Linear Regression Lab

Statistical Inference 1

9/26/2017

Contents

Note: This activity is based on the chapter 3 lab in ISLR, but is adapted to fit our course.

1 Getting the data

As in the ISLR book, let's use the data set `Boston` from the `MASS` package. This data set has information from 506 neighborhoods in Boston from the late 1970s. There are 14 variables, including `medv` (median value of owner-occupied homes in $1000s) and `lstat` (percent of households with a "low" socioeconomic status).

```
library(MASS)
names(Boston)
```

```
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "black"   "lstat"   "medv"
```

2 Calling `lm()`

```
lm.fit <- lm(medv ~ lstat)
```

```
# Two options:
lm.fit = lm(medv ~ lstat, data = Boston)

# OR
```

```
attach(Boston)
lm.fit = lm(medv ~ lstat)
```

You almost always want to save the output of the linear model call to an object (here, this is lm.fit). This is because many useful functions in R exist that use the linear model output as an argument. Pretty much every function we will learn about in this lab either depends on lm.fit or summary(lm.fit). We will almost never need the raw data after we have called lm().

3 Summary vs lm

```
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
## -15.168  -3.990  -1.318   2.034  24.500
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 34.55384    0.56263   61.41   <2e-16 ***
## lstat       -0.95005    0.03873  -24.53   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.216 on 504 degrees of freedom
## Multiple R-squared:  0.5441, Adjusted R-squared:  0.5432
## F-statistic: 601.6 on 1 and 504 DF,  p-value: < 2.2e-16
```

```
names(lm.fit)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

lm.fit prints very basic information– basically just the model and coefficient estimates. In contrast, calling summary() on a lm object provides a lot more detail, including standard error, t-statistics, p-values, R-squared, etc.

All of the information in lm() and summary() of lm can be pulled out using the appropriate R code. What you get may vary depending on which option you use. For instance, consider getting the regression coefficients...

4 Estimating the coefficients

There are multiple ways to access the coefficients for a linear model.

First, we could call `coef()` which produces a named numeric vector of coefficients. If we want to pull out a single element, we can use `[]` or `[[]]`

```
coef(lm.fit)
```

```
## (Intercept)       lstat
##  34.5538409  -0.9500494
```

```
coef(lm.fit)[2]
```

```
##      lstat
## -0.9500494
```

```
coef(lm.fit)[[2]]
```

```
## [1] -0.9500494
```

If you want to include the coefficient in some in-line R code in a R markdown file, you should use the `[[]]` which gives a single element and loses the name. For instance, the estimated coefficient for the `lstat` variable is -0.9500494.

Similarly, we could use`$` to access the coefficients, which also gives us a named numeric vector. Accessing the coefficients is the same as above.

```
lm.fit$coefficients
```

```
## (Intercept)       lstat
##  34.5538409  -0.9500494
```

Finally, we could use either of the above methods with the summary of the lm instead of the lm itself; this approach gives us a matrix rather than a vector. Incidentally, this is also how you can access the t-statistics and p-values for the coefficients:

```
lm.coef <- summary(lm.fit)$coefficients

coef(summary(lm.fit))
```

```
##               Estimate Std. Error   t value      Pr(>|t|)
## (Intercept) 34.5538409 0.56262735  61.41515 3.743081e-236
## lstat       -0.9500494 0.03873342 -24.52790  5.081103e-88
```

Now if we wanted to access the coefficient of the `lstat` variable, we would use matrix sub-setting:

```
coef(summary(lm.fit))[2, 1]
```

```
## [1] -0.9500494
```

We can use different row and column numbers to access the standard error, t-values, and p-values.

5 Making confidence intervals for beta

Making confidence intervals for the paramters is pretty straightforward. Just use the `confint` command and input the linear model object.

```
confint(lm.fit)
```

```
##                 2.5 %     97.5 %
## (Intercept) 33.448457 35.6592247
## lstat       -1.026148 -0.8739505
```

```
(lstat_ci <- confint(lm.fit, "lstat", level = 0.99))
```

```
##           0.5 %     99.5 %
## lstat -1.050199 -0.8498995
```

This produces either a data frame or a numeric vector and can be subset appropriately. For example, my 99% confidence interval for lstat goes from -1.0501992 to -0.8498995. It is also possible to choose different methods of calculating a confidence interval, but we will not focus on that yet.

If we want to make confidence intervals for Y-hat, we can use the predict function:

```
predict(lm.fit, data.frame(lstat = (c(5, 10, 15))), interval = "confidence")
```

```
##        fit      lwr      upr
## 1 29.80359 29.00741 30.59978
## 2 25.05335 24.47413 25.63256
## 3 20.30310 19.73159 20.87461
```

6 Fitted values with $\hat{Y}$

We just saw one way to get the fitted values for $\hat{Y}$ using the predict function, which gives us not only the fitted value, but also the confidence interval around that value. We can also use the makeFun function in the `mosaic` package (which is useful for coding tasks beyond just fitting y):

```
lm.predict <- makeFun(lm.fit)
lm.predict(5)
```

```
##        1
## 29.80359
```

```
lm.predict(c(5, 10, 15))
```

```
##        1        2        3
## 29.80359 25.05335 20.30310
```

```
lm.predict(c(5, 10, 15), interval = "confidence")
```

```
##        fit      lwr      upr
## 1 29.80359 29.00741 30.59978
## 2 25.05335 24.47413 25.63256
## 3 20.30310 19.73159 20.87461
```

7 Making prediction intervals

```
lm.predict(5, interval = "prediction")
```

```
##        fit      lwr      upr
## 1 29.80359 17.56567 42.04151
```

```
lm.predict(c(5, 10, 15), interval = "prediction")
```

```
##        fit       lwr      upr
## 1 29.80359 17.565675 42.04151
## 2 25.05335 12.827626 37.27907
## 3 20.30310  8.077742 32.52846
```

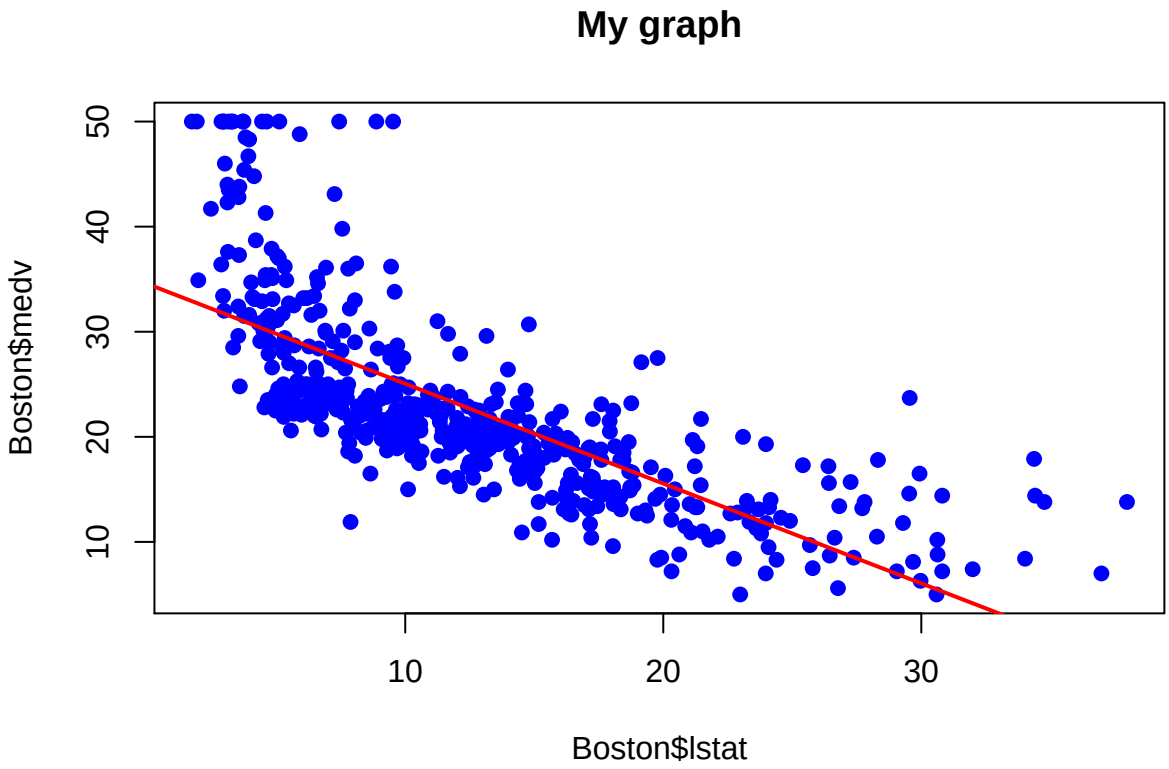
```
predict(lm.fit, data.frame(lstat = (c(5, 10, 15))), interval = "prediction")
```

```
##        fit       lwr      upr
## 1 29.80359 17.565675 42.04151
## 2 25.05335 12.827626 37.27907
## 3 20.30310  8.077742 32.52846
```

8 Making and describing scatterplots

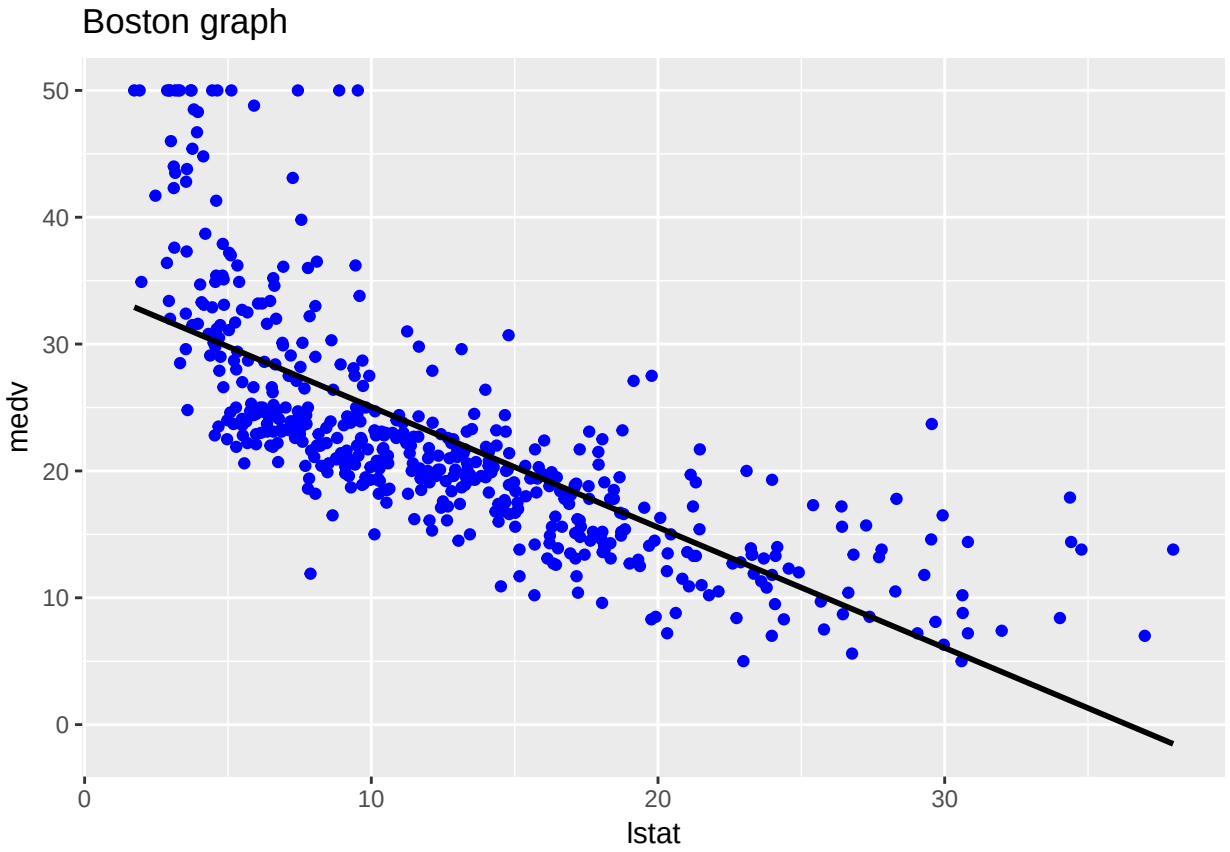
With base graphics:

```
plot(Boston$lstat, Boston$medv, pch = 16, cex = 1.1, col = "blue",
    main = "My graph")
abline(lm.fit, lwd = 2, col = "red")
```



With ggplot (make sure you've loaded the appropriate libraries first):

```
ggplot(data = Boston, aes(x = lstat, y = medv)) + geom_point(color = "blue") +
    geom_smooth(method = "lm", color = "black", se = FALSE) +
    ggtitle("Boston graph")
```



When describing a scatter plot, remember *DOTS*: *D*irection, *O*utliers, *T*rend, *S*trength

For our graph in this example: Direction: negative (as lstat gets bigger, medv gets smaller) Outliers: There appear to be some outliers at medv = 50 that don't fit the general pattern Trend: The graph appears linear– no curves or changing directions Strength: There appears to be a strong association/correlation between the two variables.

9 R^2, adjusted R^2, and correlation

Recall that R-squared is basically the percent of variation in y that is explained by x. We often report it as a decimal or as a percentage, and we use it as a measure of model fit. However, this introduces a slight problem– the more variables we add to a model, the higher R-squared will be (or at least, it will never decrease), even if the new variables are pretty much irrelevant. Adjusted R-squared is R-squared that has been penalized based on the number of parameters in the model. This means it is more helpful when choosing a model, because it strikes a balance between how well the model fits the data vs. how complicated the model is.

We can look up the R-squared value in the summary of the lm. If we want to actually retrieve the value, we can access it like this:

```
summary(lm.fit)$r.squared
```

```
## [1] 0.5441463
```

```
summary(lm.fit)$adj.r.squared
```

```
## [1] 0.5432418
```

We have also seen that in the simple linear regression model (with only one predictor variable), R-squared is the square of the correlation coefficient between x and y. This means we have two ways to find correlation:

```
sqrt(summary(lm.fit)$r.squared)  # make sure to change the sign to fit the data!
```

```
## [1] 0.7376627
```

```
cor(Boston$medv ~ Boston$lstat)  # notation from the mosaic package
```

```
## [1] -0.7376627
```

```
cor(Boston$medv, Boston$lstat)  # notation from the base stats package
```

```
## [1] -0.7376627
```

10 Test statistics and p-values– set up, calculation, interpretation

To conduct a test for $H_0 : \beta_i = 0$ vs $H_a : \beta_i \neq 0$, we can find our t-statistic and compare it to the appropriate $t_{(n-p')}$ distribution. For instance, if we want $\alpha = 0.5$ level significance in the Boston data example here to test whether $\beta_1 = 0$, we would compare our test statistic to $t_{(1-\alpha/2,n-p')} = t_{(0.975,504)}$. We can find this value with R:

```
(t_crit <- qt(0.975, 504))
```

```
## [1] 1.964682
```

If the test statistic for our test is more extreme than t_crit, then we reject the null hypothesis. In our example, the test statistic is

```
(t_stat <- summary(lm.fit)$coefficients[2, 3])
```

```
## [1] -24.5279
```

```
abs(t_stat) > t_crit
```

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

So we reject our null hypothesis at the 0.05 level of signficance and conclude that $\beta_1 \neq 0$.

We can also perform a hypothesis test just using the p-value. A p-value less than 0.05 is generally considered strong evidence to reject the null hypothesis. A p-value less than 0.01 is considered very strong evidence against the null hypothesis. In our example:

```
(pval <- summary(lm.fit)$coefficients[2, 4])
```

```
## [1] 5.081103e-88
```

```
pval < 0.01
```

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

Our p-value is basically 0, so we reject H_0.

We can also retrieve the F-statistic to perform the F-test for $H_0 : \beta_1 = 0$ against the two-sided alternative. In this example,

```
f_stat <- summary(lm.fit)$fstatistic
```

We can compare this to the F(1, 504) distribution as we did with the t-statistic an t-distribution above. As before, we reject the null hypothesis.

11 ANOVA and MS(Res)

Finding the analysis of variance table just uses the `anova` command:

```
anova(lm.fit)
```

```
## Analysis of Variance Table
##
## Response: medv
##            Df Sum Sq Mean Sq F value    Pr(>F)
## lstat       1  23244 23243.9  601.62 < 2.2e-16 ***
## Residuals 504  19472    38.6
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We see here that the MS(Res) = 38.6. We can also find this in the summary of the linear model object:

```
summary(lm.fit)$sigma  # Gives us the estimate for s
```

```
## [1] 6.21576
```

```
summary(lm.fit)$sigma^2  # Gives us the estimate for s^2 = MS(Res)
```

```
## [1] 38.63568
```

And that's pretty much everything you need to know about simple linear regression in R. The only things we haven't covered are diagnostic plots and regression diagnostics in general, because we cover those later in the course.