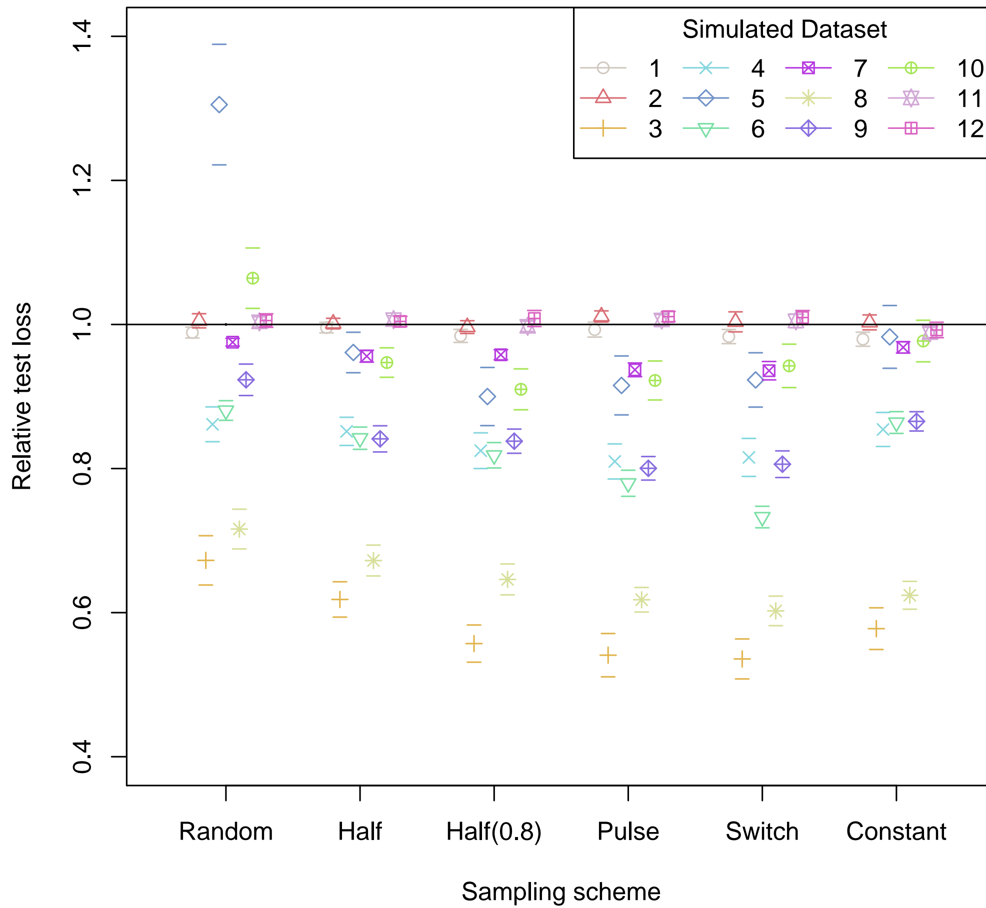

Figure 8.4: Comparison of sampling schemes over simulated data corresponding to Figure 8.3. The constant scheme represent the best constant sampling rate for each dataset, selected by cross validation over separate simulated datasets with the same distribution as the ones used for testing. For each dataset and sampling scheme combination, 50 stochastic models are trained on 50 different train/test splits, and compared with a deterministic model trained on the same split.

constant model performs descent compared to the sampling schemes, but we notice here that it is relatively high in variance. This may to some degree be reduced by limiting the set of possible sampling rates to rates close to 1, but at the cost of performance. It may be that it still would be preferred to any of the sampling schemes, but one also has to consider that this requires tuning to find the optimal constant rate. We also notice that the pulse scheme is the most consistent, way more so than constant or random sampling rates. What goes for the last three schemes, Half(1) and Switch seems slightly more consistent than Half(0.8), but all three of them are worse of than Pulse. The same pattern is seen in Figure 8.6, if we ignore the last two datasets. The pulse scheme is consistently the most consistent sampling scheme, with random and constant producing the least constant results.

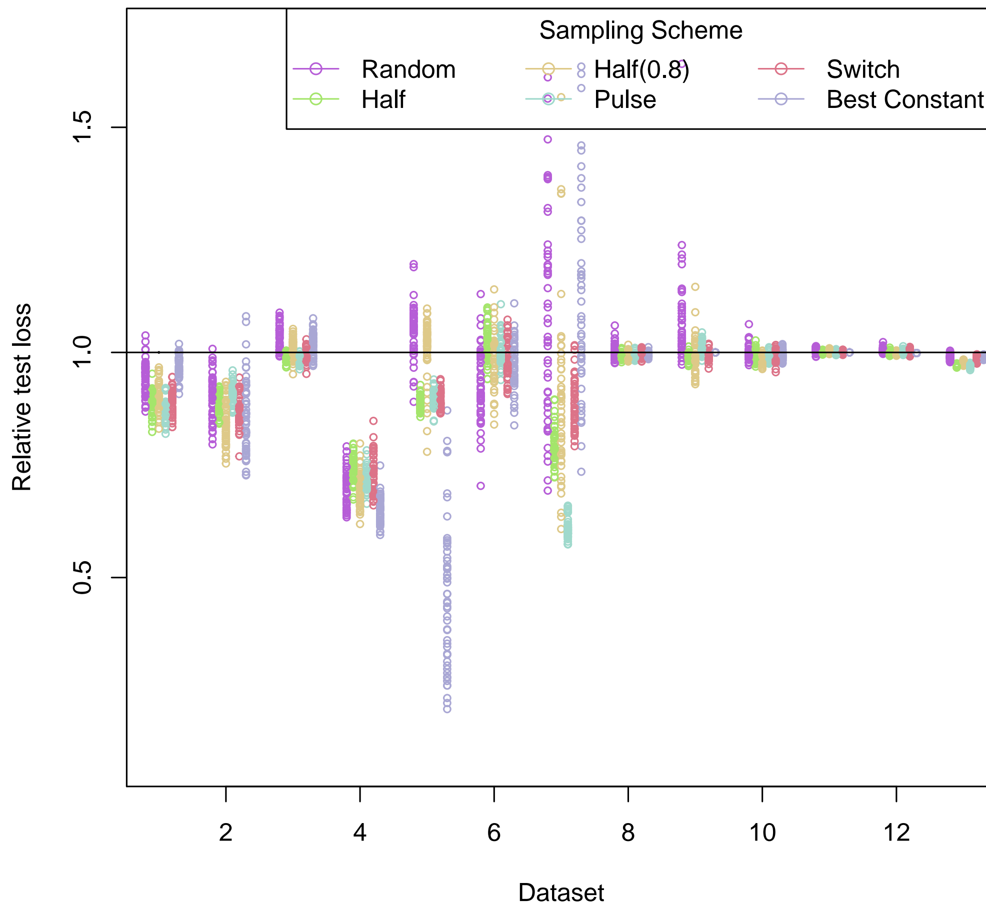

Figure 8.5: Mean squared test error for Stochastic AGTboost models for each of the five listed sampling schemes relative with deterministic AGTBoost, trained 50 times on each of the datasets listed in Table 3 for each of the different sampling schemes listed. Each training on a given dataset is done with the exact same train/test-split. The constant scheme represents the best constant sampling rate for each dataset, selected independently for each dataset by cross validation on the training data of the same split on which the models are tested.

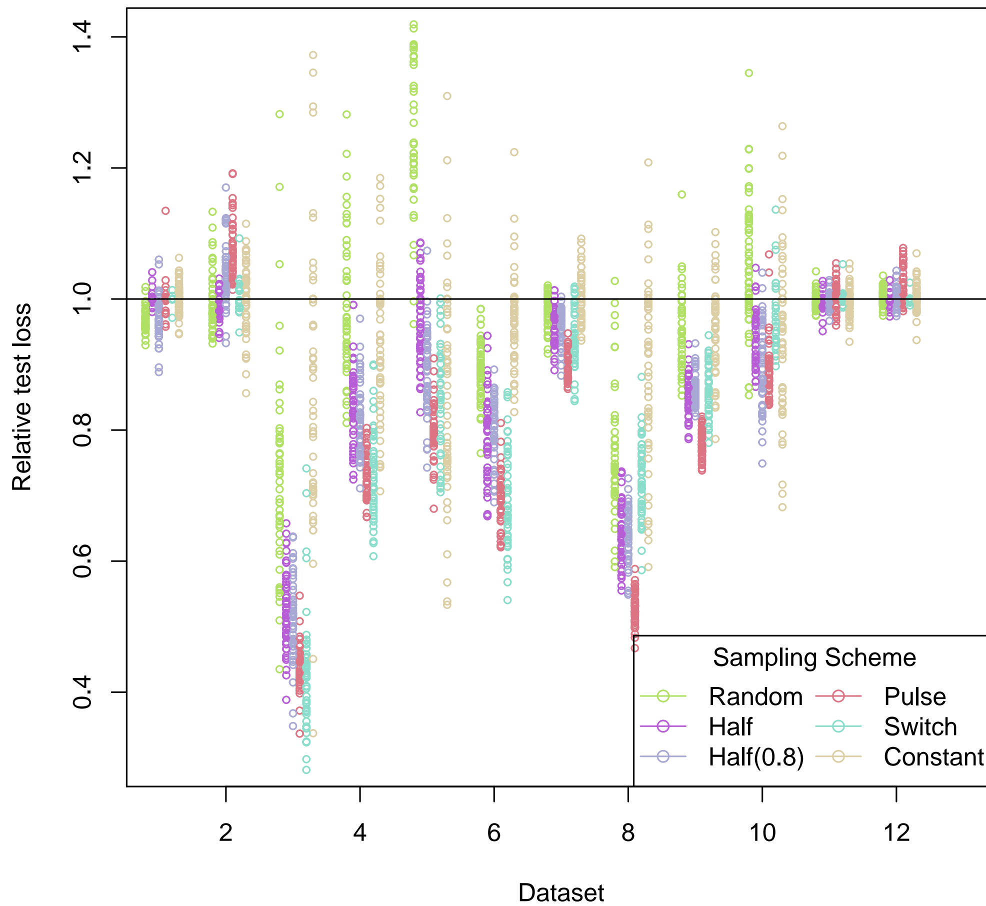

Figure 8.6: Average test mean squared error for AGTBoost trained with different sampling schemes relative to deterministic AGTBoost trained on the same datasets with the same train/test splits. For each dataset, we train 100 models on different splits of the datasets for each of the sample schemes Random, Half(1), Half(0.8), Pulse and Switch, and the test loss is averaged over the different splits.

Next, we compare our sampling schemes to XGBoost (Chen & Guestrin, 2016) and AutoXGBoost(AXGB) (Thomas et al., 2018). AXGB is an automated gradient tree boosting method, where XGBoost is tuned automatically by Bayesian optimization, a procedure where different combinations of algorithms and parameters setting are sequentially evaluated in a guided manner such that it finds a minimum in relatively few iterations (Snoek et al., 2012). However, this process still takes time, but one can stop the process at any point. For each split on which we train AXGB, we allow this optimization to go on from 10 to 100 times the time it takes to train the pulse scheme. For XGBoost, we tune both the number of trees and the sampling rate by cross validation, and for both of the mentioned methods we allow no L1 or L2 regularization or column subsampling.

We see from Figure 8.7 that XGBoost outperforms stochastic AGTBoost for all sampling schemes on most of the real datasets used. For AXGB, the picture is somewhat different and inconclusive in terms of performance relative with AGTBoost.

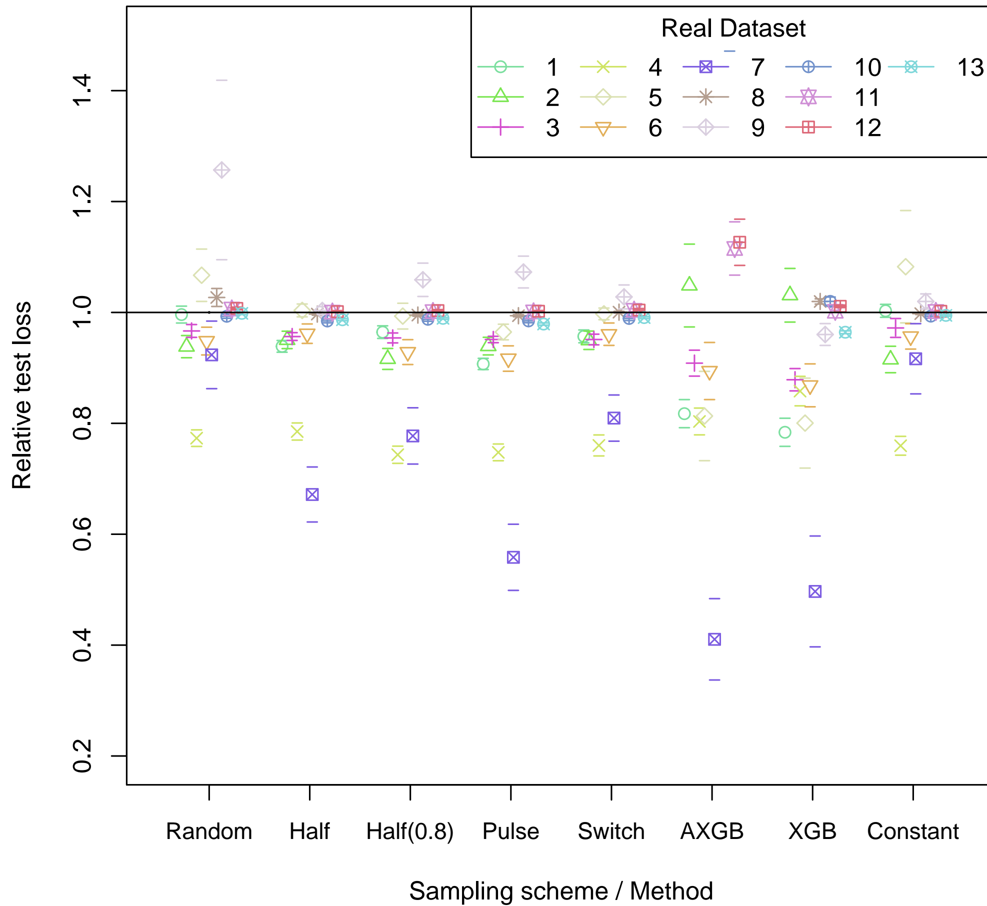

Figure 8.7: Average test mean squared error for Stochastic AGTBoost, XGBoost(XGB) and AXGB, trained with different sampling schemes relative to deterministic AGTBoost trained on the same datasets with the same train/test splits. For each dataset, we train 50 models on different splits of the datasets for each of the sample schemes Random, Half(1), Half(0.8), Pulse and Switch, and the test loss is averaged over the different splits. AXGB is on each split of each set allowed to train from 10 to 100 times as long as the slowest AGTBoost scheme dependent on the dataset.

9 Discussion

With the motivation of improving the AGTBoost algorithm by introducing stochastic row subsampling, this thesis has explored Stochastic Gradient Tree Boosting. We have shown that one can extend the information criterion from (Lunde et al., 2020) for estimating generalization loss reduction in a Gradient Tree Boosting setting to work in the stochastic setting with row subsampling. As we hoped for, we have seen that the implementation of stochastic row subsampling at each boosting iteration has also in general improved on the predictive power of AGTBoost.

The implementation of subsampling proposed allows for custom selection of sampling rate. This may be set constant over all boosting iterations, or to vary given self-defined parameters. We have tested the algorithm on a set of such sampling schemes in addition to constant sampling rates. The results indicates that a constant sampling rate allows for improvement in most cases, and our implementation include the option to set a constant self-selected sampling rate between 0 and 1. We find that overall, setting an arbitrary constant sampling rate below 1 and above 0.5 is likely to outperform the deterministic method with sampling rate equal to 1. However, we find that using a constant sampling rate favors manual tuning, as the optimal sampling rate varies over different data sets. As the sampling rate gets smaller, the difference in performance between data sets increases, and where our method with constant sampling rate for some data sets even favors sampling rates below 0.2, such low sampling rates in many cases are outperformed by the deterministic AGTBoost.

With this in mind, and recalling the philosophy behind AGTBoost, namely to have an automatic method free of manual tuning, we sought to find a way of automating the selection of sampling rate. This was not achieved for the con-

stant sampling rate, but we implemented the possibility of predefined sampling schemes, and proposed a selection of schemes, some that seems to work well over a range of data sets. We find that in general, a good sampling scheme strategy is to start at a higher sampling rate and decrease as the boosting progress. In the process we have tested sampling schemes with other strategies such as increasing sampling rates, and although some excluded schemes performed well for selected data sets, the clear pattern is that a decreasing sampling rate in most cases outperform most constant sampling rates, increasing sampling rates, and also seems to perform stable over different data sets. The problem of finding an optimal sampling rate or sampling scheme or strategy remains unsolved, and it is unsatisfying that even between the proposed sampling schemes the comparison is indecisive. Our selection of sampling schemes are limited, and it is an open problem to improve on the sampling schemes.

All of the proposed sampling schemes except for the random one, bases the sampling rate or change in sampling rate on the reduction in generalization loss over the last iterations. One possible sampling strategy that we have not tested is to let the sampling rate change systematically but without consideration of the reduction of generalization loss in the last iterations. Another possible improvement could be to let the absolute value of the sampling rate depend on this development of the estimated generalization loss over boosting iterations, or on properties of the data set.

It is theoretically trivial to extend the Stochastic AGTBoost to include column subsampling, although one can not expect this to have the same impact on predictive power as row subsampling.

Unfortunately, the testing against XGBoost and AutoXGBoost was limited due to the computational demands of tuning these two methods. That is why the testing of these are limited to only 10 splits of the included datasets.

The philosophy behind AGTBoost is as the name indicates to have a fully Automatic Gradient Tree Boosting method, and the inclusion of subsampling in terms of constant sampling rate does favor manual tuning. However, we have introduced sampling schemes in the gradient tree boosting setting, and seen that this safely can be deployed as a tuning-free stochastic variant of AGTBoost which cause overall improvement of the method without any need of manual tuning. However, finding an optimal or better sampling scheme remains an unsolved task.

References

Akaike, H. (1998). Information theory and an extension of the maximum likelihood principle. In *Selected papers of hirotugu akaike* (pp. 199–213). Springer.

Anderson, D., & Burnham, K. (2004). Model selection and multi-model inference. *Second. NY: Springer-Verlag*, *63*(2020), 10.

Baldi, P., Sadowski, P., & Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, *5*(1), 1–9.

Bartlett, P., Freund, Y., Lee, W. S., & Schapire, R. E. (1998). Boosting the margin: A new explanation for the effectiveness of voting methods. *The annals of statistics*, *26*(5), 1651–1686.

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

Breiman, L. (1996b). *Bias, variance, and arcing classifiers* (Tech. Rep.). Tech. Rep. 460, Statistics Department, University of California, Berkeley ....

Breiman, L. (1997). *Arcing the edge* (Tech. Rep.). Technical Report 486, Statistics Department, University of California at ....

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

Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (pp. 785–794).

Cox, J., Ingersoll, J., & Ross, S. (1985, 02). A theory of the term structure of interest rates. *Econometrica*, *53*, 385-407. doi: 10.2307/1911242

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and ran-

domization. *Machine learning*, *40*(2), 139–157.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., & Hutter, F. (2015). Efficient and robust automated machine learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 28). Curran Associates, Inc. Retrieved from `https://proceedings.neurips.cc/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf`

Freund, Y., Schapire, R. E., et al. (1996). Experiments with a new boosting algorithm. In *icml* (Vol. 96, pp. 148–156).

Friedman, J. H. (1999). *Greedy function approximation: a gradient boosting machine. techinical report, dept. of statistics.* Stanford University.

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational statistics & data analysis*, *38*(4), 367–378.

Gombay, E., & Horvath, L. (1990). Asymptotic distributions of maximum likelihood tests for change in the mean. *Biometrika*, *77*(2), 411–414.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, *11*(1), 10–18.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: data mining, inference, and prediction.* Springer Science & Business Media.

Huber, P. J., et al. (1967). The behavior of maximum likelihood estimates under nonstandard conditions. In *Proceedings of the fifth berkeley symposium on mathematical statistics and probability* (Vol. 1, pp. 221–233).

Hyafil, L., & Rivest, R. (1976). Constructing optimal binary search trees is np complete. *Information Processing Letters*.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction*

to statistical learning (Vol. 112). Springer.

John, G. H., & Langley, P. (1996). Static versus dynamic sampling for data mining. In *Kdd* (Vol. 96, pp. 367–370).

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, *30*, 3146–3154.

Lazarevic, A., & Obradovic, Z. (2001). Data reduction using multiple models integration. In *European conference on principles of data mining and knowledge discovery* (pp. 301–313).

Lunde, B. (2020). agtboost: Adaptive and automatic gradient boosting computations [Computer software manual]. (R package version 4.1-15)

Lunde, B. Å. S., & Kleppe, T. S. (2020). agtboost: Adaptive and automatic gradient tree boosting computations. *arXiv preprint arXiv:2008.12625*.

Lunde, B. Å. S., Kleppe, T. S., & Skaug, H. J. (2020). An information criterion for automatic gradient tree boosting. *arXiv preprint arXiv:2008.05926*.

Mason, L., Baxter, J., Bartlett, P., & Frean, M. (1999). Boosting algorithms as gradient descent in function space. In *Proc. nips* (Vol. 12, pp. 512–518).

Meir, R., & Rätsch, G. (2003). An introduction to boosting and leveraging. In *Advanced lectures on machine learning* (pp. 118–183). Springer.

Milborrow, S. (2021). Plot 'rpart' models: An enhanced version of 'plot.rpart' [Computer software manual]. (R package version 3.1.0)

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... others (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, *12*, 2825–2830.

Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2017). Catboost: unbiased boosting with categorical features. *arXiv preprint arXiv:1706.09516*.

Provost, F., Jensen, D., & Oates, T. (1999). Efficient progressive sampling. In *Proceedings of the fifth acm sigkdd international conference on knowledge discovery and data mining* (pp. 23–32).

Reyzin, L., & Schapire, R. E. (2006). How boosting the margin can also boost classifier complexity. In *Proceedings of the 23rd international conference on machine learning* (pp. 753–760).

Ridgeway, G. (1999). The state of boosting. *Computing science and statistics*, 172–181.

Sadid, M. W. H., Mondal, M. N. I., Alam, M. S., Sohail, A. S. M., & Ahmed, B. (2004). Progressive boosting for classifier committee learning. In *Asian applied computing conference* (pp. 52–58).

Schapire, R. E. (1990). The strength of weak learnability. *Machine learning*, *5*(2), 197–227.

Schwarz, G. (1978). Estimating the dimension of a model. *The annals of statistics*, 461–464.

Seni, G., & Elder, J. F. (2010). Ensemble methods in data mining: improving accuracy through combining predictions. *Synthesis lectures on data mining and knowledge discovery*, *2*(1), 1–126.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, *25*.

Terry Therneau, B. R., Beth Atkinson. (2019). Recursive partitioning and regression trees [Computer software manual]. (R package version 0.9.1)

Thomas, J., Coors, S., & Bischl, B. (2018). Automatic gradient boosting. *arXiv preprint arXiv:1807.03873*.

Thornton, C., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2013). Auto-weka: Combined selection and hyperparameter optimization of classifi-

cation algorithms. In *Proceedings of the 19th acm sigkdd international conference on knowledge discovery and data mining* (pp. 847–855).

Wilson, D. R., & Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural networks*, *16*(10), 1429–1451.