**Bagging + Random Forest**

As we’ve seen, decision trees are not the best, in general, because they overfit data. Pruning helps mitigate this. And gradboost, xgboost helps too. Another possibility for addressing this is the Bagging technique. This technique is pretty general, and can be applied to all classifiers, not just decision trees. And it can be applied to regression algorithms too. The basic idea is to construct a classifier/regressor model from, say, 100 random samples of the rows of data. And then repeat over and over again. And then take the definitive classification as the mode of all models, and the definitive regression as the mean of all models. The advantage of this technique is that, in a neighborhood where there is an outlier, most models won’t pick it up, and so it won’t sway predictions apropos classification because the models that picked it up will be outnumbered by those that didn’t. And apropos regression, it will factor in because the final prediction is the mean of all models’ predictions, not the mode, but its influence will be mitigated because its influence on the mean will be weighted by how many random models pick up the outlier. There are two ways to do this apparently. I’ll start with method 1. And I’ll discuss it in the context of decision trees.

**Method 1**

We start with the data. And we do the usual train\_test\_split thing where we split our data into a training set and a testing set. Say our data set has nrows and ncolumns (besides the outcome column). So in our illustration, nrows = 12, and ncolumns = 4. Note this would be the *training data*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa1 | xb1 | xc1 | xd1 | y1 |
| xa2 | xb2 | xc2 | xd2 | y2 |
| xa3 | xb3 | xc3 | xd3 | y3 |
| xa4 | xb4 | xc4 | xd4 | y4 |
| xa5 | xb5 | xc5 | xd5 | y5 |
| xa6 | xb6 | xc6 | xd6 | y6 |
| xa7 | xb7 | xc7 | xd7 | y7 |
| xa8 | xb8 | xc8 | xd8 | y8 |
| xa9 | xb9 | xc9 | xd9 | y9 |
| xa10 | xb10 | xc10 | xd10 | y10 |
| xa11 | xb11 | xc11 | xd11 | y11 |
| xa12 | xb12 | xc12 | xd12 | y12 |

**Create Bootstrapped Decision Tree**

Then we create an equal sized ‘boot-strapped’ data set by randomly sampling rows (*with* replacement) from our data set nrows times (I’ve seen some algorithms allow sampling less than nrows times). It is possible/likely that we will sample the same row twice when creating our ‘boot-strapped’ data set, and this is okay. We will call this data set Xtrain(1), ytrain(1).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa1 | xb1 | xc1 | xd1 | y1 |
| xa2 | xb2 | xc2 | xd2 | y2 |
| xa2 | xb2 | xc2 | xd2 | y2 |
| xa4 | xb4 | xc4 | xd4 | y4 |
| xa4 | xb4 | xc4 | xd4 | y4 |
| xa6 | xb6 | xc6 | xd6 | y6 |
| xa7 | xb7 | xc7 | xd7 | y7 |
| xa8 | xb8 | xc8 | xd8 | y8 |
| xa8 | xb8 | xc8 | xd8 | y8 |
| xa11 | xb11 | xc11 | xd11 | y11 |
| xa11 | xb11 | xc11 | xd11 | y11 |
| xa12 | xb12 | xc12 | xd12 | y12 |

Then in the generic Bagging method, we create a decision tree for Xtrain(1), ytrain(1). But our m.o. for the specifically *Random Forest* implementation of Bagging is a little different. Instead of looking for the column in Xtrain(1) which has highest information gain, or gini impurity loss vis a vis the outcome, we randomly choose just a subset of ~ √ncolumns (√4 in our illustation) of the Xtrain(1) columns and see which of *those* has the highest information gain/gini impurity loss vis a vis the outcome. And so we create the root node and leaves based on *that* column. Then going to the left leaf we pick a random subset of size ~ √ncolumns (√4 in our illustation) of the remaining columns, and find the one which has the highest information gain/gini impurity loss, and make another branch of the decision tree based on *it*. And then we go to the right leaf and do the same: pick a random subset of size √ncolumns of the remaining columns, find the one which has the highest information gain/gini impurit loss, and make another branch of the decision tree. And we just repeat until we’re done with the tree. We would probably keep restriction that we need at least some minimum number of data points per leaf, say 20.

**Create another Bootstrapped Decision Tree**

Then we create another decision tree using the same method. We randomly sample our original data set nrows times, to get another Xtrain(2), ytrain(2).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa3 | xb3 | xc3 | xd3 | y3 |
| xa3 | xb3 | xc3 | xd3 | y3 |
| xa4 | xb4 | xc4 | xd4 | y4 |
| xa5 | xb5 | xc5 | xd5 | y5 |
| xa5 | xb5 | xc5 | xd5 | y5 |
| xa6 | xb6 | xc6 | xd6 | y6 |
| xa7 | xb7 | xc7 | xd7 | y7 |
| xa9 | xb9 | xc9 | xd9 | y9 |
| xa10 | xb10 | xc10 | xd10 | y10 |
| xa11 | xb11 | xc11 | xd10 | y11 |
| xa10 | xb10 | xc10 | xd10 | y10 |
| xa12 | xb12 | xc12 | xd12 | y12 |

And then we do as before. We pick a subset of ~ √ncolumns columns from Xtrain(2), find the one most correlated with the outcome, and start a decision tree off of it. Then for the left leaf, we pick a subset of size ~ √ncolumns from the remaining columns, find the column out of that subset that is most correlated with the outcome, and make another branch of the decision tree. And we do likewise for the right leaf. Etc., until we’re done with the tree. Obviously, the number of columns left to choose from diminishes by one each level we go down the tree, and I assume once we get to the point where the number of columns left to pick from, along a particular branch, is less than √ncolumns, then we just choose all of them.

**Create another Bootstrapped Decision Tree, etc.**

And then we make another random sample Xtrain(3), ytrain(3), and build another tree. We might do this hundreds of times.

**Create different classes of Bootstrapped Decision Trees, etc.**

And we can also go back and change our model by increasing or decreasing the number of columns of Xtrain we put into the subsets. Typically we’ll do √ncolumns ± 1, √ncolumns ± 2, etc. We might do hundreds of Trees for each new class.

**Predictions for data**

To make definitive predictions on any given row of our data – either training or testing, we’d run the data through *every* model, and take as our official prediction whatever the majority outcome prediction is.



Then we could create a confusion matrix of all the predictions and that would be our accuracy test. For instance, we might have something like,



Looks like sklearn also uses this result to estimate probabilities of yes’s and no’s. To take our illustration as an example, I think it just does,



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.

**Testing the model**

We can assess our model’s performance on the training data as usual. Just make predictions on the training data and build a classification matrix or something. And we can test the model on the testing data in the same way.

**Method 2**

This way is a little different. We don’t separate our data into training/testing sets per se´. We let the random sampling kind of do that for us. This approach is useful when we have very little data to begin with, and want to use as much of it as we can to build our model. So we start with the data. Say our data set has nrows and ncolumns (besides the outcome column). So in our illustration, nrows = 12, and ncolumns = 4. Note this would now be *all* of the data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa1 | xb1 | xc1 | xd1 | y1 |
| xa2 | xb2 | xc2 | xd2 | y2 |
| xa3 | xb3 | xc3 | xd3 | y3 |
| xa4 | xb4 | xc4 | xd4 | y4 |
| xa5 | xb5 | xc5 | xd5 | y5 |
| xa6 | xb6 | xc6 | xd6 | y6 |
| xa7 | xb7 | xc7 | xd7 | y7 |
| xa8 | xb8 | xc8 | xd8 | y8 |
| xa9 | xb9 | xc9 | xd9 | y9 |
| xa10 | xb10 | xc10 | xd10 | y10 |
| xa11 | xb11 | xc11 | xd11 | y11 |
| xa12 | xb12 | xc12 | xd12 | y12 |

**Create Bootstrapped Decision Tree**

Then we create an equal sized ‘boot-strapped’ data set by randomly sampling rows (*with* replacement) from our data set nrows times (I’ve seen some algorithms allow sampling less than nrows times). It is possible/likely that we will sample the same row twice when creating our ‘boot-strapped’ data set, and this is okay. We will call this data set Xtrain(1), ytrain(1). And the rows that didn’t make it into the bootstrap data set we’ll collate and call Xtest(1) and ytest(1). So might have Xtrain(1), ytrain(1) on left, and Xtest(1), ytest(1) on right. Xtest(1) and ytest(1) are also/primarily called the out-of-bag (OOB) data. We’ll discuss what we might do with the OOB data in a bit.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa3 | xb3 | xc3 | xd3 | y3 |
| xa5 | xb5 | xc5 | xd5 | Y5 |
| xa9 | xb9 | xc9 | xd9 | y9 |
| xa10 | xb10 | xc10 | xd10 | y10 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa1 | xb1 | xc1 | xd1 | y1 |
| xa2 | xb2 | xc2 | xd2 | y2 |
| xa2 | xb2 | xc2 | xd2 | y2 |
| xa4 | xb4 | xc4 | xd4 | y4 |
| xa4 | xb4 | xc4 | xd4 | y4 |
| xa6 | xb6 | xc6 | xd6 | y6 |
| xa7 | xb7 | xc7 | xd7 | y7 |
| xa8 | xb8 | xc8 | xd8 | y8 |
| xa8 | xb8 | xc8 | xd8 | y8 |
| xa11 | xb11 | xc11 | xd11 | y11 |
| xa11 | xb11 | xc11 | xd11 | y11 |
| xa12 | xb12 | xc12 | xd12 | y12 |

Then in the generic Bagging method, we create a decision tree for Xtrain(1), ytrain(1). But our m.o. for the specifically Random Forest implementation of Bagging is a little different. Instead of looking for the column in Xtrain(1) which has highest information gain, or gini impurity loss vis a vis the outcome, we randomly choose just a subset of ~ √ncolumns (√4 in our illustation) of the Xtrain(1) columns and see which of *those* has the highest information gain/gini impurity loss vis a vis the outcome. And so we create the root node and leaves based on *that* column. Then going to the left leaf we pick a random subset of size ~ √ncolumns (√4 in our illustation) of the remaining columns, and find the one which has the highest information gain/gini impurity loss, and make another branch of the decision tree based on *it*. And then we go to the right leaf and do the same: pick a random subset of size √ncolumns of the remaining columns, find the one which has the highest information gain/gini impurit loss, and make another branch of the decision tree. And we just repeat until we’re done with the tree. We would probably keep restriction that we need at least some minimum number of data points per leaf, say 20.

**Create another Bootstrapped Decision Tree**

Then we create another decision tree using the same method. We randomly sample our original data set nrows times, to get another Xtrain(2), ytrain(2) and Xtest(2), ytest(2). Note that Xtest(2) and ytest(2) will randomly vary in size, but Xtrain(2) and ytrain(2) will always be fixed in size, to nrows.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa1 | xb1 | xc1 | xd1 | y1 |
| xa2 | xb2 | xc2 | xd2 | Y2 |
| xa8 | xb8 | xc8 | xd8 | y8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa3 | xb3 | xc3 | xd3 | y3 |
| xa3 | xb3 | xc3 | xd3 | y3 |
| xa4 | xb4 | xc4 | xd4 | y4 |
| xa5 | xb5 | xc5 | xd5 | y5 |
| xa5 | xb5 | xc5 | xd5 | y5 |
| xa6 | xb6 | xc6 | xd6 | y6 |
| xa7 | xb7 | xc7 | xd7 | y7 |
| xa9 | xb9 | xc9 | xd9 | y9 |
| xa10 | xb10 | xc10 | xd10 | y10 |
| xa11 | xb11 | xc11 | xd10 | y11 |
| xa10 | xb10 | xc10 | xd10 | y10 |
| xa12 | xb12 | xc12 | xd12 | y12 |

And then we do as before. We pick a subset of ~ √ncolumns columns from Xtrain(2), find the one most correlated with the outcome, and start a decision tree off of it. Then for the left leaf, we pick a subset of size ~ √ncolumns from the remaining columns, find the column out of that subset that is most correlated with the outcome, and make another branch of the decision tree. And we do likewise for the right leaf. Etc., until we’re done with the tree. Obviously, the number of columns left to choose from diminishes by one each level we go down the tree, and I assume once we get to the point where the number of columns left to pick from, along a particular branch, is less than √ncolumns, then we just choose all of them.

**Create another Bootstrapped Decision Tree, etc.**

And then we make another random sample Xtrain(3), ytrain(3) and Xtest(3), ytest(3), etc., and build another tree. We might do this hundreds of times.

**Create different classes of Bootstrapped Decision Trees, etc.**

And we can also go back and change our model by increasing or decreasing the number of columns of Xtrain we put into the subsets. Typically we’ll do √ncolumns ± 1, √ncolumns ± 2, etc. We might do hundreds of Trees for each new class.

**Predictions for data**

To make definitive predictions on the data row, we’d run the row through *every* model, and take as our official prediction whatever the majority outcome prediction is.



This is as before. For instance, we might have something like,



Looks like sklearn also uses this result to estimate probabilities of yes’s and no’s. To take our illustration as an example, I think it just does,



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.

**Testing the model**

And we can asses our model’s accuracy on the ‘training data’. We’d take row of the data and run it through every model for which that row appeared in Xtrain, collate the predictions, and determine the one with the highest vote. We can do this for every row. And we could build a classification matrix based off of this. To test our model on the ‘testing data’, we’d take a row of the data and run it through every model for which that row appeard in Xtest, collate the predictions, and determine the one with the highest vote. And we’d do this for every row. And then we’d build a classification matrix off of this.

**Predictions for new data with missing values using Random Forest of Decision Trees**

Say we want to make a prediction of the outcome (y) of a row with missing data, like xc.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| xa | xb | xc = ? | xd | y = ? |

If the outcome is binary, one approach we can use is to:

1. Assume y = yes, and use the methods in the Missing Data file to guess the most likely xc. Then run the data (xa, xb, xc, xd) through the random forests and tabulate how many times the prediction matched y = yes.
2. Assume y = no, and use the methods in the Missing Data file to guess the most likely xc. Then run the data (xa, xb, xc, xd) through the random forests and tabulate how many times the prediction matched y = no.
3. Whichever postulated outcome, y = yes or y = no, got the most matches, is the one we go with.

**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***

Let’s consider a data table from one of the Decision Tree files. 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 |

*First Bootstrapped Decision Tree*

So a random sample (Internet random # generator) returns the following rows: 5367123, which corresponds to this table, basically just replace row 4 with a copy of row 3. So this is our Xtrain, ytrain.

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

And our Xtest, ytest would consist of row 4.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| No | Yes | 35 | Yes |

Now apropos the training set, choose √3 ~ 2 columns at random. We’ll say Loves Popcorn and Loves Soda. Which has the largest Gini Impurity Loss: GIL(A) = GI(Y) – GI(Y|A)? Gotta calculate GI(Y|A) for each column and whichever has the smallest GI will have the largest GIL. So,



and GI(C|S) is,



So we should start with S column. And our Decision Tree so far would be, including the root node:

Diagram

Description automatically generated

Same as last time, actually, even with row 4 taken out and row 3 duplicated. And that’s ‘cause the only difference between rows 3 and 4 is the Age. Maybe not the most illustrative example then. Oh well. So then moving on to further work out the left leaf. I’ll highlight the rows we’re dealing with.

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

And we want to pick ~ √3 ~ 2 columns out of the two left. So that’s all of them, Loves Popcorn and Age. And we want to find which of them has the lowest Gini impurity. So first we’ll calculate GI(C|P). This is:



And next we have to do it for age. Can clearly see that if we split by age with a critical age of (7+18)/2 = 12.5, we will get a perfect correlation with Loves Cool as Ice.



So we’d choose to split by age, since it has the lowest GI. I’ll not concern myself with the fact that we have just one person in one of our leaves now. And our Decision Tree looks like,

Diagram

Description automatically generated with medium confidence

which is, coincidentally, the tree we got last time.

*Second Bootstrapped Decision Tree*

Now we do another random sample (Internet random # generator) of the rows in our original data table: 1245265 which corresponds to this table. So this is our Xtrain, ytrain.

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

and this is our Xtest, ytest (i.e., rows 3 and 7):

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| No | Yes | 18 | Yes |
| No | No | 83 | No |

Now apropos the training set, choose √3 ~ 2 columns at random. We’ll say Loves Popcorn and Age. And now we’ll calculate their GI values.



Its proximity to 0.5 means Popcorn is almost completely randomly associated with Loves Cool as Ice. And now we’ll do age. Technically we have to split by all possible critical ages: (7+12)/2 = 9.5, (12+35)/2 = 23.5, (35+38)/2 = 36.5, (38 + 83)/2 = 60.5, and see which split results in the lowest GI. But I think its fairly evident from the table that the (12+35)/2 = 23.5 split works best, as it seems to track best with Loves Cool as Ice. So using this Agecritical = 23.5, we have:



So Age wins. So splitting the root node by Age, we have so far:

Diagram

Description automatically generated

So then moving on to further work out the left leaf. I’ll highlight the rows we’re dealing with.

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

And we want to pick ~ √3 ~ 2 columns out of the two left. So that’s all of them, Loves Popcorn and Loves Soda. And we want to find which of them has the lowest Gini impurity. Looking at it, it’s clear this will be Loves Soda, as it’s perfectly correlated with Loves Cool as Ice. So splitting off of this, we have:

Diagram

Description automatically generated with medium confidence

*Create another Bootstrapped Decision Tree, etc.*

And we would then do another random sample, and repeat the process.

*Create different classes of Bootstrapped Decision Trees, etc.*

We put √3 ~ 2 columns in our subsets. We could go back and put 3 in – which would be all of them. Or we could go back and put 1 in – which would be pretty simple as in no need to calculate GI values.

*Testing Random Forest of Decision Trees*

Finally, to test our model, we would take a row of data from X, y, and test it on all the models for which it appears in Xtest, ytest. We’d obviously have to create a lot more decision trees for this to be a viable thing. But so far, we could test row 4 in the first DT, and we could test rows 3, 7 in the second DT.

*Predictions using Random Forest of Decision Trees*

To make actual predictions on *new* data, we’d run the data through *every* model, and take as our official prediction whatever the majority outcome prediction is.



So for instance, insofar as we’ve progressed, say we have a 44 yr old person who loves popcorn, but not soda.

Diagram

Description automatically generated with medium confidence Diagram

Description automatically generated with medium confidence

Then both DT’s predict this person Does Not Love Cool as Ice. So we’d take our official prediction to be Does Not Love Cool as Ice. Both trees are clearly working.

**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.

**Feature Importance**

Just like with regular decision trees, we can calculate the feature importance of columns in our bootstrapped/random forest model. The importance of a feature is more or less given by how correlated it is with the outcome. Per tree, the way it’s assessed is by looking at the tree and calculating the information gain when splitting on that feature. For instance, consider this node,

A diagram of a number

Description automatically generated with medium confidence

splitting on thai\_7.0 < 0.5 results in an information gain of,



We can do likewise for all the features, and then assign each feature a weight,



This is would be the feature importance for a given tree. Then to assess the importance across our entire bag/random forest, we’d say,



We can do likewise for regression models. We’d just use its own definition of information gain, which, as I recall, involves the improvement in the sum of the squared errors.

**Exploring the Model and Hyperparameters**

The RandomForrestClassifier in sklearn has a couple new hyperparameters as pertains the bagging methodology discussed above. And it also has pretty much all the old hyperparameters that the DecisionTreeClassifier had. I guess I’ll just focus on the new ones. But first, here’s the model, with its default settings on a pure N = 500 dataset.

A diagram of a forest

Description automatically generated A diagram of a forest

Description automatically generated

and an impure (10% outliers) dataset,

A diagram of a random forest

Description automatically generated A diagram of a random forest

Description automatically generated

And same apropos the three-class guys,

A diagram of a forest

Description automatically generated A diagram of different colored dots

Description automatically generated

A diagram of different colored dots

Description automatically generated A map of different colored dots

Description automatically generated

If we compare to the regular DecisionTreeClassifier, we can see it is actually doing a lot better apropos overfitting the outliers. Makes you wonder how much better SVC would do if we tried a simple bagging approach (which we can do in sklearn). Well, probably not a lot better actually. SVC didn’t have a problem with outliers (unless γ were really large). It’s really the Decision Tree that has the outlier problem. Let’s try this out on a regresion model,

A graph with a line graph and numbers

Description automatically generated with medium confidence A graph with a line graph and red dots

Description automatically generated with medium confidence

Does look like Random Forest is tamping down on the outliers. Well it is a little, but problem is that I have so few data points, the outliers are not substantially outnumbered by the legit data points, and so the outliers do factor in substantially to the prediction.

**Hyperparameter: n\_estimators**

This is the number of decision trees that go into our random forrest. The default is 100. So here’s our quadratic surface, N = 500, with a couple different n\_estimator values.

A graph of a forest plot

Description automatically generated with medium confidence A diagram of a graph

Description automatically generated with medium confidence A diagram of a graph

Description automatically generated with medium confidence

Here’s our 3-class circles guy (15% outliers),

A diagram of a forest

Description automatically generated A map of different colored circles

Description automatically generated A map of different colored dots

Description automatically generated

Can see that the tree is following the outliers less and less as n\_estimators increases.

A graph with a line graph and numbers

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

Description automatically generated A graph with red and blue dots

Description automatically generated

And same here. Doing the random forrest technique seems to work out a little better than just setting a minimum\_leaf\_sample criterion.

**Hyperparameter: bootstrap**

This determines whether we sample our rows with replacement (bootstrap=True) or w/o. The default is True. Here’s the plot again. I’ll do n\_estimators = 500, and bootstrap = True/False.

A diagram of a forest

Description automatically generated A diagram of a forest

Description automatically generated

True does seem to work a *little* better in this case. And let’s do on 3-class,

A diagram of a forest

Description automatically generated with medium confidence A map of different colored circles

Description automatically generated

And let’s try on regression,

A graph of a line graph

Description automatically generated A graph of a line graph

Description automatically generated

Again, True does seem to work better. I would imagine True works better because you want to weight the outliers according to their numbers in the actual total training data set. And most of the time, the models will miss them, because outliers are rare. But if you sample w/o replacement, then you’ll be mostly removing non-outliers, and thereby artificially enhancing the probability of catching an outlier in the next random sample. So the outliers will attain a larger influence on the models, down the line, than they should have.

**Hyperparameter: max\_samples**

This prescribes the number of rows we sample from our dataset. The default value is None, which means it will sample all rows by default. I have 500 rows in my dataset. Let’s try a few max\_sample values,

A diagram of a forest

Description automatically generated A diagram of a forest

Description automatically generated A diagram of a forest

Description automatically generated with medium confidence

The first plot has too few samples and so can’t follow the curve well. The last one has too many, relatively, and is overfitting. Clearly there is a happy middleground. Let’s do it on the three-class guy,

A diagram of different colored circles

Description automatically generated A diagram of different colored circles

Description automatically generated A diagram of a forest

Description automatically generated with medium confidence

And let’s try regression,

A graph with red and blue dots

Description automatically generated A graph with red and blue dots

Description automatically generated A graph with a line graph and red dots

Description automatically generated with medium confidence

This also seems to be efficacious. And looks like there is a happy middle ground. In fact max\_samples = 25, works pretty well. So if max\_samples is too low, then you’re not getting enough of the data to construct a good model. But if it’s too high, then the likelihood of you getting outliers in your model increases, and so skews the predictions as well.

**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. For instance, here’s the three-class guy with ccp\_alpha tuned,

A diagram of a number of dots

Description automatically generated with medium confidence

And this is a pretty good fit – could argue that the leakage of the blue guy makes sense given the data (and all the algorithms so far have wanted to do this).