Unit 4 Lecture 4: Boosting

November 7, 2023

Today, we will learn how to train and tune boosting models using the `gbm` package. Note that the `xgboost` package is more popular in practice, but the `gbm` package is simpler to use and is a stepping stone to learning `xgboost`.

First, let's load some libraries:

```r
library(gbm)
library(tidyverse)
```

Boosting models for regression

Like last time, we will be using the `hitters` data, splitting into training and testing:

```r
hitters_data <- read_csv("hitters-data.csv")

# gbm expects character features to be coded as factors
hitters_data <- hitters_data |>
  mutate(across(where(is.character), as.factor))

set.seed(1) # set seed for reproducibility
train_samples <- sample(1:nrow(hitters_data), round(0.8 * nrow(hitters_data)))
hitters_train <- hitters_data |> filter(row_number() %in% train_samples)
hitters_test <- hitters_data |> 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) but we prefer 5
- `train.fraction`: fraction of data to use as training; rest used as validation set; we leave at default of 1
- `n.cores`: how many parallel processors to use for CV; we set to 1

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

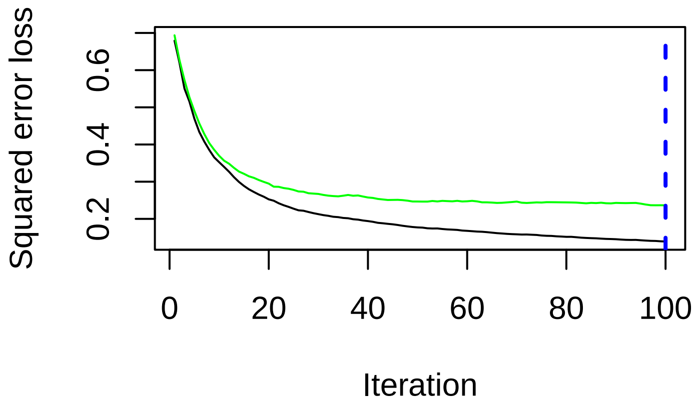
Training the model:

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

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

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)
```



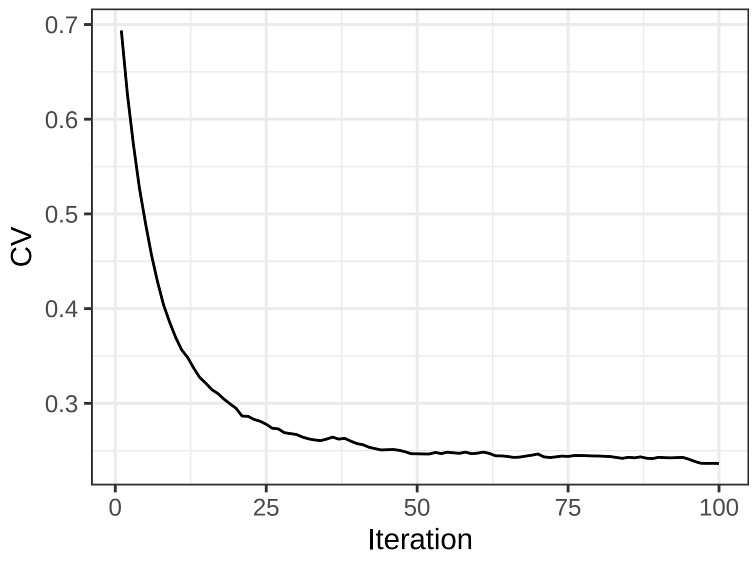
```
opt_num_trees
```

```
## [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()
```

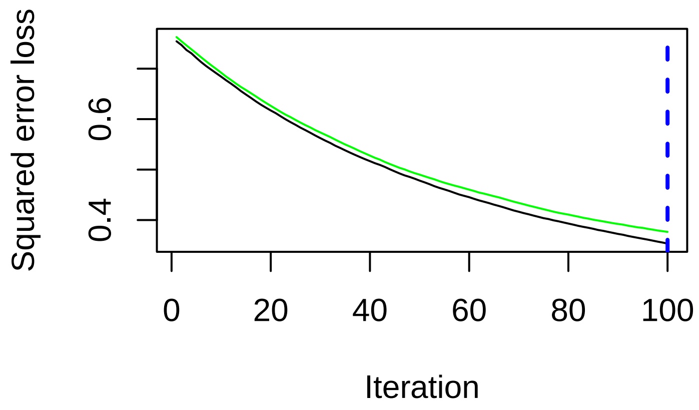


We want to make sure there are enough trees that the CV curve has reached its minimum. For example, suppose we had chosen a smaller shrinkage parameter, e.g. 0.01:

```r
set.seed(1)
gbm_fit_slow <- gbm(Salary ~ .,
  distribution = "gaussian",
  n.trees = 100,
  interaction.depth = 1,
  shrinkage = 0.01,
  cv.folds = 5,
  n.cores = 1,
  data = hitters_train
)
```

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

```r
gbm.perf(gbm_fit_slow)
```



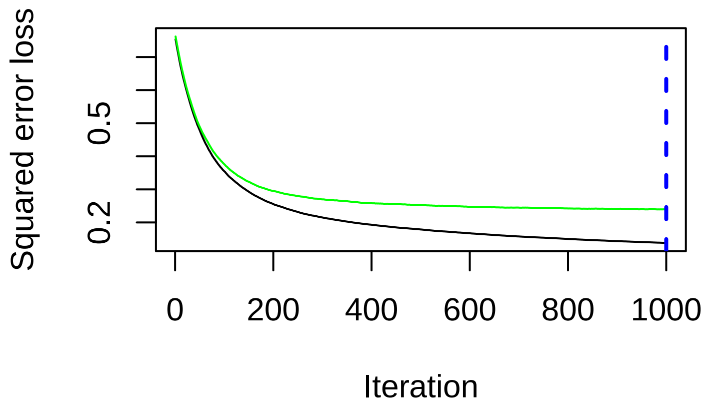
```
## [1] 100
```

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,
  n.cores = 1,
  data = hitters_train
)
```

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

```
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,
  n.cores = 1,
  data = hitters_train
)
```

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

```r
gbm_fit_2 <- gbm(Salary ~ .,
  distribution = "gaussian",
  n.trees = 100,
  interaction.depth = 2,
  shrinkage = 0.1,
  cv.folds = 5,
  n.cores = 1,
  data = hitters_train
)
```

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

```r
gbm_fit_3 <- gbm(Salary ~ .,
  distribution = "gaussian",
  n.trees = 100,
  interaction.depth = 3,
  shrinkage = 0.1,
  cv.folds = 5,
  n.cores = 1,
  data = hitters_train
)
```

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

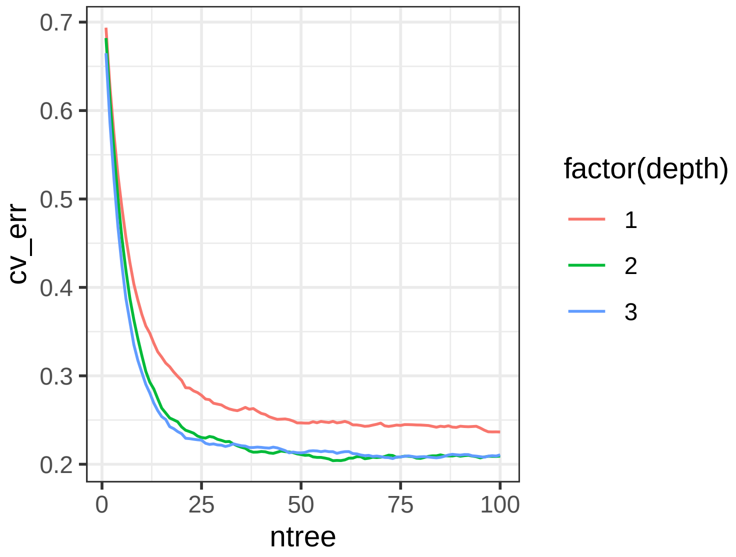
We can extract the CV errors from each of these objects by using the `cv.error` field:

```r
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
## # i 290 more rows
```

We can then plot these as follows:

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



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] 73
```

Model interpretation

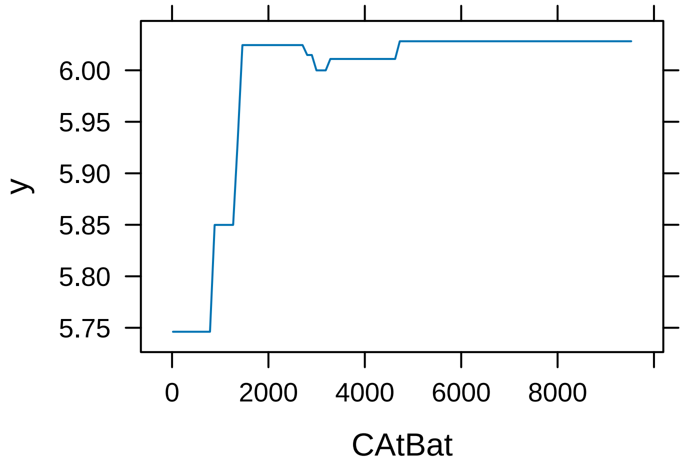
Let's now interpret our tuned model. To get the variable importance measures, we use `summary`, specifying the number of trees via the `n.trees` argument:

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

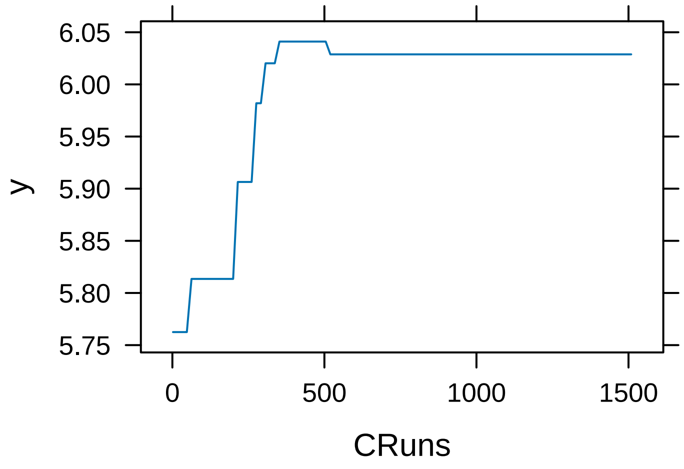
```
##                 var    rel.inf
## CRBI           CRBI 18.0869026
## CHits         CHits 15.4405163
## CAtBat       CAtBat 12.6752501
## CWalks       CWalks 12.5496768
## PutOuts     PutOuts  7.1258035
## CRuns         CRuns  6.9511709
## RBI             RBI  4.4357173
## Walks         Walks  4.4213583
## Hits           Hits  3.3555169
## Runs           Runs  2.5962647
## Years         Years  2.3012882
## HmRun         HmRun  2.1861416
## AtBat         AtBat  2.0168822
## CHmRun       CHmRun  1.8991880
## League        League  1.1888536
## Errors        Errors  1.0340862
## Division    Division  0.7605174
## Assists      Assists  0.7414765
## NewLeague NewLeague  0.2333887
```

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

```r
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:

```r
gbm_predictions <- predict(gbm_fit_optimal,
  n.trees = optimal_num_trees,
  newdata = hitters_test
)
gbm_predictions
```

```
##  [1] 6.831802 4.854461 5.101919 4.828740 5.852958 4.874117 7.062562 6.475216
##  [9] 6.276690 6.562364 7.222613 5.631181 6.319916 7.011691 5.604990 6.401199
## [17] 4.885999 6.619110 6.148456 6.070939 6.624252 5.735051 6.693219 6.371433
## [25] 6.260884 7.076805 4.622458 6.116391 6.518448 5.617082 5.074350 6.575636
## [33] 5.124591 4.867925 6.209501 6.231701 6.414159 6.478142 6.160561 6.557035
## [41] 6.321999 7.048679 6.487258 4.970457 6.406497 7.043698 5.091225 6.509705
## [49] 6.213738 5.479731 5.072152 6.747629 6.010758
```

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

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

```
## [1] 0.5225422
```

Boosting for classification

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

```
heart_data <- read_csv("heart-data.csv")
heart_data <- heart_data |>
  na.omit() |>
  mutate(AHD = ifelse(AHD == "Yes", 1, 0)) |> # gbm expects response to be 0-1,
  #  NOT factor (unlike RF)
  mutate_at(c("ChestPain", "Thal"), as.factor)

set.seed(1) # set seed for reproducibility
train_samples <- sample(1:nrow(heart_data), round(0.8 * nrow(heart_data)))
heart_train <- heart_data |> filter(row_number() %in% train_samples)
heart_test <- heart_data |> filter(!(row_number() %in% train_samples))
```

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

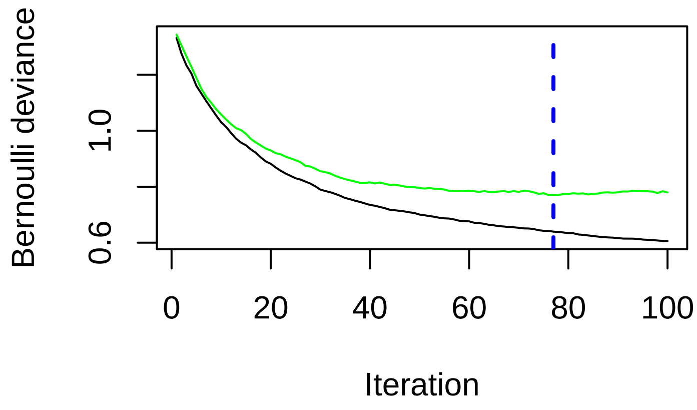
```
set.seed(1)
gbm_fit <- gbm(AHD ~ .,
  distribution = "bernoulli",
  n.trees = 100,
  interaction.depth = 1,
  shrinkage = 0.1,
  cv.folds = 5,
  n.cores = 1,
  data = heart_train
)
```

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

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] 77
```

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

```r
# 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,
  n.cores = 1,
  data = heart_train
)
```

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

```r
set.seed(1)
gbm_fit_2 <- gbm(AHD ~ .,
  distribution = "bernoulli",
  n.trees = 100,
  interaction.depth = 2,
  shrinkage = 0.1,
  cv.folds = 5,
  n.cores = 1,
  data = heart_train
)
```

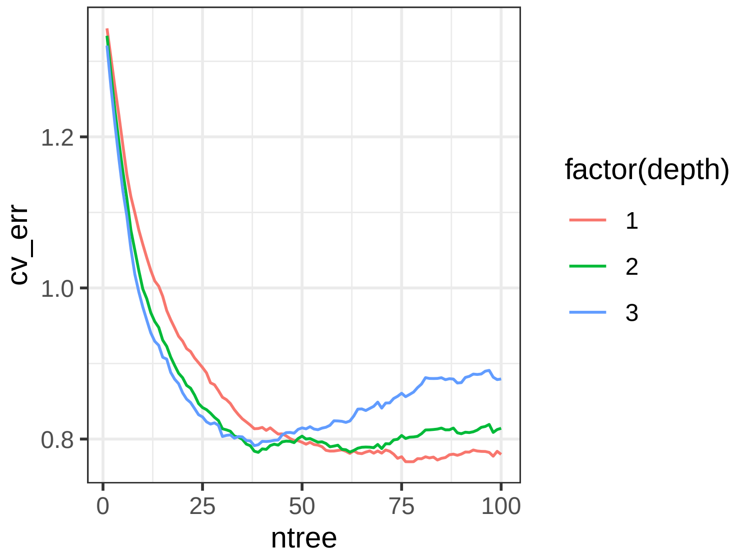
```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

```r
set.seed(1)
gbm_fit_3 <- gbm(AHD ~ .,
  distribution = "bernoulli",
  n.trees = 100,
  interaction.depth = 3,
  shrinkage = 0.1,
  cv.folds = 5,
  n.cores = 1,
  data = heart_train
)
```

```
## CV: 1
## CV: 2
## CV: 3
## CV: 4
## CV: 5
```

```
# 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()
```



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)
```

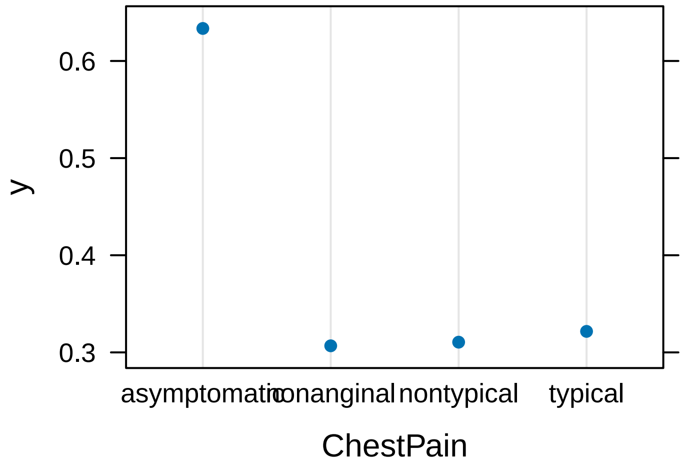
We can calculate variable importance scores as before:

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

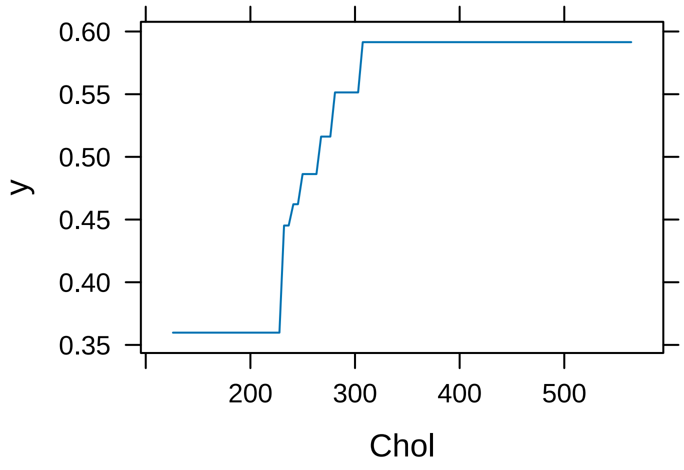
```
##                   var   rel.inf
## Ca                 Ca 20.911695
## Thal             Thal 20.715853
## ChestPain ChestPain 19.630662
## Oldpeak     Oldpeak 10.367138
## MaxHR         MaxHR  8.670123
## Slope         Slope  4.501328
## Chol           Chol  4.274883
## Sex             Sex  3.739401
## RestBP       RestBP  3.282853
## ExAng         ExAng  1.977479
## Age             Age  1.928587
## 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:

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



```r
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] 0.96620147 0.02992743 0.29116984 0.75977881 0.50966259 0.27775512
##  [7] 0.14283490 0.89740570 0.80802970 0.11460335 0.34865981 0.02499022
## [13] 0.93307631 0.31564744 0.42130879 0.96686169 0.97776659 0.18773369
## [19] 0.29644386 0.96426094 0.13281497 0.22665572 0.93670408 0.02165498
## [25] 0.95205933 0.03501518 0.67477444 0.25972467 0.04831125 0.02499022
## [31] 0.82062702 0.20394105 0.98749740 0.41927147 0.92255292 0.83583634
## [37] 0.64212727 0.05523759 0.82151034 0.47882468 0.59532237 0.09433372
## [43] 0.60389676 0.31564744 0.94746291 0.02242510 0.04894303 0.04848601
## [49] 0.53510851 0.10956111 0.25512309 0.26055317 0.25288430 0.93324646
## [55] 0.04992182 0.90241474 0.10228066 0.91000195 0.20056269
```

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] 0.1525424
```