# Gradient boosting performs gradient descent



Terence Parr and Jeremy Howard

Please send comments, suggestions, or fixes to Terence.

Contents



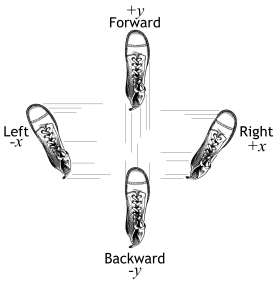
So far we've looked at GBMs that use two different direction vectors, the residual vector (Gradient boosting: Distance to target) and the sign vector (Gradient boosting: Heading in the right direction). It's natural to ask whether there are other direction vectors we can use and what effect they have on the final $F_M(X)$ predictions. Your intuition probably told you that nudging the intermediate $\hat{\mathbf{y}}$ prediction towards target $\mathbf{y}$ gradually improves model performance, but would $\hat{\mathbf{y}}$ ever stop if we kept increasing $M$? If so, where would it stop? Is there something special or interesting about the sequence of vectors that $\hat{\mathbf{y}}$ passes through? Moreover, we know that training a model on observations $(\mathbf{x}, y)$ is a matter of finding a function, $F(\mathbf{x})$, that optimizes some cost or loss function indicating how well $F$ performs. (The procedure is to tweak model $F$'s model parameters until we minimize the loss function.) What loss function then is a GBM optimizing and what is the relationship with the choice of direction vector?

To answer these questions, we're going to employ a mathematician's favorite trick: showing how our current problem is just a flavor of another well-known problem for which we have lots of useful results. Specifically, this article shows how gradient boosting machines perform an optimization technique from numerical methods called gradient or steepest descent. We'll see that a GBM training weak leaners on residual vectors optimizes the mean squared error (MSE), the $L_2$ loss, between the true target $\mathbf{y}$ and the intermediate predictions, $\hat{\mathbf{y}}_m = F_m(X)$ for observation matrix $X = [\mathbf{x}_1 \ldots \mathbf{x}_N]$. A GBM that trains weak learners on sign vectors optimizes the mean absolute error (MAE), the $L_1$ loss.

To help make the connection between gradient boosting and gradient descent, let's take a small detour to reinvent the technique of gradient descent used to optimize functions. Once we've got a good handle on both ideas, we'll see how similar they are mathematically.

## The intuition behind gradient descent

Both of us (Terence and Jeremy) have a story where we had to make our way down a mountain in the pitch black of night or dense fog. (*Spoiler alert*: we both made it back!) Obviously, the way to get down is to keep going downhill until you reach the bottom, taking steps to the left, right, forward, backward or at an angle in order to minimize the "elevation function." Rather than trying to find the best angle to step, we can treat each direction, forward/backward and left/right, separately and then combine them to obtain the best step direction. The procedure is to swing your foot forwards and backwards to figure out which way is downhill on, say, the North/South axis, then swing your foot left and right to figure out which way is downhill on the East/West axis (Boot clipart from `http://etc.usf.edu/clipart/`):




We're looking for a direction vector with components for each of the $x$ and $y$ axes that takes us downhill, according to some elevation function, $f(x, y)$. We can represent the components of the downhill direction vector as a sign vector, such as

[*which way is down in x direction*, *which way is down in y direction*] = $[-1, 1]$

which would indicate a step to the left and forward. To actually move downhill, all we have to do is add the direction vector to the current position to get the new position: $[x, y] = [x, y] + [-1, 1]$.

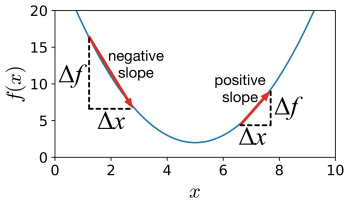
The sign vector works well but does not take into consideration the steepness of the slope in each axis. If the slope is gradual to the left ($x$) but steep to the front ($y$), we might prefer a direction vector of $[-1, 4]$. The idea is to take big steps along an axis when the slope is steep but small steps when the slope is shallow. As we approach the bottom of the mountain, we should take really small steps to avoid overshooting and going back up the other side. Updating the current position with a

direction vector containing magnitude information, proportional to the slope, automatically takes smaller steps as the slope flattens out.

Mathematically, we don't move our foot around to learn about the slope of $f(x,y)$, we compute the direction vector from the partial derivatives of $f(x,y)$ with respect to the $x$ and $y$ dimensions. Such derivatives use notation $\frac{\partial f(x,y)}{\partial x}$ and $\frac{\partial f(x,y)}{\partial y}$. If we stack those partial derivatives into a vector, it's called a *gradient* and is written:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix}$$

The gradient is is actually the opposite or negative of the direction vector we want because the slope always points in the uphill direction. This is easiest to see in two dimensions:




For a small change change in $x$ to the right, $\Delta x$, the function value $f(x)$ will go up if the slope is positive or go down if the slope is negative. Here, $\Delta f$ is negative for $x < 5$ and positive for $x > 5$. Consequently, to move downhill, not uphill, we update the current position by adding in the **negative** of the gradient.

Let's generalize our position update equation to handle more than two dimensions. Rather than $x$ and $y$, let's use $\mathbf{x}$ (in bold) to represent all of the function parameters, which means our "elevation" function to optimize now takes a vector argument, $f(\mathbf{x})$. As we update $\mathbf{x}$, we want the value of $f(\mathbf{x})$ to decrease. When it stops decreasing, $\mathbf{x}$ will have arrived at the position giving the minimum value of $f(\mathbf{x})$. Because we'll need this to show the relationship between gradient boosting and gradient descent, let's formally give the update equation. The next position of $\mathbf{x}$, at time step $t$, is given by:

$$\mathbf{x}_t = \mathbf{x}_{t-1} + (-\nabla f(\mathbf{x}_{t-1})) = \mathbf{x}_{t-1} - \nabla f(\mathbf{x}_{t-1})$$

In practice, we need to restrict the size of the steps we take from $\mathbf{x}_{t-1}$ to $\mathbf{x}_t$ by shrinking the direction vector using a learning rate, $\eta$, whose value is less than 1. (Imagine a function that is only valid between, say, -2 and 2 but whose output values range from 0 to 1,000; direction vectors derived from the slope would force overly large steps and so we attenuate the steps with the learning rate.) This brings us to the position update equation found in most literature:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \nabla f(\mathbf{x}_{t-1})$$

The key takeaways from this gradient descent discussion are:

- Minimizing a function, $f(\mathbf{x})$, means finding the $\mathbf{x}$ position where $f(\mathbf{x})$ has minimal value.
- The procedure is to pick some initial (random or best guess) position for $\mathbf{x}$ and then gradually nudge $\mathbf{x}$ in the downhill direction, which is the direction where the $f(\mathbf{x})$ value is smaller.
- The gradient of $f(\mathbf{x})$ gives us the direction of uphill and so we negate the gradient to get the downhill direction vector.
- We update position $\mathbf{x}_{t-1}$ to $\mathbf{x}_t$, where the function is lower, by adding the direction vector to $\mathbf{x}$, scaled by the learning rate, $\eta$.

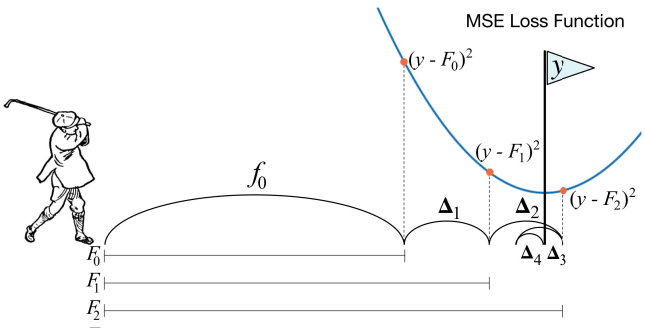
Ok, we're finally ready to show how gradient boosting is doing a particular kind of gradient descent.

## Boosting as gradient descent in prediction space

Our goal is to show that training a GBM is performing gradient-descent minimization on some loss function between our true target, $\mathbf{y}$, and our approximation, $\hat{\mathbf{y}}_m = F_m(X)$. That means showing that adding weak models, $\Delta_m$, to our GBM additive model:

$$F_m(X) = F_{m-1}(X) + \eta \Delta_m(X)$$

is performing gradient descent in some way. It makes sense that nudging our approximation, $\hat{\mathbf{y}}$, closer and closer to the true target $\mathbf{y}$ would be performing gradient descent. For example, at each step, the residual $\mathbf{y} - \hat{\mathbf{y}}$ gets smaller. We must be minimizing some function related to the distance between the true target and our approximation. Let's revisit our golfer analogy and visualize the squared error between the approximation and the true value, $(y - F_m)^2$:



Since $\hat{y} = F_m(x)$, let's think about the ball as $\hat{y}$ and the golfer nudging $\hat{y}$ towards target $y$. At each step, we have to figure out which direction to go. At $F_0$ and $F_1$, we should move $\hat{y}$ to the right; at $F_2$, we should move $\hat{y}$ to the left. How far should we move $\hat{y}$? We could move by +1 and -1, the sign of the direction, or we could take into consideration the distance of $\hat{y}$ to the target $y$, $y - \hat{y}$. That distance just happens to be in the opposite (negative) direction of the slope of the MSE loss function, $(y - \hat{y})^2$. (Recall that the derivative is positive in the uphill direction and so taking the negative gives the direction of downhill or lower cost.) So, at least in this single-observation case, adding the residual to $\hat{y}$ is subtracting the slope, which is exactly what gradient descent does.

**The key insight**
The key to unlocking the relationship for more than one observation is to see that the residual, $\mathbf{y} - \hat{\mathbf{y}}$, is a direction vector. It's not just the magnitude of the difference. Moreover, the vector points in the direction of a better approximation and, hence, a smaller loss between the true $\mathbf{y}$ and $\hat{\mathbf{y}}$ vectors. That suggests that the direction vector is also (the negative of) a loss function gradient. **Chasing the direction vector in a GBM is chasing the (negative) gradient of a loss function via gradient descent.**

In the next two sections, we'll show that the gradient of the MSE loss function is the residual direction vector and the gradient of the MAE loss function is the sign direction vector. Then, we can put it all together to show GBM is mathematically performing a gradient descent on the loss function.

### The MSE function gradient

To uncover the loss function optimized by a GBM whose $\Delta_m$ weak models are trained on the residual vector, we just have to integrate the residual $\mathbf{y} - \hat{\mathbf{y}}$. It's actually easier, though, to go the other direction and compute the gradient of the MSE loss function to show that it is the residual vector. The MSE loss function computed from $N$ observations in matrix $X = [\mathbf{x}_1 \ldots \mathbf{x}_N]$ is:

$$L(\mathbf{y}, F_M(X)) = \frac{1}{N}\sum_{i=1}^{N}(y_i - F_M(\mathbf{x}_i))^2$$

but let's substitute $\hat{\mathbf{y}}$ for the model output, $F_M(X)$, to make the equation more clear:

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

Also, since $N$ is a constant once we start boosting, and $f(x)$ and $cf(x)$ have the same $x$ minimum point, let's drop the $\frac{1}{N}$ constant:

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

Now, let's take the partial derivative of the loss function with respect to a specific $\hat{y}_j$ approximation:

$$\begin{aligned}
\frac{\partial}{\partial \hat{y}_j} L(\mathbf{y}, \hat{\mathbf{y}}) &= \frac{\partial}{\partial \hat{y}_j}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \\
&= \frac{\partial}{\partial \hat{y}_j}(y_j - \hat{y}_j)^2 \\
&= 2(y_j - \hat{y}_j)\frac{\partial}{\partial \hat{y}_j}(y_j - \hat{y}_j) \\
&= -2(y_j - \hat{y}_j)
\end{aligned}$$

(We can remove the summation because the partial derivative of $L$ for $i \neq j$ is 0.)

That means the gradient with respect to $\hat{\mathbf{y}}$ is:

$$\nabla_{\hat{\mathbf{y}}} L(\mathbf{y}, \hat{\mathbf{y}}) = -2(\mathbf{y} - \hat{\mathbf{y}})$$

Dropping the constant in front again leaves us with the gradient being the same as the residual vector: $\mathbf{y} - \hat{\mathbf{y}}$. So, chasing the residual vector in a GBM is chasing the gradient vector of the MSE $L_2$ loss function while performing gradient descent.

### The MAE function gradient

Let's see what happens with the MAE loss function:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{N}|y_i - \hat{y}_i|$$

The partial derivative with respect to a specific approximation $\hat{y}_j$ is:

$$\begin{aligned}
\frac{\partial}{\partial \hat{y}_j} L(\mathbf{y}, \hat{\mathbf{y}}) &= \frac{\partial}{\partial \hat{y}_j}\sum_{i=1}^{N}|y_i - \hat{y}_i| \\
&= \frac{\partial}{\partial \hat{y}_j}|y_j - \hat{y}_j| \\
&= sign(y_j - \hat{y}_j)\frac{\partial}{\partial \hat{y}_j}(y_j - \hat{y}_j) \\
&= -sign(y_j - \hat{y}_j)
\end{aligned}$$

giving gradient:

$$\nabla_{\hat{y}} L(\mathbf{y}, \hat{\mathbf{y}}) = -sign(\mathbf{y} - \hat{\mathbf{y}})$$

This shows that chasing the sign vector in a GBM is chasing the gradient vector of the MAE $L_1$ loss function while performing gradient descent.

### Morphing GBM into gradient descent

Now that we have all of the pieces, let's prove the general result that a GBM is performing gradient descent on a loss function that compares the target $\mathbf{y}$ to the model's previous approximation, $F_{m-1}(X)$, to get the new approximation, $F_m(X)$. To do this, we'll morph the GBM additive model recurrence relation into the gradient descent position update equation. Let's start by simplifying the GBM recurrence relation:

$$F_m(X) = F_{m-1}(X) + \eta\Delta_m(X)$$

by substituting the $m^{th}$ approximation variable as $\hat{\mathbf{y}}_m = F_m(X)$ to get:

$$\hat{\mathbf{y}}_m = \hat{\mathbf{y}}_{m-1} + \eta\Delta_m(X)$$

Since $\Delta_m$ is trained on and is an approximation to the direction vector, which is a function of $F_{m-1}(X)$, such as residual vector $\mathbf{y} - F_{m-1}(X)$, let's replace the weak model prediction with the target data it's trained on. We haven't given the direction vector a symbol yet, so for the purposes of this proof, let's refer to the direction vector as $\mathbf{r}_{m-1}$ because it is derived from $F_{m-1}(X)$. (Friedman calls $\mathbf{r}_{m-1}$ the *pseudo-response* and uses symbol $\tilde{\mathbf{y}}$, but we avoid $\tilde{\mathbf{y}}$ because it resembles our use of $\hat{y}$ too much.) Our equation now looks much simpler:

$$\hat{\mathbf{y}}_m = \hat{\mathbf{y}}_{m-1} + \eta\mathbf{r}_{m-1}$$

Next, we can flip the addition to a subtraction by subtracting the negative:

$$\hat{\mathbf{y}}_m = \hat{\mathbf{y}}_{m-1} - \eta(-\mathbf{r}_{m-1})$$

and there's nothing wrong with changing the iteration variable from *m* to *t*:

$$\hat{\mathbf{y}}_t = \hat{\mathbf{y}}_{t-1} - \eta(-\mathbf{r}_{t-1})$$

Now, compare that to the gradient descent position update equation:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta\nabla f(\mathbf{x}_{t-1})$$

Those equations are identical if we choose direction vector $\mathbf{r}_{t-1}$ to be $-\nabla f(\hat{\mathbf{y}}_{t-1})$ for some $f$, which means a GBM would be performing gradient descent using the gradient of $f$. Let $f$ be the general loss function $L(\mathbf{y}, \hat{\mathbf{y}}_{m-1})$ and we have discovered what direction vector to train our $\Delta_m$ weak models on:

$$\mathbf{r}_{m-1} = -\nabla L(\mathbf{y}, \hat{\mathbf{y}}_{m-1})$$

So, adding another $\Delta_m$ weak model to a GBM is actually adding the negative of the gradient of a loss function to get the next approximation:

| **Gradient descent** | **Gradient boosting** |
|---|---|
| $\mathbf{x}_t = \mathbf{x}_{t-1} - \eta\nabla f(\mathbf{x}_{t-1})$ | $\hat{\mathbf{y}}_m = \hat{\mathbf{y}}_{m-1} + \eta(-\nabla L(\mathbf{y}, \hat{\mathbf{y}}_{m-1}))$ |

When $L$ is the MSE loss function, $L$'s gradient is the residual vector and a gradient descent optimizer should chase that residual, which is exactly what the gradient boosting machine does as well. When $L$ is the MAE loss function, $L$'s gradient is the sign vector, leading gradient descent and gradient boosting to step using the sign vector.

The implications of all of this fancy footwork is that we can use a GBM to optimize any differentiable loss function by training our weak models on the negative of the loss function gradient (with respect to the previous approximation). Understanding this derivation from the GBM recurrence relation to gradient descent update equation is much harder to see without the $\hat{\mathbf{y}}_m = F_m(X)$ substitution, as we'll see next.

### Function space is prediction space

Most GBM articles follow Friedman's notation (on page 4, equation for $g_m(\mathbf{x}_i)$) and describe the gradient as this scary-looking expression for the partial derivative with respect to our approximation of $y_i$ for observation $\mathbf{x}_i$:

$$\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}$$

Hmm... let's see if we can tease this apart. First, evaluate the expression according to the subscript, $F(\mathbf{x}) = F_{m-1}(\mathbf{x})$:

$$\frac{\partial L(y_i, F_{m-1}(\mathbf{x}_i))}{\partial F_{m-1}(\mathbf{x}_i)}$$

Next, let's remove the *i* index variable to look at the entire gradient, instead of a specific observation's partial derivative:

$$\nabla_{F_{m-1}(X)} L(\mathbf{y}, F_{m-1}(X))$$

But, what does it mean to take the partial derivative with respect to a function, $F_{m-1}(X)$? Here is

where we find it much easier to understand the gradient expression using $\hat{\mathbf{y}}_m$, rather than $F_m(X)$. Both are just vectors, but it's easier to see that when we use a simple variable name, $\hat{\mathbf{y}}_m$. Substituting, we get a gradient expression that references two vector variables $\mathbf{y}$ and $\hat{\mathbf{y}}_{m-1}$:

$$\nabla_{\hat{\mathbf{y}}_{m-1}} L(\mathbf{y}, \hat{\mathbf{y}}_{m-1})$$

Variable $\hat{\mathbf{y}}_{m-1}$ is a position in "function space," which just means a vector result of evaluating function $F_{m-1}(X)$. This is why GBMs perform "gradient descent in function space," but it's easier to think of it as "gradient descent in prediction space" where $\hat{\mathbf{y}}_{m-1}$ is our prediction.

## How gradient boosting differs from gradient descent

Before finishing up, it's worth examining the differences between gradient descent and gradient boosting. To make things more concrete, let's consider applying gradient descent to train a neural network (NN). Training seeks to find the weights and biases, model parameters $\mathbf{x}$, that optimize the loss between the desired NN output, $\mathbf{y}$, and the current output, $\hat{\mathbf{y}}$. If we assume a squared error loss function, NN gradient descent training computes the next set of parameters by adding the residual vector, $\mathbf{y} - \hat{\mathbf{y}}$, to the current $\mathbf{x}$ (subtracting the squared error gradient).

In contrast, GBMs are meta-models consisting of multiple weak models whose output is added together to get an overall prediction. The optimization we're concerned with here occurs, not on the parameters of the weak models themselves but, instead, on the composite model prediction, $\hat{\mathbf{y}}_m = F_m(X)$. GBM training occurs on two levels then, one to train the weak models and one on the overall composite model. It is the overall training of the composite model that performs gradient descent by adding the residual vector (assuming a squared error loss function) to get the improved model prediction. Training a NN using gradient descent tweaks model parameters whereas training a GBM tweaks (boosts) the model output.

Also, training a NN with gradient descent directly adds a direction vector to the current $\mathbf{x}$, whereas training a GBM adds a weak model's approximation of the direction vector to the current output, $\hat{\mathbf{y}}$. Consequently, it's likely that a GBM's MSE and MAE will decrease monotonically during training but, given the weak approximations of our $\Delta_m$, monotonicity is not guaranteed. The GBM loss function could bounce around a bit on its way down.

One final note on training regression trees used for weak models. The interesting thing is that, regardless of the direction vector (negative gradient), regression trees can always get away with using the squared error to compute node split points; i.e., even when the overall GBM is minimizing the absolute error. The difference between optimizing MSE and MAE error for the GBM is that the weak models train on different direction vectors. How the regression trees compute splits is not a big issue since the stumps are really weak and give really noisy approximations anyway.

## Summary

This 3-part article exploded in size beyond our initial expectations, but hopefully it will provide the necessary pieces to explain how gradient boosting machines work in detail. There's a lot of math in this last chunk, but we can summarize it as follows. Every time we add a new weak model to a GBM, we hope to nudge our prediction, $\hat{\mathbf{y}}$, towards the target, $\mathbf{y}$. Prediction $\hat{\mathbf{y}}$ will take smaller and smaller steps to eventually converge on $\mathbf{y}$ (modulo noise generated by imperfect weak models, $\Delta_m$). We say that prediction $\hat{\mathbf{y}}$ sweeps through function space because $\hat{\mathbf{y}}$ it is the result of some $F_m(X)$ function evaluation. It's easier to think of this as sweeping through prediction space.

The nudges that we take are the residual or the sign vector between the true target and our approximation (for the common loss functions). We've shown these to be optimizing MSE and MAE, respectively, because the residual is the negative of the MSE gradient and the sign vector is the negative of the MAE gradient. Chasing the direction vector is, therefore, performing a gradient descent that optimizes the loss function.

We can use any differentiable loss function we want with GBMs, per the general algorithm in the next section, by using a direction vector that is the negative of the loss function's gradient. If we're satisfied with optimizing MSE or MAE, then all of the math in this last part of the 3-part article is unnecessary. We only need the math to show how to use any loss function we want. For the most part, GBM implementations will use the GBM algorithm to minimize L2 loss or GBM algorithm to minimize L1 loss.

## General algorithm with regression tree weak models

This general algorithm, derived from Friedman's Gradient Boost on page 5, assumes the use of regression trees and is more complex than the specific algorithms for $L_2$ and $L_1$ loss. We need to compute the gradient of the loss function, instead of just using the residual or sign of the residual, and we need to compute weights for regression tree leaves. Each leaf, $l$, has weight value, $w$, that minimizes the $\sum_{i \in l} L(y_i, F_{m-1}(\mathbf{x}_i) + w)$ for all $\mathbf{x}_i$ observations within that leaf.

**Algorithm:** $boost(X, \mathbf{y}, M, \eta)$ **returns**: model $F_M$
Let $F_0(X)$ be value $v$ minimizing $\sum_{i=1}^{N} L(y_i, v)$, loss across all obs.
**for** $m = 1$ **to** $M$ **do**
  Let $\mathbf{r}_{m-1} = \nabla_{\hat{\mathbf{y}}_{m-1}} L(\mathbf{y}, \hat{\mathbf{y}}_{m-1})$ where $\hat{\mathbf{y}}_{m-1} = F_{m-1}(X)$
  Train regression tree $\Delta_m$ on $\mathbf{r}_{m-1}$, minimizing squared error
  **foreach** *leaf* $l \in \Delta_m$ **do**
    Let $w$ be value minimizing $\sum_{i \in l} L(y_i, F_{m-1}(\mathbf{x}_i) + w)$ for obs. in leaf $l$
    Alter $l$ to predict $w$; i.e., (not the usual mean or median)
  **end**
  $F_m(X) = F_{m-1}(X) + \eta \Delta_m(X)$
**end**
**return** $F_M$

12 return $T_M$

To see how this algorithm reduces to that for the $L_2$ loss function we have two steps to do. First, let $L(\mathbf{y},\ \hat{\mathbf{y}}_{m-1}) = (\mathbf{y} - \hat{\mathbf{y}}_{m-1})^2$ , whose gradient gives the residual vector $\mathbf{r}_{m-1} = 2(\mathbf{y} - \hat{\mathbf{y}}_{m-1})$. Second, show that leaf weight, $w$, is the mean of the residual of the observations in each leaf becase the mean minimizes $\sum_{i \in l} L(y_i, F_{m-1}(\mathbf{x}_i) + w)$. That means minimizing:

$$\sum_{i \in l} (y_i - (F_{m-1}(\mathbf{x}_i) + w))^2$$

To find the minimal value of the function with respect to $w$, we take the partial derivative of that function with respect to $w$ and set to zero; then we solve for $w$. Here is the partial derivative:

$$\frac{\partial}{\partial w} \sum_{i \in l} (y_i - (F_{m-1}(\mathbf{x}_i) + w))^2 = 2 \sum_{i \in l} (y_i - (F_{m-1}(\mathbf{x}_i) + w)) \times \frac{\partial}{\partial w} (y_i - (F_{m-1}(\mathbf{x}_i) + w))$$

And since $\frac{\partial}{\partial w}(y_i - (F_{m-1}(\mathbf{x}_i) + w)) = 1$ , the last term drops off:

$$2 \sum_{i \in l} (y_i - F_{m-1}(\mathbf{x}_i) - w)$$

Now, set to 0 and solve for $w$:

$$\sum_{i \in l} 2F_{m-1}(\mathbf{x}_i) + 2w - 2y_i = 0$$

We can drop the constant by dividing both sides by 2:

$$\sum_{i \in l} F_{m-1}(\mathbf{x}_i) + w - y_i = 0$$

Then, pull out the $w$ term:

$$\sum_{i \in l} F_{m-1}(\mathbf{x}_i) - y_i + \sum_{i \in l} w = 0$$

and move to the other side of the equation:

$$\sum_{i \in l} (F_{m-1}(\mathbf{x}_i) - y_i) = -\sum_{i \in l} w$$

We can simplify the $w$ summation to a multiplication:

$$\sum_{i \in l} (F_{m-1}(\mathbf{x}_i) - y_i) = -n_l w \text{ where } n_l \text{ is number of obs. in } l$$

Let's also flip the order of the elements within the summation to get the target variable first:

$$\sum_{i \in l} (y_i - F_{m-1}(\mathbf{x}_i)) = n_l w$$

Divide both sides of the equation by the number of observations in the leaf:

$$w = \frac{1}{n_l} \sum_{i \in l} (y_i - F_{m-1}(\mathbf{x}_i))$$

Finally, we see that leaf weights, $w$, should be the mean when the loss function is the mean squared error:

$$w = mean(y_i - F_{m-1}(\mathbf{x}_i))$$

The mean is exactly what the leaves of the regression tree, trained on residuals, will predict.