**Gradient Boost (Classification)**

Now we’ll consider how the Gradient Boost algorithm can be adapted to classification problems. Looks like we’ll adapt the gradient boost algorithm to a logistic regression, in the context of decision trees, although it can be theoretically applied to other base classifiers, besides decision trees.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

So we want to write a decision tree to estimate the outcomes Y = Loves Cool as Ice. This is actually more easily formulated in terms of log-likelihoods λi = ln(pi/qi) where qi = 1-pi, and pi = 1, 0 according to whether event is yes, no. This is because while p and q are constrained to lie with [0,1], λ is not, and can be anywhere within (-∞, ∞). This is better because for our purposes because while our incremental algorithm might push p or q outside its range, it can’t do that to λ. The algorithm generally works like this (pictured adapted from Datacamp video). We start by calculating the overall log-likelihood λ(0), of say, y = 1. Then we calculate from this p(0), using p = eλ(1+eλ), subtract this from our data y – p0), and train a decision tree on this to get the predictions for the difference log-likelihood increments Δλ(1). With these we form λ(1) = λ(0) + αΔλ(1). α is called the learning rate (between 0 and 1), and governs how much we endorse the prediction for the differences. We calculate p(1) from this via p = eλ/(1+eλ), construct the new residuals y – p(1) and train a tree on *that*. From this we get new log-likelihood increments Δλ(2), and form λ(2) = λ(1) + αΔλ(2). We calculate p(2) from this via the aforementioned formula, construct the new residuals y – p(2), and train a tree on *that*. And the process continues like this. Eventually we come to the end – the Nth tree – and its output predictions for the data points, λ(N) = λ(N-1) + σΔλ(N), from which we get our final p(N) prediction.

A diagram of a tree

Description automatically generated

So there.

**Constructing the Decision Tree**

Our classification procedure will be not be using the entropy or gini impurity stuff from before. Instead, we’ll be using log-loss, as discussed in Decision Tree Classification file,



where the yi are our probabilities = 0, 1 of the outcome being something, and f(xi) is our estimate of the probabilities via the logistic regression function. I guess I’ll kind of recapitulate that discussion. So actually we will not be using a logistic regression function per se´. Instead we will be classifying yi’s into leaves of constant probability. And trying to find the classification which minimizes LLf(Y). I’ll try to formulate the problem in analogous fashion to the regression model, based on a single decision tree. So, changing notation a bit, let pi = f(xi) be the probabilities we’re matching,



Then say we were to group all the data into a single (root) leaf, with a single probability p. What probability would minimize LLf(Y)? Well we’d have:



and to minimize, we’d say:



where n0 = number cases where yi = 0, and n1 = number of cases where yi = 1. Note this is just the overall probability of the outcome occuring. So if making a decision tree, we’d start all the data off in the root node with:



But we’d like to improve upon this classification, so we’d want to map the data into leaves Lℓ which assign probabilities pLℓ to the points therein. And let’s see what these probabilities would be, to minimize LLf(Y|L). So we form,



and differentiate,



So we have:



This doesn’t tell us per se´ *how* to determine these leaves Lℓ which minimize SSE. We would presumably use a greedy algorithm, just like we always do, to maximize the information gain:



Instead, now we’re going to do this kind of incrementally. Generally speaking, let’s say we have a bunch of probability predictions, pi, for our data yi. And let’s write LLf(Y|p) as:



where pi is left unknown for now. We’ll want to find incremental predictions, pi, which minimize LLf (or maximize Information Gain). The procedure is very similar to the one used for Gradient Boost Regression.

***0th Tree***

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



Let’s put this in terms of log-likelihood, λ:



Let’s recall for future reference,



and then filling this into LLf,



So,



Then what value of λ(0) would minimize LLf?



where n0 = # of 0’s, n1 = # of 1’s, and n = n0 + n1. So we have,



This is just the overall log-likelihood of the outcomes themselves. I think we prefer to put things in terms of λ because λ ranges form -∞ to ∞, and so we can add increments to it without running into the boundary issues that p and q have?

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



So we have:



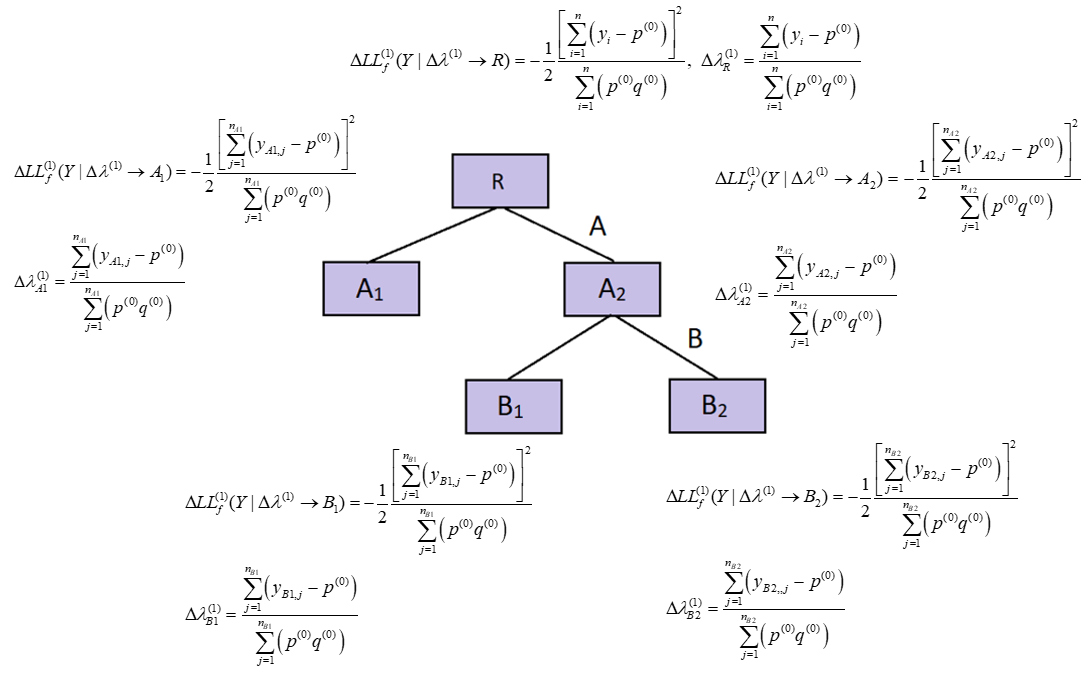
which is correct. We could simplify this quite a bit further. But I’m going to leave it like this, since we won’t be able to simplify any of the higher order Δλℓ’s past this point. Plugging this back into LLf. Well to save space, I’m going to borrow fact that when minimize a quadratic: quad = ax2 - bx + 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 we have figured out what the minimum ΔLLf 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.



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



In this diagam, L = {A1, B1, B2}. Since ΔLLf(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 ΔLLf of this node ΔLLf(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 ΔLLf(1)(Y|Δλ(1)→B1) of the group in B1, and the entropy ΔLLf(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 ΔLLf(1)(Y|Δλ(1)→B) = ΔLLf(1)(Y|Δλ(1)→B1) + ΔLLf(1)(Y|Δλ(1)→B2). And then we calculate the information gain for this split: IG(B) = ΔLLf(1)(Y|Δλ(1)→A2) – ΔLLf(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:



*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 LLf,



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. 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:



***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.

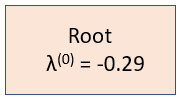
**Example**

So let’s go back to our example above,

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

*Zeroth Tree*

So λ(0) = ln(p(0)/q(0)) = ln(3/4) = -0.29. Of course this λ(0) corresponds to p(0) = 3/7, and q(0) = 4/7. Going to put our result in a root stump, even though it’s not really a part of a tree.



*First Tree*

Now gonna calculate λ(1). To do this, we need to add the residuals, y – p(0), to our table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **LC: y – p(0)** |
| Yes | Yes | 7 | 0 – 3/7 = -3/7 |
| Yes | No | 12 | 0 – 3/7 = -3/7 |
| No | Yes | 18 | 1 – 3/7 = 4/7 |
| No | Yes | 35 | 1 – 3/7 = 4/7 |
| Yes | Yes | 38 | 1 – 3/7 = 4/7 |
| Yes | No | 50 | 0 – 3/7 = -3/7 |
| No | No | 83 | 0 – 3/7 = -3/7 |

So our root node would be where we place all the data initially. The output of the root node is:



So we have, to start,

A picture containing text

Description automatically generated

And we want to figure out which column is best correlated with the residuals, as measured by the LLf guy. So we form,



If we sort by *Loves Popcorn*, this would be:



and if we sort by Loves Soda,



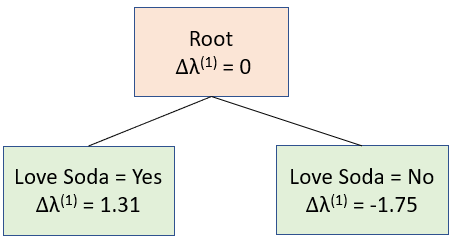
So Loves Soda is the best of the two. And the Age column doesn’t look like a serious competitor. So we’ll go with Loves Soda.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **LC: y – p(0)** |
| Yes | Yes | 7 | 0 – 3/7 = -3/7 |
| Yes | No | 12 | 0 – 3/7 = -3/7 |
| No | Yes | 18 | 1 – 3/7 = 4/7 |
| No | Yes | 35 | 1 – 3/7 = 4/7 |
| Yes | Yes | 38 | 1 – 3/7 = 4/7 |
| Yes | No | 50 | 0 – 3/7 = -3/7 |
| No | No | 83 | 0 – 3/7 = -3/7 |

Then we update log-likelihood increments,



This will update our tree to:



Let’s split the Loves Soda = Yes column by another category.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **LC: y – p(0)** |
| Yes | Yes | 7 | 0 – 3/7 = -3/7 |
| Yes | No | 12 | 0 – 3/7 = -3/7 |
| No | Yes | 18 | 1 – 3/7 = 4/7 |
| No | Yes | 35 | 1 – 3/7 = 4/7 |
| Yes | Yes | 38 | 1 – 3/7 = 4/7 |
| Yes | No | 50 | 0 – 3/7 = -3/7 |
| No | No | 83 | 0 – 3/7 = -3/7 |

The best split would be by age: Age < 12.5, Age > 12.5. This would end up putting only one row in Age < 12.5, but I guess I’ll just go with it.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **LC: y – p(0)** |
| Yes | Yes | 7 | 0 – 3/7 = -3/7 |
| Yes | No | 12 | 0 – 3/7 = -3/7 |
| No | Yes | 18 | 1 – 3/7 = 4/7 |
| No | Yes | 35 | 1 – 3/7 = 4/7 |
| Yes | Yes | 38 | 1 – 3/7 = 4/7 |
| Yes | No | 50 | 0 – 3/7 = -3/7 |
| No | No | 83 | 0 – 3/7 = -3/7 |

Let’s calculate ΔLLf(LC|Δλ(1)→LS=yes, Age),



Reassuringly, the information gain IG(LS=yes,Age) = ΔLLf(1)(LC|Δλ(1)→LS = yes) – ΔLLf(1)(LC|Δλ(1)→LS=yes,Age) = -0.84 – (-0.88) = 0.04 from splitting this node is positive. So it’s worthwhile. So now let’s look at the updated Δλ(1) predictions.



So now we have,

Diagram

Description automatically generated

Let’s presume α = 0.5 for sake of discussion, and calculate new λ’s,



and translating to probabilities,



This makes our table of predictions equal to:

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **LC: p(1)** |
| Yes | Yes | 7 | 0.24 |
| Yes | No | 12 | 0.71 |
| No | Yes | 18 | 0.71 |
| No | Yes | 35 | 0.71 |
| Yes | Yes | 38 | 0.71 |
| Yes | No | 50 | 0.24 |
| No | No | 83 | 0.24 |

Rounding to the nearest integer, these probabilities would give the correct predictions for all but the second row.

*Second Tree*

Now gonna calculate λ(2). To do this, we need to add the residuals, y – p(1), to our table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **LC: y – p(1)** |
| Yes | Yes | 7 | 0 - 0.24 = 0.76 |
| Yes | No | 12 | 0 - 0.71 = -0.71 |
| No | Yes | 18 | 1 - 0.71 = 0.29 |
| No | Yes | 35 | 1 - 0.71 = 0.29 |
| Yes | Yes | 38 | 1 - 0.71 = 0.29 |
| Yes | No | 50 | 0 - 0.24 = -0.24 |
| No | No | 83 | 0 - 0.24 = -0.24 |

So our root node would be where we place all the data initially. The output of the root node is:



So we have, to start,

A picture containing text

Description automatically generated

And we want to figure out which column is best correlated with the residuals, as measured by the LLf guy. Pretty sure this is going to be Loves Soda again.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **LC: y – p(1)** |
| Yes | Yes | 7 | 0 - 0.24 = 0.76 |
| Yes | No | 12 | 0 - 0.71 = -0.71 |
| No | Yes | 18 | 1 - 0.71 = 0.29 |
| No | Yes | 35 | 1 - 0.71 = 0.29 |
| Yes | Yes | 38 | 1 - 0.71 = 0.29 |
| Yes | No | 50 | 0 - 0.24 = -0.24 |
| No | No | 83 | 0 - 0.24 = -0.24 |

Just for the sake of it, let’s calculate:



And the prediction would be:



This will update our tree to:

Diagram

Description automatically generated

Let’s presume α = 0.5 for sake of discussion, and calculate new λ’s,



and translating to probabilities,



This makes our table of predictions equal to:

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **LC: p(2)** |
| Yes | Yes | 7 | 0.45 |
| Yes | No | 12 | 0.1 |
| No | Yes | 18 | 0.87 |
| No | Yes | 35 | 0.87 |
| Yes | Yes | 38 | 0.87 |
| Yes | No | 50 | 0.1 |
| No | No | 83 | 0.1 |

Rounding to the nearest integer, these probabilities would give the correct predictions for all rows. So our answers are clearly improving.

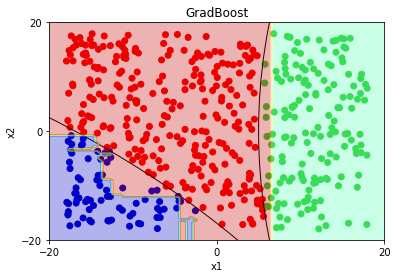
**Exploring the Model and Hyperparameters**

The GradientBoostClassifier 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 graph

Description automatically generated A diagram of a graph

Description automatically generated

 A diagram of a graph

Description automatically generated with medium confidence

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

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

A diagram of a graph

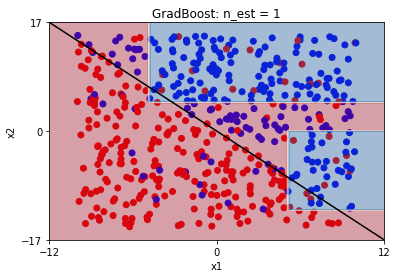
Description automatically generated with medium confidence A diagram of a graph

Description automatically generated with medium confidence

Seems fairly decent.

**Hyperparameter: n\_estimators**

This is the number of decision trees into our GradientBoost ‘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 graph of red and blue dots

Description automatically generatedA graph of a graph of a graph

Description automatically generated with medium confidence

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated A diagram of a 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. 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 0.1. When use smaller values of α, then you’d often want to use more estimators.

A diagram of a graph

Description automatically generated A diagram of red and blue dots

Description automatically generatedA diagram of red and blue dots

Description automatically generated

A diagram of a graph

Description automatically generated A diagram of a graph

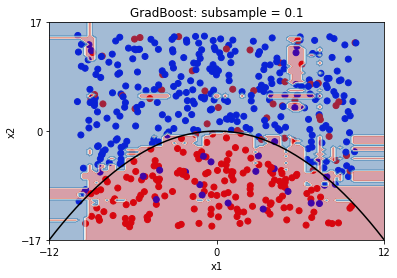
Description automatically generated with medium confidence A diagram of a graph

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 diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

A diagram of a graph

Description automatically generated with medium confidence A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated with medium confidence

Looks like a middle ground is to be had with this parameter too. The subsample = 0.5 guy looks best for the triple-class, but subsample = 1 looks best for the double-class.

**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, particularly ccp\_alpha, and min\_impurity\_decrease. You can also adjust the weight penalizing False Positives or False Negatives. Maybe this is included in the Loss function somehow?