NTNU

Norwegian University of
Science and Technology

Tree Boosting With XGBoost

Why Does XGBoost Win "Every" Machine
Learning Competition?

Didrik Nielsen

Master of Science in Physics and Mathematics
Submission date: December 2016
Supervisor: Håvard Rue, IMF

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

Tree boosting has empirically proven to be a highly effective approach to predictive modeling. It has shown remarkable results for a vast array of problems. For many years, MART has been the tree boosting method of choice. More recently, a tree boosting method known as XGBoost has gained popularity by winning numerous machine learning competitions.

In this thesis, we will investigate how XGBoost differs from the more traditional MART. We will show that XGBoost employs a boosting algorithm which we will term Newton boosting. This boosting algorithm will further be compared with the gradient boosting algorithm that MART employs. Moreover, we will discuss the regularization techniques that these methods offer and the effect these have on the models.

In addition to this, we will attempt to answer the question of why XGBoost seems to win so many competitions. To do this, we will provide some arguments for why tree boosting, and in particular XGBoost, seems to be such a highly effective and versatile approach to predictive modeling. The core argument is that tree boosting can be seen to adaptively determine the local neighbourhoods of the model. Tree boosting can thus be seen to take the bias-variance tradeoff into consideration during model fitting. XGBoost further introduces some subtle improvements which allows it to deal with the bias-variance tradeoff even more carefully.

Sammendrag

“Tree boosting” har empirisk vist seg å være en svært effektiv tilnærming til prediktiv modellering. Denne tilnærmingen har vist meget gode resultater for en lang rekke problemer. I mange år har MART vært den foretrukne “tree boosting”-metoden. Mer nylig har en “tree boosting”-metode ved navn XGBoost økt i popularitet ved å vinne en rekke maskinlæringskonkurranser.

I denne oppgaven skal vi undersøke hvordan XGBoost skiller seg fra den mer tradisjonelle MART. Vi vil vise at XGBoost benytter en “boosting”-algoritme som vi vil kalle “Newton boosting”. Denne “boosting”-algoritmen vil ytterligere bli sammenlignet med “gradient boosting”-algoritmen som MART benytter. Videre vil vi diskutere teknikker for regularisering som disse metodene tilbyr og effekten disse har på modellene.

I tillegg til dette vil vi forsøke å svare på spørsmålet om hvorfor XGBoost ser ut til å vinne så mange konkurranser. For å gjøre dette vil vi gi noen argumenter for hvorfor “tree boosting”, og særlig XGBoost, ser ut til å være en såpass effektiv og allsidig tilnærming til prediktiv modellering. Hovedargumentet er at “Tree boosting” kan sees å bestemme lokale områder i modellen adaptivt. “Tree boosting” kan dermed sees å ta hensyn til “the bias-variance tradeoff” under modelltilpasningen. XGBoost introduserer ytterligere noen små forbedringer som gjør at den kan håndtere “the bias-variance tradeoff” enda mer nøye.

Preface

This thesis concludes my master's degree in Industrial Mathematics at the Norwegian University of Science and Technology (NTNU). It was written at the Department of Mathematical Sciences during the Autumn of 2016 under the supervision of Professor Håvard Rue.

I would like to thank Professor Håvard Rue for his excellent guidance and assistance throughout the semester. Moreover, I would like to thank him for persuading me to pursue further academic work and for getting me in contact with Mohammad Emtiyaz Khan at the newly established RIKEN Center for Advanced Integrated Intelligence Research in Tokyo, where I will start working as a research assistant in 2017.

Contents

Chapter 1

Introduction

Gradient boosting is a powerful machine learning technique introduced by Friedman (2001). The technique was motivated as being a gradient descent method in function space, capable of fitting generic nonparametric predictive models. Gradient boosting has been particularly successful when applied to tree models, in which case it fits additive tree models. Friedman devised a special enhancement for this case (Friedman, 2001, 2002). We will refer to this method as MART (Multiple Additive Regression Trees), but it is also known as GBRT (Gradient Boosted Regression Trees) and GBM (Gradient Boosting Machine). More recently, a new tree boosting method has come to stage and quickly gained popularity. It goes by the name XGBoost (Chen and Guestrin, 2016), and while it is conceptually similar to Friedmans tree boosting method MART, it also differs in multiple ways.

Gradient boosting has empirically proven itself to be highly effective for a vast array of classification and regression problems. One arena where this becomes particularly apparent is the competitive machine learning scene. The foremost example of this would be *Kaggle*, a platform for data science competitions and the worlds largest community of data scientists[1]. Kaggle has on several occasions interviewed top ranking members on Kaggle, i.e. members that have achieved top scores in multiple competitions. Four members who has ranked as number one were asked the question "what are your favorite machine learning algorithms?" Their responses serve as a testemony to the effectiveness and versatility of gradient boosting of trees:

"*Again it depends on the problem, but if I have to pick one, then it is GBM (its XGBoost flavor). It is amazing how well it works in many different problems. Also I am a big fan of online SGD, factorization, neural networks, each for particular types of problems.*"

— Owen Zhang[2]

[1] *http://www.kaggle.com/about*

[2] *http://blog.kaggle.com/2015/06/22/profiling-top-kagglers-owen-zhang-currently-1-in-the-world/*

"*Gradient Boosting Machines are the best! It is amazing how GBM can deal with data at a high level of depth. And some details in an algorithm can lead to a very good generalization. GBM is great dealing with linear and non-linear features, also it can handle dense or sparse data. So it's a very good algorithm to understand core parameters and it's always a good choice if you are not sure what algorithm to use. Before I knew of GBM, I was a big fan of neural networks.*"

— Gilberto Titericz[3]

"*I like Gradient Boosting and Tree methods in general: [as they are] Scalable, Non-linear and can capture deep interactions, Less prone to outliers.*"

— Marios Michailidis[4]

"*Gradient boosted trees by far! I like GBT because it gives pretty good results right off the bat. Look at how many competitions are won using them!*"

— Lucas Eustaquio Gomes da Silva[5]

Since its introduction in 2014, XGBoost has quickly become among the most popular methods used on Kaggle. It has accumulated an impressive track record of winning competitions. For example, among the 29 challenge winning solutions posted on Kaggle during 2015, 17 used XGBoost (Chen and Guestrin, 2016). For comparison, the second most popular method was deep neural networks, which was used in 11 solutions. XGBoost has also demonstrated its effectiveness at the KDD Cup, a prestigous competition held yearly. At the KDD Cup 2015, all of the top 10 solutions used XGBoost.

In this thesis, we will determine how XGBoost differs from the more traditional MART and further attempt to understand why XGBoost wins so many competitions. To understand this, we need to understand both why tree boosting is so effective in general, but also how XGBoost differs and thus why it might be even more effective in some cases.

We will show how the boosting algorithms employed by MART and XGBoost are different. We will develop both boosting algorithms as numerical optimization methods in function space. Whereas the gradient boosting algorithm employed by MART is well known for its interpretation as a gradient descent method in function space, we will show that the boosting algorithm employed by XGBoost can be interpreted as a Newton method in function space. We hence name this form of boosting *Newton boosting*. We will further compare the properties of the Newton boosting algorithm of XGBoost with the gradient boosting algorithm which is employed by MART. Following this will be a discussion on the most common regularization techniques for additive tree models which are utilized by MART and XGBoost, with emphasis on those found only in XGBoost. Finally, we will provide

[3] *http://blog.kaggle.com/2015/11/09/profiling-top-kagglers-gilberto-titericz-new-1-in-the-world/*

[4] *http://blog.kaggle.com/2016/02/10/profiling-top-kagglers-kazanova-new-1-in-the-world/*

[5] *http://blog.kaggle.com/2016/02/22/profiling-top-kagglers-leustagos-current-7-highest-1/*

some informal arguments for why tree boosting, and in particular XGBoost, seems to be such an unreasonably effective approach to predictive modeling.

This thesis is divided into three parts. The first covers the basics of statistical learning. We here review the core concepts necessary for understanding supervised learning methods and in particular tree boosting methods. We further provide a brief overview of some commonly used loss functions and supervised learning methods. The second part introduces tree boosting. Here, we will introduce boosting and its interpretation as numerical optimization in function space. We will further discuss tree methods and introduce the core elements of the tree boosting methods MART and XGBoost. In the third and final part, we will compare the properties of the tree boosting algorithms employed by MART and XGBoost, discuss the regularization techniques they use and, finally, provide arguments for why tree boosting in general and XGBoost in particular seems to be so effective and versatile.

Part I

Statistical Learning

Chapter 2

Supervised Learning

The term ”learning” is closely related to generalization. The goal in statistical learning is to find patterns in data that will generalize well to new, unobserved data. If one is able to find patterns that generalize well, one can make accurate predictions. This is indeed the goal in *supervised learning*, the part of statistical learning concerned with establishing the relationship between a response variable Y and a set of predictor variables X. *Unsupervised learning*, on the other hand, is concerned with finding patterns in data where there is *no* predefined response variable Y and the goal is to find structure in the set of variables X. In this chapter we will discuss some core concepts in supervised learning which provides the basis needed for discussing tree boosting.

2.1 The Supervised Learning Task

Shmueli (2010) discusses the distinction between *explanatory modeling* and *predictive modeling*. In explanatory modeling we are interested in understanding the causal relationship between X and Y, whereas in predictive modeling we are interested in predicting Y and our primary interest in the predictors X is to aid us in this goal. In this thesis, we will concern ourself with predictive modeling.

Assume that we are interested in building a model for predicting the response variable $Y \in \mathcal{Y}$ using a set of covariates $X = (X_1, ..., X_p) \in \mathcal{X}$. Assume further that we have a data set at our disposal to solve the task at hand. The data set

$$\mathcal{D} = \{(Y_1, X_1), (Y_2, X_2), ..., (Y_n, X_n)\}$$

is assumed to be an i.i.d. sample of size n from a joint distribution $\mathbb{P}_{Y,X}$.

The response $Y \in \mathcal{Y}$ is also referred to as the *dependent variable* or the output variable. When $Y \in \mathcal{Y}$ can only take on a finite number of values or classes, i.e. $|\mathcal{Y}|$ is finite, we are dealing with a classification task. Otherwise, we are dealing with a regression task.

The covariates $X = (X_1, ..., X_p) \in \mathcal{X}$ are also referred to as the *predictors*, the *explanatory variables*, the *features*, the *attributes* the *independent variables* or the *input variables.*

There are multiple approaches one might take towards constructing a predictive model. One approach is to build a probabilistic model of the data generating process, often referred to as a statistical model. Once a probabilistic model is built, one might use this to make predictions. Another approach to the predictive modeling task is using the framework of statistical learning theory, which provides the theoretical basis for many modern machine learning algorithms (von Luxburg and Schoelkopf, 2008). In this framework, the task of building predictive models is referred to as supervised learning. The models built in this framework are simply predictive functions designed to make accurate predictions on new, unobserved data from $\mathbb{P}_{Y,X}$. These models need not be probabilistic models of the data generating process, but as we will see, they often end up having a probabilistic interpretation either way.

In the statistical learning framework, predictive modeling can thus be viewed as a problem of function estimation (Vapnik, 1999). The prediction accuracy of the function is measured using a *loss function*, which measures the discrepancy between the predictions and the actual outcomes.

2.2 Risk Minimization: Defining the Target

In this section, we will introduce the loss function. The loss function is the measure of prediction accuracy that we define for the problem at hand. We are ultimately interested in minimizing the expected loss, which is known as the risk. The function which minimizes the risk is known as the *target function*. This is the optimal prediction function we would like to obtain.

2.2.1 The Loss Function

Loss functions play a central role in decision theory (Young and Smith, 2005). Statistical decision theory can be viewed as a game against nature, as opposed to against other strategic players (Murphy, 2012). In this game, we have to choose an action a to take from the set of allowable actions, i.e. the action space $\mathcal{A}$. This action is subsequently judged in context of the true outcome $y \in \mathcal{Y}$, which is picked by nature. The loss function

$$L : \mathcal{Y} \times \mathcal{A} \to \mathbb{R}_+ \tag{2.1}$$

gives a quantitative measure of the loss incurred from choosing action a when the true outcome ends up being y. The lower the loss incurred, the better.

Loss functions can be used for measuring the quality of a parameter estimate. In that case, the loss measures the discrepancy between the true parameter, picked by nature, and our parameter estimate. In predictive modeling however, we are concerned with predictions. Hence, we will utilize loss functions to measure the quality of a prediction. In this case, the loss measures the discrepancy between the true outcome, picked by nature, and our prediction. The action in our case will therefore be to make a prediction a. This prediction is subsequently judged in context of the true outcome of the response variable y.

Oftentimes in the literature, predictions are denoted $\hat{y}$ instead of a. We will however stick to $a \in \mathcal{A}$ as predictions need not be members of the set $\mathcal{Y}$. For example, if we have a binary classification problem with $\mathcal{Y} = \{0, 1\}$, we could for example use probabilistic predictions where $\mathcal{A} = [0, 1]$. As such, predictions are not directly an estimate of y, but can more generally be viewed as some action taken which will be judged in context of y.

2.2.1.1 Examples

Common loss functions for regression include the squared error loss

$$L(y, a) = \frac{1}{2}(y - a)^2$$

and the absolute loss

$$L(y, a) = |y - a|$$

which measures the quadratic and absolute discrepancy between the prediction a and the true outcome y, respectively.

For classification, a common loss function is the misclassification or 0-1 loss

$$L(y, a) = \mathrm{I}(y \neq a),$$

which assigns a loss of 1 to misclassifications and a loss of 0 to correct classifications. More generally, one can also assign greater losses to certain misclassifications.

2.2.2 The Risk Function

The loss function measures the accuracy of a prediction after the outcome is observed. At the time we make the prediction however, the true outcome is still unknown, and the loss incurred is consequently a random variable $L(Y, a)$. It would therefore be useful to have a notion of an optimal action under uncertainty.

The *risk* of action a is defined as the expected loss

$$R(a) = \mathrm{E}[L(Y, a)].$$

The optimal action is defined to be the *risk minimizer* a^* (Murphy, 2012),

$$a^* = \underset{a \in \mathcal{A}}{\arg\min}\, R(a).$$

2.2.3 The Model

The make predictions depending on the input X, we will use a model

$$f : \mathcal{X} \to \mathcal{A},$$

mapping every input $x \in \mathcal{X}$ to a corresponding prediction $a \in \mathcal{A}$. Thus, for a given x, we would make the prediction

$$a = f(x),$$

and would consequently incur a loss of $L(y, f(x))$. The model is also referred to as a *hypothesis*, a *prediction function*, a *decision function* or a *decision rule*.

Let $\mathcal{A}^{\mathcal{X}}$ denote the set of all functions mapping from $\mathcal{X}$ to $\mathcal{A}$. The problem of estimating a model $\hat{f}$ can be viewed as a problem of selecting a function $\hat{f}$ from the set $\mathcal{A}^{\mathcal{X}}$ based on data. We will come back to this in Section 2.3.

2.2.4 The Target Function

In Section 2.2.2, we discussed the risk of a prediction $a \in \mathcal{A}$. Now, we will discuss the risk of a model or prediction function $f \in \mathcal{A}^{\mathcal{X}}$.

The risk of a model f is defined as the expected loss over the joint distribution $\mathbb{P}_{Y,X}$ (Tewari and Bartlett, 2014)

$$R(f) = \mathrm{E}[L(Y, f(X))].$$

The optimal model is defined to be the *risk minimizer* f^*,

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

This is also known as the *target function* and is the function we are ultimately interested in estimating. Note that

$$\begin{aligned} f^* &= \arg\min_{f \in \mathcal{A}^{\mathcal{X}}} E[L(Y, f(X))] \\ &= \arg\min_{f \in \mathcal{A}^{\mathcal{X}}} E[E[L(Y, f(X))|X]]. \end{aligned}$$

The target function can therefore be calculated pointwise as

$$f^*(x) = \arg\min_{f(x) \in \mathcal{A}} E[L(Y, f(x))|X = x] \quad \forall x \in \mathcal{X}.$$

2.2.4.1 Examples

For the squared error loss, the corresponding risk is

$$R(f) = \mathrm{E}[(Y - f(x))^2|X = x].$$

From this, it is straightforward to show that the corresponding target function is given by the conditional mean

$$f^*(x) = \mathrm{E}[Y|X = x]$$

while the risk of the target function is given by the conditional variance

$$R(f^*(x)) = \mathrm{Var}[Y|X = x].$$

For the absolute loss on the other hand, the risk is

$$R(f) = \mathrm{E}[|Y - f(x)| \Big| X = x].$$

The target function is the conditional median

$$f^*(x) = \mathrm{Med}[Y|X = x]$$

while the risk of the target function is given by the conditional mean absolute deviation

$$R(f^*(x)) = \mathrm{MAD}[Y|X = x].$$

Finally, for the misclassification loss, the risk is the conditional probability of misclassification

$$R(f) = \mathrm{Prob}[Y \neq f(x)|X = x].$$

The corresponding target function is

$$f^*(x) = \underset{y \in \mathcal{Y}}{\arg\max}\, \mathrm{Prob}[Y = y|X = x],$$

while the risk of the target function is given by

$$R(f^*(x)) = \underset{y \in \mathcal{Y}}{\arg\min}\, \mathrm{Prob}[Y = y|X = x].$$

2.3 Empirical Risk Minimization: Defining the Solution

Ultimately, we want a model which generalizes as well as possible, i.e. one that has as low true risk as possible. However, since we don't know the true risk of a model, we need to rely on empirical estimates of risk when inferring our model. In this section, we will discuss the principle of *empirical risk minimization* which is the main inductive principle used in statistical learning (Tewari and Bartlett, 2014). This inductive principle relies on minimization of the *empirical risk*.

2.3.1 Empirical Risk

Calculating the risk of a model would require complete knowledge of the true, but unknown, distribution $\mathbb{P}_{Y,X}$. This is also known as the *population distribution*, from which we assume the data was sampled. Let $\hat{\mathbb{P}}_{Y,X}$ denote the *empirical distribution*, i.e. the distribution that assigns a probability mass of $1/n$ to each data point. Our information about the true distribution is limited to the information in our empirical distribution.

The empirical risk $\hat{R}(f)$ is simply an empirical estimate of the true risk $R(f)$ of a model, where the expectation of the loss is taken with respect to the empirical distribution rather than the population distribution. The empirical risk of a function f is

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i)).$$

Note also that the empirical risk can be viewed as a random quantity, depending on the realization of the data set $\mathcal{D}$. By the strong law of large numbers we have that

$$\lim_{n\to\infty} \hat{R}(f) = R(f),$$

almost surely.

2.3.2 Empirical Risk Minimization

Empirical risk minimization (ERM) is an induction principle which relies on minimization of the empirical risk (Vapnik, 1999). The model defined by ERM is the empirical risk minimizer $\hat{f}$, which is an empirical approximation of the target function which was defined as the risk minimizer f^*.

We define the empirical risk minimizer as

$$\hat{f} = \arg\min_{f\in\mathcal{F}} \hat{R}(f),$$

where $\mathcal{F} \subseteq \mathcal{A}^{\mathcal{X}}$ is some class of functions. ERM is a criterion to select the optimal function $\hat{f}$ from a set of functions $\mathcal{F}$. The choice of $\mathcal{F}$ is of major importance.

2.3.2.1 The Naive Approach

The naive approach would be to let $\mathcal{F} = \mathcal{A}^{\mathcal{X}}$. That is, we would allow any function in $\mathcal{A}^{\mathcal{X}}$ to be a solution. In the general case where the cardinality of $\mathcal{X}$ is infinite, we would essentially attempt to estimate an infinite number of parameters using only a finite data set. The problem is thus ill-posed (Evgeniou et al., 2000; Rakotomamonjy and Canu, 2005). There will be an infinite number of solutions on the form

$$\hat{f}(x) = \begin{cases} a_i & \text{, if} \quad x = x_i \\ \text{arbitrary} & \text{, else} \end{cases}.$$

where

$$a_i = \arg\min_{a\in\mathcal{A}} \sum_{\{j:x_j=x_i\}} L(y_j, a).$$

To solve the function estimation problem we thus need to impose additional assumptions on the solution.

2.3.3 The Model Class

To make the function estimation problem well-posed, we need to restrict $\mathcal{F}$ to be a subset of the total function space, i.e. $\mathcal{F} \subset \mathcal{A}^{\mathcal{X}}$. In machine learning this is commonly referred to as the *hypothesis space*. This restriction of the function space essentially defines a class of models. We will thus refer to $\mathcal{F}$ as a *model class.*

The perhaps most popular model class is the class of linear models

$$\mathcal{F} = \{f : f(x) = \theta_0 + \sum_{j=1}^{p} \theta_j x_j, \quad \forall x \in \mathcal{X}\}.$$

This model greatly simplifies the function estimation problem to estimation a parameter vector $\theta = (\theta_0, \theta_1, ..., \theta_p)^T$.

We will come back with more examples of model classes in Section 2.5 where we provide an overview of common learning methods.

2.3.4 The Learning Algorithm

The model class together with the ERM principle reduces the learning problem to an optimization problem. The model class constitutes a set of functions which are considered candidate solutions, while the ERM principle provides us with a criterion to select a function from this set. This defines the statistical aspect of the learning problem. The computational aspect of the problem is to actually solve the the optimization problem defined by ERM. This is the job of the *learning algorithm*, which is essentially just an optimization algorithm. The learning algorithm takes a data set $\mathcal{D}$ as input and outputs a fitted model $\hat{f}$.

Most model classes will have some parameters $\theta \in \Theta$ that the learning algorithm will adjust to fit the data. In this case, it suffices to estimate the parameters $\hat{\theta}$ in order to estimate the model

$$\hat{f}(x) = f(x; \hat{\theta}).$$

Different choices of model classes and loss functions will lead to different optimization problems. These optimization problems will vary in difficulty and thus require different approaches. The simplest problems yield analytic solutions. Most problems do however require numerical methods.

When the objective function is continuous with respect to θ we get a *continuous optimization problem*. When this is not the case, we get a *discrete optimization problem*. It is often desirable to have a model class and loss function which lead to a continuous optimization problem as these are typically easier to solve than discrete optimization problems.

One notable example of a model class which leads to a discrete optimization problem is tree models. In fact, learning a tree model is an NP-complete problem (Hyafil and Rivest, 1976). To learn a tree model one thus have to make approximations to reduce the search space. We will discuss this further in Chapter 5.

Most model classes does however lead to continuous optimization problems. There are a vast set of methods for continuous optimization problems. SeeNocedal and Wright (2006) for more details. Two prominent methods that will be important for this thesis however, is the method of *gradient descent* and *Newton's method*. We will motivate gradient boosting and Newton boosting as approximate nonparametric versions of these optimization algorithms. We will come back to this in Chapter 4.

2.3.5 Connection to Maximum Likelihood Estimation

Empirical risk minimization (ERM) and maximum likelihood estimation (MLE) are closely related. In the case of i.i.d. data, MLE can be formulated as ERM by selecting an appropriate loss function. The predictive model resulting from ERM can thus be interpreted as a statistical model in this case.

2.3.5.1 Maximum Likelihood Estimation

We will here briefly review maximum likelihood estimation. Let us first consider parametric density estimation. Assume Y comes from a parametric distribution

$$Y \sim P_Y(y; \theta)$$

where $\theta \in \Theta$ are the parameters of the distribution. One approach to estimating θ is to use maximum likelihood estimation (MLE). The maximum likelihood estimate can be written as

$$\hat{\theta} = \arg\max_{\theta \in \Theta} l(\theta; y_1, ..., y_n) = \arg\max_{\theta \in \Theta} \sum_{i=1}^{n} \log P_Y(y_i; \theta).$$

Let us now generalize this and let the parameter θ depend on X. That is, let it be a function

$$\theta : \mathcal{X} \to \Theta,$$

and assume that

$$Y|X \sim P_{Y|X}(y; \theta(X)).$$

Then

$$\hat{\theta} = \arg\max_{\theta \in \Theta^{\mathcal{X}}} \sum_{i=1}^{n} \log P_{Y|X}(y_i; \theta(x_i)).$$

Rewriting this in the form

$$\hat{\theta} = \arg\min_{\theta \in \Theta^{\mathcal{X}}} \{\frac{1}{n} \sum_{i=1}^{n} - \log P_{Y|X}(y_i; \theta(x_i))\},$$

we see that this is equivalent to the empirical risk minimizer of the loss function

$$L(y, \theta(x)) = -\log P_{Y|X}(y; \theta(x)). \tag{2.2}$$

We will refer to a loss function defined in this way as a *likelihood-based loss function.*

2.3.5.2 Link Functions

When the parameter space Θ is bounded, models are typically fit on a transformed scale where estimation is convenient. This is the trick used by e.g. generalized linear models (GLMs) (Nelder and Wedderburn, 1972) and generalized additive models (GAMs) (Hastie and Tibshirani, 1986). The trick is to define a link function

$$g : \Theta \to \mathcal{A}$$

and instead estimate the function

$$f(x) = g(\theta(x)) \in \mathcal{A}$$

on the transformed scale. Link functions are also commonly used for additive tree models. Here, regression trees are estimated and added on the transformed scale. The final additive tree model can make predictions on the original scale by applying the inverse link g^{-1} to the predictions.

2.4 Common Loss Functions

The loss function chosen for a particular problem should reflect some discrepancy between the observations and the predictions that we would like to minimize. In theory we can use any loss function as defined in Equation 2.1. In practice however, there are a few popular loss functions which tend to be used for a wide variety of problems.

Many of the loss functions used in practice are likelihood-based, i.e. they can be interpretations as negative log-likelihoods. We will discuss some examples in Section 2.4.1. For classification, minimizing the number of misclassifcations might be the goal. We will discuss the misclassification loss in Section 2.4.2. Sometimes, a loss function which is more convenient to optimize is used instead of the actual loss function during estimation. We will discuss this in Section 2.4.3.

2.4.1 Likelihood-based Loss Functions

We defined the likelihood-based loss function in Equation 2.2. We will here briefly discuss some common examples.

2.4.1.1 The Gaussian Distribution

For regression, the Gaussian distribution is popularly used. That is, we assume a conditional Gaussian distribution

$$[Y|X] \sim \text{Normal}(\mu(X), \sigma^2).$$

The loss function based on this likelihood is

$$L(y, \mu(x)) = \frac{1}{2}\log 2\pi\sigma^2 + \frac{1}{2\sigma^2}(y - \mu(x))^2. \tag{2.3}$$

For the Gaussian distribution, the identity link is used. That is, the model is estimated on the original parameter scale, $f(x) = g(\mu(x)) = \mu(x)$. Assuming that σ^2 is a constant we are not interested in estimating, we can simplify the expression in Equation 2.3. Doing this we find that this loss function is equivalent to the squared error loss

$$L(y, f(x)) = (y - f(x))^2.$$

Maximum likelihood estimation with a Gaussian error assumption is thus equivalent to least-squares regression.

2.4.1.2 The Bernoulli Distribution

For binary classification, the Bernoulli/binomial distribution is useful. Letting $\mathcal{Y} = \{0, 1\}$, we can assume

$$[Y|X] \sim \text{Bernoulli}(p(X)).$$

The loss function based on this likelihood is

$$L(y, p(x)) = -y \log(p(x)) - (1 - y)\log(1 - p(x))$$

The target function can easily be shown to be

$$p^*(x) = \text{Prob}[Y = 1|X = x].$$

Since $p(x) \in [0, 1]$ is bounded, it is common to model it on the logit scale instead. We define

$$f(x) = g(p(x)) = \log \frac{p(x)}{1 - p(x)},$$

where g is the *logit link.* The loss function for the model on the logit scale can be written

$$L(y, f(x)) = \log(1 + e^{f(x)}) - yf(x).$$

Fitting linear models with this loss function gives rise to the popular logistic regression (Cox, 1958).

The loss function based on the Bernoulli likelihood is also referred to as the log-loss, the cross-entropy or the Kullback-Leibler information (Shen, 2005).

2.4.1.3 Other Distributions

Similarly, one can derive corresponding loss functions by assuming other distributions. An important class of distributions is the *exponential family.* This class includes the Gaussian and Bernoulli distribution, as well as many other important distributions. Examples include the Poisson distribution, the gamma distribution and the multiclass generalization of the Bernoulli/binomial distribution, i.e. the categorical/multinomial distribution. See e.g. Nielsen and Garcia (2009) for a more comprehensive overview.

Other important distributions include the Laplace distribution and the asymmetric Laplace distribution. The Laplace distribution can be seen to correspond to the absolute loss function, and can thus be used for estimating the conditional median. The asymmetric Laplace distribution can be seen to correspond to the loss function for quantile regression (Kozumi and Kobayashi, 2011), and can thus be used to estimate other conditional quantiles than the median (Koenker, 2005).

2.4.2 The Misclassification Loss

For classification problems, we might be interested in predicting the ”crisp” labels of new data, rather than the probabilities of the various class labels. If we are interested in minimizing the rate of misclassifications, we can use the misclassification loss. This loss function is not associated with any likelihood, but can be seen to judge the quality of some action, i.e. a crisp classification, one take based on the data.

Even though we are interested in minimizing the misclassification rate, the misclassification loss is rarely used for model estimation as it leads to an NP-hard optimization problem (Nguyen and Sanner, 2013). One therefore typically employ some surrogate loss function instead.

2.4.3 Surrogate Loss Functions

When used for estimation, some loss functions lead to problems. Typically, you run into computational issues in classification problems and robustness issues in regression problems (Steinwart, 2007). In those cases, it is common to use a *surrogate loss function* instead of the original loss function.

2.4.3.1 Surrogate Loss Functions for Regression

Consider using the squared error loss for a regression problem. If the data set contains outliers, this might severely impact the model. One might therefore consider to use the absolute loss or the Huber loss (Huber, 1964) instead as they are more robust to outliers.

2.4.3.2 Surrogate Loss Functions for Classification

Using the misclassification loss for estimation leads to an NP-hard optimization problem (Nguyen and Sanner, 2013). It is therefore common to use convex surrogates instead (Bartlett et al., 2006).

One commonly used surrogate loss is the log-loss from Section 2.4.1.2. Classifications are made by

$$c(x) = \mathrm{I}(p(x) \geq 0.5)$$

or equivalently

$$c(x) = \mathrm{sign}\{f(x)\}.$$

Any method minimizing the log-loss can thus be used for classification.

Other methods have been developed specifically for the case of binary classification with misclassification loss. These methods focus directly on determining the decision boundary between the classes that would minimize the misclassification loss.

For these methods one typically let $y \in \mathcal{Y} = \{-1, 1\}$ and let the model assign a real-valued score, i.e. $f(x) \in \mathcal{A} = \mathbb{R}$, where $f(x) = 0$ defines the decision boundary between the two classes. The 0-1 loss function can then be rewritten as

$$L(y, f(x)) = I(yf(x) < 0).$$

The quantity $yf(x)$ is called the *margin.* This quantity is positive for correct classifications.

Two prominent methods that were motivated using the margin are Support Vector Machines (SVMs) (Cortes and Vapnik, 1995) and AdaBoost (Freund and Schapire, 1995). These methods have however been shown to utilize the *hinge loss function* (Lin, 2002)

$$L(y, f(x)) = \max\{0, 1 - yf(x)\}$$

and the *exponential loss function* (Friedman et al., 2000)

$$L(y, f(x)) = e^{-yf(x)},$$

respectively. These are two other possible surrogate loss functions for the misclassification loss. As opposed to the log-loss however, these are not likelihood-based. Finally, note that the log-loss is sometimes also written as a function of the margin, i.e.

$$L(y, f(x)) = \log(1 + e^{-2yf(x)}).$$

2.5 Common Learning Methods

We will here give a brief overview of some learning methods that are commonly used. We will come back to some of them later. This section is not intended to give comprehensive introduction to any learning method nor to give a comprehensive overview of various learning methods. For more details see e.g. Hastie et al. (2009), Murphy (2012) or Kuhn and Johnson (2013).

2.5.1 The Constant

Les us begin with an extreme example of a model class, the constant. This restricts the model fitted to be a constant on the form

$$f(x) = \theta_0.$$

This reduces the estimation problem to estimating only one parameter θ_0. This model class assumes that Y is independent of all the predictor variables X. This is of course not a particularly useful model, but can be considered as a sort of base case.

2.5.2 Linear Methods

Linear methods are learning methods which employ the class of linear models. A linear model can be written on the form

$$f(x) = \theta_0 + \sum_{j=1}^{p} \theta_j x_j. \tag{2.4}$$

For this model class, the estimation problem is simplified to estimating a parameter vector $\theta = (\theta_0, \theta_1, ..., \theta_p) \in \mathbb{R}^{p+1}$.

When the squared error loss is used, we get the familiar linear regression. The parameter vector can in this case be determined analytically as

$$\hat{\theta} = (X^T X)^{-1} X^T y,$$

where $X = (x_1, ..., x_n)^T$ is the *design matrix* and $y = (y_1, ..., y_n)^T$.

More generally, if the loss function used is based on an exponential family likelihood, Equation 2.4 defines a generalized linear model. These models are typically fit using *iteratively reweighted least squares* (IRLS) (Nelder and Wedderburn, 1972), which is essentially a Newton method for optimizing the parameter vector θ.

If the loss function used is the hinge loss, Equation 2.4 is a support vector classifier or linear support vector machine. The optimization problem is a quadratic programming problem (Cortes and Vapnik, 1995). A popular learning algorithm is *sequential minimal optimization* (Platt, 1998).

2.5.3 Local Regression Methods

One simple way to introduce nonlinearities is to let the fitted function be more affected by nearby points than distant points. A *kernel function* is a function

$$\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}.$$

which measures the similarity or closeness between inputs. For local regression methods, the kernel function is used to measure closeness between points in the input space $\mathcal{X}$. Some examples include kernel regression (Nadaraya, 1964; Watson, 1964), k-Nearest-Neighbours and locally weighted regression (Cleveland and Devlin, 1988).

These methods typically require fitting a weighted average or weighted regression model at the time of prediction. The weights will generally be different for every prediction point according to which other points are nearby. These models thus require fitting a separate model for each prediction point. They thus also require keeping all of the training data in memory and are thus sometimes called *memory-based methods.* On the plus side however, they require no training.

Since distance measures tend to become less meaningful in higher dimensions (Aggarwal et al., 2001), local regression methods tend to become less useful as the dimension p grows. This phenomenon is also known as the *curse of dimensionality* (Bellman and Bellman, 1961).

2.5.4 Basis Function Expansions

The linear model in Equation 2.4 can also be extended to handle nonlinearities by using a *basis function expansion.* A basis function expansion can be written in the form

$$f(x) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x). \tag{2.5}$$

Instead of being linear in the p inputs, this function is linear in M (potentially) nonlinear functions of the input vector.

Letting

$$\theta = \big(\theta_0, \theta_1, ..., \theta_m\big)^T$$

and

$$\phi(x) = \big(1, \phi_1(x), ..., \phi_m(x)\big)^T,$$

we can rewrite Equation 2.5 as

$$f(x) = \theta^T \phi(x).$$

If we use the squared error loss function, we get an analytic solution similar to the case for linear models

$$\hat{\theta} = (\Phi^T \Phi)^{-1} \Phi^T y,$$

where $\Phi = (\phi(x_1), ..., \phi(x_n))^T$ is the design matrix and $y = (y_1, ..., y_n)^T$. Other

Depending on the choice of basis functions $\phi_1, ..., \phi_M$, this will yield different model classes. Many commonly used methods fit models which can be written as basis function expansions.

2.5.4.1 Explicit Nonlinear Terms

In the simplest case, the nonlinear basis functions are explicitly specified by the user and then fit as an ordinary linear model. Examples include polynomial and interaction terms such as x_j^2 and $x_j x_k$, and other nonlinear transformations, such as $\log x_j$ or $1/x_j$.

One drawback of this approach is that the nonlinearities have to be explicitly specified. The main drawback however is that the basis functions are global, i.e. they do not impose any locality. That is, all data points will have an equally strong effect on the fitted function everywhere in the input space $\mathcal{X}$. This can very quickly lead to nonsensible fits as an outlier in area of the input space can have a strong effect on the fit at all other points the input space. Outliers can thus have a strong negative impact on the fit. The basis functions are global for linear models as well, although they tend to be more stable due to their restricted nature.

2.5.4.2 Splines

One way to introduce locality in the function fitting is to use spline models. These models define local polynomial models in prespecified regions of the input space, where there are typically smoothness restrictions along the boundaries of these regions. Although these models can be specified for higher dimensional input, using e.g. thin-plate splines or tensor basis splines, they are most commonly used for one dimensional inputs. For the one dimensional case, the boundaries are called *knots*.

A spline model is defined by specifying the location of the knots, the order of the polynomial fits between the knots and the smoothness restrictions at the knots. Important examples include regression splines and smoothing splines (Silverman, 1984).

2.5.4.3 Kernel Methods

Kernel methods fit models which can be written as a basis function expansion where the basis functions $\phi_1, ..., \phi_M$ are defined using the kernel function

$$\phi_m(x) = \kappa(x, \mu_m)$$

and a set of M *centroids* $\mu_1, ..., \mu_M$. The centroids are usually taken to be the input points $x_1, ..., x_n$. There are multiple examples of kernels which are popularly used. Among the most common is the RBF kernel (radial basis function kernel)

$$\kappa(x, x') = \exp(-\frac{\|x - x'\|}{2\sigma^2})$$

which is a special case of the Gaussian kernel.

Popular examples of kernel methods include support vector machines (SVMs) (Cortes and Vapnik, 1995) and relevance vector machines (RVMs) (Tipping, 2001). The main drawback of these methods are that they are strongly affected by curse of dimensionality (Efron and Hastie, 2016).

2.5.5 Adaptive Basis Function Models

Another large class of models is *adaptive basis function models* (Murphy, 2012). Models in this class differ from earlier mentioned approaches in that the basis functions are themselves learnt using data and do not take a predetermined form. Popular adaptive basis function models include tree models, generalized additive models, boosting and neural networks (Murphy, 2012).

2.5.5.1 Generalized Additive Models

Generalized additive models (GAMs) are a generalization of GLMs which allow each predictor to have a nonlinear effect on the output. The model is however restricted such that each basis function depend solely on one predictor variable. We get the class of GAMs by letting $M = p$, $\phi_j(x) = \phi_j(x_j)$ and $\theta_1, ..., \theta_p = 1$. This gives us models in the form

$$f(x) = \theta_0 + \sum_{j=1}^{p} \phi_j(x_j),$$

The basis functions ϕ_j can be of any suitable model class. Typically used are scatterplot smoothers such as smoothing splines or locally weighted regression. The learning algorithm used is typically *backfitting*, together with *local scoring* for loss functions other than the squared error loss (Hastie et al., 2009). This is again a Newton method which fits adjust the fit for one predictor at a time.

2.5.5.2 Neural Networks

A *feedforward neural network* with M hidden units in the last hidden layer is a linear basis function expansion of the kind defined in Equation 2.5 where θ_0 is referred to as the *bias*, $\theta_1, ..., \theta_M$ are referred to as the *weights*. For a single-layer network, the basis functions are given by

$$\phi_m(x) = a(w_{m,0} + \sum_{j=1}^{p} w_{m,j} x_j),$$

where a is a nonlinear function called the *activation function* and $w_{m,j}, m = 1, ..., M; j = 0, ..., p$ are the weights of the hidden layer. More generally, one can have multilayer networks where the layers are stacked.

Neural networks are typically fit using *backpropagation* (Rumelhart et al., 1988) which allows one to do gradient descent steps in the joint parameter space of all biases and weights of the network.

2.5.5.3 Tree Models

Tree models, which we will discuss in more detail in Chapter 5, assume that the relationship between the response and the predictors can be modeled by locally constant fits. The input space is divided into M regions $R_1, ..., R_M$ and a constant is fit in each region. The basis functions are indicator functions $\phi_m(x) = \mathrm{I}(x \in R_m)$. With $\theta_0 = 0$, tree models thus take the form

$$f(x) = \sum_{m=1}^{M} \theta_m \mathrm{I}(x \in R_m),$$

where θ_m defines the constant fit in region R_m. There are multiple learning algorithms for fitting tree models. Among the most popular are CART (Breiman et al., 1984) and C4.5 (Quinlan, 1993).

2.5.5.4 Boosting

Finally, we have boosting which fits models which can be written in the form

$$f(x) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x).$$

Boosting is very general in the sense that you have to specify the model class of the basis functions Φ. A boosting algorithm can be viewed as a learning algorithm which approximately finds the optimal linear combination of M basis functions from this class. During fitting, the basis functions are added in a sequential manner where each basis function is learnt using a *base learner* $\mathcal{L}_\Phi$. We will come back to boosting in Chapter 4. The most popular choice of basis functions are tree models. In this case, boosting fits additive tree models. We will come back to tree boosting methods in Chapter 6.

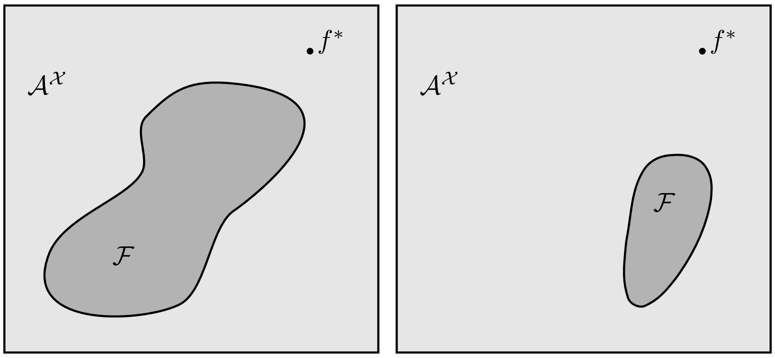
Chapter 3

Model Selection

In Section 2.5, we discussed several different learning methods. These methods employ different model classes and typically have different *hyperparameters* to control the fit. Hyperparameters are parameters which are not optimized by the learning algorithm and are also referred to as *meta-parameters* or *tuning parameters*. In this chapter, we will discuss the issue of selecting between different learning methods and their hyperparameters, which is known as *model selection*.

3.1 Model Complexity

Different model classes can be viewed as different restrictions on the function space $\mathcal{A}^{\mathcal{X}}$. An illustration of two different model classes $\mathcal{F}$ are shown in Figure 3.1. The target function f^* is also depicted. We regard functions that are depicted as equidistant to f^* as having equal risk, with risk increasing with the distance from f^*. Note that this is only an illustration as the space $\mathcal{A}^{\mathcal{X}}$ will generally be infinite-dimensional.



(a) A model class. (b) A more restricted model class.

Figure 3.1: Examples of model classes.

Most of the functions in $\mathcal{A}^{\mathcal{X}}$ will have wildly irregular structures which are very unlikely to resemble the target function f^*. The restrictions imposed on the function space by various model classes can typically be viewed as complexity restrictions of some kind (Hastie et al., 2009). Most model classes impose, either implicitly or explicitly, some simple structure that the model should have in small neighbourhoods of the input space $\mathcal{X}$, such as locally constant, locally linear or some other simple structure. The size of this neighbourhood and the local structure in the neighbourhood is related to the complexity. For example, the constant model class that we described in Section 2.5 forces the model to be globally constant, while the linear model class defines models which are globally linear. These model classes are thus very rigid as their neighbourhoods are global. By making the neighbourhoods smaller, the models will be able to fit the data more flexibly and the complexity of the model class will thus increase. Many other methods, such as local regression, splines and tree models instead define more local neighbourhoods. While local regression and splines explicitly define the size of the neighbourhoods through hyperparameters, tree methods adaptively determines the size of the neighbourhoods.

So, why should we not simply pick a complex model class which is very flexible and thus capable of fitting almost any interesting structure in the data? This will be discussed in the next section.

3.2 Generalization Performance

Since our goal is prediction, we would like to have a model that generalizes well, i.e. one that performs well on new, independent data from $\mathbb{P}_{Y,X}$. That is, we want a model $\hat{f}$ with as low true risk $R(\hat{f})$ as possible.

In statistical learning, we typically assume we have little or no knowledge of the target function f^*. We would thus like to estimate the model using as few assumptions as possible. That is, we would like to use a flexible model class capable of fitting all the interesting structure in the data. When the model class is too flexible however, it ends up fitting structure in the data that will not generalize well. This is called *overfitting* the data. If the model class is not flexible enough on the other hand, it will end up not being flexible enough to fit interesting structure in the data. This is called *underfitting* the data. There is thus a tradeoff in selecting an appropriate model complexity. This is often referred to as the *bias-variance tradeoff*, which we will come back to in Section 3.3.

Since we want to minimize the true risk of our fitted model $R(\hat{f})$, we would like to have an estimate of it. One might consider using the *training error* $\hat{R}(\hat{f})$ to estimate the true risk $R(\hat{f})$. However, since $\hat{f}$ was obtained by minimizing the training error, the training error will be a biased estimate for the generalization error $R(\hat{f})$. The more complex model class we use, the more we are able to reduce the training error. However, if the model class used is too complex, we might overfit the data, leading to poor generalization performance. We thus have that the more complex model class we use, the more biased the training error $\hat{R}(\hat{f})$ will be as an estimate of generalization error $R(\hat{f})$. This is illustrated in Figure 3.2.

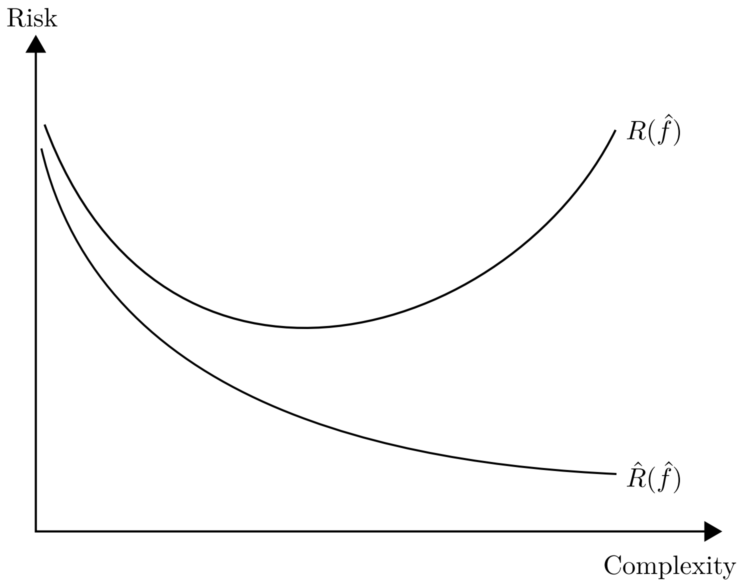


Figure 3.2: Generalization vs training error.

From this we can understand why hyperparameters cannot be optimized by the learning algorithm. Since the training error would always prefer a more complex model, we would always end up overfitting the data.

For some model classes there exist various model selection criteria which take into account the model complexity. Examples include AIC (Akaike, 1973), BIC (Schwarz, 1978), adjusted R^2 and many more. The most common approach to more reliably estimate the generalization error is however to use an independent data set to measure the risk. The data set we have available is typically split into a *training set* and a *validation set*. The model is fit by minimizing empirical risk on the training set, while the generalization error is measured by calculating the empirical risk on the validation set. The hyperparameter settings which gives the best estimated generalization can subsequently be selected. More generally, one can use *cross-validation* (Stone, 1974; Allen, 1974; Geisser, 1975).

When the generalization error is measured for the purpose of model assessment rather than model selection, the independent data set is typically referred to as a *test set* rather than a validation set.

3.3 The Bias-Variance Tradeoff

The ERM criterion is one example of a criterion used to select models from a model class. In Section 3.4, we will discuss how we can alter the ERM criterion in order to obtain models which generalize better. We will refer to the collection of the model class $\mathcal{F}$ and the criterion used to select a model from it as the *fitting procedure*.

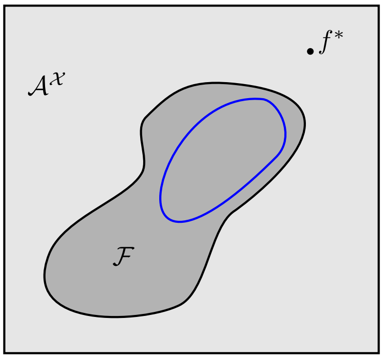


Figure 3.3: The distribution of empirical risk minimizers over the function space.

The fitting procedure can be thought of as giving rise to a probability distribution of possible models over the function space $\mathcal{A}^{\mathcal{X}}$. This distribution is induced by the distribution of possible realizations of the training data $\mathcal{D}$. The different possible realizations of the training data $\mathcal{D}$ gives rise to different empirical risk functions $\hat{R}(f)$. The different empirical risk functions in turn gives rise to different models $\hat{f} \in \mathcal{F}$ which optimize the criterion. The distribution over the function space is illustrated in Figure 3.3, where the area enclosed by the blue shape is intended to represent the area of highest density in the function space.

First of all, the choice of the model class defines which models are considered solutions. Thus, any model which is not in the model class $\mathcal{F}$ will have a probability of zero assigned to it. For example, when using the linear model class, any function which is not linear will not be considered a solution. Second of all, the criterion used to select the model will affect the distribution over the model class $\mathcal{F}$. For ERM we are minimizing the empirical risk, which is equivalent to maximizing the likelihood when the loss function is likelihood-based.

We are interested in using a procedure which will tend to give us models $\hat{f}$ that are “close to” the target function f^*, i.e. have low risk $R(\hat{f})$. Note that since $\hat{f}$ can be viewed as random, the true risk $R(\hat{f})$ will also be a random quantity. The *expected risk* $\mathrm{E}[R(\hat{f})]$ however, where the expectation is taken over the distribution of possible training data sets, is not a random quantity. A procedure with low expected risk will tend to generalize well. We thus seek to use a procedure which will have low expected risk. The expected risk can be decomposed in various ways. One way is the decomposition to approximation and estimation error, see e.g. Tewari and Bartlett (2014). Another decomposition, which is specific to the squared error loss, is the *bias-variance decomposition*. We will now show this decomposition.

3.3.1 The Bias-Variance Decomposition

The expected conditional risk at x for the squared loss can be written as

$$\begin{aligned}\mathrm{E}[R(\hat{f}(x))] &= \mathrm{E}[(Y - \hat{f}(x))^2 | X = x] \\ &= \mathrm{Var}[Y|X = x] + (\mathrm{E}[\hat{f}(x)] - f^*(x))^2 + \mathrm{Var}[\hat{f}(x)] \\ &= R(f^*) + \mathrm{Bias}[\hat{f}(x)]^2 + Var[\hat{f}(x)].\end{aligned}$$

The three terms in the decomposition are referred to as noise, bias and variance, respectively.

Note that this bias-variance decomposition is given at a particular x. When talking about the bias and variance of a procedure, we usually implicitly refer to the *average* squared bias

$$\mathrm{E}[\mathrm{Bias}[\hat{f}(X)]^2]$$

and *average* variance

$$\mathrm{E}[\mathrm{Var}[\hat{f}(X)]].$$

Note also that generalizations of the bias-variance decomposition have been proposed. One generalization for arbitrary loss functions is given by Domingos (2000).

3.3.2 The Bias-Variance Tradeoff

Choosing a suitable procedure is crucial for predictive modeling. Using a more complex procedure typically reduces the (average) bias, but usually comes at the price of a larger (average) variance. Vice versa, if we use a less complex procedure, we can reduce the (average) variance at the price of greater (average) bias. The tradeoff in selecting a procedure with the right amount of complexity is therefore typically called the *bias-variance tradeoff*. A procedure which overfits the data typically has high variance, while a procedure which underfits the data typically has large bias. The tradeoff is illustrated in Figure 3.4.

Note that the complexity tradeoff is typically referred to as the bias-variance tradeoff regardless of the loss function used (Tewari and Bartlett, 2014), even though the bias-variance decomposition is only defined for the squared error loss.

3.4 Regularization

Rosset (2003) defines regularization as any part of model building which either implicitly or explicitly takes into account the finiteness and imperfection of the data and the limited information in it. That is, we can think of regularization as any technique used to control the variance of the fit. By controlling the variance of the fit, we can control the flexibility of the procedure in order to obtain models that generalize better.

Regularization can take many forms, see e.g.(Bickel et al., 2006). In this section, we will discuss regularization by constraining and penalizing model complexity.

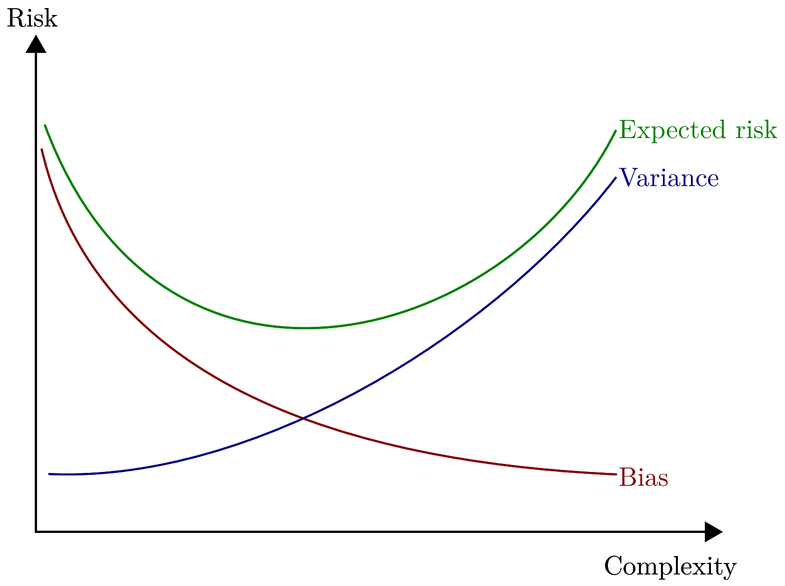


Figure 3.4: The bias-variance tradeoff.

Note that there also exist other techniques which are referred to as regularization techniques. One example, which we will come back to later, is introducing randomness in training. This is a simple and useful approach which can reduce overfitting for some procedures. Randomization is for example at the core of *bagging* (Breiman, 1996) and *Random Forests* (Breiman, 2001). It is also commonly used in boosting algorithms. We will come back to this in Section 6.2.4. The *dropout* method (Srivastava et al., 2014) used for training neural networks can also be viewed as a regularization technique based on introducing randomness during training.

3.4.1 Complexity Measures

Since we are interested in controlling complexity, we need a measure of complexity. Perhaps surprisingly, there is no universal way to define complexity (Bousquet et al., 2004). Although there is no universal definition of complexity, we can define complexity measures for specific model classes based on heuristics. That is, for a model class $\mathcal{F}$, we can define a complexity measure

$$\Omega : \mathcal{F} \to \mathbb{R}_+.$$

For linear regression for example, common complexity measures are l_0, l_1 and l_2-norm of the coefficient vector θ (excluding the intercept), where the l_0-norm is defined to measure the number of non-zero coefficients. These norms give rise to subset selection, LASSO regression and Ridge regression, respectively. The complexity measures can then be written

$$\Omega(f) = \begin{cases} \|\theta\|_0, & \text{for subset selection} \\ \|\theta\|_1, & \text{for LASSO regression} \\ \|\theta\|_2, & \text{for Ridge regression} \end{cases}$$

These complexity measures are also used by many model classes defined through basis function expansions.

Complexity measures can also be defined by the size of the local neighbourhoods or by some measure of smoothness of the model. This is typically the case for e.g. local regression methods, splines, kernel methods as well as tree methods.

3.4.2 Complexity Constraints

Constraining our model class $\mathcal{F}$ to only contain models of some maximum complexity is a simple way to regularize the problem. In some sense, the choice of some model class $\mathcal{F} \subseteq \mathcal{A}^{\mathcal{X}}$ is already a form of regularization, as we restrict our solution to be member of some set of models which is not too complex. For a model class $\mathcal{F}$, we can however typically impose further constraints on the complexity. Constraining the complexity too much will result in underfitting, while constraining it too little will result in overfitting. The strength of the constraint imposed on the model class thus becomes a hyperparameter for the learning algorithm which could be selecting using e.g. a validation set or cross-validation. By constraining the complexity by an appropriate amount, we can obtain models which generalize better to new data.

In some cases, the degree of the constraints define a nested set of model classes

$$\mathcal{F}_1 \subset \mathcal{F}_2 \subset ... \subset \mathcal{F}.$$

Here $\mathcal{F}_1$ is less complex than $\mathcal{F}_2$ and so on. This is illustrated in Figure 3.5. For linear regression for example, we can let $\mathcal{F}_1$ be the set of linear models where only one coefficient is allowed to be non-zero, $\mathcal{F}_2$ be the set where only two are allowed to be non-zero and so on. Selecting the model class with the right amount of complexity is in this case referred to as subset selection.

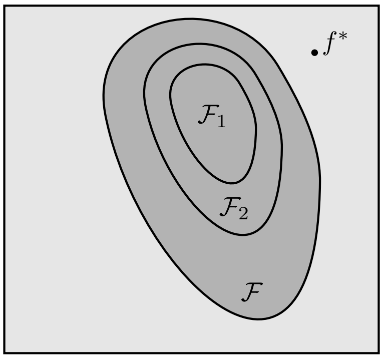


Figure 3.5: Nested model classes as a result of complexity constraints.

3.4.3 Complexity Penalization

Another way to control the complexity of the procedure is to penalize model complexity during estimation. We can define a penalized ERM criterion

$$J(f) = \hat{R}(f) + \lambda\Omega(f),$$

where $\hat{R}(f)$ is the usual empirical risk, while $\Omega(f)$ is a measure of model complexity. The model $\hat{f}$ is defined as the function $f \in \mathcal{F}$ which minimizes $J(f)$ for some fixed value of λ. The hyperparameter λ here controls how heavily complexity is penalized, thus controlling the tradeoff in model complexity and fit to the data. The training error will thus always prefer lower values of λ. An appropriate value for λ thus have to be selected using some other criterion than the training error, such as performance on an independent validation set or through cross-validation.

Penalization can be viewed as a more adaptive way of regularizing than simply constraining complexity. Penalization allow the model to fit more complex structure, but only if there is enough evidence in the data to support the additional complexity. Penalization thus simply decrease the probability of obtaining complex fits, rather than defining every model with complexity above some threshold as "off limits". In contrast to "hard" complexity constraints which shrinks the model class, "soft" complexity penalization can be seen to simply shift the distribution over the model class towards simpler models. This is illustrated in Figure 3.6.

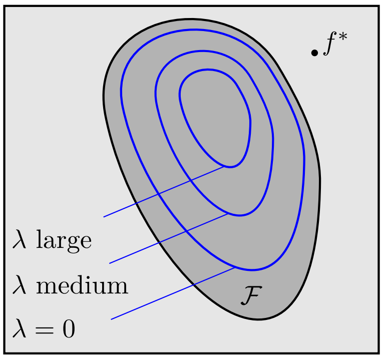


Figure 3.6: Shifting model distribution as a result of complexity penalization.

For linear regression, penalization of the l_1-norm or the l_2-norm of the coefficient vector leads to LASSO and Ridge regression, respectively. When the loss function used is likelihood-based, the LASSO and Ridge regression estimates are equivalent to MAP estimates using Laplacian and Gaussian priors for θ, respectively. Regularization thus have the Bayesian interpretation of being a prior belief that the model should be somewhat simple, as defined by the complexity measure.

Part II

Tree Boosting

Chapter 4

Boosting

Boosting refers to a class of learning algorithms that fit models by combining many simpler models (Schapire and Freund, 2012). These simpler models are typically referred to as *base models* and are learnt using a *base learner* or *weak learner*. These simpler models tend to have limited predictive ability, but when selected carefully using a boosting algorithm, they form a relatively more accurate model. This is sometimes referred to as an ensemble model as it can be viewed as an ensemble of base models. Another way to view it is that boosting algorithms are learning algorithms for fitting adaptive basis function models.

In this thesis we will focus on *gradient boosting* and what we will term *Newton boosting*. Friedman (2001) developed gradient boosting and showed that it can be interpreted as a gradient descent algorithm in function space. Newton boosting, on the other hand, is the boosting algorithm employed by XGBoost (Chen and Guestrin, 2016). We will show that this can be interpreted as a Newton method in function space and therefore name it "Newton boosting". Both of these algorithms thus have the interpretation of being numerical optimization algorithms in function space. These boosting algorithms are fairly general as they are applicable for a wide range of loss functions and base learners. Earlier boosting algorithms were however mainly focused on binary crisp classification problems. We will now briefly discuss some of these early developments before we go on to discuss numerical optimization in function space and develop gradient boosting and Newton boosting.

4.1 Early Developments

The early work on boosting focused on binary classification where the response is taken to be $\tilde{y} \in \{-1, 1\}$ and the classification is given by $c(x) \equiv \text{sign}(f(x))$ where $f(x) \in \mathbb{R}$. The quantity $\tilde{y}f(x)$ is called the margin and should be positive for the classification to be correct.

4.1.1 Boosting and AdaBoost

Boosting originated from the question posed by Kearns (1988) and Kearns and Valiant (1989) of whether a set of weak classifiers could be converted to a strong classifier. Schapire (1990) positively answered the question. Freund and Schapire (1996) presented *AdaBoost* which is regarded as the first practical boosting algorithm. This algorithm fits a weak classifier to weighted versions of the data iteratively. At each iteration, the data is reweighted such that misclassified data points receive larger weights. The resulting model can be written

$$\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=1}^{M} \hat{\theta}_m \hat{c}_m(x), \tag{4.1}$$

where $\hat{c}_m(x) \in \{-1, 1\}$ are the weak classifiers and crisp classifications are given by $\hat{c}(x) = \text{sign}(\hat{f}(x))$. This algorithm showed remarkable results and thereby attracted the attention of many researchers.

4.1.2 Forward Stagewise Additive Modeling and LogitBoost

Friedman et al. (2000) developed a statistical view of the algorithm. They showed that AdaBoost was actually minimizing the *exponential loss function*

$$L(\tilde{y}, f(x)) = \exp(-\tilde{y} f(x)).$$

In their terminology, AdaBoost fits an additive model of the form in Equation 4.1. Letting the weak classifiers be parameterized as $\hat{c}_m(x) = c(x; \hat{\gamma}_m)$, the optimization problem given by ERM is

$$\{\hat{\theta}_m, \hat{\gamma}_m\}_{m=1}^M = \underset{\{\theta_m, \gamma_m\}_{m=1}^M}{\arg\min} \sum_{i=1}^{n} L(y_i, \sum_{m=1}^{M} \theta_m c(x_i; \gamma_m)).$$

They showed that AdaBoost uses a greedy approach called *forward stagewise additive modeling* (FSAM) which iteratively fits

$$\{\hat{\theta}_m, \hat{\gamma}_m\} = \underset{\{\theta_m, \gamma_m\}}{\arg\min} \sum_{i=1}^{n} L(y_i, \hat{f}^{(m-1)}(x_i) + \theta_m c(x_i; \gamma_m))$$

at each iteration m. In addition to providing a statistical view of the AdaBoost procedure, they proposed several new boosting algorithms. Most notably, they proposed *LogitBoost* which instead of the exponential loss minimizes a second-order approximation of the log-loss at each iteration using FSAM. In fact, the LogitBoost algorithm can be viewed as a Newton boosting algorithm for the log-loss.

Breiman (1997a,b, 1998) first developed the view of boosting algorithms as numerical optimization techniques in function space. Based on this view, more general boosting algorithms that allowed for optimization of any differentiable loss function was developed simultaneously by Mason et al. (1999) and Friedman (2001).

This generalized boosting to be applicable to general regression problems and not only classification.

The algorithm presented by Friedman (2001) was gradient boosting, which has remained popular in practice. This algorithm was motivated as a gradient descent method in function space. We will now discuss numerical optimization in function space which will in turn be used for developing gradient boosting and Newton boosting in Section 4.3.

4.2 Numerical Optimization in Function Space

In this section, we will develop methods for numerical optimization in function space. More specifically, we will develop nonparametric versions of gradient descent and Newtons method which perform optimization in function space. We will in this section assume that the true risk of a model $R(f)$ is known to us. The methods developed in this chapter can thus be understood as iterative risk minimization procedures in function space.

In Section 4.3 we will transfer the methodology developed here to the practical case where the true risk $R(f)$ is unknown and we have to rely on the empirical risk $\hat{R}(f)$. By doing this we will develop gradient boosting and Newton boosting. These methods can thus be understood as iterative empirical risk minimization procedures in function space.

Before we discuss numerical optimization in function space, we will review numerical optimization in parameter space.

4.2.1 Numerical Optimization in Parameter Space

Assume in this section that we are trying to fit a model $f(x) = f(x;\theta)$ parameterized by θ. We can rewrite the risk of the model $R(f)$ as the risk of the parameter configuration, i.e.

$$R(\theta) = \mathrm{E}[L(Y, f(X;\theta)].$$

Given that $R(\theta)$ is differentiable with respect to θ, we can estimate θ using a numerical optimization algorithm such as gradient descent or Newton's method. For Newton's method, $R(\theta)$ needs to be twice differentiable.

At iteration m, the estimate of θ is updated from $\theta^{(m-1)}$ to $\theta^{(m)}$ according to

$$\theta^{(m)} = \theta^{(m-1)} + \theta_m, \tag{4.2}$$

where θ_m is the step taken at iteration m. This is the case both for gradient descent and Newton's method. The two algorithms differ in the step θ_m they take.

The resulting estimate of θ after M iterations can be written as a sum

$$\theta \equiv \theta^{(M)} = \sum_{m=0}^{M} \theta_m, \tag{4.3}$$

where θ_0 is an initial guess and $\theta_1, ..., \theta_M$ are the successive steps taken by the optimization algorithm.

4.2.1.1 Gradient Descent

Before the update at iteration m, the current estimate of θ is given by $\theta^{(m-1)}$.

At this current estimate, the direction of steepest descent of the risk is given by the negative gradient

$$-g_m = -\nabla_\theta R(\theta)\big|_{\theta=\theta^{(m-1)}}.$$

Taking a step along this direction is guaranteed to reduce risk, given that the length of the step taken is not too long. A popular way to determine the step length ρ_m to take in the steepest descent direction is to use *line search*

$$\rho_m = \arg\min_\rho R\big(\theta^{(m-1)} - \rho g_m\big).$$

The step taken at iteration m can thus be written

$$\theta_m = -\rho_m g_m,$$

Performing updates iteratively according to this yields the gradient descent algorithm.

4.2.1.2 Newton's Method

At a current estimate of θ given by $\theta^{(m-1)}$, Newton's method determines both the step direction and step length at the same time. Newton's method can be motivated as a way to approximately solve

$$\nabla_{\theta_m} R(\theta^{(m-1)} + \theta_m) = 0 \tag{4.4}$$

at each iteration to obtain θ_m. By doing a second-order Taylor expansion of $R(\theta^{(m-1)} + \theta_m)$ around $\theta^{(m-1)}$ we get

$$R(\theta^{(m-1)} + \theta_m) \approx R(\theta^{(m-1)}) + g_m^T\theta_m + \frac{1}{2}\theta_m^T H_m \theta_m,$$

where H_m is the Hessian matrix at the current estimate

$$H_m = \nabla_\theta^2 R(\theta)\big|_{\theta=\theta^{(m-1)}}.$$

Plugging this into Equation 4.4 we get

$$\nabla_{\theta_m} R(\theta^{(m-1)} + \theta_m) \approx g_m + H_m\theta_m = 0.$$

The solution to this is given by

$$\theta_m = -H_m^{-1} g_m.$$

This is the Newton step. Unlike for gradient descent, the Newton step has a "natural" step length of 1 associated with it (Nocedal and Wright, 2006). Line search steps are thus typically not used for Newton methods. From the discussion above, we can see that Newton's method is a second-order method, while gradient descent is a first-order method.

4.2.2 Numerical Optimization in Function Space

We will now discuss numerical optimization in function space. In this section we perform risk minimization of the true risk $R(f)$ where we let $f \in \mathcal{A}^{\mathcal{X}}$. We will for notational simplicity focus on the case of scalar-valued functions. For most loss functions this is all that is needed. One exception is the multinomial-based loss, for which vector-valued functions are needed. Note that minimizing

$$R(f) = \mathrm{E}[L(Y, f(X))]$$

is equivalent to minimizing

$$R(f(x)) = \mathrm{E}[L(Y, f(x))|X = x]$$

for each $x \in \mathcal{X}$.

At a current estimate $f^{(m-1)}$, the "step" f_m is taken in function space to obtain $f^{(m)}$. Analogously to parameter optimization update in Equation 4.2 we can write the update at iteration m as

$$f^{(m)}(x) = f^{(m-1)}(x) + f_m(x),$$

for each $x \in \mathcal{X}$.

Analogously to the resulting parameter optimization update in Equation 4.3, the resulting estimate of f after M iterations can be written as a sum

$$f(x) \equiv f^{(M)}(x) = \sum_{m=0}^{M} f_m(x) \tag{4.5}$$

for each $x \in \mathcal{X}$, where f_0 is an initial guess and $f_1, ..., f_M$ are the successive "steps" taken in function space.

We will now develop gradient descent and Newton's method in function space. The development of gradient descent in function space is largely based on Friedman (2001), while the corresponding development for Newton's method is a simple modification of this.

4.2.2.1 Gradient Descent

Before the update at iteration m is made, the estimate of f is given by $f^{(m-1)}$. At this current estimate $f^{(m-1)}$, the direction of steepest descent of the risk is given by the negative gradient

$$\begin{aligned} -g_m(x) &= -\left[\frac{\partial R(f(x))}{\partial f(x)}\right]_{f(x)=f^{(m-1)}(x)} \\ &= -\left[\frac{\partial \mathrm{E}[L(Y, f(x))|X = x]}{\partial f(x)}\right]_{f(x)=f^{(m-1)}(x)} \\ &= -\mathrm{E}\left[\frac{\partial L(Y, f(x))}{\partial f(x)}\Big|X = x\right]_{f(x)=f^{(m-1)}(x)} \end{aligned} \tag{4.6}$$

for each $x \in \mathcal{X}$. We here assumed sufficient regularity for differentiation and integration to be interchanged. The step length ρ_m to take in the steepest descent direction can be determined using *line search*

$$\rho_m = \arg\min_{\rho} \mathrm{E}[L(Y, f^{(m-1)}(X) - \rho g_m(X)].$$

The ”step” taken at each iteration m is then given by

$$f_m(x) = -\rho_m g_m(x).$$

Performing updates iteratively according to this yields the gradient descent algorithm in function space.

4.2.2.2 Newton's Method

At a current estimate of f given by $f^{(m-1)}$, we seek to determine the the optimal step f_m to minimize $R(f^{(m-1)} + f_m)$. That is, at step m we are trying to solve

$$\frac{\partial}{\partial f_m(x)} \mathrm{E}[L(Y, f^{(m-1)}(x) + f_m(x))|X = x] = 0 \tag{4.7}$$

for each $x \in \mathcal{X}$. By doing a second-order Taylor expansion of $\mathrm{E}[L(Y, f^{(m-1)}(x) + f_m(x))|X = x]$ around $f^{(m-1)}(x)$ we get

$$\begin{aligned} \mathrm{E}[L(Y, f^{(m-1)}(x) + f_m(x))|X = x] \approx & \mathrm{E}[L(Y, f^{(m-1)}(x))|X = x] + \\ & g_m(x) f_m(x) + \frac{1}{2} h_m(x) f_m(x)^2, \end{aligned}$$

where $h_m(x)$ is the Hessian at the current estimate

$$\begin{aligned} h_m(x) &= \left[\frac{\partial^2 R(f(x))}{\partial f(x)^2}\right]_{f(x)=f^{(m-1)}(x)} \\ &= \left[\frac{\partial^2 \mathrm{E}[L(Y, f(x))|X = x]}{\partial f(x)^2}\right]_{f(x)=f^{(m-1)}(x)} \\ &= \mathrm{E}\left[\frac{\partial^2 L(Y, f(x))}{\partial f(x)^2}\Big| X = x\right]_{f(x)=f^{(m-1)}(x)} \end{aligned}$$

where we again assume sufficient regularity conditions for differentiation and integration to be interchanged. Plugging this into Equation 4.7 we get

$$\frac{\partial}{\partial f_m(x)} \mathrm{E}[L(Y, f^{(m-1)}(x) + f_m(x))|X = x] \approx g_m(x) + h_m(x) f_m(x) = 0.$$

The solution to this is given by

$$f_m(x) = -\frac{g_m(x)}{h_m(x)},$$

which determines the Newton “step” in function space.

4.3 Boosting Algorithms

As seen from the previous sections, boosting fits ensemble models of the kind

$$f(x) = \sum_{m=0}^{M} f_m(x).$$

These can be rewritten as adaptive basis function models

$$f(x) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x),$$

where $f_0(x) = \theta_0$ and $f_m(x) = \theta_m \phi_m(x)$ for $m = 1, ...M$.

Boosting works by using a base learner $\mathcal{L}_\Phi$ to sequentially add basis functions, sometimes called *base models*, $\phi_1, ..., \phi_M \in \Phi$ that improves the fit of the current model.

Most boosting algorithms can be seen to solve

$$\{\hat{\theta}_m, \hat{\phi}_m\} = \underset{\{\theta_m, \phi_m\}}{\arg\min} \sum_{i=1}^{n} L(y_i, \hat{f}^{(m-1)}(x_i) + \theta_m \phi_m(x_i)) \tag{4.8}$$

either exactly or approximately at each iteration. AdaBoost solves Equation 4.8 exactly for the exponential loss function under the constraint that ϕ_m are classifiers with $\mathcal{A} = \{-1, 1\}$. Gradient boosting and Newton boosting on the other hand can be viewed as general algorithms that solve Equation 4.8 approximately for any suitable loss function. We will now show how gradient boosting and Newton boosting can be viewed as empirical versions of the numerical optimization algorithms we developed in Section 4.2.2.

4.3.1 Gradient Boosting

We will here develop gradient boosting. The development is based on gradient descent in function space that we derived in Section 4.2.2.1. Here, the empirical risk will take the place of the true risk used in Section 4.2.2.1. Thus, contrary to the gradient descent in function space, this procedure will be of practical use as it can learn from data.

The empirical version of the negative gradient in Equation 4.6 is given by

$$\begin{aligned} -\hat{g}_m(x_i) &= -\left[\frac{\partial \hat{R}(f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)} \\ &= -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)} \end{aligned}$$

Note that this empirical gradient is only defined at the data points $\{x_i\}_{i=1}^n$. Thus, to generalize to other points in $\mathcal{X}$ and prevent overfitting, we need to learn an approximate negative gradient using a restricted set of possible functions. We thus constrain the set of possible solutions to a set of basis functions Φ. At iteration m, the basis function $\phi_m \in \Phi$ is learnt from the data. The basis function we seek should produce output $\{\phi_m(x_i)\}_{i=1}^n$ which is most highly correlated with the negative gradient $\{-\hat{g}_m(x_i)\}_{i=1}^n$. This is obtained by

$$\hat{\phi}_m = \arg\min_{\phi \in \Phi, \beta} \sum_{i=1}^{n} \Big[\big(-\hat{g}_m(x_i)\big) - \beta\phi(x_i)\Big]^2$$

The basis function ϕ_m is learnt using a base learner where the squared error loss is used as a surrogate loss.

This procedure can be viewed as learning a constrained step direction for gradient descent, where the direction is constrained to be member of a set of basis functions Φ. The step length ρ_m to take in this step direction can subsequently be determined using *line search*

$$\hat{\rho}_m = \arg\min_{\rho} \sum_{i=1}^{n} L(y_i, \hat{f}^{(m-1)}(x_i) + \rho\hat{\phi}_m(x_i)).$$

Friedman (2001) also introduced *shrinkage*, where the step length at each iteration is multiplied by some factor $0 < \eta \leq 1$. This can be viewed as a regularization technique, as the components of the model are shrunk towards zero. The factor η is sometimes referred to as the *learning rate* as lowering it can slow down learning.

Combining all this, the "step" taken at each iteration m is given by

$$\hat{f}_m(x) = \eta\hat{\rho}_m\hat{\phi}_m(x),$$

where $0 < \eta \leq 1$ is the learning rate. Doing this iteratively yields the gradient boosting procedure, which is outlined in Algorithm 1.

The resulting model can be written as

$$\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x).$$

This can be seen as an adaptive basis function model with $\hat{f}_m(x) = \hat{\theta}_m\hat{\phi}_m(x)$ where $\hat{\theta}_m = \eta\hat{\rho}_m$ for $m = 1, ..., M$. The procedure is typically initialized using a constant, i.e. $f_0(x) = \theta_0$, where

$$\hat{\theta}_0 = \arg\min_{\theta} \sum_{i=1}^{n} L(y_i, \theta).$$

4.3.2 Newton Boosting

We will now develop what we term *Newton boosting.* The development is based on Newton's method in function space that we derived in Section 4.2.2.2, but the risk will now be exchanged with the empirical risk.

```
Algorithm 1: Gradient boosting
Input  : Data set $\mathcal{D}$.
         A loss function $L$.
         A base learner $\mathcal{L}_\Phi$.
         The number of iterations $M$.
         The learning rate $\eta$.
1 Initialize $\hat{f}^{(0)}(x) = \hat{f}_0(x) = \hat{\theta}_0 = \arg\min_{\theta} \sum_{i=1}^{n} L(y_i, \theta)$;
2 for $m$ = 1,2,..,$M$ do
3     $\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
4     $\hat{\phi}_m = \arg\min_{\phi \in \Phi, \beta} \sum_{i=1}^{n} \left[\big(-\hat{g}_m(x_i)\big) - \beta\phi(x_i)\right]^2$;
5     $\hat{\rho}_m = \arg\min_{\rho} \sum_{i=1}^{n} L(y_i, \hat{f}^{(m-1)}(x_i) + \rho\hat{\phi}_m(x_i))$;
6     $\hat{f}_m(x) = \eta\hat{\rho}_m\hat{\phi}_m(x)$;
7     $\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$;
8 end
Output: $\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x)$
```

Similar to the case for gradient boosting, we have that the empirical gradient is defined solely at the data points. For Newton's method we also need the Hessian. The empirical Hessian

$$\begin{aligned}\hat{h}_m(x_i) &= \left[\frac{\partial^2 \hat{R}(f(x_i))}{\partial f(x_i)^2}\right]_{f(x)=\hat{f}^{(m-1)}(x)} \\ &= \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2}\right]_{f(x)=\hat{f}^{(m-1)}(x)}.\end{aligned}$$

is also defined solely at the data points. We thus also need a base learner here to select a basis function from a restricted set of functions. The Newton "step" is found by solving

$$\hat{\phi}_m = \arg\min_{\phi\in\Phi} \sum_{i=1}^{n} \Big[\hat{g}_m(x_i)\phi(x_i) + \frac{1}{2}\hat{h}_m(x_i)\phi(x_i)^2\Big].$$

By completing the square, this can be rewritten as

$$\hat{\phi}_m = \arg\min_{\phi\in\Phi} \sum_{i=1}^{n} \frac{1}{2}\hat{h}_m(x_i)\Big[\big(-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)}\big) - \phi(x_i)\Big]^2.$$

Newton boosting thus amounts to a weighted least-squares regression problem at each iteration, which is solved using the base learner.

The "step" taken at each iteration m is given by

$$\hat{f}_m(x) = \eta\hat{\phi}_m(x),$$

where $0 < \eta \leq 1$ is the learning rate. Repeating this iteratively yields the Newton boosting procedure, which is outlined in Algorithm 2.

The resulting model can be written as

$$\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x).$$

This can be seen as an adaptive basis function model with $\hat{f}_m(x) = \hat{\theta}_m\hat{\phi}_m(x)$ where $\hat{\theta}_m = \eta$ for $m = 1, ..., M$. The procedure can be initialized using a constant, i.e. $f_0(x) = \theta_0$, where

$$\hat{\theta}_0 = \arg\min_{\theta} \sum_{i=1}^{n} L(y_i, \theta).$$

Note that the boosting algorithm developed here is the one used by XGBoost (Chen and Guestrin, 2016). This is also the same boosting algorithm which is at the core LogitBoost (Friedman et al., 2000). We choose to term this boosting algorithm "Newton boosting" since it can be viewed as a Newton method in function space. Other names we could have used include "second-order gradient boosting" or perhaps "Hessian boosting".

Algorithm 2: Newton boosting

```
  Input  : Data set $\mathcal{D}$.
           A loss function $L$.
           A base learner $\mathcal{L}_\Phi$.
           The number of iterations $M$.
           The learning rate $\eta$.
1 Initialize $\hat{f}^{(0)}(x) = \hat{f}_0(x) = \hat{\theta}_0 = \arg\min_{\theta} \sum_{i=1}^{n} L(y_i, \theta)$;
2 for $m$ = 1,2,..,$M$ do
3     $\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
4     $\hat{h}_m(x_i) = \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
5     $\hat{\phi}_m = \arg\min_{\phi\in\Phi} \sum_{i=1}^{n} \frac{1}{2}\hat{h}_m(x_i)\Big[\big(-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)}\big) - \phi(x_i)\Big]^2$;
6     $\hat{f}_m(x) = \eta\hat{\phi}_m(x)$;
7     $\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$;
8 end
  Output: $\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x)$
```

4.4 Base Learners

Both gradient and Newton boosting requires a base learner to learn a basis function $\phi \in \Phi$ at each iteration. This base learner should be a regression procedure as will be used to minimize (weighted) mean squared error. Apart from that, there are few requirements imposed on the base learner. The base learner is however typically chosen to be simple, i.e. have high bias, but low variance.

Bühlmann and Hothorn (2007) describes using componentwise linear models to fit linear models and componentwise smoothing splines to fit additive models. This allows one to fit potentially sparse linear or additive models where regularization is achieved through early stopping rather than penalization. While these are powerful ways to fit linear or additive models, the resulting model will still be linear or additive and thus have limited representational ability.

The most common choice of base learner for gradient and Newton boosting is a regression tree algorithm such as CART, which we will describe in Chapter 5. Friedman (2001) proposed a special enhancement to the gradient boosting procedure when the base learner is CART. We will continue the discussion of tree boosting methods in Chapter 6.

4.5 Hyperparameters

As seen from the previous sections, the two main hyperparameters of both gradient and Newton boosting are the number of iterations or basis functions M and the learning rate or shrinkage parameter η. These parameters are not independent and have to be selected jointly. We will first assume η is held fixed and discuss the effect of the number of iterations.

4.5.1 The Number of Iterations M

As the number of iterations M is increased, the complexity of the model will tend to increase. This is not very surprising as the basis function expansion will tend to have greater representational ability when the number of basis functions increases. Therefore, at some point, increasing the number of iterations further will lead to overfitting. Consequently, regularization can be achieved through early stopping (Zhang and Yu, 2005). A suitable number of iterations M is commonly determined by monitoring prediction accuracy on a validation set or through cross-validation.

4.5.2 The Learning Rate η

Friedman (2001) empirically found that smaller values of η tended to improve generalization performance. By decreasing η however, the number of iterations M required is typically increased. Thus, lowering η comes at the cost of greater computational demand.

Bühlmann and Yu (2010) suggests that the choice of η is not crucial, as long as it is sufficiently small, such as $\eta = 0.1$. Ridgeway (2006) suggests that the learning rate should be as small as possible since lower values of η tend to improve generalization performance. The marginal improvements are however decreasing. As such, one should set the learning rate η as low as you can "afford" computationally and then determining the optimal number of iterations for that η.

To get an idea of the effect of the number of iterations M and the learning rate η, we used the *xgboost* package (Chen et al., 2016) to fit the Boston Housing dataset (Lichman, 2013). The response variable *medv* was log-scaled. We used tree models with different values for the learning rate and set the number of iterations to $M = 10000$. Predictions were made for each iteration, thus giving 10000 predictions for models of increasing complexity. To get a fairly stable estimate of the out-of-sample performance, three repetitions of 10-fold cross-validation was used. The folds were the same for all learning rates. The out-of-sample RMSE (root mean squared error) was plotted against the number of iterations for the different learning rates. The result can be seen in Figure 4.1.

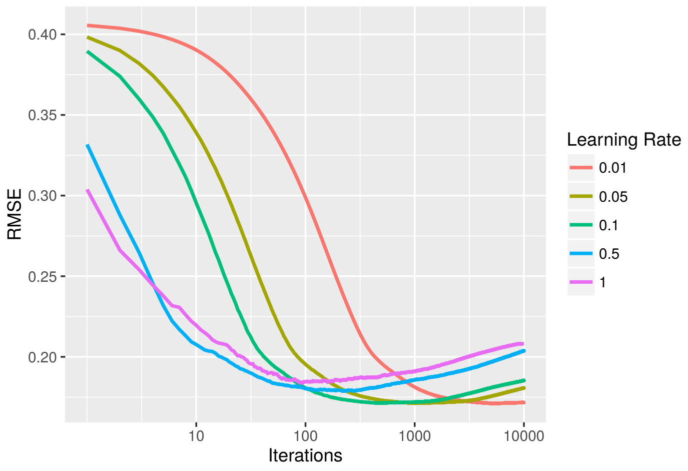


Figure 4.1: Out-of-sample RMSE for different learning rates on the Boston Housing dataset.

Chapter 5

Tree Methods

Tree models are simple, interpretable models. As the name suggests, the model takes a form that can be visualized as a tree structure. One example is shown in Figure 5.1a.

The node at the top of the tree is called the *root node*. This node has branches to nodes below it. The nodes in the tree which have branches below them are called *internal nodes* or *splits*. The nodes at the bottom of the tree are called *terminal nodes* or *leaves*. We will focus on *binary trees* where each internal node only have two branches.

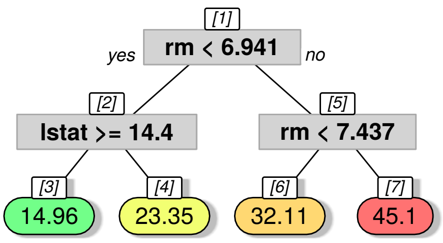
Tree models do unfortunately usually have limited predictive power. When multiple tree models are combined together as in bagged trees (Breiman, 1996), Random Forests (Breiman, 2001) or in boosting algorithms however, they tend to have very good predictive capabilities. We will here focus on the properties of tree models which are important for understanding their role in tree boosting methods.

5.1 Model Class

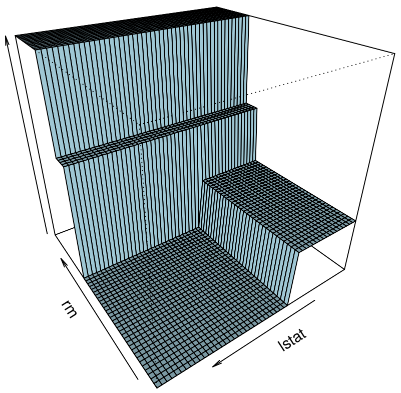
Tree models partition the feature space $\mathcal{X}$ into a set of T rectangular, non-overlapping regions $R_1, ..., R_T$ and fit a simple model in each region, such as a constant. One can also fit e.g. linear models in each region as described by Quinlan (1992). The most common choice is however to use a constant model, which is what we will focus on here. When a constant model is used, tree models can be written in the form

$$f(x) = \sum_{j=1}^{T} w_j \mathrm{I}(x \in R_j).$$

They can thus be viewed as adaptive basis function models in the form shown in Equation 2.5 with T basis functions or terminal nodes, where $\theta_j = w_j$ is the constant fit in each terminal node and $\phi_j(x) = \mathrm{I}(x \in R_j)$ is the basis function indicating whether observation x belongs in terminal node j. Note that the tree model is essentially just a piecewise constant function. This is shown in Figure 5.1b.



(a) Visualization of tree.



(b) Fitted function.

Figure 5.1: Visualization of tree model fit to the Boston Housing data.

This shows the fitted function of the tree shown in Figure 5.1a. This tree was fit to the Boston Housing data using only the predictors *lstat* and *rm* for visualization purposes.

5.2 Regularization

Regularization for tree models is commonly achieved by constraining or penalizing the complexity of the tree.

5.2.1 Model Complexity

There are multiple possible ways one could define the complexity of a tree model. The complexity can be seen to depend on the depth of the tree or the number of terminal nodes of a tree.

Complexity also generally depends on the size of the local neighbourhoods. In the case of trees, this is the size of the regions $R_1, ..., R_T$. That is, trees with smaller regions can fit local structure more closely and are thus more complex.

The complexity can also be seen to be related to the relative difference of the leaf weights $w_1, ..., w_T$. To see why this is the case, consider first a tree where all the weights are identical. This would be a globally constant model. If the weights are wildly different on the other hand, the tree model can be taken to be more complex. Defining complexity in this way is not common for individual tree models, but is used by XGBoost when fitting additive tree models.

5.2.2 Complexity Constraints

To control the flexibility when fitting a tree model, we can constrain the complexity of the tree. There are multiple possible ways to constrain the complexity of tree models. The most obvious and common way to do so is to constrain the number of terminal nodes in the tree to some maximum allowed number of terminal nodes $T_{\max}$.

Another way one could constrain the complexity of the tree fit is to restrict the minimum number of observations allowed in each terminal node $n_{\min}$. This will impose a limit on how small the neighbourhoods are allowed to be. This will also directly limit the variance of the fit as more observations will be required to estimate the leaf weights.

5.2.3 Complexity Penalization

The most common way to penalize tree complexity is to penalize the number of terminal nodes T. Breiman et al. (1984) introduced *cost-complexity pruning* which relies on this form of penalization. The objective to be minimized is given by the *cost-complexity criterion*

$$J(f) = \hat{R}(f) + \Omega(f) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \sum_{j=1}^{T} w_j \mathrm{I}(x_i \in R_j)) + \gamma T, \tag{5.1}$$

where γ is a hyperparameter known as the *complexity parameter*. We will discuss *pruning* in Section 5.3.5.

Another way to penalize tree complexity is to introduce penalization of the leaf weights $w_1, ..., w_T$. This form of penalization is used by XGBoost, but is not commonly used for single tree models. We will therefore delay discussion of this to Section 6.2.3.

5.3 Learning Algorithms

For simplicity, we will here consider learning a tree without penalization. The objective will therefore simply be the empirical risk of the tree model f, which can be written as

$$\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(x_i)) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \sum_{j=1}^{T} w_j \mathrm{I}(x_i \in R_j)). \tag{5.2}$$

Minimizing this objective function is typically computationally infeasible. Learning the optimal weights $w_1, ..., w_T$ is typically easy given the regions $R_1, ..., R_T$. Learning the optimal regions however, i.e. learning the *structure* of the tree, is hard. In fact, the problem is NP-complete (Hyafil and Rivest, 1976). Consequently, one has to simplify the problem by instead computing an approximate solution.

The are many different learning algorithms for learning tree models. Examples include CART (Classification And Regression Trees) (Breiman et al., 1984), Conditional Inference Trees (Torsten Hothorn, 2006), C4.5 (Quinlan, 1993) and CHAID (Kass, 1980). The tree boosting algorithm MART makes use CART, while XGBoost uses an algorithm closely related to CART. We will therefore focus on CART here.

CART *grows* the tree greedily in a top-down fashion using binary splits. The growing begins with only the root node. Every split parallel to the coordinate axes are considered and the split minimizing the objective is chosen. Next, every split parallel to the coordinate axes within each of the current regions are considered. The best split is chosen and this procedure is repeated until some stopping criterion is applied.

We will now discuss learning of the weights given the structure and learning the structure, followed by possible ways of dealing with missing values and categorical predictors.

5.3.1 Learning the Weights for a Given Structure

Given a region R_j, learning the weight w_j is typically straightforward. Let I_j denote the set of indices that belongs to region R_j, i.e. $x_i \in R_j$ for $i \in I_j$.

The weight is estimated by

$$\hat{w}_j = \arg\min_w \sum_{i \in I_j} L(y_i, w).$$

For the squared error loss for example, the estimated weight will simply be the average of the responses in the region. For the absolute loss on the other hand, the estimated weight will be the median of the responses.

5.3.2 Learning the Structure

For a tree model $\hat{f}$, the empirical risk is given by

$$\hat{R}(\hat{f}) = \sum_{j=1}^{T} \sum_{i \in I_j} L(y_i, \hat{w}_j) \equiv \sum_{j=1}^{T} \hat{L}_j,$$

where we let $\hat{L}_j$ denote the aggregated loss at node j.

Consider now that we are in the process of learning a tree and that the current tree model is denoted by $\hat{f}_{\text{before}}$. Let further $\hat{f}_{\text{after}}$ denote the tree model after a considered split at node k into a left node L and right node R is performed. We can write the empirical risk of these models as

$$\hat{R}(\hat{f}_{\text{before}}) = \sum_{j \neq k} \hat{L}_j + \hat{L}_k$$

and

$$\hat{R}(\hat{f}_{\text{after}}) = \sum_{j \neq k} \hat{L}_j + \hat{L}_L + \hat{L}_R.$$

The *gain* of the considered split is defined as

$$Gain = \hat{R}(\hat{f}_{\text{before}}) - \hat{R}(\hat{f}_{\text{after}}) = \hat{L}_k - (\hat{L}_L + \hat{L}_R). \tag{5.3}$$

For each split made, the gain is calculated for every possible split at every possible node and the split with maximal gain is taken.

Note that this is simply empirical risk minimization where the optimization problem is discrete. To obtain the optimal tree, every possible tree would have to be constructed and the one with minimal empirical risk selected. The optimization problem is thus simplified by greedily selecting splits which minimize the empirical risk or training error. For squared error loss, the split which minimizes MSE is selected, while for the absolute loss, the split which minimizes MAE is selected.

5.3.3 Missing Values

Many learning algorithms will have problems when faced with missing values. You are typically left with the choice of removing data points with missing values or using some procedure to impute the missing values. See Kuhn and Johnson (2013) for more details. For most tree algorithms however, this is not necessary as they have the ability to deal with missing values during training.

CART handles missing values by using so-called *surrogate variables*. For each predictor, we only use the observations for which that predictor is not missing when searching for a split. Once the primary split is chosen, one forms a list of surrogate predictors and split points. These are chosen to best mimic the split of the training data achieved by the primary split. See Breiman et al. (1984) for more details.

The tree growing algorithm used by XGBoost treats missing values by learning *default directions*. At each node there are two possible directions, left or right. When data is missing, the default direction is taken. When there is missing data during training, the direction which minimizes the objective is learnt from the data. When missing data is not present, the default direction is set to some default. See Chen and Guestrin (2016) for more details.

5.3.4 Categorical Predictors

So far, we have assumed that the predictors were either continuous or binary. When multicategorical predictors are present however, one can treat these in two different ways. Kuhn and Johnson (2013) refer to these as *grouped categories* and *independent categories*.

When using grouped categories, we treat the categorical predictor as a single entity. Thus when considering possible splits of a categorical predictor, one has to consider every way to split the categories into two groups. For a categorical predictor with K categories or classes, there are $2^{K-1} - 1$ possible partitions into two groups (Hastie et al., 2009). The number of possible partitions becomes very large for large K and we are thus likely to find a partitioning of the data that looks good, possibly leading to overfitting.

When using independent categories, a categorical predictor is encoded as K binary predictors, one for each class. This is referred to as *dummy encoding* or *one-hot encoding*. This imposes a "one-versus-all" split of the categories.

Kuhn and Johnson (2013) find that either of the approaches sometimes works better than the other and thus suggest trying both approaches during model building.

CART handles grouped categories, while the tree algorithm used by XGBoost requires independent categories.

5.3.5 Pruning

Consider now growing a tree using a penalized objective such as the one in Equation 5.1. If the maximum gain achievable through an additional split is negative, one is unable to further reduce the empirical risk through the split. One might consider stopping the tree growing. This might however be short-sighted as the potential splits further down in tree might have positive gains which makes up for the negative gain further up.

To prevent this one typically grows the tree until some stopping criteria is met and then employ *pruning* to remove nodes with negative gain. This is done in a bottom-up fashion, successively collapsing internal nodes with negative gain. When this is done according to the cost-complexity criterion in Equation 5.1, this is referred to as *cost-complexity pruning*.

One possible stopping criteria is that the maximum number of terminal nodes T_{max} is reached. Another is that further splitting would leave a terminal node with fewer number of observations than the minimum allowed n_{min}.

5.4 Benefits and Drawbacks

Hastie et al. (2009) describes tree methods as a good choice for an "off-the-shelf" method for data mining due to their many benefits. While they do have many benefits making them applicable to many data sets without the need for pre-processing, they do also have many drawbacks such as limited predictive performance. In Chapter 6, we will discuss additive tree models where tree models are used as base learners in boosting algorithms. These models inherit many of the benefits of tree models, while eliminating or reducing many of the drawbacks.

5.4.1 Benefits

Some of the benefits of tree methods are that they (Hastie et al., 2009; Murphy, 2012)

- are easily interpretable.
- are relatively fast to construct.
- can naturally deal with both continuous and categorical data.

- can naturally deal with missing data.
- are robust to outliers in the inputs.
- are invariant under monotone transformations of the inputs.
- perform implicit variable selection.
- can capture non-linear relationships in the data.
- can capture high-order interactions between inputs.
- scale well to large data sets.

5.4.2 Drawbacks

Some of the drawbacks of tree methods are that they (Hastie et al., 2009; Kuhn and Johnson, 2013; Wei-Yin Loh, 1997; Strobl et al., 2006)

- tend to select predictors with a higher number of distinct values.
- can overfit when faced with predictors with many categories.
- are unstable and have high variance.
- lack smoothness.
- have difficulty capturing additive structure.
- tend to have limited predictive performance.

Chapter 6

Tree Boosting Methods

Using trees as base models for boosting is a very popular choice. Seeing how trees have many benefits that boosted trees inherit while the predictive ability is greatly increased through boosting, this is perhaps not very surprising. The main drawback of boosted tree models compared to single tree models is that most of the interpretability is lost.

6.1 Model Class

Boosted tree models can be viewed as adaptive basis function models of the form in Equation 2.5, where the basis functions are regression trees. Regression trees can however further be viewed as adaptive basis function models. We can thus collect the constant terms as

$$\begin{aligned}
f(x) &= \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x) \\
&= \theta_0 + \sum_{m=1}^{M} \theta_m \sum_{j=1}^{T_m} \tilde{w}_{jm} \mathrm{I}(x \in R_{jm}) \\
&= \theta_0 + \sum_{m=1}^{M} \sum_{j=1}^{T_m} w_{jm} \mathrm{I}(x \in R_{jm}) \\
&= \theta_0 + \sum_{m=1}^{M} f_m(x).
\end{aligned} \tag{6.1}$$

As seen from this, boosting tree models results in a sum of multiple trees $f_1, ..., f_M$. Boosted tree models are therefore also referred to as *tree ensembles* or *additive tree models*. An additive tree model fit to the Boston Housing data is shown in Figure 6.1. This is fit to the same data as the tree model shown in Figure 5.1b. It is immediately apparent that the fit is much smoother than that of a single tree model.

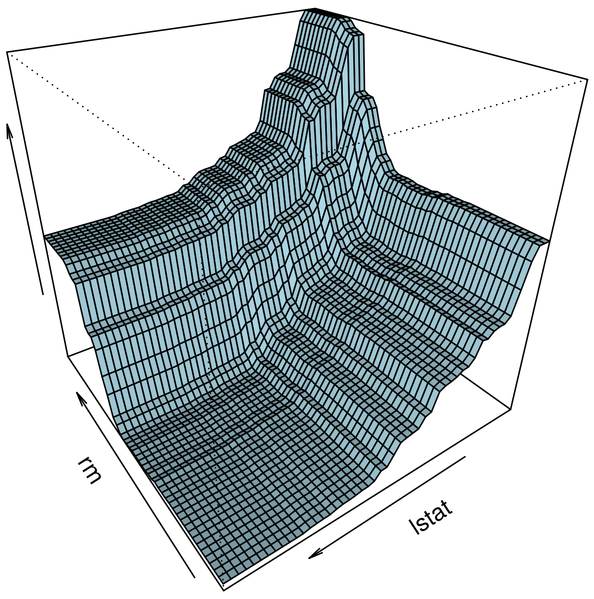


Figure 6.1: Visualization of an additive tree model fit to the Boston Housing data.

6.2 Regularization

Regularization of additive tree models can be achieved in many ways. First of all, one can regularize the basis function expansion. Second of all, one can regularize each of the basis functions, i.e. the individual tree models. Finally, one can also introduce regularization through randomization. This was first introduced by Friedman (2002) through subsampling. Note that this form of regularization is not necessarily restricted only to tree boosting, but could potentially be used with boosting in general. It is however especially useful for trees as they typically benefit the most from it. To keep discussion simple, we thus chose to delay the discussion of randomization to this section.

In this section, we will introduce the different regularization techniques used for additive tree models. In Chapter 8, we will go more in depth and discuss the effects the various regularization parameters have on the model.

6.2.1 Model Complexity

The complexity of an additive tree model can be seen to depend on many parameters. First of all, the number of trees is clearly related to the complexity.

For additive tree models, we also have the complexities of each of the individual trees. When defining the complexity of an additive tree model we can thus additionally take the complexities of the individual trees, as discussed in Section 5.2.1, into account.

6.2.2 Complexity Constraints

Constraining the complexity of an additive tree model amounts to constraining the complexity of the basis function expansion and each of the basis functions. That is, we can for example constrain the number of trees. We can further constrain the complexity of an additive tree model by restricting the maximum number of terminal nodes of each individual tree. Another way is to limit the minimum number of observations falling in any terminal node. This was discussed in Section 5.2.2.

For additive tree models, *shallow* regression trees are commonly used, i.e. regression trees with few terminal nodes. Shallow trees have low variance, but high bias compared to deeper trees. This makes sense as boosting iteratively fits new trees to reduce errors of the previous trees. As the number of boosting iterations increases, the variance thus tends to increase, while the bias tends to decrease.

6.2.3 Complexity Penalization

Both MART and XGBoost offers the possibility of constraining the complexity of the individual trees. XGBoost does however additionally offer the possibility of penalizing the complexity of the trees. Before XGBoost, complexity penalization was not commonly used for additive tree models. This is one of the core improvements of XGBoost over MART.

The penalization terms of the objective function can be written

$$\Omega(f) = \sum_{m=1}^{M} \Big[\gamma T_m + \frac{1}{2}\lambda \|w_m\|_2^2 + \alpha \|w_m\|_1\Big]. \tag{6.2}$$

The penalty is the sum of the complexity penalties of the individual trees in the additive tree model.

We see that the regularization term includes penalization of the number of terminal nodes of each individual tree through γ. This is equivalent to the cost-complexity criterion discussed in Section 5.2.3 for each individual tree. Additionally, the objective includes $l2$ regularization of the leaf weights. These two penalization terms are described by Chen and Guestrin (2016). The last term in Equation 6.2 is $l1$ regularization on the term weights. This is indeed also implemented in XGBoost [1]. We will discuss the effect of these regularization parameter further in Chapter 8.

[1] *https://github.com/dmlc/xgboost/blob/master/src/tree/param.h*

6.2.4 Randomization

Friedman (2002) proposed an extension to gradient boosting which he called *stochastic gradient boosting*. The extension was to include (row) subsampling at each boosting iteration. The idea was that introducing randomness in the learning procedure could improve generalization performance. This was inspired by the *bagging* method of Breiman (1996).

6.2.4.1 Row Subsampling

Bagging can be used to improve generalization performance for some learning algorithms. It works by drawing bootstrap samples (Efron, 1979) from the data and fitting a model using the same learning algorithm to each one. The fitted models are then combined by simple averaging to form an ensemble model. The models will differ due to being fitted to slightly altered versions of the data set. The averaging thus tend to result in reduced variance of the ensemble model. This has empirically proven to be an effective way of improving performance of some model classes such as tree models.

Friedman (2002) proposed using subsampling at each boosting iteration to fit a random subsample of the data at each iteration. This was found to yield performance improvements for many data sets. Note that subsampling draws a random sample of the data *without* replacement, while bootstrapping draws a random sample of the data *with* replacement.

Following Chen and Guestrin (2016), we will refer to this as *row subsampling*. The row subsampling fraction $0 < \omega_r \leq 1$ thus becomes a hyperparameter for the boosting procedure. Setting $\omega_r = 1$ results in the original procedure without row subsampling.

6.2.4.2 Column Subsampling

Another way to introduce randomness in a learning procedure is by randomly sampling the predictors. This is sometimes called the Random Subspace Method (Ho, 1998). Breiman (2001) combined this idea with tree bagging and thereby introduced Random Forests. Specifically, Random Forests combines bagging with a randomized tree learning algorithm that only considers a random subset of the predictors each time a split is considered. This decorrelates the fitted tree models such that the variance is further reduced through averaging. The simplicity of Random Forests together with their usually good performance has made them very popular in practice.

Following Chen and Guestrin (2016), we will refer to this as *column subsampling*. The column subsampling fraction $0 < \omega_c \leq 1$ thus becomes a hyperparameter for the boosting procedure. Setting $\omega_c = 1$ results in the original procedure without column subsampling.

MART includes row subsampling, while XGBoost includes both row and column subsampling (Chen and Guestrin, 2016).

6.3 Learning Algorithms

MART and XGBoost employ two different boosting algorithms for fitting additive tree models. We will refer to these as *gradient tree boosting* (GTB) and *Newton tree boosting* (NTB), respectively. In this section, we will develop these tree boosting algorithms. For simplicity, we will assume the objective is the empirical risk without any penalization. In Chapter 8, we will extend the discussion to the case where penalization is included.

At each iteration m, both these algorithms seek to minimize the FSAM criterion

$$J_m(\phi_m) = \sum_{i=1}^{n} L(y_i, \hat{f}^{(m-1)}(x_i) + \phi_m(x_i)). \tag{6.3}$$

For the tree boosting algorithms, the basis functions are trees

$$\phi_m(x) = \sum_{j=1}^{T} w_{jm}\mathrm{I}(x \in R_{jm}).$$

While Newton tree boosting is simply Newton boosting with trees as basis functions, gradient tree boosting is a modification of regular gradient boosting to the case where the basis functions are trees. In Chapter 7, we will compare these tree boosting algorithms and discuss the properties in more detail.

We will in this section show how Newton tree boosting and gradient tree boosting learns the tree structure and leaf weights at each iteration. We will do this in three stages. First, we will determine the leaf weights $\tilde{w}_{jm}$ for a proposed (fixed) tree structure. In the next stage, different tree structures are proposed with weights determined from the previous stage. In this stage, the tree structure and thus the regions $\hat{R}_{jm}, j = 1, ..., T$ are determined. Finally, once a tree structure is settled upon, the final leaf weights $\hat{w}_{jm}, j = 1, ..., T$ in each terminal node are determined.

6.3.1 Newton Tree Boosting

The Newton tree boosting algorithm is simply the Newton boosting shown in Algorithm 2 where the basis functions are tree models. As discussed in Section 4.3.2, Newton boosting approximates the criterion in Equation 6.3 by

$$\tilde{J}_m(\phi_m) = \sum_{i=1}^{n} \Big[\hat{g}_m(x_i)\phi_m(x_i) + \frac{1}{2}\hat{h}_m(x_i)\phi_m(x_i)^2\Big], \tag{6.4}$$

which is the second-order approximation. We will here show how to learn the structure and leaf weights of the tree according to the criterion in Equation 6.4.

6.3.1.1 Learning the Weights for a Given Structure

We first rewrite the criterion in Equation 6.4 with trees as basis functions,

$$\tilde{J}_m(\phi_m) = \sum_{i=1}^{n} \Big[\hat{g}_m(x_i)\sum_{j=1}^{T} w_{jm}\mathrm{I}(x_i \in R_{jm}) + \frac{1}{2}\hat{h}_m(x_i)\big(\sum_{j=1}^{T} w_{jm}\mathrm{I}(x_i \in R_{jm})\big)^2\Big].$$

Due to the disjoint nature of the regions of the terminal nodes, we can rewrite this criterion as

$$\begin{aligned}\tilde{J}_m(\phi_m) &= \sum_{i=1}^{n}\Big[\hat{g}_m(x_i)\sum_{j=1}^{T} w_{jm}\mathrm{I}(x_i \in R_{jm}) + \frac{1}{2}\hat{h}_m(x_i)\sum_{j=1}^{T} w_{jm}^2\mathrm{I}(x_i \in R_{jm})\Big] \\ &= \sum_{j=1}^{T}\sum_{i\in I_{jm}}\Big[\hat{g}_m(x_i)w_{jm} + \frac{1}{2}\hat{h}_m(x_i)w_{jm}^2\Big],\end{aligned}$$

where I_{jm} denote the set of indices of the x_i falling in region R_{jm}. Letting $G_{jm} = \sum_{i\in I_{jm}} \hat{g}_m(x_i)$ and $H_{jm} = \sum_{i\in I_{jm}} \hat{h}_m(x_i)$, we can rewrite this as

$$\tilde{J}_m(\phi_m) = \sum_{j=1}^{T}\Big[G_{jm}w_{jm} + \frac{1}{2}H_{jm}w_{jm}^2\Big]. \tag{6.5}$$

For a proposed, fixed structure, the weights are thus given by

$$\tilde{w}_{jm} = -\frac{G_{jm}}{H_{jm}}, \quad j = 1, ..., T. \tag{6.6}$$

6.3.1.2 Learning the Structure

As discussed in Section 5.3, learning the structure of a tree amounts to searching for splits. For each split, a series of candidate splits are proposed, and the one which minimizes empirical risk is chosen. Equivalently, we seek the split which maximizes the gain, which is empirical risk reduction of a proposed split. For Newton tree boosting, we seek the split which minimizes the criterion in Equation 6.4.

Plugging the weights from Equation 6.6 into the empirical risk in Equation 6.5, we find the criterion for a fixed structure to be

$$\tilde{J}_m(\tilde{\phi}_m) = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_{jm}^2}{H_{jm}}.$$

When searching for the optimal split, this is the criterion we seek to minimize. That is, the splits are determined by maximizing the gain given by

$$Gain = \frac{1}{2}\Big[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}\Big]. \tag{6.7}$$

6.3.1.3 Learning the Final Weights

For Newton tree boosting, the final weights are already found when searching for the structure. That is, the final weights are simply given by

$$\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}}, \quad j = 1, ..., T, \tag{6.8}$$

for the learnt structure.

6.3.1.4 The Algorithm

Summing up these steps, we get the Newton tree boosting algorithm, which is outlined in Algorithm 3.

Algorithm 3: Newton tree boosting

```
Input  : Data set $\mathcal{D}$.
         A loss function $L$.
         The number of iterations $M$.
         The learning rate $\eta$.
         The number of terminal nodes $T$
1 Initialize $\hat{f}^{(0)}(x) = \hat{f}_0(x) = \hat{\theta}_0 = \arg\min_{\theta} \sum_{i=1}^{n} L(y_i, \theta)$;
2 for $m$ = 1,2,..,$M$ do
3     $\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
4     $\hat{h}_m(x_i) = \left[\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
5     Determine the structure $\{\hat{R}_{jm}\}_{j=1}^{T}$ by selecting splits which maximize
        $Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}\right]$;
6     Determine the leaf weights $\{\hat{w}_{jm}\}_{j=1}^{T}$ for the learnt structure by
        $\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}}$;
7     $\hat{f}_m(x) = \eta \sum_{j=1}^{T} \hat{w}_{jm} \mathrm{I}(x \in \hat{R}_{jm})$;
8     $\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$;
9 end
  Output: $\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x)$
```

6.3.2 Gradient Tree Boosting

The gradient tree boosting algorithm is closely related to the gradient boosting algorithm shown in Algorithm 1. In line 4, a tree is learnt using the criterion

$$\tilde{J}_m(\phi_m) = \sum_{i=1}^{n} \Big[\big(-\hat{g}_m(x_i)\big) - \beta\phi(x_i)\Big]^2, \tag{6.9}$$

which is an approximation to the criterion in Equation 6.3. Next, in line 5, a line search step is performed. Friedman (2001) did however suggest an enhancement to this step in the case where the base models are trees. The gradient tree boosting algorithm is thus not just a special case of gradient boosting where the base models are trees, but a slightly different algorithm which is based on the general gradient boosting algorithm.

We will here develop the gradient tree boosting algorithm in a similar fashion we did for the Newton tree boosting algorithm in the previous section. That is, we will show how the structure and leaf weights of the tree is learnt in three stages. Apart from the final stage, the development is very similar, and will hence be shortened here.

6.3.2.1 Learning the Weights for a Given Structure

Doing a development similar to the one for Newton tree boosting, we can rewrite the criterion in Equation 6.9 as

$$\tilde{J}_m(\phi_m) = \sum_{j=1}^{T} \Big[G_{jm} w_{jm} + \frac{1}{2} n_{jm} w_{jm}^2 \Big], \tag{6.10}$$

where n_{jm} denote the number of points x_i falling in region R_{jm}. For a proposed, fixed structure, the weights are thus given by

$$\tilde{w}_{jm} = -\frac{G_{jm}}{n_{jm}}, \quad j = 1, ..., T. \tag{6.11}$$

6.3.2.2 Learning the Structure

Plugging the weights from Equation 6.11 into the empirical risk in Equation 6.10, we find the criterion for a fixed structure to be

$$\tilde{J}_m(\tilde{\phi}_m) = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_{jm}^2}{n_{jm}}.$$

The gain used to determine the splits are thus given by

$$Gain = \frac{1}{2} \Big[\frac{G_L^2}{n_L} + \frac{G_R^2}{n_R} - \frac{G_{jm}^2}{n_{jm}} \Big]. \tag{6.12}$$

6.3.2.3 Learning the Final Weights

In the general gradient boosting in Algorithm 1, a line search step is performed to determine the “step” length to take in function space. For gradient tree boosting however, Friedman (2001) presented a special enhancement. He noted that an additive tree model can be considered a basis function expansion where the basis functions are themselves basis function expansions. He further took the view that at each iteration, the algorithm was fitting T separate basis functions, one for each region of the tree. He therefore proposed to do T line search steps, one for each region $R_1, ..., R_T$, instead of one for the whole tree.

The optimization problem to be solved in the line search step is thus

$$\{\hat{w}_{jm}\}_{j=1}^{T} = \underset{\{w_j\}_{j=1}^{T}}{\arg\min} \sum_{i=1}^{n} L(y_i, \hat{f}^{(m-1)}(x_i) + w_j \mathrm{I}(x_i \in \hat{R}_{jm})).$$

Note however that this can be rewritten as T disjoint optimization problems

$$\hat{w}_{jm} = \underset{w_j}{\arg\min} \sum_{i \in \hat{I}_{jm}} L(y_i, \hat{f}^{(m-1)}(x_i) + w_j), \quad j = 1, ..., T. \tag{6.13}$$

The final leaf weights are thus determined using T separate line search steps. This greatly simplifies the line search step, which will often be difficult otherwise.

For e.g. the squared error loss, the final leaf weight in a region will be the average of the negative gradients in that region. For the absolute loss, the final leaf weight will be the median of the negative gradients. In some cases, the optimization problem in Equation 6.13 does not have a closed form solution. One example is the log-loss. Calculating the final leaf weights would thus require numerical optimization. Instead of doing full numerical optimization for each leaf at each iteration, which will typically be computationally expensive, Friedman (2001) suggested to simply perform a single iteration of Newton's method.

6.3.2.4 The Algorithm

Summing up these steps, we get the gradient tree boosting algorithm, which is outlined in Algorithm 4.

Algorithm 4: Gradient tree boosting

```
  Input  : Data set $\mathcal{D}$.
           A loss function $L$.
           The number of iterations $M$.
           The learning rate $\eta$.
           The number of terminal nodes $T$
1 Initialize $\hat{f}^{(0)}(x) = \hat{f}_0(x) = \hat{\theta}_0 = \arg\min_{\theta} \sum_{i=1}^{n} L(y_i, \theta)$;
2 for $m$ = 1,2,..,$M$ do
3     $\hat{g}_m(x_i) = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)}$;
4     Determine the structure $\{\hat{R}_{jm}\}_{j=1}^{T}$ by selecting splits which maximize
        $Gain = \frac{1}{2}\left[\frac{G_L^2}{n_L} + \frac{G_R^2}{n_R} - \frac{G_{jm}^2}{n_{jm}}\right]$;
5     Determine the leaf weights $\{\hat{w}_{jm}\}_{j=1}^{T}$ for the learnt structure by
        $\hat{w}_{jm} = \arg\min_{w_j} \sum_{i \in \hat{I}_{jm}} L(y_i, \hat{f}^{(m-1)}(x_i) + w_j)$;
6     $\hat{f}_m(x) = \eta \sum_{j=1}^{T} \hat{w}_{jm} \mathrm{I}(x_i \in \hat{R}_{jm})$;
7     $\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$;
8 end
  Output: $\hat{f}(x) \equiv \hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x)$
```

Part III

Discussion and Comparison

Chapter 7

Comparison of Tree Boosting Algorithms

The two tree boosting methods MART and XGBoost differ in multiple ways. They both fit additive tree models. They do however differ in regularization techniques they offer and the boosting algorithm they employ to learn the additive tree model. While MART uses gradient tree boosting (GTB) as shown in Algorithm 4, XGBoost uses Newton tree boosting (NTB) as shown in Algorithm 3. In this chapter, we will compare these tree boosting algorithms. We will show under which conditions they are equivalent and which loss functions they are applicable for.

7.1 The Commonalities

Both gradient tree boosting and Newton tree boosting fits additive tree models as defined in Equation 6.1. They are thus learning algorithms for solving the same empirical risk minimization problem

$$\{\{\hat{w}_{jm}, \hat{R}_{jm}\}_{j=1}^{T_m}\}_{m=1}^{M} = \underset{\{\{w_{jm}, R_{jm}\}_{j=1}^{T_m}\}_{m=1}^{M}}{\arg\min} \sum_{i=1}^{n} L(y_i, \sum_{m=1}^{M}\sum_{j=1}^{T_m} w_{jm}\mathrm{I}(x_i \in R_{jm})). \tag{7.1}$$

This a formidable optimization problem involving joint optimization of M trees. This optimization problem is thus simplified by instead doing forward stagewise additive modeling (FSAM). This simplies the problem by instead performing a greedy search, adding one tree at a time. At iteration m, a new tree is thus learnt using

$$\{\hat{w}_{jm}, \hat{R}_{jm}\}_{j=1}^{T_m} = \underset{\{w_{jm}, R_{jm}\}_{j=1}^{T_m}}{\arg\min} \sum_{i=1}^{n} L(y_i, \hat{f}^{(m-1)}(x_i) + \sum_{j=1}^{T_m} w_{jm}\mathrm{I}(x_i \in R_{jm})). \tag{7.2}$$

This is repeated for $m = 1, ...M$ to yield an approximation to the solution to Equation 7.1.

This is a great simplification to the original optimization problem. For many loss functions, this is however still a difficult optimization problem. Gradient tree boosting and Newton tree boosting differ in how they further simplify the optimization problem in Equation 7.2.

7.2 The Differences

At each iteration, both gradient tree boosting and Newton tree boosting approximate the optimization problem in Equation 7.2. First, they differ in the tree structures they learn. Next, they differ in how they learn the leaf weights to assign in the terminal nodes of the learnt tree structure.

7.2.1 Learning the Structure

Gradient tree boosting and Newton tree boosting optimize different criteria when learning the structure of the tree at each iteration. Gradient tree boosting learns the tree which is most highly correlated with the negative gradient of the current empirical risk. That is, gradient tree boosting learns the structure by fitting a tree model according to

$$\{\tilde{w}_{jm}, \hat{R}_{jm}\}_{j=1}^{T} = \underset{\{w_j, R_j\}_{j=1}^{T}}{\arg\min} \sum_{i=1}^{n} \frac{1}{2}\Big[-\hat{g}_m(x_i) - \sum_{j=1}^{T} w_j \mathrm{I}(x_i \in R_j)\Big]^2. \tag{7.3}$$

That is, at each iteration, a tree model is fit to the negative gradient $\{-\hat{g}_m(x_i)\}_{i=1}^{n}$ using least-squares regression.

Newton tree boosting, on the other hand, learns the tree which best fits the second-order Taylor expansion of the loss function. By completing the square, this criterion can be written in the form

$$\{\tilde{w}_{jm}, \hat{R}_{jm}\}_{j=1}^{T} = \underset{\{w_j, R_j\}_{j=1}^{T}}{\arg\min} \sum_{i=1}^{n} \frac{1}{2}\hat{h}_m(x_i)\Big[-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)} - \sum_{j=1}^{T} w_j \mathrm{I}(x_i \in R_j)\Big]^2. \tag{7.4}$$

That is, at each iteration, a tree model is fit to the negative gradient, scaled by the Hessian, $\{-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)}\}_{i=1}^{n}$ using weighted least-squares regression, where the weights are given by the Hessian $\{\hat{h}_m(x_i)\}_{i=1}^{n}$.

Comparing these, we see that the Hessian plays the role of observation weights for Newton boosting. Learning the structure amounts to searching for splits which maximize the gain. For gradient tree boosting, this gain is given by

$$Gain = \frac{1}{2}\Big[\frac{G_L^2}{n_L} + \frac{G_R^2}{n_R} - \frac{G_{jm}^2}{n_{jm}}\Big],$$

while for Newton tree boosting, the gain is given by

$$Gain = \frac{1}{2}\Big[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}\Big].$$

Newton tree boosting learns the tree structure using a higher-order approximation of the FSAM criterion. We would thus expect it to learn better tree structures than gradient tree boosting. The leaf weights learnt during the search for structure are given by

$$\tilde{w}_{jm} = -\frac{G_{jm}}{n_{jm}}$$

for gradient tree boosting and by

$$\tilde{w}_{jm} = -\frac{G_{jm}}{H_{jm}}$$

for Newton tree boosting. For gradient tree boosting these leaf weights are however subsequently readjusted.

7.2.2 Learning the Leaf Weights

After a tree structure is learnt, the leaf weights need to be determined. For Newton tree boosting, this is straightforward as the final leaf weights are determined using the same criterion that was used to determine the tree structure, i.e. the second-order approximation of the empirical risk function. Newton tree boosting can thus be seen to jointly optimize tree structure and leaf weights. That is, the final leaf weights are the same as the leaf weights learnt when searching for the tree structure, i.e.

$$\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}}.$$

Gradient tree boosting, on the other hand, uses a different criterion to learn the leaf weights. The final leaf weights are determined by separate line searches in each terminal node

$$\hat{w}_{jm} = \arg\min_{w_j} \sum_{i \in \hat{I}_{jm}} L(y_i, \hat{f}^{(m-1)}(x_i) + w_j), \quad j = 1, ..., T.$$

Considering this, gradient tree boosting can be seen to generally use a more accurate criterion to learn the leaf weights than Newton tree boosting. These more accurate leaf weights are however determined for a less accurate tree structure.

7.3 Consequences for Applicable Loss Functions

As we will now discuss, gradient tree boosting is applicable to more loss functions than Newton tree boosting. In fact, this is not only valid for tree boosting, but generally for gradient boosting and Newton boosting.

Gradient boosting only require the loss function to be differentiable. It is further beneficial if the loss function used is also convex as this yields a convex empirical risk function and thus unique solutions. This is however not strictly required.

Newton boosting however, require the loss function to be twice differentiable. Furthermore, as seen from Equation 6.4, the Hessian cannot be zero. Thus, for

Newton boosting, we additionally require the loss function used to be strictly convex.

Consider for example the absolute loss. The negative gradient for this loss function is given by

$$-\hat{g}_m(x_i) = \text{sign}(y_i - \hat{f}^{(m-1}(x_i)).$$

This is not differentiable at $x = 0$. This can however circumvented by defining the gradient to be e.g. 0 at $x = 0$. Gradient boosting can thereby handle the absolute loss function. For Newton boosting however, we run into problems as the Hessian is zero everywhere (except at $x = 0$).

We could try to bypass this by simply defining the Hessian to be some constant, such as one. Newton boosting would in this case simply fit basis functions using the same criterion as gradient boosting at each iteration, but without the subsequent line search step. The convergence would however likely be much slower as the no step length is calculated. Gradient boosting thus seems to have the upper hand when it comes to loss function which are not stricly convex. In addition to the absolute loss, other loss functions which are not strictly convex include the Huber loss and the quantile regression loss.

7.4 When are They Equivalent?

For the squared error loss, the Hessian $\hat{h}_m(x_i) = 1$ everywhere. Consequently, the the NTB criterion in 7.4 collapses to the GTB criterion in 7.3. In this case, they both fit the negative gradient. For the squared error loss, the negative gradient is simply the current residuals

$$-\hat{g}_m(x_i) = y_i - \hat{f}^{(m-1)}(x_i).$$

Plugging this into either the GTB or the NTB criteria, we see that they both solve the FSAM problem in Equation 7.2 exactly for the squared error loss. For GTB, the line search step can also be skipped, as the leaf weights learnt are already optimal.

Gradient tree boosting and Newton tree boosting are thus equivalent for the squared error loss. For other loss functions however, they will differ.

7.5 Which is Better?

If we use any other loss function than the squared error loss, GTB and NTB are not equivalent. A natural question to ask is, which is better?

Since NTB can be seen to use a second order approximation to the loss function, while GTB only use a first order approximation, we would expect NTB to learn better tree structures. NTB does however not readjust the leaf weights after the structure is learnt. In summary, we would thus expect GTB to learn a more approximate tree structure than NTB, but subsequently learn more accurate leaf weights for this approximate structure. They are thus not directly comparable, and likely to outperform each other for different problems.

7.5.1 Empirical Comparison

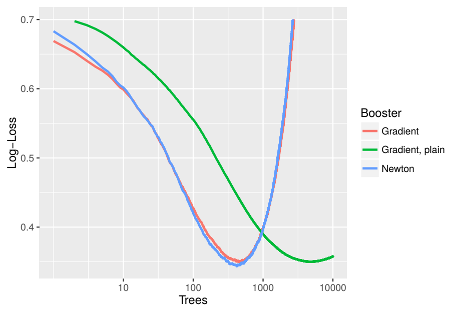
To get an idea of how they compare in practice, we will test their performance on two standard datasets, the Sonar and the Ionosphere datasets (Lichman, 2013). The log-loss was used to measure prediction accuracy.

The comparison was performed using the R programming language (R Core Team, 2016). The *gbm* package (Ridgeway, 2015) was used for the gradient tree boosting, while the *xgboost* package (Chen et al., 2016) was used for Newton tree boosting.

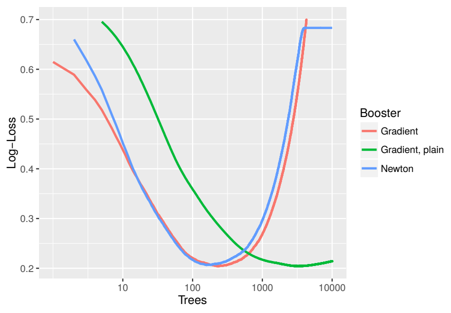
Due to implementation details of the packages, we will use only tree stumps (trees with two terminal nodes) and no other form regularization of the individual trees. Moreover, the data sets chosen have no categorical features and no missing values. The data sets were deliberately chosen this way, since categorical features and missing values are treated differently by the tree learning algorithms in the two packages.

The learning rate was set to $\eta = 0.1$ and the number of trees was set to $M = 10000$. Predictions were made at each iteration. To get a fairly stable estimate of out-of-sample perfomance, three repetitions of 10-fold cross-validation was used. The folds were the same for both GTB and NTB.

Initially, a sanity check was performed to confirm that the two implementations gave identical results for the squared error loss. The check was performed using the Boston housing dataset (Lichman, 2013). As expected, the results were indeed identical for both methods. Next, the two methods were compared for the Sonar dataset and for the Ionosphere dataset. To study the effect of line search for gradient tree boosting, one fit were also done using gradient boosting with line search. This was achieved by defining a custom objective function for the *xgboost* package. The results are shown in Figure 7.1.



(a) Sonar dataset.



(b) Ionosphere dataset.

Figure 7.1: Newton tree boosting and Gradient tree boosting with and without (plain) line search.

From Figure 7.1a, we observe that NTB seems to outperform GTB slighly for the Sonar dataset. From Figure 7.1b, we can see that NTB converges slightly faster, but GTB outperforms NTB when run for enough iterations for the Ionosphere dataset. They do however seem to have very similar performance for both problems. Newton tree boosting uses a higher-order approximation of the FSAM and should thus learn more accurate tree structures than gradient tree boosting. For the log-loss, the criterion used to learn the final leaf weights is actually the same for both. This is due to the line search step being approximated by a single Newton step for gradient tree boosting. Given this, we would apriori expect Newton tree boosting to outperform gradient tree boosting for the log-loss. Of course, due to randomness, any algorithm might perform best in practice for any particular data set.

From both Figure 7.1a and Figure 7.1b, we see that line search clearly improves the rate of convergence. In both cases, line search also seems to improve the lowest log-loss achieved. Line search thus seems to be very beneficial for gradient tree boosting.

Chapter 8

Discussion of Regularization Parameters

MART and XGBoost provides various regularization techniques for additive tree models. In this chapter, we will discuss the effect of the various regularization parameters.

We will group the regularization parameters into three categories. The first is the boosting parameters, which is the number of trees M and the learning rate η. The second is the tree parameters, which include constraints and penalties imposed on the complexities of the individual trees in the additive tree model. Finally, there is the randomization parameters which controls row and column subsampling.

8.1 Boosting Parameters

The boosting parameters are the number of boosting iterations M and the learning rate or shrinkage η. These parameters were discussed in Section 4.5 and are not specific to tree boosting. We will here add to this discussion for the case of tree boosting.

8.1.1 The Number of Trees M

The parameter M in general corresponds to the number of boosting iterations. In the case of tree boosting, M can also be seen to correspond to the number of trees in the additive tree model.

Let $\mathcal{F}_M$ denote the function space of an additive tree model consisting of M trees. Since the function space of the individual trees models are not closed under addition, we will have that

$$\mathcal{F}_1 \subset \mathcal{F}_2 \subset ... \subset \mathcal{F}_M$$

Thus, increasing the number of iterations, i.e. adding more trees, will increase the representational ability of the model. Selecting the number of trees M is thus crucial in order to achieve the right amount of representational ability. This can be viewed the "main" tuning parameter of tree boosting.

8.1.2 The Learning Rate η

The learning rate or shrinkage parameter η will generally shrink the added basis function at each iteration. For tree boosting, η can be seen to shrink the leaf weights of each of the individual trees learnt at each iteration. If η is set too high, the model will fit a lot of the structure in the data during the early iterations, thereby quickly increasing variance. As we discussed in the last section however, using a larger number of trees increases the representational ability. Thus, by lowering the learning rate η, a larger number of trees can be added before the additive tree model will start to overfit the data. This will allow the model to have greater representational ability, with smoother fits, before overfitting the data.

8.2 Tree Parameters

The tree parameters are the regularization parameters which controls the complexity of the individual trees. These parameters can be seen to control the number of terminal nodes T in the tree, the size of the leaf weights w and the minimum sum of observation weights needed in a terminal node.

The number of terminal nodes in a tree limits the order of interactions it can capture. A stump (a tree of depth two) can only capture additive effects, while deeper trees can capture higher-order interactions. The functional ANOVA decomposition (Friedman, 1991) of the target function f^* can be written

$$f^*(x) = \sum_j f_j^*(x_j) + \sum_{j,k} f_{j,k}^*(x_j, x_k) + \sum_{j,k,l} f_{j,k,l}^*(x_j, x_k, x_l) + ...$$

Here, the functions $f_1^*, ..., f_p^*$ specify the additive relationship between the response and the inputs. The second sum runs over the second-order interactions, while the third sum runs over the third-order interactions and so on. The maximum order of iterations is only limited by p. The dominant order of interaction is however typically lower than p, meaning that the target function can be approximated well by interaction terms only up to a certain order.

The size of the leaf weights $w_1, ..., w_T$ will also be related to model complexity. Smaller coefficients will yield models closer to the global constant, while larger coefficients will yield more complex models.

Lastly, the minimum sum of observation weights needed in a terminal node can directly limit the variance in the estimation of the leaf weights. For gradient boosting, the sum of the observation weights is simply the number of observations. For Newton boosting however, the sum of observation weights is the sum of the Hessians in the terminal node. For brevity, we will refer to minimum sum of observation weights needed in a terminal node as the required evidence in a terminal node.

8.2.1 Maximum Number of Terminal Nodes $T_{\max}$

We will denote the maximum number of terminal nodes by $T_{\max}$. The more terminal nodes the tree has, the more complex functions it can represent. The more terminal nodes it has, the higher the order of interactions the additive tree model can capture. As the number of terminal nodes grows, there will tend to be fewer observations in each region, giving rise to variance in estimation of the leaf weights. If the number of terminal nodes is too low, on the other hand, the tree might not be able to capture a sufficient order of interactions, giving rise to bias. There will therefore naturally be a bias-variance tradeoff in selecting the number of terminal nodes in the tree.

Friedman (2001) suggested fixing the number of terminal nodes to a constant. For MART, $T_{\max}$ determines the number of terminal nodes in each tree, i.e. $T_m = T_{\max}$ for $m = 1, ..., M$. For XGBoost, penalization might be included in the objective. In this case, the number of terminal nodes might vary among the trees. $T_{\max}$ thus only acts as an upper bound in this case.

8.2.2 Minimum Sum of Observation Weights in Leaf $h_{\min}$

We will denote the required evidence in a terminal node by $h_{\min}$. For gradient tree boosting, this is equivalent to the minimum amount of observations required in a terminal node, and could thus be denoted $n_{\min}$ in this case.

Each terminal node in the tree is associated with a region in the input space. Since we will estimate a leaf weight for each region of the input space, we would like the region to contain as much observation weight as possible. Regions containing more observation weight allow the leaf weight of the region to be estimated more reliably.

Setting a lower bound on the sum of the observation weights needed in such a region can thus limit the variance in estimation of the leaf weights. Although increasing $h_{\min}$ could reduce the variance in estimation, setting it too high can introduce significant bias. Again, there is a bias-variance tradeoff in selecting $h_{\min}$.

8.2.3 Terminal Node Penalization γ

The penalization term that XGBoost includes in the objective is shown in Equation 6.2. The first term in this is the terminal node penalization. The parameter γ controls the amount of penalization for the number of terminal nodes.

Doing a derivation similar to the one in Section 6.3.1, but now including terminal node penalization in the objective, we find that the gain is now given by

$$Gain = \frac{1}{2}\Big[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}\Big] - \gamma.$$

Penalization of the number of terminal nodes will thus increase the probability of obtaining splits with negative gain. After pruning, we will thus tend to get more shallow trees. This parameter thus affects the learning of the tree structure. The learning of the final leaf weights, on the other hand, are not directly affected.

We have previously discussed the effects of the number of terminal nodes in the tree. Introducing an objective which penalizes the number of terminal nodes gives one the ability to more adaptively select the number of terminal nodes. Depending on the complexity of the patterns found in the data set, the number of terminal nodes will adjust to fit the data in a tradeoff with model complexity.

8.2.4 $l2$ Regularization on Leaf Weights λ

The second term in Equation 6.2 is the $l2$ regularization of the leaf weights. Here λ controls the strength of the penalization.

$l2$ regularization can be seen to place a prior belief that the leaf weights should be small. Increasing λ will thus have the effect of shrinking the values of the weights towards zero. This resembles the effect of the learning rate η. However, whereas the learning rate η shrinks all leaf weights by the same factor, λ will generally shrink the leaf weights by varying amounts. The biggest difference however, is that λ will not only affect the leaf weights, but also the structure of the tree.

Doing a derivation similar to the one in Section 6.3.1, but now including $l2$ regularization of the leaf weights in the objective, we find that the gain is now given by

$$Gain = \frac{1}{2}\Big[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G_{jm}^2}{H_{jm} + \lambda}\Big].$$

Furthermore, the final leaf weights are given by

$$\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm} + \lambda}.$$

From this, we see that $l2$ regularization does indeed shrink the leaf weights. However, it also affects tree structure. In fact, it alters the gain such that different splits ends up being taken. We observe that this has the effect of trying to avoid splits which lead to regions with little evidence. Estimation of leaf weights in such regions will have high variance. However, as opposed to $h_{\min}$, no constraint is imposed, but rather penalization. If a leaf weight still were to be estimated in a region with little evidence, it will be shrunk more heavily. In addition, $l2$ regularization increases the probability of obtaining a splits with negative gains. It can thus also affect the number of terminal nodes. In summary, $l2$ regularization can be seen to combat variance in estimation of the leaf weights. In order to do this, it does not only alter the leaf weights, but also the structure of the tree.

8.2.5 $l1$ Regularization on Leaf Weights α

The third term in Equation 6.2 is the $l1$ regularization of the leaf weights. Here, α controls the strength of the penalization.

$l1$ regularization will play a similar role to $l2$ regularization. The major difference is that $l1$ regularization has the ability to shrink the leaf weights all the way to zero.

Doing a derivation similar to the one in Section 6.3.1, but now including $l1$ regularization of the leaf weights in the objective, we find that the gain is now given by

$$Gain = \frac{1}{2}\Big[\frac{T_\alpha(G_L)^2}{H_L} + \frac{T_\alpha(G_R)^2}{H_R} - \frac{T_\alpha(G_{jm})^2}{H_{jm}}\Big],$$

while the final leaf weights are given by

$$\hat{w}_{jm} = -\frac{T_\alpha(G_{jm})}{H_{jm}},$$

where

$$T_\alpha(G) = \operatorname{sign}(G)\max(0, |G| - \alpha).$$

From this, we see that as for $l2$ regularization, $l1$ regularization also affects both the structure and the leaf weights. However, instead of focusing on the Hessian in the leaves, $l1$ regularization focuses on the gradient.

8.3 Randomization Parameters

The randomization parameters refer to the parameters which introduce randomness in the learning process. This includes row subsampling as introduced by Friedman (2002), as well as column subsampling, which is included in XGBoost (Chen and Guestrin, 2016).

The fact that randomization can improve generalization performance might seem counterintuitive at first. It is however not very mysterious. By introducing randomization, the trees will indeed lose accuracy. However, they will also tend to be less similar, i.e. more diverse. The fact that they are more diverse is beneficial when they are combined together. The positive effect of diversity often outweighs the negative effects of the lost accuracy of the individual trees. The tradeoff in selecting the right amount of diversity is known as the accuracy-diversity tradeoff in literature on ensemble methods. See e.g. (Zhou, 2012) for more details.

One way to understand the accuracy-diversity tradeoff is through the bias-variance-covariance decomposition

$$\begin{aligned}\operatorname{Var}[\hat{f}(x)] &= \operatorname{Var}[\sum_{m=1}^{M} \hat{f}_m(x)] \\ &= \sum_{m=1}^{M} \operatorname{Var}[\hat{f}_m(x)] + \sum_{m=1}^{M}\sum_{m'\neq m} \operatorname{Cov}[\hat{f}_m(x), \hat{f}_{m'}(x)].\end{aligned}$$

Increasing randomization will increase the variance of the individual trees, but will tend to decrease the covariance between the trees. Oftentimes, the reduction in covariance is greater than the increase in variance. The overall effect is thus typically that the overall variance of the ensemble model, i.e. the additive tree model, is reduced.

8.3.1 Row Subsampling Fraction ω_r

Friedman (2002) found that row subsampling could yield substantial performance improvements. He found the greatest improvements for small data sets and complex base models, thus leaving him to conclude that variance reduction was a probable explanation.

According to Hastie et al. (2009), a typical value for the row subsampling fraction is $\omega_r = 0.5$, although it can be substantially smaller for large data sets. In addition to improving generalization performance, setting the row subsampling fraction lower can reduce computation time by the same fraction ω_r. This is especially useful for larger data sets.

8.3.2 Column Subsampling Fraction ω_c

Column subsampling plays a similar role to row subsampling. They both combat overfitting by introducing randomness in the learning procedure and both can reduce computation time. Chen and Guestrin (2016) states that according to user feedback, using column subsampling prevents overfitting even more so than traditional row subsampling.

Chapter 9

Why Does XGBoost Win "Every" Competition?

In this chapter, we will provide some informal arguments for why tree boosting, and especially XGBoost, performs so well in practice. Of course, for any specific data set, any method may dominate others. We will thus provide arguments as to why tree boosting seems to be such a versatile and adaptive approach, yielding good results for a wide array of problems, and not for why it is necessarily better than any other method for any specific problem.

Uncovering some possible reasons for why tree boosting is so effective are interesting for a number of reasons. First, it might improve understanding of the inner workings of tree boosting methods. Second, it can possibly aid in further development and improvement of the current tree boosting methodology. Third, by understanding the core principles of tree boosting which makes it so versatile, we might be able to construct whole new learning methods which incorporates the same core principles.

9.1 Boosting With Tree Stumps in One Dimension

Consider first the very simple case of boosting in one dimension, i.e. with only one predictor, using tree stumps, i.e. trees with only one split and two terminal nodes. We otherwise assume the model in unconstrained and unpenalized.

Does tree boosting have any particular advantages even in this simple problem? First of all, additive tree models have rich representational abilities. Although they are constrained to be piecewise constant functions, they can potentially approximate any function arbitrarily close given that enough trees are included.

The perhaps greatest benefit however is that tree boosting can be seen to adaptively determine the local neighbourhoods. We will now consider an example to clarify what we mean by this.

9.1.1 Adaptive Neighbourhoods

Let us consider a regression task and compare additive tree models with local linear regression and smoothing splines. All of these methods have regularization parameters which allows one to adjust the flexibility of the fit to the data. However, while methods such as local regression and smoothing splines can be seen to use the same amount of flexibility in the whole input space of $\mathcal{X}$, additive tree models can be seen to adjust the amount of flexibility locally in $\mathcal{X}$ to the amount which seems necessary. That is, additive tree models can be seen to adaptively determine the size of the local neighbourhoods.

Let us consider a concrete example. To make the point clear, we will design a problem where different amounts of flexibility is needed locally in $\mathcal{X}$. Let the data be generated according to

$$\begin{aligned} X &\sim \text{Uniform}(-6, 6) \\ [Y|X] &= \begin{cases} \sin(X) + \epsilon, & X \leq 0 \\ \epsilon, & 0 < X \leq \pi \\ \sin(4X) + \epsilon, & X > \pi \end{cases} \\ \epsilon &\sim N(0, 1/2). \end{aligned}$$

We will draw 500 samples from this. These samples together with the target function f^* is shown in Figure 9.1.

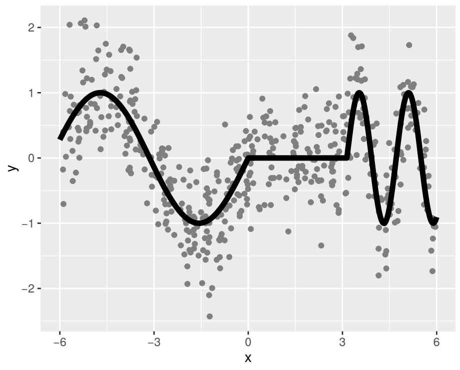
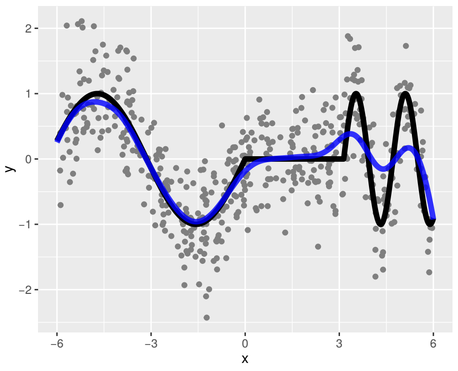


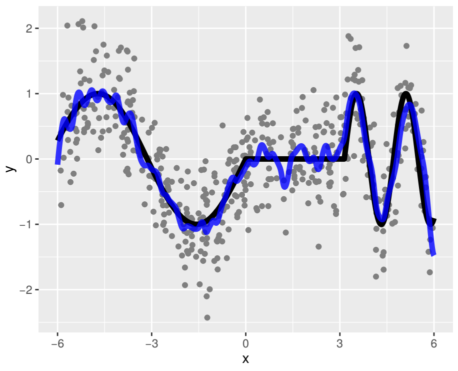
Figure 9.1: Simulated data in gray together with the target function f^* in black.

This data clearly requires a high degree of flexibility for higher values of x, less flexibility for low values of x and almost no flexibility at all for medium values of x. That is, for higher values of x, smaller neighbourhoods are needed to avoid a large bias. Keeping the neighbourhoods too small in the flat region of the target function, however, will unnecessarily increase variance.

Let us first fit local linear regression with two different degrees of flexibility to the data. The resulting fits are shown in blue in Figure 9.2. In Figure 9.2a a wider neighbourhood is used for the local regression. This seems to yield a satisfactory fit for lower values of x, but is not flexible enough to capture the more complex structure for the higher values of x. Vice versa, in Figure 9.2b, a smaller neighbourhood is used. This fit is flexible enough to capture the complex structure for the higher values of x, but seems too flexible for lower values of x. The same phenomenon is observed for smoothing splines. Figure 9.3 shows two smoothing spline fits of different flexibility.

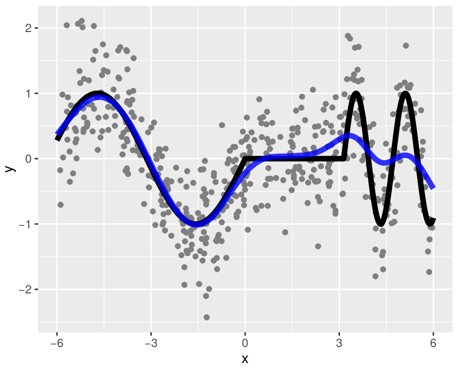


(a) Less flexible.

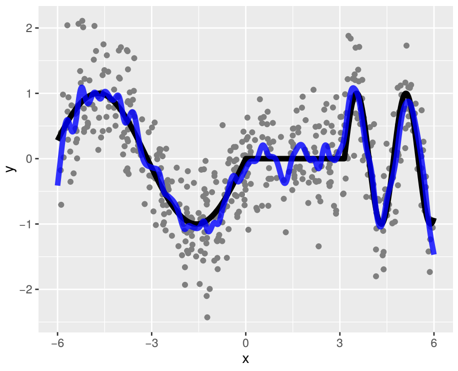


(b) More flexible.

Figure 9.2: Local linear regression with two different degrees of flexibility fit to the simulated data.



(a) Less flexible.



(b) More flexible.

Figure 9.3: Smoothing splines with two different degrees of flexibility fit to the simulated data.

Finally, we fit an additive tree model (with $\eta = 0.02, M = 4000$) to the data using tree stumps. The result is displayed in Figure 9.4. We observe that in areas with simpler structure, the additive tree model has put less effort into fitting the data. In the region where the target function is constant, for example, the model

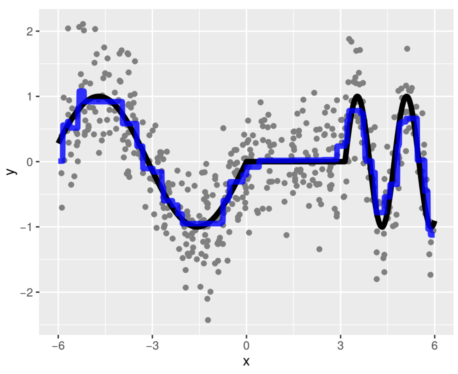


Figure 9.4: Boosted tree stumps fit to the simulated data.

has put very little effort into capturing any structure. In the region with more complex structure however, more effort has been put in to capture it. Tree boosting can thus be seen to use adaptive neighbourhood sizes depending on what seems necessary from the data. In areas where complex structure is apparent from the data, smaller neighbourhoods are used, whereas in areas where complex structure seems to be lacking, a wider neighbourhood is used.

To better understand the nature of how local neighbourhoods are determined, we will attempt make the notion of the neighbourhood more concrete. We will do this by considering the interpretation that many models, including additive tree models, local regression and smoothing splines, can be seen to make predictions using a weighted average of the training data.

9.1.2 The Weight Function Interpretation

Many models can be written in the form

$$\hat{f}(x) = \hat{w}(x)^T y,$$

where $\hat{w}(x)$ is a weight function, $\hat{w} : \mathcal{X} \to \mathbb{R}^n$. For each $x \in \mathcal{X}$, the weight function $\hat{w}(x)$ specifies the vector weights to use in the weighted average of the responses in the training data.

We can write that variance of the model as

$$\mathrm{Var}[\hat{f}(x)] = \mathrm{Var}[\sum_{i=1}^{n} \hat{w}_i(x) Y_i] = \sum_{i=1}^{n} \hat{w}_i(x)^2 \mathrm{Var}[Y_i] = \sigma^2 \sum_{i=1}^{n} \hat{w}_i(x)^2.$$

From this, we can see that in order to keep the variance low, the weights should be spread out as evenly as possible, thus keeping the neighbourhood wide. The globally constant model keeps $\hat{w}_i(x) = 1/n, \forall x \in \mathcal{X}, i \in \{1, ..., n\}$ and thus keeps the variance as low as possible. If the target function is sufficiently complex however, this model will be severely biased. To decrease the bias, the weights have to be shifted such that points which are similar or close to x receives larger weight, while distant and dissimilar points receive lower weights.

Consider for example linear regression. Predictions are given by

$$\hat{f}(x) = x^T(X^TX)^{-1}X^Ty.$$

The weight can thus be written as

$$\hat{w}(x)^T = x^T(X^TX)^{-1}X^T.$$

For local linear regression, the weight function is a simple modification of this. It can be written

$$\hat{w}(x)^T = x^T(X^TW(x)X)^{-1}X^TW(x),$$

where $W(x)$ is a diagonal matrix where diagonal element i is $\kappa(x, x_i)$ (Hastie et al., 2009). The weight functions for local linear regression at three different points for two different degrees of flexibility are shown in Figure 9.5. The two different degrees of flexibility are the same as those used in Figure 9.2. We observe that the weight function has a particular shape, regardless of the position in $x \in \mathcal{X}$. Also, as expected, the less flexible the model is, the more spread out its weights are.

Another example is smoothing splines. The weight function can in this case be written

$$\hat{w}(x)^T = \phi(x)^T(\Phi^T\Phi + \lambda\Omega)^{-1}\Phi^T,$$

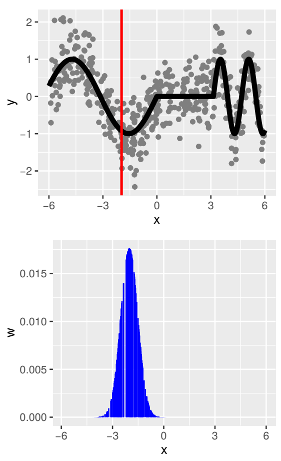
where $\Omega_{jk} = \int \phi_j''(t)\phi_k''(t)dt$ and λ is a regularization parameter which penalizes lack of smoothness (Hastie et al., 2009). In Figure 9.6, the weight functions for smoothing splines at three different points and for two different flexibilities are shown. We here observe the same phenomenon as for local linear regression, namely that the weight function takes the same form regardless of $x \in \mathcal{X}$.

For these models, and many others, $\hat{w}(x)$ is determined using only the location of the predictors x_i in the input space $\mathcal{X}$, without regard for the responses y_i. These models can thus be seen to have made up their mind about which points are similar beforehand. Similarity is often determined by some measure of closeness of points in the input space $\mathcal{X}$. Intuitively, most models will assign larger weights to data points which are determined to be closer to x.

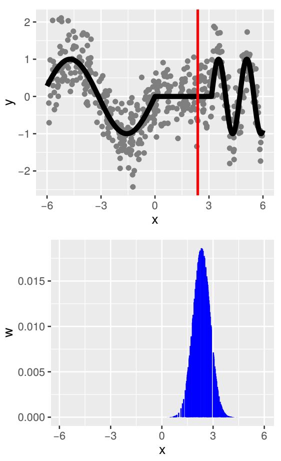
For additive tree models, on the other hand, the weight function can be seen to be determined adaptively. That is, while the other methods only take the predictors x_i in the training data into account when determining $\hat{w}(x)$, additive tree models also considers the responses in the training data. This is in fact a property of tree models, which adaptively determines neighbourhoods and fits a constant in each neighbourhood. Additive tree models inherit this from tree models and uses it to adaptively shift the weight functions at each iteration. At iteration m, boosting updates the model according to

$$\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$$
$$\hat{w}^{(m)}(x)^Ty = \hat{w}^{(m-1)}(x)^Ty + \hat{w}_m(x)^Ty.$$

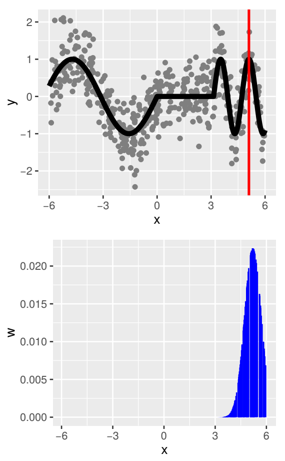
Tree boosting can thus be seen to update the weight functions at each iteration. At each iteration, the learning algorithm searches for splits that minimize the empirical



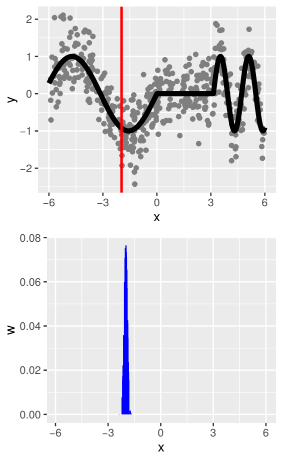
(a) Point 1, low flexibility.



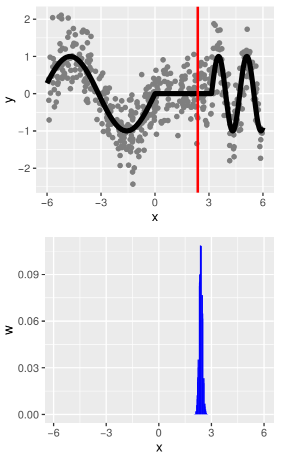
(b) Point 2, low flexibility.



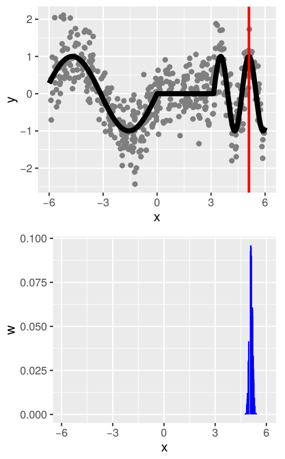
(c) Point 3, low flexibility.



(d) Point 1, high flexibility.

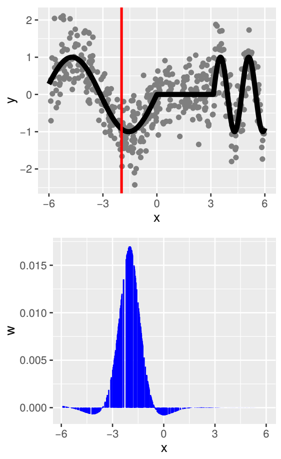


(e) Point 2, high flexibility.

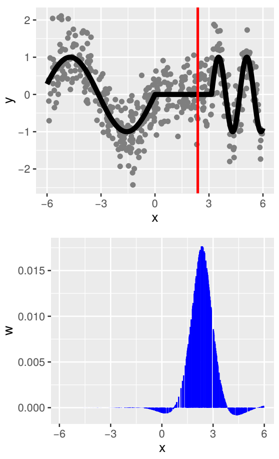


(f) Point 3, high flexibility.

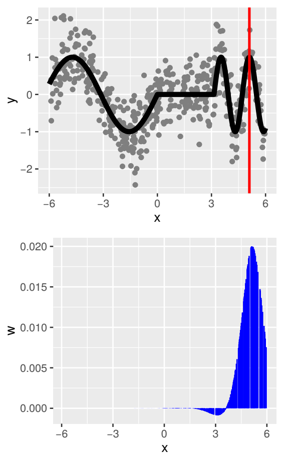
Figure 9.5: The weight function at 3 points for local linear regression with 2 different degrees of flexibility.



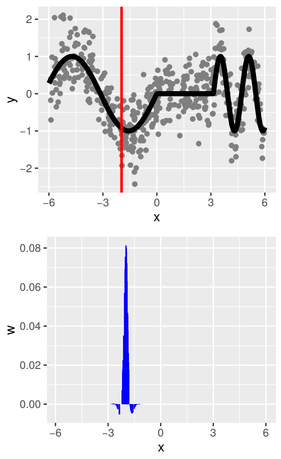
(a) Point 1, low flexibility.



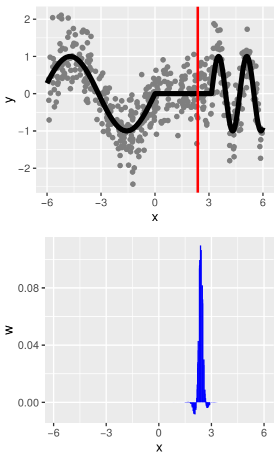
(b) Point 2, low flexibility.



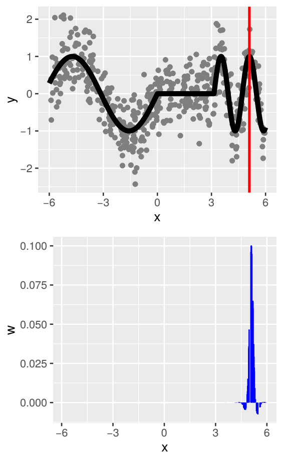
(c) Point 3, low flexibility.



(d) Point 1, high flexibility.



(e) Point 2, high flexibility.



(f) Point 3, high flexibility.

Figure 9.6: The weight function at 3 points for smoothing spline with 2 different degrees of flexibility.

risk. In this process, it can be seen to learn which points can be considered similar, thus adaptively adjusting the weight functions in order to reduce empirical risk. The additive tree model initially starts out as a global constant model with the weights spread out evenly, and thus has low variance. At each subsequent iteration, the weight functions are updated where it seems most necessary in order to reduce bias.

Tree boosting can thus be seen to directly take the bias-variance tradeoff into consideration during fitting. The neighbourhoods are kept as large as possible in order to avoid increasing variance unnecessarily, and only made smaller when complex structure seems apparent. Using smaller neighbourhoods in these areas can thus dramatically reduce bias, while only introducing some variance.

We will now show how the weight functions are adjusted by tree boosting at each iteration with the squared error loss. Consider the tree to be added at iteration m,

$$\hat{f}_m(x) = \sum_{j=1}^{T} \hat{\theta}_{jm} \mathrm{I}(x \in \hat{R}_{jm}).$$

The leaf weights for this tree is determined by

$$\hat{\theta}_{jm} = -\frac{G_{jm}}{n_{jm}} = \frac{\sum_{i \in \hat{I}_{jm}} [y_i - \hat{w}^{(m-1)}(x_i)^T y]}{n_{jm}}.$$

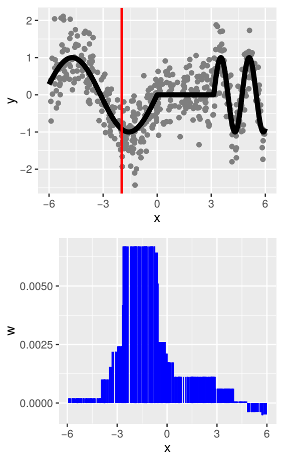
Manipulating this expression, we find that

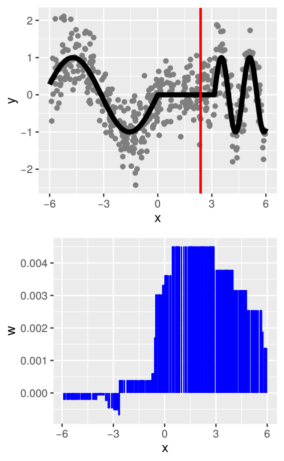
$$\hat{\theta}_{jm} = \sum_{i=1}^{n} y_i \left[\frac{\mathrm{I}(x_i \in \hat{R}_{jm}) - \sum_{k \in \hat{I}_{jm}} \hat{w}_i^{(m-1)}(x_k)}{n_{jm}} \right].$$

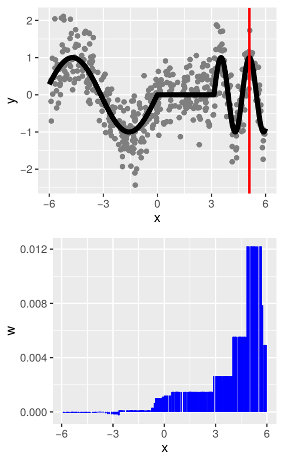
The update of element i of the weight function at x at iteration m is thus given by

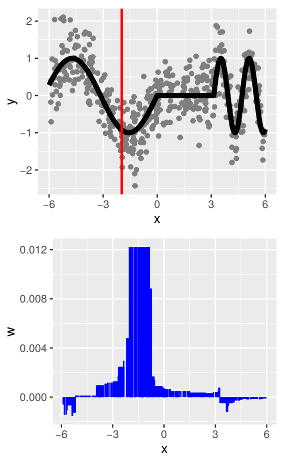
$$\hat{w}_i^{(m)}(x) = \hat{w}_i^{(m-1)}(x) + \sum_{j=1}^{T} \mathrm{I}(x \in \hat{R}_{jm}) \left[\frac{\mathrm{I}(x_i \in \hat{R}_{jm}) - \sum_{k \in \hat{I}_{jm}} \hat{w}_i^{(m-1)}(x_k)}{n_{jm}} \right].$$

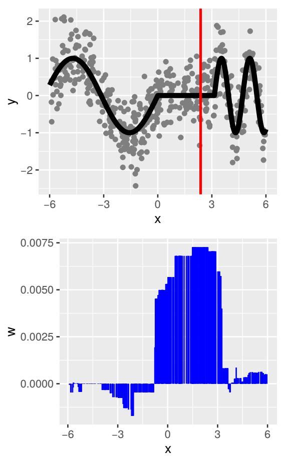
The weight functions for an additive tree model after 400 and 4000 iterations at the three different points are shown in Figure 9.7. The additive tree model shown in Figure 9.4 was for 4000 iterations. We see that the weight function is more spread out for lower iterations. The main point to observe however, is that the weight functions are different at different values of x. At the point in the flat region of the target function in Figure 9.7e, the weight function is spread out over the similar points nearby. This allows the model to calculate the prediction at this point with low variance, without introducing much bias. At the point in the region where the target function is most complex, shown in Figure 9.7f, the weight function is more peaked around x. This keeps bias low, which seems appropriate in this region of the input space. Finally, for the region where the target function is less complex, shown in Figure 9.7d, the peakedness of the weight function is somewhere between the two other. More interestingly however, the weight function is not centered around x, but seems to assign more weight to points at higher values of x. This also seems appropriate as these points are more similar, whereas in the direction of decreasing values of x, the target function changes more rapidly.


(a) Point 1, iteration 400.


(b) Point 2, iteration 400.


(c) Point 3, iteration 400.


(d) Point 1, iteration 4000.


(e) Point 2, iteration 4000.

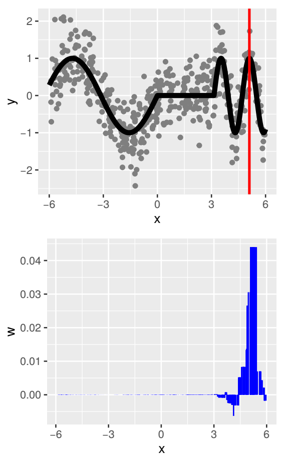

(f) Point 3, iteration 4000.

Figure 9.7: The weight function at 3 points for boosted trees after 400 and 4000 iterations.

9.2 Boosting With Tree Stumps in Multiple Dimensions

In the last section, we considered tree boosting in one dimension. Some of the greatest benefits of tree boosting are however not apparent when considering one-dimensional problems, as tree boosting is particularly useful for high-dimensional problems.

When the dimensionality of the problem increases, many methods break down. This is, as discussed earlier, due to what is known as the curse of dimensionality. Many methods rely on some measure of similarity or closeness between data points, either implicitly or explicitly, in order to introduce locality in the models. Since distance measures become less useful in higher dimensions, these methods tend to not scale well with increasing dimensionality. Techniques such as feature selection and stronger regularization might be employed to combat this. However, good results still depends crucially on good transformations and relative scalings of the features and specification of appropriate amount of flexibility for each feature. For high-dimensional problems, this can be an almost impossible task.

Tree boosting “beats” the curse of dimensionality by not relying on any distance metric. Instead, the similarity between data points are learnt from the data through adaptive adjustment of neighbourhoods. Tree boosting initially keeps the neighbourhood global in all directions. The neighbourhoods are subsequently adjusted to be smaller where it seems most necessary. It thus starts out as a globally constant model and decreases bias by decreasing neighbourhood size. The property of adaptive neighbourhoods might not be needed as often for one-dimensional problems that we discussed in the last section. When the dimensionality is increased however, it is likely to beneficial. This allows the model to adaptively determine the amount of flexibility to use for each feature. In the extreme case, the method might use no flexibility at all for some features, i.e. keep the neighbourhood globally constant along them. The additive tree model will be unaffected by these features and can thus be seen to perform automatic feature selection.

It is thus the same property of adaptively determined neighbourhoods that we discussed in the previous section that makes it “immune” to the curse of dimensionality. By using adaptive neighbourhoods, the model also becomes invariant under monotone transformations of the inputs. This can thus potentially save a lot of work spent searching for appropriate transformations. Moreover, the relative scalings features are irrelevant for the model.

So far, we have considered tree boosting using only tree stumps. Consequently, our additive tree model would only be able to capture additive structure in the data, not any interactions.

9.3 Boosting With Deeper Trees

To capture interactions, we need deeper trees. The deeper the trees are allowed to be, the higher the orders of interactions we can capture. For an additive tree model where the maximum number of terminal nodes is $T_{\max}$, the highest order of interaction that can be captured is $\max(T_{\max} - 1, p)$. There are few other methods which are able to capture high order interactions from high-dimensional data without breaking down. This is one of the great benefits of additive tree models. Again, it is due to the property of adaptive neighbourhoods that it does not break down. That is, most interactions are not modeled at all, only interactions which seem beneficial to the model is included.

Although deeper trees allow us to capture higher order interactions, which is beneficial, they also give rise to some problems. With deeper trees, the number of observations falling in each terminal node will tend to decrease. Thus, the estimated leaf weights will tend to have higher variance. Stronger regularization might therefore be required when boosting with deeper trees. Another related problem is that the model may model interactions where they are not present. Consider for simplicity a two-dimensional problem where only additive structure is present. At early boosting iterations, a lot of the structure is still left in the data. Thus, after the first split, the second split may be taken along the other feature, thereby giving rise to an apparent interaction. It can thus confuse additive structure for interactions. This will unnecessarily increase variance since the neighbourhood is not kept as wide as it could have been. This might be an area where current boosting methods might be improved.

9.4 What XGBoost Brings to the Table

All the discussion so far have been general to tree boosting and is therefore relevant for both MART and XGBoost. In summary, tree boosting is so effective because it fits additive tree models, which have rich representational ability, using adaptively determined neighbourhoods. The property of adaptive neighbourhoods makes it able to use variable degrees of flexibility in different regions of the input space. Consequently, it will be able to perform automatic feature selection and capture high-order interactions without breaking down. It can thus be seen to be robust to the curse of dimensionality.

For MART, the number of terminal nodes is kept fixed for all trees. It is not hard to understand why this might be suboptimal. For example, for high-dimensional data sets, there might be some group of features which have a high order of interaction with each other, while other features only have lower order interactions, perhaps only additive structure. We would thus like to use deeper trees for some features than for the others. If the number of terminal nodes is fixed, the tree might be forced to do further splitting when there might not be a lot of evidence for it being necessary. The variance of the additive tree model might thus increase unnecessarily.

XGBoost uses clever penalization of the individual trees. The trees are consequently allowed to have varying number of terminal nodes. Moreover, while MART uses only shrinkage to reduce the leaf weights, XGBoost can also shrink them using penalization. The benefit of this is that the leaf weights are not all shrunk by the same factor, but leaf weights estimated using less evidence in the data will be shrunk more heavily. Again, we see the bias-variance tradeoff being taken into account during model fitting. XGBoost can thus be seen to be even more adaptive to the data than MART.

In addition to this, XGBoost employs Newton boosting rather than gradient boosting. By doing this, XGBoost is likely to learn better tree structures. Since the tree structure determines the neighbourhoods, XGBoost can be expected to learn better neighbourhoods.

Finally, XGBoost includes an extra randomization parameter. This can be used to decorrelate the individual trees even further, possibly resulting in reduced overall variance of the model. Ultimately, XGBoost can be seen to be able to learn better neighbourhoods by using a higher-order approximation of the optimization problem at each iteration than MART and by determining neighbourhoods even more adaptively than MART does. The bias-variance tradeoff can thus be seen to be taken into account in almost every aspect of the learning.

Chapter 10

Conclusion

Tree boosting methods have empirically proven to be a highly effective and versatile approach to predictive modeling. For many years, MART has been a popular tree boosting method. In more recent years, a new tree boosting method by the name XGBoost has gained popularity by winning numerous machine learning competitions. In this thesis, we compared these tree boosting methods and provided arguments for why XGBoost seems to win so many competitions.

We first showed that XGBoost employs a different form of boosting than MART. While MART employs a form of gradient boosting, which is well known for its interpretation as a gradient descent method in function space, we showed that the boosting algorithm employed by XGBoost can be interpreted as Newton's method in function space. We therefore termed it Newton boosting. Moreover, we compared the properties of these boosting algorithms. We found that gradient boosting is more generally applicable as it does not require the loss function to be strictly convex. When applicable however, Newton boosting is a powerful alternative as it uses a higher-order approximation to the optimization problem to be solved at each boosting iteration. It also avoids the need of a line search step, which can involve difficult calculations in many situations.

In addition to using different boosting algorithms, MART and XGBoost also offers different regularization parameters. In particular, XGBoost can be seen to offer additional parameters not found in MART. Most importantly, it offers penalization of the individual trees in the additive tree model. These parameters will affect both the tree structure and leaf weights in order to reduce the variance in each tree. Additionally, XGBoost provides an extra randomization parameter which can be used to decorrelate the individual trees, which in turn can result in reduction of the overall variance of the additive tree model.

After determining the different boosting algorithms and regularization techniques these methods utilize and exploring the effects of these, we turned to providing arguments for why XGBoost seems to win "every" competition. To provide possible answers to this question, we first gave reasons for why tree boosting in general can be an effective approach. We provided two main arguments for this. First off, additive tree models can be seen to have rich representational abilities. Provided that enough trees of sufficient depth is combined, they are capable of closely approximating complex functional relationships, including high-order interactions. The most important argument provided for the versatility of tree boosting however, was that tree boosting methods are adaptive. Determining neighbourhoods adaptively allows tree boosting methods to use varying degrees of flexibility in different parts of the input space. They will consequently also automatically perform feature selection. This also makes tree boosting methods robust to the curse of dimensionality. Tree boosting can thus be seen actively take the bias-variance tradeoff into account when fitting models. They start out with a low variance, high bias model and gradually reduce bias by decreasing the size of neighbourhoods where it seems most necessary.

Both MART and XGBoost have these properties in common. However, compared to MART, XGBoost uses a higher-order approximation at each iteration, and can thus be expected to learn "better" tree structures. Moreover, it provides clever penalization of individual trees. As discussed earlier, this can be seen to make the method even more adaptive. It will allow the method to adaptively determine the appropriate number of terminal nodes, which might vary among trees. It will further alter the learnt tree structures and leaf weights in order to reduce variance in estimation of the individual trees. Ultimately, this makes XGBoost a highly adaptive method which carefully takes the bias-variance tradeoff into account in nearly every aspect of the learning process.

Bibliography

Aggarwal, C. C., Hinneburg, A., and Keim, D. A. (2001). *On the Surprising Behavior of Distance Metrics in High Dimensional Space*, pages 420–434. Springer Berlin Heidelberg, Berlin, Heidelberg.

Akaike, H. (1973). Information theory and an extension of the maximum likelihood principle. In Kotz, S. and Johnson, N. L., editors, *Second International Symposium on Information Theory*, pages 267–281. Springer-Verlag.

Allen, D. (1974). The relationship between variable selection and data agumentation and a method for prediction. *Technometrics*, 16(1):125–127.

Bartlett, P. L., Jordan, M. I., and McAuliffe, J. D. (2006). Convexity, classification, and risk bounds. *Journal of the American Statistical Association*, 101(473):138–156.

Bellman, R. and Bellman, R. (1961). *Adaptive Control Processes: A Guided Tour.* Princeton Legacy Library. Princeton University Press.

Bickel, P. J., Li, B., Tsybakov, A. B., van de Geer, S. A., Yu, B., Valdés, T., Rivero, C., Fan, J., and van der Vaart, A. (2006). Regularization in statistics. *Test*, 15(2):271–344.

Bousquet, O., Boucheron, S., and Lugosi, G. (2004). *Introduction to Statistical Learning Theory*, pages 169–207. Springer Berlin Heidelberg, Berlin, Heidelberg.

Breiman, L. (1996). Bagging predictors. *Mach. Learn.*, 24(2):123–140.

Breiman, L. (1997a). Arcing the edge. Technical report.

Breiman, L. (1997b). Prediction games and arcing algorithms.

Breiman, L. (1998). Arcing classifier (with discussion and a rejoinder by the author). *Ann. Statist.*, 26(3):801–849.

Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.

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

Bühlmann, P. and Hothorn, T. (2007). Boosting algorithms: Regularization, prediction and model fitting. *Statist. Sci.*, 22(4):477–505.

Bühlmann, P. and Yu, B. (2010). Boosting. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(1):69–74.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM.

Chen, T., He, T., and Benesty, M. (2016). *xgboost: Extreme Gradient Boosting*. R package version 0.4-4.

Cleveland, W. S. and Devlin, S. J. (1988). Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83:596–610.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Mach. Learn.*, 20(3):273–297.

Cox, D. R. (1958). The regression analysis of binary sequences (with discussion). *J Roy Stat Soc B*, 20:215–242.

Domingos, P. (2000). A unified bias-variance decomposition and its applications. In *IN PROC. 17TH INTERNATIONAL CONF. ON MACHINE LEARNING*, pages 231–238. Morgan Kaufmann.

Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *Ann. Statist.*, 7(1):1–26.

Efron, B. and Hastie, T. (2016). *Computer Age Statistical Inference*. Institute of Mathematical Statistics Monographs. Cambridge University Press.

Evgeniou, T., Pontil, M., and Poggio, T. (2000). Statistical learning theory: A primer. *International Journal of Computer Vision*, 38(1):9–13.

Freund, Y. and Schapire, R. E. (1995). *A desicion-theoretic generalization of online learning and an application to boosting*, pages 23–37. Springer Berlin Heidelberg, Berlin, Heidelberg.

Freund, Y. and Schapire, R. E. (1996). Experiments with a new boosting algorithm. In Saitta, L., editor, *Proceedings of the Thirteenth International Conference on Machine Learning (ICML 1996)*, pages 148–156. Morgan Kaufmann.

Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *Ann. Statist.*, 28(2):337–407.

Friedman, J. H. (1991). Multivariate adaptive regression splines. *Ann. Statist.*, 19(1):1–67.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Ann. Statist.*, 29(5):1189–1232.

Friedman, J. H. (2002). Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367–378.

Geisser, S. (1975). The predictive sample reuse method with applications. *J Am Stat Assoc*, 70(350):320–328.

Hastie, T. and Tibshirani, R. (1986). Generalized additive models. *Statistical Science*, 1:297–310.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition.* Springer Series in Statistics. Springer.

Ho, T. K. (1998). The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844.

Huber, P. J. (1964). Robust estimation of a location parameter. *Ann. Math. Statist.*, 35(1):73–101.

Hyafil, L. and Rivest, R. L. (1976). Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15 – 17.

Kass, G. V. (1980). An exploratory technique for investigating large quantities of categorical data. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 29(2):119–127.

Kearns, M. (1988). Thoughts on hypothesis boosting. Unpublished.

Kearns, M. and Valiant, L. G. (1989). Crytographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 433–444, New York, NY, USA. ACM.

Koenker, R. (2005). *Quantile Regression.* Econometric Society Monographs. Cambridge University Press.

Kozumi, H. and Kobayashi, G. (2011). Gibbs sampling methods for bayesian quantile regression. *Journal of Statistical Computation and Simulation*, 81(11):1565–1578.

Kuhn, M. and Johnson, K. (2013). *Applied Predictive Modeling.* SpringerLink : Bücher. Springer New York.

Lichman, M. (2013). UCI machine learning repository.

Lin, Y. (2002). Support vector machines and the bayes rule in classification. *Data Min. Knowl. Discov.*, 6(3):259–275.

Mason, L., Baxter, J., Bartlett, P. L., and Frean, M. R. (1999). Boosting algorithms as gradient descent. In *NIPS*.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

Nadaraya, E. A. (1964). On estimating regression. *Theory of Probability & Its Applications*, 9(1):141–142.

Nelder, J. A. and Wedderburn, R. W. M. (1972). Generalized linear models. *Journal of the Royal Statistical Society, Series A, General*, 135:370–384.

Nguyen, T. and Sanner, S. (2013). Algorithms for direct 0–1 loss optimization in binary classification. In Dasgupta, S. and Mcallester, D., editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1085–1093. JMLR Workshop and Conference Proceedings.

Nielsen, F. and Garcia, V. (2009). Statistical exponential families: A digest with flash cards. *CoRR*, abs/0911.4863.

Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York.

Platt, J. (1998). Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines. Technical report.

Quinlan, J. (1993). *C4.5: Programs for Machine Learning*. C4.5 - programs for machine learning / J. Ross Quinlan. Morgan Kaufmann Publishers.

Quinlan, R. J. (1992). Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore. World Scientific.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rakotomamonjy, A. and Canu, S. (2005). Frames, reproducing kernels, regularization and learning. *J. Mach. Learn. Res.*, 6:1485–1515.

Ridgeway, Greg with others, c. f. (2015). *gbm: Generalized Boosted Regression Models*. R package version 2.1.1.

Ridgeway, G. (2006). gbm: Generalized boosted regression models. *R package version*, 1(3).

Rosset, S. (2003). *Topics in Regularization and Boosting*. PhD thesis, Stanford university.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA.

Schapire, R. E. (1990). The strength of weak learnability. *Mach. Learn.*, 5(2):197–227.

Schapire, R. E. and Freund, Y. (2012). *Boosting: Foundations and Algorithms*. The MIT Press.

Schwarz, G. (1978). Estimating the dimension of a model. *Ann. Statist.*, 6(2):461–464.

Shen, Y. (2005). *Loss functions for binary classification and class probability estimation*. PhD thesis, University of Pennsylvania.

Shmueli, G. (2010). To explain or to predict? *Statist. Sci.*, 25(3):289–310.

Silverman, B. W. (1984). Spline smoothing: The equivalent variable kernel method. *Ann. Statist.*, 12(3):898–916.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958.

Steinwart, I. (2007). How to compare different loss functions and their risks. *Constructive Approximation*, 26(2):225–287.

Stone, M. (1974). Cross-Validatory Choice and Assessment of Statistical Predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 36(2):111–147.

Strobl, C., laure Boulesteix, A., and Augustin, T. (2006). Unbiased split selection for classification trees based on the gini index. Technical report.

Tewari, A. and Bartlett, P. L. (2014). Learning theory. In Diniz, P. S., Suykens, J. A., Chellappa, R., and Theodoridis, S., editors, *Signal Processing Theory and Machine Learning*, volume 1 of *Academic Press Library in Signal Processing*, pages 775–816. Elsevier.

Tipping, M. E. (2001). Sparse bayesian learning and the relevance vector machine. *J. Mach. Learn. Res.*, 1:211–244.

Torsten Hothorn, Kurt Hornik, A. Z. (2006). Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, 15(3):651–674.

Vapnik, V. N. (1999). An overview of statistical learning theory. *Trans. Neur. Netw.*, 10(5):988–999.

von Luxburg, U. and Schoelkopf, B. (2008). Statistical Learning Theory: Models, Concepts, and Results. *ArXiv e-prints*.

Watson, G. S. (1964). Smooth regression analysis. *SankhyĀĄ: The Indian Journal of Statistics, Series A (1961-2002)*, 26(4):359–372.

Wei-Yin Loh, Y.-S. S. (1997). Split selection methods for classification trees. *Statistica Sinica*, 7(4):815–840.

Young, G. and Smith, R. (2005). *Essentials of Statistical Inference*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press.

Zhang, T. and Yu, B. (2005). Boosting with early stopping: Convergence and consistency. *Ann. Statist.*, 33(4):1538–1579.

Zhou, Z.-H. (2012). *Ensemble Methods: Foundations and Algorithms*. Chapman & Hall/CRC, 1st edition.