# Unit 4 Lecture 3: Random forests

November 9, 2021

Today, we will learn how to train and tune random forests using the `randomForest` package.

First, let's load some libraries:

```
library(randomForest)       # install.packages("randomForest")
library(tidyverse)
```

## Random forests for regression

We will continue using the `Hitters` data from the `ISLR` package, splitting into training and testing:

```
Hitters = ISLR2::Hitters %>%
  as_tibble() %>%
  filter(!is.na(Salary)) %>%
  mutate(Salary = log(Salary)) # log-transform the salary
Hitters
```

```
## # A tibble: 263 x 20
##    AtBat  Hits HmRun  Runs   RBI Walks Years CAtBat CHits CHmRun CRuns  CRBI
##    <int> <int> <int> <int> <int> <int> <int>  <int> <int>  <int> <int> <int>
##  1   315    81     7    24    38    39    14   3449   835     69   321   414
##  2   479   130    18    66    72    76     3   1624   457     63   224   266
##  3   496   141    20    65    78    37    11   5628  1575    225   828   838
##  4   321    87    10    39    42    30     2    396   101     12    48    46
##  5   594   169     4    74    51    35    11   4408  1133     19   501   336
##  6   185    37     1    23     8    21     2    214    42      1    30     9
##  7   298    73     0    24    24     7     3    509   108      0    41    37
##  8   323    81     6    26    32     8     2    341    86      6    32    34
##  9   401    92    17    49    66    65    13   5206  1332    253   784   890
## 10   574   159    21   107    75    59    10   4631  1300     90   702   504
## # ... with 253 more rows, and 8 more variables: CWalks <int>, League <fct>,
## #   Division <fct>, PutOuts <int>, Assists <int>, Errors <int>, Salary <dbl>,
## #   NewLeague <fct>
```

```
set.seed(1) # set seed for reproducibility
train_samples = sample(1:nrow(Hitters), round(0.8*nrow(Hitters)))
Hitters_train = Hitters %>% filter(row_number() %in% train_samples)
Hitters_test = Hitters %>% filter(!(row_number() %in% train_samples))
```

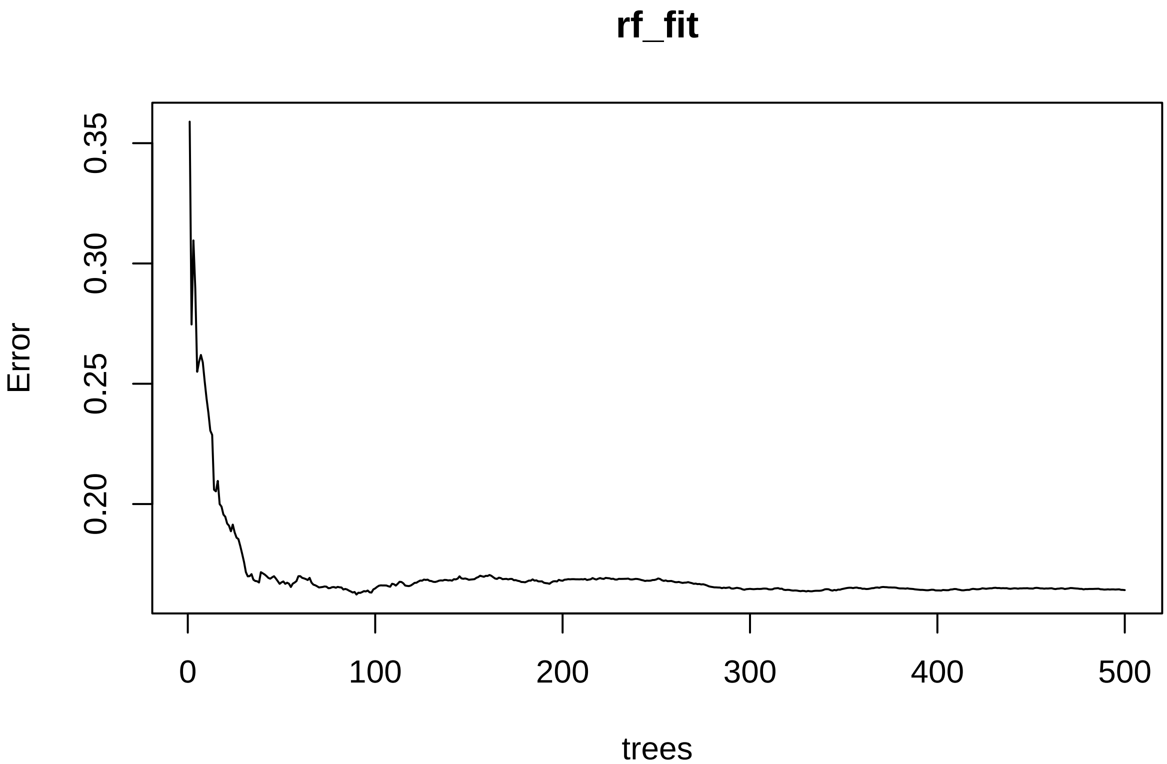
### Training a random forest

To train a random forest with default settings, we use the following syntax:

```
rf_fit = randomForest(Salary ~ ., data = Hitters_train)
?randomForest
```

We can get a quick visualization by using `plot`, which shows us the OOB error as a function of the number of trees.

```
plot(rf_fit)
```



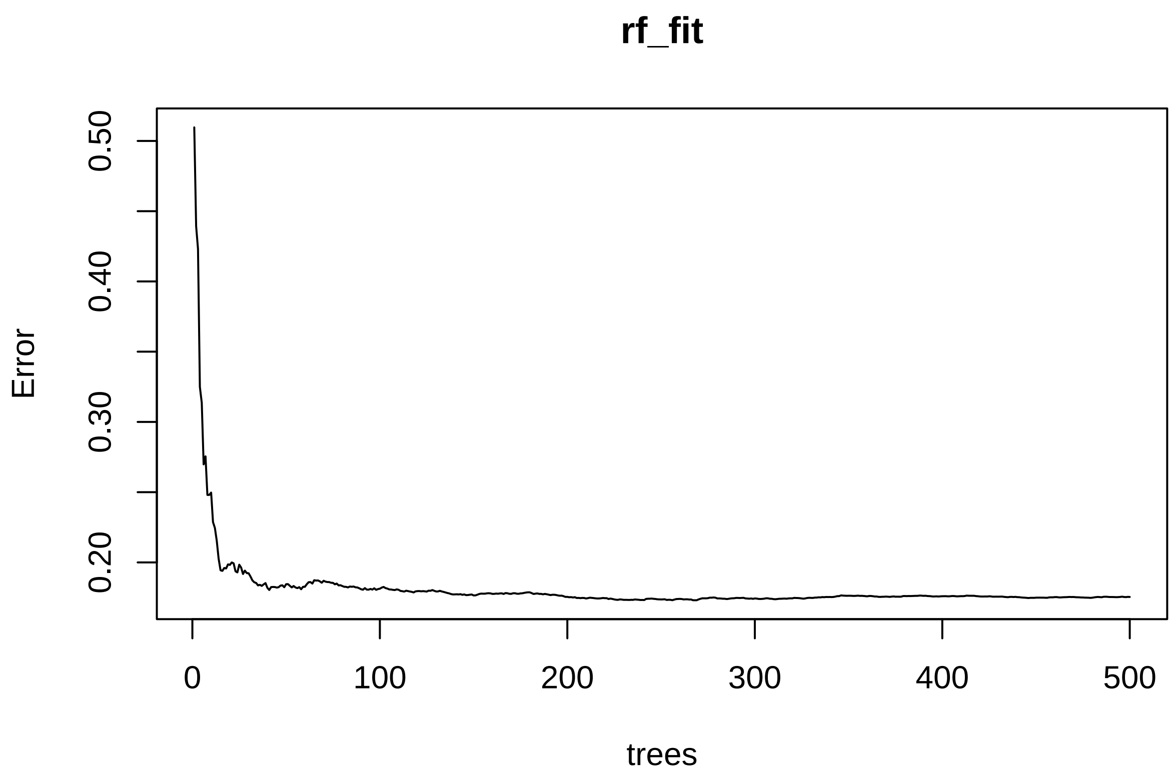

We see that this error stays flat as soon as `B` is large enough (in this case stabilizing around 100).

The key parameters controlling the random forest fit are the following:

- `mtry`: number of variables to sample for each split (called m in lecture), default `floor(p/3)` for regression and `sqrt(p)` for classification
- `nodesize`: minimum size of terminal nodes, default 1 for classification and 5 for regression
- `maxnodes`: maximum number of terminal nodes trees in the forest can have, default no maximum
- `ntree`: number of trees (called B in lecture), default 500

We might want to specify the `mtry` parameter manually. For example, to get the bagging predictions we can set `mtry = 19`, since 19 is the total number of features:

```
rf_fit = randomForest(Salary ~ ., mtry = 19, data = Hitters_train)
plot(rf_fit)
```




### Tuning the random forest

A quick-and-dirty way to tune a random forest is to try out a few different values of `mtry`:

```
rf_3 = randomForest(Salary ~ ., mtry = 3, data = Hitters_train)
rf_6 = randomForest(Salary ~ ., mtry = 6, data = Hitters_train)
rf_19 = randomForest(Salary ~ ., mtry = 19, data = Hitters_train)
```

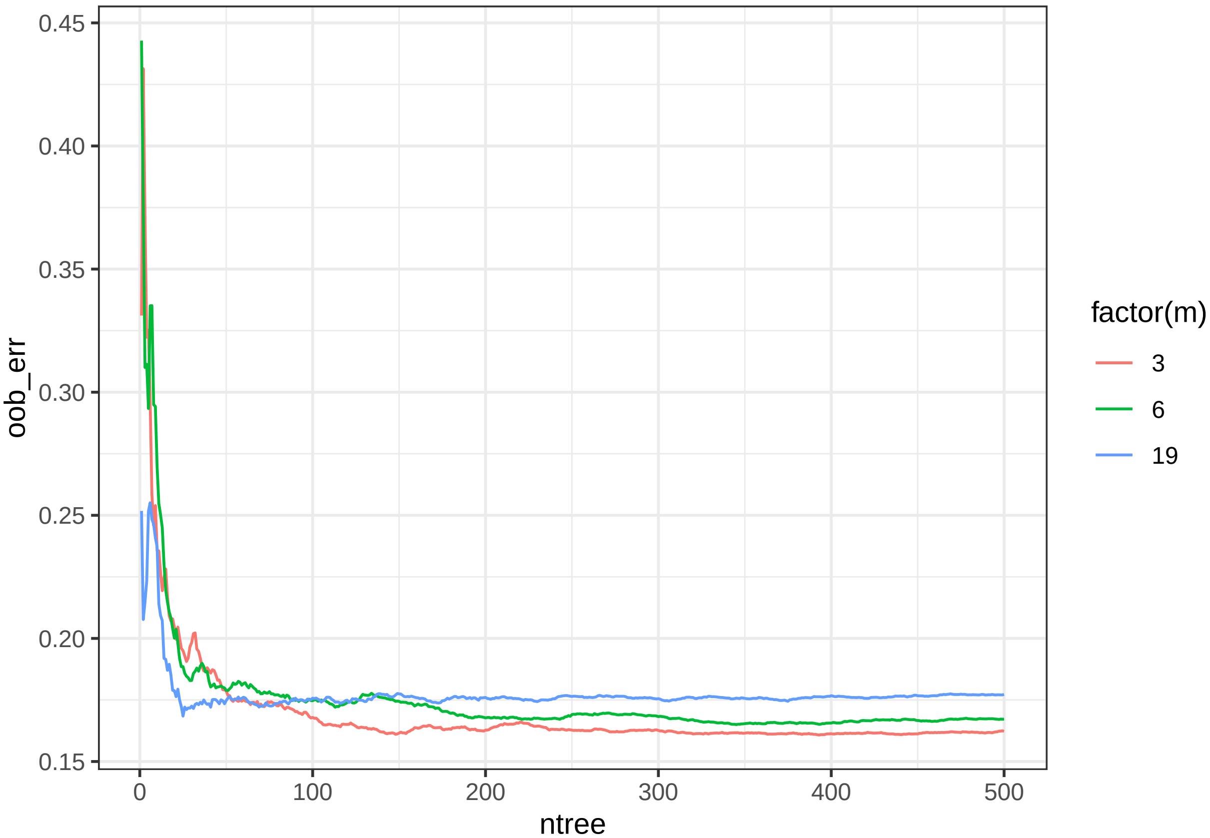
We can extract the OOB errors from each of these objects by using the `mse` field:

```
oob_errors = bind_rows(
  tibble(ntree = 1:500, oob_err = rf_3$mse, m = 3),
  tibble(ntree = 1:500, oob_err = rf_6$mse, m = 6),
  tibble(ntree = 1:500, oob_err = rf_19$mse, m = 19)
)
oob_errors
```

```
## # A tibble: 1,500 x 3
##    ntree oob_err     m
##    <int>   <dbl> <dbl>
##  1     1   0.331     3
##  2     2   0.431     3
##  3     3   0.371     3
##  4     4   0.322     3
##  5     5   0.325     3
##  6     6   0.296     3
##  7     7   0.258     3
##  8     8   0.246     3
##  9     9   0.254     3
## 10    10   0.236     3
## # ... with 1,490 more rows
```

We can then plot these as follows:

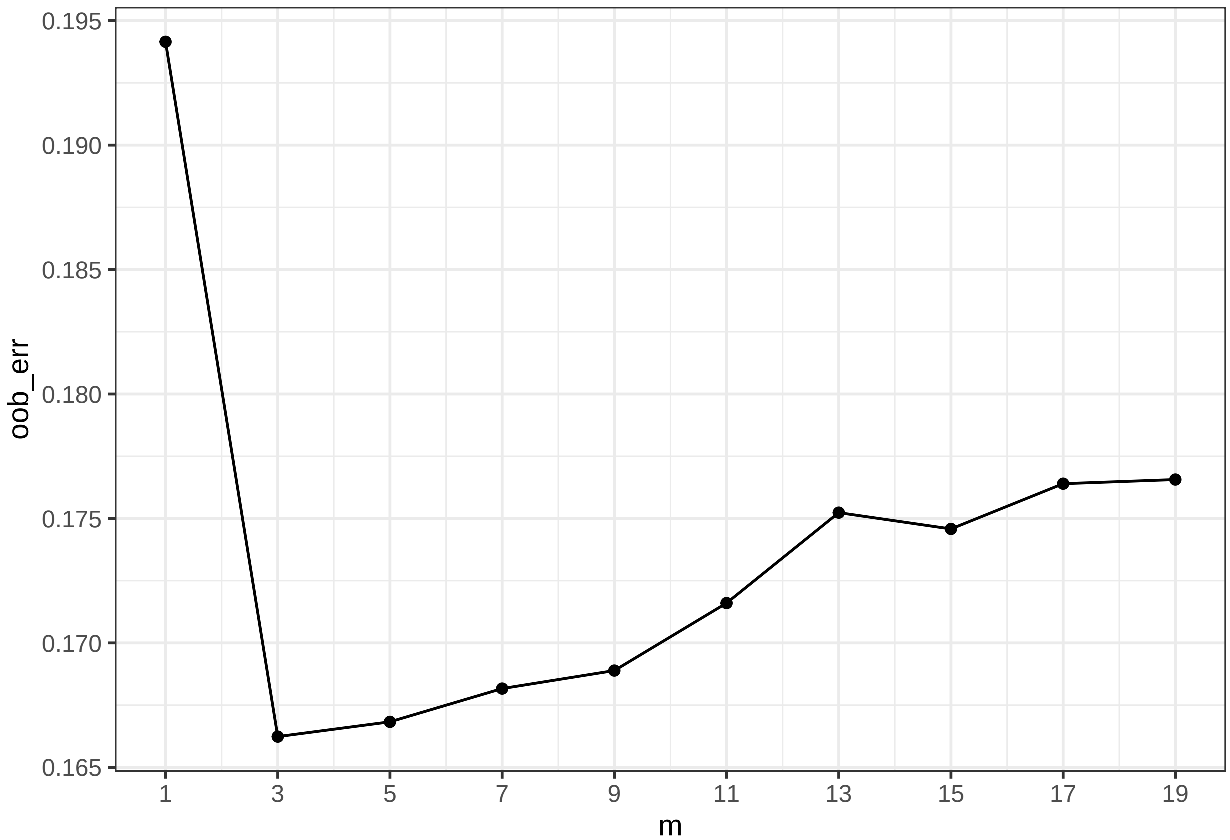
```
oob_errors %>%
  ggplot(aes(x = ntree, y = oob_err, colour = factor(m))) +
  geom_line() + theme_bw()
```




Which value of `mtry` seems to work the best here?

We can be a little more systematic in tuning the random forest by choosing a grid of values of `mtry` and plotting the OOB error for 500 trees versus `mtry`:

```
# might want to cache this chunk!
mvalues = seq(1,19, by = 2)
oob_errors = numeric(length(mvalues))
ntree = 500
for(idx in 1:length(mvalues)){
  m = mvalues[idx]
  rf_fit = randomForest(Salary ~ ., mtry = m, data = Hitters_train)
  oob_errors[idx] = rf_fit$mse[ntree]
}
tibble(m = mvalues, oob_err = oob_errors) %>%
  ggplot(aes(x = m, y = oob_err)) +
  geom_line() + geom_point() +
  scale_x_continuous(breaks = mvalues) +
  theme_bw()
```




### Variable importance

Let's go back to the default random forest fit:

```
rf_fit = randomForest(Salary ~ ., data = Hitters_train)
```

This object contains the purity-based feature importance in the `importance` field:

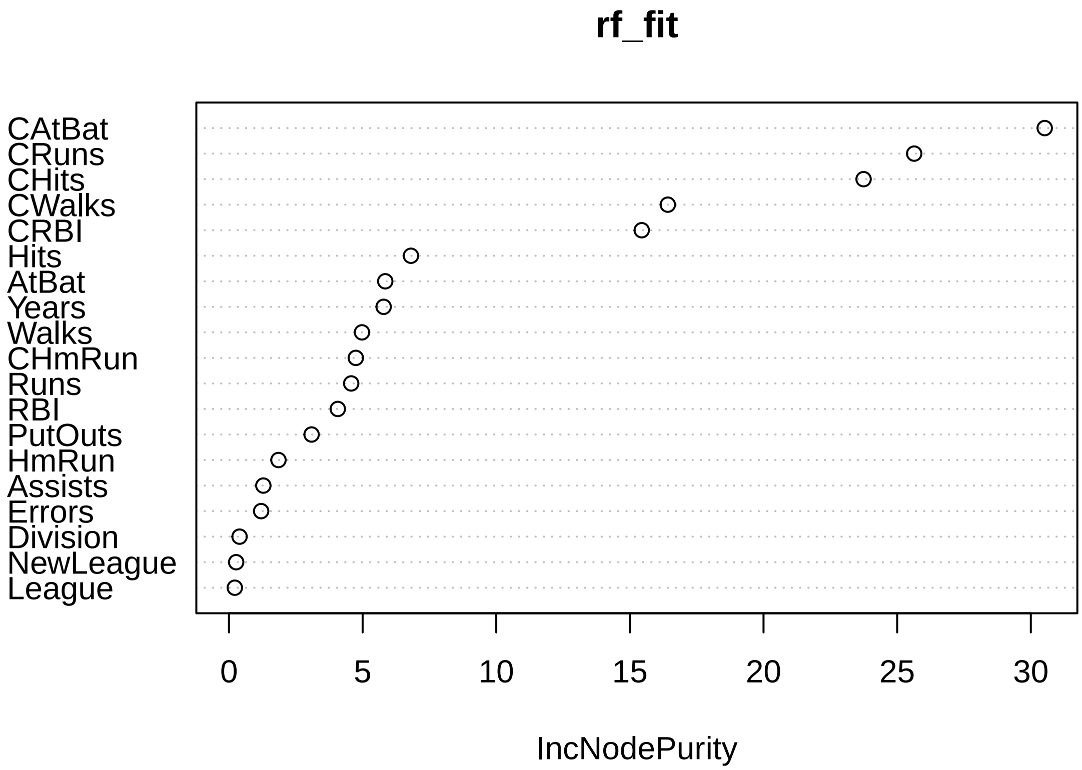
```
rf_fit$importance
```

```
##           IncNodePurity
## AtBat         5.8472732
## Hits          6.8084494
## HmRun         1.8517144
## Runs          4.5724504
## RBI           4.0710412
## Walks         4.9760854
## Years         5.7860528
## CAtBat       30.5202384
## CHits        23.7426212
## CHmRun        4.7448160
## CRuns        25.6364114
## CRBI         15.4433193
## CWalks       16.4193176
## League        0.2186973
## Division      0.3962460
## PutOuts       3.0909660
## Assists       1.2849972
```

```
## Errors        1.2032764
## NewLeague     0.2692855
```

We can visualize these importances using the built-in function called `varImpPlot`:

```
varImpPlot(rf_fit)
```




In lecture, we discussed that there were two variable importance measures. If we want to compute the second one (OOB-based importance), we need to explicitly specify this in the call to `randomForest`:

```
rf_fit = randomForest(Salary ~ ., importance = TRUE, data = Hitters_train)
```

Now let's see what the `importance` field looks like:

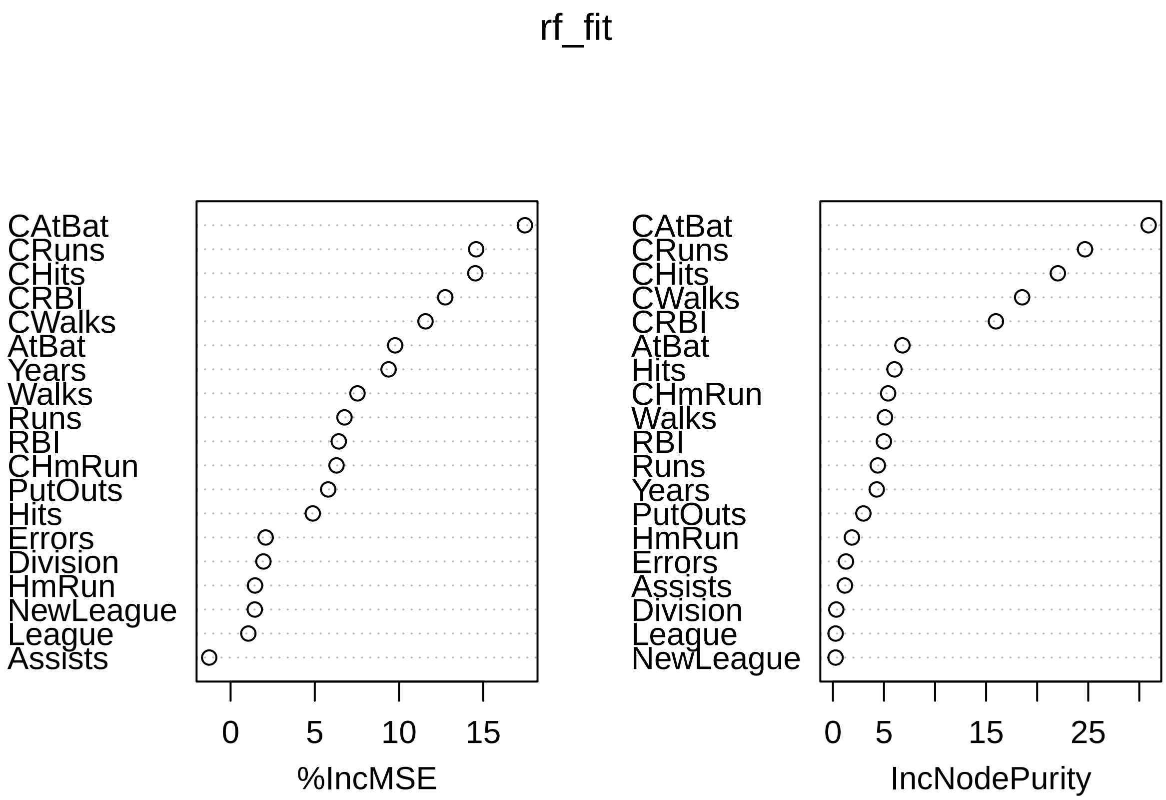
```
rf_fit$importance
```

```
##                 %IncMSE IncNodePurity
## AtBat      0.0233858662     6.8046452
## Hits       0.0108305491     6.0241269
## HmRun      0.0022536766     1.8561830
## Runs       0.0144748135     4.3898765
## RBI        0.0146433359     4.9860861
## Walks      0.0151730295     5.0956179
## Years      0.0217754325     4.2877457
## CAtBat     0.1973574070    30.9173192
## CHits      0.1380962479    22.0239109
## CHmRun     0.0210049335     5.4027865
## CRuns      0.1534388323    24.6852798
## CRBI       0.1223738644    15.9614470
## CWalks     0.0782867660    18.5284833
## League     0.0005632051     0.2584339
## Division   0.0007996822     0.3255540
```

```
## PutOuts    0.0077918413      2.9809061
## Assists   -0.0011062158      1.1736827
## Errors     0.0017946921      1.2717382
## NewLeague  0.0006349964      0.2571783
```

We see there are now two columns instead of one! We can plot both of these feature importance measures using the same syntax as above:

```
varImpPlot(rf_fit)
```




### Making predictions based on a random forest

We can make predictions using `predict`, as usual:

```
rf_predictions = predict(rf_fit, newdata = Hitters_test)
rf_predictions
```

```
##        1        2        3        4        5        6        7        8
## 6.721930 4.744138 4.484861 4.789321 5.970215 4.773344 7.075040 6.599569
##        9       10       11       12       13       14       15       16
## 5.914270 6.641251 7.176748 5.737666 6.378132 6.956755 5.787521 6.471361
##       17       18       19       20       21       22       23       24
## 4.466727 6.787044 6.271385 6.070107 6.628298 5.643797 6.703455 6.351134
##       25       26       27       28       29       30       31       32
## 6.157194 7.051782 4.626792 6.156238 6.677133 5.748750 5.147429 6.663414
##       33       34       35       36       37       38       39       40
## 5.238820 4.462744 6.200020 6.040937 6.261212 6.601225 5.950878 6.496516
##       41       42       43       44       45       46       47       48
## 6.353182 6.949286 6.544216 4.974322 6.397459 6.978974 4.872332 6.029940
##       49       50       51       52       53
```

```
## 5.899474 5.530801 5.055679 6.826845 6.053332
```

We can compute the mean-squared prediction error as usual too:

```
mean((rf_predictions - Hitters_test$Salary)^2)
```

```
## [1] 0.2439504
```

# Random forests for classification

Random forests work very similarly for classification. Let's continue with the heart disease data from last time:

```
# download the data
url = "https://raw.githubusercontent.com/JWarmenhoven/ISLR-python/master/Notebooks/Data/Heart.csv"
Heart = read_csv(url, col_types = "-iffiiiiiddiifc") %>% na.omit()

# split into train/test
set.seed(1) # set seed for reproducibility
train_samples = sample(1:nrow(Heart), round(0.8*nrow(Heart)))
Heart_train = Heart %>% filter(row_number() %in% train_samples)
Heart_test = Heart %>% filter(!(row_number() %in% train_samples))
```

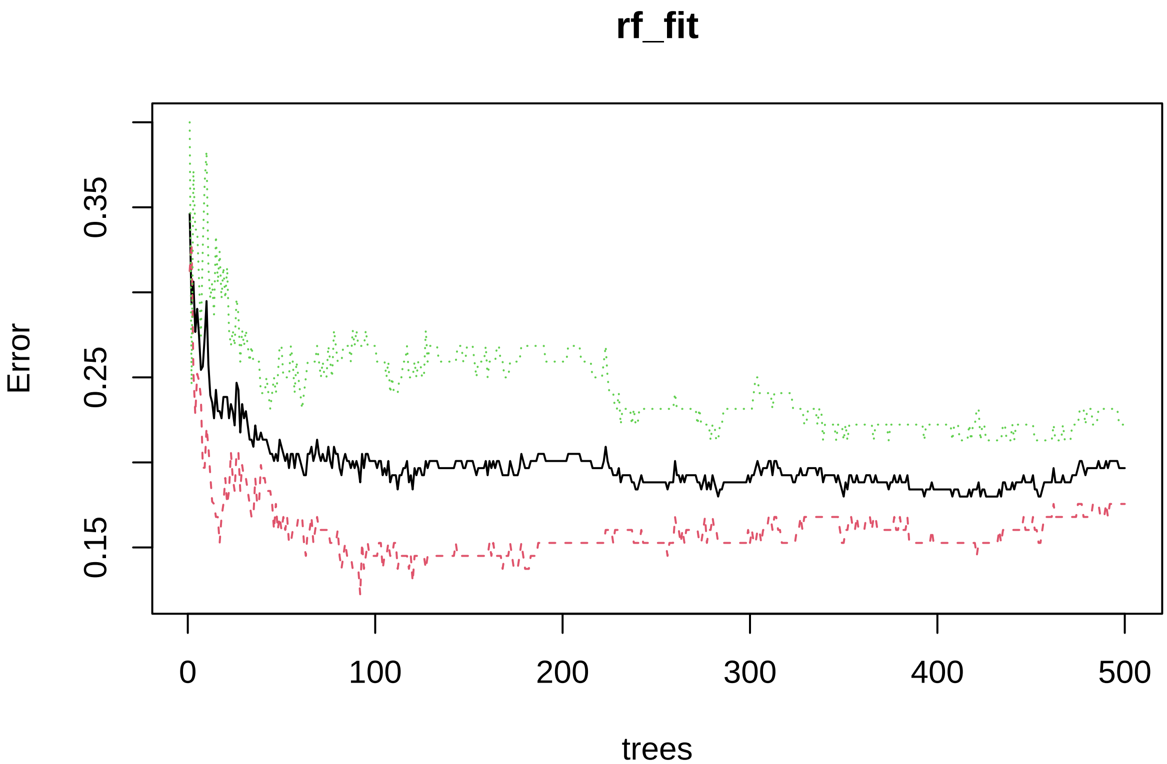
Fitting a random forest uses the same basic syntax:

```
# IMPORTANT: RESPONSE MUST BE CODED AS A FACTOR!
rf_fit = randomForest(factor(AHD) ~ ., data = Heart_train)
```

Note that for random forests the default value of `mtry` is the square root of the number of features, in this case `floor(sqrt(13)) = 3`.

When we go to make the random forest plot it looks slightly different though:

```
plot(rf_fit)
```



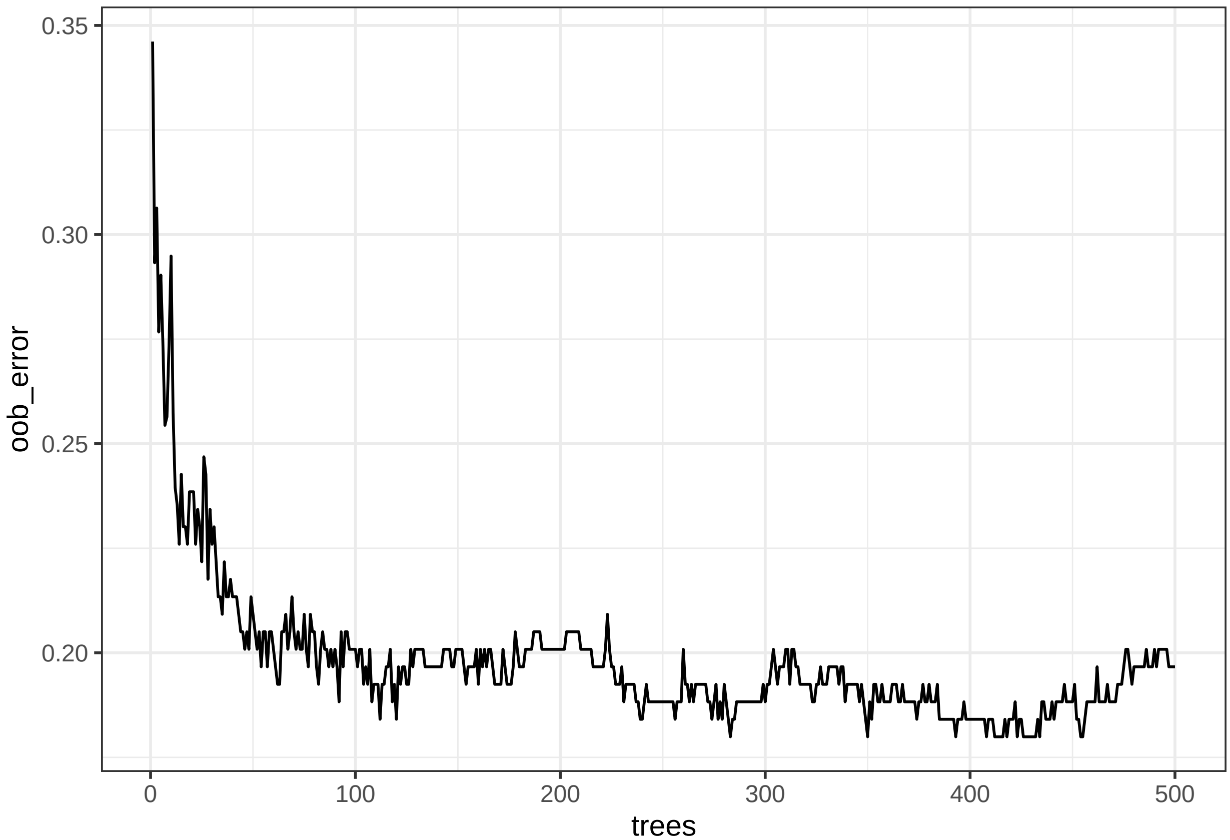

That is strange! Why does this happen? What's being plotted are three versions of the OOB error, which are stored in `rf_fit$err.rate`:

```
rf_fit$err.rate %>% head()
```

```
##            OOB        No       Yes
## [1,] 0.3461538 0.3125000 0.4000000
## [2,] 0.2932331 0.3289474 0.2456140
## [3,] 0.3063584 0.2526316 0.3717949
## [4,] 0.2766990 0.2280702 0.3369565
## [5,] 0.2903226 0.2521008 0.3367347
## [6,] 0.2743363 0.2480000 0.3069307
```

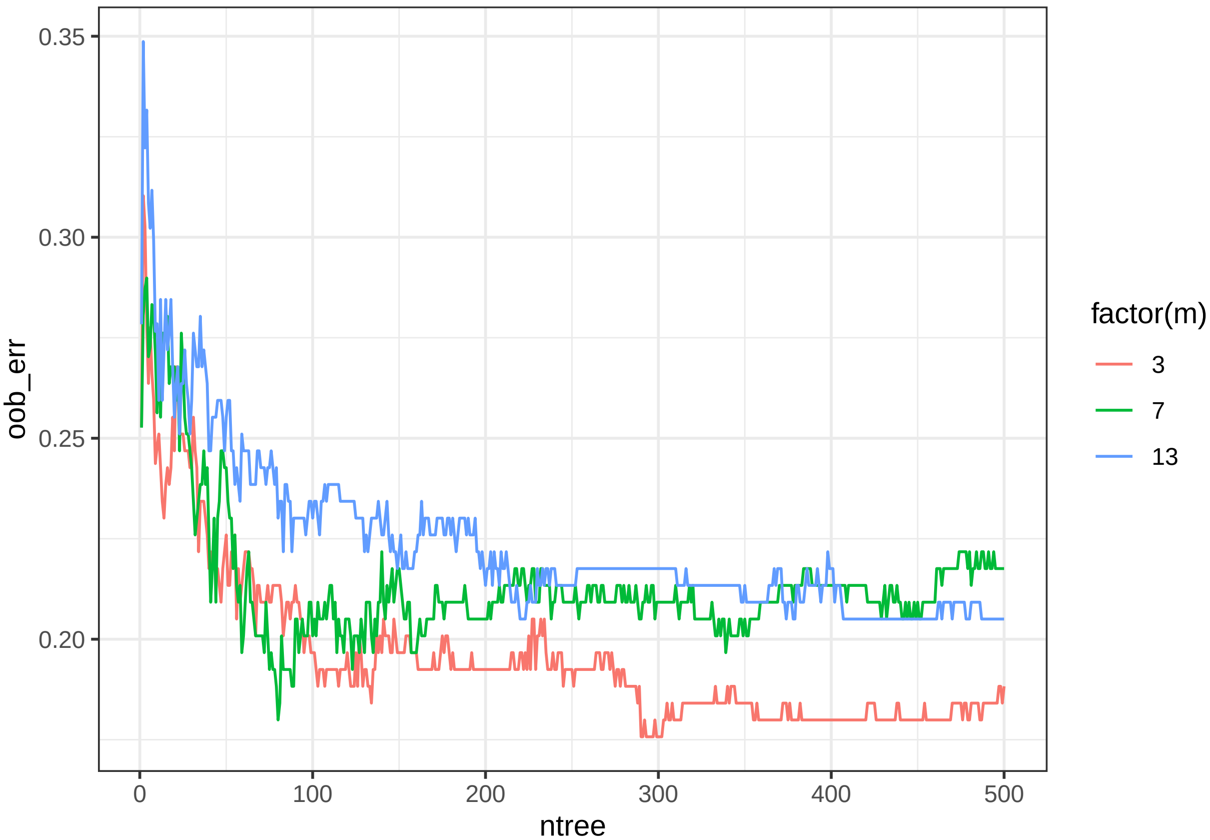
We have the OOB error column as well as two other columns, which correspond to error rates specific to each value of the response. In this class we'll ignore the latter two and focus on the OOB error, which we can plot as follows:

```
tibble(oob_error = rf_fit$err.rate[,"OOB"],
       trees = 1:500) %>%
  ggplot(aes(x = trees, y = oob_error)) + geom_line() + theme_bw()
```



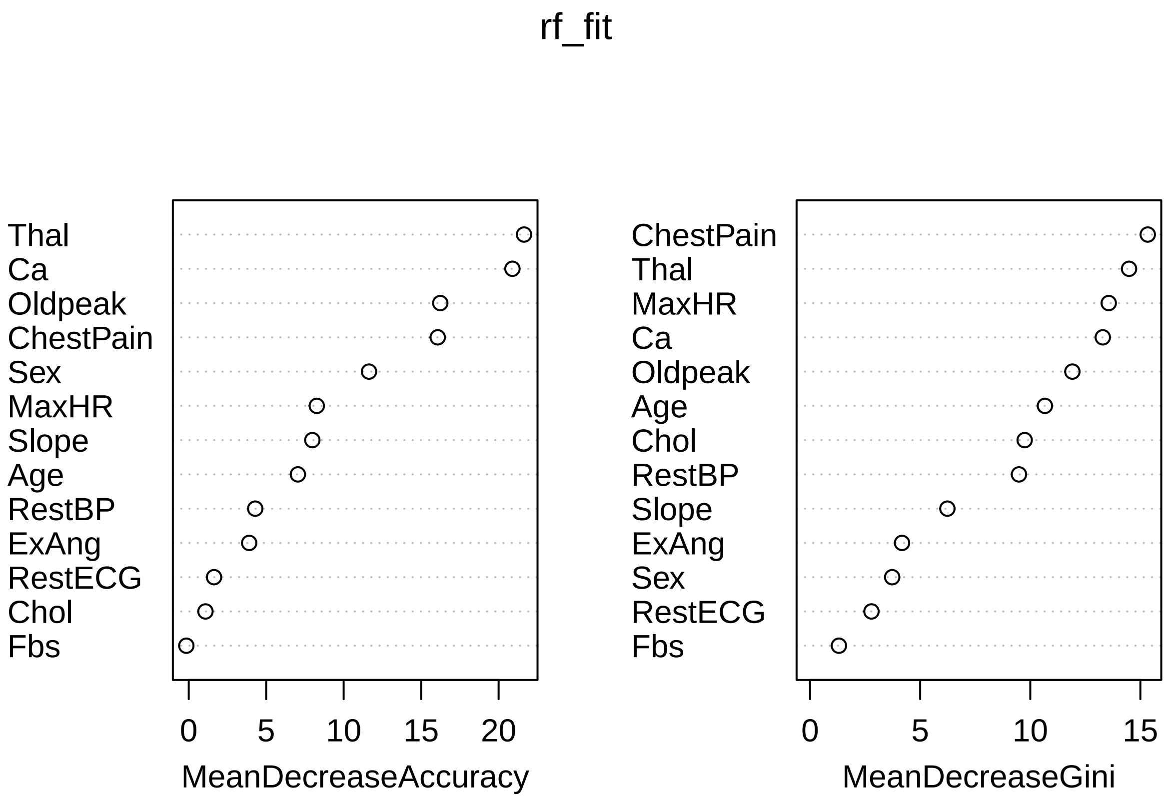

We can use the same parameters **ntree**, **mtry**, **nodesize**, and **maxnodes** as for regression random forests. For example, let's take a look at what happens when we vary **mtry**:

```
rf_3 = randomForest(factor(AHD) ~ ., mtry = 3, data = Heart_train)
rf_7 = randomForest(factor(AHD) ~ ., mtry = 7, data = Heart_train)
rf_13 = randomForest(factor(AHD) ~ ., mtry = 13, data = Heart_train)

oob_errors = bind_rows(
  tibble(ntree = 1:500, oob_err = rf_3$err.rate[,"OOB"], m = 3),
  tibble(ntree = 1:500, oob_err = rf_7$err.rate[,"OOB"], m = 7),
  tibble(ntree = 1:500, oob_err = rf_13$err.rate[,"OOB"], m = 13)
)

oob_errors %>%
  ggplot(aes(x = ntree, y = oob_err, colour = factor(m))) +
  geom_line() + theme_bw()
```




We can make variable importance plots in the same way too:

```
rf_fit = randomForest(factor(AHD) ~ ., importance = TRUE, data = Heart_train)
varImpPlot(rf_fit)
```


rf_fit
Thal
Ca
Oldpeak
ChestPain
Sex
MaxHR
Slope
Age
RestBP
ExAng
RestECG
Chol
Fbs
0 5 10 15 20
MeanDecreaseAccuracy
ChestPain
Thal
MaxHR
Ca
Oldpeak
Age
Chol
RestBP
Slope
ExAng
Sex
RestECG
Fbs
0 5 10 15
MeanDecreaseGini