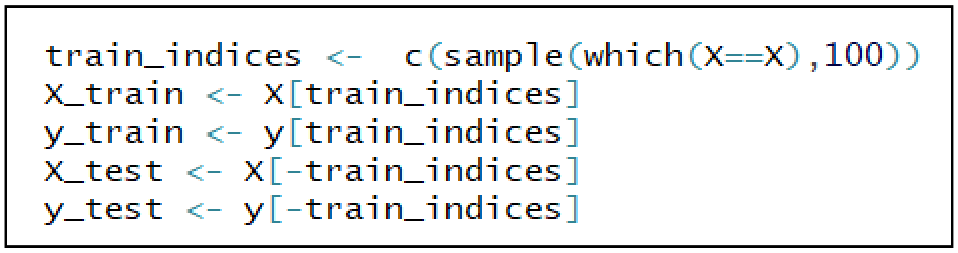
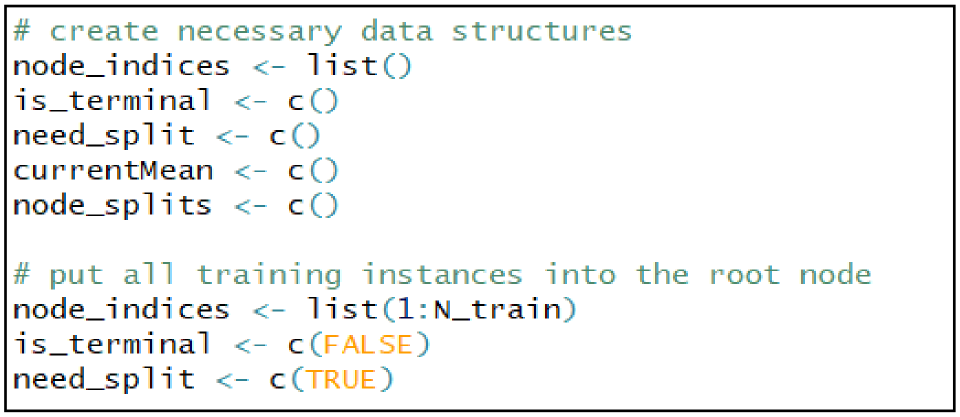
COMP421 HW05, Prepared by Arda Kırkağaç, 49799

In this homework, our task was to calculate a regression tree to estimate a continuous variable y, split between its different x values. This tree differs from a classification tree in terms of impurity calculating and defining a leaf node. We also used a pre-pruning parameter, not allowing the tree to subdivide itself if it has P or fewer data points. The data points were picked using the same procedure, where 100 random data points are chosen as training data out of 133 points, and the remaining is left as test points.

Since we are going to call this tree multiple times, it is better that a decision tree function is defined, whose only parameter is the pre-pruning value. All parameters and data structures are then defined inside this function. This function will return the RMSE value, together with some other arrays which let us plot the decision tree with data points.



```
train_indices <-  c(sample(which(X==X),100))
X_train <- X[train_indices]
y_train <- y[train_indices]
X_test <- X[-train_indices]
y_test <- y[-train_indices]
```

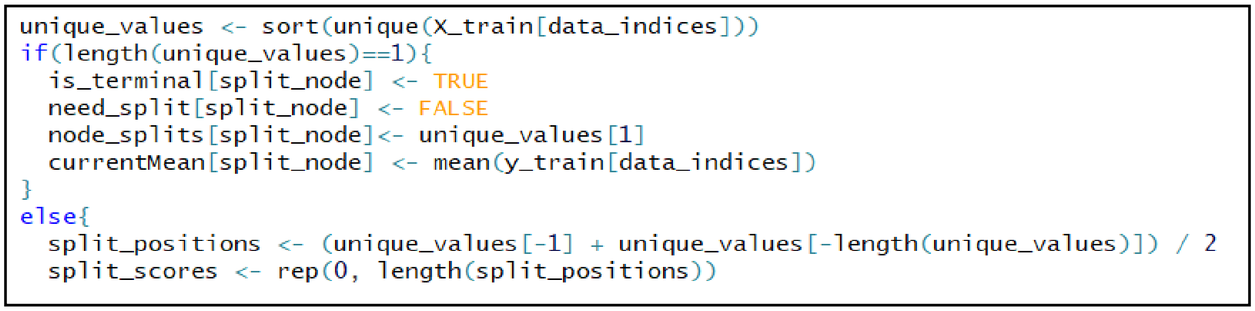


```
# create necessary data structures
node_indices <- list()
is_terminal <- c()
need_split <- c()
currentMean <- c()
node_splits <- c()

# put all training instances into the root node
node_indices <- list(1:N_train)
is_terminal <- c(FALSE)
need_split <- c(TRUE)
```

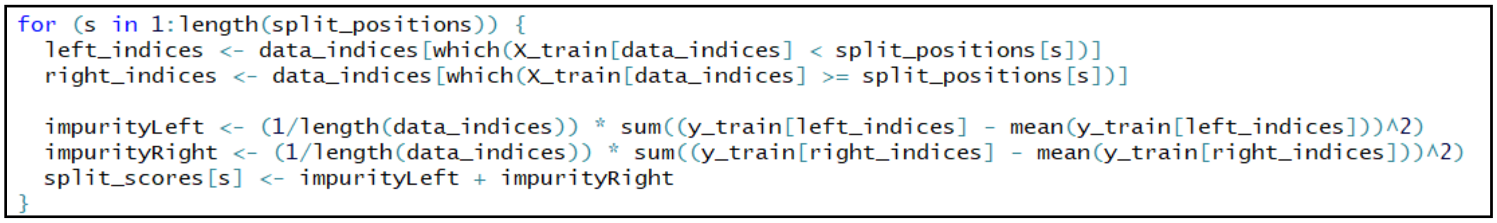
In the second figure above, currentMean() will keep the average of y values of the nodes we work on. The break condition for the iterations are the same, the algorithm will keep working until there is no nodes left to split, determined by pre-pruning parameter.

There were some problems that kept our algorithm in a loop forever. One of those problems was that the tree did not know what to do then all x values are the same for a specific node, and pre-pruning parameter is less than the number of nodes. The algorithm then tries to divide the three by using different x values, but what it actually does is iterating over the same node, since this splitting procedure will not change anything at all. Here is a solution below, which checks for the unique x values in a node to determine whether it is going to try to divide this tree or not:



```
unique_values <- sort(unique(X_train[data_indices]))
if(length(unique_values)==1){
  is_terminal[split_node] <- TRUE
  need_split[split_node] <- FALSE
  node_splits[split_node]<- unique_values[1]
  currentMean[split_node] <- mean(y_train[data_indices])
}
else{
  split_positions <- (unique_values[-1] + unique_values[-length(unique_values)]) / 2
  split_scores <- rep(0, length(split_positions))
```

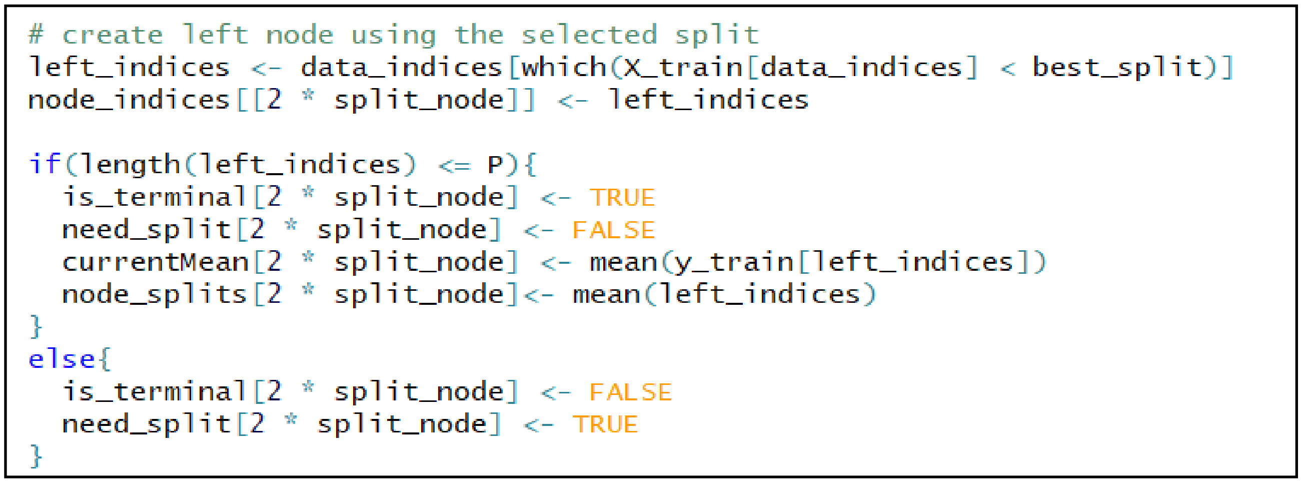
Dimensions over which we are going to split the tree is now 1, since the division is only done over x values of the input data. This makes the algorithm simpler at that point, with the only difference in the impurity calculation. The difference is that we change the classification impurity with the squared error of the y values and their estimate, and try to minimize that value.



```
for (s in 1:length(split_positions)) {
  left_indices <- data_indices[which(X_train[data_indices] < split_positions[s])]
  right_indices <- data_indices[which(X_train[data_indices] >= split_positions[s])]

  impurityLeft <- (1/length(data_indices)) * sum((y_train[left_indices] - mean(y_train[left_indices]))^2)
  impurityRight <- (1/length(data_indices)) * sum((y_train[right_indices] - mean(y_train[right_indices]))^2)
  split_scores[s] <- impurityLeft + impurityRight
}
```

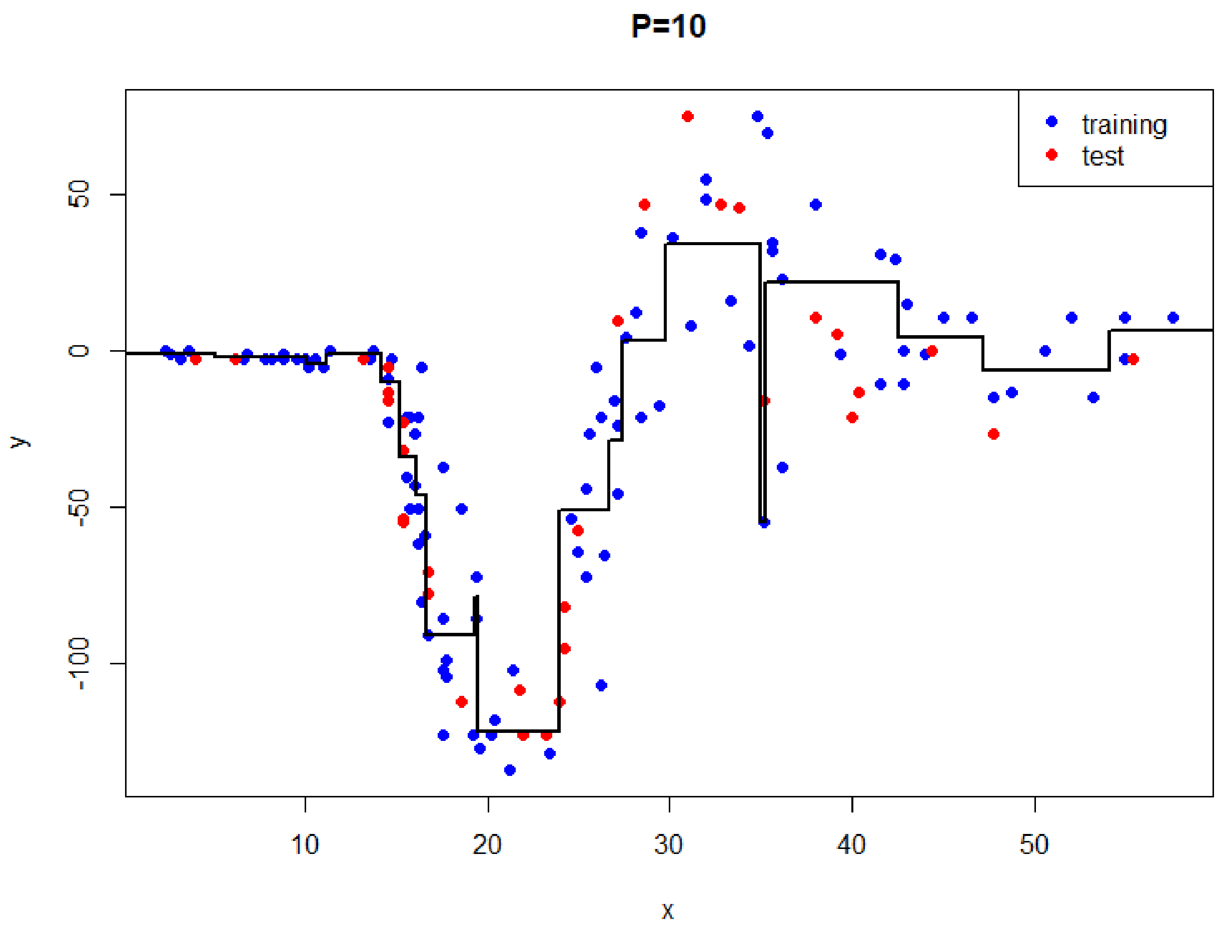
After we find the best split simply as the minimum of all split scores for the given node, the left and right child of the decision tree are created. Pre-pruning parameter comes into play at that point. Below it is shown how the left subtree is formed, and the procedure for the right subtree is the same, except the data points:



```
# create left node using the selected split
left_indices <- data_indices[which(X_train[data_indices] < best_split)]
node_indices[[2 * split_node]] <- left_indices

if(length(left_indices) <= P){
  is_terminal[2 * split_node] <- TRUE
  need_split[2 * split_node] <- FALSE
  currentMean[2 * split_node] <- mean(y_train[left_indices])
  node_splits[2 * split_node]<- mean(left_indices)
}
else{
  is_terminal[2 * split_node] <- FALSE
  need_split[2 * split_node] <- TRUE
}
```

The last thing to do inside the function is to calculate RMSE value for the test points we have, and return is_termina(), RMSE, node_splits() and currentMean(), since they will come in handy while plotting the tree estimate over the data points. This will be done outside the function after calling the output. Here are the results when we call the tree with a pre-pruning parameter of 10:



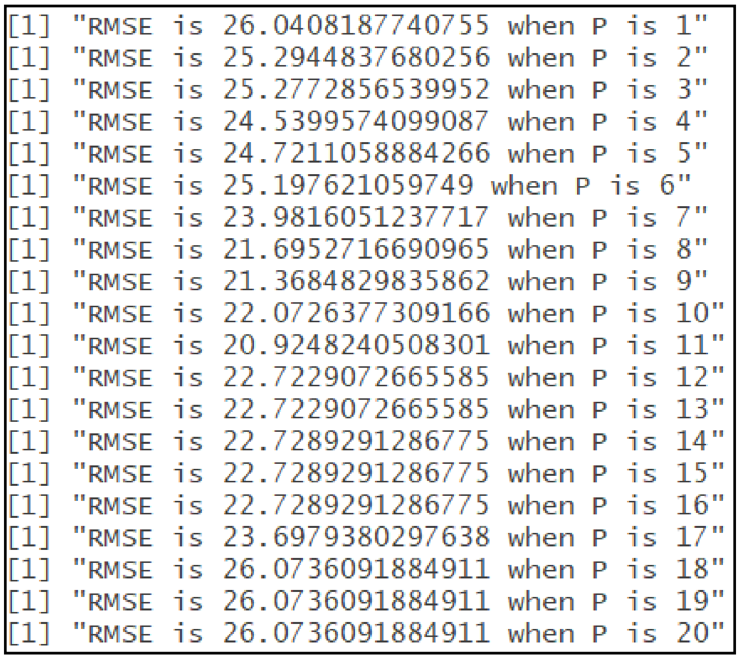
```
[1] "RMSE is 22.0726377309166 when P is 10"
```

Since we defined a function, it is quite simple now to call it for different values of P.

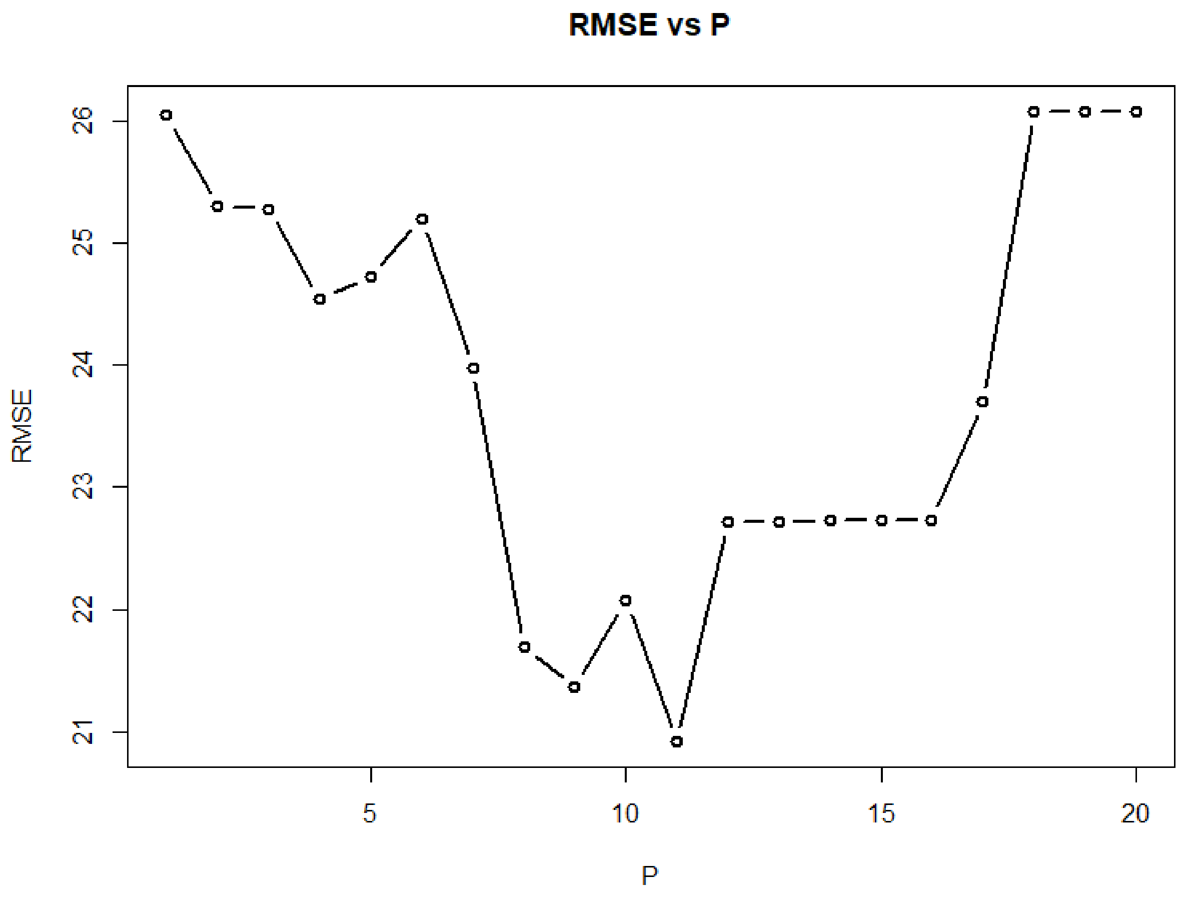
```
RMSE_value <- c()
for (P in 1:20){

  output <- decisionTree(P)
  RMSE_value[P] <- output[[1]]
}
```

Here are the outputs for different P values and the final graph which represents the error values across different parameter values:



```
[1] "RMSE is 26.0408187740755 when P is 1"
[1] "RMSE is 25.2944837680256 when P is 2"
[1] "RMSE is 25.2772856539952 when P is 3"
[1] "RMSE is 24.5399574099087 when P is 4"
[1] "RMSE is 24.7211058884266 when P is 5"
[1] "RMSE is 25.197621059749 when P is 6"
[1] "RMSE is 23.9816051237717 when P is 7"
[1] "RMSE is 21.6952716690965 when P is 8"
[1] "RMSE is 21.3684829835862 when P is 9"
[1] "RMSE is 22.0726377309166 when P is 10"
[1] "RMSE is 20.9248240508301 when P is 11"
[1] "RMSE is 22.7229072665585 when P is 12"
[1] "RMSE is 22.7229072665585 when P is 13"
[1] "RMSE is 22.7289291286775 when P is 14"
[1] "RMSE is 22.7289291286775 when P is 15"
[1] "RMSE is 22.7289291286775 when P is 16"
[1] "RMSE is 23.6979380297638 when P is 17"
[1] "RMSE is 26.0736091884911 when P is 18"
[1] "RMSE is 26.0736091884911 when P is 19"
[1] "RMSE is 26.0736091884911 when P is 20"
```


RMSE vs P
RMSE
P