# Introduction to Machine Learning

Working Group "Computational Statistics" – Bernd Bischl et al.

## Code demo for CART

Classification and decision trees are very powerful, but they do have one major drawback: they are highly unstable. We show this with the following example on the **servo** data set that offers a regression task and looks like that:

```
library(rpart)
library(dplyr)
library(rattle)
library(mlbench)
library(methods)
library(devtools)
library(BBmisc)
library(rpart.plot)
library(ggplot2)
library(mlr)


data("Servo")

# transform ordered factors to numeric
servo <- Servo %>%
  mutate_at(c("Pgain", "Vgain"), as.character) %>%
  mutate_at(c("Pgain", "Vgain"), as.numeric)
rm(Servo)
str(servo)
```

```
## 'data.frame':    167 obs. of  5 variables:
##  $ Motor: Factor w/ 5 levels "A","B","C","D",..: 5 2 4 2 4 5 3 ..
##  $ Screw: Factor w/ 5 levels "A","B","C","D",..: 5 4 4 1 2 3 1 ..
##  $ Pgain: num  5 6 4 3 6 4 3 3 ...
##  $ Vgain: num  4 5 3 2 5 3 2 2 ...
##  $ Class: num  4 11 6 48 6 20 46 49 ...
```

```
head(servo)
```

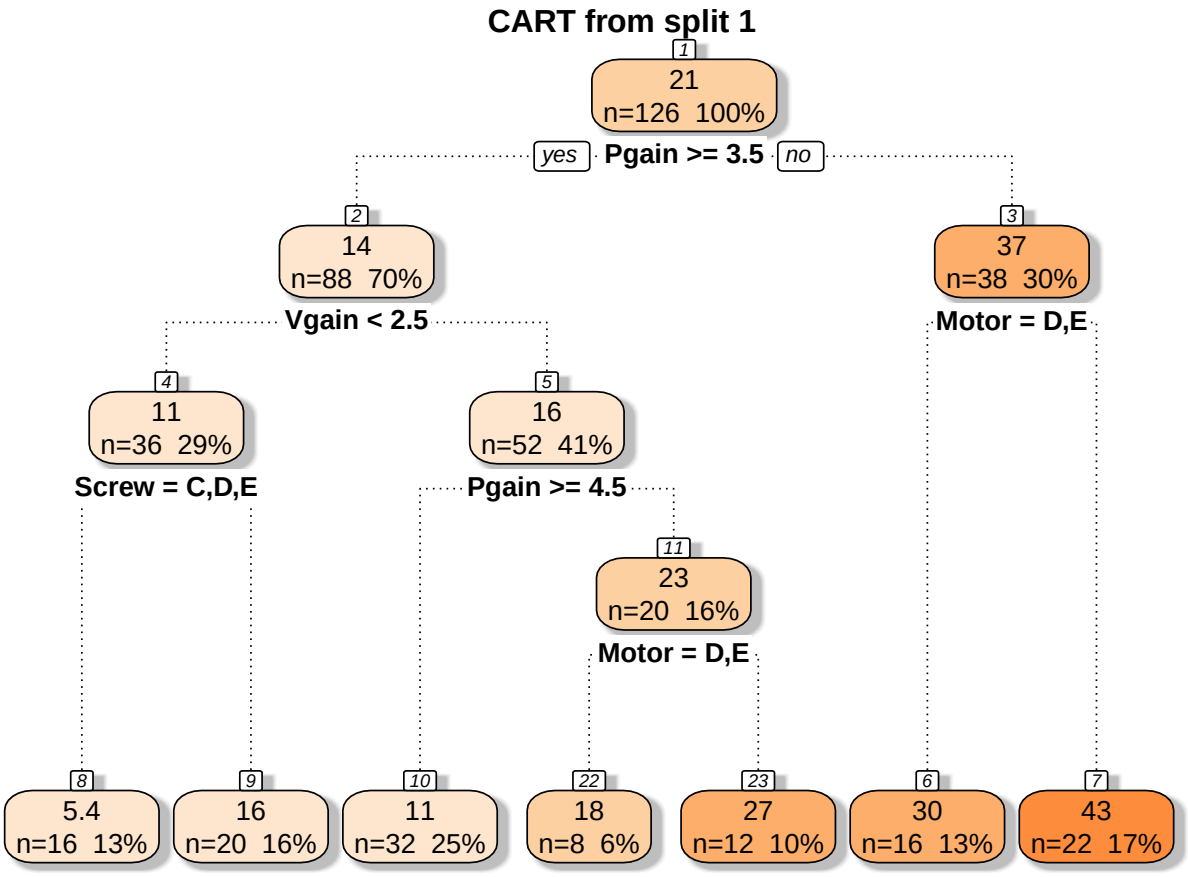
```
##   Motor Screw Pgain Vgain Class
## 1     E     E     5     4     4
## 2     B     D     6     5    11
## 3     D     D     4     3     6
## 4     B     A     3     2    48
## 5     D     B     6     5     6
## 6     E     C     4     3    20
```

We'll fit two CART's on our data, which we split in train and test with two different seeds, resulting in slightly different train and test data sets.

Check the differences in the CART architecture that was induced by those differing seeds:

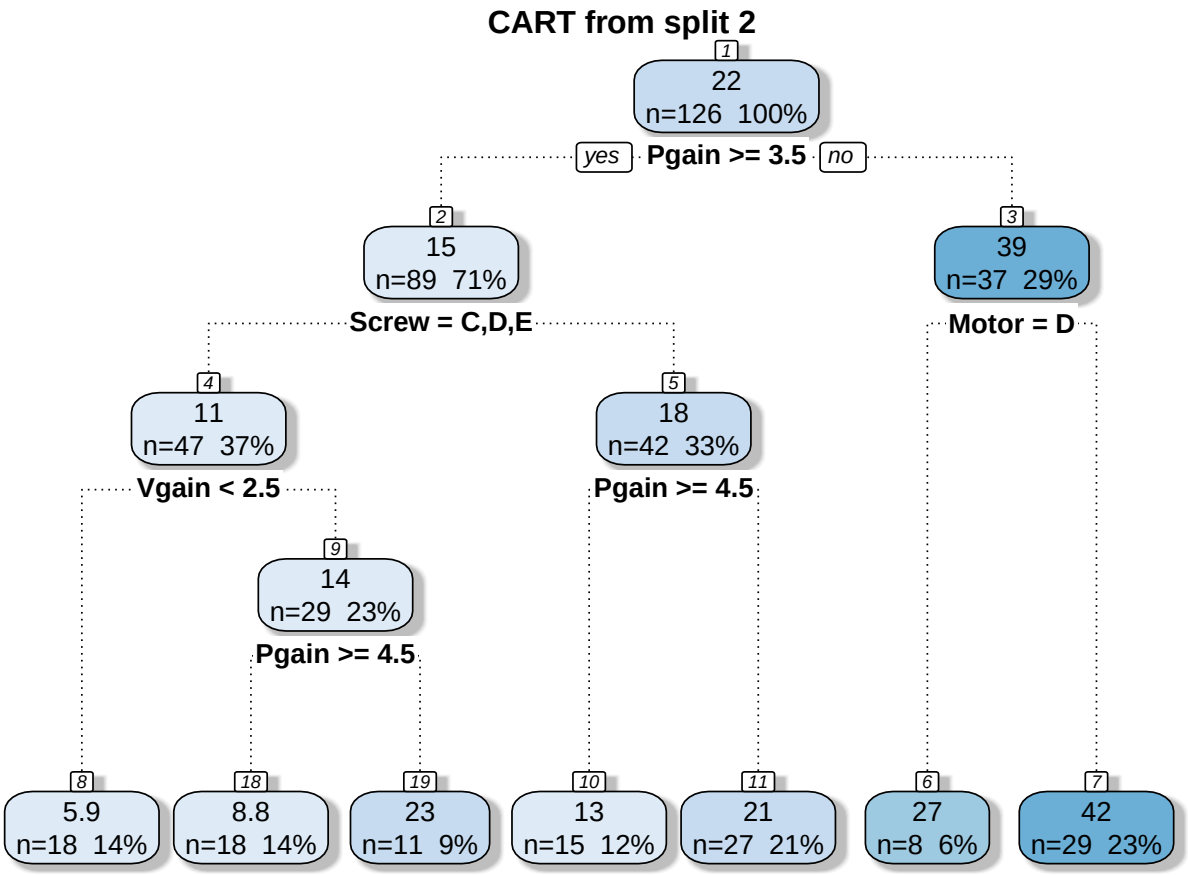
```
# split in train and test with two different seeds to show data dependency
train_size <- 3 / 4

set.seed(1333)
```

```
train_indices <- sample(
  x = seq(1, nrow(servo), by = 1),
  size = ceiling(train_size * nrow(servo)), replace = FALSE
)
train_1 <- servo[ train_indices, ]
test_1 <- servo[ -train_indices, ]

set.seed(1)
train_indices <- sample(
  x = seq(1, nrow(servo), by = 1),
  size = ceiling(train_size * nrow(servo)), replace = FALSE
)
train_2 <- servo[ train_indices, ]
test_2 <- servo[ -train_indices, ]


model_1 <- rpart(Class ~ ., data = train_1)
rattle::fancyRpartPlot(model_1,
  palettes = "Oranges", main =
    "CART from split 1\n", sub = ""
)
```




```
model_2 <- rpart(Class ~ ., data = train_2)
rattle::fancyRpartPlot(model_2,
  palettes = "Blues", main =
```

```
    "CART from split 2\n", sub = ""
)
```



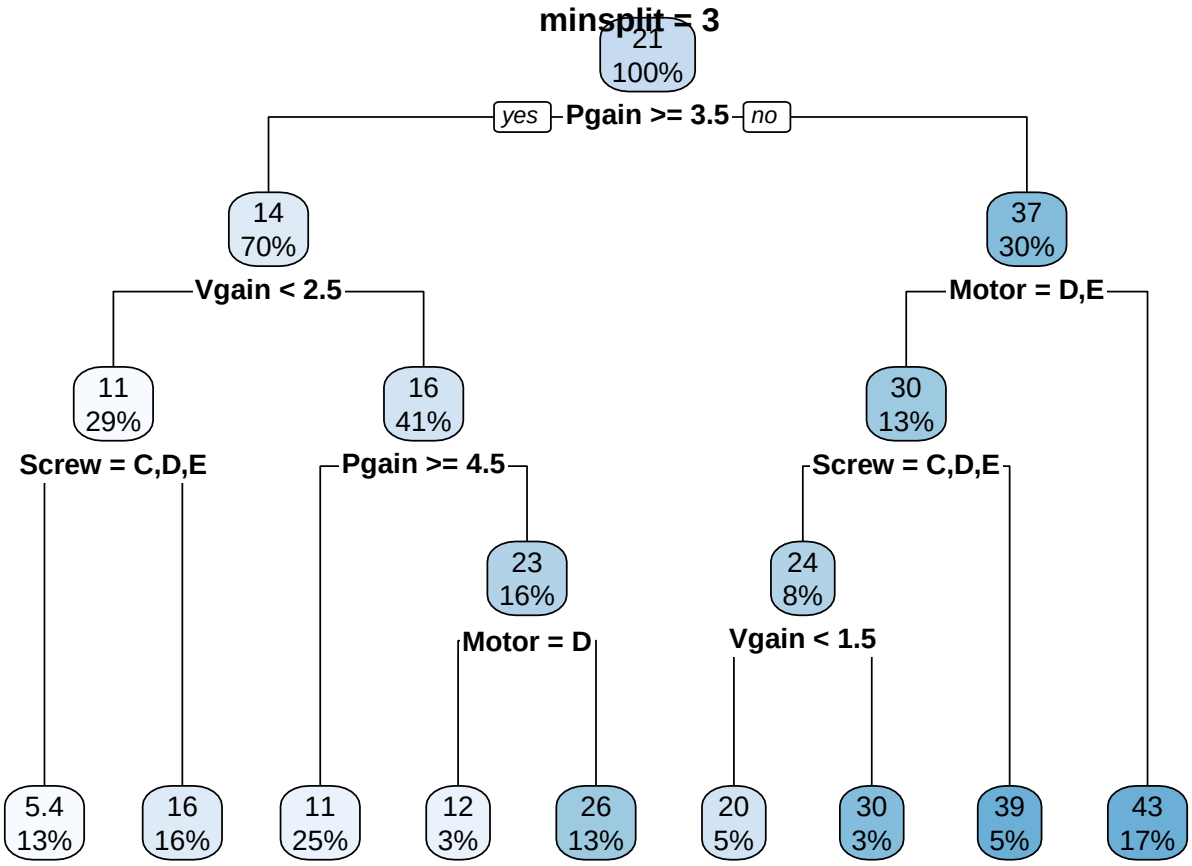

- Why is that a problem? Don't we want an algorithm that is sensitive to changes in the data?
- What can we do to address this issue?

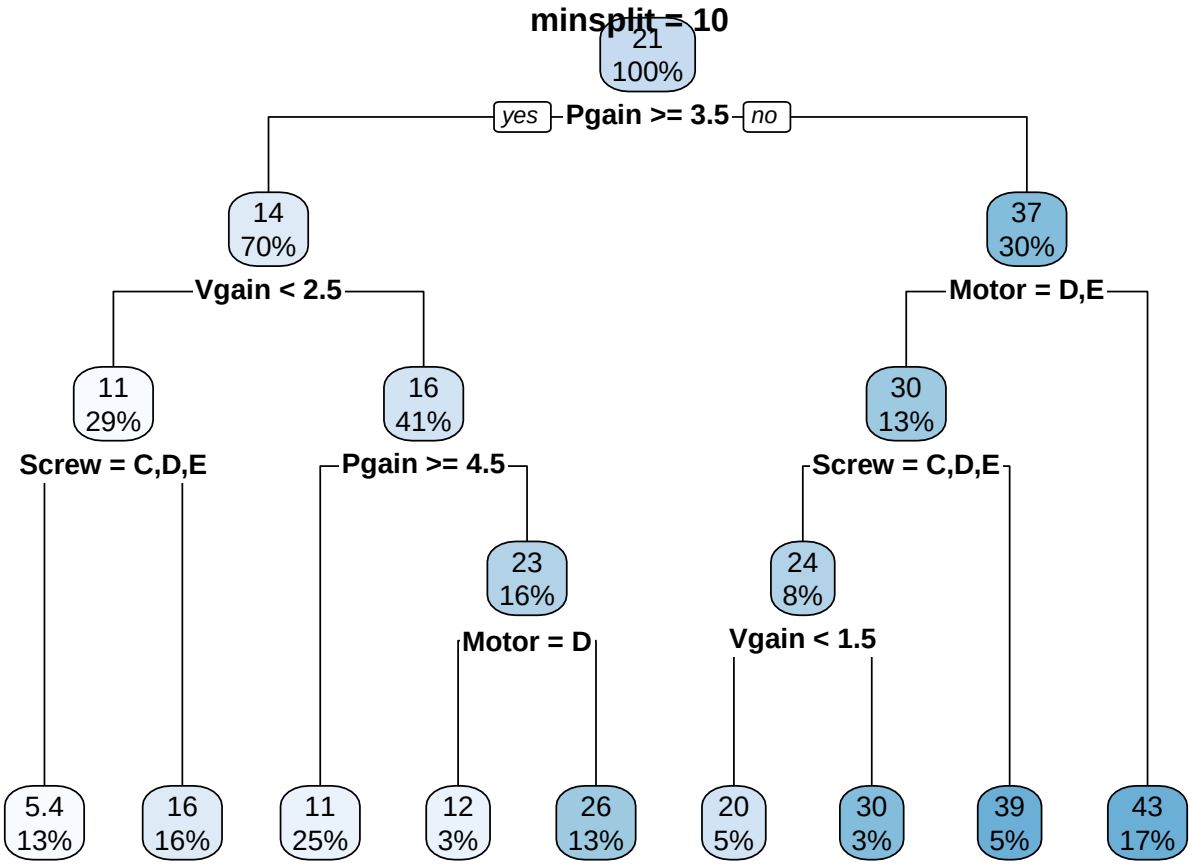
## Understanding the Hyperparameters of CARTS

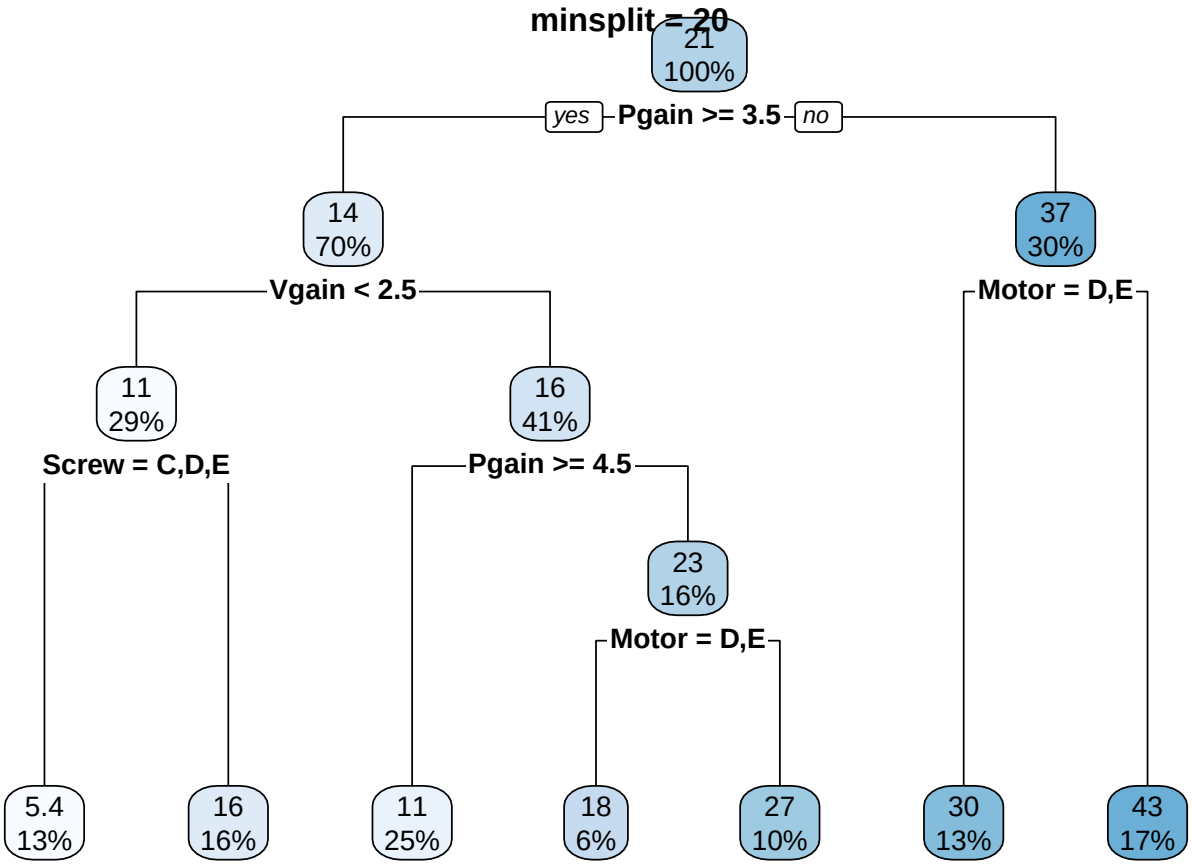
**Min split:**

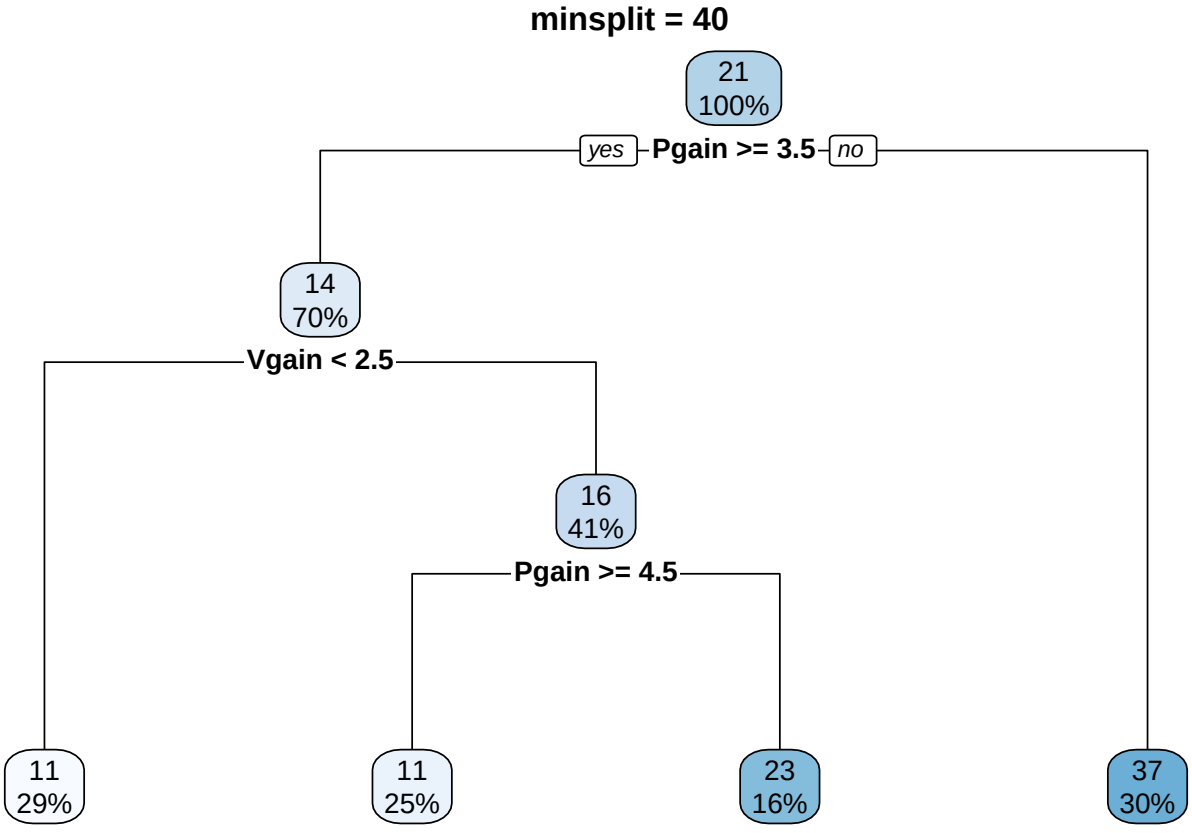
```
task <- makeRegrTask(data = train_1, target = "Class")
lrn1 <- makeLearner("regr.rpart")

plotParamSequence <- function(learner, task, param, values, plotfun, ...) {
  for (v in values) {
    xs <- list(...)
    xs[[param]] <- v
    lrn2 <- setHyperPars(learner, par.vals = xs)
    mod <- mlr::train(lrn2, task)
    plotfun(mod$learner.model)
    title(sprintf("%s = %g", param, v))
    pause()
  }
}
```

```
plotParamSequenceRPart <- function(...)
  plotParamSequence(learner = lrn1, plotfun = rpart.plot, ...)

min_splits <- c(3, 10, 20, 40)
plotParamSequenceRPart(task = task, param = "minsplit", values = min_splits)
```



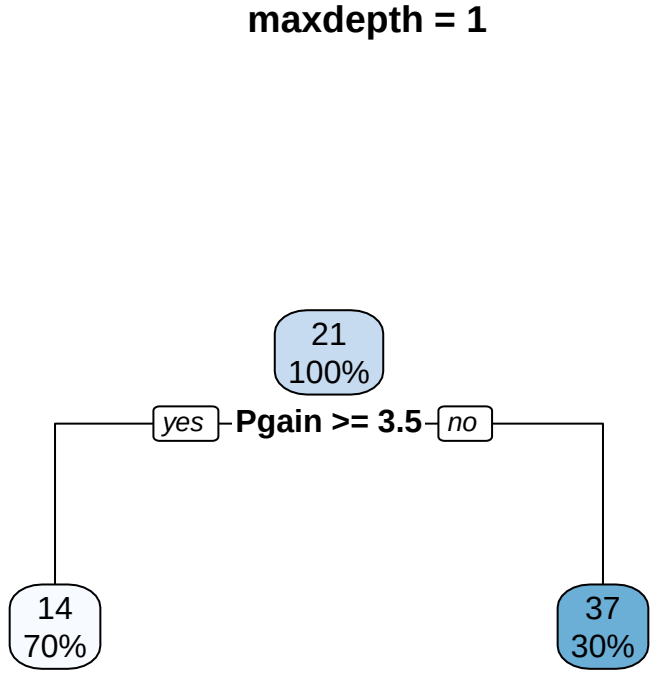

minsplit = 10
21
100%
yes
Pgain >= 3.5
no
14
70%
37
30%
Vgain < 2.5
Motor = D,E
11
29%
16
41%
30
13%
Screw = C,D,E
Pgain >= 4.5
Screw = C,D,E
23
16%
24
8%
Motor = D
Vgain < 1.5
5.4
13%
16
16%
11
25%
12
3%
26
13%
20
5%
30
3%
39
5%
43
17%

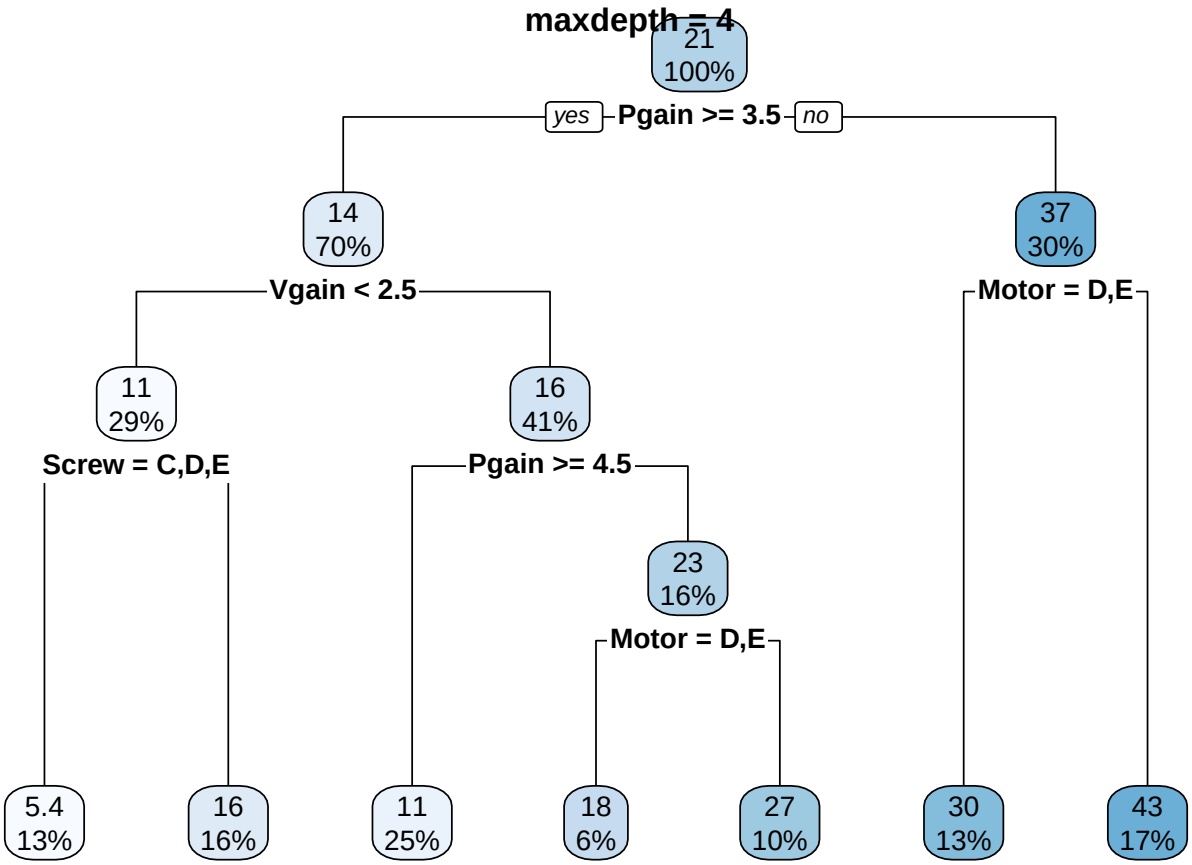

minsplit = 20
21
100%
yes
Pgain >= 3.5
no
14
70%
37
30%
Vgain < 2.5
Motor = D,E
11
29%
16
41%
Screw = C,D,E
Pgain >= 4.5
23
16%
Motor = D,E
5.4
13%
16
16%
11
25%
18
6%
27
10%
30
13%
43
17%

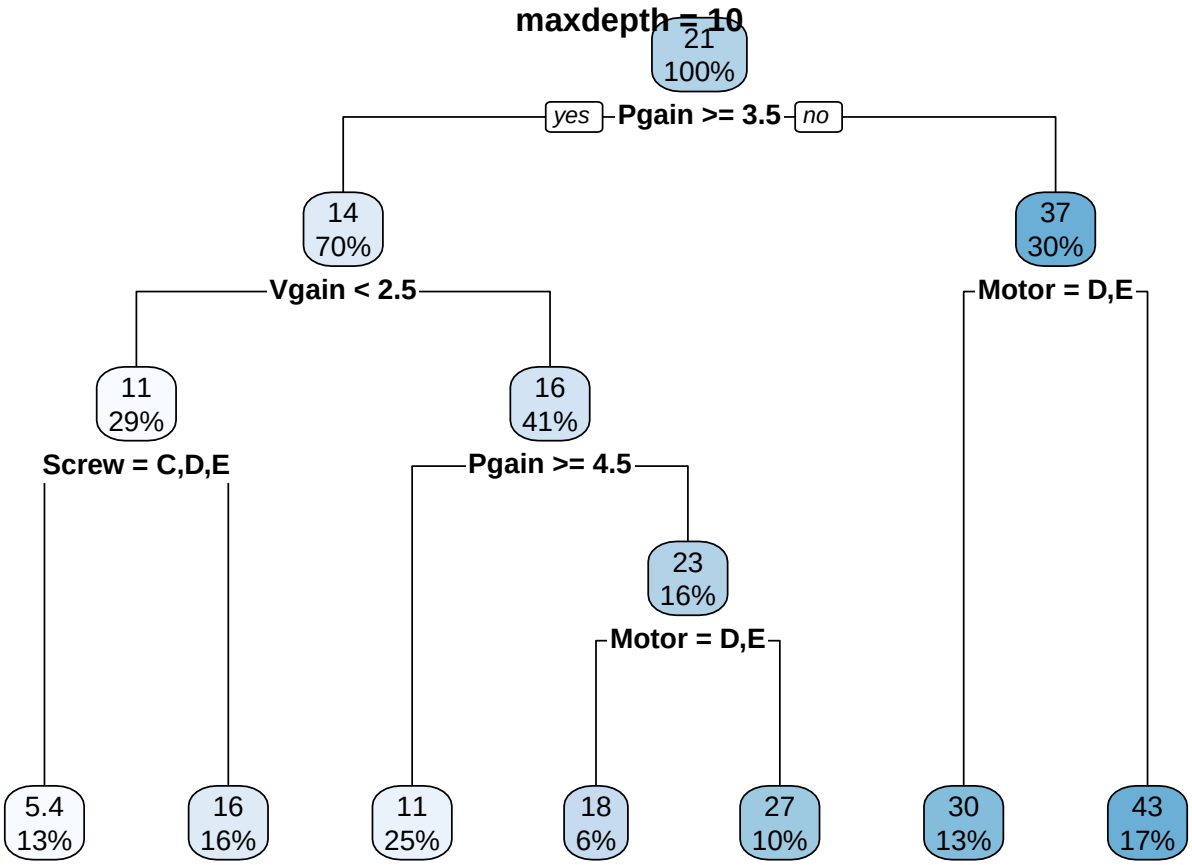

minsplit = 40
21
100%
yes
Pgain >= 3.5
no
14
70%
Vgain < 2.5
16
41%
Pgain >= 4.5
11
29%
11
25%
23
16%
37
30%

**Maximum depth of the tree**

```
task <- makeRegrTask(data = train_1, target = "Class")
lrn1 <- makeLearner("regr.rpart")

plotParamSequence <- function(learner, task, param, values, plotfun, ...) {
  for (v in values) {
    xs <- list(...)
    xs[[param]] <- v
    lrn2 <- setHyperPars(learner, par.vals = xs)
    mod <- mlr::train(lrn2, task)
    plotfun(mod$learner.model)
    title(sprintf("%s = %g", param, v))
    pause()
  }
}

plotParamSequenceRPart <- function(...)
  plotParamSequence(learner = lrn1, plotfun = rpart.plot, ...)

max_depth <- c(1, 4, 10)
plotParamSequenceRPart(task = task, param = "maxdepth", values = max_depth)
```




maxdepth = 4
21
100%
yes
Pgain >= 3.5
no
14
70%
37
30%
Vgain < 2.5
Motor = D,E
11
29%
16
41%
Screw = C,D,E
Pgain >= 4.5
23
16%
Motor = D,E
5.4
13%
16
16%
11
25%
18
6%
27
10%
30
13%
43
17%




## Invariance to monotonous feature transformation

The nice thing about CARTs is that they are invariant to scaling and other monotonous feature transformations.

We grow two trees on two data set, which only differ in the fact that the numeric features in the second data set are log-transformed.

Let's see how the architecture of the two CARTs differs (or not):

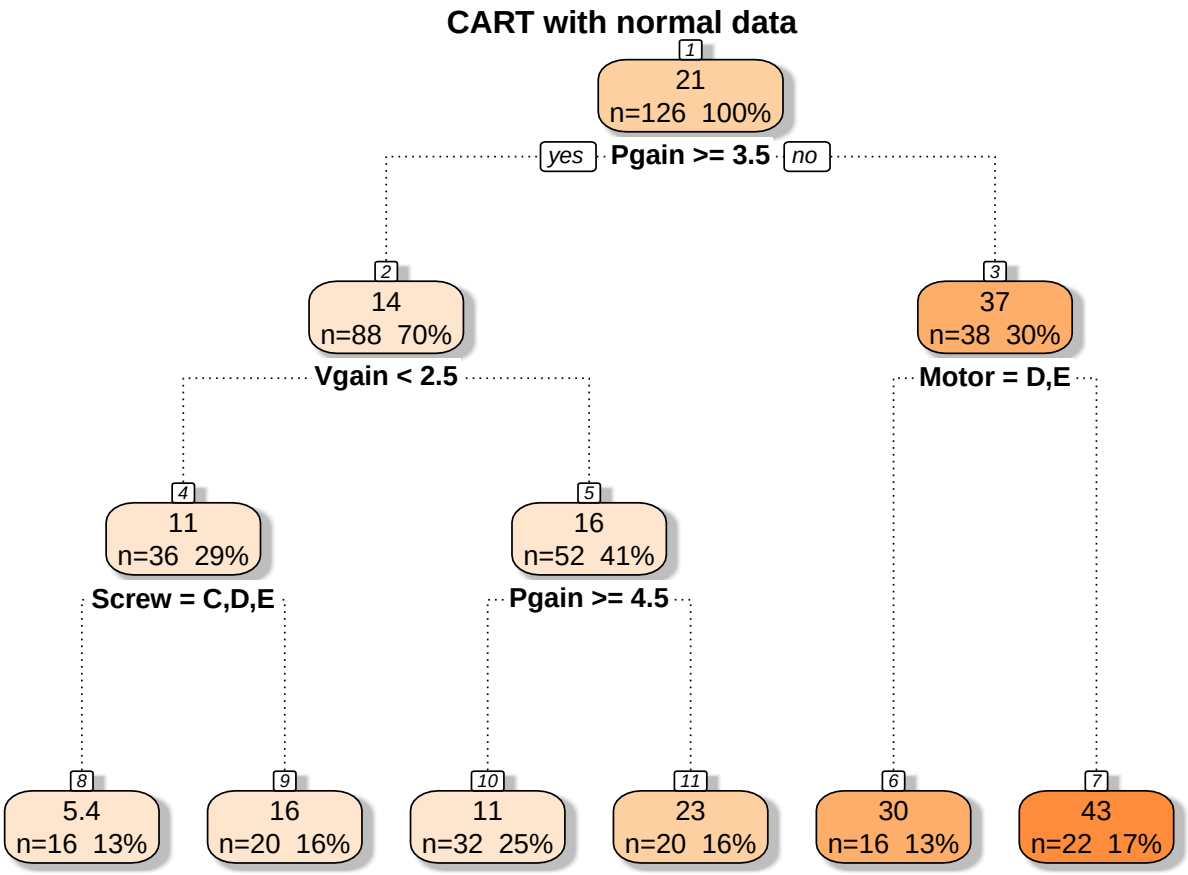
```
train_log <- train_1 %>% mutate_at(c("Pgain", "Vgain"), log)
head(train_log, 3)
```

```
##   Motor Screw Pgain Vgain Class
## 1     A     A  1.39 1.099    30
## 2     B     A  1.10 0.693    48
## 3     E     C  1.39 0.000     4
```

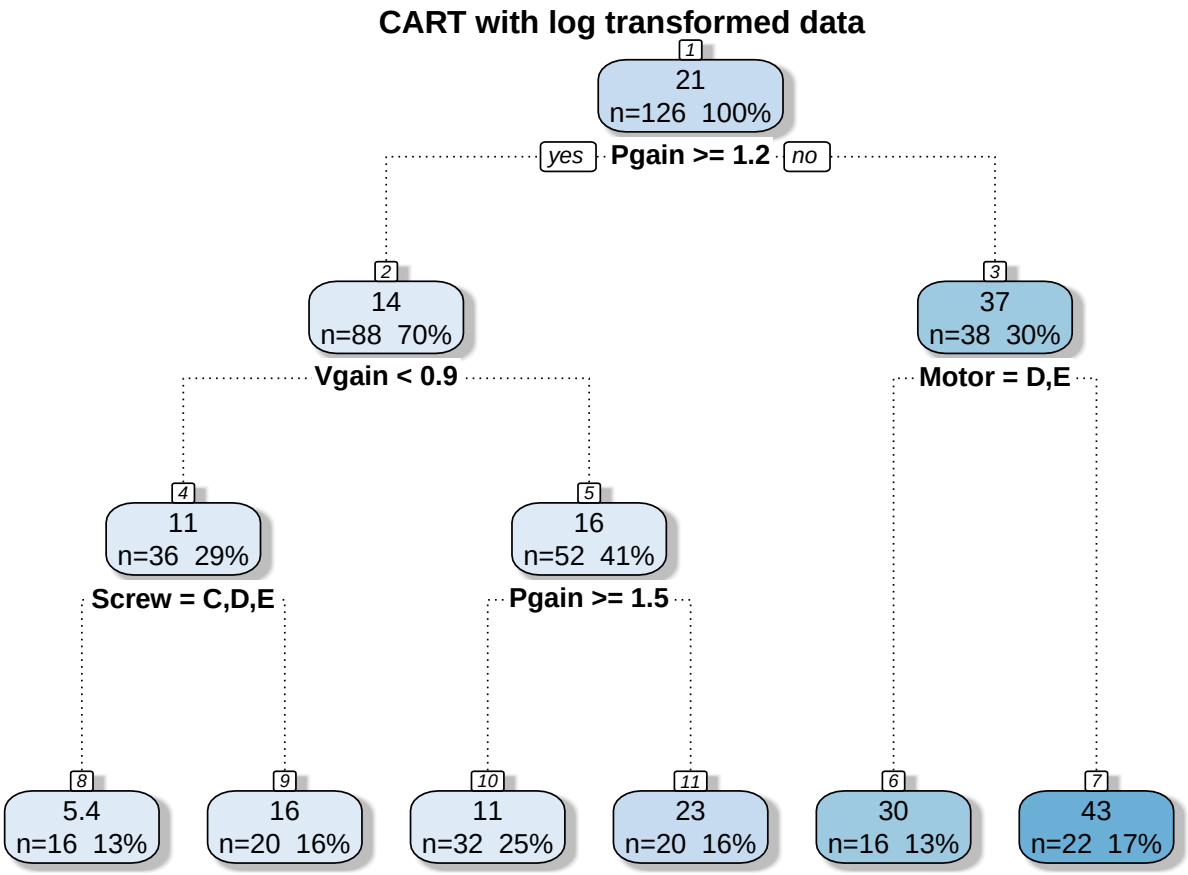
```
head(train_1, 3)
```

```
##    Motor Screw Pgain Vgain Class
## 93     A     A     4     3    30
## 4      B     A     3     2    48
## 77     E     C     4     1     4
```

```
model_1 <- rpart(Class ~ ., data = train_1, minsplit = 30)
rattle::fancyRpartPlot(model_1,
  palettes =
    "Oranges", main = "CART with normal data\n", sub = ""
)
```




```
model_log <- rpart(Class ~ ., data = train_log, minsplit = 30)
rattle::fancyRpartPlot(model_log,
  palettes = "Blues",
  main = "CART with log transformed data\n", sub = ""
)
```




## Write simple regression tree implementation

Now we want to implement a simple regression tree learner (`rt`) for numerical features:

```
# Estimate a *r*egression *t*ree, which consists of nodes.
# Every node is either a splitting node with 2 subnodes
# or a leaf / terminal node without subnodes (end of a tree branch).
# The two subnodes for each split are:
#  - true_node: the split (i.e. feature < treshold) was TRUE
#  - false_node: the split was FALSE
# The tree is built recursively by calling rt on the subset of data in each
# tree branch again.
# The final tree is then represented as a nested list of splitting nodes
# containing splitting nodes containing further  splitting nodes and leaf nodes.
# Every node itself is also simply represented as a list, see the very end of
# this function for the definition.
rt <- function(df, target, cur_depth = 0, max_depth = 30, min_split = 20) {
  feature_results <- list()

  # compute optimal split with minimal loss for every feature:
  for (feature in colnames(df)) {
    if (feature != target) {
      feature_results[[feature]] <- optimal_split(df, feature, target)
    }
  }
```

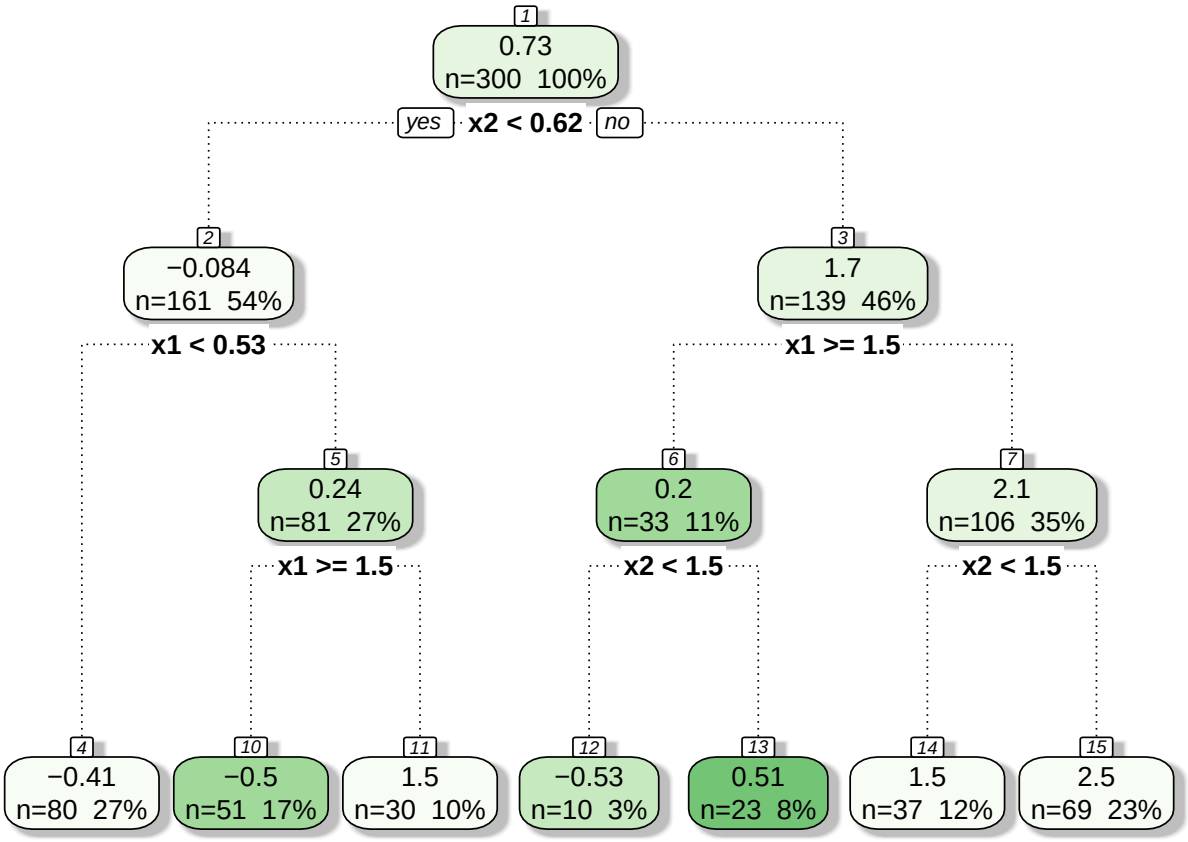
```
df_feature_results <- do.call(rbind, feature_results)
# choose the feature whose optimal split minimizes the loss:
min_loss_split <- df_feature_results[which.min(df_feature_results[, "loss"]), ]

# use this feature's split:
split_result <- split_node(df, min_loss_split)
if (cur_depth + 1 == max_depth) {
  # max_depth of the tree is reached
  true_node <- list(is_leaf = TRUE)
  false_node <- list(is_leaf = TRUE)
} else {
  if (nrow(split_result$true_split) < min_split) {
    # not enough observations to split node again
    true_node <- list(is_leaf = TRUE)
  }
  else {
    # call the rt function recursively again, using only the subset of data
    # for which the current split was TRUE, and with incremented tree depth,
    #  s.t. the function will terminate if the maximal depth is reached
    true_node <- rt(
      df = split_result$true_split,
      target = target,
      cur_depth = cur_depth + 1,
      max_depth = max_depth,
      min_split = min_split
    )
  }
  #.. now do the same things on the other node:
  if (nrow(split_result$false_split) < min_split) {
    # not enough observations to split node again
    false_node <- list(is_leaf = TRUE)
  }
  else {
    # call the rt function recursively again, using only the subset of data
    # for which the current split was FALSE, and with incremented tree depth,
    # s.t. the function will terminate if the maximal depth is reached
    false_node <- rt(
      df = split_result$false_split,
      target = target,
      cur_depth = cur_depth + 1,
      max_depth = max_depth,
      min_split = min_split
    )
  }
}

list(
  is_leaf = FALSE,  #terminal node=
  true_node = true_node,  # the tree branch for which split was TRUE
  false_node = false_node, # the tree branch for which split was FALSE
  loss = min_loss_split$loss, # the achieved loss in this split
  feature = min_loss_split$feature, #the feature used for this split
  threshold = min_loss_split$threshold, #the threshold of the feature used for the split
```

```
    num_true = min_loss_split$num_true, # the predicted value for observations if split is TRUE
    num_false = min_loss_split$num_false # the predicted value for observations in split is FALSE
  )
}

# function which splits current dataframe according to split rule
# (if one wants to implement classification trees and/or support for categorical
# features this would have to be extended -- think about how?)
split_node <- function(df, split_rule) {
  return(list(
    true_split = df[df[, split_rule$feature] < split_rule$threshold, ],
    false_split = df[df[, split_rule$feature] >= split_rule$threshold, ]
  ))
}

# function which finds the optimal split for a given feature
# (if one wants to implement classifiaction and/or support for categorical
# features this would have to be extended -- think about how?)
optimal_split <- function(df, feature, target) {
  cur_losses <- list()
  #sort data according to the feature, only keep feature and target
  df <- df[order(df[, feature]), c(feature, target)]

  split_points <- sort(unique(df[, feature]))

  # compute losses for all split points
  for (i in 2:length(split_points)) {
    is_split_true <- df[, feature] < split_points[i]
    split_true <- df[is_split_true, target]
    split_false <- df[!is_split_true, target]

    cur_losses[[as.character(i)]] <- data.frame(
      loss = l2_loss(split_true) + l2_loss(split_false),
      threshold = mean(split_points[(i - 1):i]),
      num_true = mean(split_true),
      num_false = mean(split_false)
    )
  }

  df_curr_losses <- do.call(rbind, cur_losses)
  # find split which yields the minimal loss
  min_loss <- df_curr_losses[which.min(df_curr_losses[, "loss"]), ]

  data.frame(
    loss = min_loss[1, "loss"],
    threshold = min_loss[1, "threshold"],
    num_true = min_loss[1, "num_true"],
    num_false = min_loss[1, "num_false"],
    feature = feature
  )
}

# l2 loss if target is estimated with l2-optimal constant (mean)
```

```r
l2_loss <- function(target_values) {
  sum((target_values - mean(target_values))^2)
}

# function which predicts data with regression tree
# (if one wants to implement classification and/or support for categorical
# features this would need to be extended)
predict_tree <- function(tree, df) {
  pred_values <- numeric(nrow(df))
  # loop over all observations in df
  for (i in seq_len(nrow(df))) {
    cur_node <- tree
    # follow the splits until we reach a leaf
    while (!cur_node$is_leaf) {
      if (df[i, cur_node$feature] < cur_node$threshold) {
        pred_values[i] <- cur_node$num_true
        cur_node <- cur_node$true_node
      } else {
        pred_values[i] <- cur_node$num_false
        cur_node <- cur_node$false_node
      }
    }
  }
  pred_values
}
```

Compare with CART implementation from `rpart`:

```r
# create artifical data to check algorithm
set.seed(123)
n <- 300
x1 <- runif(n, -2, 3)
x2 <- runif(n, -2, 3)
y <- ifelse(x1 > 1.5, -1, 1) +
     ifelse(x2 > 1.5, .5, -.5) +
     ifelse(x1 < .5 & x2 < .5, -1, 1) +
     rnorm(n, 0, 0.1)

train_data <- data.frame(x1, x2, y)
# compute regression tree
tree <- rt(train_data, "y", max_depth = 4)

# compare with rpart CART implementation:
rpart_tree <- rpart::rpart(y ~ x1 + x2, data = train_data, maxdepth = 4)
rattle::fancyRpartPlot(rpart_tree, sub = "")
```




```
# compare first split:
tree$feature
```

```
## [1] x2
## Levels: x1 x2
```

```
tree$threshold
```

```
## [1] 0.625
```

```
# if the subnodes of the first node were leaves, this would be the
# output prediction for the splits:
tree$num_true #if feature < threshold
```

```
## [1] -0.0842
```

```
tree$num_false #if feature > threshold
```

```
## [1] 1.68
```

```
# --> first split identical

# first splitting node on "FALSE" branch:
tree$false_node$feature
```

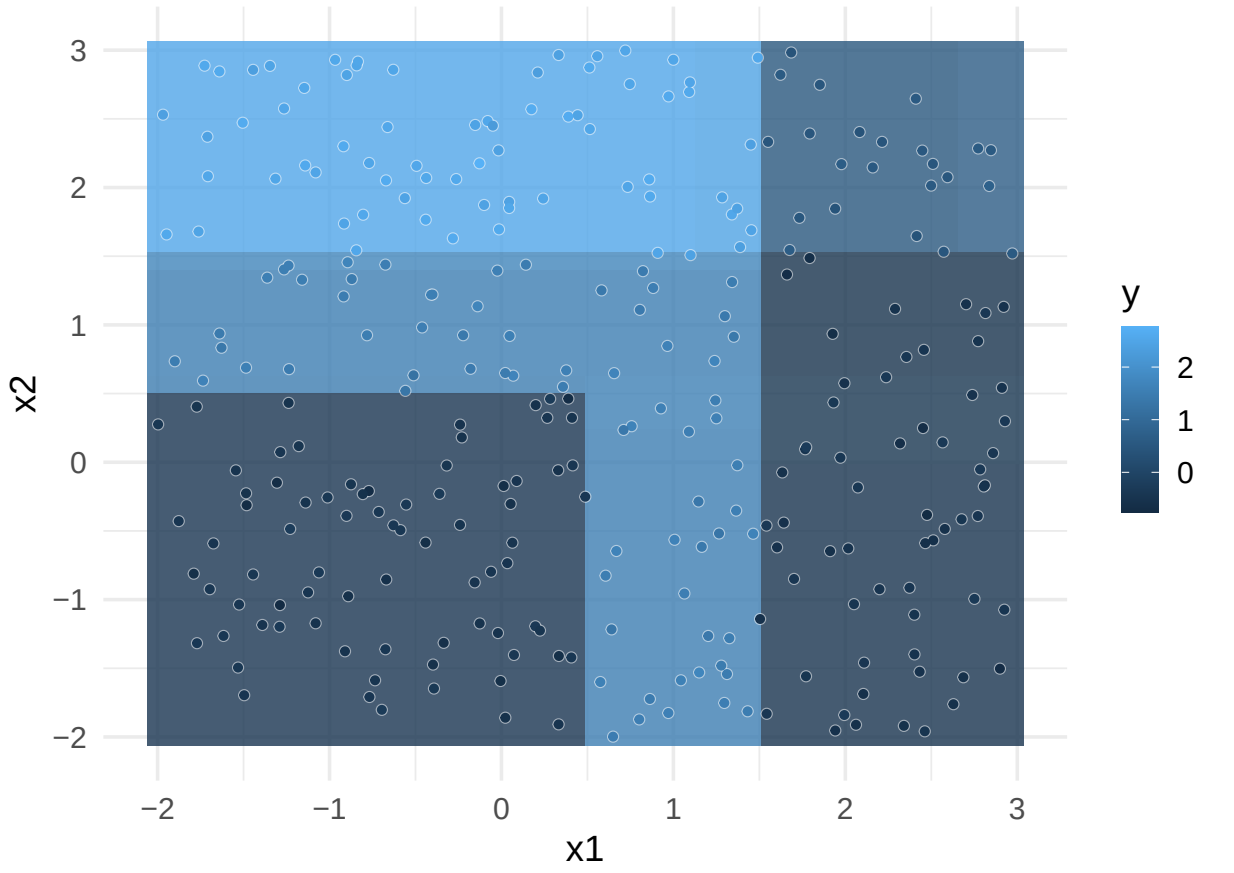
```
## [1] x1
## Levels: x1 x2
```

```
tree$false_node$threshold
```

```
## [1] 1.52
```

```
# -->  same as for rpart as well
```

```
# function for cart method with two features to visualize
# the estimated output X1 and X2 are the names of the two features to use.
plot_2D_cart <- function(train_data,
                         tree,
                         target = "y",
                         X1 = "x1",
                         X2 = "x2",
                         lengthX1 = 40,
                         lengthX2 = 40) {
  gridX1 <- seq(
    min(train_data[, X1]),
    max(train_data[, X1]),
    length.out = lengthX1
  )
  gridX2 <- seq(
    min(train_data[, X2]),
    max(train_data[, X2]),
    length.out = lengthX2
  )

  # compute grid coordinates with cartesian product
  grid_data <- expand.grid(gridX1, gridX2)
  colnames(grid_data) <- c(X1, X2)

  # predict grid cells
  grid_data <- cbind(grid_data, target = predict_tree(tree, grid_data))
  colnames(grid_data) <- c(X1, X2, target)

  min_z <- min(c(min(grid_data[, target]), min(train_data[, target])))
  max_z <- max(c(max(grid_data[, target]), max(train_data[, target])))

  ggplot() +
    geom_raster(
      data = grid_data,
      aes_string(x = X1, y = X2, fill = target),
      alpha = .8
    ) +
    geom_point(data = train_data, aes_string(x = X1, y = X2), color = "white", size = 2, alpha=.5) +
    geom_point(data = train_data, aes_string(x = X1, y = X2, color = target)) +
    scale_colour_gradient(limits = c(min_z, max_z)) +
    scale_fill_gradient(limits = c(min_z, max_z))
}


plot_2D_cart(train_data, tree)
```


3
2
1
0
−1
−2
x2
−2
−1
0
1
2
3
x1
y
2
1
0