**XG Boost (Classification)**

Now we’ll look at the extreme gradient boost algorithm for classification.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| Yes | Yes | 7 | No |
| Yes | No | 12 | No |
| No | Yes | 18 | Yes |
| No | Yes | 35 | Yes |
| Yes | Yes | 38 | Yes |
| Yes | No | 50 | No |
| No | No | 83 | No |

The general procedure is just a generalization of the gradient boost algorithm.

**Constructing the Decision Tree**

So we want to write a decision tree to estimate the outcomes Y = Loves Cool as Ice. Like before, we’ll concern ourselves with outputing a probability for the outcome. We’ll do this by incrementally minimizing the log-loss function:



where yi = 0, 1 are the outcomes, and pi the estimated probabilities of these outcomes. Like before, we’ll be formulating the procedure in terms of the log-likelihood, λ = ln(p/q), rather than p itself. Noting from previous file,



we can write LLf in terms of λ,



So we have:



I believe I wrote this a different way before, but it’s mathematically the same. To minimize the work, I’m going to convert it to:



So,



***0th Tree***

To start, we will look for a single log-likelihood/probability that will minimize the information loss. Well actually, we just say,



which implies λ(0) = ln(p(0)/q(0)) = ln(0.5/0.5) = 0. So,



***1st Tree***

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



Now we want to minimize this guy. But before we do, seems it’s common to make a quadratic approximation to LLf. This should be okay because we’re making small parameter increments and so a truncated Taylor series approximation to LLf should be accurate for our purposes. This approximation isn’t necessary *now*, but will be when we get to the 2nd tree. So might as well. Expanding out to 2nd order in Δλi(1) [why can we presume Δλi(1) is small?]:



The first term can be neglected, as it’s a constant. As for the other two, now we’ll use the fact that the increments will be the same for elements that end up in the same leaf of our decision tree. So to denote the Δλ(1)’s per leaf, I’ll write Δλ(1)i = Δλ(1)Lℓ,j, where Lℓ stands for the leaf, and j the element within the leaf. And since this is independent of j, I’ll just write Δλ(1)Lℓ. So,



Yikes. I’m going to simplify this some still, by putting it in terms of p(0) and q(0).



Then we want to minimize it. But before we do, like in the previous file, we’ll add a term that penalizes the loss for the number of leaves it has:



[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]. So we’d take the derivative w/r to ΔλLk(1) and set to 0. That’s fairly easy now,



So we have:



We could simplify this quite a bit further. But I’m going to leave it like this, since we won’t be able to simplify any of the higher order Δλℓ’s past this point. Plugging this back into LLλ. Well to save space, I’m going to borrow fact that when minimize a quadratic: quad = ax2 - bx + c, we need derivative 2ax - b = 0 → x = b/2a, and plug it back in, you get quad = a(b/2a)2 - b(b/2a) + c = -b2/4a + c. So,



Guess we could write this as:



So we have figured out what the minimum ΔLLλ would be, once classified by the decision tree into leaves. And note ΔλLℓ(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

And we want to find leaves, Lℓ, which maximize the information gain:



In this diagam, L = {A1, B1, B2}. Since ΔLLλ(1)(Y|Δλ(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 ΔLLλ of this node ΔLLλ(1)(Y|Δλ(1) → A2) as shown above. And the prediction of this node would be ΔλA2 as shown above. 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 ΔLLλ(1)(Y|Δλ(1)→B1) of the group in B1, and the loss ΔLLλ(1)(Y|Δλ(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 ΔLLλ(1)(Y|Δλ(1)→B) = ΔLLλ(1)(Y|Δλ(1)→B1) + ΔLLλ(1)(Y|Δλ(1)→B2). And then we calculate the information gain for this split: IG(B) = ΔLLλ(1)(Y|Δλ(1)→A2) – ΔLLλ(1)(Y|Δλ(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 not going to fully endorse the increment, so as not to overfit. So say rather,



where α is the learning rate. And our new probabilities will be:



**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, a leaf’s ‘cover’, defined as:



must be greater than a minimum value, usually equal to 1 by default. Might observe that the analogous term in the regression formula is just nLl, the number of terms in the leaf.

*2nd Tree*

Next, we’d like to create a decision tree to improve this classification. So we’ll look to add an increment Δλi(2) to our estimation of λi. So we’ll say λi(2) = λi(1) + Δλi(2). And fill this into LLλ,



Again we’ll expand out to quadratic order in Δλi(2),



and neglect the zeroth order term which is purely dependent on λi(1), as it’s a constant at this point. And then we’ll explicitly separate, somehow, our Δλ’s into leaves, Lℓ. So we’ll denote everything with Lℓ, j subscript, which means jth element of leaf ℓ. Note the leaves, Lℓ, in this tree, won’t be identical to the leaves in the previous tree. So we have:



I’m going to simplify this some still, by putting it in terms of p(1)ℓj and q(1)ℓj.



Then we want to minimize it. But again, before we do, we’ll add a term that discourages adding too many leaves. So,



So we’d take the derivative w/r to ΔλLk(2) and set to 0. That’s fairly easy now,



So we have:



which is correct. Plugging this back into LLf. Well to save space, I’m going to borrow fact that when minimize a quadratic: quad = ax2 + bx + c, we need derivative 2ax + b = 0 → x = -b/2a, and plug it back in, you get quad = a(-b/2a)2 + b(-b/2a) + c = -b2/4a + c. So,



Guess we could write this as:



So this is the minimum value for a given classification. But we still have to determine which of the classifications will be a minimum (i.e., largest negative). 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 not going to fully endorse the increment, so as not to overfit. So say rather,



where α is the learning rate. And our new probabilities will be:



**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, a leaf’s ‘cover’, defined as:



must be greater than a minimum value, usually equal to 1 by default. Might observe that the analogous term in the regression formula is just nLl, the number of terms in the leaf.

***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 λi(n+1) and λi(n) are ‘close enough’. Should also note that we are not required to classify something as ‘yes’ if p > 0.5, and ‘no’ if p < 0.5. We can always change threshold, according to which gives the best results. And in sklearn, we can get an array of the probabilities for each classification, pi, qi of the data X\_test.

**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,



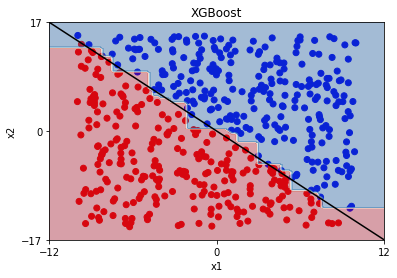
This normalizes the feature importance to 1 basically. If a feature’s importance is low, then we should consider eliminating that variable from the model. It might be irrelevant to the underlying ‘physics’, and just be fitting noise.

**Comparison to other Models**

I’m told (Datacamp) that XGBoost is best for classification problems when dealing with numeric features, or a mix of numeric and categorical features. And also when the dataset size is very large. It isn’t the best vis a vis common neural network problems like computer vision, and natural language processing.

**Exploring the Model and Hyperparameters**

The XGBoostClassifier in sklearn has a lot of hyperparameters. Let’s see how the default settings do on the N = 500 point pure dataset,

 A diagram of a red and blue graph

Description automatically generated

A diagram of a graph

Description automatically generated with medium confidence A diagram of a number of dots

Description automatically generated with medium confidence

and an impure (10% outliers for binary, 15% outliers for trinary) dataset,

A diagram of red and blue dots

Description automatically generated A diagram of a graph

Description automatically generated

A diagram of a red blue and green and yellow dotted pattern

Description automatically generated with medium confidence A diagram of a number of dots

Description automatically generated with medium confidence

Seems fairly decent. Well, a lot of overfitting as usual,

**Hyperparameter: n\_estimators**

This is the number of decision trees into our XGBoost ‘bag’. The default is 100. So here’s our linear surface, N = 500, 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 diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

A diagram of a number of dots

Description automatically generated A diagram of a number of dots

Description automatically generated with medium confidence A diagram of a number of dots

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. Can also see the risk of overfitting increases a bit as well. Interesting.

**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 diagram of red and blue dots

Description automatically generated A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

A diagram of a number of dots

Description automatically generated A diagram of a number of dots

Description automatically generated with medium confidence A diagram of a number of dots

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 diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated A diagram of red and blue dots

Description automatically generated

A diagram of a number of dots

Description automatically generated with medium confidence A diagram of a number of dots

Description automatically generated with medium confidence A diagram of a number of dots

Description automatically generated with medium confidence

So there. Can see higher lambda does lead to less overfitting. In fact, here is the λ = 100 guy,

A diagram of a number of dots

Description automatically generated

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 diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

and for the triple class guy,

A diagram of a number of dots

Description automatically generated with medium confidence A diagram of a number of dots

Description automatically generated

A diagram of a number of dots

Description automatically generated with medium confidence A diagram of a number of 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. Another thing you can do is adjust the weight penalizing False Positives or False Negatives. Maybe this is included in the Loss function somehow?