Boosting

Jianghao Chu, Tae-Hwy Lee, Aman Ullah and Ran Wang

Abstract In the era of Big Data, selecting relevant variables from a potentially large pool of candidate variables becomes a newly emerged concern in macroeconomic researches, especially when the data available is high-dimensional, i.e. the number of explanatory variables (p) is greater than the length of the sample size (n). Common approaches include factor models, the principal component analysis and regularized regressions. However, these methods require additional assumptions that are hard to verify and/or introduce biases or aggregated factors which complicate the interpretation of the estimated outputs. This chapter reviews an alternative solution, namely *Boosting*, which is able to estimate the variables of interest consistently under fairly general conditions given a large set of explanatory variables. Boosting is fast and easy to implement which makes it one of the most popular machine learning algorithms in academia and industry.

1 Introduction

The term Boosting originates from the so-called hypothesis boosting problem in the distribution-free or probably approximately correct model of learning.

Jianghao Chu
Department of Economics, University of California, Riverside, e-mail: jianghao.chu@email.ucr.edu

Tae-Hwy Lee
Department of Economics, University of California, Riverside, e-mail: tae.lee@ucr.edu

Aman Ullah
Department of Economics, University of California, Riverside, e-mail: aman.ullah@ucr.edu

Ran Wang
Department of Economics, University of California, Riverside, e-mail: ran.wang@email.ucr.edu

In this model, the learner produces a classifier based on random samples from an unknown data generating process. Samples are chosen according to a fixed but unknown and arbitrary distribution on the population. The learner's task is to find a classifier that correctly classifies new samples from the data generating process as positive or negative examples. A weak learner produces classifiers that perform only slightly better than random guessing. A strong learner, on the other hand, produces classifiers that can achieve arbitrarily high accuracy given enough samples from the data generating process.

In a seminal paper, Schapire (1990) addresses the problem of improving the accuracy of a class of classifiers that perform only slightly better than random guessing. The paper shows the existence of a weak learner implies the existence of a strong learner and vice versa. A boosting algorithm is then proposed to convert a weak learner into a strong learner. The algorithm uses *filtering* to modify the distribution of samples in such a way as to force the weak learning algorithm to focus on the harder-to-learn parts of the distribution.

Not long after the relation between weak learners and strong learners are revealed, Freund and Schapire (1997) propose the Adaptive Boost (AdaBoost) for binary classification. AdaBoost performs incredibly well in practice and stimulates the invention of boosting algorithms for multi-class classifications. On the other hand, researchers try to explain the success of AdaBoost in a more theoretical way, e.g. Friedman et al (2000), Bartlett et al (2006) and Bartlett and Traskin (2007). Further understanding of the theory behind the success of boosting algorithms in turn triggers a bloom of Boosting algorithm with better statistical properties, e.g. Friedman (2001), Bühlmann and Yu (2003) and Mease et al (2007).

Boosting is undoubtedly the most popular machine learning algorithm in the on-line data science platform such as Kaggle. It is efficient and easy to implement. There are numerous packages in Python and R which implement Boosting algorithms in one way or another, e.g. *XBoost*. In the following sections, we will introduce the AdaBoost as well as other Boosting algorithms in detail together with examples to help the readers better understand the algorithms and statistical properties of the Boosting methods.

This chapter is organized as follows. Section 1 provides an overview on the origination and development of Boosting. Sections 2 and 3 are an introduction of AdaBoost which is the first practically feasible Boosting algorithm with its variants. Section 4 introduces a Boosting algorithm for linear regressions, namely L_2Boosting. Section 5 gives a generalization of the above mentioned algorithms which is called Gradient Boosting Machine. Section 6 gives more variants of Boosting, e.g. Boosting for nonlinear models. Section 7 provides applications of the Boosting algorithms in macroeconomic studies. In section 8 we summarize.

2 AdaBoost

The first widely used Boosting algorithm is AdaBoost which solves binary classification problems with great success. A large number of important variables in economics are binary. For example, whether the economy is going into expansion or recession, whether an individual is participating in the labor force, whether a bond is going to default, and etc. Let

$$\pi(\mathbf{x}) \equiv \Pr(y = 1|\mathbf{x}),$$

so that y takes value 1 with probability $\pi(\mathbf{x})$ and -1 with probability $1 - \pi(\mathbf{x})$. The goal of the researchers is to predict the unknown value of y given known information on $\mathbf{x}$.

2.1 AdaBoost algorithm

This section introduces the AdaBoost algorithm of Freund and Schapire (1997). The algorithm of AdaBoost is shown in Algorithm 1.

Let y be the binary class taking a value in $\{-1, 1\}$ that we wish to predict. Let $f_m(\mathbf{x})$ be the weak learner (weak classifier) for the binary target y that we fit to predict using the high-dimensional covariates $\mathbf{x}$ in the mth iteration. Let err_m denote the error rate of the weak learner $f_m(\mathbf{x})$, and $E_w(\cdot)$ denote the weighted expectation (to be defined below) of the variable in the parenthesis with weight w. Note that the error rate $E_w\left[1_{(y \neq f_m(\mathbf{x}))}\right]$ is estimated by $err_m = \sum_{i=1}^{n} w_i 1_{(y_i \neq f_m(x_i))}$ with the weight w_i given by step 2(d) from the previous iteration. n is the number of observations. The symbol $1_{(\cdot)}$ is the indicator function which takes the value 1 if a logical condition inside the parenthesis is satisfied and takes the value 0 otherwise. The symbol $\text{sign}(z) = 1$ if $z > 0$, $\text{sign}(z) = -1$ if $z < 0$, and hence $\text{sign}(z) = 1_{(z>0)} - 1_{(z<0)}$.

Remark 1 *Note that the presented version of Discrete AdaBoost in Algorithm 1 as well as Real AdaBoost (RAB), LogitBoost (LB) and Gentle AdaBoost (GAB) which will be introduced later in the next section are different from their original version when they were first introduced. The original version of these algorithms only output the class label. In this paper, we follow the idea of Mease et al (2007) and modified the algorithms to output both the class label and the probability prediction. The probability prediction is attained using*

$$\hat{\pi}(\mathbf{x}) = \frac{e^{F_M(\mathbf{x})}}{e^{F_M(\mathbf{x})} + e^{-F_M(\mathbf{x})}},$$

where $F_M(\mathbf{x})$ is the sum of weak learners in the algorithms.

Remark 2 *The only hyperparameter, i.e. the user specified parameter, in the AdaBoost as well as other Boosting algorithms is the number of iterations,*

Algorithm 1 Discrete AdaBoost (DAB, Freund and Schapire, 1997)

```
1. Start with weights $w_i = \frac{1}{n}, i = 1, \ldots, n$.
2. For $m = 1$ to $M$
    a. For $j = 1$ to $k$ (for each variable)
        i. Fit the classifier $f_{mj}(x_{ij}) \in \{-1, 1\}$ using weights $w_i$ on the training data.
        ii. Compute $err_{mj} = \sum_{i=1}^{n} w_i 1_{(y_i \neq f_{mj}(x_{ji}))}$.
    b. Find $\hat{j}_m = \arg\min_j err_{mj}$
    c. Compute $c_m = \log\left(\frac{1-err_{m,\hat{j}_m}}{err_{m,\hat{j}_m}}\right)$.
    d. Set $w_i \leftarrow w_i \exp[c_m 1_{(y_i \neq f_{m,\hat{j}_m}(x_{\hat{j}_m,i}))}]$, $i = 1, \ldots, n$, and normalize so that $\sum_{i=1}^{n} w_i = 1$.
3. Output the binary classifier sign$[F_M(\mathbf{x})]$ and the class probability prediction $\hat{\pi}(\mathbf{x}) = \frac{e^{F_M(\mathbf{x})}}{e^{F_M(\mathbf{x})} + e^{-F_M(\mathbf{x})}}$ where $F_M(\mathbf{x}) = \sum_{m=1}^{M} c_m f_{m,\hat{j}_m}(x_{\hat{j}_m})$.
```

M. It is also known as the stopping rule and is commonly chosen by cross-validation as well as information criterion such as AICc (Bühlmann and Yu, 2003). The choice of the stopping rule is embedded in most implementation of AdaBoost and should not be a concern for most users. Interesting readers could check Hastie et al (2009) for more details of cross-validation.

The most widely used weak learner is the classification tree. The simplest classification tree, the stump, takes the following functional form

$$f(x_j, a) = \begin{cases} 1 & x_j > a \\ -1 & x_j < a, \end{cases}$$

where the parameter a is found by minimizing the error rate

$$\min_a \sum_{i=1}^{n} w_i 1 \left(y_i \neq f(x_{ji}, a)\right).$$

The other functional form of the stump can be shown as exchanging the greater and smaller sign in the previous from

$$f(x_j, a) = \begin{cases} 1 & x_j < a \\ -1 & x_j > a, \end{cases}$$

where the parameter a is found by minimizing the same error rate.

2.2 An example

Now we present an example given by Ng (2014) for predicting the business cycles to help the readers understand the AdaBoost algorithm. Consider classifying whether the 12 months in 2001 is in expansion or recession using three-month lagged data of the help-wanted index (HWI), new orders ($NAPM$), and the 10yr-FF spread ($SPREAD$). The data are listed in Columns 2-4 of Table 1. The NBER expansion and recession months are listed in Column 5, where 1 indicates a recession month and -1 indicates a expansion month. We use a stump as the weak learner (f). The stump uses an optimally chosen threshold to split the data into two partitions. This requires setting up a finite number of grid points for HWI, $NAPM$, and $SPREAD$, respectively, and evaluating the goodness of fit in each partition.

Table 1: An Example

	Data: Lagged 3 Months				$f_1(\mathbf{x})$	$f_2(\mathbf{x})$	$f_3(\mathbf{x})$	$f_4(\mathbf{x})$	$f_5(\mathbf{x})$
	HWI	$NAPM$	$SPREAD$	y	HWI	$NAPM$	HWI	$SPREAD$	$NAPM$
Date	-0.066	48.550	0.244		< -0.044	< 49.834	< -0.100	> -0.622	< 47.062
2001.1	0.014	51.100	-0.770	-1	-1	-1	-1	-1	-1
2001.2	-0.091	50.300	-0.790	-1	1	-1	-1	-1	-1
2001.3	0.082	52.800	-1.160	-1	-1	-1	-1	-1	-1
2001.4	-0.129	49.800	-0.820	1	1	1	1	1	1
2001.5	-0.131	50.200	-0.390	1	1	-1	1	1	1
2001.6	-0.111	47.700	-0.420	1	1	1	1	1	1
2001.7	-0.056	47.200	0.340	1	1	1	1	1	1
2001.8	-0.103	45.400	1.180	1	1	1	1	1	1
2001.9	-0.093	47.100	1.310	1	1	1	1	1	1
2001.10	-0.004	46.800	1.470	1	-1	1	-1	1	1
2001.11	-0.174	46.700	1.320	1	1	1	1	1	1
2001.12	-0.007	47.500	1.660	-1	-1	1	-1	1	-1
c					0.804	1.098	0.710	0.783	0.575
Error rate					0.167	0.100	0.138	0.155	0

Ng (2014)

The algorithm begins by assigning an equal weight of $w_i^{(1)} = \frac{1}{n}$ where $n = 12$ to each observation. For each of the grid points chosen for HWI, the sample of y values is partitioned into parts depending on whether HWI_i exceeds the grid point or not. The grid point that minimizes classification error is found to be -0.044. The procedure is repeated with $NAPM$ as a splitting variable, and then again with $SPREAD$. A comparison of the three sets of residuals reveals that splitting on the basis of HWI gives the smallest weighted error. The first weak learner thus labels Y_i to 1 if $HWI_i < -0.044$. The outcome of the decision is given in Column 6. Compared with the NBER dates in Column 5, we see that months 2 and 10 are mislabeled, giving a misclassification rate of $\frac{2}{12} = 0.167$. This is err_1 of step 2(b). Direct calculations give $c_1 = \log(\frac{1-err_1}{err_1})$ of 0.804. The weights $w_i^{(2)}$ are updated to complete step 2(d). Months 2 and 10 now each have a weight of 0.25, while the remaining 10 observations each have a weight of 0.05. Three thresholds are again computed using weights $w^{(2)}$. Of the three, the $NAPM$ split gives the smallest weighted residuals. The weak learner for step 2 is identified. The classification based on the sign of

$$F_2(\mathbf{x}) = 0.804 \cdot 1_{(HWI<-0.044)} + 1.098 \cdot 1_{(NAPM<49.834)}$$

is given in Column 7. Compared with Column 5, we see that months 5 and 12 are mislabeled. The weighted misclassification rate is decreased to 0.100. The new weights $w_t^{(3)}$ are 0.25 for months 5 and 12, 0.138 for months 2 and 10, and 0.027 for the remaining months. Three sets of weighted residuals are again determined using new thresholds. The best predictor is again HWI with a threshold of -0.100. Classification based on the sign of $F_3(\mathbf{x})$ is given in Column 8, where

$$F_3(\mathbf{x}) = 0.804 \cdot 1_{(HWI<0.044)} + 1.098 \cdot 1_{(NAPM<48.834)} + 0.710 \cdot 1_{(HWI<-0.100)}.$$

The error rate after three steps actually increases to 0.138. The weak learner in round four is $1_{(SPREAD>-0.622)}$. After $NAPM$ is selected for one more round, all recession dates are correctly classified. The strong learner is an ensemble of five weak learners defined by $\text{sign}(F_5(\mathbf{x}))$, where

$$\begin{aligned} F_5(\mathbf{x}) = {} & 0.804 \cdot 1_{(HWI<-0.044)} + 1.098 \cdot 1_{(NAPM<49.834)} + 0.710 \cdot 1_{(HWI<-0.100)} \\ & +0.783 \cdot 1_{(SPREAD>-0.622)} + 0.575 \cdot 1_{(NAPM<47.062)}. \end{aligned}$$

Note that the same variable can be chosen more than once by AdaBoost which is the key difference from other stage-wise algorithms e.g. forward stage-wise regression. The weights are adjusted at each step to focus more on the misclassified observations. The final decision is based on an ensemble of models. No single variable can yield the correct classification, which is the premise of an ensemble decision rule.

For more complicated applications, several packages in the statistical programming language `R` provide off-the-shelf implementations of AdaBoost and its variants. For example, *JOUSBoost* gives an implementation of the Discrete AdaBoost algorithm from Freund and Schapire (1997) applied to decision tree classifiers and provides a convenient function to generate test sample of the algorithms.

2.3 AdaBoost: statistical view

After AdaBoost is invented and shown to be successful, numerous papers have attempted to explain the effectiveness of the AdaBoost algorithm. In an influential paper, Friedman et al (2000) show that AdaBoost builds an additive logistic regression model

$$F_M(\mathbf{x}) = \sum_{m=1}^{M} c_m f_m(\mathbf{x})$$

via Newton-like updates for minimizing the exponential loss

$$J(F) = E\left(e^{-yF(\mathbf{x})}\middle|\,\mathbf{x}\right).$$

We hereby show the above statement using the greedy method to minimize the exponential loss function iteratively as in Friedman et al (2000).

After m iterations, the current classifier is denoted as $F_m(\mathbf{x}) = \sum_{s=1}^{m} c_s f_s(\mathbf{x})$. In the next iteration, we are seeking an update $c_{m+1} f_{m+1}(\mathbf{x})$ for the function fitted from previous iterations $F_m(\mathbf{x})$. The updated classifier would take the form

$$F_{m+1}(\mathbf{x}) = F_m(\mathbf{x}) + c_{m+1} f_{m+1}(\mathbf{x}).$$

The loss for $F_{m+1}(\mathbf{x})$ will be

$$\begin{aligned} J(F_{m+1}(\mathbf{x})) &= J(F_m(\mathbf{x}) + c_{m+1} f_{m+1}(\mathbf{x})) \\ &= E\left[e^{-y(F_m(\mathbf{x}) + c_{m+1} f_{m+1}(\mathbf{x}))}\right]. \end{aligned} \tag{1}$$

Expand w.r.t. $f_{m+1}(\mathbf{x})$

$$\begin{aligned} J(F_{m+1}(\mathbf{x})) &\approx E\left[e^{-yF_m(\mathbf{x})}\left(1 - yc_{m+1} f_{m+1}(\mathbf{x}) + \frac{y^2 c_{m+1}^2 f_{m+1}^2(\mathbf{x})}{2}\right)\right] \\ &= E\left[e^{-yF_m(\mathbf{x})}\left(1 - yc_{m+1} f_{m+1}(\mathbf{x}) + \frac{c_{m+1}^2}{2}\right)\right]. \end{aligned}$$

The last equality holds since $y \in \{-1, 1\}$, $f_{m+1}(\mathbf{x}) \in \{-1, 1\}$, and $y^2 = f_{m+1}^2(\mathbf{x}) = 1$. $f_{m+1}(\mathbf{x})$ only appears in the second term in the parenthesis, so minimizing the loss function (1) w.r.t. $f_{m+1}(\mathbf{x})$ is equivalent to maximizing the second term in the parenthesis which results in the following conditional expectation

$$\max_f E\left[e^{-yF_m(\mathbf{x})} y c_{m+1} f_{m+1}(\mathbf{x}) \,|\mathbf{x}\right].$$

For any $c > 0$ (we will prove this later), we can omit c_{m+1} in the above objective function

$$\max_f E\left[e^{-yF_m(\mathbf{x})} y f_{m+1}(\mathbf{x}) \,|\mathbf{x}\right].$$

To compare it with the Discrete AdaBoost algorithm, here we define weight $w = w(y, \mathbf{x}) = e^{-yF_m(\mathbf{x})}$. Later we will see that this weight w is equivalent to that shown in the Discrete AdaBoost algorithm. So the above optimization can be seen as maximizing a weighted conditional expectation

$$\max_f E_w \left[y f_{m+1}(\mathbf{x}) | \mathbf{x} \right], \tag{2}$$

where $E_w(y|\mathbf{x}) := \frac{E(wy|\mathbf{x})}{E(w|\mathbf{x})}$ refers to a weighted conditional expectation. Note that (2) can be re-written as

$$\begin{aligned}
& E_w \left[y f_{m+1}(\mathbf{x}) | \mathbf{x} \right] \\
&= P_w(y=1|\mathbf{x}) f_{m+1}(\mathbf{x}) - P_w(y=-1|\mathbf{x}) f_{m+1}(\mathbf{x}) \\
&= \left[P_w(y=1|\mathbf{x}) - P_w(y=-1|\mathbf{x}) \right] f_{m+1}(\mathbf{x}) \\
&= E_w(y|\mathbf{x}) f_{m+1}(\mathbf{x}),
\end{aligned}$$

where $P_w(y|\mathbf{x}) = \frac{E(w|y,\mathbf{x})P(y|\mathbf{x})}{E(w|\mathbf{x})}$. Solve the maximization problem (2). Since $f_{m+1}(\mathbf{x})$ only takes 1 or -1, it should be positive whenever $E_w(y|\mathbf{x})$ is positive and -1 whenever $E_w(y|\mathbf{x})$ is negative. The solution for $f_{m+1}(\mathbf{x})$ is

$$f_{m+1}(\mathbf{x}) = \begin{cases} 1 & E_w(y|\mathbf{x}) > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Next, we minimize the loss function (1) w.r.t. c_{m+1}

$$c_{m+1} = \arg\min_{c_{m+1}} E_w \left(e^{-c_{m+1} y f_{m+1}(\mathbf{x})} \right),$$

where

$$E_w \left(e^{-c_{m+1} y f_{m+1}(\mathbf{x})} \right) = P_w(y = f_{m+1}(\mathbf{x})) e^{-c_{m+1}} + P_w(y \neq f_{m+1}(\mathbf{x})) e^{c_{m+1}}.$$

The first order condition is

$$\frac{\partial E_w \left(e^{-c y f_{m+1}(\mathbf{x})} \right)}{\partial c_{m+1}} = -P_w(y = f_{m+1}(\mathbf{x})) e^{-c_{m+1}} + P_w(y \neq f_{m+1}(\mathbf{x})) e^{c_{m+1}}.$$

Let

$$\frac{\partial E_w \left(e^{-c_{m+1} y f_{m+1}(\mathbf{x})} \right)}{\partial c_{m+1}} = 0,$$

and thus we have

$$P_w(y = f_{m+1}(\mathbf{x})) e^{-c_{m+1}} = P_w(y \neq f_{m+1}(\mathbf{x})) e^{c_{m+1}}.$$

Solving for c_{m+1}, we obtain

$$c_{m+1} = \frac{1}{2} \log \frac{P_w(y = f_{m+1}(\mathbf{x}))}{P_w(y \neq f_{m+1}(\mathbf{x}))} = \frac{1}{2} \log \left(\frac{1 - err_{m+1}}{err_{m+1}} \right),$$

where $err_{m+1} = P_w\left(y \neq f_{m+1}\left(\mathbf{x}\right)\right)$ is the error rate of $f_{m+1}\left(\mathbf{x}\right)$. Note that $c_{m+1} > 0$ as long as the error rate is smaller than 50%. Our assumption $c_{m+1} > 0$ holds for any learner that is better than random guessing.

Now we have finished the steps of one iteration and can get our updated classifier by

$$F_{m+1}\left(\mathbf{x}\right) \leftarrow F_m\left(\mathbf{x}\right) + \left(\frac{1}{2}\log\left(\frac{1-err_{m+1}}{err_{m+1}}\right)\right) f_{m+1}\left(\mathbf{x}\right).$$

Note that in the next iteration, the weight we defined w_{m+1} will be

$$w_{m+1} = e^{-yF_{m+1}(\mathbf{x})} = e^{-y\left(F_m(\mathbf{x})+c_{m+1}f_{m+1}(\mathbf{x})\right)} = w_m \times e^{-c_{m+1}f_{m+1}(\mathbf{x})y}.$$

Since $-yf_{m+1}\left(\mathbf{x}\right) = 2 \times 1_{\{y\neq f_{m+1}(\mathbf{x})\}} - 1$, the update is equivalent to

$$w_{m+1} = w_m \times e^{\left(\log\left(\frac{1-err_{m+1}}{err_{m+1}}\right)1_{[y\neq f_{m+1}(\mathbf{x})]}\right)} = w_m \times \left(\frac{1-err_{m+1}}{err_{m+1}}\right)^{1_{[y\neq f_{m+1}(\mathbf{x})]}}.$$

Thus the function and weight update are of an identical form to those used in AdaBoost. AdaBoost could do better than any single weak classifier since it iteratively minimizes the loss function via a Newton-like procedure.

Interestingly, the function $F\left(\mathbf{x}\right)$ from minimizing the exponential loss is the same as maximizing a logistic log-likelihood. Let

$$\begin{aligned} J\left(F\left(\mathbf{x}\right)\right) &= E\left[E\left(e^{-yF(\mathbf{x})}\middle|\,\mathbf{x}\right)\right] \\ &= E\left[P\left(y=1|\mathbf{x}\right)e^{-F(\mathbf{x})} + P\left(y=-1|\mathbf{x}\right)e^{F(\mathbf{x})}\right]. \end{aligned}$$

Taking derivative w.r.t. $F\left(\mathbf{x}\right)$ and making it equal to zero, we obtain

$$\frac{\partial E\left(e^{-yF(\mathbf{x})}|\mathbf{x}\right)}{\partial F\left(\mathbf{x}\right)} = -P\left(y=1|\mathbf{x}\right)e^{-F(\mathbf{x})} + P\left(y=-1|\mathbf{x}\right)e^{F(\mathbf{x})} = 0.$$

Therefore,

$$F^*\left(\mathbf{x}\right) = \frac{1}{2}\log\left[\frac{P\left(y=1|\mathbf{x}\right)}{P\left(y=-1|\mathbf{x}\right)}\right].$$

Moreover, if the true probability is

$$P\left(y=1|\mathbf{x}\right) = \frac{e^{2F(\mathbf{x})}}{1+e^{2F(\mathbf{x})}},$$

for $Y = \frac{y+1}{2}$, the log-likelihood is

$$E\left(\log L|\mathbf{x}\right) = E\left[2YF\left(\mathbf{x}\right) - \log\left(1+e^{2F(\mathbf{x})}\right)\middle|\,\mathbf{x}\right].$$

The solution $F^*(\mathbf{x})$ that maximizes the log-likelihood must equal the $F(\mathbf{x})$ in the true model $P(y=1|\mathbf{x}) = \frac{e^{2F(\mathbf{x})}}{1+e^{2F(\mathbf{x})}}$. Hence,

$$\begin{aligned} e^{2F^*(\mathbf{x})} &= P(y=1|\mathbf{x})\left(1+e^{2F^*(\mathbf{x})}\right) \\ e^{2F^*(\mathbf{x})} &= \frac{P(y=1|\mathbf{x})}{1-P(y=1|\mathbf{x})} \\ F^*(\mathbf{x}) &= \frac{1}{2}\log\left[\frac{P(y=1|\mathbf{x})}{P(y=-1|\mathbf{x})}\right]. \end{aligned}$$

AdaBoost that minimizes the exponential loss yields the same solution as the logistic regression that maximizes the logistic log-likelihood.

From the above, we can see that AdaBoost gives high weights to and thus, focuses on the samples that are not correctly classified by the previous weak learners. This is exactly what Schapire (1990) referred to as *filtering* in Section 1.

3 Extensions To AdaBoost Algorithms

In this section, we introduce three extensions of (Discrete) AdaBoost (DAB) which is shown in Algorithm 1: namely, Real AdaBoost (RAB), LogitBoost (LB) and Gentle AdaBoost (GAB). We discuss how some aspects of the DAB may be modified to yield RAB, LB and GAB. In the previous section, we learned that Discrete AdaBoost minimizes an exponential loss via iteratively adding a binary weaker learner to the pool of weak learners. The addition of a new weak learner can be seen as taking a step on the direction that loss function descents in the Newton method. There are two major ways to extend the idea of Discrete AdaBoost. One focuses on making the minimization method more efficient by adding a more flexible weak learner. The other is to use different loss functions that may lead to better results. Next, we give an introduction to three extensions of Discrete AdaBoost.

3.1 Real AdaBoost

Algorithm 2 Real AdaBoost (RAB, Friedman et al, 2000)

```
1. Start with weights $w_i = \frac{1}{n}, i = 1, \dots, n$.
2. For $m = 1$ to $M$
    a. For $j = 1$ to $k$ (for each variable)
        i. Fit the classifier to obtain a class probability estimate $p_m(x_j) = \hat{P}_w(y = 1|x_j) \in [0, 1]$ using weights $w_i$ on the training data.
        ii. Let $f_{mj}(x_j) = \frac{1}{2} \log \frac{p_m(x_j)}{1-p_m(x_j)}$.
        iii. Compute $err_{mj} = \sum_{i=1}^{n} w_i 1_{(y_i \neq \mathtt{sign}(f_{mj}(x_{ji})))}$.
    b. Find $\hat{j}_m = \arg\min_j err_{mj}$.
    c. Set $w_i \leftarrow w_i \exp[-y_i f_{m,\hat{j}_m}(x_{\hat{j}_m,i})]$, $i = 1, \dots, n$, and normalize so that $\sum_{i=1}^{n} w_i = 1$.
3. Output the classifier sign$[F_M(\mathbf{x})]$ and the class probability prediction $\hat{\pi}(\mathbf{x}) = \frac{e^{F_M(\mathbf{x})}}{e^{F_M(\mathbf{x})}+e^{-F_M(\mathbf{x})}}$ where $F_M(\mathbf{x}) = \sum_{m=1}^{M} f_m(\mathbf{x})$.
```

Real AdaBoost that Friedman et al (2000) propose focuses solely on improving the minimization procedure of Discrete AdaBoost. In Real AdaBoost, the weak learners are continuous comparing to Discrete AdaBoost where the weak learners are binary (discrete). Real AdaBoost is minimizing the exponential loss with continuous updates where Discrete AdaBoost minimizes the exponential loss with discrete updates. Hence, Real AdaBoost is more flexible with the step size and direction of the minimization and minimizes the exponential loss faster and more accurately. However, Real AdaBoost also imposes restriction that the classifier must produces a probability prediction which reduces the flexibility of the model. As pointed out in the numerical examples by Chu et al (2018), Real AdaBoost may achieve a larger in-sample training error due to the flexibility of its model. On the other hand, this also reduces the chance of over-fitting and would in the end achieve a smaller out-of-sample test error.

3.2 LogitBoost

Friedman et al (2000) also propose LogitBoost by minimizing the Bernoulli log-likelihood via an adaptive Newton algorithm for fitting an additive logistic regression model. LogitBoost extends Discrete AdaBoost in two ways. First, it uses the Bernoulli log-likelihood instead of the exponential loss function as a loss function. Furthermore, it updates the classifier by adding a linear model instead of a binary weak learner.

Algorithm 3 LogitBoost (LB, Friedman et al, 2000)

```
1. Start with weights $w_i = \frac{1}{n}, i = 1, \ldots, n$, $F(\mathbf{x}) = 0$ and probability estimates $p(x_i) = \frac{1}{2}$.
2. For $m = 1$ to $M$
   a. Compute the working response and weights
      $$z_i = \frac{y_i^* - p(x_i)}{p(x_i)(1 - p(x_i))}$$
      $$w_i = p(x_i)(1 - p(x_i))$$
   b. For $j = 1$ to $k$ (for each variable)
      i. Fit the function $f_{mj}(x_{ji})$ by a weighted least-squares regression of $z_i$ to $x_{ji}$ using weights $w_i$ on the training data.
      ii. Compute $err_{mj} = 1 - R^2_{mj}$ where $R^2_{mj}$ is the coefficient of determination from the weighted least-squares regression.
   c. Find $\hat{j}_m = \arg\min_j err_{mj}$
   d. Update $F(\mathbf{x}) \leftarrow F(\mathbf{x}) + \frac{1}{2} f_{m,\hat{j}}(x_{\hat{j}})$ and $p(\mathbf{x}) \leftarrow \frac{e^{F(\mathbf{x})}}{e^{F(\mathbf{x})} + e^{-F(\mathbf{x})}}$.
3. Output the classifier $\text{sign}[F_M(\mathbf{x})]$ and the class probability prediction $\hat{\pi}(\mathbf{x}) = \frac{e^{F_M(\mathbf{x})}}{e^{F_M(\mathbf{x})} + e^{-F_M(\mathbf{x})}}$ where $F_M(\mathbf{x}) = \sum_{m=1}^{M} f_{m,\hat{j}_m}(x_{\hat{j}_m})$.
```

In LogitBoost, continuous weak learner is used similarly to Real AdaBoost. However, LogitBoost specifies the use of a linear weak learner while Real AdaBoost allows any weak learner that returns a probability between zero and one. A more fundamental difference here is that LogitBoost uses the Bernoulli log-likelihood as a loss function instead of the exponential loss. Hence, LogitBoost is more similar to logistic regression than Discrete AdaBoost and Real AdaBoost. As pointed out in the numerical examples by Chu et al (2018), LogitBoost has the smallest in-sample training error but the largest out-of-sample test error. This implies that while LogitBoost is the most flexible of the four, it suffers the most from over-fitting.

3.3 Gentle AdaBoost

Algorithm 4 Gentle AdaBoost (GAB, Friedman et al, 2000)

```
1. Start with weights $w_i = \frac{1}{n}, i = 1, \ldots, n$.
2. For $m = 1$ to $M$
   a. For $j = 1$ to $k$ (for each variable)
      i. Fit the regression function $f_{mj}(x_{ji})$ by weighted least-squares of $y_i$ on $x_{ji}$ using
         weights $w_i$ on the training data.
      ii. Compute $err_{mj} = 1 - R^2_{mj}$ where $R^2_{mj}$ is the coefficient of determination from
          the weighted least-squares regression.
   b. Find $\hat{j}_m = \arg\min_j err_{mj}$
   c. Set $w_i \leftarrow w_i \exp[-y_i f_{m,\hat{j}_m}(x_{\hat{j}_m,i})]$, $i = 1, \ldots, n$, and normalize so that $\sum_{i=1}^n w_i =$
      $1$.
3. Output the classifier $\text{sign}[F_M(\mathbf{x})]$ and the class probability prediction $\hat{\pi}(\mathbf{x}) =$
   $\frac{e^{F_M(\mathbf{x})}}{e^{F_M(\mathbf{x})}+e^{-F_M(\mathbf{x})}}$ where $F_M(\mathbf{x}) = \sum_{m=1}^M f_{m,\hat{j}_m}(x_{\hat{j}_m})$.
```

In Friedman et al (2000), Gentle AdaBoost extends Discrete AdaBoost in the sense that it allows each weak learner to be a linear model. This is similar to LogitBoost and more flexible than Discrete AdaBoost and Real AdaBoost. However, it is closer to Discrete AdaBoost and Real AdaBoost than LogitBoost in the sense that Gentle AdaBoost, Discrete AdaBoost and Real AdaBoost all minimize the exponential loss while LogitBoost minimizes the Bernoulli log-likelihood. On the other hand, Gentle AdaBoost is more similar to Real AdaBoost than Discrete AdaBoost since the weak learners are continuous and there is no need to find an optimal step size for each iteration because the weak learner is already optimal. As pointed out in the numerical examples by Chu et al (2018), Gentle Boost often lies between Real AdaBoost and LogitBoost in terms of in-sample training error and out-of-sample test error.

4 L_2Boosting

In addition to classification, the idea of boosting can also be applied to regressions. Bühlmann and Yu (2003) propose L_2Boosting that builds a linear model by minimizing the L_2 loss. Bühlmann (2006) further proves the consistency of L_2Boosting in terms of predictions. L_2Boosting is the simplest and perhaps most instructive Boosting algorithm for economists and econometricians. It is very useful for regression, in particular in the presence of high-dimensional explanatory variables.

We consider a simple linear regression

$$y = \mathbf{x}\beta + u,$$

where y is the dependent variable, $\mathbf{x}$ is the independent variable and $u \sim N(0,1)$. Note that the number of independent variables $\mathbf{x}$ could be high-dimensional, i.e. the number of independent variables in $\mathbf{x}$ can be larger than the number of observations.

This model, in the low dimension case, can be estimated by the ordinary least squares. We minimize the sum of squared errors

$$L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2,$$

where

$$\hat{y}_i = \mathbf{x}_i \hat{\beta}.$$

The solution to the problem is

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

The residual from the previous problem is

$$\hat{u_i} = y_i - \hat{y}_i.$$

In the high-dimension case, the ordinary least squares method falls down because the matrix $(\mathbf{X}'\mathbf{X})$ is not invertible. Hence, we need to use a modified least squares method to get over the high-dimension problem.

The basic idea of L_2Boosting is to use only one explanatory variable at a time. Since the number of variables p is larger than the length of the sample period n, the matrix $\mathbf{X}'\mathbf{X}$ is not invertible. However, if we use only one variable in one particular iteration, the matrix $\mathbf{x}_j'\mathbf{x}_j$ is a scalar and thus invertible. In order to exploit the information in the explanatory variables, in the following iterations, we can use other explanatory variables to fit the residuals which are the unexplained part from previous iterations. L_2Boosting can be seen as iteratively use the least squares technique to explain the residuals from the previous least squares regressions. In the L_2Boosting algorithm, we use the least squares technique to fit the dependent variable y with only one independent variable $\mathbf{x}_j$. Then, we iteratively take the residual from the previous regression as the new dependent variable y and fit the new dependent variable with, again, only one independent variable $\mathbf{x}_j$. The detailed description of L_2Boosting is listed in Algorithm 5.

The stopping parameter M is the main tuning parameter which can be selected using cross-validation or some information criterion in practice. Bühlmann and Yu (2003) propose to use the corrected AIC to choose the stopping parameter. According to Bühlmann and Yu (2003), the square of the bias of the L_2Boosting decays exponentially fast with increasing M, the variance increases exponentially slow with increasing M, and $\lim_{M\to\infty} MSE = \sigma^2$.

Algorithm 5 L_2Boosting (Bühlmann and Yu, 2003)

```
1. Start with $y_i$ from the training data.
2. For $m = 1$ to $M$
    a. For $j = 1$ to $k$ (for each variable)
        i. Fit the regression function $y_i = \beta_{m,0,j} + \beta_{m,j} x_{ji} + u_i$ by least-squares of $y_i$ on $x_{ji}$.
        ii. Compute $err_{mj} = 1 - R^2_{mj}$ where $R^2_{mj}$ is the coefficient of determination from the least-squares regression.
    b. Find $\hat{j}_m = \arg\min_j err_{mj}$
    c. Set $y_i \leftarrow y_i - \hat{\beta}_{m,0,\hat{j}_m} - \hat{\beta}_{m,\hat{j}_m} x_{\hat{j}_m,i}$, $i = 1, \ldots, n$.
3. Output the final regression model $F_M(\mathbf{x}) = \sum_{m=1}^{M} \beta_{m,0,\hat{j}_m} + \hat{\beta}_{m,\hat{j}_m} x_{\hat{j}_m}$.
```

L_2Boosting is computationally simple and successful if the learner is sufficiently weak. If the learner is too strong, then there will be over-fitting problem as in all the other boosting algorithms. Even though it is straightforward for econometricians to use the simple linear regression as the weak learner, Bühlmann and Yu (2003) also suggest using smoothing splines and classification and regression trees as the weak learner.

5 Gradient Boosting

This section discusses the Gradient Boosting Machine first introduced by Friedman (2001). Breiman (2004) shows that the AdaBoost algorithm can be represented as a steepest descent algorithm in function space which we call functional gradient descent (FGD). Friedman et al (2000) and Friedman (2001) then developed a more general, statistical framework which yields a direct interpretation of boosting as a method for function estimation. In their terminology, it is a 'stage-wise, additive modeling' approach. Gradient Boosting is a generalization of AdaBoost and L_2Boosting. AdaBoost is a version of Gradient Boosting that uses the exponential loss and L_2Boosting is a version of Gradient Boosting that uses the L_2 loss.

5.1 Functional gradient descent

Before we introduce the algorithm of Gradient Boosting, let us talk about functional gradient descent in a general way. We consider $F(\mathbf{x})$ to be the function of interest and minimize a risk function $R(F) = E(L(y, F))$ with respect to $F(\mathbf{x})$. For example, in the L_2Boosting, the loss function $L(y, F(\mathbf{x}))$ is the L_2 loss, i.e. $L(y, F(\mathbf{x})) = (y - F(\mathbf{x}))^2$. Notice that we do not impose any parametric assumption on the functional form of $F(\mathbf{x})$, and hence, the solution $F(\mathbf{x})$ is entirely nonparametric.

The Functional Gradient Descent minimizes the risk function $R(F)$ at each $\mathbf{x}$ directly with respect to $F(\mathbf{x})$. In each iteration m, like in gradient descent, we look for a pair of optimal direction $f_m(\mathbf{x})$ and step size c_m. The optimal direction at $\mathbf{x}$ is the direction that the loss function $R(F)$ decreases the fastest. Hence, the optimal direction

$$f_m(\mathbf{x}) = E_y \left[-\frac{\partial L(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \middle| \mathbf{x} \right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}.$$

The optimal step size c_m can be found given $f_m(\mathbf{x})$ by a line search

$$c_m = \arg\min_{c_m} E_{y,\mathbf{x}} L(y, F_{m-1}(\mathbf{x}) + c_m f_m(\mathbf{x})).$$

Next, we update the estimated function $F(\mathbf{x})$ by

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + c_m f_m(\mathbf{x}).$$

Thus, we complete one iteration of Gradient Boosting. In practice, the stopping iteration, which is the main tuning parameter, can be determined via cross-validation or some information criteria. The choice of step size c is of minor importance, as long as it is 'small', such as $c = 0.1$. A smaller value of c typically requires a larger number of boosting iterations and thus more computing time, while the predictive accuracy will be better and tend to over-fit less likely.

5.2 Gradient boosting algorithm

The algorithm of Gradient Boosting is shown in Algorithm 6.

Algorithm 6 Gradient Boosting (GB, Friedman, 2001)

```
1. Start with $F_0(\mathbf{x}) = \arg\min_{const} \sum_{i=1}^{n} L(y_i, const)$.
2. For $m = 1$ to $M$
    a. calculate the pseudo-residuals $r_i^m = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}$, $i = 1, \ldots, n$.
    b. $f_m(\mathbf{x}) = \arg\min_{f_m(\mathbf{x})} \sum_{i=1}^{N} (r_i^m - f_m(\mathbf{x}_i))^2$.
    c. $c_m = \arg\min_c \sum_{i=1}^{N} L(y_i, F_{m-1}(\mathbf{x}_i) + c_m f_m(\mathbf{x}_i))$.
    d. $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + c_m f_m(\mathbf{x})$.
3. Output $F_M(\mathbf{x}) = \sum_{m=1}^{M} c_m f_m(\mathbf{x})$.
```

In theory, any fitting criterion that estimates the conditional expectation could be used to fit the negative gradient at step $1(a)$. In Gradient Boosting, the negative gradient is also called 'pseudo-residuals' r_i^m and Gradient Boosting fits this residuals in each iteration. The most popular choice to fit the residuals is the Classification/Regression Tree which we will discuss in detail in Section 14.5.3.

5.3 Gradient boosting decision tree

Gradient Boosting Decision Tree (GBDT) or Boosting Tree is one of the most important methods for implementing nonlinear models in data mining, statistics, and econometrics. According to the results of data mining tasks at the data mining challenges platform, *Kaggle*, most of the competitors choose Boosting Tree as their basic technique to model the data for predicting tasks.

Obviously, Gradient Boosting Decision Tree combines the decision tree and gradient boosting method. The gradient boosting is the gradient descent in functional space,

$$f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \lambda_m \left(\frac{\partial L}{\partial f}\right)_m,$$

where m is the number of iteration, L is the loss function we need to optimize, λ_m is the learning rate. In each round, we find the best direction $-\left(\frac{\partial L}{\partial f}\right)_m$ to minimize the loss function. In gradient boosting, we can use some simple function to find out the best direction. That is, we use some functions to fit the 'pseudo-residuals' of the loss function. In AdaBoost, we often use the decision stump, a line or hyperplane orthogonal to only one axis, to fit the residual. In the Boosting Tree, we choose a decision tree to handle this task. Also, the decision stump could be seen as a decision tree with one root node and two leaf nodes. Thus, the Boosting Tree is a natural way to generalize Gradient Boosting.

Basically, the Boosting Tree learns an additive function, which is similar to other aggregating methods like Random Forest. But the decision trees are grown very differently among these methods. In the Boosting Tree, a new decision tree is growing based on the 'error' from the decision tree which grew in the last iteration. The updating rule comes from Gradient Boosting method and we will dive into the details later.

Suppose we need to implement a regression problem given samples $(y_i, x_i), i = 1, ..., n$. If we choose the square loss function, the 'pseudo-residual' should be $r_i^m = -\left(\frac{\partial L}{\partial f}\right)_m = -\left(\frac{\partial (y-f)^2}{\partial f}\right)_m = 2(y - f_m)$.

The algorithm of Gradient Boosting Decision Tree is shown in Algorithm 7.

Algorithm 7 Gradient Boosting Decision Tree (Tree Boost, Friedman, 2001)

```
1. Initially, estimate the first residual via $r_i^0 = -2(y_i - \bar{y}) = -2(y_i - f_1(x_i))$.
2. For $m = 1$ to $M$
    a. Based on new samples $(r_i^m, x_i), i = 1, ..., n$, fit a regression tree $h_m(\mathbf{x})$.
    b. Let $f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \lambda_m h_m(\mathbf{x})$, then, $\lambda_m = \arg\min_\lambda L(y, f_m(\mathbf{x}) + \lambda h_m(\mathbf{x}))$.
    c. Update $f_{m+1}(\mathbf{x})$ via $f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \lambda_m h_m(\mathbf{x})$.
    d. Calculate the new residual $r_i^{m+1} = -2(y_i - f_{m+1}(x_i))$, then update the new samples
       as $(r_i^{m+1}, x_i), i = 1, ....$
3. Output the Gradient Boosting Decision Tree $F_M(\mathbf{x}) = \sum_{m=1}^{M} \lambda_m f_m(\mathbf{x})$.
```

According to Algorithm 7, the main difference between Gradient Boosting and Boosting Tree is at step $1(a)$. In Boosting Tree, we use a decision tree to fit the 'residual' or the negative gradient. In other words, Boosting Tree implement the Functional Gradient Descent by following the functional gradient learned by the decision tree.

Additionally, to implement Gradient Boosting Decision Tree, we need to choose several hyperparameters: (1) N, the number of terminal nodes in trees; (2) M, the number of iterations in the boosting procedure.

Firstly, N, the number of terminal nodes in trees, controls the maximum allowed level of interaction between variables in the model. With $N = 2$ (decision stumps), no interaction between variables is allowed. With $N = 3$, the model may include effects of the interaction between up to two variables, and so on. Hastie et al (2009) comment that typically $4 < N < 8$ work well for boosting and results are fairly insensitive to the choice of N in this range, $N = 2$ is insufficient for many applications, and $N > 10$ is unlikely to be required. Figure 14.1 shows the test error curves corresponding to the different number of nodes in Boosting Tree. We can see that Boosting with decision stumps provides the best test error. When the number of nodes increases, the final test error increases, especially in boosting with trees containing 100 nodes. Thus, practically, we often choose $4 < N < 8$.

Secondly, to the number of iterations M, we will discuss that in Section 5.4.1 in detail, which is related to the regularization method in Boosting Tree.

5.4 Regularization

By following the discussion above, the Gradient Boosting Decision Trees method contains more trees when M is larger. A further issue is related to over-fitting. That is, when there are increasingly large numbers of decision trees, Boosting Tree can fit any data with zero training error, which leads to a bad test error on new samples. To prevent the model from over-fitting, we will introduce two ways to resolve this issue.

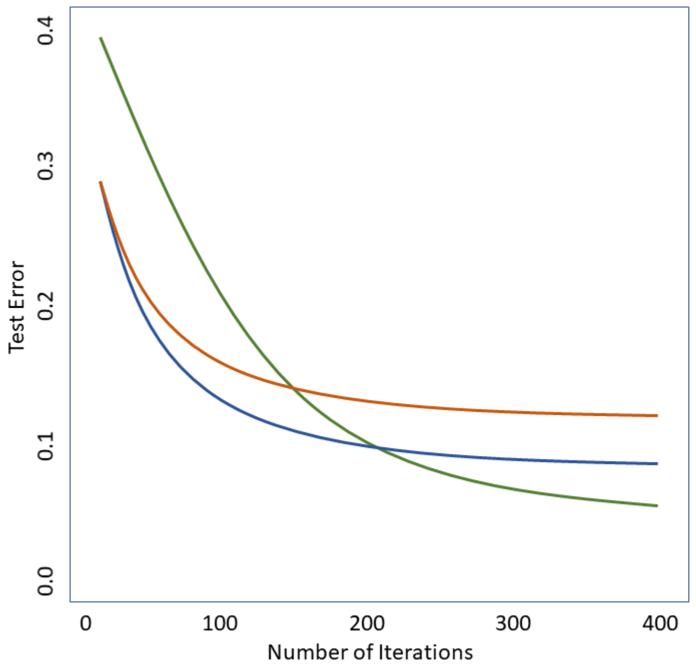

Figure 1: Illustration of Gradient Boosting Decision Trees with Different Nodes (green: decision stump; red line: tree with 10 leaf nodes; blue: tree with 100 leaf nodes)

5.4.1 Early stopping

A simple way to resolve this issue is to control the number of iterations M in the Boosting Trees. Basically, we can treat M as a hyper-parameter during the training procedure of Boosting Trees. Cross-Validation is an effective way to select hyperparameters including M. Since Boosting Trees method is equivalent to the steepest gradient descent in functional space, selecting the optimal M means that this steepest gradient descent will stop at the Mth iteration.

5.4.2 Shrinkage method

The second way to resolve the problem of over-fitting is shrinkage. That is, we add a shrinkage parameter during the training process. Let us consider the original formula for updating Boosting Trees:

$$f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \lambda_m h_m(\mathbf{x}). \tag{3}$$

In the Boosting Trees, we first fit $h_m(\mathbf{x})$ based on a decision tree. Then, we optimize λ_m for the best step size. Thus, we can shrink the step size by adding a shrinkage parameter ν:

$$f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \nu\lambda_m h_m(\mathbf{x}). \tag{4}$$

Obviously, if we set $\nu = 1$, Equation (4) is equivalent to Equation (3). Suppose we set $0 \leq \nu \leq 1$, it can shrink the optimal step size λ_m to $\nu\lambda_m$, which leads to a slower optimization. In other words, comparing to the original Boosting Tree, Shrinkage Boosting Tree learns the unknown function slower but more precise in each iteration. As a consequence, to a given $\nu < 1$, we need more steps M to minimize the error. Figure 14.2 shows this consequence. To a binary classification problem, we consider two measures: the test set deviations, which is the negative binomial log-likelihood loss on the test set, and the test set misclassification error. In the left and right panels, we can see that, with the shrinkage parameter less than 1, Boosting Tree typically need more iterations to converge but it can hit a better prediction result. Friedman (2001) found that a smaller ν will lead to a larger optimal M but the test errors in the new datasets are often better than the original Boosting Tree. Although large M may need more computational resources, this method may be inexpensive because of the faster computers.

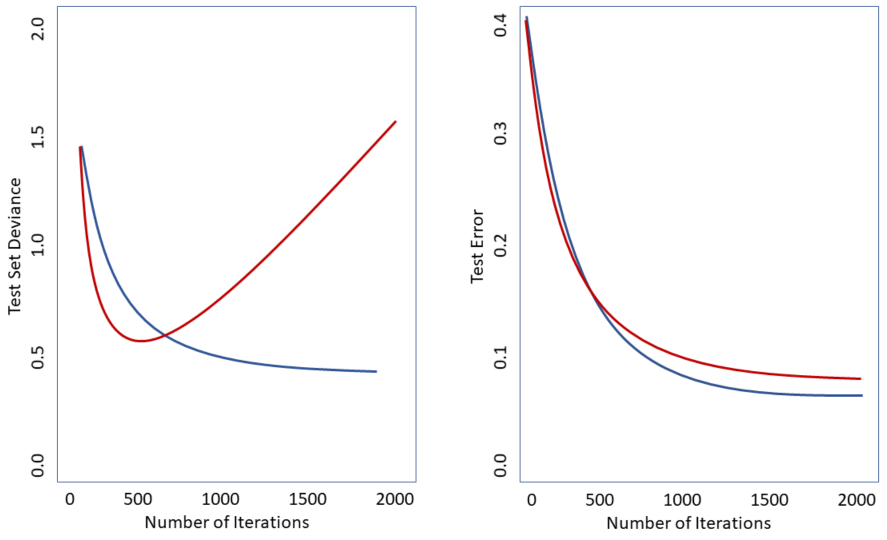

Figure 2: Gradient Boosting Decision Tree (6 leaf nodes) with Different Shrinkage Parameters (blue: Shrinkage $\nu = 0.6$; red: No shrinkage)

5.5 Variable importance

After training Boosting Tree, the next question is to identify the variable importance. Practically, we often train boosting tree on a dataset with a large number of variables and we are interested in finding important variables for analysis.

Generally, this is also an important topic in tree-based models like Random Forest discussed in chapter 13. Since Boosting Tree method is also an additive trees aggregating, we can use I_j^2 to measure the importance of a variable j:

$$I_j^2 = \frac{1}{M}\sum_{m=1}^{M} I_j^2(m),$$

and $I_j^2(m)$ is the importance of variable j for the mth decision tree:

$$I_j^2(m) = \sum_{t=1}^{T_m-1} e_t^2 I(v(t)_m = j),$$

where T_m is the number of internal nodes (non-leaf nodes) in the mth decision tree, $v(t)_m$ is the variable selected by node t, and e_t is the error improvement based on before and after splitting the space via variable $v(t)_m$.

In Random Forest or Bagging Decision Tree method, we can measure the variable importance based on the so-called Out-of-Bag errors. In Boosting Tree, since there are no Out-of-Bag samples, we can only use I_j^2. In practice, OOB-based method and I_j^2 method often provide similar results and I_j^2 works very well especially when M is very large.

Let us consider an example about the relative importance of variables for predicting spam mail via Boosting Trees. The input variable $\mathbf{x}$ could be a vector of counts of the keywords or symbols in one email. The response y is a binary variable ($Spam$, $Not\ Spam$). We regress y on $\mathbf{x}$ via Boosting Tree and then calculate the variable importance for each word or symbol. On one hand, the most important keywords and symbols may be '!', '$', 'free', that is related to money and free; on the other hand, the keywords like '3d', 'addresses' and 'labs' are not very important since they are relatively neutral. Practically, the variable importance measure often provides a result consistent with common sense.

6 Recent Topics In Boosting

In this section, we will focus on four attractive contributions of Boosting in recent years. First of all, we introduce two methods that are related to Boosting in time series and volatility models respectively. They are relevant topics in macroeconomic forecasting. The third method is called Boosting with Momentum (BOOM), which is a generalized version of Gradient Boosting and is more robust than the Gradient Boosting. The fourth method is called Multi-Layered Gradient Boosting Decision Tree, which is a deep learning method via non-differentiable Boosting Tree and shed light on representation learning in tabular data.

6.1 Boosting in nonlinear time series models

In macroeconomic forecasting, nonlinear time series models are widely used in the last 40 years. For example, Tong and Lim (1980) discuss the Threshold Autoregressive (TAR) model to describe the time dependence when the time series is higher or lower than a threshold value. Chan and Tong (1986) propose the Smooth Transition Autoregressive (STAR) model to catch the nonlinear time dependence changing continuously between two states over time. Basically, nonlinear time series models not only perform better than linear time series models but also provide a clear way to analyze the nonlinear dependence among time series data.

Although nonlinear time series models are successful in macroeconomic time series modeling, we also need to consider their assumptions and model settings so that they can work for time series modeling. Unfortunately, in the era of big data, they cannot handle the large datasets since they often contain more complicated time dependence and higher dimensional variables along time that does not satisfy the assumptions. Essentially, the Boosting method provides an effective and consistent way to handle the time series modeling among big datasets especially with relatively fewer assumptions required.

Robinzonov et al (2012) discuss the details of Boosting for nonlinear time series models. Suppose we have a bunch of time series dataset $z_t = (y_{t-1}, ..., y_{t-p}, x_{1,t-1}, ..., x_{q,t-1}, ..., x_{1,t-p}, ..., x_{q,t-p}) = (y_{t-1}, ..., y_{t-p}, \mathbf{x}_{t-1}, ..., \mathbf{x}_{t-p}) \in \mathbb{R}^{(q+1)p}$, where z_t is the information set at time t, y is a series of endogenous variable with lags of p and $(\mathbf{x}_{t-1}, ..., \mathbf{x}_{t-p})$ is a q dimensional vector series with lags of p. Consider a nonlinear time series model for the conditional mean of y_t:

$$E(y_t|z_t) = F(z_t) = F(y_{t-1}, ..., y_{t-p}, x_{1,t-1}, ..., x_{q,t-1}, ..., x_{1,t-p}, ..., x_{q,t-p}),$$

where $F(z_t)$ is an unknown nonlinear function. Chen and Tsay (1993) discuss an additive form of $F(z_t)$ for nonlinear time series modeling, which is called Nonlinear Additive Auto Regressive with exogenous variables (NAARX):

$$\begin{aligned} E(y_t|z_t) &= F(z_t) \\ &= \sum_{i=1}^{p} f_i(y_{t-i}) + \sum_{i=1}^{p} f_{1,i}(x_{1,t-i}) + ... + \sum_{i=1}^{p} f_{q,i}(x_{q,t-i}) \\ &= \sum_{i=1}^{p} f_i(y_{t-i}) + \sum_{j=1}^{q}\sum_{i=1}^{p} f_{j,i}(x_{j,t-i}). \end{aligned}$$

To optimize the best $F(z_t)$ given data, we need to minimize the loss function:

$$\hat{F}(z_t) = \arg\min_{F(z_t)} \frac{1}{T}\sum_{t=1}^{T} L(y_t, F(z_t)).$$

For example, we can use L_2 loss function $L(y_t, F(z_t)) = \frac{1}{2}(y_t - F(z_t))^2$. If we consider a parametric function $F(z_t, \beta)$, we can have the following loss function:

$$\hat{\beta} = \arg\min_{\beta} \frac{1}{T}\sum_{t=1}^{T} L(y_t, F(z_t; \beta)).$$

Since the true function of $E(y|z)$ has the additive form, the solution to the optimization problem should be represented by a sum over a bunch of estimated functions. In Boosting, we can use M different weak learner to implement:

$$F(z_t; \hat{\beta}^M) = \sum_{m=0}^{M} \nu h(z_t; \hat{\gamma}^m),$$

where ν is a shrinkage parameter for preventing over-fitting. Similar to original gradient boosting, in each iteration, we can generate a 'pseudo residual' term $r^m(z_t)$ which is:

$$r^m(z_t) = -\left.\frac{\partial L(y_t, F)}{\partial F}\right|_{F=F(z_t; \hat{\beta}^{m-1})}.$$

Thus, we can optimize $\hat{\gamma}^m$ based on the loss function:

$$\hat{\gamma}^m = \arg\min_{\gamma} \sum_{t=1}^{T} L(r^m(z_t), h(z_t; \gamma)).$$

After that, we update the $F(z_t; \hat{\beta}^m)$ as:

$$F(z_t; \hat{\beta}^m) = F(z_t; \hat{\beta}^{m-1}) + \nu h(z_t; \hat{\gamma}^m).$$

Now go back to the NAARX model. Since each function f only contains one variable, y_{t-i} or $x_{j,t-i}$, we can construct same additive form via L_2 Boosting. That is, in each iteration, we only choose one variable from the whole vector

$z_t = (y_{t-1}, ..., y_{t-p}, ..., x_{q,t-1}, ..., x_{q,t-p})$ and then fit a weak learner. This is called Component-wise Boosting.

Robinzonov et al (2012) discussed two methods of component-wise boosting with different weak learners: linear weak learner and P-Spline weak learner. The first method is called component-wise linear weak learner. For this method, we choose a linear function with one variable of z_t as a weak learner in each iteration. The algorithm of Component-wise Boosting with linear weak learner is shown in Algorithm 8.

Algorithm 8 Component-wise Boosting with Linear Weak Learner (Robinzonov et al, 2012)

```
1. Start with $y_t$ from training data.
2. For $m = 1$ to $M$
   a. For $j = 1$ to $(1+q)p$ (for each variable)
      i. Fit the regression function $y_t = \beta_{m,0,j} + \beta_{m,j} z_{j,t} + u_t$ by least-squares of $y_t$ on
         $z_{j,t}$ on the training data.
      ii. Compute $err_{mj} = 1 - R^2_{mj}$ from the weighted least-squares regression.
   b. Find $\hat{j}_m = \arg\min_j err_{mj}$.
   c. Set $y_t \leftarrow y_t - \hat{\beta}_{m,0,\hat{j}_m} - \hat{\beta}_{m,\hat{j}_m} z_{t,\hat{j}_m}$, $t = 1, \ldots, T$.
3. Output the final regression model $F_M(z) = \sum_{m=1}^{M} \beta_{m,0,\hat{j}_m} + \hat{\beta}_{m,\hat{j}_m} z_{\hat{j}_m}$.
```

Obviously, this method only provides a linear solution like an Autoregressive model with exogenous variables (ARX). We can also consider more complicated weak learner such that the nonlinear components could be caught. In the paper, P-Spline with B base learners is considered as the weak learner. The algorithm of Component-wise Boosting with P-Spline weak learner is shown in Algorithm 9

Algorithm 9 Component-wise Boosting with P-Spline Weak Learner (Robinzonov et al, 2012)

```
1. Start with $y_t$ from training data.
2. For $m = 1$ to $M$
   a. For $j = 1$ to $(1+q)p$ (for each variable)
      i. Fit the P-Spline with $B$ Base learners $\hat{y}_t = Spline_m(z_{j,t})$ by regressing $y_t$ on $z_{j,t}$
         on the training data.
      ii. Compute $err_{mj} = 1 - R^2_{mj}$ from the P-Spline regression.
   b. Find $\hat{j}_m = \arg\min_j err_{mj}$.
   c. Set $y_i \leftarrow y_t - \hat{y}_t$, $t = 1, \ldots, T$.
3. Output the final regression model $F_M(z) = \sum_{m=1}^{M} Spline_m(z_{\hat{j}_m})$.
```

6.2 Boosting in volatility models

Similarly to Boosting in nonlinear time series models for the mean, it is possible to consider Boosting in volatility models, like GARCH. Audrino and Bühlmann (2016) discussed volatility estimation via functional gradient descent for high-dimensional financial time series. Matías et al (2010) compare Boost-GARCH with other methods, like neural networks GARCH.

Let us begin with the classic $GARCH(p,q)$ model by Bollerslev (1986):

$$\begin{aligned} &y_t = \mu + e_t, t = 1, ..., T \\ &e_t \sim N(0, h_t) \\ &h_t = c + \sum_{i=1}^{p} \alpha_i e_{t-i}^2 + \sum_{j=1}^{q} \beta_j h_{t-j}. \end{aligned}$$

We can implement a Maximum Likelihood Estimation (MLE) method to estimate all the coefficients. Generally, consider a nonlinear formula of the volatility function h_t:

$$h_t = g(e_{t-1}^2, ..., e_{t-p}^2, h_{t-1}, ..., h_{t-q}) = g(E_t^2, H_t),$$

where $E_t^2 = (e_{t-1}^2, ..., e_{t-p}^2)$ and $H_t = (h_{t-1}, ..., h_{t-q})$. Similarly to NAARX model, we can consider a additive form of the function g:

$$h_t = \sum_{m=1}^{M} g_m(E_t^2, H_t).$$

For simplicity, let $p = q = 1$, we have:

$$h_t = \sum_{m=1}^{M} g_m(e_{t-1}^2, h_{t-1}).$$

Thus, we can use L_2 Boosting to approximate the formula above. Since we use MLE to estimate the original GARCH model, for Boost-GARCH, we can also introduce the likelihood function for calculating the 'pseudo residual' $r_{t,m}$ instead of using the loss function. Finally, Boost-GARCH can fit an additive nonlinear formula as the estimation of h_t:

$$\hat{h}_t = \sum_{m=1}^{M} f_m(e_{t-1}^2, h_{t-1})$$

The algorithm of Boost-GARCH $(1, 1)$ is shown in Algorithm 10.

Algorithm 10 Boost-GARCH (1,1) (Audrino and Bühlmann, 2016)

```
1. Start with estimating a linear GARCH (1,1) model:
   $$y_t = \mu + e_t, t = 1, ..., T$$
   $$e_t \sim N(0, h_t)$$
   $$h_t = c + \alpha_1 e_{t-1}^2 + \beta_1 h_{t-1}$$
2. Getting the $\hat{\mu}_0, \hat{h}_{t-1,0}$
3. For $m = 1$ to $M$
   a. Calculate the residual:
      $$e_{t,m}^2 = (y_t - \hat{\mu}_{m-1})^2$$
      $$r(\mu)_{t,m} = -\left(\frac{\partial L}{\partial \mu}\right)_m = \frac{y_t - \hat{\mu}_{t,m}}{\hat{h}_{t,m}}$$
      $$r(h)_{t,m} = -\left(\frac{\partial L}{\partial h_t}\right)_m = \frac{1}{2}\left(\frac{(y_t - \hat{\mu}_{t,m})^2}{\hat{h}_{t,m}^2} - \frac{1}{\hat{h}_{t,m}}\right)$$
   b. Fit a nonlinear base learner $\hat{y}_t = f_m(e_{t-1}^2, h_{t-1})$ by regressing $r(h)_{t,m}$ on
      $e_{t-1,m}^2, \hat{h}_{t-1,m}$ on the training data.
   c. Set $\hat{h}_{t,m} \leftarrow \hat{h}_{t,m-1} + f_m(e_{t,m-1}^2, \hat{h}_{t,m-1})$.
4. Output the final regression model $\hat{h}_t = \sum_{m=1}^{M} f_m(e_{t-1}^2, h_{t-1})$.
```

6.3 Boosting with momentum (BOOM)

In Section 5 on Gradient Boosting, we show that Gradient Boosting can be represented as a steepest gradient descent in functional space. In the optimization literature, gradient descent is widely discussed on its properties. First, gradient descent is easily revised for many optimization problems. Second, gradient descent often finds out good solutions no matter the optimization problem is convex or nonconvex.

But gradient descent also suffers from some drawbacks. Let us consider the plots of loss surface in Figure 14.4. Suppose the loss surface is convex. Obviously, gradient descent should converge to the global minimum eventually. But what we can see in the plot (a) is that the gradient descent converges very slow and the path of gradient descent is a zig-zag path. Thus, original gradient descent may spend a long time on converging to the optimal solution. Furthermore, the convergence is worse in a non-convex optimization problem.

To resolve this issue, a very practical way is to conside 'momentum' term to the gradient descent updating rule:

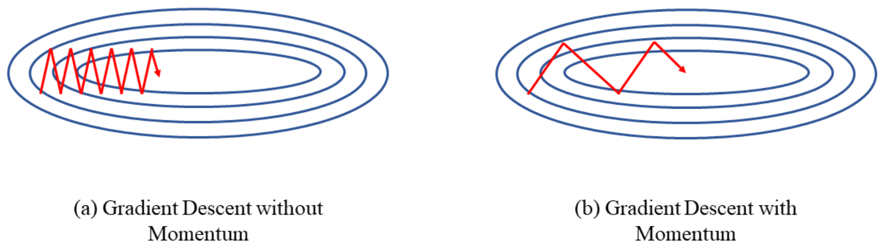
(a) Gradient Descent without Momentum
(b) Gradient Descent with Momentum

Figure 3: Gradient Descent without and with Momentum

$$\theta_{m+1} = \theta_m - \lambda V_m,$$
$$V_m = V_{m-1} + \nu \left(\frac{\partial L}{\partial \theta}\right)_m,$$

where θ_m is the parameter we want to optimize at mth iteration, V_m is the momentum term with another corresponding updating rule.

In the original gradient descent method, we have $V_m = \left(\frac{\partial L}{\partial \theta}\right)_m$. In $(m+1)$th iteration, the parameter θ_{m+1} is updated by following the gradient $\left(\frac{\partial L}{\partial \theta}\right)_m$ only. But when we consider momentum term, the parameter θ_m is updated by following the updating direction in previous iteration V_{m-1} and the gradient $\left(\frac{\partial L}{\partial \theta}\right)_m$ together. Intuitively, this is just like the effect of momentum in physics. When a ball is rolling down from the top, even though it comes to a flat surface, it keeps rolling for a while because of momentum.

Plot (b) in Figure 14.3 illustrates the difference between Gradient Descent without and with Momentum. Comparing to the path of convergence in the plot (a), if we consider momentum in gradient descent, the path becomes better and spends less time on moving to the optimal solution which is shown in plot (b).

As the generalized version of gradient descent in function space, gradient boosting may also suffer from the same problem when the loss surface is complicated. Thus, a natural way to improve the gradient boosting method is considering the momentum term in its updating rule. Mukherjee et al (2013) discuss a general analysis of a fusion of Nesterov's accelerated gradient with parallel coordinate descent. The resulting algorithm is called Boosting with Momentum (BOOM). Namely, BOOM retains the momentum and convergence properties of the accelerated gradient method while taking into account the curvature of the objective function. They also show that BOOM is especially effective in large scale learning problems. Algorithm 11 provides the procedure of BOOM via Boosting Tree.

Algorithm 11 Gradient Boosting Decision Tree with Momentum (Mukherjee et al (2013),Friedman (2002))

```
1. Initially, estimate the first residual via $r_i^0 = -2(y_i - \bar{y}) = -2(y_i - f_1(x_i))$.
2. For $m = 1$ to $M$
    a. Based on new samples $(r_i^m, x_i), i = 1, ..., n$, fit a regression tree $h_m(\mathbf{x})$.
    b. Let $V_m = V_{m-1} + \lambda_m h_m(\mathbf{x})$.
    c. Let $f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \nu V_m$, then optimize $\lambda_m$ via $\lambda_m = \arg\min_\lambda L(y, f_m(\mathbf{x}) + \nu V_m) = \arg\min_\lambda L(y, f_m(\mathbf{x}) + \nu(V_{m-1} + \lambda h_m(\mathbf{x})))$.
    d. Update $f_{m+1}(\mathbf{x})$ via $f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \nu V_m$.
    e. Calculate the new residual $r_i^{m+1} = -2(y_i - f_{m+1}(x_i))$, then update the new samples as $(r_i^{m+1}, x_i), i = 1, ..., n$.
3. Output the Gradient Boosting Decision Tree $F_M(\mathbf{x}) = \sum_{m=1}^M \nu V_m$.
```

The main difference between Boosting with Momentum and ordinary Boosting Tree is a step to update V_m. Also, we have one more hyperparameter to decide ν, which decides the fraction of gradient information saved for next iteration updating of $f_m(\mathbf{x})$. Practically, we set $0.5 < \nu < 0.9$ but it is more reasonable to tune ν via cross-validation.

This method can be generalized to Stochastic Gradient Boosting discussed by Friedman (2002). Algorithm 12 shows the procedure of BOOM via Stochastic Gradient Boosting Tree.

Algorithm 12 Stochastic Gradient Boosting Decision Tree with Momentum (Mukherjee et al (2013),Friedman (2002))

```
1. Initially, randomly select a subset of the samples $(y_i, x_i), i = 1, ..., n_s$, where $0 < n_s < n$.
2. Estimate the first residual via $r_i^0 = -2(y_i - \bar{y}) = -2(y_i - f_1(x_i))$.
3. For $m = 1$ to $M$.
    a. Based on new samples $(r_i^m, x_i), i = 1, ..., n_s$, fit a regression tree $h_m(\mathbf{x})$.
    b. Let $V_m = V_{m-1} + \lambda_m h_m(\mathbf{x})$.
    c. Let $f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \nu V_m$, then optimize $\lambda_m$ via $\lambda_m = \arg\min_\lambda L(y, f_m(\mathbf{x}) + \nu V_m) = \arg\min_\lambda L(y, f_m(\mathbf{x}) + \nu(V_{m-1} + \lambda h_m(\mathbf{x})))$.
    d. Update $f_{m+1}(\mathbf{x})$ via $f_{m+1}(\mathbf{x}) = f_m(\mathbf{x}) + \nu V_m$.
    e. Calculate the new residual $r_i^{m+1} = -2(y_i - f_{m+1}(x_i))$, then update the new samples as $(r_i^{m+1}, x_i), i = 1, ..., n_s$.
4. Output the Gradient Boosting Decision Tree $F_M(\mathbf{x}) = \sum_{m=1}^M \nu V_m$.
```

There some differences between BOOM with Boosting Tree and Stochastic Boosting Tree. In Boosting Tree, we use all the n samples to update the decision tree in each iteration. But Stochastic Boosting Tree randomly selects $\frac{n_s}{n}$ fraction of samples to grow a decision tree in each iteration. When the sample size n is increasingly large, selecting a subset of samples could be a better and more efficient way to implement the Boosting Tree algorithm.

6.4 Multi-layered gradient boosting decision tree

Last 10 years witnessed the dramatic development in the fields about deep learning, which mainly focus on distilling hierarchical features via multi-layered neural networks automatically. From 2006 deep learning methods have changed so many areas like computer vision and natural language processing.

The multi-layered representation is the key ingredient of deep neural networks. Thus, the combination of multi-layered representation and Boosting Tree are expected in handling very complicated tabular data analysis tasks. But there are few research papers exploring multi-layered representation via non-differentiable models, like Boosted Decision Tree. That is, the gradient-based optimization method which is always used in training multi-layered neural networks cannot be introduced in training multi-layered Boosting methods.

Ji et al (2018) explored one way to construct Multi-Layered Gradient Boosting Decision Tree (mGBDT) with an explicit emphasis on exploring the ability to learn hierarchical representations by stacking several layers of regression GBDTs. The model can be jointly trained by a variant of target propagation across layers, without the need to derive back-propagation or to require differentiability.

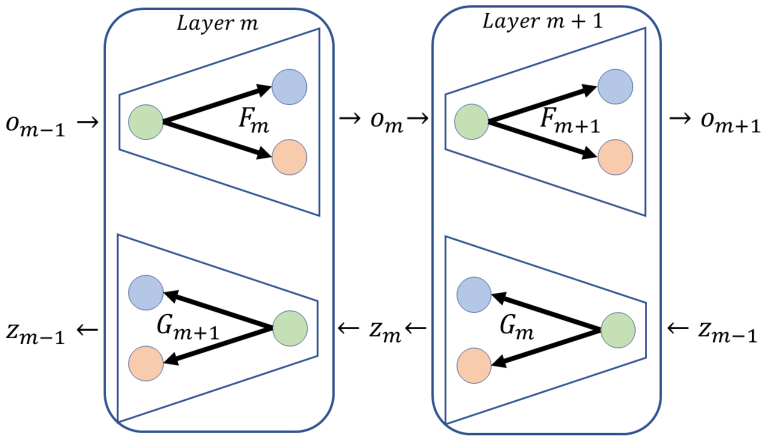

Figure 4: Illustration of Multi-Layered Gradient Boosting Decision Tree

Figure 14.4 provides the structure of a Multi-Layered Gradient Boosting Decision Tree. $F_m, m = 1, ..., M$ are the M layers of a mGBDT. Similar to the multi-layered neural networks, the input o_0 is transformed to $o_1, ..., o_M$ via $F_1, ..., F_M$. Then, the final output o_M is the prediction of the target variable y. But all the F_m are constructed via gradient boosting decision tree, we cannot training them via back-propagation method used in training multi-layered neural networks. Ji et al (2018) introduced another group of functions $G_m, m = 1, ..., M$ and corresponding variables $z_m, m = 1, ..., M$.

Intuitively, the group of function G_m are introduced for achieving back-propagation algorithm in non-differentiable Boosting Tree. To train Multi-layered Gradient Boosting Decision Tree, firstly, we use 'forward propagation' method to calculate all the $o_m, m = 1, ..., M$. Secondly, to (o_m), G_m are trained to reconstruct o_m via optimizing the loss function $L(o_m, G_m(F_m(o_m)))$. That is, we train G_m to learn 'back-propagation'. Then, after training all the $G_m, m = 1, ..., M$, we can do 'back-propagation' to generate $z_m, m = 1, ..., M$, that represents the information to each layer. Next, to the pairs of (z_m, z_{m-1}), we train F_m to optimize another loss function $L(z_m, F_m(z_{m-1}))$. Finally, we can update all the F_m and G_m via Boosting Tree method. Algorithm 13 shows the procedure of Multi-Layered Gradient Boosting Decision Tree.

Algorithm 13 Multi-Layered Gradient Boosting Decision Tree (Ji et al (2018))

```
1. Input: Number of layers $M$, layer dimension $d_m$, samples $(y_i, x_i), i = 1, ..., n$. Loss
   function $L$. Hyper-parameters $\alpha$, $\gamma$, $K_1$, $K_2$, $T$, $\sigma^2$.
2. Initially, set $F_m^0 = Initialize(M, d_m), m = 1, ..., M$;
3. For $t = 1$ to $T$
   a. Propagate the $o_0$ to calculate $o_m = F(o_{m-1}), m = 1, ..., M$
   b. $z_M^t = o_M - \alpha \frac{\partial L(y, o_M)}{\partial o_M}$
   c. For $m = M$ to 2
      i. $G_m^t = G_m^{t-1}$
      ii. $o_{m-1}^{noise} = o_{m-1} + \epsilon, \epsilon \sim N(0, diag(\sigma^2))$
      iii. $L_m^{inv} = L(o_m^{noise}, G_m^t(F_m^{t-1}(o_m^{noise})))$
      iv. for $k = 1$ to $K_1$
         A. $r_k = -\frac{\partial L_m^{inv}}{\partial G_m^t(F_m^{t-1}(o_m^{noise}))}$
         B. Fit a decision tree $h_k$ to $r_k$
         C. $G_m^t = G_m^t + \gamma h_k$
      v. $z_{m-1} = G_m^t(z_m)$
   d. For $m = 1$ to $M$
      i. $F_m^t = F_m^{t-1}$
      ii. $L_m = L(z_m^t, F_m^t(o_{m-1}))$ using gradient boosting decision tree
      iii. for $k = 1$ to $K_2$
         A. $r_k = -\frac{\partial L_m}{\partial F_j^t(o_m)}$
         B. Fit a decision tree $h_k$ to $r_k$
         C. $F_m^t = F_m^t + \gamma h_k$
      iv. $o_m = F_m^t(o_{m-1})$
4. Output the trained multi-layered gradient boosting decision tree.
```

Ji et al (2018) suggested to optimize $L^{inv} = L(o_m^{noise}, G_m^t(F_m^{t-1}(o_m^{noise})))$ instead of $L_m^{inv} = L(o_m, G_m^t(F_m^{t-1}(o_m)))$ to make the training of G_m more robust. Also, the authors found that the multi-layered gradient boosting decision tree is very robust to most hyper parameters. Without fine-tuning the parameters, this method can achieve very attractive results.

Furthermore, consider the noisy loss function from the perspective of minimizing the reconstruction error, this process could be seen as an encoding-decoding process. First, in each layer F_m encodes the input via a nonlinear transform. Then, G_m learns how to decode the transformed output back to the original input. This is similar to the Auto Encoder method in deep learning. Thus, we can also use the Multi-layered Gradient Boosting Decision Tree to do encoding-decoding, which shed a light on implementing unsupervised learning tasks in the tabular data in economics.

7 Boosting In Macroeconomics and Finance

Boosting methods are widely used in classification and regression. Gradient Boosting implemented in the packages, like *XGBoost* and *LightGBM*, is a very popular algorithm among data science competitions and industrial applications. In this section, we discuss four applications of boosting algorithms in macroeconomics.

7.1 Boosting in predicting recessions

Ng (2014) uses boosting to predict recessions 3, 6, and 12 months ahead. Boosting is used to screen as many as 1,500 potentially relevant predictors consisting of 132 real and financial time series and their lags. The sample period is 1961:12011:12. In this application, boosting is used to select relevant predictors from a set of potential predictors as well as probability estimation and prediction of the recessions. In particular, the analysis uses the Bernoulli loss function as implemented in the *GBM* package of Ridgeway (2007). The package returns the class probability instead of classifications. For recession analysis, the probability estimate is interesting in its own right, and the flexibility to choose a threshold other than one-half is convenient.

7.2 Boosting diffusion indices

Bai and Ng (2009) use boosting to select and estimate the predictors in factoraugmented autoregressions. In their application, boosting is used to make 12 months ahead of forecast on inflation, the change in Federal Funds rate, the growth rate of industrial production, the growth rate of employment, and the unemployment rate. A sample period from 1960:1 to 2003:12 was used for a total of 132 times series. They use two boosting algorithms, namely L_2Boosting and Block Boosting.

7.3 Boosting with Markov-switching

Adam et al (2017) propose a novel class of flexible latent-state time series regression models called Markov-switching generalized additive models for location, scale, and shape. In contrast to conventional Markov-switching regression models, the presented methodology allows users to model different state-dependent parameters of the response distribution - not only the mean, but also variance, skewness and kurtosis parameters - as potentially smooth functions of a given set of explanatory variables. The authors also propose an estimation approach based on the EM algorithm using the gradient boosting framework to prevent over-fitting while simultaneously performing variable selection. The feasibility of the suggested approach is assessed in simulation experiments and illustrated in a real-data setting, where the authors model the conditional distribution of the daily average price of energy in Spain over time.

7.4 Boosting in financial modeling

Rossi and Timmermann (2015) construct a new procedure for estimating the covariance risk measure in ICAPM model. First, one or more economic activity indices are extracted from macroeconomic and financial variables for estimating the covariance matrix. Second, given realized covariance matrix as the covariance matrix measure, Boosting Regression Tree is applied in projecting realized covariance matrix on the indices extracted in the first step. Lastly, predictions of the covariance matrix are made based on the nonlinear function approximated by Boosting Regression Tree and applied into the analysis of ICAPM method.

8 Summary

In this chapter, we focus on Boosting method. We start with an introduction of the well known AdaBoost. Several variants of AdaBoost, like Real AdaBoost, LogitBoost and Gentle AdaBoost are also discussed. Then, we consider in regression problem and introduce L_2 Boosting. Next, Gradient Boosting and Gradient Boosting Decision Tree are discussed in theory and practice. Then, we introduce the several variants of Gradient Boosting such as

Component-wise Boosting and Boost-GARCH for nonlinear time series modeling, Boosting with Momentum and multi-layered Boosting Tree. Finally, we discuss several applications of Boosting in macroeconomic forecasting and financial modeling.

References

Adam T, Mayr A, Kneib T (2017) Gradient boosting in Markov-switching generalized additive models for location, scale and shape

Audrino F, Bühlmann P (2016) Volatility estimation with functional gradient descent for very high-dimensional financial time series. The Journal of Computational Finance 6(3):65–89

Bai J, Ng S (2009) Boosting diffusion indices. Journal of Applied Econometrics 24(4):607–629

Bartlett PL, Traskin M (2007) AdaBoost is consistent. Journal of Machine Learning Research 8:2347–2368

Bartlett PL, Jordan MI, McAuliffe JD (2006) Convexity, classification, and risk bounds. Journal of the American Statistical Association 101(473):138–156

Bollerslev T (1986) Generalized autoregressive conditional heteroskedasticity. Journal of Econometrics 31(3):307–327

Breiman L (2004) Population theory for boosting ensembles. Annals of Statistics 32(1):1–11

Bühlmann P (2006) Boosting for high-dimensional linear models. Annals of Statistics 34(2):559–583

Bühlmann P, Yu B (2003) Boosting with the L2 loss: Regression and classification. Journal of the American Statistical Association 98(462):324–339

Chan KS, Tong H (1986) on estimating thresholds in autoregressive models. Journal of Time Series Analysis 7(3):179–190

Chen R, Tsay RS (1993) Nonlinear additive ARX models. Journal of the American Statistical Association 88(423):955–967

Chu J, Lee TH, Ullah A (2018) Component-wise AdaBoost algorithms for high-dimensional binary classification and class probability prediction. In: Handbook of Statistics, Elsevier

Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences 55:119–139

Friedman J, Hastie T, Tibshirani R (2000) Additive logistic regression: A statistical view of boosting. Annals of Statistics 28(2):337–407

Friedman JH (2001) Greedy function approximation: A gradient boosting machine. The Annals of Statistics 29:1189–1232

Friedman JH (2002) Stochastic gradient boosting. Computational Statistics and Data Analysis 38(4):367–378

Hastie T, Tibshirani R, Friedman J (2009) The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd edn. Springer

Ji F, Yang Y, Zhi-Hua Z (2018) Multi-layered gradient boosting decision trees. Proceedings of the 32nd Conference on Neural Information Processing Systems

Matías J, Febrero-Bande M, González-Manteiga W, Reboredo J (2010) Boosting GARCH and neural networks for the prediction of heteroskedastic time series. Mathematical and Computer Modelling 51(3-4):256–271

Mease D, Wyner A, Buja A (2007) Cost-weighted boosting with jittering and over/under-sampling: Jous-boost. Journal of Machine Learning Research 8:409–439

Mukherjee I, Canini K, Frongillo R, Singer Y (2013) Parallel boosting with momentum. In: Blockeel H, Kersting K, Nijssen S ŽF (ed) Machine Learning and Knowledge Discovery in Databases, vol 8190 LNAI, Springer, Berlin, Heidelberg, pp 17–32

Ng S (2014) Viewpoint: Boosting recessions. Canadian Journal of Economics 47(1):1–34

Ridgeway G (2007) Generalized boosted models: A guide to the gbm package. Tech. Rep. 4

Robinzonov N, Tutz G, Hothorn T (2012) Boosting techniques for nonlinear time series models. AStA Advances in Statistical Analysis 96(1):99–122

Rossi AG, Timmermann A (2015) Modeling covariance risk in Merton's ICAPM. Review of Financial Studies 28(5):1428–1461

Schapire RE (1990) The strength of weak learnability. Machine Learning 5:197–227

Tong H, Lim KS (1980) Threshold autoregression, limit cycles and cyclical data. Journal of the Royal Statistical Society Series B (Methodological) 42(3):245–292