Logistic Regression

In this chapter, we continue our discussion of classification. We introduce our first model for classification, logistic regression. To begin, we return to the `Default` dataset from the previous chapter.

```r
library(ISLR)
library(tibble)
as_tibble(Default)
```

```
## # A tibble: 10,000 x 4
##    default student balance income
##    <fct>   <fct>     <dbl>  <dbl>
##  1 No      No         730. 44362.
##  2 No      Yes        817. 12106.
##  3 No      No        1074. 31767.
##  4 No      No         529. 35704.
##  5 No      No         786. 38463.
##  6 No      Yes        920.  7492.
##  7 No      No         826. 24905.
##  8 No      Yes        809. 17600.
##  9 No      No        1161. 37469.
## 10 No      No           0  29275.
## # ... with 9,990 more rows
```

We also repeat the test-train split from the previous chapter.

```r
set.seed(42)
default_idx = sample(nrow(Default), 5000)
default_trn = Default[default_idx, ]
default_tst = Default[-default_idx, ]
```

Linear Regression

Before moving on to logistic regression, why not plain, old, linear regression?

```r
default_trn_lm = default_trn
default_tst_lm = default_tst
```

Since linear regression expects a numeric response variable, we coerce the response to be numeric. (Notice that we also shift the results, as we require `0` and `1`, not `1` and `2`.) Notice we have also copied the dataset so that we can return the original data with factors later.

```r
default_trn_lm$default = as.numeric(default_trn_lm$default) - 1
default_tst_lm$default = as.numeric(default_tst_lm$default) - 1
```

Why would we think this should work? Recall that,

$$\hat{\mathbb{E}}[Y \mid X = x] = X\hat{\beta}.$$

Since Y is limited to values of 0 and 1, we have

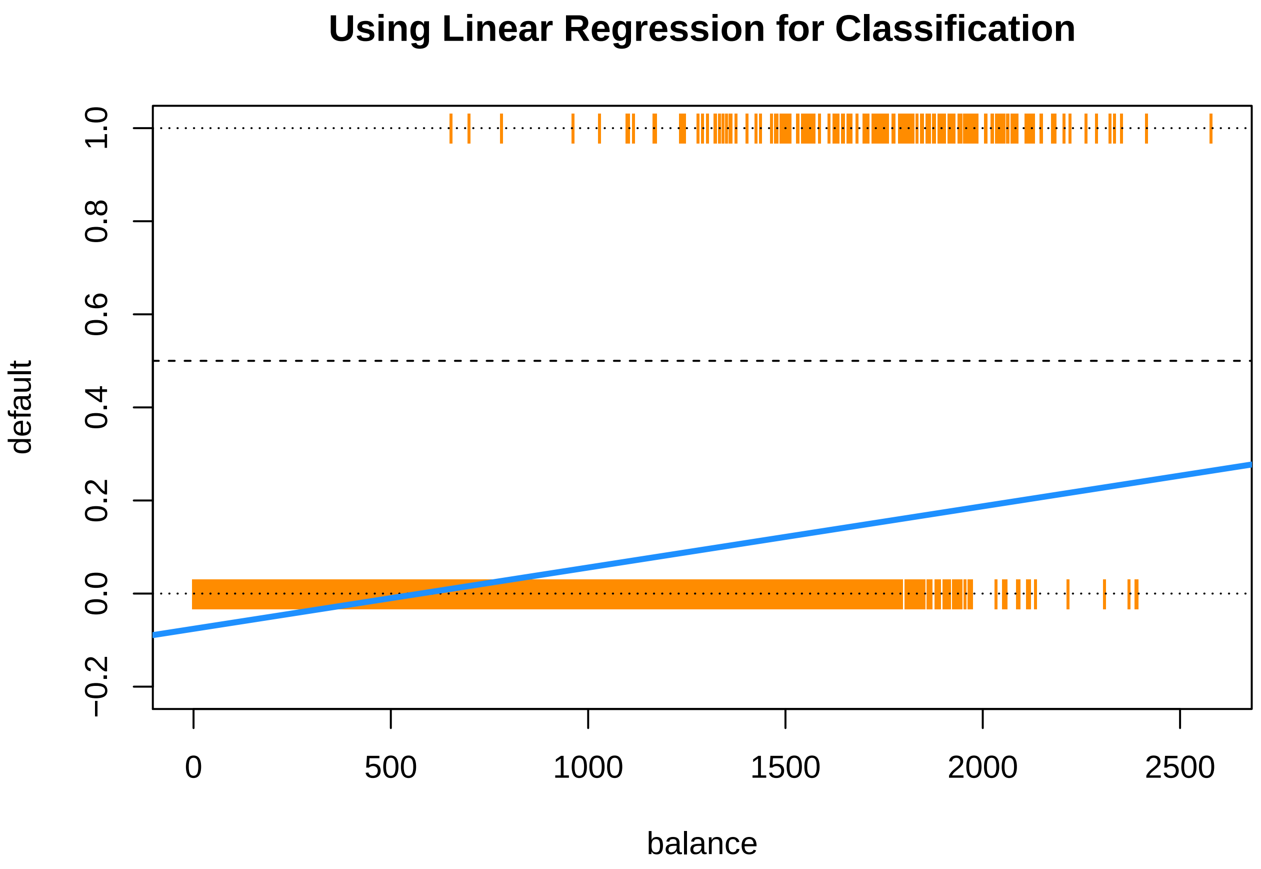
$$\mathbb{E}[Y \mid X = x] = P(Y = 1 \mid X = x).$$

It would then seem reasonable that $\mathbf{X}\hat{\beta}$ is a reasonable estimate of $P(Y = 1 \mid X = x)$. We test this on the `Default` data.

```r
model_lm = lm(default ~ balance, data = default_trn_lm)
```

Everything seems to be working, until we plot the results.

```r
plot(default ~ balance, data = default_trn_lm,
     col = "darkorange", pch = "|", ylim = c(-0.2, 1),
     main = "Using Linear Regression for Classification")
abline(h = 0, lty = 3)
abline(h = 1, lty = 3)
abline(h = 0.5, lty = 2)
abline(model_lm, lwd = 3, col = "dodgerblue")
```



Two issues arise. First, all of the predicted probabilities are below 0.5. That means, we would classify every observation as a "No". This is certainly possible, but not what we would expect.

```r
all(predict(model_lm) < 0.5)
```

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

The next, and bigger issue, is predicted probabilities less than 0.

```r
any(predict(model_lm) < 0)
```

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

Logistic Regression with `glm()`

To better estimate the probability

$$p(x) = P(Y = 1 \mid X = x)$$

we turn to logistic regression. The model is written

$$\log\left(\frac{p(x)}{1 - p(x)}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

Rearranging, we see the probabilities can be written as

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)}} = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)$$

Notice, we use the sigmoid function as shorthand notation, which appears often in deep learning literature. It takes any real input, and outputs a number between 0 and 1. How useful! (This is actualy a particular sigmoid function called the logistic function, but since it is by far the most popular sigmoid function, often sigmoid function is used to refer to the logistic function)

$$\sigma(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}$$

The model is fit by numerically maximizing the likelihood, which we will let `R` take care of.

We start with a single predictor example, again using `balance` as our single predictor.

```r
model_glm = glm(default ~ balance, data = default_trn, family = "binomial")
```

Fitting this model looks very similar to fitting a simple linear regression. Instead of `lm()` we use `glm()`. The only other difference is the use of `family = "binomial"` which indicates that we have a two-class categorical response. Using `glm()` with `family = "gaussian"` would perform the usual linear regression.

First, we can obtain the fitted coefficients the same way we did with linear regression.

```r
coef(model_glm)
```

```
##   (Intercept)       balance 
## -10.493158288   0.005424994
```

The next thing we should understand is how the `predict()` function works with `glm()`. So, let's look at some predictions.

```r
head(predict(model_glm))
```

```
##      2369      5273      9290      1252      8826       356 
## -5.376670 -4.875653 -5.018746 -4.007664 -6.538414 -6.601582
```

By default, `predict.glm()` uses `type = "link"`.

```r
head(predict(model_glm, type = "link"))
```

```
##      2369      5273      9290      1252      8826       356 
## -5.376670 -4.875653 -5.018746 -4.007664 -6.538414 -6.601582
```

That is, `R` is returning

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p$$

for each observation.

Importantly, these are **not** predicted probabilities. To obtain the predicted probabilities

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x)$$

we need to use `type = "response"`

```r
head(predict(model_glm, type = "response"))
```

```
##        2369        5273        9290        1252        8826         356
## 0.004601914 0.007572331 0.006569370 0.017851333 0.001444691 0.001356375
```

Note that these are probabilities, **not** classifications. To obtain classifications, we will need to compare to the correct cutoff value with an `ifelse()` statement.

```r
model_glm_pred = ifelse(predict(model_glm, type = "link") > 0, "Yes", "No")
# model_glm_pred = ifelse(predict(model_glm, type = "response") > 0.5, "Yes", "No")
```

The line that is run is performing

$$\hat{C}(x) = \begin{cases} 1 & \hat{f}(x) > 0 \\ 0 & \hat{f}(x) \leq 0 \end{cases}$$

where

$$\hat{f}(x) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \cdots + \hat{\beta}_p x_p.$$

The commented line, which would give the same results, is performing

$$\hat{C}(x) = \begin{cases} 1 & \hat{p}(x) > 0.5 \\ 0 & \hat{p}(x) \leq 0.5 \end{cases}$$

where

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x).$$

Once we have classifications, we can calculate metrics such as the trainging classification error rate.

```r
calc_class_err = function(actual, predicted) {
  mean(actual != predicted)
}
```

```r
calc_class_err(actual = default_trn$default, predicted = model_glm_pred)
```

```
## [1] 0.0284
```

As we saw previously, the `table()` and `confusionMatrix()` functions can be used to quickly obtain many more metrics.

```r
train_tab = table(predicted = model_glm_pred, actual = default_trn$default)
library(caret)
train_con_mat = confusionMatrix(train_tab, positive = "Yes")
c(train_con_mat$overall["Accuracy"],
  train_con_mat$byClass["Sensitivity"],
  train_con_mat$byClass["Specificity"])
```

```
##    Accuracy Sensitivity Specificity
##   0.9716000   0.2941176   0.9954451
```

We could also write a custom function for the error for use with trained logist regression models.

```
get_logistic_error = function(mod, data, res = "y", pos = 1, neg = 0, cut = 0.5) {
  probs = predict(mod, newdata = data, type = "response")
  preds = ifelse(probs > cut, pos, neg)
  calc_class_err(actual = data[, res], predicted = preds)
}
```

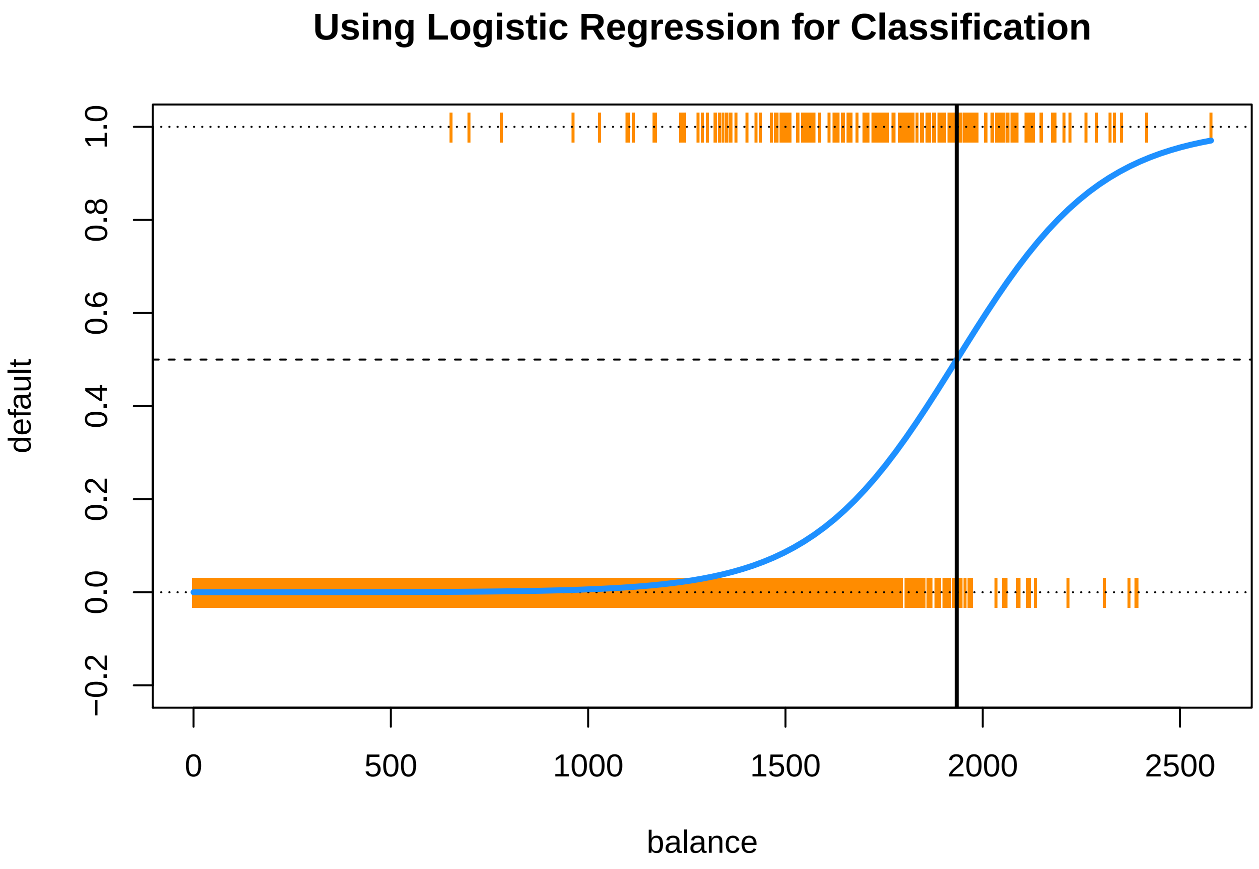
This function will be useful later when calculating train and test errors for several models at the same time.

```
get_logistic_error(model_glm, data = default_trn,
                   res = "default", pos = "Yes", neg = "No", cut = 0.5)
```

```
## [1] 0.0284
```

To see how much better logistic regression is for this task, we create the same plot we used for linear regression.

```
plot(default ~ balance, data = default_trn_lm,
     col = "darkorange", pch = "|", ylim = c(-0.2, 1),
     main = "Using Logistic Regression for Classification")
abline(h = 0, lty = 3)
abline(h = 1, lty = 3)
abline(h = 0.5, lty = 2)
curve(predict(model_glm, data.frame(balance = x), type = "response"),
      add = TRUE, lwd = 3, col = "dodgerblue")
abline(v = -coef(model_glm)[1] / coef(model_glm)[2], lwd = 2)
```



This plot contains a wealth of information.

- The orange | characters are the data, (x_i, y_i).

- The blue "curve" is the predicted probabilities given by the fitted logistic regression. That is,

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x)$$

- The solid vertical black line represents the **decision boundary**, the `balance` that obtains a predicted probability of 0.5. In this case `balance` = 1934.2247145.

The decision boundary is found by solving for points that satisfy

$$\hat{p}(x) = \hat{P}(Y = 1 \mid X = x) = 0.5$$

This is equivalent to point that satisfy

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 = 0.$$

Thus, for logistic regression with a single predictor, the decision boundary is given by the *point*

$$x_1 = \frac{-\hat{\beta}_0}{\hat{\beta}_1}.$$

The following is not run, but an alternative way to add the logistic curve to the plot.

```r
grid = seq(0, max(default_trn$balance), by = 0.01)

sigmoid = function(x) {
  1 / (1 + exp(-x))
}

lines(grid, sigmoid(coef(model_glm)[1] + coef(model_glm)[2] * grid), lwd = 3)
```

Using the usual formula syntax, it is easy to add or remove complexity from logistic regressions.

```r
model_1 = glm(default ~ 1, data = default_trn, family = "binomial")
model_2 = glm(default ~ ., data = default_trn, family = "binomial")
model_3 = glm(default ~ . ^ 2 + I(balance ^ 2),
              data = default_trn, family = "binomial")
```

Note that, using polynomial transformations of predictors will allow a linear model to have non-linear decision boundaries.

```r
model_list = list(model_1, model_2, model_3)
train_errors = sapply(model_list, get_logistic_error, data = default_trn,
                      res = "default", pos = "Yes", neg = "No", cut = 0.5)
test_errors  = sapply(model_list, get_logistic_error, data = default_tst,
                      res = "default", pos = "Yes", neg = "No", cut = 0.5)
```

Here we see the misclassification error rates for each model. The train decreases, and the test decreases, until it starts to increases. Everything we learned about the bias-variance tradeoff for regression also applies here.

```r
diff(train_errors)
```

```
## [1] -0.0066  0.0000
```

```r
diff(test_errors)
```

```
## [1] -0.0068  0.0006
```

We call `model_2` the **additive** logistic model, which we will use quite often.

ROC Curves

Let's return to our simple model with only balance as a predictor.

```r
model_glm = glm(default ~ balance, data = default_trn, family = "binomial")
```

We write a function which allows use to make predictions based on different probability cutoffs.

```r
get_logistic_pred = function(mod, data, res = "y", pos = 1, neg = 0, cut = 0.5) {
  probs = predict(mod, newdata = data, type = "response")
  ifelse(probs > cut, pos, neg)
}
```

$$\hat{C}(x) = \begin{cases} 1 & \hat{p}(x) > c \\ 0 & \hat{p}(x) \leq c \end{cases}$$

Let's use this to obtain predictions using a low, medium, and high cutoff. (0.1, 0.5, and 0.9)

```r
test_pred_10 = get_logistic_pred(model_glm, data = default_tst, res = "default",
                                 pos = "Yes", neg = "No", cut = 0.1)
test_pred_50 = get_logistic_pred(model_glm, data = default_tst, res = "default",
                                 pos = "Yes", neg = "No", cut = 0.5)
test_pred_90 = get_logistic_pred(model_glm, data = default_tst, res = "default",
                                 pos = "Yes", neg = "No", cut = 0.9)
```

Now we evaluate accuracy, sensitivity, and specificity for these classifiers.

```r
test_tab_10 = table(predicted = test_pred_10, actual = default_tst$default)
test_tab_50 = table(predicted = test_pred_50, actual = default_tst$default)
test_tab_90 = table(predicted = test_pred_90, actual = default_tst$default)

test_con_mat_10 = confusionMatrix(test_tab_10, positive = "Yes")
test_con_mat_50 = confusionMatrix(test_tab_50, positive = "Yes")
test_con_mat_90 = confusionMatrix(test_tab_90, positive = "Yes")
```

```r
metrics = rbind(

  c(test_con_mat_10$overall["Accuracy"],
    test_con_mat_10$byClass["Sensitivity"],
    test_con_mat_10$byClass["Specificity"]),

  c(test_con_mat_50$overall["Accuracy"],
    test_con_mat_50$byClass["Sensitivity"],
    test_con_mat_50$byClass["Specificity"]),

  c(test_con_mat_90$overall["Accuracy"],
    test_con_mat_90$byClass["Sensitivity"],
    test_con_mat_90$byClass["Specificity"])

)

rownames(metrics) = c("c = 0.10", "c = 0.50", "c = 0.90")
metrics
```

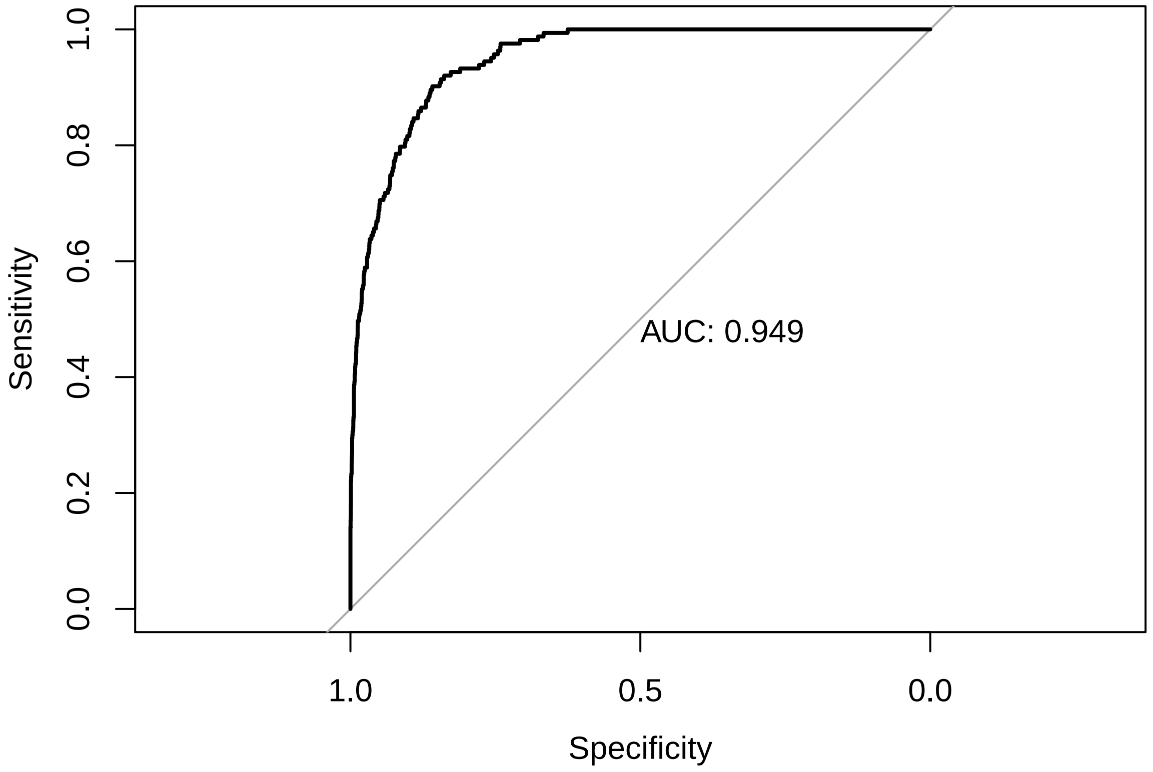
```
##          Accuracy Sensitivity Specificity
## c = 0.10   0.9328  0.71779141   0.9400455
## c = 0.50   0.9730  0.31288344   0.9952450
## c = 0.90   0.9688  0.04294479   1.0000000
```

We see then sensitivity decreases as the cutoff is increased. Conversely, specificity increases as the cutoff increases. This is useful if we are more interested in a particular error, instead of giving them equal weight.

Note that usually the best accuracy will be seen near $c = 0.50$.

Instead of manually checking cutoffs, we can create an ROC curve (receiver operating characteristic curve) which will sweep through all possible cutoffs, and plot the sensitivity and specificity.

```r
library(pROC)
test_prob = predict(model_glm, newdata = default_tst, type = "response")
test_roc = roc(default_tst$default ~ test_prob, plot = TRUE, print.auc = TRUE)
```



```r
as.numeric(test_roc$auc)
```

```
## [1] 0.9492866
```

A good model will have a high AUC, that is as often as possible a high sensitivity and specificity.

Multinomial Logistic Regression

What if the response contains more than two categories? For that we need multinomial logistic regression.

$$P(Y = k \mid X = x) = \frac{e^{\beta_{0k} + \beta_{1k}x_1 + \cdots + + \beta_{pk}x_p}}{\sum_{g=1}^{G} e^{\beta_{0g} + \beta_{1g}x_1 + \cdots + \beta_{pg}x_p}}$$

We will omit the details, as ISL has as well. If you are interested, the Wikipedia page provides a rather thorough coverage. Also note that the above is an example of the softmax function.

As an example of a dataset with a three category response, we use the `iris` dataset, which is so famous, it has its own Wikipedia entry. It is also a default dataset in `R`, so no need to load it.

Before proceeding, we test-train split this data.

```r
set.seed(430)
iris_obs = nrow(iris)
iris_idx = sample(iris_obs, size = trunc(0.50 * iris_obs))
iris_trn = iris[iris_idx, ]
iris_test = iris[-iris_idx, ]
```

To perform multinomial logistic regression, we use the `multinom` function from the `nnet` package. Training using `multinom()` is done using similar syntax to `lm()` and `glm()`. We add the `trace = FALSE` argument to suppress information about updates to the optimization routine as the model is trained.

```r
library(nnet)
model_multi = multinom(Species ~ ., data = iris_trn, trace = FALSE)
summary(model_multi)$coefficients
```

```
##            (Intercept) Sepal.Length Sepal.Width Petal.Length Petal.Width
## versicolor    16.77474    -7.855576   -13.98668     25.13860    4.270375
## virginica    -33.94895   -37.519645   -94.22846     97.82691   73.487162
```

Notice we are only given coefficients for two of the three class, much like only needing coefficients for one class in logistic regression.

A difference between `glm()` and `multinom()` is how the `predict()` function operates.

```r
head(predict(model_multi, newdata = iris_trn))
```

```
## [1] setosa     versicolor versicolor setosa     virginica  versicolor
## Levels: setosa versicolor virginica
```

```r
head(predict(model_multi, newdata = iris_trn, type = "prob"))
```

```
##           setosa   versicolor     virginica
## 1   1.000000e+00 1.910554e-16 6.118616e-176
## 92  8.542846e-22 1.000000e+00  1.372168e-18
## 77  8.343856e-23 1.000000e+00  2.527471e-14
## 38  1.000000e+00 1.481126e-16 5.777917e-180
## 108 1.835279e-73 1.403654e-36  1.000000e+00
## 83  1.256090e-16 1.000000e+00  2.223689e-32
```

Notice that by default, classifications are returned. When obtaining probabilities, we are given the predicted probability for **each** class.

Interestingly, you've just fit a neural network, and you didn't even know it! (Hence the `nnet` package.) Later we will discuss the connections between logistic regression, multinomial logistic regression, and simple neural networks.

`rmarkdown`

The `rmarkdown` file for this chapter can be found **here**. The file was created using `R` version 4.0.2. The following packages (and their dependencies) were loaded when knitting this file:

```
## [1] "nnet"    "pROC"    "caret"   "ggplot2" "lattice" "tibble"  "ISLR"
```