STAT 213: R Intro Continued
Describing Data, Chaining Commands, Markdown, Regression

Colin Reimer Dawson

Last Revised February 11, 2016

1 More R, Illustrated Through a Means Model

Last time I walked through the process of modeling a data set consisting of presidential polls taken in 2008, some before and some after the Lehman Brothers firm declared bankruptcy in mid-September. We examined and compared two models of the poll data; one with no predictors, fitting a single mean, and one with a single binary predictor, representing whether the poll was taken before or after the "Meltdown". The population models are described by

$$Y = \mu + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma_\varepsilon) \tag{1}$$

$$Y = \begin{cases} \mu_{before} + \varepsilon, & \epsilon \sim \mathcal{N}(0, \sigma_\varepsilon^{(before)}) \quad X = \text{ before} \\ \mu_{after} + \varepsilon, & \epsilon \sim \mathcal{N}(0, \sigma_\varepsilon^{(after)}) \quad X = \text{ after} \end{cases} \tag{2}$$

Let's briefly walk through the process of fitting, assessing and using these models in R.

This dataset is part of the `Stat2Data` package. The name of the dataset is `Pollster08`.

Remember that, in each new session or script, we need to bring "into view" any packages that we are going to use. The way to do this is using the `library()` function, with the package name (in quotes) as the first and only argument value.

Then, calling the `data()` function with the name of the dataset as the argument brings the *dataset* into "scope". Do both of these now:

```
library("Stat2Data")
data("Pollster08")
```

Let's look at the code book for this dataset by typing its name preceded by a question mark (**Warning:** do this line at the console, not in a script — you do not want the code book to automatically pop up every time you run your code)

```
?Pollster08
```

We can see descriptions of many different variables collected in this dataset, but the two we are interested in are `McCain` (this is our response, Y, variable), and `Meltdown` (our predictor). Looking at the code book, we see that `McCain` consists of the percentage of respondents supporting John McCain, and `Meltdown` is an "indicator" for whether the poll was taken after the meltdown or not. In stats jargon, an **indicator variable** is equal to 1 when some condition is true, and 0 otherwise; so here, 0 means "before" and 1 means "after" the meltdown.

Do (also at the console)

```
head(Pollster08)
```

to see the first few rows (cases).

1.1 Model Formulas: Boxplots and Histograms

The `mosaic` package features a set of functions for visualizing and describing data with a consistent, model-based syntax. You should already have this package installed, but since this is a new session/script, you need to activate it with `library()`. Do that now.

The first thing you should do before fitting any model is to plot the data.

Let's look at side-by-side boxplots of the data. The `bwplot()` function produces box and whisker plots. In a minute, you will type

```
bwplot(McCain ~ factor(Meltdown), data = Pollster08)
```

to create a pair of box plots of the `McCain` variable, one for each value of the `Meltdown` variable. Before you do it, let's look at the command.

The plotting command involves the function `bwplot`, an argument named `data` with the value `Pollster08`, and something new, called a `formula`. This has the form `Y ~ X`, where `Y` and `X` are the name of a response and predictor variable, respectively. Here, the response variable `Meltdown` is converted into a **factor** variable: this tells R that it is really a categorical variable, even though the entries are numbers. This part is not necessary if the variable values are text strings instead of numbers.

Go ahead and enter the command. You will see two box plots, with solid black dots representing the medians in each group, and boxes stretching from the 25th to the 75th percentile of the data in that group. Outliers are drawn as open circles.

We could also do a histogram or pair of histograms, perhaps with a best-fit normal density overlaid.

First, let's see the combined data:

```
histogram(~McCain, data = Pollster08, fit = "normal")
```

Note that with no predictors, the formula is now in the form `~ Y`. We've added an extra argument, `fit`, with the value `"normal"` to include the superimposed normal density curve.

The `histogram()` function doesn't handle a predictor variable in exactly the same way as `bwplot()`, unfortunately. There are a couple of options, but let's see how to plot just one subset of the data.

1.2 `filter()` and Command Chaining

The `filter()` function allows us to "filter" the dataset to allow only certain cases through. Type

```
filter(Pollster08, Meltdown == 0)
```

This produces a dataset that contains only those polls before the meltdown. Note the double equals sign, which indicates a TRUE or FALSE condition, rather than an argument name/value pair. To pass this dataset to the `histogram` function we will do

```
filter(Pollster08, Meltdown == 0) %>%
    histogram(~McCain, data = ., fit = "normal", main = "Before", v = 50)
```

Note the %>% operator between the two commands. This is called the **pipe** operator, and allows us to pass the result of one command as the value of an argument to another, without having to place one whole command inside the other, or having to create a new intermediate variable for the restricted dataset. Note the . as the argument value. This indicates that we want the value of the argument to be the output of the previous command. We have also included a couple of new extra arguments, one to give the plot a title, and one to draw a vertical line at 50% (to make it easier to draw visual comparisons across two plots).

Type in the "chained" commands. You should see a slightly different histogram.

Exercise 1 Modify the above command to create an analogous histogram consisting of those polls after the Meltdown.

1.3 R Markdown for Reproducible Research

Last time we learned about script files as a way to keep track of what we have done. It is good practice to record exactly what you did so that you can reproduce it later. This extends to plots and graphs as well. At this point, we have collected several plots. If we wanted to present our models and descriptions of this data, we'd like to include up-to-date plots which (a) use the most current data we have, and (b) reflect exactly any manipulations we did in the code. A script only helps us so much here: at some point we would need to cut and paste graphs into a Word document or some such writeup, and we run the risk of getting things out of sync.

A powerful solution is to use a **Markdown** document, where code, plots, and descriptive text can be interleaved in a file, which gets "compiled" (or "Knit") into a well-formated document that includes all of the above (or, you can omit the code). Every time you make a change, just re-Knit, and the changes will be updated in the document.

R Markdown Workflow

1. From the RStudio File menu, select New File > R Markdown.... Now you have several options. From the list on the lefthand side, select From Template, and then on the right, pick mosaic fancy for now, to get a template that includes a bunch of example information. As you create your own documents you will want to delete much of this extra stuff, but reading this template is a good way to learn about how Markdown works.
2. The first thing you should do after creating a new Markdown document is to save the file. Do File > Save As... and give your document a name.
3. Before editing anything, press the Knit HTML button in the document toolbar. If on the server you may get a message about a blocked pop-up. Allow it to show, and you will get a new window with a mix of code, text, and plots.
4. The first thing you should edit is the Title and Author fields at the top of the document. Give the document a title such as "R Intro Day 2", and put your name inside the quotes on the Author line. Re-Knit to see your change reflected in the document.
5. In general, every time you make a change, it is a good idea to Knit again. That way, if you get an error message, you know what caused it. As you get more experienced you may find yourself going longer between re-Knits, but at first you should do it very frequently.

Look over the original source document. One thing that jumps out is that all of the R code in the document is inside triple quote markers. These are actually backquotes, which are found on the same key as the tilde (∼) on most keyboards.

To create a new code chunk, place the cursor where you want the code to go, and click the Chunks button at the upper right of the toolbar, and select Insert Chunk (you will also see a keyboard shortcut that you could use intead if you prefer). This will create the "fence" markers that tell the compiler that anything inside is code, and anything outside is text.

Protip Just like in a script file, you should *not* include code in your Markdown document that you would not want to be run over and over again. This includes

`install.package()` lines, lines that access documentation, or generate dialogue boxes.

Exercise 2

- Edit the very first code chunk to include `library()` lines for the packages we are using. The `include = FALSE` option in curly braces tells the compiler not to display the code or any output from that chunk in the final document. It should just quietly run it for its "side-effects". Re-Knit to make sure you don't get any errors.
- Now, find the first code chunk in the section called Graphics. Add the `data()` line that brings the `Pollster08` dataset into scope, and put the commands that create the side-by-side boxplots and the three histograms there as well. Re-Knit. You should see your plots embedded in the final document.

1.4 Fitting the Models

For our simple means model, estimating parameters is a simple matter of calculating sample means and sample standard deviations. The syntax for this is exactly analogous to the syntax for making box plots and histograms: specify a model formula, either in the $Y \sim X$ or $\sim Y$ form, where Y and X are response and predictor variable names in the dataset, and include a `data` argument.

To get an overall mean and overall standard deviation, type

```
mean(~McCain, data = Pollster08)
sd(~McCain, data = Pollster08)
```

This will print out the values. (We could also have saved the values for future use by assigning the output to a named variable.)

To get means and standard deviations by group, use the $Y \sim X$ form:

```
mean(McCain ~ Meltdown, data = Pollster08)
sd(McCain ~ Meltdown, data = Pollster08)
```

We can also get both at once, and more, with

```
favstats(McCain ~ Meltdown, data = Pollster08)
```

Exercise 3

- Replace the `favstats()` line in the Markdown template with the line above, and Knit.
- Just above that code chunk, insert a new code chunk, where you include a similar `favstats()` line, but for the no predictors model. Re-Knit.
- Write a sentence below each code chunk (not itself in a chunk) describing what the data tells you.

1.5 Assessing the Models

We've already looked at histograms and boxplots of the data with superimposed Normal curves. We will skip explicit residual plots for the moment, since we have not explicitly represented our model as a model yet. For now, proceeed to the testing step.

Exercise 4 Do the t-test to see if the means significantly differ. The form is the same: model formula, and data argument. The function is called `t.test()`. Include this in a new code chunk. Re-Knit. Write a sentence or two below the code chunk describing what the results show. Re-Knit again.

2 Simple Linear Model

Let's take the other example we did in class last time: modeling incumbent presidents' re-election margin as a linear function of approval rating.

The dataset is `ElectionMargin`, and it is in the `Lock5Data` package.

Exercise 5 Create a code chunk with the relevant `library()` and `data()` commands to bring this dataset into scope.

Before we do anything else we should plot the data. Do

```
xyplot(Margin ~ Approval, data = ElectionMargin)
```

Does the data look reasonably linear?

Let's fit the linear model. It is useful to save the model object to a named variable. Do

```
## The name margin.model is arbitrary
margin.model <- lm(Margin ~ Approval, data = ElectionMargin)
```

The function `lm()` stands for linear model, and produces a linear regression model object with lots of information in it. To see the coefficients, just view `margin.model`

```
margin.model
```

We can get a prediction function from our model as follows

```
## The name f.hat could be anything
f.hat <- makeFun(margin.model)
```

Now, we get get the predicted Y value for any X value we want:

```
f.hat(Approval = 48)
```

The number 1 just indicates that it is the first value asked for. We could in principle ask for predictions for a whole vector (list) of inputs.

We can plot the regression line using `plotFun()`:

```
plotFun(f.hat(Approval) ~ Approval)
```

We don't need a `data =` argument here, because `ElectionMargin` is associated with our prediction function already.

Exercise 6 We'd like to get the line and the data on the same plot. To do this, first plot the data as above (do this in a code chunk). Then, *in the same code chunk*, add the `plotFun()` command as above, but include an extra argument, `add = TRUE` to superimpose the line on the data. Be sure that the code you need to define the model and the prediction function are included somewhere in your Markdown document as well.

2.1 Residual Plots

We can get diagnostic residual plots: the fitted values against the residuals, and a Normal Quantile-Quantile plot by simply calling the `plot()` function on our linear model object and including the argument `which=` to pick which plot we want.

```
## which = 1: residuals against fitted
plot(margin.model, which = 1)
```

```
## which = 2: Quantile-Quantile plot
plot(margin.model, which = 2)
```

Exercise 7 Put each of these in a code chunk. Specify `echo = FALSE` in the code chunk options to suppress the code itself and only include the resulting plots. Write a sentence or two describing what the residual plots tell you.