



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

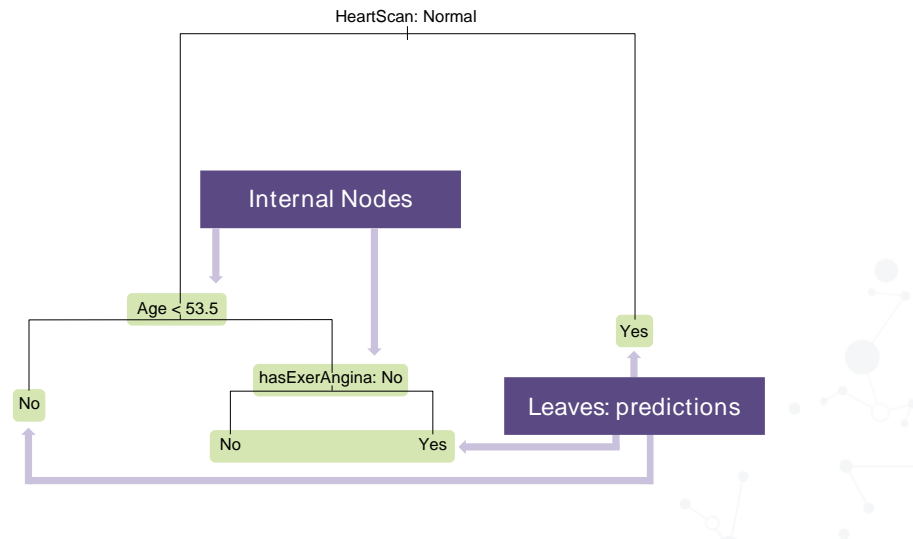
## Learning Objectives

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

- Explain how decision trees are constructed.
- Interpret decision trees.
- Explain the advantages and limitations of decision trees.
- Construct a decision tree in R.



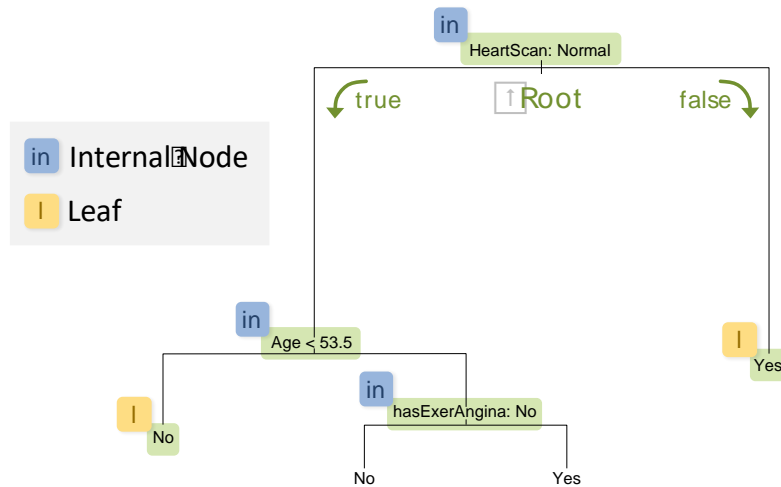
# Decision Tree Terminology



Decision trees are a method of supervised learning that can be used for either classification, or regression problems. They're represented by diagrams like this, where the end of each line is called a leaf, or a terminal node. And these represent the predictions. The places where one line splits into two are called the internal nodes. And these divide up the predictor space.

Each observation in the dataset will either have age less than 53.5 or not. Each observation will either have exer-angina equal to no, or equal to some other value.

# Interpreting Decision Trees

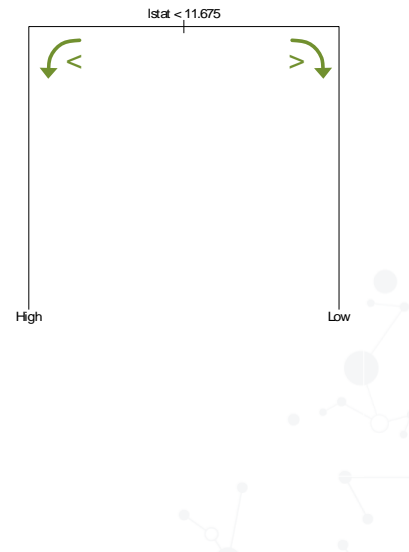
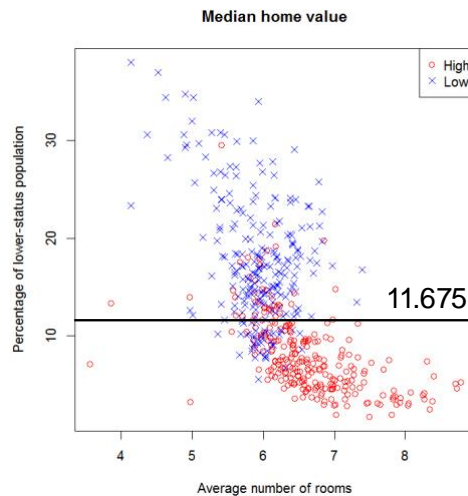


Decision trees are especially good if you want a model that's easy to interpret and communicate. Let's look at how to interpret this model, which deals with predicting whether a person has heart disease. Imagine that we're trying to predict the heart disease status of one particular person. We would start at the internal node at the very top of the tree, called the root. This is labeled with a variable heart scan, and a value, normal.

If heart scan equals normal is true for this person, then we would move to the left down this flow chart of sorts. If heart scan equals normal is false, then we would move to the right. So if this particular person does not have a normal heart scan, then our prediction is yes, they have heart disease. However, if their heart scan is normal, then we have other questions to ask. We need to know whether their age is less than 53.5. If it is, then we move to the left again, and our prediction is no.

On the other hand, if their age is greater than or equal to 53.5, then whether they should be predicted to have heart disease depends on whether or not they have exercised-induced angina.

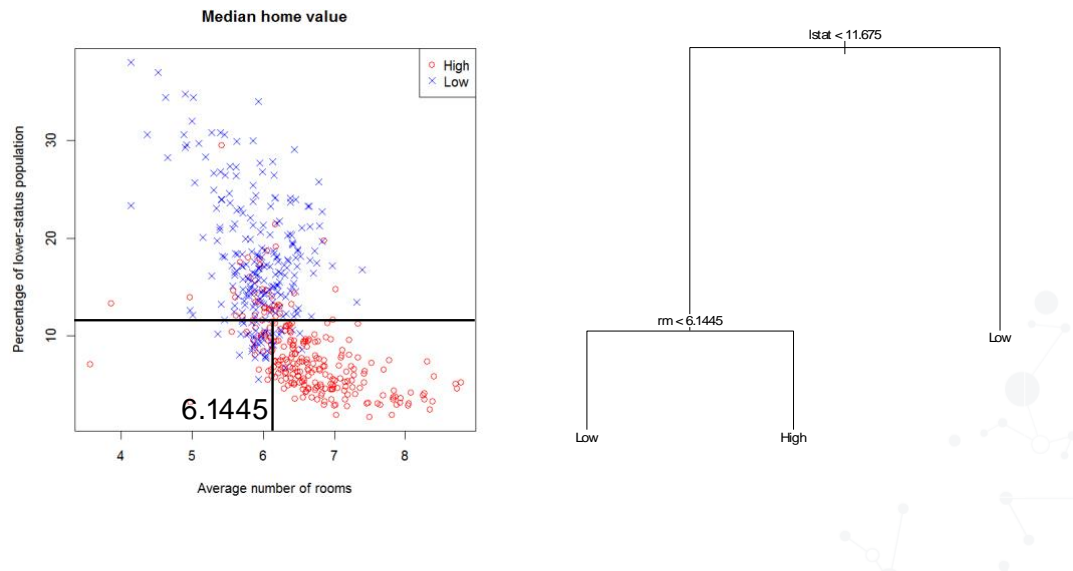
## Partitioning the Predictor Space: First Split



So how do we build the tree model? We can do it using a method called recursive binary splitting, where we start by thinking about all of the data points as belonging to a single region, and then look for the best way to split that region based on a single variable. For example, in the dataset shown here, we might decide to draw a horizontal line at a height of 11.675 for the variable lower status population. That looks like it does a decent job of separating the points with high versus low median home values. That would correspond to a tree with just two terminal nodes, also called a stump. Each of the terminal nodes corresponds to one of the regions in our new diagram.

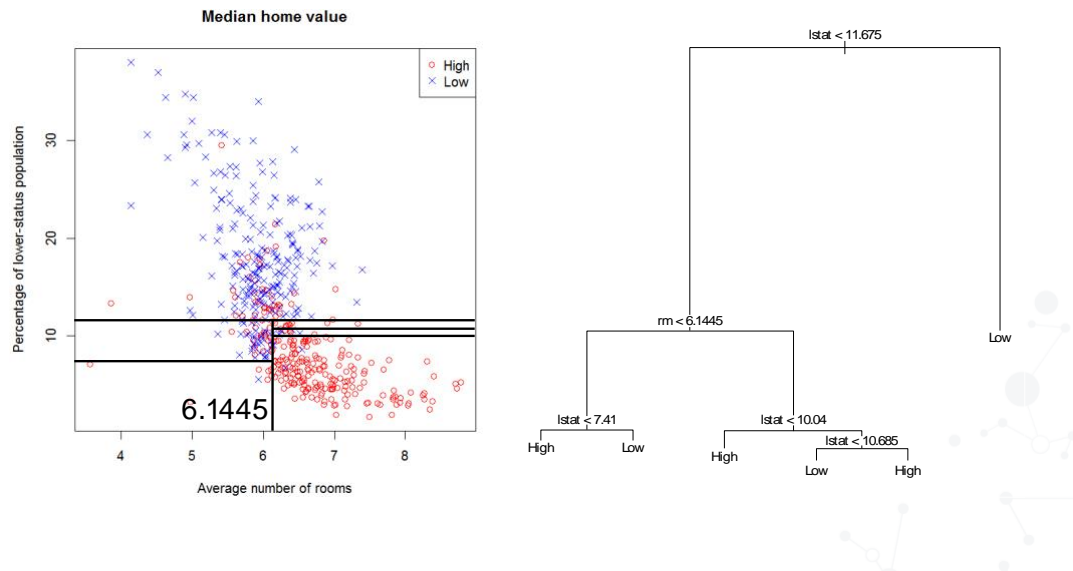
The split is based on the variable LSTAT, lower status population, less than 11.675. If that formula is true, we move to the left; and if it's false, we move to the right. And in each of the regions, we predict a value based on the more common category within that region.

## Partitioning the Predictor Space: Second Split



Next, we would look at both of our existing regions, and look for the best way to split one of those regions based on one variable. In this case, we might decide to draw a vertical line for the variable average number of rooms,  $rm$ , at the value 6.1445. Here, because we only have two predictor variables, each of our splits is represented by a line on the scatter plot. And each of the regions is a rectangle. But you can imagine that if we had three predictor variables, then the split would be a plane, and each of the regions would be a three dimensional box. And this would go on like this for multiple predictor variables.

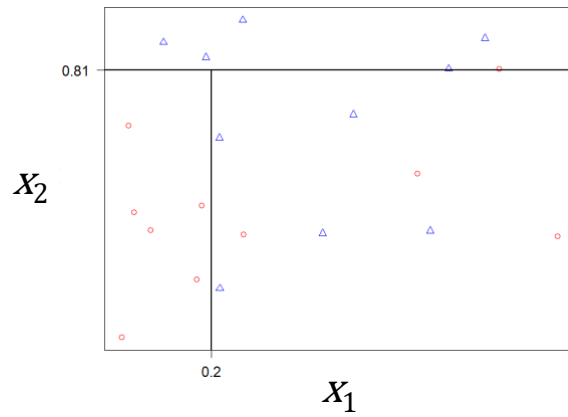
## Partitioning the Predictor Space: Additional Splits



And we keep going in this manner. At each step, we choose the best possible region to split, based on the best possible variable at the best possible value. Each split of the predictor space in the scatter plot corresponds to a split in the tree. We keep going in this process until each of the terminal nodes makes a prediction that's accurate to within a specified tolerance for error, or until each of the terminal nodes contains less than a specified number of data points, meaning there's just not enough data to make our predictions more accurate in a reliable way.

## Check Your Understanding: Partitioning the Predictor Space

Draw the decision tree that corresponds to this partition.



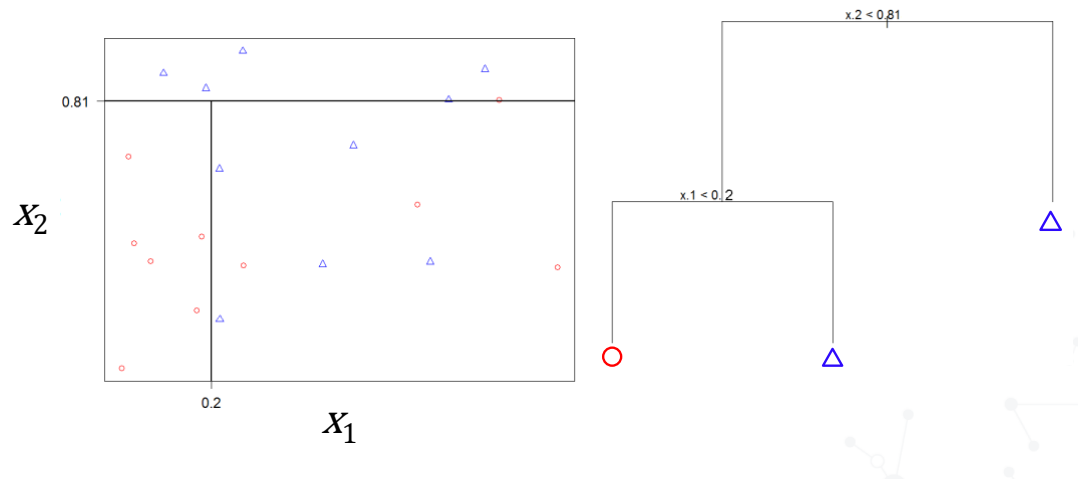
Answer on  
next slide

Quiz 3



Answer:

## Partitioning the Predictor Space

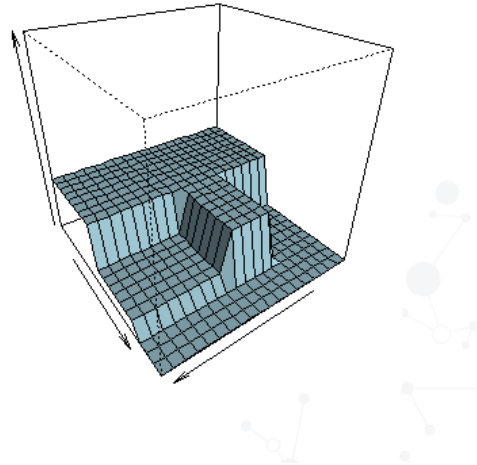


# Predictions

Classification: Most common class in that region.

Regression: Mean value in that region.

- Response modeled as constant within a region



Each terminal node in a tree corresponds to a region. For a classification problem, our prediction for a terminal node is whichever class, or category, is the most common in that region. For a regression problem, our prediction is the mean value within that region. That means that we're modeling the response variable as constant within a particular region. So we get sort of a stair-step model.

## Notes:

For more about how to make perspective graphs, see the document ["Plotting rpart trees with the rpart.plot package"](#) by Stephen Milborrow.

## Choosing an Optimal Split: Regression Trees

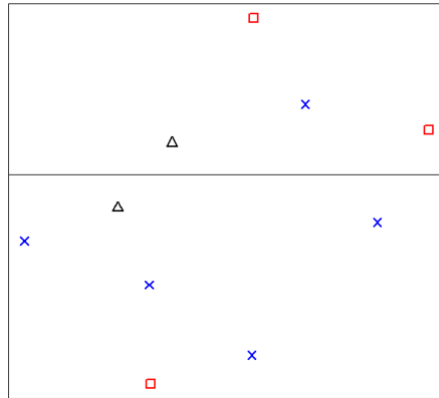
Minimize residual sum of squares (RSS)


$$\sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{j=1}^J \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$


So what criteria do we use to decide the best possible way to split a region? For a regression problem, we choose the split that minimizes the residual sum of squares, as shown here. This is the usual formula for a residual sum of squares. If we divided it by  $n$ , we'd get the mean squared error. However, for decision trees, we know that each point within the same region is predicted to have the same value,  $\hat{y}_i$ . So we can rewrite the residual sum of squares as shown on the right.

## Choosing an Optimal Split: Classification Trees

Classification Error Rate: proportion of all data points that are not the most common class in their region.



← Region 1: 2/4 ( $\neq$   / total)

← Region 2: 2/6 ( $\neq$   / total)

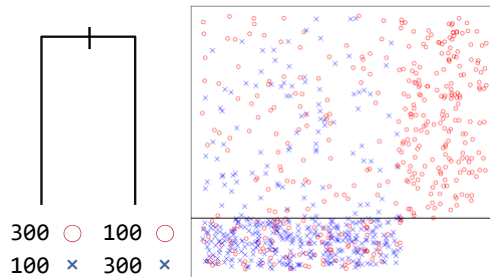
Overall: 4/10

For classification problems, it's tempting to choose splits based on the classification error rate, meaning the proportion of all data points that are not the most common class in their region. For example, in the diagram shown here, the top region has red squares as the most common class. So every point in that region would be predicted to be a red square. However, there are two points that are not red squares, so the classification error rate for that region would be 2 out of 4.

The classification error rate for the bottom region would be 2 out of 6, and the overall classification error rate would be 4 out of 10.

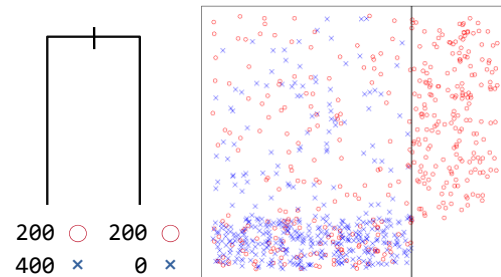
## Classification Error Rate is Not Sensitive Enough

Overall classification error rate =  $\frac{1}{4}$



Overall classification error rate =  $\frac{1}{4}$

One region has perfect prediction accuracy on training set



However, it turns out that the classification error rate is not sensitive enough to generate the best splits reliably. For example, suppose we have a dataset of 400 red circles and 400 blue x's, and we're considering two possible ways of splitting the data. Both of these proposals have the same overall classification error rate of  $\frac{1}{4}$ . But the proposed split shown on the right produces a region that has 200 red circles and no blue x's. So for that region, the predictions would be perfect for the training set, or in other words, we would say that that node has perfect node purity.

This is a very good thing to have in a split, but the classification error rate can't distinguish between this proposed split and the version on the left.

## Better Measures of Split Optimality

Gini Index:

$$\sum_{k=1}^K \hat{p}_{mk} (1 - \hat{p}_{mk})$$

Cross-entropy:

$$-\sum_{k=1}^K \hat{p}_{mk} \log \hat{p}_{mk}$$

$$\hat{p}_{mk}$$

Proportion of points in  
region  $m$  from class  $k$

Two better measures of split optimality for classification trees are the Gini index, and the cross entropy, which is closely related to the deviance. They're both based on  $\hat{p}_{mk}$ , the proportion of points in region  $m$  from class  $k$ . And they both take on very small values if one of the classes has a proportion near 1, and the other classes have proportions near 0. So they can both be used to measure node purity.

## Question 1

### DS740 - Decision Trees

🔊 Question for Self Assessment: Multiple Choice

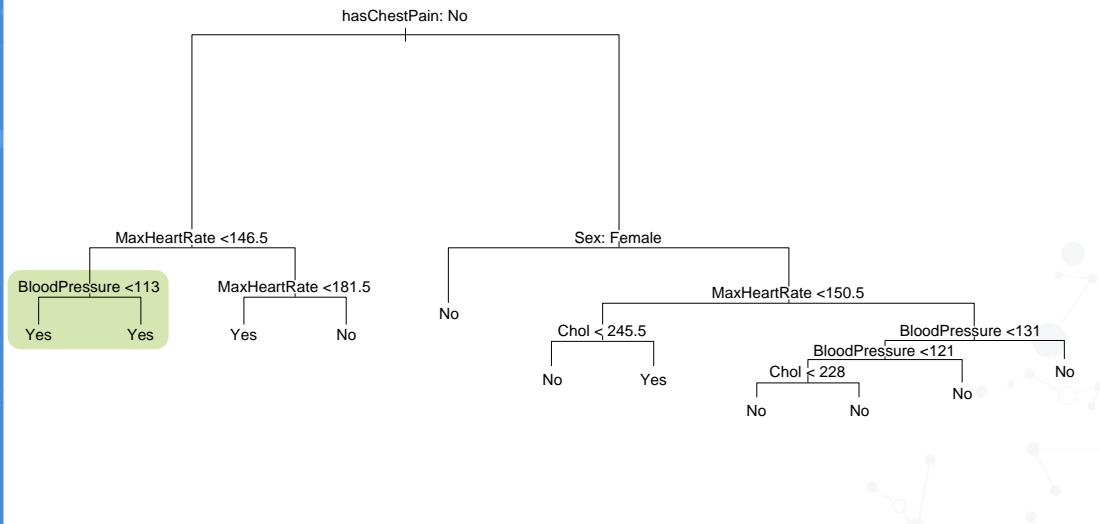
**What would be the Gini index of a region with 200 data points from class 1 and 400 data points from class 2? (Assume there are only 2 classes.)**

- ☐ 2/9
- ☐ 1/3
- ☐ 4/9

SUBMIT

Answer is at the end of this transcript

## Complicated Trees



Trees that result from recursive binary splitting can get very complicated. In some cases, you may even end up with splits that lead to two leaves with the same prediction, as shown here. This reflects a difference in the confidence of predictions. For example, in this dataset, among people with a blood pressure greater than 113, 90% of them had heart disease. However, among people with a blood pressure of less than 113, only 57% of them had heart disease. So in both cases, having heart disease, yes, was the more common class. But our level of confidence in the prediction was much higher for people with higher blood pressure.

A bigger problem with complicated trees-- besides being complicated to interpret-- is that they can lead to over-fitting to the training data, which can increase the variance of the model.



## Pruning

Minimize

$$\alpha|T| + \sum_{M=1}^{|T|} \sum_{i \in R_m} (y_i - \hat{y}_{R_m})^2$$

$|T|$  = number of leaves

$\alpha$  = penalty parameter



We can penalize trees that are more complex. Instead of minimizing the residual sum of squares by itself, we can minimize the residual sum of squares plus alpha times the size of  $t$ , where the size of the tree  $t$ , is measured by the number of leaves, and alpha is a penalty parameter. If alpha equals 0, we would just get back the original fully complicated tree. We can choose alpha by cross-validation.

## Sample Problem

Use decision trees 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 :0.04383	Mean :0.04383
3rd Qu.:0.03555	3rd Qu.:0.04795	3rd Qu.:0.05795
Max. :0.13710	Max. :0.23390	Max. :0.30590



## DS740 - Decision Trees

```
11
12 {r, message = FALSE}
13 library(tree)
14 library(readr)
15
16
17 # Can we predict whether an object is rock or metal based
   on its sonar reading?
18
19 {r}
20 sonar = read_csv("sonar_rock_metal.csv",
21                  col_names = FALSE)
22 head(sonar)
23
24 {r}
25 dim(sonar)
26
```

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

Let's use a decision tree to predict whether objects are rock or metal based on their sonar readings. I've already loaded the packages, tree to work with a single decision tree, and readr, which will allow us to read in the data using read\_csv. The sonar rock metal data set doesn't have a header row with the column names. So I'm using the additional argument col\_names equals false.

You can see that what this does is it gives each column the name x followed by its column number. The sonar data set contains 208 rows or observations and 61 columns. Columns 1 through 60 are numerical predictor variables, giving information about the sonar readings. And column 61 is a categorical response variable that equals M for metal and R for rock.

Next, let's divide the data set into a training set and test set. I'm putting 140 rows into the training set. Then to fit the decision tree on the training set, we'll use the function tree. The first argument here is a formula, very similar to the formulas that we've seen for, say, linear regression, where we have the response variable, or y variable, squiggle the predictor variables.

So here, I'm using all of the other variables as predictor variables denoted by the period, and because these column number 61 was read in as a character variable from read\_csv, I'm using the function factor to temporarily convert it into a factor variable for purposes of fitting the model. Then we have the arguments data equals the rows of the

sonar data set that are in the training set, which we defined as our vector of trues and falses when we created the training set.

So here's the summary of the tree. It's telling us which columns or variables were used to create the tree. It has a tree with 12 terminal nodes or leaves, and it's telling us the misclassification error rate. Don't pay too much attention to this number, because remember that this is the error rate on just the training set. So we expect it to be smaller than what the error rate would be on new data.

We can graph the tree to get a better sense of what our model is doing. We can do that using the plot function and simply giving it the argument of the tree object that we just created, and then the text function will label each of the leaf nodes with the predictions, and it will label each of the interior nodes with the variable being used for the split. The argument pretty equal zero can sometimes make those labels a bit easier to read.

So here's our tree, and you can see that, even though I used the function par to set the graphing parameters, to set the margins of the graph equal to just 0.1 on all four sides, we're still getting some information cut off at the top and bottom. When this happens, it's often helpful to print your graph to a PDF using the function PDF. And the argument here is just the name of the file that you want to create in order to save the graph.

Be careful, because this will overwrite any existing file of this name in the working directory. And once you're done creating the PDF, you need to use the function dev.off to close the graphing utility so that you're able to open it and view the PDF in another way. So then we can go into our File Explorer and view the PDF.

So here, we have our decision tree. So for example, you can see that the first split is being done based on variable x52. If that's less than 0.00935, then we're going to the left and checking the value of variable x11. Unfortunately, because we don't have column names here, we can't do a lot of interpretation on what this means to have the 51st or 52nd, or 11th sonar reading be a particular value.

## DS740 - Decision Trees

```
59
60
61
62 |
63
64 {r}
65 preds = predict(my_tree, newdata = sonar[!in_train, ],
66                 type = "class")
67     # type = "class" returns predicted response classes
68     # type = "vector" returns predicted probabilities
69
70
71
72
73
74
```

To measure how well our model is doing on the test data set, we first need to compute the predictions on that test data set. So we can do that using the predict function, as we've done before, with new data equal to sonar square bracket exclamation point in train, comma. So remember that in train was a vector of trues and falses to say whether each data point was in the training set.

So the exclamation point is negating each of those. So it's turning all of the falses into trues, so we're going to be taking all of the rows, where the data point was not in the training set, which means it was in the test set. Here, I'm using type equal to class. That will get me the predicted response classes. That is, metal or rock.

Alternatively, we could use type equal to vector to get the predicted probabilities of belonging to one of the classes. We can compute the confusion matrix using the table function, as we've done in the past. And we can then compute the accuracy of our model using the diag function to get the diagonal elements from our confusion matrix, 29 and 18, and then taking the sum of those.

So that's the total number of observations in the test set that were correctly classified, and then we're dividing by the size of our test set. So this is our accuracy, and 1 minus that will then give us the misclassification error rate on the test set. We could also have done this by taking 6 plus 15 and dividing that by 68, but I like this method, because it lets me recompute the error rate programmatically so that I don't need to go in and edit the numbers 6 and 15 each time I change the random seed.

So here we see that we have an error rate of about 30%. So that's quite a bit worse on new data than our model was doing on the training set itself. If we want to do better,

we might be able to do that by choosing an optimal number of leaf nodes for our tree using tenfold cross-validation. So we can do this using the `cv.tree` function.

The first argument is the existing tree that we created that we might want to prune or make smaller, and because this was a classification problem, we have the second argument `fun`, or function, equals `prune.misclass`. If this were a regression problem, that is using a decision tree to predict a quantitative response variable, we would instead use `prune.tree`.

So here, we can see the different sizes of tree that were tested, as measured by the number of leaf nodes. The deviance, or error, and the value of `k`, which is analogous to, but not exactly the same as the `alpha` value that we saw in the slides. We can also graph the results of our cross-validation using the `plot` function.

So here, we see that the error rate decreases as we increase the number of leaf nodes, and then it might increase again. As our tree becomes too complicated, it starts to overfit to the training data. In this model, it looks pretty clear that 4 and 6 are the best sizes of tree that give us a deviance of 43.

If we want to extract that programmatically, we can do that using the `min` function to find the minimum value of the deviance and then the `which` function will tell us which elements of the deviance were equal to the minimum. So in other words, that's looking at our deviance here, saying, hey, 43 is the minimum, and that occurs at locations 4 and 5 in this vector.

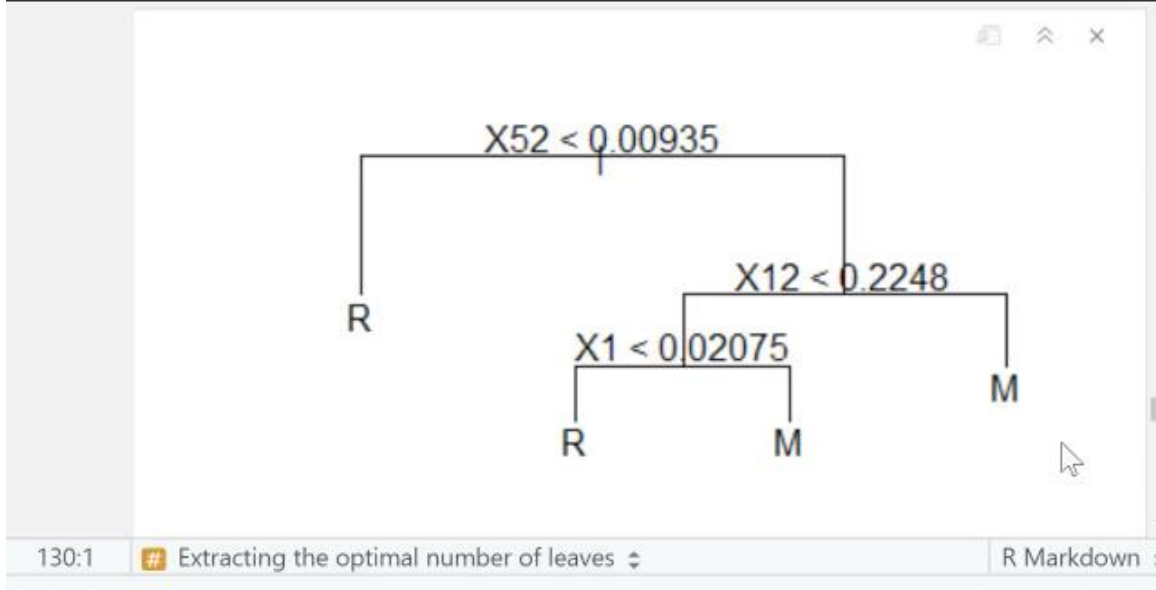
We could also do this using `which.min`, but that would only tell us about the first location where the minimum happened. And in this case, we actually had two locations that were equal to the minimum. Finally, we can use square brackets on the size of `sonar.dv`, our variable name, to pick out the values of size that correspond to those minimum deviance values.

So here, we see that the fourth and fifth elements of `size` were, indeed, the trees with 6 and 4 leaves. I typically prefer to go with the smaller tree that has the minimum, because that's going to be simpler and easier to interpret. We can take a look at that best tree by first creating it using `prune.misclass`. Again, here, if we were doing a regression problem, we would have `prune.tree`.

We're pruning our initial existing tree, and we have the argument `best` equal to 4. So that's the number of leaf nodes that we want to end up with in our

pruned tree, that we selected based on the cross-validation. And then we'll plot our tree and add labels to it, just like we did before. So here, we end up with a much simpler tree, where we can see that, if column 52 is less than 0.00935, we go to the left here and there are no more decisions to make. We just predict that that object is a rock.

## DS740 - Decision Trees



Now, you might be wishing for a bit more information about this tree. For example, how many observations are in each of these leaf nodes? You might be able to get this information by looking at the contents of the variable where the tree is stored. But in my case, I get an error message saying, argument 1 of type 'list' cannot be handled by cat.

Put simply, `r` is having a hard time figuring out how to display data of this type. This seems to have to do with a conflict between the tree package and some of the Tidyverse packages, such as `readr`. So what I'm going to do instead is, copy the name of this variable down into the console and type, `$frame`.

And this will show me details about the tree. Each of these rows corresponds to either a split or a leaf. So for example, we can see that at the top of the tree, we have, our first split is based on variable `x52`. And that had 140 data points in it. That makes sense. That was the size of our training data set.

We can see the deviance and the direction of the splits. So the left one was for values less than 0.00935. And the right one was for values greater than 0.00935.

And then, we can see the proportion of data at that node that was in each of the two categories, M and R. So for example, this first leaf node that we got to by following the left split at the first known-- so we'll display that again-- which led us left to a prediction of `r` here, that had 61 data points in it. And about 82% of them were rocks.

Down here at the 12th row, we see that 100% of the observations were rocks, but there were only 11 observations that reached that point. So for this leaf node down here,



where the prediction is rock, we're very confident about that prediction. But there aren't very many data points in the data set that followed that path through the tree.

Here's what the contents of `prune-sonar` look like when I don't load the `readr` package, and instead, just read in the data using `read.csv`. This is the same information as in `prune_sonar` `$frame`. It's just organized a bit more nicely. Here, the leaf nodes are denoted by asterisks. And we have some indentation to tell us which nodes are contained underneath other ones.

So we start with 140 data points up at the top, at the root node. If variable 52 is less than 0.00935, then we end up at a leaf node with a prediction of rock. And the probabilities or proportions belonging to the two categories are 0.18 and 0.8197. Going the other direction, if variable 52 is greater than 0.00935, then we end up at a non-leaf node that then contains additional splits below it, checking variable 12, and then, variable 1.

In this example, we started with an initial tree, and then pruned it to make it simpler. But what if your initial tree was already too simple for your application? The default in `r` for the `tree` function is to make splits, but stop if any new split would create a node with fewer than five observations or with a deviance that's less than 1% of the deviance of the root node.

If this results in a tree that's too simple for your application, you can change these defaults. You would do this using the function `tree.control`, used as an argument to the `control` argument name for the `tree` function. Here, I'm setting the number of observations to 208. That was the number of data points in our whole data set, which is what I'm working with here. So not just the training data set now.

The minimum deviance is 0. So the function will continue making splits until the deviance doesn't decrease at all. And the minimum size, equal to 2. Meaning that the function will continue making splits as long as there are at least two observations per node.

And I'm plotting this tree in a PDF. So we'll go look at that. So what you can see here is a much more complicated tree, because we now have it set up so that each of these leaf nodes perfectly classifies the data points that reach that point in the tree. This tree is much more complicated than the one that we ended up with. And it's probably going to have a higher error rate on new data points, because it's probably over-fit to our current data set.

## Decision trees using caret

```
set.seed(789)
data_used = Boston

ctrl = trainControl(method = "cv", number = 5)
fit_crim = train(crim ~ .,
                 data = data_used,
                 method = "rpart",
                 na.action = na.exclude,
                 tuneGrid = expand.grid(cp = seq(0, .15, by = .01)),
                 trControl = ctrl)

fit_crim
```

The example code for this lesson includes an example of how to build a decision tree using caret. You won't need to do this for this lesson's homework assignment, but depending on what data set you choose, it may be useful to you on the project.

### Notes:

```
library(caret)
library(MASS) # for Boston data
library(rpart) # a different package for making trees

set.seed(789)
data_used = Boston

ctrl = trainControl(method = "cv", number = 5)
fit_crim = train(crim ~ .,
                 data = data_used,
                 method = "rpart",
                 na.action = na.exclude,
                 tuneGrid = expand.grid(cp = seq(0, .15, by =
.01)),
                 trControl = ctrl)

fit_crim
```



## + Advantages

- Easy to interpret and communicate
- Resistant to predictor outliers, skew
- Automatic feature selection

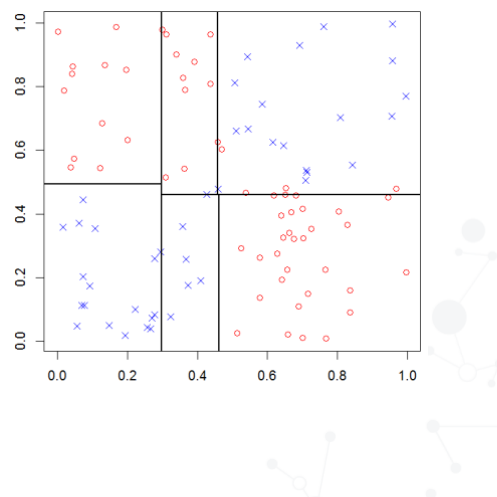
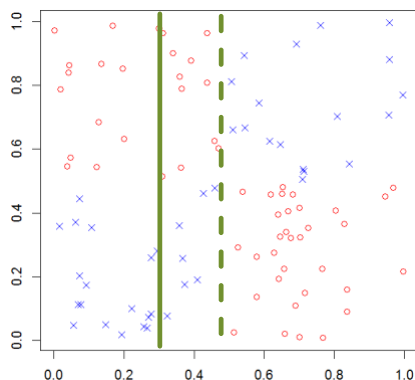


Decision trees have several advantages. They're easy to interpret and communicate. And in particular, they model interactions in an intuitive way, where one variable only matters if another variable takes on a particular value. Decision trees are also resistant to outliers and skew among the predictor variables. You don't need to transform the predictor variables before analysis.

It is still a good idea to transform the response variable in a regression problem if it's skewed, so that the predictions made up of the mean from each region are a good representation. Decision trees also perform automatic feature selection. If a particular variable doesn't contribute much to making a prediction, it doesn't get included in the tree. That means that decision trees are a good choice if you suspect that some of your predictor variables may be pure noise from the perspective of predicting the response.

## Limitations One

### Greedy



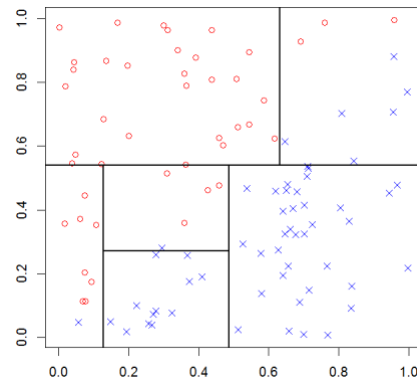
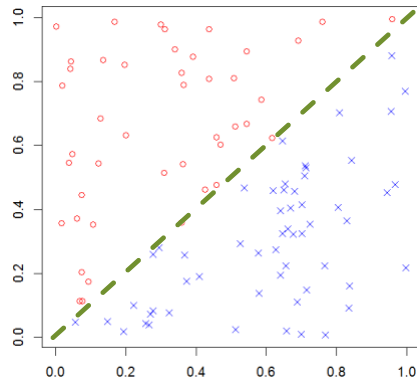
One of the limitations of recursive binary splitting is that it's a greedy algorithm. At each step, it just makes the best possible choice for the current step without looking any farther ahead. For example, suppose we have a dataset as shown here, with blue x's in the upper right quadrant and lower left quadrant, and red circles in the upper left and lower right.

A greedy algorithm isn't able to pay attention to the fact that if we divide the dataset at  $x$  equals 0.5, the next step after that is going to be great. Instead, it just makes the best possible choice for the current step, which ends up looking kind of random at  $x$  equals 0.3. This means that the regions and the resulting tree of recursive binary splitting can end up being more complicated than necessary.

Sometimes, you or a subject matter expert might be able to interpret the resulting tree in a simplified way, but this has to be done on a case-by-case basis. It's more likely that you would want to use a support vector machine for a dataset like this.

## Limitations Two

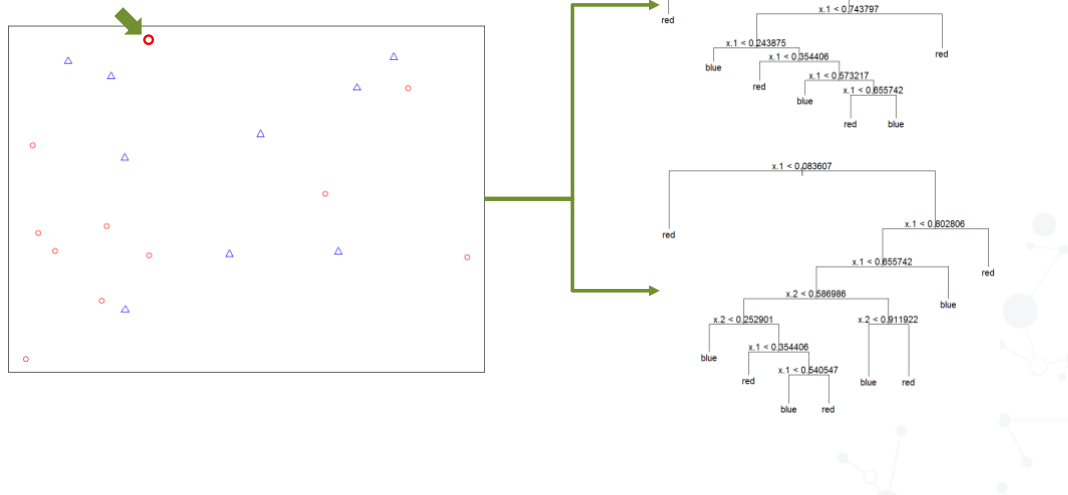
Splits based on a single variable



Another limitation of basic decision trees is that the splits are based on a single variable at a time. That means they tend not to be very good if the true relationship between the predictors and the response variable is linear. For example, if the true division between two categories is a diagonal line, we have to approximate that with a series of horizontal and vertical lines. So we end up with a more complicated division than necessary. In this situation, linear discriminant analysis, or a support vector classifier would be a good idea.

## – Limitations Three

Sensitive to changes in data or variables



Another limitation of basic decision trees is that they tend to be sensitive to changes in the data or in the variables included. For example, the dataset shown on the left results in the decision tree shown on the right. But if we change one data point from being a blue triangle to being a red circle, the decision tree totally changes. This results in high variance for the model, and can result in poor accuracy when predicting new data points. Bagging and boosting are two ways to reduce the effect of this limitation.

## Summary One

- Decision trees are constructed by partitioning the predictor space into boxes, based on 1 variable at a time.
- At each step in the construction, we choose the split that provides the greatest reduction of the MSE (for regression problems) or Gini index or cross-entropy (for classification).
- Pruning the decision tree can reduce the variance of the model.



## Summary Two

- Decision trees are easy to interpret, resistant to predictor outliers, and incorporate variable selection.
- Other methods may perform better when there are linear relationships or complex interactions.





## Question 1 Answer

### DS740 - Decision Trees

Feedback for Self Assessment

✓ Correct!

What would be the Gini index of a region with 200 data points from class 1 and 400 data points from class 2? (Assume there are only 2 classes.)

Your answer:

4/9

Correct answer:

4/9

Feedback:

Correct! The Gini index of this region is  $(200/600) * (400/600) + (400/600) * (200/600) = 4/9$ .