

Technology Arts Sciences TH Köln

Data-driven Modelling and Optimization

Exercise Random Forests (RF)

Group A – Boston Housing data

Submitted to:

Prof. Dr. Wolfgang Konen

Submitted by:

Gayathri Adhimoolam	11136292
Sunitha Radhakrishnan	11135196
Reeshika Sundram	11134553
Daniel Zito	11136529

Index

Task 2 – Random Forest concepts	3
2.1 Basic Principles	3
2.2 How RF manages to avoid overfitting?	3
2.3 Bootstrapping, OOB Prediction and OOB error	4
2.4 Sample Implementation of the above quantities using IRIS dataset	6
Task 3 – Quantity Importance in Random Forest	9
3.1 Analysing the df2 data frame	10
3.2 Assessing the importance of variables manually	10
3.3 Assessing the importance of variables through random forest	13
Task 4 – Random Forest with Boston Housing data	15
4.1 Introduction	15
4.2 Data Exploration	16
4.3 Random Forest Application	17
4.4 Kaggle Leaderboard Comparison	22

Random Forest

Task 2 – Random Forest concepts

2.1 Basic Principle

Random forest predictors are an ensemble (group of items viewed as a whole) learning approach based on regression or classification trees. Instead of building one classification tree (classifier), the RF algorithm builds multiple classifiers using randomly selected subsets of the observations and random subsets of the predictor variables.

Random Forest is Basically done in 3 Steps:

- Bootstrap Sampling
- Building the Models
- Bootstrap Aggregating

In case of **regression** trees: The predictions from the group of trees are then averaged

In case of **classification** trees: The predictions are tallied using a voting system.

Why Random Forest?

- RF is efficient to support flexible modelling strategies.
- RF can detect and make use of more complex relationships among the variables.
- RF is unexcelled (better than any other example) in accuracy among current algorithms and does not overfit.
- RF can interpolate missing value and maintain high accuracy even when a large proportion of the data are missing.
- RF can handle thousands of input variables without variable exclusion. It runs efficiently on large data bases.
- RF can also handle a spectrum of response types, including categorical, numeric, ratings, and survival data.
- Another advantage of the RF is that it requires only two user- defined parameters (The number of trees and the number of randomly selected predictive variables used to split the nodes) to be defined. These two parameters should be optimized in order to improve predictive accuracy.

2.2 How RF manages to avoid overfitting?

Parameters in Random Forest:

- Number of trees, t
- Size of Sample, n
- No. of variables in each sample, m (Total M and $m < M$)

Bagging is done at two levels:

1. Data Selection
2. Variable Selection
 - There is no pruning involved. So, a single tree highly over fits the data. Low bias and High Variance.
 - The construction of 't' trees ensures that each tree uses a different set of data and different set of variables. Each individual tree will have Low Bias and High Variance.
 - If we take a tree in consideration it will overfit the model.

Taking the average over all trees (ensembling), each fitted to a subset of the original data set, we arrive at one bagged predictor (red line). Clearly, the mean is more Stable and hence does not overfit the model.

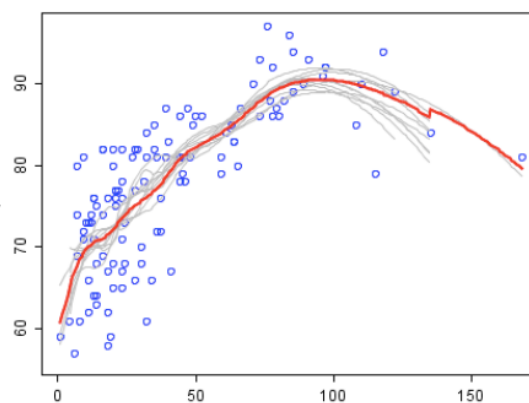


Figure 2.2.1 – Example of less overfit when we take average over all trees

There are also different strategies to manage overfitting of the Decision trees and to get more stabilized model. In the concept of random forest, it was said that the forest error rate depends on two things:

- The correlation between any two trees in the forest. Increasing the correlation increases the forest error rate.
- The strength of each individual tree in the forest. A tree with a low error rate is a strong classifier. Increasing the strength of the individual trees decreases the forest error rate.

Reducing 'm' reduces both the correlation and the strength. Increasing it increases both. Increasing 'n' also increases the correlation among different trees.

The strength of ensemble lies in that all trees are very different, which means low m and low n. (In short: Bagging ensures that all trees are different). However, keeping them very low would mean that every tree is missing on some crucial information. This would increase the error rate.

2.3 Bootstrapping, OOB Prediction and OOB error

When the training set for the current tree is drawn by sampling with replacement, known as **bootstrapping**.

This procedure involves:

1. Choose a number of bootstrap samples to perform
2. Choose a sample size

For each bootstrap sample

- Draw a sample with replacement with the chosen size
- Fit a model on the data sample (in our case, Random Forest)
- Estimate the skill of the model on the out-of-bag sample.
- Calculate the mean of the sample of model skill estimate

OOB Prediction

When the training set for the current tree is drawn by sampling with replacement, about one-third of the cases are left out of the sample. These left out samples are called **OOB (out-of-bag)** data. It is also used to get estimates of variable importance. These out of bag samples act as **OOB predictors**.

The out-of-bag (OOB) error estimate

For each bootstrap sample, there is one third of data which was not used in the creation of the tree, i.e., it was out of the bag sample. This data is referred to as out of bag data. In order to get an unbiased measure of the accuracy of the model over test data, **out-of-bag-error** is used. The out of bag data is passed for each tree is passed through that tree and the outputs are aggregated to give out of bag error. This percentage error is quite effective in estimating the error in the testing set and does not require further cross validation.

In other words,

In random forests, there is no need for cross-validation or a separate test set to get an unbiased estimate of the test set error. It is estimated internally, during the run, as follows:

Each tree is constructed using a different bootstrap sample from the original data. About one-third of the cases are left out (known as the **OOB data**) of the bootstrap sample and not used in the construction of the decision tree.

When these left out samples (Out of bag samples) are tested over Decision trees (**Fit in bag models**) it produces class errors due to the differences in prediction of classes by OOB samples. The mean of these class error gives **Out of Bag Error**. (*This is explained in the figure2 below*)

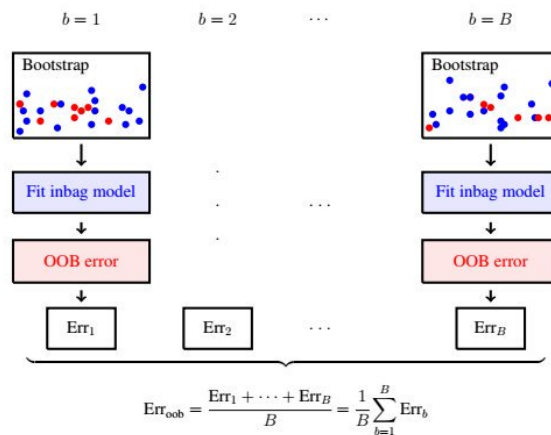


Figure 2.3.1 – OOB error estimates

2.4 Sample Implementation of the above quantities using IRIS dataset

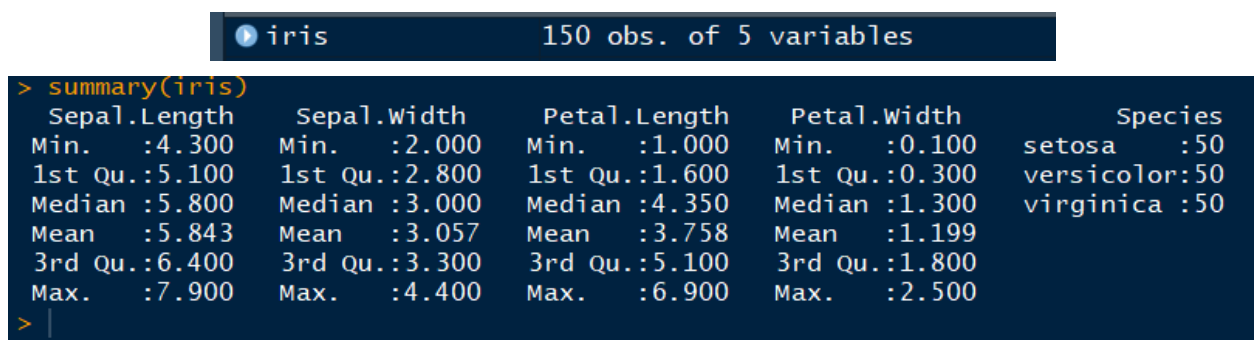


Figure 2.4.1 – Output and summary of Iris data frame

The `summary(iris)` command gives us the summarized information of “iris” dataset. We are predicting Species (response variable in our case) with help of predictors Sepal.Length, Sepal.Width, Petal.length and Petal.width.

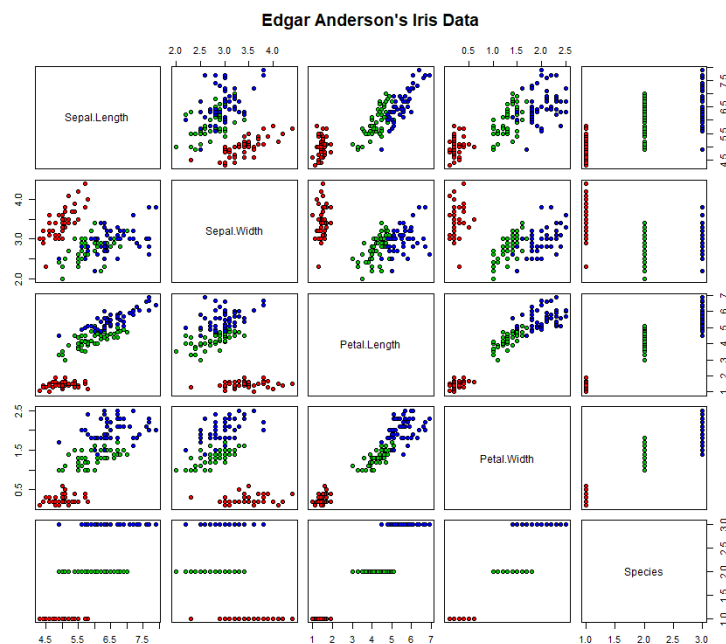


Figure 2.4.2 – Iris pairs scatter plot

From the above plot, we can see that there is a high correlation between the petal length and petal width. Hence the two input variables cannot be neglected when building any model (random forest in our case).

Building random forest model

```
> iris_rf
Call:
randomForest(formula = Species ~ ., data = trainData, ntree = 100,      proximity = TRUE)
  Type of random forest: classification
    Number of trees: 100
No. of variables tried at each split: 2

  OOB estimate of error rate: 6.6%
Confusion matrix:
      setosa versicolor virginica class.error
setosa      34         0         0 0.00000000
versicolor   0        31         3 0.08823529
virginica     0         4        34 0.10526316
>
```

Figure 2.4.2 – Random Forest implementation

Observations:

From the above code and output we can infer that IRIS dataset has

- 150 observations with 5 variables
- Classification type of random forest
- We are predicting species (response variable) with help of predictors sepal length, sepal width, petal length and petal width
- The no, of variable tried at each split is 2, this is also a good splitting of input variable as our total no of variable is 5
- The size of tree we have used is 100 because the tree size should not be larger to avoid overfitting. Hence the OOB error rate is 6.6% in our case.
- Confusion matrix gives us the OOB predictions (class error) for each species. We can see that setosa has 0 error and hence it is a strong classifier.

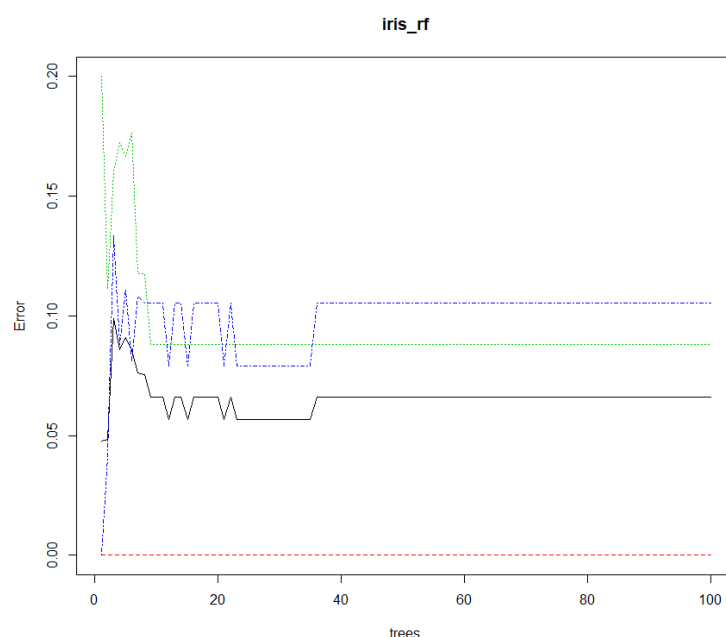


Figure 2.4.3 – Plot of iris_rf model with training data showing the error rate

From the graph above:

Black: Overall OOB error rate

Red: OOB prediction for Setosa

Green: OOB prediction for Versicolor

Blue: OOB prediction for Virginica

- Setosa was correctly predicted by the OOB sample Data Set hence its class error is 0.
- For Versicolor (Green) at t=10 the error stabilises.
- For Virginica (Blue) at t=38 the error stabilises.
- Overall, the OOB Error stabilises at t=40 giving OOB error as 6.6%

Variable Importance Output

Below output shows the percentage mean decrease in Node purity of the input variables in the random forest model of iris data set.

Mean Decrease in Gini is the average (**mean**) of a variable's total **decrease** in node impurity, weighted by the proportion of samples reaching that node in each individual decision tree in the random forest. A **higher Mean Decrease in Gini** indicates **higher** variable **importance**.

```
> importance(iris_rf)
              MeanDecreaseGini
Sepal.Length      7.038479
Sepal.Width       1.618058
Petal.Length     25.960256
Petal.Width      35.357805
```

Figure 2.4.4 – Importance for iris data frame

Hence, petal length and petal width have high variable importance.

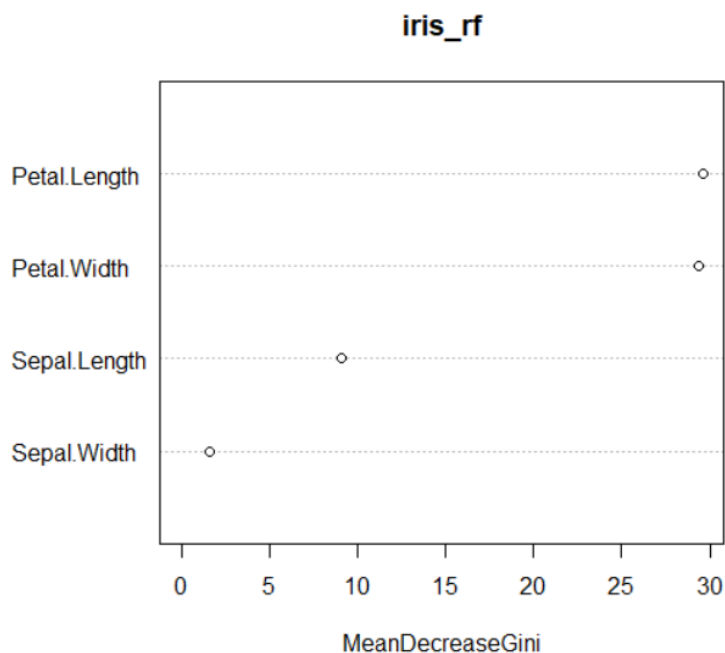


Figure 2.4.5 – Importance for iris data frame plot

Confusion matrix for test data. This shows that there is no class error when taking test data set and predicting over the training data set model, we can conclude that our Random forest model worked well and all the classes are correctly predicted by the test Data.

```
> table(irisPred, testData$Species)

irisPred      setosa versicolor virginica
setosa         16         0         0
versicolor     0         16         0
virginica       0         0         12
> #Try to see the margin, positive or negative, if positive it means correct classification
> plot(margin(iris_rf, testData$Species))
```

Figure 2.4.6 – Confusion matrix for the test set

Task 3 – Quantity Importance in Random Forest

There are two measures of importance given for each variable in the random forest.

- How much the accuracy decreases when the variable is excluded
- Decrease of Gini impurity when a variable is chosen to split a node

Importance for categorical outcomes

- **Accuracy-based importance**

Every tree has its own OOB sample through which the importance of a variable can be calculated. After measuring the prediction accuracy of the OOB sample, the values in it are shuffled randomly, keeping all other variables the same. Finally, the decrease in prediction accuracy on the shuffled data is measured. This importance measure is also broken down by outcome class.

Here, random shuffling means that, on average, the shuffled variable has no predictive power. This importance is a measure of by how much removing a variable decreases accuracy, and vice versa.

- **Gini-based importance**

When a tree is built, the decision about which variable to split uses a calculation of the Gini impurity.

The sum of the Gini decrease across every tree of the forest is accumulated every time that variable is chosen to split a node. For every variable, the sum of the Gini decrease across every tree of the forest is accumulated every time that variable is chosen to split a node. The sum is divided by the number of trees in the forest to give an average.

Importance for numeric outcomes

- Percentage increase in mean square error is analogous to accuracy-based importance and is calculated by shuffling the values of the out-of-bag samples.

- Increase in node purity is analogous to Gini-based importance and is calculated based on the reduction in sum of squared errors whenever a variable is chosen to split.

3.1 Analysing the df2 data frame

For accessing the df2 data frame, we loaded it from the RFimportance2.Rdata.

Analysing the hidden values

The df2 data frame has inputs of xin2, xin3, xin4, xin5 and output out. The information given is that out is the sum of all outputs. So, in order to check the information, we run a function in R-studio to check if the difference between the sum of inputs and the out is zero or not.

The below figure shows the function used to calculate the difference for each row in our observations and is stored in a vector x for future use.

```
difference <- function(df2) {
  x <- numeric() #This will be a vector containing the values of the hidden variables
  for (row in 1:nrow(df2)) {
    diff <- df2[row, ncol(df2)] - sum(df2[row, 1:(ncol(df2) - 1)]) #sum:sums all the input variables in the dataframe
    x <- c(x, diff)
  }
  return(x)
}
```

Figure 3.1.1 – Function for calculating differences

Now, in order to check if the difference is zero or not, the vector x is passed to the function “check”. The function check is given as below:

```
#Function that checks if there are hidden values based on the differences previously calculated
check <- function(x) {
  flag <- FALSE #If there are hidden values this flag turns into TRUE
  if (any(round(x, digits = 10) != 0)) { #We evaluate the difference up to 10 decimal digits to ignore computational approximations
    print("The dataframe contains hidden values")
    return(flag = TRUE)
  } else {
    print("The dataframe DOES NOT contain hidden values")
    return(flag)
  }
}
```

```
> check(x)
[1] "The dataframe contains hidden values"
```

Figure 3.1.2 – Function for checking whether there are hidden variables

So, the data frame contains hidden values. These hidden values or the difference vector x is added to be the new input variable xin1 to our data frame df2.

3.2 Assessing the importance of variables manually

- Box plot

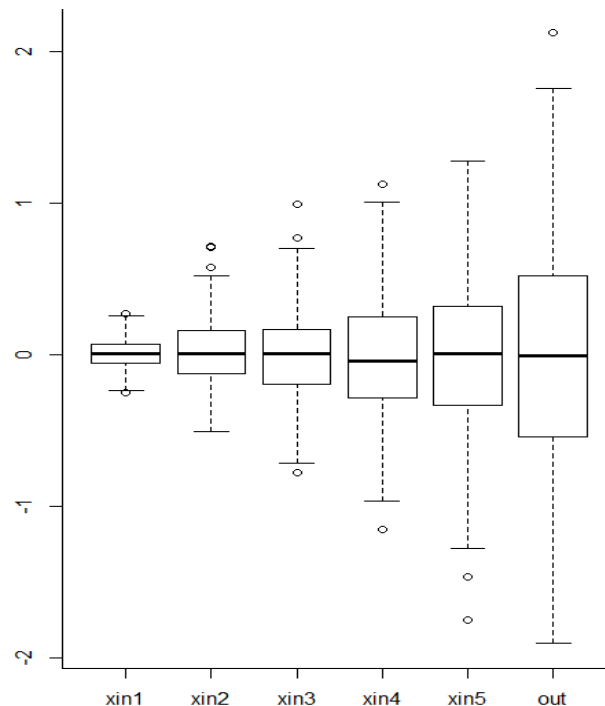


Figure 3.2.1 – df2 Boxplot

From the box plot, the range, the quartiles and the median can be visualized. It can be concluded from the above figure that the importance of input variables can be given as below:

$$\text{xin1} < \text{xin2} < \text{xin3} < \text{xin4} < \text{xin5}$$

- Correlations

The correlation between each input variable and output is shown below:

```
> correlations <- cor(df2, df2$out)
> correlations
      [,1]
xin1 0.1111709
xin2 0.2613448
xin3 0.3819207
xin4 0.5070466
xin5 0.6993189
out  1.0000000
```

Figure 3.2.2 – Correlations between input and output variable

The variable importance of input variables from the correlation table can be written as,

$$\text{xin1} < \text{xin2} < \text{xin3} < \text{xin4} < \text{xin5}$$

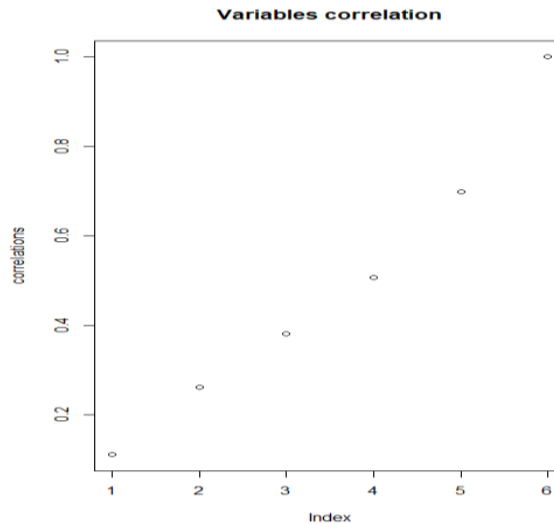


Figure 3.2.3 – Correlations between input and output variable plot

The above figure is a graphical representation of the correlation between input variables versus the output variable.

- **Linear model**

In order to see the impact of each variable in our output data, we fit a linear model for each input variable to the output variable and summarize the result as a table. The function used is shown in the below figure:

```
#The function below calculates t-statistic value, p-value and significance for each single variable with respect to the output
tps_table <- function(df) {
  parameters <- NULL
  for (i in 1:(ncol(df) - 1)) {
    print(i)
    xin <- df[, i] #Copying all the values for the ith variable to be processed in the linear model
    fit_lm <- lm(df$out ~ xin, data = df) #xin iterates from xin1 (or xin2) to xin5
    print(summary(fit_lm))
    t_val <- coef(summary(fit_lm))[-1, "t value"] #Extracting t-statistic values for the ith predictor
    p_val <- coef(summary(fit_lm))[-1, "Pr(>|t|)"] #Extracting p-value for the ith predictor
    significance <- sig_code(p_val) #Based on the p-values returns a significant values in stars code (see function below)
    parameters <- rbind(parameters, cbind(t_val, p_val, significance)) #Combining t-statistic values, p-values and significance in a tab
  }
  return(parameters)
}
```

```
> features_df2
```

	t_val	p_val	significance
[1,]	"1.93107676903519"	"0.0544218603908506"	."
[2,]	"4.67395195113991"	"4.48141147193823e-06"	****
[3,]	"7.13374708809335"	"7.44667600651864e-12"	****
[4,]	"10.1552292666293"	"5.30007246563379e-21"	****
[5,]	"16.8885713043498"	"2.32841829876099e-45"	****

Figure 3.2.4 – Linear model, t_val, p_val and significance

It is seen that the input variable xin5, xin4, xin3 and xin2 has a greater significance. From the t-value, the order of variable importance seems to be the same as inferred from the boxplot and the correlation table. A graphical representation is given below:

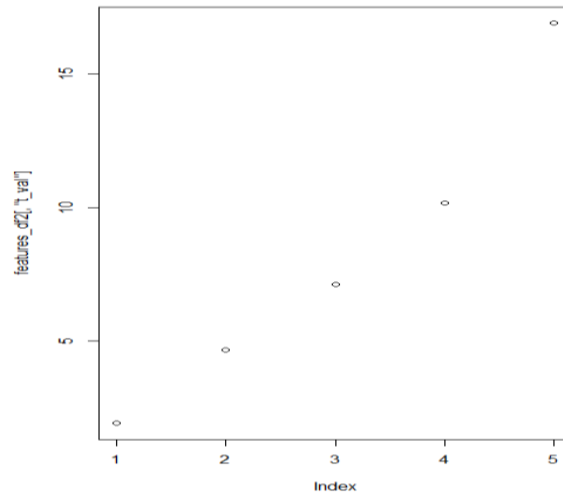


Figure 3.2.5 – Significance plot

3.3 Assessing the importance of variables through random forest

- To start with, a random forest for the original dataset without xin1 is created.

```
> df2_or_RF <- randomForest(df2_original$out ~ ., data = df2_original, importance = TRUE)
> df2_or_RF
```

Call:

```
randomForest(formula = df2_original$out ~ ., data = df2_original, importance = TRUE)
Type of random forest: regression
Number of trees: 500
```

No. of variables tried at each split: 1

```
Mean of squared residuals: 0.0894138
% Var explained: 83.59
```

Figure 3.3.1 – Random Forest implementation

The importance of variables as per the original dataset is given below

```
> importance(df2_or_RF)
      %IncMSE  IncNodePurity
xin2 23.82616      24.25825
xin3 33.06664      29.88252
xin4 56.43444      41.08701
xin5 62.89758      60.52779
```

Figure 3.3.2 – Variable importance

The above table gives the percentage increase in MSE and the Increase in Node purity of the input variables in the original dataset. It can be observed that the importance of variables can be given as $xin2 < xin3 < xin4 < xin5$.

A graphical representation of the above table is shown in the below figure

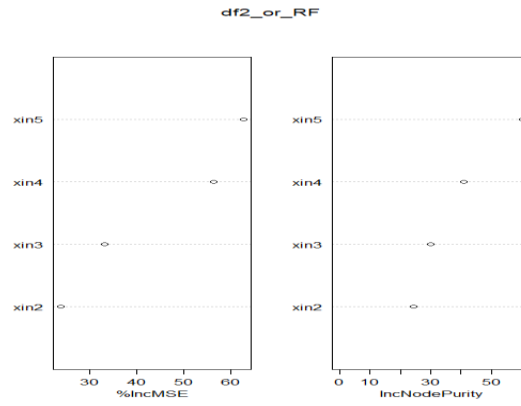


Figure 3.3.3 – Variable importance plot

- Now, a random forest for the dataset with xin1 is created.

```
> df2_RF <- randomForest(df2$out ~ ., data = df2, importance = TRUE)
> df2_RF

Call:
randomForest(formula = df2$out ~ ., data = df2, importance = TRUE)
Type of random forest: regression
Number of trees: 500
No. of variables tried at each split: 1

Mean of squared residuals: 0.1112712
% Var explained: 79.57
```

Figure 3.3.4 – Random Forest implementation

The importance of variables as per the new dataset is given below

```
> importance(df2_RF)
      %IncMSE IncNodePurity
xin1  3.265012    15.30535
xin2 20.220595    21.57691
xin3 30.043610    26.63347
xin4 46.053889    35.90279
xin5 53.415902    54.99198
```

Figure 3.3.5 – Variable importance

The above table gives the percentage increase in MSE and the Increase in Node purity of the input variables in the new dataset. It can be observed that the importance of variables can be given as $xin1 < xin2 < xin3 < xin4 < xin5$.

A graphical representation of the above table is shown in the below figure

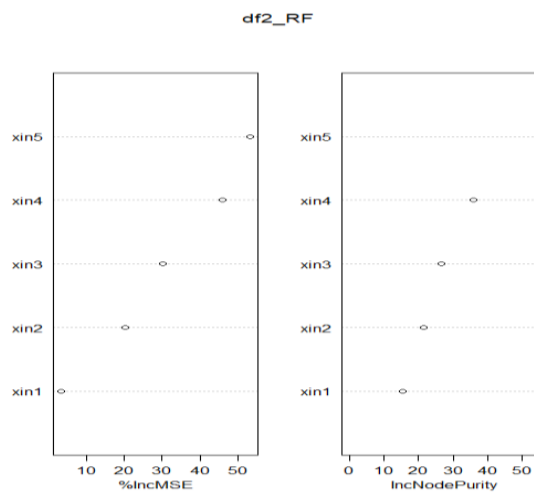


Figure 3.3.6 – Variable importance plot

- From the above two tables, it can be observed that the importance of the input variables `xin2`, `xin3`, `xin4`, `xin5` has decreased when `xin1` was added. So, when `xin1` is excluded, the random forest seems to be better.

Conclusion for the comparison between manual and random forest analysis

The predictions made manually and the actual calculations through random forest for the variable importance seems to be matching.

Task 4 – Random Forest with Boston Housing data

4.1 Introduction

Boston Housing Price dataset consists of *506 observations* of *14 attributes*. The **median value of house price** in \$1000s, denoted by **medv**, is the outcome or the *response* in our model. Below is a brief description of each *predictor* and the outcome in our dataset:

- | | |
|--------------------|--|
| 1. crim | - per capita crime rate by town |
| 2. zn | - proportion of residential land zoned for lots over 25,000 sq.ft |
| 3. indus | - proportion of non-retail business acres per town |
| 4. chas | - Charles River dummy variable (1 if tract bounds river; else 0) |
| 5. nox | - nitric oxides concentration (parts per 10 million) |
| 6. rm | - average number of rooms per dwelling |
| 7. age | - proportion of owner-occupied units built prior to 1940 |
| 8. dis | - weighted distances to five Boston employment centres |
| 9. rad | - index of accessibility to radial highways |
| 10. tax | - full-value property-tax rate per \$10.000 |
| 11. ptratio | - pupil-teacher ratio by town |
| 12. black | - $1000(B_k - 0.63)^2$ where B_k is the proportion of blacks by town |
| 13. lstat | - % lower status of the population |
| 14. medv | - Median value of owner-occupied homes in \$1000's (<i>outcome</i>) |

4.2 Data Exploration

We now explore the *structure* of our data frame. This is done in *R* with the function `str()`. Below, the output compactly provides the relevant information of our data frame, such as number of observations, number of variables, names of each column, class of each column, and sample values from each column.

```
> str(Boston)
'data.frame': 506 obs. of 14 variables:
 $ crim : num 0.00632 0.02731 0.02729 0.03237 0.06905 ...
 $ zn : num 18 0 0 0 0 0 12.5 12.5 12.5 ...
 $ indus : num 2.31 7.07 7.07 2.18 2.18 2.18 7.87 7.87 7.87 ...
 $ chas : int 0 0 0 0 0 0 0 0 0 ...
 $ nox : num 0.538 0.469 0.469 0.458 0.458 0.458 0.524 0.524 0.524 ...
 $ rm : num 6.58 6.42 7.18 7 7.15 ...
 $ age : num 65.2 78.9 61.1 45.8 54.2 58.7 66.6 96.1 100 85.9 ...
 $ dis : num 4.09 4.97 4.97 6.06 6.06 ...
 $ rad : int 1 2 2 3 3 3 5 5 5 ...
 $ tax : num 296 242 242 222 222 222 311 311 311 311 ...
 $ ptratio: num 15.3 17.8 17.8 18.7 18.7 18.7 15.2 15.2 15.2 15.2 ...
 $ black : num 397 397 393 395 397 ...
 $ lstat : num 4.98 9.14 4.03 2.94 5.33 ...
 $ medv : num 24 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 ...
```

Figure 4.2.1 – Structure of Boston dataset

To get more detailed statistical information from each column, `summary()` function can be used. It displays the summary statistics like minimum value, maximum value, median, mean, and the 1st and 3rd quartile values for each column in our dataset. *Summary* also provides information about the missing values, if any is present. We see that there are no missing values in any of our variables.

```
> summary(Boston) #No missing values in any of our variables
      crim      zn      indus      chas      nox      rm      age
Min.   : 0.00632 Min.   : 0.00 Min.   : 0.46 Min.   :0.00000 Min.   :0.3850 Min.   :3.561 Min.   : 2.90
1st Qu.: 0.08204 1st Qu.: 0.00 1st Qu.: 5.19 1st Qu.:0.00000 1st Qu.:0.4490 1st Qu.:5.886 1st Qu.: 45.02
Median : 0.25651 Median : 0.00 Median : 9.69 Median :0.00000 Median :0.5380 Median :6.208 Median : 77.50
Mean   : 3.61352 Mean   :11.36 Mean  :11.14 Mean   :0.06917 Mean   :0.5547 Mean   :6.285 Mean   : 68.57
3rd Qu.: 3.67708 3rd Qu.:12.50 3rd Qu.:18.10 3rd Qu.:0.00000 3rd Qu.:0.6240 3rd Qu.:6.623 3rd Qu.: 94.08
Max.   :88.97620 Max.   :100.00 Max.   :27.74 Max.   :1.00000 Max.   :0.8710 Max.   :8.780 Max.   :100.00

      dis      rad      tax      ptratio      black      lstat      medv
Min.   : 1.130 Min.   : 1.000 Min.   :187.0 Min.   :12.60 Min.   : 0.32 Min.   : 1.73 Min.   : 5.00
1st Qu.: 2.100 1st Qu.: 4.000 1st Qu.:279.0 1st Qu.:17.40 1st Qu.:375.38 1st Qu.: 6.95 1st Qu.:17.02
Median : 3.207 Median : 5.000 Median :330.0 Median :19.05 Median :391.44 Median :11.36 Median :21.20
Mean   : 3.795 Mean   : 9.549 Mean  :408.2 Mean   :18.46 Mean   :356.67 Mean   :12.65 Mean   :22.53
3rd Qu.: 5.188 3rd Qu.:24.000 3rd Qu.:666.0 3rd Qu.:20.20 3rd Qu.:396.23 3rd Qu.:16.95 3rd Qu.:25.00
Max.   :12.127 Max.   :24.000 Max.   :711.0 Max.   :22.00 Max.   :396.90 Max.   :37.97 Max.   :50.00
```

Figure x – Summary of Boston dataset

For a better understanding of our response variable **medv**, let us show its *distribution* and its *density*. In the graph below, the area in red represents the distribution, the bars of the histogram the percentage a same value is present in our dataset and the red dash line the *median* value of the variable.

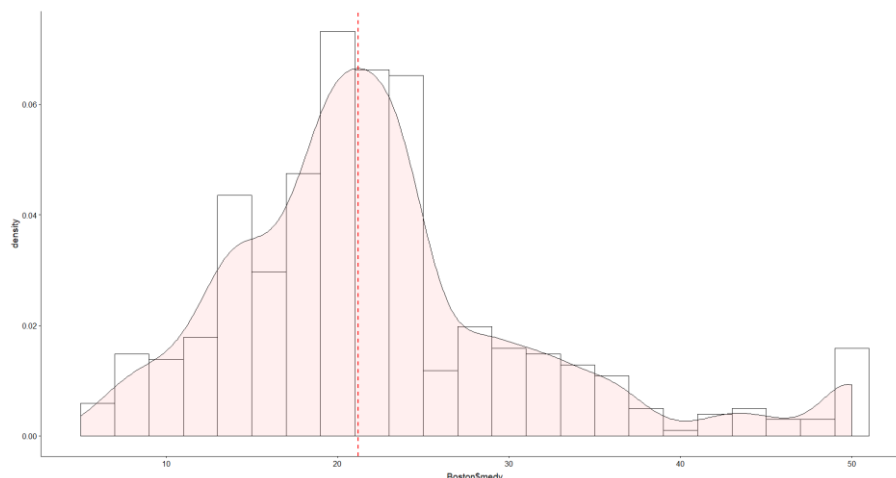


Figure 4.2.2 – Distribution (histogram) and density (red area) of medv variable

As shown in the graph above, the median value of Boston Housing Price is skewed to the right, with several outliers to the right (boxplot below).

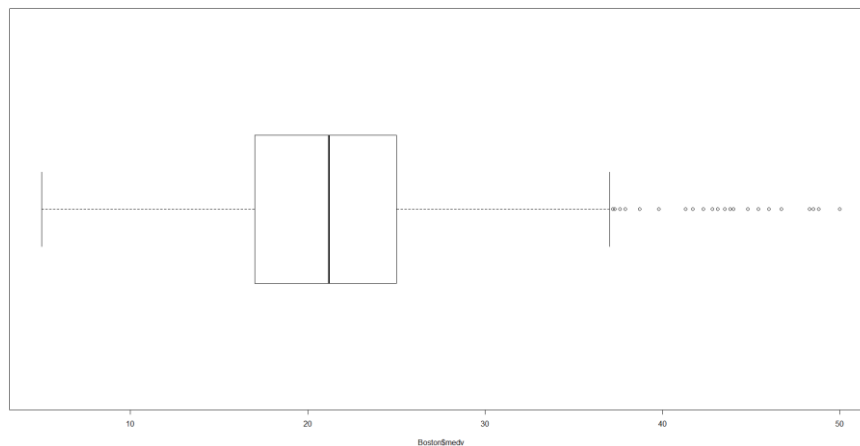


Figure 4.2.3 – Boxplot of medv variable

We now check the correlation of the input variables with the output variable. This gives us an idea about the impact a predictor in our dataset has on the response variable. This is done in *R* with the function `cor()`.

```
> correlations <- cor(Boston, Boston$medv)
> correlations
      [,1]
crim  -0.3883046
zn     0.3604453
indus  -0.4837252
chas   0.1752602
nox    -0.4273208
rm     0.6953599
age    -0.3769546
dis     0.2499287
rad    -0.3816262
tax    -0.4685359
ptratio -0.5077867
black   0.3334608
lstat  -0.7376627
medv   1.0000000
```

Figure 4.2.4 – Correlations of predictors with response variable

As it is shown in the picture above, the predictor that shows the strongest positive correlation with the medv is **rm** (0.6953599), while **lstat** shows the strongest negative correlation (-0.7376627). The predictor with the least correlation with medv is **chas** (0.1752602).

4.3 Random Forest Application

In this chapter we show a Random Forest algorithm application for the given data frame, Boston Housing Price. This is a *regression* problem and the quantity to optimize is the *RMSE*.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$$RMSE = \sqrt{MSE}$$

Where:

- n is the number of data points (observations)
- y_i represents the observed value
- \hat{y}_i represents the predicted value

To *train*, *tune* and *test* our algorithm we split the whole dataset (*506 observations*) in three different subsets: **train** (70% ca.*), **validation** (15% ca.*) and **test** (15% ca.*).

These percentages refer to the sample probability. In our case **train** contains *343 observations*, **validation** *84 observations* and **test** *79 observations* (`set.seed(6)`).

The validation set is used for parameter tuning since tuning our algorithm directly on the test set is not a good practice, it may result in *overfitting*.

```
#To train, tune and test our method, we split the dataset in three subsets. train (70%), validation (15%) and test (15%) set
split_index <- sample(1:3, size = nrow(Boston), prob = c(0.70, 0.15, 0.15), replace = TRUE)
train <- Boston[split_index == 1, ] #Training dataset
validation <- Boston[split_index == 2, ] #Validation dataset (used for parameter tuning)
test <- Boston[split_index == 3, ] #Testing dataset
```

Figure 4.3.1 – Splitting train, validation and test set in R

We now proceed by fitting the Random Forest model on the train set. We then calculate the RMSE by predicting the values of the test set.

For this first part no tuning is done. Hence, our algorithm runs with standard values such as `ntree = 500` and `mtry = 4`. Where, `ntree` is the number of trees (size of the forest) and `mtry` is the number of variables (integer) tried at each split, by default for regression problem its value is $m/3$, where m is the number of variables in the dataset (*14* in our case).

Since our goal is to predict *medv*, the `randomForest()` is written as follows:

```
fit_rf <- randomForest(formula = medv ~ ., data = train)
```

The output of the function is shown below.

```
> fit_rf <- randomForest(formula = medv ~ ., data = train)
> fit_rf

Call:
randomForest(formula = medv ~ ., data = train)
Type of random forest: regression
Number of trees: 500
No. of variables tried at each split: 4

Mean of squared residuals: 11.81814
% Var explained: 84.65
```

Figure 4.3.2 – Random Forest output

As shown in the picture: Type of random forest: regression, Number of trees: *500*, No. of variable tried at each split: *4*.

Moreover, Mean of squared residuals: *11.81814* and % Var explained: *84.65*.

Below a plot of Mean of squared residuals against random forest size (no. trees).

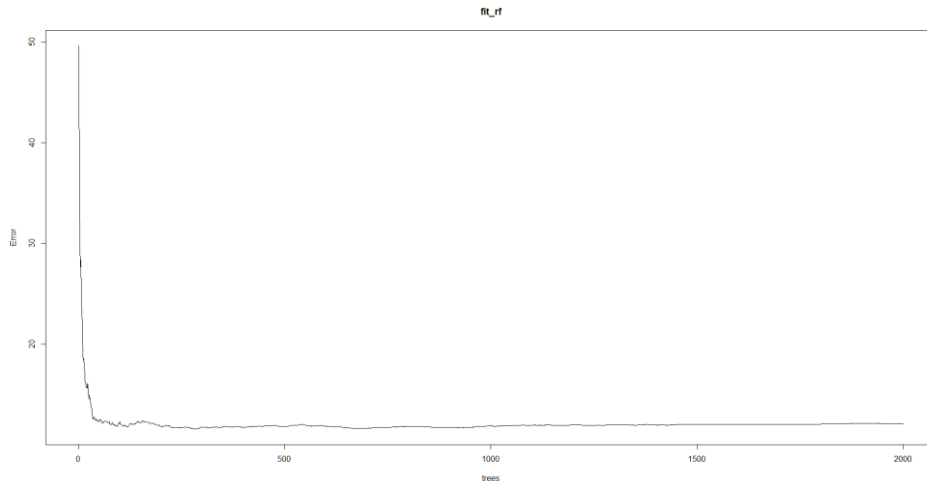


Figure 4.3.3 – Mean of squared residuals vs number of trees

As it is shown by the plot above, the error seems being constant/slightly increasing after a certain number of trees in the forest. Hence, 500 trees seems to be a reasonable value for our Random Forest as the error appears to be minimum.

We now validate of model with the help of the validation set. As already written, the same test is used for parameter tuning. Recall, the quantity to optimize is the RMSE.

```
prediction <- predict(fit_rf, newdata = validation)
mse <- sum(((prediction) - validation$medv)^2) / length(validation$medv)
rmse <- sqrt(mse)
rmse
```

Figure 4.3.4 – Prediction and RMSE of the validation set

The RMSE value generated by the code above is 2.816524. For the test set, we get a RMSE = 3.494445.

Random Forest outliers handling

We start now tuning our model. Before starting with the actual parameter tuning, we investigate what happens when we exclude **outliers** from our train dataset for the response variable medv.

An *outlier* is a data point (observation) that differs significantly from the other observations. It may be due to noisy measurements or experimental errors. Since outliers may cause serious problems in statistical analyses, it may be a good idea to exclude them while building an algorithm.

In *R*, outliers are detected and shown with `boxplot()` according to a method called *Tukey's fences*. This method is based on the *interquartile range (IQR)* where, for example, if Q_1 and Q_3 are the *lower* and *upper quartiles* respectively, then one could define an outlier to be any observation outside the range:

$$[Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)]$$

for some nonnegative constant k . For *Tukey's method*, $k = 1.5$ indicates an outlier.

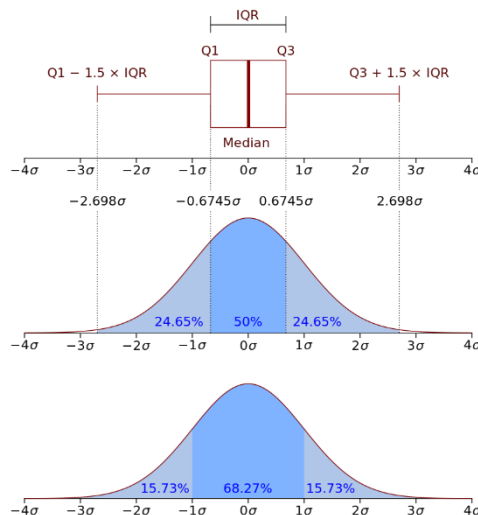


Figure 4.3.5 – Tukey's fences

Nevertheless, since validation and test dataset are randomly sampled from the whole dataset, we expect that excluding the outliers does not improve our method. In other words, we expect that training a model with a set that does not include outliers performs poorer on validation set and test set as they do contain outliers. Again, in short, outliers are part of the data.

Moreover, by looking at the *medv* values, outliers do not seem being measurement errors. In fact, they are quite reasonable values of the real world. They only appear out of the range for how it is defined by the *Tukey's fences*. Thus, excluding outliers may not be a good idea.

The code below shows how we exclude outliers from our train dataset (20 outliers found).

```
outliers <- boxplot(train$medv)$out
train[which(train$medv %in% outliers), ]
train_no_out <- train[- which(train$medv %in% outliers), ]
```

Figure 4.3.6 – Outliers exclusion from the train set

As shown before, we build our model on the new train without outliers dataset. The output is shown below.

```
> randomForest(formula = medv ~ ., data = train_no_out)
```

Call:

```
randomForest(formula = medv ~ ., data = train_no_out)
Type of random forest: regression
Number of trees: 500
No. of variables tried at each split: 4
```

```
Mean of squared residuals: 6.887547
% Var explained: 84.27
```

Figure 4.3.7 – Random Forest output of train without outliers dataset

Mean of squared residuals has decreased from *11.81814* to *6.887547*. This indicates our model seems getting better in terms of error. Yet, this is true only for the train dataset so far. We then use the new model to predict the values for the validation set and to calculate the RMSE based on these predictions.

After calculating the RMSE for the validation dataset with the new method based on the train dataset without outliers, we found out algorithms performed worse. RMSE increased to 5.503951 from 2.816524. Recall, outliers have been removed only from the train dataset. Hence, validation (and test) data set still contains outliers.

This increment in the RMSE indicates removing outliers for this dataset does not improve of prediction. Therefore, we keep the outliers in our dataset and proceed with further tuning.

Random Forest parameter tuning

We go now deeper into the tuning of the method. For such a task we use Sequential Parameter Optimization Toolbox (**SPOT**) available in *R*.

The set of parameters we want to tune is composed of the **number of trees** `ntree` and the **number of variables tried at each split** `mtry`.

Few other parameters which would make sense to tune are: Subset of most important variables, based on concept of *importance* in Random Forest, instead of using the complete set of variables ($13 + 1$ in our case) and the minimum size of terminal nodes `nodesize`. Nevertheless, since our dataset is composed of only 14 variables and 506 observations, using the all set of predictors, instead of a subset, seems being a reasonable approach. Moreover, the `nodesize` is not taken into account as the default value 5 seems to be reasonable.

The *search space* for `ntree` is defined in a range from 100 to 2000 trees while for `mtry` in a range from 3 to 10 variables.

Below is the implementation in *R* of the SPOT function to be optimize in terms of the parameters `ntree` and `mtry` with respect to the RMSE value calculated for the validation set.

```
#params is a vector cointaining the parameters to optimize. In our case ntree and mtry
predictResultsRFWithParameters <- function(params) {
  fit_rf <- randomForest(formula = medv ~ ., data = train, ntree = params[1], mtry = params[2])
  prediction <- predict(fit_rf, newdata = validation)
  mse <- sum((prediction - validation$medv)^2) / length(validation$medv)
  rmse <- sqrt(mse)
  rmse
}

wrappedFunForSPOT <- function(x) {
  apply(x, 1, FUN = predictResultsRFWithParameters)
}

#Our ntree search space is 100 to 2000, 3 to 10 for mtry, 1 to 10 for nodesize
spotRes <- spot(x = NULL, wrappedFunForSPOT, lower = c(100, 3), upper = c(2000, 10),
  control = list(funEvals = 50, types = c("integer", "integer"), plots = T,
  seedFun = 6))

best_rmse <- min(spotRes$y) #This is based on the validaton dataset
best_ntree <- spotRes$xbest[1]
best_mtry <- spotRes$xbest[2]
```

Figure 4.3.8 – SPOT implementation for `ntree` and `mtry` tuning

The result given by the SPOT implementation is as follows:

```
best_ntree = 1565
best_mtry = 5
```

Now we can build our model with the tuned parameters shown above and, finally, test it on the test data.

```
fit_rf <- randomForest(formula = medv ~ ., data = train, ntree = best_ntree, mtry = best_mtry)
prediction <- predict(fit_rf, newdata = test)
mse <- sum(((prediction) - test$medv)^2) / length(test$medv)
rmse <- sqrt(mse)
rmse
```

Figure 4.3.9 – randomForest() after SPOT tuning

As output of the code above, RMSE = 3.451913. Only a slight improvement after the tuning. RMSE has decreased from 3.494445 before tuning.

Our model works already well with standard defined values of randomForest().

4.4 Kaggle Leaderboard Comparison

We now compare our method with the Kaggle leaderboard for the Boston Housing Price dataset.

Before comparing the result, we get with the results in the board, we change the size of our train and test set according to the one actually used for the competition, approximately 50% test data*.

*<https://www.kaggle.com/c/boston-housing/leaderboard>

The value that we get is 3.047185. This would lead us in 6th position. Not bad as first try 😊

Public Leaderboard

Private Leaderboard

This leaderboard is calculated with approximately 50% of the test data.

The final results will be based on the other 50%, so the final standings may be different.

[Raw Data](#)

[Refresh](#)








#	Team Name	Kernel	Team Members	Score ?	Entries	Last
1	amyaramine			0.00000	3	3y
2	VjKR			0.00000	10	3y
3	MayankSatnalika			1.69715	1	3y
4	Luis Bronchal			2.21189	6	3y
5	yaseen hussain			3.04352	2	3y
6	FlorentRambaud			3.07276	4	3y
7	Valentino1992			3.08600	12	3y

Figure 4.4.1 – Kaggle public leaderbord for Boston Housing Price

References

- i. *DisplayR* - <https://www.displayr.com/how-is-variable-importance-calculated-for-a-random-forest/>
- ii. *RPubs* - <https://rpubs.com/chocka314/251613>
- iii. *Wikipedia* - https://en.wikipedia.org/wiki/Interquartile_range