07 - Trees and Forests

ml4econ, HUJI 2024

Itamar Caspi

May 11, 2025 (updated: 2025-05-11)

Setting Up with the Pacman Package

Leverage the power of the pacman package. It auto-loads and installs packages as needed.

```
if (!require("pacman")) install.packages("pacman")

pacman::p_load(
  tidyverse,   # for data wrangling and visualization
  broom,       # for tidy model output
  rpart,       # for estimating CART
  rpart.plot,  # for plotting rpart objects
  ranger,      # for estimating random forests
  vip,         # for variable importance plots
  knitr,       # for displaying nice tables
  here         # for referencing folders and files
)

# remotes::install_github("grantmcdermott/parttree")
library(parttree)
```

Set a theme for `ggplot` (Relevant only for the presentation), and set a seed for replication

```
theme_set(theme_grey(20))
set.seed(1203)
```

Outline

- Stratification
- Regression Trees
- Classification Trees
- Random Forests
- Other Ensemble Methods

Applications of Decision Trees

Apply decision trees to both regression and classification tasks.

- We'll first tackle regression problems using the Boston dataset.
- Next, we'll transition to classification, using the Titanic dataset.

Stratification

Boston Housing (Again)

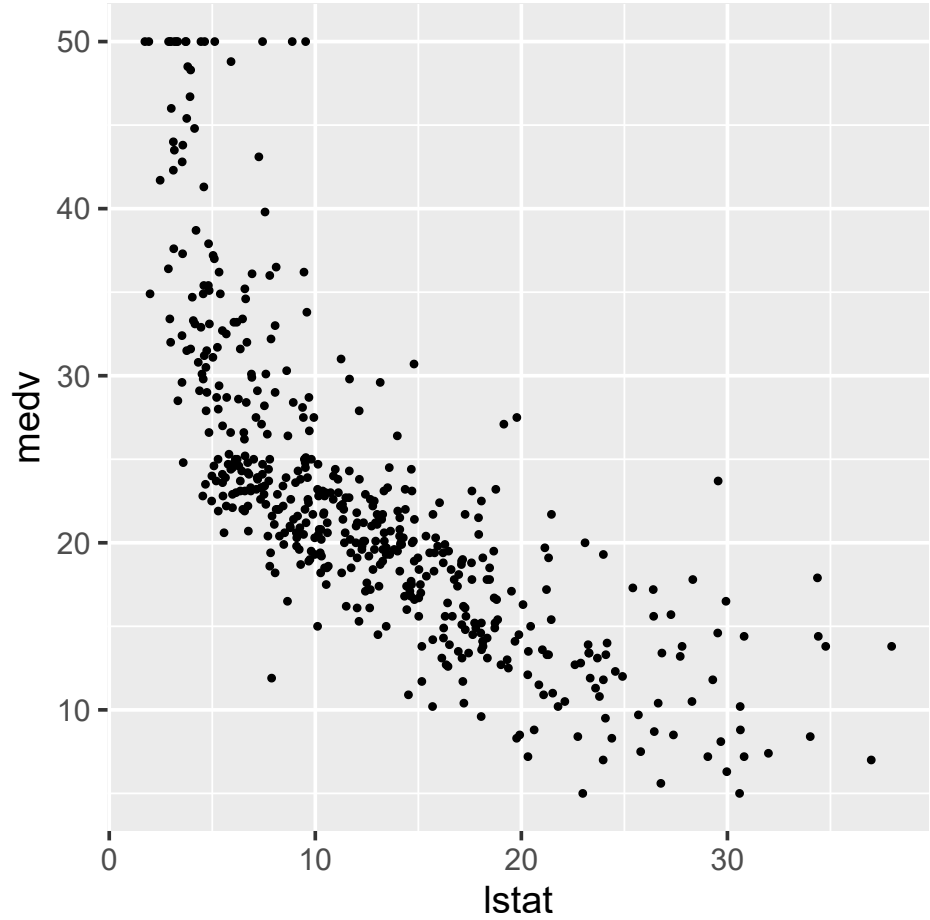
Load the data

```
boston <-
  here("07-trees-forests/data","BostonHousing.csv") %>%
  read_csv()
```

```
## Rows: 506 Columns: 14
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (14): crim, zn, indus, chas, nox, rm, age, dis, rad, tax, ptratio, b, lstat, medv
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Recall the Nonlinear Association Between `lstat` and `medv`

```
boston %>%
  ggplot(aes(lstat, medv)) +
  geom_point()
```



A Two-way Split of `lstat`

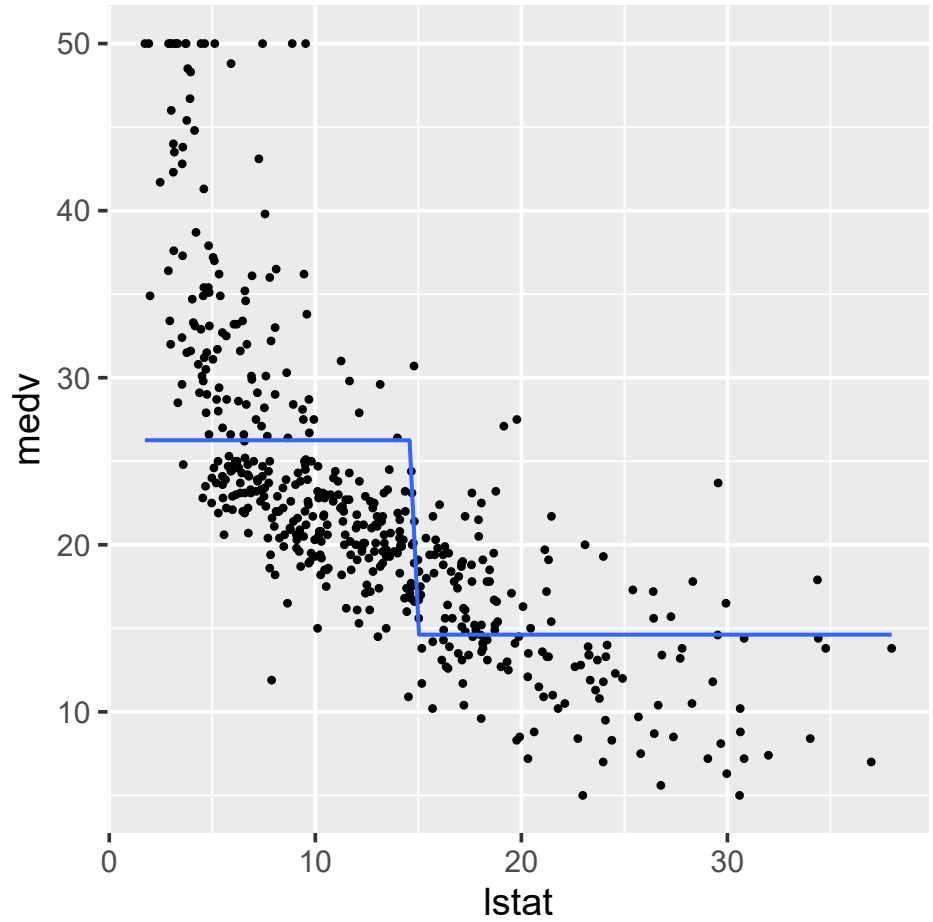
Define D_i as a two-way "split" dummy variable as follows:

$$D_i = \begin{cases} 1 & \text{if } lstat_i > 15 \\ 0 & \text{otherwise}, \end{cases}$$

The blue step-function on the left represents the fitted value from:

$$medv_i = \beta_0 + \beta_1 D_i + \varepsilon_i$$

Note: Predictions derive from the average of $medv_i$ within each "region".



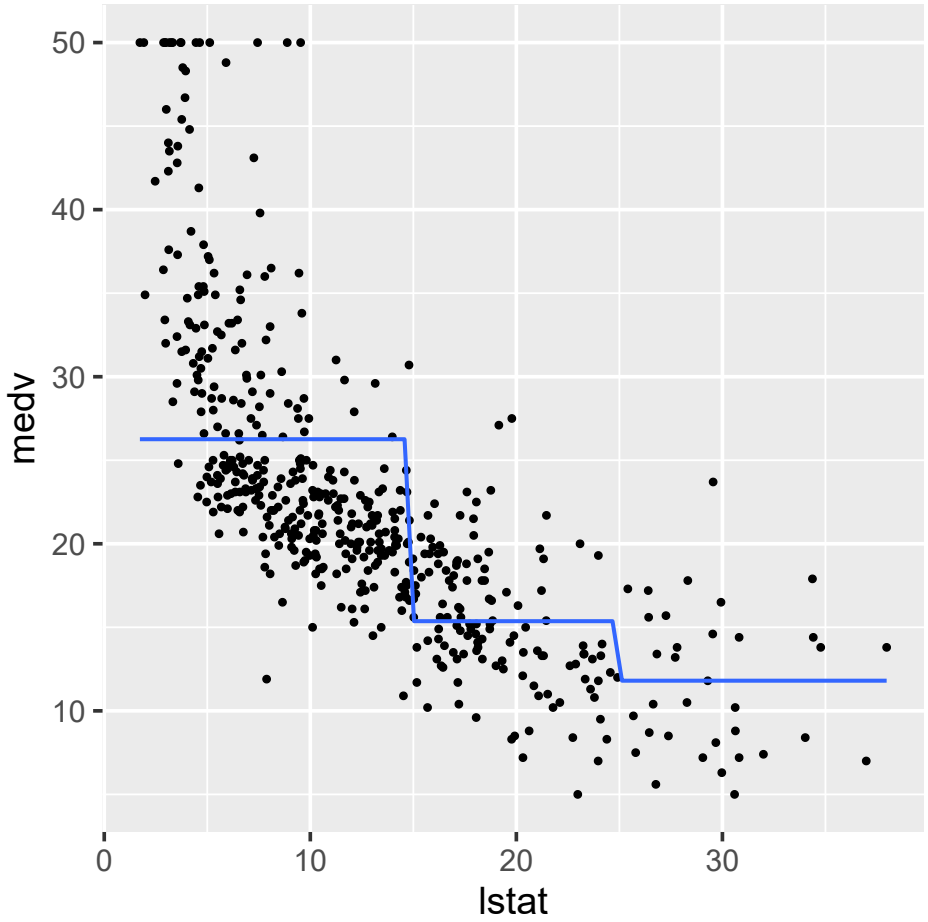
Three-way split

Next, we'll experiment with a three-way split:

$$D_{1i} = \begin{cases} 1 & \text{if } lstat_i < 15 \\ 0 & \text{otherwise ,} \end{cases} \qquad D_{2i} = \begin{cases} 1 & \text{if } lstat_i > 25 \\ 0 & \text{otherwise,} \end{cases}$$

```
boston %>%
  ggplot(aes(lstat, medv)) +
  geom_point() +
  geom_smooth(
    method = lm,
    se = FALSE,
    formula = y ~ (x>25) + (x<=25 & x>=15) + (x<15)
  )
```

Note: Predictions still stem from the average of $medv_i$ within each "region".

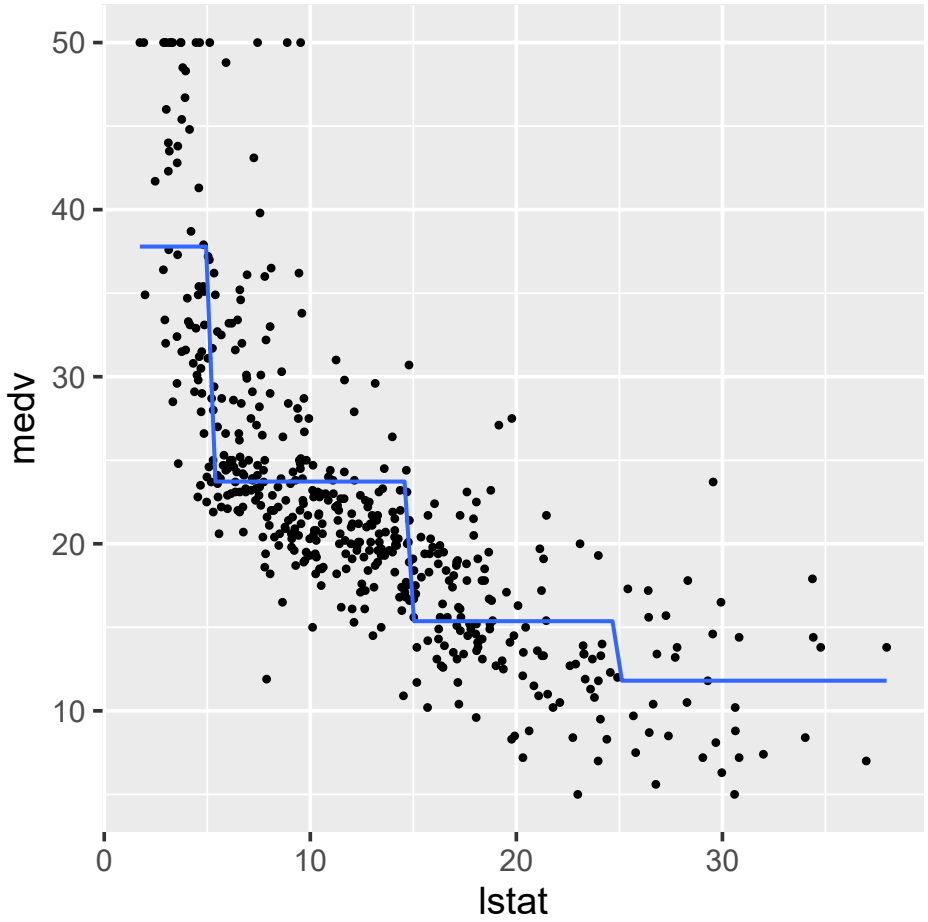


Four-way split

You get the point

```
boston %>%
  ggplot(aes(lstat, medv)) +
  geom_point() +
  geom_smooth(
    method = lm,
    se = FALSE,
    formula = y ~ (x>25) + (x<=25 & x>=15) + (x<15 &
  )
```

- The greater the number of splits, the better the fit. But what does this imply for prediction?



Challenges with Splits

Using splits in general introduces three key questions:

- Where should the split be?
- How many splits are optimal?
- How to make predictions within each node?

These queries fall within the scope of the decision tree framework.

Regression Trees

Classification and Regression Trees (CART)

Breiman et al., 1984, proposed the following fundamental concept:

1. Divide the feature space $x_1, x_2, \ldots, x_p$ into M distinct, non-overlapping regions (rectangles), $R_1, R_2, \ldots, R_M$.
2. For any observation that falls into region R_j, make the same prediction (be it regression or classification). For instance, for a continuous y,

$$\hat{y}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i$$

   where y_m is a test observation in region R_m.

The Splitting Process: How does it work?

- Checking all possible partitions of the feature space proves impractical. (But why?)
- Instead, the CART algorithm adopts a greedy approach.
- Starting with the entire dataset, define a splitting variable j and split point s. Construct the pair of half-planes:

$$R_1(j, s) = \{x | x_j \leq s\}, \qquad R_2(j, s) = \{x | x_j > s\}$$

- Identify the predictor j^* and split s^* that breaks the data into two regions $R_1(j^*, s^*)$ and $R_2(j^*, s^*)$. The partition should minimize the overall sum of squared errors:

$$\text{RSS} = \sum_{i \in R_1(j^*, s^*)} (y_i - \bar{y}_1)^2 + \sum_{i \in R_2(j^*, s^*)} (y_i - \bar{y}_2)^2$$

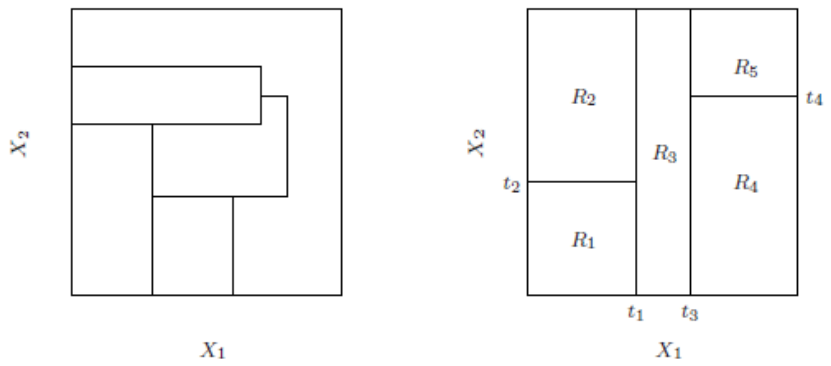
  Here, $\bar{y}_1$ and $\bar{y}_2$ denote the averages of training set outcomes within each group.

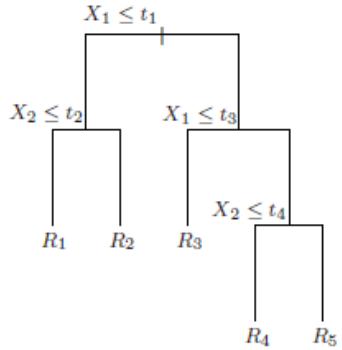
The CART Algorithm

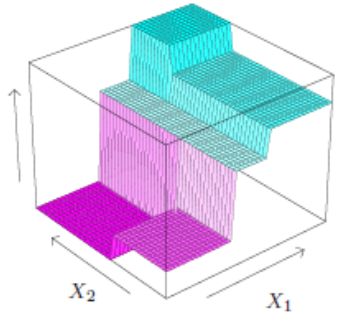
Begin with the root node, containing the complete sample. Then:

1. Identify the single RSS minimizing split for this node.
2. Split the parent node into left and right nodes.
3. Apply Steps 1 and 2 to each child node.
4. Repeat until reaching a leaf node of a pre-defined minimum size (for instance, cease splitting when fewer than 10 observations remain in each leaf).

Partitioning of Feature Space







Source: ESL, pp. 308.

- **Top right:** CART's partition of a 2-D feature space.
- **Top left:** A general partition that CART can't produce.
- **Bottom left:** The tree corresponding to the partition in the top right.
- **Bottom right:** Prediction surface.

How large should we grow the tree?

- Large tree - overfit. Small tree - high variance.
- The tree's level of *expressiveness* is captured by its size (the number of terminal nodes).
- Common practice: Build a large tree and prune the tree backwards using *cost-complexity pruning*.

Cost-Complexity Pruning

The cost complexity criterion for a tree T is:

$$\mathrm{RSS}_{cp}(T) = \mathrm{RSS}(T) + cp|T|$$

where:

- RSS denotes the sum of squared error for tree T.
- $|T|$ represents the number of terminal nodes in tree T.
- cp is the complexity parameter.

In the CART algorithm, the penalty depends on the count of terminal nodes.

Note: cp and $|T|$ function similarly to λ and $\|\boldsymbol{\beta}\|_1$ in the lasso.

The Complexity Parameter cp

The complexity parameter, free of units, ranges from 0 to 1:

- At $cp = 0$, we get a fully saturated tree.
- At $cp = 1$, we see no splits. In other words, we predict the unconditional mean.

Boston with $cp = 1$

The R equivalent of the CART algorithm is `{rpart}`. Using the `rpart()` function, we can estimate a tree easily:

```
tree_fit <- rpart(
  medv ~ lstat,
  data = boston,
  control = rpart.control(cp = 1)
)
```

Remember, setting $cp = 1$ results in no splits.

To plot a tree, we utilize the `{rpart.plot}` package.

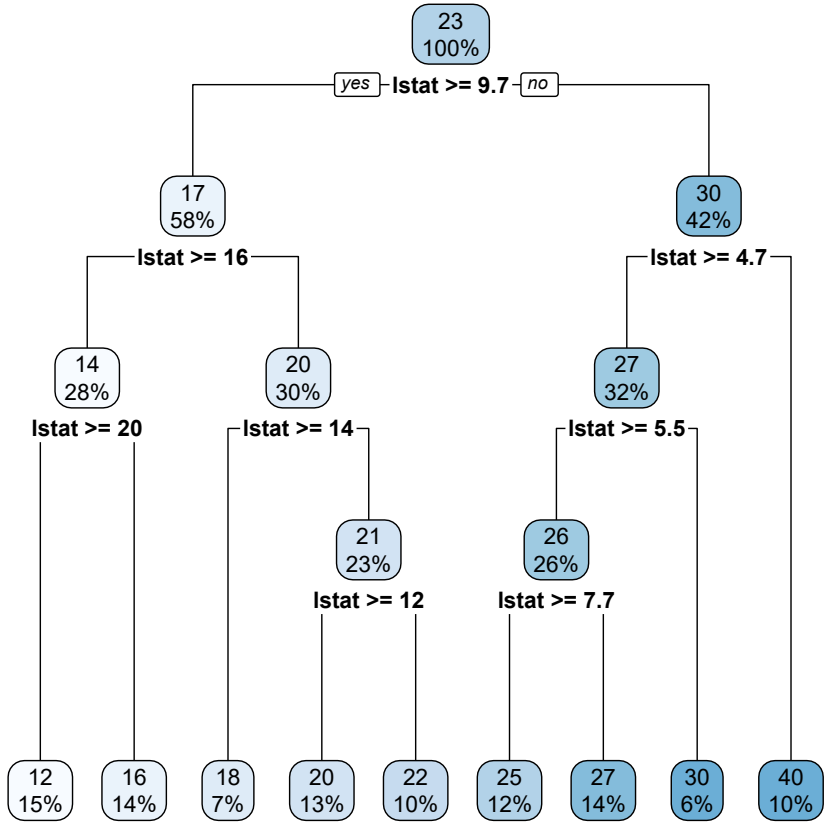
```
rpart.plot(tree_fit, cex = 2)
```

Boston with $cp = 0$

```
tree_fit <- rpart(
  medv ~ lstat,
  data = boston,
  control = rpart.control(cp = 0, minsplit = 80)
)
rpart.plot(tree_fit, cex = 1)
```

Setting $cp = 0$ leads to a saturated tree.

Please note that we've adjusted the minimum split criterion to 80 to prevent over-cluttering (excessive splits) in the figure on the right.



Tuning the Complexity Parameter cp

Breiman et al. (1984) recommend a cross-validation method to determine the optimal[*] cp:

- For any cp value, there exists a unique subtree T_{cp} that minimizes cost complexity $\text{RSS}_{cp}(T)$.
- To identify the best subtree, evaluate the data across a range of cp values. This process generates a finite sequence of subtrees, which includes T_{cp}.
- Estimate cp through cross-validation: select the value $\hat{cp}$ that minimizes the cross-validated RSS. The final tree becomes $T_{\hat{cp}}$.

[*] Breiman et al. (1984) also suggest employing the 1se heuristic. In other words, find the smallest tree that falls within one standard error of the tree with the smallest absolute error.

Understanding the Complexity Parameter cp

Think of the decision tree as a game of *Twenty Questions*.

- The cp decides the number of questions allowed before guessing.
    - If cp is small: We ask many questions; Our tree becomes detailed and large.
    - If cp is large:We ask fewer questions; Our tree is smaller and simpler.
- Too many questions (small cp) can lead to confusion and overfitting.
- Too few questions (large cp) might miss important details.

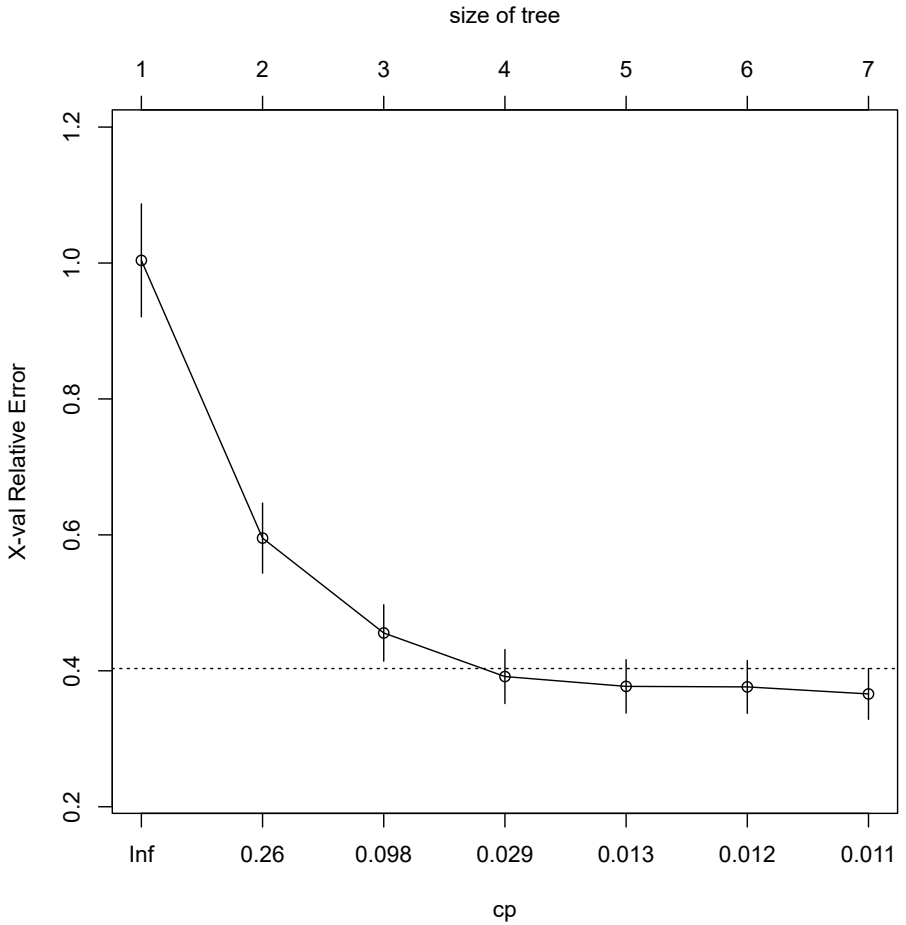
Tuning cp is about balance: Ask enough questions to understand the data. Avoid overcomplication to prevent overfitting.

Boston tree cross validation

The `{rpart}` package's `plotcp()` function provides a visual depiction of the cross-validation results within an `rpart` object.

```
tree_fit <- rpart(
  medv ~ lstat,
  data = boston
)
plotcp(tree_fit)
```

$cp = 0.029$ is the 1se optimal cp.



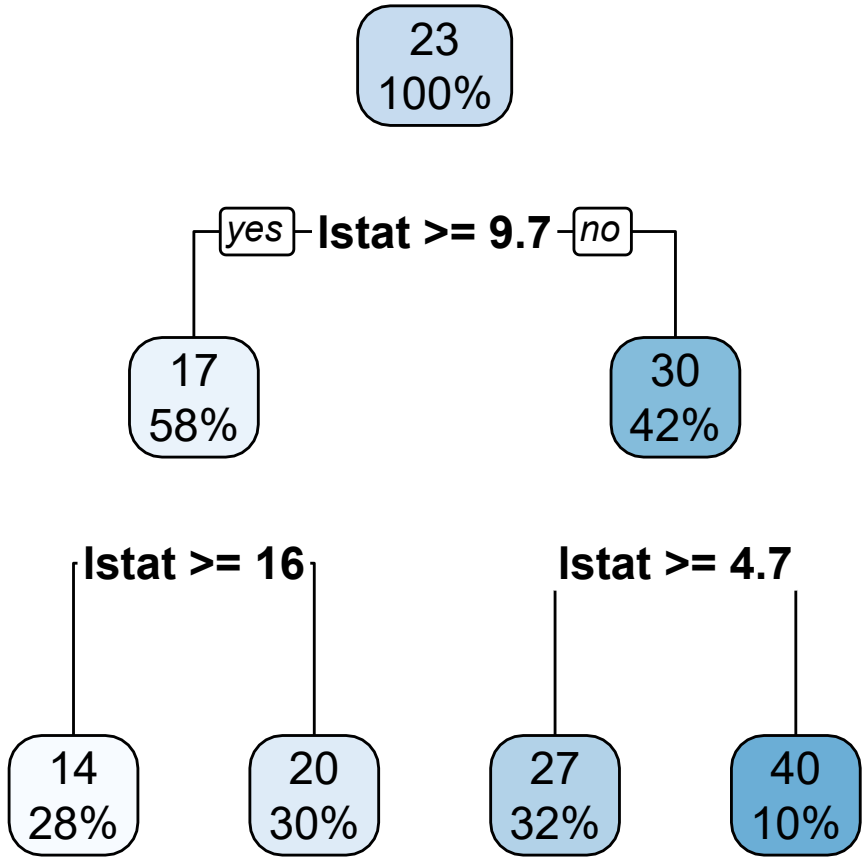
Boston pruned tree

We now proceed to pruning the tree using the `prune()` function (also from `{rpart}`), where we set `cp = 0.029`:

```
tree_prune <- prune(tree_fit, cp = 0.029)
```

And now we can plot the pruned tree:

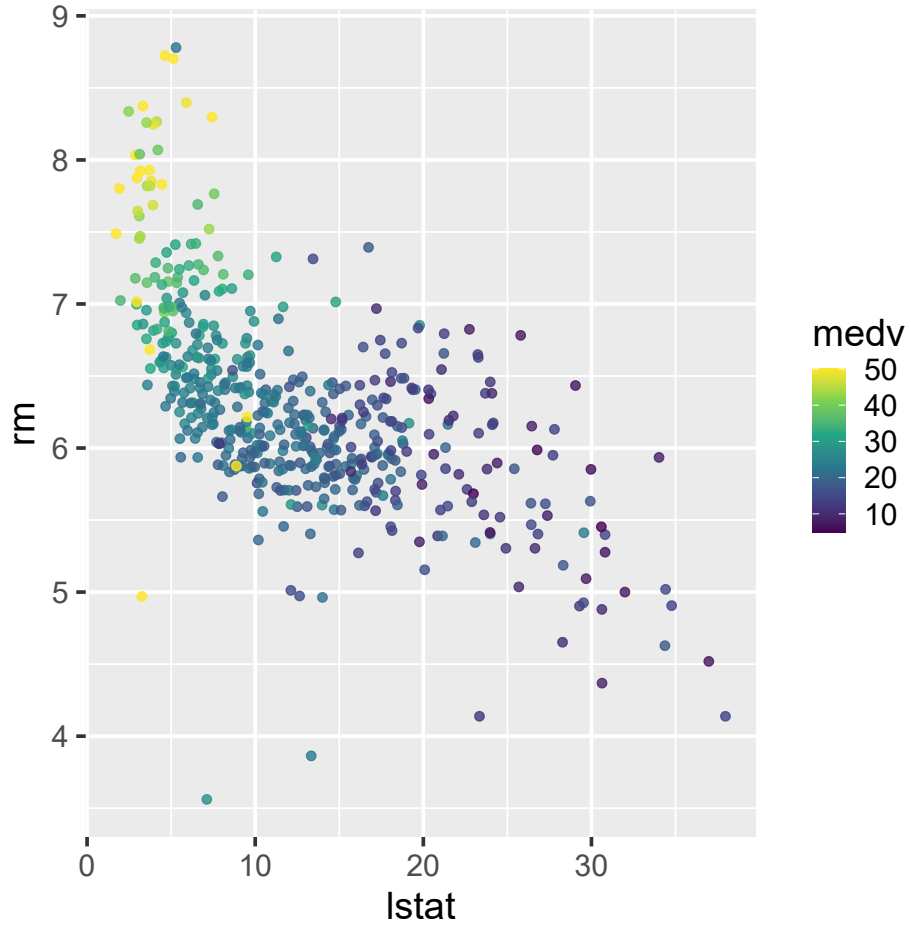
```
rpart.plot(tree_prune, cex = 2)
```



Trees with multiple features

How would you partition the data?

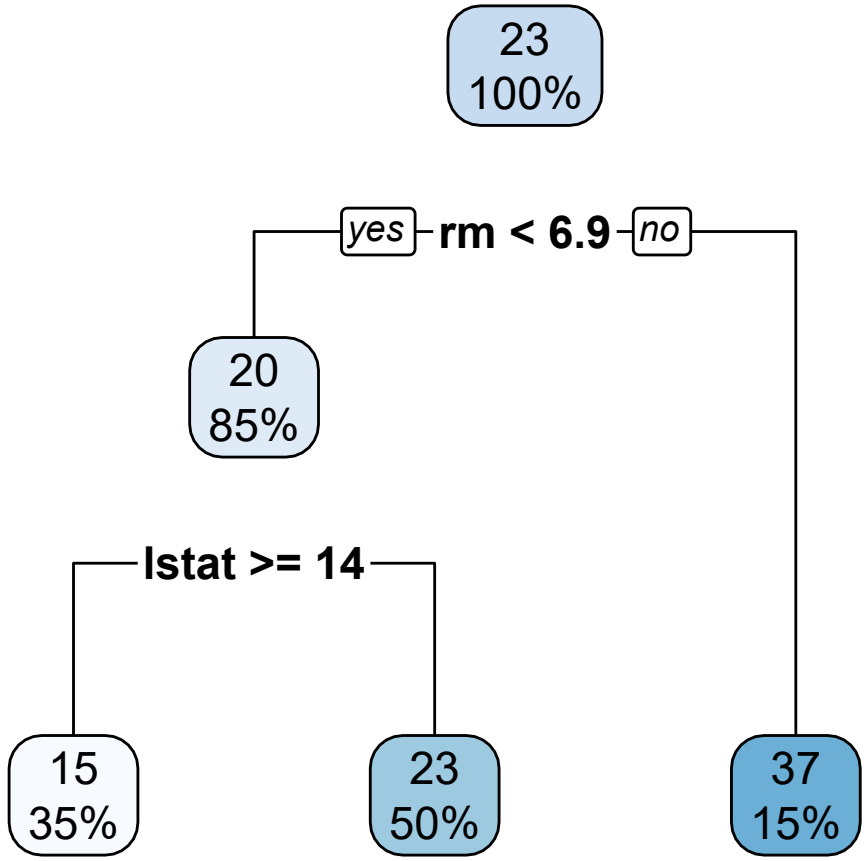
```
boston %>%
  ggplot(aes(lstat, rm, color = medv)) +
  scale_color_viridis_c() +
  geom_point(size = 2, alpha = 0.8)
```



`rpart` with many features

```
tree_fit <- rpart(
  medv ~ lstat + rm,
  data = boston,
  control = rpart.control(cp = 0.15),
  method = "anova"
)
rpart.plot(tree_fit, cex = 2)
```

For now, we will ignore the `cp = 0.15` argument.



Assessing Variable Importance

- After estimating a tree, we often want to evaluate how significant each feature is to our prediction.
- Variable Importance Measure: Commonly, we use the total decrease in the RSS caused by splits over a given variable. (Breiman et al., 1984.)
- Rule of Thumb: Variables that appear higher or more frequently in the tree are generally more important than those that appear lower or less frequently.

Boston variable importance

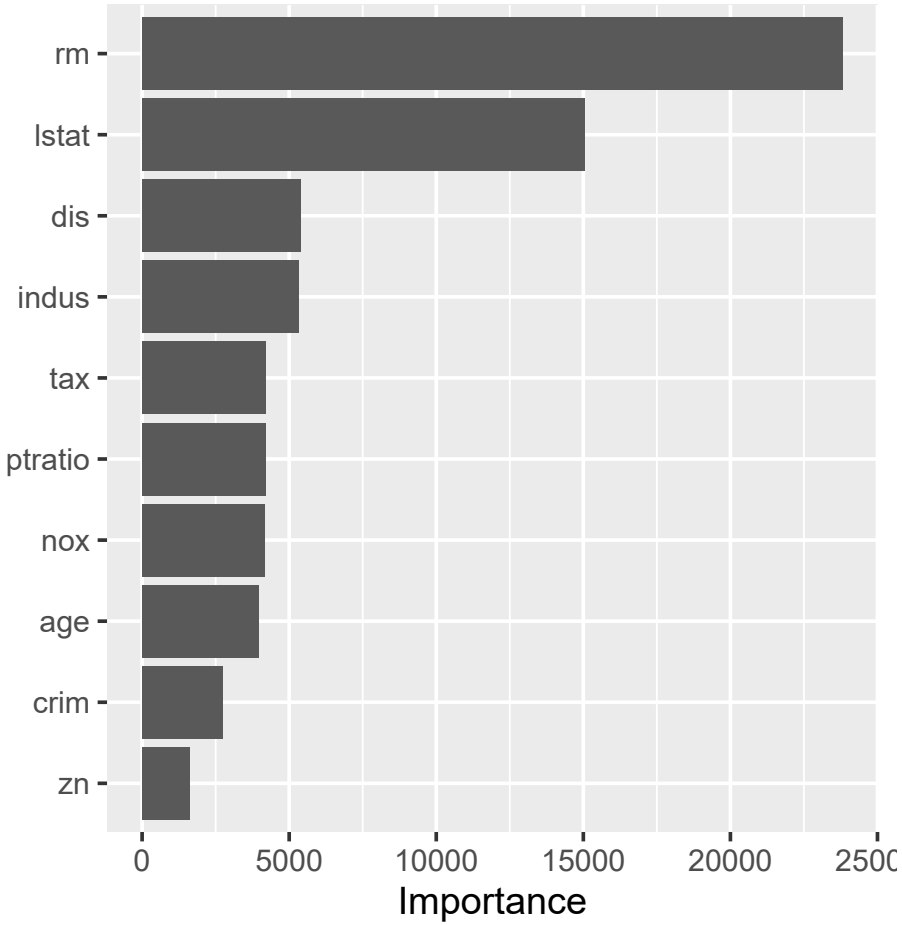
We'll now fit a tree using all features from the Boston dataset:

```
tree_all_vars <- rpart(medv ~ ., data = boston)
```

We'll use the `{vip}` package to display variable importance for the fitted tree:

```
vip(tree_all_vars)
```

As we can see, `rm` (the number of rooms) is the most significant feature in predicting `medv`.



Classification Trees

Adjustment to classification tasks: Splits

- Instead of RSS, splits are typically based on the *Gini index* (a.k.a *node purity*), defined by

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

  a measure of total variance across the total classes (this is `rpart`'s default.)

- An alternative to the Gini index is *cross-entropy*, given by

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

Adjustment for Classification: Prediction

- In classification tasks, the prediction method differs from regression. Instead of relying on the average y in region R_m, predictions are made using a majority rule.
- Each observation belongs to the most frequently occurring class among the training observations in its corresponding region.

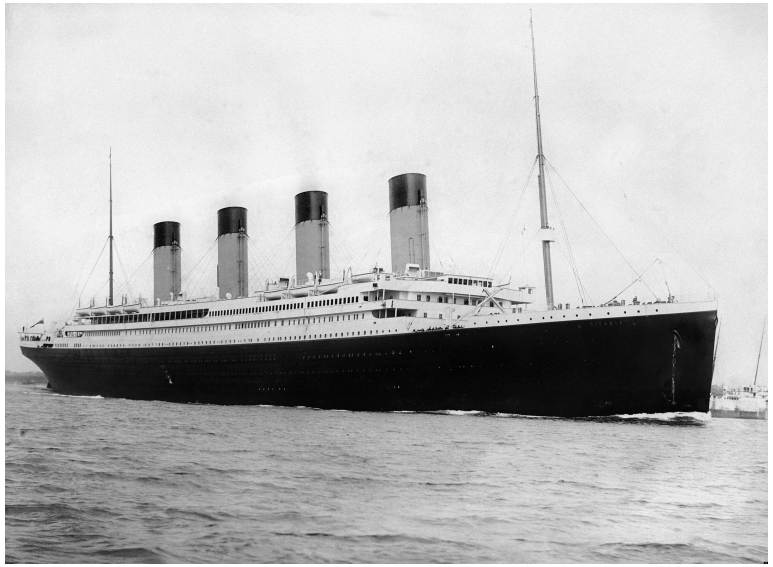
Adjustment for Classification: Variable Importance

- In classification tasks, variable importance is determined by measuring how much the Gini index or cross-entropy is reduced by splits over a specific variable.

Example of Classification Trees: The Titanic Dataset

"The RMS Titanic was a British passenger liner that sank in the North Atlantic Ocean in the early morning hours of 15 April 1912, after it collided with an iceberg during its maiden voyage from Southampton to New York City. There were an estimated 2,224 passengers and crew aboard the ship, and more than 1,500 died, making it one of the deadliest commercial peacetime maritime disasters in modern history."

— Wikipedia



Loading the Data

Let's reproduce the results from Varian's (2014) "Big data: New tricks for econometrics":

```
titanic_raw <-
  here("07-trees-forests/data", "titanic_varian.csv") %>%
  read_csv()
```

```
titanic_raw %>% glimpse()
```

```
## Rows: 1,309
## Columns: 14
## $ pclass     [3m [38;5;246m<dbl> [39m [23m 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
## $ survived   [3m [38;5;246m<dbl> [39m [23m 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1,~
## $ name       [3m [38;5;246m<chr> [39m [23m "Allen, Miss. Elisabeth Walton", "Allison, Master. Hudson Trevor", "~
## $ sex        [3m [38;5;246m<chr> [39m [23m "female", "male", "female", "male", "female", "male", "female", "mal~
## $ age        [3m [38;5;246m<dbl> [39m [23m 29.00, 0.92, 2.00, 30.00, 25.00, 48.00, 63.00, 39.00, 53.00, 71.00, ~
## $ sibsp      [3m [38;5;246m<dbl> [39m [23m 0, 1, 1, 1, 1, 0, 1, 0, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,~
## $ parch      [3m [38;5;246m<dbl> [39m [23m 0, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0,~
## $ ticket     [3m [38;5;246m<chr> [39m [23m "24160", "113781", "113781", "113781", "113781", "19952", "13502", "~
## $ fare       [3m [38;5;246m<dbl> [39m [23m 211.3375, 151.5500, 151.5500, 151.5500, 151.5500, 26.5500, 77.9583, ~
## $ cabin      [3m [38;5;246m<chr> [39m [23m "B5", "C22 C26", "C22 C26", "C22 C26", "C22 C26", "E12", "D7", "A36"~
## $ embarked   [3m [38;5;246m<chr> [39m [23m "S", "S", "S", "S", "S", "S", "S", "S", "S", "C", "C", "C", "C", "S"~
## $ boat       [3m [38;5;246m<chr> [39m [23m "2", "11", NA, NA, NA, "3", "10", NA, "D", NA, NA, "4", "9", "6", "B~
## $ body       [3m [38;5;246m<dbl> [39m [23m NA, NA, NA, 135, NA, NA, NA, NA, NA, 22, 124, NA, NA, NA, NA, NA, NA~
## $ home.dest  [3m [38;5;246m<chr> [39m [23m "St Louis, MO", "Montreal, PQ / Chesterville, ON", "Montreal, PQ / C~
```

Data Details

In this lecture, our focus will be on a single outcome variable and two features:

Variable	Role	Definition	Values
`survived`	Outcome	Survival	0 = No, 1 = Yes
`age`	Feature	Age in years	
`pclass`	Feature	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd

Our goal: Predict the likelihood of passenger survival based on their age and ticket class.

Preprocessing

To facilitate further analysis, let's handle missing values (`NA`) and convert the `survived` variable to a factor:

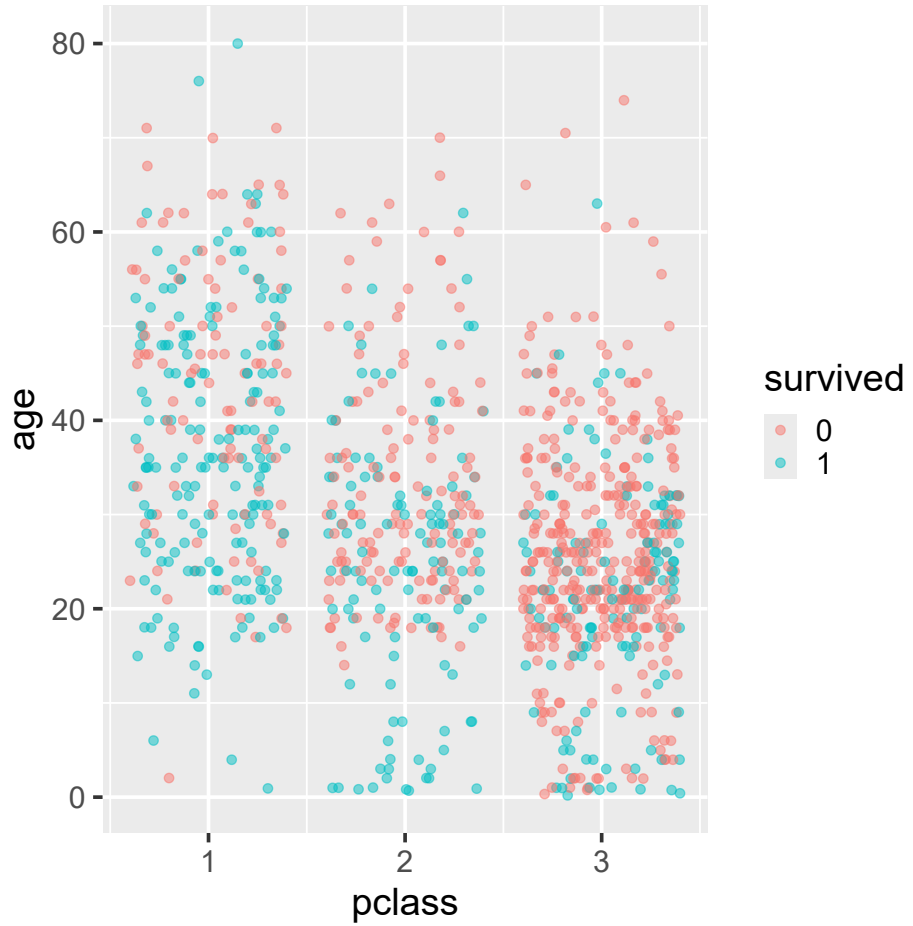
```
titanic <-titanic_raw %>%
  select(survived, age, pclass) %>%
  drop_na() %>%
  mutate(
    survived = as_factor(survived),
  )

titanic
```

```
## # A tibble: 1,046 x 3
##    survived   age pclass
##    <fct>    <dbl>  <dbl>
##  1 1        29         1
##  2 1         0.92      1
##  3 0         2         1
##  4 0        30         1
##  5 0        25         1
##  6 1        48         1
##  7 1        63         1
##  8 0        39         1
##  9 1        53         1
## 10 0        71         1
## # i 1,036 more rows
```

Partition

How would you stratify the data?

```
titanic %>%
  ggplot(aes(pclass, age, color = survived)) +
  geom_jitter(alpha = 0.5, size = 2)
```



Estimate , Prune, and Plot the Tree

Fit the tree

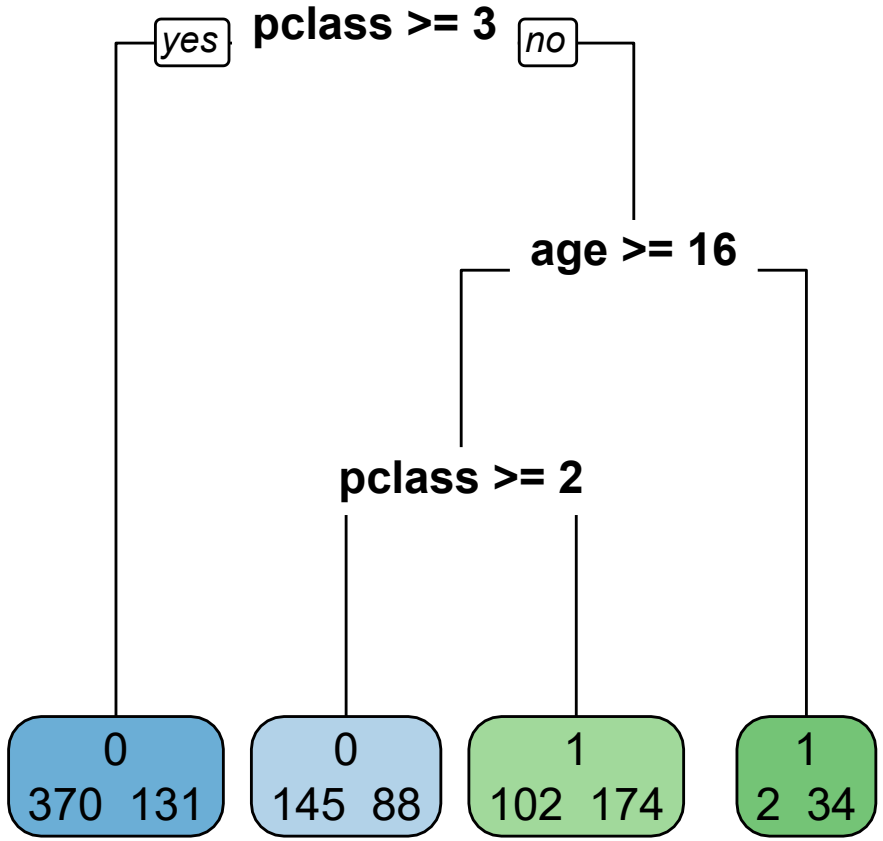
```
rpart_model <- rpart(
  survived ~ pclass + age,
  data = titanic,
  method = "class"
)
```

Prune

```
rpart_prune <- prune(rpart_model, cp = .038)
```

Plot

```
rpart.plot(rpart_prune, type=0, extra=1, cex = 2)
```

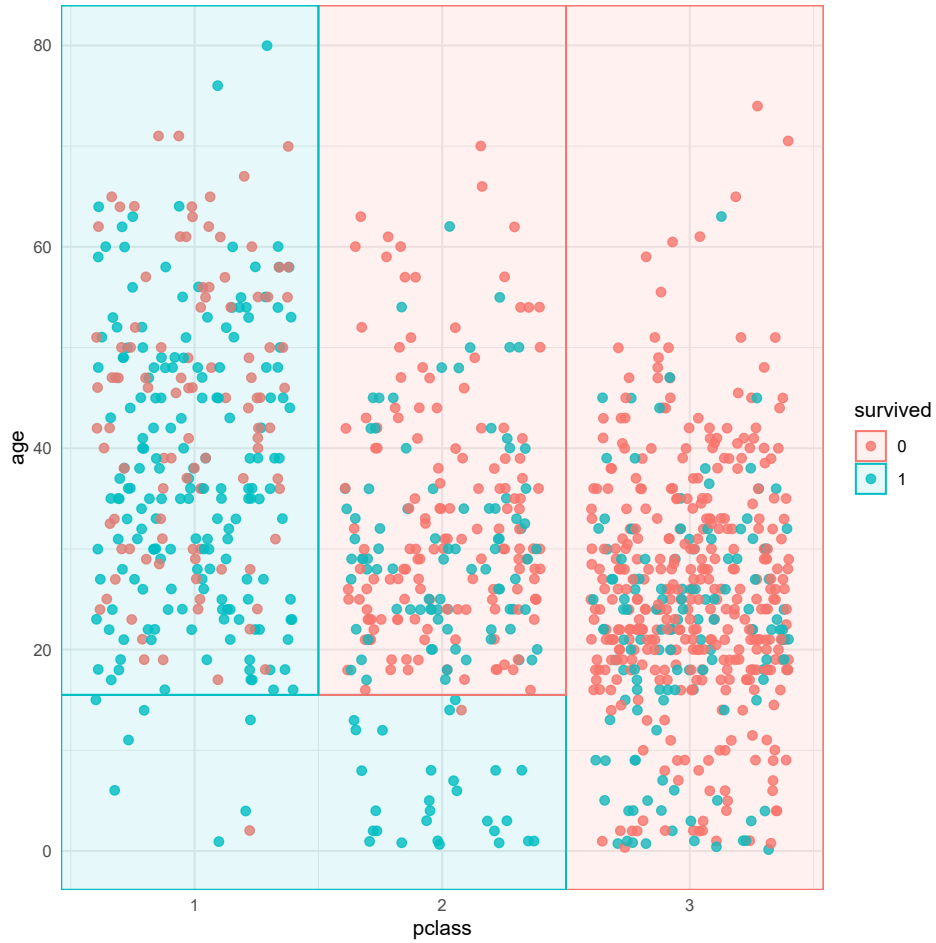


Partitioning a 2-D Feature Space

This is how the tree partitions the data

```
titanic %>%
  ggplot(aes(pclass, age, color = survived)) +
  geom_jitter(alpha = 0.8, size = 2) +
  geom_parttree(data = rpart_prune, aes(fill=survive
  theme_minimal()
```

where I've used the `geom_parttree()` function from the {parttree} package (in development.)



Recall: Trees Stratify the Feature Space

Now, let's generate partition dummies that correspond to the partitioning of our tree:

```
titanic_lm <-
  titanic %>%
  mutate(
    survived = as.numeric(survived) - 1,
    class_3 = if_else(pclass == 3, 1, 0),
    class_1_or_2_age_below_16 = if_else(pclass %in% c(1,2) & age < 16, 1, 0),
    class_1_age_above_16 = if_else(pclass == 1 & age >=16, 1, 0),
    class_2_age_above_16 = if_else(pclass == 2 & age >=16, 1, 0),
  ) %>%
  select(survived, starts_with("class_"))

titanic_lm %>% glimpse()
```

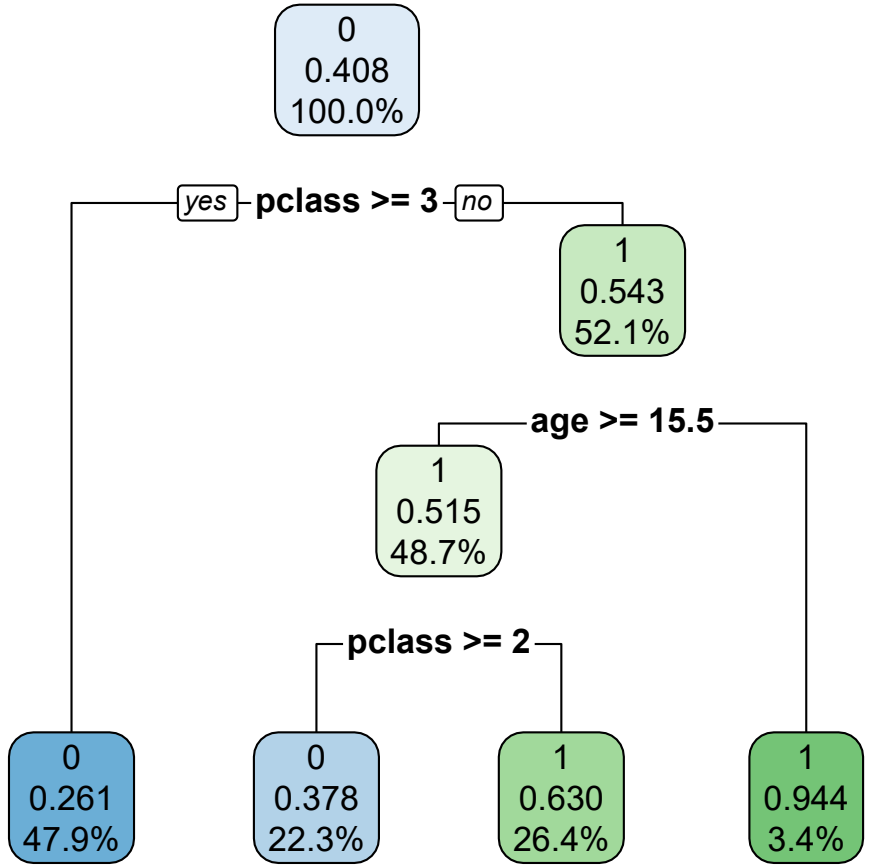
```
## Rows: 1,046
## Columns: 5
## $ survived                 <dbl> 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1~
## $ class_3                  <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ class_1_or_2_age_below_16 <dbl> 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ class_1_age_above_16      <dbl> 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ class_2_age_above_16      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
```

Representing trees as linear regressions

We can now estimate a linear model using our set of partition dummies and compare the results to our tree.

```
lm(survived ~ . -1, data = titanic_lm) %>%
  tidy() %>%
  select(term, estimate, std.error)
```

```
## # A tibble: 4 x 3
##   term                        estimate std.error
##   <chr>                          <dbl>     <dbl>
## 1 class_3                        0.261    0.0204
## 2 class_1_or_2_age_below_16      0.944    0.0762
## 3 class_1_age_above_16           0.630    0.0275
## 4 class_2_age_above_16           0.378    0.0299
```



Random Forests

Trees: Pros and Cons

Pros:

- Intuitive (more than regression?)
- Interpretable
- Nonparametric (no bookkeeping)

Cons:

- Prone to overfitting
- Typically exhibit poor predictive performance

Random Forests: Basic Idea

- According to Breiman (2001), instead of using a single tree, random forests average the predictions of multiple trees. Each tree is fitted to a bootstrapped training sample and uses a subset of the feature space for each split.

- The intuition behind random forests is to reduce variance and combat overfitting by averaging multiple predictions that are noisy and weakly correlated with each other.

The Random Forests Algorithm

Suppose B is the number of bootstrapped samples, representing the number of trees in the forest (typically thousands).

For each $b = 1$ to B:

1. Sample n observations from the data, allowing replacement (bootstrap sampling).

2. Grow a tree T_b using the bootstrap sample. At each split, randomly select a subset of m features (a common choice is $m \approx \sqrt{p}$, where p is the dimension of x).

3. Utilize typical stopping criteria for tree models to determine when a tree is complete. However, do not perform pruning.

Making Predictions

Regression Forests:

- For each observation, the prediction is based on the average of B individual tree predictions:

$$\hat{f}_{\text{RF}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

Here, $\hat{f}^{*b}(x)$ represents the prediction based on the b-th tree in the forest.

Classification Forests:

- For each test observation, the predicted class is determined by taking a majority vote from the predictions of all B trees.

Out-of-Bag Error Estimation

- Due to the bootstrap sampling, on average, each bagged tree utilizes approximately two-thirds of the observations from the training set.
- As a result, we can utilize the remaining one-third of the observations as an "out-of-bag" (OOB) validation set.
- With this approach, for the i^{th} observation, we obtain, on average, B predictions from the trees that were not trained on that specific observation, forming the OOB set.

Fitting Forests using `ranger`

Fitting forests in R is straightforward with the {ranger} package, which utilizes the same syntax as {rpart}.

```
rf_fit <- ranger(
  formula = medv ~ .,
  data = boston,
  mtry = 3,
  num.trees = 1000,
  importance = "impurity"
)
```

where:

- `num.trees` is the argument for B, representing the number of trees in the forest.
- `mtry` is the argument for m, which determines the number of features randomly selected before each split.

Please note that the `importance` argument will be used later to construct variable importance measures.

The output of the model

```
rf_fit
```

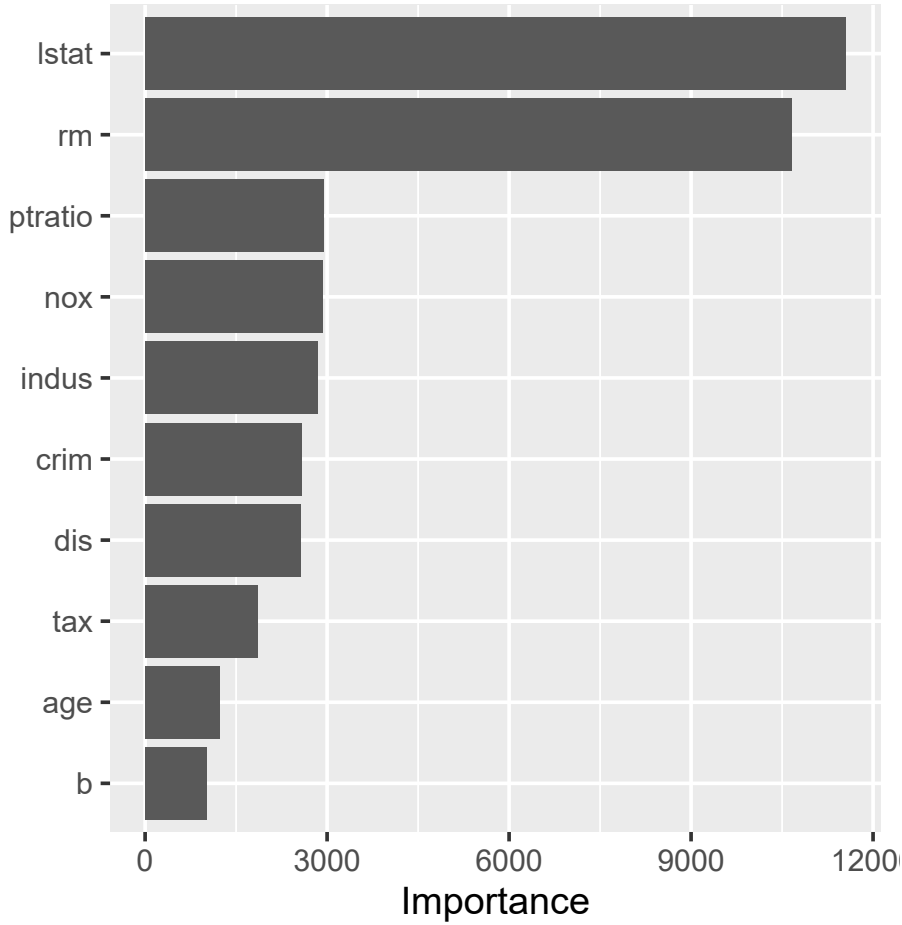
```
## Ranger result
##
## Call:
##  ranger(formula = medv ~ ., data = boston, mtry = 3, num.trees = 1000,      importance = "impurity")
##
## Type:                             Regression
## Number of trees:                  1000
## Sample size:                      506
## Number of independent variables:  13
## Mtry:                             3
## Target node size:                 5
## Variable importance mode:         impurity
## Splitrule:                        variance
## OOB prediction error (MSE):       10.32576
## R squared (OOB):                  0.877927
```

Variable importance

The concept remains the same as in trees, but now we average the effect of a variable over the B trees.

```
rf_fit %>%
  vip()
```

Based on our forest, both `lstat` and `rm` are identified as outperforming the other features.



Other Ensemble Methods

Bagging and Boosting

- Bagging (Bootstrap Aggregating): Similar to random forests, but with $m = p$, where m is the number of features. However, bagging is generally inferior to random forests because the trees tend to be correlated.

- Boosting: This is an example of a "slow learner" approach, where each tree is grown using information from previously grown trees. Boosting can be estimated using the `{gbm}` package.

Note: Boosting algorithms, in general, are highly popular and often achieve high predictive performance.

```
slides |> end()
```

Source code

References

- Breiman L, Friedman J, Olshen R, Stone C (1984). *Classification and Regression Trees*. Chapman and Hall, New York.
- Breiman, L. (2001). Random forests. *Machine learning*, 45(1), 5-32.
- Varian, H. R. (2014). Big data: New tricks for econometrics. *Journal of Economic Perspectives*, 28(2), 3-28.