**Gradient Boost (Regression)**

Now we’ll look at another method based on Decision Trees (or adapted to). Like with Random Forest and Adaboost, it replaces a single, probably overfit, tree, with an ensemble of underfit trees. And it uses the ensemble to make a prediction on the data. We’ll be looking at it in the context of Regression – i.e., trying to predict numerical values. So we’d have a table of values and such,

|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **Weight (kg)** |
| 1.6 | Blue | Male | 88 |
| 1.6 | Green | Female | 76 |
| 1.5 | Blue | Female | 56 |
| 1.8 | Red | Male | 73 |
| 1.5 | Green | Male | 77 |
| 1.4 | Blue | Female | 57 |

And we want to come up with an ensemble of deciision trees to estimate the outcomes Y = Weight. The algorithm basically works like this (picture adapted from Datacamp video). We start by calculating the average of all data points, and call this our zeroth order prediction f(0). Then we subtract this from our data y – f(0), and train a decision tree on this to get the predictions for the difference Δf(1). Once trained, we construct predictions for the data points themselves f(1) = f(0) + αΔf(1). α is called the learning rate (between 0 and 1), and governs how much we endorse the prediction for the differences. Then we construct the new residuals y – f(1) and train a tree on *that*, and the process starts over. Eventually we come to the end – the Nth tree – and its output predictions for the data points.

A diagram of a graph

Description automatically generated

We already discussed the basic concepts of SSE, predicted outcome, and Information Gain, when we covered normal regression trees. So I’ll just jump into how we construct the tree. Those concepts are a little modified from before though, as well as how we construct the tree. I think it’s easiest to just to it all together.

**Constructing the Decision Tree**

So before, we started off with all of our data in a root node with SSEm(Y) given by:



and we looked to make a classification of the data into leaves Lℓ so that SSEm(Y|L) was given by:



which maximized the information gain:



Instead, now we’re going to do this kind of incrementally. And from a slightly more general, enlightening, perspective. Generally speaking, let’s say we have a bunch of predictions, fi, for our data yi. And let’s write SSEm(Y|f) as:



where fi is left unknown for now. We’ll want to find incremental predictions, fi, which minimize the sum of the squared errors.

***0th Tree***

Before using any trees, we start with a simple classification whereby we predict everything to be a single constant parameter, so fi = f(0). So our decision tree is just a bare leaf. And presently, our SSEm is:



If we minimize SSEm(Y|f(0)) w/r to f(0), we’ll see that:



So the single parameter that best predicts all the data is simply the average of all the data, as is intuitive.



Now with this in hand, we’ll proceed to the….

***1st Tree***

So we’d like to construct a tree to improve this classification. So we’ll look for an increment, Δfi(1), that we can add to our present prediction, f(0), to get a new prediction: fi(1) = f(0) + Δfi(1).



Now we’ll look to classify these increments Δf(1)i into some assortment of leaves, L = {L1, L2, …}. But before we do, let’s expand this out in powers of Δfi(1), out to second order. Of course that’s as high as it goes anyway.



The first term, being a constant regardless of how we classify data into leaves, is a constant that doesn’t matter. So we can restrict our attention to:



where yLℓ,j denotes the jth outcome in the ℓth leaf. And similarly ΔfLℓ,j(1) is the jth increment in the ℓth leaf. But given some ℓ, ΔfLℓ,j(1) = ΔfLℓ,j´(1) for any two j and j´, as we said. So we don’t need to keep the j index here, and we just write: ΔfLℓ(1). When we minimize, we’ll have:



So we have:



This says that the parameters that minimize the loss are the ‘averages’ of the things (increments in this case) in each leaf. Plugging this back into ΔSSEm(1)(Y|Δf(1)→L), we have:



So we have:



Note ΔfLℓ(1) is the Lℓ leaf’s *prediction* for the increment. So diagrammatically, we’ll start our decision tree by putting all the data in the root node. And then we’re looking to categorize the data into categories/leaves.

Diagram, schematic

Description automatically generated

[note ΔfR(1) = 0] And we want to find leaves, Lℓ, which maximize the information gain:



In this diagam, L = {A1, B1, B2}. Since ΔSSEm­(1)(Y|Δf(1) → R) is fixed w/r to this tree, we basically just need to extremize the second term. We will typically use a greedy algorithm to find these leaves. It won’t be guaranteed to produce the correct result, but I don’t think we actually care, because we won’t usually fit a tree precisely to the data anyway (that’d be overfitting). The way we use the greedy algorithm is as follows. Say we’re at some node, A2 (and this could be the root of course). In this node are all rows with value A = A2. These rows have outcomes Yj. We can calculate the ΔSSE of this node ΔSSEm(1)(Y|Δf(1) → A2) = nA2-1 [Σj=1n\_A2 (yA2\_j – f(0))]2 as shown. And the prediction of this node would be fA2 = (1/nA2)Σj=1n\_A2 (yA2\_j – f(0)), i.e., the average value of Y deviation in this subset A2. To make further progress classifying the data, we can split Yj’s into two groups according to the B values: B1, B2. And we can calculate the ΔSSEm(1)(Y|Δf(1) → B1) of the group in B1, and the entropy ΔSSEm(1)(Y|Δf(1) → B2) of the group in B2 [note that I’m notationally eliding the fact that the Y’s in B1 or B2 must also be in A2 – just makes notation too cumbersome]. Summed together this gives the total ΔSSEm(1)(Y|Δf(1) → B) = ΔSSEm(1)(Y|Δf(1)→ B1) + ΔSSEm(1)(Y|Δf(1) → B2). And then we calculate the information gain for this split: IG(B) = ΔSSEm(1)(Y|Δf(1) → A2) – ΔSSEm(1)(Y|Δf(1) → B). And we choose the column B for which IG(B) is largest. And we continue in like fashion until we’ve broken the tree down as far as we want to go. And certainly if we should find that the information gain were ever negative, then we wouldn’t bother making the split. Once this classification is found, we’d ostensibly say,



But we won’t wholly endorse the jump, as that would overfit. So we say,



***2nd Tree***

So we’d like to construct a tree to improve this classification. So we’ll look for an increment, Δf(2), that we can add to our present prediction, f(1), to get a new prediction: fi(2) = fi(1) + Δfi(2).



Now we’ll look to classify these increments Δf(2)I into leaves. The increments will be the same for elements that end up in the same leaf of our decision tree. As before, though, we’ll start by expanding in powers of Δfi(2),



The first term, being a constant regardless of how we classify data into leaves, is a constant that doesn’t matter. So we can restrict our attention to:



So note Δf(2)i only depends on the leaf that it’s being classified into – so we write Δf(2)Lℓ, but Δf(1)i depends will depend on the leaf, and on which element it is within the leaf, because these new leaves aren’t the same as the old leaves. When we minimize this w/r to ΔfLℓ(2), we’ll have:



So we have:



This says that the parameters that minimize the loss are the ‘averages’ of the things (increments in this case) in each leaf. Plugging this back into ΔSSEm(2)(Y|Δf(2)→L), we have:



So we have:



Note ΔfLℓ(2) is the Lℓ leaf’s *prediction* for the increment. So diagrammatically, we’ll start our decision tree by putting all the data in the root node. And then we’re looking to categorize the data into categories/leaves.

Diagram, schematic

Description automatically generated

[Note ΔfR(2) = 0 – wait, is it?] And we want to find leaves, Lℓ, which maximize the information gain:



In this diagam, L = {A1, B1, B2}. We will typically use a greedy algorithm to find these leaves, as discussed above. Once this classification is found, we’d ostensibly say,



But we won’t wholly endorse the jump, as that would overfit. So we say,



***3rd Tree, etc.***

And we’d proceed likewise for all other trees. Presuming we stop after n trees, we’d have:



We’d stop when the predictions fi(n+1) and fi(n) are ‘close enough’.

**Feature Importance**

Like with decision trees, we can calculate the feature importance of a category/column/feature. I think the only difference is that now we sum over all trees in the ensemble.



This normalizes the feature importance to 1 basically. If a feature’s importance is low, then we should consider eliminating that variable from the model. It might be irrelevant to the underlying ‘physics’, and just be fitting noise.

**Example**

For example, let’s take this table, where we try to predict weight from height, favorite color, and gender.

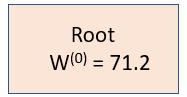
|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **Weight (kg)** |
| 1.6 | Blue | Male | 88 |
| 1.6 | Green | Female | 76 |
| 1.5 | Blue | Female | 56 |
| 1.8 | Red | Male | 73 |
| 1.5 | Green | Male | 77 |
| 1.4 | Blue | Female | 57 |

*Zeroth Tree*

First, we have to find the average weight. This is:



I’ll put this result in a tree stump, even though it’s not formally part of a tree,



And we’ll move on to the first tree,

*First Tree*

Can see that we will be fitting our decision to the residuals W – W(0). So let’s replace the weight column with the residuals. This is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **– W(0) (kg)** |
| 1.6 | Blue | Male | 16.8 |
| 1.6 | Green | Female | 4.8 |
| 1.5 | Blue | Female | -15.2 |
| 1.8 | Red | Male | 1.8 |
| 1.5 | Green | Male | 5.8 |
| 1.4 | Blue | Female | -14.2 |

Our root node would be where we place all the data initially, and output ΔWR(1) = (1/n)Σi(ΔWi) = 0. So,

A picture containing text

Description automatically generated

Now we want to do a little better and split the root node. So we want to figure out the column most closely correlated with W – W(0). That’ll be the one with the largest information gain, largest ΔSSEm(1)(W|ΔW(1)→L) value. Apropos height, it looks like the best split is at Hcrit = 1.55m. We can group favorite color by Blue/Not Blue, Green/Not Green, or Red/Not Red. And gender is obviously either Male or Female. Don’t want to go through all the details. But turns out the split with the lowest ΔSSEm is by gender.

|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **– W(0) (kg)** |
| 1.6 | Blue | Male | 16.8 |
| 1.6 | Green | Female | 4.8 |
| 1.5 | Blue | Female | -15.2 |
| 1.8 | Red | Male | 1.8 |
| 1.5 | Green | Male | 5.8 |
| 1.4 | Blue | Female | -14.2 |

So I’ll calculate this one.



and their SSEm is:



So presently, our decision tree for the residuals is:

Diagram

Description automatically generated

We would probably keep going down one more level, but I’ll just stop with the stump, to keep it brief. And our decision tree’s predictions would be (going to set α = 0.5 for illustration´s sake):



And so the table of prediction is so far:

|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **W(1) (kg)** |
| 1.6 | Blue | Male | 75.3 |
| 1.6 | Green | Female | 67.1 |
| 1.5 | Blue | Female | 67.1 |
| 1.8 | Red | Male | 75.3 |
| 1.5 | Green | Male | 75.3 |
| 1.4 | Blue | Female | 67.1 |

Okay.

*Second Tree*

Now to do the second tree, we have to determine the residuals from the first tree’s predictions. These are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **W – W(1) (kg)** |
| 1.6 | Blue | Male | 8.6 |
| 1.6 | Green | Female | 13 |
| 1.5 | Blue | Female | -7 |
| 1.8 | Red | Male | -6.4 |
| 1.5 | Green | Male | -2.4 |
| 1.4 | Blue | Female | -6 |

and now we have to train a tree on these residuals. Our root node would be where we place all the data initially, and output ΔWR(2) = (1/n)Σi(W-W(1)) = 0. So,

A picture containing text

Description automatically generated

Now we want to do a little better and split the root node. Looks like height is best now.

|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **W – W(1) (kg)** |
| 1.6 | Blue | Male | 8.6 |
| 1.6 | Green | Female | 13 |
| 1.5 | Blue | Female | -7 |
| 1.8 | Red | Male | -6.4 |
| 1.5 | Green | Male | -2.4 |
| 1.4 | Blue | Female | -6 |

And just to practice doing the ΔSSEm(2)(W|ΔW(2) → H) calculating, we have:



and their SSEm is:



Okay, well our Decision Tree is now,

Diagram

Description automatically generated

Again, we would probably keep going down one more level, but I’ll just stop with the stump, to keep it brief. And our decision tree’s predictions would be (going to set α = 0.5 for illustration´s sake):



And so the table of prediction is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **W(2) (kg)** |
| 1.6 | Blue | Male | 75.3 + 2.55 = 77.9 |
| 1.6 | Green | Female | 67.1 + 2.55 = 69.6 |
| 1.5 | Blue | Female | 67.1 – 2.55 = 64.6 |
| 1.8 | Red | Male | 75.3 + 2.55 = 77.9 |
| 1.5 | Green | Male | 75.3 – 2.55 = 72.8 |
| 1.4 | Blue | Female | 67.1 – 2.55 = 64.5 |

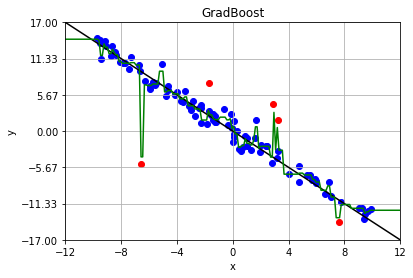
*Third Tree, etc.*

Now to do the third tree, we have to determine the residuals from the second tree’s predictions. These are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Height (m)** | **Favorite Color** | **Gender** | **W – W(2) (kg)** |
| 1.6 | Blue | Male | 10.1 |
| 1.6 | Green | Female | 6.4 |
| 1.5 | Blue | Female | -8.6 |
| 1.8 | Red | Male | -4.9 |
| 1.5 | Green | Male | 4.2 |
| 1.4 | Blue | Female | -7.5 |

**Exploring the Model and Hyperparameters**

The GradientBoostRegressor in sklearn has a lot of hyperparameters.

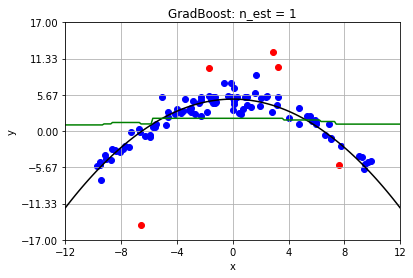
 A graph with blue and red dots

Description automatically generated

Still wants to grab the outliers. But otherwise, I think gradboost has smoothed out the fluctuations from the noise pretty well compared to decision tree regressor.

**Hyperparameter: n\_estimators**

This is the number of decision trees into our GradientBoost ‘bag’. The default is 100. So here’s our quadratic curve, with a couple different n\_estimator values. I don’t know what the max\_depth of each tree is, but I think it’s around 8 or so. So that’s why we get some level of detail, even with n\_est = 1.

 A graph of a graph with red and blue dots

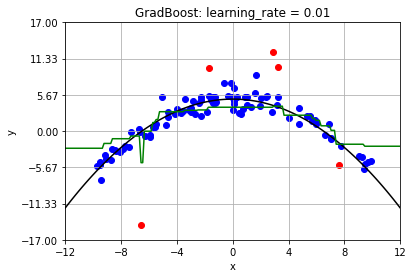
Description automatically generated A graph of a graph with red and blue dots

Description automatically generated

With n\_estimators=1, we don’t get enough of a sample of the data. So we don’t have a very good fit. It get’s better as n\_estimators increases. Maybe a little more overfitting as n\_estimators increases.

**Hyperparameter: learning\_rate**

This is the α guy mentioned up above. It modulates how much the weights get adjusted after each iteration. Larger α often means adhering more closely to data, and carries risk of overfitting. The default value is 0.1. When use smaller values of α, then you’d often want to use more estimators.

 A graph of a graph with red and blue dots

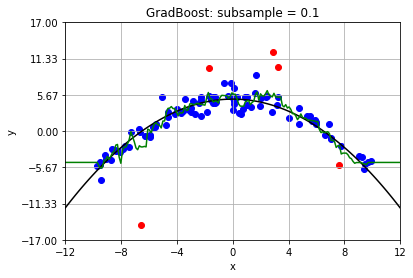
Description automatically generated A graph of a graph with a line graph

Description automatically generated with medium confidence

So there. Can see one probably doesn’t want to go too high on alpha, or we risk following data too closely.

**Hyperparameter: subsample**

This is the fraction of rows to be randomly sampled, for the construction of each of the n\_estimators decision trees I guess. This is like other similar options available to Bagging, Random Forest, and Adaboost, which are methods likewise based on an ensemble of trees. The default is subsample = 1. Here’s a couple different values.

 A graph with blue and red dots

Description automatically generated A graph of a graph with blue and red dots

Description automatically generated

Looks like one advantage of having small subsample is that the likelihood of getting an outlier decreases. And so the first guy looks pretty good.

**Hyperparameter: others**

There are lots of other hyperparameters. All of these are common to the regular decision tree classifier. And they have to do with the parameters which prescribe how we construct each of the n\_estimator trees in our GradientBoost ‘bag’. I guess I’ll not go over these, but one would surely want to include them when looking for the optimal combination of hyperparameters.