R: Learning by Exapmle
Data Management and Analysis

Dave Armstrong
University of Western Ontario
Department of Political Science

e: dave.armstrong@uwo.ca
w: www.quantoid.net/teachicpsr/rbyexample

Contents

1 The Basics **4**
1.1 Getting R . . . 4
1.2 Using **R** . . . 6
1.3 Assigning Output to Objects . . . 6
1.4 Reading in your Data . . . 8
1.5 SPSS . . . 9
1.6 Function, Syntax and Arguments . . . 10
1.7 Stata . . . 12
1.8 Excel . . . 12
1.9 Data Types in **R** . . . 13
1.10 Examining Data . . . 14
1.11 Saving & Writing . . . 16
1.11.1 Where does R store things? . . . 16
1.12 Writing . . . 17
1.13 Saving . . . 17
1.14 Recoding and Adding New Variables . . . 17
1.15 Missing Data . . . 22
1.16 Filtering with Logical Expressions and Sorting . . . 23
1.17 Sorting . . . 24
1.18 Summarising by Groups . . . 25

2 Merging Datasets **29**

3 Statistics **30**
3.1 Cross-tabulations and Categorical Measures of Association . . . 30
3.1.1 Measures of Association . . . 34
3.2 Continuous-Categorical Measures of Association . . . 35
3.3 Linear Models . . . 36

3.3.1 Adjusting the base category . . . 37
3.3.2 Model Diagnostics . . . 38
3.3.3 Predict after lm . . . 44
3.3.4 Linear Hypothesis Tests . . . 48
3.3.5 Factors and Interactions . . . 49
3.3.6 Non-linearity: Transformations and Polynomials . . . 55
3.3.7 Testing Between Models . . . 59
3.4 GLMs and the Like . . . 63
3.4.1 Binary DV Models . . . 63
3.5 Ordinal DV Models . . . 68
3.6 Multinomial DV . . . 73
3.7 Survival Models . . . 78
3.8 Multilevel Models . . . 85
3.9 Factor Analysis and SEM . . . 95

4 Miscellaneous Statistical Stuff 103
4.1 Heteroskedasticity Robust Standard Errors . . . 103
4.2 Clustered Standard Errors . . . 104
4.3 Weighting . . . 105

5 Finding Packages on CRAN 109

6 Warnings and Errors 110

7 Troubleshooting 111

8 Help! 119
8.1 Books . . . 119
8.2 Web . . . 120

9 Brief Primer on Good Graphics 120
9.1 Graphical Perception . . . 121
9.2 Advice . . . 122

10 Graphics Philosophies 123

11 The Plot Function 124
11.1 getting familiar with the function . . . 124
11.2 Default Plotting Methods . . . 126
11.3 Controlling the Plotting Region . . . 129
11.4 Example of Building a Scatterplot . . . 129
11.4.1 Adding a Legend . . . 134
11.4.2 Adding a Regression Line . . . 136
11.4.3 Identifying Points in the Plot . . . 137
11.5 Other Plots . . . 138

12 ggplots **141**
12.1 Scatterplot . . . 142
12.1.1 Bar Graph . . . 147
12.2 Other Plots . . . 149
12.2.1 Histograms and Barplots . . . 149
12.2.2 Dotplot . . . 150
12.3 Faceting . . . 153
12.4 Bringing Lots of Elements Together . . . 156

13 Maps **158**

14 Reproducibility and Tables from R to Other Software **166**

15 Reproducible Research **170**

16 Web Sites to Data **171**
16.1 Importing HTML Tables . . . 171
16.2 Scraping Websites for Content . . . 173
16.2.1 Text (Pre-)Processing . . . 174
16.3 Loops . . . 175
16.3.1 Example: Permutation Test of Significance for Cramer's V. . . . 177
16.4 Loops Example: Web Spidering . . . 178
16.5 If-then Statements . . . 179

17 Repeated Calculations **180**
17.1 apply and its relatives . . . 180
17.1.1 by . . . 181
17.1.2 List Apply Functions . . . 184

18 Basic Function Writing **184**
18.1 Example: Calculating a Mean . . . 185
18.2 Changing Existing Function Defaults . . . 185
18.3 .First and .Last functions in R. . . . 187

Introduction

Rather than slides, I have decided to distribute handouts that have more prose in them than slides would permit. The idea is to provide something that will serve as a slightly more comprehensive reference, than would slides, when you return home. If you're reading this, you want to learn R, either of your own accord or under duress. Here are some of the reasons that I use R:

- It's open source (that means FREE!)
- Rapid development in statistical routines/capabilities.
- Great graphs (including interactive and 3D displays) without (as much) hassle.

- Multiple datasets open at once (I know, SAS users will wonder why this is such a big deal).
- Save entire workspace, including multiple datasets, all models, etc...
- Easily programmable/customizable; easily see the contents (guts) of any function.
- Easy integration with LaTeX and Markdown.

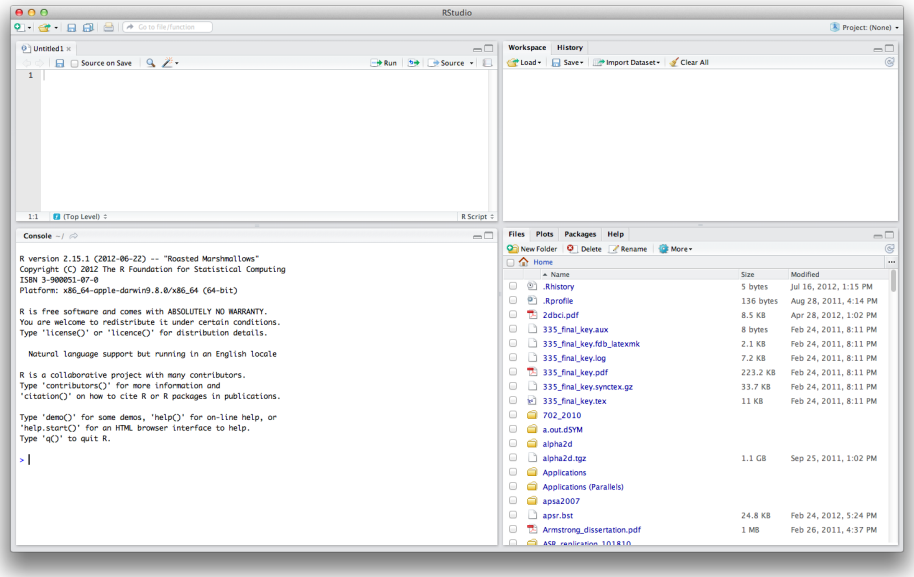
1 The Basics

1.1 Getting R

R is an object-oriented statistical programming environment. It remains largely command-line driven.[1] **R** is open-source (i.e., free) and downloadable from http://www.cran.r-project.org. Click the link for your operating system. In Windows, click on the link for `base` and then the link for "Download R 3.6.0 for Windows". Once it is done, double-click on the resulting file and that will guide you through the installation process. There are some decisions to be made, but if you're unsure, following the defaults is generally not a bad idea. In Windows, you have to choose between MDI mode (Multiple Document Interface) where graphs and help files open in their own windows or SDI mode where graphs and help files open as sub-windows in the **R** window. For Mac users, click on the link for "Download R for Mac" on the CRAN home page and then click the "R-3.6.0.pkg" link (to get the latest version, you'll need $\geq$ El Capitan). For older versions of the OS, between Maverics and El Capitan, you can download "R-3.3.3.pkg" from the same page.

You may also want to download RStudio https://www.rstudio.com/products/rstudio/download/#download, an Integrated Development Environment (IDE) for R. This application sits on top of your existing R installation (i.e., it also requires you to install R separately) to provide some nice text editing functions along with some other nice features. This is one of the better free R-editing environment and one that is worth checking out. The interface looks like this:

[1] There are a couple of attempts at generating point-and-click GUIs for **R**, but these are almost necessarily limited in scope and tend to be geared toward undergraduate research methods students. Some examples are RCommander, Deducer and SciViews.



Some people have had trouble with R Studio, especially when it is installed on a server, though sometimes on their own machines, too. If you're on a Mac and Rstudio starts to feel "laggy", you can solve the problem by opening the terminal and typing the following:

```
RSTUDIO_NO_ACCESSIBILITY=1 /Applications/RStudio.app/Contents/MacOS/RStudio
```

This will open a new RStudio window that will hopefully work better. Alternatives are to use R's built-in editor which you can get by typing "ctrl + n" on Windows or "command + n" on the mac when you're in an active R session, or to use another IDE, like Atom, Sublime or Microsoft VS Code (which I use now). Note, RStudio is like a viewer for R. It is R, just with some added convenience features.

Like Stata and SAS, **R** has an active user-developer community. This is attractive as the types of models and situations **R** can deal with is always expanding. Unlike Stata, in **R**, you have to load the packages you need before you're able to access the commands within those packages. All openly distributed packages are available from the Comprehensive **R** Archive Network, though some of them come with the Base version of R. To see what packages you have available, type `library()` There are two related functions that you will need to obtain new packages for R. Alternatively, in RStudio, you can click on the "Packages" tab (which should be in the same pane with files, plots, help and viewer). Here is a brief discussion of installing and using R packages. First, a bit of terminology. Packages are to R's library as books are to your university's library. R's library comprises packages. So, to refer to the `MASS` *library* would technically be incorrect, you should call in the `MASS` *package*.

- `install.packages()` will download the relevant source code from R and install it on your machine. This step only has to be done once until you upgrade to a new minor (on Windows) or major (on all OSs) version of R. For example, if you upgrade from 3.5.0 to 3.5.1, all of the packages you downloaded will still be available on macOS, but you will have to download them anew in Windows. In this step, a dialog box will ask you to choose a CRAN mirror - this is one of many sites

that maintain complete archives of all of R's user-developed packages. Usually, the advice is to pick one close to you (or the cloud option).

- `library()` will make the commands in the packages you downloaded available to you in the current R session (a new session starts each time R is started and continues until that instance of R is terminated). As suggested this has to be done (when you want to use functions other than those loaded automatically) each time you start R. There is an option to have R load packages automatically on startup by modifying the `.RProfile` file (more on that later).

1.2 Using R

The "object-oriented" nature of R means that you're generally saving the results of commands into objects that you can access whenever you want and manipulate with other commands. **R** is a case-sensitive environment, so be careful how you name and access objects in the space and be careful how you call functions `lm()` $\neq$ `LM()`.

There are a few tips that don't really belong anywhere, but are nonetheless important, so I'll just mention them here and you can refer back when they become relevant.

- In RStudio, if you position your cursor in a line you want to execute (or block text you want to execute), then hit `ctrl+enter` on a PC or `command+enter` on the mac, the functions will be automatically executed.
- You can return to the command you previously entered in the R console by hitting the "up arrow" (similar to "Page Up" in Stata).
- You can find out what directory **R** is in by typing `getwd()`. In RStudio, this is visible in gray right underneath the `Console` tab label.
- You can set the working directory of **R** by typing `setwd(path)` where `path` is the full path to the directory you want to use. The directories must be separated by forward slashes `/` and the entire string must be in quotes (either double or single). For example: `setwd("C:/users/armstrod/desktop")`. You can also do this in RStudio with

  Session→Set Working Directory→Choose one of Three Options

- To see the values in any object, just type that object's name into the command window and hit enter (or look in the object browser in RStudio). You can also type `browseEnv()`, which will initiate a page that identifies the elements (and some of their properties) in your workspace.

1.3 Assigning Output to Objects

R can be used as a big calculator. By typing `2+2` into **R**, you will get the following output:

```
2+2

## [1] 4
```

After my input of `2+2`, **R** has provided the output of 4, the evaluation of that mathematical expression. **R** just prints this output to the console. Doing it this way, the output is not saved *per se*. Notice, that unlike Stata, you do not have to ask **R** to "display" anything in Stata, you would have to type `display 2+2` to get the same result.

Often times, we want to save the output so we can look at it later. The assignment character in **R** is `<-` (the less-than sign directly followed by the minus sign). You may hear me say "X gets 10," in **R**, this would translate to

```
X <- 10
X

## [1] 10
```

You can also use the `=` as the assignment character. When I started using R, people wren't doing this, so I haven't changed over yet, but the following is an equivalent way of specifying the above statement:

```
X = 10
X

## [1] 10
```

As with any convention that doesn't matter much, there are dogmatic adherents on either side of the debate. Some argue that the code is easier to read using the arrow. Others argue that using a single keystroke to produce the assignment character is more efficient. In truth, both are probably right. Your choice is really a matter of taste.

To assign the output of an evaluated function to an object, just put the object on the left-hand side of the arrow and the function on the right-hand side.

```
X <- 4+4
```

Now the object `X` contains the evaluation of the expression `4+4` or 8. We see the contents of `X` simply by typing its name at the command prompt and hitting enter. In the above command, we're assigning the output (or result) of the command `4+4` to `X`.

```
X

## [1] 8
```

1.4 Reading in your Data

Before we move on to more complicated operations and more intricacies of dealing with data, the one thing everyone wants to know is - "How do I get my data into **R**?" As it turns out, the answer is - "quite easily." While there are many packages that read in data, the `rio` package is a comprehensive solution to importing data of all kinds. To install the `rio` package, then, once that is done, you can do the following in R:

```
library(rio)
```

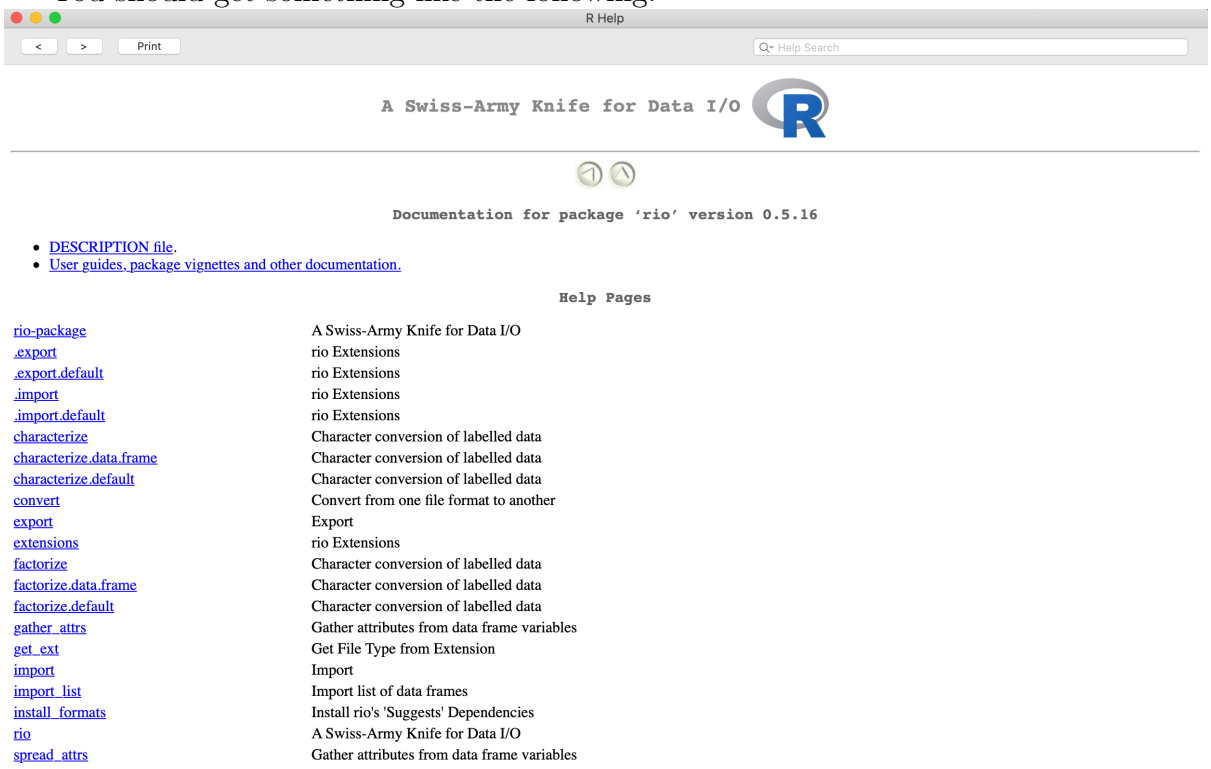
The first time you do this, you'll likely have to download some other importing formats, which you can do as follows in R:

```
install_formats()
```

Generally, you will get prompted to do this. By looking at the help page for the package, we can see what functions are available. In R, you can do this with

```
help(package="rio")
```

You should get something like the following:



The `import()` function is the one that imports your data from various sources. Looking at the help file for the `import` function will show you the types of data that can be imported. We'll show a couple of examples of how it works. [2]

[2] There are also `export` and `convert` functions that will write data out to lots of formats and convert from one to another.

The dataset we'll be using here has three variables - `x1`, (a numeric variable), `x2` (a labeled numeric variable [0=none, 1=some]) and `x3` a string variable ("no" and "yes"). I've called this dataset `r_example.sav` (SPSS) and `r_example.dta` (Stata).

R has lots of different data structures available (e.g., arrays, lists, ect...). The one that we are going to be concerned with right now is the *data frame*; the **R** terminology for a dataset. A data frame can have different types of variables in it (i.e., character and numeric). It is rectangular (i.e., all rows have the same number of columns and all columns have the same number of rows. There are some more distinctions that make the data frame special, but we'll talk about those later.

1.5 SPSS

Let's start with the SPSS dataset:

```
spss.dat <- import("r_example.sav")
```

The first argument (and only one we'll use) is the name of the dataset. If the dataset is in **R**'s current working directory, then only the file name is needed. If the files is not in **R**'s working directory, then we have to put the full path to the dataset. Either way, the full path or the file name, both have to be in either double or single quotes. We can put the output of this in an object called `spss.dat`. This name is arbitrary.

To see what the data frame looks like, you simply type the name of the object at the command prompt and hit enter:

```
spss.dat
```

```
##    x1 x2  x3
## 1   1  0 yes
## 2   2  0  no
## 3   3  1  no
## 4   4  0 yes
## 5   3  0  no
## 6   4  0 yes
## 7   1  1 yes
## 8   2  1 yes
## 9   5  1  no
## 10  6  0  no
```

If we wanted to look at a single variable from the data frame, we could use the dollar sign `$`, to extract a single variable from a data frame:

```
spss.dat$x1
```

```
##  [1] 1 2 3 4 3 4 1 2 5 6
## attr(,"label")
## [1] "x1"
## attr(,"format.spss")
## [1] "F8.2"
```

1.6 Function, Syntax and Arguments

Even though R is developed by many people, the syntax across commands is quite unified, or probably as much as it can be. Each function in R has a number of acceptable arguments - parameters you can specify that govern and modify the behavior of the function. In R, arguments are specified as first by supplying the name of the argument you are specifying and then by specifying the value(s) you want to apply to that argument. Let's take a pretty easy example first, `mean`. There are two ways to figure out what arguments are available for the function `mean()`. One is to look at its help file, by typing `?mean` or `help(mean)`.

You can see that the `mean()` function takes at least three arguments - `x`, the vector of values for which you want the mean calculated, `trim` - the proportion of data trimmed from each end if you want a trimmed mean. You can see in the help file that the default value for `trim` is 0. Finally, you can specify what you want to be done with missing data with `na.rm` the missing data can either be listwise deleted (if the argument is `TRUE`) or not (if the argument is "FALSE", the default). Arguments can either be specified explicitly by their names or, so long as they are specified in order, they can be given without their name. The arguments should be separated by commas. For example:

```
mean(spss.dat$x1)

## [1] 3.1

mean(x=spss.dat$x1)

## [1] 3.1

mean(na.rm=TRUE, x=spss.dat$x1)

## [1] 3.1
```

You will notice that we specified two different types of arguments above.

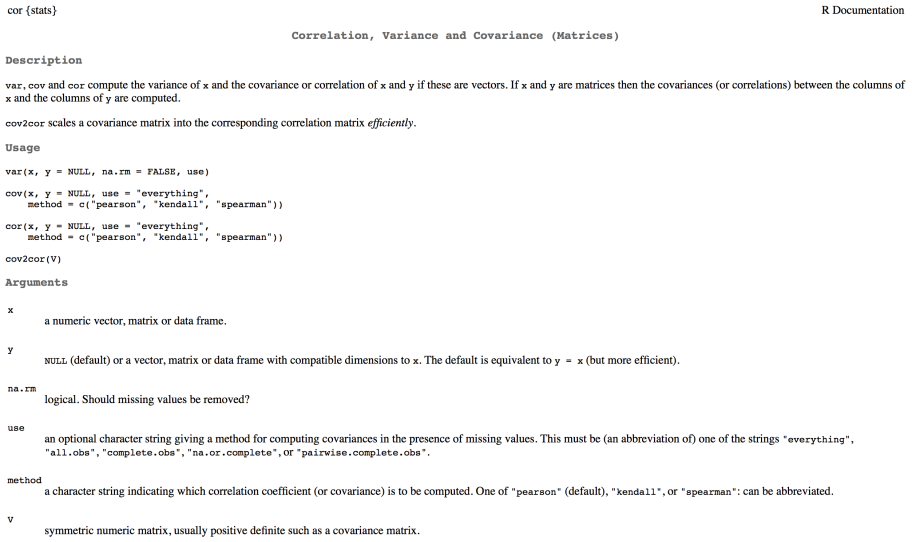
- The `x` argument wanted a vector of values and we provided a variable from our dataset. Whenever the argument is something that R recognizes as an object or is a function that R can interpret, then quotes are not needed.
- The `na.rm` argument is called a logical argument because it can be either `TRUE` (remove missing data) or `FALSE` (do not remove missing data). Note that logical arguments do not get put in quotation marks because R understands what `TRUE` and `FALSE` mean. In most cases, these can be abbreviated with `T` and `F` unless you have redefined those letters.

```
mean(spss.dat$x1, na.rm=T)

## [1] 3.1
```

Note that assigning `T` or `F` to be something is not necessarily great form as people might sometimes be accustomed to using these as shortcuts to logical values. This could result in the following sort of problem:

```
T <- "something"
> mean(spss.dat$x1, na.rm=T)
Error in if (na.rm) x <- x[!is.na(x)] :
  argument is not interpretable as logical
```

Arguments can also be character strings (i.e., words that are in quotations, either single or double). Let's consider the correlation function, `cor()`. This function again wants and `x` and `y` to correlate (though there are other ways of specifying it, too), as well as character string arguments for `use` and `method`. If you look just at the help file, you will see the following:



cor {stats} R Documentation

Correlation, Variance and Covariance (Matrices)

Description

`var`, `cov` and `cor` compute the variance of `x` and the covariance or correlation of `x` and `y` if these are vectors. If `x` and `y` are matrices then the covariances (or correlations) between the columns of `x` and the columns of `y` are computed.

`cov2cor` scales a covariance matrix into the corresponding correlation matrix *efficiently*.

Usage

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cov2cor(V)
```

Arguments

`x`
: a numeric vector, matrix or data frame.

`y`
: `NULL` (default) or a vector, matrix or data frame with compatible dimensions to `x`. The default is equivalent to `y = x` (but more efficient).

`na.rm`
: logical. Should missing values be removed?

`use`
: an optional character string giving a method for computing covariances in the presence of missing values. This must be (an abbreviation of) one of the strings `"everything"`, `"all.obs"`, `"complete.obs"`, `"na.or.complete"`, or `"pairwise.complete.obs"`.

`method`
: a character string indicating which correlation coefficient (or covariance) is to be computed. One of `"pearson"` (default), `"kendall"`, or `"spearman"`: can be abbreviated.

`V`
: symmetric numeric matrix, usually positive definite such as a covariance matrix.

what you will see is that both `use` and `method` have default values. For `use` the default value is `'everything'` and the default for `method` is `'pearson'`. The help file gives more information (particularly in the "Details" section) about what all of the various options mean.

```
cor(spss.dat$x1, spss.dat$x2, use="complete.obs",
    method="spearman")

## [1] -0.1798608
```

There are other types of arguments as well, but one of the most common is a formula. This generally represents situations where one variable can be considered a dependent variable and the other(s) independent variable(s). For example, if we wanted to run a linear model of $x1$ on $x2$ from the data above, we would do:

```
lm(x1 ~ x2, data=spss.dat)

##
## Call:
## lm(formula = x1 ~ x2, data = spss.dat)
##
## Coefficients:
## (Intercept)           x2
##      3.3333      -0.5833
```

where the formula is specified as `y ~ x`. The dependent variable is on the left-hand side of the formula and the independent variable(s) are on the right-hand side of the formula. If we had more than one independent variable, we could separate the independent variables with a plus (`+`) if we wanted the relationship to be additive (e.g., `y ~ x1 + x2`) and an asterisk (`*`) if we wanted the relation ship to be multiplicative (e.g., `y ~ x1 * x2`).

1.7 Stata

The basic operations here are pretty similar when reading in Stata datasets. The only difference is there is a different command - `read_dta`. You can see what the optional arguments are for the function by typing `help(read_dta)`. There are a couple of differences here. There is an `encoding` argument (only potentially needed for files before v. 14 that were encoded in something other than UTF-8).

```
stata.dat <- import("r_example.dta")
stata.dat

##    x1 x2  x3
## 1   1  0 yes
## 2   2  0  no
## 3   3  1  no
## 4   4  0 yes
## 5   3  0  no
## 6   4  0 yes
## 7   1  1 yes
## 8   2  1 yes
## 9   5  1  no
## 10  6  0  no
```

1.8 Excel

There are a couple of different ways to get information in to R from excel - either directly from whe workbook or from a `.csv` file from one of the sheets.

```
csv.dat <- import("r_example.csv")
csv.dat

##    x1   x2  x3
## 1   1 none yes
## 2   2 none  no
## 3   3 some  no
## 4   4 none yes
## 5   3 none  no
## 6   4 none yes
## 7   1 some yes
## 8   2 some yes
## 9   5 some  no
## 10  6 none  no
```

With an excel workbook, you'll also need to provide the `which` argument, which should be the sheet name in quotes or number (not in quotes).

```
xls.dat <- import("r_example.xlsx", which="Sheet1")
xls.dat <- import("r_example.xlsx", which=1)
xls.dat

##    x1   x2  x3
## 1   1 none yes
## 2   2 none  no
## 3   3 some  no
## 4   4 none yes
## 5   3 none  no
## 6   4 none yes
## 7   1 some yes
## 8   2 some yes
## 9   5 some  no
## 10  6 none  no
```

1.9 Data Types in R

This is a convenient time to talk about different types of data in **R**. There are basically three different types of variables - numeric variables, factors and character strings.

- Numeric variables would be something like GDP/capita, age or income (in $). Generally, these variables do not contain labels because they have many unique values. Dummy variables are also numeric with values 0 and 1. **R** will only do mathematical operations on numeric variables (e.g., mean, variance, etc...).
- Factors are variables like social class or party for which you voted. When you think about how to include variables in a model, factors are variables that you

would include by making a set of category dummy variables. Factors in **R** look like numeric variables with value labels in either Stata or SPSS. That is to say that there is a numbering scheme where each unique label value gets a unique number (all non-labeled values are coded as missing). Unlike in those other programs, **R** will not let you perform mathematical operations on factors.

- Character strings are simply text. There is no numbering scheme with corresponding labels, the value in each cell is simply that cell's text, not a number with a corresponding label like in a factor.

1.10 Examining Data

There are a few different methods for examining the properties of your data. The first will tell you what type of data are in your data frame and gives a sense of what some representative values are.

```
str(stata.dat)
```

```
## 'data.frame': 10 obs. of  3 variables:
##  $ x1: num  1 2 3 4 3 4 1 2 5 6
##   ..- attr(*, "label")= chr "First variable"
##   ..- attr(*, "format.stata")= chr "%8.0g"
##  $ x2: num  0 0 1 0 0 0 1 1 1 0
##   ..- attr(*, "label")= chr "Second variable"
##   ..- attr(*, "format.stata")= chr "%8.0g"
##   ..- attr(*, "labels")= Named num  0 1
##   .. ..- attr(*, "names")= chr  "none" "some"
##  $ x3: chr  "yes" "no" "no" "yes" ...
##   ..- attr(*, "label")= chr "Third variable"
##   ..- attr(*, "format.stata")= chr "%3s"
```

The second method is a numerical summary. This gives a five number summary + mean for quantitative variables, a frequency distribution for factors and minimal information for character vectors.

```
summary(stata.dat)
```

```
##        x1              x2           x3
##  Min.   :1.0   Min.   :0.0   Length:10
##  1st Qu.:2.0   1st Qu.:0.0   Class :character
##  Median :3.0   Median :0.0   Mode  :character
##  Mean   :3.1   Mean   :0.4
##  3rd Qu.:4.0   3rd Qu.:1.0
##  Max.   :6.0   Max.   :1.0
```

You could also use the `describe` function from the `Hmisc` package (which you'll have to install before you load it the first time):

```
library(psych)
describe(stata.dat)

##     vars  n mean   sd median trimmed  mad min  max range skew kurtosis
## x1     1 10  3.1 1.66      3    3.00 1.48   1    6     5 0.25    -1.34
## x2     2 10  0.4 0.52      0    0.38 0.00   0    1     1 0.35    -2.05
## x3*    3 10  NaN   NA     NA     NaN   NA Inf -Inf  -Inf   NA       NA
##       se
## x1  0.53
## x2  0.16
## x3*   NA
```

You can also describe data by groups, with the `describeBy()` function in the `psych` package:

```
describeBy(stata.dat, group="x2")

##
##  Descriptive statistics by group
## group: 0
##     vars n mean   sd median trimmed  mad min  max range skew kurtosis   se
## x1     1 6 3.33 1.75    3.5    3.33 1.48   1    6     5 0.14    -1.52 0.71
## x2     2 6 0.00 0.00    0.0    0.00 0.00   0    0     0  NaN      NaN 0.00
## x3*    3 6  NaN   NA     NA     NaN   NA Inf -Inf  -Inf   NA       NA   NA
## ------------------------------------------------------------
## group: 1
##     vars n mean   sd median trimmed  mad min  max range skew kurtosis   se
## x1     1 4 2.75 1.71    2.5    2.75 1.48   1    5     4 0.28    -1.96 0.85
## x2     2 4 1.00 0.00    1.0    1.00 0.00   1    1     0  NaN      NaN 0.00
## x3*    3 4  NaN   NA     NA     NaN   NA Inf -Inf  -Inf   NA       NA   NA
```

In the dataset returned by the `import` function, often factors will be represented by a `labelled` class variable that is numeric, but contains information on the labelling of the numbers. Unless you generally want those variables treated numerically, you may want to chance those into factors, which you can do with the `factorize()` function in the `rio` package.

```
stata.fdat <- factorize(stata.dat)
```

Note, you could overwrite the existing data if you like by putting `stata.dat` on the left-hand side of the assignment arrow.

Note that, `none` is the reference category. In R, it is always the first level that is the reference level and unless an alternative is specified, this is the first level alphabetically. This is largely irrelevant (at least from a statistical point of view), but can be changed with the `relevel` function:

```
levels(stata.fdat$x2)

## [1] "none" "some"

stata.fdat$x2 <- relevel(stata.fdat$x2, ref="some")
```

1.11 Saving & Writing

1.11.1 Where does R store things?

- Files you ask R to save are stored in R's working directory. By default, this is your home directory (on the mac mine is `/Users/armstrod` and on Windows it is `C:\Users\armstrod\documents`).
- If you invoke R from a different directory, that will be the default working directory.
- You can find out what R's working directory is with:

```
getwd()

## [1] "/Users/david/Dropbox (DaveArmstrong)/IntroR/Boulder"
```

- You can change the working directory with:
  - RStudio: Session → Chose Working Directory
  - Mac:

```
setwd("/Users/armstrod/Dropbox/IntroR")
```

  - Windows:

```
setwd("C:/Users/armstrod/Dropbox/IntroR")
```

  Note the forward slashes even in the Windows path. You could also do `C:\\users\\armstrod\\Dropbox\\IntroR`. For those of you who would prefer to browse to a directory, you could do that with

  - Mac:

```
library(tcltk)
setwd(tk_choose.dir())
```

  - Windows:

```
setwd(choose.dir())
```

There are a number of different ways to save data from R. You can either write it out to its own file readable by other software (e.g., .dta, .csv, .dbf), you can save a single dataset as an R dataset or you can save the entire workspace (i.e., all the objects) so everything is available to you when you load the workspace again (.RData or .rda).

1.12 Writing

You can write data out with the `export()` function in the `rio` package. You can write out to any of the following formats - .csv, .xlsx, .json, .rda, .sas7bdta, .sav, .dta, text. The function will pick the appropriate type based on the extension of the file you're exporting to. You can also specify it directly with the `format` function.

```
export(stata.dat, file="stata_out.dta")
```

1.13 Saving

- You can save the entire R workspace with `save.image()` where the only argument needed is a filename (e.g., `save.image('myWorkspace.RData')`). This will allow you to load all objects in your workspace whenever you want. You can do this with `load('myWorkspace.RData')`.
- You can save a single object or a small set of objects with `save()` e.g., `save(spss.dat, stata.dat, file='myStuff.rda')` would save just those two data frames in a file called `myStuff.rda` which you could also get back into R with `load()`.

You try it

1. Read in the data file `mtcars.dta` that was in your zip file and save it to an object.
   - Print the contents of the data frame.
   - Use some of the summarizing functions to learn about the properties of the data.
   - Save the data file as an R data set.
2. Read in your own dataset and learn about some of its properties.

1.14 Recoding and Adding New Variables

To demonstrate a couple of the features of **R**, we will add a variable to the dataset. Let's add a dummy variable that has zero for the first five cases and one for the last five cases. Unlike SPSS and Stata, there's not a particularly good spreadsheet-type data editor in **R**. For us, it is easier to make an object that looks the way we want, and then append that object to the dataset. If this is the strategy we adopt, first we need to make the object. What we want is a string of numbers (five zeros and five ones). To do this, we need to use **R**'s concatenate function, `c()`. I'll show this to you, then we'll discuss.

```
x4 <- c(0,0,0,0,0,1,1,1,1,1)
x4

##  [1] 0 0 0 0 0 1 1 1 1 1
```

What this did is make *one* object, called `x4` that is a string of numbers as above. Specifically, this is a vector with a length of ten (that is, it has ten entries). Now, we need to assign a new variable in the dataset the values of `x4`. We can do this as follows:

```
stata.dat <- import("r_example.dta")
stata.dat <- factorize(stata.dat)
stata.dat$x4 <- x4
stata.dat

##    x1   x2  x3 x4
## 1   1 none yes  0
## 2   2 none  no  0
## 3   3 some  no  0
## 4   4 none yes  0
## 5   3 none  no  0
## 6   4 none yes  1
## 7   1 some yes  1
## 8   2 some yes  1
## 9   5 some  no  1
## 10  6 none  no  1
```

Recoding and making new variables that are functions of existing variables are two relatively common operations as well. These are relatively easily done in **R**, though perhaps not as easily as in Stata and SPSS. First, generating new variables. As we saw above, we can generate a new variable simply by giving the new variable object in the dataset some values. We can also do this when creating transformations of existing variables. For example:

```
stata.dat$log_x1 <- log(stata.dat$x1)
stata.dat

##    x1   x2  x3 x4    log_x1
## 1   1 none yes  0 0.0000000
## 2   2 none  no  0 0.6931472
## 3   3 some  no  0 1.0986123
## 4   4 none yes  0 1.3862944
## 5   3 none  no  0 1.0986123
## 6   4 none yes  1 1.3862944
## 7   1 some yes  1 0.0000000
## 8   2 some yes  1 0.6931472
## 9   5 some  no  1 1.6094379
## 10  6 none  no  1 1.7917595
```

In the first command above, I generated the new variable (`log_x1`) as the log of the variable `x1`. Now, both of variables exist in the dataset `stata.dat`.

Recoding variables is a bit more cumbersome. There are commands in the `car` library (written by John Fox) that make these operations more user-friendly. To make those commands accessible, we first have to load the library with: `library(car)`. Then, we can see what the command structure looks like by looking at `help(recode)`. Let's now say that we want to make a new variable were values of one and 2 on `x1` are coded as 1 and values 3-6 are coded 2. We could do this with the recode command as follows:

```
recode(stata.dat$x1, "c(1,2)=1; c(3,4,5,6)=2")

##  [1] 1 1 2 2 2 2 1 1 2 2
## attr(,"label")
## [1] "First variable"
## attr(,"format.stata")
## [1] "%8.0g"
```

Here, the recodes amount to a vector of values and then the new value that is to be assigned to each of the existing values. The old/new combinations are each separated by a semi-colon and the entire recoding statement is put in double-quotes. Since I have not assigned the recode to an object, it simply prints the recode on the screen. It gives me a chance to, "try before I buy". If I'm happy with the output, I can now assign that recode to a new object.

```
stata.dat$recoded_x1 <- recode(stata.dat$x1,
        "c(1,2)=1; c(3,4,5,6)=2")
stata.dat

##    x1   x2  x3 x4    log_x1 recoded_x1
## 1   1 none yes  0 0.0000000          1
## 2   2 none  no  0 0.6931472          1
## 3   3 some  no  0 1.0986123          2
## 4   4 none yes  0 1.3862944          2
## 5   3 none  no  0 1.0986123          2
## 6   4 none yes  1 1.3862944          2
## 7   1 some yes  1 0.0000000          1
## 8   2 some yes  1 0.6931472          1
## 9   5 some  no  1 1.6094379          2
## 10  6 none  no  1 1.7917595          2
```

You can also recode entire ranges of values as well. Let's imagine that we want to recode `log_x1` such that anything greater than zero and less than 1.5 is a 1 and that anything greater than or equal to 1.5 is a 2. We could do that as follows:

```
recode(stata.dat$log_x1, "0=0; 0:1.5=1; 1.5:hi = 2")
```

```
##  [1] 0 1 1 1 1 1 0 1 2 2
## attr(,"label")
## [1] "First variable"
## attr(,"format.stata")
## [1] "%8.0g"
```

```
cbind(stata.dat$log_x1, recode(stata.dat$log_x1,
        "0=0; 0:1.5=1; 1.5:hi = 2"))
```

```
##             [,1] [,2]
##  [1,] 0.0000000    0
##  [2,] 0.6931472    1
##  [3,] 1.0986123    1
##  [4,] 1.3862944    1
##  [5,] 1.0986123    1
##  [6,] 1.3862944    1
##  [7,] 0.0000000    0
##  [8,] 0.6931472    1
##  [9,] 1.6094379    2
## [10,] 1.7917595    2
```

There are some other functions that can help change the nature of your data, too. One particularly useful one is `binVariable` from the `RcmdrMisc` package. There are a couple of main arguments (aside from the data). The `bins` argument specifies how many bins (number of groups + 1) you want and the `method` argument tells R whether you want groups with roughly equal intervals (`intervals`, the default) or groups with roughly equal counts (`proportions`). There is an optional argument `labels` that will give the labels to attach to each of the categories that is created.

```
library(RcmdrMisc)
data(Duncan)
incgroup <- binVariable(Duncan$income, bins=4, method="intervals")
table(incgroup)
```

```
## incgroup
##  1  2  3  4
## 16  9  8 12
```

```
incgroup2 <- binVariable(Duncan$income, bins=4, method="proportions")
table(incgroup2)
```

```
## incgroup2
##  1  2  3  4
## 15  9 11 10
```

You try it

Read in the `nes1996.dta` file, do the following:

1. Examine the data, both the properties of the data frame and the numerical summary.

2. Recode the `lrself` variable (left-right self-placement) such that the values 0 to 3 (inclusive) are "left", 4 to 6 (inclusive) are "center" and 7 to 10 (inclusive) are "right".

3. Recode the `race` variable into a dummy indicating whether observations are white or non-white.

4. Create an age-group variable from the variable `age` with five roughly evenly-sized groups.

1.15 Missing Data

In **R**, missing data are indicated with `NA` (similar to the `.`, or `.a`, `.b`, etc..., in Stata). The dataset `r_example_miss.dta`, looks like this in Stata:

```
. list

     +-----------------+
     | x1     x2    x3 |
     |-----------------|
  1. |  1   none   yes |
  2. |  2   none    no |
  3. |  .   some    no |
  4. |  4      .   yes |
  5. |  3   none    no |
     |-----------------|
  6. |  4   none   yes |
  7. |  1   some   yes |
  8. |  2   some   yes |
  9. |  5   some    no |
 10. |  6   none    no |
     +-----------------+
```

Notice that it looks like values are missing on all three variables. Let's read the data into **R** and see what happens.

```
stata2.dat <- import("r_example_miss.dta")
stata2.dat <- factorize(stata2.dat)
stata2.dat
```

```
##    x1   x2  x3
## 1   1 none yes
## 2   2 none  no
## 3  NA some  no
## 4   4 <NA> yes
## 5   3 none  no
## 6   4 none yes
## 7   1 some yes
## 8   2 some yes
## 9   5 some  no
## 10  6 none  no
```

Notice that the missing elements are `NA`.

There are a few different methods for dealing with missing values, though they produce the same statistical result, they have different post-estimation behavior. These are specified through the `na.action` argument to modeling commands and you can see how

these work by using the help functions: `?na.action`. In lots of the things we do, we will have to give the argument `na.rm=TRUE` to remove the missing data from the calculation (i.e., listwise delete).

1.16 Filtering with Logical Expressions and Sorting

A logical expression is one that evaluates to either `TRUE` (the condition is met) or `FALSE` (the condition is not met). There are a few operators you need to know (which are the same as the operators in Stata or SPSS).

EQUALITY `==` (two equal signs) is the symbol for logical equality. `A == B` evaluates to `TRUE` if `A` is equivalent to `B` and evaluates to `FALSE` otherwise.

INEQUALITY `!=` is the command for inequality. `A != B` evaluates to `TRUE` when `A` is not equivalent to `B`.

AND `&` is the conjunction operator. `A & B` would evaluate to `TRUE` if both `A` and `B` were met. It would evaluate to `FALSE` if either `A` and/or `B` were not met.

OR `|` (the pipe character) is the logical or operator. `A | B` would evaluate to `TRUE` if either `A` and/or `B` is met and would evaluate to `FALSE` only if neither `A` nor `B` were met.

NOT `!` (the exclamation point) is the character for logical negation. `!(A & B)` is the mirror image of `(A & B)` such that the latter evaluates to `TRUE` when the former evaluates to `FALSE`.

When using these with variables, the conditions for factors and character strings should be specified with characters. With numeric variables, the conditions should be specified using numbers. A few examples will help to illuminate things here.

```
stata.dat$x3 == "yes"

##  [1]  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE

stata.dat$x2_fac == "none"

## logical(0)

stata.dat$x2 == 1

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

stata.dat$x1 == 2

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

the `which()` command will return the observation numbers for which the logical expression evaluates to `TRUE`.

```r
which(stata.dat$x3 == "yes")

## [1] 1 4 6 7 8

which(stata.dat$x2_fac == "none")

## integer(0)

which(stata.dat$x2 == 1)

## integer(0)

which(stata.dat$x1 == 2)

## [1] 2 8
```

You can use a logical expression to subset a matrix and you will only see the observations where the conditional statement evaluates to `TRUE`. Let's use this to subset our dataset.

```r
stata.dat[which(stata.dat$x1 == 1 & stata.dat$x2 == "none"), ]

##   x1   x2  x3 x4 log_x1 recoded_x1
## 1  1 none yes  0      0          1
```

You can't evaluate whether values are finite, missing or null with the == construct. Instead, there are functions that do this.

```r
is.na(stata2.dat$x2)

##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE

is.finite(stata2.dat$x2)

##  [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

is.null(stata2.dat$x2)

## [1] FALSE
```

There are lots of other `is.` functions that you could use, too. For example, `is.factor()` and `is.numeric()` are commonly used ones.

1.17 Sorting

The `Sort` (note the capital "S") in the `DescTools` package allows us to sort vectors, matrices, tables or data frames.

```
library(DescTools)
stata.dat_sorted <- Sort(stata.dat, c("x1", "x2"), decreasing=FALSE)
stata.dat_sorted
```

```
##    x1   x2  x3 x4    log_x1 recoded_x1
## 1   1 none yes  0 0.0000000          1
## 7   1 some yes  1 0.0000000          1
## 2   2 none  no  0 0.6931472          1
## 8   2 some yes  1 0.6931472          1
## 5   3 none  no  0 1.0986123          2
## 3   3 some  no  0 1.0986123          2
## 4   4 none yes  0 1.3862944          2
## 6   4 none yes  1 1.3862944          2
## 9   5 some  no  1 1.6094379          2
## 10  6 none  no  1 1.7917595          2
```

Now, there are two datasets in our workspace, one ordered on `x1` and one original one. Both contain exactly the same information, but sorted a different way.

You try it

Using the data object that contains the `nes1996.dta` file, do the following:

1. Find the observations where both of the following conditions hold simultaneously:
   - `educ` is equal to `3. High school diploma or equivalency te`
   - `hhincome` is equal to `1. A. None or less than 2,999`
2. Save the results of the above into a new data object and print the data object.
3. Usiong your own data, try recoding a couple of variables.

1.18 Summarising by Groups

One of the things we might often want to do is to summarize or collapse data by groups. There are lots of methods to do this. Some of them, like `by` are more flexible, but produce output that still requires some massaging to be useful. Others, such as `aggregate` (which has syntax that is similar to `by`), produce a dataframe as output, but is less flexible in how it operates. There is a function in the `dplyr` package called `summarise` that summarizes data by groups (among other things). To use this effectively, however, we need to learn something else first - the 'pipe'. The pipe is implemented in `magrittr`

and is an important part of how many people argue we should be executing multiple sequential functions, rather than nesting them inside of each other. The pipe character is `%>%` and what it does is it passes whatever happens on its left side to the function on its right side.

Here's a simple example. I've got some data on strikes which you can read in as follows:

```
strikes <- import("https://quantoid.net/files/rbe/strikes_small.rda")
```

If I wanted to figure out how many strikes happened in Denmark over the time-period of the dataset, I could sum up the `strike_vol` variable for all of the observations where the country is Denmark. If I were going to try to do this in a single set of nested functions, I could do the following:

```
sum(strikes[which(strikes$country == "Denmark"), "strike_vol"])

## [1] 4205
```

An alternative with pipes would look like the following:

```
library(dplyr)
library(magrittr)
strikes %>% filter(country=="Denmark") %>% select(strike_vol) %>% sum

## [1] 4205
```

The above basically says, take the `strikes` data and use it as the data in the `filter` command, that is filtering on country. Then, take the filtered data and pass it to the `select` function where we choose one variable (`strike_vol`). Finally, pass that one variable to the function `sum`. This works well when the first argument of the function we're executing is the data that gets passed from the previous position in the pipe. However, what if we had missing data and wanted to listwise delete it? You can always use the period (`.`) to stand in for whatever is passed from the previous position in the pipe:

```
strikes %>% filter(country=="Denmark") %>%
    select(strike_vol) %>% sum(., na.rm=TRUE)

## [1] 4205
```

Now, if we wanted to figure out how many strikes were in each country in the dataset, we can replace the `filter` command above with the `group_by` function and replace the `sum` command with the appropriate `summarise` function.

```
strikes %>% group_by(country) %>%
    summarise(n_strike = sum(strike_vol))
```

```
## # A tibble: 18 x 2
##    country     n_strike
##    <fct>          <dbl>
##  1 Australia      10557
##  2 Austria          246
##  3 Belgium         2770
##  4 Canada         17035
##  5 Denmark         4205
##  6 Finland         8808
##  7 France         10840
##  8 Germany          805
##  9 Ireland        10437
## 10 Italy          29376
## 11 Japan           2229
## 12 Netherlands      570
## 13 New Zealand     5405
## 14 Norway           796
## 15 Sweden          1967
## 16 Switzerland       24
## 17 UK              8405
## 18 USA             5541
```

If we wanted to save those data for later, we could simply assign the output from the entire string to an object.

```
tmp <- strikes %>% group_by(country) %>%
    summarise(n_strike = sum(strike_vol))
tmp
```

```
## # A tibble: 18 x 2
##    country     n_strike
##    <fct>          <dbl>
##  1 Australia      10557
##  2 Austria          246
##  3 Belgium         2770
##  4 Canada         17035
##  5 Denmark         4205
##  6 Finland         8808
##  7 France         10840
##  8 Germany          805
##  9 Ireland        10437
## 10 Italy          29376
## 11 Japan           2229
## 12 Netherlands      570
## 13 New Zealand     5405
## 14 Norway           796
```

```
## 15 Sweden          1967
## 16 Switzerland       24
## 17 UK              8405
## 18 USA             5541
```

The only restriction on the arguments to summarise is that the value produced has to be a scalar (i.e., a single value). This would prevent us from using the `ci` function in the `gmodels` package to generate the confidence interval. However, we could still do this by executing the function and pulling out on the value that we want. Here's what the output to `ci` looks like.

```
library(gmodels)
ci(strikes$strike_vol)

##   Estimate   CI lower   CI upper Std. Error
##  340.95455  282.15610  399.75299   29.89631
```

Note that the second element of the vector is the lower bound and the third element is the upper bound. If we wanted a data frame with the average along with the lower and upper confidence bounds, too, we could do the following:

```
tmp <- strikes %>% group_by(country) %>%
    summarise(mean_strike = mean(strike_vol),
    lwr = ci(strike_vol)[2], upr=ci(strike_vol)[3])
tmp

## # A tibble: 18 x 4
##    country     mean_strike      lwr     upr
##    <fct>             <dbl>    <dbl>   <dbl>
##  1 Australia         459    342.     576.
##  2 Austria            10.7    3.01    18.4
##  3 Belgium           252.   170.     334.
##  4 Canada            710.   565.     854.
##  5 Denmark           263.   -18.5    544.
##  6 Finland           383.   193.     573.
##  7 France            452.  -138.    1042.
##  8 Germany            50.3    4.42    96.2
##  9 Ireland           652.   432.     872.
## 10 Italy            1224    978.    1470.
## 11 Japan              92.9   59.6    126.
## 12 Netherlands        23.8    9.96    37.5
## 13 New Zealand       338.   273.     402.
## 14 Norway             49.8   15.8     83.7
## 15 Sweden             82.0  -17.1    181.
## 16 Switzerland         1.5    0.452    2.55
## 17 UK                525.   327.     724.
## 18 USA               346.   216.     477.
```

2 Merging Datasets

Merging datasets is relatively easy in R. Just like any other package, all you need are variables to merge on. There are several functions that do merging. We'll use the ones from the `dplyr`. Each one has two arguments, `x` (the first dataset) and `y` (the second dataset). Here's how the functions perform.

Table 1: Workings of the join functions from the dplyr package

	Observations in `x`	Observations in `y`
`left_join`	all retained	those in `x` retained
`right_join`	those in `y` retained	all retained
`full_join`	all retained	all retained
`inner_join`	only those in both `x` and `y` retained	only those in both `x` and `y` retained

By default, the data are joined on all matching variable names. Otherwise, the `by` argument allows you to specify the merging variables.

```
polity <- import("https://quantoid.net/files/rbe/polity_small.dta")
ciri <- import("https://quantoid.net/files/rbe/ciri_small.dta")
```

```
lmerge <- left_join(polity, ciri)
rmerge <- right_join(polity, ciri)
fmerge <- full_join(polity, ciri)
imerge <- inner_join(polity, ciri)
nrow(lmerge)

## [1] 12590

nrow(rmerge)

## [1] 4027

nrow(fmerge)

## [1] 13005

nrow(imerge)

## [1] 3612
```

A couple of notes here.

- This preserves all duplicates. There are 3 duplicate country-years in the polity dataset and 52 duplicate country-years in the ciri dataset. We could find duplicates as follows:

```
dups <- which(duplicated(lmerge[,c("ccode", "year")]))
dups

##  [1] 5800 6402 8690 9034 9036 9038 9040 9042 9044 9046 9048 9050 9052 9054
## [15] 9056 9058 9060 9062 9080 9082 9084 9086 9088 9090 9092 9094 9096 9098
## [29] 9100 9102 9104 9106 9108 9110 9112 9114 9116 9118 9120 9122 9124 9126
## [43] 9128 9144 9146 9148 9150 9152 9154 9156 9158 9160 9162 9164 9166
```

- The `by =` variables need to have the same names. By default (without the argument specified), the command looks for the intersecting column names across the two datasets.

If we wanted to drop the duplicated years, we could do that with

```
lmerge2 <- lmerge[-dups, ]
```

The merging of many-to-one data happens exactly the same way. The result is that the smaller dataset elements get replicated for all of the observations in the bigger data with the same matching variables.

3 Statistics

Below, we will go over a set of common statistical routines.

3.1 Cross-tabulations and Categorical Measures of Association

There are (at least) three different methods for making cross-tabs in R. The simplest method is with the `table()` function. For this exercise, we'll use the GSS data from 2012.

```
gss <- import("GSS2012.dta")
gss$happy <- factorize(gss$happy)
gss$mar1 <- factorize(gss$mar1)
tab <- table(gss$happy, gss$mar1)
tab

##
##                    married widowed divorced separated never married
##   very happy           381      37       74        13            80
##   pretty happy         504     105      180        38           241
##   not at all happy      76      35       55        24            73
```

If you want marginal values on the table, you can add those with the `addmargins` function.

```
addmargins(tab)
```

```
##
##                   married widowed divorced separated never married  Sum
##   very happy          381      37       74        13            80  585
##   pretty happy        504     105      180        38           241 1068
##   not at all happy     76      35       55        24            73  263
##   Sum                 961     177      309        75           394 1916
```

Finally, for now, if you wanted either row or column proportions, you could obtain that information by using the `prop.table` function:

```
round(prop.table(tab, margin=2), 3)
```

```
##
##                   married widowed divorced separated never married
##   very happy        0.396   0.209    0.239     0.173         0.203
##   pretty happy      0.524   0.593    0.583     0.507         0.612
##   not at all happy  0.079   0.198    0.178     0.320         0.185
```

The `margin=2` argument is for column percentages, `margin=1` will give you row percentages.

You can accomplish the same thing with the (more versatile) `xtabs` function.

```
xt1 <- xtabs(~ happy + mar1, data=gss)
xt1
```

```
##                   mar1
## happy              married widowed divorced separated never married
##   very happy           381      37       74        13            80
##   pretty happy         504     105      180        38           241
##   not at all happy      76      35       55        24            73
```

The nice thing about `xtabs` is that it also works with already aggregated data.

```
gss.ag <- import("GSS2012ag.dta")
gss.ag$happy <- factorize(gss.ag$happy)
gss.ag$mar1 <- factorize(gss.ag$mar1)
head(gss.ag)
```

```
##            mar1        happy class freq
## 1       married   very happy     1   14
## 2       widowed   very happy     1    4
## 3      divorced   very happy     1    8
## 4     separated   very happy     1    3
## 5 never married   very happy     1    8
## 6       married pretty happy     1   33
```

```
xt <- xtabs(freq ~ happy + mar1, data=gss.ag )
xt
```

```
##                   mar1
## happy              married widowed divorced separated never married
##   very happy           381      37       74        13            80
##   pretty happy         504     105      180        38           241
##   not at all happy      76      35       55        24            73
```

It can also produce tables in more than two dimensions:

```
gss$class <- factorize(gss$class)
xt2 <- xtabs(freq ~ happy + mar1 + class, data=gss.ag)
xt2

## , , class = 1
##
##                   mar1
## happy              married widowed divorced separated never married
##   very happy            14       4        8         3             8
##   pretty happy          33      10       22         6            31
##   not at all happy      11       6       13         8            21
##
## , , class = 2
##
##                   mar1
## happy              married widowed divorced separated never married
##   very happy           144       9       27         6            40
##   pretty happy         200      39       93        20           128
##   not at all happy      39      14       30        10            32
##
## , , class = 3
##
##                   mar1
## happy              married widowed divorced separated never married
##   very happy           200      22       34         3            27
##   pretty happy         263      54       62        11            73
##   not at all happy      25      14        9         6            20
##
## , , class = 4
##
##                   mar1
## happy              married widowed divorced separated never married
##   very happy            23       2        5         1             5
##   pretty happy           8       2        3         1             9
##   not at all happy       1       1        3         0             0
```

You can use the `ftable` command to "flatten" the table:

```
ftable(xt2, row.vars=c("class", "mar1"))
```

```
##                     happy very happy pretty happy not at all happy
## class mar1
## 1     married                      14           33               11
##       widowed                       4           10                6
##       divorced                      8           22               13
##       separated                     3            6                8
##       never married                 8           31               21
## 2     married                     144          200               39
##       widowed                       9           39               14
##       divorced                     27           93               30
##       separated                     6           20               10
##       never married                40          128               32
## 3     married                     200          263               25
##       widowed                      22           54               14
##       divorced                     34           62                9
##       separated                     3           11                6
##       never married                27           73               20
## 4     married                      23            8                1
##       widowed                       2            2                1
##       divorced                      5            3                3
##       separated                     1            1                0
##       never married                 5            9                0
```

The `CrossTable` function in the `gmodels` package can also be quite helpful, in that it a) presents row, column and cell percentages as well as expected counts and χ^2 contributions and b) it can produce tests of independence. This can be used either on raw data or on existing tables.

```
library(gmodels)
with(gss, CrossTable(happy, mar1))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## | Chi-square contribution |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  1916
##
##
##                | mar1
##          happy |      married |      widowed |     divorced |    separated | never married |    Row Total |
## ---------------|--------------|--------------|--------------|--------------|---------------|---------------|
##     very happy |          381 |           37 |           74 |           13 |            80 |           585 |
##                |       26.144 |        5.374 |        4.387 |        4.279 |        13.499 |               |
##                |        0.651 |        0.063 |        0.126 |        0.022 |         0.137 |         0.305 |
##                |        0.396 |        0.209 |        0.239 |        0.173 |         0.203 |               |
##                |        0.199 |        0.019 |        0.039 |        0.007 |         0.042 |               |
## ---------------|--------------|--------------|--------------|--------------|---------------|---------------|
##    pretty happy |          504 |          105 |          180 |           38 |           241 |          1068 |
##                |        1.873 |        0.407 |        0.350 |        0.346 |         2.081 |               |
##                |        0.472 |        0.098 |        0.169 |        0.036 |         0.226 |         0.557 |
##                |        0.524 |        0.593 |        0.583 |        0.507 |         0.612 |               |
##                |        0.263 |        0.055 |        0.094 |        0.020 |         0.126 |               |
## ---------------|--------------|--------------|--------------|--------------|---------------|---------------|
```

```
## not at all happy |            76 |            35 |            55 |            24 |            73 |           263 |
##                  |        23.699 |         4.716 |         3.734 |        18.245 |         6.617 |               |
##                  |         0.289 |         0.133 |         0.209 |         0.091 |         0.278 |         0.137 |
##                  |         0.079 |         0.198 |         0.178 |         0.320 |         0.185 |               |
##                  |         0.040 |         0.018 |         0.029 |         0.013 |         0.038 |               |
## -----------------|---------------|---------------|---------------|---------------|---------------|---------------|
##     Column Total |           961 |           177 |           309 |            75 |           394 |          1916 |
##                  |         0.502 |         0.092 |         0.161 |         0.039 |         0.206 |               |
## -----------------|---------------|---------------|---------------|---------------|---------------|---------------|
##
##


with(gss, CrossTable(happy, mar1, expected=TRUE, prop.r=FALSE,
    prop.t=FALSE, chisq=TRUE))


##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |              Expected N |
## | Chi-square contribution |
## |           N / Col Total |
## |-------------------------|
##
##
## Total Observations in Table:  1916
##
##
##                  | mar1
##            happy |       married |       widowed |      divorced |     separated | never married |     Row Total |
## -----------------|---------------|---------------|---------------|---------------|---------------|---------------|
##       very happy |           381 |            37 |            74 |            13 |            80 |           585 |
##                  |       293.416 |        54.042 |        94.345 |        22.899 |       120.297 |               |
##                  |        26.144 |         5.374 |         4.387 |         4.279 |        13.499 |               |
##                  |         0.396 |         0.209 |         0.239 |         0.173 |         0.203 |               |
## -----------------|---------------|---------------|---------------|---------------|---------------|---------------|
##     pretty happy |           504 |           105 |           180 |            38 |           241 |          1068 |
##                  |       535.672 |        98.662 |       172.240 |        41.806 |       219.620 |               |
##                  |         1.873 |         0.407 |         0.350 |         0.346 |         2.081 |               |
##                  |         0.524 |         0.593 |         0.583 |         0.507 |         0.612 |               |
## -----------------|---------------|---------------|---------------|---------------|---------------|---------------|
## not at all happy |            76 |            35 |            55 |            24 |            73 |           263 |
##                  |       131.912 |        24.296 |        42.415 |        10.295 |        54.082 |               |
##                  |        23.699 |         4.716 |         3.734 |        18.245 |         6.617 |               |
##                  |         0.079 |         0.198 |         0.178 |         0.320 |         0.185 |               |
## -----------------|---------------|---------------|---------------|---------------|---------------|---------------|
##     Column Total |           961 |           177 |           309 |            75 |           394 |          1916 |
##                  |         0.502 |         0.092 |         0.161 |         0.039 |         0.206 |               |
## -----------------|---------------|---------------|---------------|---------------|---------------|---------------|
##
##
## Statistics for All Table Factors
##
##
## Pearson's Chi-squared test
## ------------------------------------------------------------
## Chi^2 =  115.7517     d.f. =  8     p =  2.493911e-21
##
##
##
```

You try it

Using the data object that contains the `nes1996.dta` file, do the following:

1. Create a cross-tabulation of `race` and `votetri`
2. Add to the cross-tabulation above the `gender` variable. Print the "flattened" table.

3.1.1 Measures of Association

As you saw above, specifying `chisq=T` in the call to `CrossTable` gives you Pearson's χ^2 statistic. You can also get Fisher's exact test with `fisher=T` and McNemar's test with `mcnemar=T`. Many of the other measures of association for cross-tabulations are also

available, but not generally in the same place. For example, you can get phi, Cramer's V and the contingency coefficient with the `assocstats` function in the `vcd` package:

```
library(vcd)
summary(assocstats(xt1))

##
## Call: xtabs(formula = ~happy + mar1, data = gss)
## Number of cases in table: 1916
## Number of factors: 2
## Test for independence of all factors:
##  Chisq = 115.75, df = 8, p-value = 2.494e-21
##                     X^2 df P(> X^2)
## Likelihood Ratio 114.84  8        0
## Pearson          115.75  8        0
##
## Phi-Coefficient   : NA
## Contingency Coeff.: 0.239
## Cramer's V        : 0.174
```

The `vcd` package also has a function called `Kappa` that calculates the κ statistic.

For rank-ordered correlations, the `corr.test` function has methods `spearman` and `kendall` that produce rank-order correlation statistics

3.2 Continuous-Categorical Measures of Association

t-tests can be done easily in R.

```
gss$veryhappy <- recode(gss$happy, "'very happy' = 1;
    c('pretty happy', 'not at all happy') = 0; else=NA")
ttres <- t.test(realinc ~ veryhappy, data=gss, var.equal=F)
ttres

##
##  Welch Two Sample t-test
##
## data:  realinc by veryhappy
## t = -5.5683, df = 828.86, p-value = 3.476e-08
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -17000.287  -8138.728
## sample estimates:
## mean in group 0 mean in group 1
##        30695.07        43264.58
```

3.3 Linear Models

There are tons of linear models presentation and diagnostic tools in **R**. We will first look at how to estimate a linear model using the `Duncan` data from the `car` package. These are OD Duncan's data on occupational prestige.

```
type Type of occupation. A factor with the following levels:
     'prof', professional and managerial; 'wc', white-collar;
     'bc', blue-collar.

income Percent of males in occupation earning $3500 or more in
     1950.

education Percent of males in occupation in 1950 who were
     high-school graduates.

prestige Percent of raters in NORC study rating occupation as
     excellent or good in prestige.
```

This will give me a chance to show how factors work in the linear model context.

At the heart of the modeling functions in **R** is the *formula*. Particularly the dependent variable is given first then a tilde `~` and the independent variables are then given separated by `+`. For example: `prestige ~ income + type` is a formula. Now, we have to tell **R** in what context it should evaluate that formula. For our purposes today, we'll be using the `lm` function. This will estimate an OLS regression (unless otherwise indicated with weights).

```
library(car)
data(Duncan)
lm(prestige ~ income + type,data=Duncan)

##
## Call:
## lm(formula = prestige ~ income + type, data = Duncan)
##
## Coefficients:
## (Intercept)       income     typeprof       typewc
##      6.7039       0.6758      33.1557      -4.2772

mod <- lm(prestige ~ income + type,data=Duncan)
summary(mod)

##
## Call:
## lm(formula = prestige ~ income + type, data = Duncan)
##
## Residuals:
```

```
##     Min      1Q  Median      3Q     Max
## -23.243  -6.841  -0.544   4.295  32.949
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  6.70386    3.22408   2.079   0.0439 *
## income       0.67579    0.09377   7.207 8.43e-09 ***
## typeprof    33.15567    4.83190   6.862 2.58e-08 ***
## typewc      -4.27720    5.54974  -0.771   0.4453
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.68 on 41 degrees of freedom
## Multiple R-squared:  0.893,Adjusted R-squared:  0.8852
## F-statistic:   114 on 3 and 41 DF,  p-value: < 2.2e-16
```

We have saved our model object as `mod`. If we want to see what pieces of information are in the little box labeled `mod`, we can simply type the following:

```
names(mod)
```

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

3.3.1 Adjusting the base category

It is relatively easy to adjust the base category of the factor here. We simply need to manipulate the variable's contrasts. There are many different types of these, but the one we most usually think of are *treatment contrasts*. Treatment contrasts make dummy variables for all but one level of the categorical variable, leaving one out as the base category. The first level is the one that is left out by default. We can see what the dummy variables will look like by typing:

```
contrasts(Duncan$type)
```

```
##      prof wc
## bc      0  0
## prof    1  0
## wc      0  1
```

We can modify these by using the `relevel` command. This command takes arguments - the variable and the new base level (specified as the numeric value or the level label). For example

```
data(Duncan)
Duncan$type2 <- relevel(Duncan$type, "prof")
lm(prestige ~ income + type2, data=Duncan)

##
## Call:
## lm(formula = prestige ~ income + type2, data = Duncan)
##
## Coefficients:
## (Intercept)       income      type2bc      type2wc
##     39.8595       0.6758     -33.1557     -37.4329
```

If you wanted deviation or effects coding, you could chose the `contr.sum` contrasts:

```
Duncan$type3 <- Duncan$type
contrasts(Duncan$type3) <- 'contr.sum'
lm(prestige ~ type3, data=Duncan)

##
## Call:
## lm(formula = prestige ~ type3, data = Duncan)
##
## Coefficients:
## (Intercept)       type31       type32
##       46.62       -23.86        33.82
```

Typing `?contrasts` will give you the help file on the different types of contrasts available in R.

You try it

Using the data object that contains the `nes1996.dta` file, do the following:

1. Estimate and summarize a linear regression of left-right self-placement on age, education, gender, race (3-categories) and income.

2. Change the base-category of the education variable to `'3. High school diploma or equivalency te'` and re-estimate the model.

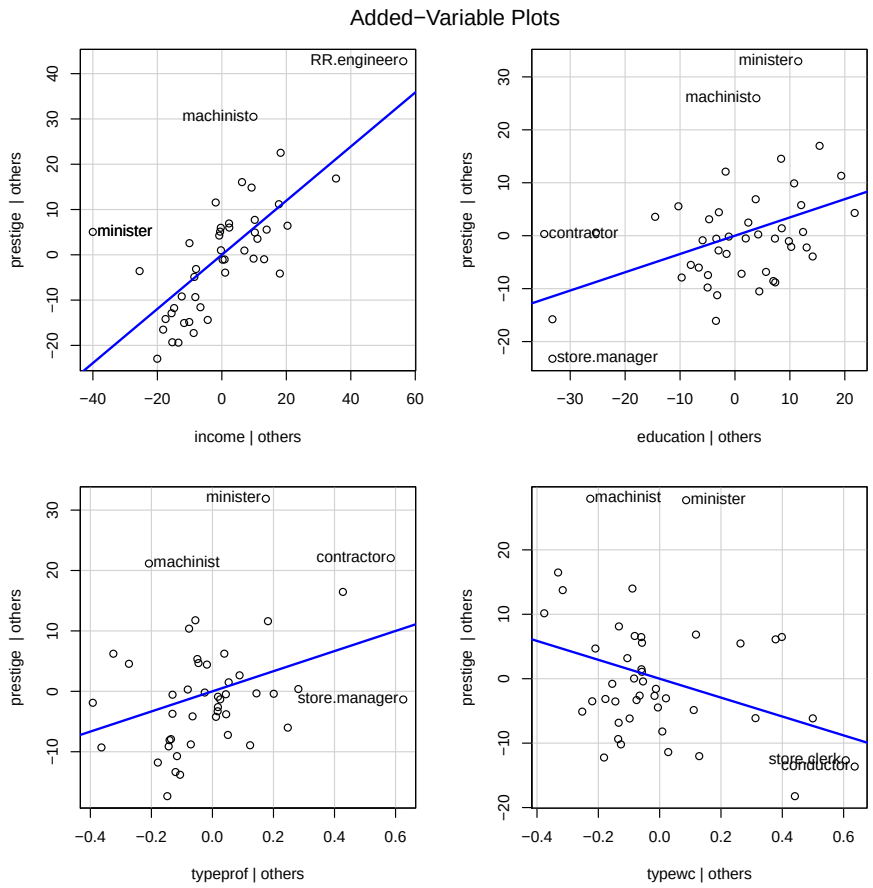
3.3.2 Model Diagnostics

There are both numeric and graphical techniques to help figure out whether there are problems with model specification. Most of these are in the `car` package.

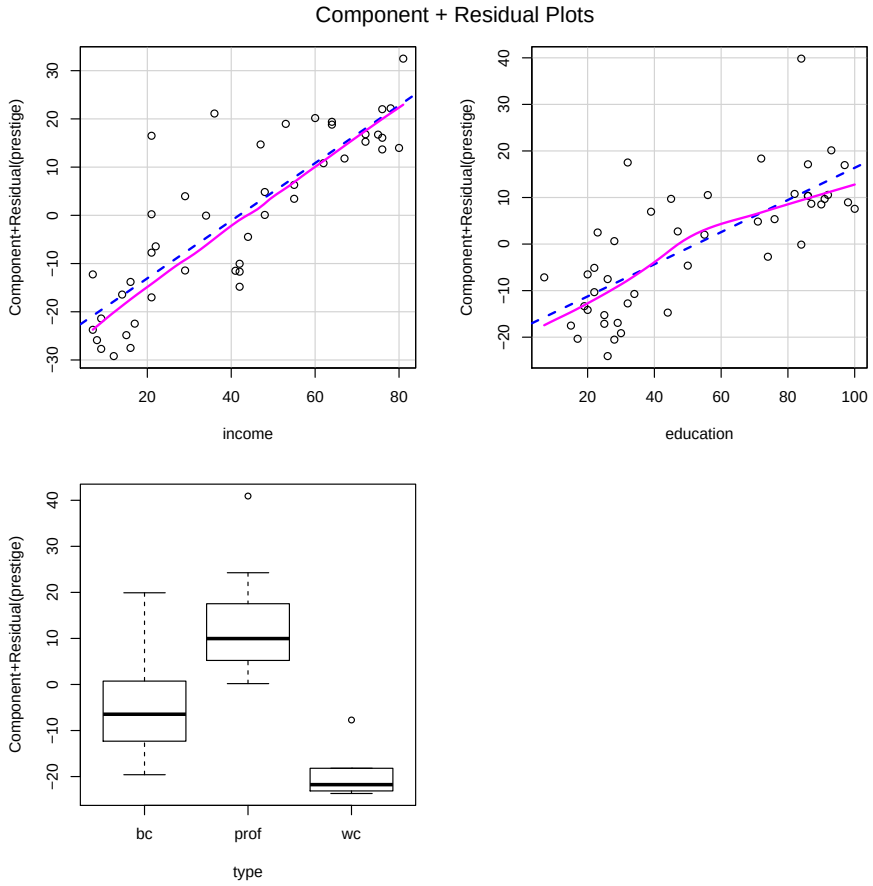
```r
mod <- lm(prestige ~ income + education + type, data=Duncan)
ncvTest(mod, var.formula = ~ income + education + type)

## Non-constant Variance Score Test
## Variance formula: ~ income + education + type
## Chisquare = 5.729855, Df = 4, p = 0.22025

outlierTest(mod)

##          rstudent unadjusted p-value Bonferonni p
## minister 3.829396         0.00045443      0.02045
```

We can also make some diagnostic plots (added variable plots and component+residual plots)

```
avPlots(mod)
```

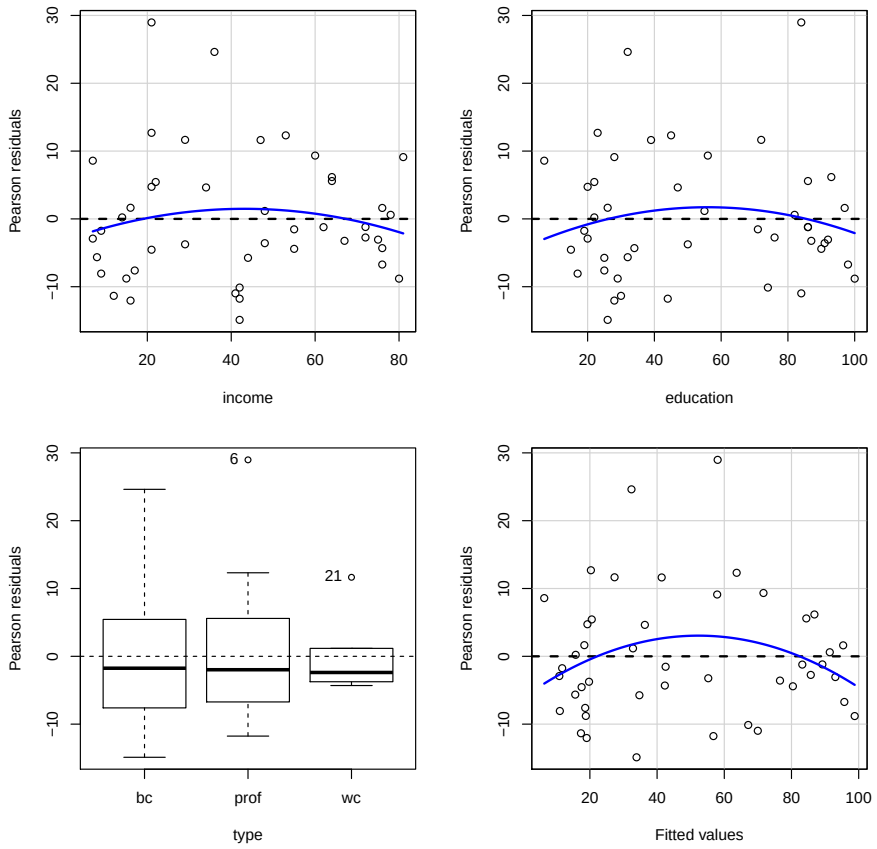


```
crPlots(mod)
```


Component + Residual Plots
Component+Residual(prestige)
income
education
type
bc
prof
wc

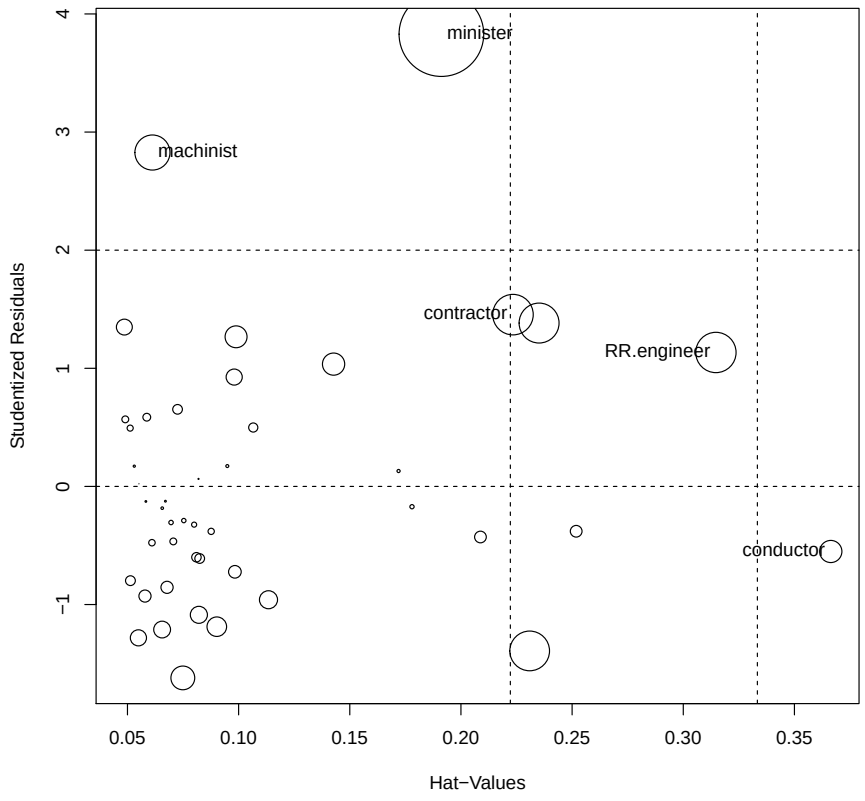
We can make the residuals-versus-fitted plot to look for patterns in the residuals

```
residualPlots(mod)

##             Test stat Pr(>|Test stat|)
## income        -0.8443           0.4037
## education     -0.8347           0.4090
## type
## Tukey test    -1.6035           0.1088
```



Finally, we could make influence plots to look for potential outliers and influential observations:

```
influencePlot(mod)

##                StudRes       Hat      CookD
## minister     3.8293960 0.1912053 0.51680533
## conductor   -0.5505711 0.3663519 0.03567303
## contractor   1.4543682 0.2234009 0.11839260
## RR.engineer  1.1339763 0.3146829 0.11725367
## machinist    2.8268000 0.0612292 0.08872915
```



You try it

Using the data object that contains the `nes1996.dta` file, do the following:

1. Use the methods we discussed above to diagnose any particular problems with the model. What are your conclusions about model specification?

3.3.3 Predict after lm

There are a number of different ways you can get fitted values after you've estimated a linear model. If you only want to see the fitted values for each observation, you can use

```
data(Duncan)
Duncan$type <- relevel(Duncan$type, "prof")
mod <- lm(prestige ~ income + type, data=Duncan)
summary(mod)

##
## Call:
## lm(formula = prestige ~ income + type, data = Duncan)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -23.243  -6.841  -0.544   4.295  32.949
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  39.85953    6.16834   6.462 9.53e-08 ***
## income        0.67579    0.09377   7.207 8.43e-09 ***
## typebc      -33.15567    4.83190  -6.862 2.58e-08 ***
## typewc      -37.43286    5.11026  -7.325 5.75e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.68 on 41 degrees of freedom
## Multiple R-squared:  0.893,Adjusted R-squared:  0.8852
## F-statistic:   114 on 3 and 41 DF,  p-value: < 2.2e-16

mod$fitted

##         accountant              pilot          architect
##           81.75848           88.51638           90.54374
##             author            chemist           minister
##           77.02795           83.11006           54.05111
##          professor             dentist           reporter
##           83.11006           93.92269           47.70456
##           engineer         undertaker             lawyer
##           88.51638           68.24269           91.21953
##          physician     welfare.worker            teacher
##           91.21953           67.56690           72.29743
##          conductor         contractor      factory.owner
##           53.78667           75.67637           80.40690
##      store.manager             banker         bookkeeper
##           68.24269           92.57111           22.02456
```

```
##       mail.carrier    insurance.agent        store.clerk
##           34.86456           39.59509           22.02456
##          carpenter        electrician        RR.engineer
##           20.89544           38.46597           61.44281
##          machinist     auto.repairman            plumber
##           31.03228           21.57123           36.43860
##  gas.stn.attendant         coal.miner streetcar.motorman
##           16.84070           11.43438           35.08702
##        taxi.driver       truck.driver   machine.operator
##           12.78596           20.89544           20.89544
##             barber          bartender        shoe.shiner
##           17.51649           17.51649           12.78596
##               cook         soda.clerk            watchman
##           16.16491           14.81333           18.19228
##            janitor          policeman              waiter
##           11.43438           29.68070           12.11017

fitted(mod)

##         accountant              pilot          architect
##           81.75848           88.51638           90.54374
##             author            chemist           minister
##           77.02795           83.11006           54.05111
##          professor             dentist           reporter
##           83.11006           93.92269           47.70456
##           engineer         undertaker             lawyer
##           88.51638           68.24269           91.21953
##          physician     welfare.worker            teacher
##           91.21953           67.56690           72.29743
##          conductor         contractor      factory.owner
##           53.78667           75.67637           80.40690
##      store.manager             banker         bookkeeper
##           68.24269           92.57111           22.02456
##       mail.carrier    insurance.agent        store.clerk
##           34.86456           39.59509           22.02456
##          carpenter        electrician        RR.engineer
##           20.89544           38.46597           61.44281
##          machinist     auto.repairman            plumber
##           31.03228           21.57123           36.43860
##  gas.stn.attendant         coal.miner streetcar.motorman
##           16.84070           11.43438           35.08702
##        taxi.driver       truck.driver   machine.operator
##           12.78596           20.89544           20.89544
##             barber          bartender        shoe.shiner
##           17.51649           17.51649           12.78596
##               cook         soda.clerk            watchman
```

```
##           16.16491           14.81333           18.19228
##            janitor          policeman             waiter
##           11.43438           29.68070           12.11017
```

Both of the above commands will produce the same output - a predicted value for each observation. We could also get fitted values using the `predict` command which will also calculate standard errors or confidence bounds.

```
pred <- predict(mod)
head(pred)

## accountant      pilot  architect     author    chemist   minister
##   81.75848   88.51638   90.54374   77.02795   83.11006   54.05111

pred.se <- predict(mod, se.fit=TRUE)
head(pred.se$fit)

## accountant      pilot  architect     author    chemist   minister
##   81.75848   88.51638   90.54374   77.02795   83.11006   54.05111

head(pred.se$se.fit)

## [1] 2.523522 2.754890 2.880750 2.561183 2.543959 4.443785

pred.mean.ci <- predict(mod, interval="confidence")
head(pred.mean.ci)

##                fit      lwr      upr
## accountant 81.75848 76.66212 86.85484
## pilot      88.51638 82.95276 94.07999
## architect  90.54374 84.72595 96.36154
## author     77.02795 71.85554 82.20037
## chemist    83.11006 77.97243 88.24769
## minister   54.05111 45.07670 63.02551

pred.ind.ci <- predict(mod, interval="prediction")
head(pred.ind.ci)

##                fit      lwr       upr
## accountant 81.75848 59.59898 103.91798
## pilot      88.51638 66.24477 110.78798
## architect  90.54374 68.20728 112.88020
## author     77.02795 54.85083  99.20507
## chemist    83.11006 60.94103 105.27909
## minister   54.05111 30.69280  77.40942
```

Notice, that when the original data are used, the fourth option indicates that these

confidence intervals are for future predictions. They are not meant to say something interesting about the observed data, rather about future or hypothetical cases.

It is also possible to make out-of-sample predictions. To do this, you need to make a new data frame that has the same variables that are in your model, with values at which you want to get predictions. Let's say that above, we wanted to get predictions for a single observation that had an income value of 50 and had "blue collar" as the type. We could do the following:

```
newdat <- data.frame(
        income = 50,
        type = "bc")
predict(mod, newdat, interval="confidence")

##        fit      lwr      upr
## 1 40.49334 33.64969 47.33698
```

If we wanted to get predictions for "blue collar" occupations with incomes of 40, 50 and 60, we could do that as follows:

```
newdat <- data.frame(
        income = c(40, 50, 60),
        type = c("bc", "bc", "bc"))
predict(mod, newdat, interval="confidence")

##        fit      lwr      upr
## 1 33.73544 28.11384 39.35704
## 2 40.49334 33.64969 47.33698
## 3 47.25123 38.93011 55.57236
```

You try it

Using the data object that contains the `nes1996.dta` and the model you estimated above, do the following:

1. Create predictions and confidence intervals for the following two different hypothetical respondents:
    - 25 year old, black female with a BA-level degree whose household income is between $40,000 and $44,999.
    - 75 year old, white male with a high school diploma whose household income is between $50,000 and $59,999.

3.3.4 Linear Hypothesis Tests

You can use the `linearHypothesis` command in R to test any hypothesis you want about any linear combination of parameters (this is akin to `lincom` in Stata). For example, let's say that we wanted to test the hypothesis that $\beta_{\text{income}} = 1$, we could do:

```
linearHypothesis(mod, "income = 1")

## Linear hypothesis test
##
## Hypothesis:
## income = 1
##
## Model 1: restricted model
## Model 2: prestige ~ income + type
##
##   Res.Df    RSS Df Sum of Sq      F   Pr(>F)
## 1     42 6038.3
## 2     41 4675.2  1    1363.1 11.954 0.001284 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

If you wanted to test whether $\beta_{\text{bc}} = \beta_{\text{wc}}$, you could do:

```
linearHypothesis(mod, "typebc=typewc")

## Linear hypothesis test
##
## Hypothesis:
## typebc - typewc = 0
##
## Model 1: restricted model
## Model 2: prestige ~ income + type
##
##   Res.Df    RSS Df Sum of Sq     F Pr(>F)
## 1     42 4742.9
## 2     41 4675.2  1    67.731 0.594 0.4453
```

If we wanted to test whether both were simultaneously zero, rather than just the same, we could use:

```
linearHypothesis(mod, c("typebc=0", "typewc=0"))

## Linear hypothesis test
##
## Hypothesis:
## typebc = 0
```

```
## typewc = 0
##
## Model 1: restricted model
## Model 2: prestige ~ income + type
##
##   Res.Df     RSS Df Sum of Sq      F    Pr(>F)
## 1     43 13022.8
## 2     41  4675.2  2    8347.6 36.603 7.575e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Or, we could use `Anova` to accomplish the same goal:

```
Anova(mod)

## Anova Table (Type II tests)
##
## Response: prestige
##           Sum Sq Df F value    Pr(>F)
## income    5922.4  1  51.938 8.428e-09 ***
## type      8347.6  2  36.603 7.575e-10 ***
## Residuals 4675.2 41
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

You try it

Using the data object that contains the `nes1996.dta` and the model you estimated above, do the following:

1. Test whether the effect of a BA level degree on left-right self-placement is the same as a high school diploma.

3.3.5 Factors and Interactions

We saw above how to change the base category. We can also discuss some other methods to present pairwise differences implied by the coefficients (i.e., dealing with the reference category problem). Here, we'll use a dataset with some more interesting categorical variables (interesting = more values).

```
library(car)
data(Ornstein)
mod <- lm(interlocks ~ nation + sector +
    log(assets), data=Ornstein)
summary(mod)
```

```
##
## Call:
## lm(formula = interlocks ~ nation + sector + log(assets), data = Ornstein)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -41.936  -6.169  -0.140   4.745  47.440
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -28.4430     4.9272  -5.773 2.47e-08 ***
## nationOTH    -3.0533     3.0872  -0.989 0.323676
## nationUK     -5.3294     3.0714  -1.735 0.084030 .
## nationUS     -8.4913     1.7174  -4.944 1.46e-06 ***
## sectorBNK    17.3227     5.1847   3.341 0.000971 ***
## sectorCON    -2.7127     5.4241  -0.500 0.617463
## sectorFIN    -1.2745     3.4121  -0.374 0.709100
## sectorHLD    -2.2916     4.6132  -0.497 0.619835
## sectorMAN     1.2440     2.3666   0.526 0.599621
## sectorMER    -0.8801     3.0346  -0.290 0.772058
## sectorMIN     1.7566     2.4448   0.719 0.473153
## sectorTRN     1.8888     3.3023   0.572 0.567888
## sectorWOD     5.1056     3.0990   1.647 0.100801
## log(assets)   5.9908     0.6814   8.792 3.24e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.26 on 234 degrees of freedom
## Multiple R-squared:  0.5353,Adjusted R-squared:  0.5094
## F-statistic: 20.73 on 13 and 234 DF,  p-value: < 2.2e-16
```

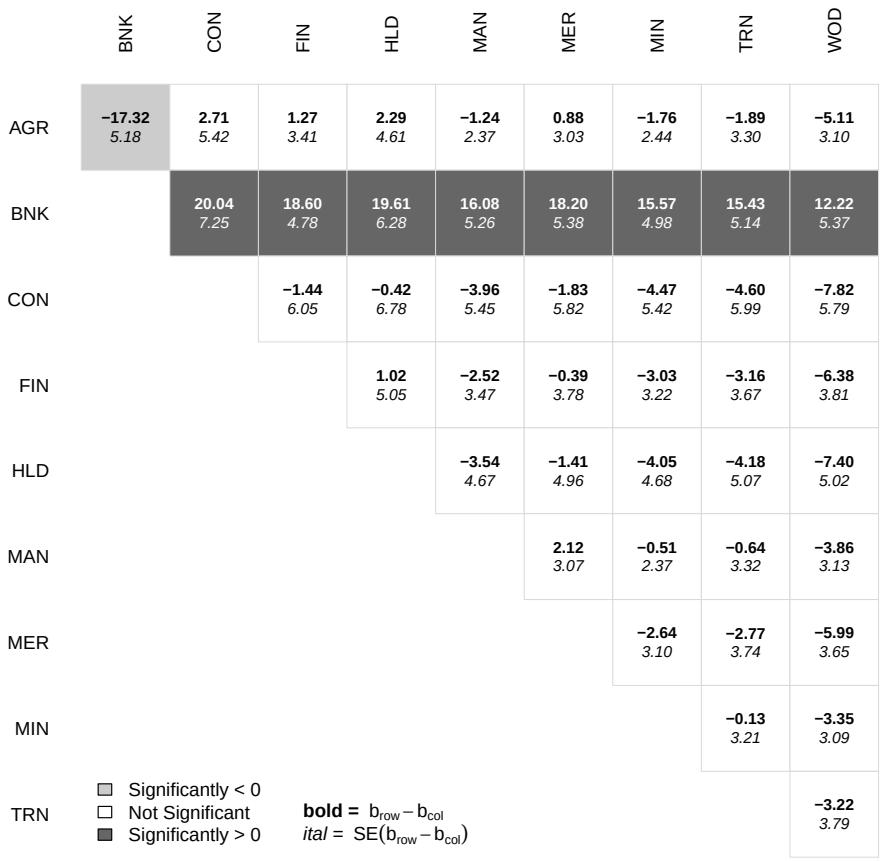
Note that 9 of the $\frac{10-9}{2} = 45$ pairwise comparisons are presented in the output. To see all 45, you could use the `glht` function (general linear hypothesis test) in the `multcomp` package.

```
library(multcomp)
glht(mod, linfct=mcp(sector = "Tukey"))

##
##   General Linear Hypotheses
##
## Multiple Comparisons of Means: Tukey Contrasts
##
##
## Linear Hypotheses:
##                Estimate
```

```
## BNK - AGR == 0  17.3227
## CON - AGR == 0  -2.7127
## FIN - AGR == 0  -1.2745
## HLD - AGR == 0  -2.2916
## MAN - AGR == 0   1.2440
## MER - AGR == 0  -0.8801
## MIN - AGR == 0   1.7566
## TRN - AGR == 0   1.8888
## WOD - AGR == 0   5.1056
## CON - BNK == 0 -20.0354
## FIN - BNK == 0 -18.5972
## HLD - BNK == 0 -19.6143
## MAN - BNK == 0 -16.0787
## MER - BNK == 0 -18.2028
## MIN - BNK == 0 -15.5661
## TRN - BNK == 0 -15.4339
## WOD - BNK == 0 -12.2171
## FIN - CON == 0   1.4382
## HLD - CON == 0   0.4211
## MAN - CON == 0   3.9567
## MER - CON == 0   1.8326
## MIN - CON == 0   4.4693
## TRN - CON == 0   4.6015
## WOD - CON == 0   7.8183
## HLD - FIN == 0  -1.0171
## MAN - FIN == 0   2.5185
## MER - FIN == 0   0.3944
## MIN - FIN == 0   3.0311
## TRN - FIN == 0   3.1633
## WOD - FIN == 0   6.3801
## MAN - HLD == 0   3.5356
## MER - HLD == 0   1.4115
## MIN - HLD == 0   4.0482
## TRN - HLD == 0   4.1804
## WOD - HLD == 0   7.3972
## MER - MAN == 0  -2.1241
## MIN - MAN == 0   0.5126
## TRN - MAN == 0   0.6448
## WOD - MAN == 0   3.8616
## MIN - MER == 0   2.6367
## TRN - MER == 0   2.7690
## WOD - MER == 0   5.9857
## TRN - MIN == 0   0.1322
## WOD - MIN == 0   3.3490
## WOD - TRN == 0   3.2168
```

To present these visually, you could use the `factorplot` function in the package of the same name.

```
library(factorplot)
fp <- factorplot(mod, factor.var = "sector")
plot(fp)
```



There are other methods for displaying these types of comparisons (I wrote a paper about this in the R Journal)

You try it

Using the data object that contains the `nes1996.dta` and the model you estimated above, do the following:

1. Look at all of the pairwise differences between education level in the most recently estimated model.

2. Present the differences you identify above graphically. What do you conclude about the effect of education?

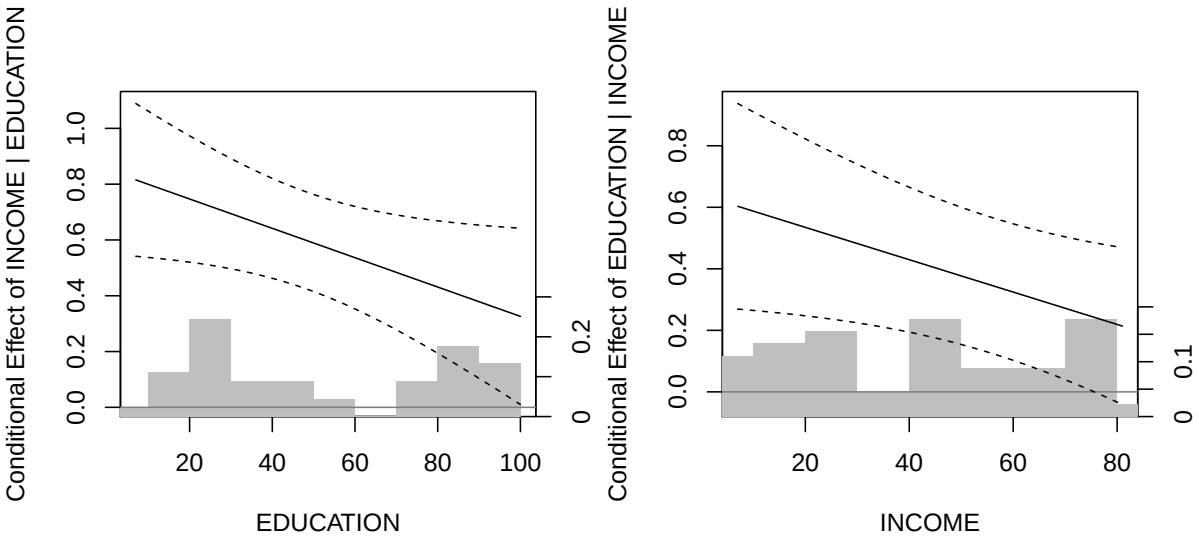
Interactions can also be handled easily in R. Simply replace a `+` between two variables with a `*`. This will include both "main" effects and all of the required product regressors.

```
mod <- lm(prestige ~ income*education + type, data= Duncan)
```

Recently, Clark, Golder and Milton (JOP) proposed a method for presenting continuous-by-continuous interactions in linear models. I have implemented their advice in DAMisc's `DAintfun2` function. First, you'll need to install the package from GitHub:

```
library(devtools)
install_github("davidaarmstrong/damisc")
```

```
library(DAMisc)
DAintfun2(mod, c("income", "education"), rug=F, hist=T,
    scale.hist=.3)
```



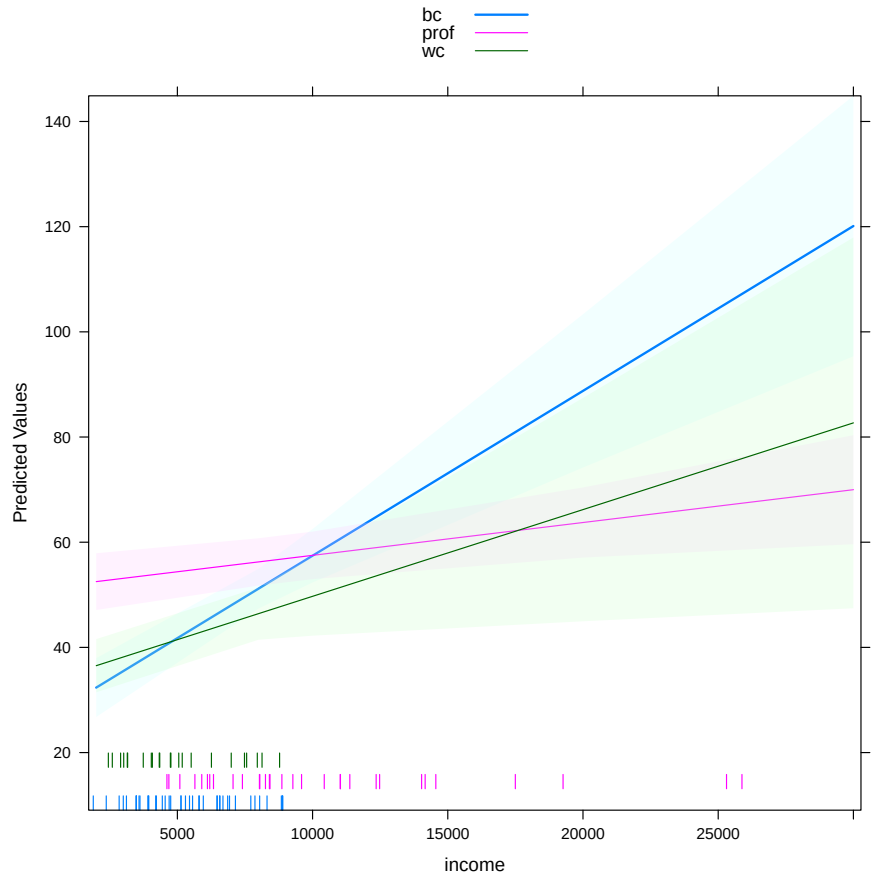
When interactions include a continuous variable and a categorical variable, the presentation becomes a bit more challenging. There are two different types of things we might want to know - the effect of the continuous variable given then different categories of the categorical variable or the difference between the categories given the continuous variable. The `intQualQuant` function provides all of this information (here, we'll use the Prestige data).

```
mod2 <- lm(prestige ~ income*type + education, data=Prestige)
```

```
intQualQuant(mod2,
    c("income", "type"),
    type="slopes",
    plot=FALSE)
```

```
## Conditional Effect of  income  given  type
##            eff        se tstat    pvalue
## bc   0.0031344 0.0005215 6.010 3.788e-08
## prof 0.0006242 0.0002217 2.816 5.957e-03
## wc   0.0016489 0.0007089 2.326 2.225e-02
```

```
intQualQuant(mod2,
    c("income", "type"),
    type="slopes",
    plot=TRUE)
```



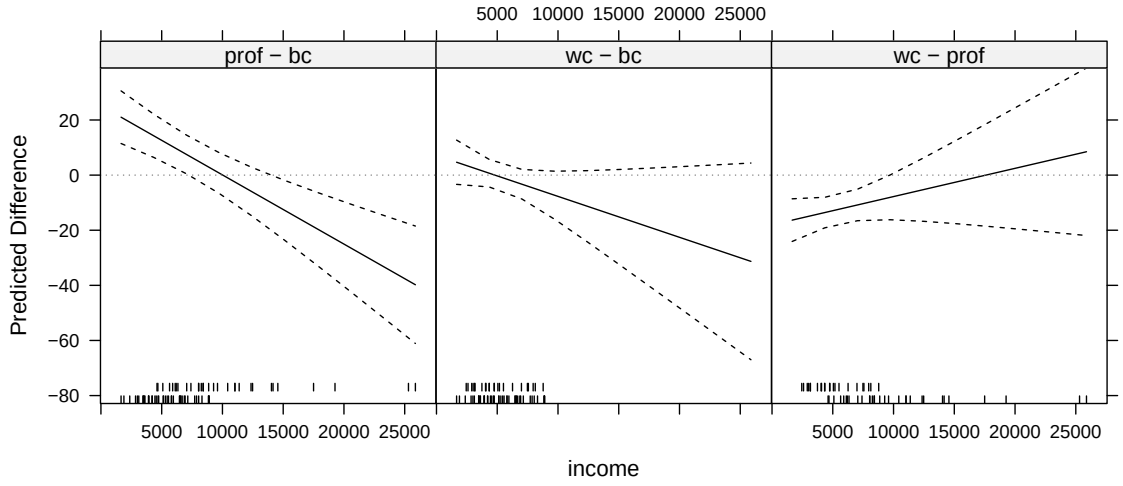
We can also look at the difference between categories given the continuous variable.

```
intQualQuant(mod2,
    c("income", "type"),
    n = 3,
    type="facs",
    plot=FALSE)

## Conditional Effect of  type  given  income
```

```
##       fit se.fit     x  contrast   lower   upper
## 1  21.016  4.803  1656 prof - bc  11.475  30.556
## 2  -9.386  4.918 13768 prof - bc -19.156   0.383
## 3 -39.788 10.725 25879 prof - bc -61.093 -18.484
## 4   4.677  4.043  1656   wc - bc  -3.353  12.708
## 5 -13.315  7.617 13768   wc - bc -28.446   1.816
## 6 -31.307 17.967 25879   wc - bc -66.996   4.381
## 7 -16.338  3.898  1656 wc - prof -24.081  -8.595
## 8  -3.929  6.659 13768 wc - prof -17.156   9.299
## 9   8.481 15.307 25879 wc - prof -21.925  38.887
```

```
p <- intQualQuant(mod2, c("income", "type"), type="facs", plot=TRUE)
```

```
update(p, layout=c(3,1), aspect=1)
```



You try it

Using the data object that contains the `nes1996.dta` and the model you estimated above, do the following:

1. Estimate a model of left-right self-placement on the interaction of age and race along with the income, education and gender variables. Figure out the effect of age given race and the effect of race given age.

3.3.6 Non-linearity: Transformations and Polynomials

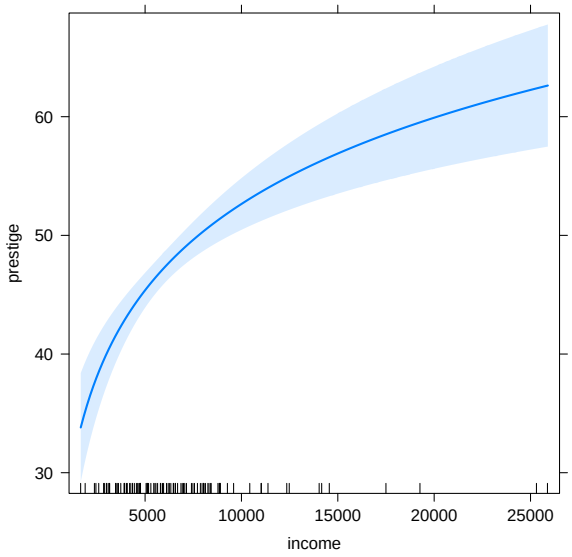
As we saw above, functional non-linear transformations (i.e., those that are themselves functions that return the transformed values, like `log()`, `sqrt()`) can be done inside the model formula. For example:

```
transmod <- lm(prestige ~ log(income) + education + type, data=Prestige)
```

There are a couple of different functions that will plot model effects - particularly useful for non-linearities. The `effects` package has been around a bit longer and applies to more types of models (though not all of them). The `visreg` package does similar things on relatively fewer models. However, the real benefit with the `visreg` package is that it will produce ggPlot objects and not just lattice objects, like the `effects` package. More on the differences and distinctions between these two later.

From the model above, the effect of `income` could be plotted with the `effects` package.[3]

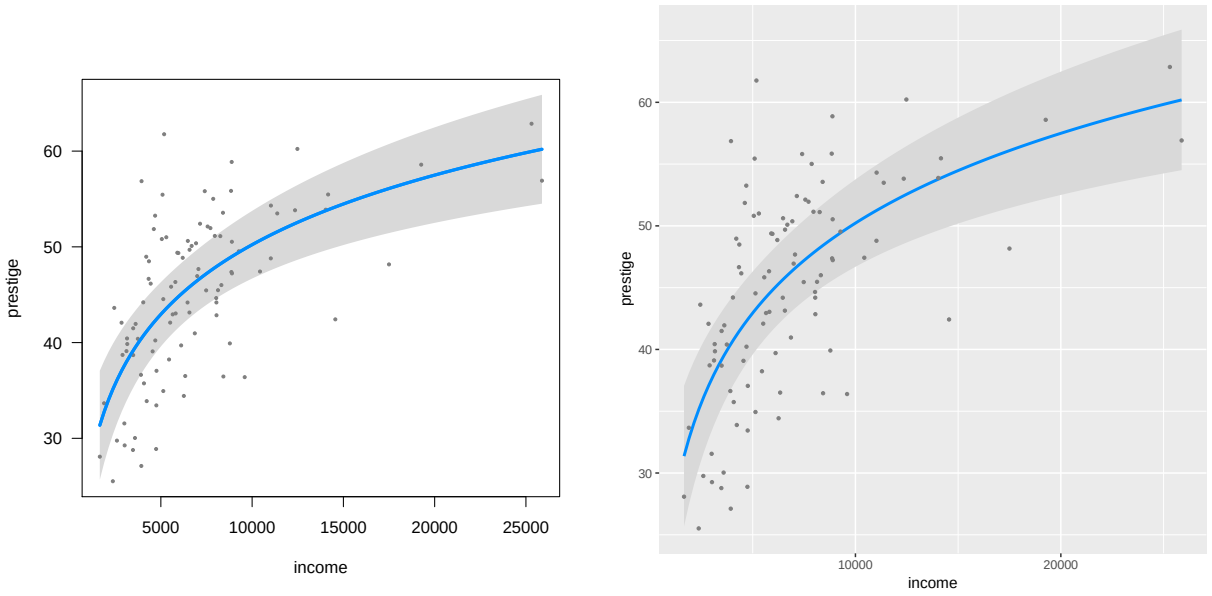
```
library(effects)
plot(predictorEffect("income", transmod, xlevels=25), main="")
```




```
library(visreg)
visreg(transmod, "income")
```

```
visreg(transmod, "income", gg=TRUE)
```

[3] We'll talk lots more about this and how it works in the visualization part of the workshop.

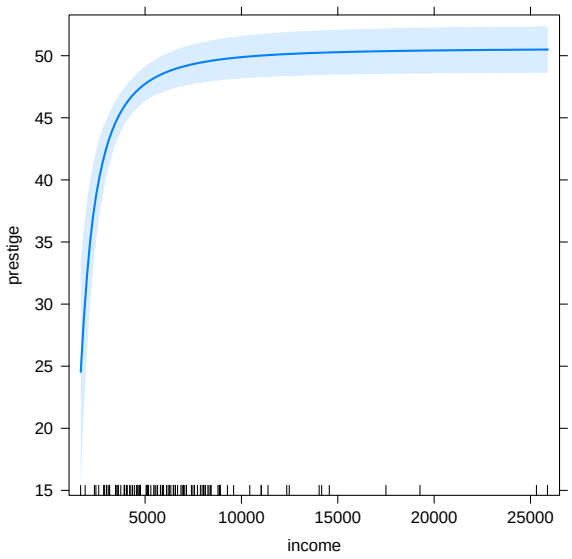


You could also include transformations (or other mathematical functions) using the `I()` function. For example:

```
transmod2 <- lm(prestige ~ I(1/income^2) + education + type, data=Prestige)
```

and that effect is just as easily plotted:

```
plot(predictorEffect("income", transmod2, xlevels=25), main="")
```



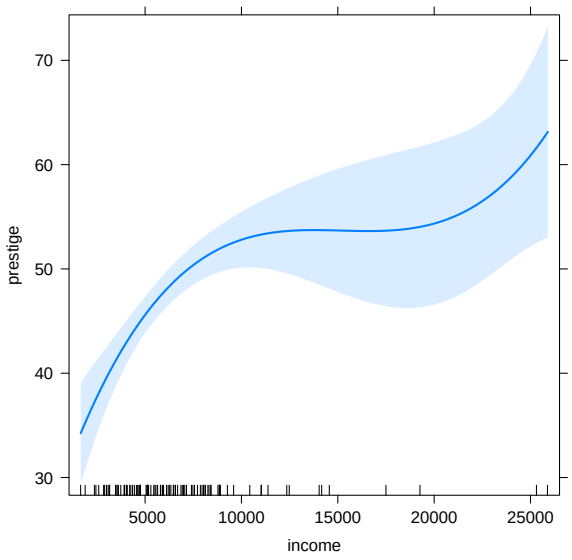
Polynomials can be included through the `poly()` function. By default, this creates orthogonal polynomials (think of it as using the principal components of your matrix of polynomial regressors). This has both some benefits and drawbacks, but you can include the raw values with the argument `raw=TRUE`.

```r
polymod <- lm(prestige ~ poly(income, 3) + education + type, data=Prestige)
summary(polymod)
```

```
##
## Call:
## lm(formula = prestige ~ poly(income, 3) + education + type, data = Prestige)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -14.262  -4.297   1.077   3.890  18.086
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)        8.5944     5.5368   1.552  0.12408
## poly(income, 3)1  50.0355     9.2846   5.389 5.53e-07 ***
## poly(income, 3)2 -23.6628     7.6143  -3.108  0.00252 **
## poly(income, 3)3  17.3362     7.5946   2.283  0.02478 *
## education          3.3907     0.6333   5.354 6.41e-07 ***
## typeprof           6.8031     3.6688   1.854  0.06693 .
## typewc            -1.6090     2.4391  -0.660  0.51112
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.68 on 91 degrees of freedom
##   (4 observations deleted due to missingness)
## Multiple R-squared:  0.8568,Adjusted R-squared:  0.8473
## F-statistic: 90.73 on 6 and 91 DF,  p-value: < 2.2e-16
```

Again, the effects can be plotted with the effects package.

```r
plot(predictorEffect("income", polymod, xlevels=25), main="")
```



You try it

Using the data object that contains the `nes1996.dta`, do the following:

1. Estimate a model of left-right self-placement as a function of a polynomial in age, education, gender, income and race.

2. Plot the effect of the polynomial.

3.3.7 Testing Between Models

Nested model testing in R is easy with `anova`. Before we do that, let's consider what the different types of `anova` and `Anova` functions do. The `anova` command (when executed on a single model) gives Type I sequential sums of squares. This is rarely (if ever) what we want as social scientists using observational data in models with control variables. When executed on two models, the `anova` function just calculates the F-test of the two models relative to each other. The `Anova` function calculates either Type II or Type III sums of squares. The Type II control for lower-order terms, but not higher-order ones. For example, if you had the model:

$$y = b_0 + b_1x_1 + b_2x_2 + b_3x_1x_2 + b_4x_3 + e,$$

when testing for the significance of x_1 using Type II sums of squares, x_2 and x_3 would be controlled for, but not x_1x_2 (the product regressor). Using Type III sums of squares, all regressors are controlled for when testing for the significance of any other variable. Consider one of the models from above:

```
anova(mod2)

## Analysis of Variance Table
##
## Response: prestige
##             Df  Sum Sq Mean Sq F value    Pr(>F)
## income       1 14021.6 14021.6 336.555 < 2.2e-16 ***
## type         2  7988.5  3994.3  95.873 < 2.2e-16 ***
## education    1  1655.5  1655.5  39.736 1.024e-08 ***
## income:type  2   890.0   445.0  10.681 6.809e-05 ***
## Residuals   91  3791.3    41.7
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Anova(mod2)

## Anova Table (Type II tests)
##
## Response: prestige
##             Sum Sq Df F value    Pr(>F)
## income      1058.8  1 25.4132 2.342e-06 ***
## type         591.2  2  7.0947   0.00137 **
## education   1068.0  1 25.6344 2.142e-06 ***
## income:type  890.0  2 10.6814 6.809e-05 ***
## Residuals   3791.3 91
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Anova(mod2, type="III")

## Anova Table (Type III tests)
##
## Response: prestige
##             Sum Sq Df F value    Pr(>F)
## (Intercept)   76.9  1  1.8458    0.1776
## income      1504.8  1 36.1180 3.788e-08 ***
## type        1003.4  2 12.0427 2.291e-05 ***
## education   1068.0  1 25.6344 2.142e-06 ***
## income:type  890.0  2 10.6814 6.809e-05 ***
## Residuals   3791.3 91
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

For two nested models, we could use the `anova` function:

```
tmp <- Prestige[complete.cases(Prestige[,c("income", "prestige",
    "education", "women", "type")]), ]
m1 <- lm(prestige ~ income + education, data=tmp)
m2 <- lm(prestige ~ income + education + women + type, data=tmp)
anova(m1, m2, test="F")

## Analysis of Variance Table
##
## Model 1: prestige ~ income + education
## Model 2: prestige ~ income + education + women + type
##   Res.Df    RSS Df Sum of Sq      F  Pr(>F)
## 1     95 5272.4
## 2     92 4679.0  3    593.45 3.8896 0.01149 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

For two non-nested models, the `lmtest` package has a number of options. For example:

```
m1 <- lm(prestige ~ education + type, data=tmp)
m2 <- lm(prestige ~ income + women, data=tmp)
library(lmtest)
encomptest(m1, m2, Prestige)

## Encompassing test
##
## Model 1: prestige ~ education + type
## Model 2: prestige ~ income + women
## Model E: prestige ~ education + type + income + women
##           Res.Df Df      F    Pr(>F)
## M1 vs. ME     92 -2 10.431 8.258e-05 ***
## M2 vs. ME     92 -3 53.481 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

jtest(m1, m2, Prestige)

## J test
##
## Model 1: prestige ~ education + type
## Model 2: prestige ~ income + women
##                  Estimate Std. Error t value  Pr(>|t|)
## M1 + fitted(M2)  0.33741    0.078230  4.3131 4.006e-05 ***
## M2 + fitted(M1)  0.81914    0.064216 12.7559 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

coxtest(m1, m2, Prestige)
```

```
## Cox test
##
## Model 1: prestige ~ education + type
## Model 2: prestige ~ income + women
##                 Estimate Std. Error  z value  Pr(>|z|)
## fitted(M1) ~ M2  -16.196     3.1413  -5.1559 2.525e-07 ***
## fitted(M2) ~ M1  -64.119     3.2608 -19.6638 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The `games` package has some better-known tests, the Vuong test and Clarke's distribution free test

```
library(games)
vuong(m1, m2)

##
## Vuong test for non-nested models
##
## Model 1 log-likelihood: -339
## Model 2 log-likelihood: -378
## Observations: 98
## Test statistic: 3.7
##
## Model 1 is preferred (p = 0.00024)

clarke(m1, m2)

##
## Clarke test for non-nested models
##
## Model 1 log-likelihood: -339
## Model 2 log-likelihood: -378
## Observations: 98
## Test statistic: 74 (76%)
##
## Model 1 is preferred (p = 4.2e-07)
```

You try it

Using the data object that contains the `nes1996.dta`, do the following:

1. Estimate two models of left-right self-placement:

   M1: include age, gender and race

   M2: include hhincome and education

2. Test to see whether one model is supported more than the other.

3.4 GLMs and the Like

3.4.1 Binary DV Models

GLMs (e.g., logit/probit, poisson/negative binomial, gamma) are easy to estimate with R. You can use the `glm()` function.

```
data(Mroz)
logit.mod <- glm(lfp ~ k5 + k618 + log(age) + wc + lwg*asinh(inc),
    data=Mroz, family=binomial(link="logit"))
probit.mod <- glm(lfp ~ k5 + k618 + log(age) + wc + lwg*asinh(inc),
    data=Mroz, family=binomial(link="probit"))
```

The `effects` package is one way of calculating predicted probabilities with measures of uncertainty:

```
library(effects)
eff <- predictorEffect("age", logit.mod, focal.levels=10)
summary(eff)

##
##  age effect
## age
##        30        33        37        40        43        47        50
## 0.7476576 0.6977984 0.6312423 0.5826375 0.5360439 0.4779514 0.4377960
##        53        57        60
## 0.4007029 0.3559769 0.3258431
##
##  Lower 95 Percent Confidence Limits
## age
##        30        33        37        40        43        47        50
## 0.6655259 0.6289454 0.5796400 0.5398455 0.4950074 0.4290016 0.3797376
##        53        57        60
## 0.3337507 0.2793653 0.2440438
```

```
##
##  Upper 95 Percent Confidence Limits
## age
##        30        33        37        40        43        47        50
## 0.8152209 0.7587753 0.6800070 0.6242201 0.5765981 0.5273284 0.4976097
##        53        57        60
## 0.4715795 0.4407484 0.4198332
```

If you want to calculate effects for a particular type of observation, you need to know what the columns of the *model matrix* or *design matrix* are called. You can find this out with the following:

```
names(coef(logit.mod))
```

```
## [1] "(Intercept)"    "k5"             "k618"           "log(age)"
## [5] "wcyes"          "lwg"            "asinh(inc)"     "lwg:asinh(inc)"
```

You can use these names in a vector that gets passed to `given.values`:

```
e <- predictorEffect("age", logit.mod, focal.levels=5,
      fixed.predictors=list(
          given.values=c(k5 = 0,
          k618 = 2, "asinh(inc)" = asinh(10))))
do.call(cbind, summary(e)[c("effect", "lower", "upper")])
```

```
##       effect     lower     upper
## 30 0.8713466 0.8096600 0.9151368
## 38 0.7849160 0.7278793 0.8327449
## 45 0.7010412 0.6357253 0.7590839
## 52 0.6163420 0.5283668 0.6973082
## 60 0.5249076 0.4091994 0.6380018
```

You try it

Using the data object that contains the `nes1996.dta`, do the following:

1. Add to your dataset a new dummy variable that indicates a vote for Bill Clinton (1) versus all other candidates (0).

2. Estimate a logistic regression model of vote for Bill Clinton on left-right self-placement, age, education, race, gender and income.

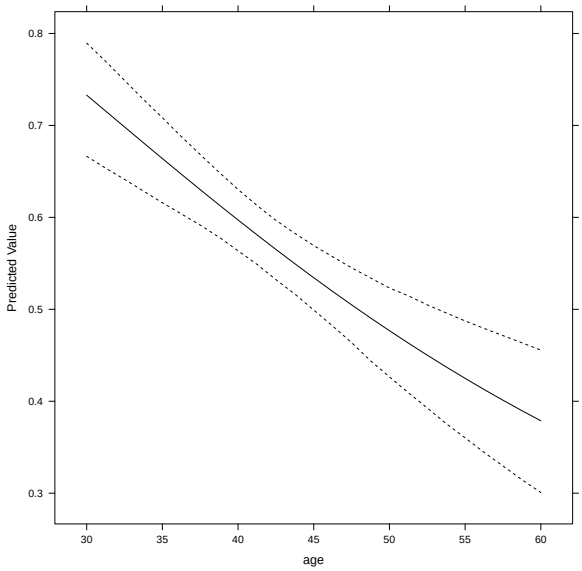
3. What is the effect of `education`? What is the size in the gender gap in the probability of voting for Clinton?

You might also want to consider first differences between predicted probabilities. This would be like what the SPost set of functions in Stata do. I wrote a function for the `DAMisc` package that simulates confidence intervals for first differences in probabilities based on the "marginal effects at reasonable values" approach. The `glmChange` function does this. Some recent work encourages researchers to adopt an "average marginal effects" approach. This is what the `glmChange2` function does.

```
glmChange(logit.mod, Mroz, diffchange="sd", sim=TRUE)

## $diffs
##            min       max        diff       lower       upper
## k5   0.6752487 0.4895261 -0.18572252 -0.25552355 -0.11254854
## k618 0.5931192 0.5776770 -0.01544223 -0.09054625  0.05671729
## age  0.6463212 0.5275628 -0.11875843 -0.19272858 -0.03908577
## wc   0.5854191 0.7698764  0.18445727  0.10769214  0.25920607
## lwg  0.5402439 0.6292021  0.08895815  0.01177340  0.16209720
## inc  0.6532116 0.5347779 -0.11843371 -0.19326728 -0.04140371
##
## $minmax
##                  k5     k618      age  wc       lwg     inc
## typical  0.0000000 1.000000 43.00000  no 1.0684031 17.7000
## min     -0.2619795 0.340063 38.96371  no 0.7746249 11.8826
## max      0.2619795 1.659937 47.03629 yes 1.3621813 23.5174
##
## attr(,"class")
## [1] "change"

glmChange2(logit.mod, "age", Mroz, change="sd")

##           mean      lower       upper
## age -0.1031133 -0.1405158 -0.06285143
```

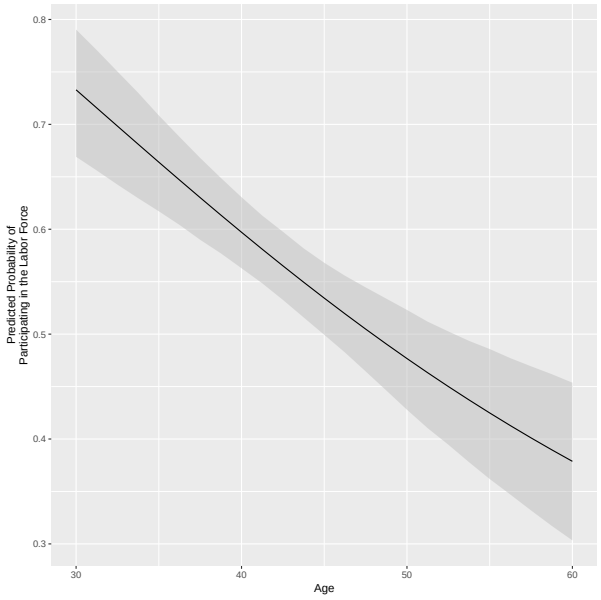
If you like the average marginal effects approach, you can use the `aveEffPlot` function from the `DAMisc` package.

```
aveEffPlot(logit.mod, "age", Mroz)
```



You could also ask it to just return the values that you could use in your own plot:

```
aep2 <- aveEffPlot(logit.mod, "age", Mroz, plot=F)
library(ggplot2)
ggplot(aep2, aes(x=s, y=mean, ymin = lower, ymax=upper)) +
    geom_ribbon(fill="gray75", alpha=.5) +
    geom_line() +
    xlab("Age") +
    ylab("Predicted Probability of\nParticipating in the Labor Force")
```



You can also get some model fit statistics with:

```
binfit(logit.mod)

##                    Names1    vals1                        Names2    vals2
```

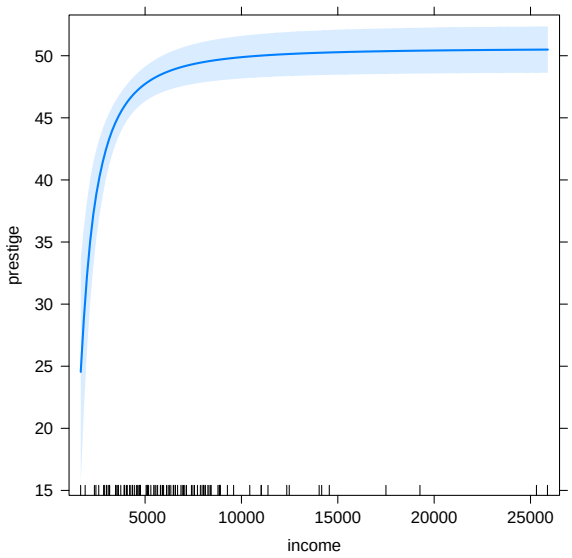
```
## 1 Log-Lik Intercept Only: -514.873          Log-Lik Full Model: -452.709
## 2                 D(745):  905.417                       LR(7):  124.329
## 3                                                  Prob > LR:    0.000
## 4          McFadden's R2:    0.121           McFadden's Adk R2:    0.105
## 5      ML (Cox-Snell) R2:    0.152 Cragg-Uhler (Nagelkerke) R2:    0.204
## 6  McKelvey & Zavoina R2:    0.218                   Efron's R2:    0.155
## 7               Count R2:    0.697                Adj Count R2:    0.298
## 8                    BIC:  958.410                         AIC:  921.417
```

```
pre(logit.mod)
```

```
## mod1:  lfp ~ k5 + k618 + log(age) + wc + lwg * asinh(inc)
## mod2:  lfp ~ 1
##
## Analytical Results
##  PMC =  0.568
##  PCP =  0.697
##  PRE =  0.298
## ePMC =  0.509
## ePCP =  0.585
## ePRE =  0.154
```

Finally, we could evaluate the interaction in the model in a couple of different ways. We could look at the effect plot to see whether there is a difference in the effects across the levels of the conditioning variable.

```
plot(effect("lwg*asinh(inc)", logit.mod, xlevels=6), type="response", main="")
```



We could also look at the second difference for the effect - the difference in first differences for the focal variable when holding the conditioning variable at a low value relative to a high value. The `secondDiff()` function in the `DAMisc` package does this. It calculates the average second difference which is held in the element `avg` and it calculates the second difference for each individual observation, held in the element `ind`.

```
secd <- secondDiff(logit.mod, c("lwg", "inc"), Mroz)
mean(secd$avg > 0)

## [1] NaN

mean(secd$ind$pval < .05)

## [1] 0
```

You try it

Using the data object that contains the `nes1996.dta`, do the following:

1. Using the model you estimated above, what is the effect of race on vote? What about gender and income?

2. How well does you model fit?

3.5 Ordinal DV Models

One function for fitting ordinal dv models `polr` is in the `MASS` package. This has been the default, more or less, for years. While this function certainly does what it says, there are models that have more flexibility in the `ordinal` package - including the generalized ordered logit (which allows for the calculation of the Brant test) and mixed ordinal DV models (which we'll come back to later on).

```
library(ordinal)
library(DAMisc)
data(france)
ologit.mod <- clm(retnat ~ lrself + male + age, data=france)
summary(ologit.mod)

## formula: retnat ~ lrself + male + age
## data:    france
##
##  link  threshold nobs logLik  AIC     niter max.grad cond.H
##  logit flexible  542  -566.76 1143.53 5(0)  3.36e-13 5.7e+04
##
## Coefficients:
##         Estimate Std. Error z value Pr(>|z|)
## lrself -0.154106   0.037345  -4.127 3.68e-05 ***
## male   -0.342544   0.162003  -2.114   0.0345 *
## age     0.010258   0.004911   2.089   0.0367 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Threshold coefficients:
##             Estimate Std. Error z value
## Better|Same  -1.6237     0.2951  -5.503
## Same|Worse    0.3419     0.2857   1.197
```

You can also use the `effects` package here.

```
eff2 <- effect("lrself", ologit.mod, xlevels=list(lrself=c(1,5,10)))
summary(eff2)

##
## lrself effect (probability) for Better
## lrself
##         1         5        10
## 0.1218185 0.2044194 0.3570088
##
## lrself effect (probability) for Same
## lrself
##         1         5        10
## 0.2351498 0.3025490 0.3326240
##
## lrself effect (probability) for Worse
## lrself
##         1         5        10
## 0.6430316 0.4930316 0.3103672
##
##  Lower 95 Percent Confidence Limits for Better
## lrself
##          1          5         10
## 0.08929251 0.17371206 0.26619482
##
##  Lower 95 Percent Confidence Limits for Same
## lrself
##         1         5        10
## 0.1906681 0.2582969 0.2861687
##
##  Lower 95 Percent Confidence Limits for Worse
## lrself
##         1         5        10
## 0.5693401 0.4462952 0.2233103
##
##  Upper 95 Percent Confidence Limits for Better
## lrself
##         1         5        10
## 0.1640583 0.2389850 0.4594081
```

```
##
##  Upper 95 Percent Confidence Limits for Same
## lrself
##         1         5        10
## 0.2863374 0.3507968 0.3825789
##
##  Upper 95 Percent Confidence Limits for Worse
## lrself
##         1         5        10
## 0.7105248 0.5398901 0.4133029
```

The plotting here has more options having to do with how you want multiple lines plotted. The options are juxtaposed (the default, figure 2(a)), superposed (figure 2(b)) or stacked (figure 2(c)).

These are done with the following commands, respectively. In each of the commands, the rug plot can be turned off by giving the plot function the argument `rug=FALSE`.

```
plot(effect("lrself", ologit.mod, xlevels=25))
```

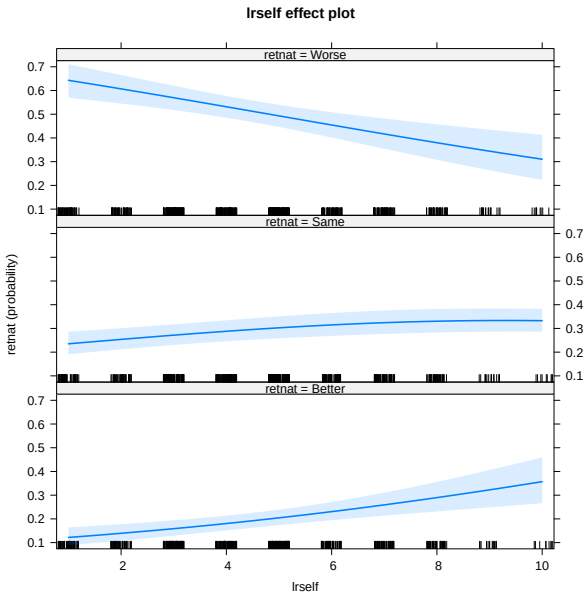
```
plot(effect("lrself", ologit.mod, xlevels=25), multiline=TRUE)
```

```
plot(effect("lrself", ologit.mod, xlevels=25), style="stacked")
```

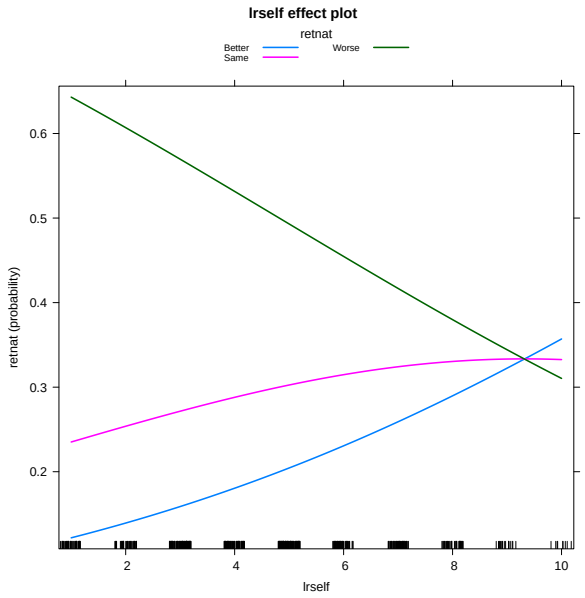
The `clm` function also does the generalized ordered logit model wherein some of the coefficients are allowed to change across response categories. This is essentially a ordinal-multinomial model hybrid.

```
gologit.mod <- clm(retnat ~   male + age, nominal = ~ lrself,
    data=france)
summary(gologit.mod)

## formula: retnat ~ male + age
## nominal: ~lrself
## data:    france
##
##  link  threshold nobs logLik  AIC     niter max.grad cond.H
##  logit flexible  542  -566.20 1144.40 5(0)  4.44e-14 5.8e+04
##
## Coefficients:
##       Estimate Std. Error z value Pr(>|z|)
## male -0.345173   0.162037  -2.130   0.0332 *
## age   0.010367   0.004912   2.111   0.0348 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

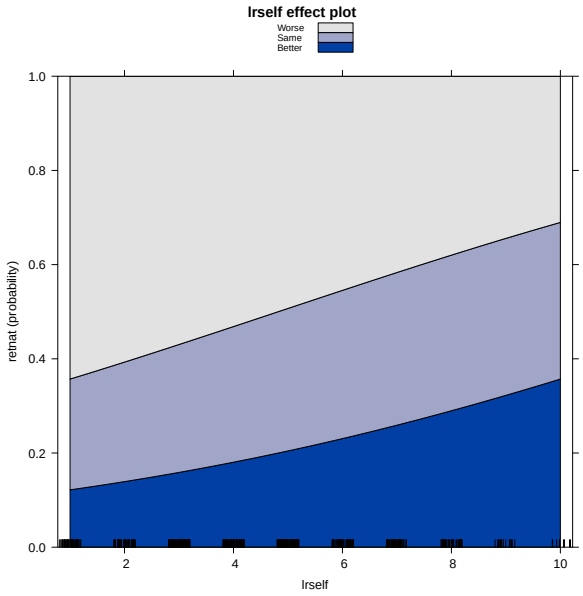
Figure 1: Effects plots for Ordered Logistic Regression Models



(a) Juxtaposed



(b) Superposed



(c) Stacked

```
##
## Threshold coefficients:
##                          Estimate Std. Error z value
## Better|Same.(Intercept) -1.78421    0.33463  -5.332
## Same|Worse.(Intercept)   0.44284    0.30183   1.467
## Better|Same.lrself       0.18569    0.04792   3.875
## Same|Worse.lrself        0.13092    0.04290   3.052
```

It is simple to do a test between the of the generalization:

```
anova(ologit.mod, gologit.mod)

## Likelihood ratio tests of cumulative link models:
##
##             formula:                      nominal: link: threshold:
## ologit.mod  retnat ~ lrself + male + age ~1       logit flexible
## gologit.mod retnat ~ male + age          ~lrself  logit flexible
##
##             no.par    AIC  logLik LR.stat df Pr(>Chisq)
## ologit.mod       5 1143.5 -566.76
## gologit.mod      6 1144.4 -566.20  1.1232  1     0.2892
```

There are also similar functions to those discussed above for binary models for considering ordinal models. For these functions, though, you'll have to use the `polr` function. For example, we can look at model fit with:

```
library(MASS)
polr.mod <- polr(retnat ~ lrself + male + age, data=france)
ordfit(polr.mod)

##                       Estimate
## Count R2              0.465
## Count R2 (Adj)        0.049
## ML R2                 0.045
## McFadden R2           0.021
## McFadden R2 (Adj)     0.013
## McKelvey & Zavoina R2 0.049
```

You can also get predicted first differences of both kinds:

```
print(ordChange(polr.mod, france, diffchange="sd", sim=TRUE))

##        Better  Same    Worse
## lrself  0.062*  0.010  -0.072*
## male    0.030*  0.006  -0.036*
## age    -0.027* -0.011   0.038*

print(ordChange2(polr.mod, "lrself", france, diffchange="sd"))

##        Better  Same    Worse
## lrself  0.062*  0.006  -0.068*
```

3.6 Multinomial DV

There are two functions (at least) for estimating the multinomial logistic regression model. The `multinom` function in the `nnet` package is easy to use, but less flexible than other alternatives. The `mlogit` function in the package of the same name is more flexible and not much harder to deal with, though the `effects` package doesn't interact well with output from `mlogit`, so if you want a traditional effects graph, then you should use the `multinom` function. We'll show an example of that first:

```
library(nnet)
data(Chile)
mnl.mod <- multinom(vote ~ log(income) + age + sex + education,
    data=Chile)

## # weights:  28 (18 variable)
## initial  value 3381.171947
## iter  10 value 3029.110953
## iter  20 value 2962.887890
## final  value 2962.537389
## converged

mnlSig(mnl.mod)

##   (Intercept) log(income)    age    sexM educationPS educationS
## N       1.079      -0.014  0.011   0.565*       0.359     -0.226
## U       1.276      -0.054  0.025* -0.236      -0.993*    -0.668*
## Y      -0.374       0.158  0.024* -0.002      -0.745*    -0.889*
```

You can do predicted probabilities for MNL with the same types of commands as above.

```
eff3 <- predictorEffect("education", mnl.mod, focal.levels=5)
summary(eff3)

##
## education effect (probability) for A
## education
##          P         PS          S
## 0.05067585 0.07009838 0.08965062
##
## education effect (probability) for N
## education
##         P        PS         S
## 0.2578039 0.5108144 0.3637748
##
## education effect (probability) for U
## education
##         P        PS         S
```

```
## 0.2435060 0.1247853 0.2208474
##
## education effect (probability) for Y
## education
##         P        PS         S
## 0.4480143 0.2943020 0.3257271
##
##  Lower 95 Percent Confidence Limits for A
## education
##          P         PS          S
## 0.03638388 0.04900163 0.07242528
##
##  Lower 95 Percent Confidence Limits for N
## education
##         P        PS         S
## 0.2246936 0.4608968 0.3326770
##
##  Lower 95 Percent Confidence Limits for U
## education
##          P         PS          S
## 0.21013254 0.09507359 0.19490609
##
##  Lower 95 Percent Confidence Limits for Y
## education
##         P        PS         S
## 0.4071336 0.2511650 0.2957926
##
##  Upper 95 Percent Confidence Limits for A
## education
##          P         PS          S
## 0.07017302 0.09932928 0.11048479
##
##  Upper 95 Percent Confidence Limits for N
## education
##         P        PS         S
## 0.2939434 0.5605171 0.3960544
##
##  Upper 95 Percent Confidence Limits for U
## education
##         P        PS         S
## 0.2802990 0.1621189 0.2491729
##
##  Upper 95 Percent Confidence Limits for Y
## education
##         P        PS         S
```

```
## 0.4896098 0.3414692 0.3571545
```

To use the `mlogit` function, you need to load the `mlogit` package and then use the `mlogit.data` function to transform your data into a format that the function can use:

```
data(Chile)
library(mlogit)
chile.ml <- mlogit.data(Chile, shape="wide", choice="vote")
mnl.mod2 <- mlogit(vote ~ 0 | log(income) + age + sex + education,
    data= chile.ml)
summary(mnl.mod2)

##
## Call:
## mlogit(formula = vote ~ 0 | log(income) + age + sex + education,
##     data = chile.ml, method = "nr")
##
## Frequencies of alternatives:
##        A        N        U        Y
## 0.073391 0.355474 0.227962 0.343173
##
## nr method
## 6 iterations, 0h:0m:0s
## g'(-H)^-1g = 2.07E-06
## successive function values within tolerance limits
##
## Coefficients :
##                  Estimate Std. Error z-value  Pr(>|z|)
## N:(intercept)  1.0800825  0.9159147  1.1792 0.2383030
## U:(intercept)  1.2768887  0.9587379  1.3318 0.1829117
## Y:(intercept) -0.3728719  0.9240548 -0.4035 0.6865679
## N:log(income) -0.0138212  0.0950387 -0.1454 0.8843736
## U:log(income) -0.0536474  0.0997201 -0.5380 0.5905911
## Y:log(income)  0.1580681  0.0958376  1.6493 0.0990794 .
## N:age          0.0108886  0.0065742  1.6563 0.0976665 .
## U:age          0.0252555  0.0067812  3.7244 0.0001958 ***
## Y:age          0.0235988  0.0065503  3.6027 0.0003150 ***
## N:sexM         0.5647097  0.1663498  3.3947 0.0006870 ***
## U:sexM        -0.2362981  0.1761208 -1.3417 0.1796992
## Y:sexM        -0.0021624  0.1678632 -0.0129 0.9897219
## N:educationPS  0.3593581  0.2886259  1.2451 0.2131078
## U:educationPS -0.9930253  0.3186012 -3.1168 0.0018281 **
## Y:educationPS -0.7446480  0.2933443 -2.5385 0.0111336 *
## N:educationS  -0.2261543  0.2131965 -1.0608 0.2887906
## U:educationS  -0.6681666  0.2183014 -3.0608 0.0022078 **
## Y:educationS  -0.8892455  0.2118204 -4.1981 2.692e-05 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Log-Likelihood: -2962.5
## McFadden R^2:  0.038623
## Likelihood ratio test : chisq = 238.04 (p.value = < 2.22e-16)
```

If you wanted to test whether the effect of `age` on the choice of `N` vs `A`, was the same as the effect of `age` on the choice of `Y` vs `A`, then you could use the `linearHypothesis` function from the `car` package.

```
linearHypothesis(mnl.mod2, "N:age = Y:age")

## Linear hypothesis test
##
## Hypothesis:
## N:age - Y:age = 0
##
## Model 1: restricted model
## Model 2: vote ~ 0 | log(income) + age + sex + education
##
##   Res.Df Df  Chisq Pr(>Chisq)
## 1   2422
## 2   2421  1 12.147  0.0004917 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The `mlogit` package also does conditional (alternative-specific) logit models, too. In the dataset `Car` that comes with the `mlogit` package, there are several variables, but the ones we'll use are `college` (college education) and `coml5` (commute less than 5 miles/day). These variables do not change by individual across choices. The variables `pricez` and `costz` (where `z` stands in for the choice) vary by choice, but not by individual. First, we need to make the data amenable to the `mlogit` function.

```
data("Car", package = "mlogit")
Car <- mlogit.data(Car, varying = 5:70, shape = "wide", sep = "",
                   choice = "choice", alt.levels = 1:6)
```

Now, we can run the model:

```
carmod <- mlogit(choice ~ cost+price | college + coml5, data=Car)
summary(carmod)

##
## Call:
## mlogit(formula = choice ~ cost + price | college + coml5, data = Car,
```

```
##     method = "nr")
##
## Frequencies of alternatives:
##        1        2        3        4        5        6
## 0.190589 0.057800 0.288999 0.074989 0.322089 0.065535
##
## nr method
## 5 iterations, 0h:0m:0s
## g'(-H)^-1g = 1.63E-05
## successive function values within tolerance limits
##
## Coefficients :
##                  Estimate Std. Error z-value  Pr(>|z|)
## 2:(intercept) -0.9585274  0.1551549 -6.1779 6.497e-10 ***
## 3:(intercept)  0.5039224  0.1019177  4.9444 7.638e-07 ***
## 4:(intercept) -0.6731628  0.1417569 -4.7487 2.047e-06 ***
## 5:(intercept)  0.6398613  0.0982459  6.5129 7.374e-11 ***
## 6:(intercept) -0.7045481  0.1414793 -4.9799 6.363e-07 ***
## cost          -0.0707752  0.0072787 -9.7236 < 2.2e-16 ***
## price         -0.1916731  0.0265703 -7.2138 5.440e-13 ***
## 2:college     -0.1627939  0.1654676 -0.9838  0.325193
## 3:college     -0.0195603  0.1064413 -0.1838  0.854197
## 4:college     -0.2070364  0.1500382 -1.3799  0.167620
## 5:college     -0.1642816  0.1025024 -1.6027  0.108999
## 6:college     -0.4381037  0.1507657 -2.9059  0.003662 **
## 2:coml5       -0.3307923  0.1513886 -2.1851  0.028885 *
## 3:coml5       -0.0948075  0.0912542 -1.0389  0.298833
## 4:coml5       -0.1767738  0.1347870 -1.3115  0.189687
## 5:coml5        0.1307450  0.0885182  1.4770  0.139665
## 6:coml5        0.0042508  0.1389393  0.0306  0.975593
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Log-Likelihood: -7252.5
## McFadden R^2:  0.011953
## Likelihood ratio test : chisq = 175.48 (p.value = < 2.22e-16)
```

While the effects package doesn't work here, there is a function called `effects` that is defined for these objects that gives the marginal effect (first derivative of the probability with respect to a covariate). These values are often better viewed in the aggregate (i.e., the average marginal effect).

```
carme <- effects(carmod, covariate="college", type="rr", data=Car)
me <- round(cbind(colMeans(carme), apply(carme, 2, sd)), 3)
colnames(me) <- c("Effect", "SD")
me
```

```
##   Effect    SD
## 1  0.078 0.052
## 2 -0.040 0.023
## 3  0.078 0.035
## 4 -0.074 0.040
## 5 -0.038 0.025
## 6 -0.249 0.139
```

3.7 Survival Models

R has a suite of functions for doing survival analysis. The ones we'll look at are in the `survival` package. In survival analysis, we are trying to predict the length of time it takes something to happen.

- The thing for which we're waiting to happen is called a failure (even if it's a good thing).
- Not all observations have to fail (though generally we assume that they will fail by $t = \infty$, but all observations have to have not failed for at least some time (i.e., they cannot enter the sample with failure).

The basic idea here is that we want to know

$$Pr(\text{failure}_t|\text{No Failure}_{t-1})$$

- Here t is an index for time, implying that our data are temporally organized.

The desired result here is a curve that tells us the probability of surviving until a given time. This is called a survival curve.

- We might also be interested in the hazard function: the probability that the observation fails around time t divided by the probability that they are still alive at time t.

We can capture the probability that observations still at risk at time t (i.e., they did not fail at time $t-1$) will remain alive in time t with:

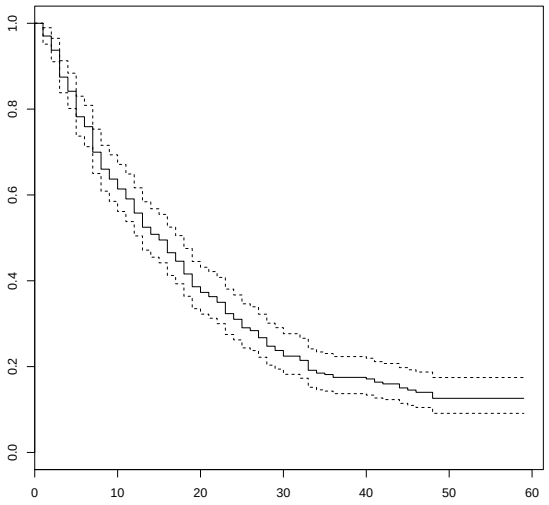
$$\hat{S}(t) = \prod_{t_i \leq t} \frac{n_i - d_i}{n_i} \tag{1}$$

Where the numerator indicates the number of at risk observations that remain alive and the denominator indicates the total number of at risk observations. These probabilities are multiplied across all previous time-periods. The first thing we need to do is read in the data and create a survival object

```
dat <- import("cabinet2.dta")
dat$invest <- factorize(dat$invest)
S <- Surv(dat$durat, dat$censor)
```

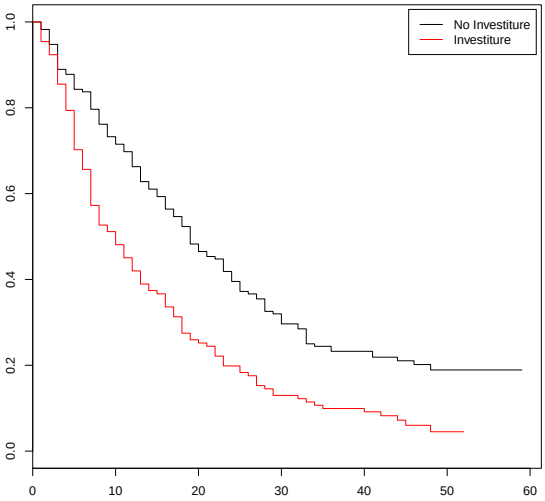
The empirical (Kaplan-Meier) curve can be made with R's survival package.

```
library(survival)
out <- survfit(S ~ 1)
plot(out)
```



We can also plot empirical curves by other covariates:

```
out2 <- survfit(
    Surv(durat, censor) ~ invest,
    data=dat)
plot(out2, col=c("black", "red"))
legend("topright", c("No Investiture",
    "Investiture"), col=c("black", "red"),
    lty = c(1,1), inset=.01)
```



If we want a more inferential method, we can estimate a survival regression model. The simplest parametric model is the *exponential* model, where exponential refers to the distribution.

- The exponential distribution has one parameter, the rate λ.
- Here, $h(t) = \lambda$ and the expected duration time is $\frac{1}{\lambda}$.
- $S(t) = exp(-\lambda(t))$
- $f(t) = \lambda(t)exp(-\lambda(t))$

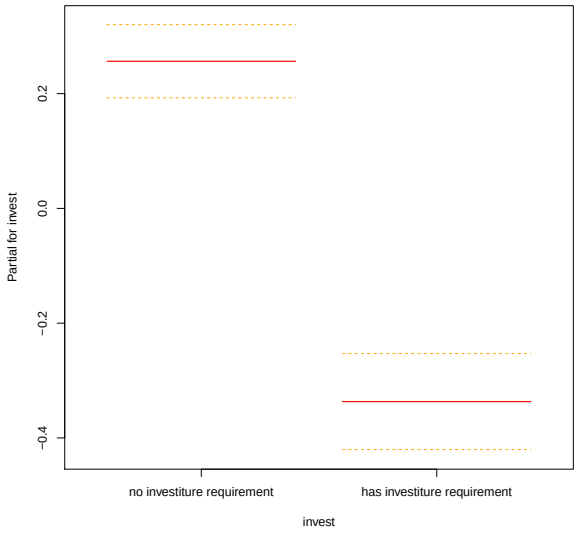
We can parameterize the distribution by making λ a function of covariates in the following way: $\lambda = exp(-\boldsymbol{x}\beta) = h(t|\boldsymbol{X})$.

```
mod <- survreg(Surv(durat, censor) ~ invest, data=dat,
    dist="exponential")
summary(mod)

##
## Call:
## survreg(formula = Surv(durat, censor) ~ invest, data = dat, dist = "exponential")
##                                  Value Std. Error     z       p
## (Intercept)                     3.3400     0.0854 39.09 < 2e-16
## investhas investiture requirement -0.5931     0.1242 -4.78 1.8e-06
##
## Scale fixed at 1
##
## Exponential distribution
## Loglik(model)= -1055.4   Loglik(intercept only)= -1066.6
##  Chisq= 22.24 on 1 degrees of freedom, p= 2.4e-06
```

```
## Number of Newton-Raphson Iterations: 4
## n= 303
```

We could then make a plot of the survival curves as a function of investitute (`invest`).

```
termplot(mod, dat, term=1, se=2)
```

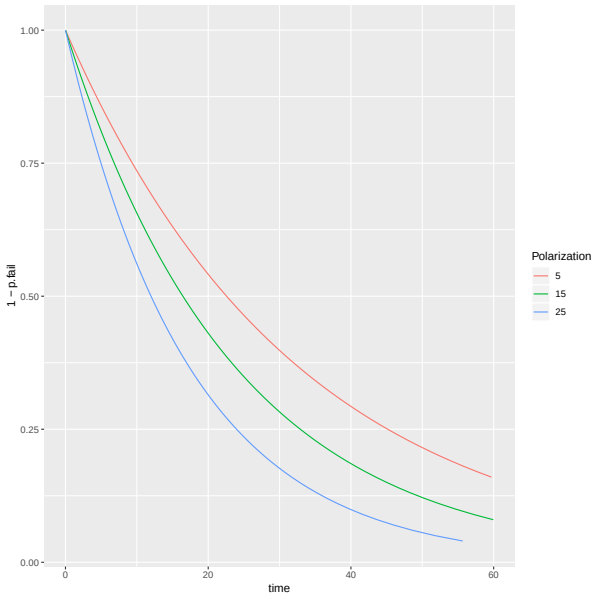


We could also use continuous variables in our models and make plots of their effects, too.

```
mod2 <- survreg(Surv(durat, censor) ~ invest + polar, data=dat,
    dist="exponential")
```

```
tmp <- data.frame(
    invest = factor(c(0), levels=c(0,1),
    labels=levels(dat$invest)),
    polar = c(5,15,25))
p <- seq(.99,0,by=-.01)
preds <- predict(mod2, newdata=tmp,
    type="quantile", p=p)
plot.data <- data.frame(
    p.fail = rep(p, each=nrow(preds)),
    time = c(preds))
plot.data$polar <- rep(
    tmp$polar, ncol(preds))
plot.data <- plot.data[
    which(plot.data$time < 60), ]
ggplot(plot.data, aes(x=time, y=1-p.fail,
        colour=as.factor(polar))) +
```

```
    geom_line() +
    scale_colour_discrete(name="Polarization")
```



The exponential model assumes a constant hazard over time.

- Here, the baseline hazard is thought to be always increasing or always decreasing (i.e., monotonic, including flat [like the exponential]).
- Here, the hazard is given by $h(t) : \lambda p(\lambda t)^{p-1}$ for $\lambda > 0, t > 0, p > 0$.
- p is the shape parameter, where $p > 1$ indicates increasing hazard with time, $p < 1$ indicates decreasing hazard with time and $p = 1$ indicates constant hazard over time.
- The survivor function is $S(t) = exp(-(\lambda t)^p)$.
- The hazard function is $pt^{p-1}exp(\alpha \boldsymbol{x})$
- The Weibull model, though more flexible than the exponential model, is still a proportional hazards model.

There are two different parameterizations of the model, the Proportional Hazards (PH) parameterization and the Accelerated Failure Time (AFT) parameterization.

	PH	AFT
λ	$exp(xb)$	$exp(-pxb)$
$h(t)$	$p\lambda t^{p-1}$	$p\lambda t^{p-1}$
$S(t)$	$exp(-\lambda t^p)$	$exp(-\lambda t^p)$
$E(T)$	$exp\left(\frac{-1}{p}xb\right)\Gamma\left(1+\frac{1}{p}\right)$	$exp(xb)\Gamma\left(1+\frac{1}{p}\right)$

```
mod3 <- survreg(Surv(durat, censor) ~ invest + polar, data=dat,
    dist="weibull")
summary(mod3)

##
## Call:
## survreg(formula = Surv(durat, censor) ~ invest + polar, data = dat,
##     dist = "weibull")
##                                     Value Std. Error     z       p
## (Intercept)                       3.63341    0.08541 42.54 < 2e-16
## investhas investiture requirement -0.30063    0.11397 -2.64 0.00835
## polar                             -0.03014    0.00432 -6.98 2.9e-12
## Log(scale)                        -0.17605    0.05035 -3.50 0.00047
##
## Scale= 0.839
##
## Weibull distribution
## Loglik(model)= -1031.1   Loglik(intercept only)= -1065.9
##  Chisq= 69.62 on 2 degrees of freedom, p= 7.6e-16
## Number of Newton-Raphson Iterations: 5
## n= 303
```

Here, the `Scale` term is $\frac{1}{p}$, so $p = 1.19$, meaning increasing hazards with time. Further, R gives you the AFT parameterization.

The weibull and exponential models assume a monotonic form of the baseline hazard (i.e., the hazard when all variables are zero).

- Further, the baseline hazard is assumed to have a particular distributional form based on the distribution chosen.
- The Cox model assumes nothing about the functional or distributional form of the baseline hazard $h_0(t)$

In the Cox model, the hazard is given by:

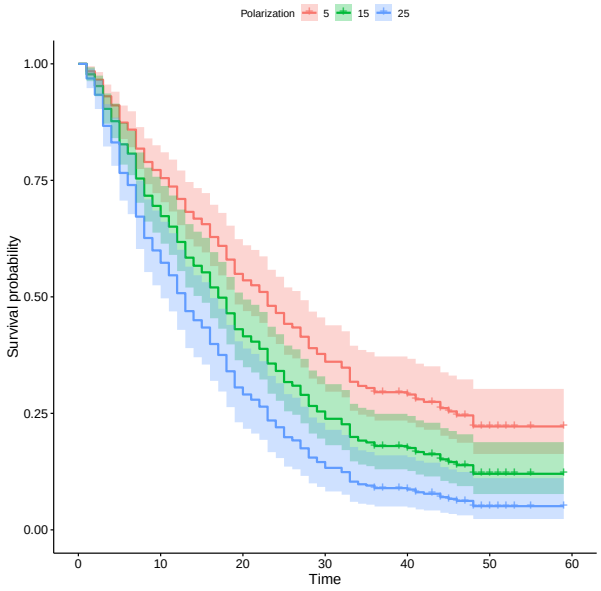
$$h_i(t) = exp(\boldsymbol{x}\beta)h_0(t) \tag{2}$$

Here, $h_0(t)$ is not parameterized - there are no distributional assumptions made about it.

```
mod <- coxph(Surv(durat, censor) ~ invest + polar, data=dat)
summary(mod)

## Call:
## coxph(formula = Surv(durat, censor) ~ invest + polar, data = dat)
##
```

```
##   n= 303, number of events= 260
##
##                                     coef exp(coef) se(coef)     z
## investhas investiture requirement 0.353460  1.423985 0.135881 2.601
## polar                             0.034121  1.034710 0.005317 6.417
##                                   Pr(>|z|)
## investhas investiture requirement  0.00929 **
## polar                             1.39e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##                                   exp(coef) exp(-coef) lower .95 upper .95
## investhas investiture requirement     1.424     0.7023     1.091     1.859
## polar                                 1.035     0.9665     1.024     1.046
##
## Concordance= 0.648  (se = 0.017 )
## Rsquare= 0.187   (max possible= 1 )
## Likelihood ratio test= 62.83  on 2 df,   p=2e-14
## Wald test            = 68.48  on 2 df,   p=1e-15
## Score (logrank) test = 71.58  on 2 df,   p=3e-16
```

```
library(survminer)
tmp <- data.frame(
    invest = factor(c(0), levels=c(0,1),
    labels=levels(dat$invest)),
    polar = c(5,15,25))
sfmod <- survfit(mod, newdata=tmp)
ggsurvplot(sfmod, data=tmp)$plot +
    scale_colour_discrete(name="Polarization", labels = c("5", "15", "25")) +
    scale_fill_discrete(name="Polarization", labels = c("5", "15", "25"))
```



3.8 Multilevel Models

R has good functionality when it comes to multilevel models. There are options for estimating linear and non-linear GLMs as well as ordinal and multinomial models (in a somewhat different context). We're going to use the `mlbook` data from the second edition of Tom Snijders and Roel Bosker's book "Multilevel Analysis". For those interested, there are great ML resources for R on the book's website: https://www.stats.ox.ac.uk/~snijders/mlbook.htm#data.

```
mlbook <- import("mlbook2_r.dat", header=T)
```

The main package gor estimating multilevel models (in the frequentist context, at least) is the `lme4` package. First, we'll just talk about estimating a two-level model with only level-1 covariates.

```
library(lme4)
m1 <- lmer(langpost ~ iq_verb + sch_iqv + (1|schoolnr), data=mlbook)
summary(m1)

## Linear mixed model fit by REML ['lmerMod']
## Formula: langpost ~ iq_verb + sch_iqv + (1 | schoolnr)
##    Data: mlbook
##
## REML criterion at convergence: 24893.9
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.2201 -0.6399  0.0631  0.7054  3.2173
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  schoolnr (Intercept)  8.785   2.964
##  Residual             40.442   6.359
## Number of obs: 3758, groups:  schoolnr, 211
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)  41.1132     0.2329 176.526
## iq_verb       2.4536     0.0555  44.212
## sch_iqv       1.3127     0.2627   4.997
##
## Correlation of Fixed Effects:
##         (Intr) iq_vrb
## iq_verb -0.007
## sch_iqv  0.043 -0.209
```

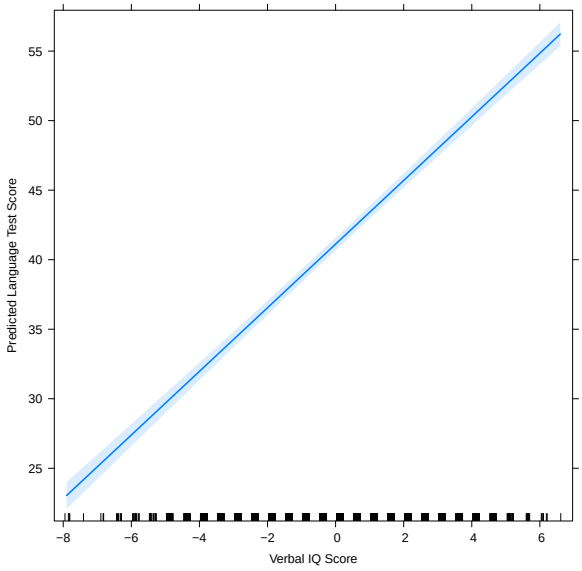
What you'll notice is that there are no p-values on the model output summary. To some in the R community, this is a feature rather than a flaw because there is debate about what the appropriate reference distribution is for these tests. However, that debate doesn't help you understand whether things are significant or not. To solve this problem, there is a function package called `lmerTest`, which implements a test of these coefficients. Once you load the package, you simply estimate the `lmer` model again and this time when you summarize it, it will show you p-values.[4]

```
library(lmerTest)
m1 <- lmer(langpost ~ iq_verb + sex + ses + (1|schoolnr), data=mlbook)
summary(m1)

## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula: langpost ~ iq_verb + sex + ses + (1 | schoolnr)
##    Data: mlbook
##
## REML criterion at convergence: 24572.4
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.1030 -0.6321  0.0717  0.6914  3.4569
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  schoolnr (Intercept)  9.402   3.066
##  Residual             36.795   6.066
## Number of obs: 3758, groups:  schoolnr, 211
##
## Fixed effects:
##              Estimate Std. Error        df t value Pr(>|t|)
## (Intercept) 3.994e+01  2.560e-01 2.759e+02  156.01   <2e-16 ***
## iq_verb     2.288e+00  5.427e-02 3.706e+03   42.17   <2e-16 ***
## sex         2.408e+00  2.030e-01 3.624e+03   11.86   <2e-16 ***
## ses         1.631e-01  1.103e-02 3.739e+03   14.78   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##         (Intr) iq_vrb sex
## iq_verb -0.014
## sex     -0.381  0.034
## ses      0.024 -0.292 -0.031
```

[4]You only have to estimate the model once after the package is loaded. We're only estimating it again because the package hadn't been loaded yet.

The effects package works here, too.

```
e <- predictorEffect("iq_verb", m1)
plot(e, main = "", xlab="Verbal IQ Score",
    ylab = "Predicted Language Test Score")
```



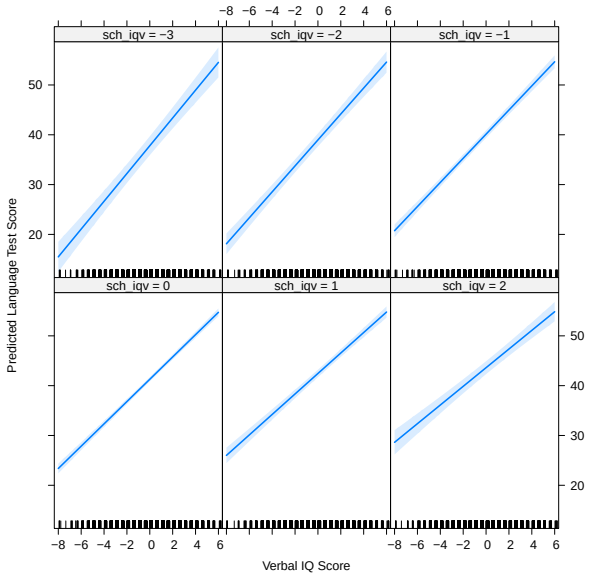
Including level-2 variables and even cross-level interactions is also easy here.

```
m2 <- lmer(langpost ~ iq_verb*sch_iqv + sex + ses + sch_ses +  minority + sch_min +
summary(m2)

## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula: langpost ~ iq_verb * sch_iqv + sex + ses + sch_ses + minority +
##     sch_min + (1 | schoolnr)
##    Data: mlbook
##
## REML criterion at convergence: 24550.8
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.1157 -0.6327  0.0673  0.6924  3.6143
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  schoolnr (Intercept)  8.465   2.909
##  Residual             36.747   6.062
## Number of obs: 3758, groups:  schoolnr, 211
##
## Fixed effects:
```

```
##                   Estimate Std. Error         df t value Pr(>|t|)
## (Intercept)       40.03585    0.26867  273.57904 149.017  < 2e-16 ***
## iq_verb            2.23924    0.05560 3548.19312  40.277  < 2e-16 ***
## sch_iqv            1.16111    0.31355  231.81828   3.703 0.000266 ***
## sex                2.41326    0.20280 3628.81491  11.900  < 2e-16 ***
## ses                0.16566    0.01148 3543.52242  14.436  < 2e-16 ***
## sch_ses           -0.09175    0.04432  228.02804  -2.070 0.039547 *
## minority           0.11470    0.57617 3582.24576   0.199 0.842213
## sch_min            1.19596    1.89132  269.87733   0.632 0.527698
## iq_verb:sch_iqv   -0.18322    0.05569 3332.83260  -3.290 0.001012 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr) iq_vrb sch_qv sex    ses    sch_ss minrty sch_mn
## iq_verb     -0.023
## sch_iqv     -0.093 -0.169
## sex         -0.368  0.034 -0.002
## ses          0.012 -0.265  0.044 -0.027
## sch_ses      0.000  0.064 -0.473 -0.006 -0.256
## minority     0.011  0.116 -0.034  0.003  0.086 -0.017
## sch_min     -0.346 -0.042  0.258  0.016 -0.023  0.081 -0.274
## iq_vrb:sch_ -0.104  0.057  0.089 -0.011 -0.014 -0.012 -0.094 -0.115
```

You can also plot the effect of the cross-level interaction:

```
e <- effect("iq_verb*sch_iqv", m2,
    xlevels=list(sch_iqv = c(-3,-2,-1,0,1,2),
    iq_verb = seq(-8, 6, length=25)))
plot(e,  as.table=T,
    main = "", xlab="Verbal IQ Score",
    ylab = "Predicted Language Test Score")
```



With one little command, you could also add group-means of variables to the dataset:

```
addGmean <- function(vars, groupvars, data, stat = "mean", na.rm=TRUE, ...){
    # makes sure the plyr package is loaded
    require(plyr)
    # creates a formula from the vector of input variables.
    form <- as.formula(paste(" ~ ", paste(vars, collapse="+"), sep=""))
    # creates the design matrix from the formula.
    X <- model.matrix(form, data)[,-1]
    # creates a unique grouping variable based on
    # the input grouping variables.
    g <- apply(data[, groupvars, drop=FALSE], 1, paste, collapse=":")
    # calculates the statistic for each variable for each group
    bys <- by(X, list(g), apply, 2, stat, na.rm=na.rm, ...)
    # creates a matrix from the results.
    bys <- do.call(rbind, bys)
    # adds a variable "g" to the data set that is just a copy of
    # the grouping variable
    data$g <- g
    # converts the matrix of results into a data frame.
    meandata <- as.data.frame(bys)
    # appends "_gm" to the end of each variable name
    names(meandata) <- paste(names(meandata), "_gm", sep="")
    # puts the grouping variable into the dataset.
    meandata$g <- rownames(meandata)
    # merges the individual and group-level data together
    out <- join(data, meandata, by="g")
    # returns the merged data,
    return(out)
}
```

If you copy and paste the above command into R, it will make it available to you (we'll talk more later on about how you could always make sure it was loaded). Then, you could use the command to add group means to you dataset and then make both the between and within variables.

```
new <- addGmean(c("iq_verb", "ses", "sex", "minority"),
    "schoolnr", mlbook)
new$iqv_w <- new$iq_verb - new$iq_verb_gm
new$sex_w <- new$sex - new$sex_gm
new$ses_w <- new$ses - new$ses_gm
new$min_w <- new$minority - new$minority_gm
m3 <- lmer(langpost ~ iqv_w + iq_verb_gm + sex_w + sex_gm +
     min_w + minority_gm + ses_w + ses_gm +  (1|schoolnr), data=new)
summary(m3)

## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula:
## langpost ~ iqv_w + iq_verb_gm + sex_w + sex_gm + min_w + minority_gm +
##     ses_w + ses_gm + (1 | schoolnr)
##    Data: new
##
## REML criterion at convergence: 24555.3
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.1048 -0.6299  0.0708  0.6891  3.4571
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  schoolnr (Intercept)  8.838   2.973
##  Residual             36.793   6.066
## Number of obs: 3758, groups:  schoolnr, 211
##
## Fixed effects:
##               Estimate Std. Error         df t value Pr(>|t|)
## (Intercept)   39.35372    0.87775  224.14387  44.835   <2e-16 ***
## iqv_w          2.25023    0.05557 3541.36099  40.495   <2e-16 ***
## iq_verb_gm     3.47030    0.31195  219.02545  11.125   <2e-16 ***
## sex_w          2.38858    0.20436 3541.36099  11.688   <2e-16 ***
## sex_gm         3.54642    1.72557  224.87064   2.055    0.041 *
## min_w         -0.06974    0.57535 3541.36099  -0.121    0.904
## minority_gm    0.56204    1.89904  222.00697   0.296    0.768
## ses_w          0.16513    0.01148 3541.36099  14.380   <2e-16 ***
## ses_gm         0.06908    0.04421  199.21218   1.562    0.120
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr) iqv_w  iq_vr_ sex_w  sex_gm min_w  mnrty_ ses_w
## iqv_w        0.000
## iq_verb_gm  -0.058  0.000
## sex_w        0.000  0.035  0.000
## sex_gm      -0.958  0.000  0.027  0.000
## min_w        0.000  0.122  0.000  0.003  0.000
## minority_gm -0.214  0.000  0.258  0.000  0.107  0.000
## ses_w        0.000 -0.264  0.000 -0.027  0.000  0.085  0.000
## ses_gm       0.106  0.000 -0.498  0.000 -0.108  0.000  0.074  0.000
```

You could even test the equivalence of the between and within effects with `linearHypothesis`.

```
linearHypothesis(m3, "iqv_w = iq_verb_gm")

## Linear hypothesis test
##
## Hypothesis:
## iqv_w - iq_verb_gm = 0
##
## Model 1: restricted model
## Model 2: langpost ~ iqv_w + iq_verb_gm + sex_w + sex_gm + min_w + minority_gm +
##     ses_w + ses_gm + (1 | schoolnr)
##
##   Df  Chisq Pr(>Chisq)
## 1
## 2  1 14.826  0.0001179 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

linearHypothesis(m3, "min_w = minority_gm")

## Linear hypothesis test
##
## Hypothesis:
## min_w - minority_gm = 0
##
## Model 1: restricted model
## Model 2: langpost ~ iqv_w + iq_verb_gm + sex_w + sex_gm + min_w + minority_gm +
##     ses_w + ses_gm + (1 | schoolnr)
##
##   Df  Chisq Pr(>Chisq)
## 1
## 2  1 0.1014     0.7502
```

Adding random coefficients is as easy as adding another variable to the formula inside the random component. For example, if we wanted to add a random coefficient for `iqv_w` and allow that to be predicted by `iq_verb_gm`, then we could do that as follows:

```
m4 <- lmer(langpost ~ iqv_w*iq_verb_gm + sex_w + sex_gm +
     min_w + minority_gm + ses_w + ses_gm +  (1 + iqv_w|schoolnr),
     data=new)
summary(m4)

## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula:
## langpost ~ iqv_w * iq_verb_gm + sex_w + sex_gm + min_w + minority_gm +
##     ses_w + ses_gm + (1 + iqv_w | schoolnr)
##    Data: new
##
## REML criterion at convergence: 24526.7
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.1201 -0.6297  0.0726  0.6835  3.0316
##
## Random effects:
##  Groups   Name        Variance Std.Dev. Corr
##  schoolnr (Intercept)  8.946   2.9910
##           iqv_w        0.188   0.4335   -0.70
##  Residual             36.083   6.0069
## Number of obs: 3758, groups:  schoolnr, 211
##
## Fixed effects:
##                    Estimate Std. Error         df t value Pr(>|t|)
## (Intercept)        39.22722    0.84962  232.30831  46.170   <2e-16 ***
## iqv_w               2.27104    0.06397  197.15338  35.500   <2e-16 ***
## iq_verb_gm          3.43997    0.30715  228.15952  11.200   <2e-16 ***
## sex_w               2.37629    0.20346 3536.35974  11.680   <2e-16 ***
## sex_gm              3.97564    1.66790  232.32685   2.384   0.0179 *
## min_w               0.12227    0.58238 3273.74604   0.210   0.8337
## minority_gm        -1.19277    1.78364  194.44191  -0.669   0.5045
## ses_w               0.16626    0.01143 3536.93552  14.545   <2e-16 ***
## ses_gm              0.04483    0.04210  196.29448   1.065   0.2882
## iqv_w:iq_verb_gm   -0.13398    0.06995  211.18343  -1.915   0.0568 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr) iqv_w  iq_vr_ sex_w  sex_gm min_w  mnrty_ ses_w  ses_gm
```

```
## iqv_w       -0.068
## iq_verb_gm  -0.049 -0.009
## sex_w        0.000  0.030  0.004
## sex_gm      -0.956 -0.013  0.020  0.000
## min_w       -0.010  0.101  0.019  0.006  0.005
## minority_gm -0.205 -0.013  0.247  0.000  0.101  0.047
## ses_w        0.006 -0.229  0.008 -0.025 -0.006  0.084 -0.002
## ses_gm       0.101 -0.001 -0.484 -0.006 -0.102 -0.008  0.070 -0.015
## iqv_w:q_vr_ -0.003  0.023 -0.227 -0.022  0.004 -0.101 -0.029 -0.017 -0.028
```

The structure of the formula makes it easy to also add either fully or partially crossed random effects. For example, in a time-series cross-sectional dataset, you might want random effects for both unit and time.

```
tscs <- import("ajpsdata.dta")
mod <- lmer(sd ~ cwarcow + iwarcow + milcontr + logpop +
    logpcgnp + demtri + (1|ccode) + (1|year), data=tscs)
summary(mod)

## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula: sd ~ cwarcow + iwarcow + milcontr + logpop + logpcgnp + demtri +
##     (1 | ccode) + (1 | year)
##    Data: tscs
##
## REML criterion at convergence: 5159.5
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.2793 -0.5385 -0.0518  0.4999  4.7932
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  ccode    (Intercept) 0.22291  0.4721
##  year     (Intercept) 0.05445  0.2334
##  Residual             0.37409  0.6116
## Number of obs: 2550, groups:  ccode, 147; year, 21
##
## Fixed effects:
##               Estimate Std. Error         df t value Pr(>|t|)
## (Intercept)    0.88832    0.46391  193.41774   1.915  0.05699 .
## cwarcow        1.08027    0.06668 2373.98862  16.200  < 2e-16 ***
## iwarcow        0.21503    0.07509 2531.99207   2.864  0.00422 **
## milcontr       0.21214    0.05035 2016.46616   4.214 2.62e-05 ***
## logpop         0.17633    0.02557  169.32743   6.897 1.01e-10 ***
## logpcgnp      -0.18733    0.02622  403.56501  -7.144 4.27e-12 ***
```

```
## demtri        -0.39198    0.04228  873.72373  -9.271  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##          (Intr) cwarcw iwarcw mlcntr logpop lgpcgn
## cwarcow   0.010
## iwarcow   0.033  0.032
## milcontr -0.099  0.056 -0.024
## logpop   -0.907 -0.061 -0.027  0.012
## logpcgnp -0.462  0.072 -0.050  0.112  0.079
## demtri    0.095  0.023  0.057  0.265 -0.008 -0.350
```

There are also functions that will do random effects models (particularly random intercepts) for ordinal and nominal data, too.

If you wanted to be Bayesian instead, you could use the `brms` package:

```
library(brms)
m3a <- brm(langpost ~ iqv_w + iq_verb_gm + sex_w + sex_gm +
     min_w + minority_gm + ses_w + ses_gm +  (1|schoolnr), data=new, chains=2)
```

Then, we can summarize the model and test between versus within effects.

```
summary(m3a)
```

```
##  Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: langpost ~ iqv_w + iq_verb_gm + sex_w + sex_gm + min_w + minority_gm + se
##    Data: new (Number of observations: 3758)
## Samples: 2 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 2000
##
## Group-Level Effects:
## ~schoolnr (Number of levels: 211)
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sd(Intercept)     2.99      0.19     2.63     3.37        621 1.00
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      39.30      0.86    37.69    40.97        677 1.00
## iqv_w           2.25      0.06     2.14     2.36       3170 1.00
## iq_verb_gm      3.47      0.30     2.84     4.05        493 1.00
## sex_w           2.39      0.20     2.00     2.78       3003 1.00
## sex_gm          3.66      1.72     0.41     6.98        675 1.00
## min_w          -0.08      0.59    -1.28     1.06       4542 1.00
## minority_gm     0.63      1.87    -3.01     4.26        770 1.01
```

```
## ses_w           0.17      0.01     0.14     0.19       2997 1.00
## ses_gm          0.07      0.04    -0.02     0.16        494 1.00
##
## Family Specific Parameters:
##       Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma     6.07      0.07     5.94     6.22       3293 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

```
hypothesis(m3a, "iqv_w = iq_verb_gm")
```

```
## Hypothesis Tests for class b:
##                 Hypothesis Estimate Est.Error CI.Lower CI.Upper Evid.Ratio
## 1 (iqv_w)-(iq_verb_gm) = 0    -1.22      0.31    -1.83     -0.6         NA
##   Post.Prob Star
## 1        NA    *
## ---
## '*': The expected value under the hypothesis lies outside the 95%-CI.
## Posterior probabilities of point hypotheses assume equal prior probabilities.
```

One of the big benefits of the `brm` function is that it fits a number of random-effect models that takes advantage of the ease of specifying these models in the Bayesian context. These include all of the usual GLMs, but also zero-inflated and hurdle poisson and NB models, multinomial and ordered dependent variables as well as exponential and Weibull models (and many others). This is a really flexible modeling strategy that can be accessed and engaged pretty simply.

3.9 Factor Analysis and SEM

One of the places where development continues rapidly is around measurement and latent variable issues. First, we'll talk about factor analysis and then move on to the SEM setting. While R has a bult-in factor analysis routine called `factanal`, there is a more flexible routine in the `psych` pacakge called `fa`.[5]

We'll start with factor analysis by using some data on political democracy from Ken Bollen.

```
library(lavaan)
data(PoliticalDemocracy)
library(psych)
fa_dem65 <- fa(PoliticalDemocracy[,c("y5", "y6", "y7", "y8")],
    SMC=TRUE, nfactors=1, fm="wls", scores="regression", cor="mixed")
```

[5]Note, that you've got access to the eigen decomposition routine with the `eigen` function and the singular value decomposition function with `svd`. There is also `princomp`, a principal components routine.

```
##
## mixed.cor is deprecated, please use mixedCor.

fa_dem65

## Factor Analysis using method =  wls
## Call: fa(r = PoliticalDemocracy[, c("y5", "y6", "y7", "y8")], nfactors = 1,
##     scores = "regression", SMC = TRUE, fm = "wls", cor = "mixed")
## Standardized loadings (pattern matrix) based upon correlation matrix
##    WLS1   h2   u2 com
## y5 0.75 0.56 0.44   1
## y6 0.79 0.62 0.38   1
## y7 0.82 0.67 0.33   1
## y8 0.89 0.80 0.20   1
##
##                WLS1
## SS loadings    2.65
## Proportion Var 0.66
##
## Mean item complexity =  1
## Test of the hypothesis that 1 factor is sufficient.
##
## The degrees of freedom for the null model are  6  and the objective function was
## The degrees of freedom for the model are 2  and the objective function was  0.09
##
## The root mean square of the residuals (RMSR) is  0.04
## The df corrected root mean square of the residuals is  0.07
##
## The harmonic number of observations is  75 with the empirical chi square  1.58  wi
## The total number of observations was  75  with Likelihood Chi Square =  6.31  with
##
## Tucker Lewis Index of factoring reliability =  0.917
## RMSEA index =  0.176  and the 90 % confidence intervals are  0.027 0.329
## BIC =  -2.32
## Fit based upon off diagonal values = 1
## Measures of factor score adequacy
##                                                 WLS1
## Correlation of (regression) scores with factors   0.95
## Multiple R square of scores with factors          0.90
## Minimum correlation of possible factor scores     0.80
```

The scores are stored as the `scores` element of the created object (in this case, `fa_dem65`). The `scores` element will be a $N \times$ #factors matrix of scores, so it can be directly included in the original data. For example:

```
fa_dem60 <- fa(PoliticalDemocracy[,c("y1", "y2", "y3", "y4")],
    SMC=TRUE, nfactors=1, fm="wls", scores="regression", cor="mixed")

##
## mixed.cor is deprecated, please use mixedCor.

fa_ind60 <- fa(PoliticalDemocracy[,c("x1", "x2", "x3")],
    SMC=TRUE, nfactors=1, fm="wls", scores="regression", cor="mixed")

##
## mixed.cor is deprecated, please use mixedCor.

PoliticalDemocracy$dem65 <- c(fa_dem65$scores)
PoliticalDemocracy$dem60 <- c(fa_dem60$scores)
PoliticalDemocracy$ind60 <- c(fa_ind60$scores)
```

The most robust and flexible package for doing SEMs in R is called `lavaan`, which has been in development for some time, but is moving toward being a one-for-one replacement for MPlus, which would be a big cost savings for lots of people. Lavaan has both CFA and SEM capabilities. The main parts of the model code are the `=~` which identifies the latent variables (LHS) and their indicators (RHS), the `~~` which frees variances or covariances to be estimates and `~`, which defines the predictive part of the models that include the latent variables. Here's an example from Bollen's book.

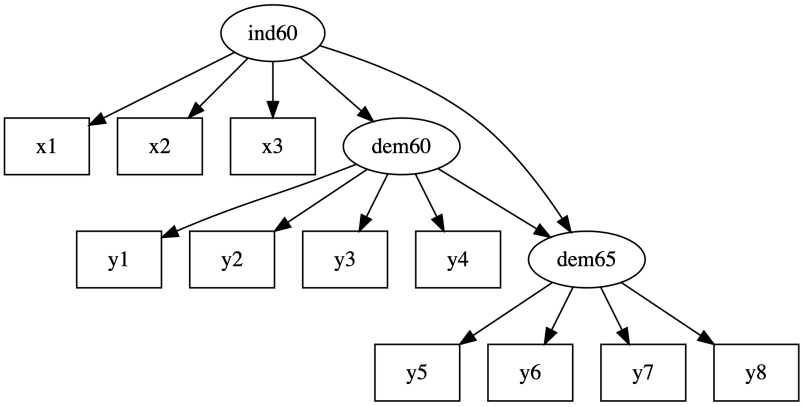
```
library(lavaan)
model <- '
  # measurement model
    ind60 =~ x1 + x2 + x3
    dem60 =~ y1 + y2 + y3 + y4
    dem65 =~ y5 + y6 + y7 + y8
  # regressions
    dem60 ~ a1*ind60
    dem65 ~ b1*ind60 + b2*dem60
  # residual correlations
    y1 ~~ y5
    y2 ~~ y4 + y6
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
'
fit <- sem(model, data=PoliticalDemocracy, mimic="MPlus")
summary(fit)

## lavaan 0.6-3 ended normally after 83 iterations
##
##   Optimization method                           NLMINB
```

```
##   Number of free parameters                         42
##
##   Number of observations                            75
##   Number of missing patterns                         1
##
##   Estimator                                         ML
##   Model Fit Test Statistic                      38.125
##   Degrees of freedom                                35
##   P-value (Chi-square)                           0.329
##
## Parameter Estimates:
##
##   Information                                 Observed
##   Observed information based on                Hessian
##   Standard Errors                             Standard
##
## Latent Variables:
##                   Estimate  Std.Err  z-value  P(>|z|)
##   ind60 =~
##     x1                1.000
##     x2                2.180    0.139   15.685    0.000
##     x3                1.819    0.152   11.949    0.000
##   dem60 =~
##     y1                1.000
##     y2                1.257    0.185    6.775    0.000
##     y3                1.058    0.148    7.131    0.000
##     y4                1.265    0.151    8.391    0.000
##   dem65 =~
##     y5                1.000
##     y6                1.186    0.171    6.920    0.000
##     y7                1.280    0.160    7.978    0.000
##     y8                1.266    0.163    7.756    0.000
##
## Regressions:
##                   Estimate  Std.Err  z-value  P(>|z|)
##   dem60 ~
##     ind60     (a1)    1.483    0.397    3.733    0.000
##   dem65 ~
##     ind60     (b1)    0.572    0.234    2.449    0.014
##     dem60     (b2)    0.837    0.099    8.476    0.000
##
## Covariances:
##                   Estimate  Std.Err  z-value  P(>|z|)
##  .y1 ~~
##    .y5                0.624    0.369    1.690    0.091
```

```
##   .y2 ~~
##     .y4               1.313    0.699    1.879    0.060
##     .y6               2.153    0.726    2.964    0.003
##   .y3 ~~
##     .y7               0.795    0.621    1.280    0.201
##   .y4 ~~
##     .y8               0.348    0.458    0.761    0.447
##   .y6 ~~
##     .y8               1.356    0.572    2.371    0.018
##
## Intercepts:
##                    Estimate  Std.Err  z-value  P(>|z|)
##     .x1               5.054    0.084   60.127    0.000
##     .x2               4.792    0.173   27.657    0.000
##     .x3               3.558    0.161   22.066    0.000
##     .y1               5.465    0.302   18.104    0.000
##     .y2               4.256    0.450    9.461    0.000
##     .y3               6.563    0.376   17.460    0.000
##     .y4               4.453    0.384   11.598    0.000
##     .y5               5.136    0.301   17.092    0.000
##     .y6               2.978    0.386    7.717    0.000
##     .y7               6.196    0.377   16.427    0.000
##     .y8               4.043    0.371   10.889    0.000
##      ind60            0.000
##     .dem60            0.000
##     .dem65            0.000
##
## Variances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##     .x1               0.082    0.020    4.136    0.000
##     .x2               0.120    0.070    1.712    0.087
##     .x3               0.467    0.089    5.233    0.000
##     .y1               1.891    0.469    4.035    0.000
##     .y2               7.373    1.346    5.479    0.000
##     .y3               5.067    0.968    5.233    0.000
##     .y4               3.148    0.756    4.165    0.000
##     .y5               2.351    0.489    4.810    0.000
##     .y6               4.954    0.895    5.532    0.000
##     .y7               3.431    0.728    4.715    0.000
##     .y8               3.254    0.707    4.603    0.000
##      ind60            0.448    0.087    5.170    0.000
##     .dem60            3.956    0.945    4.188    0.000
##     .dem65            0.172    0.220    0.783    0.434
```

You could plot out the path diagram with the `lavaanPlot()` function in the package of the same name:

```
library(lavaanPlot)
lavaanPlot(model=fit)
```



You can also use Lavaan to do mediation analysis. As you can see in the above plot, the `ind60` variable has an indirect effect through `dem60` plus a direct effect on `dem65`. By adding a couple of lines to our model, we can calculate the indirect and total effects.

```
library(lavaan)
med.model <- '
  # measurement model
    ind60 =~ x1 + x2 + x3
    dem60 =~ y1 + y2 + y3 + y4
    dem65 =~ y5 + y6 + y7 + y8
  # regressions
    dem60 ~ a1*ind60
    dem65 ~ b1*ind60 + b2*dem60
  # residual correlations
    y1 ~~ y5
    y2 ~~ y4 + y6
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
  # Indirect Effect of ind60
   ab := a1*b2
  # Total Effect of ind60
   total := (a1*b2) + b1
'
fit2 <- sem(med.model, data=PoliticalDemocracy, mimic="MPlus")
summary(fit2)

## lavaan 0.6-3 ended normally after 83 iterations
##
##   Optimization method                           NLMINB
##   Number of free parameters                         42
```

```
##
##   Number of observations                           75
##   Number of missing patterns                        1
##
##   Estimator                                        ML
##   Model Fit Test Statistic                     38.125
##   Degrees of freedom                               35
##   P-value (Chi-square)                          0.329
##
## Parameter Estimates:
##
##   Information                                 Observed
##   Observed information based on                Hessian
##   Standard Errors                             Standard
##
## Latent Variables:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   ind60 =~
##     x1                1.000
##     x2                2.180    0.139   15.685    0.000
##     x3                1.819    0.152   11.949    0.000
##   dem60 =~
##     y1                1.000
##     y2                1.257    0.185    6.775    0.000
##     y3                1.058    0.148    7.131    0.000
##     y4                1.265    0.151    8.391    0.000
##   dem65 =~
##     y5                1.000
##     y6                1.186    0.171    6.920    0.000
##     y7                1.280    0.160    7.978    0.000
##     y8                1.266    0.163    7.756    0.000
##
## Regressions:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   dem60 ~
##     ind60     (a1)    1.483    0.397    3.733    0.000
##   dem65 ~
##     ind60     (b1)    0.572    0.234    2.449    0.014
##     dem60     (b2)    0.837    0.099    8.476    0.000
##
## Covariances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##  .y1 ~~
##    .y5                0.624    0.369    1.690    0.091
##  .y2 ~~
```

```
##    .y4                1.313    0.699    1.879    0.060
##    .y6                2.153    0.726    2.964    0.003
##  .y3 ~~
##    .y7                0.795    0.621    1.280    0.201
##  .y4 ~~
##    .y8                0.348    0.458    0.761    0.447
##  .y6 ~~
##    .y8                1.356    0.572    2.371    0.018
##
## Intercepts:
##                   Estimate  Std.Err  z-value  P(>|z|)
##    .x1                5.054    0.084   60.127    0.000
##    .x2                4.792    0.173   27.657    0.000
##    .x3                3.558    0.161   22.066    0.000
##    .y1                5.465    0.302   18.104    0.000
##    .y2                4.256    0.450    9.461    0.000
##    .y3                6.563    0.376   17.460    0.000
##    .y4                4.453    0.384   11.598    0.000
##    .y5                5.136    0.301   17.092    0.000
##    .y6                2.978    0.386    7.717    0.000
##    .y7                6.196    0.377   16.427    0.000
##    .y8                4.043    0.371   10.889    0.000
##     ind60             0.000
##    .dem60             0.000
##    .dem65             0.000
##
## Variances:
##                   Estimate  Std.Err  z-value  P(>|z|)
##    .x1                0.082    0.020    4.136    0.000
##    .x2                0.120    0.070    1.712    0.087
##    .x3                0.467    0.089    5.233    0.000
##    .y1                1.891    0.469    4.035    0.000
##    .y2                7.373    1.346    5.479    0.000
##    .y3                5.067    0.968    5.233    0.000
##    .y4                3.148    0.756    4.165    0.000
##    .y5                2.351    0.489    4.810    0.000
##    .y6                4.954    0.895    5.532    0.000
##    .y7                3.431    0.728    4.715    0.000
##    .y8                3.254    0.707    4.603    0.000
##     ind60             0.448    0.087    5.170    0.000
##    .dem60             3.956    0.945    4.188    0.000
##    .dem65             0.172    0.220    0.783    0.434
##
## Defined Parameters:
##                   Estimate  Std.Err  z-value  P(>|z|)
```

```
##     ab                1.242    0.357    3.480    0.001
##     total             1.814    0.381    4.756    0.000
```

You can get a bunch of fit measures with the `fitMeasures` function

```
fitMeasures(fit)
```

```
##                 npar                 fmin                chisq
##               42.000                0.254               38.125
##                   df               pvalue       baseline.chisq
##               35.000                0.329              730.654
##          baseline.df     baseline.pvalue                  cfi
##               55.000                0.000                0.995
##                  tli                 nnfi                  rfi
##                0.993                0.993                0.918
##                  nfi                 pnfi                  ifi
##                0.948                0.603                0.996
##                  rni                 logl   unrestricted.logl
##                0.995            -1547.791            -1528.728
##                  aic                  bic               ntotal
##             3179.582             3276.916               75.000
##                 bic2                rmsea      rmsea.ci.lower
##             3144.543                0.035                0.000
##      rmsea.ci.upper        rmsea.pvalue                  rmr
##                0.092                0.611                0.256
##           rmr_nomean                 srmr        srmr_bentler
##                0.276                0.041                0.041
## srmr_bentler_nomean                 crmr         crmr_nomean
##                0.044                0.045                0.049
##           srmr_mplus   srmr_mplus_nomean                cn_05
##                0.041                0.045               98.970
##                cn_01                  gfi                 agfi
##              113.803                0.996                0.991
##                 pgfi                  mfi                 ecvi
##                0.453                0.979                1.628
```

There are also commands that will produce modification indices (`modificationindices()`).

4 Miscellaneous Statistical Stuff

There are a few other statistical matters that are often part of your statistical analysis workflow that we haven't touched on yet. The first is robust/clustered standard errors.

4.1 Heteroskedasticity Robust Standard Errors

Did you know that there are 7 kinds of heteroskedasticity consistent standard errors? They go by the names HC0, HC1, HC2, HC3, HC4, HC4m and HC5. They are all riffs

on the same basic idea - that we can use the residual (and sometimes the hat-value as a measure of leverage) to give the appropriate weight to different values in calculating the coefficient variances. In Political Science, we have often been apprised of at least HC0 and HC1 standard errors. In the social sciences more generally, robust standard errors through HC3 have been discussed. The more recent incarnations often go unexamined. All of these are available in the `sandwich` package. There is a useful function called `coeftest` that is in the `lmtest` package that will summarize the coefficients using robust variance-covariance matrices.

Using the strike volume models from above, we could look at the robust summaries.

```
library(stargazer)
library(sandwich)
library(lmtest)
mod1 <- lm(log(strike_vol + 1) ~ inflation +
    unemployment + sdlab_rep, data=strikes)
coeftest(mod1, .vcov=vcovHC, type="HC3")

##
## t test of coefficients:
##
##               Estimate Std. Error t value  Pr(>|t|)
## (Intercept)  2.4183539  0.3890087  6.2167 1.453e-09 ***
## inflation    0.1463045  0.0226996  6.4452 3.851e-10 ***
## unemployment 0.2104343  0.0311168  6.7627 5.738e-11 ***
## sdlab_rep    0.0014674  0.0076656  0.1914    0.8483
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

4.2 Clustered Standard Errors

The `multiwayvcov` package has a function that allows clustering corrections to standard errors. The function is `cluster.vcov` and allows potentially multiple clustering variables. The strikes data are time-series cross-sectional in nature and so we might want to cluster standard errors on country. We could accomplish that as follows:

```
library(multiwayvcov)
mod1 <- lm(log(strike_vol + 1) ~ inflation +
    unemployment + sdlab_rep, data=strikes)
coeftest(mod1, cluster.vcov, cluster=strikes$ccode)

##
## t test of coefficients:
##
```

```
##               Estimate Std. Error t value  Pr(>|t|)
## (Intercept)  2.4183539  1.2235211  1.9766 0.0488814 *
## inflation    0.1463045  0.0384244  3.8076 0.0001658 ***
## unemployment 0.2104343  0.0749549  2.8075 0.0052745 **
## sdlab_rep    0.0014674  0.0229305  0.0640 0.9490113
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Again, generating publication quality tables with these values would work the same way as above. You would just replace the `vcovHC` function with `cluster.vcov(mod1, strikes$ccode)`.

To cluster on two variables, you simply need to give a matrix (or data frame) rather than a vector to the `cluster` argument.

```
coeftest(mod1, cluster.vcov, cluster=strikes[,c("ccode", "year")])

##
## t test of coefficients:
##
##               Estimate Std. Error t value  Pr(>|t|)
## (Intercept)  2.4183539  1.2252916  1.9737 0.0492069 *
## inflation    0.1463045  0.0435762  3.3574 0.0008737 ***
## unemployment 0.2104343  0.0755991  2.7836 0.0056703 **
## sdlab_rep    0.0014674  0.0224213  0.0654 0.9478550
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

In either event, generating the right values is relatively easy. For those interested in more complicated TSCS designs, the `plm` package has a number of methods for estimating different TSCS models.

4.3 Weighting

Simple weighting in R is relatively easy as most modeling functions have a `weight` argument. We can use the 2012 GSS as an example:

```
gss <- import("GSS2012.dta")
umod <- lm(realinc ~ educ*age + sex, data=gss)
wmod <- lm(realinc ~ educ*age + sex, data=gss, weight=gss$wtss)
stargazer(umod, wmod, type="text")

##
## ===============================================================
##                                     Dependent variable:
##                                 ---------------------------
##                                           realinc
```

```
##                                     (1)            (2)
## ------------------------------------------------------------
## educ                             5,619.782***  5,153.598***
##                                   (935.915)     (1,044.475)
##
## age                                 77.571        -50.975
##                                   (241.850)      (280.399)
##
## sex                              -5,413.636***  -8,136.969***
##                                   (1,775.422)    (1,947.624)
##
## educ:age                            -6.525         6.448
##                                    (17.873)      (20.666)
##
## Constant                         -33,124.420**   -19,661.850
##                                   (13,016.430)  (14,514.430)
##
## ------------------------------------------------------------
## Observations                        1,711          1,711
## R2                                  0.162          0.151
## Adjusted R2                         0.160          0.149
## Residual Std. Error (df = 1706)   36,560.000    39,793.750
## F Statistic (df = 4; 1706)         82.711***      75.565***
## ============================================================
## Note:                            *p<0.1; **p<0.05; ***p<0.01
```

The weights here produce the same result as analytical weights `[aw=wtss]` in Stata. The equivalence between what's going on in Stata and R does get a bit more complicated. If you have a more compicated sampling scheme (e.g., cluster or stratified random sample), then the `survey` package gives similar functionality to the `svyset` functions in Stata (though is perhaps a bit less comprehensive). Here's a quick example with the GSS data from above just using a single weight variable with no clusters.

```
library(survey)
gss$voteob <- car::recode(gss$pres08, "1=1; 2:4=0; else=NA")
d <- svydesign(id=~1,strata=NULL, weights=~wtss, data=gss)
smod <- svyglm(voteob ~ realinc + age + sex, design=d, family=binomial)
summary(smod)

##
## Call:
## svyglm(formula = voteob ~ realinc + age + sex, design = d, family = binomial)
##
## Survey design:
## svydesign(id = ~1, strata = NULL, weights = ~wtss, data = gss)
##
```

```
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  8.785e-01  3.245e-01   2.708 0.006883 **
## realinc     -6.172e-06  1.726e-06  -3.577 0.000363 ***
## age         -4.922e-03  4.259e-03  -1.156 0.248110
## sex          3.192e-02  1.432e-01   0.223 0.823674
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 0.9482928)
##
## Number of Fisher Scoring iterations: 4

summary(wmod)

##
## Call:
## lm(formula = realinc ~ educ * age + sex, data = gss, weights = gss$wtss)
##
## Weighted Residuals:
##     Min      1Q  Median      3Q     Max
## -113034  -21948  -12339    1102  321601
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -19661.849  14514.433  -1.355    0.176
## educ          5153.598   1044.475   4.934 8.83e-07 ***
## age            -50.975    280.399  -0.182    0.856
## sex          -8136.969   1947.624  -4.178 3.09e-05 ***
## educ:age         6.448     20.666   0.312    0.755
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 39790 on 1706 degrees of freedom
##   (205 observations deleted due to missingness)
## Multiple R-squared:  0.1505,Adjusted R-squared:  0.1485
## F-statistic: 75.56 on 4 and 1706 DF,  p-value: < 2.2e-16
```

You'll notice here that the the standard errors are different. Using the survey package gives you a result that is equivalent to using `[pw=wtss]` in Stata. So, you should be able to produce results like thos you want with one or the other method in R.

If you did a weighted cross-tab with the `xtabs` function, what you would find is that these results correspond to the ones you would generate if you used importance weights, `[iw=wtss]`.

```
xtabs(wtss ~ pres08 + sex, data=gss)

##       sex
## pres08          1          2
##      1 306.064833 394.434634
##      2 199.887618 246.103177
##      3  18.474857   6.590370
##      4   3.857657   3.757275
```

The `weights` package has a number of simple inferential functions (correlations, t-test, χ^2) for weighted data. Here are a few examples:

```
library(weights)
library(dplyr)
xf <- filter(gss, sex == 2)
yf <- filter(gss, sex == 1)
x <- xf$realinc
wtx <- xf$wtss
y <- yf$realinc
wty <- yf$wtss
wtd.t.test(x, y, wtx, wty)

## $test
## [1] "Two Sample Weighted T-Test (Welch)"
##
## $coefficients
##       t.value            df       p.value
## -3.763769e+00  1.508738e+03  1.737870e-04
##
## $additional
## Difference     Mean.x     Mean.y   Std. Err
##  -8100.843  36120.827  44221.669   2152.322
```

Here's another example:

```
wtd.cor(log(gss$age), log(gss$realinc), gss$wtss)

##   correlation    std.err  t.value      p.value
## Y   0.1005515 0.02406702 4.177978 3.089603e-05
```

and a χ^2 example:

```
wtd.chi.sq(gss$pres08, gss$sex)

##     Chisq        df   p.value
## 5.0215284 3.0000000 0.1702275
```

There are also lots of functions available in the `survey` package for doing bivariate and multiple relationships with weighted data.

5 Finding Packages on CRAN

R's open source nature means that active development on new and improved R functions is going on all over the world. This means that new functions are being written all of the time and a comprehensive list of R's capabilities would be ephemeral at best. There are currently 5662 packages available from CRAN (and others available from other repositories) each consisting of many functions. While this has many benefits for both individual work and science more generally, it does making finding functions a bit more difficult. There are many resources available, though.

RSeek (rseek.org) is a website that searches R's help files (including those of the add on packages), Support forums, books, blogs and google (all under separate headings) to find what you're looking for.

Crantastic crantastic.org is a website that organizes R packages by tags and permits user reviews. The search facilities here seem a bit less powerful than RSeek.

?? The `??` character (or the function `help.search()`) will search *local* help files for for a particular word.

Packages from CRAN can be installed with `install.packages('package name')`.[6] Generally, packages are built as binary files, meaning they have already been compiled and they are ready to install directly. Binaries for Windows cannot be installed on Macs and vice versa. Packages are also distributed as source code that can be compiled directly on your machine. There is often little need for this unless packages are available in an older version, but not for your current version. You could build the package directly yourself.

- If when you try to install a package from CRAN, it tells you that it doesn't exist for your version, try the following: `install.packages('package name', type='source')`. This will download the source package and try to compile it on your machine.

When you try to install a package from CRAN, you will have to choose a CRAN mirror (i.e., a website from which to download the package). The advice is generally to use one that is geographically close, but there is also a cloud-based option which is the first on the list. This is as good as any to use.

Development happens in other places than on CRAN. Technically, the people who maintain CRAN only want developers to upload new versions a few times a year. Development happens in other places like R-forge and github. One place to search for new or development versions of packages is on R-forge (r-forge.r-project.org). To install packages from R-forge, you could do:
`install.packages('package name', repos='r-forge.r-project.org)`.

Git is a popular source code management (SCM) system and many using Git have projects on Github, a web-based hosting service for that uses Git for version control. Packages can be installed from github using the `devtools` package.

[6] Singe and double quotes are both acceptable in R as quotation characters, but not mixed up. That is, if you open with a double quote, you also have to close with a double quote.

```
library(devtools)
install_github('davidaarmstrong/damisc')
```

Once packages have been installed, (with some version of `install.packages` or `install_github`), they need to be loaded so you can have the required functionality with `library('package name')`. The `install.packages` or equivalent function only needs to be run once for each version of R. Once you've downloaded and installed the package, it doesn't need to be installed again. However, the `library('package name')` statement has to be issued in each session (i.e., each time you turn R on).

There are ways to have R automatically load packages every time you open the software. To do this, follow these steps.

1. Open a file in whichever R editor you are using (R's internal or RStudio).
2. Save the file as `.Rprofile` in your home directory (i.e., R's default working directory).
3. In that file, put the following:

   ```
   dp <- getOption("defaultPackages")
   options(defaultPackages=c(dp, "foreign", "MASS", "DAMisc"))
   ```

   and then save the file.

Now, the next time you open R the packages you specified will be automatically loaded.

6 Warnings and Errors

Most of the time, if R provides no output than it means you did something right. R will generally offer two kinds of "you did something wrong" messages.

- Warnings - these messages suggest that things may not have proceeded exactly as expected, but output was still produced. Unless you are familiar with the warning and the circumstances under which it is produced, you should look at the results to ensure they make sense.

```
mat <- matrix(c(
   100, 25, 3, 11), ncol=2, byrow=T)
chisq.test(mat)
```

```
Pearson's Chi-squared test with Yates' continuity correction
```

```
data:  mat
X-squared = 19.5568, df = 1, p-value = 9.765e-06

Warning message:
In chisq.test(mat) : Chi-squared approximation may be incorrect
```

Note, the warning suggests the χ^2 approximation may be incorrect (it doesn't say why, but it is because there are expected counts of less than five). The output was produced, but R is cautioning you to ensure that you take a second look at your results to make sure they are what you want.

- Errors - these are messages that indicate R was unable to perform the task you asked. Generally, no output beyond the error message is produced. Some error messages can be helpful and others, not so much. It depends on how the function was programmed (developers write their own warning/error messages). For example, what if I try to correlate a numeric vector with a vector of words:

```
x <- c(1,2,3,4,5)
y <- c("a", "b", "c", "d", "e")
cor(x,y)
```

```
> cor(x,y)
Error in cor(x, y) : 'y' must be numeric
```

Note here that R is telling you that `y` has to be numeric and until you make it numeric, R can't produce any output for the function.

7 Troubleshooting

In the example above with `chisq.test`, you may wonder why it gives you that error. You could look in the help file, but that might not provide the relevant information (in this case, it does not). You could then look at the code. If you type `chisq.test` at the command prompt (without the parenthesesS) and hit enter, R will provide you with the source code. You could then search for your warning and you would find the following.

```
names(PARAMETER) <- "df"
if (any(E < 5) && is.finite(PARAMETER))
   warning("Chi-squared approximation may be incorrect")
```

The term `PARAMETER` is the degrees of freedom and `E` is the expected count. We know this either by looking through the code or just from the context. So, the warning message is produced in the case where any expected values are less than five and the degrees of freedom is finite. Presumably, we have a sense of what the theoretical problems are and could remedy them. By using a (Monte Carlo) simulated p-value, the warning disappears. The reason it goes away is that when you simulate the p-value, the function sets `PARAMETER` to `NA` (missing), so it fails the `is.finite(PARAMETER)` part of the condition to trigger the warning.

```
chisq.test(mat, simulate.p.value=TRUE)

##
##  Pearson's Chi-squared test with simulated p-value (based on 2000
##  replicates)
##
## data:  mat
## X-squared = 22.505, df = NA, p-value = 0.0004998
```

Ultimately, the code is the definitive source for what the function is doing and when. Unfortunately, for many of us, myself included at times, the code is not always that illuminating. Sometimes, typing the function at the command line and hitting enter will not provide you with all of the code. For example,

```
mean

## standardGeneric for "mean" defined from package "base"
##
## function (x, ...)
## standardGeneric("mean")
## <environment: 0x7ffc605aaf28>
## Methods may be defined for arguments: x
## Use  showMethods("mean")  for currently available ones.
```

Here, the "guts" of the function are `UseMethod(mean)`. This suggests that `mean` is a generic function and works differently on different objects. If you want to see what kinds of objects the `mean` function works on, you could do:

```
methods("mean")

##  [1] mean,ANY-method          mean,Matrix-method
##  [3] mean,Raster-method       mean,sparseMatrix-method
##  [5] mean,sparseVector-method mean.Date
##  [7] mean.default             mean.difftime
##  [9] mean.IDate*              mean.mlogit*
## [11] mean.mlogit.data*        mean.POSIXct
## [13] mean.POSIXlt             mean.quosure*
## [15] mean.rpar*               mean.yearmon*
## [17] mean.yearqtr*            mean.zoo*
## see '?methods' for accessing help and source code
```

Since we generally want the mean of an object of class `numeric`, we would look for `mean.numeric`. Since it's not there, we would look at `mean.default`.

```
mean.default
```

```
## function (x, trim = 0, na.rm = FALSE, ...)
## {
##     if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
##         warning("argument is not numeric or logical: returning NA")
##         return(NA_real_)
##     }
##     if (na.rm)
##         x <- x[!is.na(x)]
##     if (!is.numeric(trim) || length(trim) != 1L)
##         stop("'trim' must be numeric of length one")
##     n <- length(x)
##     if (trim > 0 && n) {
##         if (is.complex(x))
##             stop("trimmed means are not defined for complex data")
##         if (anyNA(x))
##             return(NA_real_)
##         if (trim >= 0.5)
##             return(stats::median(x, na.rm = FALSE))
##         lo <- floor(n * trim) + 1
##         hi <- n + 1 - lo
##         x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
##     }
##     .Internal(mean(x))
## }
## <bytecode: 0x7ffc47dba018>
## <environment: namespace:base>
```

It is also possible that some code you want to see is "hidden". The reasons for this are technical and not that interesting, so we won't go into them. Let's imagine you loaded the `car` package with `library(car)` and wanted to see if there was any method for plotting a data frame (e.g., `Duncan`).

```
methods("plot")
```

```
##   [1] plot,ANY,ANY-method
##   [2] plot,ANY,brob-method
##   [3] plot,ANY,glub-method
##   [4] plot,brob,ANY-method
##   [5] plot,brob,missing-method
##   [6] plot,color,ANY-method
##   [7] plot,Extent,missing-method
##   [8] plot,glub,ANY-method
##   [9] plot,glub,missing-method
##  [10] plot,parboot,missing-method
```

```
##  [11] plot,profile,missing-method
##  [12] plot,profile.mle,missing-method
##  [13] plot,pvgam,ANY-method
##  [14] plot,qrrvglm,ANY-method
##  [15] plot,Raster,ANY-method
##  [16] plot,Raster,Raster-method
##  [17] plot,rcim,ANY-method
##  [18] plot,rcim0,ANY-method
##  [19] plot,rrvgam,ANY-method
##  [20] plot,Spatial,missing-method
##  [21] plot,SpatialGrid,missing-method
##  [22] plot,SpatialGridDataFrame,missing-method
##  [23] plot,SpatialLines,missing-method
##  [24] plot,SpatialMultiPoints,missing-method
##  [25] plot,SpatialPixels,missing-method
##  [26] plot,SpatialPixelsDataFrame,missing-method
##  [27] plot,SpatialPoints,missing-method
##  [28] plot,SpatialPolygons,missing-method
##  [29] plot,stanfit,missing-method
##  [30] plot,timeDate,ANY-method
##  [31] plot,timeSeries,ANY-method
##  [32] plot,unmarkedFit,missing-method
##  [33] plot,unmarkedFitOccuMulti,missing-method
##  [34] plot,unmarkedFrame,missing-method
##  [35] plot,unmarkedFrameOccuMulti,missing-method
##  [36] plot,unmarkedRanef,missing-method
##  [37] plot,vgam,ANY-method
##  [38] plot,vglm,ANY-method
##  [39] plot,vlm,ANY-method
##  [40] plot,vsmooth.spline,ANY-method
##  [41] plot.aareg*
##  [42] plot.acf*
##  [43] plot.ACF*
##  [44] plot.agnes*
##  [45] plot.areg*
##  [46] plot.areg.boot*
##  [47] plot.aregImpute*
##  [48] plot.augPred*
##  [49] plot.bagplot
##  [50] plot.balsos*
##  [51] plot.bayesplot_grid*
##  [52] plot.bayesplot_scheme*
##  [53] plot.bclust*
##  [54] plot.biVar*
##  [55] plot.boot*
```

```
##  [56] plot.brmsfit*
##  [57] plot.brmshypothesis*
##  [58] plot.brmsMarginalEffects*
##  [59] plot.cld*
##  [60] plot.clusGap*
##  [61] plot.coef.mer*
##  [62] plot.cohesiveBlocks*
##  [63] plot.communities*
##  [64] plot.compareFits*
##  [65] plot.Conf
##  [66] plot.confint.glht*
##  [67] plot.correspondence*
##  [68] plot.cox.zph*
##  [69] plot.cuminc*
##  [70] plot.curveRep*
##  [71] plot.data.frame*
##  [72] plot.decomposed.ts*
##  [73] plot.default
##  [74] plot.dendrogram*
##  [75] plot.density*
##  [76] plot.Desc*
##  [77] plot.describe*
##  [78] plot.diana*
##  [79] plot.diffci*
##  [80] plot.drawPlot*
##  [81] plot.ecdf
##  [82] plot.eff*
##  [83] plot.efflist*
##  [84] plot.effpoly*
##  [85] plot.factor*
##  [86] plot.factorplot*
##  [87] plot.fANCOVA*
##  [88] plot.formula*
##  [89] plot.function
##  [90] plot.gam*
##  [91] plot.gbayes*
##  [92] plot.ggplot*
##  [93] plot.glht*
##  [94] plot.gls*
##  [95] plot.goodfit*
##  [96] plot.gtable*
##  [97] plot.hcl_palettes*
##  [98] plot.hclust*
##  [99] plot.histogram*
## [100] plot.HoltWinters*
```

```
## [101] plot.ica*
## [102] plot.ideal*
## [103] plot.igraph*
## [104] plot.InformativeTesting*
## [105] plot.intervals.lmList*
## [106] plot.irt
## [107] plot.isoreg*
## [108] plot.jam*
## [109] plot.Lc
## [110] plot.Lclist
## [111] plot.lda*
## [112] plot.limdil*
## [113] plot.lm*
## [114] plot.lme*
## [115] plot.lmList*
## [116] plot.lmList4*
## [117] plot.lmList4.confint*
## [118] plot.loddsratio*
## [119] plot.loess
## [120] plot.loglm*
## [121] plot.loo*
## [122] plot.ls_means*
## [123] plot.mca*
## [124] plot.mcmc*
## [125] plot.mcmc.list*
## [126] plot.medpolish*
## [127] plot.merMod*
## [128] plot.mids*
## [129] plot.mitml*
## [130] plot.mlm*
## [131] plot.mlm.efflist*
## [132] plot.mlogit*
## [133] plot.mona*
## [134] plot.MPP*
## [135] plot.nffGroupedData*
## [136] plot.nfnGroupedData*
## [137] plot.nls*
## [138] plot.nmGroupedData*
## [139] plot.OddsRatio*
## [140] plot.opscale*
## [141] plot.palette
## [142] plot.partition*
## [143] plot.pdMat*
## [144] plot.poly
## [145] plot.poly.parallel
```

```
## [146] plot.PostHocTest
## [147] plot.PP*
## [148] plot.ppr*
## [149] plot.prcomp*
## [150] plot.predict.crr*
## [151] plot.predict.ideal*
## [152] plot.predictoreff*
## [153] plot.predictorefflist*
## [154] plot.predProbs*
## [155] plot.preplot.predProbs*
## [156] plot.princomp*
## [157] plot.profile*
## [158] plot.profile.clm*
## [159] plot.profile.clm2*
## [160] plot.profile.clmm2*
## [161] plot.profile.game*
## [162] plot.profile.nls*
## [163] plot.psis*
## [164] plot.psis_loo*
## [165] plot.psych
## [166] plot.Quantile2*
## [167] plot.R6*
## [168] plot.ranef.lme*
## [169] plot.ranef.lmList*
## [170] plot.ranef.mer*
## [171] plot.raster*
## [172] plot.replot_xts*
## [173] plot.residuals
## [174] plot.ridgelm*
## [175] plot.rm.boot*
## [176] plot.rpar*
## [177] plot.rpart*
## [178] plot.rrvgam*
## [179] plot.seatsVotes*
## [180] plot.sePP*
## [181] plot.shingle*
## [182] plot.silhouette*
## [183] plot.simulate.lme*
## [184] plot.sir*
## [185] plot.slice.clm*
## [186] plot.SOM*
## [187] plot.somgrid*
## [188] plot.spec*
## [189] plot.spline*
## [190] plot.step_list*
```

```
## [191] plot.stepfun
## [192] plot.stft*
## [193] plot.stl*
## [194] plot.structable*
## [195] plot.summary.formula.response*
## [196] plot.summary.formula.reverse*
## [197] plot.summaryM*
## [198] plot.summaryP*
## [199] plot.summaryS*
## [200] plot.Surv*
## [201] plot.surv_cutpoint*
## [202] plot.survfit*
## [203] plot.svm*
## [204] plot.svrepstat*
## [205] plot.svyby*
## [206] plot.svycdf*
## [207] plot.svykm*
## [208] plot.svykmlist*
## [209] plot.svysmooth*
## [210] plot.svystat*
## [211] plot.table*
## [212] plot.timeSeries*
## [213] plot.TMod*
## [214] plot.transcan*
## [215] plot.trellis*
## [216] plot.ts
## [217] plot.tskernel*
## [218] plot.TukeyHSD*
## [219] plot.tune*
## [220] plot.varclus*
## [221] plot.Variogram*
## [222] plot.vgam*
## [223] plot.visreg*
## [224] plot.visreg2d*
## [225] plot.visregList*
## [226] plot.xts*
## [227] plot.xyVector*
## [228] plot.zoo
## see '?methods' for accessing help and source code
```

What you see is that there is a `plot.data.frame`, but it has an asterisk which indicates that it is "Non-visible". So, if you type `plot.data.frame` at the command line and hit enter, R will tell you that it cannot find `plot.data.frame`. To see the code for invisible functions, you could look at:

```
getAnywhere(plot.data.frame)

## A single object matching 'plot.data.frame' was found
## It was found in the following places
##   registered S3 method for plot from namespace graphics
##   namespace:graphics
## with value
##
## function (x, ...)
## {
##     plot2 <- function(x, xlab = names(x)[1L], ylab = names(x)[2L],
##         ...) plot(x[[1L]], x[[2L]], xlab = xlab, ylab = ylab,
##         ...)
##     if (!is.data.frame(x))
##         stop("'plot.data.frame' applied to non data frame")
##     if (ncol(x) == 1) {
##         x1 <- x[[1L]]
##         if (class(x1)[1L] %in% c("integer", "numeric"))
##             stripchart(x1, ...)
##         else plot(x1, ...)
##     }
##     else if (ncol(x) == 2) {
##         plot2(x, ...)
##     }
##     else {
##         pairs(data.matrix(x), ...)
##     }
## }
## <bytecode: 0x7ffc3a39d520>
## <environment: namespace:graphics>
```

8 Help!

There are lots of ways to get help for R. First, let me suggest a couple of books.

8.1 Books

- Kabacoff, Robert. 2014. *R In Action,* 2^{nd} *ed.* Manning.
- Fox, John and Sanford Weisberg. 2011. *An R Companion to Applied Regression,* 2^{nd} *ed.* Sage.
- Monogan, James. 2015. *Political Analysis Using R.* Springer (forthcoming this Fall).

Both are wonderful books. Kabacoff's is more of a "from scratch" book, providing some detail about the basics that John's book doesn't. Kabacoff also has a website called Quick R http://www.statmethods.net/ that has some nice examples and code that could prove useful. John's has some introductory material, but is a bit more focused on regression than Kabacoff's.

8.2 Web

There are also lots of internet resources.

Stack Overflow The `r` tag at stack overflow refers to questions relating to R (http://stackoverflow.com/questions/tagged/r). In the top-right corner of the page, there is a search bar that will allow you to search through "r"-tagged questions. So, here you can see if your question has already been asked and answered or post a new question if it hasn't.

Rseek We talked about http://www.rseek.org for finding packages, but it is also useful for getting help if you need it.

RSiteSearch This can be invoked from wihin R as `RSiteSearch('linear')`.

UCLA IDRE UCLA's Institute for Digital Research and Education (http://www.ats.ucla.edu/stat/r/) has some nice tools for learning R, too.

R Mailing List R has a number of targeted mailing lists (along with a general help list) that are intended to allow users to ask questions (http://www.r-project.org/mail.html). There are links to instructions and a posting guide which you should follow to the best of your ability. Failure to follow these guidelines will likely result in you being excoriated by the people on the list. People who answer the questions are doing so on their free time, so make it as easy as possible for them to do that. In general, it is good practice if you're asking someone to help you that you provide them with the means to reproduce the problem.

9 Brief Primer on Good Graphics

While we could define graphs in lots of different ways, Kosslyn (1994, 2) defines graphs as:

"a visual display that illustrates one or more relationships among numbers"

He goes on to define a graph as "successful" if:

"the pattern,trend or comparison it presents can be immediately apprehended."

Graphs work by encoding quantitative and qualitative information with different graphical elements - points (plotting symbols), lines (patterns), colors, etc...

- Graphical perception is the visual decoding of this encoded information

- For our graphs to be successful, readers must be able to easily and efficiently decoded the quantitative and qualitative information.

Shah (2002) identifies three components to the process of visual decoding

1. Identify important features of the display (what Cleveland calls "Detection") → Characteristics of the display
2. Relate the visual feature to the phenomenon it represents in the observed world → Knowledge of graphs
3. Interpret the relationships interps of concepts being quantified → Knowledge of content (i.e., concepts being discussed).

9.1 Graphical Perception

We can use lots of different elements to encode numeric information

- Angle
- Area
- Color hue
- Color saturation
- Density
- Length (distance)
- Position along a common scale
- Position along identical, non-aligned scales
- Slope
- Volume

Others

- Volume is used rarely to encode quantitative information in scientific graphs.
- Color hue (i.e., different colors) are often used to encode categorical (not quantitative) information. Cleveland does not discuss this much (book published in 1985).
- Color saturation (i.e., the intensity of the color) can be used to encode quantitative information.
  - Be careful about making graphs color-blind friendly
  - Realize that many graphs will may appear in color on the screen, but may not be printed in color
  - See http://colorbrewer2.org for more advice

Further, our visual processing system is drawn to make comparisons within rather than between groups

- When observations are clustered together, our minds tend to think they belong to the same group.

- Groups should be formed such that the intended comparisons are within, rather than between groups.

Cleveland identified a set of elementary perception tasks and the ordering (below from easiest to hardest) of their difficulty.

1. Position along a common scale
2. Position along identical, nonaligned scale
3. Length
4. Angle - Slope
5. Area
6. Volume
7. Color hue, color saturation, density

9.2 Advice

Plotting Symbols:

- Use open plotting symbols (rather than filled in)

Legends:

- Legends should be comprehensive and informative - should make the graph stand alone.
- Legends should:
  - Describe everything that is graphed
  - Draw attention to important aspects
  - Describe conclusions.

  Readers should not be left "guessing" about what the graphical elements mean.

Error Bars:

Error bars (or regions) should be clearly labeled so as to convey their meaning. Common error bars are:

- $\pm$ 1 SD.
- $\pm$ 1 SE
- 95% Confidence Interval.

Axes (scales):

- Choose tick marks to include all (or nearly all) of the range of the data
- Choose scales so that the data fill up as much of the data region as possible
- Sometimes two different scales can be more informative
- Axes should be labeled (including units where applicable).
- Use appropriate (i.e., the same) scales when comparing.
- Do not feel obliged to force scale lines to include zero.
- Use data transformations (e.g., logs) to improve resolution.

Aspect Ratio:

Cleveland found through experimentation, that 45° was the angle at which we can best discern differences in slope. Thus, he developed a technique called "banking to 45°" that finds the optimal aspect ratio for graphs.

- If you're using `lattice`, then using the option `aspect= 'xy'` will do the banking for you.
- If you're using some different graphing tool (like R's traditional graphics engine), then setting the aspect ratio to one will come close enough, usually.

10 Graphics Philosophies

R has four different graphics systems - base, grid, lattice and ggplot2, all with different philosophies about how graphs are to be made.

base The base system works like painting - you can sequentially add elements to a graph, but it is quite hard to take elements away (in fact, it is impossible). Layers can be added until the graph conveys just what the user wants.

lattice The lattice system is based on Cleveland's Trellis system developed at Bell Labs and is built on top of the grid graphics system. These are particularly good for grouped data where multi-panel displays are needed/desired. These operate more like setting up a track of dominoes when you were a kid. You line them all up and then knock the first one down and all the others fall as you've arranged them. If you messed up the arrangement, it isn't going to be as cool/pretty/interesting as you thought. Lattice graphs work the same way. All elements of the graph must be specified in a single call to the graphics command and if you don't do it right, it will not be as cool/pretty/interesting as you want.

ggplot2 Hadley Wickham wrote ggplot2 and describes it in his 2009 book as an implementation of Leeland Wilkinson's "Grammar of graphics". This builds a comprehensive grammar- (read model-) based system for generating graphs.

grid The grid system was written by Paul Murrell and is described in his 2006 book, which is now in its second edition. This system, while very flexible, is not something that users often interact with. We are much better off interacting with grid through lattice.

The base graphics system, lattice and grid are all downloaded automatically when you download R. ggplot2 must be downloaded separately and all but the base system must be loaded when you want to use them with the `library()` command.

As you can imagine, these different systems offer quite different solutions to creating high quality graphics. Depending on what exactly you're trying to do, some things are more difficult, or impossible, in one of these systems, but not in the other. We will certainly see examples of this as the course progresses. I have had a change of heart about these various systems recently. Originally, I did everything I possibly could in the traditional graphics system and then only later moved to the lattice system. This probably makes sense as the lattice system requires considerably more programming, but does lots more neat stuff. More recently still, I have moved much of my plotting to ggplot because I think that is where the discipline is headed, in general. If we have to focus on one advanced graphics platform, ggplot is probably the best one to focus on.

We are going to start by talking about the traditional graphics system. We will spend some time later talking about what other kinds of graphs can do and why we might want to use them, but for now, it's just the traditional graphics.

11 The Plot Function

For the most part, in the traditional graphics system, graphs are initially made with the `plot` function, though there are others, too. Then additional elements can be added as you see fit.

11.1 getting familiar with the function

Let's take a look at the help file for the plot command.

```
     x: the coordinates of points in the plot. Alternatively, a
        single plotting structure, function or _any R object with a
        'plot' method_ can be provided.

     y: the y coordinates of points in the plot, _optional_ if 'x' is
        an appropriate structure.

   ...: Arguments to be passed to methods, such as graphical
        parameters (see 'par'). Many methods will accept the
        following arguments:
```

```
'type' what type of plot should be drawn.  Possible types are

        *  '"p"' for *p*oints,

        *  '"l"' for *l*ines,

        *  '"b"' for *b*oth,

        *  '"c"' for the lines part alone of '"b"',

        *  '"o"' for both ë*o*verplottedí,

        *  '"h"' for ë*h*istogramí like (or ëhigh-densityí)
           vertical lines,

        *  '"s"' for stair *s*teps,

        *  '"S"' for other *s*teps, see ëDetailsí below,

        *  '"n"' for no plotting.

     All other 'type's give a warning or an error; using, e.g.,
     'type = "punkte"' being equivalent to 'type = "p"' for S
     compatibility.

'main' an overall title for the plot: see 'title'.

'sub' a sub title for the plot: see 'title'.

'xlab' a title for the x axis: see 'title'.

'ylab' a title for the y axis: see 'title'.

'asp' the y/x aspect ratio, see 'plot.window'.
```

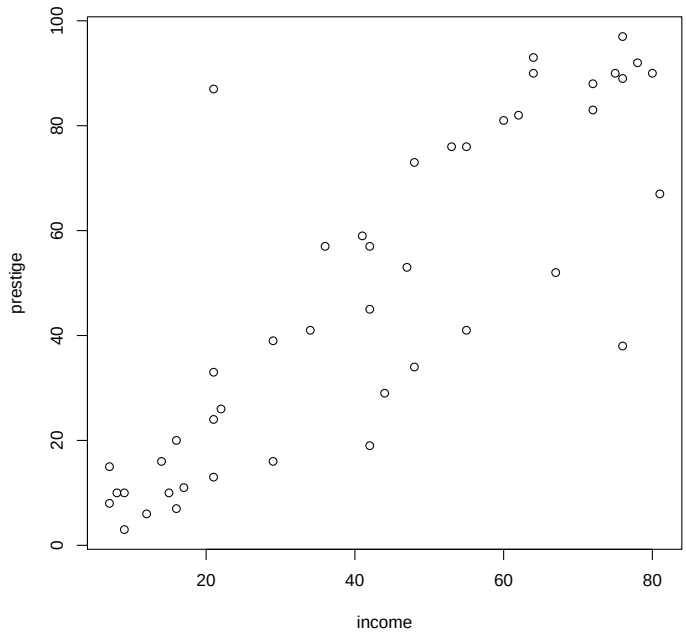
Now, we can load the Duncan data again and see how the plotting function works. The two following commands produce the same output within the plotting region, but have different axis labels.

```
library(car)
plot(prestige ~ income, data=Duncan)
```

```
plot(Duncan$income, Duncan$prestige)
```

There are a couple of interesting features here.

Figure 2: Scatterplot of Income and Prestige from the Duncan data



- You can either specify the plot with a formula as the first argument, `y ~ x` or with `x,y` as the first two arguments. The `data` argument only exists when you use the former method. Using the latter method, you have to provide the data in `dataset$variable` format, unless the data are attached.
- Whatever names you have for `x` and `y` will be printed as the x- and y-labels. These can be controlled with the `xlab` and `ylab` commands.
- The plotting symbols, by default are open circles. This can also be controlled with the `pch` option (more on this later) .

11.2 Default Plotting Methods

In **R**, there are default methods for plotting all types of variables. By *default method* I simply mean that **R** looks at the context in which you're asking for a graph and then makes what it thinks is a reasonable graph given the different types of data. All of the figures in 3 were called simply by using the `plot` command. **R** figured out by the types of variables being used what plot was most appropriate.

```
plot(Duncan$income)
```

```
plot(Duncan$type)
```

```
plot(income ~ education, data=Duncan)
```

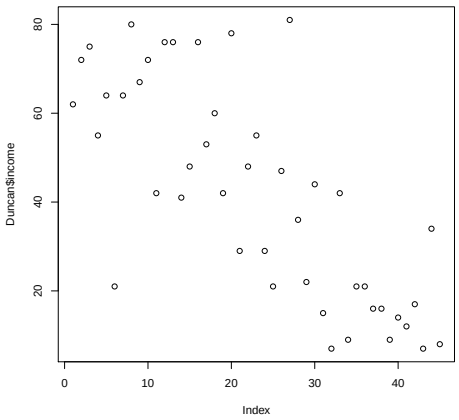
```
Duncan$inc.cat <- cut(Duncan$income, 3)
par(las=2, mar=c(5,7,2,3))
plot(inc.cat ~ type, data=Duncan, xlab="", ylab="")
```

```
plot(income ~ type, data= Duncan, xlab="")
```

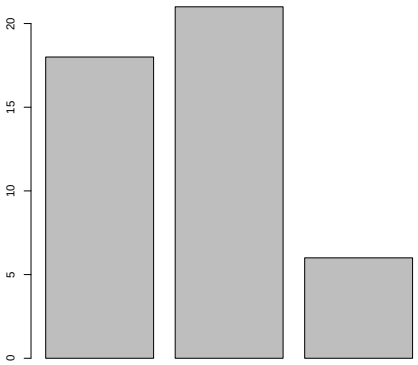
You try it

1. Load the R data file `strikes_small.rda` that was in your zip file.
2. Look through the variables by using `str` and `searchVarLabels(strikes, '')` (make sure the DAMisc package is loaded to use `searchVarLabels`). Using the data, make examples of all the plots above.

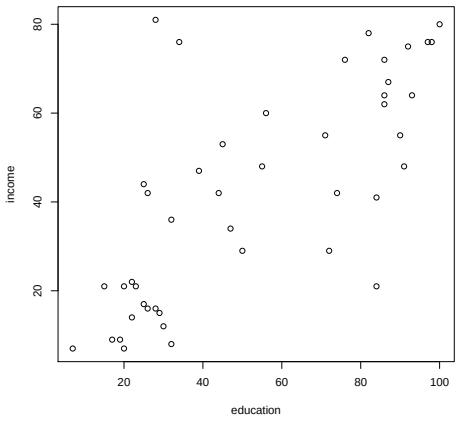
Figure 3: Default Plotting Methods



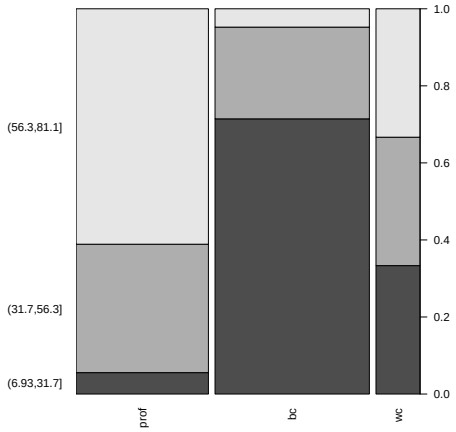
(a) One Numeric - scatterplot (with `index` as the x-axis)



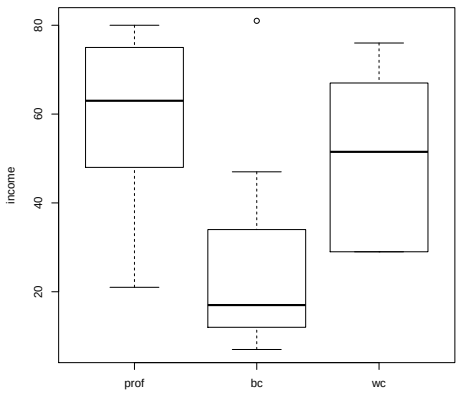
(b) One factor - histogram



(c) Two Numeric Variables - scatterplot



(d) Two Factors - mosaic plot



(e) One Numeric, One Factor - boxplot

11.3 Controlling the Plotting Region

There are a number of commands we can use to control the size and features of the plotting region.

- We can control the limits of the x- and y-axis with `xlim` and `ylim`, respectively. Here, the limits are specified with a vector of indicating the desired minimum and maximum value of the axis.

```
plot(prestige ~ income, data=Duncan, xlim=c(0,100))
```

11.4 Example of Building a Scatterplot

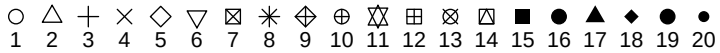
In **R**, you can open a plotting window and set its dimensions without actually making the points appear in the space. You can do this by specifying `type='n'`. You can also remove the axes from the space, by issuing the command `axes=F`. Finally, you could remove any axis labels by adding the commands `xlab='', ylab=''`. The command, then, would look something like this:

```
plot(prestige ~ income, data=Duncan, type="n", xlab='', ylab='', axes=F)
```

This will open a graphics window that is completely blank. One reason that you may want to do this is to be able to control, more precisely, the elements of the graph. Let's talk about adding some elements back in.

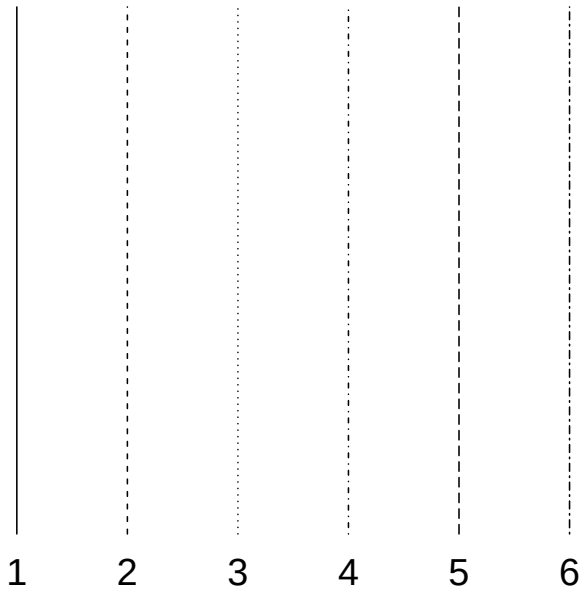
We can add the points by using the `points` command. You can see the arguments to the points command by typing `help(points)`. The first two arguments to the command are `x` and `y`. You can also change the plotting symbol and the color.

- Plotting Symbols are governed by the `pch` argument. Below is a description of some common plotting symbols.



- Color - Colors are controlled by the `col` command. (see help for `rainbow`, `colors`). The colors can be specified by `col = 'black'` or `col = 'red'`. Colors can also be specified with `rgb()` for red-green-blue saturation with optional transparency parameter.

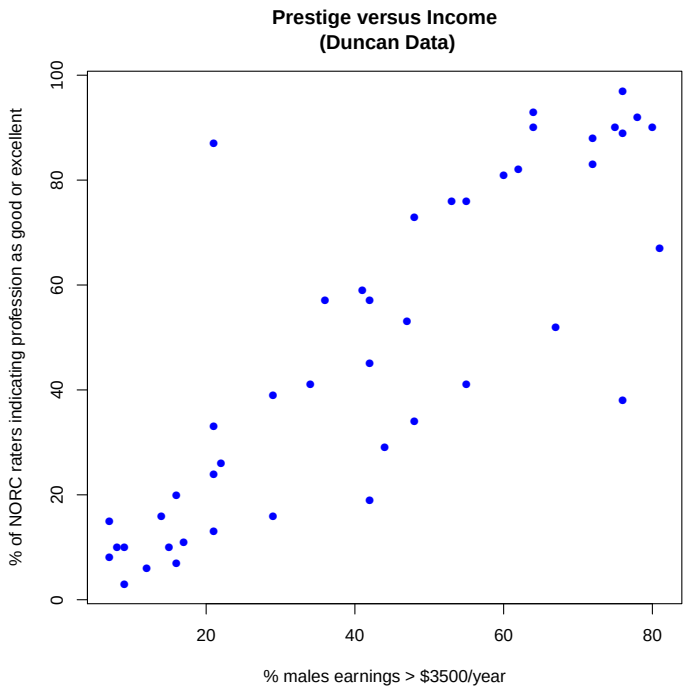
- Lines - there are six different line-types which should be specified with the corresponding number:



- Line-width, controlled with `lwd` defaults to 1, so numbers bigger than 1 are thicker than the default and numbers smaller than 1 are thinner.
- Character Expansion is controlled by `cex`. This controls the size of the plotting symbols. The default is 1. Numbers in the range (0,1) make plotting symbols smaller than the default and values > 1 make the plotting symbols bigger than they would be otherwise.

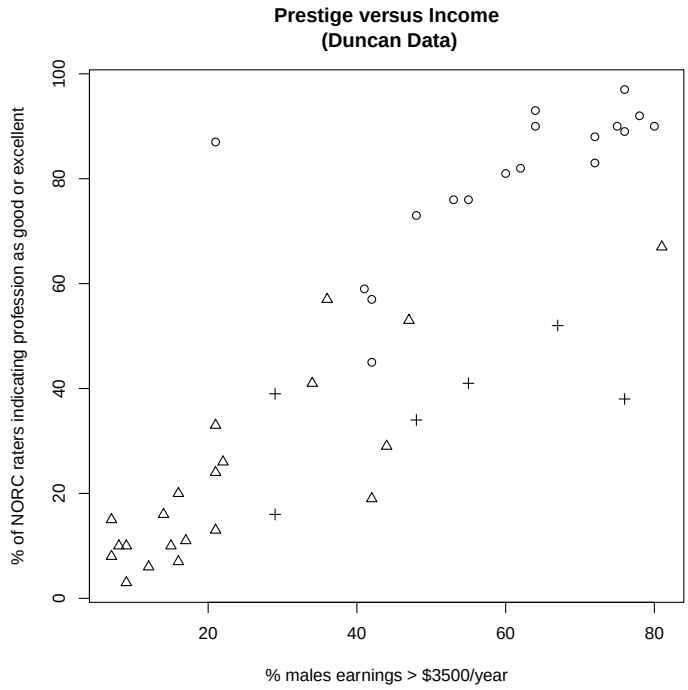
There are lots of interesting things we can do with these. Let's start by changing the plotting character and color of points.

```
plot(prestige ~ income, data=Duncan, pch=16, col="blue",
    xlab='% males earnings > $3500/year',
        ylab='% of NORC raters indicating profession as good or excellent',
        main = 'Prestige versus Income\n (Duncan Data)')
```



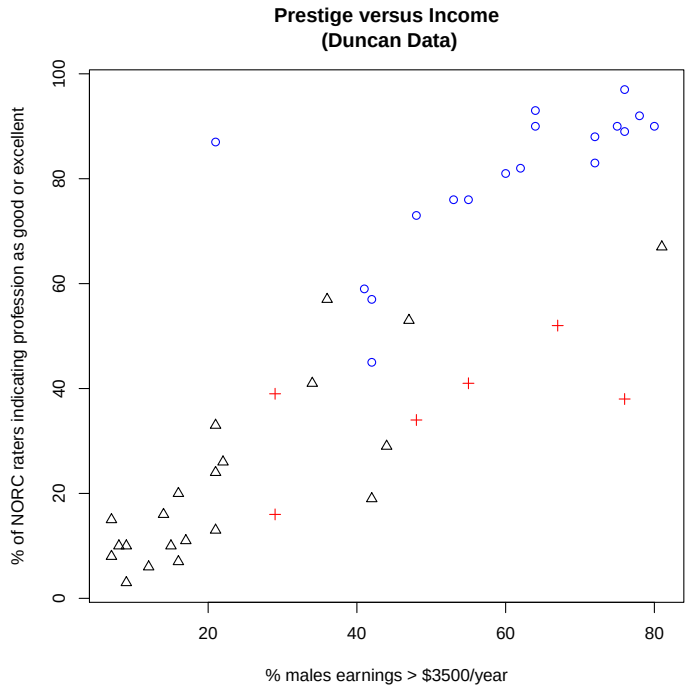
Above, we simply plotted the points, so only two variables are involved here. What if we wanted to include a third variable? We could include information relating to occupation type by coloring the points different for different types of occupations and perhaps using different plotting symbols. To do this, we would have to specify the `pch` argument differently, but could use the rest of the commands we specified above to make the plot.

```
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
    xlab='% males earnings > $3500/year',
        ylab='% of NORC raters indicating profession as good or excellent',
        main = 'Prestige versus Income\n (Duncan Data)')
```



Basically, what you are doing is you are specifying the plotting character for each point. This is much easier in the more advanced systems, but we'll get there eventually. We could also do this with colors.

```
cols <- c("blue", "black", "red")
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
        col = cols[as.numeric(Duncan$type)],
    xlab='% males earnings > $3500/year',
        ylab='% of NORC raters indicating profession as good or excellent',
        main = 'Prestige versus Income\n (Duncan Data)')
```



You're doing this by allowing our vector of the three colors to be indexed by the occupation type. Nifty, right!?

You try it

Using the `strikes_small.rda` object, do the following:

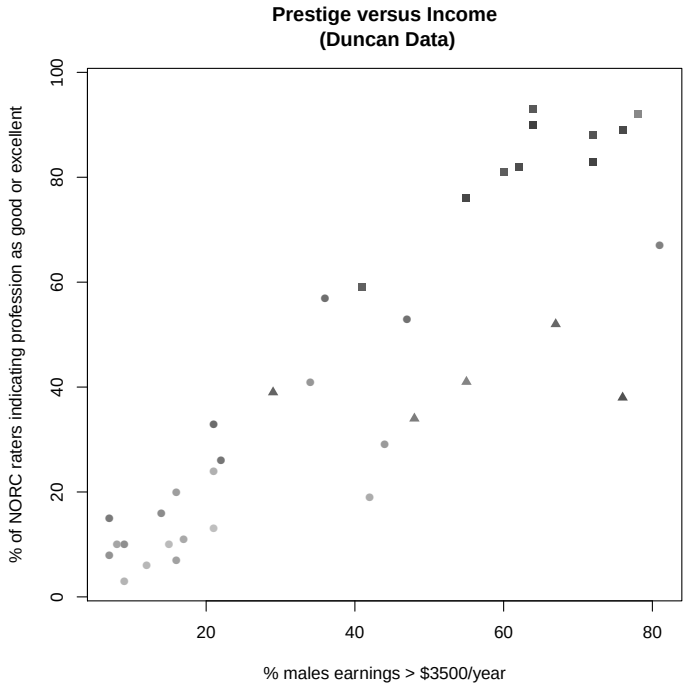
1. Plot the log of strike volume (+1) against one of the quantitative variables in the dataset.

2. Make a factor variable out of `sdlab_rep` such that there are three evenly-sized groups. Change the plotting symbols in the graph above based on the values of this variable.

3. Make the colors of the plotting symbols above a function of the factor you created in the last step.

You can also encode quantitative information in color with `colorRampPalette`.

```
myRamp <- colorRampPalette(c("gray75", "gray25"))
cols <- myRamp(length(unique(Duncan$education)))
pchs <- c(15,16,17)
plot(prestige ~ income, data=Duncan,
        pch=pchs[as.numeric(Duncan$type)],
        col = cols[order(Duncan$education)],
    xlab='% males earnings > $3500/year',
        ylab='% of NORC raters indicating profession as good or excellent',
        main = 'Prestige versus Income\n (Duncan Data)')
```



Next, we can talk about adding elements to the plots.

11.4.1 Adding a Legend

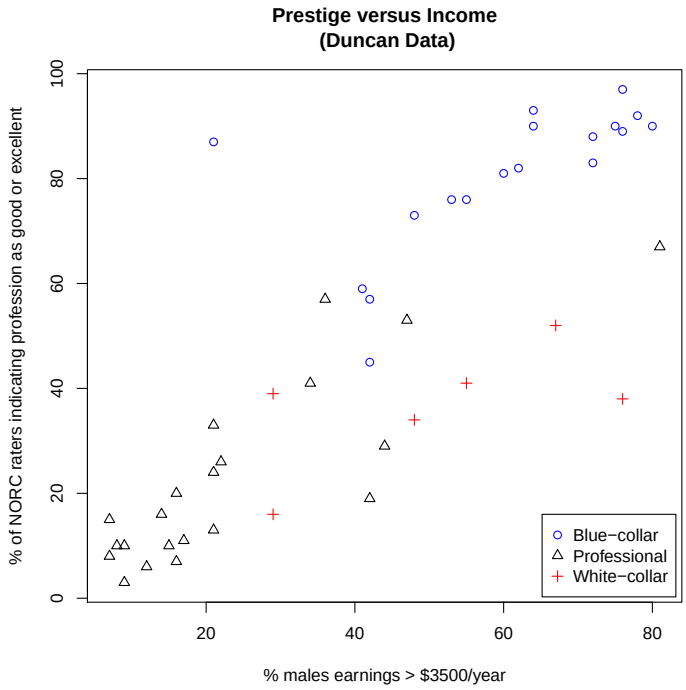
We know what the points mean, but we can't very well expect other people to know this unless we tell them. There is a function called `legend` that allows us to make a legend and stick it in the plot. The `legend` command has a number of arguments that should be specified.

- The first argument is the location of the plot. This can be specified in a number of ways. If you provide x and y coordinate values (with `x=#` and `y=#`), then this gives the coordinates for the top-left corner of the box containing the legend information. Otherwise, you can specify the location with `topleft`, `top`, `topright`, `right`, `bottomright`, `bottom`, `bottomleft`, `left` and **R** will put the legend near the edge of the plot in this position.

- The `legend` argument gives the text you want to be displayed for each point/line you're describing.
- The point symbol and color are given by a vector to `pch` and `col`.
- If instead of points, you have lines, you can give the argument `lty` with the different line types being used (more on this later).
- `inset` gives the fraction of the plot region between the edges of the legend and the box around the plotting region (default is 0).

We can include a legend in our plot as follows:

```
cols <- c("blue", "black", "red")
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
        col = cols[as.numeric(Duncan$type)],
    xlab='% males earnings > $3500/year',
        ylab='% of NORC raters indicating profession as good or excellent',
        main = 'Prestige versus Income\n (Duncan Data)')
legend("bottomright", c("Blue-collar", "Professional", "White-collar"),
    pch=1:3, col=cols, inset=.01)
```



You try it

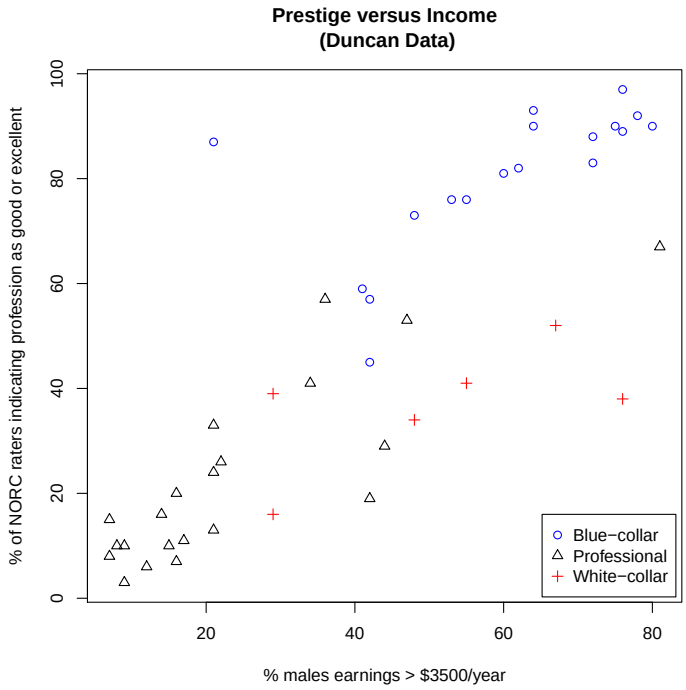
Using the `strikes_small.rda` object, do the following:

1. Add a legend to the plot you made in the previous exercise

11.4.2 Adding a Regression Line

We can add a regression line to the previous graph with `abline()` which adds a line of specified slope and intercept to the plot.

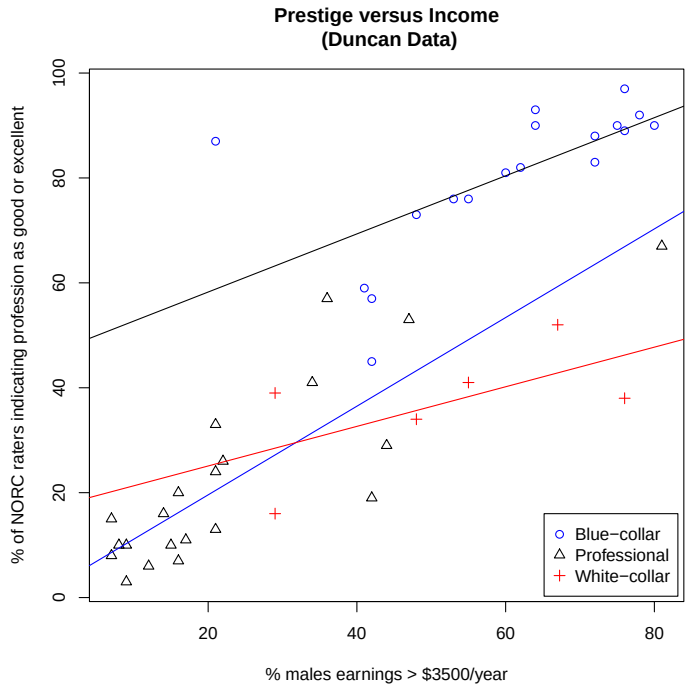
```
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
  col = cols[as.numeric(Duncan$type)],
    xlab='% males earnings > $3500/year',
  ylab='% of NORC raters indicating profession as good or excellent',
  main = 'Prestige versus Income\n (Duncan Data)')
legend("bottomright", c("Blue-collar", "Professional", "White-collar"),
    pch=1:3, col=cols, inset=.01)
```



If we wanted to add a different regression line for each occupation type, we can do that with three different calls to `abline()`.

```
plot(prestige ~ income, data=Duncan, pch=as.numeric(Duncan$type),
  col = cols[as.numeric(Duncan$type)],
    xlab='% males earnings > $3500/year',
  ylab='% of NORC raters indicating profession as good or excellent',
  main = 'Prestige versus Income\n (Duncan Data)')
legend("bottomright", c("Blue-collar", "Professional", "White-collar"),
    pch=1:3, col=cols, inset=.01)
abline(lm(prestige ~ income, data=Duncan,
        subset=Duncan$type == "bc"), col=cols[1])
abline(lm(prestige ~ income, data=Duncan,
        subset=Duncan$type == "prof"), col=cols[2])
abline(lm(prestige ~ income, data=Duncan,
```

```
        subset=Duncan$type == "wc"), col=cols[3])
```



11.4.3 Identifying Points in the Plot

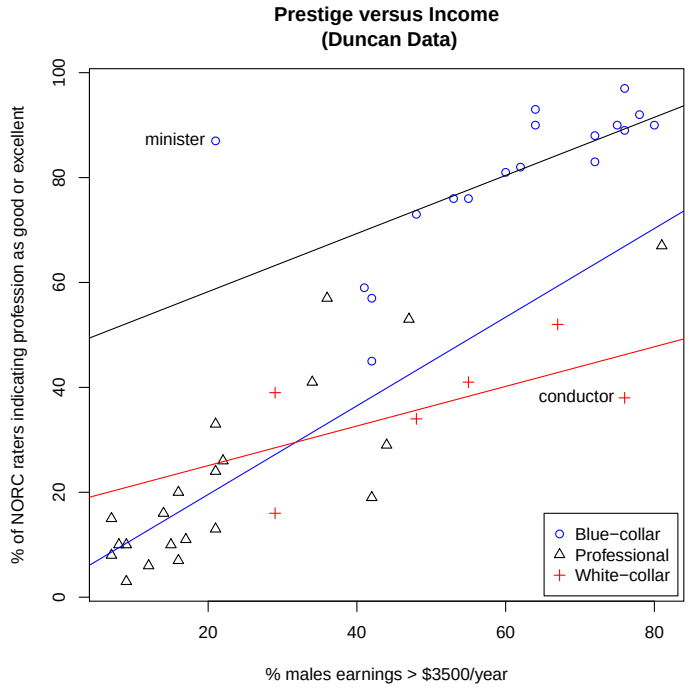
Points in the plot can be identified with the function `identify`. The `identify` will print a label next to points that you click on giving an indication of which point exactly is plotted. The important arguments are:

- Location is given by an `x` and `y` variable. These should be the same `x` and `y` variables you used to make the plot.
- The labels, given with the `label` option will provide **R** with text to print by each identified point.
- `n` gives the number of points we want to identify.

Let's try to identify the two most outlying points on our graph.

```
identify(x=Duncan$income, y=Duncan$prestige,
    labels=rownames(Duncan), n=2)
```

You can see here that the `minister` has more prestige on average than other jobs with similar percentage of males making over $3500 and that the conductor (railroad) has less prestige than its income would suggest.



You don't have to set `n` to any number in particular, just remember to shut off the identifying if you don't set it. On a mac, just right-clicking in the plot will turn off the identifying function. On Windows, you can right-click in the plot, then a dialog will pop up asking if you want to stop or continue. Not surprisingly, if you want to stop identifying, just click "stop". There is another function called `locator()` which will simply return the (x,y) coordinate of the point clicked in the plotting region.

You try it

Using the `strikes_small.rda` object, do the following:

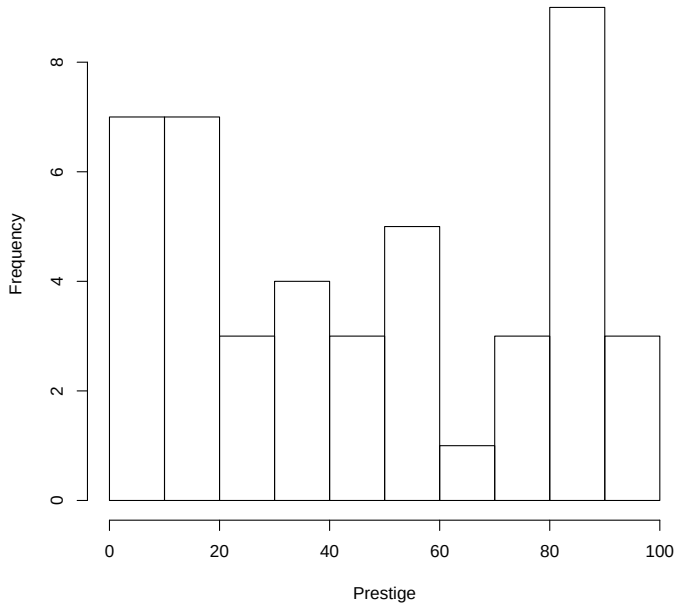
1. Identify points in the previous plot you made by country and year. Note, you will have to make a variable that is country and year with `paste(strikes$abb, strikes$year, sep=:")"`

11.5 Other Plots

There are a few other plots that are both common and useful. I will talk about histograms, bar plots, density estimates and dot plots. Histograms can be done with the `hist()` command:

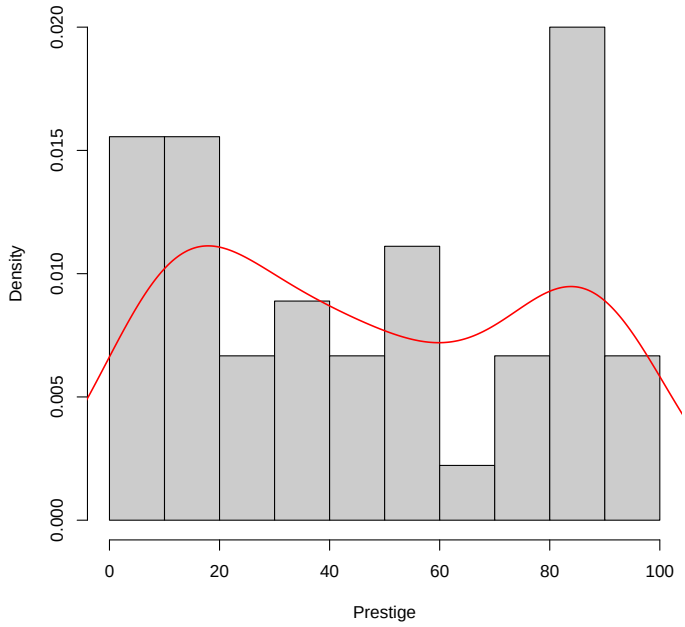
```
hist(Duncan$prestige, nclass=10, xlab="Prestige", main="")
```

The `nclass` argument tells R how many bins you want. For whatever reason, it doesn't always give you exactly that many bars, but it does come close usually.



If you wanted to impose a kernel density estimate over the bars, you could use `lines()` with the `density()` command. Note that for this to work, you need the original histogram to be in density (rather than frequency) units. You can accomplish this by specifying the `freq=FALSE` argument.

```
hist(Duncan$prestige, nclass=10, xlab="Prestige",
        main="", freq=FALSE, col="gray80")
lines(density(Duncan$prestige), col="red", lwd=1.5)
```



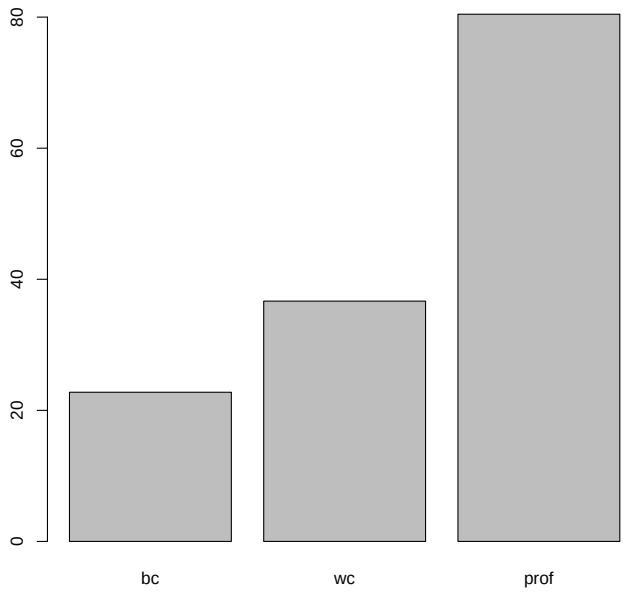
You try it

Using the `strikes_small.rda` object, do the following:

1. Make a histogram of strike volume and impose a kernel density estimate.

Bar plots can be made first by aggregating a continuous variable over the values of some factor, then using barplot.

```
library(dplyr)
bp1 <- Duncan %>% group_by(type) %>% summarise(mp = mean(prestige))
bp1 <- bp1[order(bp1$mp), ]
barplot(bp1$mp, horiz=F, names.arg = bp1$type)

## Error in bp1$mp:  $ operator is invalid for atomic vectors
```

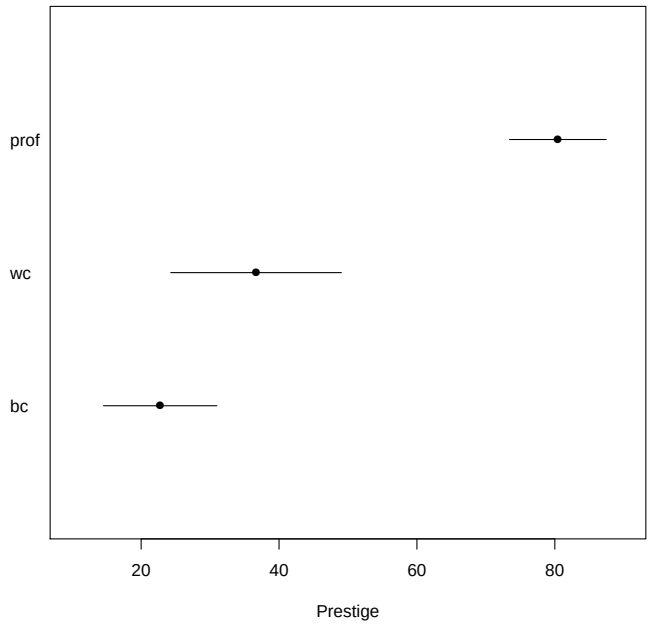


A dot-plot, often showing a set of estimates and confidence bounds arrayed relative to a categorical variable, is relatively common these days. Let's do this by first making the confidence interval of repression for each region. Then we can make the dot-plot.

```
library(gmodels)
ag <- Duncan %>% group_by(type) %>% summarize(mp = mean(prestige),
    lwr = ci(prestige)[2], upr = ci(prestige)[3])

## Error in summarize(., mp = mean(prestige), lwr = ci(prestige)[2], upr =
ci(prestige)[3]):  argument "by" is missing, with no default

ag

## Error in eval(expr, envir, enclos):  object 'ag' not found
```

Now, we can make a dotplot:

```
ag <- ag[order(ag$mp), ]

## Error in eval(expr, envir, enclos):  object 'ag' not found

dotchart(ag$mp, ag$type, xlim=c(10,90),
        lcolor=NA, pch=16, xlab = "Prestige")

## Error in dotchart(ag$mp, ag$type, xlim = c(10, 90), lcolor = NA, pch = 16,
:  object 'ag' not found

segments(ag$lwr, 1:3, ag$upr, 1:3)

## Error in segments(ag$lwr, 1:3, ag$upr, 1:3):  object 'ag' not found
```



You try it

Using the `strikes_small.rda` object, do the following:

1. Make a bar plot of average strike volume by country.
2. Make a dot plot of average strike volume by country.

12 ggplots

I have long had a preference for the lattice package's graphcis because I think Cleveland's theoretical work is most compelling. However, `ggplot` is really quite dominant now and

growing moreso, so in the interest of showing you the most up-to-date softwarem we'll start there. According to Wickham (2009, p. 3), "a statistical graphic is a mapping from data to aesthetic attributes (color, shape size) of geometric objects (points, lines, bars)." The plot may contain transformations of the data that inform the *scale* or the coordinate system. Finally, faceting, or grouping, can generate the same plot for different subsets of the data. These components can be combined in various ways to make graphics. The main elements of a ggplot are:

data/mappings The data that you want to represent visually.

geoms represent the elements you see on the plot (lines, points, shapes, etc...)

scales map the values of the data into an aesthetic space through colors, sizes or shapes. They also draw legends or axes that allow users to read original data values from the plot.

coord the coordinate system describes how coordinates are mapped to a two-dimensional graph.

facet describes how to subset the data.

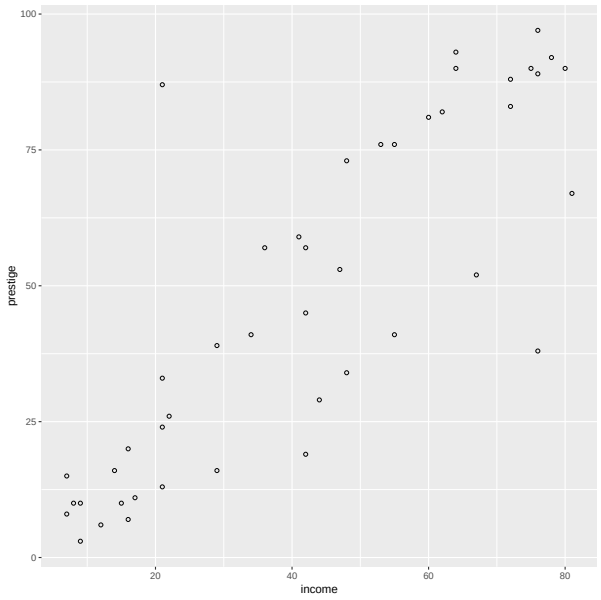
12.1 Scatterplot

Let's go back to our scatterplot with the Duncan data. As you'll see, the more complicated aspects of what we did above will become quite easy here. First, we need to initialize the plot with the `ggplot()` function.

```
g <- ggplot(Duncan, aes(x=income, y=prestige))
```

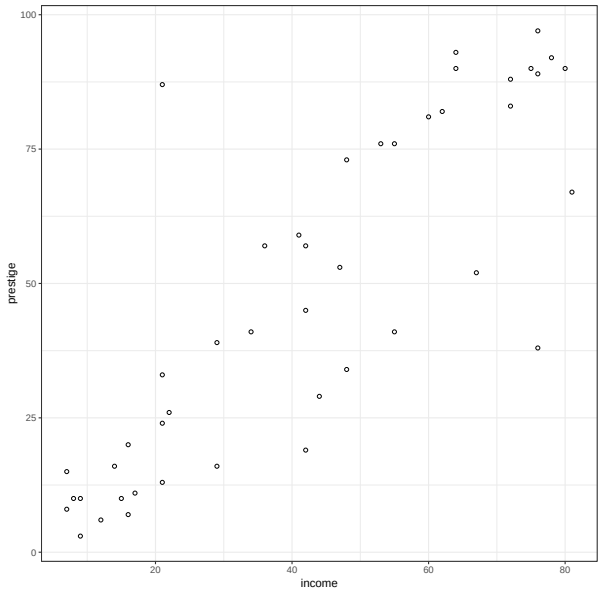
The above won't produce any output, we need to add geometric elements to the plot for it to display information.

```
g <- g + geom_point(pch=1)
g
```



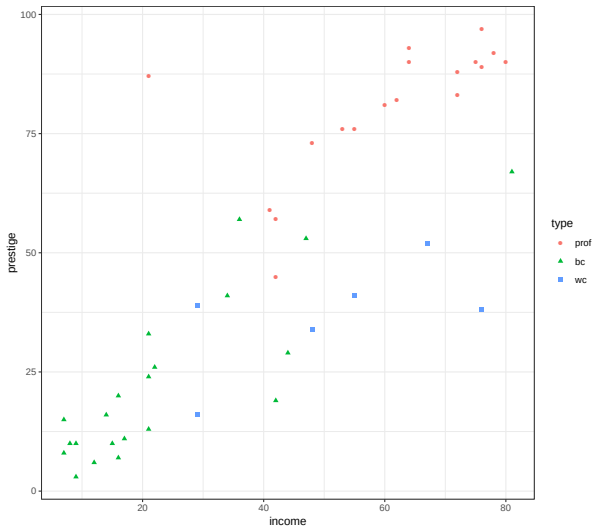
There are some things not to like about this plot. I'm not a huge fan of the gray background (I'd prefer white) and I want the aspect ratio to be 1. I could implement both of these with the following:

```
g <- g + theme_bw() + theme(aspect.ratio=1)
g
```



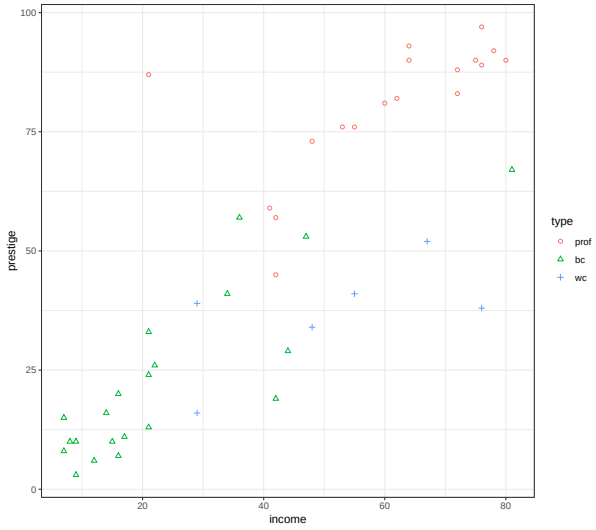
This gets us in pretty good shape. Now, we can start adding in some of the other elements (like symbols and colors that depend on occupation type).

```
g <-  ggplot(Duncan, aes(x=income, y=prestige, colour=type, shape=type)) +
    geom_point() + theme_bw() + theme(aspect.ratio=1)
g
```



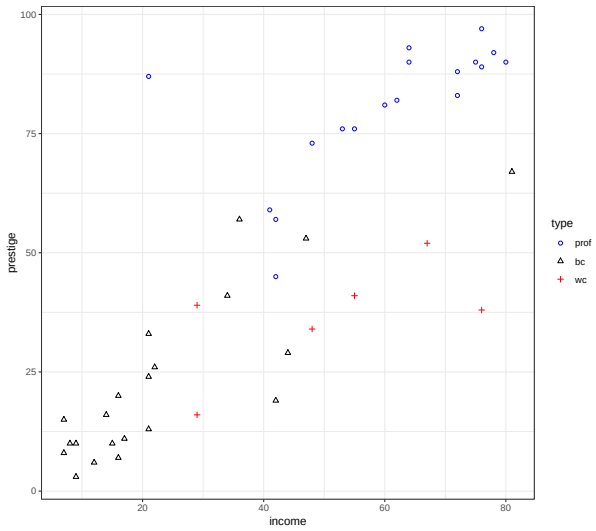
If you don't like the colors or shapes, those can both be changed like the following:

```
g <- ggplot(Duncan, aes(x=income, y=prestige, colour=type, shape=type)) +
    geom_point() + theme_bw() + theme(aspect.ratio=1) + scale_shape_manual(values=c(1
g
```



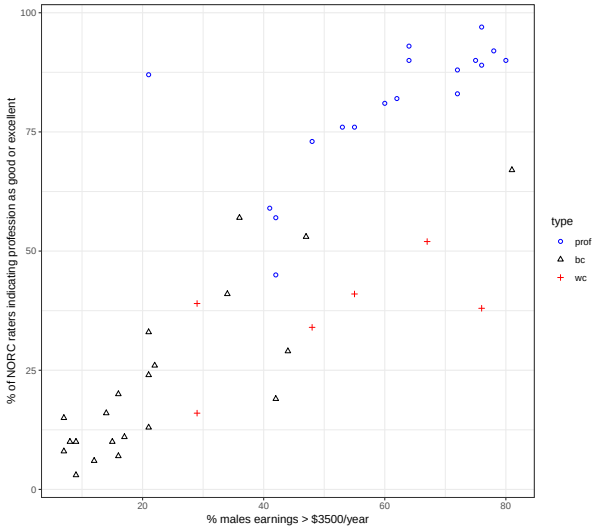
The `scale_colour_manual()` function will change the colors if you want different colors.

```
g <- ggplot(Duncan, aes(x=income, y=prestige, colour=type, shape=type)) +
    geom_point() + theme_bw() + theme(aspect.ratio=1) +
    scale_shape_manual(values=c(1,2,3)) +
    scale_colour_manual(values=c("blue", "black", "red"))
g
```



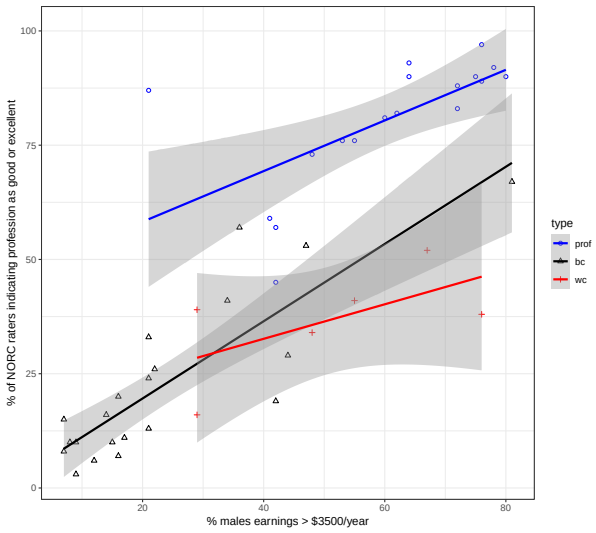
Finally, to make the plot just like before, we can use the `xlab()` and `ylab()` functions.

```
g <- ggplot(Duncan, aes(x=income, y=prestige, colour=type, shape=type)) +
    geom_point() + theme_bw() + theme(aspect.ratio=1) +
    scale_shape_manual(values=c(1,2,3)) +
    scale_colour_manual(values=c("blue", "black", "red")) +
    xlab('% males earnings > $3500/year') +
    ylab('% of NORC raters indicating profession as good or excellent')
g
```



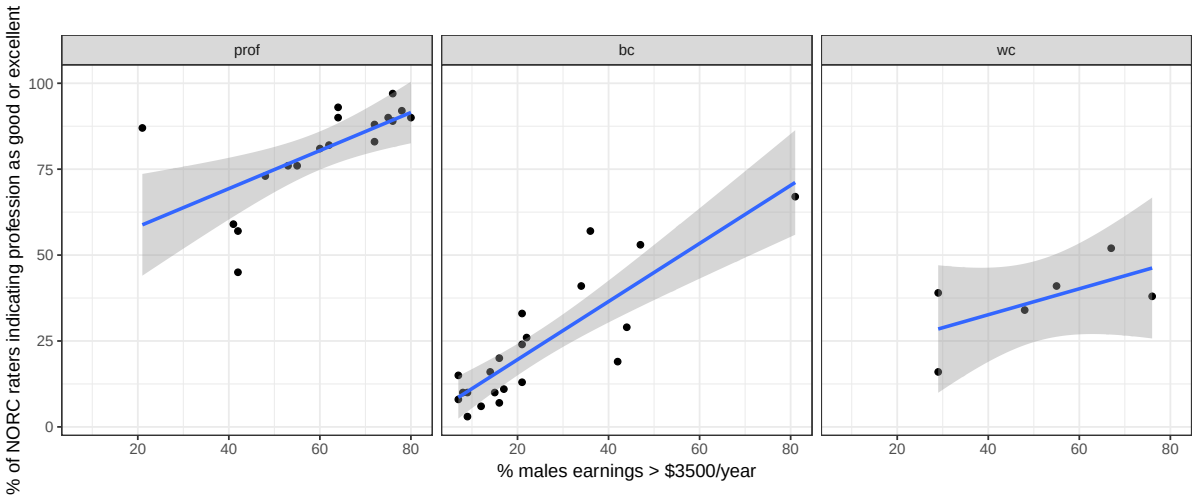
The real value comes here when adding in a regression line. This can be done with just adding `geom_smooth()`.

```
g <- ggplot(Duncan, aes(x=income, y=prestige, colour=type, shape=type)) +
    geom_point() + geom_smooth(method="lm") +
    theme_bw() + theme(aspect.ratio=1) +
    scale_shape_manual(values=c(1,2,3)) +
    scale_colour_manual(values=c("blue", "black", "red")) +
    xlab('% males earnings > $3500/year') +
    ylab('% of NORC raters indicating profession as good or excellent')
g
```



These plots have all been superposed - the lines and points are all in the same region. Another option is to juxtapose the plot. We can do this with the `facet_wrap()` function.

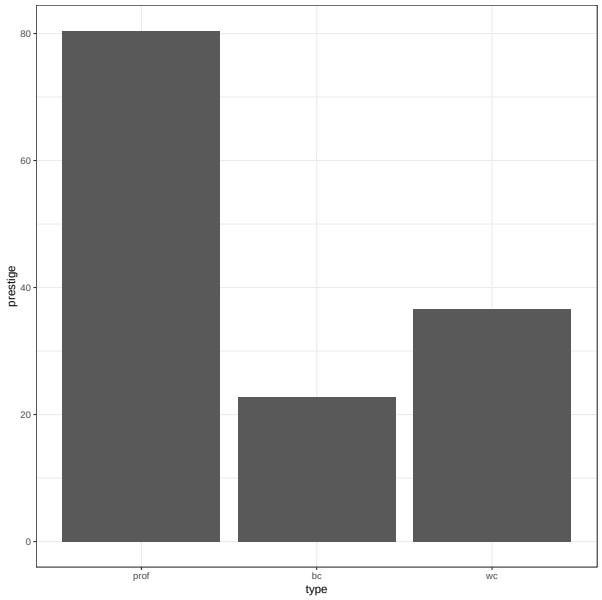
```
g <- ggplot(Duncan, aes(x=income, y=prestige)) +
    geom_point() + geom_smooth(method="lm") +
    theme_bw() + theme(aspect.ratio=1) +
    facet_wrap(~type) +
    xlab('% males earnings > $3500/year') +
    ylab('% of NORC raters indicating profession as good or excellent')
g
```



12.1.1 Bar Graph

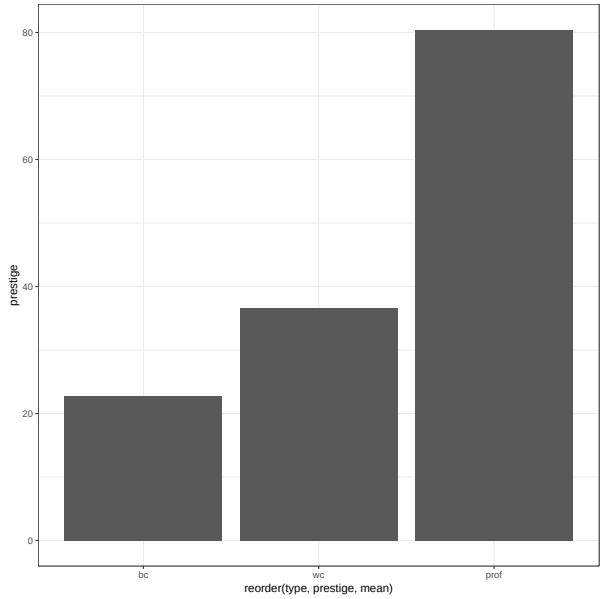
One of the benefits of the `ggplot` tools is that they can do both the data manipulation and the plotting all in one.

```
ggplot(Duncan, aes(x=type, y=prestige)) +
    stat_summary(geom="bar", fun.y=mean) +
    theme_bw()
```



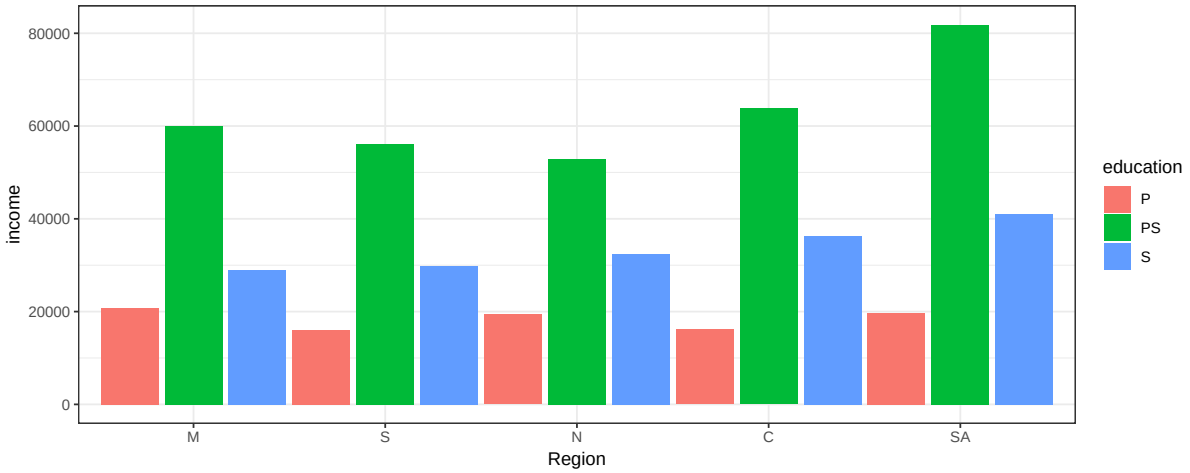
If you want to order the bars by height, you can do that in the aesthetic part of the function.

```
ggplot(Duncan, aes(x=reorder(type, prestige, mean), y=prestige)) +
    stat_summary(geom="bar", fun.y=mean) +
    theme_bw()
```



Side-by-side barplots can be accomplished in a lot the same way as in Lattice.

```
p <- ggplot(subset(Chile, !is.na(education)), aes(x=reorder(region, income, mean, na.
    stat_summary(fun.y=mean, geom="bar", position = position_dodge(1)) +
    xlab("Region") +
    theme_bw()
p
```

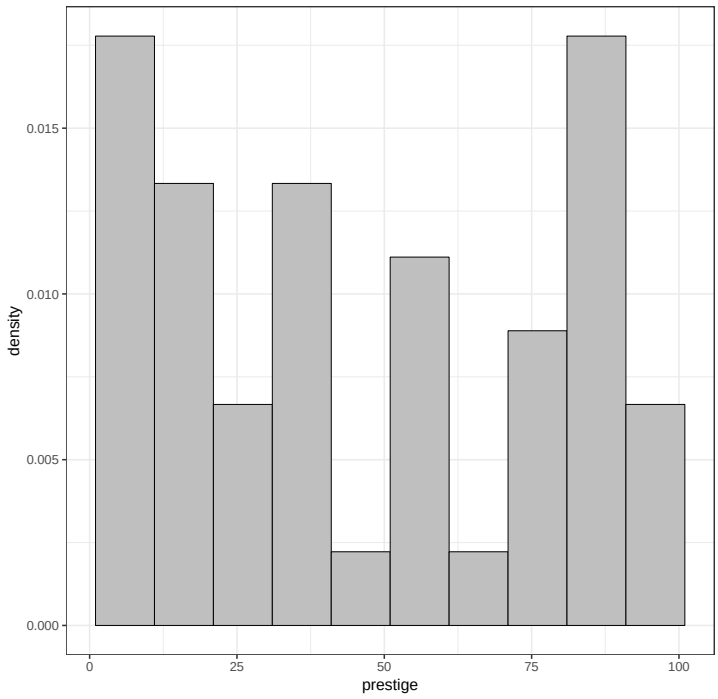


12.2 Other Plots

You can also make the whole set of other plots with `ggplot2` as well. Here are a few examples.

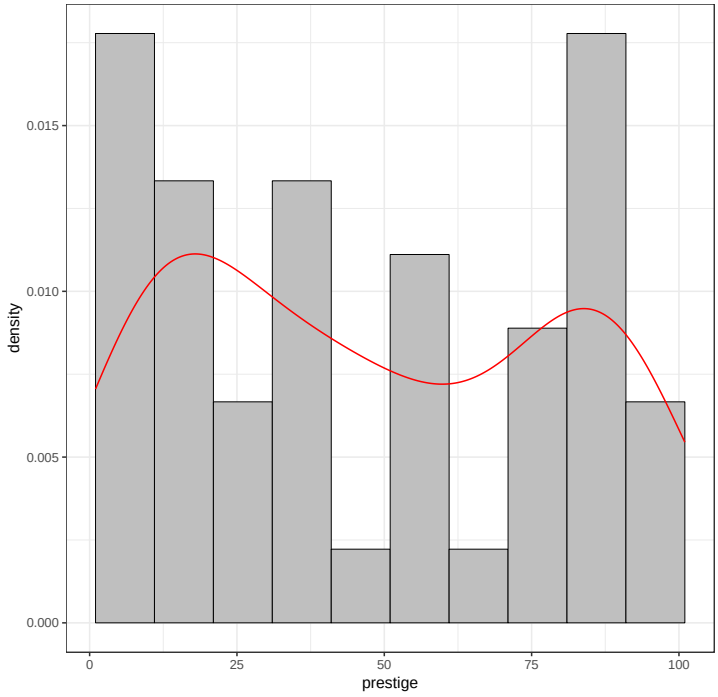
12.2.1 Histograms and Barplots

```
p <- ggplot(Duncan, aes(prestige))
p2 <- p + xlim(1,101) + geom_histogram(aes(y = ..density..),
        color="black", size=.25, fill="gray75", breaks=seq(1,101,by=10)) +
  theme_bw()  + theme(aspect.ratio=1)
p2
```



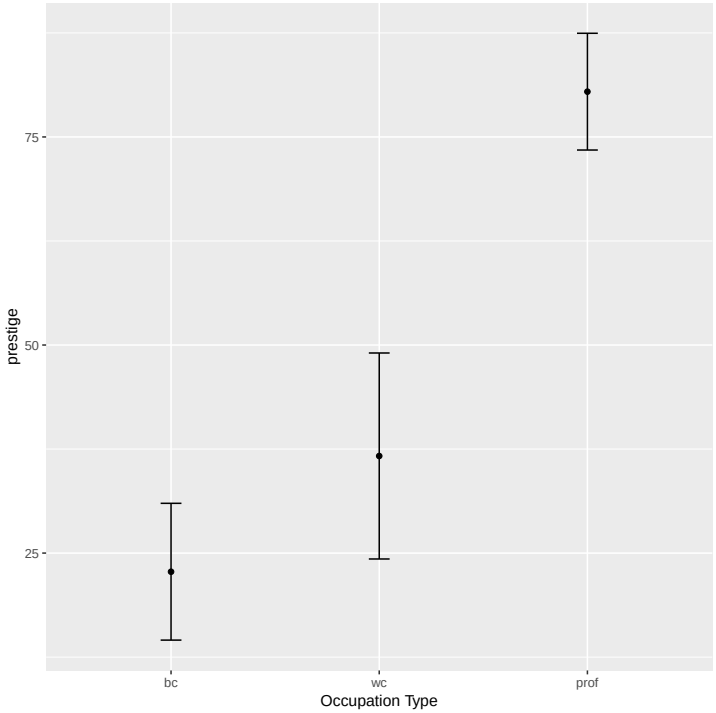
and now with density lines

```
p2 + stat_density(geom = "line", col="red")
```



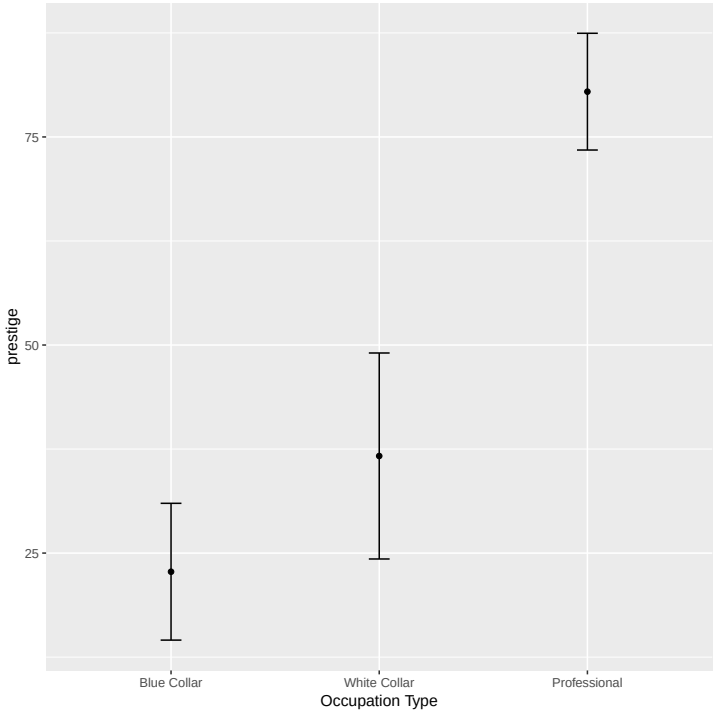
12.2.2 Dotplot

```
ggplot(Duncan, aes(x=reorder(type, prestige, mean), y=prestige)) +
    # add an error-bar geometry
    # that uses the mean_cl_normal
    # function from the Hmisc package.
    stat_summary(geom="errorbar", fun.data=mean_cl_normal, width=.1) +
    # add points at the mean of
    # prestige for each type.
    stat_summary(geom="point", fun.y=mean) +
    # add an x-label.
    xlab("Occupation Type")
```



One thing we might want to change is the labels on the bars. Perhaps we would want them to be "Blue Collar", "White Collar", and "Professional". To do this we could either change the levels of the variable (which would persist beyond the plotting command we're using) or we could issue a call to `scale_x_discerte()` which would allow us to change the labels as we do below:

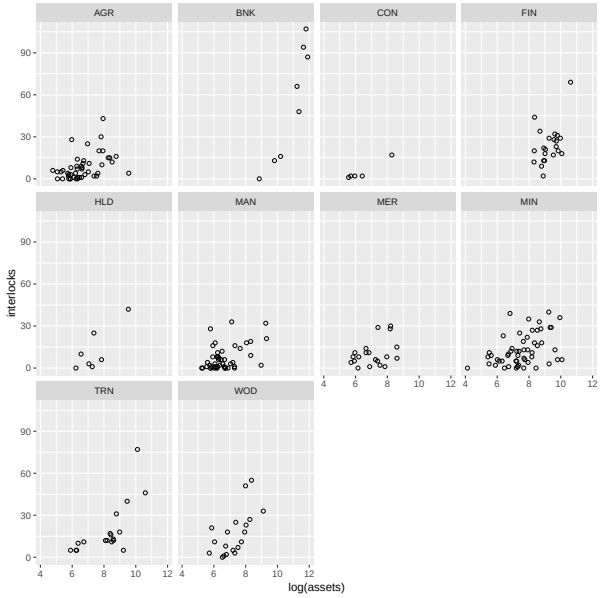
```
ggplot(Duncan, aes(x=reorder(type, prestige, mean), y=prestige)) +
    stat_summary(geom="errorbar", fun.data=mean_cl_normal, width=.1) +
    stat_summary(geom="point", fun.y=mean) +
    xlab("Occupation Type") +
    scale_x_discrete(labels=c("Blue Collar", "White Collar", "Professional"))
```


75
50
25
prestige
Blue Collar
White Collar
Professional
Occupation Type

12.3 Faceting

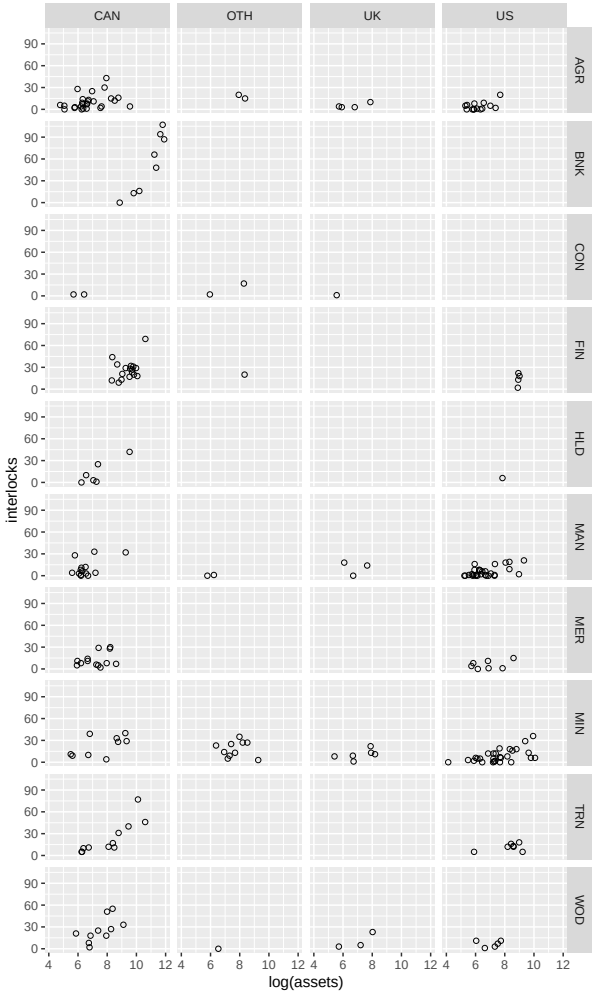
`ggplot2` is also really good with dependent data. There are two different commands that do faceting (juxtaposition rather than superposition). These are the `facet_wrap()` and `facet_grid()` functions. The `facet_wrap()` function takes a single sided formula as its argument and it will then make as many panels as groups wrapping them on to new lines if necessary. The `facet_grid()` function creates a grid of panels from two (or more) variables. For example, `facet_wrap(~z)` would make a panel in the plot for each different group of the variable z, putting some on the second or third lines if necessary. Here's an example:

```
library(car)
data(Ornstein)
ggplot(Ornstein, aes(x=log(assets), y=interlocks)) +
    geom_point(pch=1) + facet_wrap(~sector)
```



The facet grid works differently, by making the rows of the display relate to the variable on the left-hand side of the tilde and the columns to the variable on the right-hand side, like below:

```
library(car)
data(Ornstein)
ggplot(Ornstein, aes(x=log(assets), y=interlocks)) +
    geom_point(pch=1) +
    facet_grid(sector ~ nation)
```



Adding a regression line in each panel is also easy:

```
ggplot(Ornstein, aes(x=log(assets), y=interlocks)) +
    geom_point(pch=1) +
    facet_wrap(~sector) +
    geom_smooth(method="lm", se=FALSE, fullrange=TRUE)
```


AGR
BNK
CON
FIN
HLD
MAN
MER
MIN
TRN
WOD
interlocks
log(assets)
100
0
−100
4
6
8
10
12

12.4 Bringing Lots of Elements Together

There are other ways to do this, but just imagine that we wanted to show predictions for two models, one linear in income and one using a cubic polynomial. This would require us take several steps on our way to making a plot of the results.

1. Estimate both models:

```
mod1 <- lm(prestige ~ income + education + type, data=Duncan)
mod2 <- lm(prestige ~ poly(income, 3) + education + type, data=Duncan)
```

2. Calculate the effects in both models:

```
e1 <- predictorEffect("income", mod1, focal.levels=25)
e2 <- predictorEffect("income", mod2, focal.levels=25)
```

3. Take the results of the effects and put them into a dataset.

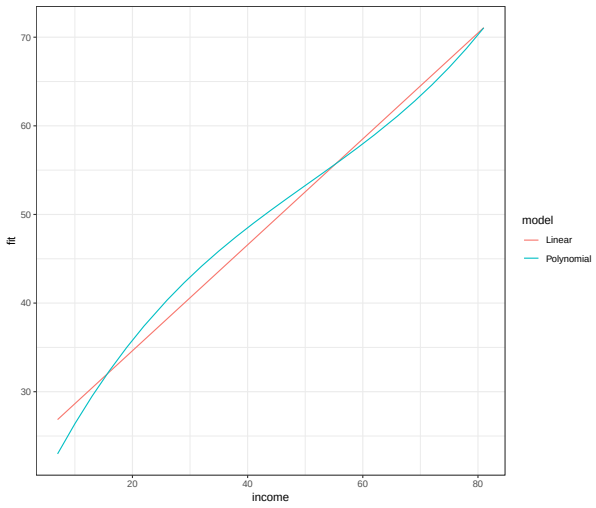
```
ed1 <- do.call("data.frame", e1[c("x", "fit", "lower", "upper" )])
ed1$model <- factor(1, levels=1:2, labels=c("Linear", "Polynomial"))
ed2 <- do.call("data.frame", e2[c("x", "fit", "lower", "upper" )])
ed2$model <- factor(2, levels=1:2, labels=c("Linear", "Polynomial"))
```

4. Append the datasets to make a single dataset.

```
plot.dat <- rbind(ed1, ed2)
```

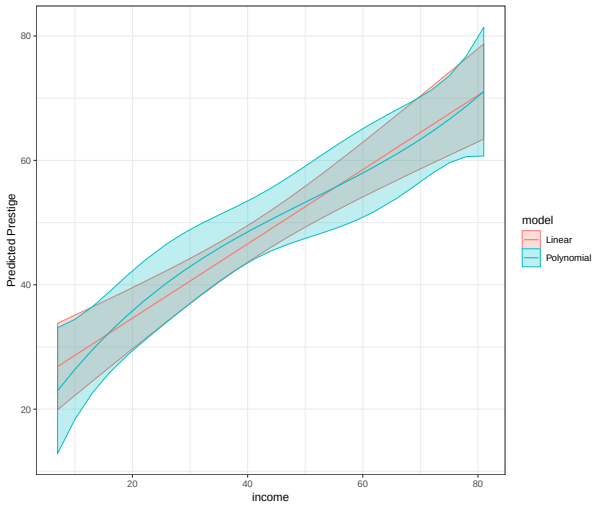
5. Make the Plot, first, let's just make the lines:

```
g <- ggplot(plot.dat, aes(x=income, y=fit, colour=model)) +
    geom_line() +
    theme_bw() +
    theme(aspect.ratio=1)
g
```



Now, if we wanted to put confidence bounds around them, we could use the ribbon geometry.

```
g <- ggplot(plot.dat, aes(x=income, y=fit, colour=model)) +
    geom_ribbon(aes(ymin = lower, ymax=upper, fill=model),
        alpha=.25, size=0) +
    geom_line() +
    ylab("Predicted Prestige") +
    theme_bw() +
    theme(aspect.ratio=1)
g
```



There is likely a way either using some of the `stat_` functions or with `visreg`, but sometimes the easiest thing is to put together pieces that you already know.

13 Maps

One of the real benefits on R is that it is relatively easy to make good looking maps. You'll need a shape file (and the associated auxiliary files) and potentially some extra data you want to plot. The most recent advancement in organizing spatial data in R is called a "simple features dataframe". This method for organizing and managing data is operationalized in the **sf** package. The novelty here is that all of the geographic information is represented in a single column (variable) in the data frame. This makes subsetting and other data management tasks a bit easier.

Before trying to install the **sf** package, please look at the instructions, which differ for Windows and Mac and are available here: https://github.com/r-spatial/sf. These require the installation of some additional software.

I'm going to show you a couple of different examples here. The first one is the ideal situation where you have a consistent, standardized numeric identifier in both the spatial data and the data you want to plot. After some googling, I ended up finding the county boundary files (along with lots of other us geographies) here: https://www.census.gov/geographies/mapping-files/time-series/geo/carto-boundary-file.html

This downloaded a .zip archive that contained 7 files. I've put a link to the .zip file on the course website.

```
library(sf)
counties <- st_read("county_boundaries/cb_2018_us_county_20m.shp")

## Reading layer `cb_2018_us_county_20m' from data source `/Users/david/Dropbox (DaveArmstro
## Simple feature collection with 3220 features and 9 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -179.1743 ymin: 17.91377 xmax: 179.7739 ymax: 71.35256
## epsg (SRID):    4269
## proj4string:    +proj=longlat +datum=NAD83 +no_defs

head(counties)

## Simple feature collection with 6 features and 9 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -99.20277 ymin: 31.44832 xmax: -76.22014 ymax: 41.59003
## epsg (SRID):    4269
## proj4string:    +proj=longlat +datum=NAD83 +no_defs
##   STATEFP COUNTYFP COUNTYNS       AFFGEOID GEOID     NAME LSAD      ALAND
## 1      37      017 01026336 0500000US37017 37017   Bladen   06 2265887723
## 2      37      167 01025844 0500000US37167 37167   Stanly   06 1023370459
## 3      39      153 01074088 0500000US39153 39153   Summit   06 1069181981
## 4      42      113 01213687 0500000US42113 42113 Sullivan   06 1165338428
## 5      48      459 01384015 0500000US48459 48459   Upshur   06 1509910100
## 6      48      049 01383810 0500000US48049 48049    Brown   06 2446120250
```

```
##     AWATER                       geometry
## 1 33010866 MULTIPOLYGON (((-78.902 34....
## 2 25242751 MULTIPOLYGON (((-80.49737 3...
## 3 18958267 MULTIPOLYGON (((-81.68699 4...
## 4  6617028 MULTIPOLYGON (((-76.81373 4...
## 5 24878888 MULTIPOLYGON (((-95.15274 3...
## 6 32375524 MULTIPOLYGON (((-99.19587 3...
```

Next, we'll need to read in some data to plot. I've grabbed the 2016 crime data from ICPSR's data archive.

```
library(rio)
crime <- import("crime_data_2016.csv")
```

The main thing we need to be aware of is that the FIPS codes are named different things in the two data sets and we need to change that.

```
names(crime)[which(names(crime) == "FIPS_ST")] <- "STATEFP"
names(crime)[which(names(crime) == "FIPS_CTY")] <- "COUNTYFP"
```

One thing that we would notice when we tried to merge these together is that the variables are of two different types. We need to change the one in the counties data to be integers. This is a strange case because the numbers that are the levels of the factors are not necessarily the same as the numeric values of the factor (I know, this is confusing). What we can do is the following:

```
counties$STATEFP <- as.integer(as.character(counties$STATEFP))
counties$COUNTYFP <- as.integer(as.character(counties$COUNTYFP))
```

Now, we can merge the two datasets together.

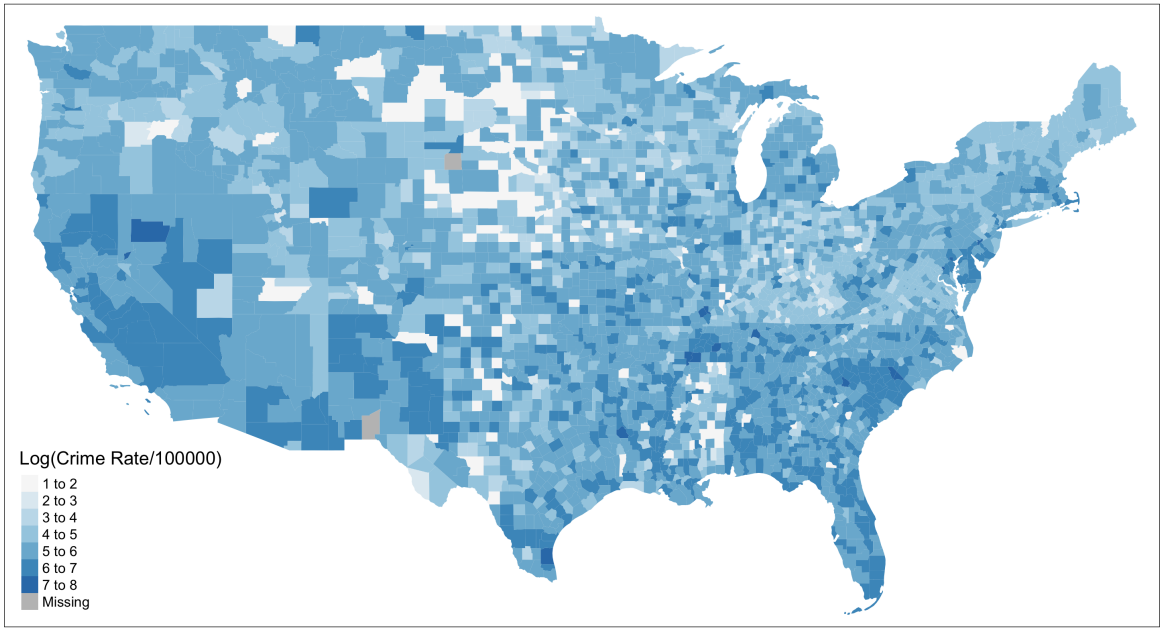
```
library(dplyr)
counties <- left_join(counties, crime)
```

To make the plotting easier, we're going to take out Hawaii and Alaska:

```
counties <- counties[-which(counties$STATEFP %in% c(2, 15, 66, 72)), ]
```

Finally, we can make the map. There are *lots* of ways to do this. One that looks pretty good right out of the box is the one made with `tmap()` which is what we'll discuss.

```
library(tmap)
counties$logcrime <- log(counties$crime_rate_per_100000)
tm_shape(counties) + tm_fill(col="logcrime", palette = "RdBu", title="Log(Crime Rate/100000)
```

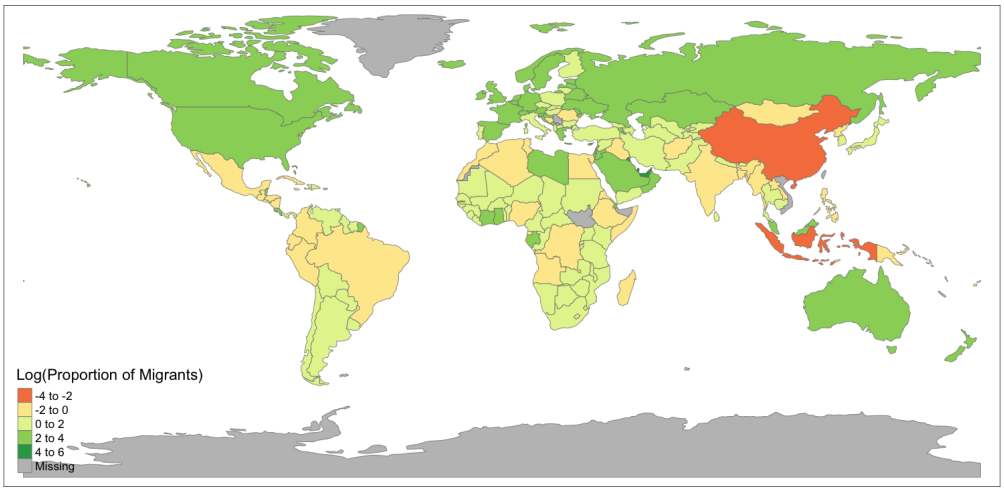


You'll notice that what happened above was relatively simple. Matching up the places was relatively easy and didn't really require any manual changes. An example of the opposite is any time we want to match up countries. There are loads of different country codes and even lots of different ways to spell and format country names. Unless you happen on to two datasets that use the same numeric codes, there may be a bit of manual matching. There is a package called `countrycode` that does a lot of the work. The World Bank data already has

```
data(world, package="spData")
wbmig <- import("wbmig.dta")
library(countrycode)
world$ccode <- countrycode(world$iso_a2, "iso2c", "cown")
wbmig$ccode <- as.integer(wbmig$ccode)
wbmig <- subset(wbmig, !is.na(ccode))
world <- left_join(world, wbmig)
world$logmig <- log(world$wbpropmig)
```

Here, the data managing is, by far, the hardest part. Making the map is actually quite easy.

```
tm1 <- tm_shape(world) +
    tm_fill(col="logmig", title="Log(Proportion of Migrants)") +
    tm_borders(col="gray50", lwd=.5)
```



The output from tmap can also be made into an interactive leaflet map, with a single command:

```
tmap_leaflet(tm1)
```

This will open up a web browser with your map. All javascript libraries required to render the map will be in the map's root directory. This makes it relatively easy to host the map on your website.

R also has lots of spatial statistics routines. First, we have to convert the simple features dataframe into a spatial polygons data fgrame. This can be done as follows:

```
worldsp <- as(world, "Spatial")
class(worldsp)

## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
```

One simple thing that you might have to figure out which polygons are neighbors. You could do this as follows and we can find out which polygons border which others with the following:

```
library(spdep)
nl <- poly2nb(worldsp)
worldsp@data[172, ]

##     iso_a2 name_long continent region_un      subregion              type
## 172     UG    Uganda    Africa    Africa Eastern Africa Sovereign country
##     area_km2      pop lifeExp gdpPercap ccode Country_Name Country_Code
## 172 245768.5 38833338  59.224  1637.275   500          201          199
##     wbpropmig    logmig
## 172  2.271231 0.8203218

worldsp@data[nl[[172]], ]
```

```
##     iso_a2                         name_long continent region_un
## 2       TZ                          Tanzania    Africa    Africa
## 12      CD Democratic Republic of the Congo    Africa    Africa
## 13      CD Democratic Republic of the Congo    Africa    Africa
## 15      KE                             Kenya    Africa    Africa
## 173     RW                            Rwanda    Africa    Africa
## 180     SS                       South Sudan    Africa    Africa
##          subregion                type   area_km2      pop lifeExp gdpPercap
## 2   Eastern Africa Sovereign country  932745.79 52234869  64.163 2402.0994
## 12   Middle Africa Sovereign country 2323492.48 73722860  58.782  785.3473
## 13   Middle Africa Sovereign country 2323492.48 73722860  58.782  785.3473
## 15  Eastern Africa Sovereign country  590836.91 46024250  66.242 2753.2361
## 173 Eastern Africa Sovereign country   23365.41 11345357  66.188 1629.8689
## 180 Eastern Africa Sovereign country  624909.10 11530971  55.817 1935.8794
##     ccode Country_Name Country_Code  wbpropmig     logmig
## 2     510          190          198  2.0546392  0.7201002
## 12    490           40           36 44.3563753  3.7922564
## 13    490           45          213  0.8886225 -0.1180827
## 15    501          100          100  2.2077830  0.7919888
## 173   517          160          165  4.6211463  1.5306428
## 180   626          176          176         NA         NA
```

A spatial weight matrix allows us to capture, in a matrix form, the neighbor-relationships between all observations.

```
wmat <- nb2listw(nl, zero.policy=TRUE)
print.listw(wmat, zero.policy=TRUE)

## Characteristics of weights list object:
## Neighbour list object:
## Number of regions: 180
## Number of nonzero links: 672
## Percentage nonzero weights: 2.074074
## Average number of links: 3.733333
## 21 regions with no links:
## 1 21 22 24 25 47 48 49 80 91 138 139 140 141 142 144 148 151 159 163 179
##
## Weights style: W
## Weights constants summary:
##     n    nn  S0       S1       S2
## W 159 25281 159 97.80043 700.6661
```

Here, weights are $\frac{1}{\#Neighbors}$; for binary weights use `style='B'` as an argument to the `nb2listw` function.

Just like we have to care about temporal autocorrelation, we should also care about spatial autocorrelation - the non-independence of residuals for geographically proximate units. Moran's I is a measure of this:

$$I = \frac{n}{\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}} \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(y_i - \bar{y})(y_j - \bar{y})}{\sum_{i=1}^{n}(y_i = \bar{y})^2}$$

In R, you can calculate Moran's I as follows:

```
moran.test(worldsp@data$logmig, wmat, zero.policy=TRUE, na.action=na.omit)

##
##  Moran I test under randomisation
##
## data:  worldsp@data$logmig
## weights: wmat
## omitted: 3, 22, 23, 24, 25, 45, 47, 81, 96, 138, 144, 153, 163, 164, 171, 176, 178, 180 r
##
##
## Moran I statistic standard deviate = 4.7019, p-value = 1.289e-06
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##       0.301267282      -0.006849315       0.004294289
```

These are for continuous variables, the `joincount.test()` and `joincount.multi()` functions could be used for categorical variables. If the underlying distributional assumptions are dubious, a permutation test is also possible.

```
moran.mc(worldsp@data$logmig, wmat, zero.policy=TRUE, na.action=na.omit, nsim=1000)

##
##  Monte-Carlo simulation of Moran I
##
## data:  worldsp@data$logmig
## weights: wmat
## omitted: 3, 22, 23, 24, 25, 45, 47, 81, 96, 138, 144, 153, 163, 164, 171, 176, 178, 180
## number of simulations + 1: 1001
##
## statistic = 0.28897, observed rank = 1001, p-value = 0.000999
## alternative hypothesis: greater
```

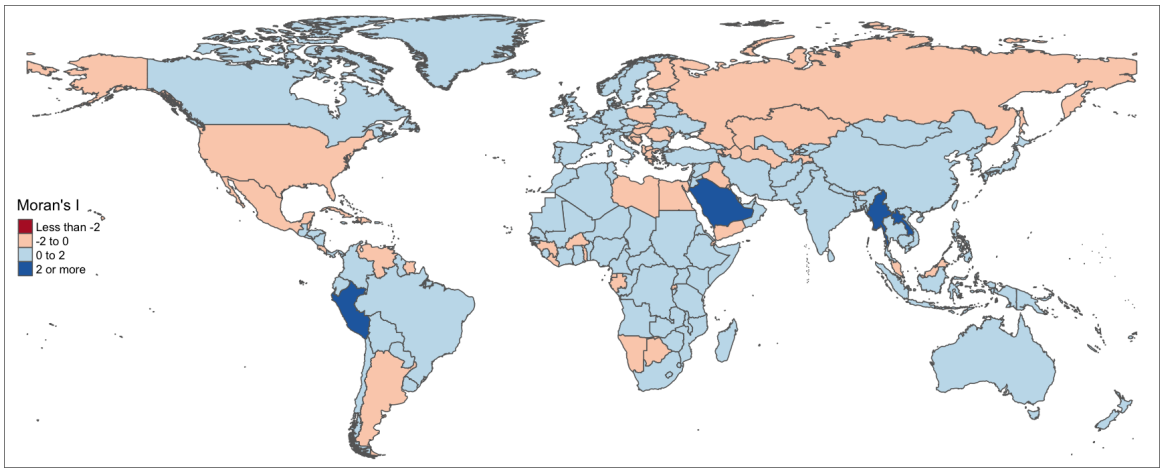
We can also look at the local Moran's I statistics that aggregate to produce the global estimate:

$$I_i = \frac{(y_i - \bar{y}) \sum_{j=1}^{n} w_{ij}(y_j - \bar{y})}{\frac{\sum_{i=1}^{n}(y_i - \bar{y})^2}{n}}$$

```
worldsp2 <- as(world[which(!is.na(world$logmig)), ], "Spatial")
l <- localmoran(worldsp2$logmig, nb2listw(poly2nb(worldsp2),
    style="C", zero.policy=T), zero.policy=TRUE, na.action=na.omit)
worldsp2@data$localI <- l[,1]
```

We can then use the `tmap` package to map the local autocorrelation.

```
worldsf2 <- as(worldsp2, "sf")
tm_shape(worldsf2) + tm_fill(col="localI",
    palette = "RdBu", title="Moran's I",
    breaks= c(-Inf, -2, 0, 2, Inf)) + tm_borders() +
    tm_layout(legend.position = c("left", "center"))
```



Using the `world` data that is in the `spData` package, we can do more interesting statistical things (because there are more interesting variables). Using the same process as above, we can make the neighbors list and then make a neighborhood mean of a variable that could be used in a regression model.

```
data(world)
world <- as(world, "Spatial")
nl <- poly2nb(world)
wmat <- nb2listw(nl, zero.policy=TRUE)
localStat <- function(x, neigh, stat="mean", ...){
    out <- sapply(nl, function(z)do.call(stat, list(x=x[z], ...)))
    return(out)
}
world@data$lifeExp_L1 <- localStat(world@data$lifeExp, nl, na.rm=T)
slmod <- lm(lifeExp ~ lifeExp_L1 + log(gdpPercap) + log(pop), data=world@data)
summary(slmod)

##
## Call:
## lm(formula = lifeExp ~ lifeExp_L1 + log(gdpPercap) + log(pop),
##     data = world@data)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -9.988 -1.984  0.107  2.113  9.253
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -9.86752    4.39534  -2.245   0.0263 *
```

```
## lifeExp_L1      0.68393    0.05755  11.884  < 2e-16 ***
## log(gdpPercap)  2.70317    0.34868   7.753 1.63e-12 ***
## log(pop)        0.46101    0.19105   2.413   0.0171 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.473 on 141 degrees of freedom
##   (32 observations deleted due to missingness)
## Multiple R-squared:  0.8279,Adjusted R-squared:  0.8243
## F-statistic: 226.1 on 3 and 141 DF,  p-value: < 2.2e-16
```

We could also use the bespoke spatial regression function:

```
summary(slmod2 <- lagsarlm(lifeExp ~ log(pop) + log(gdpPercap), data=world@data,
    listw = wmat, zero.policy=T))

##
## Call:
## lagsarlm(formula = lifeExp ~ log(pop) + log(gdpPercap), data = world@data,
##     listw = wmat, zero.policy = T)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -19.9247  -2.1895   0.8774   3.0410   7.9731
##
## Type: lag
## Regions with no neighbours included:
##  1 20 46 47 79 90 97 136 137 138 139 145 148 156 162 176
## Coefficients: (asymptotic standard errors)
##                Estimate Std. Error z value  Pr(>|z|)
## (Intercept)    15.66959    5.12973  3.0547  0.002253
## log(pop)        0.28423    0.24261  1.1716  0.241377
## log(gdpPercap)  5.59441    0.32494 17.2166 < 2.2e-16
##
## Rho: -0.0078364, LR test value: 0.20186, p-value: 0.65322
## Asymptotic standard error: 0.017465
##     z-value: -0.44869, p-value: 0.65365
## Wald statistic: 0.20132, p-value: 0.65365
##
## Log likelihood: -479.2826 for lag model
## ML residual variance (sigma squared): 23.409, (sigma: 4.8383)
## Number of observations: 160
## Number of parameters estimated: 5
## AIC: 968.57, (AIC for lm: 966.77)
## LM test for residual autocorrelation
## test value: 54.805, p-value: 1.3312e-13
```

14 Reproducibility and Tables from R to Other Software

One thing that is important to figure out is how to easily export tables and other output from R to whatever software you tend to write in. The good news is that there are lots of options here. The bad news is that the ease of integrating R output into your document is more or less inversely proportional to how easy it is to interact with the writing software. By that, I mean that it is easiest to get R tables and figures into LaTeX. In fact, it is possible (as I've done in these handouts) write a single document that has all prose and code that is processed by R and then LaTeXto produce both the analysis and the final document. This is great, by LaTeXhas a pretty steep learning curve relative to other composition software (though not compared to other markup languages). The next easiest thing is to write in Markdown. Markdown is a lightweight typesetting environment that is intended to produce output for the web, but can produce it in other formats, too. Just like with LaTeX, RMarkdown incorporates R code into a markdown document and then it executes the analysis and typesets the document. We'll talk more about this in a bit. Finally, there are ways to get your tables into Word, but it's a bit more of a hassle, though we'll discuss that, too.

There is an interesting package called `stargazer` that allows you to make publication quality tables. These export to LaTeX, html or text, though the text option which might be the most interesting option for many people doesn't work well with Word. Below, we'll talk about some other options. Considering the two models estimated below, we could do the following:

```
library(rio)
strikes <- import("https://quantoid.net/files/rbe/strikes_small.rda")
library(stargazer)
mod1 <- lm(log(strike_vol + 1) ~ inflation +
    unemployment + sdlab_rep, data=strikes)
mod2 <- lm(log(strike_vol + 1) ~ union_cent +
    unemployment + sdlab_rep, data=strikes)
stargazer(mod1, mod2, type="text")

##
## ===============================================================
##                                   Dependent variable:
##                               ----------------------------
##                                   log(strike_vol + 1)
##                                     (1)            (2)
## ---------------------------------------------------------------
## inflation                          0.146***
##                                    (0.023)
##
## union_cent                                        -2.683***
##                                                    (0.301)
##
## unemployment                       0.210***       0.174***
##                                    (0.031)        (0.030)
##
## sdlab_rep                           0.001         -0.008
##                                    (0.008)        (0.007)
##
## Constant                           2.418***       5.303***
##                                    (0.389)        (0.394)
##
## ---------------------------------------------------------------
## Observations                         352            352
## R2                                  0.234          0.302
## Adjusted R2                         0.227          0.296
## Residual Std. Error (df = 348)      1.835          1.752
## F Statistic (df = 3; 348)         35.452***      50.130***
## ===============================================================
## Note:                           *p<0.1; **p<0.05; ***p<0.01
```

There are a few things that we might want to change here. First, we might want to put different variable names in.

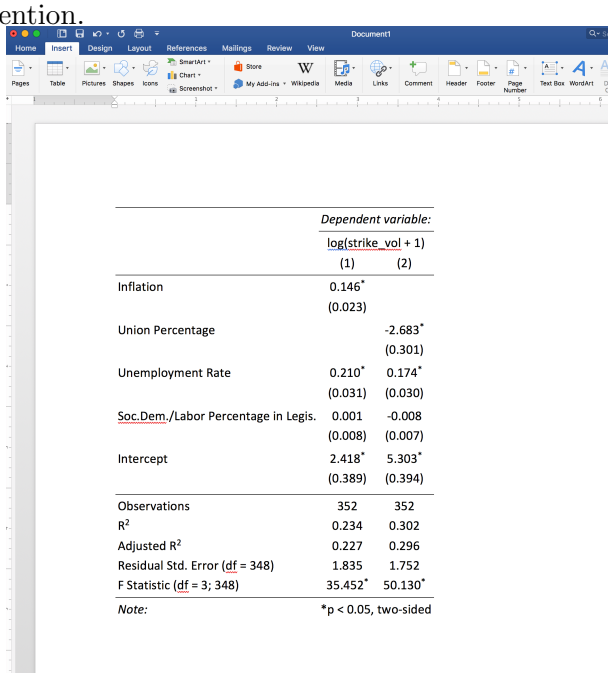
```
stargazer(mod1, mod2, type="text", notes.append=F,
  star.cutoffs = .05, notes="*p < 0.05, two-sided",
  covariate.labels=c("Inflation", "Union Percentage",
    "Unemployment Rate", "Soc.Dem./Labor Percentage in Legis.",
    "Intercept"))
```

```
##
## ============================================================
##                                        Dependent variable:
##                                    ----------------------
##                                     log(strike_vol + 1)
##                                        (1)        (2)
## ------------------------------------------------------------
## Inflation                             0.146*
##                                       (0.023)
##
## Union Percentage                                 -2.683*
##                                                  (0.301)
##
## Unemployment Rate                     0.210*     0.174*
##                                       (0.031)    (0.030)
##
## Soc.Dem./Labor Percentage in Legis.    0.001     -0.008
##                                       (0.008)    (0.007)
##
## Intercept                             2.418*     5.303*
##                                       (0.389)    (0.394)
##
## ------------------------------------------------------------
## Observations                            352        352
## R2                                     0.234      0.302
## Adjusted R2                            0.227      0.296
## Residual Std. Error (df = 348)         1.835      1.752
## F Statistic (df = 3; 348)             35.452*    50.130*
## ============================================================
## Note:                                *p < 0.05, two-sided
```

Once you've got the table how you like it, you can export it to a file in html format as follows:

```
stargazer(mod1, mod2, type="html", notes.append=F,
    out = "table.html", star.cutoffs = .05,
    notes="*p < 0.05, two-sided",
    covariate.labels=c("Inflation", "Union Percentage",
    "Unemployment Rate", "Soc.Dem./Labor Percentage in Legis.",
    "Intercept"))
```

Then, you can go to Word and choose Insert ¿¿ Object ¿¿ Text from File ¿¿ and browse to the `table.html` file to which you just saved the output. This will produce the following output in Word with no intervention.



	Dependent variable:	
	log(strike_vol + 1)	
	(1)	(2)
Inflation	0.146*	
	(0.023)	
Union Percentage		-2.683*
		(0.301)
Unemployment Rate	0.210*	0.174*
	(0.031)	(0.030)
Soc.Dem./Labor Percentage in Legis.	0.001	-0.008
	(0.008)	(0.007)
Intercept	2.418*	5.303*
	(0.389)	(0.394)
Observations	352	352
R^2	0.234	0.302
Adjusted R^2	0.227	0.296
Residual Std. Error (df = 348)	1.835	1.752
F Statistic (df = 3; 348)	35.452*	50.130*
Note:	*p < 0.05, two-sided	

If you wanted stargazer to take account of these robust standard errors in its summary, you can do that. First, you need to get the robust standard errors by taking the square root (`sqrt`) of the diagonal (`diag`) of the robust variance-covariance matrix (`vcovHC`)

```
library(stargazer)
library(sandwich)
library(lmtest)
mod1 <- lm(log(strike_vol + 1) ~ inflation +
    unemployment + sdlab_rep, data=strikes)
mod1 <- lm(log(strike_vol + 1) ~ inflation +
    unemployment + sdlab_rep, data=strikes)
mod2 <- lm(log(strike_vol + 1) ~ union_cent +
    unemployment + sdlab_rep, data=strikes)
se1 <- mod1 %>% vcovHC(., type="HC3") %>% diag %>% sqrt
se2 <- mod2 %>% vcovHC(., type="HC3") %>% diag %>% sqrt
```

```
library(stargazer)
stargazer(mod1, mod2, type="text", notes.append=F,
    star.cutoffs = .05,
    notes="*p < 0.05, two-sided",
    covariate.labels=c("Inflation", "Union Percentage",
    "Unemployment Rate", "Soc.Dem./Labor Percentage in Legis.",
    "Intercept"), se=list(se1, se2))
```

```
##
## ===========================================================
##                                      Dependent variable:
##                                    ----------------------
##                                    log(strike_vol + 1)
##                                       (1)        (2)
## -----------------------------------------------------------
## Inflation                            0.146*
##                                      (0.019)
##
## Union Percentage                                -2.683*
##                                                 (0.273)
##
## Unemployment Rate                    0.210*     0.174*
##                                      (0.030)    (0.030)
##
## Soc.Dem./Labor Percentage in Legis.   0.001     -0.008
##                                      (0.007)    (0.007)
##
## Intercept                            2.418*     5.303*
##                                      (0.378)    (0.406)
##
## -----------------------------------------------------------
## Observations                           352        352
## R2                                    0.234      0.302
## Adjusted R2                           0.227      0.296
## Residual Std. Error (df = 348)        1.835      1.752
## F Statistic (df = 3; 348)            35.452*    50.130*
## ===========================================================
## Note:                               *p < 0.05, two-sided
```

15 Reproducible Research

There are now entire books written on implementing reproducible research designs in R:

- "Reproducible Research with R and R Studio" by Christopher Gandrud
- "Dynamic Documents with R and knitr" by Yihui Xie
- "Implementing Reproducible Research" by Victoria Stodden, Friedrich Leisch and Roger D. Peng

We will touch on a couple of aspects of this today. Here are a few basic pointers.

- At the end of your project, you should be able to run a script that reproduces your results without intervention (i.e., you shouldn't have to do anything aside from run the script).
- It is good practice to have your code build data from scratch wherever possible. That is, if you download the Polity dataset and the CIRI dataset and some data from the World Bank, your code should start with reading in and managing these data to produce a final merged dataset.

Rstudio offers a couple of ways to make reproducible research with Rmarkdown and knitr - we can make article-like documents or presentations. To make an article-like document in Rstudio, click on the button for a new document and selected new Rmarkdown document.

For now, I've got three example documents:

- One article-type document (`reproducible.Rmd`),
- One presentation doucment (`reproducible_present.Rmd`),
- One article-type document in LaTeX

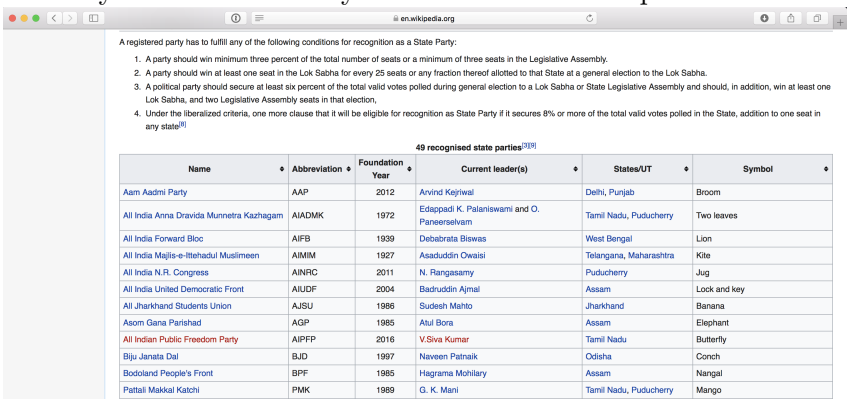
Hopefully get you going in the right direction. We will look through those documents and discuss as necessary.

16 Web Sites to Data

The web is a great source of data. When we think about getting data from the web, we could mean a couple of different things. First, we could mean something relatively simple like automatically importing a table from a website into your stats software. The other thing we could mean is downloading text from a webiste (or social media platform such as Twitter) and then trying to turn that text into data. We'll talk about both different things in turn.

16.1 Importing HTML Tables

Importing HTML tables is relatively easy, though there are lots of cases where it csn get more complicsted. This works best when the every row of the table has the same number of columns and every cell has only one row. Let's say that we wanted to import this table into R:



en.wikipedia.org

A registered party has to fulfill any of the following conditions for recognition as a State Party:

1. A party should win minimum three percent of the total number of seats or a minimum of three seats in the Legislative Assembly.
2. A party should win at least one seat in the Lok Sabha for every 25 seats or any fraction thereof allotted to that State at a general election to the Lok Sabha.
3. A political party should secure at least six percent of the total valid votes polled during general election to a Lok Sabha or State Legislative Assembly and should, in addition, win at least one Lok Sabha, and two Legislative Assembly seats in that election,
4. Under the liberalized criteria, one more clause that it will be eligible for recognition as State Party if it secures 8% or more of the total valid votes polled in the State, addition to one seat in any state[8]

49 recognised state parties[3][9]

Name	Abbreviation	Foundation Year	Current leader(s)	States/UT	Symbol
Aam Aadmi Party	AAP	2012	Arvind Kejriwal	Delhi, Punjab	Broom
All India Anna Dravida Munnetra Kazhagam	AIADMK	1972	Edappadi K. Palaniswami and O. Paneerselvam	Tamil Nadu, Puducherry	Two leaves
All India Forward Bloc	AIFB	1939	Debabrata Biswas	West Bengal	Lion
All India Majlis-e-Ittehadul Muslimeen	AIMIM	1927	Asaduddin Owaisi	Telangana, Maharashtra	Kite
All India N.R. Congress	AINRC	2011	N. Rangasamy	Puducherry	Jug
All India United Democratic Front	AIUDF	2004	Badruddin Ajmal	Assam	Lock and key
All Jharkhand Students Union	AJSU	1986	Sudesh Mahto	Jharkhand	Banana
Asom Gana Parishad	AGP	1985	Atul Bora	Assam	Elephant
All Indian Public Freedom Party	AIPFP	2016	V.Siva Kumar	Tamil Nadu	Butterfly
Biju Janata Dal	BJD	1997	Naveen Patnaik	Odisha	Conch
Bodoland People's Front	BPF	1985	Hagrama Mohilary	Assam	Nangal
Pattali Makkal Katchi	PMK	1989	G. K. Mani	Tamil Nadu, Puducherry	Mango

The process for doing this is to first read in the entire website and then pulling out the tables. Unless the html table tag has a unique class or other identifier that you could drill down on, the function will grab all of the tables and you'll have to figure out which one you want. To read in websites, we'll use the `rvest` package.

```
library(rvest)
india <- read_html("https://en.wikipedia.org/wiki/List_of_political_parties_in_India")
tabs <- html_table(india, fill=T)
head(tabs[[3]])
```

```
##                                     Name Abbreviation Foundation  year
## 1                          Aam Aadmi Party          AAP             2012
## 2 All India Anna Dravida Munnetra Kazhagam       AIADMK             1972
## 3                   All India Forward Bloc         AIFB             1939
## 4   All India Majlis-e-Ittehadul Muslimeen        AIMIM             1927
## 5                  All India N.R. Congress        AINRC             2011
## 6        All India United Democratic Front        AIUDF             2004
##                             Current leader(s)             States/UT
## 1                             Arvind Kejriwal          Delhi, Punjab
## 2 Edappadi K. Palaniswami and O. Paneerselvam Tamil Nadu, Puducherry
## 3                           Debabrata Biswas            West Bengal
## 4                           Asaduddin Owaisi              Telangana
## 5                              N. Rangaswamy             Puducherry
## 6                           Badruddin Ajmal                  Assam
##   Symbol
## 1
## 2
## 3
## 4
## 5
## 6
```

Here's an example of a table that works less well:

Let's try the same thing:

```
library(rvest)
uk <- read_html("https://en.wikipedia.org/wiki/List_of_Prime_Ministers_of_the_United_Kingdom")
tabs <- html_table(uk, fill=T)
colnames(tabs[[2]]) <- c("", "Portrait", "PM", "Start", "End",
    "Year", "portfolio", "party", "ministry", "monarch", "note")
tabs[[2]] <- tabs[[2]][-1, ]
```

```
tabs[[2]][1:2, ]

##    Portrait
## 2
## 3
##
## 2 The Right HonourableSir Robert Walpole1st Earl of OrfordKGKBPCMP for King's Lynn[§](167
## 3 The Right HonourableSir Robert Walpole1st Earl of OrfordKGKBPCMP for King's Lynn[§](167
##         Start             End Year
## 2 3 April1721 11 February1742 1722
## 3 3 April1721 11 February1742 1727
##
## 2 Chancellor of the ExchequerFirst Lord of the TreasuryLeader of the House of Commons (17
## 3 Chancellor of the ExchequerFirst Lord of the TreasuryLeader of the House of Commons (17
##   party          ministry              monarch note
## 2  Whig Walpole{Townshend  George I(1714{1727) [30]
## 3  Whig Walpole{Townshend George II(1727{1760) [30]
```

This works less well, though the information is there, so it is likely easier than typing it in yourself.

16.2 Scraping Websites for Content

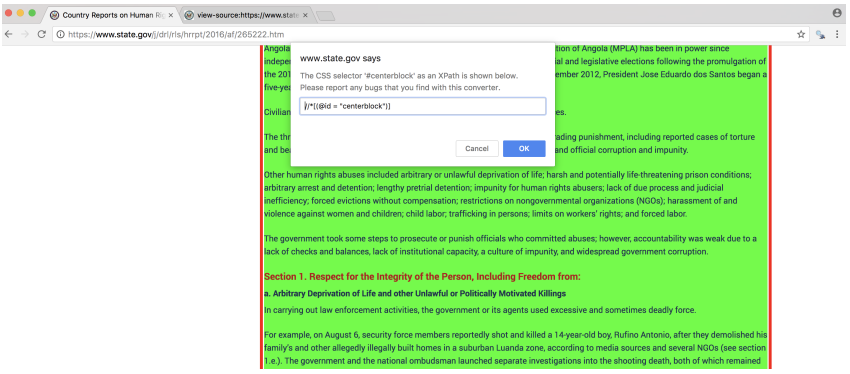
Website and social media scraping has really been a game-changer in generating new data to test our social scientigfic theories. R has good facilities for handling text data. It's probably not quite as good as python, but the startup cost is lower.

One thing that's really important here is that you need to make sure you know what the terms of service for the site you're scraping allow. This is sometimes not particularly easy, but it is important. There was recently some discussion on the POLMETH listserv about how Facebook prohibits scraping its sites in any other way than using their own API. In some cases, you have access to APIs through R (e.g., twitteR).

In any event, **rvest** is the way that we read text data in, too. In fact, we read in the website the same way we did when scraping a table.

```
library(rvest)
h <- read_html("https://www.state.gov/reports/2018-country-reports-on-human-rights-practices
```

The next step after this is to highlight the text that you want. There are a couple of ways to do this. One way is to dig around in the html code to find the right tag that isolates your text. This is sometimes not that difficult, but if you're not familiar with how html works (at least in principle), it can be pretty daunting to try to figure it out. The other option is to use the "selector gadget" tool: http://selectorgadget.com/. When I look at the state department website above and turn on the selector gadget tool, I isolated the block tha I wanted and hit the "xpath" button. It produce the following:



You can translate that information into R's language as follows:

```
text <- html_node(h, xpath = "//div[starts-with(@class, 'report__content')]") %>% html_text
```

Now, in the `text` object, is all of the text of the country report. In terms of getting the text in, this is all that is required.

16.2.1 Text (Pre-)Processing

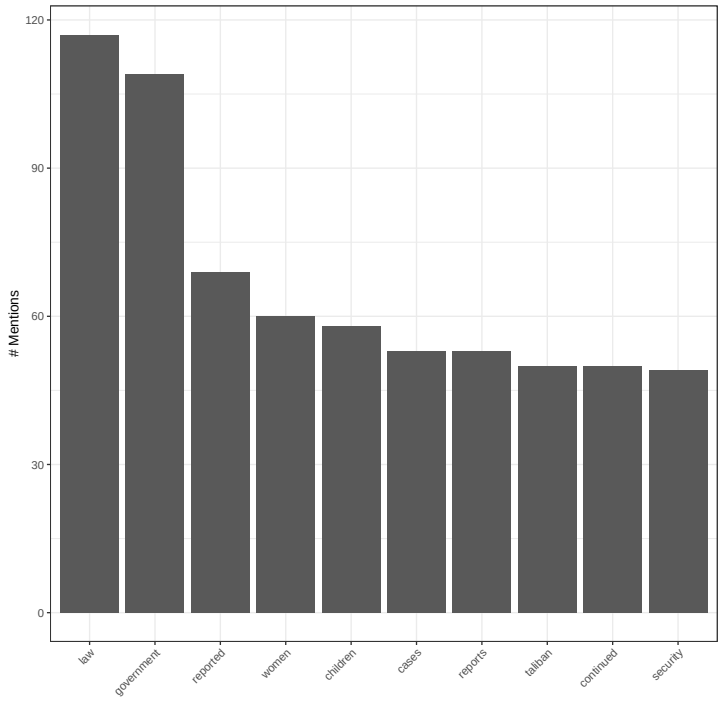
The next step in the process is to process the text and turn ito into a document-feature matrix. To do this, we'll use the `quanteda` package. R also has another package to deal with textual data (`tm`), but `quanteda` is a bit easier to work with. In the `quanteda` package, there is a function called `dfm` which produces a document-feature matrix from unstructured text. This function also does the text pre-processing, too. You can choose what pre-processing you do by specifying the following arguments:

- To stem the document use, `stem = TRUE`
- To remove stop words use, `remove = stopwords()`
- To remove numbers use, `remove_numbers = TRUE`
- To remove punctuation use, `remove_punct = TRUE`
- To remove symbols use, `remove_symbols = TRUE`,
- To keep n-grans of order higher than 1 use, `ngrams = 1:3` (e.g., to keep single words, bi-grams and tri-grams).
- To convert all letters to lower case use, `tolower = TRUE`

```
library(quanteda)
dfmat <- dfm(text,
    tolower=TRUE, remove=stopwords(),
    remove_numbers=TRUE, remove_punct=TRUE,
    remove_symbols=TRUE, ngrams=1)
topfeatures(dfmat, n=10)

##        law government   reported      women   children      cases
##        117        109         69         60         58         53
##    reports    taliban  continued   security
##         53         50         50         49
```

Using what we learned earlier, we could make a bar plot of the results.

```
library(tibble)
x <- sort(topfeatures(dfmat, n=10), decreasing=T)
tmp <- tibble(
    label = factor(names(x), levels=names(x)),
    y=x)
g <- ggplot(data=tmp, aes(x=label, y=y))
g + geom_bar(stat="identity") + xlab("") +
    theme_bw() + ylab("# Mentions") +
    # here, we can turn the axis tick-mark
    # labels to 45 degrees to prevent
    # overplotting.
    theme(axis.text.x =
        element_text(angle = 45, hjust = 1))
```



One of the nice things about the `quanteda` package is that it includes both supervised (e.g., naive bayes classification, word scores) and unsupervised (e.g., wordfish) that can be applied to text. We could go into this a bit more if you want, but I'll leave it there for now.

If we wanted to bring in all of the country reports,

16.3 Loops

The ultimate in flexibility in repeated calculations is a loop. Loops will work in all cases where apply works and in some cases where it doesn't. You will often times hear about what massive speed increases are to be reaped from using apply instead of a loop. While this can certainly be true, it is not necessarily true. There are instances where apply is actually slower than a loop. I don't have a particularly good feel for when one will be faster than the other, but so far as I can tell from my experience, when the function is manipulating a lot of data, meaning that a lot has to be stored in memory, then loops might actually be faster.

The `for` loop is a very useful tool when dealing with aggregating over units or performing operations multiple times either on the same set of data or on different sets of data. To show the basic structure of a loop, consider the following example:

```
n <- c("one", "two", "three", "four")
for(i in 1:4){
    # print the ith value of the
    # n-vector of words.
    print(n[i])
}

## [1] "one"
## [1] "two"
## [1] "three"
## [1] "four"
```

Here, the solitary character `i` holds the place for the numbers 1, 2, 3, and 4 in turn. It would be equivalent to do the following:

```
n[1]

## [1] "one"

n[2]

## [1] "two"

n[3]

## [1] "three"

n[4]

## [1] "four"
```

Though you often see `i` used as an index, you could perform the same task with:

```
for(fred in 1:4){
    print(n[fred])
}

## [1] "one"
## [1] "two"
## [1] "three"
## [1] "four"
```

There is another loop structure called `while` which will iterate until a predefined condition is met. The `for` loop is best when the number of iterations is know (i.e., you want to do something exactly n times) and `while` is best when you need to converge on a solution. For example, let's say we wanted to know how long it would take us drawing random poisson variates (with mean 3) such that the sum of draws was at least 100.

```
i <- 0
sumpois <- 0
while(sumpois < 100){
    # increment the counter
    i <- i+1
    # add a random poisson(3) draw to the
    # previous sumpois objet
    sumpois <- sumpois + rpois(1, 3)
}
i

## [1] 38
```

If we did the above while loop over and over again, we would find different answers. So, to be as efficient as possible, we want to only go as far as we need to.

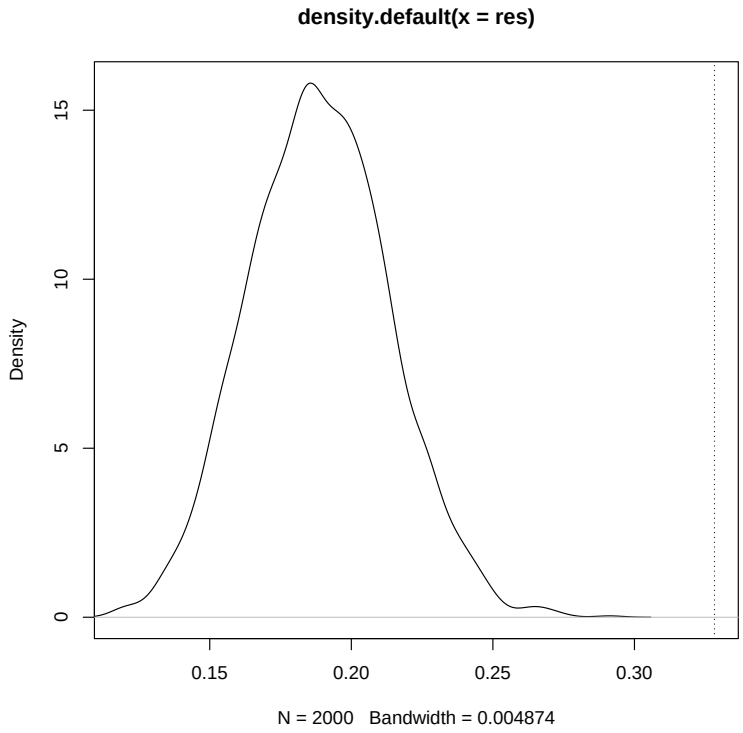
16.3.1 Example: Permutation Test of Significance for Cramer's V.

A more practical (and certainly useful) application of the loop is in simulation. Let's say that you had a cross-tabulation (in this case, of `nation` and `sector` from the Ornstein data) and we want to calculate Cramer's V and figure out whether it is significant. Perhaps we don't know the sampling distribution of V. We could use a permutation test to figure it out. What we want to do is to keep `nation` as it is and randomly assign values of `sector`, so that we know there is independence between the two variables. Then we calculate V and save it. We repeat this process many times. Then we can see where our observed statistic falls in this sampling distribution we are making.

```
library(vcd)
x <- Ornstein$sector
y <- Ornstein$nation
res <- rep(NA, 2000)
for(i in 1:2000){
    # make a temporary x value that is
    # our original x-values randomly
    # rearranged.
    tmpX <- sample(x, length(x),replace=F)
    # find the cramer's v from the
    # original y and the reordered x;
    # store the result in the ith value of res
    res[i] <- assocstats(table(tmpX, y))$cramer
}
mean(res > assocstats(table(x,y))$cramer)

## [1] 0
```

Or, we could make a graph:

```
plot(density(res), xlim=range(c(res, assocstats(table(x,y))$cramer)))
abline(v=assocstats(table(x,y))$cramer, lty=3)
```



16.4 Loops Example: Web Spidering

One of the things you might want to be able to do is find and follow links. We can find all of the links in a particular page as follows:

```
h1 <- read_html("https://www.state.gov/reports/2018-country-reports-on-human-rights-practice
links <- html_nodes(h1, xpath = "//option[starts-with(@value, 'https://www.state.gov/reports
countries <- html_nodes(h1, xpath = "//option[starts-with(@value, 'https://www.state.gov/rep
```

The next step is to loop through all of the links and get the text from their pages. We can use the same functions that we used above to grab the text.

```
texts <- list()

for(i in 1:length(links)){

    ## read in the url in the ith link
    h1 <- read_html(links[i])

    ## pull the text from the <div> that has a class that starts with 'report_content'
    texts[[i]] <- html_node(h1, xpath = "//div[starts-with(@class, 'report__content')]") %>%
}
names(texts) <- countries
```

The next thing we need to do is to create the document feature matrix. To do this, we need to change the list of texts to a vector of texts. Then, we can use the `dfm` function to process the data

```
texts <- do.call("c", texts)
library(quanteda)
dfmat <- dfm(texts,
    tolower=TRUE, remove=stopwords(),
    remove_numbers=TRUE, remove_punct=TRUE,
    remove_symbols=TRUE, ngrams=1)
```

After this, you could move on to whatever other analysis you wanted.

16.5 If-then Statements

There are two ways to do if-then statements - the `ifelse` function and two separate statements `if()` and `else`. The `ifelse` is a vectorized function, so it operates on a vector. Here's a simple example:

```
x <- c(1, 2, 3,4,5)
ifelse(x < 3, 0, 1)

## [1] 0 0 1 1 1
```

There are three arguments to `ifelse`, The first is an expression that is evaluated. The second argument is the value to return if the expression evaluates to `TRUE` and the third argument is what to return if the statement if `FALSE`. You can also use the variable in the expression:

```
x <- c(1, 2, 3,4,5)
ifelse(x < 3, 0, x)

## [1] 0 0 3 4 5
```

This only works if you're what you want to do is return a scalar (a single value) for each value of the original value. If you want to do something more complicated, you can use separate `if()` and `else` statements (though, you don't always have to use an `else` if you use an `if`). For example, imagine that we wanted to find out if an observation is missing and if it is, we want to fill it in with a random draw from "similar observations".

```
library(car)
tmp <- Duncan[,c("income", "type")]
tmp$income[c(3,40)] <- NA
tmp$income_imp <- tmp$income
for(i in 1:nrow(tmp)){
    # evaluate whether the ith value of
    # tmp£income is missing
    if(is.na(tmp$income[i])){
        # if it is missing, fill in
        # the ith value of tmp£income_imp
        # with a randomly drawn value using
        # sample.
        tmp$income_imp[i] <- sample(
            # the values you're sampling are
```

```
            # the valid values of income from
            # occupations with the same type.
            na.omit(tmp$income[which(tmp$type == tmp$type[i])]), 1)
    # close the if loop
    }
    # open the else loop
    else{
    # here, the else loop is blank because we
    # don't want to do anything if the ith value
    # of tmp£income isn't missing.  We could just
    # as easily omitted the else block altogether.
    # close the else block
    }
}
```

Notice that the `else` condition is blank. We could just have easily (in fact more easily) have just left out the `else` condition and it would be equivalent.

17 Repeated Calculations

Here, we're going to talk about doing lots of calculations at once. These operations can range from quite simple to quite complicated. In fact, we'll generally be talking about executing some function multiple times.

17.1 apply and its relatives

The `apply` function performs a function for some *margin* of a matrix (or array). The command is as follows:

```
apply(data, margin, function)
```

Where `data` is the matrix, `margin` is 1 for rows and 2 for columns and `function` is some function that is either defined within the command or some function that is defined elsewhere that is called in the function. I'll show a couple of examples below.

```
set.seed(10)
mat <- matrix(runif(25,1,5), ncol=5)
colmeans <- apply(mat, 2, mean)
colmeans

## [1] 2.615514 2.454218 2.829502 2.583852 3.413282

rowmeans <- apply(mat, 1, mean)
rowmeans

## [1] 3.142481 2.453034 2.481607 3.127373 2.691873
```

Notice that this is much faster than specifying a loop over the five columns. You can read the second line above as

- We want to apply the function `mean` to the columns (`margin = 2`) to the matrix `mat`.

The fourth line can be read similarly, though the margin changes to 1, so we are applying the function to the rows instead of the columns.

What if there are some missing values in the matrix? If this happens, the mean function will return `NA` for the row or column that contains the missing value unless `na.rm=T` is offered as an argument to the command `mean`. You can do this in apply. The additional arguments come after the function name. For example:

```
colmeans <- apply(mat, 2, mean, na.rm=T)
rowmeans <- apply(mat, 1, mean, na.rm=T)
```

Equivalently, this could be done with:

```
colmeans <- apply(mat, 2, function(z)mean(z))
colmeans

## [1] 2.615514 2.454218 2.829502 2.583852 3.413282

rowmeans <- apply(mat, 1, function(z)mean(z))
rowmeans

## [1] 3.142481 2.453034 2.481607 3.127373 2.691873
```

These exampled are a bit contrived because there are functions called `colMeans` and `rowMeans` that do the same. However, replacing `mean` with `median` would grab the column or row medians. This can also be any function that returns a vector.

```
library(car)
apply(Duncan[,c("income", "education", "prestige")],
    2, quantile, c(.25,.5, .75), na.rm=T)

##     income education prestige
## 25%     21        26       16
## 50%     42        45       41
## 75%     64        84       81
```

17.1.1 by

The `by` command is very flexible about both input data and about what kind of objects get returned. The `by` command can take a a matrix or data frame as its first argument. As suggested above, `by` returns a list rather than a data frame, but I'll show you below how we can change it back to a matrix.

```
by1 <- by(Duncan$education, list(Duncan$type), mean)
by1
```

```
## : prof
## [1] 81.33333
## ------------------------------------------------------------
## : bc
## [1] 25.33333
## ------------------------------------------------------------
## : wc
## [1] 61.5
```

We can make this into a vector simply by typing:

```
by1.vec <- c(by1)
by1.vec

##     prof       bc       wc
## 81.33333 25.33333 61.50000
```

One of the benefits of the `by` command is it allows you to provide a matrix as the first argument to the command and allows the function to perform matrix operations. One simple thing we might want to do is take the mean of three columns of a matrix for the values of another variable. Specifically, we might want to know the means of prestige, education and income for the values of type in the Duncan dataset.

```
by2 <- by(Duncan[,c("prestige", "income", "education")],
    list(Duncan$type),
    function(x)apply(x, 2, mean))
by2

## : prof
##  prestige    income education
##  80.44444  60.05556  81.33333
## ------------------------------------------------------------
## : bc
##  prestige    income education
##  22.76190  23.76190  25.33333
## ------------------------------------------------------------
## : wc
##  prestige    income education
##  36.66667  50.66667  61.50000
```

Now, what if we want this to be a 3×3 matrix?

```
mat <- do.call(rbind, by2)
mat

##      prestige   income education
## prof 80.44444 60.05556  81.33333
## bc   22.76190 23.76190  25.33333
## wc   36.66667 50.66667  61.50000
```

We could also do something more complicated, like estimate a linear model for all of the different values of a categorical variable.

```
mods <- by(Duncan, list(Duncan$type),
    function(x)lm(prestige ~ income + education, data=x))

mods

## : prof
##
## Call:
## lm(formula = prestige ~ income + education, data = x)
##
## Coefficients:
## (Intercept)       income    education
##     28.0573       0.4143       0.3382
##
## ------------------------------------------------------------
## : bc
##
## Call:
## lm(formula = prestige ~ income + education, data = x)
##
## Coefficients:
## (Intercept)       income    education
##     -3.9505       0.7834       0.3196
##
## ------------------------------------------------------------
## : wc
##
## Call:
## lm(formula = prestige ~ income + education, data = x)
##
## Coefficients:
## (Intercept)       income    education
##    -10.9937       0.4231       0.4264

summary(mods[[1]])

##
## Call:
## lm(formula = prestige ~ income + education, data = x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -15.338  -5.216  -0.416   5.920  21.833
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept)  28.0573    12.9430   2.168   0.0467 *
## income        0.4143     0.1637   2.530   0.0231 *
## education     0.3382     0.1590   2.127   0.0504 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.1 on 15 degrees of freedom
## Multiple R-squared:  0.5478,Adjusted R-squared:  0.4875
## F-statistic: 9.086 on 2 and 15 DF,  p-value: 0.002599
```

17.1.2 List Apply Functions

There are two different apply functions that operate on lists. They are essentially the same function, but differ in the nature of the output. The `lapply` function executes a specified function on every element of the input list and it always returns a list, regardless of the simplicity of the output. The `sapply` function does similar things as `lapply`, but returns the simplest possible data structure. For example, if it can return a matrix instead of a list, it will. See below.

```
lapply(mods, coef)

## $prof
## (Intercept)      income   education
##  28.0572630   0.4142683   0.3382139
##
## $bc
## (Intercept)      income   education
##  -3.9505429   0.7834109   0.3196229
##
## $wc
## (Intercept)      income   education
## -10.9937457   0.4231025   0.4263938

sapply(mods, coef)

##                    prof         bc          wc
## (Intercept) 28.0572630 -3.9505429 -10.9937457
## income       0.4142683  0.7834109   0.4231025
## education    0.3382139  0.3196229   0.4263938
```

18 Basic Function Writing

There are a few different situations in which we might want to write our own functions. First, we might want to be able to execute multiple commands with a single command. Second, we might want to use **R**'s ability to do repeated calculations with `apply`, `by` or `for` to do something many times with a single command. Finally, we might want to make complicated graphs with the `lattice` package which will require writing a function.

The function `function()` in **R** is what you need to write functions. The most important thing when writing a function is that you know what you want **R** to do. The *very* basic structure of a function is

```
myfun <- function(x){
    do something to/with/using x
}
```

Not surprisingly, it's the `do something` part that you need to come up with. We will start out with some relatively simple examples, but these can get arbitrarily complicated.

18.1 Example: Calculating a Mean

To give a simple example, let's think about making a function that calculates the mean (not that we need to, but just because we can). We need to get the sum of a variable and the number of observations in the variable to make the mean

```
mymean <- function(x){
        # calculate the sum
        sum.x <- sum(x, na.rm=T)
        # find the N by counting up all of the non-missing entries
        n.x <- sum(!is.na(x))
        # calculate the mean as the sum divided by n
        mean.x <- sum.x/n.x
        # print the mean
        mean.x
}
```

Notice here, that we are making the function take a single argument `x`. Whenever `x` shows up in the "guts" of the function, it will stand in for the variable whose mean we are trying to calculate.

```
z <- 1:6
mymean(z)

## [1] 3.5

mean(z)

## [1] 3.5
```

18.2 Changing Existing Function Defaults

You might want to know if you can make a function that sets parameters of another function so we don't have to. Let's take, for instance, the `CrossTable` function from the `gmodels` package. Let's say that we only want column percentages and cell counts along with the chi-squared statistic and it's p-value. If we look at the help file for the `CrossTable` command, we can see what arguments we need to set.

```
CrossTable(x, y, digits=3, max.width = 5, expected=FALSE, prop.r=TRUE,
           prop.c=TRUE, prop.t=TRUE, prop.chisq=TRUE, chisq = FALSE,
           fisher=FALSE, mcnemar=FALSE, resid=FALSE, sresid=FALSE,
           asresid=FALSE, missing.include=FALSE,format=c("SAS","SPSS"),
           dnn = NULL, ...)
```

Notice here, we will probably want to change some of the options that are `TRUE` by default to `FALSE`.

In **R**, the defaults are set with the equal sign (`=`) in the command. For example in the `CrossTable` command, `x` and `y` have no defaults. That is to say, the command will not substitute a value for you if you do not provide arguments `x` and `y`. However, if you do not specify the `digits` argument, it defaults to `3`. The argument `prop.r` defaults to `TRUE`, but we want to set it to `FALSE`.

```
CrossTable2 <- function(x,y){
require("gmodels")
    CrossTable(x=x,y=y, prop.r=F, prop.t=F, prop.chisq=F,chisq=T,
        format="SPSS")
}
```

If we run the command now, we can see the results:

```
library(car)
data(Duncan)
Duncan$ed.cat <- cut(Duncan$education, 2)
CrossTable2(Duncan$ed.cat, Duncan$type)
```

```
##
##    Cell Contents
## |-------------------------|
## |                   Count |
## |          Column Percent |
## |-------------------------|
##
## Total Observations in Table:  45
##
##              | y
##            x |       bc  |     prof  |       wc  | Row Total |
## -------------|-----------|-----------|-----------|-----------|
##  (6.91,53.5] |       21  |        2  |        2  |       25  |
##              |  100.000% |   11.111% |   33.333% |           |
## -------------|-----------|-----------|-----------|-----------|
##   (53.5,100] |        0  |       16  |        4  |       20  |
##              |    0.000% |   88.889% |   66.667% |           |
## -------------|-----------|-----------|-----------|-----------|
## Column Total |       21  |       18  |        6  |       45  |
##              |   46.667% |   40.000% |   13.333% |           |
## -------------|-----------|-----------|-----------|-----------|
##
##
## Statistics for All Table Factors
##
##
## Pearson's Chi-squared test
## ------------------------------------------------------------
## Chi^2 =  32.4     d.f. =  2     p =  9.213601e-08
##
##
##
##        Minimum expected frequency: 2.666667
## Cells with Expected Frequency < 5: 2 of 6 (33.33333%)
```

You can see that we've controlled the output and presented exactly what we wanted. Notice that we've simply included `x` and `y` as arguments and we pass them to the original command. It's a bit more explicit what is going on if we do the following:

There are two other pieces of this command.

- `require('gmodels')` - does the following. If the `gmodels` library is loaded, then it does nothing. If it is not loaded, it loads it.
- `print(xt)` prints the object that was created on the previous line.

18.3 .First and .Last functions in R.

Some people have asked how they can get R to load packages that they use frequently automatically on start. This is pretty easy, but requires a tiny little bit of programming. First you need to have a file in your home directory (or whatever directory R starts in)[7] The file's name should be `.RProfile`. Notice, that nothing comes before the extention; the file name starts with a dot. In that file, you can set preferences and generally tell R how you want it to be initialized when it opens. There are two functions you can use in the `.RProfile` file called `.First` and `.Last` which tell R what to do when you open R initially and close R, respectively.

Let's say that I wanted to load the `foreign` package every time I started R. My `.RProfile` function would look something like this:

```
.First <- function(){
    # load a package; you could also do
    # this with the defaultPackages option.
        library(foreign)
        # set the working directory automatically
    Bdir <- "C:/Users/armstrod/Dropbox/IntroR/Boulder/"
}
```

You could also set options like default paper size, use of fancy quotation marks, etc..., but you don't have to do that. The `.RProfile` file can be as long or short as you want. The `.Last` function is specified in exactly the same way, but it executes automatically on closing, rather than opening, R.

[7] You can find this information by opening R and immediately typing `getwd()`; this will tell you where the home directory is.