A Robust Boosting Algorithm for Chemical Modeling

*Ville A. Satopää, Richard D. De Veaux

\textsuperscript{a}Department of Statistics
The Wharton School of the University of Pennsylvania
Philadelphia, PA 19104-6340, USA,
satopaa@wharton.upenn.edu
(413) 652 0540

\textsuperscript{b}Williams College, Williamstown, MA, 01267, USA,
deveaux@williams.edu
Abstract

Bagging and boosting have become increasingly important ensemble methods for combining models in the data mining and machine learning literature. We review the basic ideas of these methods, propose a new robust boosting algorithm based on a non-convex loss function and compare the performance of these methods to both simulated and real data sets with and without contamination.

Key words: Bagging, Boosting, Data Mining, Machine Learning, Robust

1. Introduction

Data mining methods have become increasingly important in the analysis of scientific data, especially in the field of bioinformatics (e.g. [1], [2], [3], [4]). Recently, much attention has been given to the so-called ensemble methods introduced in the 1990s. Now, only a few decades later, the literature on these methods is large (e.g. [5], [6], [7], [8]). The overall goal in ensemble learning is to construct a collection of “weak learners” such as decision trees with only a few (or a single) splits. These smaller models are then combined into a master model usually by weighting the individual predictions. In the case of a continuous response (a regression problem), the final predictions are the (possibly weighted) average of the individual predictions. In the case of a categorical response variable (a classification problem), the final prediction is the weighted or modal response from the collection of models. Two of the most popular ensemble methods are “bagging” (bootstrap aggregating) and “boosting.” Bagging works by resampling the data, fitting a model to each resulting data set, and then averaging the final predictions of the individual models. Boosting iteratively fits a model to a data set with weights that are proportional to the accuracy of the models from the earlier iterations. Iterating in this manner it produces a series of models that are then combined for the final prediction.

2. Bagging

The idea of bagging is simple: Given a training set of data \( \{(y_i, x_i)\}^n \), produce \( i=1, \ldots, k \) samples of size \( n \) by sampling from the training set uniformly and with replacement. That is, each of the \( k \) new samples is the same size as the original training set, but some rows are sampled more than once and some rows not at all. Each resulting sample is called a bootstrap sample. While bootstrap samples have been used to study the properties of many statistical estimators, their purpose here is to simply create slightly different versions of the original training set. Bagging produces \( k \) slightly different models (typical decision trees) by training one model per bootstrap sample. The outputs of these models are then combined into a final prediction by averaging the individual \( k \) predictions.

In 2001 Leo Breiman ([8]) proposed an improvement to bagging with decision trees: consider only a randomly selected subset of the predictors for each split of the decision tree. The resulting collection of trees is called a random forest.
3. Boosting

The theory behind boosting is easy to understand via a binary classification problem. Therefore for the time being assume that the goal is to classify the members of some population into two categories. For instance, the goal might be to determine whether a medical patient has a certain disease or not. Typically these two categories are given numerical representations such that the positive outcome (the patient has the disease) equals 1 and the negative outcome (the patient does not have the disease) equals to $-1$. Using this notation, each example can be represented with a pair $(y, x)$, where $y \in \{-1, 1\}$ and $x \in \mathbb{R}^p$.

The boosting algorithm starts with a constant function, e.g. the mean or median of the response values. After this, the algorithm proceeds iteratively. During every iteration it trains a weak learner (defined as a rule that can classify examples slightly better than random guessing) on a training set that weights more heavily those examples that the previous weak learners found difficult to classify correctly. Iterating in this manner produces a set of weak learners that can be viewed as a committee of classifiers working together to correctly classify each training example. Within the committee each weak learner has a vote on the final prediction. These votes are typically weighted such that weak learners that perform well with respect to the training set have more relative influence on the final prediction. The weighted predictions are then added together. The sign of this sum forms the final prediction (resulting into a prediction of either $+1$ or $-1$) of the committee.

Freund ([9]) developed a boosting algorithm for binary classification. This algorithm, named AdaBoost, was the first adaptive boosting algorithm in the sense that misclassified examples are weighted more heavily than the correctly classified examples for the subsequent iterations.

3.1 AdaBoost

In binary classification the performance of a classifier can be measured by the fraction of incorrect predictions on the training set $\{(x_i, y_i)\}_{i=1}^n$. This measure, called the training error, is defined for any function $f$ as follows.

\[
\text{training error} (f) = \frac{1}{n} \sum_{i=1}^{n} I(y_i \neq f(x_i))
\]
The goal is to construct a set of weak learners that can be combined into a master function with very low training error. The AdaBoost algorithm takes the following iterative approach. The algorithm starts with a uniform distribution over the training set giving each observation equal weight. After this, each successive iteration fits a weak learner on a training set that weights more heavily those examples that the previous weak learners found difficult to classify correctly. The algorithm runs for a total of $M$ iterations producing a sequence of $M$ weak learners. These $M$ weak learners are combined into a master function through a weighted majority vote. More formally, if $F(x): \mathbb{R} \rightarrow \{-1, 1\}$ is the master function, then

$$F(x) = \text{sign} \left( \sum_{m=1}^{M} \gamma_m f_m(x) \right)$$

where $\gamma_m$ represents the weight associated with the $m$th weak learner $f_m$. These weights correlate negatively with the training errors. Therefore those weak learners that perform well on the training set have higher weights and thus relatively more influence on the final prediction. For more details, see the full specification of AdaBoost in Algorithm 1.

While AdaBoost was designed for binary classification problems, boosting theory has been extended to regression problems as well. Since in regression each example is represented with a pair $(y, x)$, where $y \in \mathbb{R}$ and $x \in \mathbb{R}^p$, the master function should output a real-valued prediction based on any given vector of features. The training error of the master function is typically measured by summing the differences between the predictions and the respective training examples via the mean squared error:

$$\text{training error} \ (F) = \frac{1}{n} \sum_{i=1}^{n} (y_i - F(x_i))^2$$

**Algorithm 1 AdaBoost**

Start with $M > 0,$ $\{(y_i, x_i)\}_{i=1}^{n}$

1. Initialize the training set weights $w_i^1 = \frac{1}{n}$ for $i = 1, 2, ..., n$.

2. For: $m = 1, 2, ..., M$ do

   1. Fit a weak learner $f_m$ to the training set with weights $\{w_i^m\}_{i=1}^{n}$
   2. Compute $\varepsilon_m = \frac{\sum_{i=1}^{n} w_i^m \mathbbm{1}(y_i \neq f_m(x_i))}{\sum_{i=1}^{n} w_i^m}$
   3. Compute $\gamma_m = \log(\frac{1-\varepsilon_m}{\varepsilon_m})$
   4. Set $w_i^{m+1} = w_i^m \exp[\gamma_m \mathbbm{1}(y_i \neq f(x_i))]$ for $i = 1, 2, ..., n$
   5. Normalize the weights to add to one: $w_i^{m+1} = \frac{w_i^{m+1}}{\sum_{i=1}^{n} w_i^{m+1}}$

3. End for

4. Return $F_M(x) = \text{sign}(\sum_{m=1}^{M} \gamma_m f_m(x))$
3.2 Gradient Boosting Machine

Friedman ([10]) showed that the Gradient Boosting Machine (or simply Gradient Boosting) can be applied to both classification and regression problems with good empirical performance. Overall, Gradient Boosting allows the development of a wide range of different boosting algorithms by simply specifying the following two components.

1. A function class $H$ of weak learners. For instance, the class of decision trees with only a few splits is a popular choice.
2. A non-negative, differentiable loss function $L(y, F(x))$. The loss function is the criterion that the boosting algorithm uses for determining how far the master function is from making a correct prediction on any given training example. In regression it is natural to use a loss function that correlates positively with the size of the absolute residual. For instance, the square error loss $(y_i - F(x_i))^2$ and the absolute difference $|y_i - F(x_i)|$ are popular choices for regression. In classification the role of the residual is replaced by the margin that is defined for example $(y_i, x_i)$ as

$$
\text{margin}[(y_i, x_i)] = y_i \sum_{m=1}^{M} \gamma_m f_m(x_i)
$$

Since a large negative margin corresponds to a very poor overall fit, the loss should be a decreasing function in the margin.

Given these two components the Gradient Boosting algorithm sequentially builds up a master function that minimizes the average loss:

$$
L = \frac{1}{n} \sum_{i=1}^{n} L(y_i, F(x_i))
$$

**Algorithm 2 Gradient Boosting**

Start with: $M > 0, \{(y_i, x_i)\}_{i=1}^{n}$

1. Initialize the function with a constant value: $F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i - \gamma)$
2. For $m = 1, 2, ..., M$ do
   1. Compute the pseudo-residuals: $\eta_{im} = -\left[ \frac{\partial L(y_i - F(x_i))}{\partial F(x_i)} \right]_{F(x_i) = F_{m-1}(x)}$ for $i = 1, 2, ..., n$
   2. Fit a weak learner $f_m(x)$ to the pseudo-residuals $\{(\eta_{im}, x_i)\}_{i=1}^{n}$
   3. Compute the step size: $\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i - F_{m-1}(x_i) - \gamma f_m(x_i))$
1. Initialize: $F_0(x) = \phi(x)$

2. For $m = 1$ to $M$
   
   3. \textbf{End for}
   
   4. Return $F_M(x)$

Gradient Boosting minimizes the average loss by starting with a constant function $F_0$, and then updating it in a greedy fashion:

$$
F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i - \gamma)
$$

$$
F_m(x) = F_{m-1}(x) + \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{n} L(y_i - (F_{m-1}(x_i) + f(x_i))
$$

Finding this weak learner is a very difficult optimization problem. Fortunately, this search can be simplified by incorporating a steepest descent step in the optimization procedure. In order to understand how this is done assume for the time being that making predictions on the training examples is the only purpose of the master function. Then the model can be updated in the following fashion. By setting $F = \{F(x_i)\}_{i=1}^{n}$, the gradient of $L(F)$ evaluated at $F = F_{m-1}$ is denoted by $\nabla_m$.

The elements of $\nabla_m$ are

$$
\frac{\partial L(y_i - F(x_i))}{\partial F(x_i)} \bigg|_{F(x_i) = F_{m-1}(x_i)} \quad \text{for } i = 1, 2, ..., n
$$

Using this gradient, the predicted values can be moved closer to the training examples with

$$
F = F_{m-1} - \gamma_m \nabla_m
$$

where

$$
\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i - (F_{m-1}(x_i) - \gamma g_m))
$$

The negative gradient $-\nabla_m$ can be seen as the local direction in $\mathbb{R}^n$ for which the loss function is decreasing most rapidly at $F = F_{m-1}$. Therefore updating the fitted values in this way takes a step of size $\gamma_m$ in the direction with the steepest descent.

Notice that this procedure does not fit weak learners at any point. Instead, it starts with the predictions given by the constant function and then iteratively adds small quantities to these predictions until they match with the training examples. Therefore the transition from the initial predictions to the final predictions can be viewed as a series of gradient values. These transitions can be stored by choosing the weak learner $f_m \in \mathcal{H}$ that most closely approximates $\nabla_m$ during iteration $m$. In practice, this means fitting $f_m$ to the pseudo-residuals $\{(g_{im}, x_i)\}_{i=1}^{n}$. This produces a sequence of weak learners that ultimately form the master function. In classification this is done through a weighted majority vote:
In regression the master function must remain real-valued. Therefore the final master function is an additive model of the weak learners:

\[ F(x) = \text{sign} \left( \sum_{m=1}^{M} \gamma_m f_m (x) \right) \]

Notice that Gradient Boosting is different from Adaboost in a sense that it does not alter the weights of the training examples during the boosting procedure. Instead, it distributes the weights evenly across the training set and modifies the training examples themselves.

### 3.3 Boosting in Regression

Duffy and Helmbold ([7]) propose a boosting variant, named ExpLev, that modifies both the examples and the weights such that the algorithm can make progress in a gradient boosting-like manner:

#### Algorithm 3 ExpLev

**Start with:** \( \eta > 0 \), a training set \( \{(y_i, x_i)\}_{i=1}^{n} \) and an initial function estimate \( F_1(x) \)

1. Set \( s = \frac{1}{\eta} \) and \( t = 1 \)
2. Compute the residuals \( \eta_i = (y_i - F_t(x_i)) \) for \( i = 1, 2, \ldots, n \)
3. **while** \( \max(\{\eta_i\}_{i=1}^{n}) > \eta \) and \( |F_{t+1}(x) - F_t(x)| > \varepsilon \) **do**
   1. Increment \( t = t + 1 \)
   2. Find gradients: \( \nabla_i L_{\exp} = -s \exp (s \eta_i) + s \exp (-s \eta_i) \) for \( i = 1, 2, \ldots, n \)
   3. Compute the weights \( w_i = \frac{|\nabla_i L_{\exp}|}{\sum_{i=1}^{n} |\nabla_i L_{\exp}|} \) for \( i = 1, 2, \ldots, n \)
   4. Compute the pseudo-residuals: \( \tilde{y}_i = \text{sign}(\eta_i) \) for \( i = 1, 2, \ldots, n \)
   5. Fit a weak learner \( f_t(x) \) to the pseudo-residuals: \( \{(\tilde{y}_i, x_i)\}_{i=1}^{n} \) with weights \( \{w_i\}_{i=1}^{n} \)
   6. Compute the step size: \( \gamma_t = \arg \min_{\gamma} \sum_{i=1}^{n} L_{\exp}(y_i - (F_{t-1}(x_i) - \gamma f_t(x_i))) \)
   7. Update the master function: \( F_t(x) = F_{t-1}(x) + \gamma_t f_t(x) \)
   8. Compute the residuals: \( \tilde{y}_i = y_i - F_t(x_i) \)
Similarly to *Gradient Boosting*, the *ExpLev* algorithm repeatedly fits a weak learner to the pseudo-residuals. In this case, however, each pseudo-residual is equal to the sign of the respective negative gradient value. By weighting these pseudo-residuals proportional to the negative gradient, the weak learner is encouraged to return a prediction that correlates positively with the negative gradient both in sign and magnitude. Therefore fitting a weak learner in this way is not vastly different from fitting a weak learner directly to the negative gradient.

4. **Robust Boosting**

Many classical modeling methods rely heavily on assumptions that are rarely met in practice. For instance, it is often assumed that the errors in the training set are normally distributed. As a result, classical methods can end up performing very poorly on noisy real-world datasets. For instance, it has been shown that the performance of many boosting algorithms degrades dramatically in the presence of outliers ([5]). To illustrate, consider the *AdaBoost* algorithm. Since *AdaBoost* weights the misclassified training examples more heavily for the subsequent iterations, it can end up assigning an unboundedly large weight to an outlier. Such a heavily weighted outlier can pull the master function far off from the true underlying trend of the data. As a result, the master function ends up having poor predictive performance.

Robust methods provide an alternative to the classical methods by constructing models that are unaffected by outliers. A common approach is to identify and remove the outliers from the training set before the fitting process. In some cases, however, it can be beneficial to allow the outlier detection mechanism to access and make use of the fitting process. This section explains how the boosting procedure can use the intermediate master functions for outlier detection. These outliers are then excluded from the fitting process of the next weak learner via a carefully chosen loss function. It turns out that the convexity of the loss function is a very important determinant of robustness. The next subsection briefly discusses the relative robustness of convex and non-convex loss functions.

4.1 **Convexity**

Most boosting algorithms today can be viewed as methods that minimize some sort of a convex loss function. Convex loss functions are particularly popular because they can be minimized very efficiently using the gradient descent methods. Recently, however, Long and Servedio ([11]) showed that any boosting algorithm that is based on a convex loss function can be defeated by random noise in the training set. Therefore it is not possible to use a convex loss function to construct a highly robust boosting algorithm.

Freund ([12]) developed a classification boosting algorithm called *Boost-By-Majority* (BBM). This algorithm uses a loss function that resembles the sigmoid function rotated by the Y-axis (Figure 1a) and a weight function that is based on the derivative of the loss function (Figure 1b). Notice that the weights as a function of the margin increase only until the inflection point of the loss function. Beyond
this point the weights begin to decrease indefinitely. Therefore at some point the margin becomes negative enough for the boosting algorithm to suspect that the training example might be inherently incorrect and hence should not be given any weight in the fitting process of the next weak learner.

{Figure 1 here}

This seems like the kind of an approach that any robust boosting algorithm should take. Unfortunately, a local minimum of a non-convex loss function is not necessarily global. Therefore the gradient descent methods are not guaranteed to converge to a global minimum. The next section introduces the loss function used by the TrimBoost algorithm. Even though this loss function is non-convex, it can be efficiently minimized with the gradient descent methods.

4.2 Trimmed Mean Loss

The loss function should be chosen carefully such that it leads to an easy construction and analysis of the resulting boosting algorithm. In other words, it is not enough for the loss function to be non-negative, differentiable, and decreasing towards zero. In addition, it should be easy to work with. The trimmed mean loss function $L_{trim}$ (Figure 2a) is a simple piecewise function with appropriate properties for a more robust regression boosting algorithm. It is defined as

$$L_{trim}(y_i, F(x_i)) = \begin{cases} (y_i - F(x_i))^2, & \text{if } |y_i - F(x_i)| \leq c, \\ \frac{c^2}{2}, & \text{otherwise.} \end{cases}$$

where the critical value $c$ is non-negative. Notice that $L_{trim}$ is (piecewise) differentiable with respect to $F(x)$. This derivative is used as a basis for the weight function $W_{trim}$ (Figure 2b).

$$W_{trim}(y_i, F(x_i)) = \begin{cases} 2 |y_i - F(x_i)|, & \text{if } |y_i - F(x_i)| \leq c, \\ 0, & \text{otherwise.} \end{cases}$$

{Figure 2 here}
The \emph{TrimBoost} algorithm assigns weight by first evaluating the weight function \(W_{tr,t}\) for each training example and then normalizing these values so that they all add up to one. Notice that the weight function increases linearly until the absolute residual \(|y_i - F(x_i)| = c\). After this point the absolute residual becomes large enough for the boosting procedure to suspect that the example is inherently different from the rest of the training examples. As a result, the example is classified as an outlier and not given any weight in the fitting process of the next weak learner. Since the choice of \(c\) plays a central role in this procedure, it should be chosen very carefully. The next section describes several criteria for updating this critical value.

4.3 Outlier Detection

During each iteration the value of \(c\) needs to be updated such that as few outliers and as many non-outliers as possible are included in the fitting process of the next weak learner. While it is not known which examples are outliers and which are not, a good estimation can be achieved by updating the value of \(c\) based on the assumption that the current master function is a good approximation of the true underlying trend of the data. Under this assumption, any relatively large absolute residual can be viewed as a large deviation from the true underlying trend. Therefore a well-chosen robust measure of scale of the absolute residuals provides a natural solution to separating the outliers from the non-outliers.

Many different criteria for updating \(c\) have been proposed in the past. For instance, Jerome Friedman developed a regression boosting algorithm based on the Huber loss function (Figure 3) ([10]). This gradient descent boosting algorithm sets \(c\) equal to the \(\alpha\)-quantile of the absolute residuals. While this algorithm is fairly robust compared to earlier regression boosting algorithms, it suffers from a practical inconvenience: it requires the user to pre-specify \(\alpha\). An alternative approach is to first assume that the outliers follow a different distribution than the non-outliers, and then use Chebyshev’s inequality combined with a robust measure of scale to approximate the likelihood of an example following the distribution of the non-outliers. If this likelihood is too low, the example is likely to be an outlier and hence can be discarded from the fitting process of the next weak learner. While this approach is theoretically appealing, it is too conservative to work in practice because Chebyshev’s inequality is known to typically provide very loose bounds.

The update rule used by the \emph{TrimBoost} algorithm is based on John Tukey’s famous construction of the box plot. According to Tukey any example that is \(1.5 \times IQR\), where \(IQR\) is the interquartile range, away from the closest quartile is nominated as an outlier ([13]). Therefore in the beginning of iteration \(m\) the value of \(c\) is updated by setting

\[
c = 1.5 \times IQR\left(||y_i - F_{m-1}(x_i)||_{P=1}\right)
\]

This update rule is particularly nice because it does not require the user to pre-specify any parameters. It can also tolerate a high level of noise. In order to see this, notice that if 25\% of the examples are allowed to be arbitrarily large so that they produce very large absolute residuals, then these residuals
are likely to fall in the upper quartile of the absolute residuals. Since the value of $c$ is completely determined by the middle 50% of the absolute residuals, these large training examples do not affect the size of $c$. If any more examples are allowed to be arbitrarily large, the IQR of the absolute residuals will begin to grow. As a result, the algorithm may end up assigning unboundedly large weight to some of the noisy examples. This leads to the following observation.

**Observation 1:** The TrimBoost algorithm can be expected to handle noise up to 25%.

Chapter 6 shows that the TrimBoost algorithm is in fact able to tolerate such a high level of noise in practice. Improved noise tolerance, however, is only one of the many properties of the TrimBoost algorithm. The next chapter gives more details by stating several theorems on the behavior of the TrimBoost algorithm.

5. Methodology

At iteration $m$, the TrimBoost algorithm fits a weak learner $f_m$ to a modified version of the training set $\{(\hat{y}_i, x_i)\}_{i=1}^n$ where $\hat{y}_i = \text{sign}(y_i - F_{m-1}(x_i))$. Each example in this modified training set is weighted proportional to the first derivative of the trimmed mean loss function. Using an approach similar to the ExpLev algorithm, the TrimBoost algorithm minimizes the average trimmed mean loss:

$$L_{tr} = \frac{1}{n} \sum_{i=1}^n L_{tr_i}(y_i, F_m(x_i))$$

$$= \frac{1}{n} \sum_{i=1}^n \left( \sum_{j \neq i}^n \eta_j^2 + \sum_{|j| < c} \eta_j^2 \right). \quad (1)$$

where the residual is defined by the current master function $\eta_i = y_i - F_m(x_i)$. The optimal value for the coefficient $\gamma_m$ can be found directly by minimizing the average loss with respect to $\gamma_m$. At first this optimization may seem difficult because the partition into two separate sums in equation (1) depends on $\gamma_m$. Fortunately, this dependency can be eliminated by deriving an upper bound that determines the partition based on the previous residuals $\{(y_i - F_{m-1}(x_i))\}_{i=1}^n$. The likelihood that this upper bound is an equality increases as the boosting algorithm makes progress and becomes more certain about which training examples are outliers. Using this upper bound, any iterative search for the optimal coefficient $\gamma_m$ can be replaced with the following closed-form solution.
Boosting algorithms tend to measure the performance of the weak learners slightly differently. This measure, generally known as the “edge,” is often defined such that it simplifies the notation of the theoretical analysis of the algorithm. In this paper, the edge is defined as follows:

\[
\epsilon_m = \frac{\sum_{|\mathcal{S}|} r_i f_m(x_i)}{\sqrt{\sum_{|\mathcal{S}|} r_i f_m(x_i)^2}}
\]

This measure of edge correlates positively with the weak learner’s performance on the training set. Therefore as long as the weak learner performs reasonably well on the training set, the edge can be assumed to be positive. This is important because the average loss decreases at least by a factor of \(1 - \epsilon_m^2\) per iteration. In other words, during every iteration the average loss decreases by a multiplicative factor that depends on the performance of the weak learner. More importantly, if it can be assumed that the edges across a finite number of iterations are bounded below by \(\epsilon_{\min} > 0\) and that each response value \(\{y_i\}_{i=1}^n\) is bounded between \(-B\) and \(B\), then for any \(\rho > 0\) after \(T\) iterations, the master function \(F_T(x)\) has training error \(\hat{\mathcal{J}}_T(F_T) \leq \rho\) over the training set \(\mathcal{S}\). The training error is measured by the mean squared error over those training examples whose associated absolute residuals are less than or equal to \(\epsilon\).

**Definition 1** Given a training set \(\mathcal{S}\) of size \(n\) and a function \(f: \mathbb{R}^p \rightarrow \mathbb{R}\), the training error of \(f\) on \(\mathcal{S}\) is

\[
\hat{\mathcal{J}}_S (f) = \frac{\sum_{r \in \mathcal{S}} r^2}{\sum_{r \in \mathcal{S}} r}.
\]

This is a natural definition of the training error for the TrimBoost algorithm because training examples whose absolute residuals are larger than \(\epsilon\) are not part of the fitting process.
5.1 Convergence

Convergence results imply that if the master function is in some relatively simple hypothesis space and if the master function has a small training error over a large enough set of IID training examples, then the master function is expected to have a small squared error over the entire domain of $\mathbf{x}$ with very high probability. This expected error, called the generalization error, is defined as follows.

**Definition 2** Let be a probability distribution on $\mathbb{R}^p \times \mathbb{R}$. The generalization error of a function $f : \mathbb{R}^p \rightarrow \mathbb{R}$ with respect to $P$ is

$$e_P(f) = E \left[ (y - f(x))^2 \right]$$

In the presence of outliers it cannot be assumed that all the training examples are IID with respect to some distribution $P$. It is reasonable, however, to assume that some subset of the training set is IID with respect to $P$. If the training error over such a subset can be minimized, the master function should be expected to perform well across the entire domain of $\mathbf{x}$ as long as the subset is large enough (for large enough see [7]). It turns out that minimizing the training error $\hat{e}_P(F)$ also minimizes the training error over those training examples that are IID with respect to $P$. To be more precise, assume that the fitting process is guaranteed to include at least $n_2$ training examples that are IID with respect to $P$. If $S_{i.i.d}$ is the (unobservable) set of training examples that are IID with respect to $P$, then for any $\rho > 0$

$$\rho \geq \hat{e}_P(F)$$

implies

$$\frac{\rho}{n_2} \geq \hat{e}_{S_{i.i.d}}(F)$$

(2)

Since each term on the left hand side of (2) is fixed, the training error $\hat{e}_{S_{i.i.d}}(F)$ can be made arbitrarily small.

5.2 Specification

Algorithm 4 gives the full specification of the TrimBoost algorithm. Notice that the master function is initialized to the median of the response values. If the initialization was not based on a robust measure of location, any outliers in the training set could end up pulling the initial master function far off from the true underlying trend of the data. In order to avoid such an un-robust start, a robust measure, such as the median, should be used for initializing the master function.
Algorithm 4 TrimBoost

Start with $M > 1$, a training set $\{(y_i, x_i)\}_{i=1}^n$

1. Initialize the master function: $F_0(x_i) = \text{median}(\{y_i\}_{i=1}^n)$
2. For $m = 1, 2, ..., M$ do
   1. Compute the residuals: $\eta_i = y_i - F_{m-1}(x_i)$ for $i=1,2,...,n$
   2. Set $c = 1.5 \times IQR(\{\eta_i\}_{i=1}^n)$
   3. Find the gradients: $\nabla_i L = \begin{cases} -2\eta_i, & \text{if } |\eta_i| \leq c, \\ 0, & \text{otherwise} \end{cases}$ for $i=1,2,...,n$
   4. Compute the weights: $w_i = \frac{|\eta_i|}{\sum_{k=1}^n |\eta_k|}$ for $i=1,2,...,n$
   5. Compute the pseudo-residuals: $\tilde{y}_i = \text{sign}(\eta_i)$ for $i=1,2,...,n$
   6. Fit a weak learner $f_m(x)$ to the pseudo-residuals $\{(\tilde{y}_i, x_i)\}_{i=1}^n$ with weights $\{w_i\}_{i=1}^n$
   7. Set $\gamma_m = \frac{\sum_{r \in \mathcal{S}} r_{r \in \mathcal{S}}}{\sum_{r \in \mathcal{S}} \sum_{r \in \mathcal{S}}}$
   8. Update the master function: $F_m(x) = F_{m-1}(x) + \gamma_m f_m(x)$
3. \textbf{end for}
4. \textbf{return} $F_M(x)$

6. Empirical Analysis

This chapter evaluates the behavior and performance of the TrimBoost algorithm on both real-world and synthetic datasets. The results show that TrimBoost is more robust than several of the following widely used regression algorithms.

Gradient Descent Boosting: The evaluation includes two different version of the gradient boosting machine: The first one uses the square error (Gaussian) loss function and serves as a reality check. The second one gains more robustness by using the Huber loss function ([10]). Figure 4 summarizes the connection between these two loss functions and the loss function used by TrimBoost. Notice that the main differences occur when the absolute residual is larger than the critical value $c$. The gradient boosting with the Huber loss function uses the same update rule for the critical value $c$ as the TrimBoost algorithm. Throughout this chapter the weak learner is a stump, i.e. a decision tree with depth one. The boosting algorithms used a total of 10,000 decision stumps with terminal node values calculated by averaging the training examples falling into each terminal node.
Random Forest: By sampling with replacement from the training set, the random forest algorithm first constructs some fixed number, say $M$, samples. Then a random-forest tree is grown to each of these $M$ samples by recursively repeating the following steps for each terminal node of the tree until the node contains at most some fixed number of training examples.

1. Select a subset of $m$ features from the set of $p$ features $\{x_i\}_{i=1}^p$.
2. Among these $m$ features choose the feature that produces the best split according to some information criterion.
3. Based on the chosen feature, split the node into two child nodes. A prediction for some unseen point $\mathbf{x}$ is made by averaging the individual predictions of the $M$ trees.

Overall, random forest is an efficient and accurate learning algorithm that offers an excellent benchmark outside the boosting paradigm. Throughout this chapter the random forest algorithm used a total of 100 random trees of any complexity.

Robust Linear Regression: Often the coefficients of a linear regression model are estimated by minimizing the square error loss over the training set. Under this criterion, however, outliers can have a considerable influence on the final model. Therefore more robust criteria should be used. Throughout this evaluation, the robust linear regression model is fit by minimizing the Huber loss function (see Figure 4).

Neural Network: A neural network with a single hidden layer consists of three levels of nodes: the input level, the hidden level and the output level. At any given level, the nodes are fully connected to the neighboring levels. The nodes at the input level take in the feature values and produce an output. These output values then become the input of the nodes at the hidden level. Similarly, the hidden nodes produce output that is further passed on to the output nodes. The output nodes produce the final output of the neural network. Figure 5 illustrates a neural network that outputs $\hat{y}$ based on three explanatory variables $x_1$, $x_2$, and $x_3$. Overall, neural networks are widely used as a benchmark for performance comparisons. Throughout this chapter the neural networks consisted of a single hidden layer with 10 hidden nodes.
6.1 Real-World Data

This section evaluates the performance of the TrimBoost algorithm on three datasets that are publicly available on the University of California, Irvine (UCI) Machine Learning Repository ([14]).

6.1.1 Wine Quality Data

The first two datasets contain chemical and quality measurements on wine samples with 1,599 red and 4,898 white wine samples respectively. Each wine sample is represented with a quality measure and 11 continuous features that are based on physicochemical tests. Since the quality measure is a score between 0 (very bad) and 10 (excellent), each example is a pair \((y_i, x_i)\), where \(y_i \in \{0, 1, \ldots, 10\}\) and \(x_i \in \mathbb{R}^{11}\). The overall goal is to model wine quality based on physicochemical tests.

The evaluation procedure is repeated separately for the two datasets as follows: First, a 10-fold cross-validation is repeated 10 times for each of the algorithms. Recall that a 10-fold cross-validation partitions the original dataset into 10 subsamples. One of these subsamples is used for testing, and the remaining nine are used for training. Since each of the subsamples is used for testing exactly once, repeating a 10-fold cross-validation for 10 times produces a total of a 100 testing errors. These testing errors are used as a baseline for assessing the impact of noise on each of the algorithms. This is done by repeating a 10-fold cross-validation for 10 times for each of the algorithms under varying levels of contamination in the training set. Any degradation in performance relative to the baseline reflects reduced robustness.

The training set is contaminated by first selecting some fraction of the training examples uniformly at random and then changing their response values as much as possible. More specifically, the contamination is done using the following function.

\[
\text{Contaminate}(y) = \begin{cases} 
10, & \text{if } y < 5, \\
0, & \text{if } y > 5, \\
\text{Unif}([0,10]), & \text{if } y = 5 
\end{cases}
\]

**Red Wine Results**: Figure 6 shows the box plots of a 100 testing errors under varying levels of contamination in the red wine data set. Notice how the performances of the gradient boosting with the squared error (Gaussian) loss, the neural networks, and the random forests degrade linearly in the level of contamination. While the gradient boosting with the Huber loss function is slightly more robust, it does not compare to robust regression and TrimBoost that are not affected by the contamination.

**White Wine Results**: Figure 7 depicts the box plots of a 100 testing errors under varying levels of contamination in the white wine dataset. Since this dataset is nearly three times as large as the red wine dataset, it is no surprise that the testing errors are uniformly smaller. Apart from this, the results are very similar on both datasets.

On both datasets the neural network algorithm produced a few very large testing errors making it very difficult to visualize the actual differences between the algorithms. For this reason, these large errors were excluded from the box plots.
6.1.2 Concrete Data

This section evaluates the performance of the TrimBoost algorithm on measurements of concrete. This dataset is also found in the UCI repository. The response represents concrete compressive strength that is highly nonlinear function of eight continuous feature values including age and amounts of ingredients used in the concrete. This dataset contains a total of 1,030 examples. The goal is to model the concrete compressive strength based on the feature values.

Similarly to the previous section, ten 10-fold cross-validations are run on the concrete dataset under varying levels of contamination in the training set. The training set is contaminated by first selecting some fraction of the training examples uniformly at random and then adding random noise \( e \sim N(0,1000) \) to these examples. The testing set is not contaminated.

**Concrete Results:** Figure 8 shows the box plots of a 100 testing errors under varying levels of contamination in the concrete dataset. Only robust regression and TrimBoost are able to tolerate the contamination. Since the response is nonlinear of the feature values, it is no surprise that the TrimBoost algorithm is able to outperform robust regression by a wide margin. Notice that TrimBoost outperforms the random forest algorithm under any level of contamination. Here the testing errors have been log-transformed due to large differences in the testing errors.

6.2 Synthetic Data

This section compares the algorithms on synthetic datasets. The first simulation gives robust linear regression an advantage by generating data from an underlying linear function. The second simulation adds an extra layer of complexity by changing the correlation structure of the true underlying function at every iteration. Generating data in this manner imitates the idea of a random function generator proposed by Friedman ([10]).

6.2.1 Linear Data
The training set is generated from

\[ y_i = x_{i1} + x_{i2} + \ldots + x_{ip} + \epsilon_i \]

(3)

where \( \epsilon_i \sim N(0,1) \) and \( x_{ij} \sim N(0,10) \) for \( j = 1, ..., p \). The training set is contaminated by replacing a given fraction of the error terms with \( \epsilon_i' \sim N(0,100) \). Testing is performed over a 1,000 examples generated directly from (3) without contamination.

**First Linear Data Results:** Figure 9 shows the box plots of a 1,000 testing errors under varying levels of contamination. This simulation used five features \( (p = 5) \) and a training set of size 100 \( (n = 100) \). The performances of the gradient boosting, the neural networks, and the random forests degrade linearly in the level of contamination. Similarly to the previous section, robust regression and TrimBoost are not affected by the contamination.

**Second Linear Data Results:** Figure 10 depicts the box plots of a 1,000 testing errors of the second simulation that used ten features \( (p = 10) \) and a training set of size 1,000 \( (n = 1000) \). The results are very similar to the ones shown in Figure 6. In fact, only the gradient boosting with the Huber loss function shows increased robustness when compared to the results in Figure 9.

6.2.2 Random Function Generation

Since the choice of the underlying true function that generates the data is one of the most important factors affecting the relative performance of the algorithms, Friedman suggested evaluating different algorithms on a set of randomly generated functions ([10]). Each of these functions takes an additive form

\[ F^*(x) = \sum_{i=1}^{20} y_i^* f_i^*(x_i) \]

where the true coefficients and true weak learners are marked with an asterisk. The coefficients \( \{ y_i^*\}_{i=1}^{20} \) are generated from a continuous uniform distribution \( y_i^* \sim \mathcal{U}(-1,1) \). The secondary features \( z_i \) are a random subset of size \( n_t \) of the original features \( x_i \). These sizes are determined by \( n_t = \lfloor 1.5 + r \rfloor \), where \( r \sim \mathcal{E}(\lambda = 3) \). Therefore there is a great chance that at least one of the functions \( f_i^*(z_i) \) involves a higher order interaction in the features. Each \( f_i^*(z_i) \) is represented by the \( n_t \)-dimensional Gaussian function

\[ f_i^*(z_i) = \exp \left( -\frac{1}{2} (z_i - \mu_i)^T \mathcal{V}_i (z_i - \mu_i) \right). \]
where the means \( \{\mu_i\}_{i=1}^{10} \) generated from the standard multivariate normal distribution. The covariance matrix \( V \) is randomly generated using the Gramian matrix ([15]). Throughout this simulation ten features are used. Their joint distribution is chosen to be the standard multivariate normal \( x \sim N(\mathbf{0}, \Gamma) \). A total of 5,000 examples are used for training and 2,500 examples are used for testing. The training set was contaminated in the same way as in the previous section.

**Random Function Results:** Figure 11 depicts the box plots of the testing errors over a hundred different randomly generated functions. Notice how the gradient boosting with the squared error (Gaussian) loss, the neural networks, and the random forests tolerate contamination very poorly. In contrast to the earlier plots, the gradient boosting with the Huber loss and the \( \text{TrimBoost} \) algorithm perform better than robust regression under any level of contamination. In order to make these differences more clear, Figure 12 shows only the box plots for the gradient boosting with the Huber loss, \( \text{TrimBoost} \), and robust regression. Based on this plot, the gradient boosting with the Huber loss has a slight edge over the \( \text{TrimBoost} \) algorithm. On the other hand, the performance of the gradient boosting with the Huber loss begins to degrade slightly as the contamination increases beyond 10%. Meanwhile, robust regression and \( \text{TrimBoost} \) are not affected by the contamination.

7. **Concluding Remarks**

This paper introduces a novel regression boosting algorithm, named \( \text{TrimBoost} \), that incorporates outlier detection and exclusion via a non-convex loss function. The detection is based on the \( 1.5 \times IQR \) heuristic proposed by John Tukey in his construction of the box plot ([13]). While this heuristic can work well in many different circumstances, it is important to keep in mind that outlier detection is always a subjective exercise. For this reason, it is not difficult to construct a dataset on which the \( \text{TrimBoost} \) algorithm might would perform poorly with a fixed rule. In addition, many improvements can be incorporated in the algorithm. For instance, the choice of initial constant learner could be optimized by cross-validation.

Nevertheless, based on empirical evidence, \( \text{TrimBoost} \) seems to work well in practice: it is able to maintain the high performance typical to boosting as long as the training set contains no more than 25% random noise. With such a high breakdown point, the \( \text{TrimBoost} \) algorithm offers a highly robust meta-algorithm that can be applied to a wide range of real world datasets.

**References**


Figure 1. The loss and weight functions of the Boost-By-Majority algorithm.
(a) Loss Functions

(b) Weight Function

Figure 2. The loss and weight functions with $c = 5$
Figure 3. The Huber loss function with $c = 5$. 
Figure 4. Gaussian, Trimmed, and Huber loss with $c = 3$. 

![Graph showing Gaussian, Trimmed, and Huber loss with $c = 3$.](image)
Figure 5. An example of a one-hidden-layer neural network in three input variables and one output variable.
Figure 6. Performances on the red wine data set
Figure 7. Performances on the white wine data set
Figure 8. Performances on the concrete dataset
Figure 9. Performances on the linear data with $p = 5$ and $n = 100$
Figure 10. Performances on the linear data with $p = 10$ and $n = 1000$
Figure 11. Performances under the random function generation
Figure 12. A detailed view of Figure 10