Machine Learning 2ST129 26605 HT2023 Assignment 2

Anonymous Student

November 19, 2023

Contents

General Information

- Time used for reading: 3 hours
- Time used for basic assignment: 20 hours
- Time used for extra assignment 8 hours
- Good with lab: Very practical. Now i feel like i got a better understanding of how decision trees and randomforests work.
- Things to improve in the lab: Maybe more code in the code sketch or more hints on how to handle odd situations that can arise. I think that roughly half of the time went only to debug things, for example smaller problems like handling ties, to larger problems where for example I could get correct results in the task 1, but the functions would not work for the bootstrapped data sets because the tree_split would just `next` all iterations and you could end up in a infinite while loop or get different errors. Though more hints were added later (like the tie), I just felt that I understood the concept of how to do things but got stuck on debugging things. Also I did not understand the hint in task 3.3 to look at figure 15.1 in the hastie et al; since that plot was about the test error for different number of trees for bagging, RF, and Gradient boosting but the task only ask you to present the RMSE.

1 Task 1: decision trees

```
library(uuml)
library(tidyverse)

 data("Hitters")
```

1.1 1.1 test set

```
# Remove NA values
Hitters <- Hitters[complete.cases(Hitters),]
# Create test and training set
X_test <- as.matrix(Hitters[1:30, c("Years", "Hits")])
y_test <- Hitters[1:30, c("Salary")]
X_train <- as.matrix(Hitters[31:nrow(Hitters), c("Years", "Hits")])
y_train <- Hitters[31:nrow(Hitters), c("Salary")]
```

1.2 1.2 tree split

Now we want to implement a function to split observations binary greedily.

```
SS_func <- function(y_vec, y_bar){
  Sum_square <- sum((y_vec - y_bar)^2)
  return(Sum_square)
}

assertions <- function(X,y,l){
  checkmate::assert_matrix(X)
  checkmate::assert_numeric(y, len = nrow(X))
  checkmate::assert_int(l)
}


#' function to split observations
#' @param X Design Matrix
#' @param y variable
#' @param l leaf size. Minimum 1
#'
```

```
tree_split <- function(X, y, l){
assertions(X=X,y=y,l=l)

  SS <<- matrix(Inf, nrow = nrow(X), ncol = ncol(X))

  for (j in 1:ncol(X)) {
    for (k in 1:nrow(X)) {
      s <- X[k, j]  # Split point
      R1 <- which(X[, j] <= s)
      R2 <- which(X[, j] > s)

      # Handle if R1 or R2 is smaller than the leaf size l
      if (length(R1) < l || length(R2) < l) next
        c1 <- mean(y[R1])
        # Compute c2
        c2 <- mean(y[R2])
        # Compute the SS
        SS[k, j] <- SS_func(y[R1], c1) + SS_func(y[R2],c2)
      }
    }

# The final results for the min SS
 min_SS <- min(SS)
 if(min_SS == Inf) {
   return(NULL)
 } # in case it cannot find any split such that both regions are > l
 #example n=15, l = 5, but it cannot split it into two regions such that both
 # regions have n >5, then it would just next all iterations and SS = inf

 index <- which(SS == min_SS, arr.ind = TRUE)[1,]
 #in case of ties,  index the first row
  k <- index[1]
  j <- index[2] #which column was used for the split-point
  s <- X[k,j]# What value of X was used as the split-point
  R1 <- which(X[,j] <= s)
  R2 <- which(X[,j] > s)

  result <- list(j=j, s = s,
                 R1 = R1, R2 = R2,  SS = min_SS)

  return(result)
  }

X_check <- as.matrix(Hitters[31: 50, c("Years", "Hits")])
y_check <- Hitters[31: 50, c("Salary")]
# These are the names of the players we look at
rownames(Hitters)[31: 50]
```

```
 [1] "-Bob Melvin"       "-BillyJo Robidoux" "-Bill Schroeder"
 [4] "-Chris Bando"      "-Chris Brown"      "-Carmen Castillo"
 [7] "-Chili Davis"      "-Carlton Fisk"     "-Curt Ford"
[10] "-Carney Lansford"  "-Chet Lemon"       "-Candy Maldonado"
[13] "-Carmelo Martinez" "-Craig Reynolds"   "-Cal Ripken"
[16] "-Cory Snyder"      "-Chris Speier"     "-Curt Wilkerson"
[19] "-Dave Anderson"    "-Don Baylor"
```

```
tree_split(X_check, y_check, l = 5)
```

```
$j
```

```
col
  1

$s
[1] 5

$R1
      -Bob Melvin -BillyJo Robidoux   -Bill Schroeder      -Chris Brown
                1                 2                 3                 5
 -Carmen Castillo        -Curt Ford -Carmelo Martinez      -Cory Snyder
                6                 9                13                16
  -Curt Wilkerson    -Dave Anderson
               18                19

$R2
    -Chris Bando     -Chili Davis    -Carlton Fisk -Carney Lansford
               4                7                8               10
     -Chet Lemon -Candy Maldonado  -Craig Reynolds      -Cal Ripken
              11               12               14               15
   -Chris Speier      -Don Baylor
              17               20

$SS
[1] 1346633
```

```
tree_split(X_check, y_check, l = 1)
```

```
$j
col
  2

$s
[1] 132

$R1
      -Bob Melvin -BillyJo Robidoux   -Bill Schroeder      -Chris Bando
                1                 2                 3                 4
     -Chris Brown  -Carmen Castillo     -Carlton Fisk        -Curt Ford
                5                 6                 8                 9
      -Chet Lemon  -Candy Maldonado -Carmelo Martinez   -Craig Reynolds
               11                12                13                14
     -Cory Snyder     -Chris Speier   -Curt Wilkerson    -Dave Anderson
               16                17                18                19

$R2
    -Chili Davis -Carney Lansford      -Cal Ripken      -Don Baylor
               7               10               15               20

$SS
[1] 904383.4
```

1.3 1.3-1.4

Based on the whole training set, the results are:

```
first_train_split <-tree_split(X_train, y_train, l = 5)
first_train_split$j
```

```
col
  1
```

```
first_train_split$s

[1] 4

first_train_split$SS

[1] 38464163
```

The split is for col 1, or the variable Years with the value of 4. The sum of squares is 38464163

1.4 1.5 grow tree

Now we want to use the function `tree_split()` to create a function `grow_tree()`. This is done by first creating two helper functions `grow_node` which uses the `tree_split` function to grow a node and return the results in a list, as well as the function `gamma_leaf` which computes the γ-value and returns the results in a list as well. This is just do make the `grow_tree()` function more readable.

```
#' function to grow the tree nodes , for the first if statement in grow_tree
grow_node <- function(X, y, l, S_m, R_i ){
     new_split <- tree_split(X[S_m[[1]],, drop=FALSE], y[S_m[[1]]], l)
     if(is.null(new_split)){
       return(NULL) #as before, in case it cannot find any splits and
       #the preallocated SS==Inf
       }
      new_results <- data.frame(
        j = new_split$j,
        s = new_split$s,
        R1_i = R_i,
        R2_i = R_i+1,
        gamma = NA
      )
      return(list(new_results, new_split$R1, new_split$R2))
}
#' function to calculate the gamma value. The else statement in grow_tree
gamma_leaf <- function(y, S_m){
  gamma <- mean(y[S_m[[1]]])

  new_results <- data.frame(
        j = NA,
        s = NA,
        R1_i = NA,
        R2_i = NA,
        gamma = gamma
      )
  return(new_results)
}

grow_tree <- function(X, y, l) {
  assertions(X = X, y = y, l = l)

  # Initialize the tree with the first split
  init <- tree_split(X, y, l)

  # Initialize S_m to store the set of observation indices
  S_m <- list(init$R1, init$R2)
  R_i <- 2
  # Initialize results data frame
  results <- data.frame(j = init$j, s = init$s,
                      R1_i = R_i, R2_i = R_i+1, gamma = NA)
  # Main loop to grow the tree
  while (length(S_m) > 0) {
    # As long as not all parts of the tree have been handled,
    # we will either split or compute gamma
    if (length(S_m[[1]]) >= 2 * l) {
      R_i <- R_i + 2
      new_split <- grow_node(X=X, y=y, l=l,
                             S_m = S_m, R_i = R_i)


    if(is.null(new_split)){ #this is added in case S_m[[1]]) >= 2 * l
     # but the grow_node (tree split() cannot make any split of the observations
     # such that both regions > l. Then we treat it as a leaf)
    leaf <- gamma_leaf(y=y, S_m = S_m)
    results <- rbind(results, leaf)
    R_i <- R_i - 2
    }

    else{
       results <- rbind(results,new_split[[1]])
         S_m <- c(S_m, list(S_m[[1]][new_split[[2]]], S_m[[1]][new_split[[3]]]))
    }
  # Add R1 and R2 to S_m
    #the long indexes is to make sure we index on the correct value for the whole
   #data X, not the ones used for the current split only

    S_m <- S_m[-1]

    } else {
    # Compute gamma for leaf node
    leaf <- gamma_leaf(y=y, S_m = S_m)
    results <- rbind(results, leaf)
    S_m <- S_m[-1]
    }
  }
  rownames(results) <- 1:nrow(results)
  return(results)
}
```

```
 tr <- grow_tree(as.matrix(X_check), y_check, l= 5)
print(tr)
```

```
   j   s R1_i R2_i    gamma
1  1   5    2    3       NA
2  1   3    4    5       NA
3  2 101    6    7       NA
4 NA  NA   NA   NA 106.5000
5 NA  NA   NA   NA 244.5000
6 NA  NA   NA   NA 509.3334
7 NA  NA   NA   NA 946.0000
```

```
  grow_tree(as.matrix(X_check), y_check, l= 1)
```

```
    j   s R1_i R2_i    gamma
1   2 132    2    3       NA
2   1   5    4    5       NA
3   2 146    6    7       NA
4   1   2    8    9       NA
5   1  16   10   11       NA
6   1   6   12   13       NA
```

```
7    1   6   14   15        NA
8    2  53   16   17        NA
9    2  56   18   19        NA
10   2  78   20   21        NA
11 NA  NA   NA   NA  875.000
12 NA  NA   NA   NA  815.000
13 NA  NA   NA   NA  950.000
14 NA  NA   NA   NA 1350.000
15 NA  NA   NA   NA 1200.000
16   2  41   22   23        NA
17   1   1   24   25        NA
18   2  46   26   27        NA
19   1   3   28   29        NA
20   2  68   30   31        NA
21   1   6   32   33        NA
22 NA  NA   NA   NA   67.500
23 NA  NA   NA   NA   70.000
24 NA  NA   NA   NA   90.000
25 NA  NA   NA   NA   90.000
26 NA  NA   NA   NA  180.000
27   2  53   34   35        NA
28 NA  NA   NA   NA  215.000
29   1   4   36   37        NA
30   1   6   38   39        NA
31 NA  NA   NA   NA  416.667
32 NA  NA   NA   NA  415.000
33 NA  NA   NA   NA  675.000
34 NA  NA   NA   NA  225.000
35 NA  NA   NA   NA  230.000
36 NA  NA   NA   NA  340.000
37 NA  NA   NA   NA  247.500
38 NA  NA   NA   NA  305.000
39 NA  NA   NA   NA  275.000
```

1.5 1.6

Here we implement a function to predict the value given observations. The first function `check_leaf()` works for a single observation as well as a given value for the row of the whole tree. It works by checking whether the observation is in a leaf containing a gamma value, which corresponds to the predicted value, and then returns `TRUE`. Else it means that the observation is currently in a node and hence it checks which region to assign the observation to next (the new row number). This is done iteratively in the `predict_with_tree()` function such that it iterates over all the different rows until it is assigned to a leaf.

```r
#' function to check whether a single observation is in a leaf or not given the
#' row of the tree to search in.
#' if not, then it returns the next region to search, else returns TRUE
check_leaf <- function(x, tree, n_row){
  new_row <- n_row
  gamma_na <- is.na(tree[n_row,]$gamma)
  if (!gamma_na){
    return( TRUE)
  }

  variable_to_split <- tree$j[n_row]
  value_split <- tree$s[n_row]

  if (x[variable_to_split] <= value_split){
    new_row <- tree$R1_i[n_row]
  }
    else new_row <- tree$R2_i[n_row]
  return(new_row)
}


predict_with_tree <- function(new_data, tree){
  checkmate::assert_matrix(new_data)
  predictions <- numeric(nrow(new_data))

  for(i in 1:length(predictions)){
    row=1
    not_in_leaf <- TRUE

    while(not_in_leaf){
      iter_res <- check_leaf(x = new_data[i,], tree=tree, n_row=row)
      if (iter_res ==TRUE){
        not_in_leaf <- FALSE
        predicted_value <- tree[row,]$gamma
      }
      else row <- iter_res
    }
  predictions[i] <- predicted_value
  }
  return(predictions)
}
```

```r
X_new <- as.matrix(Hitters[51:52, c("Years", "Hits")])

y_new <- Hitters[51:52, c("Salary")]
predict_with_tree(X_new, tr)
```

```
[1] 106.5 244.5
```

```r
rmse(x=c(75, 105, 33), y = c(102, 99, 43))
```

```
[1] 16.98038
```

Now to test compute the root mean square error for the test set predictions based on the full training set tree.

```r
full_train_tree <- grow_tree(X=X_train, y = y_train, l = 5)
test_predictions <- predict_with_tree(X_test, tree= full_train_tree)
rmse(y_test, test_predictions) %>%
  print()
```

```
[1] 322.2891
```

The RMSE is 322

2 Task 2

2.1 2.1

```r
library(randomForest)
data("Hitters")
Hitters <- Hitters[complete.cases(Hitters),]
dat_test <- Hitters[1:30, c("Salary", "Years", "Hits")]
dat_train <- Hitters[31:nrow(Hitters), c("Salary", "Years", "Hits")]
Hitters.rf <- randomForest(Salary ~ Years + Hits, data=dat_train, mtry=1)
```

2.2 2.2

Looking at the fitted random forest model gives the following:

```
Hitters.rf


Call:
 randomForest(formula = Salary ~ Years + Hits, data = dat_train,      mtry = 1)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 1

          Mean of squared residuals: 141924.5
                    % Var explained: 34.67
```

The number of variables used at teach split is 1. This is because the default setting is approximately $p/3$ in the regression setting and where p is the number of variables. If we want we could just change it by setting the argument `mtry` to something else. Or well, we do not really have that much of a choice due to our limited number of variables, but we could do like this if we wanted to use 2 variables per split instead.

```
randomForest(Salary ~ Years + Hits, data=dat_train, mtry=2)


Call:
 randomForest(formula = Salary ~ Years + Hits, data = dat_train,      mtry = 2)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 2

          Mean of squared residuals: 154941.7
                    % Var explained: 28.68
```

2.3 2.3

Now we want to predict on the test set

```
RF_preds <- predict(Hitters.rf, newdata=dat_test)
rmse(x = dat_test$Salary, y = RF_preds)

[1] 238.1639
```

Hence the RMSE is 234.6

2.4 2.4

Now we use xgboost to fit a boosted regression tree.

```
library(xgboost)
X_test <- dat_test[, c("Years", "Hits")]
y_test <- dat_test[, c("Salary")]
X_train <- dat_train[, c("Years", "Hits")]
y_train <- dat_train[, c("Salary")]
xgb <- xgboost(as.matrix(X_train), as.matrix(y_train), nrounds = 200)
```

```
xgb_preds <- predict(xgb, newdata=as.matrix(X_test))
rmse(x= y_test, y = xgb_preds) %>%
  print()

[1] 227.5978
```

This yields a slightly lower root mean square error of 227.6

3 3

3.1 Bagged trees

3.1.1 3.1.1

The way this is implemented here is through a series of functions. First we create a `sample_func()` which just samples new observations from given data X and y and returns a data frame. Then a `bootstrap_func()` which takes an additional parameter B which creates B bootstrapped samples and returns a list, each element a data frame with new observations. Then a function `grow_tree_on_boot` which is just a wrapper for the previously defined function `grow_tree()` but made simpler to use with a data frame instead. Then `train_bagged_trees()` which just uses those functions and applies it more general so that it returns a list of bagged trees that have been trained on the bootstrapped samples.

```r
X_test <- as.matrix(Hitters[1:30, c("Years", "Hits")])
y_test <- Hitters[1:30, c("Salary")]
X_train <- as.matrix(Hitters[31:nrow(Hitters), c("Years", "Hits")])
y_train <- Hitters[31:nrow(Hitters), c("Salary")]
```

```r
#'function to sample new data given covariates X and response y
#' @return Note: it returns a dataframe
sample_func <- function(X, y) {
  n <- length(y)
  sampled_rows <- sample(1:n, size = n, replace = TRUE)
  new_X <- X[sampled_rows, ]
  new_y <- y[sampled_rows]

  # Create a tibble with the same column names as X
  new_X <- as_tibble(new_X) %>%
    setNames(colnames(X))

  # Combine X and y into a new tibble
  new_df <- bind_cols(tibble(y = new_y), new_X)

  return(new_df)
}

#' function to create B number of bootstrapped sets
#' @param B number of bootstraps
#' @return a list with bootstraped datasets as the elements
bootstrap_func <- function(X,y, B){

  bootstrapped_datasets <- replicate(B, sample_func(X,y), simplify = FALSE)
  #returns the bootstrapped sets in a list
  return(bootstrapped_datasets)
}
####

#just a wrapper function for the grow_tree funcion to specify $y as y argument
#and all other variables as the X argument in matrix, which makes it easier to use
grow_tree_on_boot <- function(dataset, l) {
    grow_tree(X = as.matrix(dataset %>% select(-y)), y = dataset$y, l = l)
}

###

#' function to create bagged trees based on the bootstrapped datasets
#' parameters are the same as before, with l = leaf size
#' @return a list with bagged trees
#'
```

```
train_bagged_trees <- function(X, y, l, B){
  bootstrapped_sets <- bootstrap_func( X=X, y=y, B=B)
   bagged_trees <- lapply(bootstrapped_sets, grow_tree_on_boot, l = l)
  return(bagged_trees)
}
```

3.1.2 3.1.2

Now we create a function `predict_with_bagged_trees()` which is basically a wrapper for the previously `predict_with_tree()` but generalized to work on a list of bagged trees.

Then the `boot_and_predict()` function just combines both the `train_bagged_trees()` and `predict_with_bagged_trees()` functions so that it first creates bagged trees and then predicts in the same function. Then finally, in the `task_3.3_main()` the use is extended to work on multiple values of B.

```
#' @param bagged_trees returned results from grow_tree_on_boot with
#' B number of trees
#' @param new_obs new data to predict on with length K
#' @return mean predicted values of the trees on new observations

predict_with_bagged_trees <- function(bagged_trees, new_obs){

  all_tree_preds <- lapply(bagged_trees, FUN = function(tree){
  predict_with_tree(new_data = new_obs, tree=tree)
})

result_matrix <- do.call(rbind, all_tree_preds)
#matrix where columns are the new observations and rows the predicted value
#from each tree

mean_preds <- colMeans(result_matrix)
return(mean_preds)
}


####

#' function which combines train_bagged_trees and predict
#'  @return RMSE for the predictions
boot_and_predict <- function(X_train,y_train,l, B, X_test, y_test){

  trees <- train_bagged_trees(X=X_train, y=y_train, l=l, B=B)
  preds <- predict_with_bagged_trees(bagged_trees = trees, new_obs=X_test)
  rmse <- uuml::rmse(x= y_test, y= preds)
  return(rmse)
}


task_3.3_main <- function(X_train, y_train, X_test, y_test, l, B){
  res<- sapply(B, FUN = boot_and_predict, X_train=X_train, y_train = y_train,
              X_test = X_test, y_test = y_test, l= l)
  return(res)
}
```

3.1.3 3.1.3

Now that we have the functions the results are:

```
set.seed(111)
boot_results <- task_3.3_main(X_train = X_train, y_train =  y_train,
                              y_test = y_test, X_test = X_test,
              l= 5, B= c(1,10, 100,500, 1000))

boot_results

[1] 308.4493 304.7681 282.7706 275.1588 278.9853
```

Hence looking at the results,the RMSE is around 300 for all different values for B. B=1 has the largest value, whereas B=500 and B = 1000 has the lowest. But the difference is not that much

3.2 3.2 Random Forest

3.2.1 3.2.1

Now to implement a random forest function. We can basically reuse the previous functions but with small tweaks. For the `grow_tree()` function we just add that it should split the variables at random according to how many we set as the `m`-argument. Thus the following lines were added at two parts of the code:

```
vars_to_split <- sample(1:n_cols, size=m, replace=FALSE)
X_subset <- X[,c(vars_to_split), drop=FALSE]
```

and then subsequent changes so that it uses `X_subset` instead of the whole X-set instead. Hence when we call the `tree_split()`, (which is itself wrapped with the `grow_node()` function) inside the `grow_tree()` function, then it will only consider the m randomly selected variables.

```
grow_tree_m <- function(X,y,l, m) {
  assertions(X = X, y = y, l = l)
  n_cols <- ncol(X)
  stopifnot(m <= n_cols)
  #added this for the RF implementation. also inside the while-loop
 vars_to_split <- sample(1:n_cols, size=m, replace=FALSE)
 X_subset <- X[,c(vars_to_split), drop=FALSE]
  # Initialize the tree with the first split
  init <- tree_split(X_subset, y, l)

  # Initialize S_m to store the set of observation indices
  S_m <- list(init$R1, init$R2)
  R_i <- 2
  # Initialize results data frame
  results <- data.frame(j = init$j, s = init$s,
                        R1_i = R_i, R2_i = R_i+1, gamma = NA)
  # Main loop to grow the tree

  while (length(S_m) > 0) {
     vars_to_split <- sample(1:n_cols, size=m, replace=FALSE)
    X_subset <- X[,c(vars_to_split), drop=FALSE]
    # As long as not all parts of the tree have been handled,
    # we will either split or compute gamma
    if (length(S_m[[1]]) >= 2 * l) {
      R_i <- R_i + 2
      new_split <- grow_node(X=X_subset, y=y, l=l,
                             S_m = S_m, R_i = R_i)


    if(is.null(new_split)){ #this is added in case S_m[[1]]) >= 2 * l
     # but the grow_node (tree split() cannot make any split of the observations
     # such that both regions > l. Then we treat it as a leaf)
    leaf <- gamma_leaf(y=y, S_m = S_m)
    results <- rbind(results, leaf)
    R_i <- R_i - 2
    }

    else{
       results <- rbind(results,new_split[[1]])
         S_m <- c(S_m, list(S_m[[1]][new_split[[2]]], S_m[[1]][new_split[[3]]]))
    }
  # Add R1 and R2 to S_m
    #the long indexes is to make sure we index on the correct value for the whole
   #data X, not the ones used for the current split only

    S_m <- S_m[-1]


    } else {
    # Compute gamma for leaf node
    leaf <- gamma_leaf(y=y, S_m = S_m)
    results <- rbind(results, leaf)
    S_m <- S_m[-1]
    }
  }
  rownames(results) <- 1:nrow(results)
  return(results)
}
```

3.2.2 3.2.2

Next we just update the `grow_tree_on_boot()` function to act as a wrapper for `grow_tree_m()` instead. The reason this function exists is basically since the returned results from the bootstrap function is a data frame, so `grow_tree_on_boot()` just splits it into the y and X parts for the `grow_tree_m()` function.

Then inside the `new_randomForest()` function, we just do the same as before. First create some bootstrapped samples with the `bootstrap_func`, then creating bagged trees through the new `grow_tree_m()` (by extension through other functions) and then we can just reuse the `predict_with_bagged_trees()` to get predictions for the `y_test` and then compute the RMSE.

```
#just a wrapper function for the grow_tree_m funcion to specify $y as y argument
#and all other variables as the X argument in matrix, which makes it easier to use
grow_tree_on_boot_m <- function(dataset, l, m) {
    grow_tree_m(X = as.matrix(dataset %>% select(-y)), y = dataset$y, l = l, m=m)
}

new_randomForest <- function(X_train, y_train, X_test, y_test, l, m, B){

    bootstrapped_sets <- bootstrap_func( X=X_train, y=y_train, B=B)

   bagged_trees_m <- lapply(bootstrapped_sets, grow_tree_on_boot_m, l = l, m=m)

    preds <- predict_with_bagged_trees(bagged_trees = bagged_trees_m,
                                       new_obs=X_test)
  rmse <- uuml::rmse(x= y_test, y= preds)

   return(rmse)
}
```

3.2.3 3.2.3

Now we test it with $m = 1$ and $B = 100$ and predict on the test data.

```
set.seed(111)
RF_res_1 <- new_randomForest(X_train=X_train, y_train=y_train,
                    X_test=X_test, y_test = y_test,
                    l=5, m =1, B = 100)
RF_res_1
```

```
[1] 307.4823
```

```
RF_res_2 <- new_randomForest(X_train=X_train, y_train=y_train,
                    X_test=X_test, y_test = y_test,
                    l=5, m =2, B = 100)
RF_res_2
```

```
[1] 253.577
```

Hence, the RMSE for m=1 proved to be slightly worse than previously for the bagged trees with a value of 307. But its still in the same neighborhood since it was around 300 earlier. However, when testing for with m=2 it proved to be better at 253, though still slightly worse than the actual `randomForest()` function.