Unit 4 Lecture 4: Boosting

November 11, 2021

Today, we will learn how to train and tune boosting models using the `gbm` package.

First, let's load some libraries:

```r
library(gbm)        # install.packages("gbm")
library(tidyverse)
```

Boosting models for regression

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

```r
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>
```

```r
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 gradient boosting model

Arguments:

- `distribution`: "gaussian" for continuous responses; "bernoulli" for binary responses
- `n.trees`: maximum number of trees to try; defaults to 100 but this is normally not enough trees
- `interaction.depth`: interaction depth; defaults to 1
- `shrinkage`: shrinkage parameter lambda: defaults to 0.1

- `bag.fraction`: subsampling fraction pi; defaults to 0.5
- `cv.folds`: number of CV folds to use; defaults to 0 (i.e. no CV)
- `train.fraction`: fraction of data to use as training; rest used as validation set

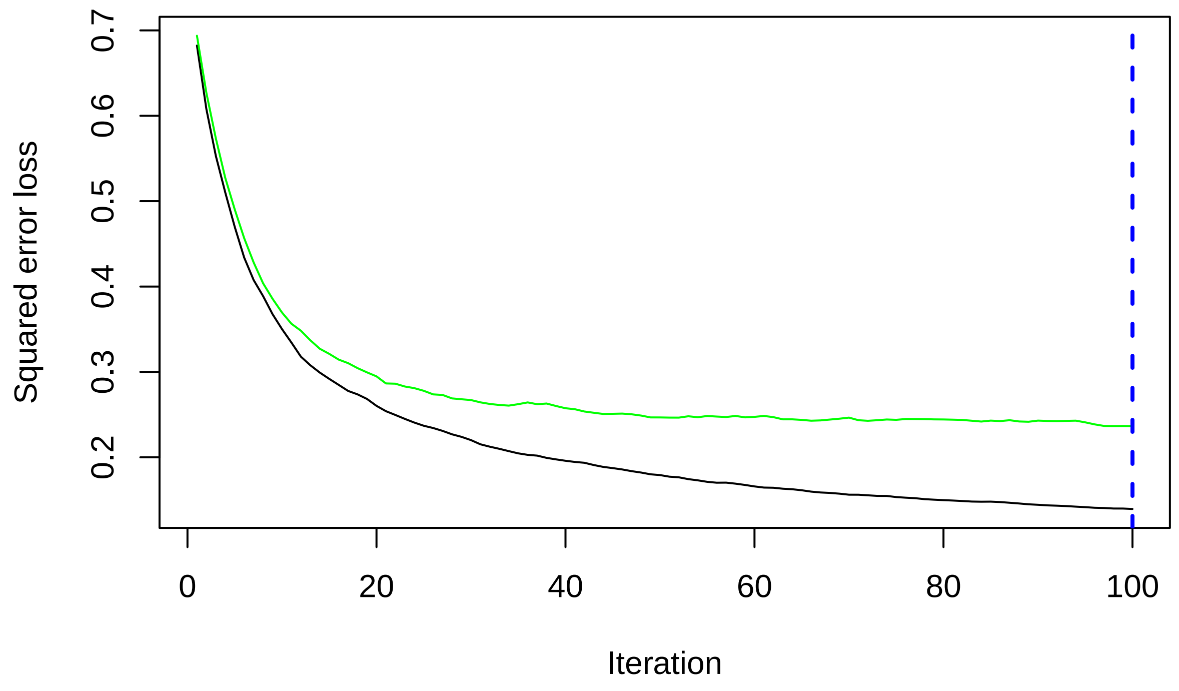
```
# read more about the inputs and outputs, bells and whistles of gbm
?gbm
```

Training the model:

```
set.seed(1)
gbm_fit = gbm(Salary ~ .,
              distribution = "gaussian",
              n.trees = 100,
              interaction.depth = 1,
              shrinkage = 0.1,
              cv.folds = 5,
              data = Hitters_train)
```

We can visualize the CV error using `gbm.perf`, which both makes a plot and outputs the optimal number of trees:

```
opt_num_trees = gbm.perf(gbm_fit)
```



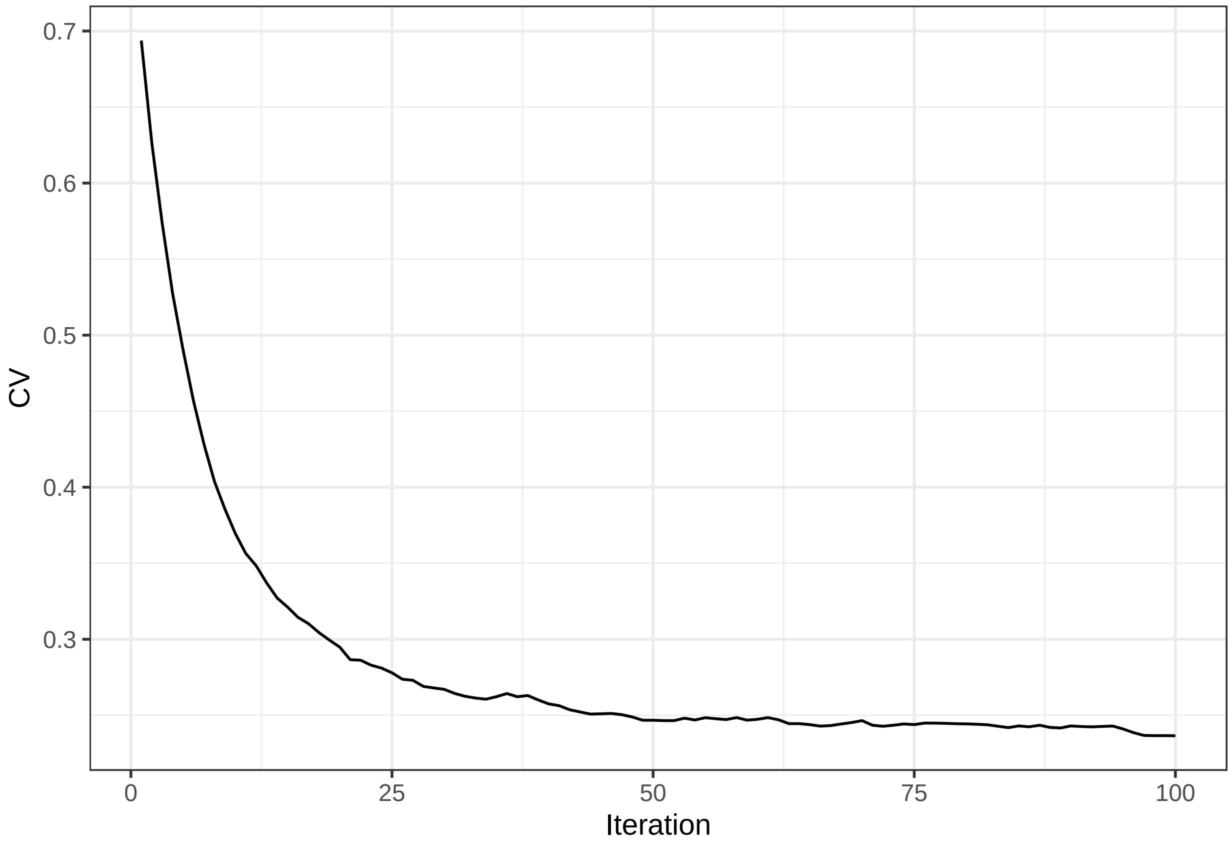
```
opt_num_trees
```

```
## [1] 100
```

The green curve is the CV error; the black curve is the training error. The dashed blue line indicates the minimum of the CV error.

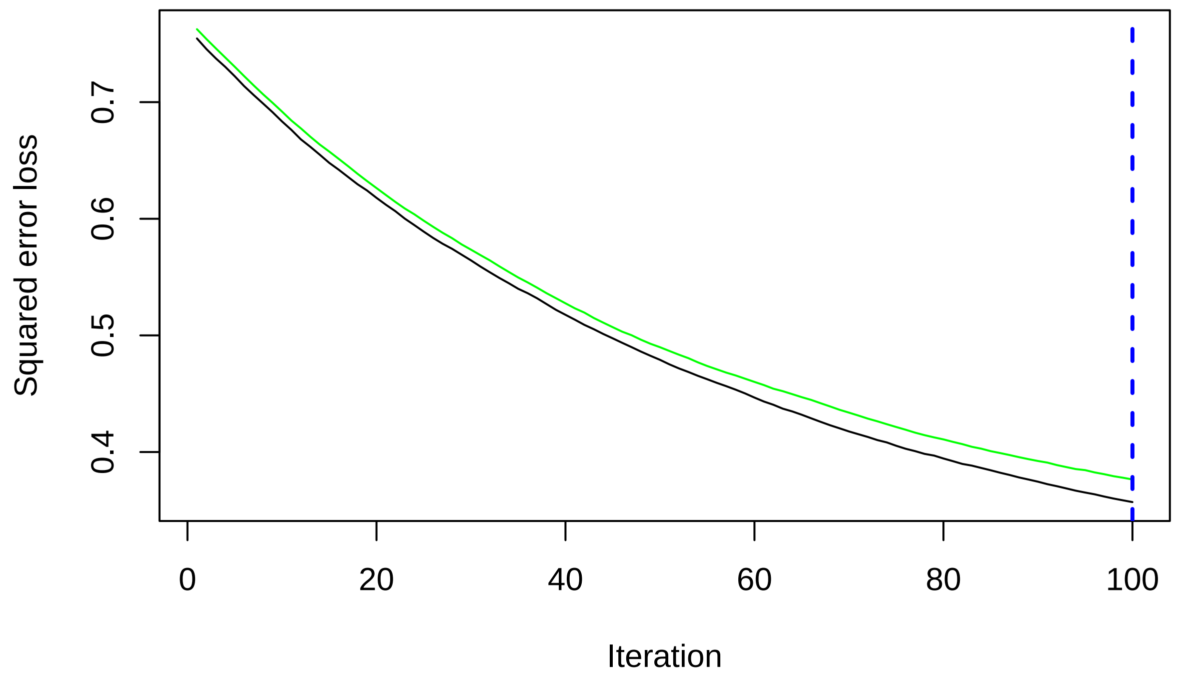
Note that `gbm_fit$cv.error` also contains the CV errors, so these can be plotted manually as well:

```
ntrees = 100
tibble(Iteration = 1:ntrees, CV = gbm_fit$cv.error) %>%
  ggplot(aes(x = Iteration, y = CV)) + geom_line() +
  theme_bw()
```



We want to make sure there are enough trees that the CV curve has reached its minimum. For example, suppose we had chosen a smaller shrinkage parameter, e.g. 0.01:

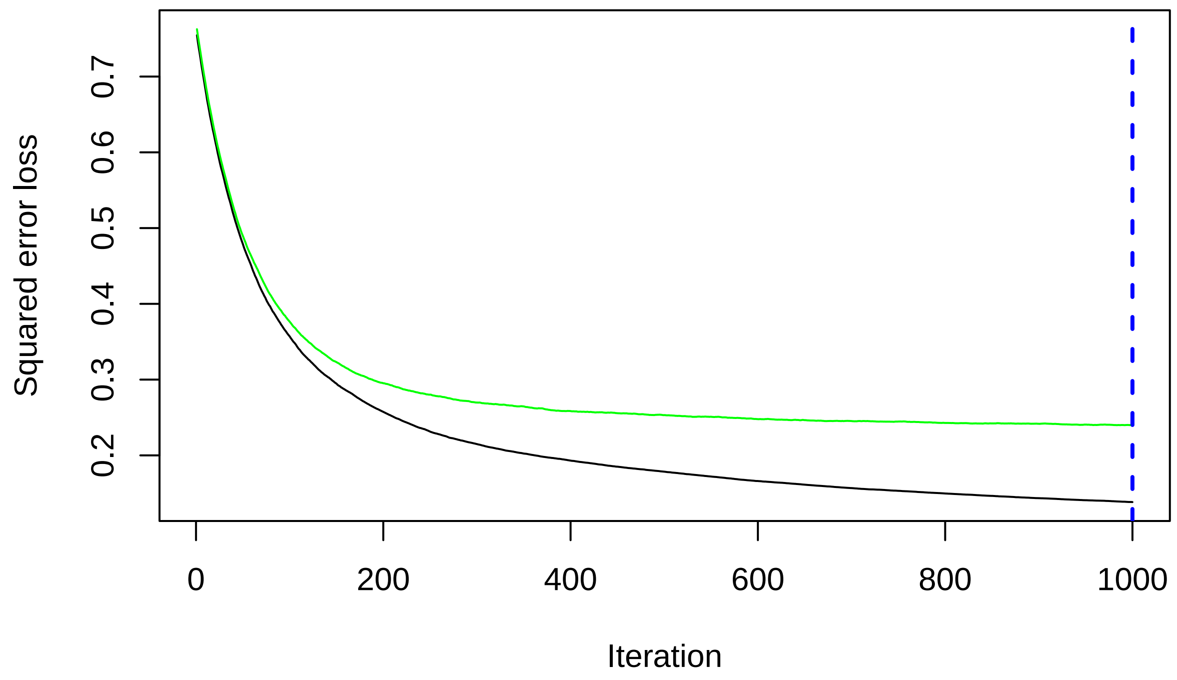
```
set.seed(1)
gbm_fit_slow = gbm(Salary ~ .,
                   distribution = "gaussian",
                   n.trees = 100,
                   interaction.depth = 1,
                   shrinkage = 0.01,
                   cv.folds = 5,
                   data = Hitters_train)
gbm.perf(gbm_fit_slow)
```



```
## [1] 100
```

We see that 100 is not enough trees for lambda = 0.01. In this case, we would need to increase the number of trees:

```
set.seed(1)
gbm_fit_slow = gbm(Salary ~ .,
                   distribution = "gaussian",
                   n.trees = 1000,
                   interaction.depth = 1,
                   shrinkage = 0.01,
                   cv.folds = 5,
                   data = Hitters_train)
gbm.perf(gbm_fit_slow)
```



```
## [1] 1000
```

Tuning the interaction depth

The quick way to tune the interaction depth is to try out a few different values:

```
set.seed(1)
gbm_fit_1 = gbm(Salary ~ .,
              distribution = "gaussian",
              n.trees = 100,
              interaction.depth = 1,
              shrinkage = 0.1,
              cv.folds = 5,
              data = Hitters_train)
gbm_fit_2 = gbm(Salary ~ .,
              distribution = "gaussian",
              n.trees = 100,
              interaction.depth = 2,
              shrinkage = 0.1,
              cv.folds = 5,
              data = Hitters_train)
gbm_fit_3 = gbm(Salary ~ .,
              distribution = "gaussian",
              n.trees = 100,
              interaction.depth = 3,
              shrinkage = 0.1,
              cv.folds = 5,
              data = Hitters_train)
```

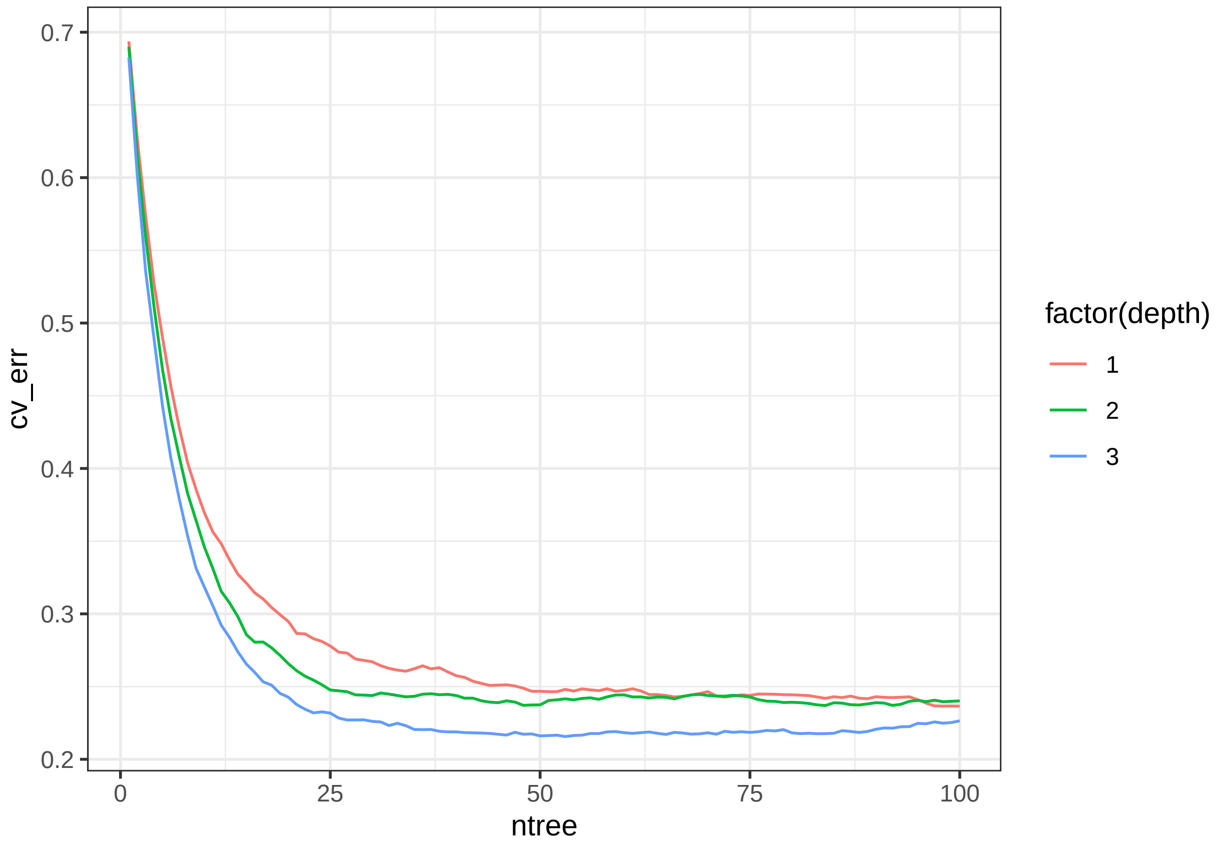
We can extract the CV errors from each of these objects by using the `cv.error` field:

```
ntrees = 100
cv_errors = bind_rows(
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_1$cv.error, depth = 1),
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_2$cv.error, depth = 2),
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_3$cv.error, depth = 3)
)
cv_errors
```

```
## # A tibble: 300 x 3
##    ntree cv_err depth
##    <int>  <dbl> <dbl>
##  1     1  0.694     1
##  2     2  0.627     1
##  3     3  0.574     1
##  4     4  0.527     1
##  5     5  0.490     1
##  6     6  0.457     1
##  7     7  0.428     1
##  8     8  0.404     1
##  9     9  0.386     1
## 10    10  0.370     1
## # ... with 290 more rows
```

We can then plot these as follows:

```
cv_errors %>%
  ggplot(aes(x = ntree, y = cv_err, colour = factor(depth))) +
  geom_line() + theme_bw()
```



Which value of `interaction.depth` seems to work the best here?

Let's save the optimal model and optimal number of trees (note `plot.it = FALSE` in `gbm.perf` to extract the optimal number of trees without making the CV plot again):

```
gbm_fit_optimal = gbm_fit_3
optimal_num_trees = gbm.perf(gbm_fit_3, plot.it = FALSE)
optimal_num_trees
```

```
## [1] 53
```

Model interpretation

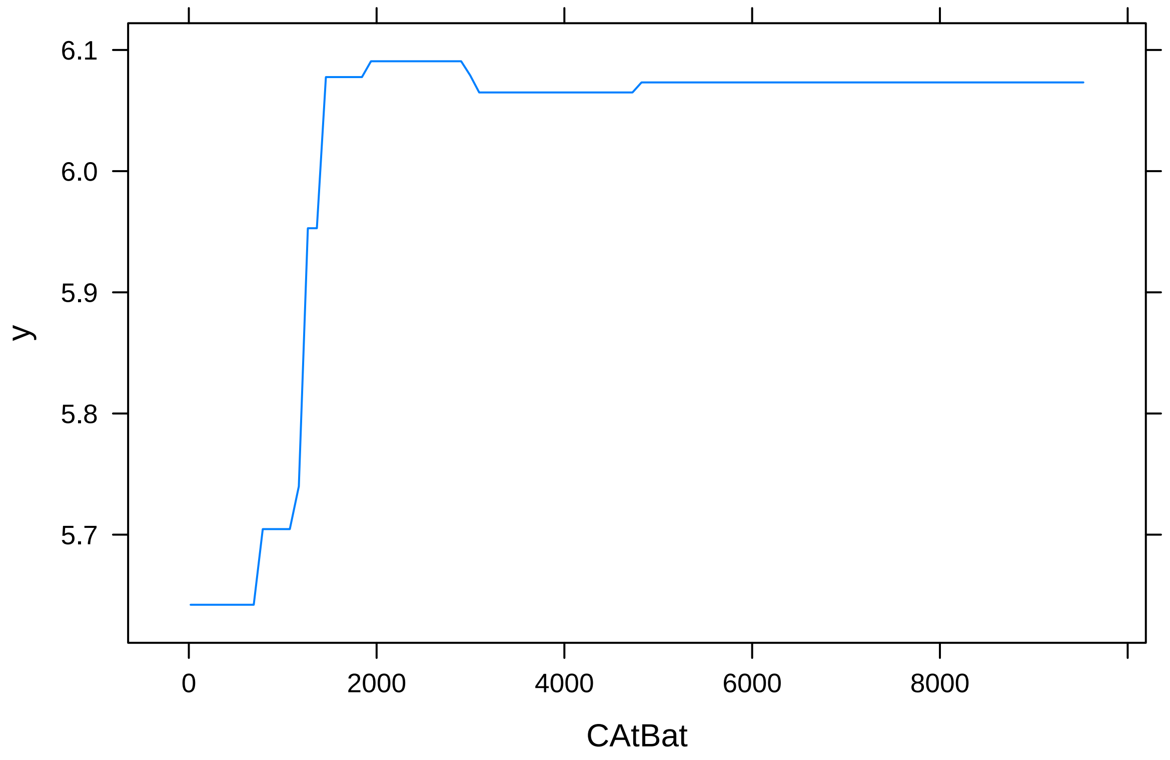
Let's now interpret our tuned model. To get the variable importance measures, we use `summary`, specifying the number of trees via the `n.trees` argument:

```
summary(gbm_fit_optimal, n.trees = optimal_num_trees, plotit = FALSE)
```

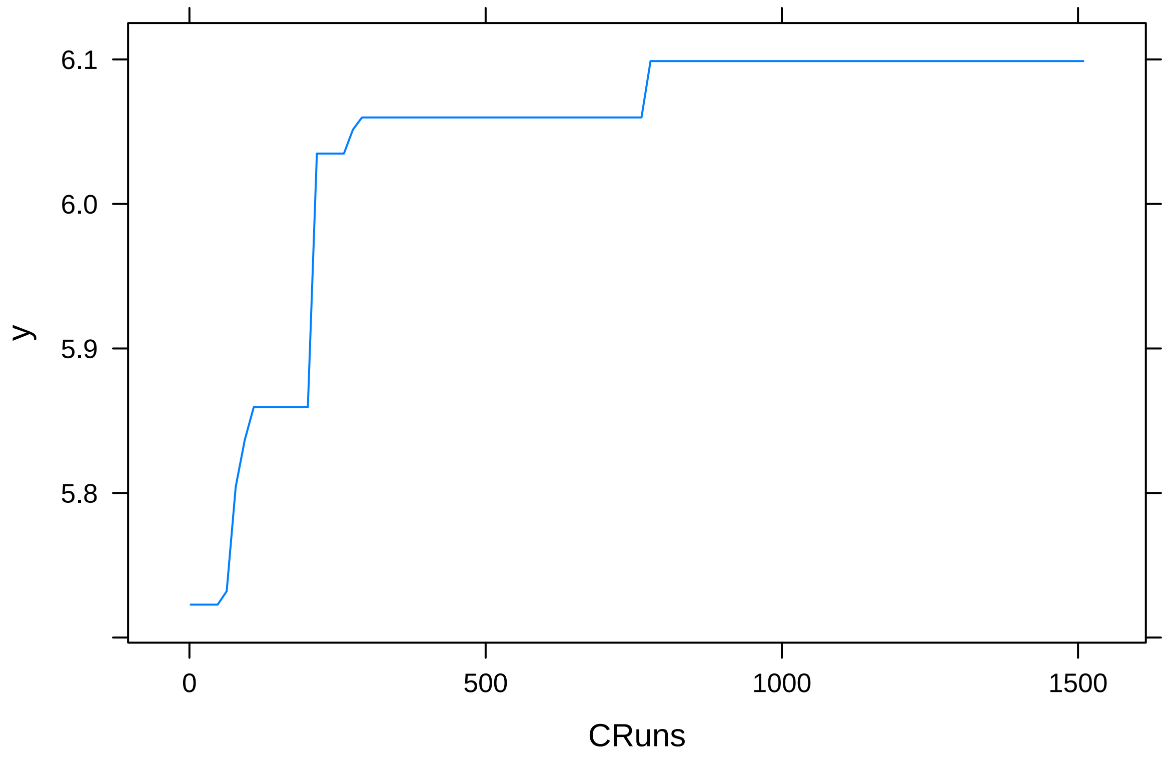
```
##                 var    rel.inf
## CAtBat       CAtBat 24.5153770
## CRBI           CRBI 16.6073144
## CRuns         CRuns 11.5190958
## CWalks       CWalks  9.7032929
## CHits         CHits  6.6707587
## AtBat         AtBat  4.4347641
## PutOuts     PutOuts  4.3226905
## Hits           Hits  4.2193568
## Years         Years  3.1751279
## Walks         Walks  3.0915187
## CHmRun       CHmRun  3.0041789
## RBI             RBI  2.9560862
## Runs           Runs  2.4589581
## HmRun         HmRun  0.9662435
## Errors       Errors  0.8391951
## League       League  0.6814164
## Division   Division  0.4754207
## Assists     Assists  0.3592044
## NewLeague NewLeague  0.0000000
```

We can also make the partial dependence plots for the different features using `plot`:

```
plot(gbm_fit_optimal, i.var = "CAtBat", n.trees = optimal_num_trees)
```



```r
plot(gbm_fit_optimal, i.var = "CRuns", n.trees = optimal_num_trees)
```



Making predictions based on a boosting model:

We can make predictions using `predict`, as usual, but we need to specify the number of trees to use:

```
gbm_predictions = predict(gbm_fit_optimal, n.trees = optimal_num_trees,
                          newdata = Hitters_test)
gbm_predictions
```

```
##  [1] 6.831876 5.032180 5.191472 4.754915 5.705454 4.752605 6.919533 6.559619
##  [9] 6.192010 6.721912 7.220870 5.651122 6.370257 7.014860 5.777815 6.290083
## [17] 4.879809 6.639346 6.262634 6.077145 6.603744 5.611133 6.626498 6.276845
## [25] 6.179101 7.002552 4.539483 6.083082 6.650163 5.686692 4.986215 6.487983
## [33] 4.986992 4.800090 6.171913 6.127455 6.327928 6.440638 6.250032 6.742604
## [41] 6.423631 6.971434 6.360514 4.732555 6.585064 6.999726 4.971630 6.290141
## [49] 6.219813 5.460875 5.226644 6.807000 6.175467
```

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

```
sqrt(mean((gbm_predictions - Hitters_test$Salary)^2))
```

```
## [1] 0.5334763
```

Boosting for classification

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

```
url = "https://raw.githubusercontent.com/JWarmenhoven/ISLR-python/master/Notebooks/Data/Heart.csv"
Heart = read_csv(url, col_types = "-iffiiiiiddiifc") %>%
  na.omit() %>%
  mutate(AHD = ifelse(AHD == "Yes", 1, 0))  # gbm expects response to be 0-1,
                                            #  NOT factor (unlike RF)

# 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))
```

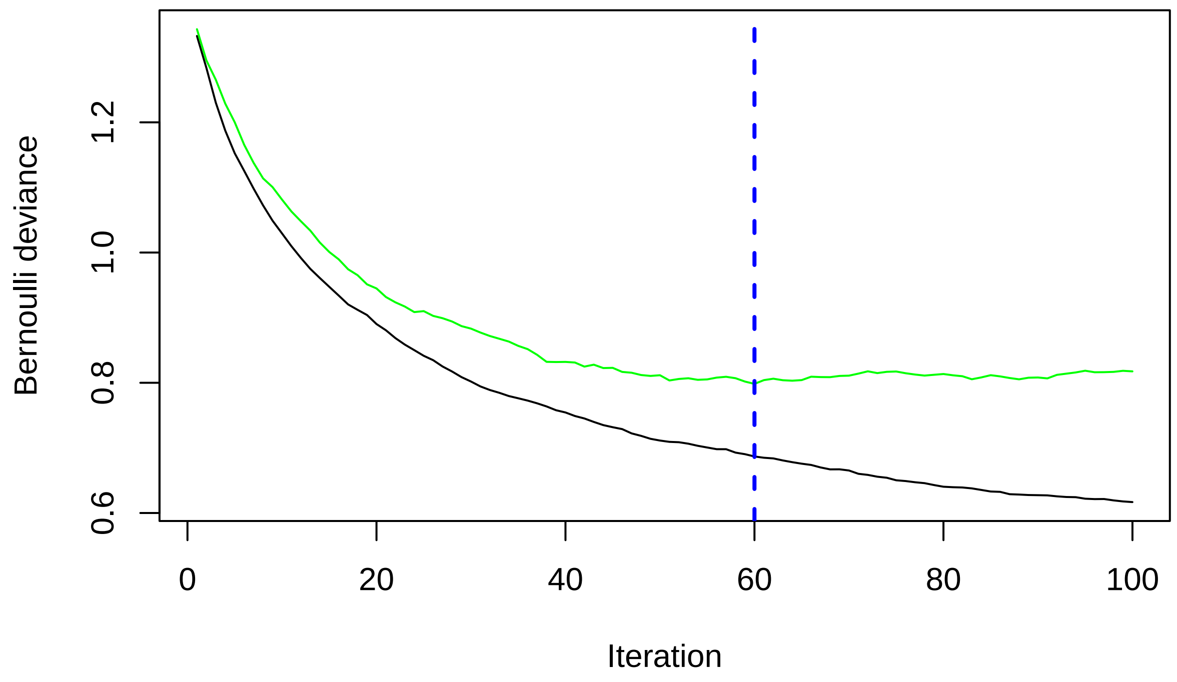
Fitting a boosting model uses the same basic syntax, but with `distribution = "bernoulli"`:

```
set.seed(1)
gbm_fit = gbm(AHD ~ .,
              distribution = "bernoulli",
              n.trees = 100,
              interaction.depth = 1,
              shrinkage = 0.1,
              cv.folds = 5,
              data = Heart_train)
```

Common pitfalls when fitting a `gbm`:

- The binary response is coded as a `character`, e.g. "Yes"/"No".
- The binary response is coded as a `factor`.
- Any of the features are coded as strings, rather than factors.

```
gbm.perf(gbm_fit)
```



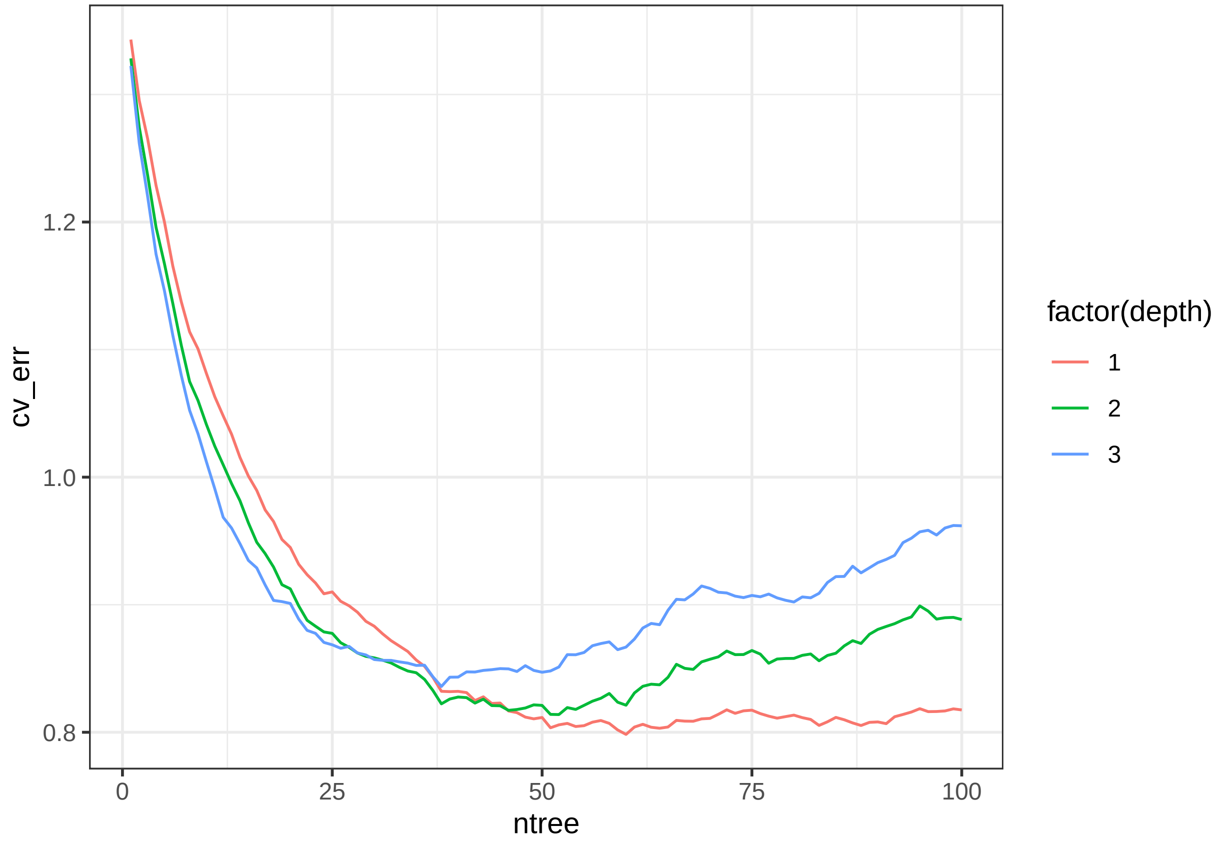
```
## [1] 60
```

We can tune the interaction depth in the same way as before:

```
# try a few values
set.seed(1)
gbm_fit_1 = gbm(AHD ~ .,
              distribution = "bernoulli",
              n.trees = 100,
              interaction.depth = 1,
              shrinkage = 0.1,
              cv.folds = 5,
              data = Heart_train)
set.seed(1)
gbm_fit_2 = gbm(AHD ~ .,
              distribution = "bernoulli",
              n.trees = 100,
              interaction.depth = 2,
              shrinkage = 0.1,
              cv.folds = 5,
              data = Heart_train)
set.seed(1)
gbm_fit_3 = gbm(AHD ~ .,
              distribution = "bernoulli",
              n.trees = 100,
              interaction.depth = 3,
              shrinkage = 0.1,
              cv.folds = 5,
              data = Heart_train)

# extract CV errors
ntrees = 100
cv_errors = bind_rows(
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_1$cv.error, depth = 1),
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_2$cv.error, depth = 2),
  tibble(ntree = 1:ntrees, cv_err = gbm_fit_3$cv.error, depth = 3)
)

# plot CV errors
cv_errors %>%
  ggplot(aes(x = ntree, y = cv_err, colour = factor(depth))) +
  geom_line() + theme_bw()
```



Aha! We see some overfitting! For which values of interaction depth do we see more overfitting, and why? What is the optimal interaction depth?

```
gbm_fit_optimal = gbm_fit_1
optimal_num_trees = gbm.perf(gbm_fit_1, plot.it = FALSE)
```

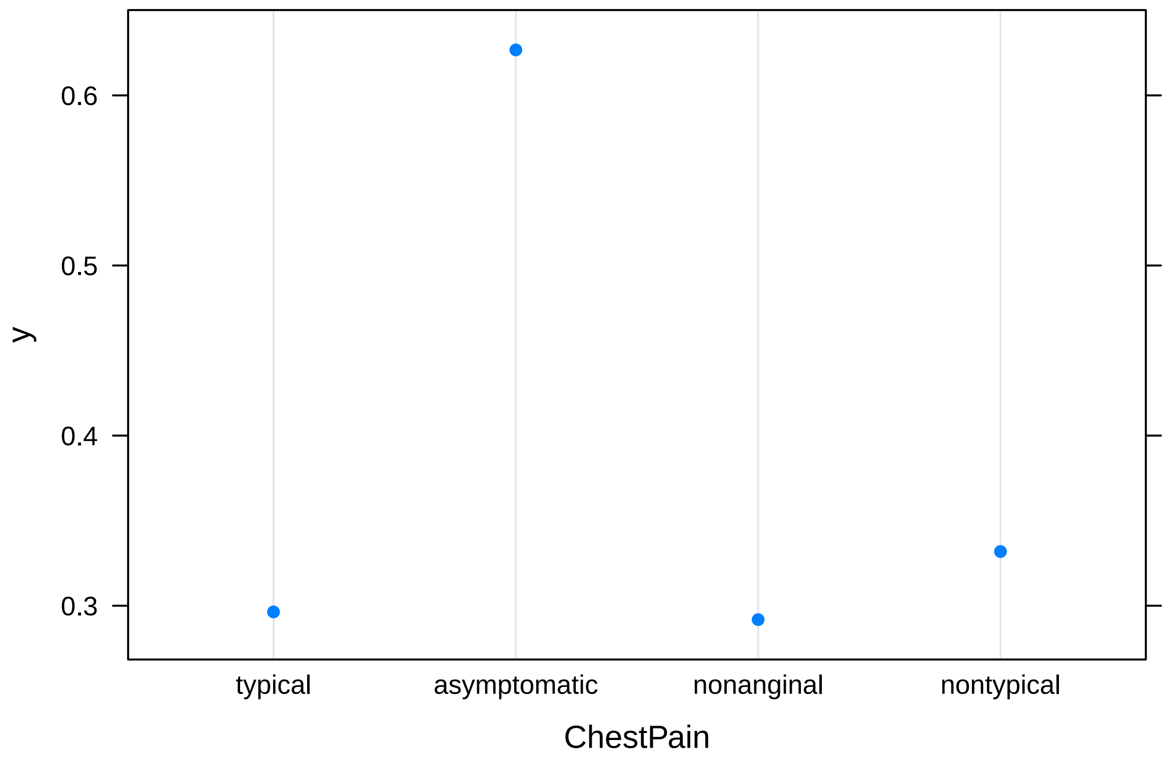
We can calculate variable importance scores as before:

```
summary(gbm_fit_optimal, n.trees = optimal_num_trees, plotit = FALSE)
```

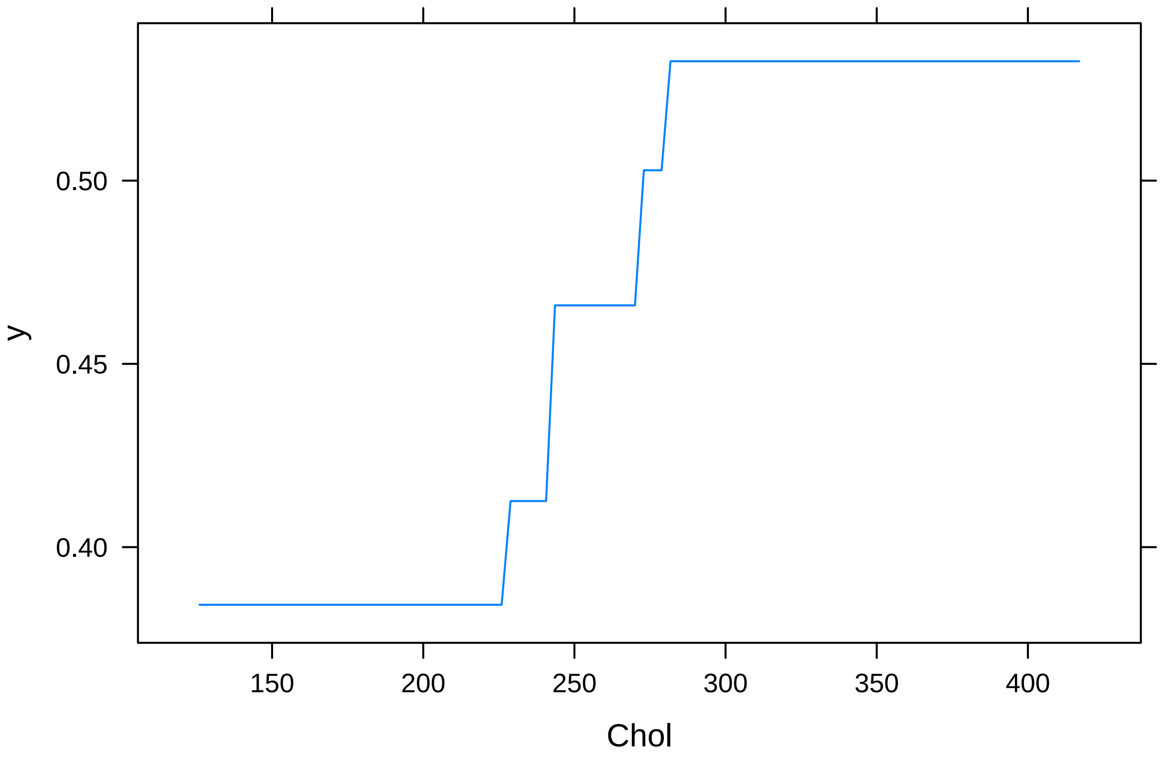
```
##                 var   rel.inf
## ChestPain ChestPain 26.171634
## Thal           Thal 21.926979
## Ca               Ca 21.021617
## Oldpeak     Oldpeak  7.777384
## MaxHR         MaxHR  6.794431
## Chol           Chol  3.920631
## Slope         Slope  3.283046
## RestBP       RestBP  3.008152
## Sex             Sex  2.769791
## ExAng         ExAng  1.857264
## Age             Age  1.469070
## Fbs             Fbs  0.000000
## RestECG     RestECG  0.000000
```

For the partial dependence plots, it's useful to specify `type = "response"` so we can interpret the y axis on the probability scale:

```
plot(gbm_fit_optimal, i.var = "ChestPain", n.trees = optimal_num_trees, type = "response")
```



```
plot(gbm_fit_optimal, i.var = "Chol", n.trees = optimal_num_trees, type = "response")
```



To make predictions, use the same syntax as before but with `type = "response"` to get predictions on the probability scale:

```
gbm_probabilities = predict(gbm_fit_optimal, n.trees = optimal_num_trees,
                            type = "response", newdata = Heart_test)
gbm_probabilities
```

```
##  [1] 0.95737346 0.05912815 0.33020657 0.76891179 0.47295576 0.27863385
##  [7] 0.19513646 0.26055091 0.88884444 0.72941877 0.08203111 0.54234917
## [13] 0.05981156 0.92066149 0.39445784 0.36753679 0.94231197 0.97648564
## [19] 0.18233581 0.23391324 0.92945600 0.14398180 0.38773436 0.41093601
## [25] 0.05416493 0.91803106 0.23112832 0.96456990 0.85078912 0.71560082
## [31] 0.95463630 0.95027231 0.27237411 0.73277821 0.03889593 0.35264965
## [37] 0.87589252 0.86364608 0.92799472 0.33914436 0.08598292 0.92220713
## [43] 0.41009369 0.08898101 0.96951434 0.03503601 0.12429338 0.26230606
## [49] 0.93001342 0.08791087 0.69489085 0.83768148 0.04893160 0.06336597
## [55] 0.05542077 0.90798838 0.28925915 0.20442883 0.31517227 0.65976992
```

We can then threshold the probabilities at 0.5 as usual and calculate the misclassification error:

```
gbm_predictions = as.numeric(gbm_probabilities > 0.5)
mean(gbm_predictions != Heart_test$AHD)
```

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