A Brief Introduction to Gradient Boosting

Abhishek Allamsetty

5/23/2018

1 Background

We see many Kaggle winners and high praise over utilizing a specific model known as a Gradient Boosting Model (GBM) algorithm. Although GBM is being used everywhere, many users treat it as a black box and run the models with pre-built libraries. That usage is fine and functional, but I think we can benefit a great deal by taking a deeper look into the model and really understanding the ways in which it works. The purpose of this lecture is to help simplify a supposedly complex algorithm, and help us all understand this powerful tool just a little bit better.

2 Motivation

We'll start with a basic example. We want to predict a person's age based on whether they play video games, enjoy to garden, and whether or not they like to wear hats. What we want to do is minimize our squared error. We have the following nine training samples to build our model:

3 Introduction

The main causes of difference in actual and predicted values when trying to predict the target variable utilizing any ML technique are: **noise, variance, and bias**. A way to help reduce all of the factors but noise would be through ensemble.

An ensemble is a collection of predictors that are taken together, or a mean of all of the predictions, to gather a final prediciton. We use ensembles because there are a lot of predictors trying to predict some target variable that'll performa a better job than a single predictor alone. We can classify ensembling into:

Intuitively, we might expect that the people who like to garder are probably older, the people who like video games are probably younger, and *LikesHats* is probably just some random noise.

PersonID	**Age**	**LikesGardening**	**PlaysVideoGames**	**LikesHats**
1	13	False	True	True
2	14	False	True	False
3	15	False	True	False
4	25	True	True	True
5	35	False	True	True
6	49	True	False	False
7	68	True	True	True
8	71	True	False	False
9	73	True	False	True

Feature	False	True
LikesGardening	{13, 14, 15, 35}	{25, 49, 68, 71, 73}
PlaysVideoGames	{49, 71, 73}	{13, 14, 15, 25, 35, 68}
LikesHats	{14, 15, 49, 71}	{13, 25, 35, 68, 73}

Let's take a quick look at our data to check these assumptions:

Now, we can model our data with a regression tree. We'll start off with the requirement that the terminal nodes have at least three samples. So that means our regression tree will make its first and last split on *LikesGardening*.

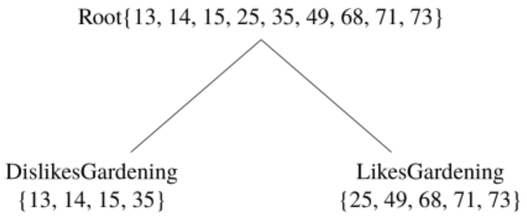

Root{13, 14, 15, 25, 35, 49, 68, 71, 73}
DislikesGardening {13, 14, 15, 35}
LikesGardening {25, 49, 68, 71, 73}

Figure 1: Tree 1

This is nice, but it's missing some important information from our *LikesVideoGames* feature. Let's try allowing terminal nodes to have 2 samples.

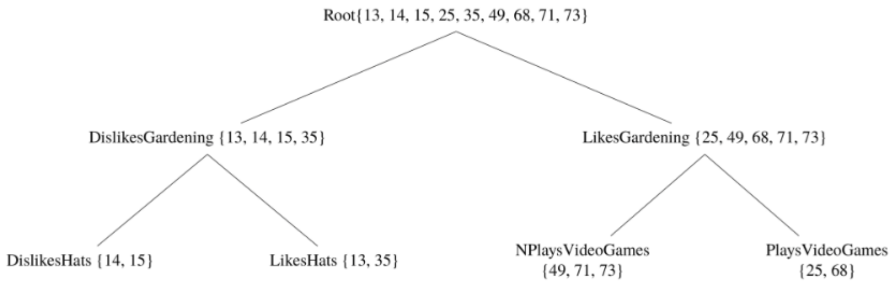
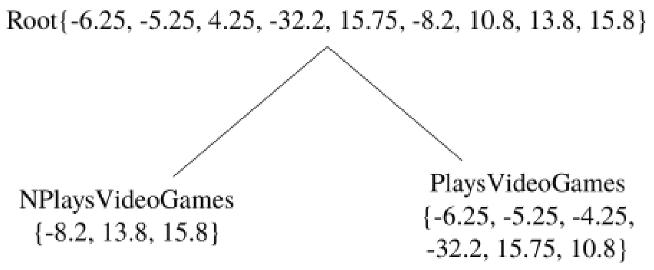

Root{13, 14, 15, 25, 35, 49, 68, 71, 73}
DislikesGardening {13, 14, 15, 35}
LikesGardening {25, 49, 68, 71, 73}
DislikesHats {14, 15}
LikesHats {13, 35}
NPlaysVideoGames {49, 71, 73}
PlaysVideoGames {25, 68}

Figure 2: Overfit Tree

Here, we end up attaining some information about *PlaysVideoGames*, but we also pick up random information from *LikesHats*, this is a good indication that we're overfitting and our tree is splitting random noise.

This is the big drawback when it comes to using a regression tree, **it fails to include predictive power from multiple, overlapping regions of the feature space**. Let's say we measure the training errors from our first tree:

Now,we can fit a second regression tree to the residuals of the first tree.


Root{-6.25, -5.25, 4.25, -32.2, 15.75, -8.2, 10.8, 13.8, 15.8}
NPlaysVideoGames {-8.2, 13.8, 15.8}
PlaysVideoGames {-6.25, -5.25, -4.25, -32.2, 15.75, 10.8}

PersonID	Age	Tree 1 Prediction	Tree 1 Residual
1	13	19.25	-6.25
2	14	19.25	-5.25
3	15	19.25	-4.25
4	25	57.2	-32.2
5	35	19.25	15.75
6	49	57.2	-8.2
7	68	57.2	10.8
8	71	57.2	13.8
9	73	57.2	15.8

You can notice that this tree doesn't include *LikesHats* even though our overfitted regression tree above did. The reason is because this regression tree is able to consider *LikesHats* and *PlaysVideoGames* with respect to all of the training samples, contrary to our overfit regression tree, which only considered each feature within a small region of the space we input, which allows for random noise to select *LikesHats* as a splitting feature.

Now, lets improve our initial predictions from our first tree by adding the "error-correcting" predictions from this tree.

PersonID	Age	Tree1 Prediction	Tree1 Residual	Tree2 Prediction	Combined Prediction	Final Residual
1	13	19.25	-6.25	-3.567	15.68	2.683
2	14	19.25	-5.25	-3.567	15.68	1.683
3	15	19.25	-4.25	-3.567	15.68	0.6833
4	25	57.2	-32.2	-3.567	53.68	28.63
5	35	19.25	15.75	-3.567	15.68	-19.32
6	49	57.2	-8.2	7.133	64.33	15.33
7	68	57.2	10.8	-3.567	53.63	-14.37
8	71	57.2	13.8	7.133	64.33	-6.667
9	73	57.2	15.8	7.133	64.33	-8.667

4 Gradient Boosting: 1

Inspired by the thought process above, we make our first naive formalization of gradient boosting. What we basically need to do is:

1. Fit a model to the data, $F_1(x) = y$
2. Fit a model to the residuals, $h_1(x) = y - F_1(x)$
3. Create a new model, $F_2(x) = F_1(x) + h_1(x)$

It isn't hard to see how we can generalize this idea by adding more models that correct the errors of the previous model, specifically we can allude to:

$$F(x) = F_1(x) \mapsto F_2(x) = F_1(x) + h_1(x) \ldots \mapsto F_M(x) = F_{M-1}(x) + h_{M-1}(x)$$

where $F_1(x)$ is an initial model to fit y

Since we initialize our procedure by fitting $F_1(x)$, our task at each step is to find $h_m(x) = y - F_m(x)$.

Tree 1 SSE	Combined SSE
1994	1765

PersonID	**Age**	F_0	**Residual0**
1	13	35	-22
2	14	35	-21
3	15	35	-21
4	25	35	-10
5	35	35	0
6	49	35	14
7	68	35	33
8	71	35	36
9	73	35	38

We must notice something. h_m is just a model. We have not defined anything that requires it to be a tree-based model. This is one of the concepts that work gradient boosting to an advantage. It acts as a framework to iteratively improve any weak learner. What this allows us to claim is that in theory, a well written gradient boosting module would allow you to plug in various types of weak learners, however most often than not, h_m is almost always a tree based learner, so we can interpret h_m as a regression tree like the one we started out with as our example.

5 Gradient Boosting: 2

Now what we want to do is tweak our model to conform to most gradient boosting implementations. We want to initialize our model with a single prediction value, and since our main task at the moment is to minimize squared error, we'll initialize F with the mean of the trading target values.

$$F_0(x) = arg_\gamma min \sum_{i=1}^{n} L(y_i, \gamma) = arg_\gamma min \sum_{i=1}^{n} (\gamma - y_i)^2 = \frac{1}{n} \sum_{i=1}^{n} y_i$$

We are now able to recursively define each subsequent F_m, just like we did before.

$$F_{m+1}(x) = F_m(x) + h_m(x) = y \text{ for } m \geq 0$$

Where h_m comes from a class of base learning regression trees

6 Gradient Boosting: 3

Up until now, we've been building a model that minimizes squared error, but what if we want to minimize absolute error? We are going to use an interesting method to do this. To determine F_0, we start by choosing a minimizer for absolute error. This will be $median(y) = 35$. Now, we can measure the residuals, $y - F_0$.

Let's consider the first and fourth training samples. They have F_0 residuals of -22 and -10 respectively. Now suppose we can make each prediction one unit closer to its target.

Our respective squared error reductions would be 43 and 19, while the respective absolute error reductions would be 1 and 1. So using a regression tree, which intrinsically minimizes squared error, will be focused on reducing the residual of the first training sample. If we want to minimize absolute error, moving each prediction one unit closer to the target produces an equal reduction in the cost function.

7 Gradient Descent

Let us establish this idea of the minimization of absolute error through exploring the concept of gradient descent. For example,

$$L(x_1, x_2) = \frac{1}{2}(x_1 - 15)^2 + \frac{1}{2}(x_2 - 25)^2$$

What we want to is find a pair (x_1, x_2) that minimizes L. Notice, we can interpret this function as calculating the squared error for hte two data points, 15 and 25 given two values for prediction, x_1 and x_2. Although we are able to directly minimize this function, gradient descent will let us minimize much more complicated loss functions that aren't able to directly minimize.

Steps to Initialization:

- Number of Iteration Steps: $M = 100$
- Starting Point: $s^0 = (0, 0)$
- Step Size $\gamma = 0.1$

For iteration $m = 1$ to M:

- Calculate the gradient of L at the point s^{m-1}
- "Step" in the direction of greatest descent (the negative gradient), with step size γ. Or, $s^m = s^{m-1} - \gamma \nabla L(s^{m-1})$

If γ is small, and M is big enough, s^M will be the location of the minimum value of L.

8 How we can use Gradient Descent

Now we can begin to use the concept of gradient descent in our GBM. The function we ultimately want to minimize is L with a starting point at $F_0(x)$. For iteration, $m = 1$, we compute the gradient of L with respect to $F_0(x)$. We then fit a weak learner to the gradient components.

In the case of a regression tree, the leaf nodes produce an average gradient among samples with similar features. For each leaf, we step in the direction of the average gradient, using line search to determine the magnitude of each step. The result is F_1, we iterate this process until we have F_M.

What we have just done is to modify our gradient boosting algorithm so that it works with any differentiable loss function. Let us reformulate our GBM with the ideas we have gathered.

We initialize our model with a constant value:

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

For m=1 to M: We want to compute the pseudo residuals: $r_pseudo = -[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}]_{F(x)=F_{m-1}(x)}$ for $i = 1, \ldots, n$.

We want to fit our base learner, $h_m(x)$ to our pseudo residuals, then compute our step magnitude multiplier γ_m

Finally, we update:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

Lets see how our current gradient boosting method affects the results in our sample problem for both squared and absolute error.

Age	F_0	$\textbf{PseudoResidual}_0$	h_0	γ_0	F_1	$\textbf{PseudoResidual}_1$	h_1	γ_1	F_2
13	40.33	-27.33	-21.08	1	19.25	-6.25	-3.567	1	15.68
14	40.33	-26.33	-21.08	1	19.25	-5.25	-3.567	1	15.68
15	40.33	-25.33	-21.08	1	19.25	-4.25	-3.567	1	15.68
25	40.33	-15.33	16.87	1	57.2	-32.2	-3.567	1	53.63
35	40.33	-5.333	-21.08	1	19.25	15.75	-3.567	1	15.68
49	40.33	8.667	16.87	1	57.2	-8.2	7.133	1	64.33
68	40.33	27.67	16.87	1	57.2	10.8	-3.567	1	53.63
71	40.33	30.67	16.87	1	57.2	13.8	7.133	1	64.33
73	40.33	32.67	16.87	1	57.2	15.8	7.133	1	64.33

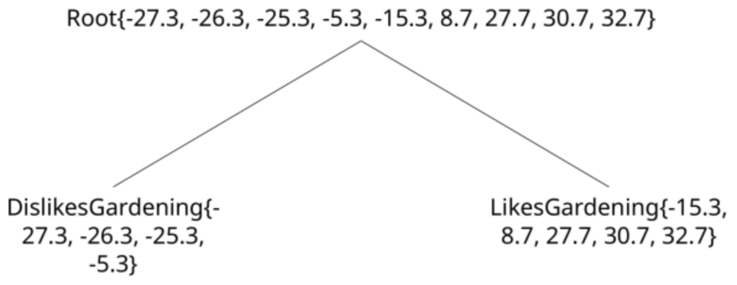

Root{-27.3, -26.3, -25.3, -5.3, -15.3, 8.7, 27.7, 30.7, 32.7}
DislikesGardening{-27.3, -26.3, -25.3, -5.3}
LikesGardening{-15.3, 8.7, 27.7, 30.7, 32.7}

Figure 3: h_0

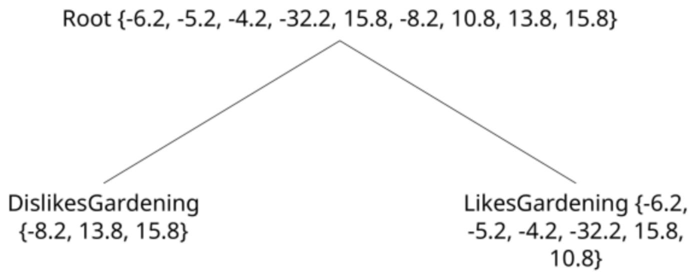

Root {-6.2, -5.2, -4.2, -32.2, 15.8, -8.2, 10.8, 13.8, 15.8}
DislikesGardening {-8.2, 13.8, 15.8}
LikesGardening {-6.2, -5.2, -4.2, -32.2, 15.8, 10.8}

Figure 4: h_1

Age	F_0	$\textbf{PseudoResidual}_0$	h_0	γ_0	F_1	$\textbf{PseudoResidual}_1$	h_1	γ_1	F_2
13	35	-1	-1	20.5	14.5	-1	-0.3333	0.75	14.25
14	35	-1	-1	20.5	14.5	-1	-0.3333	0.75	14.25
15	35	-1	-1	20.5	14.5	1	-0.3333	0.75	14.25
25	35	-1	0.6	55	68	-1	-0.3333	0.75	67.75
35	35	-1	-1	20.5	14.5	1	-0.3333	0.75	14.25
49	35	1	0.6	55	68	-1	0.3333	9	71
68	35	1	0.6	55	68	-1	-0.3333	0.75	67.75
71	35	1	0.6	55	68	1	0.3333	9	71
73	35	1	0.6	55	68	1	0.3333	9	71

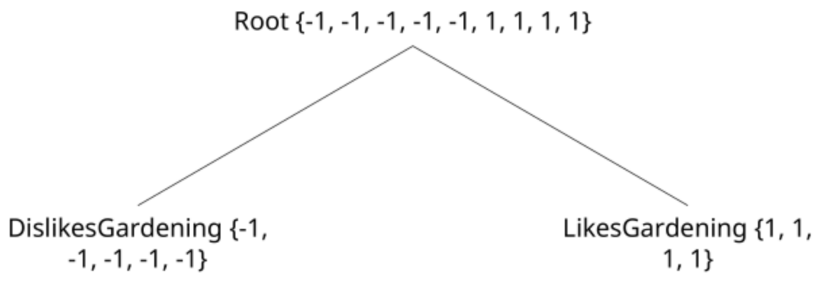

Root {-1, -1, -1, -1, -1, 1, 1, 1, 1}
DislikesGardening {-1, -1, -1, -1, -1}
LikesGardening {1, 1, 1, 1}

Figure 5: h_0

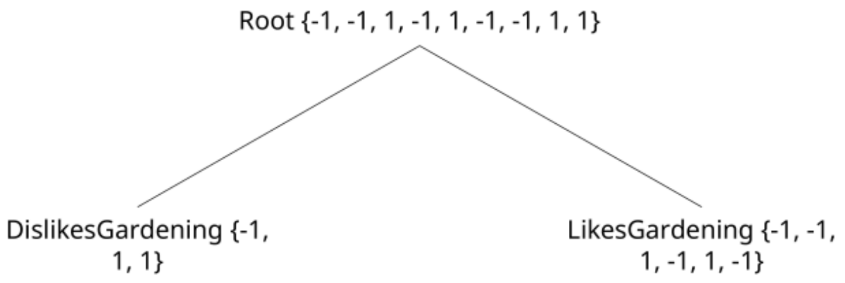

Root {-1, -1, 1, -1, 1, -1, -1, 1, 1}
DislikesGardening {-1, 1, 1}
LikesGardening {-1, -1, 1, -1, 1, -1}

Figure 6: h_1

9 Final Concepts

There is another concept known as shrinkage that rounds out the concept of gradient boosting. Basically, for each gradient step, the step magnitude is multiplied by a factor between 0 and 1 known as the learning rate. So each gradient step is shrunken by some factor. The general consensus on this practice in addition to gradient boosting is that it causes sample predictions to more slowly converge towards observed values. As the slow convergence occurs, the samples get closer to their target and end up being grouped together into larger and larger leaves, resulting in a natural regularization effect.

Finally, row sampling and column sampling, or the ability for a GBM to sample the data rows and columns before each boosting iteration has proven to be effective. What makes this sampling method particularly effective as it results in more different tree splits, which means more overall distribution of information in the model.

In conclusion, gradient boosting proves to be incredibly effective in practice. The most popular implementation, known as XGBoost, is used in a number of winning Kaggle solutions. XGBoost utilizes several methods to further increase speed and accuracy of traditional gradient boosting, namely 2^{nd} order gradient descent. Some new competitors, such as LightGBM by Microsoft are also gaining traction. Gradient boosting can be utilized as a classification and ranking model as well, as long as there exists a differentiable loss function for the algorithm to minimize, we are able to use gradient boosting.