UNIVERSITY OF WISCONSIN
DATA SCIENCE

DS 740

Data Mining

Boosting

Important note: Transcripts are **not** substitutes for textbook assignments.

Learning Objectives

By the end of this lesson, you will be able to:

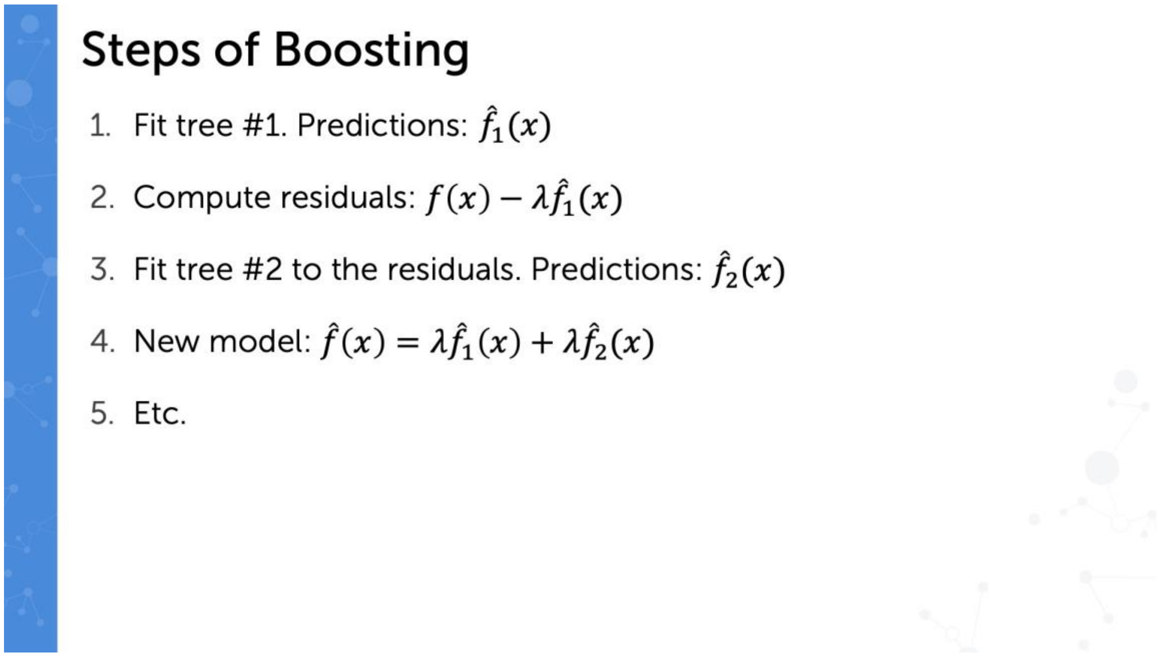
- Explain how boosting works.
- Select reasonable initial values for the tuning parameters for boosting.
- Perform boosting in R.

Boosting

Bagging: fit multiple trees, based on *different subsets of data*.

Boosting: fit multiple trees, *sequentially*.

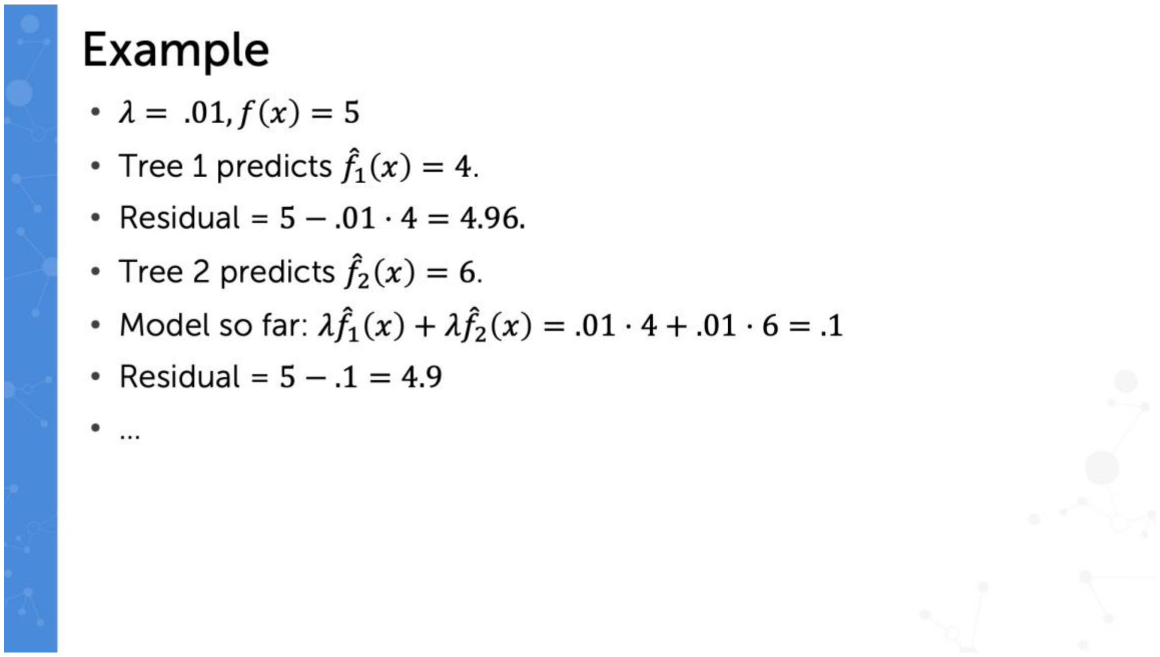
Boosting is similar to bagging in that we fit multiple models and use them to create a composite model with lower variance than any individual model. Also like bagging, boosting can be applied to many different data mining algorithms, but it's especially useful when applied to decision trees. The difference between these two methods is that in bagging, the order of the trees we fit was random. Whereas in boosting, we fit multiple trees sequentially. Each tree that we build depends on the previous trees.



Steps of Boosting

1. Fit tree #1. Predictions: $\hat{f}_1(x)$
2. Compute residuals: $f(x) - \lambda\hat{f}_1(x)$
3. Fit tree #2 to the residuals. Predictions: $\hat{f}_2(x)$
4. New model: $\hat{f}(x) = \lambda\hat{f}_1(x) + \lambda\hat{f}_2(x)$
5. Etc.

The way boosting works is we'll start by fitting a single tree. And getting some predictions, call them f hat sub 1. Then we'll compute the residuals by taking the true response values f minus our predictions f hat sub 1 times a shrinkage parameter lambda. Then we'll fit a second tree to the residuals, getting us a new set of predictions f hat sub 2. And we'll create a new model by taking the sum of the predictions from tree 1 and tree 2 times our shrinkage parameter. We'll continue this process by fitting trees to the residuals and creating a new model formed out of the sum of the f hats times our shrinkage parameter.

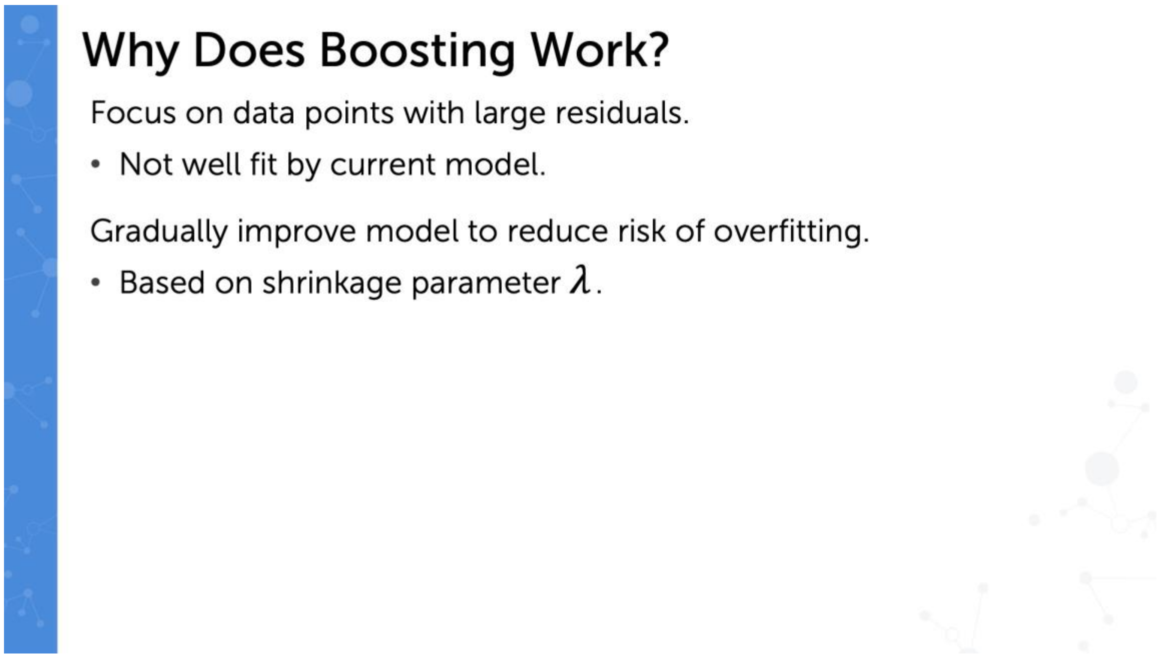


Example

- $\lambda = .01, f(x) = 5$
- Tree 1 predicts $\hat{f}_1(x) = 4$.
- Residual = $5 - .01 \cdot 4 = 4.96$.
- Tree 2 predicts $\hat{f}_2(x) = 6$.
- Model so far: $\lambda \hat{f}_1(x) + \lambda \hat{f}_2(x) = .01 \cdot 4 + .01 \cdot 6 = .1$
- Residual = $5 - .1 = 4.9$
- …

For example, suppose that lambda equals 0.01. And for a particular data point, the true value of the response variable, f of x equals 5. Suppose that our first tree predicts a response value for this data point of 4. Then the residual will be the true response value minus lambda times the predicted value so 5 minus 0.1 times 4, or 4.96. Then, we build another tree to try to predict the residual, which is 4.96.

Suppose that this second tree predicts a response value of 6. Then our model so far for this data point is the sum of lambda times the predicted values from each of trees one and two. So 0.1 times 4 plus 0.01 times 6, giving us a predicted value of 0.1. So the residual for this data point is now 5 minus 0.1, or 4.9. We would continue in this way building tree three to try to predict this residual of 4.9.



Why Does Boosting Work?

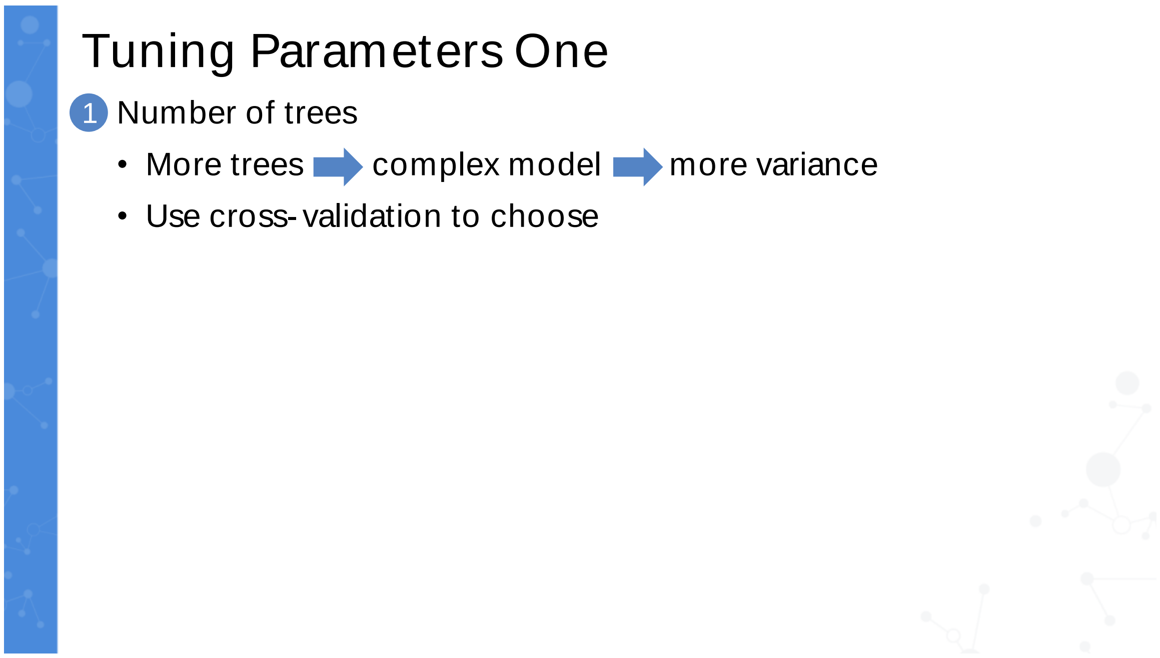
Focus on data points with large residuals.

- Not well fit by current model.

Gradually improve model to reduce risk of overfitting.

- Based on shrinkage parameter λ.

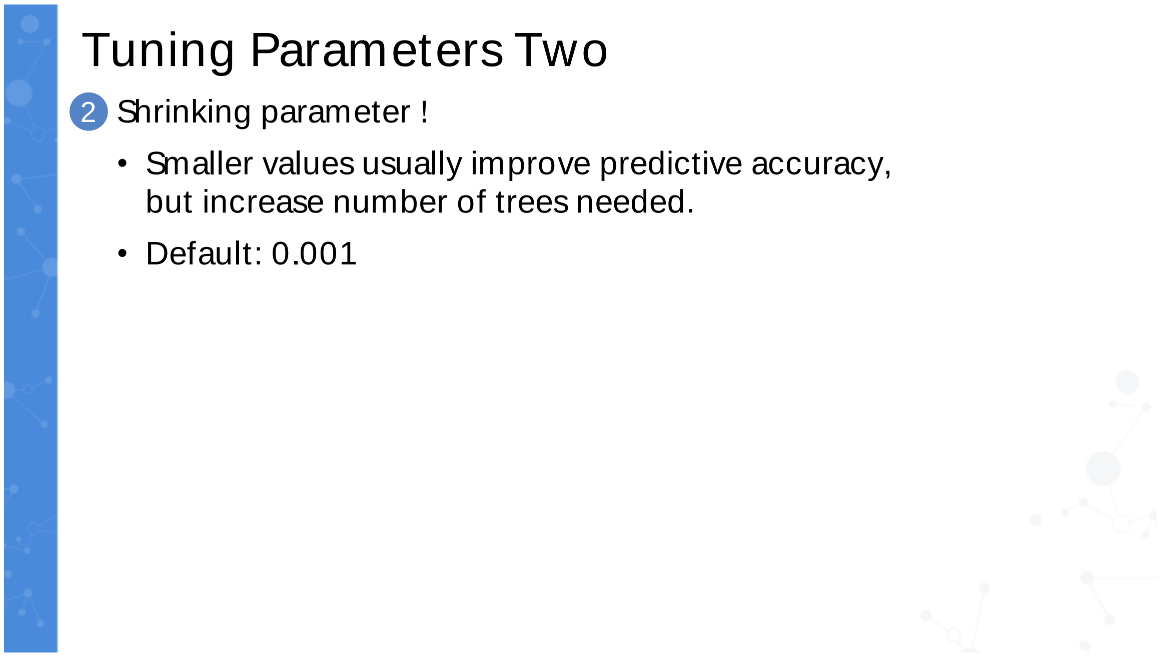
The idea of boosting is to focus on data points that have residuals that are large in absolute value, that is, data points that are not well fit by the current model so you can build a new model that fits them better. Using our shrinkage parameter lambda, we're improving the model gradually to sneak up on the best possible model. This sort of slow learning approach is especially good when you have a large number of predictor variables.



Tuning Parameters One

1. Number of trees
   - More trees $\rightarrow$ complex model $\rightarrow$ more variance
   - Use cross-validation to choose

In boosting, there are three tuning parameters to choose using cross validation. The first is the number of trees. The more trees you use, the more complex your model is because you're summing more f hats, which can increase the variance of your model. So unlike with bagging, in boosting it's a good idea to choose the number of trees using cross validation.



Tuning Parameters Two

2 Shrinking parameter !

- Smaller values usually improve predictive accuracy, but increase number of trees needed.
- Default: 0.001

The second parameter is the shrinkage parameter lambda. Smaller values of lambda tend to be better for predictive accuracy, but they increase the number of trees that you need which makes the boosting analysis take longer. One recommendation is to choose lambda to be small enough so that you need at least 1,000 trees before the cross-validation error starts to increase. The default value in R is lambda equals 0.001.

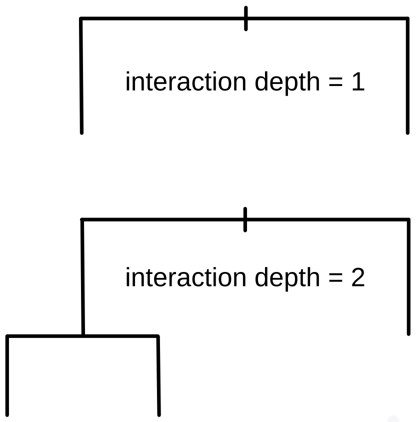
Notes:

For more about choosing lambda, see Generalized Boosted Models: A guide to the gbm package - Greg Ridgeway (PDF) and A working guide to boosted regression trees - J. Elith, J. R. Leathwick, T. Hastie

Tuning Parameters Three

3 Interaction depth

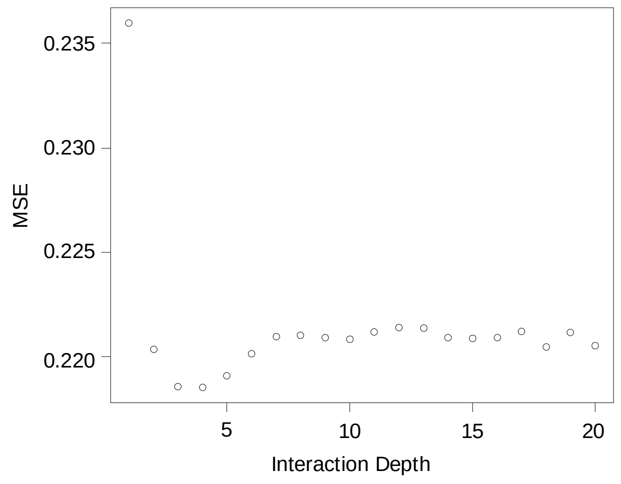
- Number of splits
- Generally < 10
- Consider complexity of the interactions you expect



The third tuning parameter is the interaction depth or the number of splits in each tree that the boosting analysis produces. To choose the interaction depth, you can use cross validation. And to decide on an initial guess, consider the complexity of the interactions you expect among the predictor variables.

For example, if you expect that the true model is additive with no interactions among the variables, then it makes sense to start with an interaction depth of 1, where each tree is a stump. It's rare to have an interaction depth of 10 or more produce any significant improvement in the model compared to values less than 10. In particular, small datasets, those with less than about 250 data points, generally don't benefit much from an interaction depth of more than 3.

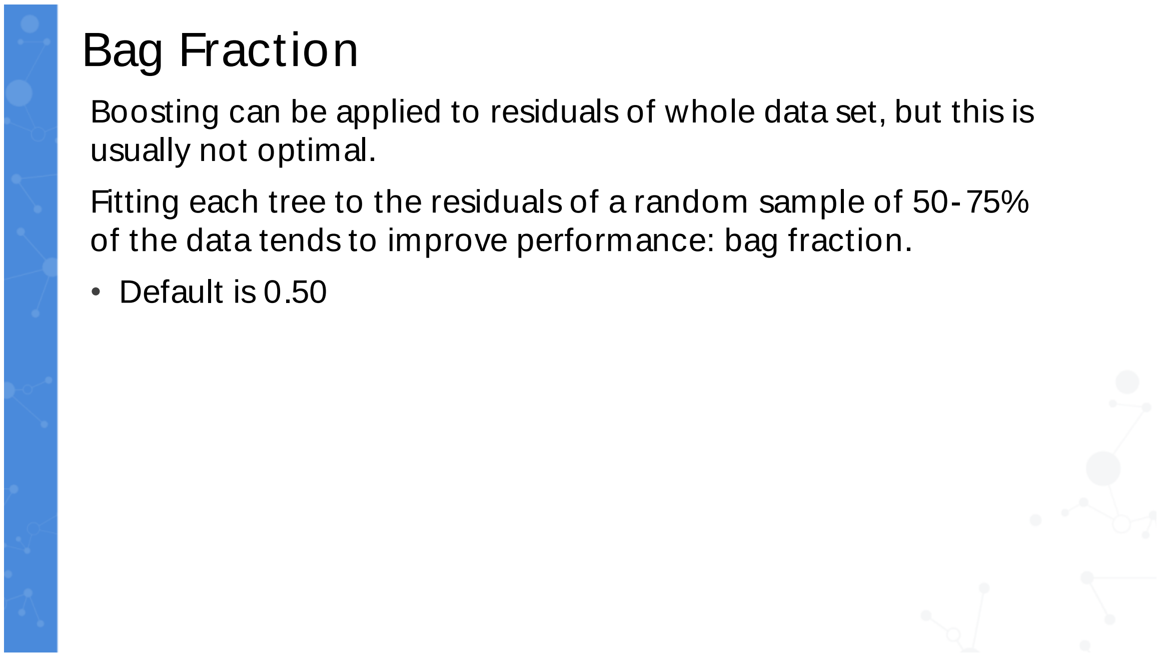
Interaction Depth



If the true underlying model has no interactions or very simple interactions, it is possible for the error to be higher with higher interaction depths.

Notes:

See Fig. 8.11 in our textbook.



Bag Fraction

Boosting can be applied to residuals of whole data set, but this is usually not optimal.

Fitting each tree to the residuals of a random sample of 50-75% of the data tends to improve performance: bag fraction.

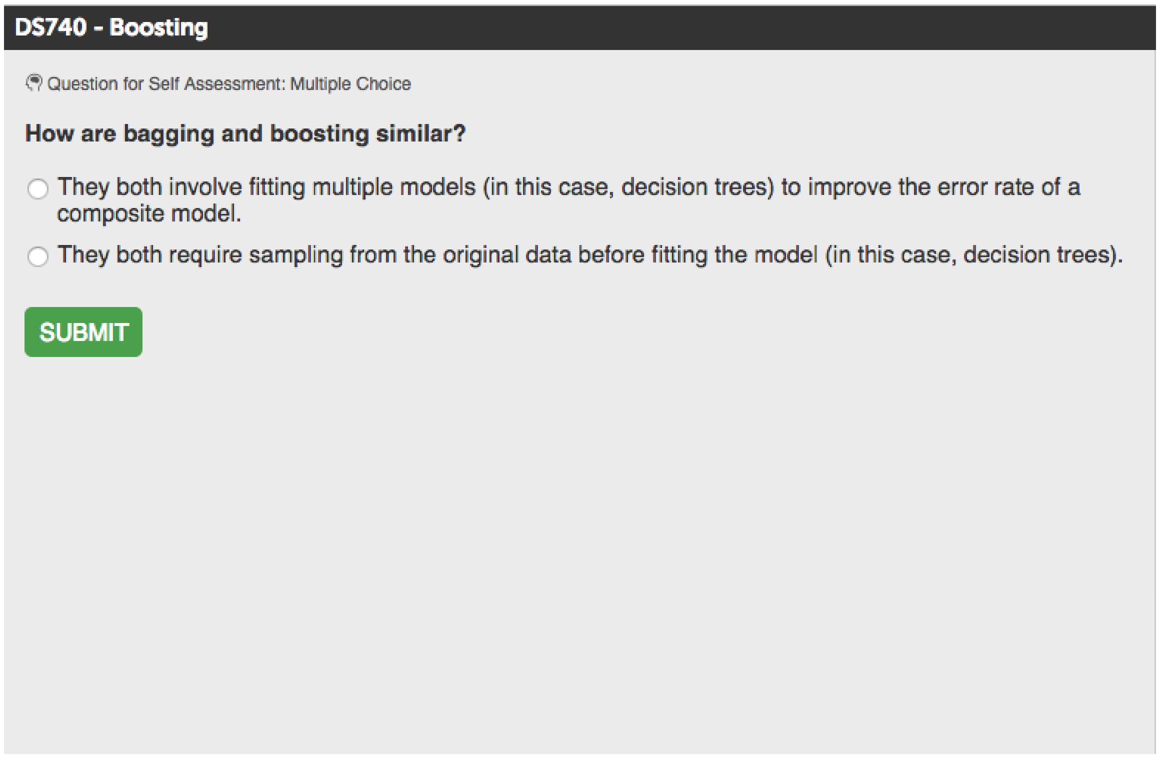
- Default is 0.50

Unlike bagging, in boosting, each tree can be fit to the residuals of the whole dataset. However, fitting each tree to the residuals of a random sample of 50% to 75% of the data tends to improve the performance. We call this percentage the bag fraction. The default in R is a bag fraction of 50%

Notes:

For more about the bag fraction, see Stochastic gradient boosting - Jerome H. Friedman

Question 1


DS740 - Boosting

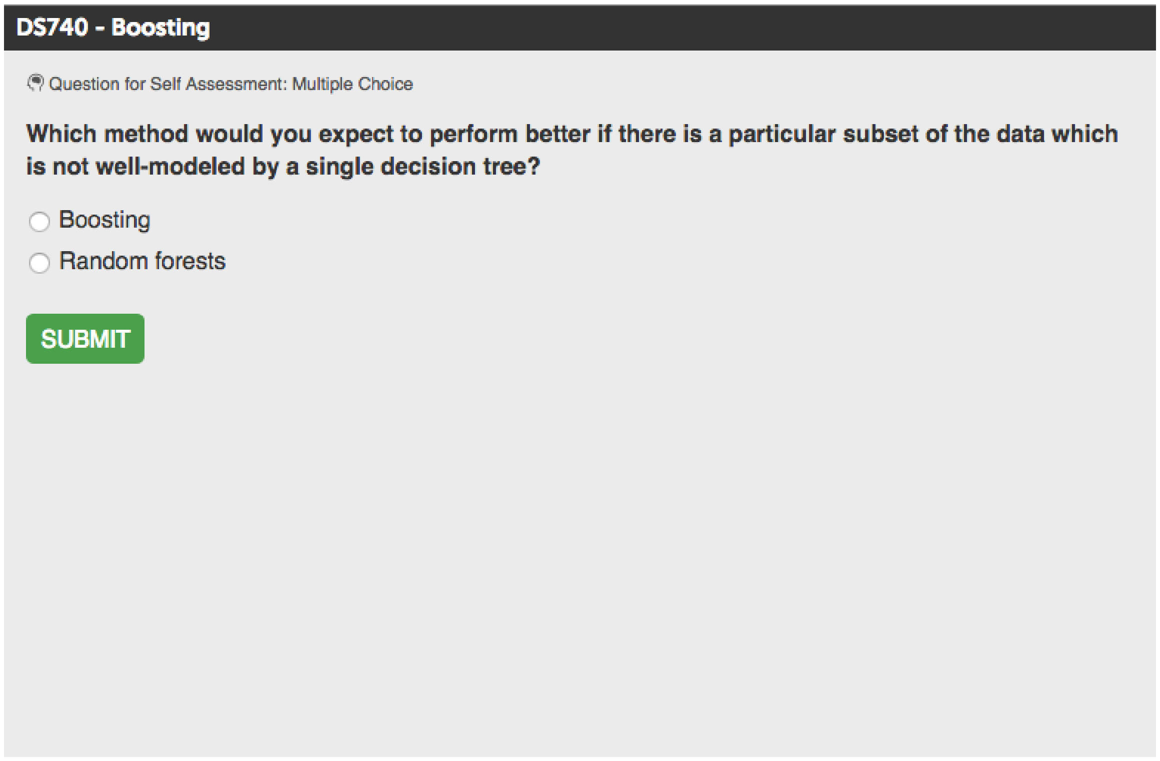
Question for Self Assessment: Multiple Choice

How are bagging and boosting similar?

- ◯ They both involve fitting multiple models (in this case, decision trees) to improve the error rate of a composite model.
- ◯ They both require sampling from the original data before fitting the model (in this case, decision trees).

Answer is at the end of the transcript

Question 2


DS740 - Boosting

Question for Self Assessment: Multiple Choice

Which method would you expect to perform better if there is a particular subset of the data which is not well-modeled by a single decision tree?

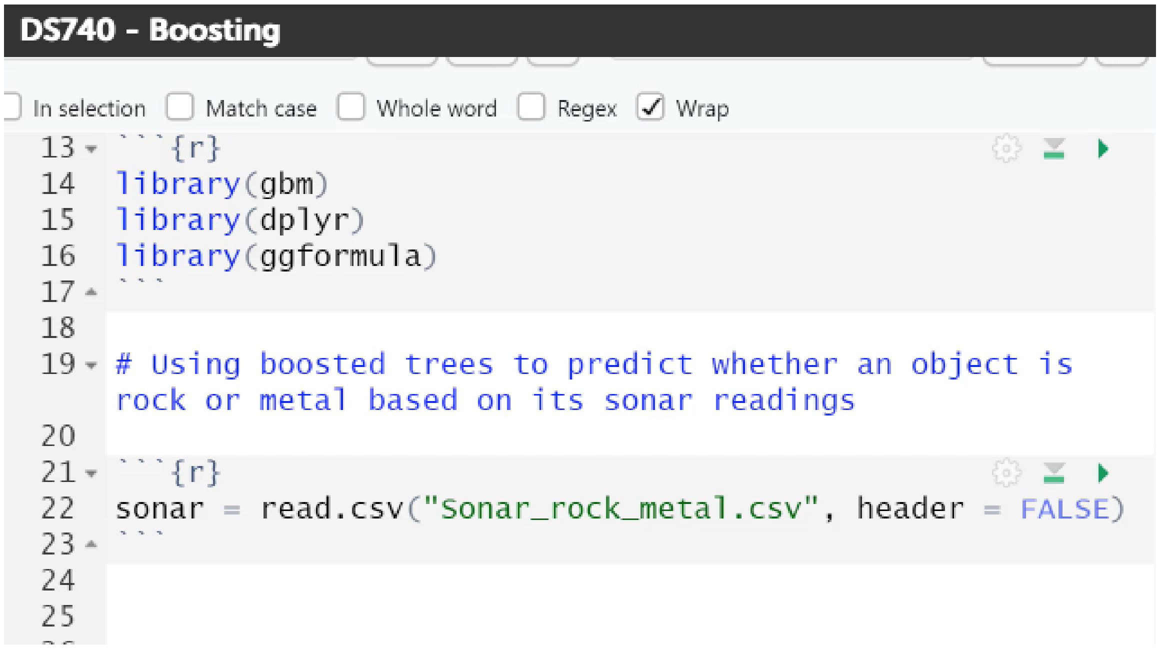
- ○ Boosting
- ○ Random forests

Answer is at the end of the transcript

Sample Problem

Use boosting to predict whether an object is made of rock or metal based on sonar data.

```
> sonar = read.csv("Sonar_rock_metal.csv", header=F)
> summary(sonar)
       V1                V2                V3
 Min.   :0.00150   Min.   :0.00060   Min.   :0.00150
 1st Qu.:0.01335   1st Qu.:0.01645   1st Qu.:0.01895
 Median :0.02280   Median :0.03080   Median :0.03430
 Mean   :0.02916   Mean   :0.03844   Mean   :0.04383
 3rd Qu.:0.03555   3rd Qu.:0.04795   3rd Qu.:0.05795
 Max.   :0.13710   Max.   :0.23390   Max.   :0.30590
```



~~~
```{r}
library(gbm)
library(dplyr)
library(ggformula)
~~~

Using boosted trees to predict whether an object is rock or metal based on its sonar readings

```{r}
sonar = read.csv("Sonar_rock_metal.csv", header = FALSE)
```
```

This slide represents a video/screencast in the lecture. The transcript does not substitute video content.

Let's use boosted trees to predict whether objects are rock or metal based on their sonar readings. I've already loaded the package gbm, which stands for Gradient Boosting Machine, which we'll use to build the boosted trees. I've also loaded the packages dplyr and ggformula, which we'll use to manipulate the data set and make some graphs.

I'm going to read in the data set using read.csv with the argument, header equals false, because the sonar rock metal data set doesn't have a header row containing column names. You'll recall that the functions tree for a single decision tree and random forest both expect the response variable for a classification problem to be a factor variable. In contrast, gbm expects the response variable to be numeric, even when we're doing classification.

So we need to start by redefining our response variable, v61, as a numeric variable. I'm doing that using mutate and case_when, and I'll let 1 represent medals and 0 represent rocks. So here I'm defining a new variable called "object" to store the numeric version of v61. And I'm including this additional row in my case_when, which simply takes anything that's not a metal or rock and turns it into NA for missing value, with the specific type of NA being a real number to fit with the numeric values that we're using for metal and rock. And this is just a safeguard in case there's anything in the data set that I wasn't expecting that's not a metal or rock.
```

After I've created my new numeric version of the response variable, I'm piping the results into the select function and using minus v61 to remove the variable v61 from the data set. This is because our new variable object will be perfectly associated with v61, so if we left v61 in there, we could get perfect predictions, which isn't realistic. Alternatively, instead of removing v61 from the data set, we could exclude it from the analysis by changing our formula that we're going to use in GBM to have period to indicate all of the other variables being predictors, but then minus v61 to exclude v61 at the time that we're creating the model.

To build our boosted trees, we can use the function gbm, which stands for Generalized Boosted Model. Because I removed the variable v61 from the data set, I can use the formula object, squiggle dot, where the dot indicates that I want to use everything in the data set except for the response variable object as a predictor variable. This is a classification problem, so we'll use distribution equals bernoulli. If we were instead predicting a quantitative response variable, we'd use distribution equals gaussian.

Here I'm going to use 1000 trees, a shrinkage parameter, lambda, of 0.001, and an interaction depth, or depth of the tree, of 3. If we wanted to get predictions for each of our data points, we could do that using the predict function, using n.trees equal to the same number of trees we use to build the model and the argument type equals response.

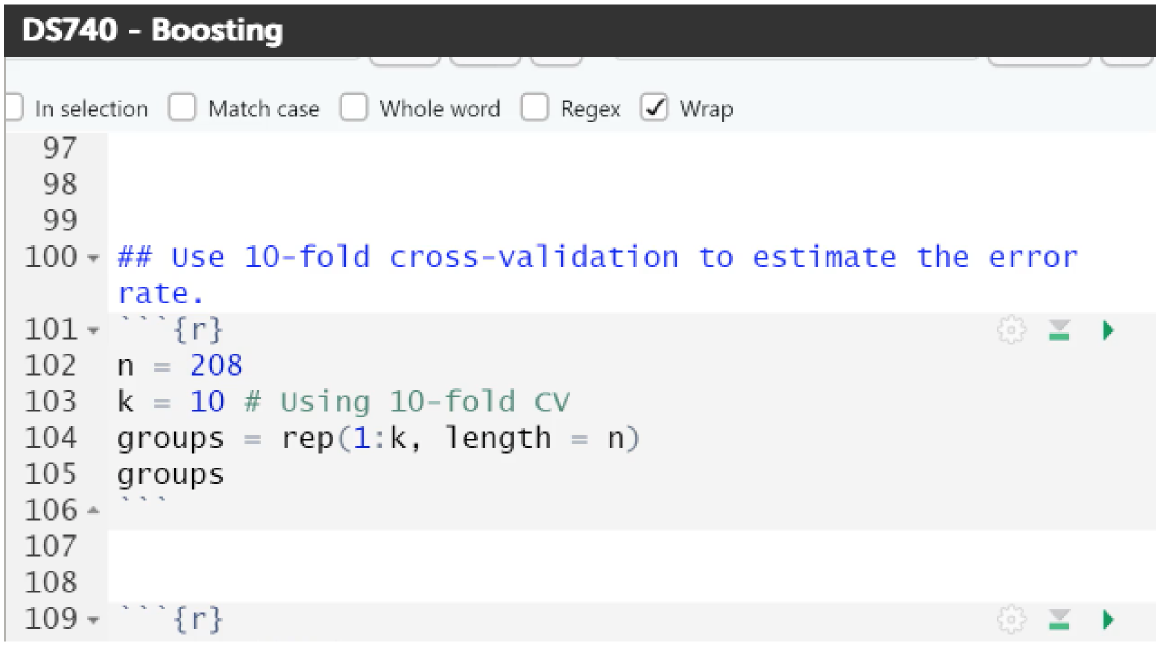
But remember that when you're making predictions for a data set that matches the data set that you use to build the model, your predictions are going to tend to be more accurate than they would be on new data. So this might give you an overly rosy sense of how well your model is doing. If we just look at the contents of our variable where we stored the model, in this case, boost, we get a brief summary of our model, reminding us that we used 1000 trees with a bernoulli loss function-- so we did classification-- and that there were 60 predictor variables.

We can get a bit more information by using the summary function. This shows us a list of the variables that were used as predictors, sorted by their relative influence or importance. And we can also see a graph of their importance as shown here. This graph can be pretty hard to read if there are a lot of predictor variables, so it might be helpful to simplify it down to just the top 10 most important predictor variables.

So we can do that using dplyr to select the top 10 variables from that list, and then using the gf_col function from ggformula to plot the relative influence as a function of which variable it is. So gf_col will produce a bar plot or column plot, but the difference from gf_bar is that the heights of the bars are given by the variable that's shown first in the formula, not by how many times a particular value occurs in the variable that happens on the right-hand side of the formula.

If we just did this, then the order of the bars gets reordered, so we're no longer looking at them in strictly decreasing order of relative influence. So I think the graph is easier to read and compare if we first reorder the variable "var," which, remember, tells the name of the predictor variable, in order of decreasing relative influence. So this shows us that V11 and V12 are the most important variables in this boosted tree model, similar to what we saw in the random forest model.

We can also view the marginal effects of our predictor variables, so this is similar to the partial dependence plot that we saw for random forests. For our variable V11, the most important variable, we get a very similar result to the partial dependence plot from random forests, showing that the y variable or response is positively associated with V11. In contrast, if we look at the marginal effect of V36, another one of our important predictor variables, we see that the y variable is negatively associated with V36. Remember that we coded the response variable as 1 being metal and 0 being rock, so this is showing that data points with larger values of V36 are more likely to be rocks.



~~~
## Use 10-fold cross-validation to estimate the error rate.
```{r}
n = 208
k = 10 # Using 10-fold CV
groups = rep(1:k, length = n)
groups
~~~

```{r}
```

This slide represents a video/screencast in the lecture. The transcript does not substitute video content.

Let's use 10-f cross-validation to get an honest assessment of how well our model will do on new data. We'll start with a little bit of setup, letting n be the size of the data set, k being the number of folds, so I'll do 10-fold cross-validation. And we'll let groups be a vector of values from 1 up to k, so in this case 1 up to 10, with length equal to n.

So you can see that groups simply contains the numbers 1 through 10 repeated over and over until we get a length of 208, the same as the size of our data set. Next we'll set the seed and use the sample function to reorder our group's variable. And we'll initialize a variable called boost_predict to store the predictions for each of our data points. So this is a vector with length equal to n.

Then we have a for loop where we're iterating over the different folds. And we'll set group i to be a vector of trues and falses where it's true for all of the values where cvgroups is equal to ii, the index of which fold we're looking at.

Next, we want to perform boosting on everything that's in the current training set, that is everything that's not in groupi, and then use that model to predict the values for groupi. So to do this, I'm going to go back up and copy the code that we used before to build our boosted model.

There are a couple of things we need to change. One is that, instead of building our model using the whole data set, we instead just want to use everything that's not in groupi. Oops. So that would be exclamation point groupi to negate all of the trues and falses so that everything that was false in groupi, mean it was not in the current fold, now becomes true so it will be used in my data set. Then we want to only make predictions for the values that are in group i. So oops.

Here we'll use new data equals data used square bracket groupi to only make predictions for the values where groupi equals true. And we want to store these predictions in our vector boost_predict. Specifically, we want to store them in the positions of boost predict that correspond to the data points that we're making predictions for.

Then we can go ahead and run this code. This will take a little bit of time because we need to build 1,000 trees 10 times. If we look at the first five values of boost_predict, we see that what this type equals response argument is doing is giving us the predicted probability of belonging to the metal category, which was the category that we assigned as a value of 1.

So now we can build a confusion matrix. Because we got the predictions as probabilities rather than a classification m or f, we need to use some sort of threshold to decide how we're classifying the data points in our confusion matrix. A threshold of 0.5 is a common value to use, but you could vary this using a rock curve or cross-validation.

So here's our confusion matrix. And as we've done in the past, we can sum the diagonal elements and divide by our total data set size to get the accuracy and then take 1 minus that to get the classification error rate. So in this case, we have an honest error rate of about 23%.

An alternative use of cross-validation is to test different values of a tuning parameter, such as the interaction depth, and decide which one is the best. We can do this by taking the same code that we used to do cross-validation for model assessment and wrapping it in outer for loop that iterates over possible values of the tuning parameter. In this case, the depth.

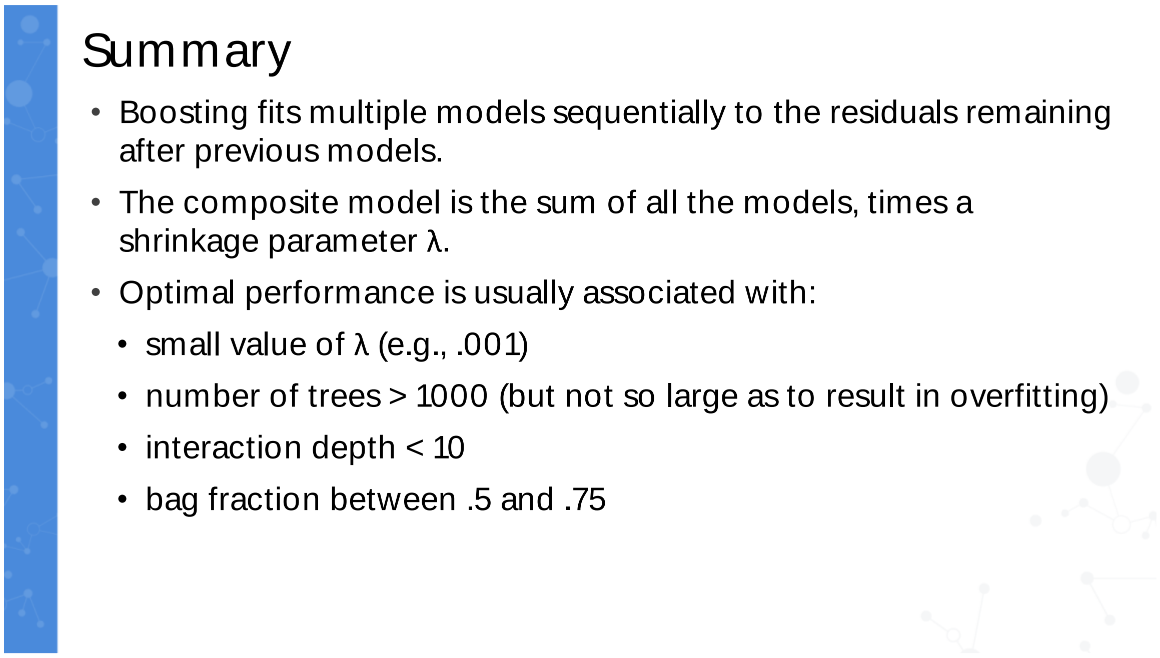
We'll use the same setup as before with two additional pieces of information. One is creating a vector of depths that we want to test. So I'll test tree sizes from 1 up to 20. And I have an additional vector here to store the error from each value of the interaction depth.

Down at the bottom of my for loop-- so this is between the two loops. So I've already completed iteration over all 10 folds for a given interaction depth but I haven't yet completed iteration over all the interaction depths. Here is where I'm computing the confusion matrix for that interaction depth and storing the error as the jj-th position in my error vector.

Here we're testing 20 different values of interaction depth so this code will take approximately 20 times as long to run as our initial cross-validation where we were only using a single interaction depth. Once our cross-validation has finished running, we can use the function gf_point to make a point graph of the error as a function of the depth.

For this data set, we see that the highest error occurred at the beginning when the depth of the tree was one or two. After that, it looks like the error rate levels out at around 22%. A tree depth of 15 did give us the lowest error rate in this case. But that error rate is pretty close to the error rates of the surrounding depths so it seems likely that if we used a different random seed, we would get a different optimum depth. In this case, I'd say that a tree depth of about four is probably going to be good to give us a near lowest error rate with simple trees.

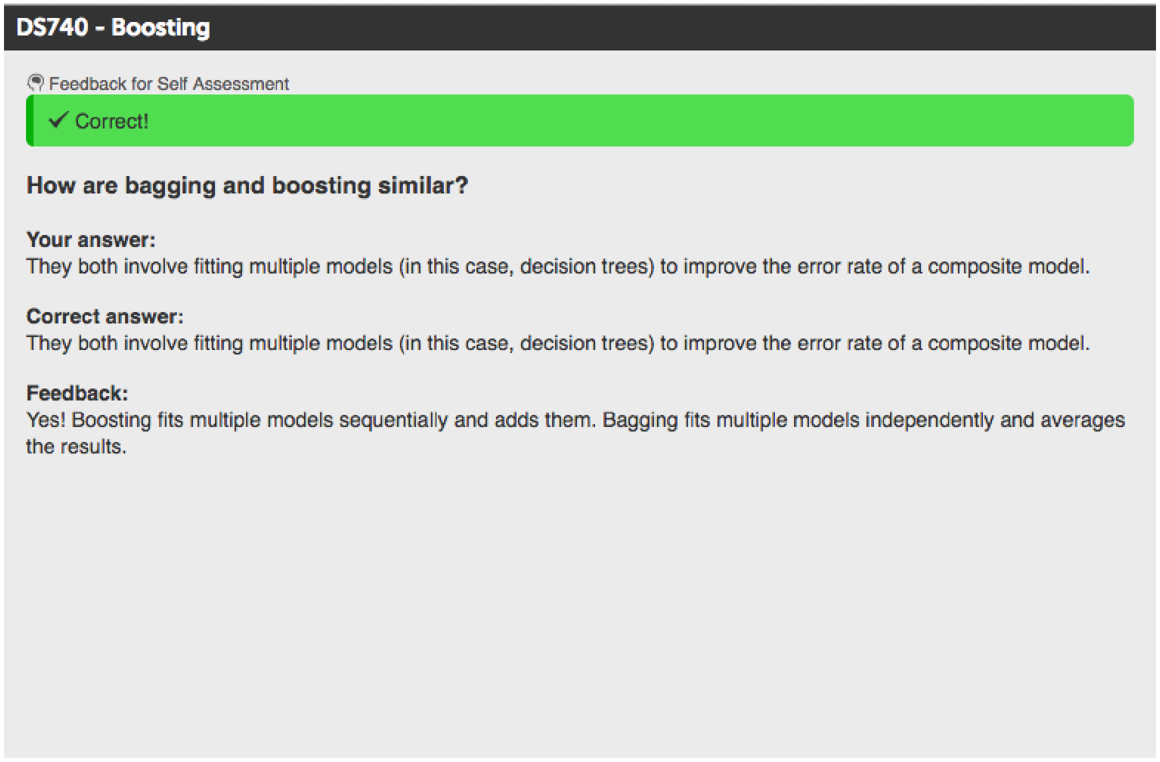
The R markdown document for this lesson also includes an example of how to do boosting in Caret using the xgboost package. You won't need this for the homework assignment for this lesson but, depending on what data set you choose, it might be useful to you on the project.



Summary

- Boosting fits multiple models sequentially to the residuals remaining after previous models.
- The composite model is the sum of all the models, times a shrinkage parameter λ.
- Optimal performance is usually associated with:
  - small value of λ (e.g., .001)
  - number of trees > 1000 (but not so large as to result in overfitting)
  - interaction depth < 10
  - bag fraction between .5 and .75

Question 1 Answer



DS740 - Boosting

Feedback for Self Assessment

✓ Correct!

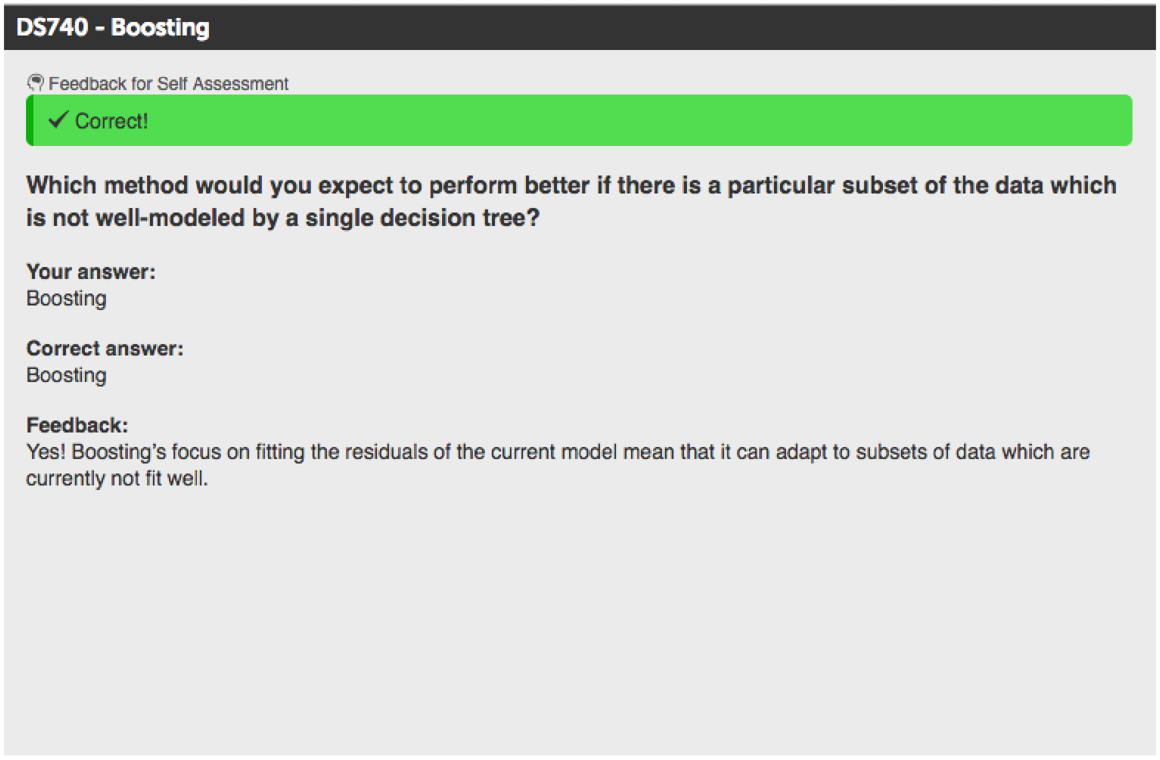
How are bagging and boosting similar?

Your answer:
They both involve fitting multiple models (in this case, decision trees) to improve the error rate of a composite model.

Correct answer:
They both involve fitting multiple models (in this case, decision trees) to improve the error rate of a composite model.

Feedback:
Yes! Boosting fits multiple models sequentially and adds them. Bagging fits multiple models independently and averages the results.

Question 2 Answer



DS740 - Boosting

Feedback for Self Assessment

✓ Correct!

Which method would you expect to perform better if there is a particular subset of the data which is not well-modeled by a single decision tree?

Your answer:
Boosting

Correct answer:
Boosting

Feedback:
Yes! Boosting's focus on fitting the residuals of the current model mean that it can adapt to subsets of data which are currently not fit well.