Contents

1 Ensemble **2**

2 Bagging **2**

2.1 Regression . . . 2

2.2 Classification . . . 2

2.3 Out-Of-Bag Estimation . . . 3

3 Random Forest **4**

3.1 Extra-Tree . . . 4

4 Boosting **5**

4.1 Adaptive Boosting . . . 5

4.1.1 Two-Class Problem . . . 5

4.1.2 Multi-Class Problem . . . 7

4.1.3 Visual Intuition . . . 8

4.2 Scenario For Gradient Boosting . . . 9

4.2.1 Taylor Approximation . . . 9

4.2.2 Gradient Descent In Functional Space . . . 10

4.3 Gradient Boosting Tree . . . 10

4.3.1 Square Loss . . . 11

4.3.2 Absolute Loss . . . 12

4.3.3 Huber Loss . . . 12

4.3.4 Pseudo Code . . . 12

4.3.5 Hyper-parameters . . . 12

1 Ensemble

An **ensemble method** is an approach that combines many simple "building block" models in order to obtain a single and potentially very powerful model. These simple building block models are sometimes known as **weak learners**, since they may lead to mediocre predictions on their own.

2 Bagging

Bootstrap aggregation, or **bagging**, is a general-purpose procedure for reducing the variance (overfitting the specific split of training data) of a statistical learning method.

2.1 Regression

Recall that given a set of n independent observations $Z_1, ..., Z_n$, each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by $\frac{\sigma^2}{n}$. In other words, **averaging** a set of observations reduces variance. Hence a natural way to reduce the variance and increase the test set accuracy of a statistical learning method is to 1) take many training sets from the population, 2) build a separate prediction model using each training set, 3) and average the resulting predictions. We calculate $\hat{f}^1(x), \hat{f}^2(x), \ldots, \hat{f}^B(x)$ response functions using B separate training sets, and average them in order to obtain a single low-variance statistical learning model, given by

$$\hat{f}_{\text{avg}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(x)$$

To obtain different random subsets of the training set, we can bootstrap, by taking repeated samples from the (single) training set. We generate B different bootstrapped training data sets. We then train our method on the b^{th} bootstrapped training set in order to get $\hat{f}^{*b}(x)$, and finally average all the predictions, to obtain

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

This is called bagging. The **aggregation function** in this case is a measure of central tendency— the mean. To apply bagging to decision trees, we simply construct B decision trees using B bootstrapped training sets, and average the resulting predictions. These trees are grown deep, and are not pruned. Hence each individual tree has high variance (likely overfitting the data) but low bias. Averaging these B trees reduces the variance.

2.2 Classification

Thus far, we have described the bagging procedure in the regression context, to predict a **quantitative** outcome Y. If Y is **qualitative**, we can use a voting classifier. For a given test observation, we can record the class predicted by each of the B trees, and take a majority vote: the overall prediction is the most commonly occurring majority class among the B predictions. This is also known as the the **hard voting** classifier. If all B classifiers are able to estimate class probabilities, then another approach is to predict the class with the highest class probability, averaged over all B individual classifiers. This is called **soft voting**.

2.3 Out-Of-Bag Estimation

There is a very straightforward way to estimate the test error of a bagged model without the need to perform cross-validation or the validation set approach. The key to bagging is that trees are repeatedly fit to bootstrapped subsets of the observations. Suppose that we obtain a bootstrap sample from a set of n observations. We will now derive the probability that a given observation is part of a bootstrap sample. Assuming that each selection from the set of n observations is independent, the probability that the j^{th} observation is not in the bootstrap sample is:

$$p_j(n) = \prod_{i=1}^{n} \pi_j = \left(1 - \frac{1}{n}\right)\left(1 - \frac{1}{n}\right)\cdots\left(1 - \frac{1}{n}\right) = \left(1 - \frac{1}{n}\right)^n$$

Recall that the exponential function satisfies:

$$e^x = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n$$

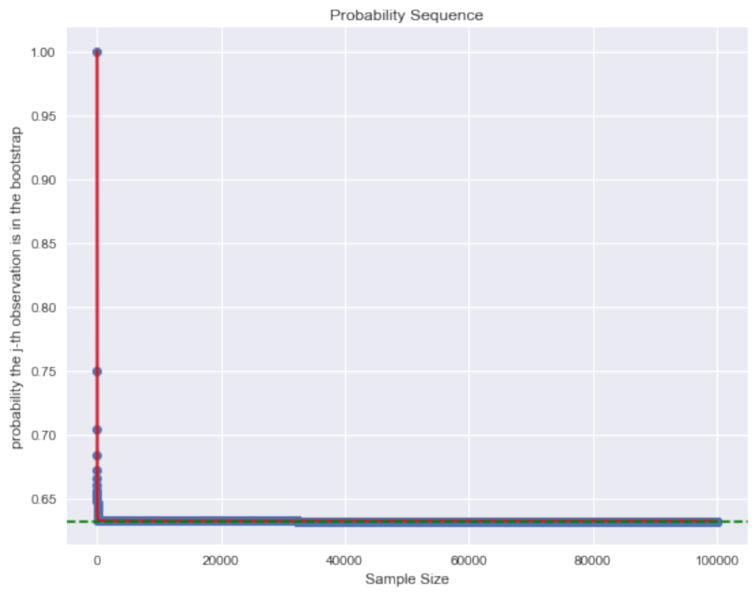
Therefore, letting $x = -1$

$$e^{-1} = \lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^n$$

and the probability that the j^{th} observation is **not** in the bootstrap sample is

$$p_j := \lim_{n\to\infty} p_j(n) = e^{-1} \approx 0.36787944117$$

The probability that the j^{th} observation **is** in the bootstrap sample converges as $n \to \infty$:



This means that each bagged tree makes use of around two-thirds ($1 - 0.36787944117 = 0.6321205588$) of the observations. The remaining one-third of the observations not used to fit a given bagged tree are referred to as the **out-of-bag** (OOB) observations.

- We can predict the response for the i^{th} observation using each of the trees in which that observation was OOB. This will yield around $\frac{B}{3}$ predictions for the i^{th} observation.
- To obtain a single prediction for the i^{th} observation, we can then average these $\frac{B}{3}$ predicted responses (if regression is the goal) or can take a majority vote from $\frac{B}{3}$ votes (if classification is the goal). This leads to a single OOB prediction for the i^{th} observation.

An OOB prediction can be obtained in this way for each of the n observations, from which the overall OOB MSE (for a regression problem) or classification error (for a classification problem) can be computed. The resulting OOB error is a valid estimate of the test error for the bagged model, since the estimated response or classification for each observation is predicted using **only the trees that were not fit using that observation**. With B sufficiently large, OOB error is virtually equivalent to leave-one-out cross-validation error.

3 Random Forest

Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors. Then in the collection of bagged trees, most or all of the trees will use this strong predictor in the top split. Consequently, all of the bagged trees will look quite similar to each other. Hence the predictions from the bagged trees will be highly correlated. Averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities.

Random forests overcome this problem by forcing each split to consider only a subset of the predictors. Therefore, on average $\frac{(pm)}{p}$ of the splits will not even consider the strong predictor, and so other predictors will have more of a chance. The main difference between bagging and random forests is the choice of predictor subset size m. For instance, if a random forest is built using $m = p$, then this amounts simply to bagging.

3.1 Extra-Tree

In a Random Forest, at each node, only a random subset of the features is considered for splitting. It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular Decision Trees do).

A forest of such extremely random trees is called an **Extremely Randomized Trees ensemble** (or Extra-Trees for short). Extra-Trees are much faster to train than regular Random Forests, because finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

4 Boosting

4.1 Adaptive Boosting

4.1.1 Two-Class Problem

1. Initialize the observation weights $w_i = \frac{1}{n}$, for $i = 1, 2, \ldots, n$.

2. For $m = 1$ to M predictors or weak learners:

   (a) Fit a classifier $T^{(m)}(\boldsymbol{x})$ to the training data using weights w_i.

   (b) Compute the weighted error rate of the m^{th} predictor

   $$err^{(m)} = \frac{\sum_{i=1}^{n} w_i \mathbb{I}\left(c_i \neq T^{(m)}\left(\boldsymbol{x}_i\right)\right)}{\sum_{i=1}^{n} w_i}$$

   where

   - $err^{(m)}$ is the error rate of the m^{th} predictor
   - $\mathbb{I}\left(c_i \neq T^{(m)}\left(\boldsymbol{x}_i\right)\right)$ is an indicator function that takes the value of 1 when the training instance is misclassified and 0 if it is correctly classified
   - $\sum_{i=1}^{n} w_i \mathbb{I}\left(c_i \neq T^{(m)}\left(\boldsymbol{x}_i\right)\right)$ is the sum of the weights of misclassified instances
   - $\sum_{i=1}^{n} w_i$ is the sum of all n instance weights

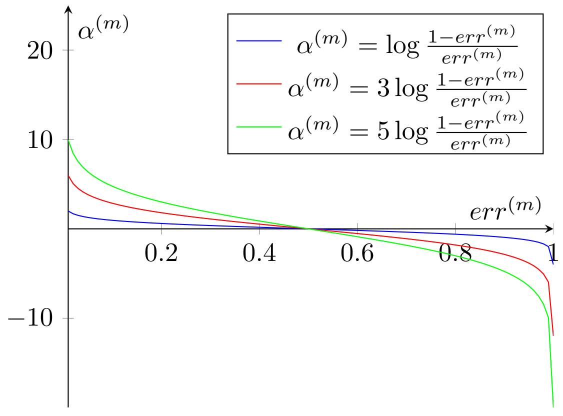
   (c) Compute the m^{th} predictor's weight using its error rate

   $$\alpha^{(m)} = \log \frac{1 - err^{(m)}}{err^{(m)}}$$

   Some implementations use a learning rate hyperparameter η such that this formula becomes

   $$\alpha^{(m)} = \eta \log \frac{1 - err^{(m)}}{err^{(m)}}$$

   The m^{th} predictor's weight has the following inverse relationship with its error rate (how many instances it misclassifies as a percentage of total number of instances):



   Two importance notes:

- **The larger the error rate $err^{(m)}$ of a predictor, the smaller the weight $\alpha^{(m)}$ that such predictor receives.** If the weak learner misclassifies a lot, we want to minimize its influence on the final model.
- For larger values of the learning parameter η, predictors with larger error rates $err^{(m)}$ are penalized more and the weights $\alpha^{(m)}$ for such weak learners are lower (blue always has lower weights than red and green). **With larger η, predictors with smaller error rates get larger weights (see green). Thus, increasing the learning parameter η reduces bias, but may lead to over-fitting.**

(d) Update the weights using the following rule

$$w_i \leftarrow w_i + \exp\left(\alpha^{(m)} + \mathbb{I}\left(c_i \neq T^{(m)}\left(\boldsymbol{x}_i\right)\right)\right)$$

for $i = 1, 2, \ldots, n$. This is so that the weights of the misclassified instances are boosted.

(e) Re-normalize each instance weight w_i by dividing by $\sum_{i=1}^{n} w_i$.

3. Finally, to make predictions, AdaBoost simply computes the predictions of all the m predictors and weighs them using the predictor weights $\alpha^{(m)}$. The predicted class is the one that receives the majority of weighted votes

$$C(\boldsymbol{x}) = \arg\max_k \sum_{m=1}^{M} \alpha^{(m)} \cdot \mathbb{I}\left(T^{(m)}(\boldsymbol{x}) = k\right)$$

Summary

The AdaBoost algorithm is an iterative procedure that tries to approximate the Bayes classifier $C^*(x)$ by combining many weak classifiers. Starting with the unweighted training sample, the AdaBoost builds a classifier, for example a classification tree, that produces class labels. If a training data point is misclassified, the weight of that training data point is increased (boosted). A second classifier is built using the new weights, which are no longer equal. Again, misclassified training data have their weights boosted and the procedure is repeated. **Typically, one may build 500 or 1000 classifiers this way.** A score $\alpha^{(m)}$ is assigned to each classifier, and the final classifier is defined as the linear combination of the classifiers from each stage.

Note

Note that the theory of AdaBoost assumes that the error of each weak classifier $err^{(m)}$ is less than $\frac{1}{2}$ (or equivalently $\alpha^{(m)} > 0$, see figure above), with respect to the distribution on which it was trained. This assumption is easily satisfied for two-class classification problems, because the error rate of random guessing is $\frac{1}{2}$. However, it is much harder to achieve in the multi-class case, where the random guessing error rate is $\frac{K-1}{K}$. The main disadvantage of AdaBoost is that it is unable to handle weak learners with an error rate greater than 1/2. As a result, AdaBoost may easily fail in the multi-class case.

4.1.2 Multi-Class Problem

The **SAMME** algorithm— Stagewise Additive Modeling using a Multi-class Exponential Loss function—can be employed to solve for multi-class problems.

1. Initialize the observation weights $w_i = \frac{1}{n}$, for $i = 1, 2, \ldots, n$.
2. For $m = 1$ to M predictors or weak learners:
   (a) Fit a classifier $T^{(m)}(\boldsymbol{x})$ to the training data using weights w_i.

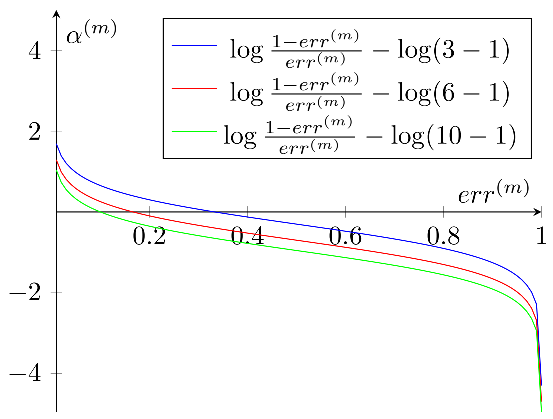
   (b) Compute the weighted error rate of the m^{th} predictor

   $$err^{(m)} = \frac{\sum_{i=1}^{n} w_i \mathbb{I}\left(c_i \neq T^{(m)}\left(\boldsymbol{x}_i\right)\right)}{\sum_{i=1}^{n} w_i}$$

   (c) Compute the m^{th} predictor's weight using its error rate

   $$\alpha^{(m)} = \log \frac{1 - err^{(m)}}{err^{(m)}} - \log(K-1).$$

   where K is the numbers of classes. The m^{th} predictor's weight has the following inverse relationship with its error rate:

   Two importance notes:

   - Notice that the x-intercepts for $K = 3, 6, 10$ are $\frac{1}{3} \approx 0.3333$ (blue), $\frac{1}{6} \approx 0.1667$ (red), and $\frac{1}{10} = 0.1$ (green), respectively. These are the probabilities of random guesses.

   (d) Update the weights using the following rule

   $$w_i \leftarrow w_i + \exp\left(\alpha^{(m)} + \mathbb{I}\left(c_i \neq T^{(m)}\left(\boldsymbol{x}_i\right)\right)\right)$$

   for $i = 1, 2, \ldots, n$. This is so that the weights of the misclassified instances are boosted.

   (e) Re-normalize each instance weight w_i by dividing by $\sum_{i=1}^{n} w_i$.

3. Output

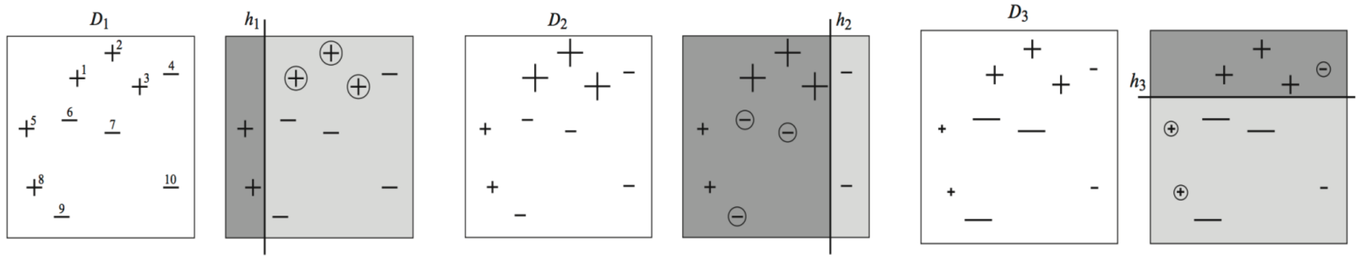
$$C(\boldsymbol{x}) = \arg\max_k \sum_{m=1}^{M} \alpha^{(m)} \cdot \mathbb{I}\left(T^{(m)}(\boldsymbol{x}) = k\right)$$

Summary

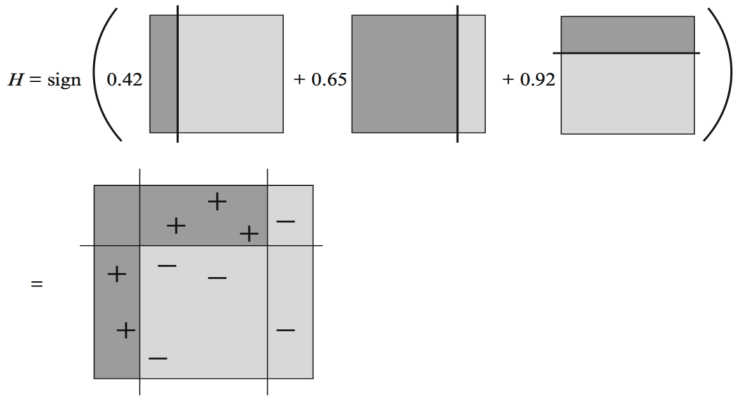
One immediate consequence is that now in order for each weak learner's weight $\alpha^{(m)}$ to be positive (above the x-axis $\frac{1}{K}$), we only need $(1 - err^{(m)}) > \frac{1}{K}$. In other words, the accuracy of each weak classifier only has to be better than random guessing (with probability $\frac{1}{K}$) rather than $\frac{1}{2}$, as in the two-class case.

4.1.3 Visual Intuition

The AdaBoost algorithm fits an additive model (ensemble) $\sum_t \rho_t h_t(x)$ in a forward stage-wise manner. In each stage, a weak learner is introduced to compensate for the shortcomings of existing weak learners. The "shortcomings" are identified by high-weight data points. The classification problem below has two classes $+$ and $-$.



- The first learner h_1 miss-classifies three training samples for class $+$; these samples are boosted in D_2.
- The second learner h_2 compensates and now classifies all instance of class $+$ correctly, but, in doing so, it miss-classifies three training samples for the other class $-$. These samples are boosted in D_3.
- Finally, the third learning h_3 then compensates for the mistakes of h_2.



4.2 Scenario For Gradient Boosting

Hypothesis class $\mathbb{H}$, whose set of classifiers has large bias and the training error is high (e.g. CART trees with very limited depth.) Can weak learners (H) be combined to generate a strong learner with low bias?

<u>**Solution**</u>

Create ensemble classifier $H_T(\vec{x}) = \sum_{t=1}^{T} \alpha_t h_t(\vec{x})$. This ensemble classifier is built in an iterative fashion. In iteration t we add the classifier $\alpha_t h_t(\vec{x})$ to the ensemble. At test time, we evaluate all classifier and return the weighted sum. The process of constructing such an ensemble in a additive, stage-wise fashion is very similar to gradient descent. However, instead of updating the model parameters in each iteration, we add functions to our ensemble. Let ℓ denote a (convex and differentiable) loss function. With a little abuse of notation we write

$$\ell(H) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(H\left(\mathbf{x}_i\right), y_i\right)$$

Assume we have already finished t iterations and already have an ensemble classifier $H_t(\vec{x})$. Now in iteration $t+1$ we want to add one more weak learner h_{t+1} to the ensemble. We search for the weak learner $h \in \mathbb{H}$ that minimizes the loss function the most,

$$h_{t+1} = \operatorname{argmin}_{h \in \mathbb{H}} \ell\left(H_t + \alpha h_t\right)$$

Once h_{t+1} has been found, we add it to our ensemble, i.e. $H_{t+1} := H_t + \alpha h$. In order to find such $h \in \mathbb{H}$, we use gradient descent in the functional space. In the functional space, the inner product of two real-valued functions g and h can be defined as $\langle h, g \rangle = \int_x h(x) g(x) dx$. Since we are dealing with training data that are vectors, we define $\langle h, g \rangle = \sum_{i=1}^{n} h\left(\mathbf{x}_i\right) g\left(\mathbf{x}_i\right)$.

4.2.1 Taylor Approximation

Taylor's approximation can be used to minimize a function ℓ. Provided that the norm $\|\mathbf{s}\|_2$ is small (i.e. $\mathbf{w}+\mathbf{s}$ is very close to $\mathbf{w}$), we can approximate the function $\ell(\mathbf{w}+\mathbf{s})$ by its first and second derivatives:

$$\ell(\mathbf{w}+\mathbf{s}) \approx \ell(\mathbf{w}) + g(\mathbf{w})^\top \mathbf{s}$$
$$\ell(\mathbf{w}+\mathbf{s}) \approx \ell(\mathbf{w}) + g(\mathbf{w})^\top \mathbf{s} + \frac{1}{2} \mathbf{s}^\top H(\mathbf{w}) \mathbf{s}$$

Here, $g(\mathbf{w}) = \nabla \ell(\mathbf{w})$ is the gradient and $H(\mathbf{w}) = \nabla^2 \ell(\mathbf{w})$ is the Hessian of ℓ. Both approximations are valid if $\|\mathbf{s}\|_2$ is small, but the second one assumes that ℓ is twice differentiable and is more expensive to compute but also more accurate than only using gradient.

4.2.2 Gradient Descent In Functional Space

Given H, we want to find the step-size α and (weak learner) h to minimize the loss $\ell(H+\alpha h)$. Using Taylor Approximation on $\ell(H+\alpha h)$:

$$\ell(H+\alpha h) \approx \ell(H) + \langle \nabla \ell(H), \alpha h \rangle$$
$$\ell(H+\alpha h) \approx \ell(H) + \alpha \langle \nabla \ell(H), h \rangle$$

This approximation (of ℓ as a linear function) only holds within a small region around $\ell(H)$, i. as long as α is small. We therefore fix it to a small constant (e.g. $\alpha \approx 0.1$). With the step-size α fixed, we can use the approximation above to find an almost optimal h that minimizes the loss function the most:

$$\begin{aligned}\operatorname{argmin}_{h\in\mathbb{H}} \ell(H+\alpha h) &\approx \operatorname{argmin}_{h\in\mathbb{H}} \cancel{\ell(H)} + \langle \nabla \ell(H), \alpha h \rangle \\ &\approx \operatorname{argmin}_{h\in\mathbb{H}} \langle \nabla \ell(H), \alpha h \rangle \\ &= \text{Because we have n training samples } i, ..., n \text{, we can evaluate the function } h \\ &\quad \text{and the gradient vector } \nabla \ell(H) \text{ using the training data;} \\ &\quad \text{then, the inner product of two functions reduces to the inner product of two vectors} \\ &= \operatorname{argmin}_{h\in\mathbb{H}} \sum_{i=1}^{n} \frac{\partial \ell}{\partial \left[H\left(\mathbf{x}_i\right)\right]} \cdot \alpha h\left(\mathbf{x}_i\right)\end{aligned}$$

The reason $\ell(H)$ is dropped in the minimization is because we can write each prediction as an input to the loss function:

$$\ell(H) = \sum_{i=1}^{n} \ell\left(H\left(\mathbf{x}_i\right)\right) = \ell\left(H\left(x_1\right), \ldots, H\left(x_n\right)\right)$$

So $\ell(H)$ is simply a constant that becomes zero. To solve the minimization,

$$h_{t+1} = \operatorname{argmin}_{h\in\mathbb{H}} \sum_{i=1}^{n} \underbrace{\frac{\partial \ell}{\partial \left[H\left(\mathbf{x}_i\right)\right]}}_{r_i} \alpha h(x)$$

we need a function $\mathrm{A}\left(\left\{\left(\mathbf{x}_1, r_1\right), \ldots, \left(\mathbf{x}_n, r_n\right)\right\}\right) = \operatorname{argmin}_{h\in\mathbb{H}} \sum_{i=1}^{n} r_i \alpha h\left(\mathbf{x}_i\right)$. Note that α can also be dropped from the minimization problem.

4.3 Gradient Boosting Tree

- Classification ($y_i \in \{+1, -1\}$) or (even multi-dimensional) regression $\left(y_i \in \mathcal{R}^k\right)$
- Weak learners, $h \in \mathbb{H}$, are regressors, $h(\mathbf{x}) \in \mathcal{R}, \forall \mathbf{x}$, typically fixed-depth (e.g. depth=4) regression trees.
- Step size or shrinkage α is fixed to a small constant (hyper-parameter).
- Loss function: Any differentiable convex loss that decomposes over the samples $\mathcal{L}(H) = \sum_{i=1}^{n} \ell\left(H\left(\mathbf{x}_i\right)\right)$

In order to use regression trees for gradient boosting, we must be able to find a tree $h()$ that maximizes $h = \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^{n} r_i h(\mathbf{x}_i)$ where $r_i = \frac{\partial \ell}{\partial H(\mathbf{x}_i)}$. Two assumptions:

1. Assume that $\sum_{i=1}^{n} h^2(\mathbf{x}_i) = \text{constant}$. This is simple to do (we normalize the predictions of $h(\mathbf{x}_i)$) and important because we could always decrease $\sum_{i=1}^{n} h(\mathbf{x}_i) r_i$ by rescaling h with a large constant. By fixing $\sum_{i=1}^{n} h^2(\mathbf{x}_i)$ to a constant we are essentially fixing the vector h (the predictions of the next weak learner) to lie on a circle, and we are only concerned with its direction but not its length.

2. We can define the negative gradient $r_i = \frac{\partial \ell}{\partial H(\mathbf{x}_i)}$ as $t_i = -r_i$.

Then letting $h(\mathbf{x}_i)$ be the weak learner that minimizes the loss function the most,

$$
\begin{aligned}
&\operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^{n} r_i h(\mathbf{x}_i) \qquad \text{(This is the original formulation)} \\
&= \operatorname{argmin}_{h \in \mathbb{H}} -2 \sum_{i=1}^{n} t_i h(\mathbf{x}_i) \qquad \text{(Swapping in } t_i \text{ for } -r_i \text{ and multiplying by 2, which is a constant)} \\
&= \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^{n} \underbrace{t_i^2}_{\text{constant}} -2 t_i h(\mathbf{x}_i) + \underbrace{(h(\mathbf{x}_i))^2}_{\text{constant}} \qquad \text{(Adding constant } \sum_i t_i^2 + h(\mathbf{x}_i)^2\text{)} \\
&= \operatorname{argmin}_{h \in \mathbb{H}} \sum_{i=1}^{n} (h(\mathbf{x}_i) - t_i)^2
\end{aligned}
$$

In other words, we can feed in the value t_i (negative gradients evaluated on the training set) as targets (response or label) for each $\mathbf{x}_i$. During each iteration, we build a new tree for a different set of "targets" (response or label) $t_1, \ldots, t_n$.

4.3.1 Square Loss

If the loss function ℓ is the squared loss, i.e. $\ell(H) = \frac{1}{2} \sum_{i=1}^{n} (H(\mathbf{x}_i) - y_i)^2$, then it is easy to show that

$$
\begin{aligned}
\frac{\partial \ell}{\partial H(\mathbf{x}_i)} &= \frac{\cancel{2}}{\cancel{2}} (H(\mathbf{x}_i) - y_i)^{\cancel{2}} \\
&= (H(\mathbf{x}_i) - y_i)
\end{aligned}
$$

And therefore

$$
\begin{aligned}
t_i &= -\frac{\partial \ell}{H(\mathbf{x}_i)} \\
&= -(H(\mathbf{x}_i) - y_i) \\
&= y_i - H(\mathbf{x}_i)
\end{aligned}
$$

This above step shows that, in the regression context with squared loss, we are really just fitting trees $h(\mathbf{x}_i)$ with fixed depths to the training data (X, t_i). That is, we fit a tree using the current residuals, rather than the outcome Y, as the response. However, it is important to note that we can use any other differentiable and convex loss function ℓ, and the solution for our next weak learner $h()$ will always be the regression tree minimizing the squared loss.

4.3.2 Absolute Loss

For absolute loss function, the negative gradient is:

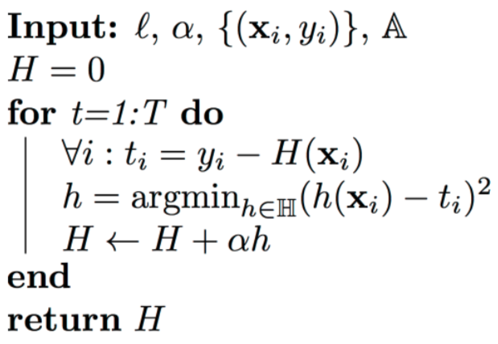
$$-g\left(x_i\right)=-\frac{\partial \ell}{\partial H\left(\mathbf{x}_i\right)}=\operatorname{sign}\left(y_i-H\left(\mathbf{x}_i\right)\right)$$

4.3.3 Huber Loss

For Huber loss function, the negative gradient:

$$\begin{aligned} -g\left(x_i\right) &= -\frac{\partial \ell}{\partial H\left(\mathbf{x}_i\right)} \\ &= \begin{cases} y_i-H\left(\mathbf{x}_i\right) & \left|y_i-H\left(\mathbf{x}_i\right)\right| \leq \delta \\ \delta \operatorname{sign}\left(y_i-H\left(\mathbf{x}_i\right)\right) & \left|y_i-H\left(\mathbf{x}_i\right)\right| > \delta \end{cases} \end{aligned}$$

4.3.4 Pseudo Code



```
**Input:** $\ell$, $\alpha$, $\{(\mathbf{x}_i, y_i)\}$, $\mathbb{A}$
$H = 0$
**for** $t$=1:$T$ **do**
    $\forall i : t_i = y_i - H(\mathbf{x}_i)$
    $h = \text{argmin}_{h \in \mathbb{H}} (h(\mathbf{x}_i) - t_i)^2$
    $H \leftarrow H + \alpha h$
**end**
**return** $H$
```

- For all weak learners $1, ..., T$, compute the residuals for all n training samples.
- Find the next weak learner h such that the loss function is minimized the most.
- Update H by adding in a *shrunken* version of the new tree or weak learner.
- The boosted model is then simply:

$$H_T(\vec{x})=\sum_{t=1}^{T} \alpha_t h_t(\vec{x})$$

4.3.5 Hyper-parameters

The three hyper-parameters to consider for gradient boosting trees are:

1. The number of trees T. Unlike bagging and random forests, boosting can overfit if T is too large (higher variance and lower bias), although this over-fitting tends to occur slowly if at all. We use cross-validation to select T.

2. The shrinkage or step-size parameter α, a small positive number. This controls the rate at which boosting learns. Typical values are 0.01 or 0.001, and the right choice can depend on the problem. There is a **trade-off** in the sense that very small α (reducing variance but increases bias) can require using a very large number of trees T in order to achieve good performance.

3. The number d of splits in each tree, which controls the complexity of the boosted ensemble. Often $d = 1$ (or $d = 4$) works well, in which case each tree is a **stump**, consisting of a single split. In this case, the boosted ensemble is fitting an additive model, since each term involves only a single variable. More generally d is the **interaction depth**, and controls the interaction order of the boosted model, since d splits can involve at most d variables.