

Random Forests tutorial

Gabriel Dansereau

March 5th 2020

Acknowledgments:

Awesome resources who inspired this tutorial, check them out!: - https://uc-r.github.io/random_forests - <https://koalaverse.github.io/machine-learning-in-R/random-forest.html> - <https://www.blopig.com/blog/2017/04/a-very-basic-introduction-to-random-forests-using-r/> - <https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5>

Random forest example

Why are Random Forests a good choice: - Easy to use & work with - Easy to program - Fast - Good performance - Not necessarily a lot of tuning - Based on a simple concept

What are Random Forests? - Models trained to predict categorical/numeric variable - Easy to run, good predictive performance without a lot of tuning, fast implementation - An ensemble method: aggregating predictions made by single models (decision trees) - 2 special features: bootstrap aggregating, variable randomization - Can determine variable importance

Data Preparation

```
# Load required packages
library(randomForest) # main package for random forests

## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.

library(rpart) # package for single decision trees
library(caret) # only needed for one function in this case

## Loading required package: lattice
## Loading required package: ggplot2
##
## Attaching package: 'ggplot2'

## The following object is masked from 'package:randomForest':
##
##     margin

## Prepare data
# Load datasets
red_dataset <- read.csv("http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv")
white_dataset <- read.csv("http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv")

# Create working dataframes
```

```

red_wine <- red_dataset
white_wine <- white_dataset

# Add wine type
red_wine$type <- factor("red")
white_wine$type <- factor("white")
# Extract index of wine type
type_ind <- which(names(white_wine) == "type")

# Join datasets
both_wine <- rbind(red_wine, white_wine)

# Backup dataframe with quality as numeric
both_wine_numeric <- both_wine
# Convert quality to factor
both_wine$quality <- as.factor(both_wine$quality)
# Extract index of wine quality
quality_ind <- which(names(both_wine) == "quality")

# Take a peak at the data
head(both_wine)

##      fixed.acidity volatile.acidity citric.acid residual.sugar chlorides
## 1           7.4           0.70           0.00           1.9       0.076
## 2           7.8           0.88           0.00           2.6       0.098
## 3           7.8           0.76           0.04           2.3       0.092
## 4          11.2           0.28           0.56           1.9       0.075
## 5           7.4           0.70           0.00           1.9       0.076
## 6           7.4           0.66           0.00           1.8       0.075
##      free.sulfur.dioxide total.sulfur.dioxide density    pH sulphates alcohol
## 1              11              34 0.9978 3.51       0.56      9.4
## 2              25              67 0.9968 3.20       0.68      9.8
## 3              15              54 0.9970 3.26       0.65      9.8
## 4              17              60 0.9980 3.16       0.58      9.8
## 5              11              34 0.9978 3.51       0.56      9.4
## 6              13              40 0.9978 3.51       0.56      9.4
##      quality type
## 1          5 red
## 2          5 red
## 3          5 red
## 4          6 red
## 5          5 red
## 6          5 red

tail(both_wine)

##      fixed.acidity volatile.acidity citric.acid residual.sugar chlorides
## 6492           6.5           0.23           0.38           1.3       0.032
## 6493           6.2           0.21           0.29           1.6       0.039
## 6494           6.6           0.32           0.36           8.0       0.047
## 6495           6.5           0.24           0.19           1.2       0.041
## 6496           5.5           0.29           0.30           1.1       0.022
## 6497           6.0           0.21           0.38           0.8       0.020
##      free.sulfur.dioxide total.sulfur.dioxide density    pH sulphates
## 6492              29              112 0.99298 3.29       0.54

```

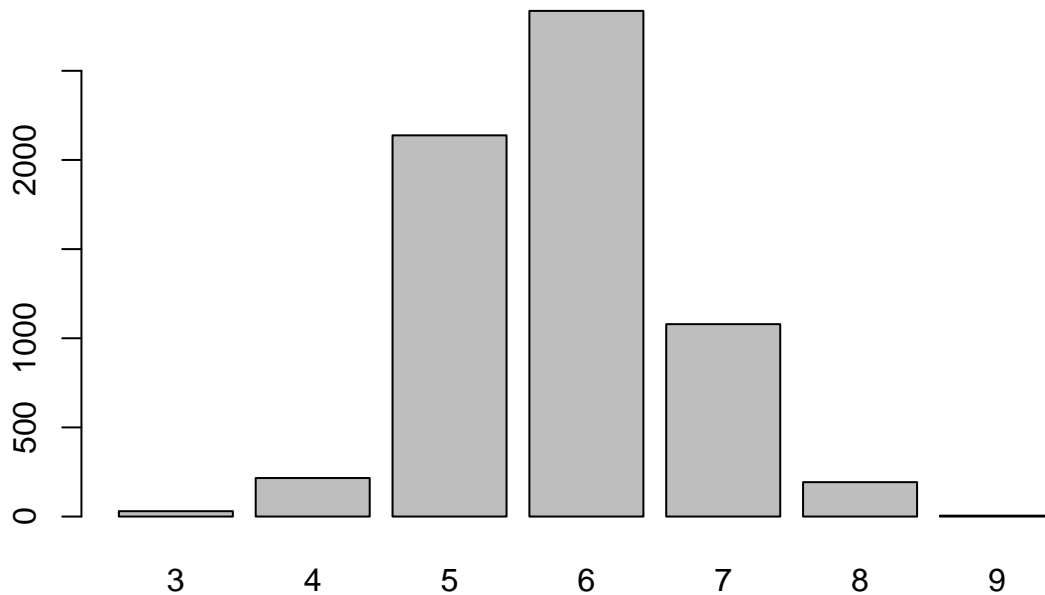
```
## 6493          24          92 0.99114 3.27      0.50
## 6494          57         168 0.99490 3.15      0.46
## 6495          30         111 0.99254 2.99      0.46
## 6496          20         110 0.98869 3.34      0.38
## 6497          22          98 0.98941 3.26      0.32
##      alcohol quality  type
## 6492      9.7       5 white
## 6493     11.2       6 white
## 6494      9.6       5 white
## 6495      9.4       6 white
## 6496     12.8       7 white
## 6497     11.8       6 white
```

```
# Check data structure
```

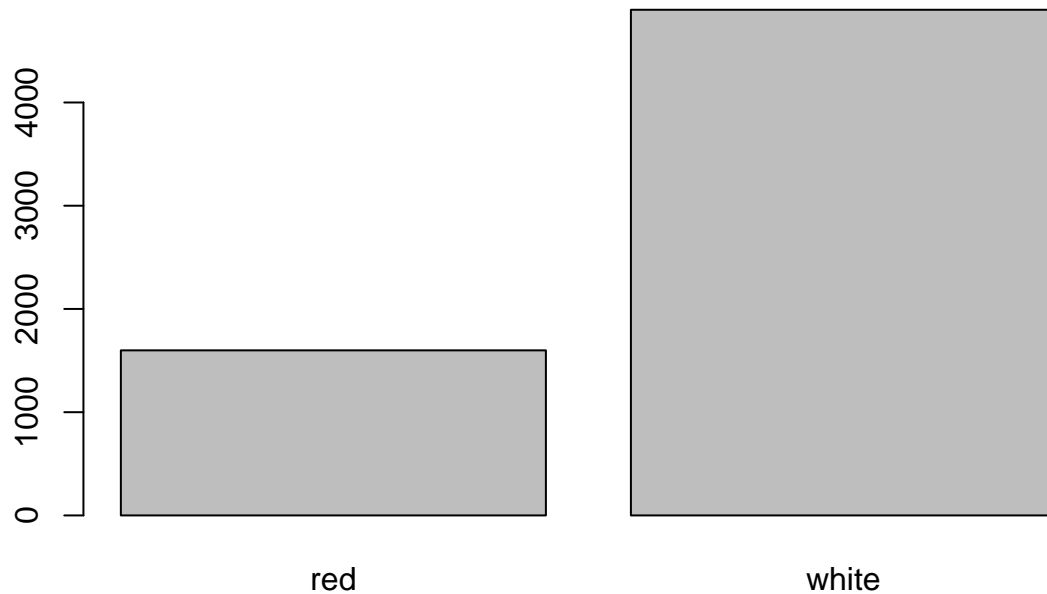
```
table(both_wine$type, both_wine$quality)
```

```
##
##           3    4    5    6    7    8    9
##  red      10   53  681  638  199   18   0
##  white    20  163 1457 2198  880  175   5
```

```
barplot(table(both_wine$quality))
```



```
barplot(table(both_wine$type))
```

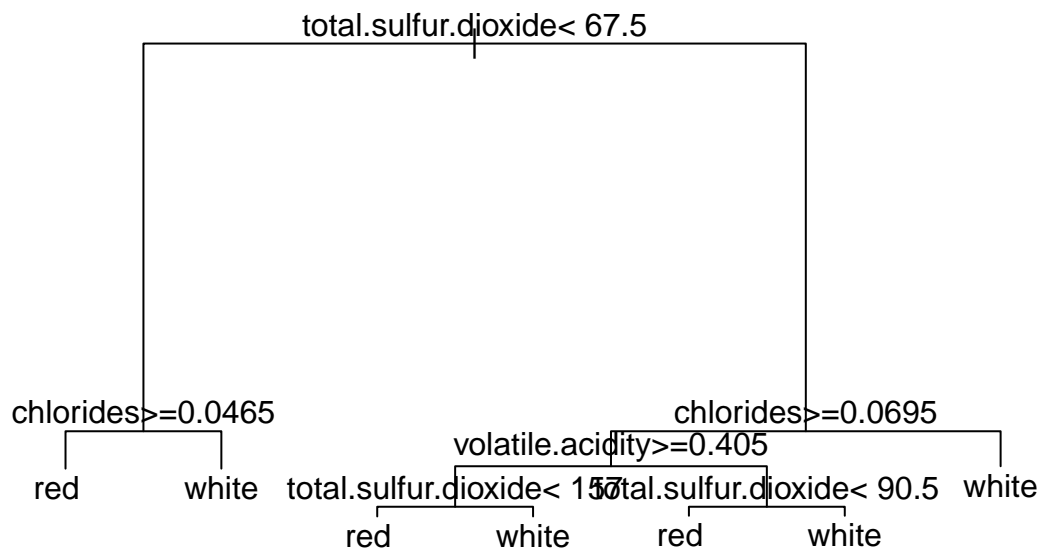


Decision Tree Example

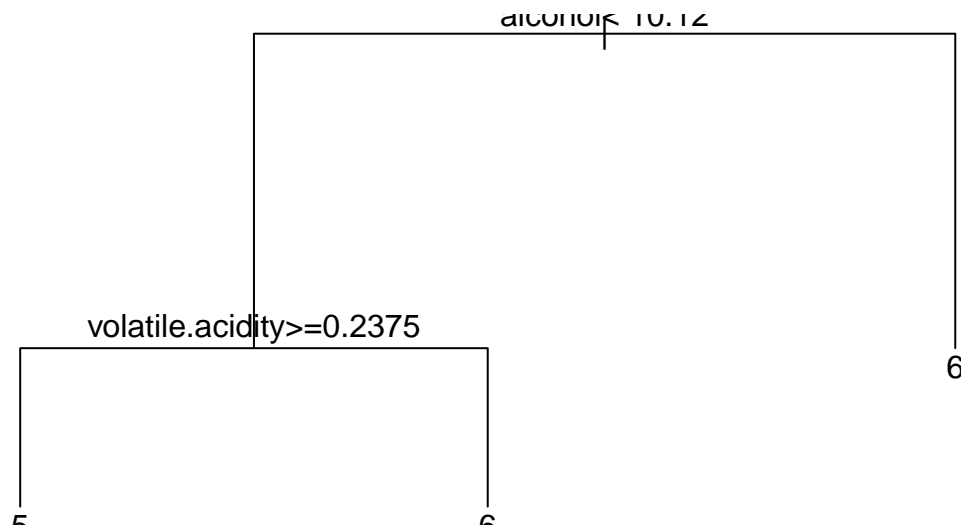
Decision Trees: - Predict a categorical/numeric response variable - Split points: nodes where decision is made
 - Terminal/leaf node: node without descendant - Pros: handling huge datasets, mixed predictors, ignore redundant variables - Cons: unstable trees, poor predictive accuracy & unstable predictions, high variance

Criterion & Best Split: - Gini Impurity: how often random element incorrectly labeled if labeled randomly, given label distribution - $\sum(f_i * (1 - f_i))$ - Entropy: uncertainty in data set S , \sim data set (S), classes in S (X), ratio elem in x to elem in S $p(x)$, 0 = perfect classification - Information gain: how much uncertainty reduced after splitting S on attribute A (diff in entropy) - Regression trees: minimize overall SSE (value vs constant for all trees) - Pruning, optimal subtree & cost complexity criterion: grow deep, prune back based on criterion (ex. number of terminal nodes)

```
## Decision Tree Example
set.seed(42)
decision_tree <- rpart(type ~ ., data = both_wine)
plot(decision_tree)
text(decision_tree, xpd = NA)
```



```
set.seed(42)
decision_tree <- rpart(as.factor(quality) ~ ., data = both_wine)
plot(decision_tree)
text(decision_tree)
```



```
# library(png)
# img <- readPNG("./code/random-forests.png")
# grid::grid.raster(img)
```

Modelling wine type - Random Forest

Ensemble method: - Way to aggregate method from various models, should return better one

Random forests: - Bagging (bootstrap aggregating): sampling with replacement, predict unseen samples by average/majority) - BUT still tree correlation, ~ same structure as all vars available at each split, doesn't optimally reduce variance of predictive values - Random Subspace/split-variable randomization: random subset of m features (variables) at each split (feature bagging) - Corrects overfitting, decorrelates trees, reduces variance - m : $m = p/3$ for regression trees, $m = \sqrt{p}$ for classification (?), $m = p$ is bagging - OOB estimates: no need for cross-validation or separate test set for unbiased estimate of test set error, because estimated internally (tested in 1/3 of trees) - Efficient, no data sacrificed - BUT still differences between OOB error vs test error -> test set important if multiple models compared, complex loss functions -> cross-validation still possible - Variable importance: count correct OOB cases, permute var, recount correct -> big difference = important variable - Local importance score: ~ same as variable importance with percentage of votes instead of counts (?) - Overfitting: RF models more robust, possible with deep trees, but adding trees generally increases performance

```
### Prep the data
train_inds <- sample(1:nrow(both_wine), 0.7*nrow(both_wine))
train_wine <- both_wine[train_inds,]
table(train_wine$type, train_wine$quality)
```

```
##
##           3     4     5     6     7     8     9
##  red       9    35   468  450  138   12    0
##  white    15   107  1036 1517  635  122    3
```

```
valid_wine <- both_wine[-train_inds,]
table(train_wine$type, train_wine$quality)
```

```
##
##           3      4      5      6      7      8      9
##  red       9     35    468    450    138    12     0
##  white    15    107   1036   1517    635   122     3

table(valid_wine$type, valid_wine$quality)

##
##           3      4      5      6      7      8      9
##  red       1     18   213   188     61     6     0
##  white     5     56   421   681   245    53     2

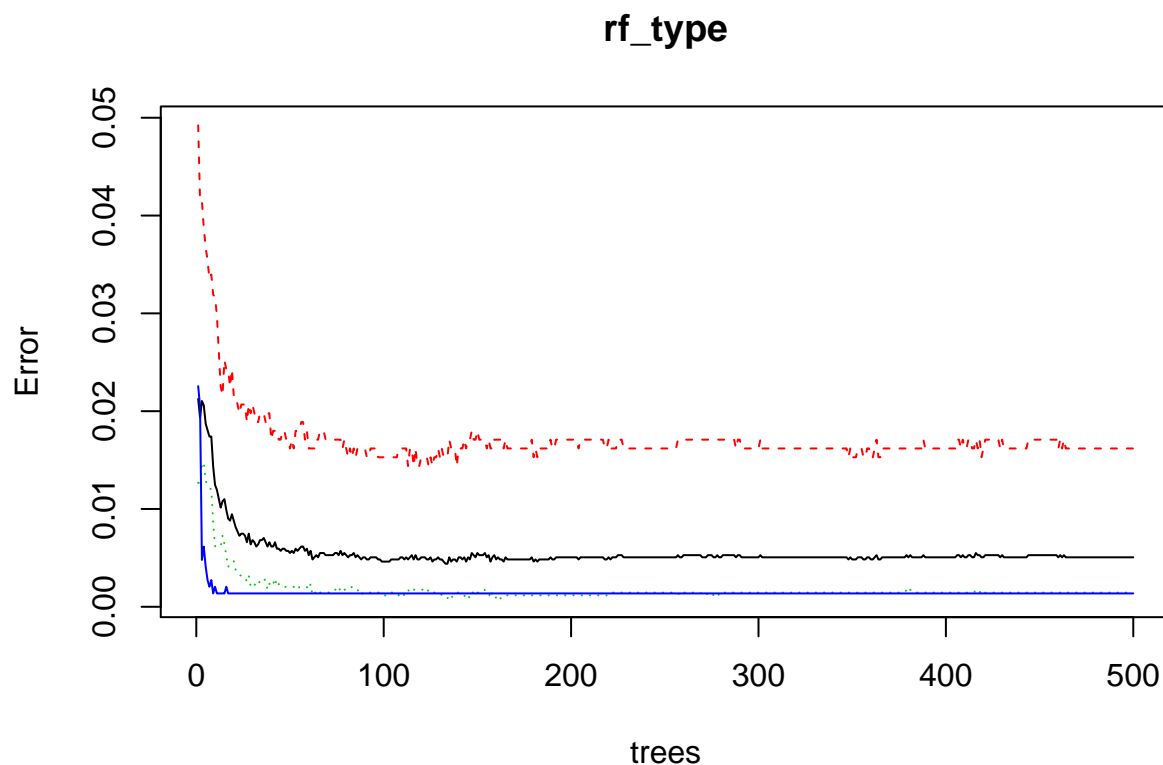
## Random Forest with default parameters
set.seed(42)
rf_type <- randomForest(type ~ ., data = train_wine, importance = TRUE)
rf_type

##
## Call:
## randomForest(formula = type ~ ., data = train_wine, importance = TRUE)
##           Type of random forest: classification
##           Number of trees: 500
## No. of variables tried at each split: 3
##
##           OOB estimate of  error rate: 0.51%
## Confusion matrix:
##           red white class.error
## red   1094     18 0.016187050
## white     5   3430 0.001455604

rf_type_test <- randomForest(type ~ ., data = train_wine, importance = TRUE,
                             xtest = valid_wine[, -type_ind], ytest = valid_wine$type)
rf_type_test

##
## Call:
## randomForest(formula = type ~ ., data = train_wine, importance = TRUE,           xtest = valid_wine[, -t
##           Type of random forest: classification
##           Number of trees: 500
## No. of variables tried at each split: 3
##
##           OOB estimate of  error rate: 0.48%
## Confusion matrix:
##           red white class.error
## red   1095     17 0.015287770
## white     5   3430 0.001455604
##           Test set error rate: 0.62%
## Confusion matrix:
##           red white class.error
## red    477     10 0.020533881
## white     2   1461 0.001367054

plot(rf_type)
lines(1:500, rf_type_test$test$err.rate[,3], col="blue")
```



```
pred_type <- predict(rf_type, valid_wine)
confusionMatrix(pred_type, valid_wine$type)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  red white
##      red    479     2
##      white     8  1461
##
##              Accuracy : 0.9949
##              95% CI : (0.9906, 0.9975)
##      No Information Rate : 0.7503
##      P-Value [Acc > NIR] : <2e-16
##
##              Kappa : 0.9863
##
##  Mcnemar's Test P-Value : 0.1138
##
##              Sensitivity : 0.9836
##              Specificity : 0.9986
##              Pos Pred Value : 0.9958
##              Neg Pred Value : 0.9946
##              Prevalence : 0.2497
##              Detection Rate : 0.2456
##              Detection Prevalence : 0.2467
##              Balanced Accuracy : 0.9911
##
##              'Positive' Class : red
##
```

```
## Evaluate variable importance
```

```
# Show values
```

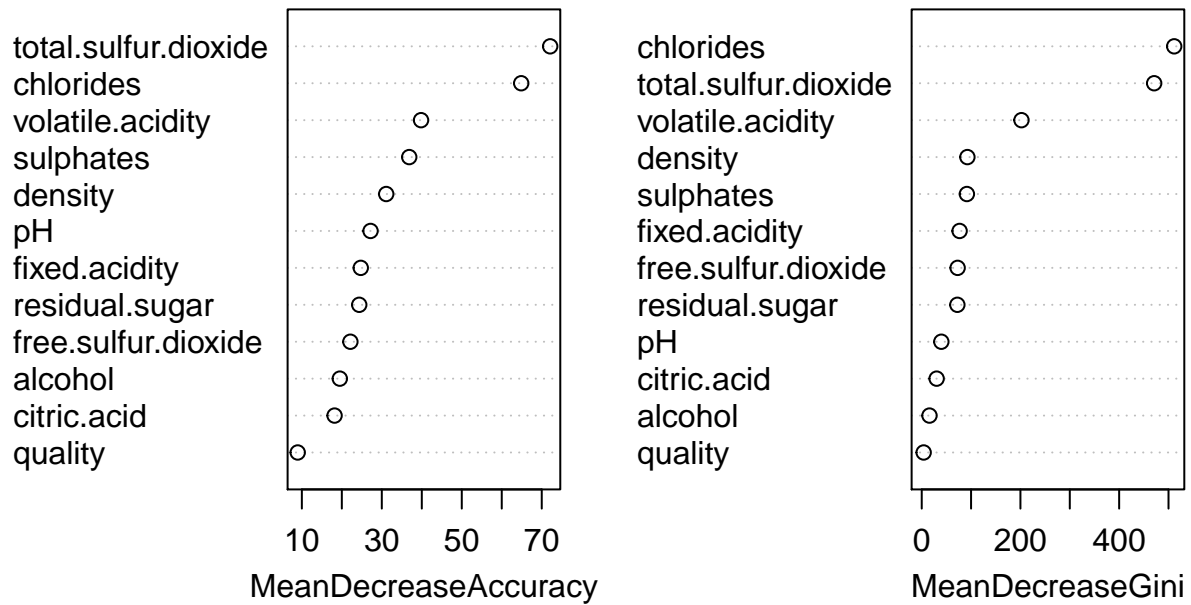
```
importance(rf_type)
```

##	red	white	MeanDecreaseAccuracy
## fixed.acidity	17.744607	19.494451	24.736270
## volatile.acidity	32.098593	29.926100	39.817575
## citric.acid	13.717021	15.045659	18.184691
## residual.sugar	24.687933	13.396438	24.351897
## chlorides	56.364875	46.980504	64.869303
## free.sulfur.dioxide	13.975108	18.733853	22.143245
## total.sulfur.dioxide	59.883587	52.049825	72.111832
## density	33.106756	18.063159	31.103137
## pH	22.566391	20.580303	27.180315
## sulphates	30.035325	30.733502	36.857690
## alcohol	17.753073	14.158994	19.498302
## quality	6.129025	7.280307	8.964397
##	MeanDecreaseGini		
## fixed.acidity	76.73472		
## volatile.acidity	202.08023		
## citric.acid	30.23785		
## residual.sugar	72.21333		
## chlorides	511.54393		
## free.sulfur.dioxide	72.74952		
## total.sulfur.dioxide	470.93669		
## density	92.54132		
## pH	39.74136		
## sulphates	91.48173		
## alcohol	15.74687		
## quality	3.92047		

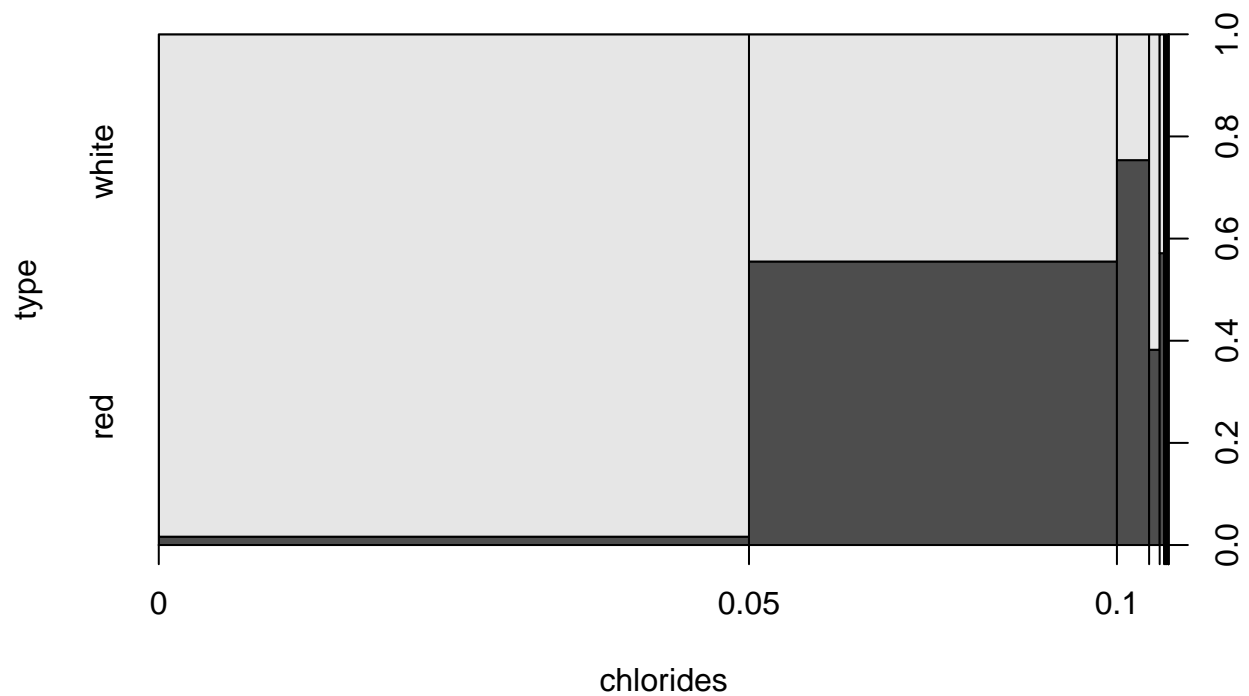
```
# Plot importance values
```

```
varImpPlot(rf_type) # total.sulfur.dioxides & chlorides are clearly the best
```

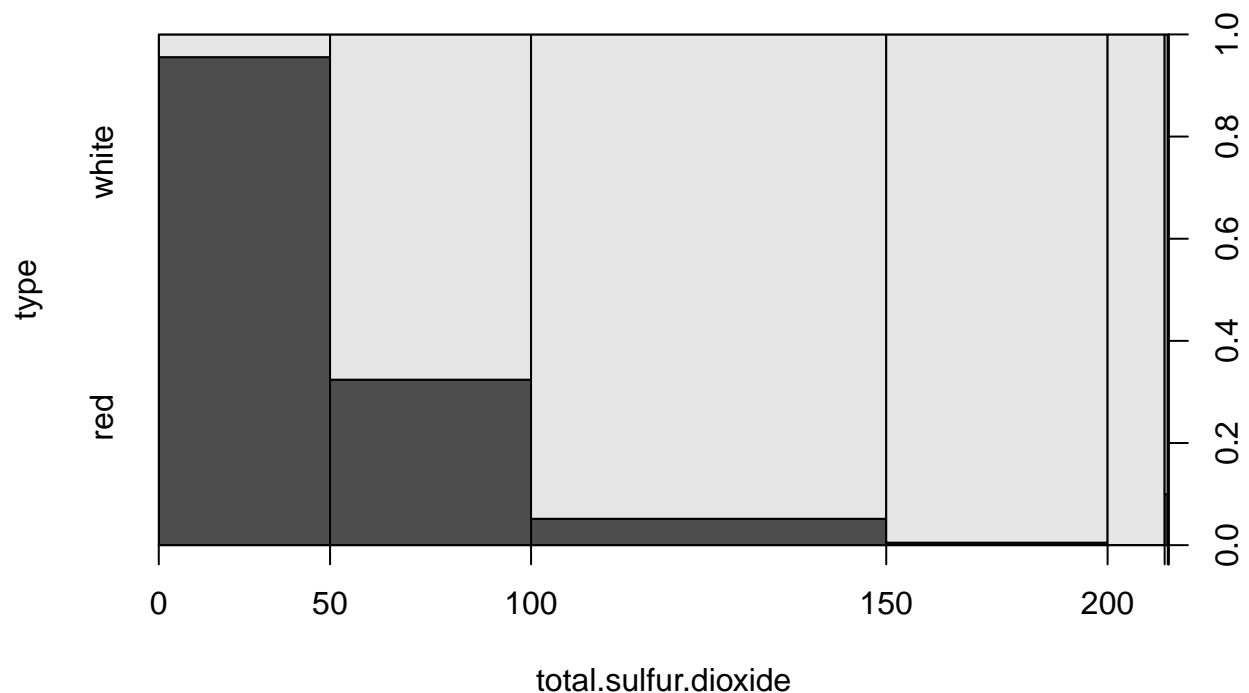

rf_type



```
# Plot type as function of best parameters
plot(type ~ chlorides, data = both_wine)
```



```
plot(type ~ total.sulfur.dioxide, data = both_wine)
```



Modeling wine quality - Random Forest (classification analysis)

Tune model

```
## 1. Model with default parameters
set.seed(42)
rf_qual <- randomForest(quality ~ ., data = both_wine)
rf_qual

##
## Call:
## randomForest(formula = quality ~ ., data = both_wine)
##           Type of random forest: classification
##           Number of trees: 500
## No. of variables tried at each split: 3
##
##           OOB estimate of  error rate: 28.6%
## Confusion matrix:
##   3  4   5   6   7   8  9 class.error
## 3 0  1   17   12   0  0  0  1.0000000
## 4 1 32   111   68   4  0  0  0.8518519
## 5 0  9 1589   529  11  0  0  0.2567820
## 6 0  4  377 2305 148  2  0  0.1872355
## 7 0  0   21  419 631  8  0  0.4151993
## 8 0  0    1   66  44 82  0  0.5751295
## 9 0  0    0    3   2  0  0  1.0000000

# default ntree = 500
# default mtry = 3
# OOB error rate = 28.52%%

## 2. Choose ntree value (where OOB error rate stabilizes at minimum)
```

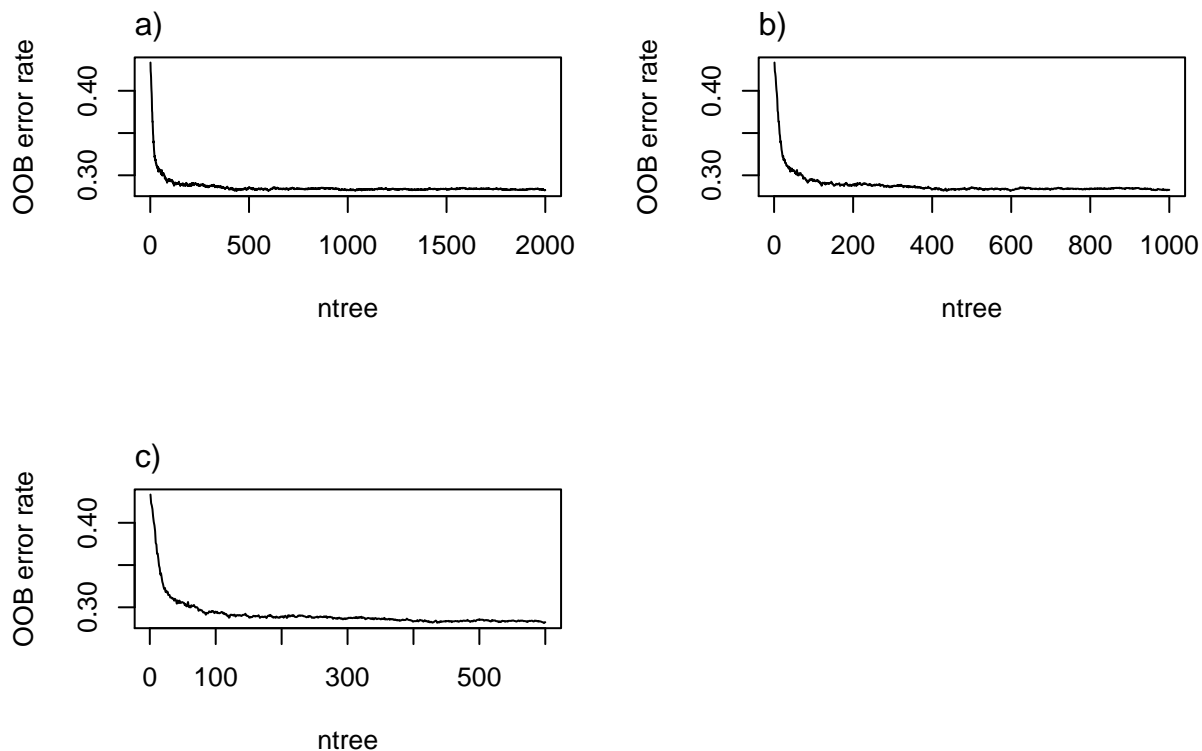
```

# Test model with ntree = 2000, 1000, and 600
set.seed(42)
rf_qual_2000 <- randomForest(as.factor(quality) ~ ., data = both_wine, ntree = 2000)
set.seed(42)
rf_qual_1000 <- randomForest(as.factor(quality) ~ ., data = both_wine, ntree = 1000)
set.seed(42)
rf_qual_600 <- randomForest(as.factor(quality) ~ ., data = both_wine, ntree = 600)

# Plot error rate ~ ntree

par(mfrow=c(2,2))
plot(rf_qual_2000$err.rate[, "OOB"], type = "l", xlab = "ntree",
     ylab = "OOB error rate")
mtext("a)", line = 0.5, adj = 0)
plot(rf_qual_1000$err.rate[, "OOB"], type = "l", xlab = "ntree",
     ylab = "OOB error rate")
mtext("b)", line = 0.5, adj = 0)
plot(rf_qual_600$err.rate[, "OOB"], type = "l", xlab = "ntree",
     ylab = "OOB error rate")
mtext("c)", line = 0.5, adj = 0)
par(mfrow=c(1,1))

```



```

# stabilizes around n=500, same as default value. Let's select it

## 3. Find optimal mtry value
set.seed(42)
mtry <- tuneRF(x = both_wine[-which(names(both_wine)=="quality")],
               y = as.factor(both_wine$quality),
               ntreeTry=500, mtryStart = 5, stepFactor=1.5, improve=0.01, trace=TRUE, plot=TRUE)

```

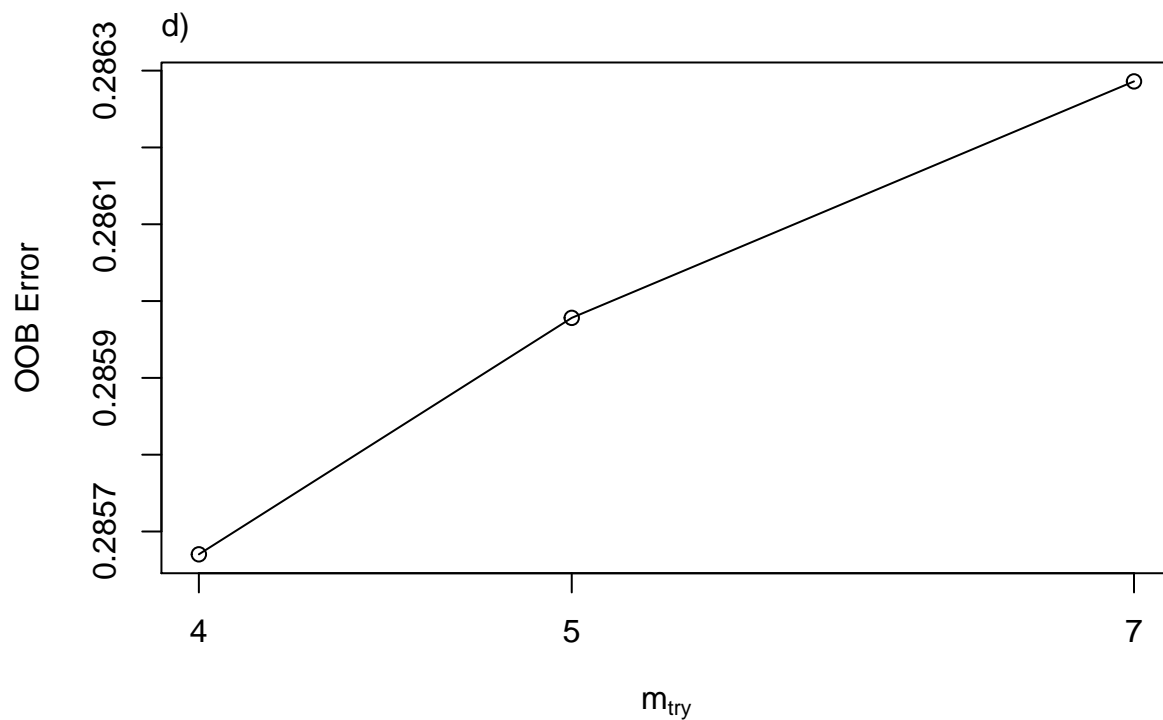
```
## mtry = 5   OOB error = 28.6%
## Searching left ...
## mtry = 4     OOB error = 28.57%
## 0.001076426 0.01
## Searching right ...
## mtry = 7     OOB error = 28.63%
## -0.001076426 0.01
```

```
# Plot mtry values
```

```
mtry
```

```
##      mtry  OOBError
## 4.00B    4 0.2856703
## 5.00B    5 0.2859781
## 7.00B    7 0.2862860
```

```
mtext("d)", line = 0.5, adj = 0)
```



```
# Best mtry = 3, same as default
```

Modeling wine quality - Random Forest (regression analysis)

```
## Modeling wine quality - Random Forest (regression analysis)
```

```
## Random forest with default parameters
```

```
set.seed(42)
```

```
rf_qual_reg <- randomForest(quality ~ ., data = both_wine_numeric, importance=TRUE)
```

```
rf_qual_reg
```

```
##
```

```
## Call:
```

```
## randomForest(formula = quality ~ ., data = both_wine_numeric, importance = TRUE)
```

```

##                Type of random forest: regression
##                Number of trees: 500
## No. of variables tried at each split: 4
##
##                Mean of squared residuals: 0.3388956
##                % Var explained: 55.55

# default ntree = 500
# default mtry = 4
# Mean of squared residuals = 0.339
# % Var explained = 55.55

# mtry <- tuneRF(x = both_wine_numeric[, -quality_ind],
#                y = both_wine_numeric$quality,
#                ntreeTry=500, mtryStart = 5, stepFactor=1.5, improve=0.01, trace=TRUE, plot=TRUE)

## Verify model accuracy with predicted values as integers
# Extract OOB testing and predicted values from model
testing_qual_reg <- rf_qual_reg$y
predicted_qual_reg <- rf_qual_reg$predicted

## Function to rearrange numeric predictions to integers as in
## Cortez et al. 2009
# Tolerance range is used to determine if predicted value is correct
rf_reg_accuracy <- function(tolerance){ # Wrapped in a function

  # Loop for all predicted values
  for(j in 1:length(predicted_qual_reg)){

    # Conditional operators to determine if predicted value is within tolerance
    # range of real value
    if(abs(predicted_qual_reg[j]-testing_qual_reg[j]) < tolerance) {
      # if TRUE, predicted value is within tolerance range and considered
      # correct; # hence, real value is selected
      predicted_qual_reg[j] <- testing_qual_reg[j]
    } else {
      # if FALSE, predicted value is incorrect; hence, predicted value is simply
      # rounded to closest integer
      predicted_qual_reg[j] <- round(predicted_qual_reg[j], digits = 0)
    }

  }

  # Define datasets as factors with same levels (warnings if not)
  testing_qual_reg <- as.factor(testing_qual_reg)
  predicted_qual_reg <- factor(predicted_qual_reg,
                              levels = levels(testing_qual_reg))

  # Check confusion matrix of predicted values
  print(confusionMatrix(predicted_qual_reg, testing_qual_reg))
}

# Verify model accuracy
library(caret)
rf_reg_accuracy(tolerance = 0.5) # accuracy = 0.700, similar to classification

```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    3    4    5    6    7    8    9
##           3    0    0    0    0    0    0
##           4    1    6    2    0    0    0
##           5   19   160 1574   358   12    3
##           6   10    48   551 2324   465   50
##           7    0    2    11   154   602  107
##           8    0    0    0    0    0   33
##           9    0    0    0    0    0    0
##
## Overall Statistics
##
##           Accuracy : 0.6986
##           95% CI : (0.6873, 0.7098)
##           No Information Rate : 0.4365
##           P-Value [Acc > NIR] : < 2.2e-16
##
##           Kappa : 0.5276
##
## Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##           Class: 3  Class: 4  Class: 5  Class: 6  Class: 7
## Sensitivity      0.000000 0.0277778  0.7362  0.8195  0.55792
## Specificity      1.000000 0.9995224  0.8734  0.6927  0.94869
## Pos Pred Value      NaN 0.6666667  0.7404  0.6738  0.68409
## Neg Pred Value      0.995382 0.9676326  0.8710  0.8320  0.91508
## Prevalence        0.004618 0.0332461  0.3291  0.4365  0.16608
## Detection Rate      0.000000 0.0009235  0.2423  0.3577  0.09266
## Detection Prevalence 0.000000 0.0013853  0.3272  0.5309  0.13545
## Balanced Accuracy   0.500000 0.5136501  0.8048  0.7561  0.75331
##           Class: 8  Class: 9
## Sensitivity      0.170984 0.0000000
## Specificity      1.000000 1.0000000
## Pos Pred Value      1.000000      NaN
## Neg Pred Value      0.975248 0.9992304
## Prevalence        0.029706 0.0007696
## Detection Rate      0.005079 0.0000000
## Detection Prevalence 0.005079 0.0000000
## Balanced Accuracy   0.585492 0.5000000
```

```
rf_reg_accuracy(tolerance = 1.0) # accuracy = 0.913
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    3    4    5    6    7    8    9
##           3    0    0    0    0    0    0
##           4    1   69    0    0    0    0
##           5   19   97 2019   15   12    3
##           6   10   48  108 2814  119   50
##           7    0    2   11    7  948   51
```

```
##           8      0      0      0      0      0      89      0
##           9      0      0      0      0      0      0      0
##
## Overall Statistics
##
##           Accuracy : 0.9141
##           95% CI : (0.907, 0.9208)
##           No Information Rate : 0.4365
##           P-Value [Acc > NIR] : < 2.2e-16
##
##           Kappa : 0.8682
##
## McNemar's Test P-Value : NA
##
## Statistics by Class:
##
##           Class: 3 Class: 4 Class: 5 Class: 6 Class: 7 Class: 8
## Sensitivity      0.000000  0.31944  0.9443  0.9922  0.8786  0.46114
## Specificity      1.000000  0.99984  0.9665  0.9082  0.9862  1.00000
## Pos Pred Value    NaN      0.98571  0.9326  0.8933  0.9267  1.00000
## Neg Pred Value    0.995382  0.97713  0.9725  0.9934  0.9761  0.98377
## Prevalence        0.004618  0.03325  0.3291  0.4365  0.1661  0.02971
## Detection Rate    0.000000  0.01062  0.3108  0.4331  0.1459  0.01370
## Detection Prevalence 0.000000  0.01077  0.3332  0.4848  0.1575  0.01370
## Balanced Accuracy  0.500000  0.65964  0.9554  0.9502  0.9324  0.73057
##           Class: 9
## Sensitivity      0.0000000
## Specificity      1.0000000
## Pos Pred Value    NaN
## Neg Pred Value    0.9992304
## Prevalence        0.0007696
## Detection Rate    0.0000000
## Detection Prevalence 0.0000000
## Balanced Accuracy  0.5000000
```

```
## Evaluate variable importance
```

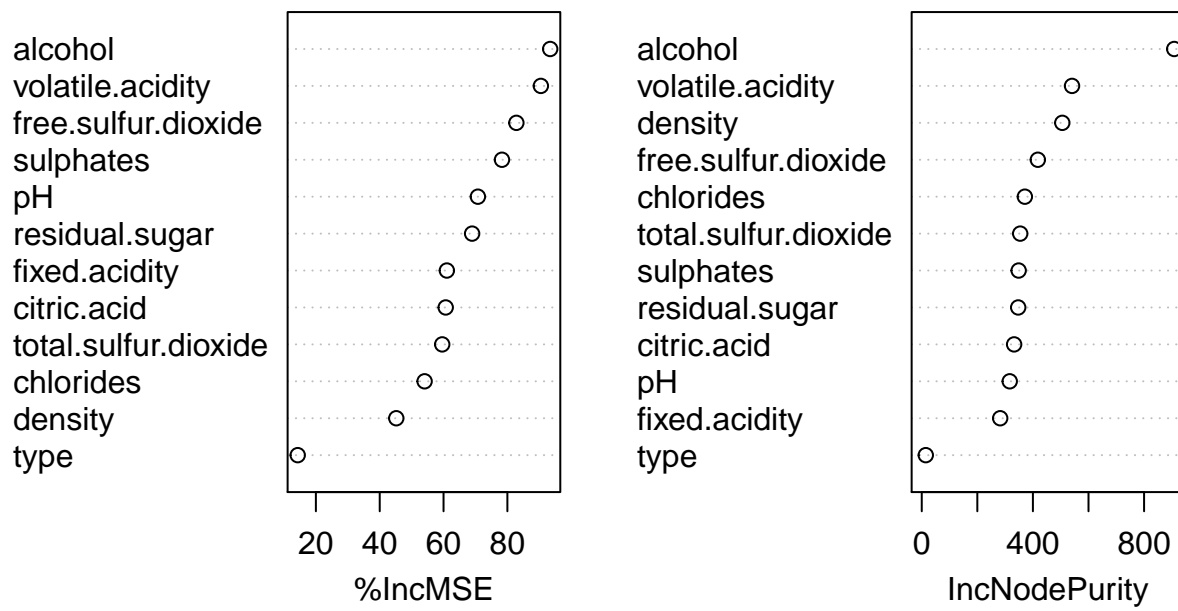
```
# Show values
```

```
importance(rf_qual_reg)
```

```
##           %IncMSE IncNodePurity
## fixed.acidity      61.02999      282.24308
## volatile.acidity    90.48706      540.75877
## citric.acid         60.66902      332.37285
## residual.sugar      68.95193      347.08741
## chlorides           54.06412      371.02795
## free.sulfur.dioxide 82.81470      417.69809
## total.sulfur.dioxide 59.57786      354.29125
## density             45.18951      505.73845
## pH                  70.75994      316.61830
## sulphates           78.29823      348.70583
## alcohol             93.41057      908.41400
## type                14.30684       14.49352
```

```
# Plot importance values
```

```
varImpPlot(rf_qual_reg, main = NULL) # alcohol is the best predictor
```



```
library(ranger)
```

```
Ranger
```

```
##
```

```
## Attaching package: 'ranger'
```

```
## The following object is masked from 'package:randomForest':
```

```
##
```

```
## importance
```

```
# time models
```

```
system.time(rf_qual <- randomForest(as.factor(quality) ~ ., data = train_wine))
```

```
## user system elapsed
```

```
## 3.687 0.040 3.728
```

```
system.time(ranger_qual <- ranger(as.factor(quality) ~ ., data = train_wine, importance = "impurity"))
```

```
## user system elapsed
```

```
## 6.257 0.067 0.640
```

```
## Fine tuning
```

```
# hyperparameter grid search
```

```
hyper_grid <- expand.grid(
```

```
  mtry = seq(2, 6, by = 1),
```

```
  node_size = seq(1, 5, by = 1),
```

```
  sampe_size = c(.55, .632, .70),
```

```
  OOB = 0
```

```
)
```

```
# total number of combinations
```

```
nrow(hyper_grid)
```

```
## [1] 75
```



```
## [1] 160

system.time(
  for(i in 1:nrow(hyper_grid)) {

    # train model
    model <- ranger(formula = as.factor(quality) ~ .,
                     data = train_wine,
                     num.trees = 500,
                     mtry = hyper_grid$mtry[i],
                     min.node.size = hyper_grid$node_size[i],
                     sample.fraction = hyper_grid$sampe_size[i],
                     importance = "impurity",
                     seed = 42
                    )

    # add OOB error to grid
    hyper_grid$OOB[i] <- model$prediction.error
  }
)
```

```
##      user  system elapsed
## 390.122   1.916   39.613
```

```
hyper_grid
```

```
##      mtry node_size sampe_size      OOB
## 1       2         1      0.550 0.3217506
## 2       3         1      0.550 0.3206510
## 3       4         1      0.550 0.3243897
## 4       5         1      0.550 0.3268089
## 5       6         1      0.550 0.3254893
## 6       2         2      0.550 0.3272487
## 7       3         2      0.550 0.3276886
## 8       4         2      0.550 0.3305476
## 9       5         2      0.550 0.3301078
## 10      6         2      0.550 0.3325269
## 11      2         3      0.550 0.3301078
## 12      3         3      0.550 0.3287882
## 13      4         3      0.550 0.3316472
## 14      5         3      0.550 0.3316472
## 15      6         3      0.550 0.3367055
## 16      2         4      0.550 0.3296679
## 17      3         4      0.550 0.3422037
## 18      4         4      0.550 0.3384649
## 19      5         4      0.550 0.3415439
## 20      6         4      0.550 0.3384649
## 21      2         5      0.550 0.3422037
## 22      3         5      0.550 0.3408841
## 23      4         5      0.550 0.3408841
## 24      5         5      0.550 0.3466022
## 25      6         5      0.550 0.3424236
## 26      2         1      0.632 0.3206510
## 27      3         1      0.632 0.3235100
## 28      4         1      0.632 0.3239499
```

## 29	5	1	0.632 0.3237299
## 30	6	1	0.632 0.3215307
## 31	2	2	0.632 0.3208709
## 32	3	2	0.632 0.3241698
## 33	4	2	0.632 0.3230702
## 34	5	2	0.632 0.3268089
## 35	6	2	0.632 0.3268089
## 36	2	3	0.632 0.3285683
## 37	3	3	0.632 0.3290081
## 38	4	3	0.632 0.3235100
## 39	5	3	0.632 0.3285683
## 40	6	3	0.632 0.3259292
## 41	2	4	0.632 0.3263690
## 42	3	4	0.632 0.3347262
## 43	4	4	0.632 0.3272487
## 44	5	4	0.632 0.3358258
## 45	6	4	0.632 0.3303277
## 46	2	5	0.632 0.3349461
## 47	3	5	0.632 0.3362657
## 48	4	5	0.632 0.3345063
## 49	5	5	0.632 0.3369254
## 50	6	5	0.632 0.3345063
## 51	2	1	0.700 0.3202111
## 52	3	1	0.700 0.3237299
## 53	4	1	0.700 0.3221905
## 54	5	1	0.700 0.3210908
## 55	6	1	0.700 0.3309875
## 56	2	2	0.700 0.3208709
## 57	3	2	0.700 0.3257093
## 58	4	2	0.700 0.3221905
## 59	5	2	0.700 0.3237299
## 60	6	2	0.700 0.3305476
## 61	2	3	0.700 0.3217506
## 62	3	3	0.700 0.3246096
## 63	4	3	0.700 0.3254893
## 64	5	3	0.700 0.3259292
## 65	6	3	0.700 0.3301078
## 66	2	4	0.700 0.3259292
## 67	3	4	0.700 0.3246096
## 68	4	4	0.700 0.3303277
## 69	5	4	0.700 0.3285683
## 70	6	4	0.700 0.3303277
## 71	2	5	0.700 0.3316472
## 72	3	5	0.700 0.3331867
## 73	4	5	0.700 0.3294480
## 74	5	5	0.700 0.3360457
## 75	6	5	0.700 0.3371454

AUC