3.6 Lab: Linear Regression

3.6.1 Libraries

The `library()` function is used to load *libraries*, or groups of functions and data sets that are not included in the base `R` distribution. Basic functions that perform least squares linear regression and other simple analyses come standard with the base distribution, but more exotic functions require additional libraries. Here we load the `MASS` package, which is a very large collection of data sets and functions. We also load the `ISLR2` package, which includes the data sets associated with this book.

`library()`

```
> library(MASS)
> library(ISLR2)
```

If you receive an error message when loading any of these libraries, it likely indicates that the corresponding library has not yet been installed on your system. Some libraries, such as `MASS`, come with `R` and do not need to be separately installed on your computer. However, other packages, such as

`ISLR2`, must be downloaded the first time they are used. This can be done directly from within `R`. For example, on a Windows system, select the `Install package` option under the `Packages` tab. After you select any mirror site, a list of available packages will appear. Simply select the package you wish to install and `R` will automatically download the package. Alternatively, this can be done at the `R` command line via `install.packages("ISLR2")`. This installation only needs to be done the first time you use a package. However, the `library()` function must be called within each `R` session.

3.6.2 Simple Linear Regression

The `ISLR2` library contains the `Boston` data set, which records `medv` (median house value) for 506 census tracts in Boston. We will seek to predict `medv` using 12 predictors such as `rm` (average number of rooms per house), `age` (average age of houses), and `lstat` (percent of households with low socioeconomic status).

```
> head(Boston)
     crim zn indus chas   nox    rm  age    dis rad tax
1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296
2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242
3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242
4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222
5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222
6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222
  ptratio lstat medv
1    15.3  4.98 24.0
2    17.8  9.14 21.6
3    17.8  4.03 34.7
4    18.7  2.94 33.4
5    18.7  5.33 36.2
6    18.7  5.21 28.7
```

To find out more about the data set, we can type `?Boston`.

We will start by using the `lm()` function to fit a simple linear regression model, with `medv` as the response and `lstat` as the predictor. The basic syntax is `lm(y ∼ x, data)`, where `y` is the response, `x` is the predictor, and `data` is the data set in which these two variables are kept. `lm()`

```
> lm.fit <- lm(medv ∼ lstat)
Error in eval(expr, envir, enclos) : Object "medv" not found
```

The command causes an error because `R` does not know where to find the variables `medv` and `lstat`. The next line tells `R` that the variables are in `Boston`. If we attach `Boston`, the first line works fine because `R` now recognizes the variables.

```
> lm.fit <- lm(medv ∼ lstat, data = Boston)
> attach(Boston)
> lm.fit <- lm(medv ∼ lstat)
```

If we type `lm.fit`, some basic information about the model is output. For more detailed information, we use `summary(lm.fit)`. This gives us p-values and standard errors for the coefficients, as well as the R^2 statistic and F-statistic for the model.

```
> lm.fit

Call:
lm(formula = medv ~ lstat)

Coefficients:
(Intercept)        lstat
      34.55        -0.95

> summary(lm.fit)

Call:
lm(formula = medv ~ lstat)

Residuals:
   Min      1Q  Median      3Q     Max
-15.17   -3.99   -1.32    2.03   24.50

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  34.5538     0.5626    61.4   <2e-16 ***
lstat        -0.9500     0.0387   -24.5   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 6.22 on 504 degrees of freedom
Multiple R-squared: 0.544,      Adjusted R-squared: 0.543
F-statistic:  602 on 1 and 504 DF,  p-value: < 2e-16
```

We can use the `names()` function in order to find out what other pieces of information are stored in `lm.fit`. Although we can extract these quantities by name—e.g. `lm.fit$coefficients`—it is safer to use the extractor functions like `coef()` to access them.

`names()`

`coef()`

```
> names(lm.fit)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
> coef(lm.fit)
(Intercept)       lstat
      34.55       -0.95
```

In order to obtain a confidence interval for the coefficient estimates, we can use the `confint()` command.

`confint()`

```
> confint(lm.fit)
            2.5 % 97.5 %
(Intercept) 33.45 35.659
```

```
lstat        -1.03 -0.874
```

The `predict()` function can be used to produce confidence intervals and prediction intervals for the prediction of `medv` for a given value of `lstat`. `predict()`

```
> predict(lm.fit, data.frame(lstat = (c(5, 10, 15))),
    interval = "confidence")
    fit   lwr   upr
1 29.80 29.01 30.60
2 25.05 24.47 25.63
3 20.30 19.73 20.87
> predict(lm.fit, data.frame(lstat = (c(5, 10, 15))),
    interval = "prediction")
    fit    lwr   upr
1 29.80 17.566 42.04
2 25.05 12.828 37.28
3 20.30  8.078 32.53
```

For instance, the 95 % confidence interval associated with a `lstat` value of 10 is $(24.47, 25.63)$, and the 95 % prediction interval is $(12.828, 37.28)$. As expected, the confidence and prediction intervals are centered around the same point (a predicted value of 25.05 for `medv` when `lstat` equals 10), but the latter are substantially wider.

We will now plot `medv` and `lstat` along with the least squares regression line using the `plot()` and `abline()` functions. `abline()`

```
> plot(lstat, medv)
> abline(lm.fit)
```

There is some evidence for non-linearity in the relationship between `lstat` and `medv`. We will explore this issue later in this lab.

The `abline()` function can be used to draw any line, not just the least squares regression line. To draw a line with intercept `a` and slope `b`, we type `abline(a, b)`. Below we experiment with some additional settings for plotting lines and points. The `lwd = 3` command causes the width of the regression line to be increased by a factor of 3; this works for the `plot()` and `lines()` functions also. We can also use the `pch` option to create different plotting symbols.

```
> abline(lm.fit, lwd = 3)
> abline(lm.fit, lwd = 3, col = "red")
> plot(lstat, medv, col = "red")
> plot(lstat, medv, pch = 20)
> plot(lstat, medv, pch = "+")
> plot(1:20, 1:20, pch = 1:20)
```

Next we examine some diagnostic plots, several of which were discussed in Section 3.3.3. Four diagnostic plots are automatically produced by applying the `plot()` function directly to the output from `lm()`. In general, this command will produce one plot at a time, and hitting *Enter* will generate the next plot. However, it is often convenient to view all four plots together. We can achieve this by using the `par()` and `mfrow()` functions, which tell `R` `par()` `mfrow()`

to split the display screen into separate panels so that multiple plots can be viewed simultaneously. For example, `par(mfrow = c(2, 2))` divides the plotting region into a 2×2 grid of panels.

```
> par(mfrow = c(2, 2))
> plot(lm.fit)
```

Alternatively, we can compute the residuals from a linear regression fit using the `residuals()` function. The function `rstudent()` will return the studentized residuals, and we can use this function to plot the residuals against the fitted values.

`residuals()`
`rstudent()`

```
> plot(predict(lm.fit), residuals(lm.fit))
> plot(predict(lm.fit), rstudent(lm.fit))
```

On the basis of the residual plots, there is some evidence of non-linearity. Leverage statistics can be computed for any number of predictors using the `hatvalues()` function.

`hatvalues()`

```
> plot(hatvalues(lm.fit))
> which.max(hatvalues(lm.fit))
375
```

The `which.max()` function identifies the index of the largest element of a vector. In this case, it tells us which observation has the largest leverage statistic.

`which.max()`

3.6.3 Multiple Linear Regression

In order to fit a multiple linear regression model using least squares, we again use the `lm()` function. The syntax `lm(y ∼ x1 + x2 + x3)` is used to fit a model with three predictors, `x1`, `x2`, and `x3`. The `summary()` function now outputs the regression coefficients for all the predictors.

```
> lm.fit <- lm(medv ∼ lstat + age, data = Boston)
> summary(lm.fit)

Call:
lm(formula = medv ∼ lstat + age, data = Boston)

Residuals:
   Min     1Q Median     3Q    Max
-15.98  -3.98  -1.28   1.97  23.16

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  33.2228     0.7308   45.46   <2e-16 ***
lstat        -1.0321     0.0482  -21.42   <2e-16 ***
age           0.0345     0.0122    2.83   0.0049 **
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 6.17 on 503 degrees of freedom
```

```
Multiple R-squared: 0.551,      Adjusted R-squared: 0.549
F-statistic:  309 on 2 and 503 DF,  p-value: < 2e-16
```

The `Boston` data set contains 12 variables, and so it would be cumbersome to have to type all of these in order to perform a regression using all of the predictors. Instead, we can use the following short-hand:

```
> lm.fit <- lm(medv ~ ., data = Boston)
> summary(lm.fit)

Call:
lm(formula = medv ~ ., data = Boston)

Residuals:
    Min      1Q  Median      3Q     Max
-15.130  -2.767  -0.581   1.941  26.253

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  41.61727    4.93604    8.43  3.8e-16 ***
crim         -0.12139    0.03300   -3.68  0.00026 ***
zn            0.04696    0.01388    3.38  0.00077 ***
indus         0.01347    0.06214    0.22  0.82852
chas          2.83999    0.87001    3.26  0.00117 **
nox         -18.75802    3.85135   -4.87  1.5e-06 ***
rm            3.65812    0.42025    8.70  < 2e-16 ***
age           0.00361    0.01333    0.27  0.78659
dis          -1.49075    0.20162   -7.39  6.2e-13 ***
rad           0.28940    0.06691    4.33  1.8e-05 ***
tax          -0.01268    0.00380   -3.34  0.00091 ***
ptratio      -0.93753    0.13221   -7.09  4.6e-12 ***
lstat        -0.55202    0.05066  -10.90  < 2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 4.8 on 493 degrees of freedom
Multiple R-squared:  0.734,     Adjusted R-squared:  0.728
F-statistic:  114 on 12 and 493 DF,  p-value: < 2e-16
```

We can access the individual components of a summary object by name (type `?summary.lm` to see what is available). Hence `summary(lm.fit)$r.sq` gives us the R^2, and `summary(lm.fit)$sigma` gives us the RSE. The `vif()` function, part of the `car` package, can be used to compute variance inflation factors. Most VIF's are low to moderate for this data. The `car` package is not part of the base `R` installation so it must be downloaded the first time you use it via the `install.packages()` function in `R`.

`vif()`

```
> library(car)
> vif(lm.fit)
   crim      zn   indus    chas     nox      rm     age     dis
   1.77    2.30    3.99    1.07    4.37    1.91    3.09    3.95
    rad     tax ptratio   lstat
   7.45    9.00    1.80    2.87
```

What if we would like to perform a regression using all of the variables but one? For example, in the above regression output, `age` has a high p-value. So we may wish to run a regression excluding this predictor. The following syntax results in a regression using all predictors except `age`.

```
> lm.fit1 <- lm(medv ∼ . - age, data = Boston)
> summary(lm.fit1)
...
```

Alternatively, the `update()` function can be used.

`update()`

```
> lm.fit1 <- update(lm.fit, ∼ . - age)
```

3.6.4 Interaction Terms

It is easy to include interaction terms in a linear model using the `lm()` function. The syntax `lstat:black` tells `R` to include an interaction term between `lstat` and `black`. The syntax `lstat * age` simultaneously includes `lstat`, `age`, and the interaction term `lstat`×`age` as predictors; it is a shorthand for `lstat + age + lstat:age`.

```
> summary(lm(medv ∼ lstat * age, data = Boston))

Call:
lm(formula = medv ∼ lstat * age, data = Boston)

Residuals:
   Min     1Q Median     3Q    Max
-15.81  -4.04  -1.33   2.08  27.55

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 36.088536   1.469835   24.55  < 2e-16 ***
lstat       -1.392117   0.167456   -8.31  8.8e-16 ***
age         -0.000721   0.019879   -0.04    0.971
lstat:age    0.004156   0.001852    2.24    0.025 *
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 6.15 on 502 degrees of freedom
Multiple R-squared: 0.556,  Adjusted R-squared: 0.553
F-statistic:  209 on 3 and 502 DF,  p-value: < 2e-16
```

3.6.5 Non-linear Transformations of the Predictors

The `lm()` function can also accommodate non-linear transformations of the predictors. For instance, given a predictor X, we can create a predictor X^2 using `I(X^2)`. The function `I()` is needed since the `^` has a special meaning in a formula object; wrapping as we do allows the standard usage in `R`, which is to raise `X` to the power `2`. We now perform a regression of `medv` onto `lstat` and $\texttt{lstat}^2$.

`I()`

```
> lm.fit2 <- lm(medv ~ lstat + I(lstat^2))
> summary(lm.fit2)

Call:
lm(formula = medv ~ lstat + I(lstat^2))

Residuals:
   Min     1Q Median     3Q    Max
-15.28  -3.83  -0.53   2.31  25.41

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 42.86201    0.87208    49.1   <2e-16 ***
lstat       -2.33282    0.12380   -18.8   <2e-16 ***
I(lstat^2)   0.04355    0.00375    11.6   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 5.52 on 503 degrees of freedom
Multiple R-squared: 0.641,  Adjusted R-squared: 0.639
F-statistic:  449 on 2 and 503 DF,  p-value: < 2e-16
```

The near-zero p-value associated with the quadratic term suggests that it leads to an improved model. We use the `anova()` function to further quantify the extent to which the quadratic fit is superior to the linear fit. `anova()`

```
> lm.fit <- lm(medv ~ lstat)
> anova(lm.fit, lm.fit2)
Analysis of Variance Table

Model 1: medv ~ lstat
Model 2: medv ~ lstat + I(lstat^2)
  Res.Df   RSS Df Sum of Sq   F Pr(>F)
1    504 19472
2    503 15347  1      4125 135 <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

Here Model 1 represents the linear submodel containing only one predictor, `lstat`, while Model 2 corresponds to the larger quadratic model that has two predictors, `lstat` and `lstat`^{2}. The `anova()` function performs a hypothesis test comparing the two models. The null hypothesis is that the two models fit the data equally well, and the alternative hypothesis is that the full model is superior. Here the F-statistic is 135 and the associated p-value is virtually zero. This provides very clear evidence that the model containing the predictors `lstat` and `lstat`^{2} is far superior to the model that only contains the predictor `lstat`. This is not surprising, since earlier we saw evidence for non-linearity in the relationship between `medv` and `lstat`. If we type

```
> par(mfrow = c(2, 2))
> plot(lm.fit2)
```

then we see that when the lstat^2 term is included in the model, there is little discernible pattern in the residuals.

In order to create a cubic fit, we can include a predictor of the form I(X^3). However, this approach can start to get cumbersome for higher-order polynomials. A better approach involves using the poly() function to create the polynomial within lm(). For example, the following command produces a fifth-order polynomial fit: poly()

```
> lm.fit5 <- lm(medv ~ poly(lstat, 5))
> summary(lm.fit5)

Call:
lm(formula = medv ~ poly(lstat, 5))

Residuals:
    Min       1Q   Median       3Q      Max
-13.543   -3.104   -0.705    2.084   27.115

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)       22.533      0.232   97.20  < 2e-16 ***
poly(lstat, 5)1 -152.460      5.215  -29.24  < 2e-16 ***
poly(lstat, 5)2   64.227      5.215   12.32  < 2e-16 ***
poly(lstat, 5)3  -27.051      5.215   -5.19  3.1e-07 ***
poly(lstat, 5)4   25.452      5.215    4.88  1.4e-06 ***
poly(lstat, 5)5  -19.252      5.215   -3.69  0.00025 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 5.21 on 500 degrees of freedom
Multiple R-squared: 0.682,   Adjusted R-squared: 0.679
F-statistic:  214 on 5 and 500 DF,  p-value: < 2e-16
```

This suggests that including additional polynomial terms, up to fifth order, leads to an improvement in the model fit! However, further investigation of the data reveals that no polynomial terms beyond fifth order have significant p-values in a regression fit.

By default, the poly() function orthogonalizes the predictors: this means that the features output by this function are not simply a sequence of powers of the argument. However, a linear model applied to the output of the poly() function will have the same fitted values as a linear model applied to the raw polynomials (although the coefficient estimates, standard errors, and p-values will differ). In order to obtain the raw polynomials from the poly() function, the argument raw = TRUE must be used.

Of course, we are in no way restricted to using polynomial transformations of the predictors. Here we try a log transformation.

```
> summary(lm(medv ~ log(rm), data = Boston))
...
```