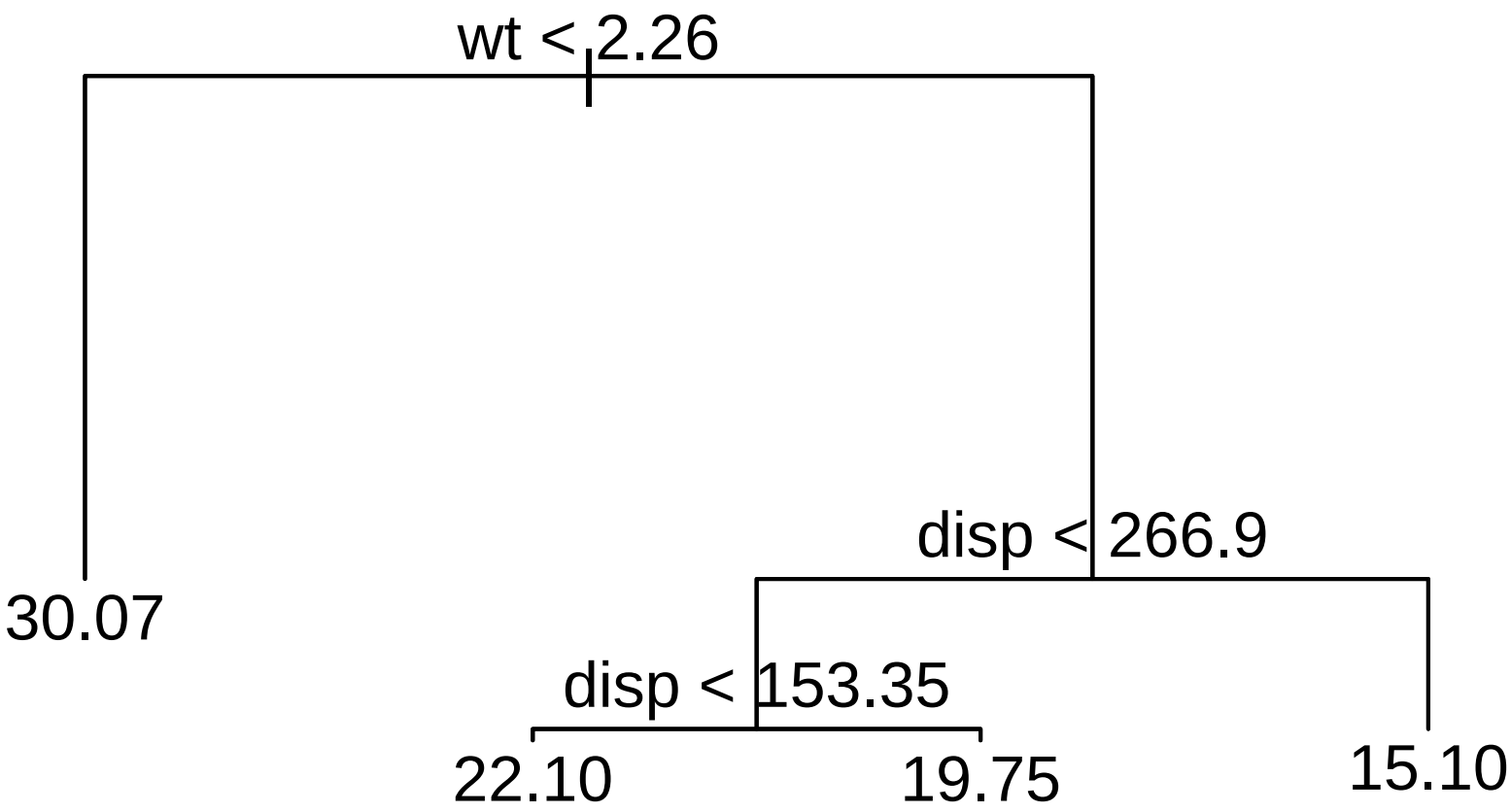
Regression Trees 02

Dr. Saad

Regression Tree (RT) 02 Motivation Example:

Recall: Regression Tree belongs to Tree Based Models or **CART** family. The response variable is **continuous**. `tree` or `rpart` packages are used to fit **RT Example:**

```
library(tree)
rt <- tree(mpg ~ wt + disp , data = mtcars)
plot(rt)
text(rt)
```



rpart package

rpart: Stands for **Recursive Partitioning**. This package is used for advanced Classification and Regression Trees techniques. We will focus on this package throughout this course. **rpart.plot** is another package for plotting the trees.

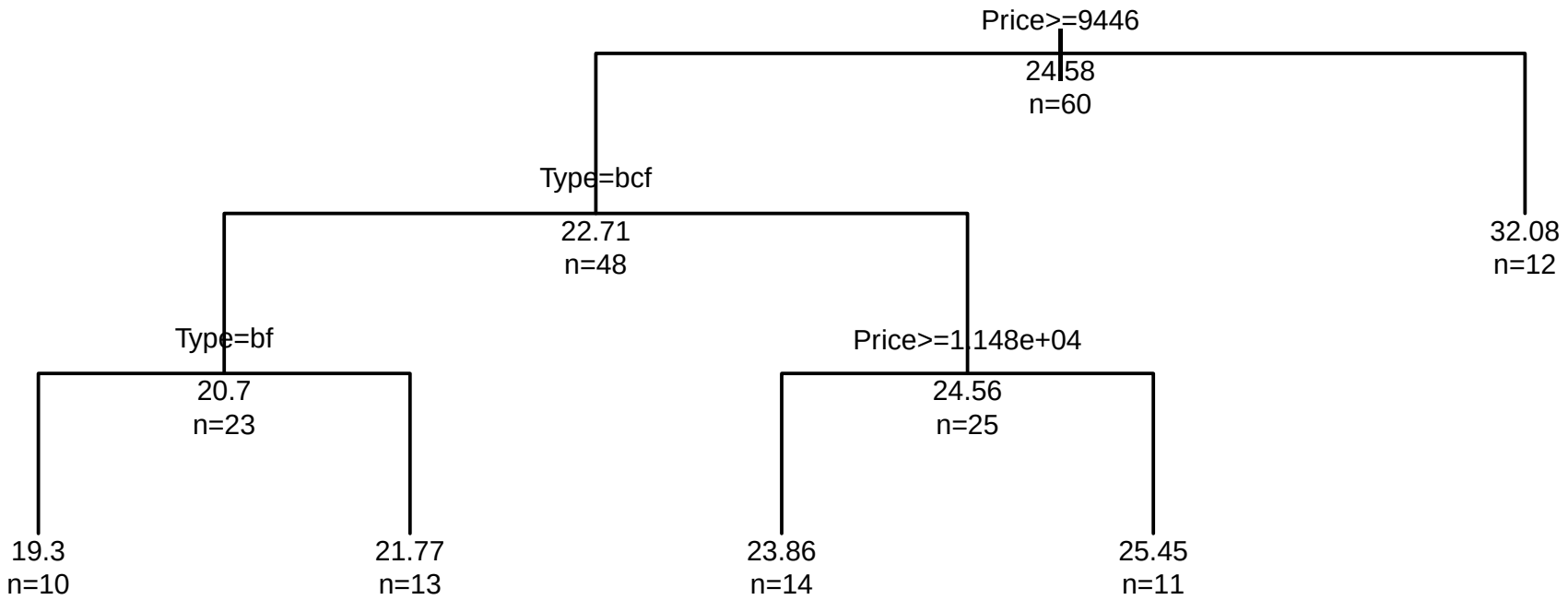
Fitting Regression Tree With rpart package

1. Use the **rpart()** function
2. The first argument is a **formula**
3. Set **method** argument to **anova** because we are doing regression.
4. pass the data to the **data** argument.
5. Use plot() function to plot the model
6. Use text() function to add text to the model

```
library(rpart)
library(rpart.plot)
data("cu.summary")
rt_rpart <- rpart(Mileage ~ Price + Country + Reliability + Type,
                  method = 'anova',
                  data = cu.summary)
#
plot(rt_rpart, uniform = TRUE, margin = 0.1)
text(rt_rpart, use.n = TRUE, all = TRUE, cex=.75)
```

Running RT example:

```
library(rpart)
library(rpart.plot)
data("cu.summary")
rt_rpart <- rpart(Mileage ~ Price + Country + Reliability + Type,
                  method = 'anova',
                  data = cu.summary)
#
plot(rt_rpart, uniform = TRUE, margin = 0.1)
text(rt_rpart, use.n = TRUE, all = TRUE, cex=.5)
```



Process of Building Regression Trees

- **Step:01** Split the values of predictors (vector space) **recursively** X_1, X_2, ..., X_p into J distinct, non-overlapping, regions.
- **Step: 02** . For new observations that falls into the region Rj, we make the same prediction, which is the **mean of response values** for the training observations in Rj.

Example:

- Suppose we have two features X_1 and X_2.
- We perform **step1**, we get for instance two regions R_1 and R_2.
- In R_1 we have $\bar{y}_1$, suppose **3** , and in R_2 we have $\bar{y}_2$, suppose **7**.
- We make predictions like this: if we have a new observation $X_1 = x$, so if $x \in R_1$ then we predict the response as **3**. but if $x \in R_2$ then we predict the response as **7**.

Process of Building Regression Trees Continue

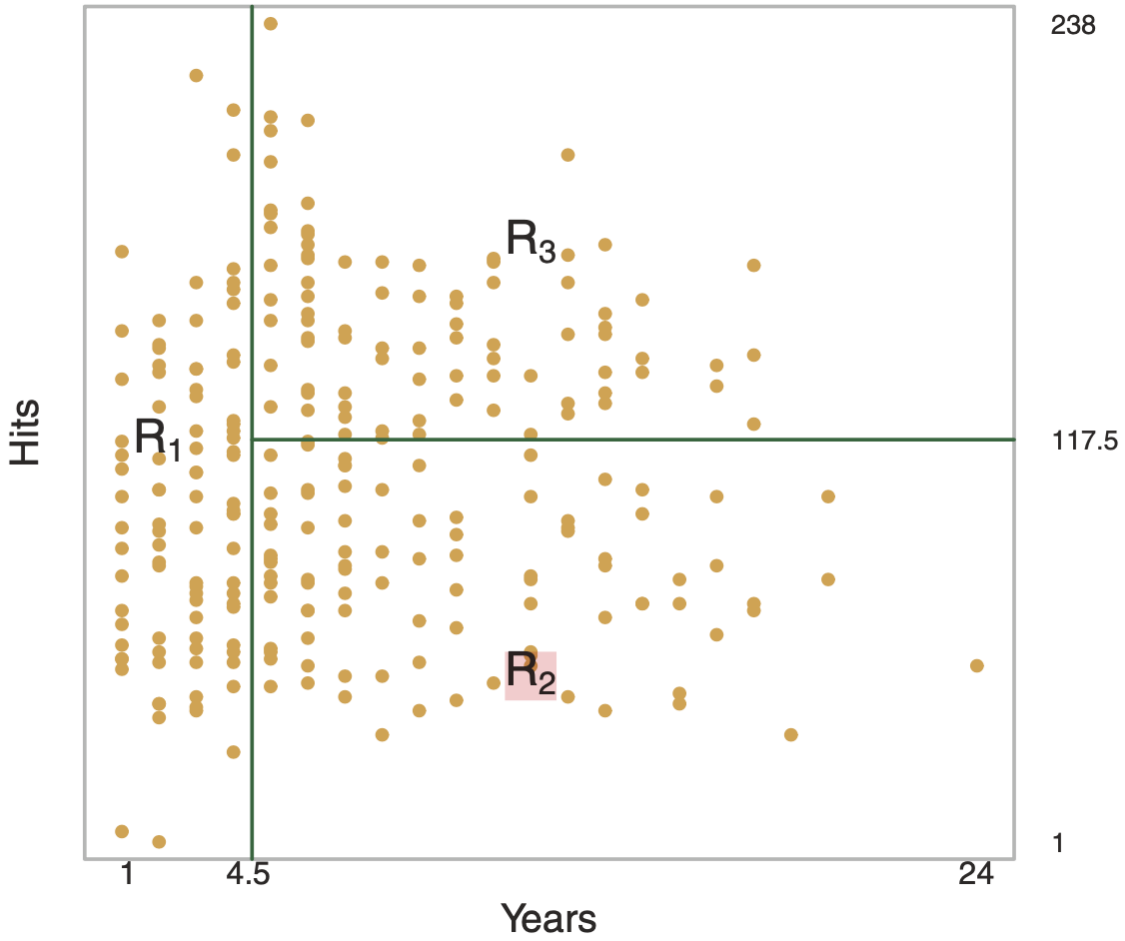


Figure 1: Three-region partition

Process of Building Regression Trees Continue

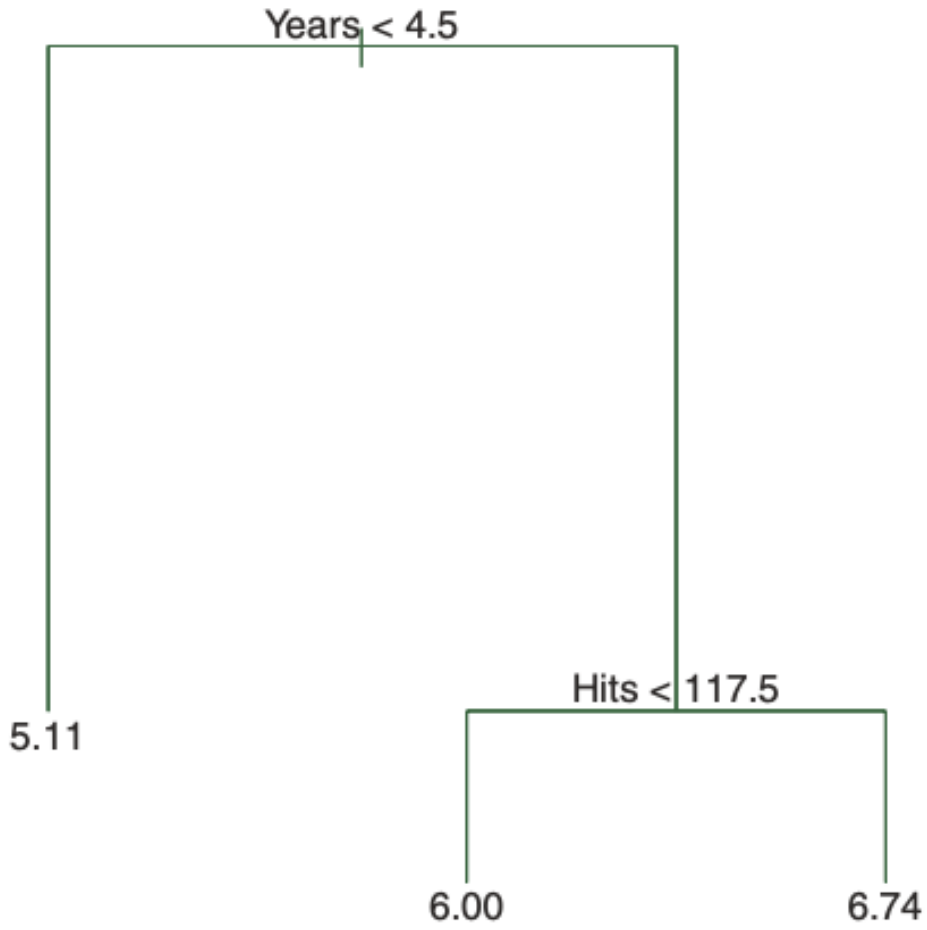


Figure 2: Regression Tree

Binary Split with More Regions

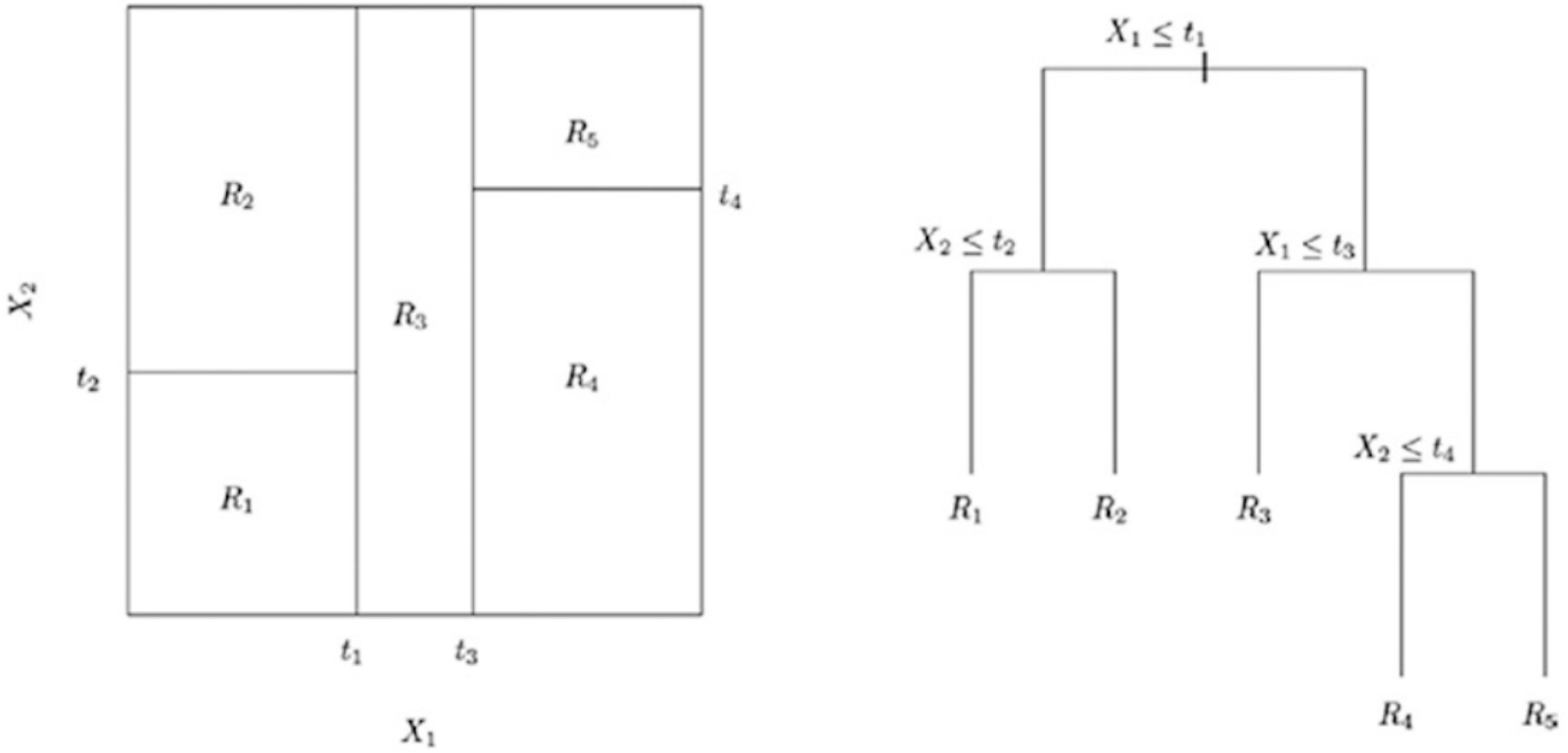


Figure 3: Recursive binary Split and its corresponding tree for two-dimensional feature space

Recursive Binary Splitting

Q: How do we construct the regions $R_1, R_2, ..., R_j$

A: Using **Recursive Binary Splitting**, or **Top-down, greedy** approach.

The Process of Recursive Binary Splitting:

- Consider all the predictor we have $X_1, X_2, ..., X_p$
- Find a cutpoint s (where the split is made).
- s is a cutpoint that splits the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j >= s\}$ that leads to the greatest reduction in **RSS**.
- We consider all possible values of s for each predictor X_j.
- Choose the predictor X_j and a cutpoint s that minimizes the **RSS (Residual Some of Squares)**

The RSS formula is given as follows

$$\sum_{i:x_i R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

$\hat{y}_{R_1}$ is the mean response for the training observations in $R_1(j, s)$. And $\hat{y}_{R_2}$ is the mean response for the training observations in $R_2(j, s)$.

Recursive Binary Splitting Continue

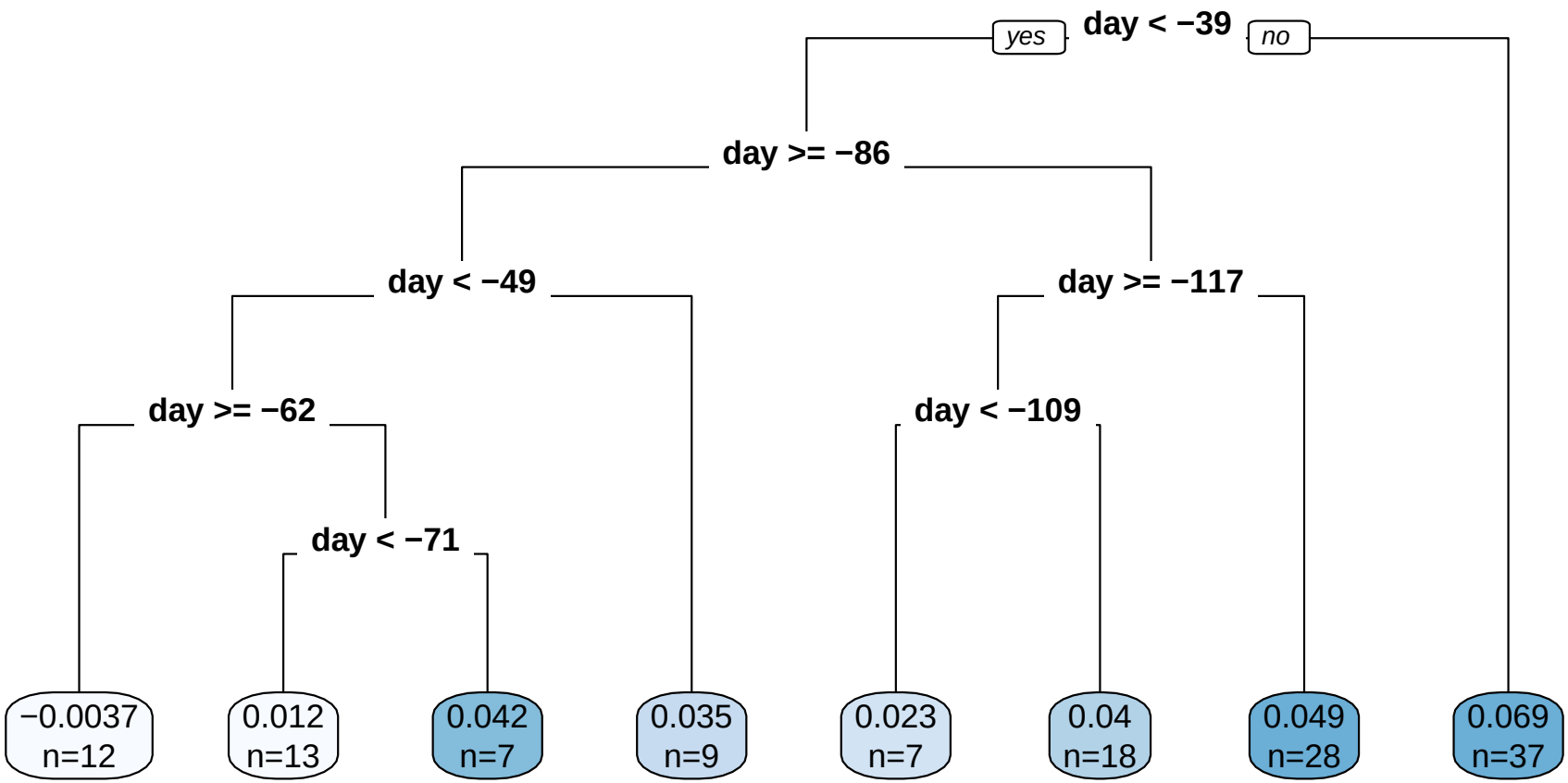
- In the first step we have two regions R_1 and R_2. For each region the mean for response variable will be calculated.
- The process will be repeated **recursively** for the new regions R_1 and R_2, which each region might be split into two other regions.
- The process will continue until it reaches some conditions, they are called stopping conditions (which we will see in a later slide)
- Once the regions $R_1, ..., R_j$ have been created, the predicted response for new observations will be the mean of the training data in the region to which that new data point belongs.

Example of Recursive Binary Splitting

2008 polls in US Election Between Obama and McCain

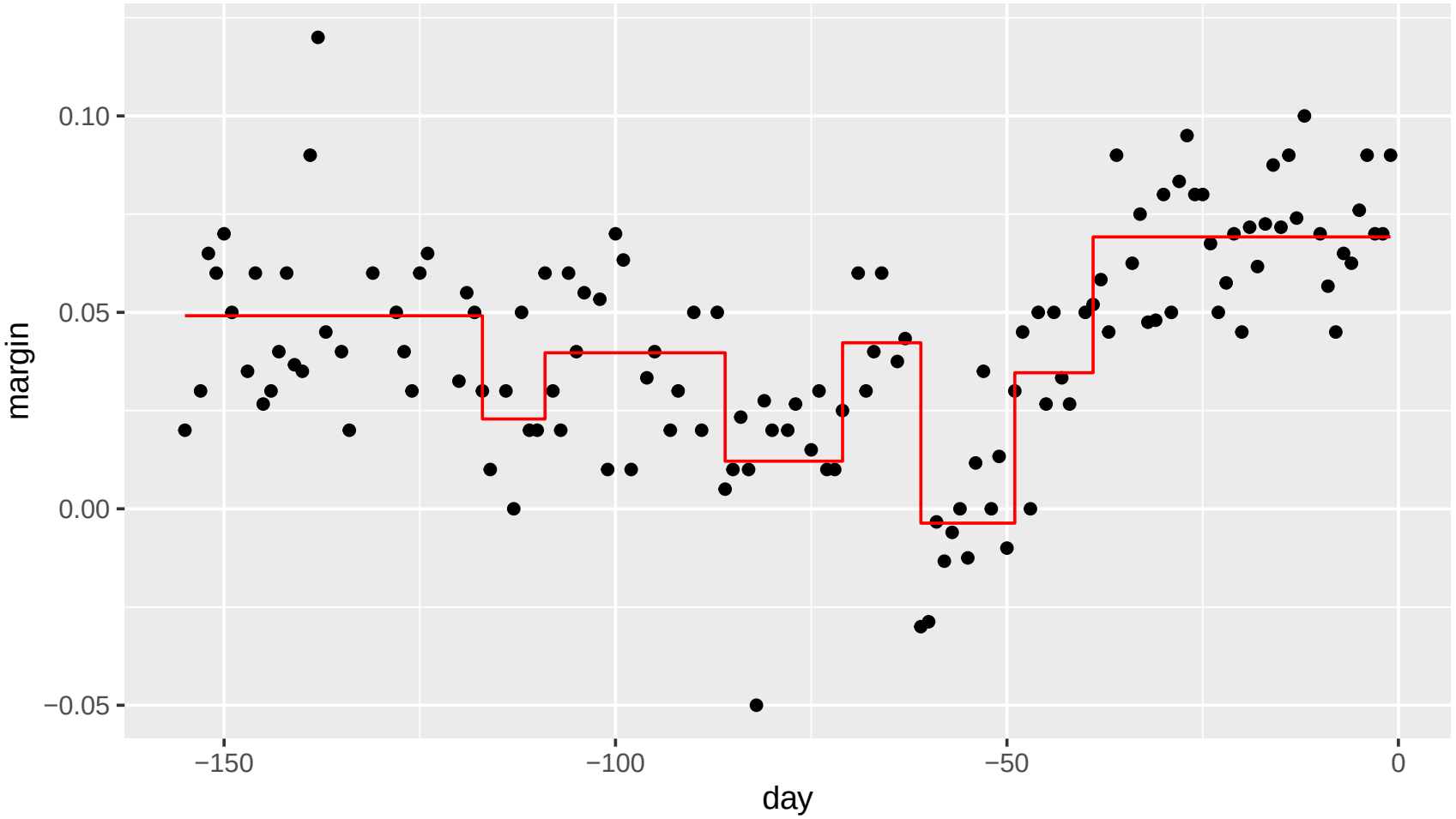
```
library(dslabs, quietly = TRUE)
data("polls_2008")
rt_fit <- rpart(margin ~ . , method = 'anova', data = polls_2008)

rpart.plot(rt_fit,yesno = 1, type = 0, extra = 1,  cex = 0.75)
```



Prediction with Regression Tree

```
polls_2008 %>% mutate(y_hat = predict(rt_fit)) %>%
  ggplot() +
  geom_point(aes(day, margin)) +
  geom_step(aes(day, y_hat), color = "red")
```



Questions:

1. **in the previous example, why does splitting stop at 8 leaves?**
2. **What will happen if we keep splitting?**

The Answer:

If the algorithm keeps splitting until each point is in its own partition, the **RSS** will be **zero**, since the mean of a value is the same value. **Isn't this what we want!**. Minimizing **RSS**, means we are doing the greatest job. However, this is not the case, because **Regression Trees** tend to **overfit** easily.

Now we answer the first question, the algorithm stopped splitting at **8** because of some controlling parameters. We see them through **rpart** function

Rpart control

Now we answer the first question, the algorithm stopped splitting at **8** because of some controlling parameters. We see them through **rpart** function

- **Control** parameter: this argument is passed to the **rpart()** through a function **rpart.control()**
- The **rpart.control()** function:

```
args(rpart.control)
```

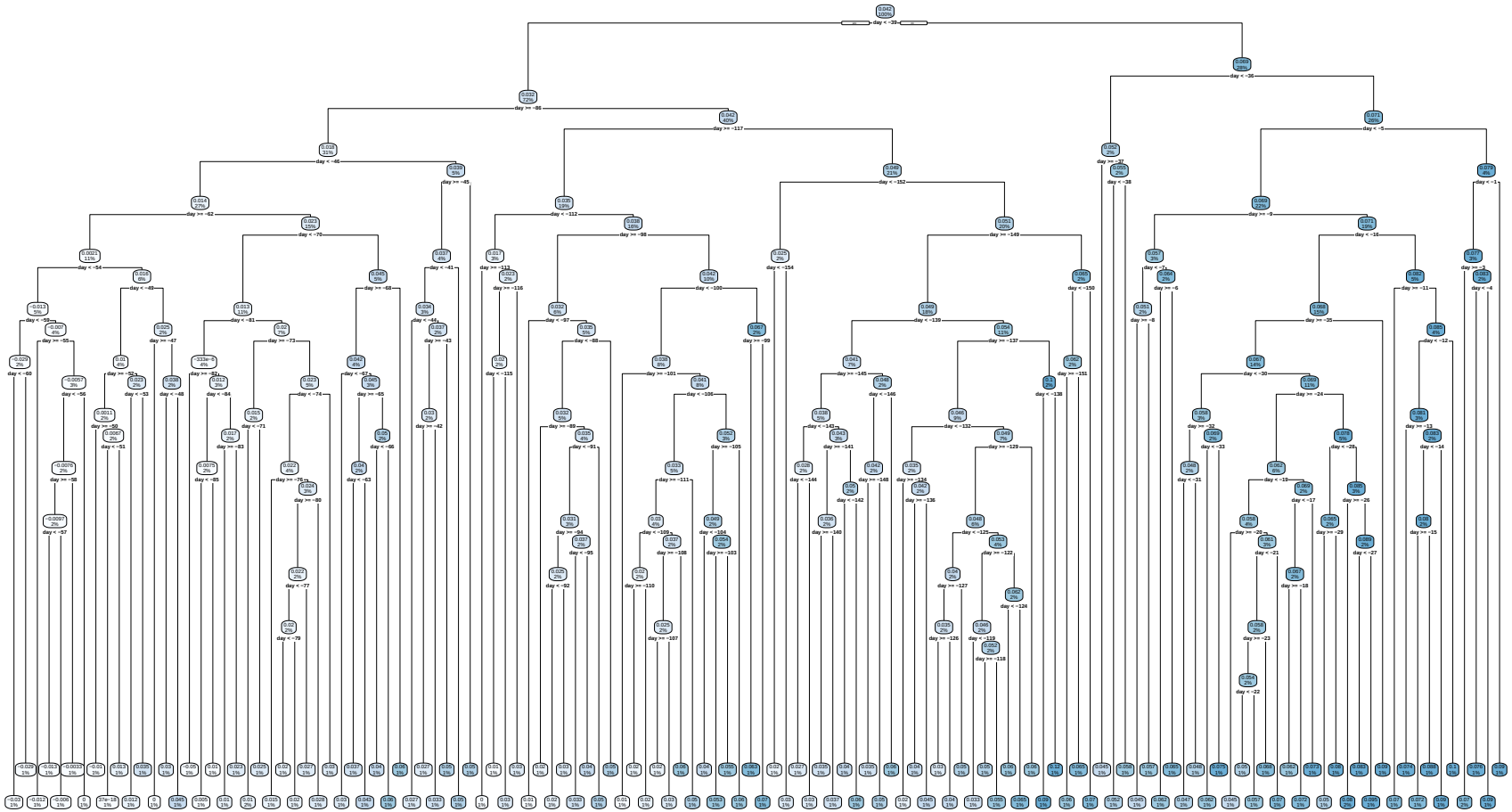
```
| function (minsplit = 20L, minbucket = round(minsplit/3), cp = 0.01,
|     maxcompete = 4L, maxsurrogate = 5L, usesurrogate = 2L, xval = 10L,
|     surrogatestyle = 0L, maxdepth = 30L, ...)
| NULL
```

- **Complexity Parameter, "cp"**: Every time we split into two regions, the **RSS** decreases, **but by how much?**, **cp** controls by how much the **RSS** should decrease to continue splitting. If the decrease in RSS is more than the **cp** then the split will stop. **Note, the smaller cp the more complex the tree is (more splits)**
- **Minsplit**: the minimum number of observations that must exist in a node in order to attempt another split.
- **Minbucket**: the minimum number of observations in a terminal node.
- **Maxdepth**: depth of a tree, root node is 0. It limits the maximum number of nodes between a leaf node and the root node.

Application of rpart.control

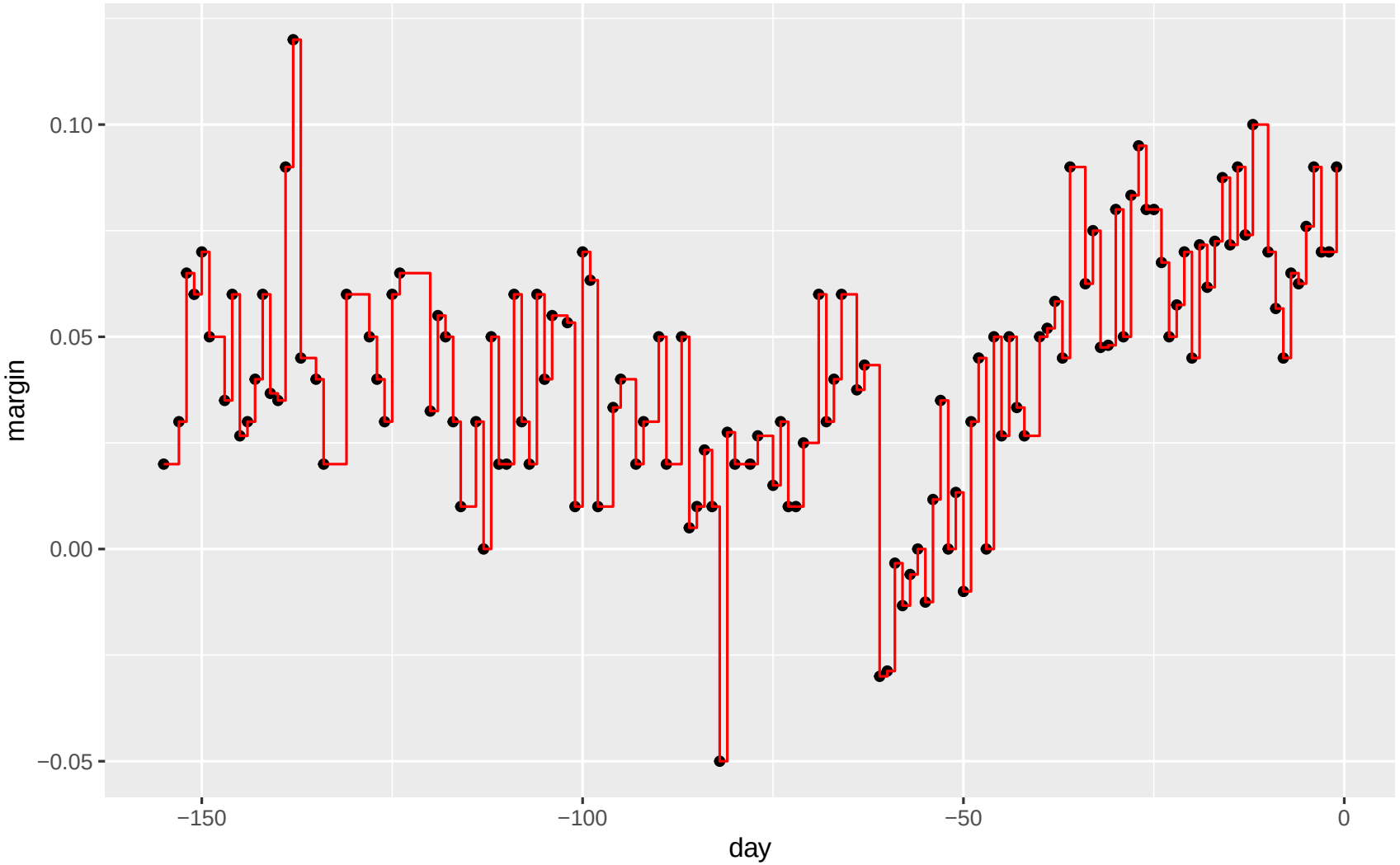
This example will show overfitting

```
cp_fit <- rpart(margin ~ . , data = polls_2008,
                control = rpart.control(cp = 0, minsplit = 2))
rpart.plot(cp_fit)
```



Plotting the prediction

```
polls_2008 %>% mutate(y_hat = predict(cp_fit)) %>%
  ggplot() +
  geom_point(aes(day, margin)) +
  geom_step(aes(day,y_hat), color = "red")
```



Useful Functions for Gridsearch.

You may already have asked yourself, how can I determine cp or minsplit?

first we need to familiarize ourselves with few functions

1. **seq() function**: it generates a sequence of numbers, it takes a start (from), **end** (to), step (by), and optional **length**.

Examples:

```
# Generate a sequence from 1 to 10
seq(1, 10)   #by is 1 by default
```

```
|  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# Generate a sequence from 1 to 50 by 5
seq(from = 1, to = 50, by = 5)  # by = 5
```

```
|  [1]  1  6 11 16 21 26 31 36 41 46
```

```
# Generate 10 number between 0 and 1
seq(0, 1, length = 10)
```

```
|  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
|  [8] 0.7777778 0.8888889 1.0000000
```

Useful Functions for Gridsearch (continue)

What if we want all combinations between sequences of numbers?

In Regression tree we search for combinations among two or more **hyper-parameters** (we will thoroughly discuss them later)

2. expand.grid() function: it generates all combinations between sequences

Examples

```
a <- 1:3
b <- c(2, 5)
expand.grid(a, b)
```

```
|   Var1 Var2
| 1    1    2
| 2    2    2
| 3    3    2
| 4    1    5
| 5    2    5
| 6    3    5
```

```
# we can use expand_grid from tidyr package
expand_grid(a, b)
```

```
| # A tibble: 6 x 2
|       a     b
|   <int> <dbl>
| 1     1     2
| 2     1     5
| 3     2     2
| 4     2     5
| 5     3     2
| 6     3     5
```

Gridsearch for Regression Tree

- Suppose we want to search for a best combination between **cp** and **minsplit**
- First: We create a grid as follows

```
cp <- c(0.001, 0.01, 0.1)
ms <- c(10, 20, 50)
# Grid
grid <- expand.grid(cp, ms)
grid
```

```
|    Var1 Var2
| 1 0.001   10
| 2 0.010   10
| 3 0.100   10
| 4 0.001   20
| 5 0.010   20
| 6 0.100   20
| 7 0.001   50
| 8 0.010   50
| 9 0.100   50
```

Training Regression Trees with Caret

Training a regression tree with caret `train()` function is straightforward. It can be done as follows:

1. Pass the formula of our model as a first argument
2. Pass method = "rpart"
3. pass the data to data arg
4. We use **TuneGrid** argument in the train function, **We must pass a data.frame** to it with named columns with the necessary hyperparameter names

The syntax is as follows:

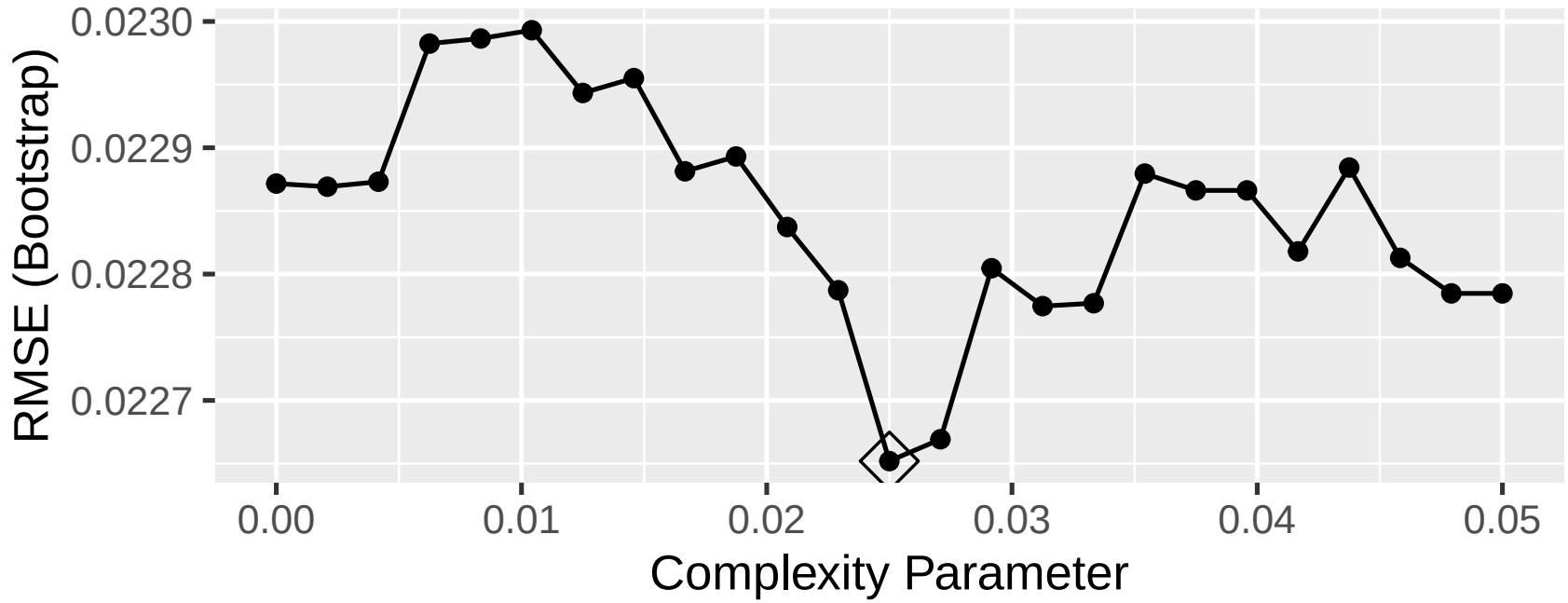
```
train_rpart <- train(margin ~ . ,
                     method = "rpart",
                     TuneGrid = data.frame(cp = seq(0, 0.09, length = 10)),
                     data = polls_2008)
```

Cross Validation for determining complexity parameter

```
library(caret)
train_rpart <- train(margin ~ .,
                     method = "rpart",
                     tuneGrid = data.frame(cp = seq(0, 0.05, len = 25)),
                     data = polls_2008)
names(train_rpart)
```

```
|  [1] "method"       "modelInfo"    "modelType"    "results"      "pred"
|  [6] "bestTune"     "call"         "dots"         "metric"       "control"
| [11] "finalModel"   "preProcess"   "trainingData" "resample"     "resampledCM"
| [16] "perfNames"    "maximize"     "yLimits"      "times"        "levels"
| [21] "terms"        "coefnames"    "xlevels"
```

```
ggplot(train_rpart, highlight = TRUE)
```



How Cross-Validation Works for cp Tuning with Train()

Train() has an argument `trControl = trainControl()` argument; Thus, it suffices for us to understand how `trainControl()` word.

```
args(trainControl)
```

```
| function (method = "boot", number = ifelse(grepl("cv", method),
|     10, 25), repeats = ifelse(grepl("[d_]cv$", method), 1, NA),
|     p = 0.75, search = "grid", initialWindow = NULL, horizon = 1,
|     fixedWindow = TRUE, skip = 0, verboseIter = FALSE, returnData = TRUE,
|     returnResamp = "final", savePredictions = FALSE, classProbs = FALSE,
|     summaryFunction = defaultSummary, selectionFunction = "best",
|     preProcOptions = list(thresh = 0.95, ICAcomp = 3, k = 5,
|         freqCut = 95/5, uniqueCut = 10, cutoff = 0.9), sampling = NULL,
|     index = NULL, indexOut = NULL, indexFinal = NULL, timingSamps = 0,
|     predictionBounds = rep(FALSE, 2), seeds = NA, adaptive = list(min = 5,
|         alpha = 0.05, method = "gls", complete = TRUE), trim = FALSE,
|     allowParallel = TRUE)
| NULL
```

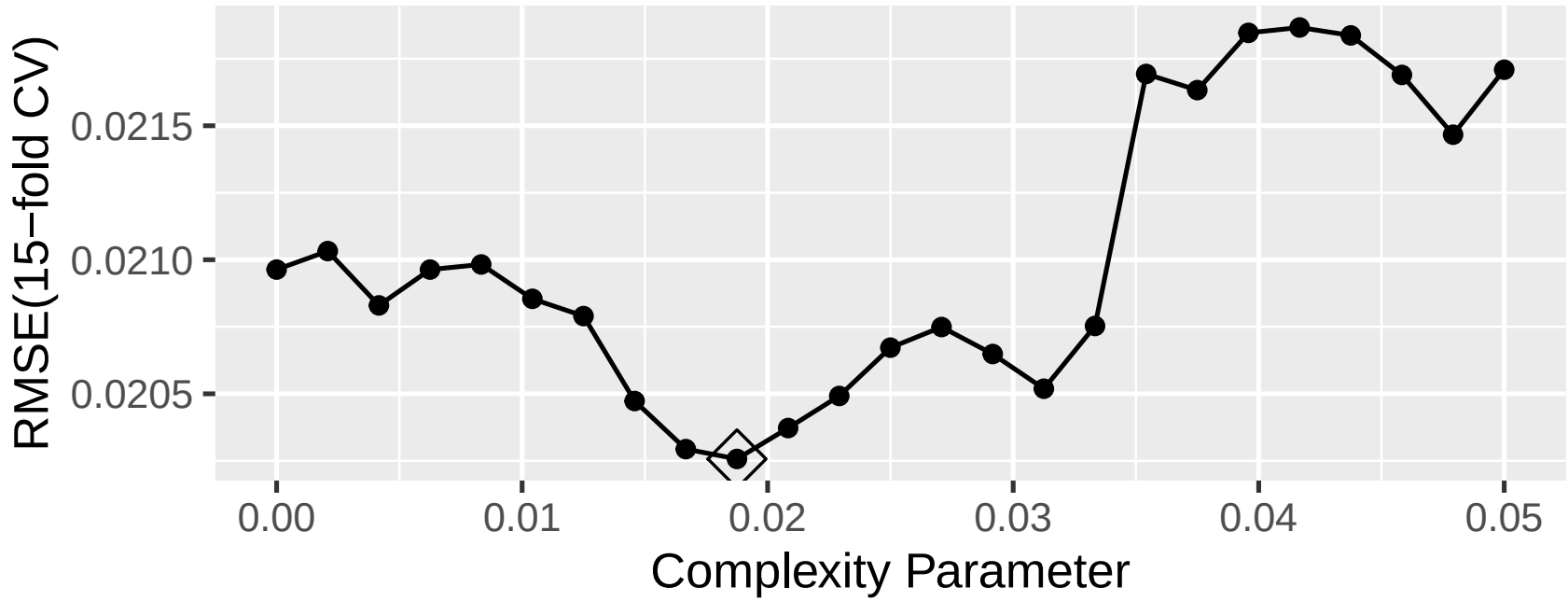
The default behavior for resampling of train function is **bootstrapping** method. So, if we want to change it, we can pass **cv** (cross-validation) to method argument. Note as well, if `method='boot'` train will do **25-fold** validation, but if `method="cv"`, a 10-fold cross-validation will be used. Of course we can control that as well as follows:

```
ctr <- trainControl(method = "cv", number = 15)
```

CrossValidation **cv** Example 01

we are going to do a 15-fold **cv** with the previous example:

```
cv_model <- train(margin ~ .,
                  method = "rpart",
                  tuneGrid = data.frame(cp = seq(0, 0.05, len = 25)),
                  data = polls_2008,
                trControl = trainControl(method = "cv",
                                         number = 15))
ggplot(cv_model, highlight = TRUE) + ylab("RMSE(15-fold CV)")
```



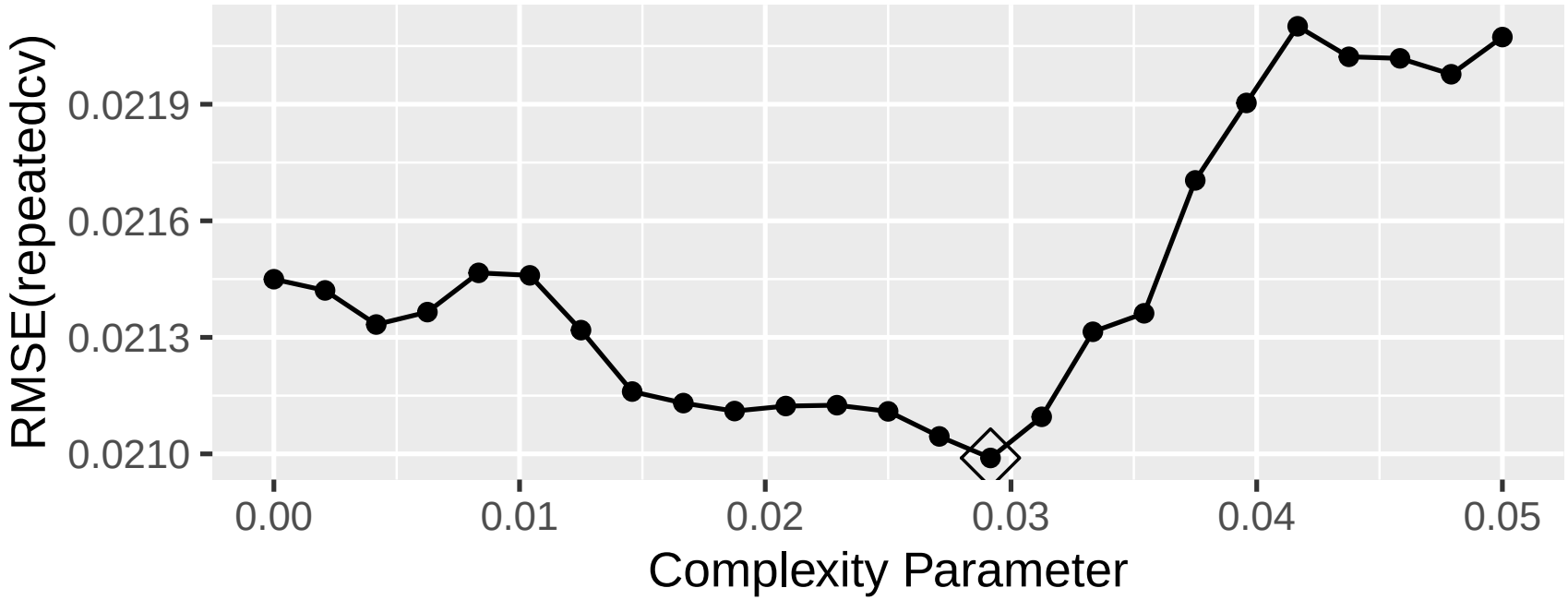

```
cv_model$bestTune
```

```
|          cp
| 10 0.01875
```

Repeated CrossValidation **repeatedcv** Example 02

We do 5 repeats of 10-Fold CV for the previous example

```
rep_cv_model <- train(margin ~ .,
                      method = "rpart",
                      tuneGrid = data.frame(cp = seq(0, 0.05, len = 25)),
                      data = polls_2008,
                    trControl = trainControl(method = "repeatedcv",
                                             repeats = 5))
ggplot(rep_cv_model, highlight = TRUE) + ylab("RMSE(repeatedcv)")
```



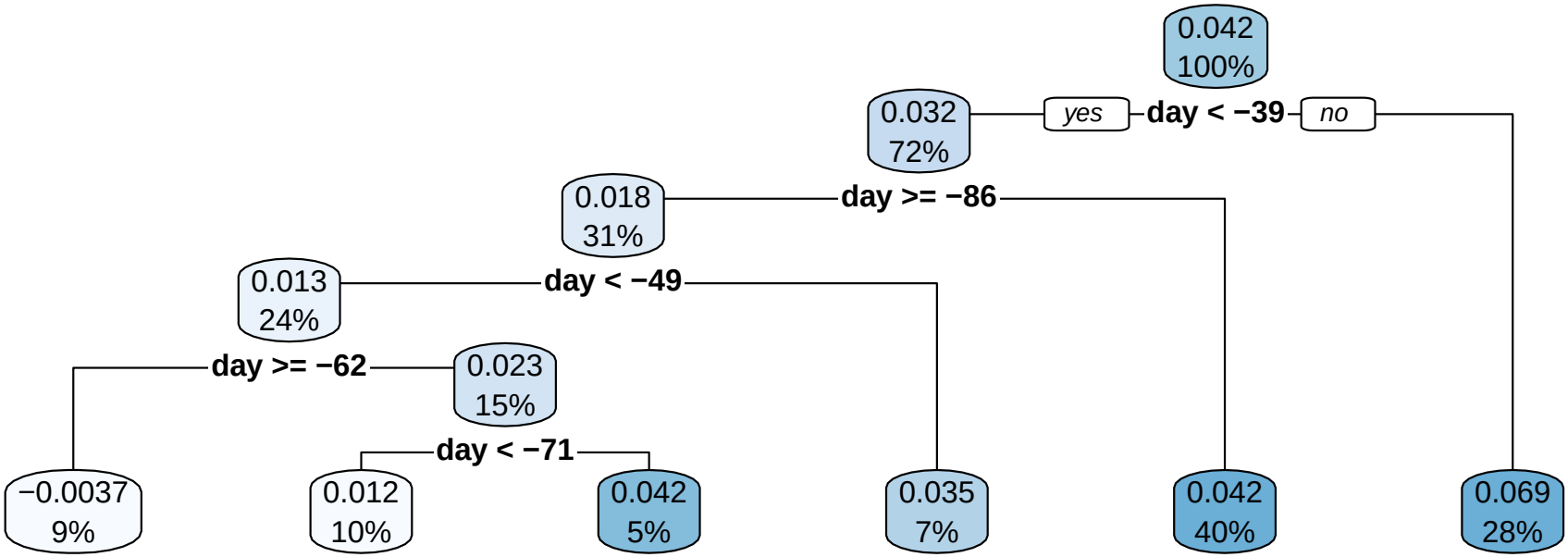

```
rep_cv_model$bestTune
```

```
|           cp
| 15 0.02916667
```

Finding the best Regression Tree Model

After fitting the regression tree model with different **cp** values, we can access the final model through **finalModel** attribute.

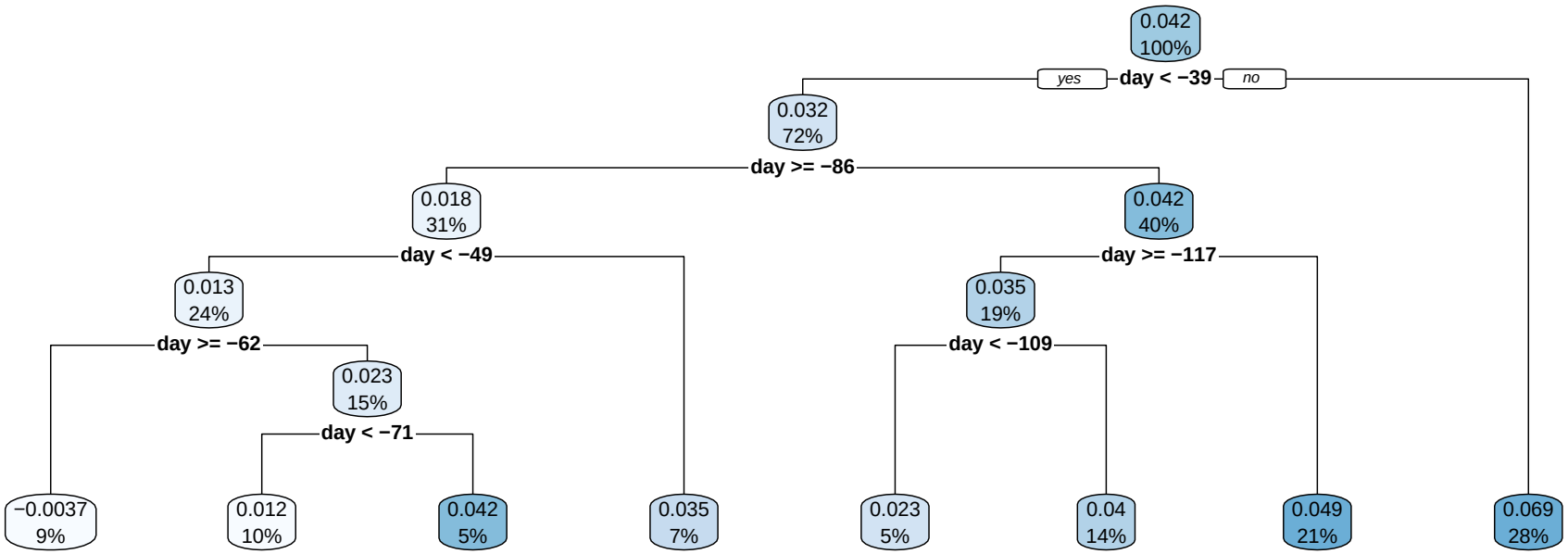
```
rpart.plot(rep_cv_model$finalModel)
```



Pruning a Regression Tree Model

If we already have a tree model, and we want for some reason to apply a different **cp** value. For instance, a higher **cp** value to make the tree smaller and less flexible. we can use the function **prune()** from rpart package.

```
tree_fit <- rpart(margin ~ . , data = polls_2008)
pruned_tree <- prune(tree_fit, cp = 0.01)
rpart.plot(pruned_tree)
```



Applying a Higher cp Value

```
tree_fit <- rpart(margin ~ . , data = polls_2008)
pruned_tree <- prune(tree_fit, cp = 0.03)
rpart.plot(pruned_tree)
```