Regression Modelling to Predict the Age of the Abaolne based on its Physical Characteristics - Log Transformed Response

Minoli Munasinghe

2023-04-10

Read the dataset from UCI machine learning repository

```
# Read the dataset into a data frame
abalone_df <- read.csv("http://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data",
# Print first five rows in the data set
head(abalone_df)
```

```
##   V1    V2    V3    V4     V5     V6     V7    V8 V9
## 1  M 0.455 0.365 0.095 0.5140 0.2245 0.1010 0.150 15
## 2  M 0.350 0.265 0.090 0.2255 0.0995 0.0485 0.070  7
## 3  F 0.530 0.420 0.135 0.6770 0.2565 0.1415 0.210  9
## 4  M 0.440 0.365 0.125 0.5160 0.2155 0.1140 0.155 10
## 5  I 0.330 0.255 0.080 0.2050 0.0895 0.0395 0.055  7
## 6  I 0.425 0.300 0.095 0.3515 0.1410 0.0775 0.120  8
```

Add column names to the data set

```
# add column names
names(abalone_df) <- c("sex", "length", "diameter", "height", "weight.whole",
                       "weight.shucked", "weight.viscera", "weight.shell", "rings")
```

Get the number of observations in the data

```
# numbers of rows in the data set
nrow(abalone_df)
```

```
## [1] 4177
```

Get the number of columns in the data

```
ncol(abalone_df)
```

```
## [1] 9
```

The data set has 4177 observations with 9 variables

Get the number of missing values

```r
# Check for the missing values
sum(is.na(abalone_df))
```

```
## [1] 0
```

The data set does not have any missing values

Get the structure of data

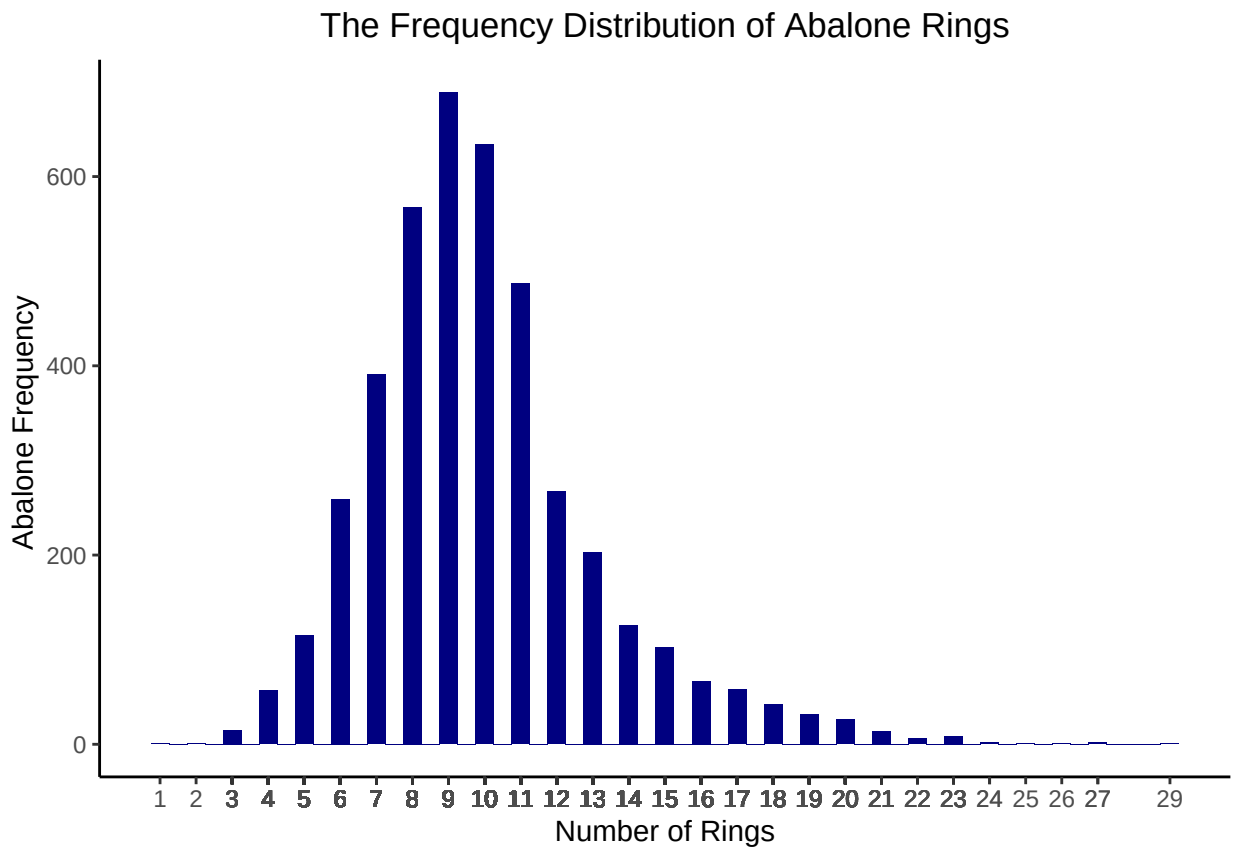
```r
str(abalone_df)
```

```
## 'data.frame':    4177 obs. of  9 variables:
##  $ sex           : chr  "M" "M" "F" "M" ...
##  $ length        : num  0.455 0.35 0.53 0.44 0.33 0.425 0.53 0.545 0.475 0.55 ...
##  $ diameter      : num  0.365 0.265 0.42 0.365 0.255 0.3 0.415 0.425 0.37 0.44 ...
##  $ height        : num  0.095 0.09 0.135 0.125 0.08 0.095 0.15 0.125 0.125 0.15 ...
##  $ weight.whole  : num  0.514 0.226 0.677 0.516 0.205 ...
##  $ weight.shucked: num  0.2245 0.0995 0.2565 0.2155 0.0895 ...
##  $ weight.viscera: num  0.101 0.0485 0.1415 0.114 0.0395 ...
##  $ weight.shell  : num  0.15 0.07 0.21 0.155 0.055 0.12 0.33 0.26 0.165 0.32 ...
##  $ rings         : int  15 7 9 10 7 8 20 16 9 19 ...
```

According to the data, there is one categorical variable (sex) and all the oher variables are continuous. The response variable of the model is rings.

The distribution of response variable

```r
library(ggplot2)
ggplot(aes(x = rings),data = abalone_df)+
geom_histogram(binwidth = 0.5,fill="navy blue") +
scale_x_continuous(breaks = abalone_df$rings)+
ylab("Abalone Frequency ")+
xlab("Number of Rings")+
ggtitle("The Frequency Distribution of Abalone Rings")+
theme_classic() +
theme(plot.title = element_text(hjust = 0.5))
```

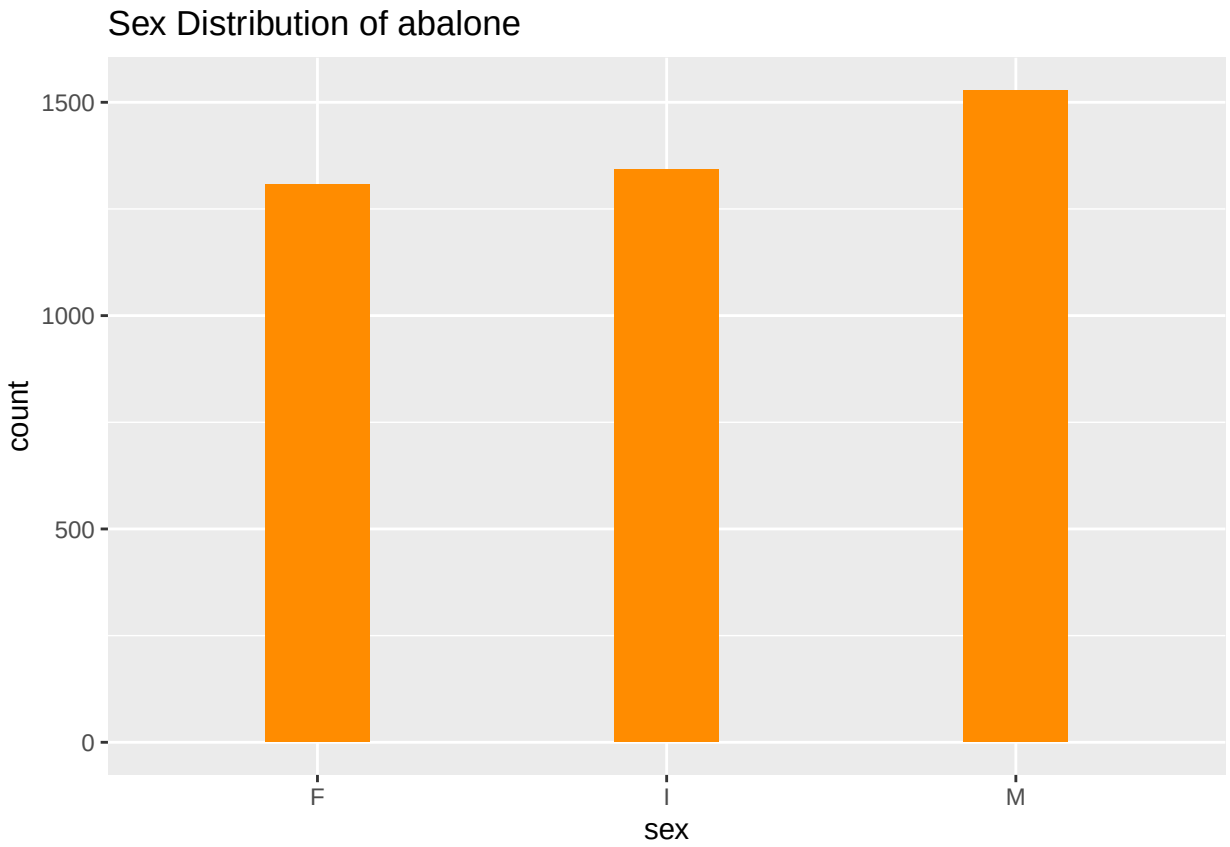



```
library(moments)
skewness(abalone_df$rings)
```

```
## [1] 1.113702
```

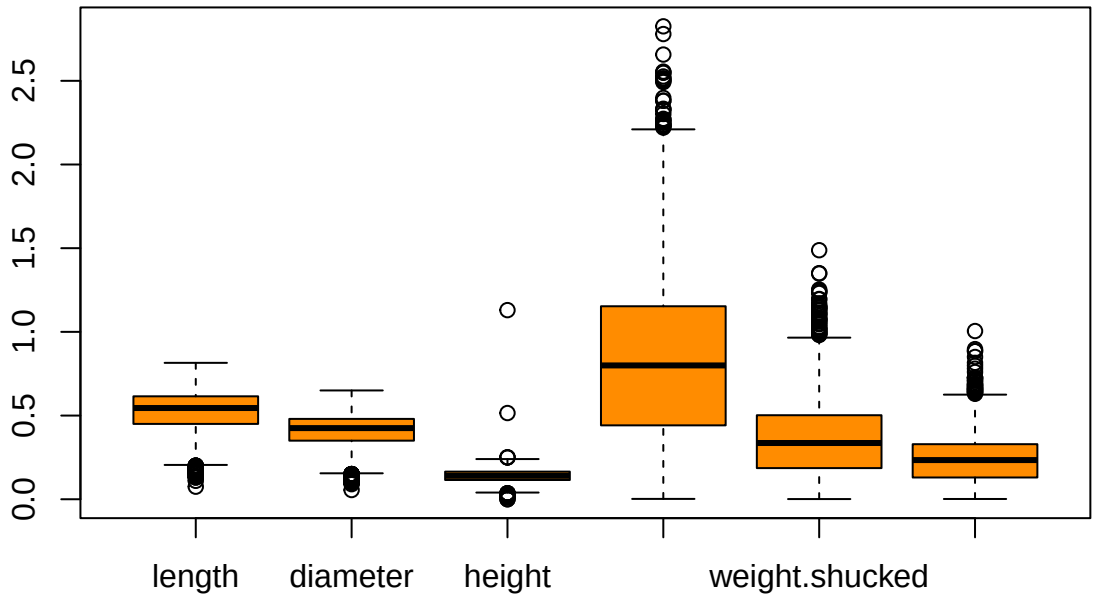
The distribution of observations by number of rings is observed using the above histogram and according to its skewness value of 1.1, we can consider that it is moderately positively skewed and the distribution is also not symmetrically distributed around the mean.

```
library(ggplot2)
ggplot(aes(x = sex),data = abalone_df)+
  geom_bar(stat="count",width = 0.3,fill = "dark orange")+
  theme(legend.position = "None")+
  ggtitle("Sex Distribution of abalone")
```



Observe the distribution of other variables

```
boxplot(abalone_df[,c(2,3,4,5,6,8)],col = "dark orange") # boxplot to check possible outliers
```



According to the boxplots drawn for all the other variables, we can see that all variables have outlier values in the data.

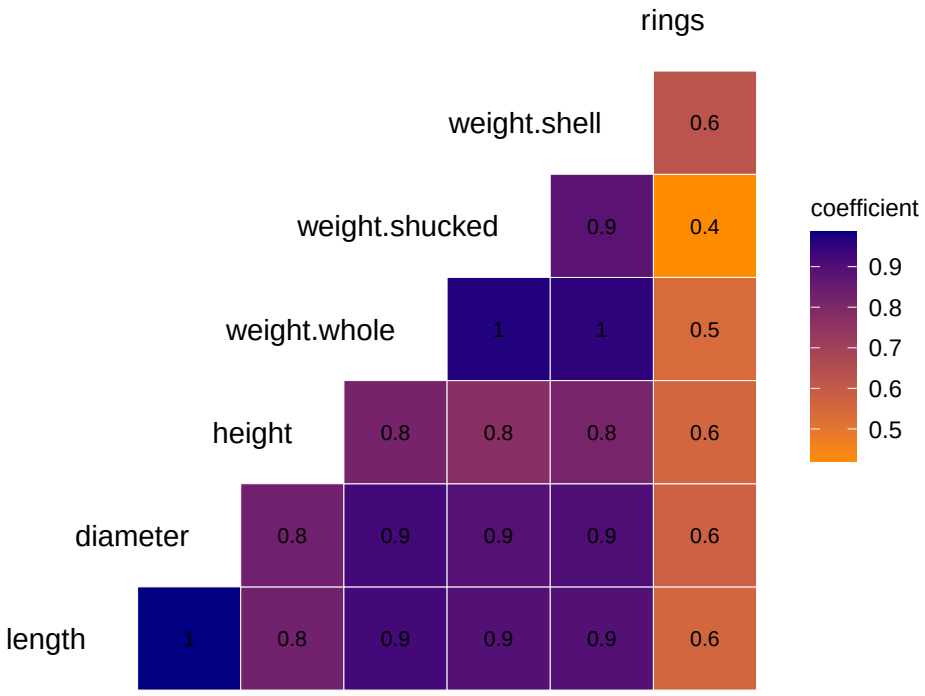
Correlation plot to observe the relationship between variables

```
library(GGally)
```

```
## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2
```

```
ggcorr(abalone_df[,c(2,3,4,5,6,8,9)], label = T, label_size = 2.9, hjust = 1, layout.exp = 3) +
  scale_fill_gradient(low = "dark orange", high = "navy blue")
```

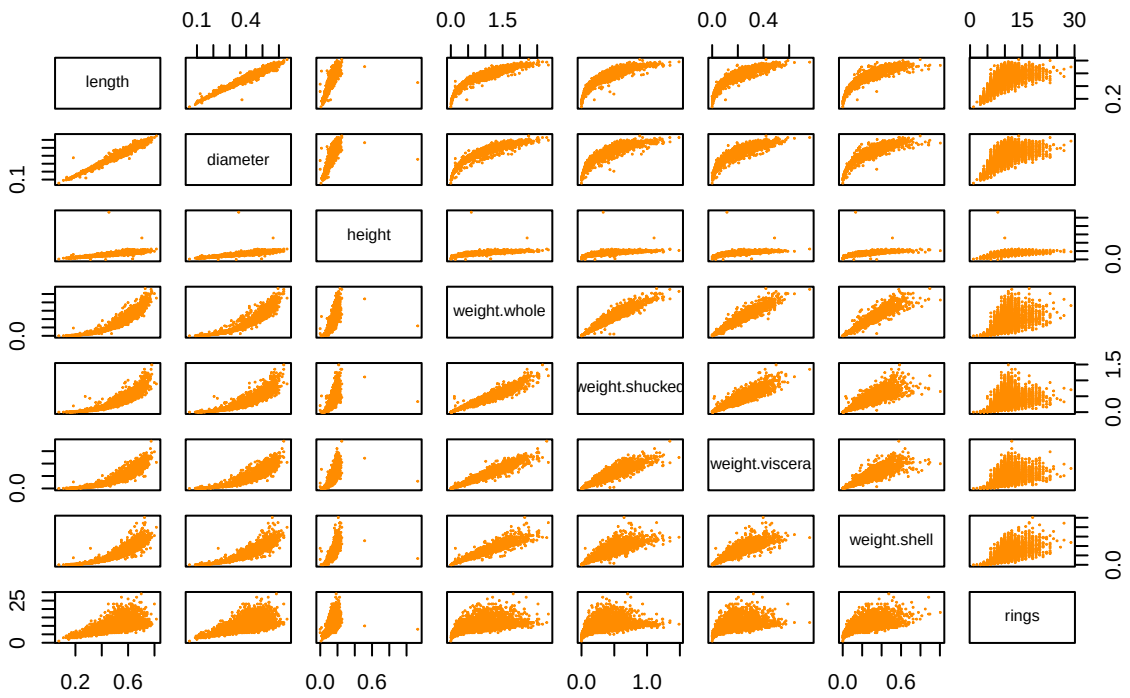
```
## Scale for fill is already present.
## Adding another scale for fill, which will replace the existing scale.
```



According to the correlation plot, we can see that the predictor variables are highly correlated and therefore multicollinearity exists.

Pairwise Scatterplot

```
# Plot the scatter matrix
pairs(abalone_df[,2:9], main = "Scatterplot Matrix of Abalone Data",col = "dark orange",cex=0.1,cex.mai
```



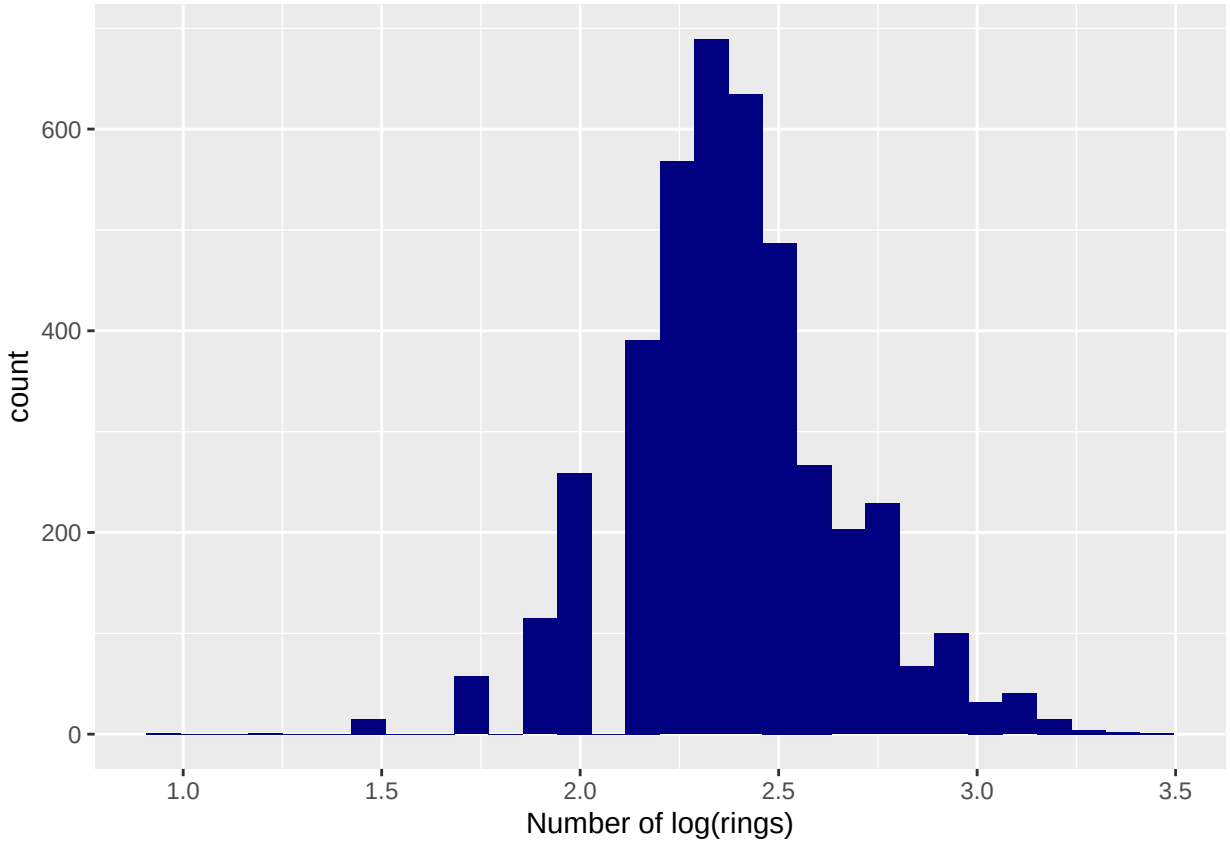

```
abalone_df$trn_rings = log(abalone_df$rings+1.495)
skewness(abalone_df$trn_rings)
```

```
## [1] 0.0008371152
```

The distribution of Log transformation of the rings

```
library(ggplot2)
ggplot(aes(x = trn_rings),data = abalone_df)+
geom_histogram(fill="Navy blue")+
xlab("Number of log(rings)")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
ggtitle("The Frequency Distribution of Log of Abalone Rings")+
theme_classic() +
theme(plot.title = element_text(hjust = 0.5))
```

```
## NULL
```

Handling the outlier data

```
find_outliers = function(x){
  q1 = quantile(x,probs = 0.25)
  q3 = quantile(x,probs = 0.75)

  IQR = q3 - q1

  status = x < q1 - (IQR*1.5) | x > q3 + (IQR*1.5)

}

remove_outliers=function(processed_data,cols = names(processed_data)) {

    for (col in cols) {
        processed_data = processed_data[!find_outliers(processed_data[[col]]), ]
    }
  return(processed_data)
}
```

```
abalone_df = remove_outliers(abalone_df,cols =
                                    c("length","diameter","height","weight.whole",
                                     "weight.shucked","weight.viscera","weight.shell"))
dim(abalone_df)
```

```
## [1] 4013   10
```

According to the results, current dataset without outliers has 4013 records, thus only 164 records have been removed due to outlier detection.

Convert the categorical variable *sex* into numeric variable

```
# Convert sex to factor
sex_factor <- as.factor(abalone_df$sex)

# Convert factor levels to numeric values
abalone_df["sex"] = as.numeric(sex_factor)
```

Data Normalization

```
# Standard Scaling the data
abalone_df = scale(abalone_df,scale = TRUE,center = TRUE)
```

Split the data set in to training and testing set

```
set.seed(2023)

# Remove the 9th column (rings) from the dataset
abalone_df <- subset(abalone_df, select = -rings)

# Split data into training & testing set
sample <- sample(c(TRUE, FALSE), nrow(abalone_df), replace=TRUE, prob=c(0.80,0.20))
train_set <- abalone_df[sample, ]
test_set <- abalone_df[!sample, ]

# Separate predictor and response data from training set
x_train <- train_set[,-9]
y_train <- train_set[,9]

# Separate predictor and response data from testing set
x_test <- test_set[,-9]
y_test <- test_set[,9]
```

Fit the multiple linear regression model with training data including all predictor variables (Transformed response variable - log(rings))

```
# ------------------------------Multiple Linear Regression ----------------------------------------

# The start time to train the model
start = Sys.time()
# Train Multiple Linear regression model
```

```
ml_model <- lm(trn_rings~., data = as.data.frame(train_set))
# End time to finish the training
end = Sys.time()
# Time taken to train the model
mlr.time = end-start

summary(ml_model)
```

```
##
## Call:
## lm(formula = trn_rings ~ ., data = as.data.frame(train_set))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.80733 -0.45401 -0.07893  0.36652  2.81267
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)     0.003169   0.011734   0.270    0.787
## sex             0.009402   0.011799   0.797    0.426
## length          0.012161   0.072853   0.167    0.867
## diameter        0.415778   0.073669   5.644 1.81e-08 ***
## height          0.300328   0.029370  10.226  < 2e-16 ***
## weight.whole    1.101572   0.117733   9.357  < 2e-16 ***
## weight.shucked -1.262270   0.060354 -20.914  < 2e-16 ***
## weight.viscera -0.284688   0.047215  -6.030 1.83e-09 ***
## weight.shell    0.313982   0.054937   5.715 1.19e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6678 on 3231 degrees of freedom
## Multiple R-squared:  0.5629, Adjusted R-squared:  0.5619
## F-statistic: 520.2 on 8 and 3231 DF,  p-value: < 2.2e-16
```

The adjusted R squared value of training set is 56.2% with a RMSE of 66.7%. According to the summary results, length and diameter are not significant variables of the model.

Making predictions from the trained multiple linear regression model on the testing set

```
library(ModelMetrics)
```

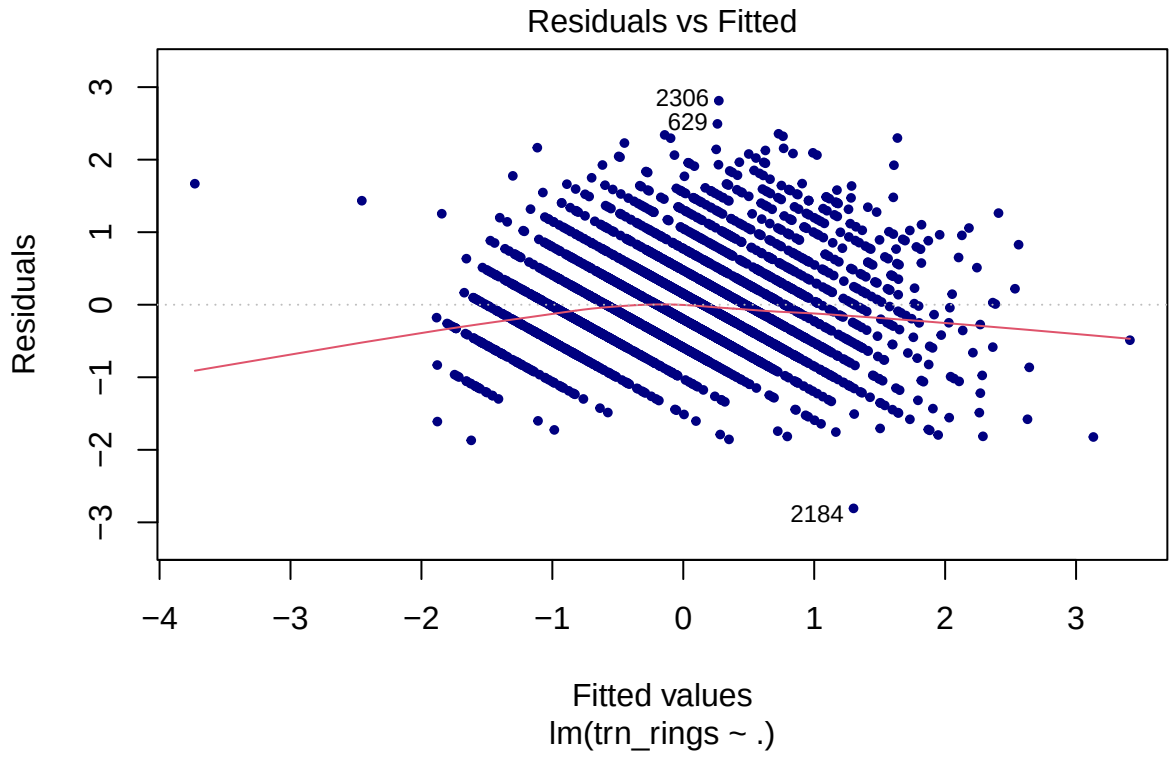
```
##
## Attaching package: 'ModelMetrics'

## The following object is masked from 'package:base':
##
##     kappa
```

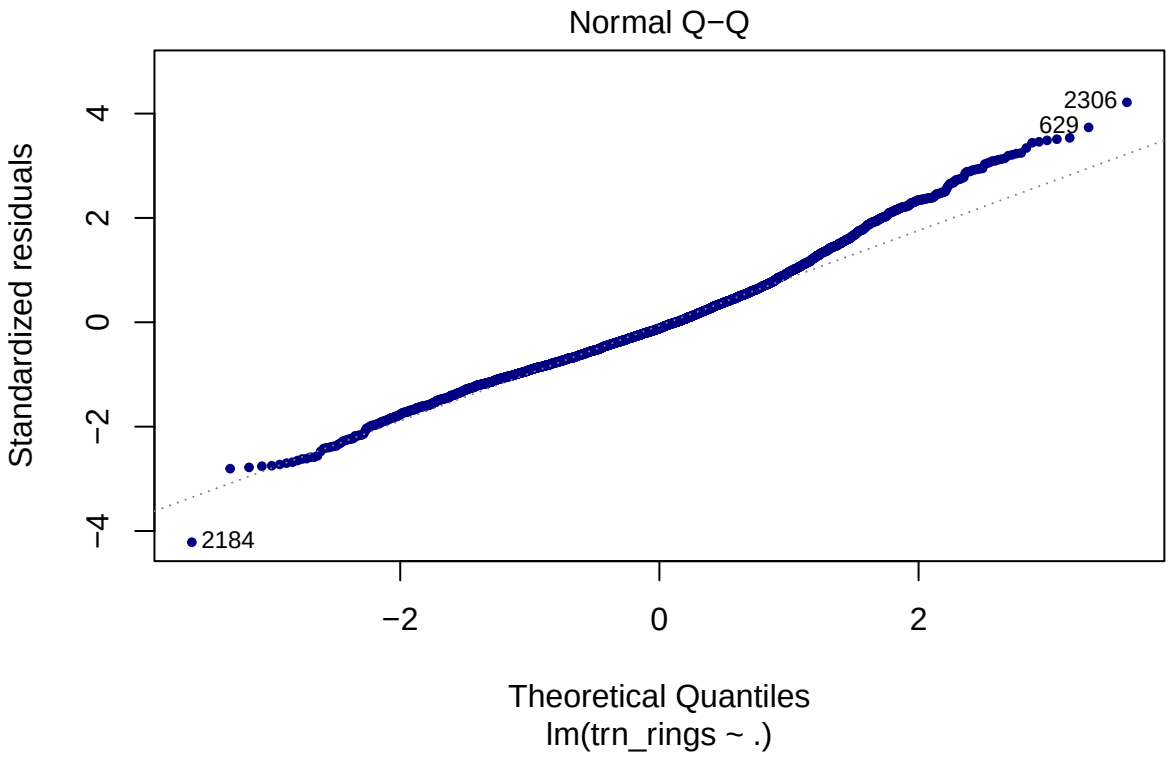
```
# R squared on train data
mlr.tr.Rsquare = summary(ml_model)$adj.r.squared
# RMSE on train data
mlr.tr.RMSE = 0.6678
```

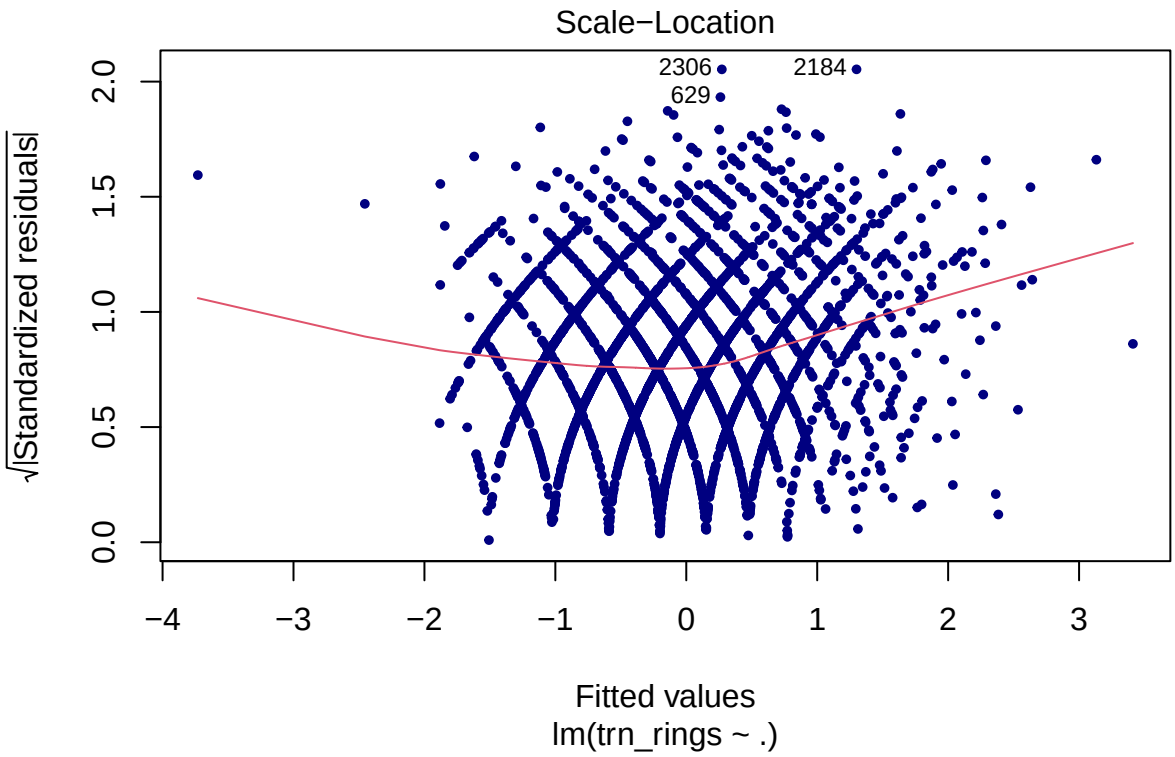
```r
# Predict on test data
y_pred <- predict(ml_model, newdata=as.data.frame(x_test))

# RMSE and R squared for test set
data.frame(
mlr.te.RMSE = caret::RMSE(y_pred, y_test),
mlr.te.Rsquare = caret::R2(y_pred,y_test)
)
```

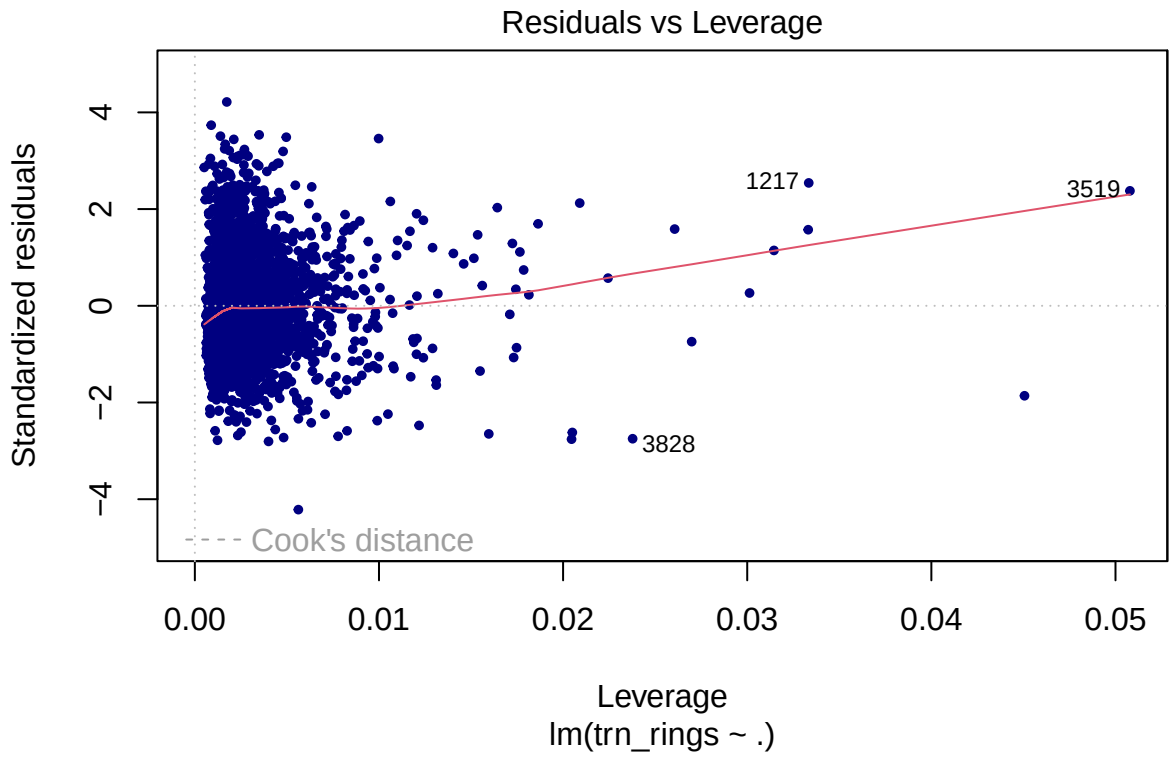
```
##   mlr.te.RMSE mlr.te.Rsquare
## 1   0.6760124      0.5064648
```

```r
plot(ml_model, col = "navy blue",cex = 0.8,pch=20)
```




Normal Q−Q
Standardized residuals
4
2
0
−2
−4
2306
629
2184
−2
0
2
Theoretical Quantiles
lm(trn_rings ~ .)


Scale−Location
2306
2184
629
√|Standardized residuals|
0.0
0.5
1.0
1.5
2.0
−4
−3
−2
−1
0
1
2
3
Fitted values
lm(trn_rings ~ .)



We can see that the assumptions of the linear regression is violated and therefore, we first need to remove multicollinearity among the predictor variables, so that the correlation among the predictors will be reduced to a considerable amount to get a better fitted model

To remove the multicolinearity, I would choose principal component regressor, ridge regualarization and lasso regualrization

PCR - Reducing the dimensions would transform the correlated predictor variables into uncorrelated principal components which would reduce the multicolinearity.

The optimum number of components in the PCR is obtained from hyper parameter tuning

```
# Principal Component Regression with hyper parameter tuning using 10-fold cross validation

library(caret)
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following objects are masked from 'package:ModelMetrics':
##
##     confusionMatrix, precision, recall, sensitivity, specificity
```

```
# Start time to train the PCR
start = Sys.time()

# define hyperparameters for the number of components
hyperparams <- expand.grid(ncomp = c(1, 2, 3, 4,5,6,7,8))

# Train PCR model using cross validation
pcr.model <- train(trn_rings~., data = as.data.frame(train_set), method = "pcr",
                   scale = TRUE,
                   trControl = trainControl("cv", number = 10),
                   tuneGrid=hyperparams
)
# End time to finish the training
end = Sys.time()
# Time taken to train the model
pcr.time_taken = end-start
```

```
# Print the best tuning parameter (number of components) that minimize the cross-validation error, RMSE
pcr.model$bestTune
```

```
##   ncomp
## 8     8
```

```
# Summarize the final PCR model
summary(pcr.model$finalModel)
```

```
## Data:    X dimension: 3240 8
##  Y dimension: 3240 1
## Fit method: svdpc
## Number of components considered: 8
## TRAINING: % variance explained
##           1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X           81.28    93.77    96.16    97.79    98.91    99.74    99.91
## .outcome    37.03    37.05    49.41    49.50    54.50    54.89    54.99
##           8 comps
## X          100.00
## .outcome    56.29
```

According to the results, optimum number of components selected is 8, and 56% of the variability in the log rings can be explained by the predictor variables.

Making predictions on the training and testing data

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
# Get the R squared and RMSE on training set
train_pred <- pcr.model %>% predict(x_train)
data.frame(
pcr.tr.RMSE = caret::RMSE(train_pred, y_train),
pcr.tr.Rsquare = caret::R2(train_pred,y_train)
)
```

```
##   pcr.tr.RMSE pcr.tr.Rsquare
## 1   0.6669062      0.5629435
```

```
# Make predictions on test data
test_pred <- pcr.model %>% predict(x_test)
data.frame(
pcr.te.RMSE = caret::RMSE(test_pred, y_test),
pcr.te.Rsquare = caret::R2(test_pred,y_test)
)
```

```
##   pcr.te.RMSE pcr.te.Rsquare
## 1   0.6760124      0.5064648
```

Ridge Regression model

A ridge regression model is fitted with hyperparameter tuning of lambda using 10-fold cross validation

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-6
```

```
# Start time
start  = Sys.time()

# define grid of hyper parameters
grid <- expand.grid(alpha = 0, lambda = seq(0, 1, by = 0.01))

# implement 10-fold cross validation
ctrl <- trainControl(method = "cv", number = 10)

# train the ridge regression model using 10-fold cross validation
ridge.model <- train(trn_rings ~ ., data = train_set, method = "glmnet", trControl = ctrl, tuneGrid = g

# Optimized alpha
ridge.model$bestTune$alpha
```

```
## [1] 0
```

```
# Optimized lambda
ridge.model$bestTune$lambda
```

```
## [1] 0.06
```

```
# Fit the final model on the training data using optimal lambda
ridge.model <- glmnet(x = as.matrix(x_train), y = as.matrix(y_train), alpha = 0, lambda = ridge.model$b

# End time
end = Sys.time()
# Time taken to train the ridge model
ridge.time_taken = end - start
```

Making predictions on the training data and testing data using ridge regression model

```
library(dplyr)

# Make predictions on train_set
train_pred <- ridge.model %>% predict(as.matrix(x_train))
data.frame(
ridge.tr.RMSE = caret::RMSE(train_pred, y_train),
ridge.tr.Rsquared = caret::R2(train_pred,y_train)
)
```

```
##   ridge.tr.RMSE        s0
## 1      0.687204 0.5395245
```

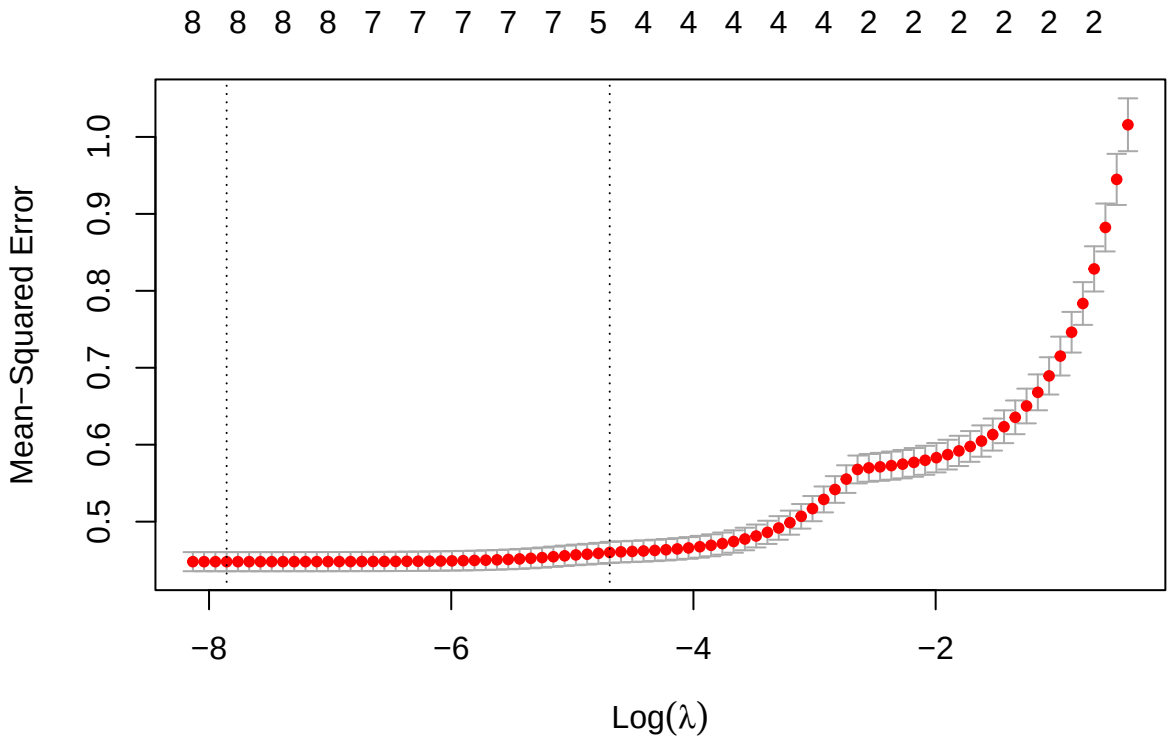
```
# Make predictions on the test data
y_test = as.matrix(y_test)
test_pred <- ridge.model %>% predict(as.matrix(x_test))
data.frame(
ridge.te.RMSE = caret::RMSE(test_pred, y_test),
ridge.te.Rsquared = caret::R2(test_pred,y_test)
)
```

```
##   ridge.te.RMSE        s0
## 1     0.6887168 0.4870175
```

Lasso Regression

Lasso Regressor is fitted with hyper parameter tuning of lambda using 10-fold cross validation

```
# Start time
start = Sys.time()

# Train the lasso model
cv.lasso <- cv.glmnet(x = x_train, y = as.matrix(y_train), alpha =1, lambda = NULL)
plot(cv.lasso)
```




```
# Optimal lambda
cv.lasso$lambda.min
```

```
## [1] 0.0003879473
```

```
# Fit the final model on the training data using optimal lambda
lasso.model <- glmnet(x = x_train, y = as.matrix(y_train), alpha = 1, lambda = cv.lasso$lambda.min)

# End time
end = Sys.time()
# Time taken to train the model
lasso.time_taken = end - start
```

The optimal lambda obtained from hyper parameter tuning for lasso regression model is 0.00029

The trained model on optimum lambda is used to make the predictions on training and testing data

```
train_pred <- lasso.model %>% predict(x_train)
data.frame(
lasso.tr.RMSE = caret::RMSE(train_pred, y_train),
lasso.tr.Rsquared = caret::R2(train_pred,y_train)
)
```

```
##   lasso.tr.RMSE        s0
## 1     0.6669464 0.5628945
```

```
# Make predictions on the test data

predictions <- lasso.model %>% predict(x_test)
data.frame(
lasso.te.RMSE = caret::RMSE(predictions, y_test),
lasso.te.Rsquared = caret::R2(predictions,y_test)
)
```

```
##   lasso.te.RMSE       s0
## 1     0.6758612 0.506529
```

Support Vector Regressor

A support vector regressor is fitted with tuning the hyperparameters (C,Sigma) using 10-fold cross validation

```
#Load Library
library(e1071)
```

```
##
## Attaching package: 'e1071'

## The following objects are masked from 'package:moments':
##
##     kurtosis, moment, skewness
```

```
library(caret)

# Start time
start = Sys.time()
# Define grid for hyper parameter tuning
grid <- expand.grid(sigma = c(0.01, 0.1, 1),
                    C = c(0.1, 1, 10))

# train svm model with cross validation & defined grid of hyper parameters
ctrl <- trainControl(method = "cv", number = 10)
svm.model <- train(trn_rings ~ ., data = train_set, method = "svmRadial", trControl = ctrl, tuneGrid = g


# Optimum parameters
best_c <- svm.model$bestTune$C
best_sigma <- svm.model$bestTune$sigma

# train svm model with optimum parameters
svm.model <- train(trn_rings ~ ., data = train_set, kernel = "radial", cost = best_c, epsilon = best_sig

# end time
end = Sys.time()
# Time taken to train
svm.time_taken = end - start

# Prediction on training data
train_pred = predict(svm.model, x_train)
```

```
data.frame(
  svm.tr.RMSE = caret::RMSE(train_pred, y_train),
  svm.tr.Rsquare = caret::R2(train_pred, y_train)
)
```

```
##   svm.tr.RMSE svm.tr.Rsquare
## 1   0.2998703       0.927161
```

```
#Prediction on testing data
test_pred = predict(svm.model, x_test)
data.frame(
  svm.te.RMSE = caret::RMSE(test_pred, y_test),
  svm.te.Rsquare = caret::R2(test_pred, y_test)
)
```

```
##   svm.te.RMSE svm.te.Rsquare
## 1    0.630382      0.5711228
```

```
# Optimum parameters
best_c
```

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

```
best_sigma
```

```
## [1] 0.1
```

This is computationally expensive method.

Random Forest Regressor

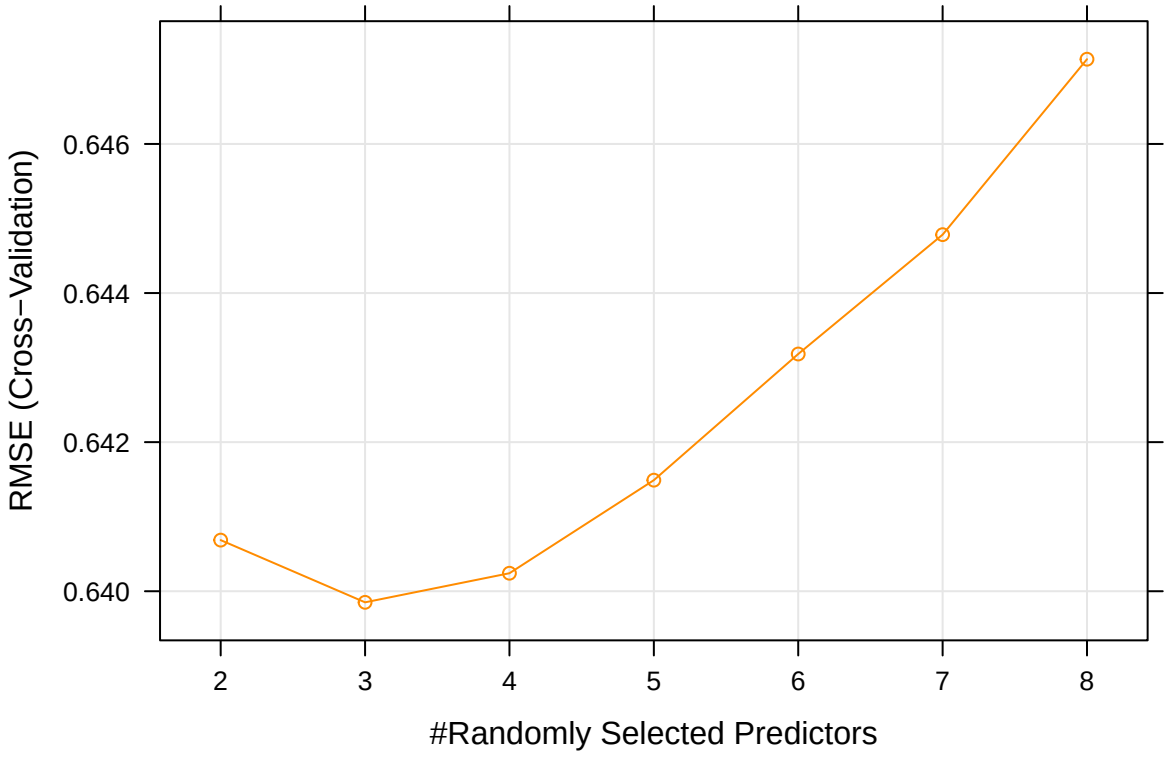
```
# ---------------------------------------Random Forest Regressor ---------------------------
library(caret)


# Start time
start = Sys.time()

train.control <- trainControl(method = "cv", number = 10)
# Train the model
rf.model <- train(trn_rings ~., data = train_set, method = "rf",
                      trControl = train.control,
                      tuneLength=20)
```

```
## note: only 7 unique complexity parameters in default grid. Truncating the grid to 7 .
```

```
print(rf.model) # Summarize the results
```

```
## Random Forest
##
## 3240 samples
##    8 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2916, 2916, 2916, 2917, 2916, 2915, ...
## Resampling results across tuning parameters:
##
##   mtry  RMSE       Rsquared   MAE
##   2     0.6406857  0.5963796  0.4913078
##   3     0.6398510  0.5972318  0.4905082
##   4     0.6402411  0.5968272  0.4904695
##   5     0.6414899  0.5953720  0.4919011
##   6     0.6431821  0.5932803  0.4933833
##   7     0.6447835  0.5914076  0.4944942
##   8     0.6471372  0.5887270  0.4964135
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 3.
```

```
plot(rf.model, col = "dark orange")
```



```
# end time
end = Sys.time()
# Execution time for RF
rf.time = end-start

# Make predictions on train data
train_pred <- rf.model %>% predict(x_train)

data.frame(
  RMSE = caret::RMSE(train_pred, y_train),
  Rsquare = caret::R2(train_pred, y_train)
)
```

```
##        RMSE   Rsquare
## 1 0.2871294 0.9324538
```

```
# Make predictions on test data
test_pred <- rf.model %>% predict(x_test)

data.frame(
  RMSE = caret::RMSE(test_pred, y_test),
  Rsquare = caret::R2(test_pred, y_test)
)
```

```
##        RMSE   Rsquare
## 1 0.6288769 0.5741445
```

```
varImp(rf.model)
```

```
## rf variable importance
##
##                Overall
## weight.shell    100.00
## weight.whole     43.92
## height           34.32
## weight.shucked   34.08
## weight.viscera   33.22
## diameter         29.50
## length           13.28
## sex               0.00
```

```
varImp(rf.model,conditional = TRUE)
```

```
## rf variable importance
##
##                Overall
## weight.shell    100.00
## weight.whole     43.92
## height           34.32
## weight.shucked   34.08
## weight.viscera   33.22
```

```
## diameter         29.50
## length           13.28
## sex               0.00
```

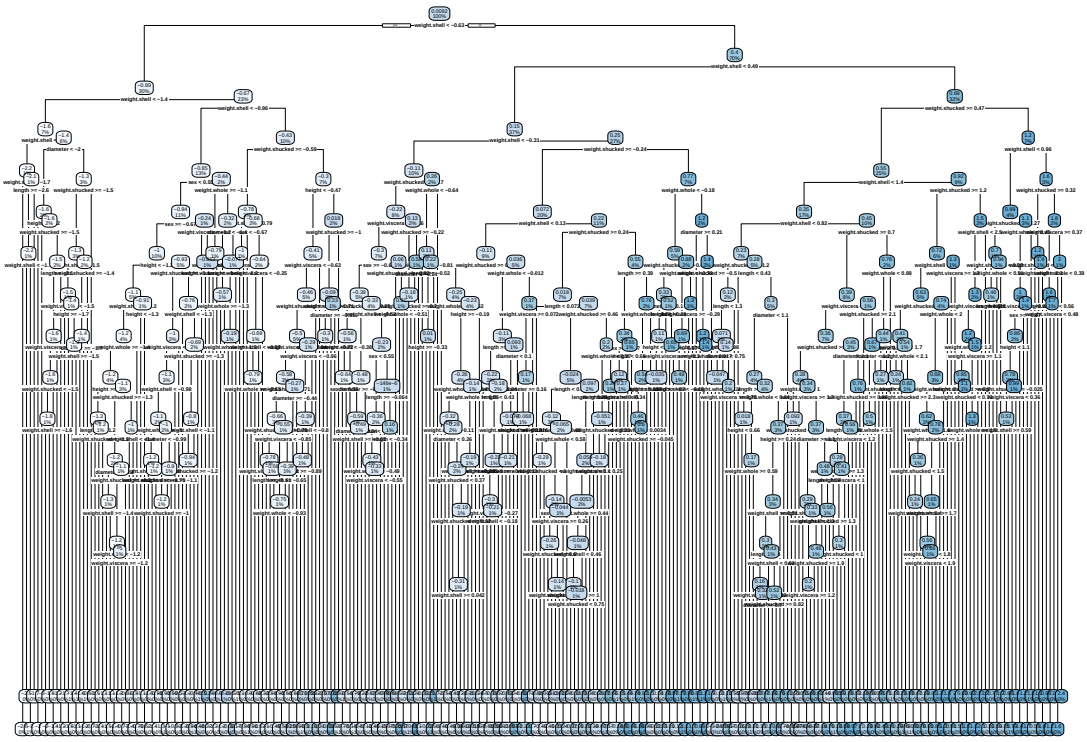
Decision Tree Regressor

```
# Load libraries
library(rpart)
library(rpart.plot)
library(caret)

# Set up cross-validation
ctrl <- trainControl(method = "cv", number = 10)

# Define hyperparameter grid
grid <- expand.grid(cp = seq(0, 0.1, by = 0.01))


# Start time
start = Sys.time()

# Tune hyperparameters using cross-validation
dt_model <- train(trn_rings ~ .,
                  data = train_set,
                  method = "rpart",
                  trControl = ctrl,
                  tuneGrid = grid)

# Print the best parameters
print(dt_model$bestTune)
```

```
##   cp
## 1  0
```

```
# Train the final model using the best parameters
final_dt_model <- rpart(trn_rings ~ .,
                        data = as.data.frame(train_set),
                        cp = dt_model$bestTune$cp)

# end time
end = Sys.time()
# Execution time for RF
dt.time = end-start

# Plot the tree
rpart.plot(final_dt_model)
```

```
## Warning: labs do not fit even at cex 0.15, there may be some overplotting
```



```
# Make predictions on train data
train_pred <- final_dt_model %>% predict(as.data.frame(x_train))

data.frame(
  RMSE = caret::RMSE(train_pred, y_train),
  Rsquare = caret::R2(train_pred, y_train)
)
```

```
##        RMSE   Rsquare
## 1 0.5086363 0.7457724
```

```
# Make predictions on test data
test_pred <- final_dt_model %>% predict(as.data.frame(x_test))

data.frame(
  RMSE = caret::RMSE(test_pred, y_test),
  Rsquare = caret::R2(test_pred, y_test)
)
```

```
##       RMSE   Rsquare
## 1 0.714013 0.4849381
```

Neural Network

```r
library(nnet)

# Start time
start = Sys.time()

ctrl <- trainControl(method = "cv", number = 10,
                     search = "grid", summaryFunction = defaultSummary)

grid <- expand.grid(size = c(5,6,7,8), decay = seq(0.1, 0.2, 0.3))

model <- train(trn_rings ~ ., data = train_set, method = "nnet", tuneGrid = grid,
               trControl = ctrl)
```

```
## # weights:  51
## initial  value 3805.133671
## iter  10 value 2631.759919
## iter  20 value 2264.725203
## iter  30 value 2176.536821
## iter  40 value 2163.829767
## iter  50 value 2160.169408
## iter  60 value 2154.428093
## iter  70 value 2152.364967
## iter  80 value 2149.875925
## iter  90 value 2148.286592
## iter 100 value 2147.786012
## final  value 2147.786012
## stopped after 100 iterations
## # weights:  61
## initial  value 3464.724597
## iter  10 value 2979.710850
## iter  20 value 2335.532572
## iter  30 value 2197.164085
## iter  40 value 2166.622917
## iter  50 value 2157.336502
## iter  60 value 2153.673590
## iter  70 value 2149.097289
## iter  80 value 2147.339871
## iter  90 value 2146.367787
## iter 100 value 2145.724310
## final  value 2145.724310
## stopped after 100 iterations
## # weights:  71
## initial  value 4310.113161
## iter  10 value 2513.332709
## iter  20 value 2252.636416
## iter  30 value 2171.607746
## iter  40 value 2155.687320
## iter  50 value 2150.495469
## iter  60 value 2147.668441
## iter  70 value 2145.707609
## iter  80 value 2143.354747
## iter  90 value 2141.656528
## iter 100 value 2140.223177
```

```
## final  value 2140.223177
## stopped after 100 iterations
## # weights:  81
## initial  value 3308.711842
## iter  10 value 2896.905578
## iter  20 value 2285.098042
## iter  30 value 2191.422695
## iter  40 value 2158.069098
## iter  50 value 2151.076690
## iter  60 value 2147.925421
## iter  70 value 2144.799881
## iter  80 value 2143.723679
## iter  90 value 2142.517683
## iter 100 value 2140.777077
## final  value 2140.777077
## stopped after 100 iterations
## # weights:  51
## initial  value 2990.716101
## iter  10 value 2400.415120
## iter  20 value 2244.773368
## iter  30 value 2196.061155
## iter  40 value 2184.387316
## iter  50 value 2181.298422
## iter  60 value 2176.321703
## iter  70 value 2173.624448
## iter  80 value 2170.693851
## iter  90 value 2169.931266
## iter 100 value 2168.792751
## final  value 2168.792751
## stopped after 100 iterations
## # weights:  61
## initial  value 3635.535289
## iter  10 value 2999.801145
## iter  20 value 2318.737264
## iter  30 value 2197.402622
## iter  40 value 2175.552441
## iter  50 value 2171.181884
## iter  60 value 2168.883849
## iter  70 value 2166.783699
## iter  80 value 2165.545170
## iter  90 value 2164.892267
## iter 100 value 2164.382683
## final  value 2164.382683
## stopped after 100 iterations
## # weights:  71
## initial  value 2963.372282
## iter  10 value 2249.303609
## iter  20 value 2203.228797
## iter  30 value 2185.587755
## iter  40 value 2181.833456
## iter  50 value 2180.006334
## iter  60 value 2178.322998
## iter  70 value 2169.011625
## iter  80 value 2164.923395
```

```
## iter  90 value 2163.595308
## iter 100 value 2162.128871
## final  value 2162.128871
## stopped after 100 iterations
## # weights:  81
## initial  value 3656.799595
## iter  10 value 3001.232396
## iter  20 value 2314.686961
## iter  30 value 2211.071341
## iter  40 value 2177.761601
## iter  50 value 2168.685848
## iter  60 value 2164.631469
## iter  70 value 2163.891346
## iter  80 value 2163.178393
## iter  90 value 2162.717325
## iter 100 value 2162.484069
## final  value 2162.484069
## stopped after 100 iterations
## # weights:  51
## initial  value 3431.474955
## iter  10 value 2734.585462
## iter  20 value 2242.861796
## iter  30 value 2177.353953
## iter  40 value 2164.658043
## iter  50 value 2162.126726
## iter  60 value 2160.397571
## iter  70 value 2155.914601
## iter  80 value 2151.540614
## iter  90 value 2149.435081
## iter 100 value 2147.187343
## final  value 2147.187343
## stopped after 100 iterations
## # weights:  61
## initial  value 3710.160471
## iter  10 value 2726.803674
## iter  20 value 2292.280196
## iter  30 value 2177.144869
## iter  40 value 2153.410246
## iter  50 value 2142.718299
## iter  60 value 2140.803394
## iter  70 value 2140.282159
## iter  80 value 2140.143797
## iter  90 value 2140.027170
## iter 100 value 2139.941349
## final  value 2139.941349
## stopped after 100 iterations
## # weights:  71
## initial  value 3603.329272
## iter  10 value 2640.359572
## iter  20 value 2221.960328
## iter  30 value 2162.088087
## iter  40 value 2155.723947
## iter  50 value 2153.628229
## iter  60 value 2150.379222
```

```
## iter  70 value 2145.650198
## iter  80 value 2143.549937
## iter  90 value 2142.113021
## iter 100 value 2141.593176
## final  value 2141.593176
## stopped after 100 iterations
## # weights:  81
## initial  value 3522.153701
## iter  10 value 2687.998029
## iter  20 value 2199.958911
## iter  30 value 2163.135523
## iter  40 value 2151.183735
## iter  50 value 2146.369399
## iter  60 value 2143.750990
## iter  70 value 2139.552919
## iter  80 value 2138.042710
## iter  90 value 2137.250084
## iter 100 value 2136.737948
## final  value 2136.737948
## stopped after 100 iterations
## # weights:  51
## initial  value 3841.947782
## iter  10 value 2894.850016
## iter  20 value 2264.576076
## iter  30 value 2188.685803
## iter  40 value 2168.791965
## iter  50 value 2152.193166
## iter  60 value 2149.652616
## iter  70 value 2148.053506
## iter  80 value 2147.608522
## iter  90 value 2147.570505
## iter 100 value 2147.566157
## final  value 2147.566157
## stopped after 100 iterations
## # weights:  61
## initial  value 3849.756480
## iter  10 value 2725.577292
## iter  20 value 2353.757328
## iter  30 value 2189.327112
## iter  40 value 2165.948979
## iter  50 value 2153.960935
## iter  60 value 2146.743571
## iter  70 value 2144.888443
## iter  80 value 2143.313611
## iter  90 value 2142.316996
## iter 100 value 2142.128374
## final  value 2142.128374
## stopped after 100 iterations
## # weights:  71
## initial  value 3304.266546
## iter  10 value 2886.872229
## iter  20 value 2416.868528
## iter  30 value 2203.606482
## iter  40 value 2164.087926
```

```
## iter  50 value 2153.730865
## iter  60 value 2150.604160
## iter  70 value 2146.066630
## iter  80 value 2143.394081
## iter  90 value 2141.073959
## iter 100 value 2139.988287
## final  value 2139.988287
## stopped after 100 iterations
## # weights:  81
## initial  value 3089.856524
## iter  10 value 2595.192790
## iter  20 value 2245.936463
## iter  30 value 2173.072813
## iter  40 value 2155.985071
## iter  50 value 2149.623508
## iter  60 value 2147.131275
## iter  70 value 2145.417649
## iter  80 value 2144.384889
## iter  90 value 2143.392692
## iter 100 value 2141.815206
## final  value 2141.815206
## stopped after 100 iterations
## # weights:  51
## initial  value 4148.102660
## iter  10 value 2455.288705
## iter  20 value 2256.836291
## iter  30 value 2203.244837
## iter  40 value 2191.313450
## iter  50 value 2185.393336
## iter  60 value 2182.883839
## iter  70 value 2180.225902
## iter  80 value 2176.451362
## iter  90 value 2175.361520
## iter 100 value 2175.272836
## final  value 2175.272836
## stopped after 100 iterations
## # weights:  61
## initial  value 4312.096244
## iter  10 value 3012.820879
## iter  20 value 2314.827209
## iter  30 value 2224.810333
## iter  40 value 2205.845127
## iter  50 value 2193.326731
## iter  60 value 2182.996080
## iter  70 value 2177.570297
## iter  80 value 2175.306110
## iter  90 value 2174.663668
## iter 100 value 2174.513348
## final  value 2174.513348
## stopped after 100 iterations
## # weights:  71
## initial  value 4283.923930
## iter  10 value 3008.607405
## iter  20 value 2251.249503
```

```
## iter  30 value 2202.725847
## iter  40 value 2185.313033
## iter  50 value 2177.660398
## iter  60 value 2174.633632
## iter  70 value 2173.342971
## iter  80 value 2172.600897
## iter  90 value 2172.419051
## iter 100 value 2172.362056
## final  value 2172.362056
## stopped after 100 iterations
## # weights:  81
## initial  value 3287.271815
## iter  10 value 2736.323147
## iter  20 value 2295.066221
## iter  30 value 2219.280637
## iter  40 value 2193.843606
## iter  50 value 2181.963275
## iter  60 value 2178.882053
## iter  70 value 2178.092507
## iter  80 value 2176.062094
## iter  90 value 2175.601587
## iter 100 value 2174.730451
## final  value 2174.730451
## stopped after 100 iterations
## # weights:  51
## initial  value 3717.055018
## iter  10 value 2810.256968
## iter  20 value 2254.339288
## iter  30 value 2187.186339
## iter  40 value 2168.233773
## iter  50 value 2166.018751
## iter  60 value 2164.149089
## iter  70 value 2162.989276
## iter  80 value 2161.457346
## iter  90 value 2160.943741
## iter 100 value 2159.923670
## final  value 2159.923670
## stopped after 100 iterations
## # weights:  61
## initial  value 3050.890082
## iter  10 value 2790.699352
## iter  20 value 2221.925355
## iter  30 value 2182.002109
## iter  40 value 2170.539046
## iter  50 value 2166.091471
## iter  60 value 2160.436778
## iter  70 value 2155.834669
## iter  80 value 2155.146695
## iter  90 value 2154.707511
## iter 100 value 2153.955695
## final  value 2153.955695
## stopped after 100 iterations
## # weights:  71
## initial  value 3643.989111
```

```
## iter  10 value 2913.163047
## iter  20 value 2279.364510
## iter  30 value 2191.353082
## iter  40 value 2167.260081
## iter  50 value 2161.010486
## iter  60 value 2157.607073
## iter  70 value 2155.951729
## iter  80 value 2155.232451
## iter  90 value 2153.925874
## iter 100 value 2153.666568
## final  value 2153.666568
## stopped after 100 iterations
## # weights:  81
## initial  value 3805.607928
## iter  10 value 3003.568541
## iter  20 value 2350.109285
## iter  30 value 2219.459016
## iter  40 value 2184.635957
## iter  50 value 2166.727739
## iter  60 value 2156.810790
## iter  70 value 2154.419355
## iter  80 value 2153.733241
## iter  90 value 2153.299233
## iter 100 value 2152.754811
## final  value 2152.754811
## stopped after 100 iterations
## # weights:  51
## initial  value 4102.691894
## iter  10 value 2421.901572
## iter  20 value 2248.847702
## iter  30 value 2211.274097
## iter  40 value 2200.902648
## iter  50 value 2190.179137
## iter  60 value 2188.259358
## iter  70 value 2187.400251
## iter  80 value 2186.829796
## iter  90 value 2186.681529
## iter 100 value 2186.673008
## final  value 2186.673008
## stopped after 100 iterations
## # weights:  61
## initial  value 3962.162425
## iter  10 value 2527.334425
## iter  20 value 2283.324331
## iter  30 value 2214.667245
## iter  40 value 2201.940896
## iter  50 value 2196.890248
## iter  60 value 2190.516640
## iter  70 value 2186.383523
## iter  80 value 2184.581474
## iter  90 value 2183.145686
## iter 100 value 2182.858658
## final  value 2182.858658
## stopped after 100 iterations
```

```
## # weights:  71
## initial  value 3426.410654
## iter  10 value 2817.090312
## iter  20 value 2259.295127
## iter  30 value 2211.194468
## iter  40 value 2203.583918
## iter  50 value 2190.241602
## iter  60 value 2183.863475
## iter  70 value 2182.787137
## iter  80 value 2181.899080
## iter  90 value 2181.350314
## iter 100 value 2181.176758
## final  value 2181.176758
## stopped after 100 iterations
## # weights:  81
## initial  value 3285.540119
## iter  10 value 2992.782424
## iter  20 value 2264.228695
## iter  30 value 2215.850623
## iter  40 value 2202.028123
## iter  50 value 2194.714357
## iter  60 value 2190.267326
## iter  70 value 2186.430674
## iter  80 value 2183.865200
## iter  90 value 2182.067790
## iter 100 value 2181.215102
## final  value 2181.215102
## stopped after 100 iterations
## # weights:  51
## initial  value 4538.391654
## iter  10 value 2335.589611
## iter  20 value 2225.585142
## iter  30 value 2181.972484
## iter  40 value 2173.956119
## iter  50 value 2171.031106
## iter  60 value 2170.528826
## iter  70 value 2170.414269
## iter  80 value 2170.339561
## iter  90 value 2169.272645
## iter 100 value 2163.920211
## final  value 2163.920211
## stopped after 100 iterations
## # weights:  61
## initial  value 3825.612789
## iter  10 value 2943.336291
## iter  20 value 2532.186135
## iter  30 value 2248.515855
## iter  40 value 2176.851466
## iter  50 value 2165.495339
## iter  60 value 2161.989423
## iter  70 value 2160.050255
## iter  80 value 2158.522069
## iter  90 value 2157.852887
## iter 100 value 2157.178934
```

```
## final  value 2157.178934
## stopped after 100 iterations
## # weights:  71
## initial  value 4406.987883
## iter  10 value 2977.945042
## iter  20 value 2279.875387
## iter  30 value 2205.681329
## iter  40 value 2179.627337
## iter  50 value 2172.177468
## iter  60 value 2167.418522
## iter  70 value 2160.717755
## iter  80 value 2158.609732
## iter  90 value 2155.309120
## iter 100 value 2153.273779
## final  value 2153.273779
## stopped after 100 iterations
## # weights:  81
## initial  value 4376.502895
## iter  10 value 2936.851790
## iter  20 value 2288.163990
## iter  30 value 2194.817712
## iter  40 value 2166.181078
## iter  50 value 2156.619436
## iter  60 value 2154.829062
## iter  70 value 2153.946413
## iter  80 value 2152.856486
## iter  90 value 2151.979850
## iter 100 value 2151.591149
## final  value 2151.591149
## stopped after 100 iterations
## # weights:  51
## initial  value 3590.750543
## iter  10 value 2949.753032
## iter  20 value 2294.002266
## iter  30 value 2228.761717
## iter  40 value 2185.613349
## iter  50 value 2176.019224
## iter  60 value 2172.643999
## iter  70 value 2172.141053
## iter  80 value 2171.919131
## iter  90 value 2171.729213
## iter 100 value 2171.596990
## final  value 2171.596990
## stopped after 100 iterations
## # weights:  61
## initial  value 3432.478557
## iter  10 value 2920.378636
## iter  20 value 2235.954884
## iter  30 value 2197.395784
## iter  40 value 2187.749732
## iter  50 value 2177.262188
## iter  60 value 2173.490133
## iter  70 value 2171.977947
## iter  80 value 2171.372589
```

```
## iter  90 value 2168.080794
## iter 100 value 2162.481783
## final  value 2162.481783
## stopped after 100 iterations
## # weights:  71
## initial  value 3226.597697
## iter  10 value 2961.581665
## iter  20 value 2484.302086
## iter  30 value 2243.784520
## iter  40 value 2189.240179
## iter  50 value 2177.586895
## iter  60 value 2174.832722
## iter  70 value 2170.803498
## iter  80 value 2164.664175
## iter  90 value 2160.808670
## iter 100 value 2159.188911
## final  value 2159.188911
## stopped after 100 iterations
## # weights:  81
## initial  value 3041.333247
## iter  10 value 2942.736667
## iter  20 value 2346.715402
## iter  30 value 2221.794668
## iter  40 value 2182.687735
## iter  50 value 2172.646961
## iter  60 value 2168.967487
## iter  70 value 2165.663670
## iter  80 value 2163.560660
## iter  90 value 2162.334713
## iter 100 value 2161.317326
## final  value 2161.317326
## stopped after 100 iterations
## # weights:  51
## initial  value 3424.332712
## iter  10 value 2734.060667
## iter  20 value 2329.223221
## iter  30 value 2179.049300
## iter  40 value 2155.508130
## iter  50 value 2141.828787
## iter  60 value 2138.678563
## iter  70 value 2137.607965
## iter  80 value 2137.196939
## iter  90 value 2137.152727
## iter 100 value 2137.147874
## final  value 2137.147874
## stopped after 100 iterations
## # weights:  61
## initial  value 3735.712625
## iter  10 value 2704.147372
## iter  20 value 2307.030353
## iter  30 value 2192.917118
## iter  40 value 2149.504587
## iter  50 value 2143.203020
## iter  60 value 2140.163138
```

```
## iter  70 value 2137.654560
## iter  80 value 2136.726907
## iter  90 value 2136.532392
## iter 100 value 2136.440983
## final  value 2136.440983
## stopped after 100 iterations
## # weights:  71
## initial  value 3253.797943
## iter  10 value 2697.199983
## iter  20 value 2198.348649
## iter  30 value 2155.939916
## iter  40 value 2138.895821
## iter  50 value 2134.115618
## iter  60 value 2132.793180
## iter  70 value 2132.403147
## iter  80 value 2132.160779
## iter  90 value 2132.072203
## iter 100 value 2131.978868
## final  value 2131.978868
## stopped after 100 iterations
## # weights:  81
## initial  value 3640.212593
## iter  10 value 2883.333058
## iter  20 value 2228.818174
## iter  30 value 2174.150837
## iter  40 value 2158.048613
## iter  50 value 2147.143978
## iter  60 value 2137.506588
## iter  70 value 2134.703421
## iter  80 value 2133.792319
## iter  90 value 2133.202437
## iter 100 value 2132.744552
## final  value 2132.744552
## stopped after 100 iterations
## # weights:  61
## initial  value 4039.775727
## iter  10 value 2602.560550
## iter  20 value 2451.186519
## iter  30 value 2409.666674
## iter  40 value 2399.719828
## iter  50 value 2393.943910
## iter  60 value 2392.577486
## iter  70 value 2392.229019
## iter  80 value 2392.001791
## iter  90 value 2391.907162
## iter 100 value 2391.870078
## final  value 2391.870078
## stopped after 100 iterations
```

```r
best_size = model$bestTune$size
best_decay = model$bestTune$decay

nn.model <- nnet(trn_rings ~ ., data = train_set, size = best_size, decay = best_decay, maxit = 1000)
```

```
## # weights:  61
## initial  value 3576.969387
## iter  10 value 2887.945120
## iter  20 value 2548.289797
## iter  30 value 2437.129019
## iter  40 value 2421.454143
## iter  50 value 2410.727033
## iter  60 value 2403.423830
## iter  70 value 2398.443827
## iter  80 value 2395.352966
## iter  90 value 2393.870168
## iter 100 value 2392.866613
## iter 110 value 2391.932289
## iter 120 value 2391.444617
## iter 130 value 2391.219669
## iter 140 value 2391.173615
## iter 150 value 2391.131595
## final  value 2391.128361
## converged
```

```
# end time
end = Sys.time()
# Execution time for RF
nn.time = end-start

# Make predictions on train data
train_pred <- nn.model %>% predict(as.data.frame(x_train))

data.frame(
  RMSE = caret::RMSE(train_pred, y_train),
  Rsquare = caret::R2(train_pred, y_train)
)
```

```
##        RMSE   Rsquare
## 1 0.8530333 0.4679529
```

```
# Make predictions on test data
test_pred <- nn.model %>% predict(as.data.frame(x_test))

data.frame(
  RMSE = caret::RMSE(test_pred, y_test),
  Rsquare = caret::R2(test_pred, y_test)
)
```

```
##        RMSE  Rsquare
## 1 0.8439022 0.407513
```

Bayesian Regression

```
suppressPackageStartupMessages(library(rstanarm))

# Start time
```

```r
start = Sys.time()

model_bayes<- stan_glm(trn_rings~., data=as.data.frame(train_set), family = gaussian(),
prior=normal(0, 5),seed=111)
```

```
##
## SAMPLING FOR MODEL 'continuous' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 0.001 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 10 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 1: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 1: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 1: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 1: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 1: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 1: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 1: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 1: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 1: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 1: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 1: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 0.426 seconds (Warm-up)
## Chain 1:                0.672 seconds (Sampling)
## Chain 1:                1.098 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'continuous' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 0 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 2: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 2: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 2: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 2: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 2: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 2: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 2: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 2: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 2: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 2: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 2: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 0.475 seconds (Warm-up)
## Chain 2:                0.712 seconds (Sampling)
```

```
## Chain 2:                1.187 seconds (Total)
## Chain 2:
##
## SAMPLING FOR MODEL 'continuous' NOW (CHAIN 3).
## Chain 3:
## Chain 3: Gradient evaluation took 0 seconds
## Chain 3: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 3: Adjust your expectations accordingly!
## Chain 3:
## Chain 3:
## Chain 3: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 3: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 3: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 3: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 3: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 3: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 3: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 3: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 3: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 3: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 3: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 3: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 3:
## Chain 3:  Elapsed Time: 0.433 seconds (Warm-up)
## Chain 3:                0.816 seconds (Sampling)
## Chain 3:                1.249 seconds (Total)
## Chain 3:
##
## SAMPLING FOR MODEL 'continuous' NOW (CHAIN 4).
## Chain 4:
## Chain 4: Gradient evaluation took 0 seconds
## Chain 4: 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Chain 4: Adjust your expectations accordingly!
## Chain 4:
## Chain 4:
## Chain 4: Iteration:    1 / 2000 [  0%]  (Warmup)
## Chain 4: Iteration:  200 / 2000 [ 10%]  (Warmup)
## Chain 4: Iteration:  400 / 2000 [ 20%]  (Warmup)
## Chain 4: Iteration:  600 / 2000 [ 30%]  (Warmup)
## Chain 4: Iteration:  800 / 2000 [ 40%]  (Warmup)
## Chain 4: Iteration: 1000 / 2000 [ 50%]  (Warmup)
## Chain 4: Iteration: 1001 / 2000 [ 50%]  (Sampling)
## Chain 4: Iteration: 1200 / 2000 [ 60%]  (Sampling)
## Chain 4: Iteration: 1400 / 2000 [ 70%]  (Sampling)
## Chain 4: Iteration: 1600 / 2000 [ 80%]  (Sampling)
## Chain 4: Iteration: 1800 / 2000 [ 90%]  (Sampling)
## Chain 4: Iteration: 2000 / 2000 [100%]  (Sampling)
## Chain 4:
## Chain 4:  Elapsed Time: 0.453 seconds (Warm-up)
## Chain 4:                0.751 seconds (Sampling)
## Chain 4:                1.204 seconds (Total)
## Chain 4:
```

```
end = Sys.time()
bayes.time = end-start
```

```
summary(model_bayes)
```

```
##
## Model Info:
##  function:     stan_glm
##  family:       gaussian [identity]
##  formula:      trn_rings ~ .
##  algorithm:    sampling
##  sample:       4000 (posterior sample size)
##  priors:       see help('prior_summary')
##  observations: 3240
##  predictors:   9
##
## Estimates:
##                   mean   sd   10%   50%   90%
## (Intercept)      0.0    0.0  0.0   0.0   0.0
## sex              0.0    0.0  0.0   0.0   0.0
## length           0.0    0.1 -0.1   0.0   0.1
## diameter         0.4    0.1  0.3   0.4   0.5
## height           0.3    0.0  0.3   0.3   0.3
## weight.whole     1.1    0.1  0.9   1.1   1.3
## weight.shucked -1.3    0.1 -1.3  -1.3  -1.2
## weight.viscera -0.3    0.0 -0.3  -0.3  -0.2
## weight.shell     0.3    0.1  0.2   0.3   0.4
## sigma            0.7    0.0  0.7   0.7   0.7
##
## Fit Diagnostics:
##            mean   sd   10%   50%   90%
## mean_PPD 0.0    0.0  0.0   0.0   0.0
##
## The mean_ppd is the sample average posterior predictive distribution of the outcome variable (for de
##
## MCMC diagnostics
##                mcse Rhat n_eff
## (Intercept)    0.0  1.0  3139
## sex            0.0  1.0  5963
## length         0.0  1.0  2010
## diameter       0.0  1.0  2079
## height         0.0  1.0  4229
## weight.whole   0.0  1.0  1476
## weight.shucked 0.0  1.0  1804
## weight.viscera 0.0  1.0  2790
## weight.shell   0.0  1.0  1850
## sigma          0.0  1.0  3499
## mean_PPD       0.0  1.0  3631
## log-posterior  0.1  1.0  1688
##
## For each parameter, mcse is Monte Carlo standard error, n_eff is a crude measure of effective sample
```

posterior_predictive checks

```
y_pred = predict(model_bayes,newdata = as.data.frame(x_test))
data.frame(
  RMSE = caret::RMSE(y_pred, y_test),
  Rsquare = caret::R2(y_pred, y_test)
)
```

```
##        RMSE   Rsquare
## 1 0.6759975 0.5064838
```

```
y_pred_train = predict(model_bayes,newdata = as.data.frame(x_train))
data.frame(
  RMSE = caret::RMSE(y_pred_train, y_train),
  Rsquare = caret::R2(y_pred_train, y_train)
)
```

```
##        RMSE   Rsquare
## 1 0.6669065 0.5629431
```

The execution time of each model

```
print("Multiple Linear Regression Model")
```

```
## [1] "Multiple Linear Regression Model"
```

```
mlr.time
```

```
## Time difference of 0.02656293 secs
```

```
print("Principal Component Regressor")
```

```
## [1] "Principal Component Regressor"
```

```
pcr.time_taken
```

```
## Time difference of 0.7460201 secs
```

```
print("Ridge Regressor")
```

```
## [1] "Ridge Regressor"
```

```
ridge.time_taken
```

```
## Time difference of 1.282827 secs
```

```
print("Lasso Regressor")
```

```
## [1] "Lasso Regressor"
```

```
lasso.time_taken
```

```
## Time difference of 0.4396169 secs
```

```
print("Support Vector Regressor")
```

```
## [1] "Support Vector Regressor"
```

```
svm.time_taken
```

```
## Time difference of 27.3901 mins
```

```
print("Random Forest Regressor")
```

```
## [1] "Random Forest Regressor"
```

```
rf.time
```

```
## Time difference of 10.05714 mins
```

```
print("Decision Tree Regressor")
```

```
## [1] "Decision Tree Regressor"
```

```
dt.time
```

```
## Time difference of 0.9678252 secs
```

```
print("Neural Network")
```

```
## [1] "Neural Network"
```

```
nn.time
```

```
## Time difference of 16.44814 secs
```

```
print("Bayesian Regression")
```

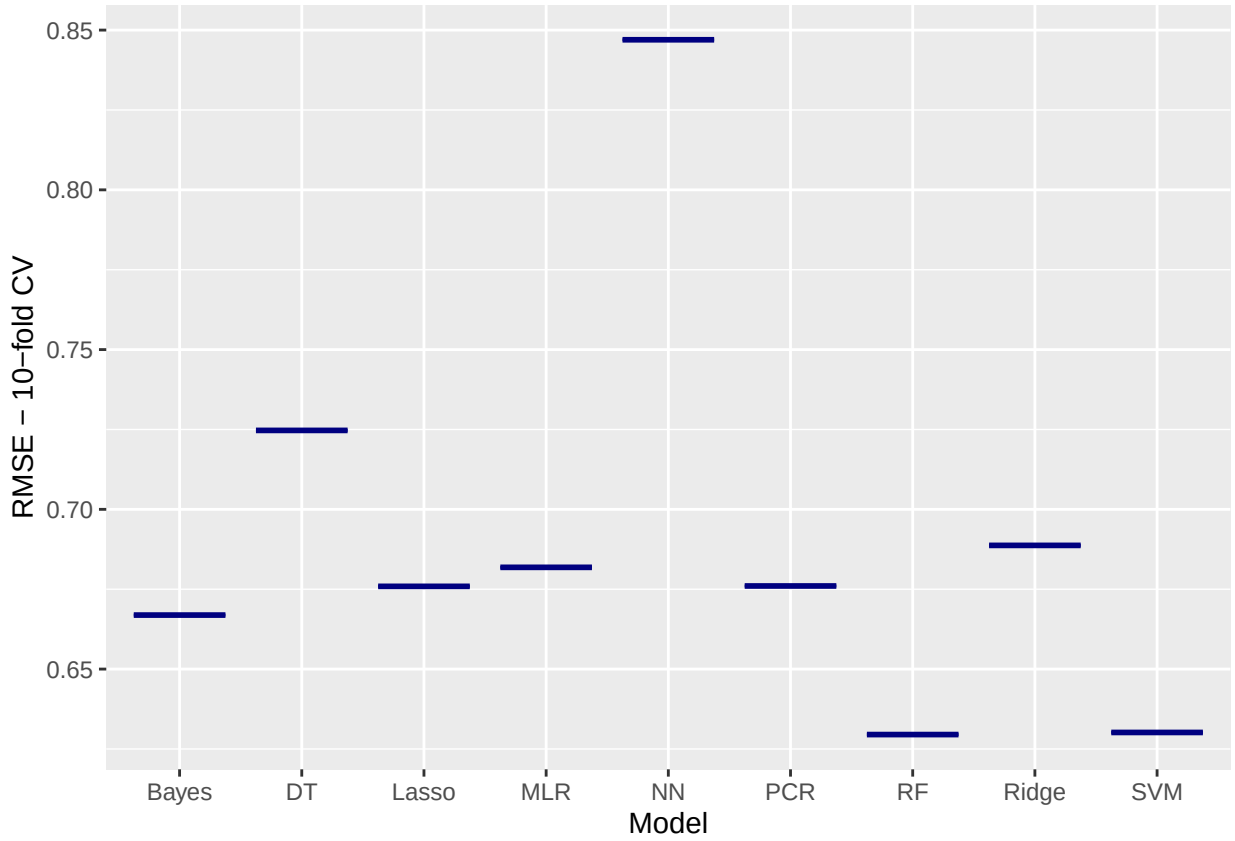
```
## [1] "Bayesian Regression"
```

```
bayes.time
```

```
## Time difference of 5.031238 secs
```

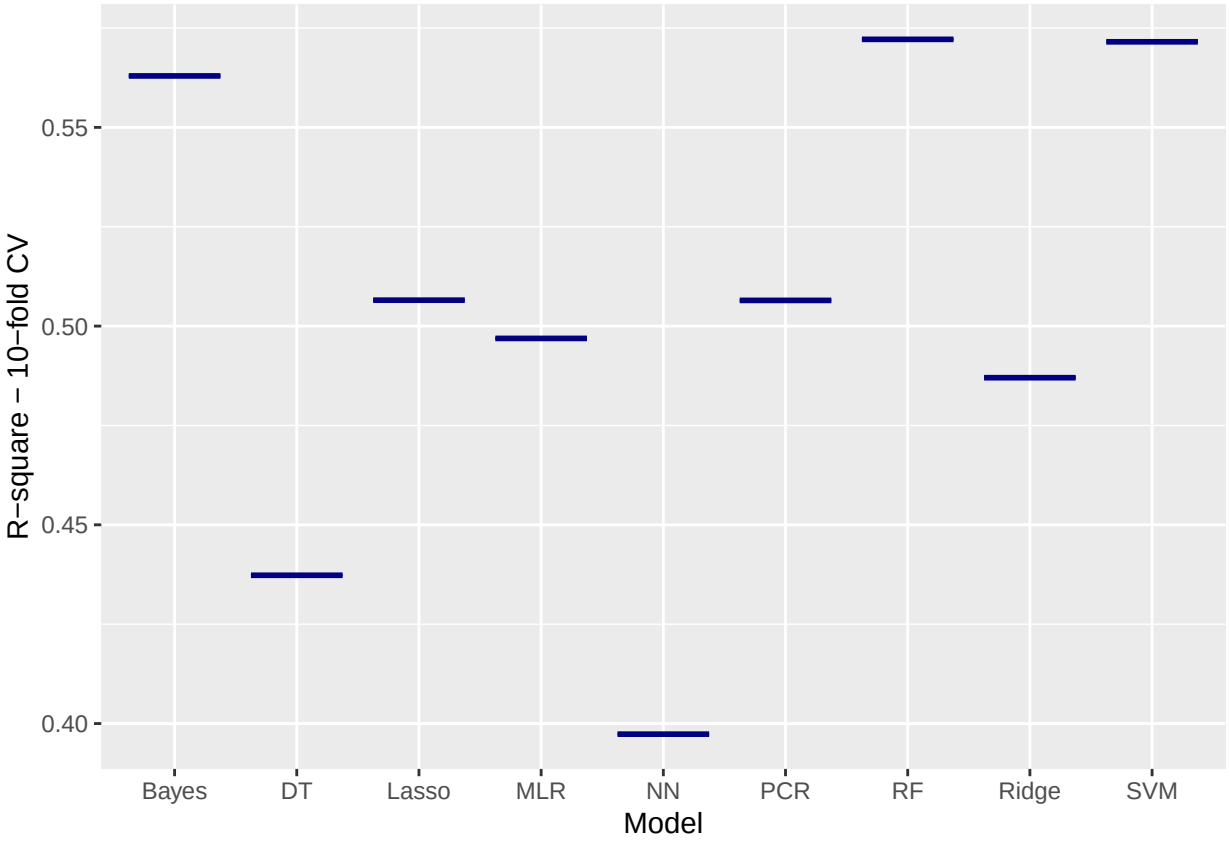
Boxplot representation of RMSE values of each model

```
RMSE_data <- data.frame(Model = c("MLR", "PCR", "Ridge","Lasso","SVM","DT","RF","NN","Bayes"),
                        RMSE = c(0.6818485, 0.6760124 , 0.6887168,0.6758897,0.6301614 ,0.7247068,0.629501

ggplot(RMSE_data, aes(x = Model, y = RMSE)) + geom_boxplot(col = "navy blue") + ylab("RMSE - 10-fold CV
```



The R squared values of all the models are obtained in a boxplot to obtain a clear understanding about the performance of the model.

```
R2_data <- data.frame(Model = c("MLR", "PCR", "Ridge","Lasso","SVM","DT","RF","NN","Bayes"),
                      Rsquare = c(0.4968994, 0.5064648  , 0.4870175,0.5065215,0.5715283 ,   0.4373023,0

ggplot(R2_data, aes(x = Model, y = Rsquare)) + geom_boxplot(col = "navy blue") + ylab("R-square - 10-fo
```


R−square − 10−fold CV
0.55
0.50
0.45
0.40
Bayes
DT
Lasso
MLR
NN
PCR
RF
Ridge
SVM
Model