Lab 2: The Linear Model in R and the Tidyverse

Sean Norton (adapted from code by Isabel Laterzo and Simon Hoellerbauer)

1/20/2022

This lab will introduce you to R's base `lm()` function, which is used to fit OLS models. In addition, we will briefly use some `tidyverse` functions, which can be useful for manipulating data and cleaning text in R. (I am personally a tidyverse skeptic, but draw your own conclusions.) We will also use `ggplot2` again to continue working on your plotting skills.

To ensure that everyone is attempting the labs, you will be turning this in for a completion grade on Sakai by 5 PM on Monday. I will not be checking your code for accuracy, but please review the answer key regardless. Please turn the notebook in as a PDF file.

Like last time, I'll give a brief introduction to this lab and then you will be free to work independently.

Don't forget to set your working directory!

#Part 1: LM in R

First, we're going to simulate some fake data. *This is a very useful skill, both for the problem sets and your future work!* By creating a data set where the parameters are known, you can check whether a model is capable of correctly recovering the parameters.

We'll first specify the β parameters for our fake data, create a fake input matrix X, and then simulate a random error term. Using these values, we'll simulate a vector of outcomes, Y.

Let's set our seed to 1017, representing the date I first set a random seed in R many years ago.

We'll use only a single predictor for simplicity's sake. Draw X_1 from a normal distribution with mean 5 and standard deviation 1 - get 100 draws. (NB: while in statistical texts the normal distribution is commonly parameterized using the variance, in R it's parameterized using the standard deviation. Make sure to check you're using the correct value if asked to draw from a specific normal distribution.) We'll set the standard deviation to be 3.4 (for simulating the error).

Let's set the intercept to be 5 and β_1 to be 1.7

Why do we need to simulate error for the purposes of fitting a model? What would happen if we didn't?

Now we need to simulate our outcome, which is a linear product of the inputs, the coefficients, and random error. Simulate the outcome y below.

Let's investigate the relationship between the predictor and our outcome. Create a scatter plot of y vs. X_1 below using `ggplot()`. You'll need to coerce the values to a dataframe in order to use `ggplot2`.

This should look like exactly what we intended it to be - a linear relationship, measured with noise.

Now let's see if R's `lm()` function actually works! (Hopefully it does, because otherwise we're going to have to rework the entire course.)

Use R's help docs to figure out how estimate a linear model on our dataframe that we created above. The only relevant arguments to `lm()` for our purposes are `formula` and `data`. Note that you can use arguments in R positionally - you can simply input the formula as the first argument, add a comma, and then input the data - but it's generally better practice to specify argument names to avoid mistakes and make your code more readable.

Make sure to assign the result of `lm()` to an object. You can then check the results using the `summary()` command.

How does that compare to the relationship we specified? Do the true values fall within the 95% confidence interval?

This is also a great example of why its more honest to report effect sizes as a range than a point estimate. Our point estimate of the true value of β_1 is off by roughly .3, or 17% higher than the true effect. That's not an inconsequential difference in magnitude!

#Part 2: Combining LM and the tidyverse

Some of you might already use the `tidyverse` and its associated packages (if you do not have it installed yet, I suggest doing `install.packages("tidyverse")`) The tidyverse allows us to use the piping operator `%>%`, which can be a useful piece of code. It allows you to take code that would previously involve many nested functions in break them up into separate function calls, with the pipe operator passing the result of each function to the next. This can make code more readable, and is similar to the pipe operator (`|`) in shell-scripting languages like bash.

Of course, the exact same thing could be accomplished by simply saving results to objects and breaking heavily-nested workflows into separate lines. The `%>%` is also a special type of function (a so-called "infix operator") and behind-the-hood, it is extremely complex. In my personal opinion, using the tidyverse pipe operator fixes a problem that does not exist and introduces new problems, notably that its error messages are obscure and it is not infrequently buggy. Nevertheless, it is commonly used, many people find it useful, and I'm here to help you learn how to code, not command you on how to do it! Workflows that only pipe a few functions together are unlikely to run into these problems.

(And I will admit that many of the data cleaning functions are useful because they replace multiple base R functions.)

```
library(tidyverse)
```

Let's show a basic example of a pipe. This example is pretty simplistic and doesn't gain you much, but you get the general idea. So long as you are absolutely clear that you know what value is being passed through `%>%` (in this case the mpg variable), the pipe operator can let you write some pretty, readable code. If you're ever not sure what value is being passed, either double-check each step of the process or just don't use the piping operator.

```
data(mtcars)

#normal
sd(mtcars$mpg)
```

```
## [1] 6.026948
```

```
#piping
mtcars$mpg %>% sd()
```

```
## [1] 6.026948
```

The `dplyr` package in the `tidyverse` is by far the most popular and useful (after `ggplot2`.) `.dplyr` allows us to manipulate our data in a structured and simple way, and particularly shines in its ability to apply transformations to specific subsets of data with far fewer function calls than in base R. The key functions are `mutate()`, which creates new variables, `select()`, which allows us to pick variable by characteristics of their names, `filter()`, which allows us to choose rows according to criteria we define, `summarize()`, which allows us to "reduce multiple values down to a single summary", and `group_by`, which groups the data frame by a defined criteria and then applies all subsequent function calls by group rather than to the dataframe as a whole.

Let's see an example of how this can be convenient. We'll group cars by the number of gears and then get the mean MPG for cars with the same number of gears.

There you have it - cars with 4 gears are the most efficient for a reason that probably makes sense if you understand how cars work.

RStudio makes useful cheat sheets for their packages, including `dplyr`.

As we dive into learning linear models in `R`, we're going to use some data from the `mtcars`. But first, lets use the `tidyverse` to clean it up a bit. First lets take a look at it (its readily available so you don't really need to load it in!).

Great. Let's say that we work at a car company and we've been ordered to analyze the role that `disp` and `cyl` play in influencing `mpg`. However, we're only interested in observations where `cyl` is greater than 4. Let's use `dplyr` and the `tidyverse` to only keep observations where `cyl` is greater than 4.

Hint: the `dplyr` functions you are interested in here are `filter()` and `select()` - don't forget to call their help files if you need more information (e.g., `?filter()` in your console).

Now that we have our data ready, let's model it. Use your recently learned `lm()` skills to regress `mpg` on `disp` and `cyl`. Use `summary()` to review the results.

Remember, use the filtered data frame, not `mtcars`.

The `summary` function provides lots of valuable information about the model we've created. We can also extract different information if we need, for example, just the coefficients, or just the residuals. Use `str()` on the model object. What basic R data type does the model object look like? Run `typeof()` on the model object to confirm your suspicion.

Using the correct subset notation for this type of object (`str` should give you a nice hint if you forgot), extract the coefficients, the residuals, and $\hat{y}$ (`fitted.values`). It is better to use predict than fitted.values (you will use predict in the problem set)

(`str()` is a very useful function for displaying the "shape" of any object in R - if you're trying to extract something from an object and can't find it, `str()` will help you poke around.)

What if we wanted to include an interaction term? Use `*` for the interaction term. You should always include both variables in the interaction term as independent regressors in the model. `lm()` will do this for you automatically if you forget, but not all the modeling functions we use this semester will bail you out!

How might we present these results in a table? Both `xtable()` and `stargazer()` are functions that produce LaTeX code that could be copy-pasted into a .tex document or included in an Rmarkdown document. Try both of these on one of the model object we just created above (with the interaction term). Note, these are VERY flexible and can be customized to make very nice tables, the ones we are generating here are just basic.

#Part 3: ggplot

`ggplot` is very important, but requires some time and patience to get to know well. We won't have time to dive fully into it, but we will begin reviewing the following code so you become more familiar. R has its own built-in plot function, but ggplot is far more flexible and creates far better plots.

For a comprehensive 'cheat sheet', see: https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf

Let's start by making a basic scatter plot of displacement (`disp`) versus miles per gallon (`mpg`). Let's make it more fun/useful by coloring the points by the number of cylinders in each car.

This should reveal some basic insights into the ggplot syntax:

- A plot always starts by initializing the blank plot with a call to `ggplot()`
- The basic plot object is defined using a dataframe (`data`) and an aesthetic (`aes`)
- `aes` includes the variables that will be used in the plot
- Layers are added to this basic plot object using `+` and a `geom_` function

- `geom_` functions take a variety of options and can even take their own aesthetics
- `theme_` functions change the basic theme of the plot to make it better looking; these are highly customizable, and you can even write your own to make sure all your plots look consistent.

Let's make this plot much more useful. Let's add labels to the axes and the scale of the plot. Fill in the code below with useful labels and change the size of the points in `geom_point()` to make them more visible.

You can also assign this plot to an object and save that object as a .pdf.

What if we wanted to plot the linear relationship between our x and y?

Whoa, you can add new layers to an existing plot without completely redoing the plot? That seems neat and useful!

Right now, we're using color as the aesthetic mapped to a specific explanatory variable (color represents the number of cylinders). We can play around with this:

Obviously, some plotting choices are going to be more useful than others. You should always endeavor to make your plots communicate the intended insight in the clearest way possible. The perfect plot doesn't require reviewers to read the accompanying part of the text to understand it (because they probably won't read the accompanying text). Also, while colors are fun, be aware that most journals will not print color plots in the print edition (for the 5 people who still read those) and some journals are no fun and do not allow color at all.

Finally, one reason we might be interested in models is for prediction! (Predictive checks using held-out data can also be a useful way to check model robustness.) Let's check our model's predictions versus the actual values. The `predict` function should come in handy here. (Note that we don't actually need to use it since the fitted values are part of the model object, but then you wouldn't get to use this function.)

How should we interpret this plot? What would a perfect fit look like?