**Adaboost**

We’ll look at *another* way to mitigate the overfitting issue, using Adaboost (Adaptive Boosting). Adaboost is an algorithm to build a strong learner out of a bunch of weak learners. We’ll explore this in the context of Decision Trees, though, like with Bagging, it can be applied to all different classifiers. A weak learner is an algorithm that produces the correct classification better than 50% of the time, but not much better. In the context of decision trees, a weak learner would be a tree stump – just the first node and (two) leaves of a decision tree, but nothing more. Adaboost creates an ensemble of such tree stumps, each with a different weight, and the final classification of the ensemble is just the weighted sum of the classifications. So this is like the Bagging method, but instead of all models in the bag getting the same weight, these have different weights (or, *say*, in this context). Also, each model influence the say of the next model, as well as the data its trained on. So each model, boosts the next one, in a sense, which is why it’s called boosting.

I’ll sketch the algorithm, or at least one way to do it. It starts with the data table, assigns weights to the rows (initially, all identical so weight = 1/nrows), constructs a decision tree stump from it, calculates how well it did via an error rate, and finally calculates a ‘say’ for it, i.e., how, much weight its decision should be assigned based on the error rate. Then it assigns new weights to the rows, emphasizing the rows in which it did poorly, and constructs a new decision tree stump fom *it*, calculates its ‘say’ based on its error rate, and then assigns new weights to the rows again, giving higher weight to the ones it did poorly on. This process continues for some predetermined time. Finally, all the decision trees are collated, with their assigned ‘says’, and the overall prediction for a row is taken to be the one with the highest net say.



Here’s a diagram I got off of Datacamp. Can see we feed data into Predictor1 (tree stump usually), its success rate and ‘say’ is calculated (that’s what α1 is representing), and its success rate goes into calculating the weights, W(2), for the data in the second round. Then this weighted data is fed into Predictor2, its success rate ans ‘say’ is calculated (that’s α2), and its success rate goes into calculating the weights, W(3) for the data in the third round, etc.

A diagram of a graph

Description automatically generated

To go into a little more detail, suppose we are somewhere in the middle of the process, with our nth data table, and the weights.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** | **weights** |
| Yes | Yes | 7 | No | w1(n) |
| Yes | No | 12 | No | w2(n) |
| No | Yes | 18 | Yes | w3(n) |
| No | Yes | 35 | Yes | w4(n) |
| Yes | Yes | 38 | Yes | w5(n) |
| Yes | No | 50 | No | w6(n) |
| No | No | 83 | No | w7(n) |

Now to construct our nth decision tree stump, we investigate to see which column is most closely correlated with the outcome. But instead of calculating the gini impurity loss or information gain, we calculate the weighted error rate (see Decision Tree file). That’s what I’ve seen done in most of the examples. The *weighted* error rate is different from the error rate in that instead of adding up all the classification errors and dividing by nrows, we add up all the weights × errors (which is the same as the former prescription, if the weights are all identical). So,



where εi = 1 if the row is misclassified and 0 otherwise, and subscript incor denotes the incorrect rows’ weights. Might also define a ‘success’ rate, for correctly classifying,



where subscript cor denotes the correct rows’ weights. So we find a mapping from one column’s values to the Outcome’s values with the lowest Error rate. Once found we create that decision tree stump, and then we have to calculate its ‘say’, or how much weight to give its classification when combined with all the other decision trees we make. The amount of ‘say’, is:



The ‘say’ function looks like this.

Chart

Description automatically generated

So when E ~ 0 (and C ~ 1) the amount of say the tree stump will have is large. When error ~ 1/2, which no better than random, then amount of say is 0. And when error ~ 1, so that nearly all samples are incorrectly classified, then the amount of say is very negative (seems you could then just reverse the predictions and get a good model – so that large negative values of say are kind of good?). FWIW, we’ll recognize the error function as the inverse of the logistic function,



Anyway, having consructed our tree, and its ‘say’, we move on to the next tree. To construct the next tree, we have to construct new weights for the rows. The recursion formula for the new weights is:



where wi(n) is the vector of old weights, and Z is a normalization factor to ensure Σiwi(n+1) = 1. We can actually work out this normalization factor, and simplify this formula for the new weights:



This brings our weight formula to:



So finally:



[α is the learning rate. In sklearn, its default value is 1]

Evidently, if a decision tree is good enough, so that C(n) > E(n), then 1/E(n) > 1/C(n), and so the rows it got wrong will be given a higher weight, and the rows it gone right will be given a lower weight. I guess our inequality should be the case, because a decision tree stump is supposed to be a weak learner, meaning that it does get, though perhaps not by much, more correct answers than incorrect. This would seem to suggest that if this isn’t the case, then maybe the algorithm fails? Anyway, so we’ll update the weights in our new data table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** | **weights** |
| Yes | Yes | 7 | No | w1(n+1) |
| Yes | No | 12 | No | w2(n+1) |
| No | Yes | 18 | Yes | w3(n+1) |
| No | Yes | 35 | Yes | w4(n+1) |
| Yes | Yes | 38 | Yes | w5(n+1) |
| Yes | No | 50 | No | w6(n+1) |
| No | No | 83 | No | w7(n+1) |

And we’ll repeat the process. Then we’ll collate all the trees together and write, for our decision:



where max means the decision which has the max net weight. All the trees will output some specific decision on any data row fed to them, and we’d just see which decision has the most net weight behind it. For instance, we’d end up with something like,



Looks like sklearn also uses this result to estimate probabilities. And I think it converts the final decision coefficients to probabilities using the softmax function. To take our example above as an illustration, we’d have:



Given this, we can set different probability cutoffs, pcut, for delineating when we might decide yes vs. no. Typically pcut = 0.5, but could be anything in principle. And different cutoffs will be more useful than others, especially if we’re looking to minimize False Positives or False Negatives, etc. Let’s try to work out an example,

**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, times their say? 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***

We’ll consider this data table. We wanted to predict whether someone would like that movie based on their preference for Popcorn, Soda, and their age.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 Decision Tree Stump*

So first, we assign weights to the rows. At first all rows will be counted equally. And since we have 7 rows, the weights (normalized) will be 1/7.



and then to remind us of this fact, we’ll write:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** | **weights** |
| Yes | Yes | 7 | No | 1/7 |
| Yes | No | 12 | No | 1/7 |
| No | Yes | 18 | Yes | 1/7 |
| No | Yes | 35 | Yes | 1/7 |
| Yes | Yes | 38 | Yes | 1/7 |
| Yes | No | 50 | No | 1/7 |
| No | No | 83 | No | 1/7 |

Like usual, we’ll want to find the column which is most highly correlated with the outcome, as measured by weighted Error Rate though. Let’s try Loves Popcorn. The best rule seems to be: Loves Popcorn = Yes → Loves Cool as Ice = No. And the weighted error rate is:



The best rule for Loves Soda seems to be: Loves Soda = Yes → Loves Cool as Ice = Yes. And the weighted error rate is:



And the best rule for Age seems to be:



So the error rate of soda is the lowest (we also found its gini impurity to be lowest, way back when),



and we can write our tree stump, and calculate its ‘say’.

Diagram

Description automatically generated

where ‘say’ is:



*First Decision Tree Stump*

Now we have to update the weights on our data table and create the first/second tree. To update the weights, we need to isolate the correctly, and uncorrectly classified rows. The incorrectly classified row has the slash.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** | **weights** |
| Yes | Yes | 7 | No | 1/7 |
| Yes | No | 12 | No | 1/7 |
| No | Yes | 18 | Yes | 1/7 |
| No | Yes | 35 | Yes | 1/7 |
| Yes | Yes | 38 | Yes | 1/7 |
| Yes | No | 50 | No | 1/7 |
| No | No | 83 | No | 1/7 |

So the first one is not correctly classified. And the new weights are (we’ll set α = 1):



Can see that rows which are incorrectly classified have higher weights now. And so our new weighted data table is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** | **weights** |
| Yes | Yes | 7 | No | 1/2 |
| Yes | No | 12 | No | 1/12 |
| No | Yes | 18 | Yes | 1/12 |
| No | Yes | 35 | Yes | 1/12 |
| Yes | Yes | 38 | Yes | 1/12 |
| Yes | No | 50 | No | 1/12 |
| No | No | 83 | No | 1/12 |

Now we have to see which column is most correlated with the outcome, vis a vis the weighted error rate metric. We’ll look at Loves Popcorn first. The best rule for it is:



The best rule for Loves Soda now seems, given the large weight of the first row, to be: Loves Soda = Yes → Loves Cool as Ice = No. Well actually, the other rule Loves Soda = Yes → Loves Cool as Ice = Yes is just as good/bad. They have the same error rate:



And the best rule for Age still seems to be Age > 15 = Yes → Loves Cool as Ice = Yes:



So we have a tie for lowest error rate. I guess I’ll use Popcorn. So our error is:



So our new tree stump, and ‘say’ would be:

Diagram

Description automatically generated

where the say is:



*Second Decision Tree Stump*

And we repeat. First we identify the incorrectly classified rows,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** | **weights** |
| Yes | Yes | 7 | No | 1/2 |
| Yes | No | 12 | No | 1/12 |
| No | Yes | 18 | Yes | 1/12 |
| No | Yes | 35 | Yes | 1/12 |
| Yes | Yes | 38 | Yes | 1/12 |
| Yes | No | 50 | No | 1/12 |
| No | No | 83 | No | 1/12 |

and update the weights (we’ll set α = 1),



which comes to:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** | **weights** |
| Yes | Yes | 7 | No | 3/10 |
| Yes | No | 12 | No | 1/20 |
| No | Yes | 18 | Yes | 1/20 |
| No | Yes | 35 | Yes | 1/20 |
| Yes | Yes | 38 | Yes | 1/4 |
| Yes | No | 50 | No | 1/20 |
| No | No | 83 | No | 1/4 |

Reassuringly, the weights still add up to 1, as they should. Not going to bother anymore, ‘cause now it looks hard. But we’d keep going the same way we have.

*Adding the stumps together*

If we content ourselves with just the first two stumps, then our decision algorithm is:



and we go with whatever outcome has the largest weight. So if we were to try to classify that

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| Yes | Yes | 7 | No |

we’d get:



So we’d still go with ‘Yes’, even though the correct answer is ‘No’. But the ‘No’ weight is much higher than it would be w/o DT(1). So I imagine that with a couple more trees, we’d eventually overturn the influence of DT(0) and get the correct answer.

**Error Rate**

I’ve seen another way to quantify the correlation of a column with the outcome. This kind of makes the most sense of all, really. It’s called the error rate, E. Could also define a correct classifications rate, C = 1 – E.



And it seems to work like this. We take a column, like Outlook perhaps, and see if we can create a rule to map its value to the Play value.

Table

Description automatically generated

There seem to be six possible mappings, excluding possibility that all Outlook possibilities result in Yes or No, in which case there would be no dependence of Play on Outlook:



And the error rate of any of the mappings would just be the fraction of errors the mapping incurs:



So the rule:



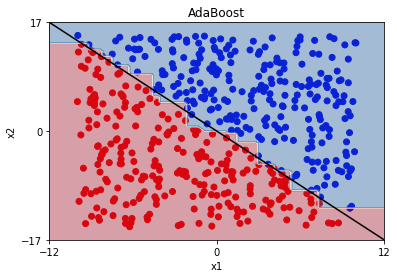
has the lowest error rate. And we’d start a decision tree node off of this guy. And continue to break it down similarly.

**How do we modify this algorithm when we’re doing decision tree regression?**

We do the same as above, and take the average of the predictions, instead of the mode.

**Exploring the Model and Hyperparameters**

The AdaBoostClassifier in sklearn basically takes as input any other classifier you want, through the **base\_estimator = model(hyperparameters=…)** argument. Apparently the default setting is a two-leaf node decision tree. Let’s see how the default settings do on the N = 500 point pure dataset,

 A diagram of a graph

Description automatically generated

and an impure (10% outliers) dataset,

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

Seems kind of iffy so far. And let’s do the three-class guys,

A diagram of a graph

Description automatically generated A diagram of a diagram with circles and a number of dots

Description automatically generated with medium confidence

and usual 15% outliers,

A diagram of a diagram of a substance

Description automatically generated with medium confidenceA diagram of a diagram of a number of dots

Description automatically generated with medium confidence

Yikes. And let’s try Regression,

A graph with blue and red dots

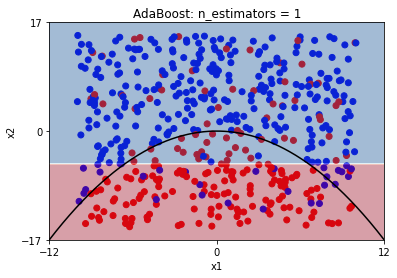
Description automatically generated A graph with blue and red dots

Description automatically generated

Well, this is an improvement, to some degree, over regular decision tree.

**Hyperparameter: n\_estimators**

This is the number of decision trees that go into our adaboost ‘bagger’. The default is 50. So here’s our quadratic surface, N = 500, with a couple different n\_estimator values. And also the N = 500 three-class guy,

 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 with medium confidence A diagram of 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 (since our base\_estimator is a two-leaf tree). 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. And with regression,

A graph with red and blue dots

Description automatically generated A graph with blue and red dots

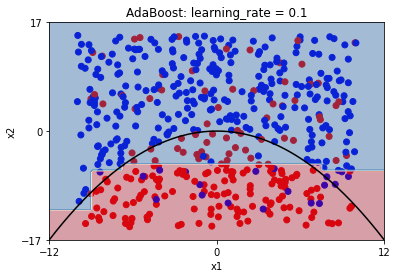
Description automatically generated A graph with red and blue dots

Description automatically generated

Also seems to be a good middle ground somewhere.

**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 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 a graph

Description automatically generated

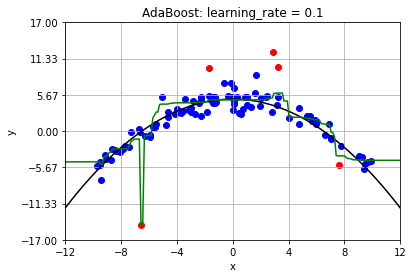
A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

Definitely jumped the gun with the large learning rate. And did the quasi-linear three-class by mistake, but the circle three-class looks way worse. Not sure why this is not performing well. I must not be in the range range for these hyperparameters? Or something? And let’s do regression,

 A graph with red and blue dots

Description automatically generated A graph with red and blue dots

Description automatically generated

And we see the same problem with super high learning rate as had with classification.

**Hyperparameter: others**

Many of the hyperparameters available to regular decision trees are available here too, like min\_impurity\_decrease, min\_samples\_split, ccp\_alpha, etc. Looks like it’d be best to use RandomForrest in conjunction with, say, an optimum value of ccp\_alpha.