All You Need to Know about Gradient Boosting Algorithm − Part 2. Classification

Algorithm explained with an example, math, and code

Tomonori Masui · Follow

Published in Towards Data Science

13 min read · Feb 7, 2022

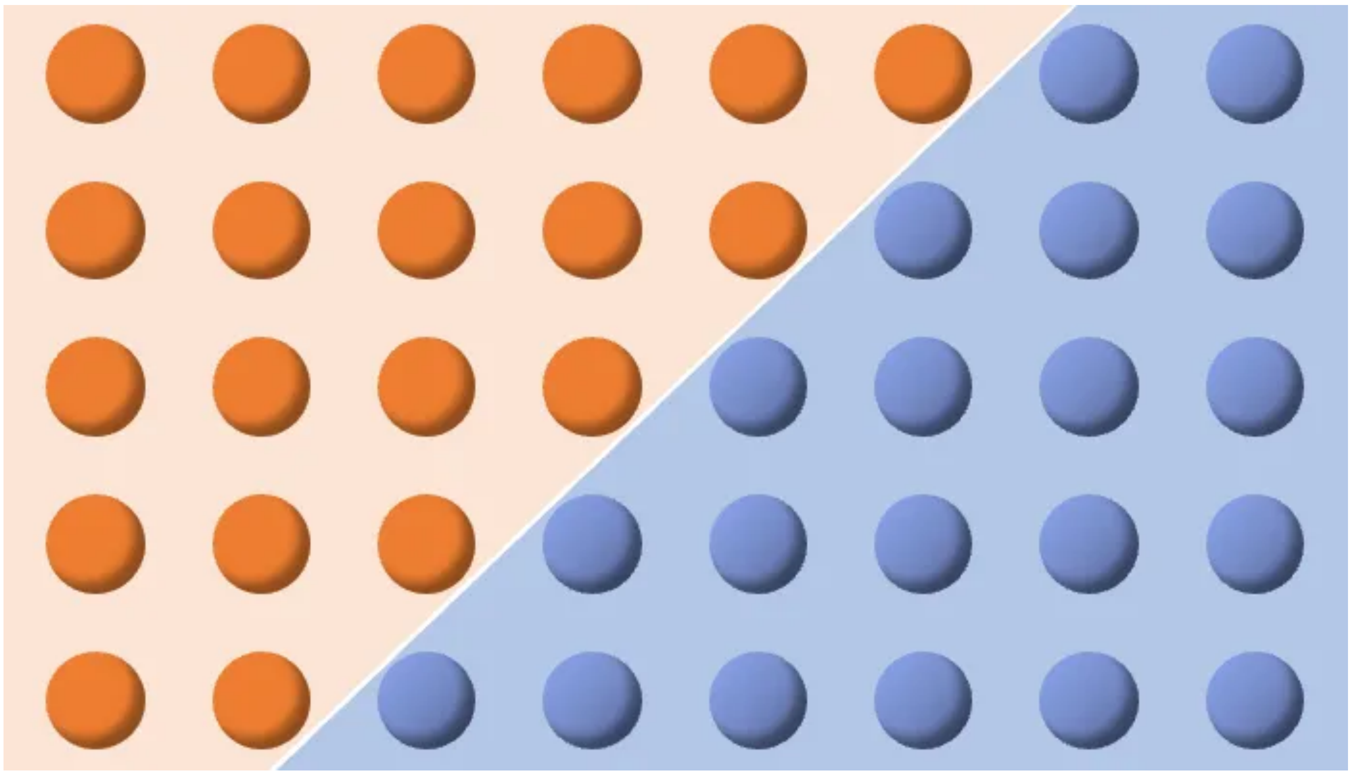


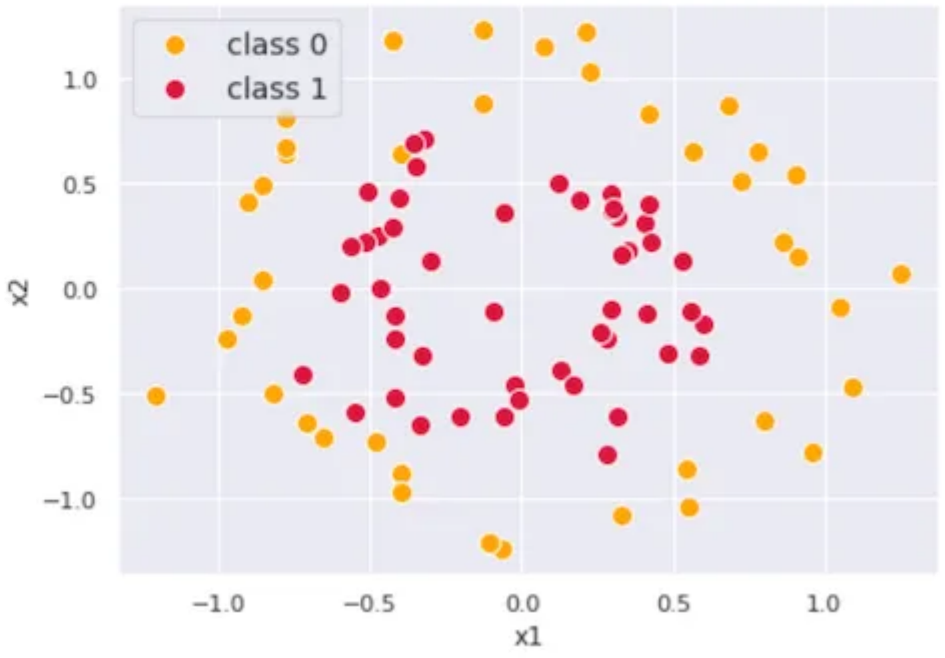
Image by author

In the Part 1 article, we learned the gradient boosting regression algorithm in its detail. As we reviewed in that post, the algorithm is flexible enough to deal with any loss functions as long as it is differentiable. That means if we just replace the loss function used for regression, specifically mean squared loss, with a loss function that deals with classification problems, we can perform classification without changing the algorithm itself. Even though the base algorithm is the same, there are still some differences that we want to know. In this post, we will dive into all the details of the classification algorithm.

Algorithm with an Example

Gradient boosting is one of the variants of ensemble methods where you create multiple weak models (they are often decision trees) and combine them to get better performance as a whole. In this section, we are building a gradient boosting classification model using very simple example data to intuitively understand how it works.

The picture below shows the sample data. It has the binary class y (0 and 1) and two features x_1 and x_2.

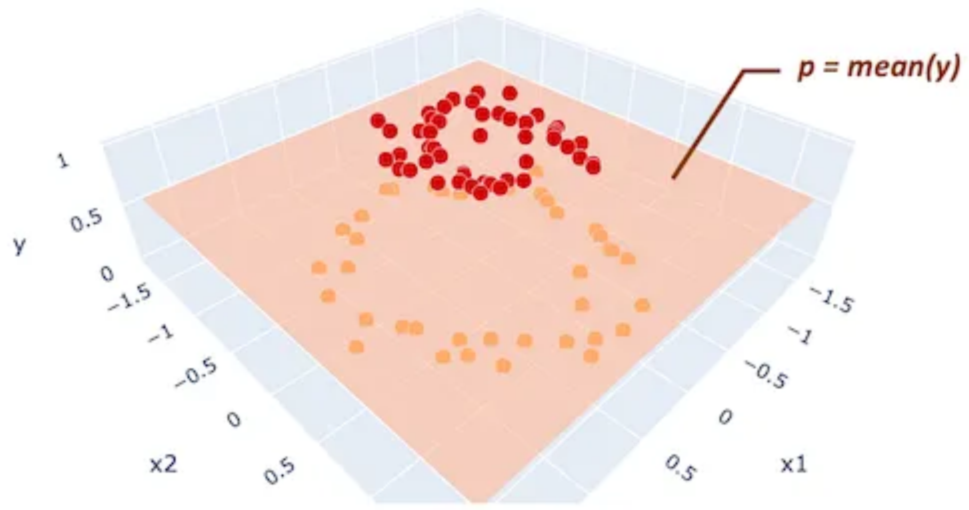


Sample for the classification problem (Image by author)

Our goal is to build a gradient boosting model that classifies those two classes. The first step is making a uniform prediction on a probability of class 1 (we will call it p) for all the data points. The most reasonable value for the uniform prediction might be the proportion of class 1 which is just a mean of y.

$$p = P(y = 1) = \bar{y}$$

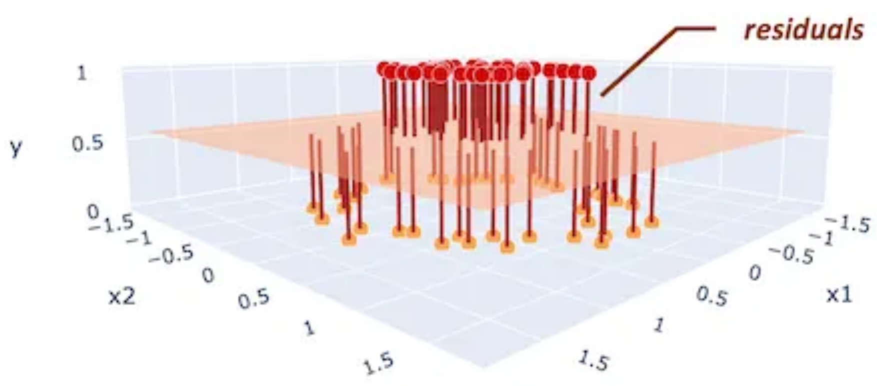
Here is a 3D representation of the data and the initial prediction. At this moment, the prediction is just a plane that has the uniform value $p = mean(y)$ on the y axis all the time.



Prediction plane (Image by author)

In our data, the mean of y is 0.56. As it is bigger than 0.5, everything is classified into class 1 with this initial prediction. Some of you might feel that this uniform value prediction does not make sense, but don't worry. We will improve our prediction as we add more weak models to it.

To improve our prediction quality, we might want to focus on the residuals (i.e. prediction error) from our initial prediction as that is what we want to minimize. The residuals are defined as $r_i = y_i - p$ (i represents the index of each data point). In the figure below, the residuals are shown as the brown lines that are the perpendicular lines from each data point to the prediction plane.

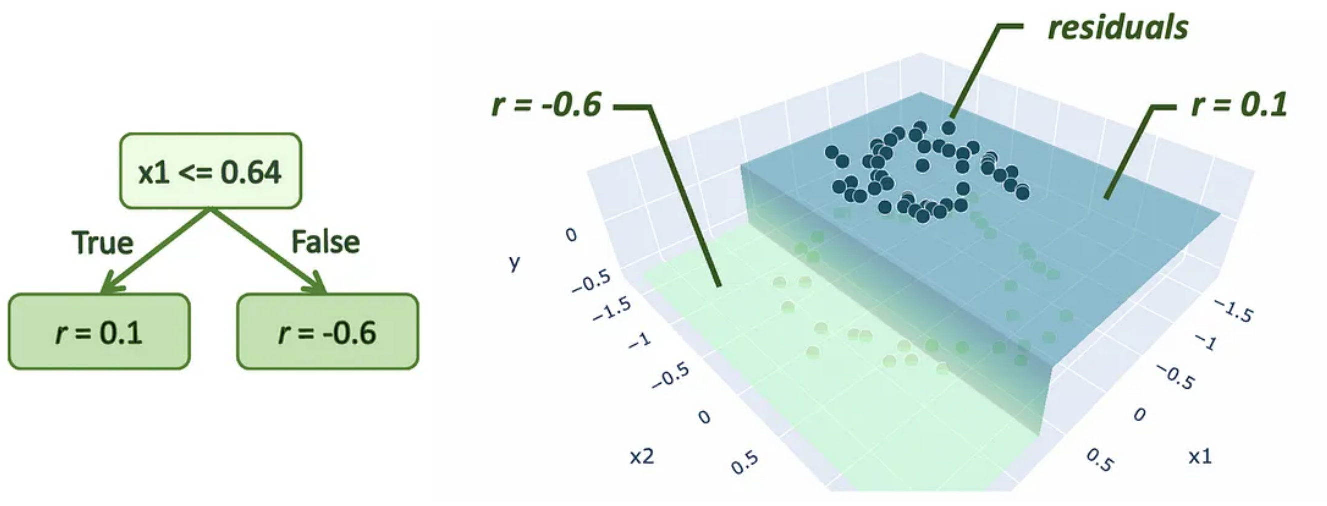


Residuals (Image by author)

To minimize these residuals, we are building a regression tree model with both x_1 and x_2 as its features and the residuals r as its target. If we can build a tree that finds some patterns between x and r, we can reduce the residuals from the initial prediction p by utilizing those found patterns.

To simplify the demonstration, we are building very simple trees each of that only has one split and two terminal nodes which is called "stump". Please note that gradient boosting trees usually have a little deeper trees such as ones with 8 to 32 terminal nodes.

Here we are creating the first tree predicting the residuals with two different values $r = \{0.1, -0.6\}$.



Created tree (Image by author)

You might now think we want to add these predicted values to our initial prediction p to reduce its residuals if you already read the post talking about the regression algorithm, but things are slightly different with classification. The values (we call it γ gamma) that we are adding to our initial prediction is computed in the following formula:

$$\underbrace{\gamma_j}_{\text{γ is computed for each terminal node j}} = \frac{\sum_{x_i \in R_j} (y_i - p)}{\underbrace{\sum_{x_i \in R_j}}_{\text{Aggregating for all the data points x_i that belongs to terminal node j}} p(1-p)}$$

$\Sigma x_i \in R_j$ means we are aggregating the values in the sigma Σ on all the sample x_i s that belong to the terminal node R_j . j represents the index of each terminal node. You might notice that the numerator of the fraction is the sum of the residuals in the terminal node j . We will go through all the calculations that give us this formula in the next section, but let's just use it to calculate γ for now. Below is the computed values of γ_1 and γ_2 .

$$\gamma_1 = \frac{\sum_{x_i \in R_1} (y_i - 0.56)}{\sum_{x_i \in R_1} 0.56 \cdot (1 - 0.56)} = 0.3$$

$$\gamma_2 = \frac{\sum_{x_i \in R_2} (y_i - 0.56)}{\sum_{x_i \in R_2} 0.56 \cdot (1 - 0.56)} = -2.2$$

This γ is not simply added to our initial prediction p . Instead, we are converting p into log-odds (we will call this log-odds converted value $F(x)$), then adding γ to it. For those who are not familiar with log-odds, it is defined below. You might have seen it used in logistic regression.

$$log(odds) = log\left(\frac{p}{1-p}\right)$$

log-odds

One more tweak on the prediction update is that γ is scaled down by **learning rate** ν, which ranges between 0 and 1, before it is added to the log-odds-converted prediction $F(x)$. This helps the model not to overfit the training data.

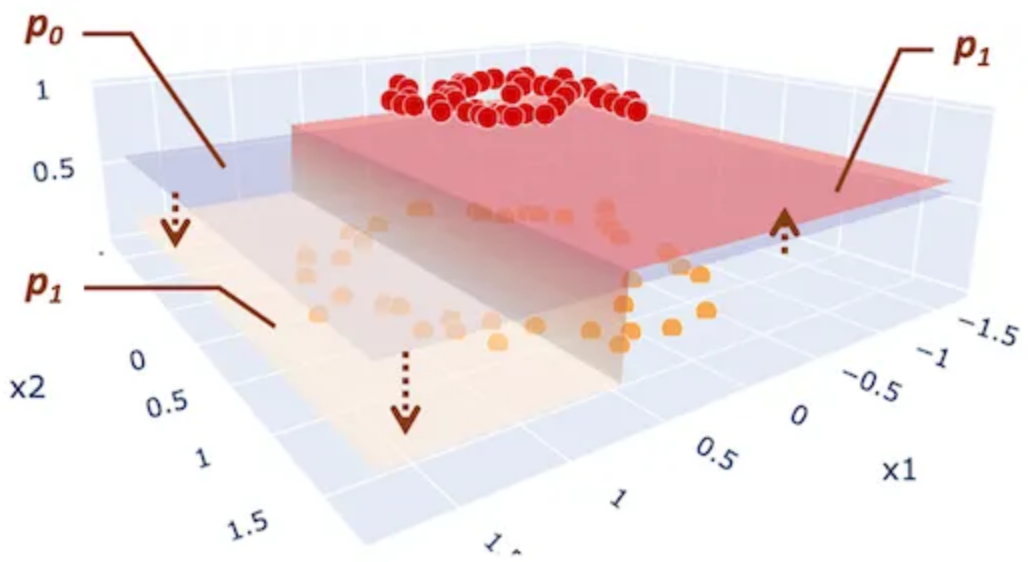
$$\underbrace{F_1(x)}_{\text{Updated prediction}} = \underbrace{F_0(x)}_{\text{Initial prediction}} + \underbrace{\nu}_{\text{Learning rate}} \cdot \gamma$$

In this example, we use a relatively big learning rate $\nu = 0.9$ to make the optimization process easier to understand, but it is usually supposed to be much smaller values such as 0.1.

By substituting actual values for the variables in the right side of the above equation, we get our updated prediction $F_1(x)$.

$$F_1(x) = \begin{cases} log\left(\frac{0.56}{1-0.56}\right) + 0.9 \cdot 0.3 = 0.5 & if\ x_1 \leq 0.64 \\ log\left(\frac{0.56}{1-0.56}\right) - 0.9 \cdot 2.2 = -1.7 & otherwise \end{cases}$$

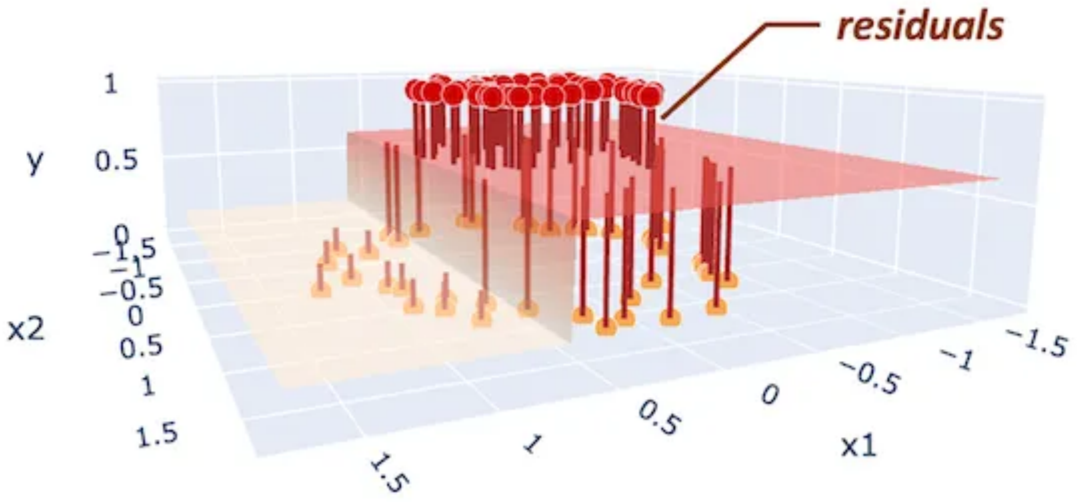
If we convert log-odds $F(x)$ back into the predicted probability $p(x)$ (we will cover how we can convert it in the next section), it looks like a stair-like object below.



Updated prediction plane (Image by author)

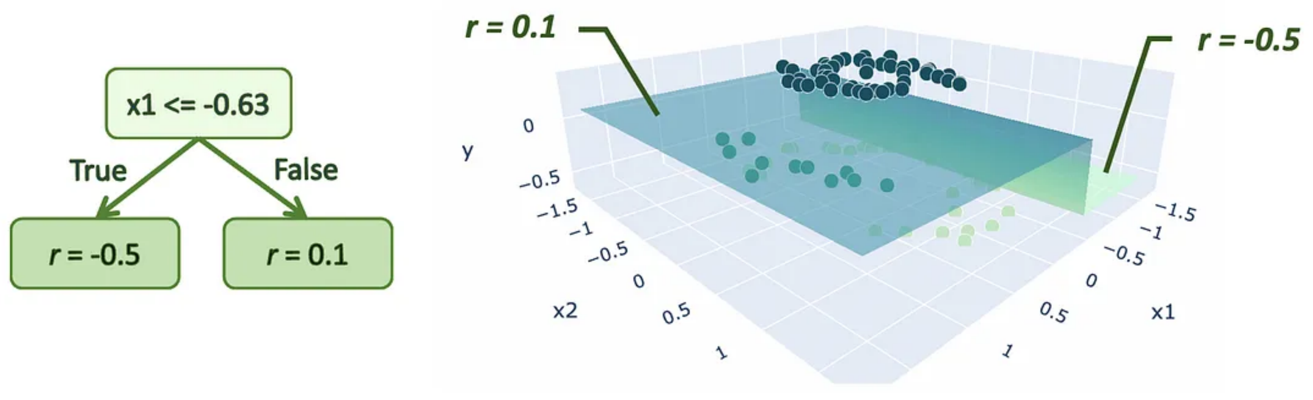
The purple-colored plane is the initial prediction p_0 and it is updated to the red and yellow plane p_1.

Now, the updated residuals r looks like this:



Updated residuals (Image by author)

In the next step, we are creating a regression tree again using the same x_1 and x_2 as the features and the updated residuals r as its target. Here is the created tree:



Created tree (Image by author)

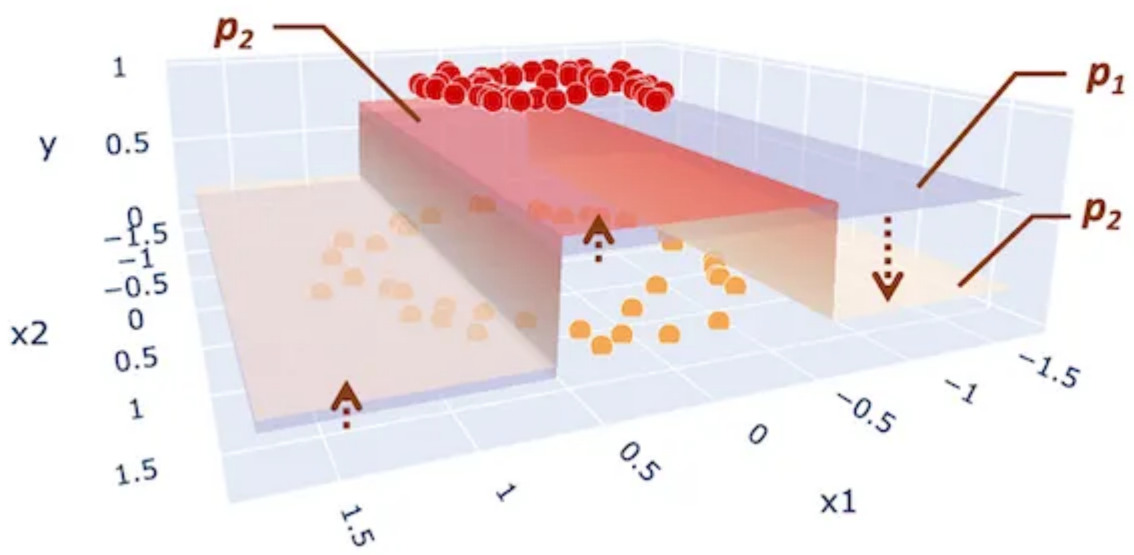
We apply the same formula to compute γ. The calculated γ along with the updated prediction $F_2(x)$ are as follows.

$$F_2(x) = \begin{cases} F_1(x) - \nu \cdot \boxed{2.3} = 0.5 - 0.9 \cdot 2.3 = -1.6 & \textit{if } x_1 \le -0.63 \\ F_1(x) + \nu \cdot \boxed{0.4} = 0.5 + 0.9 \cdot 0.4 = 0.9 & \textit{else if } -0.63 < x_1 \le 0.64 \\ F_1(x) + \nu \cdot \boxed{0.4} = -1.7 + 0.9 \cdot 0.4 = -1.3 & \textit{otherwise} \end{cases}$$

These are γ computed with this formula:

$$\gamma_j = \frac{\sum_{x_i \in R_j} (y_i - p)}{\sum_{x_i \in R_j} p(1-p)}$$

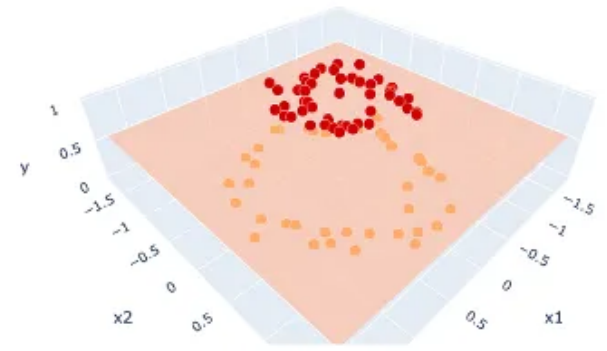
Again, if we convert log-odds $F_2(x)$ back into the predicted probability $p_2(x)$, it looks like something below.



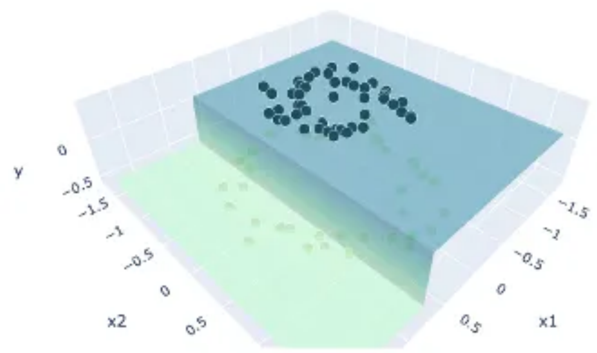
Updated prediction plane (Image by author)

We iterate these steps until the model prediction stops improving. The figures below show the optimization process from 0 to 4 iterations.

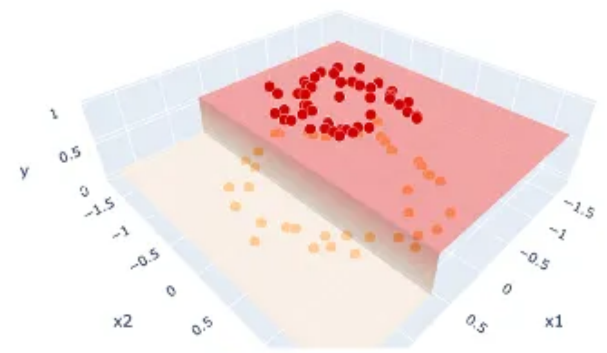
Predictions of iteration 0



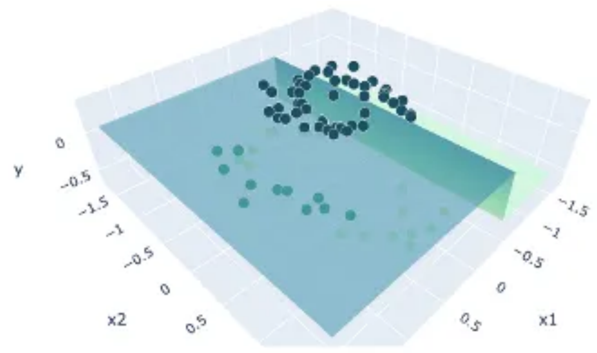
Residuals of iteration 1



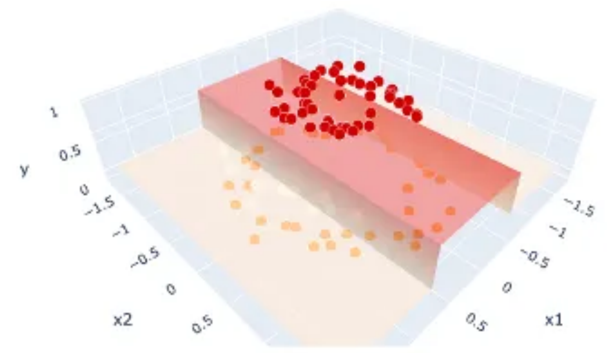
Predictions of iteration 1



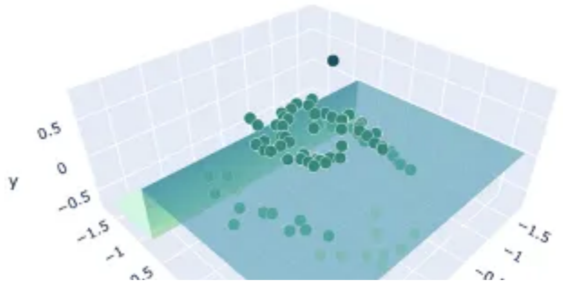
Residuals of iteration 2



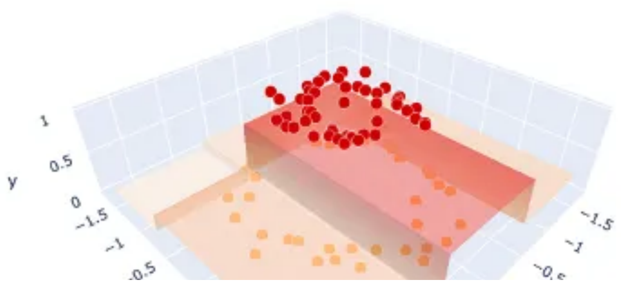
Predictions of iteration 2

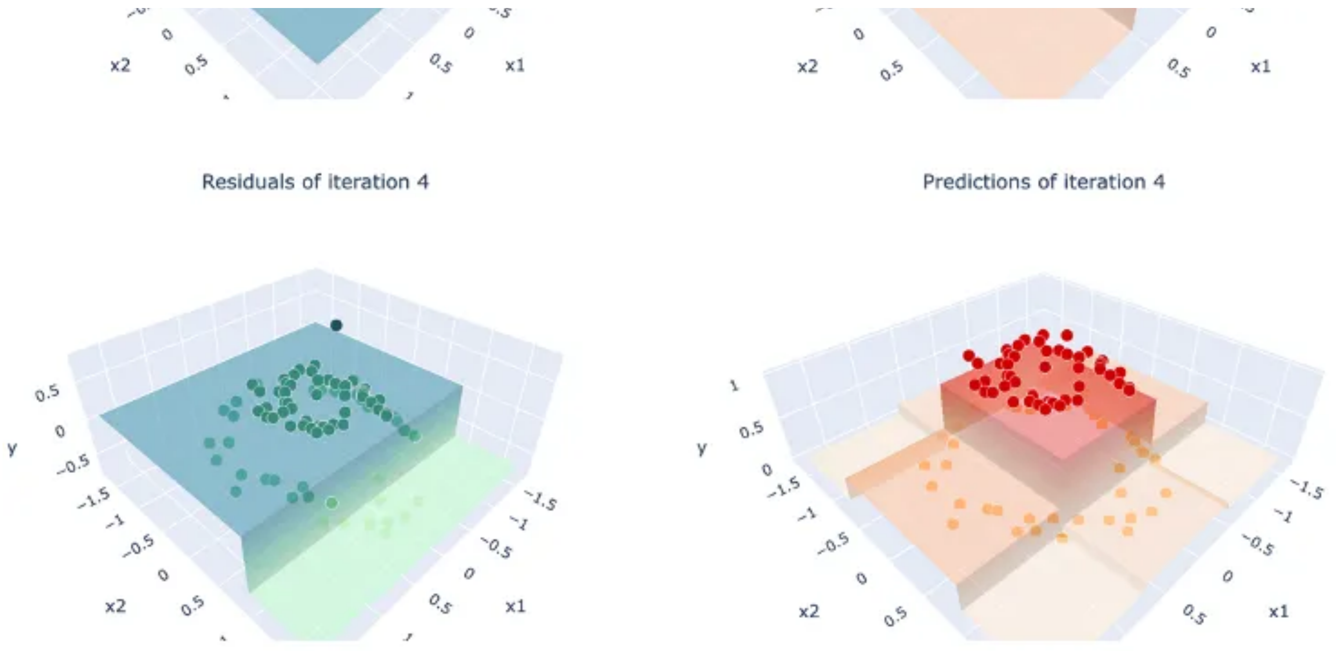


Residuals of iteration 3



Predictions of iteration 3

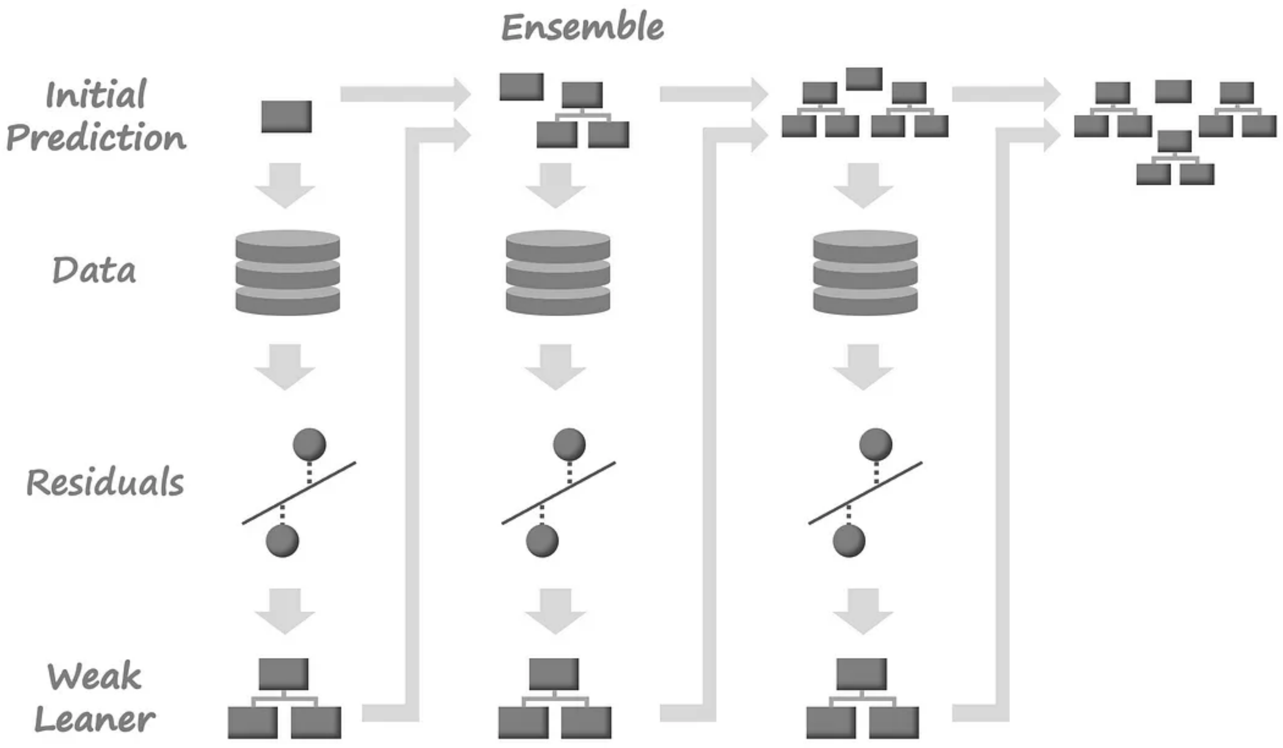




Prediction update through iterations (Image by author)

You can see the combined prediction $p(x)$ (red and yellow plane) is getting closer to our target y as we add more trees into the combined model. This is how gradient boosting works to predict complex targets by combining multiple weak models.

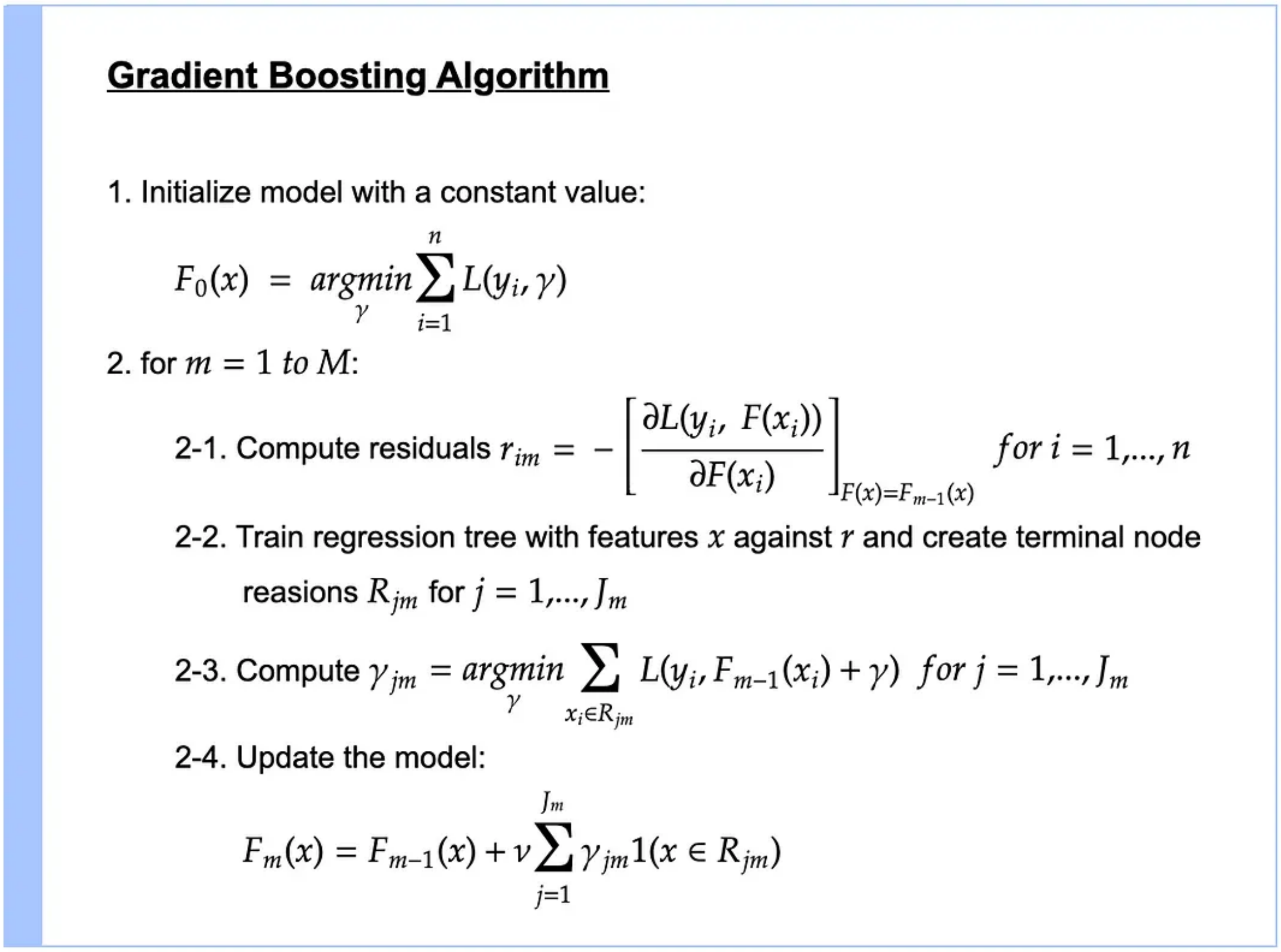
The image below summarizes the whole process of the algorithm.



Process of the algorithm (Image by Author)

Math

In this section, we are learning a more generalized algorithm by looking at its math details. Here is the whole algorithm in math formulas.



Gradient Boosting Algorithm

1. Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

2. for $m = 1 \; to \; M$:

2-1. Compute residuals $r_{im} = -\left[\frac{\partial L(y_i,\; F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad for \; i = 1,\ldots,n$

2-2. Train regression tree with features x against r and create terminal node reasions R_{jm} for $j = 1,\ldots,J_m$

2-3. Compute $\gamma_{jm} = \underset{\gamma}{argmin} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma) \quad for \; j = 1,\ldots,J_m$

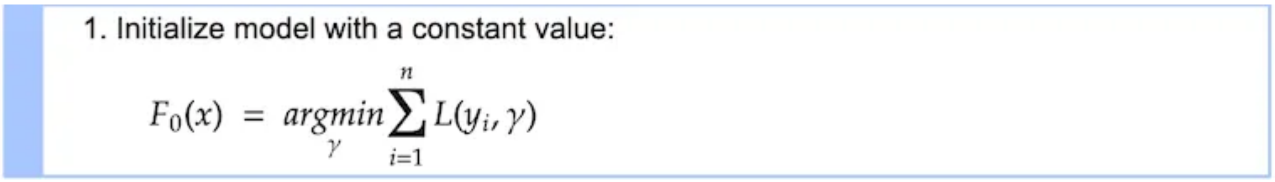
2-4. Update the model:

$$F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$$

Source: adapted from Wikipedia and Friedman's paper

Let's take a close look at it line by line.

Step 1



1. Initialize model with a constant value:

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma)$$

The first step is creating an initial constant prediction value F_0. L is the loss function and we are using log loss (or more generally called cross-entropy loss) for it.

$$L = -(y_i \cdot log(p) + (1 - y_i) \cdot log(1 - p))$$

Log Loss

y_i is our classification target and it is either 0 or 1. p is the predicted probability of class 1. You might see L taking different values depending on the target class y_i.

$$L = \begin{cases} -log(p) & if\ y_i = 1 \\ -log(1-p) & if\ y_i = 0 \end{cases}$$

As `-log(x)` is the decreasing function of x, the better the prediction (i.e. increasing p for $y_i=1$), the smaller loss we will have.

`argmin` means we are searching for the value γ *(gamma)* that minimizes $\Sigma L(y_i,\gamma)$. While it is more straightforward to assume γ is the predicted probability p, we assume γ is **log-odds** as it makes all the following computations easier. For those who forgot the log-odds definition which we review in the previous section, it is defined as `log(odds) = log(p/(1-p))`.

To be able to solve the `argmin` problem in terms of log-odds, we are transforming the loss function into the function of log-odds.

$$\begin{aligned}
L &= -\left(y_i \cdot log(p) + (1 - y_i) \cdot log(1 - p)\right) \\
&= -\left(y_i \cdot (\underline{log(p)} - \underline{log(1 - p)}) + log(1 - p)\right) \\
&= -\left(y_i \cdot log\left(\frac{p}{1 - p}\right) + log(1 - p)\right) \\
&= -\left(y_i \cdot log(odds) + log(1 - p)\right)
\end{aligned}$$

Now, we might want to replace p in the above equation with something that is expressed in terms of log-odds. By transforming the log-odds expression shown earlier, p can be represented by log-odds:

$$\begin{aligned}
&log\left(\frac{p}{1 - p}\right) = log(odds) \\
&\frac{p}{1 - p} = e^{log(odds)} \\
&p = (1 - p)e^{log(odds)} \\
&\left(1 + e^{log(odds)}\right)p = e^{log(odds)} \\
&p = \frac{e^{log(odds)}}{1 + e^{log(odds)}}
\end{aligned}$$

Then, we are substituting this value for p in the previous L equation and simplying it.

$$L = -\left(y_i \cdot log(odds) + log\left(1 - \overset{\text{p is replaced with this}}{\boxed{\frac{e^{log(odds)}}{1 + e^{log(odds)}}}}\right)\right)$$

$$= -\left(y_i \cdot log(odds) + \underline{log\left(\frac{1}{1 + e^{log(odds)}}\right)}\right)$$

$$= -\left(y_i \cdot log(odds) + \underbrace{log(1)}_{=0} - log\left(1 + e^{log(odds)}\right)\right)$$

$$= -\left(y_i \cdot log(odds) - log\left(1 + e^{log(odds)}\right)\right)$$

Now we are finding γ (please remember we are assuming it is log-odds) that minimizes ΣL. We are taking a derivative of ΣL with respect to log-odds.

$$\frac{\partial}{\partial log(odds)} \sum_{i=1}^{n} L = -\frac{\partial}{\partial log(odds)} \sum_{i=1}^{n} \left[y_i \cdot log(odds) - \underbrace{log\left(1 + e^{log(odds)}\right)}_{\substack{\text{Applying chain rule to} \\ \text{get derivative of this} \\ \frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)}}\right]$$

$$= -\sum_{i=1}^{n} y_i + n \underline{\frac{e^{log(odds)}}{1 + e^{log(odds)}}}$$

$$= -\sum_{i=1}^{n} y_i + np$$

In the equations above, we replaced the fraction containing log-odds with p to simplify the equation. Next, we are setting $\partial \Sigma L / \partial log(odds)$ equal to 0 and solving it for p.

$$-\sum_{i=1}^{n} y_i + np = 0$$

$$np = \sum_{i=1}^{n} y_i$$

$$p = \frac{1}{n}\sum_{i=1}^{n} y_i = \bar{y}$$

In this binary classification problem, `y` is either 0 or 1. So, the mean of `y` is actually the proportion of class 1. You might now see why we used `p = mean(y)` for our initial prediction.

As `γ` is log-odds instead of probability `p`, we are converting it into log-odds.

$$F_0(x) = \overset{*}{\gamma} = log\left(\frac{\bar{y}}{1-\bar{y}}\right)$$

Step2

2. for $m = 1$ *to* M:

The whole step2 processes from 2–1 to 2–4 are iterated `M` times. `M` denotes the number of trees we are creating and the small `m` represents the index of each tree.

Step 2–1

2-1. Compute residuals $r_{im} = -\left[\frac{\partial L(y_i,\ F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ $for\ i = 1,\ldots,n$

We are calculating residuals r_{im} by taking a derivative of the loss function with respect to the previous prediction F_{m-1} and multiplying it by −1. As you can see in the subscript index, r_{im} is computed for each single sample i. Some of you might be wondering why we are calling this r_{im} residuals. This value is actually **negative gradient** that gives us the directions (+/−) and the magnitude in which the loss function can be minimized. You will see why we are calling it residuals shortly. By the way, this technique where you use a gradient to minimize the loss on your model is very similar to gradient descent technique which is typically used to optimize neural networks. (In fact, they are slightly different from each other. If you are interested, please look at this article detailing that topic.)

Let's compute the residuals here. F_{m-1} in the equation means the prediction from the previous step. In this first iteration, it is F_0. As in the previous step, we are taking a derivative of L with respect to log-odds instead of p since our prediction F_m is log-odds. Below we are using L expressed by log-odds which we got in the previous step.

$$\begin{aligned} r_{im} &= -\frac{\partial}{\partial log(odds)} L \\ &= \frac{\partial}{\partial log(odds)} \left[y_i \cdot log(odds) - log\left(1 + e^{log(odds)}\right) \right] \\ &= y_i - \frac{e^{log(odds)}}{1 + e^{log(odds)}} \end{aligned}$$

In the previous step, we also got this equation:

$$p = \frac{e^{log(odds)}}{1 + e^{log(odds)}}$$

So, we can replace the second term in r_{im} equation with p.

$$r_{im} = y_i - p$$

You might now see why we call r residuals. This also gives us interesting insight that the negative gradient that provides us the direction and the magnitude to which the loss is minimized is actually just residuals.

Step 2–2

2-2. Train regression tree with features x against r and create terminal node reasions R_{jm} for $j = 1,\ldots,J_m$

j represents a terminal node (i.e. leave) in the tree, m denotes the tree index, and capital J means the total number of leaves.

Step 2–3

2-3. Compute $\gamma_{jm} = \underset{\gamma}{argmin} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma) \;\; for\ j = 1,\ldots,J_m$

We are searching for γ_{jm} that minimizes the loss function on each terminal node j. $\Sigma_{x_i \in R_{jm}} L$ means we are aggregating the loss on all the x_i s that belong to the terminal node R_{jm}. Let's plugin the loss function into the equation.

$$\begin{aligned}\gamma_{jm} &= \underset{\gamma}{argmin} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma) \\ &= \underset{\gamma}{argmin} \sum_{x_i \in R_{jm}} -\left(y(F_{m-1}(x_i) + \gamma) - log\left(1 + e^{F_{m-1}(x_i)+\gamma}\right)\right)\end{aligned}$$

Solving this equation for γ_{jm} is going to be very hard. To make it more easily solvable, we are making the approximation of L by using the second-order Taylor polynomial. A Taylor polynomial is a way to approximate any function as a polynomial with an infinite/finite number of terms. While we are not looking into its details here, you can look at this tutorial which explains the idea nicely if you are interested.

Here is the approximation of L using the second-order Taylor polynomial:

$$L(y,\ F_{m-1}(x_i)+\gamma) \approx L(y, F_{m-1}(x_i)) + \frac{\partial}{\partial F}L(y_i, F_{m-1}(x_i))\gamma + \frac{1}{2}\frac{\partial^2 L(y_i, F_{m-1}(x_i))}{\partial F^2}\gamma^2$$

We are substituting this approximation for L in the equation of γ_{jm}, then finding the value of γ_{jm} that makes the derivative of Σ(*) equal zero.

$$\frac{\partial}{\partial \gamma}\sum_{x_i \in R_{jm}}\left(\underbrace{L(y_i, F_{m-1}(x_i))}_{\text{Derivative of this term is zero}} + \frac{\partial}{\partial F}L(y_i, F_{m-1}(x_i))\gamma + \frac{1}{2}\frac{\partial^2 L(y_i, F_{m-1}(x_i))}{\partial F^2}\gamma^2\right) = 0$$

$$\sum_{x_i \in R_{jm}}\left(\frac{\partial}{\partial F}L(y_i, F_{m-1}(x_i)) + \frac{\partial^2 L(y_i, F_{m-1}(x_i))}{\partial F^2}\gamma\right) = 0$$

$$\sum_{x_i \in R_{jm}}\frac{\partial^2 L(y_i, F_{m-1}(x_i))}{\partial F^2}\gamma = -\sum_{x_i \in R_{jm}}\frac{\partial}{\partial F}L(y_i, F_{m-1}(x_i))$$

$$\gamma = \frac{-\sum_{x_i \in R_{jm}}\frac{\partial}{\partial F}L(y_i, F_{m-1}(x_i))}{\sum_{x_i \in R_{jm}}\frac{\partial^2}{\partial F^2}L(y_i, F_{m-1}(x_i))}$$

As we already computed $\partial L/\partial F$ in the previous step as below:

$$\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = -(y_i - p)$$

We are substituting this for $\partial L/\partial F$ in the γ equation.

Now we have gone through the whole steps. To get the best model performance, we want to iterate step 2 M times, which means adding M trees to the combined model. In reality, you might often want to add more than 100 trees to get the best model performance.

If you read my article for a regression algorithm, you might feel the math computation of the classification algorithm is much more complicated than the regression. While `argmin` and the derivative computation of the log-loss function is complicated, the underlining math algorithm which is shown at the beginning of this section is exactly the same. That is actually the elegance of the gradient boosting algorithm as exactly the same algorithm works for any loss function as long as it is differentiable. In fact, popular gradient boosting implementations such as XGBoost or LightGBM have a wide variety of loss functions, so you can choose whatever loss functions that suit your problem (see the various loss functions available in XGBoost or LightGBM).

$$
\begin{aligned}
\gamma &= \frac{\sum_{x_i \in R_{jm}} (y_i - p)}{\sum_{x_i \in R_{jm}} \frac{\partial}{\partial F}(-y_i + \boxed{p})} \\
&= \frac{\sum_{x_i \in R_{jm}} (y_i - p)}{\sum_{x_i \in R_{jm}} \frac{\partial}{\partial F}\left(-y_i + \boxed{\frac{e^{\log(odds)}}{1+e^{\log(odds)}}}\right)} \\
&= \frac{\sum_{x_i \in R_{jm}} (y_i - p)}{\sum_{x_i \in R_{jm}} \frac{\partial}{\partial F}\left(\underbrace{-y_i}_{\text{Derivative of y is zero}} + e^{\log(odds)}\left(1 + e^{\log(odds)}\right)^{-1}\right)} \\
&= \frac{\sum_{x_i \in R_{jm}} (y_i - p)}{\sum_{x_i \in R_{jm}} \boxed{\frac{\partial}{\partial F}\left(e^{\log(odds)}\left(1 + e^{\log(odds)}\right)^{-1}\right)}} \\
& \qquad \text{Apply product rule: } d(u \cdot v)/dx = du/dx \cdot v + u \cdot dv/dx \\
&= \frac{y_i - p}{\sum_{x_i \in R_{jm}} \boxed{\left(e^{\log(odds)}\left(1 + e^{\log(odds)}\right)^{-1} - e^{2\log(odds)}\left(1 + e^{\log(odds)}\right)^{-2}\right)}} \\
&= \frac{\sum_{x_i \in R_{jm}} (y_i - p)}{\sum_{x_i \in R_{jm}} \left(\underbrace{\frac{e^{\log(odds)}}{1+e^{\log(odds)}}}_{= p} - \left(\underbrace{\frac{e^{\log(odds)}}{1+e^{\log(odds)}}}_{= p}\right)^2\right)} \\
&= \frac{\sum_{x_i \in R_{jm}} (y_i - p)}{\sum_{x_i \in R_{jm}} \left(p - p^2\right)} \\
&= \frac{\sum_{x_i \in R_{jm}} (y_i - p)}{\sum_{x_i \in R_{jm}} p(1 - p)}
\end{aligned}
$$

Finally, we got this simplified equation for the value of γ_{jm} which we used in the previous section.

Step 2–4

2-4. Update the model:

$$F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} 1(x \in R_{jm})$$

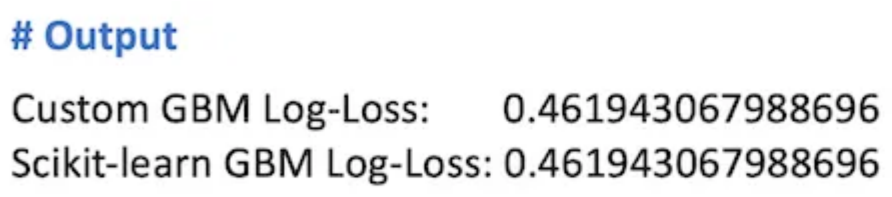
In the final step, we are updating the prediction of the combined model F_m. $\gamma_{jm}1(x \in R_{jm})$ means that we pick the value γ_{jm} if a given x falls in a terminal node R_{jm}. As all the terminal nodes are exclusive, any given single x falls into only a single terminal node and corresponding γ_{jm} is added to the previous prediction F_{m-1} and then it makes the updated prediction F_m.

As mentioned in the previous section, ν is learning rate ranging between 0 and 1 which controls the degree of contribution of the additional tree prediction γ to the combined prediction F_m. A smaller learning rate reduces the effect of the additional tree prediction, but it basically also reduces the chance of the model overfitting to the training data.

in

Please note that all the trained trees are stored in `self.trees` list object and it is retrieved when we make predictions with `predict_proba` method.

Next, we are checking if our `CustomGradientBoostingClassifier` performs as the same as `GradientBoostingClassifier` from scikit-learn by looking at their log-loss on our data.



```
# Output
Custom GBM Log-Loss:       0.461943067988696
Scikit-learn GBM Log-Loss: 0.461943067988696
```

As you can see in the output above, both models have exactly the same log-loss.

Recommended Resources

In this blog post, we have reviewed all the details of the gradient boosting classification algorithm. If you are also interested in the regression algorithm, please look at the Part 1 article.

All You Need to Know about Gradient Boosting Algorithm − Part 1. Regression

Algorithm explained with an example, math, and code

towardsdatascience.com

There are also some other great resources if you want further details of the algorithm:

- **StatQuest, Gradient Boost Part3 and Part 4**
  These are the YouTube videos explaining the gradient boosting classification algorithm with great visuals in a beginner-friendly way.

- **Terence Parr and Jeremy Howard, How to explain gradient boosting**
  While this article focuses on gradient boosting regression instead of classification, it nicely explains every detail of the algorithm.

- **Jerome Friedman, Greedy Function Approximation: A Gradient Boosting Machine**
  This is the original paper from Friedman. While it is a little hard to understand, it surely shows the flexibility of the algorithm where he shows a generalized algorithm that can deal with any type of problem having a differentiable loss

function.

You can also look at the full Python code in the Google Colab link or the Github link below.

Google Colaboratory

Gradient Boosting Algorithm — Part2. Classification

Gradient Boosting Algorithm — Classification at tomonori-masui/gradient-boosting

github.com

References

- Jerome Friedman, Greedy Function Approximation: A Gradient Boosting Machine

- Terence Parr and Jeremy Howard, How to explain gradient boosting

- Matt Bowers, How to Build a Gradient Boosting Machine from Scratch

- Wikipedia, Gradient boosting