CART Coding

Qi Wang

2022/7/18

Here are some coding tricks I learned from the code in this website https://github.com/ebprado/MOTR-BART/blob/master/MOTR%20BART%20paper/MOTR_BART.R .

1."vector" Function

This function is useful when using vector('list', length = 2). This will create a list like vector but we don't need the "$" to subtract the elements in this vector. Instead, we use "[[x]]", this will become more convenient to store some values especially when multiple indexes of an observation is stored. For example:

```
a <- vector('list', 3)
a[[1]] <- c(1,1,2,3,4)
a[[2]] <- matrix(c(4,4,5,6),2,2)
a[[3]] <- array( c(7,7,8,9,0,1), c(2,2,2))
```

This is actually like a vector of all kinds of things including matrices or arrays. When I want to get the first part of information, I may use:

```
a[[1]]
```

```
## [1] 1 1 2 3 4
```

When I want to get the second part of information, I may use:

```
a[[2]]
```

```
##      [,1] [,2]
## [1,]    4    5
## [2,]    4    6
```

However, when I want to get the information of the matrix which is the second element of the whole vector, I will use this:

```
a[[2]][1,1]
```

```
## [1] 4
```

```
a[[2]][1,]
```

```
## [1] 4 5
```

This is how "vector" function works in this code, which is a very important function.

2.Create a function that stores stump information for each Tree

This step is to keep track of whether this node is a terminal node, it is just a stump model so it only store information for one node. Here, we create a information box for all trees.

```
num_trees <- 5
x1 <- rnorm(1000, mean = 9, sd = 3)
x2 <- rnorm(1000, mean = 30, sd = 10)
y <- rnorm(1000, mean = 2*x1 + 3*x2, sd = 1)
x <- cbind(x1, x2)
```

```
create_stump <- function(x, y, num_trees = 1){

all_tree <- vector("list", length = num_trees)

# Make each tree information stored in all_tree[[i]]
# This is like creating a box, but it's empty for further storage

for (i in 1:num_trees) {

  # Make each element has two indexes
  all_tree[[i]] <- vector("list",2)
  names(all_tree[[i]]) <- c("info", "node")

  # Create tree information storage
  all_tree[[i]][[1]] <- matrix(NA, ncol = 8, nrow = 1)

  # Create node indices storage
  all_tree[[i]][[2]] <- rep(1, length(y))

  # Create column names for information matrix
  colnames(all_tree[[i]][[1]]) <- c("terminal","child_L","child_R","parent","split_variable","split_valu

  all_tree[[i]][[1]][1,] <- c(1, rep(NA, 5), 0, length(y))



}

return(all_tree)

}
```

3. Create a function to update trees in the M-H step.

3.1 "Switch" Function

This function is like choosing different directions of the result. For example,

```
switch (1,"A","B","C")
```

```
## [1] "A"
```

```
switch (2,"A","B","C")
```

```
## [1] "B"
```

```
switch (3,"A","B","C")
```

```
## [1] "C"
```

However, in this code, the first argument can also be a string, for example,

```
switch ("A", "A" = 4, "B" = 5, "C" = "Hello World")
```

```
## [1] 4
```

```
switch ("B", "A" = 4, "B" = 5, "C" = "Hello World")
```

```
## [1] 5
```

```
switch ("C", "A" = 4, "B" = 5, "C" = "Hello World")
```

```
## [1] "Hello World"
```

3.2 Updating Tree

Therefore, we can simply use this function to create a tree update function, which depends on the method of the decision. If the method of updating is "grow", then we will grow the tree, and so on. So the update function should have the shape like this:

```
update_tree <- function(x, y, type = c("grow", # For growing the existing tree
                                       "prune", # For merging one pair of the terminal node
                                       "change", # Change the splitting variable and splitting rule
                                       "swap", # Swap the splitting rules for two pairs of terminal nod
                                       ), curr_tree # The current sets of trees
                                        , node_min_size # The minimum size of a node to grow, avoid the

                        ){
```

```
  new_tree <- switch (type, grow = grow_tree(x, y, curr_tree, node_min_size),
                            prune = prune_tree(x, y, curr_tree),
                            change = change_tree(x, y, curr_tree, node_min_size),
                            swap = swap_tree(x, y, curr_tree, node_min_size))
  return(new_tree)
                                        }
```

Then we have to figure our how to grow, prune, change or swap a tree.

4. Grow Tree Function

4.1 Quick Notes(R trick)

1. We can quickly get the elements corresponding to some categories by using the "==", just a[xx == yy] if a is a vector. No need to use a[which[xx == yy]].

```
c(1,2,3,4,5,6)[c(T,F,T,F,T,F)]
```

```
## [1] 1 3 5
```

2. We can quickly pick out one column with the column name if the matrix has a column name.

```
A <- matrix(c(1,2,3,4), 2,2, byrow = TRUE)
colnames(A) <- c("A", "B")
A
```

```
##      A B
## [1,] 1 2
## [2,] 3 4
```

```
A[,"A"]
```

```
## [1] 1 3
```

```
A[,"B"]
```

```
## [1] 2 4
```

3. We can use "any" function to judge whether there exists something that satisfies our expectation, for example,

```
if(any(c(1,2,3)>1)){print("Exist one value bigger than 1")}
```

```
## [1] "Exist one value bigger than 1"
```

```r
if(any(c(1,2,3)<=1)){print("Exist one value smaller than or equal to 1")}
```

```
## [1] "Exist one value smaller than or equal to 1"
```

4.2 Grow Tree Function

We first get the available terminal nodes list which can grow, then get the available rules that we can use, then select the spliting variables and rules uniformly.

```r
grow_tree <- function(x, y, curr_tree, node_min_size){

  # Give new tree information box
  new_tree <- curr_tree

  # Get the terminal nodes list
  terminal_node <- which(as.numeric(new_tree$info[,"terminal"]) == 1)

  # Get the terminal node size
  terminal_node_size <- new_tree$info[terminal_node,"node_size"]

  # Initialize
  available_values <- NULL
  max_bad_trees <- 10
  count_bad_trees <- 0
  bad_trees = TRUE

  # If it's a bad_tree, for example, the size of one new terminal node is too small, then we need to re

  while (bad_trees) {

    # Information box for new tree
    new_tree <- curr_tree

    # Add two extra rows to the tree information, because we are now having more nodes of the tree.
    new_tree$info <- rbind(new_tree$info, c(1, rep(NA, 7)), c(1, rep(NA, 7)))

    # Choose a random terminal to split, if it has been smaller than the minimum size of our requiremen
    split_node <- sample(terminal_node, 1, prob = as.integer(as.numeric(terminal_node_size))> node_min_

    # Choose a variable to split
    split_var <- sample(1:ncol(x),1)

    # The next step guaranteed that we are having a nice splitting value.
    # Available values are the range of the selected variables belonging to this node. Noting that node
    available_values <- sort(unique(x[new_tree$node == split_node, split_var]))

    # Select an available value to split

    if (length(available_values)==1){
      split_val <- available_values[1]
    }else if(length(available_values)==2){
        split_val <- available_values[2]
```

```
    }else{
      # We cannot select the biggest or smallest value as the splitting rule
        split_val <- gdata::resample(available_values[-c(1, length(available_values))],1)
    }
       # Make sure the current parent exist there. If it's the root node, then it's NA.
      curr_parent <- new_tree$info[split_node, "parent"]
      new_tree$info[split_node, 1:6] <- c(0, # Because now it's not terminal node
                                          nrow(new_tree$info)-1, # The second last row is the left chil
                                          nrow(new_tree$info), # The last row is the right child
                                          curr_parent, # Record the current parent
                                          split_var, split_val)


      # Fill the parent of these two child nodes
      new_tree$info[nrow(new_tree$info)-1, "parent"] <- split_node
      new_tree$info[nrow(new_tree$info), "parent"] <- split_node

      #Fill the details including updating the node indices
      new_tree <- fill_tree_details(new_tree, x)

      # Check for bad trees, if it's a bad tree, then we cannot use this.
      if(any(new_tree$info[,"node_size"]<= node_min_size)){

        # Count how many bad trees that we have generated
        count_bad_trees = count_bad_trees + 1
      }else{
        bad_trees = FALSE
      }

      # If too many bad trees are generated, we return the current tree, which means now the trees has
      if(count_bad_trees == max_bad_trees){return(curr_tree)}


}
        return(new_tree)
}
```

5. Fill Detailed Tree Function

Even if we have updated the tree information, we haven't updated the nodes that the observation belongs to, i.e., new_tree$node. In this section, we create a function to fill the tree details. Also, we update the node size.

```
fill_tree_details <- function(curr_tree, x){

  # Collect old information from the tree
  tree_info <- curr_tree$info

  # Create a new matrix to overwrite the information
  new_tree_info <- tree_info

  # Start with default node
```

```
  node <- rep(1, nrow(x))

  # Get the number of observations falling in each node.
  # But we don't need the first node, because it's the root so all the observations will fall into the

    for (i in 2:nrow(tree_info)) {

    #Get the parent
    curr_parent <- as.numeric(tree_info[i,"parent"])

    #Get the splitting variable and splitting value
    split_var <- as.numeric(tree_info[curr_parent, "split_variable"])
    split_val <- as.numeric(tree_info[curr_parent, "split_value"])
    direction <- ifelse(tree_info[curr_parent, "child_L"] == i, "L", "R")
    if(direction == "L"){
      # if the direction is the left, then it should use "smaller than" to confirm that this observatio
      # x[node == curr_parent,] selects the observations who belongs to the parent node.
      # Be careful that the "node" updated every time in the loop!
      # The node now haven't been updated, so it describes the category that this observation belongs t
      #So it's reasonable to say node == curr_parent because we need to filter among all the nodes belo
      new_tree_info[i, "node_size"] <- sum(x[node == curr_parent, split_var] < split_val)
      # This step we update the node indices for the i th row. So we now only update those who belongs
      node[node == curr_parent][x[node == curr_parent, split_var]<split_val] <- i
    }else{

      #Same as left, but change the inequality direction
      new_tree_info[i, "node_size"] <- sum(x[node == curr_parent, split_var] >= split_val)
      node[node == curr_parent][x[node == curr_parent, split_var] >= split_val] <- i

    }

  }

  return(list(info = new_tree_info, node = node))


}
```

Testing for Code from part 1-5

```
tree <- create_stump(x,y)[[1]]


tree_up_1 <- update_tree(x,y,type = "grow", curr_tree = tree, node_min_size = 10)

tree_up_2 <- update_tree(x,y,type =  "grow", curr_tree = tree_up_1, node_min_size = 10)

tree_up_3 <- update_tree(x,y, type = "grow", curr_tree = tree_up_2, node_min_size = 10)

print(tree)
```

```
## $info
```

```
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        1      NA      NA     NA              NA          NA  0      1000
##
## $node
##    [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##   [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##   [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [112] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [149] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [186] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [223] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [260] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [297] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [334] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [371] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [408] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [445] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [482] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [519] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [556] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [593] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [630] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [667] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [704] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [741] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [778] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [815] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [852] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [889] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [926] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [963] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [1000] 1
```

```
print(tree_up_1)
```

```
## $info
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        0       2       3     NA              2    40.93968  0      1000
## [2,]        1      NA      NA      1             NA          NA NA       888
## [3,]        1      NA      NA      1             NA          NA NA       112
##
## $node
##    [1] 2 3 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 3 2 2 2 2 2 2 2 3 2 2 2 2 2
##   [38] 3 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 2 2 2 2 2 2 2 2
##   [75] 2 3 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [112] 2 2 2 2 2 2 2 2 2 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 2 2
##  [149] 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2
##  [186] 2 3 2 2 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2
##  [223] 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2
##  [260] 2 3 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [297] 2 3 2 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2
##  [334] 2 2 2 2 2 2 2 2 3 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 3 2 2 2 3 2 3 2 2 2
##  [371] 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2
##  [408] 2 2 3 2 2 2 2 2 2 2 2 2 2 2 3 3 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2
```

```
##  [445] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3
##  [482] 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 2 2 2 3 2 2 3 2 2 2 2 2 2 3 3 2 2 2 2 2 2 3
##  [519] 3 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2 2 2
##  [556] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2 2 2 3 2 2 2 2
##  [593] 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2 2 3 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [630] 2 2 2 2 2 2 2 3 2 2 2 2 3 2 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2
##  [667] 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 3 2 2 2 2 2 2 3 3 2 2 2 2 2 2 2 3 2 2
##  [704] 2 2 3 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3
##  [741] 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2 2
##  [778] 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 3 2 2 3 3 2 2 2 2
##  [815] 2 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2
##  [852] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 3 2 2 2
##  [889] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2
##  [926] 2 2 2 3 3 2 2 2 2 2 2 3 2 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [963] 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 3 2 2 3 2 2 2 2 2 2
## [1000] 2
```

```
print(tree_up_2)
```

```
## $info
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        0       2       3     NA              2   40.939684  0      1000
## [2,]        1      NA      NA      1             NA          NA NA       888
## [3,]        0       4       5      1              1    7.136951 NA       112
## [4,]        1      NA      NA      3             NA          NA NA        25
## [5,]        1      NA      NA      3             NA          NA NA        87
##
## $node
##    [1] 2 4 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 5 2 2 2 5 2 2 2 2 2 2 2 5 2 2 2 2 2
##   [38] 4 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 5 5 2 2 2 2 2 2 2 2 2 2 2 2
##   [75] 2 5 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [112] 2 2 2 2 2 2 2 2 2 2 5 2 2 5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 5 2 2 2 2 2 2
##  [149] 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2
##  [186] 2 5 2 2 2 5 5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2
##  [223] 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2
##  [260] 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [297] 2 5 2 5 5 5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2
##  [334] 2 2 2 2 2 2 2 2 5 2 2 2 2 5 2 2 2 2 2 2 2 2 2 5 2 2 5 2 2 2 4 2 5 2 2 2 2
##  [371] 2 2 2 2 2 2 2 2 2 2 2 2 5 4 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 2 2
##  [408] 2 2 4 2 2 2 2 2 2 2 2 2 2 2 5 5 2 5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 5 2
##  [445] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 4
##  [482] 2 2 2 2 5 2 5 2 2 2 2 2 2 2 2 2 2 2 4 2 2 2 4 2 2 2 2 2 2 2 5 5 2 2 2 2 2 2 4
##  [519] 5 2 2 5 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 4 2 2 2 2 2 4 2 2 2 2 2 2 2
##  [556] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 5 2 2 2 2
##  [593] 2 2 2 2 2 2 2 2 2 5 2 5 2 2 2 2 2 2 5 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2
##  [630] 2 2 2 2 2 2 2 5 2 2 2 2 2 5 2 4 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2
##  [667] 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 4 2 2 2 5 2 2 2 2 2 2 5 5 2 2 2 2 2 2 2 5 2 2
##  [704] 2 2 5 2 2 2 2 2 2 2 4 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 4
##  [741] 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 5 2 2 2 2 2 5 2 2 2 2 2 2
##  [778] 2 2 2 2 2 5 2 5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 5 2 2 5 2 2 2 5 5 2 2 2 2
##  [815] 2 2 2 2 2 2 2 2 2 5 2 4 2 2 2 2 2 2 2 2 5 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [852] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 4 2 2 5 2 2 2
##  [889] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 5 2
##  [926] 2 2 2 5 5 2 2 2 2 2 2 2 5 2 5 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

```
##  [963] 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 2 2 2 2 2 2 2 2 2 5 2 2 2 5 2 2 5 2 2 2 2 2 2
## [1000] 2
```

```
print(tree_up_3)
```

```
## $info
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        0       2       3     NA              2   40.939684  0      1000
## [2,]        0       6       7      1              2   29.329720 NA       888
## [3,]        0       4       5      1              1    7.136951 NA       112
## [4,]        1      NA      NA      3             NA          NA NA        25
## [5,]        1      NA      NA      3             NA          NA NA        87
## [6,]        1      NA      NA      2             NA          NA NA       465
## [7,]        1      NA      NA      2             NA          NA NA       423
##
## $node
##    [1] 7 4 6 6 6 7 7 6 6 5 7 6 7 6 6 6 7 6 6 5 7 7 6 5 7 6 6 6 6 6 7 5 7 6 7 7 7
##   [38] 4 6 7 7 7 6 6 7 4 6 7 6 6 7 7 6 6 7 6 6 7 7 6 5 5 6 6 6 6 6 6 7 6 6 7 6 6
##   [75] 7 5 7 6 7 7 7 7 6 6 5 7 7 6 6 6 6 6 6 6 7 6 6 7 7 7 6 6 6 7 7 7 6 6 7 6 6
##  [112] 7 7 7 7 6 6 7 7 6 5 6 7 5 7 6 7 7 6 7 7 7 6 6 6 6 6 7 6 6 6 7 5 5 6 6 7 6 6
##  [149] 6 7 6 7 7 6 6 5 6 7 7 7 7 7 7 7 6 5 7 6 6 7 6 6 7 7 7 5 7 6 7 6 6 7 7 7 6
##  [186] 6 5 6 7 5 5 6 7 6 6 7 7 7 6 6 6 6 7 6 6 6 6 7 6 6 6 6 6 6 7 5 6 7 6 7 7 6 6 6
##  [223] 7 6 7 6 5 7 6 6 6 7 7 7 6 4 6 6 7 6 6 6 6 7 7 6 6 7 7 6 7 6 6 6 5 7 6 6 6
##  [260] 6 4 6 7 7 7 7 7 6 7 6 7 7 7 4 6 6 6 7 6 7 6 6 7 6 7 7 6 6 6 6 6 6 7 6 7 6
##  [297] 6 5 7 5 5 5 7 7 7 6 7 6 7 7 6 7 6 7 7 6 7 7 6 7 7 6 7 5 6 6 7 7 7 6 6 6 7
##  [334] 7 6 7 6 6 6 7 6 5 6 6 6 6 5 7 7 6 7 7 7 7 7 6 6 5 7 7 5 6 7 6 4 6 5 6 7 6 7
##  [371] 6 6 6 7 7 7 7 7 7 6 6 5 4 7 6 7 7 6 6 6 6 6 7 7 7 5 6 6 6 7 7 7 6 6 7 7 7
##  [408] 6 7 4 7 7 7 6 7 7 6 7 7 6 5 5 7 5 7 6 6 7 6 7 7 7 6 6 7 6 6 6 7 7 7 5 5 6
##  [445] 6 7 7 6 6 7 7 6 7 6 6 6 6 6 7 7 6 6 6 6 6 6 6 6 7 7 6 7 7 7 7 7 6 6 7 6 6 7 6 4
##  [482] 6 7 7 7 5 6 5 7 7 6 7 6 7 6 6 7 6 7 4 7 6 4 7 7 6 6 6 7 5 5 6 6 7 7 6 7 4
##  [519] 5 7 6 5 7 6 6 7 7 7 6 6 7 5 7 7 6 6 7 6 6 7 7 4 6 7 7 6 7 4 7 7 6 6 6 7 6
##  [556] 7 6 6 6 6 6 7 6 6 7 7 6 7 6 6 6 7 5 7 7 6 6 7 5 7 7 6 6 7 6 7 7 5 6 7 7 7
##  [593] 6 6 6 6 6 7 6 6 5 7 5 6 7 6 7 6 5 7 7 6 7 7 7 4 7 6 6 6 7 7 6 7 7 6 6 7 7
##  [630] 7 7 7 6 7 6 5 7 6 6 7 5 6 4 6 7 4 7 7 7 6 7 7 6 6 6 7 7 7 7 5 7 7 7 6 6 6
##  [667] 6 6 7 6 6 6 6 5 7 7 7 6 7 7 7 7 4 7 7 5 7 7 6 7 7 5 5 6 7 7 6 6 7 6 5 7 7
##  [704] 7 7 5 6 7 6 7 6 7 4 6 7 6 6 7 7 7 6 7 5 6 6 6 6 6 6 7 6 6 6 6 6 7 7 6 7 6 4
##  [741] 7 6 4 7 6 7 6 7 6 6 7 6 6 7 6 7 6 6 6 7 7 5 7 6 7 5 6 6 6 6 5 6 7 6 6 6 6
##  [778] 7 6 7 6 5 6 5 6 7 7 6 7 7 6 6 7 7 6 7 7 5 7 6 6 7 5 7 6 5 7 7 5 5 6 6 6 7
##  [815] 6 6 6 6 7 7 6 6 5 7 4 7 6 6 6 7 6 6 5 6 6 6 6 5 7 6 6 6 6 6 7 6 7 7 6 7 7
##  [852] 6 6 6 7 7 7 7 7 6 7 7 6 7 7 6 5 6 6 6 6 7 7 7 6 7 7 6 6 6 5 4 7 6 5 7 7 6
##  [889] 7 6 6 6 7 7 7 7 7 6 7 6 7 6 7 7 7 7 7 7 7 7 6 6 6 6 7 7 6 5 6 6 7 6 7 5 7
##  [926] 7 6 7 5 5 7 7 6 6 6 7 5 6 5 6 6 4 6 6 6 6 7 6 7 6 6 7 7 6 7 6 6 6 7 6 6 6
##  [963] 7 7 7 6 6 6 6 7 6 6 7 5 6 7 7 6 7 6 6 7 6 6 6 5 6 6 7 5 6 7 5 7 7 7 6 6 7
## [1000] 6
```

6. Prune Tree Function

6.1 Coding for Prune Tree Function

In this section, we are going to write some function that prunes a tree, i.e., merge two terminal nodes who has the same parent.

```r
prune_tree <- function(x, y, curr_tree){

  # To begin with, we create a holder for the new tree, where we can store the information about the ne
  new_tree <- curr_tree

  # If the tree has been the stump, we cannot prune it anymore.
  if(nrow(new_tree$info) == 1){ return(new_tree)}

  # Then, we will get the list of the terminal nodes.
  terminal_nodes <- which(as.numeric(new_tree$info[,"terminal"])==1)

  # Then we randomly pick up a terminal to prune, but it's important that both of the terminal points h
  bad_node <- TRUE
  while (bad_node) {

    # Randomly pick up a point to prune
    selected_node <- sample(terminal_nodes, 1)

    # Then, find the parent of this node
    selected_parent <- as.numeric(new_tree$info[selected_node, "parent"])

    # Then, get the children of this parent
    children_left <- as.numeric(new_tree$info[selected_parent,"child_L"])
    children_right <- as.numeric(new_tree$info[selected_parent, "child_R"])

    # Check whether they are both terminal nodes
    children_left_ter <- as.numeric(new_tree$info[children_left,"terminal"])
    children_right_ter <- as.numeric(new_tree$info[children_right, "terminal"])

    # If both of the nodes are terminal nodes, then it's okay
    if( children_right_ter + children_left_ter == 2 ){ bad_node <- FALSE }


  }

  # After selecting the two terminal nodes, we need to delete the two rows of these nodes.
  new_tree$info <- new_tree$info[-c(children_left, children_right),]

  # Update the information for the parent node since now it's a terminal node, so it does not have the
  new_tree$info[selected_parent,c("terminal", "child_L", "child_R", "split_variable", "split_value")] <-

  # If the tree comes back to a stump, there is no need to fill the tree details.
  if(nrow(new_tree$info)==1){new_tree$node <- rep(1, nrow(x))}else{

    # Since we have deleted several rows, there are some row indices changing, if we didn't delete the
    if(selected_node <= nrow(new_tree$info)){

      # Find those whose parents are affected by deleting rows
      bad_parents <- which(as.numeric(new_tree$info[,"parent"])>= selected_node)

      # Since the deleting rows must be continuous two rows, -2 is to make sure the parents are correct
      new_tree$info[bad_parents, "parent"] <- as.numeric(new_tree$info[bad_parents, "parent"]) - 2
```

```
      for (i in selected_node:nrow( new_tree$info )) {

        # Update the child index who has been affected.
        # First, find the parent of the ith row, because we have updated parents before, this row now h
        curr_parent <- as.numeric(new_tree$info[i, "parent"])

        # Then, find the correct children index of the parent row to update the wrong children row inde
        curr_children <-  which(as.numeric(new_tree$info[,"parent"])==curr_parent)

        # Then, update the row indexes.
        new_tree$info[curr_parent, c("child_L", "child_R")] <- sort(curr_children)

      } # End loop for updating children nodes.

    } # End loop for updating tree information

      # Fill tree details
      new_tree <- fill_tree_details(new_tree, x)
  }

      return(new_tree)

}
```

6.2 Testing for Prune Tree Function

```
prune_tree(x, y, tree_up_2)
```

```
## $info
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        0       2       3     NA              2    40.93968  0      1000
## [2,]        1      NA      NA      1             NA          NA NA       888
## [3,]        1      NA      NA      1             NA          NA NA       112
##
## $node
##    [1] 2 3 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 3 2 2 2 2 2 2 3 2 2 2 2 2
##   [38] 3 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2
##   [75] 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [112] 2 2 2 2 2 2 2 2 2 2 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 2 2
##  [149] 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2
##  [186] 2 3 2 2 2 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2
##  [223] 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2
##  [260] 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [297] 2 3 2 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2
##  [334] 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 3 2 2 2 3 2 2 2 3 2 3
##  [371] 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2
##  [408] 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3
##  [445] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3
##  [482] 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 3 2 2 2 2 2 2 2 3 3 2 2 2
##  [519] 3 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2
##  [556] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 3 2 2 2 2 2 2 2 3 2 2
```

```
##  [593] 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2 2 3 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2
##  [630] 2 2 2 2 2 2 3 2 2 2 2 3 2 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2
##  [667] 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 3 2 2 3 2 2 2 2 2 3 3 2 2 2 2 2 2 2 3 2 2
##  [704] 2 2 3 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3
##  [741] 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 3 2 2 2 2 3 2 2 2 2 2 2
##  [778] 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 3 2 2 3 2 2 3 3 2 2 2 2
##  [815] 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 3 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [852] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 3 2 2 2
##  [889] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2
##  [926] 2 2 2 3 3 2 2 2 2 2 2 3 2 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [963] 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 3 2 2 3 2 2 2 2 2 2
## [1000] 2
```

7. Changing Tree

7.1 Preparation

7.1.1 R Coding Tricks - Recursive

In this function, we need to get the children of a parent, and then get the children of the children, which sounds very complicated. However, there is a kind of function in coding, that allows functions to call themselves again and agian, called "recursive" function. For further information, refer to: https://www.datamentor.io/r-programming/recursion/ .

Here are some examples for this kind of function:

```
recursive_fac <- function(a){

  if(a == 0) {return(1)}
  if(a != 0) {return(a * recursive_fac(a-1))}

}

recursive_fac(5)
```

```
## [1] 120
```

```
factorial(5)
```

```
## [1] 120
```

7.1.2 Preparation Function: Get Children

So if we changed one parent splitting variable and splitting rule, all the children and grandchildren and so on will be affected. We should corrected the affected childrens. So we have to use a function to get the children and the list of the childrens. Recursive function is a good way to achieve this.

```
get_children <- function(tree_info, parent){

  all_chilren <- NULL
```

```
  # If the current node is the terminal node, then return the children list and the parent so far.
  if(as.numeric(tree_info[parent, "terminal"]) == 1){return(c(all_chilren, parent))}else{

    # If the current node is not the terminal node, then we have to recursively get the node list.
    curr_child_left <- as.numeric(tree_info[parent, "child_L"])
    curr_child_right <- as.numeric(tree_info[parent, "child_R"])

    # Return the children and the children of the children recursively
    return(c(all_chilren, get_children(tree_info,curr_child_left), get_children(tree_info, curr_child_ri

  }

}
```

```
tree_up_3
```

```
## $info
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        0       2       3     NA              2   40.939684  0      1000
## [2,]        0       6       7      1              2   29.329720 NA       888
## [3,]        0       4       5      1              1    7.136951 NA       112
## [4,]        1      NA      NA      3             NA          NA NA        25
## [5,]        1      NA      NA      3             NA          NA NA        87
## [6,]        1      NA      NA      2             NA          NA NA       465
## [7,]        1      NA      NA      2             NA          NA NA       423
##
## $node
##    [1] 7 4 6 6 6 7 7 6 6 5 7 6 7 6 6 6 7 6 6 5 7 7 6 5 7 6 6 6 6 6 7 5 7 6 7 7 7
##   [38] 4 6 7 7 7 6 6 7 4 6 7 6 6 7 7 6 6 7 6 6 7 7 6 5 5 6 6 6 6 6 6 7 6 6 7 6 6
##   [75] 7 5 7 6 7 7 7 7 6 6 5 7 7 6 6 6 6 6 6 6 7 6 6 7 7 7 6 6 6 7 7 7 6 6 7 6 6
##  [112] 7 7 7 7 6 6 7 7 6 5 6 7 5 7 6 7 7 6 7 7 7 6 6 6 6 6 7 6 6 7 5 5 6 6 7 6 6
##  [149] 6 7 6 7 7 6 6 5 6 7 7 7 7 7 7 7 6 5 7 6 6 7 6 6 7 7 7 5 7 6 7 6 6 7 7 7 6
##  [186] 6 5 6 7 5 5 6 7 6 6 7 7 7 6 6 6 6 7 6 6 6 6 7 6 6 6 6 6 6 7 5 6 7 6 7 7 6 6 6
##  [223] 7 6 7 6 5 7 6 6 6 7 7 7 6 4 6 6 7 6 6 6 6 6 7 7 6 6 7 7 6 7 6 6 6 5 7 6 6 6
##  [260] 6 4 6 7 7 7 7 7 6 7 6 7 7 7 4 6 6 6 7 6 7 6 6 7 6 7 7 6 6 6 6 6 6 7 6 7 6
##  [297] 6 5 7 5 5 5 7 7 7 6 7 6 7 7 6 7 6 7 7 6 7 7 6 7 7 6 7 5 6 6 7 7 7 6 6 6 7
##  [334] 7 6 7 6 6 6 7 6 5 6 6 6 6 5 7 7 6 7 7 7 7 6 6 5 7 7 5 6 7 6 4 6 5 6 7 6 7
##  [371] 6 6 6 7 7 7 7 7 7 6 6 5 4 7 6 7 7 6 6 6 6 6 7 7 7 5 6 6 6 7 7 7 6 6 7 7 7
##  [408] 6 7 4 7 7 7 6 7 7 6 7 7 6 5 5 7 5 7 6 6 7 6 7 7 7 6 6 7 6 6 6 7 7 7 5 5 6
##  [445] 6 7 7 6 6 7 7 6 7 6 6 6 6 6 7 7 6 6 6 6 6 6 6 6 7 7 6 7 7 7 7 6 6 7 6 6 7 6 4
##  [482] 6 7 7 7 5 6 5 7 7 6 7 6 7 6 6 7 6 7 4 7 6 4 7 7 6 6 6 7 5 5 6 6 7 7 6 7 4
##  [519] 5 7 6 5 7 6 6 7 7 7 6 6 7 5 7 7 6 6 7 6 6 7 7 4 6 7 7 6 7 4 7 7 6 6 6 7 6
##  [556] 7 6 6 6 6 6 7 6 6 7 7 6 7 6 6 6 7 5 7 7 6 6 7 5 7 7 6 6 7 6 7 7 5 6 7 7 7
##  [593] 6 6 6 6 6 7 6 6 5 7 5 6 7 6 7 6 5 7 7 6 7 7 7 4 7 6 6 6 7 7 6 7 7 6 6 7 7
##  [630] 7 7 7 6 7 6 5 7 6 6 7 5 6 4 6 7 4 7 7 7 6 7 7 6 6 6 7 7 7 7 5 7 7 7 6 6 6
##  [667] 6 6 7 6 6 6 6 5 7 7 7 6 7 7 7 7 4 7 7 5 7 7 6 7 7 5 5 6 7 7 6 6 7 6 5 7 7
##  [704] 7 7 5 6 7 6 7 6 7 4 6 7 6 6 7 7 7 6 7 5 6 6 6 6 6 7 6 6 6 6 6 7 7 6 7 6 4
##  [741] 7 6 4 7 6 7 6 7 6 6 7 6 6 7 6 7 6 6 6 7 7 5 7 6 7 5 6 6 6 6 5 6 7 6 6 6 6
##  [778] 7 6 7 6 5 6 5 6 7 7 6 7 7 6 6 7 7 6 7 7 5 7 6 6 7 5 7 6 5 7 7 5 5 6 6 6 7
##  [815] 6 6 6 6 7 7 6 6 5 7 4 7 6 6 6 7 6 6 5 6 6 6 6 5 7 6 6 6 6 6 7 6 7 7 6 7 7
##  [852] 6 6 6 7 7 7 7 7 6 7 7 6 7 7 6 5 6 6 6 6 7 7 7 6 7 7 6 6 6 5 4 7 6 5 7 7 6
##  [889] 7 6 6 6 7 7 7 7 7 6 7 6 7 6 7 7 7 7 7 7 7 7 6 6 6 6 7 7 6 5 6 6 7 6 7 5 7
##  [926] 7 6 7 5 5 7 7 6 6 6 7 5 6 5 6 6 4 6 6 6 6 7 6 7 6 6 7 7 6 7 6 6 6 7 6 6 6
```

```
##  [963] 7 7 7 6 6 6 6 7 6 6 7 5 6 7 7 6 7 6 6 7 6 6 6 5 6 6 7 5 6 7 5 7 7 7 6 6 7
## [1000] 6
```

```
get_children(tree_up_3$info, 1)
```

```
## [1] 6 7 4 5
```

7.1.3 New Function - "Match"

"Match" is a function that look for which elements of the second vector match the first vector. For example:

```
match(1, c(2,3,4,1))
```

```
## [1] 4
```

It shows that the fourth element of the second vector match the first vector , 1.

```
match(c(1,2), c(2,3,4,5,1))
```

```
## [1] 5 1
```

This shows that the fifth element in the vector is 1, and the first matches 2.

Also, it can be written as A %in% B. If elements of A is in B, then it will return TRUE, otherwise FALSE.

7.2 Coding for Changing Tree

```
change_tree <- function(x, y, curr_tree, node_min_size){

  # In this function, since we are changing the tree structure, we have to make sure that the new tree
  # It has at least some observations in each terminal node, which is decided by the "node_min_size".

  # If it's a stump, there is nothing to change.
  if(nrow(curr_tree$info) == 1){return(curr_tree)}

  # Then, create a information box for the new three
  new_tree <- curr_tree

  # Since changing means change out the split variable and split rule, we need to make sure that this i
  internal_node <- which(as.numeric(new_tree$info[,"terminal"]) == 0)
  terminal_node <- which(as.numeric(new_tree$info[,"terminal"]) == 1)

  # Then, we can use a while loop to get a good tree.
  max_bad <- 100
  count_bad <- 0
  bad_tree <- TRUE

  while (bad_tree) {
    # When it's a bad tree, our changing has to be reset to make a new tree.
```

```
    new_tree <- curr_tree

    # Then we select a internal node to change.
    selected_node <- sample(internal_node, 1)

    # Use the get_children function to get all the children nodes of this node
    all_children <- get_children(new_tree$info, selected_node)

    # First, we find the nodes that corresponding to these children, then we delete those who are not i
    # This step is used to getting the further available value because we have to first find which poin
    use_node <- !is.na(match(new_tree$node, all_children))

    # Then we create a new split variable and a new split value.
    available_values <- NULL
    new_split <- sample(1:ncol(x), 1)

    # By using the selected points, we can get the available values.
    available_values <- sort(unique(x[use_node, new_split]))

    if(length(available_values) == 1){split_val <- available_values[1]}else if(length(available_values)
      split_val <- available_values[2]
    }else{

      split_val <- gdata::resample(available_values[-c(1, length(available_values))], 1)
    }

    # Then, we can update the tree information
    new_tree$info[selected_node, c("split_variable", "split_value")] <- c(new_split, split_val)

    # Updating the tree details using the fill tree function
    new_tree <- fill_tree_details(new_tree, x)

    # Now, we finished changing the splitting variable and splitting rule, but we have to check whether
    if(any(new_tree$info[terminal_node,"node_size"] <= node_min_size)){count_bad <- count_bad + 1}else{
      bad_tree <- FALSE
    }

    if(count_bad == max_bad){return(curr_tree)}

  }


  return(new_tree)

}
```

7.3 Check the Code

```
change_tree(x,y,tree_up_1, node_min_size = 10)
```

```
## $info
```

```
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        0       2       3     NA              1    13.75963  0      1000
## [2,]        1      NA      NA      1             NA          NA NA       937
## [3,]        1      NA      NA      1             NA          NA NA        63
##
## $node
##    [1] 2 2 2 3 2 2 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 3 2 2 2
##   [38] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##   [75] 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 3
##  [112] 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 2 2 2 2
##  [149] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2
##  [186] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2
##  [223] 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [260] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 2
##  [297] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [334] 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [371] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2
##  [408] 2 2 2 2 2 2 2 2 2 2 2 2 3 3 2 2 2 3 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2
##  [445] 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2
##  [482] 3 2 3 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2
##  [519] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [556] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [593] 2 3 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2
##  [630] 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [667] 2 2 2 3 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [704] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2
##  [741] 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [778] 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2
##  [815] 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2
##  [852] 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [889] 2 2 2 2 2 2 2 2 2 3 2 2 2 3 2 2 2 2 2 2 3 2 2 2 3 2 2 2 2 2 2 2 2 3 3 2
##  [926] 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [963] 2 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [1000] 2
```

```
change_tree(x,y,tree_up_2, node_min_size = 10)
```

```
## $info
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        0       2       3     NA              2   25.687516  0      1000
## [2,]        1      NA      NA      1             NA          NA NA       338
## [3,]        0       4       5      1              1    7.136951 NA       662
## [4,]        1      NA      NA      3             NA          NA NA       177
## [5,]        1      NA      NA      3             NA          NA NA       485
##
## $node
##    [1] 4 4 2 5 5 5 5 2 2 5 5 2 5 5 5 4 5 2 2 5 5 4 2 5 5 2 2 2 2 5 5 5 5 2 5 4 4
##   [38] 4 2 4 4 5 4 2 5 4 2 5 4 2 4 4 2 2 4 4 2 5 5 5 5 5 2 2 5 2 5 5 5 2 2 5 2 2
##   [75] 5 5 5 2 4 5 5 4 5 5 5 5 5 2 2 2 2 2 2 2 5 2 4 5 5 5 2 4 4 5 5 4 5 2 5 5 5
##  [112] 5 5 5 5 2 5 5 5 4 5 2 4 5 5 2 5 5 4 5 4 5 5 2 5 2 2 5 5 5 5 5 5 2 2 5 2 5
##  [149] 2 5 2 5 5 2 2 5 5 5 5 5 5 5 5 4 2 5 5 2 2 5 4 2 4 5 5 5 5 5 4 2 2 5 4 5 2
##  [186] 5 5 4 5 5 5 2 5 2 2 5 5 5 2 2 2 2 5 5 2 2 4 5 2 2 4 5 5 5 2 5 2 5 5 2 5 2
##  [223] 5 2 5 2 5 5 2 2 2 5 5 5 4 4 4 2 5 2 2 2 2 5 4 2 2 5 4 4 4 5 4 5 5 4 2 2 5
##  [260] 2 4 2 4 5 5 5 5 2 5 5 5 4 5 4 5 2 5 5 2 5 2 5 5 2 4 5 2 2 2 2 5 2 5 2 4 2
```

```
##  [297] 2 5 4 5 5 5 5 5 4 2 5 2 5 5 2 4 2 5 5 2 5 5 2 5 4 2 5 5 5 5 4 4 5 5 2 2 4
##  [334] 5 2 4 2 5 2 5 2 5 2 2 2 2 5 5 5 5 5 5 5 5 2 2 5 5 4 5 2 4 4 4 2 5 2 5 2 4
##  [371] 2 2 5 5 5 4 5 4 5 2 2 5 4 4 2 5 5 2 5 2 5 2 4 5 5 5 2 2 2 4 5 4 2 2 5 5 5
##  [408] 2 5 4 5 5 5 2 4 4 2 5 4 5 5 5 5 5 5 2 2 5 2 5 5 5 2 2 5 2 2 2 4 5 4 5 5 2
##  [445] 2 5 4 5 5 5 5 2 4 2 2 2 2 5 5 2 2 4 2 4 5 5 5 4 2 5 5 5 5 4 5 5 5 2 5 5 4
##  [482] 2 5 5 5 5 2 5 5 4 2 5 5 5 2 2 4 2 5 4 5 2 4 5 5 2 2 2 5 5 5 2 2 5 5 2 5 4
##  [519] 5 5 2 5 4 2 5 5 5 5 2 2 5 5 5 4 2 2 5 2 5 5 5 4 2 5 4 2 5 4 5 4 2 2 4 4 5
##  [556] 5 4 2 5 5 2 5 2 2 5 5 2 5 2 2 5 5 5 5 5 5 2 5 5 5 4 2 2 4 2 5 5 5 2 4 5 5
##  [593] 2 2 2 2 2 4 4 2 5 5 5 2 4 4 4 2 5 5 4 2 5 4 5 4 5 2 5 2 5 5 2 4 5 2 2 4 5
##  [630] 5 5 4 2 5 5 5 5 4 4 5 5 2 4 2 5 4 4 5 4 2 5 5 5 2 2 5 5 4 5 5 4 5 5 2 2 2
##  [667] 2 2 5 2 5 5 2 5 4 4 4 5 4 5 5 5 4 5 4 5 5 5 4 5 5 5 5 2 5 5 2 4 5 5 5 5 4
##  [704] 4 5 5 5 5 2 5 2 5 4 2 4 2 2 5 5 5 2 5 5 2 2 2 2 5 5 2 2 4 2 2 4 5 2 4 2 4
##  [741] 5 5 4 5 2 5 2 4 5 2 5 4 2 5 2 5 5 5 2 4 4 5 5 2 5 5 4 2 2 2 5 5 4 2 2 2 2
##  [778] 5 2 5 2 5 2 5 2 4 5 4 5 5 2 5 5 4 2 4 4 5 5 2 2 5 5 5 2 5 4 5 5 5 2 2 5 4
##  [815] 2 2 2 2 5 4 2 2 5 5 4 5 2 5 2 5 5 4 5 2 2 5 2 5 5 2 2 4 2 2 5 2 5 4 2 5 4
##  [852] 2 5 2 5 5 4 4 4 2 5 5 2 5 4 2 5 5 5 2 2 5 4 5 2 5 4 2 5 2 5 4 5 2 5 5 5 5
##  [889] 4 2 2 5 4 5 5 5 5 5 5 2 4 2 5 5 4 4 5 5 5 5 2 5 2 2 5 5 4 5 5 2 4 2 5 5 5
##  [926] 4 2 4 5 5 5 4 2 2 2 5 5 2 5 5 5 4 2 4 2 2 5 2 4 2 2 4 5 2 5 2 2 5 5 2 2 2
##  [963] 5 5 5 2 5 2 2 4 2 2 5 5 2 5 5 2 5 2 2 5 2 2 2 5 2 2 5 5 2 5 5 5 4 5 2 2 5
## [1000] 2
```

```
change_tree(x,y,tree_up_3, node_min_size = 10)
```

```
## $info
##      terminal child_L child_R parent split_variable split_value mu node_size
## [1,]        0       2       3     NA              2    40.93968  0      1000
## [2,]        0       6       7      1              2    29.32972 NA       888
## [3,]        0       4       5      1              2    42.12474 NA       112
## [4,]        1      NA      NA      3             NA          NA NA        23
## [5,]        1      NA      NA      3             NA          NA NA        89
## [6,]        1      NA      NA      2             NA          NA NA       465
## [7,]        1      NA      NA      2             NA          NA NA       423
##
## $node
##    [1] 7 5 6 6 6 7 7 6 6 4 7 6 7 6 6 6 7 6 6 5 7 7 6 5 7 6 6 6 6 6 7 4 7 6 7 7 7
##   [38] 5 6 7 7 7 6 6 7 5 6 7 6 6 7 7 6 6 7 6 6 7 7 6 5 5 6 6 6 6 6 6 7 6 6 7 6 6
##   [75] 7 5 7 6 7 7 7 7 6 6 4 7 7 6 6 6 6 6 6 6 7 6 6 7 7 7 6 6 6 7 7 7 6 6 7 6 6
##  [112] 7 7 7 7 6 6 7 7 6 5 6 7 5 7 6 7 7 6 7 7 7 6 6 6 6 6 7 6 6 7 4 5 6 6 7 6 6
##  [149] 6 7 6 7 7 6 6 4 6 7 7 7 7 7 7 7 6 5 7 6 6 7 6 6 7 7 7 5 7 6 7 6 6 7 7 7 6
##  [186] 6 5 6 7 5 5 6 7 6 6 7 7 7 6 6 6 6 7 6 6 6 6 7 6 6 6 6 6 6 7 5 6 7 6 7 7 6 6 6
##  [223] 7 6 7 6 5 7 6 6 6 7 7 7 6 5 6 6 7 6 6 6 6 6 7 7 6 6 7 7 6 7 6 6 6 5 7 6 6 6
##  [260] 6 5 6 7 7 7 7 7 6 7 6 7 7 7 5 6 6 6 7 6 7 6 6 7 6 7 7 6 6 6 6 6 6 7 6 7 6
##  [297] 6 5 7 5 5 5 7 7 7 6 7 6 7 7 6 7 6 7 7 6 7 7 6 7 7 6 7 5 6 6 7 7 7 6 6 6 7
##  [334] 7 6 7 6 6 6 7 6 5 6 6 6 6 6 5 7 7 6 7 7 7 7 7 6 6 5 7 7 4 6 7 6 5 6 5 6 7 6 7
##  [371] 6 6 6 7 7 7 7 7 7 7 6 6 4 5 7 6 7 7 6 6 6 6 6 7 7 7 5 6 6 6 7 7 7 6 6 7 7 7
##  [408] 6 7 5 7 7 7 6 7 7 6 7 7 6 5 5 7 4 7 6 6 7 6 7 7 7 6 6 7 6 6 6 7 7 7 5 4 6
##  [445] 6 7 7 6 6 7 7 6 7 6 6 6 6 7 7 6 6 6 6 6 6 6 7 7 6 7 7 7 7 6 6 7 6 6 7 6 5
##  [482] 6 7 7 7 4 6 4 7 7 6 7 6 7 6 6 7 6 7 5 7 6 5 7 7 6 6 6 7 5 5 6 6 7 7 6 7 5
##  [519] 5 7 6 5 7 6 6 7 7 7 6 6 7 5 7 7 6 6 7 6 6 7 7 5 6 7 7 6 7 4 7 7 6 6 6 7 6
##  [556] 7 6 6 6 6 6 7 6 6 7 7 6 7 6 6 6 7 5 7 7 6 6 7 5 7 7 6 6 7 6 7 7 5 6 7 7 7
##  [593] 6 6 6 6 6 7 6 6 4 7 5 6 7 6 7 6 5 7 7 6 7 7 7 5 7 6 6 6 7 7 6 7 7 6 6 7 7
##  [630] 7 7 7 6 7 6 5 7 6 6 7 5 6 5 6 7 4 7 7 7 6 7 7 6 6 6 7 7 7 7 5 7 7 7 6 6 6
##  [667] 6 6 7 6 6 6 6 5 7 7 7 6 7 7 7 7 5 7 7 5 7 7 6 7 7 5 5 6 7 7 6 6 7 6 4 7 7
##  [704] 7 7 4 6 7 6 7 6 7 4 6 7 6 6 7 7 7 6 7 5 6 6 6 6 6 7 6 6 6 6 6 7 7 6 7 6 5
```

```
##  [741] 7 6 5 7 6 7 6 7 6 6 7 6 6 7 6 7 6 6 6 7 7 4 7 6 7 5 6 6 6 6 5 6 7 6 6 6 6
##  [778] 7 6 7 6 5 6 5 6 7 7 6 7 7 6 6 7 7 6 7 7 5 7 6 6 7 5 7 6 5 7 7 4 5 6 6 6 7
##  [815] 6 6 6 6 7 7 6 6 5 7 5 7 6 6 6 7 6 6 5 6 6 6 6 5 7 6 6 6 6 6 7 6 7 7 6 7 7
##  [852] 6 6 6 7 7 7 7 7 6 7 7 6 7 7 6 5 6 6 6 6 7 7 7 6 7 7 6 6 6 4 5 7 6 5 7 7 6
##  [889] 7 6 6 6 7 7 7 7 7 6 7 6 7 6 7 7 7 7 7 7 7 7 6 6 6 6 7 7 6 4 6 6 7 6 7 4 7
##  [926] 7 6 7 5 5 7 7 6 6 6 7 5 6 5 6 6 5 6 6 6 6 7 6 7 6 6 7 7 6 7 6 6 6 7 6 6 6
##  [963] 7 7 7 6 6 6 6 7 6 6 7 5 6 7 7 6 7 6 6 7 6 6 6 5 6 6 7 4 6 7 5 7 7 7 6 6 7
## [1000] 6
```

8. Swap the Tree Function

8.1 Swap the Tree Function

In the swap setting, we swap two neighboring internal nodes around their split values and split variables, so we have to find the neighbouring internal nodes, then switch the splitting rules.

```r
swap_tree <- function(x, y, curr_tree, node_min_size){

  # Swap means we have to find two neighboring internal nodes, and then swaps their splitting values an

  # If the tree is a stump, then we cannot swap anymore
  if(nrow(curr_tree$info) == 1){ return(curr_tree) }

  # Create an information box for new tree
  new_tree <- curr_tree

  # Same as "change", we need to find which nodes are internal and which are terminal
  internal_node <- which(as.numeric(new_tree$info[,"terminal"]) == 0)
  terminal_node <- which(as.numeric(new_tree$info[,"terminal"]) == 1)

  # If the tree is too small, like it only has one or two internal node, then swapping is useless since
  if(length(internal_node) < 3 ){ return(curr_tree)}

  # Then we need to find a pair of neighboring internal nodes
  parent_internal <- as.numeric(new_tree$info[internal_node, "parent"])

  # This step, we bind two internal nodes by column and create the pairs of internal nodes.
  # -1 because the root node doesn't have a parent, so it's useless as a child.
  # Be careful that this step creates a p*2 matrix, because internal_node is a vector, and parent is al
  # This matrix describes:
      # 1st column: Which points are internal nodes
      # 2nd column: Which nodes are the parent of the 1st column correspondingly.
  pairs_internal <- cbind(internal_node, parent_internal)[-1,]


  # Then create a loop to get good trees.
  # Set the maximum number of bad trees.

  max_bad <- 10
  count_bad <- 0
  bad_tree <- TRUE
```

```
  while (bad_tree) {
    new_tree <- curr_tree

    # Pick up a random pair
    selected_node <- sample(1:nrow(pairs_internal),1)

    # Get the split variable and split value for this pair
    # 1 for some internal node, 2 for the parent of this node
    swap_1_rule <- as.numeric(new_tree$info[pairs_internal[selected_node,1], c("split_variable", "split
    swap_2_rule <- as.numeric(new_tree$info[pairs_internal[selected_node,2], c("split_variable", "split

    # Change the tree information matrix, interchange the splitting rules
    new_tree$info[pairs_internal[selected_node,1], c("split_variable", "split_value")] <- swap_2_rule
    new_tree$info[pairs_internal[selected_node,2], c("split_variable", "split_value")] <- swap_1_rule


    # Then we should fill in the tree details
    new_tree <- fill_tree_details(new_tree, x)

    # Check whether it's a bad tree
    if(any(as.numeric(new_tree$info[,"node_size"]) <= node_min_size)){ count_bad <- count_bad + 1}else{
      bad_tree <- FALSE
    }

    if(max_bad == count_bad){ return(curr_tree)}

  }

  return(new_tree)

}
```

8.2 Testing Swap Function

```
tree_up_4 <- grow_tree(x,y,tree_up_3, node_min_size = 10)
swap_tree(x,y,curr_tree = tree_up_4, node_min_size = 10)
```

```
## $info
##       terminal child_L child_R parent split_variable split_value mu node_size
##  [1,]        0       2       3     NA              1    7.136951  0      1000
##  [2,]        0       6       7      1              2   29.329720 NA       265
##  [3,]        0       4       5      1              2   40.939684 NA       735
##  [4,]        1      NA      NA      3             NA          NA NA       648
##  [5,]        1      NA      NA      3             NA          NA NA        87
##  [6,]        0       8       9      2              2   21.472649 NA       124
##  [7,]        1      NA      NA      2             NA          NA NA       141
##  [8,]        1      NA      NA      6             NA          NA NA        52
##  [9,]        1      NA      NA      6             NA          NA NA        72
##
## $node
##    [1] 7 7 4 4 4 4 4 9 8 5 4 8 4 4 4 9 4 4 8 5 4 7 4 5 4 4 9 4 4 4 4 5 4 4 4 7 7
```

```
##   [38] 7 4 7 7 4 9 8 4 7 4 4 9 4 7 7 4 8 7 9 8 4 4 4 5 5 9 8 4 4 4 4 4 4 4 4 4 4
##   [75] 4 5 4 8 7 4 4 7 4 4 5 4 4 4 8 4 8 4 4 4 4 4 9 4 4 4 4 9 9 4 4 7 4 4 4 4 4
##  [112] 4 4 4 4 4 4 4 4 9 5 4 7 5 4 4 4 4 9 4 7 4 4 4 4 4 9 4 4 4 4 5 5 4 8 4 8 4
##  [149] 9 4 4 4 4 8 4 5 4 4 4 4 4 4 4 7 4 5 4 4 4 4 9 4 7 4 4 5 4 4 7 4 9 4 7 4 4
##  [186] 4 5 9 4 5 5 9 4 4 4 4 4 4 4 4 4 4 4 4 4 4 7 4 4 4 9 4 4 5 4 4 4 4 4 4 4 4
##  [223] 4 4 4 4 5 4 4 4 8 4 4 4 9 7 9 4 4 4 4 4 4 4 7 8 4 4 7 9 7 4 9 4 5 7 4 8 4
##  [260] 4 7 4 7 4 4 4 4 4 4 4 4 7 4 7 4 8 4 4 4 4 4 4 4 4 7 4 4 4 8 4 4 8 4 4 7 4
##  [297] 4 5 7 5 5 5 4 4 7 4 4 4 4 4 4 7 4 4 4 8 4 4 8 4 7 9 4 5 4 4 7 7 4 4 4 4 7
##  [334] 4 4 7 4 4 4 4 4 5 4 4 4 4 5 4 4 4 4 4 4 4 4 4 5 4 7 5 4 7 9 7 4 5 9 4 9 7
##  [371] 4 4 4 4 4 7 4 7 4 4 4 5 7 7 8 4 4 4 4 9 4 4 7 4 4 5 8 4 4 7 4 7 4 4 4 4 4
##  [408] 4 4 7 4 4 4 4 7 7 4 4 7 4 5 5 4 5 4 4 9 4 4 4 4 4 8 4 4 9 4 4 7 4 7 5 5 4
##  [445] 4 4 7 4 4 4 4 4 7 4 4 4 9 4 4 8 4 9 8 9 4 4 4 7 9 4 4 4 4 9 4 4 4 9 4 4 7
##  [482] 4 4 4 4 5 8 5 4 7 8 4 4 4 8 4 7 4 4 7 4 4 7 4 4 4 4 4 4 4 5 5 8 4 4 4 4 4 7
##  [519] 5 4 4 5 7 4 4 4 4 4 4 8 4 5 4 7 9 4 4 4 4 4 4 7 4 4 7 4 4 7 4 7 4 8 9 7 4
##  [556] 4 9 4 4 4 4 4 8 4 4 4 9 4 4 4 4 4 5 4 4 4 4 4 5 4 7 4 4 7 4 4 4 5 4 7 4 4
##  [593] 4 4 4 9 4 7 9 4 5 4 5 4 7 9 7 8 5 4 7 4 4 7 4 7 4 9 4 9 4 4 8 7 4 4 4 7 4
##  [630] 4 4 7 9 4 4 5 4 9 9 4 5 4 7 4 4 7 7 4 7 4 4 4 4 4 4 4 4 7 4 5 7 4 4 9 8 4
##  [667] 8 4 4 4 4 4 4 5 7 7 7 4 7 4 4 4 7 4 7 5 4 4 9 4 4 5 5 4 4 4 8 9 4 4 5 4 7
##  [704] 7 4 5 4 4 9 4 4 4 7 4 7 4 8 4 4 4 9 4 5 4 4 4 4 4 4 8 8 9 4 4 7 4 4 7 4 7
##  [741] 4 4 7 4 4 4 4 7 4 9 4 9 4 4 4 4 4 4 9 7 7 5 4 4 4 5 9 8 4 4 5 4 7 4 4 9 4
##  [778] 4 4 4 4 5 4 5 4 7 4 9 4 4 4 4 4 4 7 4 7 7 5 4 4 4 4 4 5 4 4 5 7 4 5 5 4 9 4 7
##  [815] 4 4 4 4 4 7 4 4 5 4 7 4 4 4 4 4 4 4 9 5 8 4 4 4 5 4 4 4 4 9 9 4 4 4 4 7 4 4 7
##  [852] 4 4 9 4 4 7 7 7 4 4 4 4 4 4 7 9 5 4 4 8 4 4 7 4 4 4 7 4 4 4 5 7 4 4 5 4 4 4
##  [889] 7 4 4 4 7 4 4 4 4 4 4 4 7 4 4 4 7 7 4 4 4 4 8 4 4 4 4 4 4 9 5 4 4 7 4 4 5 4
##  [926] 7 8 7 5 5 4 7 8 4 8 4 5 4 5 4 4 7 4 9 9 4 4 4 7 4 4 7 4 8 4 4 8 4 4 8 4 4
##  [963] 4 4 4 4 4 8 4 7 4 9 4 5 9 4 4 4 4 4 4 4 4 4 4 5 4 9 4 5 4 4 5 4 7 4 4 4 4
## [1000] 4
```

9. Full Conditional Distribution

Since the distribution of the tree is not in a closed form, we need to get the exact value of the $p(Tree|others)$. But we can get a proportional number to use the MH steps. Before getting the code to calculate the tree full conditional, the other parameters could be presented in a closed form as follows.

This model can be described as follows:

$$Y_{ij} \sim^{i.i.d} N(\mu_i, \sigma_i^2)$$

where μ_i is the mean of the i^{th} terminal node, and the σ_i^2 is the variance of the i^{th} terminal node. We can further set the model as follows:

$$\mu_i \sim^{i.i.d} N(\theta, \tau_\mu), \qquad \sigma_i \sim^{ind} IG(a_i, b_i)$$

However, we can also set a flat prior on μ_i and σ_i as follows:

$$p(\mu_i, \sigma_i^2) \propto \frac{1}{\sigma_i^2}$$

Actually, we can further set a hierarchical model and put priors on θ, τ_μ and other parameters. But here we just set them as hyperparameters and fixed.

Therefore, the posterior distribution of each parameter can be presented as follows if the tree structure is given:

$$p(\mu_i|others) \propto exp(-\frac{\sum_{j=1}^{n_i}(y_{ij}-\mu_i)^2}{2\sigma_i^2} - \frac{(\mu_i-\theta)^2}{2\tau_\mu^2}) \propto exp\left(-\frac{1}{2}\left((\frac{n_i}{\sigma_i^2}+\frac{1}{\tau_\mu^2})\mu_i^2 - 2\mu_i(\frac{\sum_{j=1}^{n_i}y_{ij}}{\sigma_i^2}+\frac{\theta}{\tau_\mu^2})\right)\right)$$

$$\mu_i \sim N(\frac{\frac{\sum_{j=1}^{n_i}y_{ij}}{\sigma_i^2}+\frac{\theta}{\tau_\mu^2}}{\frac{n_i}{\sigma_i^2}+\frac{1}{\tau_\mu^2}}, \frac{1}{\frac{n_i}{\sigma_i^2}+\frac{1}{\tau_\mu^2}})$$

For σ_i^2:

$$p(\sigma_i^2|others) \propto (\sigma_i^2)^{-\frac{n_i}{2}} exp(-\frac{\sum_{j=1}^{n_i}(y_{ij}-\mu_i)^2}{2}(\sigma_i^2)^{-1}) \times (\sigma_i^2)^{-a_i+1} exp(-\frac{1}{b_i}(\sigma_i^2)^{-1})$$

$$\sigma_i^2 \sim IG(\frac{n_i}{2}+a_i, \frac{\sum_{j=1}^{n_i}(y_{ij}-\mu_i)^2}{2}+b_i)$$

Here is the case that I regard only μ_i and σ_i as unknown parameters. If we further assume all σ_i^2 are equal to each other, and $\theta = 0$, the full conditional posterior of μ_i and σ^2 become:

$$\mu_i \sim N(\frac{\frac{\sum_{j=1}^{n_i}y_{ij}}{\sigma_i^2}}{\frac{n_i}{\sigma_i^2}+\frac{1}{\tau_\mu^2}}, \frac{1}{\frac{n_i}{\sigma_i^2}+\frac{1}{\tau_\mu^2}})$$

If we use a reparameterization that $\tau_\mu^2 = \frac{\sigma^2}{a}$, the distribution could be written as:

$$\mu_i \sim N(\frac{\sum_{i=1}^{n_i}y_{ij}}{n_i+a}, \frac{\sigma_i^2}{n_i+a})$$

$$\sigma^2 \sim IG(\frac{N}{2}+a, \frac{\sum_{i=1}^{I}\sum_{j=1}^{n_i}(y_{ij}-\mu_i)^2}{2}+b)$$

```
loglike <- function(tree, x, y, mu, sigma){
  # Here, both sigma and mu are vector of length how many tree terminals. And for jth point in ith each
  # Be careful that sigma now is the variance instead of standard deviation.
  # Also, if all the terminal node observations share the same variance, then sigma is a single number.



  # Select those nodes which are terminal
  terminal_node <- which(as.numeric(tree$info[,"terminal"])==1)
  internal_node <- which(as.numeric(tree$info[,"terminal"])==0)
  log_post <- NULL

  # Then, get the posterior density in each node.

  for (i in 1:length(terminal_node)) {
    node_i_all <- y[which(tree$node==terminal_node[i])]
    log_post[i] <- mvtnorm::dmvnorm(node_i_all, mean = rep(mu[i], length(node_i_all)), sigma = diag(sig

  }

  return(sum(log_post))

}
```

```r
tree_marginal <- function(tree, x, y, c, alpha, beta){
  # Calculate the marginal distribution for the tree to get a sample from.
  I <- length(table(tree$node))
  ni <- table(tree$node)
  terminal_node <- which(tree$info[,"terminal"]== 1 )
  part1 <- -sum(log(sqrt(ni+c)))
  part2 <- lgamma(length(y)/2 + alpha)

  M <- 0

  for (i in 1:I) {
    y_i <- y[tree$node==terminal_node[i]]
    M <- M + (sum(y_i^2)-sum(y_i)^2/(ni[i] + c))/2

  }

  part3 <- (length(y)/2 + alpha)*log(beta+M)

  # Get the log marginal likelihood
  Log_mar <- part1 + part2 - part3
  return(Log_mar)
}
```

```r
# Since the distribution of mu and sigma are conjugate, we can directly sample from the posterior distr
# This is a function that sample the mean of each terminal node as a vector together.



fill_update_mu <- function(y, sigma, c, curr_tree){
  ni <- table(curr_tree$node)
  mean_vec <- NULL
  var_vec <- NULL
  # Update each mean and variance, since they are iid distributed, I can generate them together by usin
  for (i in 1:length(ni)) {
    mean_vec[i] <- (sum(y[curr_tree$node==names(ni)[i]])) /
                   (ni[i] + c )
    var_vec[i] <- sigma / (ni[i] + c )
  }

  mu_update <- mvtnorm::rmvnorm(1, mean = mean_vec,  sigma = diag(var_vec, ncol = length(ni)))
  new_tree <- curr_tree

  new_tree$info[which(as.numeric(new_tree$info[,"terminal"])==1), "mu"] <- mu_update
  return(new_tree)
}

sample_sigma <- function(y, mu, alpha_sig, beta_sig, tree, c){

  # Get the posterior parameters for the inverse gamma distribution
  ni <- table(tree$node)
  alpha <- length(y)/2 + length(ni)/2 + alpha_sig
  SS <- NULL
  for (i in 1:length(ni)) {
```

```
    SS[i] <- sum((y[tree$node==ni[i]]-mu[i])^2)
  }
  beta <- sum(SS)/2 + c*sum(mu^2)/2 + beta_sig
  sigma_update <- 1/rgamma(1, shape = alpha, rate = beta)
  return(sigma_update)
}
```

10. Get the prior distribution of the tree p(tree)

Remember that the splitting probability can be written as:

$$\alpha(1+d)^{-\beta}$$

```
get_tree_prior <- function(tree, alpha, beta){
  # First, we need to work the depth of the tree
  # So we need to find the level of each node, then the depth is the maximum of the level

  level <- rep(NA, nrow(tree$info))
  # The first node is the 0 depth of the tree
  level[1] <- 0

  if(nrow(tree$info)==1){return(log(1-alpha))} # Because the tree depth is 0.

  for (i in 2:nrow(tree$info)) {
    # We need to first find the current parent
    curr_parent <- as.numeric(tree$info[i,"parent"])

    # Then this child must have one more level than its parent
    level[i] <- level[curr_parent] + 1
  }

  # If we only compute the internal nodes
  internal_node <- which(as.numeric(tree$info[,"terminal"])==0)
  log_prior <- 0
  for (i in 1:length(internal_node)) {
    log_prior <- log_prior + log(alpha) - beta*log(1 + level[internal_node[i]])
  }

  # Also, in each terminal node, it does not split
  terminal_node <- which(as.numeric(tree$info[,"terminal"])==1)
  for (i in 1:length(terminal_node)) {
    log_prior <- log_prior + log(1-alpha*(1+level[terminal_node[i]])^(-beta))

  }

  return(log_prior)
}
```

11. Train the Model

```
train_bart <- function(x, y, num_trees = 1, # number of trees
                       control = list(node_min_size = 5), # control the minimum terminal node size
                       hyper = list(alpha = 0.95, beta = 2, c = 10, alpha_sig=0.01, beta_sig=0.01), # G
                       init = list(sigma = 1), # sigma2 initial value
                       mcmc_par = list(iter = 1000, # number of total iterations
                                       burn = 100, # number of burn-in
                                       thin = 1) # how to thin the chain

){

  # Scale the covariates
  # We need to record how we scaled them so that we can predict new observations.

  # Only calculate those columns who are numeric


  # Get the columns that are numeric
  # Create a holder

  if(is.null(ncol(x))){
    center_x_num = mean(x)
    scale_x_num = sd(x)
    }else{
      center_x <- apply(x, 2, mean)
      scale_x <- apply(x, 2, sd)}

    x <- scale(x)

      center_y <- mean(y)
      scale_y <- sd(y)
      y <- scale(y)




  # Extract control parameters
  node_min_size <- control$node_min_size

  # Extract initial values
  sigma <- init$sigma
  log_lik <- 0

  # Extract hyperparameters
  alpha <- hyper$alpha
  beta <- hyper$beta
  c <- hyper$c
  alpha_sig <- hyper$alpha_sig
  beta_sig <- hyper$beta_sig
```

```r
# Extract MCMC details
iter <- mcmc_par$iter
burn <- mcmc_par$burn
thin <- mcmc_par$thin
totIter <- burn + iter*thin

# Create containers
store_size <- iter
tree_store <- vector("list", store_size)
sigma2_store <- rep(NA, store_size)
log_like_store <- rep(NA, store_size)
num_node <- rep(NA, store_size)
pred <- matrix(NA, nrow = store_size, ncol = length(y))
log_mar_tree <- rep(NA, store_size)

if(num_trees == 1 ){full_condition_store <- rep(NA, store_size)}else{
  full_condition_store <- matrix(NA, ncol = num_trees, nrow = store_size)
}


# Create a tree stump
curr_tree <- create_stump(num_trees = num_trees, y = y, x = x)[[1]]

# Initialize
new_tree <- curr_tree


pb = utils::txtProgressBar(min = 1, max = totIter,
                           style = 3, width = 60,
                           title = 'Running Models...')


# MCMC Iteration
for(i in 1:totIter){
  utils::setTxtProgressBar(pb, i)
  if((i > burn ) & ((i-burn)%%thin == 0 )){

    curr <- (i-burn)/thin
    tree_store[[curr]] <- curr_tree
    sigma2_store[curr] <- sigma
    log_like_store[curr] <- log_lik
    pred[curr,] <- prediction
    num_node[curr] <- num.node
    log_mar_tree[curr] <- log_mar_tree_curr
  }

  # Propose a new tree.
  # To prevent the tree from being too small, we need to grow at the beginning

  type <- sample(c("grow","prune","change","swap"), 1)

  if(i < max( floor(0.1*burn), 10) ){type = "grow"}
```

```r
# This is a proposed tree, but we need to figure out whether we should use this tree.
new_tree <- update_tree(x,y, type = type, curr_tree = curr_tree, node_min_size = node_min_size)


# New tree, compute the log of marginalized likelihood plus the log of the tree prior
# First, subtract the mean of the tree



# Use Metropolis-Hasting to sample a new tree here, by using the marginal distribution of tree

# Get the new log probability
l_new <- tree_marginal(new_tree, x, y, c, alpha, beta) + get_tree_prior(new_tree, alpha = alpha, be

# Get the old log probability
l_old <- tree_marginal(curr_tree, x, y, c, alpha, beta) + get_tree_prior(curr_tree, alpha = alpha,


# Since the proposal is symmetric, we can record the transition probability as follows:
a <- exp(l_new - l_old)
l <- runif(1)



# Need to save the full conditional to check the convergence
if(  (i>burn) & ( ((i-burn)%%thin) ==0) ) {full_condition_store[curr] <- l_old}

if(a > l){curr_tree <- new_tree }

# Then, we should update the mu for each terminal node and update the sigma

curr_tree <- fill_update_mu(y, sigma = sigma, c = c, curr_tree = curr_tree)
mu <- curr_tree$info[which(as.numeric(curr_tree$info[,"terminal"])==1),"mu"]

sigma <- sample_sigma(y, mu = mu, alpha_sig, beta_sig, tree = curr_tree, c)

# Get the log likelihood for the model

log_lik <- get_like(curr_tree = curr_tree, x, y, sigma = sigma)


#For now, I will also get the prediction of this iteration
# We have subtracted mu in the previous step

all_mean <- curr_tree$info[,"mu"]
pred_mean <- all_mean[curr_tree$node]
prediction_scaled <- rnorm(length(y), mean = pred_mean, sd = sqrt(sigma))
prediction <- prediction_scaled*scale_y + center_y


num.node <- sum(as.numeric(curr_tree$info[,"terminal"]==1))

log_mar_tree_curr <- tree_marginal(curr_tree, x, y, c, alpha, beta) + get_tree_prior(curr_tree, alp
```

```
  }
  # End the iteration loop
  cat('\n')

  return(list(
    tree = tree_store,
    sigma2_scaled = sigma2_store,
    log_like = log_like_store,
    center_x = center_x,
    scale_x = scale_x,
    center_y = center_y,
    scale_y = scale_y,
    iter = iter,
    burn = burn,
    thin = thin,
    store_size = store_size,
    prediction = pred,
    num_node = num_node,
    log_mar_tree = log_mar_tree

  ))


}
```

```
get_like <- function(curr_tree, x, y, sigma){

  # Find which node is terminal node and find its mean

  terminal_node <- which(as.numeric(curr_tree$info[,"terminal"])==1)
  terminal_mean <- curr_tree$info[terminal_node, "mu"]

  res <- 0

  for (i in 1:length(terminal_node)) {

    group_index <- which(curr_tree$node == terminal_node[i])
    res <- res + sum(dnorm(y[group_index], mean = terminal_mean[i], sd = sqrt(sigma), log = TRUE))

  }

  return(res)


}
```

12. Simulated Data Set

A same simulated model as the paper presented was as follows:

$$Y = f(X_1, X_2) + 2\epsilon,\ \epsilon \sim N(0, 1)$$

$$f(X_1, X_2) = \begin{cases} 8.0 \ \ if \ \ X_1 \leq 5.0 \ and \ X_2 \in \{A, B\} \\ 2.0 \ \ if \ \ X_1 > 5.0 \ and \ X_2 \in \{A, B\} \\ 1.0 \ \ if \ \ X_1 \leq 3.0 \ and \ X_2 \in \{C, D\} \\ 5.0 \ \ if \ \ 3.0 < X_1 \leq 7.0 \ and \ X_2 \in \{C, D\} \\ 8.0 \ \ if \ \ X_1 > 7.0 \ and \ X_2 \in \{C, D\} \end{cases}$$

Let X_1 be the sequence from 0.1 to 10, with a break of 0.1, 100 in total. Let X_2 be a repeated uniform random sample from A, B, C, D.

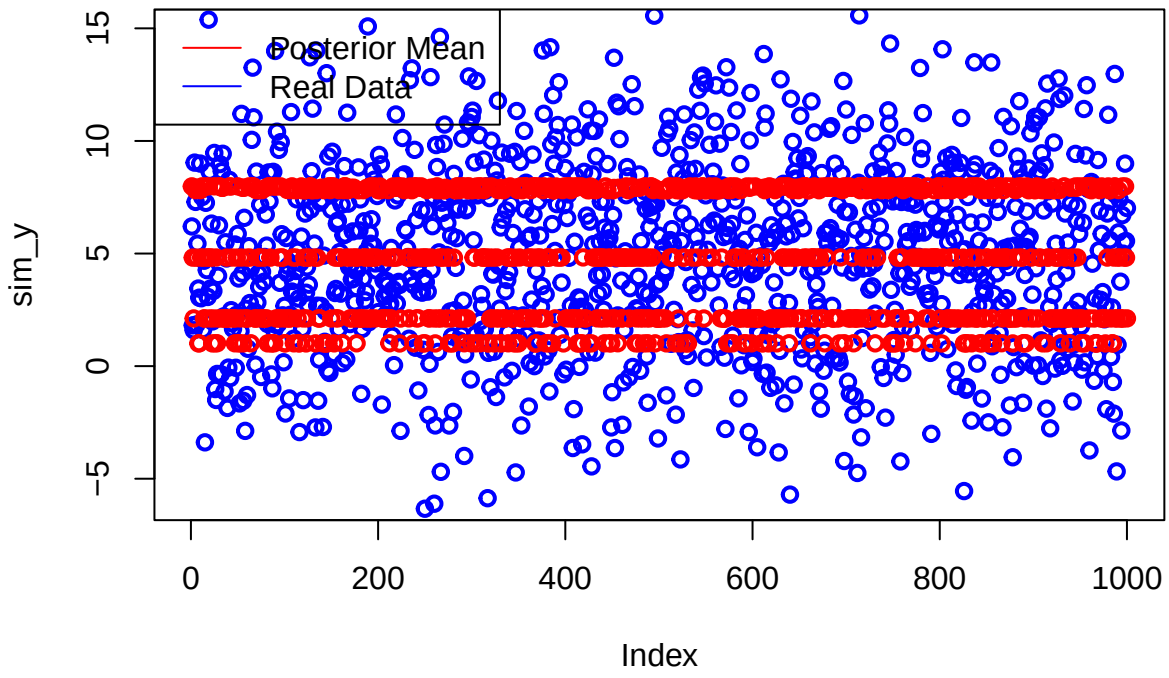
```
set.seed(0)
x_grid <- seq(from = 0.5, to = 9.5, by = 1)
x1 <- sample(x_grid, 1000, replace = TRUE)
x2 <- sample(1:4, replace = TRUE, size = 1000)
y <- rep(NA, 1000)
res <- 3*rnorm(1000, mean = 0, sd = 1)

y[ x1<=5 & (x2==1 | x2==2)] <- 8
y[ x1>5 & (x2==1 | x2==2)] <- 2
y[ x1<=3 & (x2==3 | x2==4)] <- 1
y[ (x1>3 & x1<=7) & (x2==3 | x2==4)] <- 5
y[ x1>7 & (x2==3 | x2==4)] <- 8

cat <- c("A","B","C","D")
x2 <- cat[x2]
sim_dat <- data.frame(y = as.numeric(y+res), x1= as.numeric(x1), x2=x2)
sim_x <- model.matrix(~ -1 + sim_dat[,2]+sim_dat[,3])
sim_y <- sim_dat[,1]
```

13. Simulated Prediction

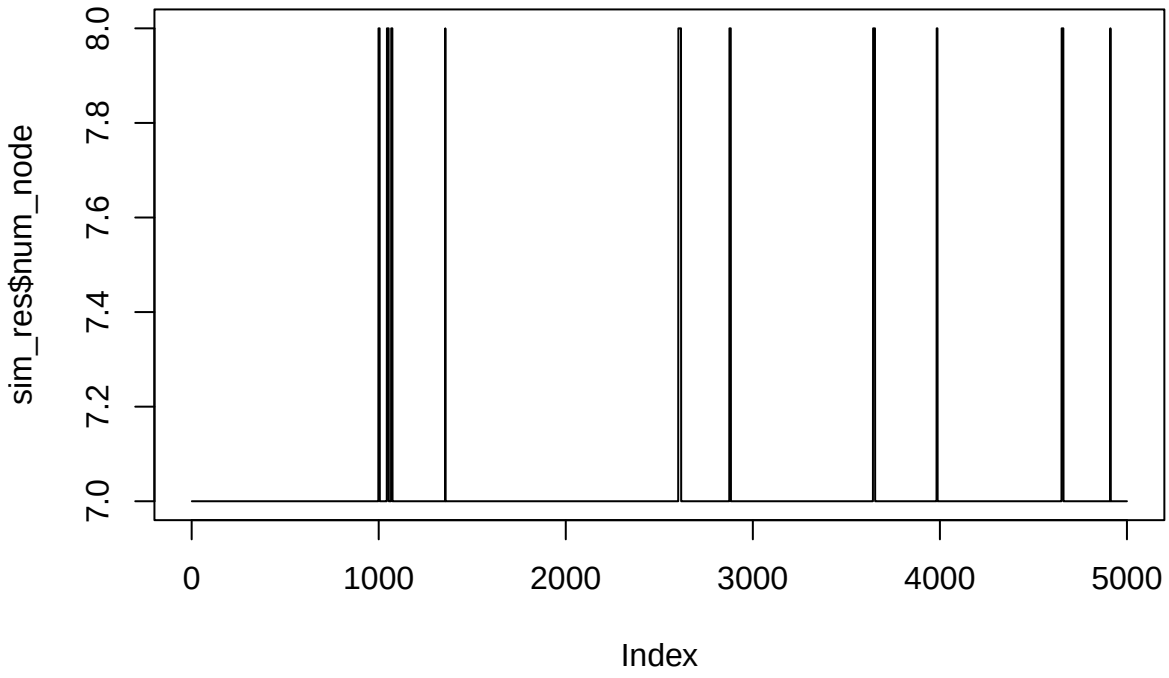
```
sim_res <- train_bart(x = sim_x, y = sim_y, mcmc_par = list(iter = 5000, burn = 2000, thin = 2),hyper =
```

```
##   |                                                                      |
```

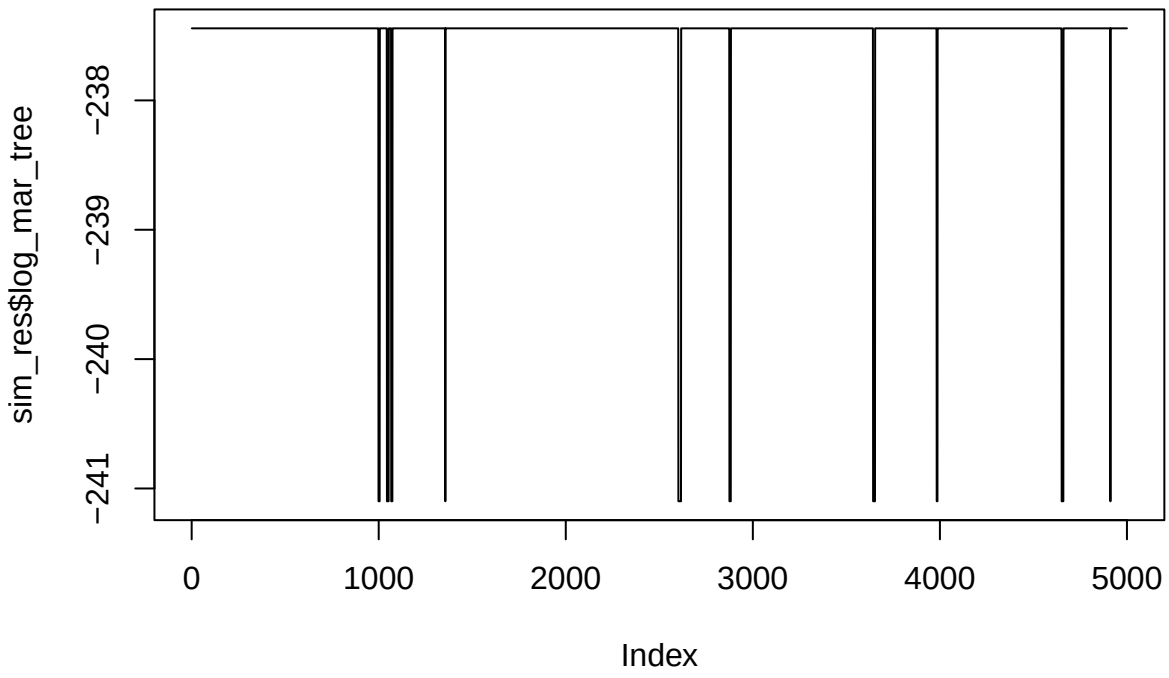
```
pred_y_mean_all <- apply(sim_res$prediction, 2, mean)

plot(sim_y, type = "p", col = "blue", lwd = 2, ylim = c(-6, 15))
lines(pred_y_mean_all, type = "p", col = "red", lwd = 2)
legend("topleft",c("Posterior Mean","Real Data"), col = c("red","blue"), lty = c(1,1))
```



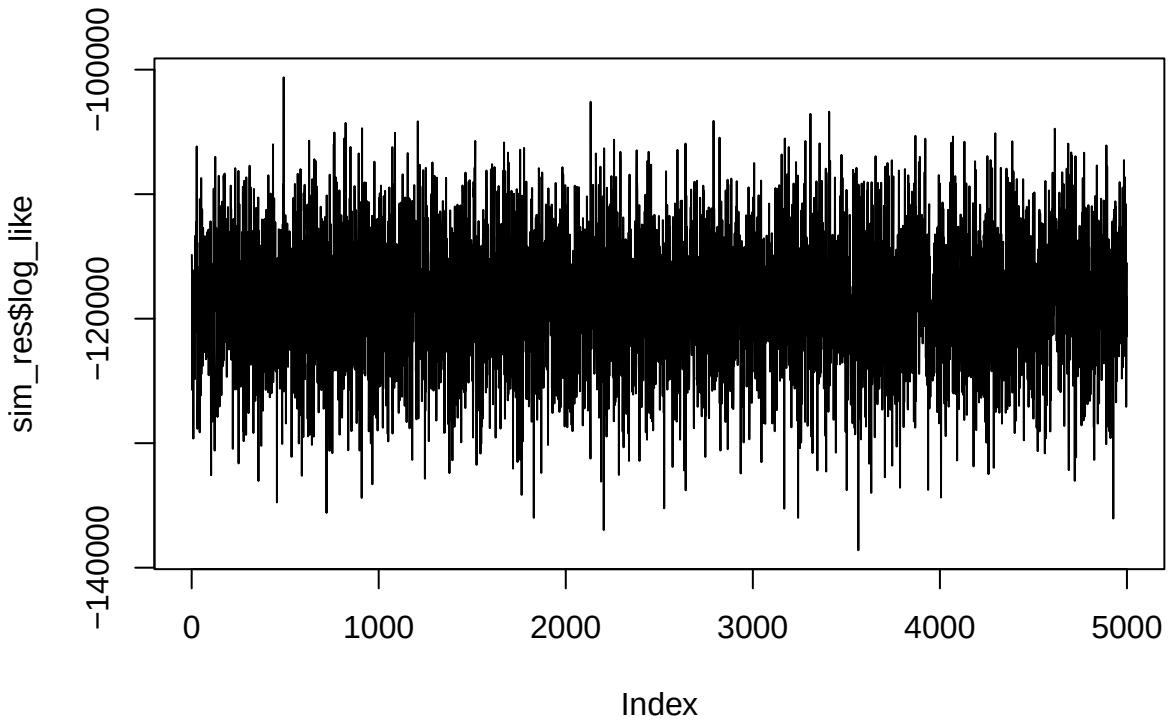

```
plot(sim_res$num_node, type = 'l')
```




```
plot(sim_res$log_mar_tree, type = 'l')
```




```
plot(sim_res$log_like, type = 'l')
```

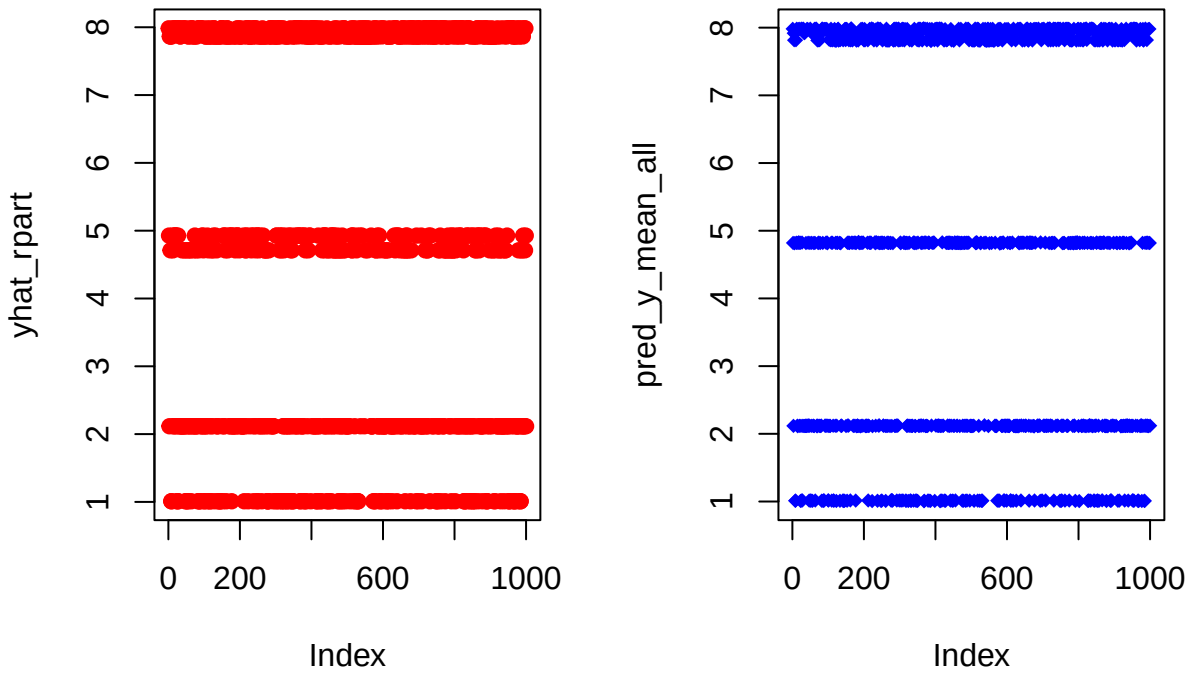


It seems that due to the big residual I added, the overall trend of y is not so obvious, however, I changed the residual to have a normal with mean 0 and var 0.01 and tried again, the result is as follows.

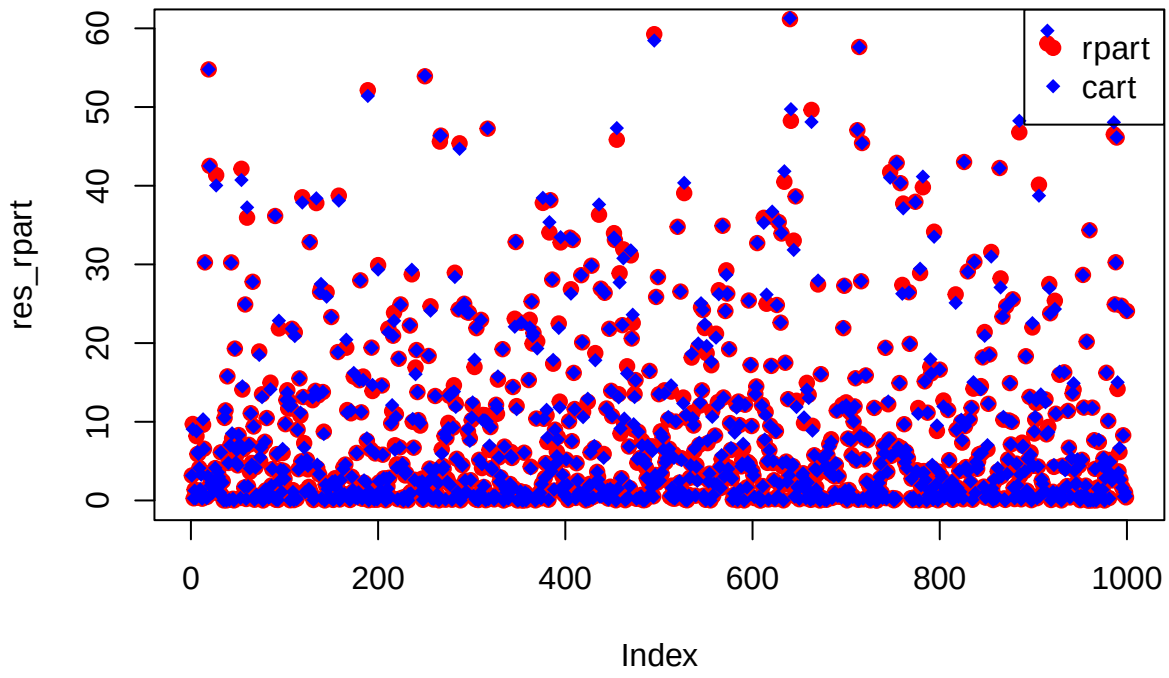
14. Compare with rpart(All Data Included as Train Set)

In this section, I am going to compare the prediction squared error with the rpart function.

```
library(rpart)
rmod <- rpart(y~., data = sim_dat)
yhat_rpart <- predict(rmod, sim_dat[,2:3])

par(mfrow = c(1,2))
#plot(yhat_rpart, type = 'p', pch = 19, col = "red")
#lines(pred_y_mean_all, type = 'p', pch = 18, col = "blue")
#legend("topright", c("rpart","cart"), pch = c(19,18), col = c("red","blue"))


plot(yhat_rpart, type = 'p', pch = 19, col = "red")
plot(pred_y_mean_all, type = 'p', pch = 18, col = "blue")
```




```
res_rpart <- (sim_dat[,1] - yhat_rpart)^2
res_cart <- (pred_y_mean_all - sim_dat[,1])^2
plot(res_rpart, type = 'p', pch = 19, col = "red", ylim = c(-0.1,60))
lines(res_cart, type = 'p', pch = 18, col = "blue")
legend("topright", c("rpart","cart"), pch = c(19,18), col = c("red","blue"))
```




```
sum(res_rpart)
```

```
## [1] 9610.162
```

```
sum(res_cart)
```

```
## [1] 9612.263
```

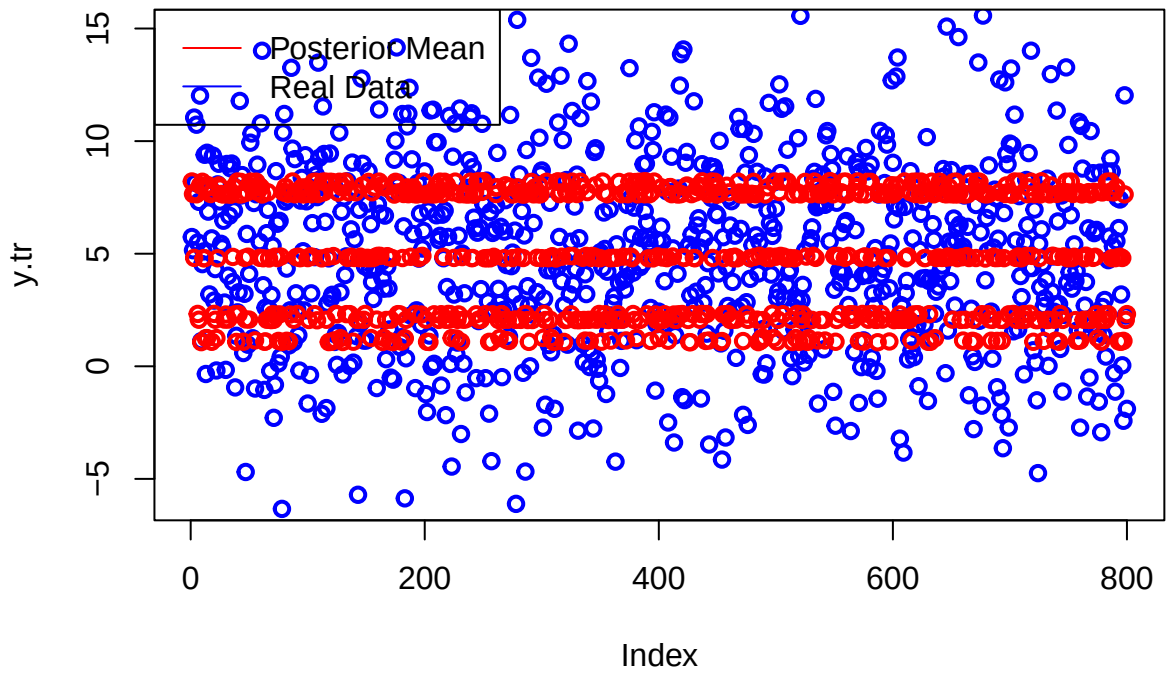
15. Out of Sample Prediction of CART

Here, we are going to use the result of the bart function to predict the out of sample observations. 800 of the data was randomly selected as train set and the rest as validation set.

```
tr.index <- sample(size = 800, 1:1000, replace = FALSE)
x.tr <- sim_x[tr.index,]
x.va <- sim_x[-tr.index,]
y.tr <- sim_y[tr.index]
y.va <- sim_y[-tr.index]
cv.bart <- train_bart(x.tr, y.tr, mcmc_par = list(iter = 5000, burn = 200, thin = 2),hyper = list(alpha
```

```
##   |                                                                      |
```

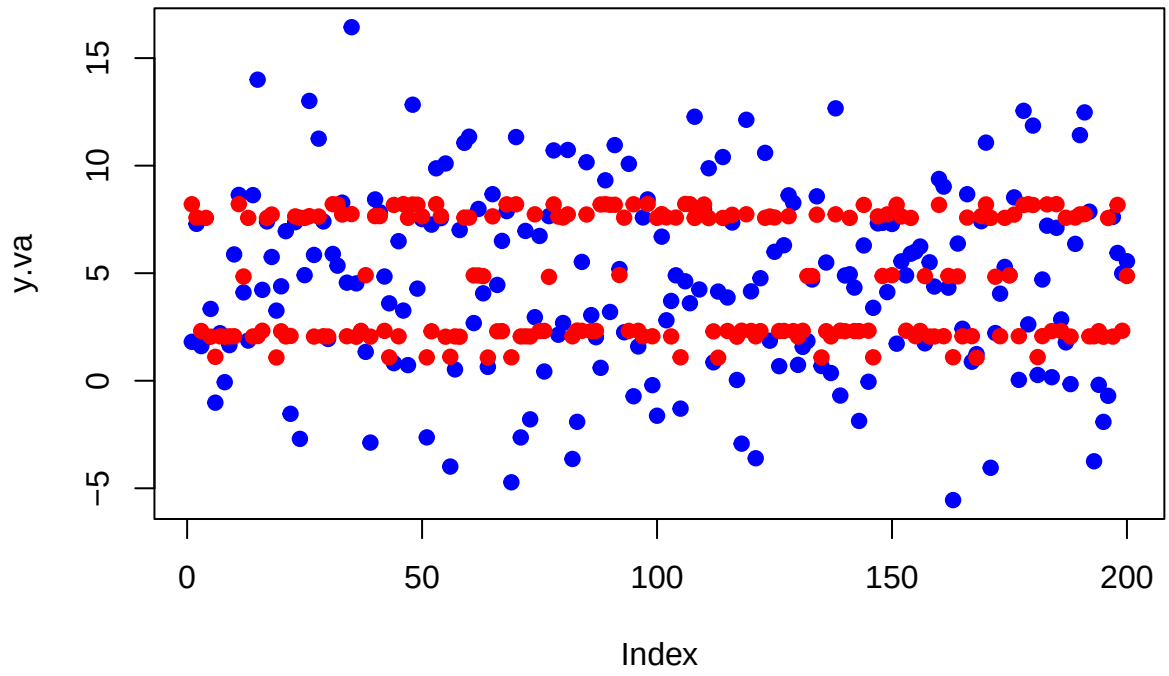
```r
pred_y_mean_in <- apply(cv.bart$prediction, 2, mean)

plot(y.tr, type = "p", col = "blue", lwd = 2, ylim = c(-6, 15))
lines(pred_y_mean_in, type = "p", col = "red", lwd = 2)
legend("topleft",c("Posterior Mean","Real Data"), col = c("red","blue"), lty = c(1,1))
```




```r
bart_oo_pred<- function(bart_result, x.va){

  store_size <- length(bart_result$tree)
  xva.scaled <- x.va
  center_y <- bart_result$center_y
  scale_y <- bart_result$scale_y
  for (j in 1:ncol(x.va)) {
     xva.scaled[,j] <- (x.va[,j]-bart_result$center_x[j])/bart_result$scale_x[j]
  }

  pred <- matrix(NA, nrow = store_size, ncol = nrow(x.va))
  for (i in 1:store_size) {
    curr_tree <- bart_result$tree[[i]]
    curr_sig <- bart_result$sigma2_scaled[i]
    all_mu <- curr_tree$info[,"mu"]
    pred_node <- fill_tree_details(curr_tree, xva.scaled)$node
    pred_mean <- all_mu[pred_node]
    pred_scaled <- rnorm(n = nrow(x.va), mean = pred_mean, sd = sqrt(curr_sig))
    pred[i,] <- pred_scaled * scale_y + center_y
```

```
  }

  return(pred)


}
```

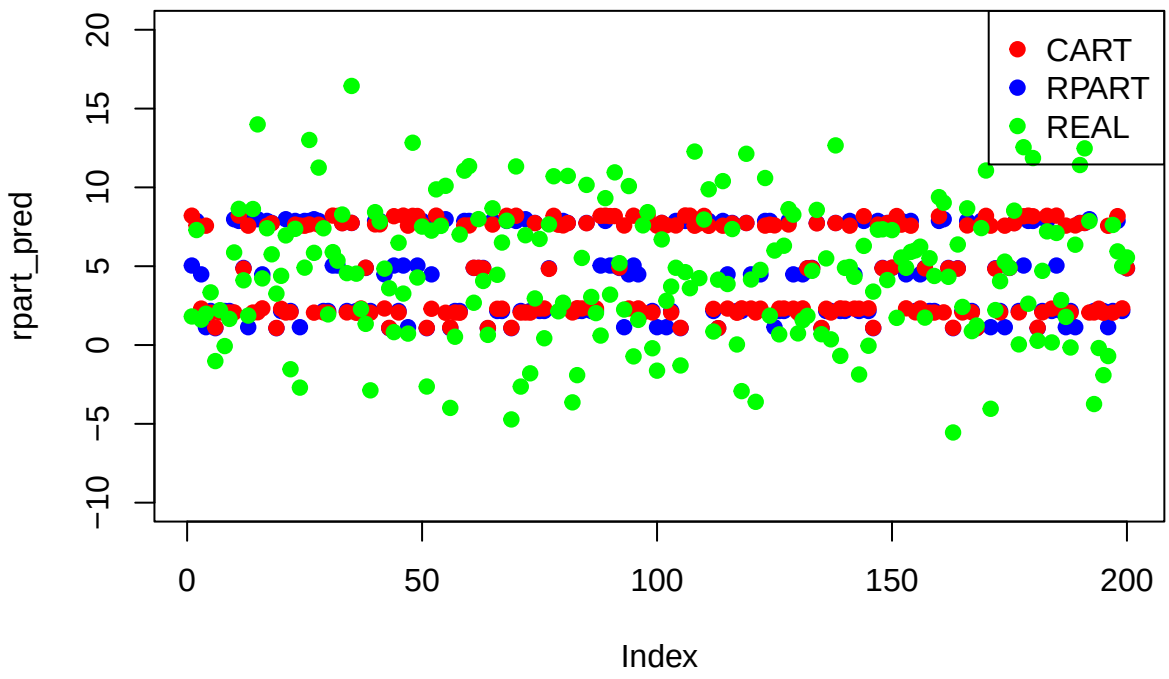
```
oo_pred <- bart_oo_pred(cv.bart, x.va)
pos_oo_mean <- apply(oo_pred, 2, mean)

plot(y.va, col = "blue", pch = 19)
lines(pos_oo_mean, col = "red", pch = 19, type = "p")
```



16. Out of Sample Prediction of Rpart

```
tr_dat <- as.data.frame(cbind(y.tr, x.tr))
colnames(tr_dat) <- c("y","x1","A","B","C","D")
colnames(x.va) <- c("x1","A","B","C","D")
cv_mod <- rpart(y ~., data = tr_dat)
rpart_pred <- predict(cv_mod, newdata = as.data.frame(x.va))
```

```
plot(rpart_pred, type='p', col = "blue", pch = 19, ylim = c(-10,20))
lines(pos_oo_mean, col = "red", pch = 19, type = "p")
lines(y.va, col = "green", pch = 19, type = "p")
legend("topright", c("CART","RPART","REAL"), col = c("red","blue","green"), pch = c(19,19,19))
```




```
pred_err_rpart <- sum((rpart_pred-y.va)^2)
pred_err_cart <- sum((pos_oo_mean-y.va)^2)
paste("pred_err_cart = ",pred_err_cart)
```

```
## [1] "pred_err_cart =  2995.06039971267"
```

```
paste("pred_err_rpart = ",pred_err_rpart)
```

```
## [1] "pred_err_rpart =  1896.91970264028"
```