**XG Boost (Regression)**

Now we’ll look at a variant of Gradient Boost – Extreme Gradient Boost. And we’ll be looking at it in the context of Regression – i.e., trying to predict numerical values. Generally works like this. So we 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 |

Again, it can be used with more than just decision trees. For instance, it can be used with linear models as the base regressor/weak learner. But since Gradient Boosting just linearly combines its weak learners, a gradient boosted linear model is just itself a linear model (so we might as well use linear regression, or Lasso/Ridge regression instead). So we’ll just focus on using decision trees as the base regressor. And we want to write a decision tree to estimate the outcomes Y = weights.

**Constructing the Decision Tree**

So we have our typical loss function,



***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) = 0.5. Not sure why we use 0.5. But we do. So,



***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). The increments will be the same for elements that end up in the same leaf of our decision tree. But we’re also going to add a λ term to SSE to discourage proliferation of leaves. This is similar to how we use a λ term in Ridge/Lasso regression to avoid overfitting data points with sloped lines. And this will update our Loss function to:



[There is also an option for an L1 regularization term. I imagine it comes into the SSE like αΣ|Δfi|? And if we included it, then we’d update SSEλ → SSEλα. But I’m not going to bother with it as we’ve got enough going on] The restriction that the sum run over only unique values of fi means that we’re basically counting each leaf, but not the terms within the leaves. And we want to determine the fi which will minimize this loss function on average. First we’ll expand out to 2nd order in Δfi(1). Of course that’s as high as it goes anyway.



The first term is a constant that doesn’t matter, so we can restrict our attention to:



And now we’ll explicitly throw the fi into leaves,



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’ll 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’ (modulated by λ) of the things (increments in this case) in each leaf. Plugging this back into ΔSSEλ, 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

Description automatically generated

[note ΔfR(1) ≠ 0 anymore, since f(0)is arbitrarily set to 0.5] And we want to find leaves, Lℓ, which maximize the information gain:



In this diagam, L = {A1, B1, B2}. Since ΔSSEλ­(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 ΔSSEλ(1)(Y|Δf(1) → A2) = (nA2+λ)-1[Σj=1n\_A2 (yA2\_j – f(0))]2 as shown. And the prediction of this node would be fA2 = (nA2+λ)-1Σj=1n\_A2 (yA2\_j – f(0)). 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 ΔSSEλ(1)(Y|Δf(1) → B1) of the group in B1, and the entropy ΔSSEλ(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 ΔSSEλ(1)(Y|Δf(1) → B) = ΔSSEλ(1)(Y|Δf(1)→ B1) + ΔSSEλ(1)(Y|Δf(1) → B2). And then we calculate the information gain for this split: IG(B) = ΔSSEλ(1)(Y|Δf(1) → A2) – ΔSSEλ(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,



**Note #1**: In practice we prune the trees to avoid overfitting. When we’re finished with the tree, we start from the bottom, and if the gain from splitting a node is less than some user-defined value γ, then we would erase that split. And we’d work our way up the tree eliminating splits with gains < γ. We’ll stop as soon as we get to a tree whose leaves all have gains > γ (even if further up the tree there are gains < γ). This is where the regularization parameter λ comes in, as the larger λ is, the smaller the ΔSSEλ for a given leaf will be, and so the more likely it is to be dropped according to the gamma parameter.

**Note #2**: Apparently, in this algorithm, there is a minimum number of rows that can be in a node, leaf, or whatever. But I think the default is set to 1. So yeah.

*2nd Tree, etc.*

Then we’ll look at adding a second increment to SSEλ(Y|f).:



And we want to determine the fi which will minimize this loss function on average. First we’ll expand out to 2nd order in Δfi(2).



The first term is a constant that doesn’t matter, so we can restrict our attention to:



And now we’ll explicitly throw the fi into leaves,



When we minimize, we’ll have:



So we have:



Plugging this back into ΔSSEλ, 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

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,



**Note #1**: In practice we prune the trees to avoid overfitting. When we’re finished with the tree, we start from the bottom, and if the gain from splitting a node is less than some user-defined value γ, then we would erase that split. And we’d work our way up the tree eliminating splits with gains < γ. We’ll stop as soon as we get to a tree whose leaves all have gains > γ (even if further up the tree there are gains < γ). This is where the regularization parameter λ comes in, as the larger λ is, the smaller the ΔSSEλ for a given leaf will be, and so the more likely it is to be dropped according to the gamma parameter.

**Note #2**: Apparently, in this algorithm, there is a minimum number of rows that can be in a node, leaf, or whatever. But I think the default is set to 1. So yeah.

***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. So,



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.

**Exploring the Model and Hyperparameters**

The XGBoostClassifier in sklearn has a lot of hyperparameters. Let’s see how the default settings do on our linear and quadratic curves

A graph with lines and dots

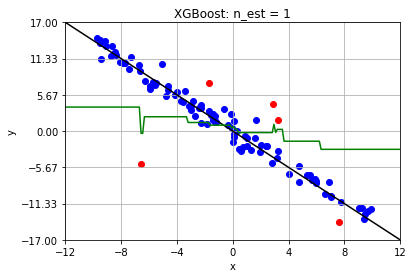
Description automatically generated A graph with lines and dots

Description automatically generated

Seems typical so far.

**Hyperparameter: n\_estimators**

This is the number of decision trees into our XGBoost ‘bag’. The default is 100. So here’s our 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 a line graph

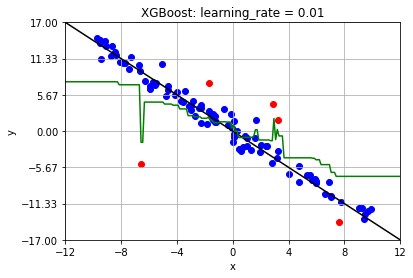
Description automatically generated with medium confidence A graph of a graph with a line graph

Description automatically generated with medium confidence

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.

**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 0.3, Chat says. When use smaller values of α, then you’d often want to use more estimators.

 A graph with a line graph and numbers

Description automatically generated with medium confidence 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: reg\_lambda**

This is the λ guy mentioned up above. It modulates the penalty given to having so many leaves in the tree. The default value is 1, Chat says. Smaller values of λ would mean less regularization, and so more closely fitting, perhaps overfitting.

A graph with lines and dots

Description automatically generated A graph with blue and red dots

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

Description automatically generated

So there. Can see higher lambda does lead to less overfitting. Could also try to vary the L1 regularization parameter, α.

**Hyperparameter: gamma**

This guy ensures a minimum improvement in the loss function to authorize splitting a leaf node. Increasing it will reduce overfitting, as we wouldn’t then authorize a split if we’re just gaining a single data point, say. I think the default value of γ is 0.3? Anyway, here’s a few plots,

A graph of a graph with a line graph

Description automatically generated with medium confidence A graph with blue and red dots

Description automatically generated

A graph with blue and red dots

Description automatically generated A graph with blue and red dots

Description automatically generated

Can see higher γ means less overfitting. And really large γ makes it start underfitting. Not sure how gamma differs from similar parameter in DecisionTreeClassifier. Maybe it doesn’t differ.

**Hyperparameter: others**

There are lots of other hyperparameters, common to other ‘bagging’ classifiers. Like with Random Forest, and tell it to choose a random subset of some percentage, x, of the columns when constructing a tree, or when constructing a level of the tree, etc. Can also do normal DecisionTree things like specifying the maximum depth of the trees. Can also change the base model from decision tree regressor to linear regressor. This didn’t seem to make much difference to my illustrative data set above.