Analytics 1 BADM 371

Tobin Turner

2021-01-12

Contents

Chapter 1

Before we start...

This is a book designed to support BADM 371 Analytics 1. It will serve (*I truly hope*) as a living document. See an opportunity to make something better or clearer or more instructive? Tell me.

Chapter 2

Welcome, friend.

2.1 Is your life is about to change? Yes. YES.

Let's chat.

2.2 Where does this course fit in the big picture?

This course can serves as a first course in an undergraduate analytics very similar to data science) and highly related to statistics. More importantly my hope is that this course is *a gateway to an appreciation of what data can do to change the (your!) world.*

2.3 Why R?

Unlike most other software designed specifically for teaching statistics, R is free and open source, powerful, flexible, and relevant beyond the introductory statistics classroom. Arguments against using and teaching R at especially the introductory statistics level generally cluster around the following two points: teaching programming in addition to statistical concepts is challenging and the command line is more intimidating to beginners than the graphical user interface (GUI) most point-and-click type software offer.

One solution for these concerns is to avoid hands-on data analysis completely. If we do not ask our students to start with raw data and instead always provide them with small, tidy rectangles of data then there is never really a need for statistical software beyond spreadsheet or graphing calculator. This is not what we want in a modern statistics course and is a disservice to students.

Another solution is to use traditional point-and-click software for data analysis. The typical argument is that the GUI is easier for students to learn and so they can spend more time on statistical concepts. However, this ignores the fact that these software tools also have nontrivial learning curves. In fact, teaching specific data analysis tasks using such software often requires lengthy step-by-step instructions, with annotated screenshots, for navigating menus and other interface elements. Also, it is not uncommon that instructions for one task do not easily extend to another. Replacing such instructions with just a few lines of R code actually makes the instructional materials more concise and less intimidating.

Many in the statistics education community are in favor of teaching R (or some other programming language, like Python) in upper level statistics courses, however the value of using R in introductory statistics courses is not as widely accepted. We acknowledge that this addition can be burdensome, however we would argue that learning a tool that is applicable beyond the introductory statistics course and that enhances students' problem solving skills is a burden worth bearing.

2.4 Why not language X?

There are a number of other great programming tools out there that can also be used for introducing students to data science, e.g. Python. These materials are designed for teaching data science with R. A great example of a similar curriculum using Python is Data 8 designed at University of California, Berkeley.

2.5 Why RStudio?

The RStudio IDE includes a viewable environment, a file browser, data viewer, and a plotting pane, which makes it less intimidating than the bare R shell. Additionally, since it is a full fledged IDE, it also features integrated help, syntax highlighting, and context-aware tab completion, which are all powerful tools that help flatten the learning curve. RStudio also has direct integration with other critically important tools for teaching computing best practices and reproducible research.

Our recommendation is that students access the RStudio IDE through a centralized RStudio server instance or using RStudio Cloud. We describe this in further detail in the Infrastructure section.

It should be noted that we do not want to completely dissuade students from downloading and installing R and RStudio locally, we just do not want it to be a prerequisite for getting started. We have found that teaching personal setup is best done progressively throughout a semester, usually via one-on-one

interactions during office hours or after class. Our goal is that all students will be able to continue using R in any setting.

2.6 Learn more

If you would like to learn more about the design philosophy behind the course as well as implementation details, we recommend the following paper that is freely available online.

> Mine Çetinkaya-Rundel & Victoria Ellison (In press), A fresh look at introductory data science, *Journal of Statistics Education*. doi.org/10.1080/10691898.2020.1804497.

Chapter 3

Get your R act together & Tools of the Trade

3.1 Install R and RStudio

3.1.1 R and RStudio

- Install R, a free software environment for statistical computing and graphics from [CRAN] https://cran.r-project.org/, the Comprehensive R Archive Network. I **highly recommend** you install a precompiled binary distribution for your operating system – use the links up at the top of the CRAN page linked above!

- Install RStudio's IDE (stands for *integrated development environment*), a powerful user interface for R. https://rstudio.com/products/rstudio/download/ Get the apprpriate Mac or Windows Edition of RStudio Desktop.

    - RStudio comes with a **text editor**, so there is no immediate need to install a separate stand-alone editor.
    - RStudio can **interface with Git(Hub)**. However, you must do all the Git(Hub) set up [described elsewhere][happy-git] before you can take advantage of this. This course will not cover github, but we can chat about it if you like.

If you have a pre-existing installation of R and/or RStudio, we **highly recommend** that you reinstall both and get as current as possible. It can be considerably harder to run old software than new.

- If you upgrade R, you will need to update any packages you have installed. The command below should get you started, though you may need to specify more arguments if, e.g., you have been using a non-default library for your packages.

```
update.packages(ask = FALSE, checkBuilt = TRUE)
```

Note: this will only look for updates on CRAN. So if you use a package that lives *only* on GitHub or if you want a development version from GitHub, you will need to update manually, e.g. via `devtools::install_github()`.

3.1.2 Testing testing

- Do whatever is appropriate for your OS to launch RStudio. You should get a window similar to the screenshot you see [here][rstudio-workbench], but yours will be more boring because you haven't written any code or made any figures yet!
- Put your cursor in the pane labelled Console, which is where you interact with the live R process. Create a simple object with code like `x <- 2 * 4` (followed by enter or return). Then inspect the `x` object by typing `x` followed by enter or return. You should see the value 8 print to screen. If yes, you've succeeded in installing R and RStudio.

3.1.3 Add-on packages

R is an extensible system and many people share useful code they have developed as a *package* via CRAN and GitHub. To install a package from CRAN, for example the [dplyr][dplyr-cran] package for data manipulation, here is one way to do it in the R console (there are others).

```
install.packages("dplyr", dependencies = TRUE)
```

By including `dependencies = TRUE`, we are being explicit and extra-careful to install any additional packages the target package, dplyr in the example above, needs to have around.

You could use the above method to install the following packages, all of which we will use:

- tidyr, [package webpage][tidyr-web]
- ggplot2, [package webpage][ggplot2-web]

Chapter 4

R basics and workflows

4.1 Basics of working with R at the command line and RStudio goodies

Launch RStudio/R.

Notice the default panes:

- Console (entire left)
- Environment/History (tabbed in upper right)
- Files/Plots/Packages/Help (tabbed in lower right)

FYI: you can change the default location of the panes, among many other things: [Customizing RStudio][rstudio-customizing].

Go into the Console, where we interact with the live R process.

Make an assignment and then inspect the object you just created:

```
x <- 3 * 4
x
```

```
## [1] 12
```

All R statements where you create objects – "assignments" – have this form:

```
objectName <- value
```

and in my head I hear, e.g., "x gets 12".

You will make lots of assignments and the operator `<-` is a pain to type. Don't be lazy and use `=`, although it would work, because it will just sow confusion later. Instead, utilize RStudio's keyboard shortcut: Alt + - (the minus sign).

Notice that RStudio automagically surrounds `<-` with spaces, which demonstrates a useful code formatting practice. Code is miserable to read on a good day. Give your eyes a break and use spaces.

RStudio offers many handy [keyboard shortcuts][rstudio-key-shortcuts]. Also, Alt+Shift+K brings up a keyboard shortcut reference card.

Object names cannot start with a digit and cannot contain certain other characters such as a comma or a space. You will be wise to adopt a [convention for demarcating words][wiki-snake-case] in names.

```
i_use_snake_case
other.people.use.periods
evenOthersUseCamelCase
```

Make another assignment:

```
this_is_a_really_long_name <- 2.5
```

To inspect this, try out RStudio's completion facility: type the first few characters, press TAB, add characters until you disambiguate, then press return.

Make another assignment:

```
turner_rocks <- 2 ^ 3
```

Let's try to inspect:

```
turnerrocks
```

```
## Error in eval(expr, envir, enclos): object 'turnerrocks' not found
```

```
Turner_rocks
```

```
## Error in eval(expr, envir, enclos): object 'Turner_rocks' not found
```

```
turner_rocks
```

```
## [1] 8
```

Implicit contract with the computer / scripting language: Computer will do tedious computation for you. In return, you will be completely precise in your instructions. Typos matter. Case matters. Get better at typing.

R has a mind-blowing collection of built-in functions that are accessed like so:

```
functionName(arg1 = val1, arg2 = val2, and so on)
```

Let's try using `seq()` which makes regular sequences of numbers and, while we're at it, demo more helpful features of RStudio.

Type `se` and hit TAB. A pop up shows you possible completions. Specify `seq()` by typing more to disambiguate or using the up/down arrows to select. Notice the floating tool-tip-type help that pops up, reminding you of a function's arguments. If you want even more help, press F1 as directed to get the full documentation in the help tab of the lower right pane. Now open the parentheses and notice the automatic addition of the closing parenthesis and the placement of cursor in the middle. Type the arguments `1, 10` and hit return. RStudio also exits the parenthetical expression for you. IDEs are great.

```
seq(1, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

The above also demonstrates something about how R resolves function arguments. You can always specify in `name = value` form. But if you do not, R attempts to resolve by position. So above, it is assumed that we want a sequence `from = 1` that goes `to = 10`. Since we didn't specify step size, the default value of `by` in the function definition is used, which ends up being 1 in this case. For functions I call often, I might use this resolve by position for the first argument or maybe the first two. After that, I always use `name = value`.

Make this assignment and notice similar help with quotation marks.

```
yo <- "hello world"
yo
```

```
## [1] "hello world"
```

If you just make an assignment, you don't get to see the value, so then you're tempted to immediately inspect.

```
y <- seq(1, 10)
y
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

This common action can be shortened by surrounding the assignment with parentheses, which causes assignment and "print to screen" to happen.

```
(y <- seq(1, 10))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Not all functions have (or require) arguments:

```
date()
```

```
## [1] "Tue Jan 12 14:18:22 2021"
```

Now look at your workspace – in the upper right pane. The workspace is where user-defined objects accumulate. You can also get a listing of these objects with commands:

```
objects()
```

```
## [1] "this_is_a_really_long_name" "turner_rocks"
## [3] "x"                          "y"
## [5] "yo"
```

```
ls()
```

```
## [1] "this_is_a_really_long_name" "turner_rocks"
## [3] "x"                          "y"
## [5] "yo"
```

If you want to remove the object named `y`, you can do this:

```
rm(y)
```

To remove everything:

```
rm(list = ls())
```

or click the broom in RStudio's Environment pane.

4.2 Workspace and working directory

One day you will need to quit R, go do something else and return to your analysis later.

One day you will have multiple analyses going that use R and you want to keep them separate.

One day you will need to bring data from the outside world into R and send numerical results and figures from R back out into the world.

To handle these real life situations, you need to make two decisions:

- What about your analysis is "real", i.e. will you save it as your lasting record of what happened?
- Where does your analysis "live"?

4.2.1 Workspace, `.RData`

As a beginning R user, it's OK to consider your workspace "real". *Very soon*, I urge you to evolve to the next level, where you consider your saved R scripts as "real". (In either case, of course the input data is very much real and requires preservation!) With the input data and the R code you used, you can reproduce *everything*. You can make your analysis fancier. You can get to the bottom of puzzling results and discover and fix bugs in your code. You can reuse the code to conduct similar analyses in new projects. You can remake a figure with different aspect ratio or save is as TIFF instead of PDF. You are ready to take questions. You are ready for the future.

If you regard your workspace as "real" (saving and reloading all the time), if you need to redo analysis … you're going to either redo a lot of typing (making mistakes all the way) or will have to mine your R history for the commands you used. Rather than [becoming an expert on managing the R history][rstudio-command-history], a better use of your time and psychic energy is to keep your "good" R code in a script for future reuse.

Because it can be useful sometimes, note the commands you've recently run appear in the History pane.

But you don't have to choose right now and the two strategies are not incompatible. Let's demo the save / reload the workspace approach.

Upon quitting R, you have to decide if you want to save your workspace, for potential restoration the next time you launch R. Depending on your set up, R or your IDE, e.g. RStudio, will probably prompt you to make this decision.

Quit R/RStudio, either from the menu, using a keyboard shortcut, or by typing `q()` in the Console. You'll get a prompt like this:

Save workspace image to ~/.Rdata?

Note where the workspace image is to be saved and then click "Save".

Using your favorite method, visit the directory where image was saved and verify there is a file named `.RData`. You will also see a file `.Rhistory`, holding the commands submitted in your recent session.

Restart RStudio. In the Console you will see a line like this:

```
[Workspace loaded from ~/.RData]
```

indicating that your workspace has been restored. Look in the Workspace pane and you'll see the same objects as before. In the History tab of the same pane, you should also see your command history. You're back in business. This way of starting and stopping analytical work will not serve you well for long but it's a start.

4.2.2 Working directory

Any process running on your computer has a notion of its "working directory". In R, this is where R will look, by default, for files you ask it to load. It also where, by default, any files you write to disk will go. Chances are your current working directory is the directory we inspected above, i.e. the one where RStudio wanted to save the workspace.

You can explicitly check your working directory with:

```
getwd()
```

It is also displayed at the top of the RStudio console.

As a beginning R user, it's OK let your home directory or any other weird directory on your computer be R's working directory. *Very soon*, I urge you to evolve to the next level, where you organize your analytical projects into directories and, when working on project A, set R's working directory to the associated directory.

Although I do not recommend it, in case you're curious, you can set R's working directory at the command line like so:

```
setwd("~/myCoolProject")
```

Although I do not recommend it, you can also use RStudio's Files pane to navigate to a directory and then set it as working directory from the menu: *Session > Set Working Directory > To Files Pane Location*. (You'll see even

more options there). Or within the Files pane, choose "More" and "Set As Working Directory".

But there's a better way. A way that also puts you on the path to managing your R work like an expert.

4.3 RStudio projects

Keeping all the files associated with a project organized together – input data, R scripts, analytical results, figures – is such a wise and common practice that RStudio has built-in support for this via its [*projects*][rstudio-using-projects].

Let's make one to use for the rest of this workshop/class. Do this: *File > New Project...*. The directory name you choose here will be the project name. Call it whatever you want (or follow me for convenience).

I created a directory and, therefore RStudio project, called `swc` in my `tmp` directory, FYI.

```
setwd("~/tmp/swc")
```

Now check that the "home" directory for your project is the working directory of our current R process:

```
getwd()
```

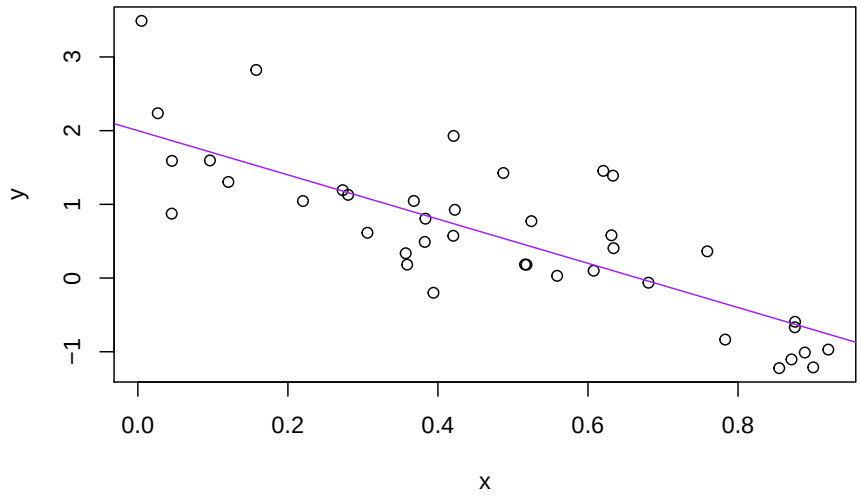
I can't print my output here because this document itself does not reside in the RStudio Project we just created.

Let's enter a few commands in the Console, as if we are just beginning a project:

```
a <- 2
b <- -3
sig_sq <- 0.5
x <- runif(40)
y <- a + b * x + rnorm(40, sd = sqrt(sig_sq))
(avg_x <- mean(x))
```

```
## [1] 0.4806322
```

```
write(avg_x, "avg_x.txt")
plot(x, y)
abline(a, b, col = "purple")
```




```
dev.print(pdf, "toy_line_plot.pdf")
```

```
## pdf
##   2
```

Let's say this is a good start of an analysis and your ready to start preserving the logic and code. Visit the History tab of the upper right pane. Select these commands. Click "To Source". Now you have a new pane containing a nascent R script. Click on the floppy disk to save. Give it a name ending in `.R` or `.r`, I used `toy-line.r` and note that, by default, it will go in the directory associated with your project.

Quit RStudio. Inspect the folder associated with your project if you wish. Maybe view the PDF in an external viewer.

Restart RStudio. Notice that things, by default, restore to where we were earlier, e.g. objects in the workspace, the command history, which files are open for editing, where we are in the file system browser, the working directory for the R process, etc. These are all Good Things.

Change some things about your code. Top priority would be to set a sample size `n` at the top, e.g. `n <- 40`, and then replace all the hard-wired 40's with `n`. Change some other minor-but-detectable stuff, e.g. alter the sample size `n`, the slope of the line `b`,the color of the line … whatever. Practice the different ways to re-run the code:

- Walk through line by line by keyboard shortcut (Command+Enter) or mouse (click "Run" in the upper right corner of editor pane).
- Source the entire document – equivalent to entering `source('toy-line.r')` in the Console – by keyboard shortcut (Shift+Command+S) or mouse (click "Source" in the upper right corner of editor pane or select from the mini-menu accessible from the associated down triangle).
- Source with echo from the Source mini-menu.

Visit your figure in an external viewer to verify that the PDF is changing as you expect.

In your favorite OS-specific way, search your files for `toy_line_plot.pdf` and presumably you will find the PDF itself (no surprise) but *also the script that created it (`toy-line.r`)*. This latter phenomenon is a huge win. One day you will want to remake a figure or just simply understand where it came from. If you rigorously save figures to file **with R code and not ever ever ever the mouse or the clipboard**, you will sing my praises one day. Trust me.

4.4 Stuff

It is traditional to save R scripts with a `.R` or `.r` suffix. Follow this convention unless you have some extraordinary reason not to.

Comments start with one or more `#` symbols. Use them. RStudio helps you (de)comment selected lines with Ctrl+Shift+C (Windows and Linux) or Command+Shift+C (Mac).

Clean out the workspace, i.e. pretend like you've just revisited this project after a long absence. The broom icon or `rm(list = ls())`. Good idea to do this, restart R (available from the Session menu), re-run your analysis to truly check that the code you're saving is complete and correct (or at least rule out obvious problems!).

This workflow will serve you well in the future:

- Create an RStudio project for an analytical project
- Keep inputs there (we'll soon talk about importing)
- Keep scripts there; edit them, run them in bits or as a whole from there
- Keep outputs there (like the PDF written above)

Avoid using the mouse for pieces of your analytical workflow, such as loading a dataset or saving a figure. Terribly important for reproducibility and for making it possible to retrospectively determine how a numerical table or PDF

was actually produced (searching on local disk on filename, among `.R` files, will lead to the relevant script).

Many long-time users never save the workspace, never save `.RData` files (I'm one of them), never save or consult the history. Once/if you get to that point, there are options available in RStudio to disable the loading of `.RData` and permanently suppress the prompt on exit to save the workspace (go to *Tools* > *Options* > *General*).

For the record, when loading data into R and/or writing outputs to file, you can always specify the absolute path and thereby insulate yourself from the current working directory. This is rarely necessary when using RStudio projects properly.

Chapter 5

Conquering R

5.1 Summaries and Subscripting

Let's suppose we've collected some data from an experiment and stored them in an object x:

```
x<-c(7.5,8.2,3.1,5.6,8.2,9.3,6.5,7.0,9.3,1.2,14.5,6.2)
```

Some simple summary statistics of these data can be produced:

```
mean(x)
```

```
## [1] 7.216667
```

```
var(x)
```

```
## [1] 11.00879
```

```
sd(x)
```

```
## [1] 3.317949
```

```
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.200   6.050   7.250   7.217   8.475  14.500
```

which should all be self explanatory.

It may be, however, that we subsequently learn that the first 6 data correspond to measurements made on one machine, and the second six on another machine.

This might suggest summarizing the two sets of data separately, so we would need to extract from x the two relevant subvectors. This is achieved by subscripting:

```
x[1:6]
```

```
## [1] 7.5 8.2 3.1 5.6 8.2 9.3
```

```
x[7:12]
```

```
## [1]  6.5  7.0  9.3  1.2 14.5  6.2
```

```
summary(x[1:6])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   3.100   6.075   7.850   6.983   8.200   9.300
```

```
summary(x[7:12])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.200   6.275   6.750   7.450   8.725  14.500
```

Other subsets can be created in the obvious way. For example:

```
x[c(2,4,9)]
```

```
## [1] 8.2 5.6 9.3
```

Negative integers can be used to exclude particular elements. For example

```
x[-(1:6)]
```

```
## [1]  6.5  7.0  9.3  1.2 14.5  6.2
```

has the same effect as

```
x[7:12]
```

```
## [1]  6.5  7.0  9.3  1.2 14.5  6.2
```

5.1.1 Exercises

1. If x<- c(5,9,2,3,4,6,7,0,8,12,2,9) decide what each of the following is and use R to check your answers:

(a) x[2]
(b) x[2:4]
(c) x[c(2,3,6)]
(d) x[c(1:5,10:12)]
(e) x[-(10:12)]

2. The data y<-c(33,44,29,16,25,45,33,19,54,22,21,49,11,24,56) contain sales of milk in gallons for 5 days in three different shops (the frst 3 values are for shops 1,2 and 3 on Monday, etc.) Produce a statistical summary of the sales for each day of the week and also for each shop.

5.1.2 Matrices

Matrices can be created in R in a variety of ways. Perhaps the simplest is to create the columns and then glue them together with the command cbind. For example, Matrices can be created in R in a variety of ways. Perhaps the simplest is to create the columns and then glue them together with the command cbind. For example,

```
x<-c(5,7,9)
y<-c(6,3,4)
z<-cbind(x,y)
z
```

```
##      x y
## [1,] 5 6
## [2,] 7 3
## [3,] 9 4
```

The dimension of a matrix can be checked with the dim command:

```
dim(z)
```

```
## [1] 3 2
```

[1] 3 2 i.e., three rows and two columns. There is a similar command, rbind, for building matrices by gluing rows together.

The functions cbind and rbind can also be applied to matrices themselves (provided the dimensions match) to form larger matrices. For example,

```
rbind(z,z)
```

```
##      x y
## [1,] 5 6
## [2,] 7 3
## [3,] 9 4
## [4,] 5 6
## [5,] 7 3
## [6,] 9 4
```

Matrices can also be built by explicit construction via the function matrix. For example,

```
z<-matrix(c(5,7,9,6,3,4),nrow=3)
```

results in a matrix z identical to z above. Notice that the dimension of the matrix is determined by the size of the vector and the requirement that the number of rows is 3, as specified by the argument nrow=3. As an alternative we could have specified the number of columns with the argument ncol=2 (obviously, it is unnecessary to give both). Notice that the matrix is 'flled up' column-wise. If instead you wish to fill up row-wise, add the option byrow=T. For example,

```
z<-matrix(c(5,7,9,6,3,4),nr=3,byrow=T)
z
```

```
##      [,1] [,2]
## [1,]    5    7
## [2,]    9    6
## [3,]    3    4
```

Notice that the argument nrow has been abbreviated to nr. Such abbreviations are always possible for function arguments provided it induces no ambiguity - if in doubt always use the full argument name.

As usual, R will try to interpret operations on matrices in a natural way. For example, with z as above, and

```
y<-matrix(c(1,3,0,9,5,-1),nrow=3,byrow=T)
y
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    0    9
## [3,]    5   -1
```

we obtain

```
y+z
```

```
##      [,1] [,2]
## [1,]    6   10
## [2,]    9   15
## [3,]    8    3
```

and

```
y*z
```

```
##      [,1] [,2]
## [1,]    5   21
## [2,]    0   54
## [3,]   15   -4
```

Other useful functions on matrices are to transpose a matrix:

```
z
```

```
##      [,1] [,2]
## [1,]    5    7
## [2,]    9    6
## [3,]    3    4
```

```
t(z)
```

```
##      [,1] [,2] [,3]
## [1,]    5    9    3
## [2,]    7    6    4
```

As with vectors it is useful to be able to extract sub-components of matrices. In this case, we may wish to pick out individual elements, rows or columns. As before, the [ ] notation is used to subscript. The following examples should make things clear:

```
z[1,1]
```

```
## [1] 5
```

```
z[c(2,3),2]
```

```
## [1] 6 4
```

```
z[,2]
```

```
## [1] 7 6 4
```

```
z[1:2,]
```

```
##      [,1] [,2]
## [1,]    5    7
## [2,]    9    6
```

So, in particular, it is necessary to specify which rows and columns are required, whilst omitting the integer for either dimension implies that every element in that dimension is selected.

5.1.3 Exercises

1. Create this matrix in R

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    7    8   11   -5
## [2,]    3    8    6    3   -9
## [3,]    0   11   14   14   14
```

2. Create in R these matrices:

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    8   11
## [3,]    5    9
```

```
##      [,1] [,2]
## [1,]    6    8
## [2,]    2    1
## [3,]    1   -7
```

3. Calculate the following and check your answers in R:

(a) 2*x
(b) x*x
(c) y%*%x
(d) x%*%y
(e) t(y)
(f) solve(x)

```
##      [,1] [,2]
## [1,]    2   14
## [2,]   16   22
## [3,]   10   18
```

```
##      [,1] [,2]
## [1,]    1   49
## [2,]   64  121
## [3,]   25   81
```

```
##      [,1] [,2] [,3]
## [1,]    6    2    1
## [2,]    8    1   -7
```

4. With x and y as above, calculate the effect of the following subscript operations and check your answers in R.

(a) x[1,]
(b) x[2,]
(c) x[,2]
(d) y[1,2]
(e) y[,2:3]

Chapter 6

Final Words

We have finished a nice book.

Chapter 7

Lab 1 – GDH

GDH provides ice cream for its wonderful customers. I LOVE GDH. Do you love it as much as me (let's discuss)?

In the last three years GDH used ice cream, in pounds, by month, as shown in the attached file.

```
##  Month.Name year1 year2 year3
##         Jan    60    67    64
##         Feb    68    67    72
##         Mar    83    62    61
##         Apr   102    95   107
##         May    95   105   101
##         Jun    57    89    75
##         Jul    61    57    81
##         Aug   109   109   104
##         Sep    56    86    88
##         Oct    53    53    65
##         Nov    74    72    72
##         Dec    73    64    65
```

Please do the following.

GDH provides ice cream cones for its customers. In the last three years GDH used ice cream, in pounds, by month, as shown in the attached file.

1. In R, create the above data frame and name it ice.cream
2. What is another way you could have created the same data set?
3. Using R, what is the mean using for the months of Feb and Oct?
4. Create a chart showing ice cream use over time.
5. Which year used the most ice cream?

6. Which month has the highest standard deviation of ice cream use?
7. Which year has the highest standard deviation of ice cream use?
8. Also, you May want to check out this link to look at something called dataframes that may help with this assignment (but is not absolutely necessary) https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/data.frame

Chapter 8

Introduction to linear models

Linear regression is a very powerful statistical technique. Many people have some familiarity with regression just from reading the news, where straight lines are overlaid on scatterplots. Linear models can be used for prediction or to evaluate whether there is a linear relationship between two numerical variables.

Residual Analysis in Linear Regression

Linear regression is a statistical method for for modelling the linear relationship between a dependent variable y (i.e. the one we want to predict) and one or more explanatory or independent variables(X).

This vignette (https://rpubs.com/iabrady/residual-analysis) will explain how residual plots generated by the regression function can be used to validate that some of the assumptions that are made about the dataset indicating it is suitable for a linear regression are met.

If you have ever wondered what these mean and how they can help - this is a small guide!

There are a number of assupmtions we made about the data and these must be met for a linear model to work successfully and the standard residual plots can help validate some of these. These are:

The dataset must have some linear relationship Multivariate normality - the dataset variables must be statistically Normally Distributed (i.e. resembling a Bell Curve) It must have no or little multicollinearity - this means the independent variables must not be too highly correlated with each other. This can be tested with a Correlation matrix and other tests No auto-correlation - Auto-correlation occurs when the residuals are not independent from each other. For

instance, this typically occurs in stock prices, where the price is not independent from the previous price. Homoscedasticity - meaning that the residuals are equally distributed across the regression line i.e. above and below the regression line and the variance of the residuals should be the same for all predicted scores along the regression line.

Four standard plots can be accessed using the plot() function with the fit variable once the model is generated. These can be used to show if there are problems with the dataset and the model produced that need to be considered in looking at the validity of the model. These are:

```
Residuals vs Fitted Plot
Normal Q-Q (quantile-quantile) Plot
Scale-Location
Residuals vs Leverage
```

The mtcars dataset is used as an example to show the residual plots. The dataset describes the attibutes of various cars and how these relate to the dependent variable mpg i.e. how to things like weight, no of cylinders and no of gears affect miles per gallon (mpg). For this example we will use mpg (mpg) vs weight (wt).

8.1 Fitting a line, residuals, and correlation

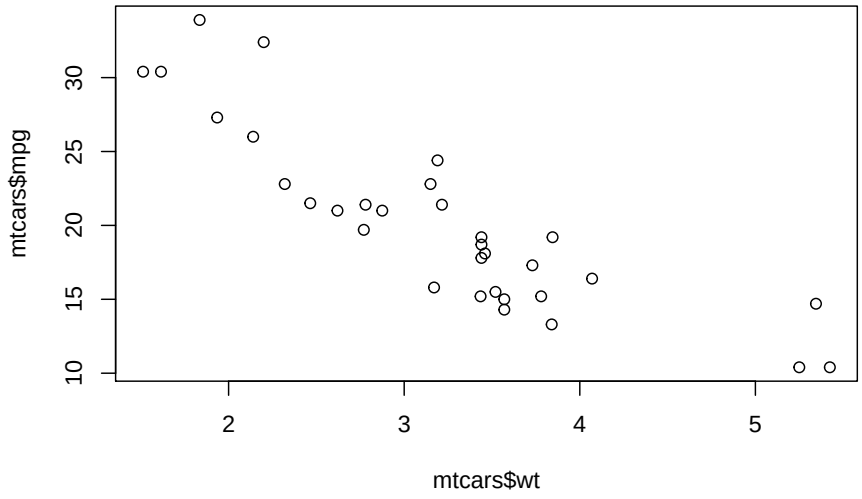
It's helpful to think deeply about the line fitting process. In this section, we define the form of a linear model, explore criteria for what makes a good fit, and introduce a new statistic called *correlation*.

8.1.1 Fitting a line to data

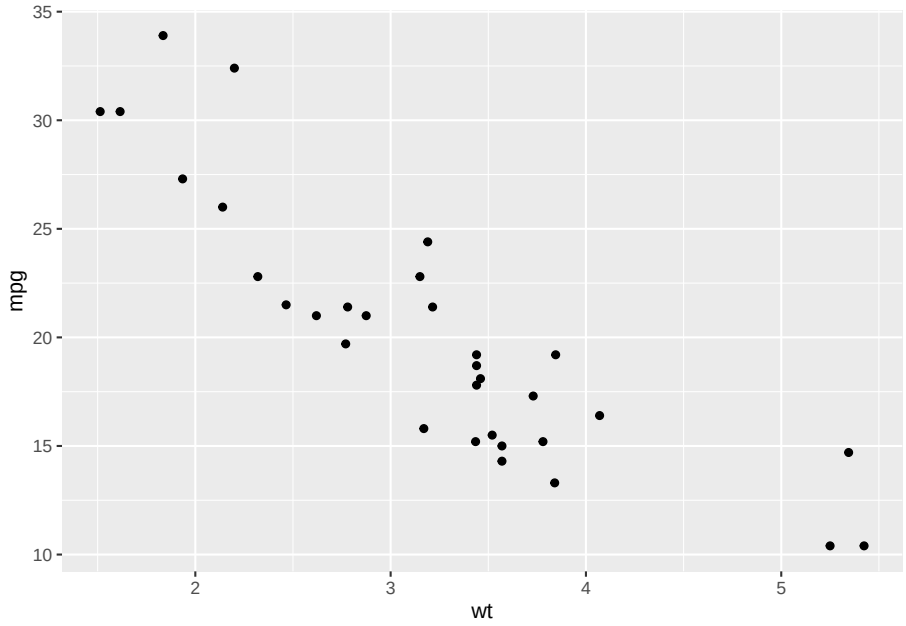
```
data(mtcars)
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
plot(mtcars$wt,mtcars$mpg)
```




```
library(ggplot2)
# Basic scatter plot
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point()
```

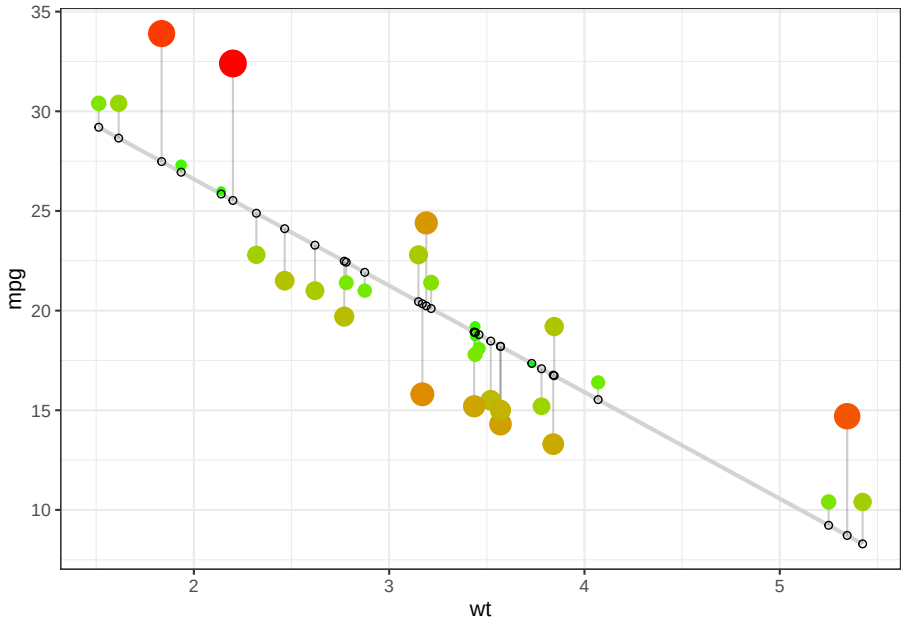


###Fitting the Regression Line and its Residuals

Using the mtcars dataset we can use the lm linear regression function to fit a regression line and then plot it to see the results. The plot shows a good looking regression line.

The plot shows graphically the size of the residual value using a colour code (red is longer line to green - smaller line) and size of point. The size of residual is the length of the vertical line from the point to where it meets the regression line.

```
## `geom_smooth()` using formula 'y ~ x'
```



Based on this graph, what mpg would you predict for a car weighing 4.5 (lbs in 1,000's)?

```
data("women")
head(women)
```

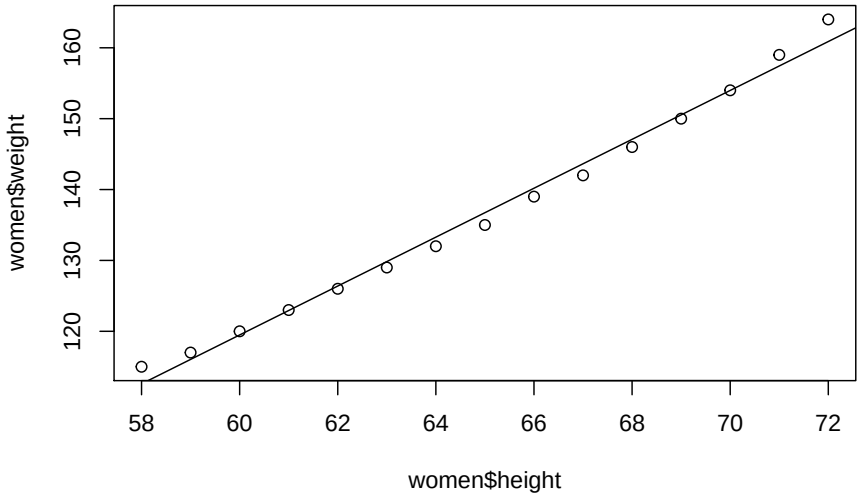
```
##   height weight
## 1     58    115
## 2     59    117
## 3     60    120
## 4     61    123
## 5     62    126
## 6     63    129
```

```
plot(women$height,women$weight)
lm1 <- lm(women$weight~women$height)
summary(lm1)
```

```
##
## Call:
## lm(formula = women$weight ~ women$height)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
```

```
## -1.7333 -1.1333 -0.3833  0.7417  3.1167
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -87.51667    5.93694  -14.74 1.71e-09 ***
## women$height   3.45000    0.09114   37.85 1.09e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.525 on 13 degrees of freedom
## Multiple R-squared:  0.991,  Adjusted R-squared:  0.9903
## F-statistic:  1433 on 1 and 13 DF,  p-value: 1.091e-14
```

```
abline(lm1)
```



shows two variables whose relationship can be modeled nearly perfectly with a straight line. The equation for the line is $y = -87.51667 + 3.45000x$. Consider what a perfect linear relationship means: we know the exact value of y just by knowing the value of x. Perfect fit is unrealistic in almost any natural process. For example, if we took family income (x), this value would provide some useful information about how much financial support a college may offer a prospective student (y). However, the prediction would be far from perfect, since other factors play a role in financial support beyond a family's finances.

Linear regression is the statistical method for fitting a line to data where the

relationship between two variables, x and y, can be modeled by a straight line with some error:

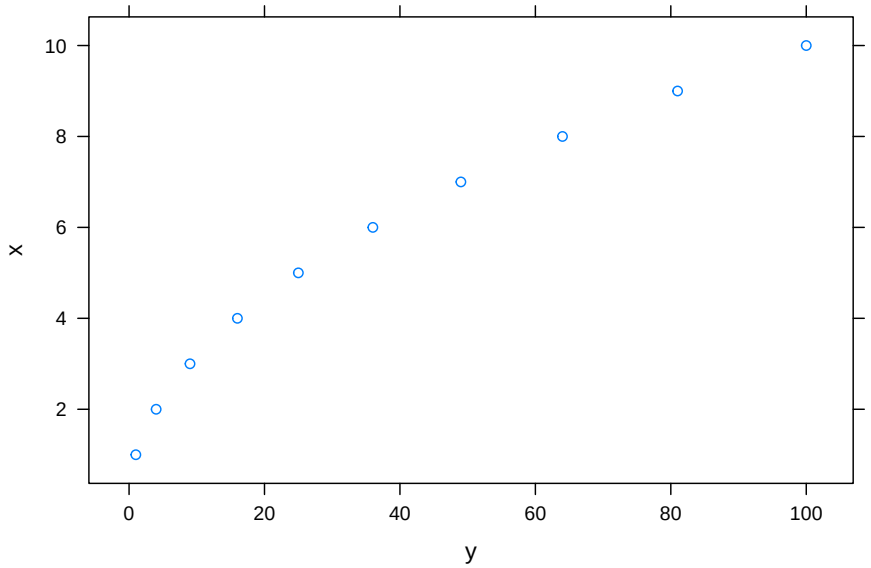
$$y = \beta_0 + \beta_1 x + \varepsilon$$

The values β_0 and β_1 represent the model's parameters (β is the Greek letter *beta*), and the error is represented by ε (the Greek letter *epsilon*). The parameters are estimated using data, and we write their point estimates as b_0 and b_1. When we use x to predict y, we usually call x the explanatory or **predictor** variable, and we call y the response; we also often drop the ϵ term when writing down the model since our main focus is often on the prediction of the average outcome.

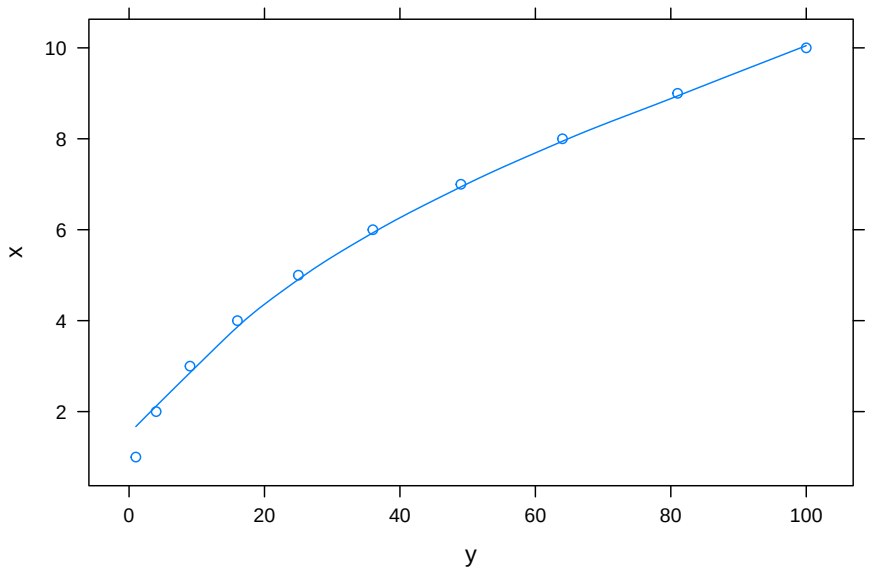
It is rare for all of the data to fall perfectly on a straight line. Instead, it's more common for data to appear as a *cloud of points*, such as those examples shown in Figure **??**. In each case, the data fall around a straight line, even if none of the observations fall exactly on the line. The first plot shows a relatively strong downward linear trend, where the remaining variability in the data around the line is minor relative to the strength of the relationship between x and y. The second plot shows an upward trend that, while evident, is not as strong as the first. The last plot shows a very weak downward trend in the data, so slight we can hardly notice it. In each of these examples, we will have some uncertainty regarding our estimates of the model parameters, β_0 and β_1. For instance, we might wonder, should we move the line up or down a little, or should we tilt it more or less? As we move forward in this chapter, we will learn about criteria for line-fitting, and we will also learn about the uncertainty associated with estimates of model parameters.

There are also cases where fitting a straight line to the data, even if there is a clear relationship between the variables, is not helpful. One such case is shown in Figure **??** where there is a very clear relationship between the variables even though the trend is not linear.

```
library(lattice)
par(mfrow=c(2,1))
vals<-data.frame(x=1:10,y=(1:10)^2)
xyplot(x~y,data=vals)
```




```
vals<-data.frame(x=1:10,y=(1:10)^2)
xyplot(x~y,data=vals,type=c("p","smooth"))
```



We discuss nonlinear trends briefly here but details of fitting nonlinear models

are saved for a later course.

```
Now (?) might be a great time for us to check out this website: http://guessthecorrelation.com/
```

8.2 Excercises & Homework

Your homework due before class tomorrow is to watch these videos which are posted under the linear regression header on Brightspace and then do the following homework: Videos to watch: 1. Linear regression women 2. Best fit line women

Once you have watched these videos, and you can refer to them as often as you would like, please answer and do the following:

1. Use linear regression to predict the weight of a woman who is 100 inches tall.
2. Use linear regression to predict the height of the woman who weighs 200 pounds.
3. Use linear regression to predict the height of a woman who weighs 5 pounds.
4. Use linear regression to predict the weight of a woman who is 200 inches tall.
5. Plot weight on the X axes and height on the y-axes and create a best fit line on your plot.
6. Plot height on the y-axes and wait on the X axes and create a best fit line on your plot.
7. Add a another column to the women dataframe called GPA which is these 15 numbers: 1.5,4,2,3.7,4,1, 3, 2.5, 3.8, 0.8, 2, 4, 1, 3, 2.
8. Use GPA to predict height. Is GPA a significant predictor and how do you know? Draw a best fit line on this relationship.
9. Use GPA to predict a weight. Is GPA a significant predictor and how do you know? Draw a bested line on this relationship, too.
10. Predict the height of a person with a GPA of 4.0.

Chapter 9

Managing Data Frames with the dplyr package

OK. This is the easiset way to me. Based on SQL. Let's chat.

9.1 Data Frames

The data frame is a key data structure in statistics and in R. The basic structure of a data frame is that there is one observation per row and each column represents a variable, a measure, feature, or characteristic of that observation. R has an internal implementation of data frames that is likely the one you will use most often. However, there are packages on CRAN that implement data frames via things like relational databases that allow you to operate on very very large data frames (but we won't discuss them here).

Given the importance of managing data frames, it's important that we have good tools for dealing with them. In previous chapters we have already discussed some tools like the use of [and $ operators to extract subsets of data frames. However, other operations, like filtering, re-ordering, and collapsing, can often be tedious operations in R whose syntax is not very intuitive. The dplyr package is designed to mitigate a lot of these problems and to provide a highly optimized set of routines specifically for dealing with data frames.

9.2 The dplyr Package

The dplyr package was developed by Hadley Wickham of RStudio and is an optimized and distilled version of his plyr package. The dplyr package does

not provide any "new" functionality to R per se, in the sense that everything dplyr does could already be done with base R, but it greatly simplifies existing functionality in R. One important contribution of the dplyr package is that it provides a "grammar" (in particular, verbs) for data manipulation and for operating on data frames. With this grammar, you can sensibly communicate what it is that you are doing to a data frame that other people can understand (assuming they also know the grammar). This is useful because it provides an abstraction for data manipulation that previously did not exist. Another useful contribution is that the dplyr functions are very fast, as many key operations are coded in C++.

9.3 dplyr Grammar

Some of the key "verbs" provided by the dplyr package are - select: return a subset of the columns of a data frame, using a flexible notation

- filter: extract a subset of rows from a data frame based on logical conditions
- arrange: reorder rows of a data frame
- rename: rename variables in a data frame
- mutate: add new variables/columns or transform existing variables
- summarise / summarize: generate summary statistics of different variables in the data frame, possibly within strata
- %>%: the "pipe" operator is used to connect multiple verb actions together into a pipeline

```
Let's hang out here for a while.  PLEASE make sure you know what each of the verbs abo
```

> I repeat. Let's hang out here for a while. PLEASE make sure you know the verbs above. Get this.

The dplyr package as a number of its own data types that it takes advantage of. For example, there is a handy print method that prevents you from printing a lot of data to the console. Most of the time, these additional data types are transparent to the user and do not need to be worried about.

9.4 Common dplyr Function Properties

All of the functions that we will discuss in this tutorial will have a few common characteristics. In particular,

1. The first argument is a data frame.

2. The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the $ operator (just use the column names).
3. The return result of a function is a new data frame
4. Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be tidy[^1]. In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.

9.5 Installing the dplyr package

The dplyr package can be installed from CRAN or from GitHub using the devtools package and the install_github() function. The GitHub repository will usually contain the latest updates to the package and the development version. To install from CRAN, just run

```
library(dplyr)
```

After installing the package it is important that you load it into your R session with the library() function.

```
library(dplyr)
```

```
The following object is masked from 'package:stats':
filter
The following objects are masked from 'package:base':
intersect, setdiff, setequal, union
```

You may get some warnings when the package is loaded because there are functions in the dplyr package that have the same name as functions in other packages. For now you can ignore the warnings.

9.6 select()

For the examples in this chapter we will be using a dataset containing air pollution and temperature data for the city of Chicago [3] in the U.S. The dataset is available from my web site. After unzipping the archive, you can load the data into R using the readRDS() function.

> chicago <- readRDS("chicago.rds") You can see some basic characteristics of the dataset with the dim() and str() functions.

> dim(chicago) [1] 6940 8 str(chicago) 'data.frame': 6940 obs. of 8 variables: $ city : chr "chic" "chic" "chic" "chic" ... $ tmpd : num 31.5 33 33 29 32 40 34.5 29 26.5 32.5 ... $ dptp : num 31.5 29.9 27.4 28.6 28.9 ... $ date : Date, format: "1987-01-01" "1987-01-02" ... $ pm25tmean2: num NA NA NA NA NA NA NA NA NA NA ... $ pm10tmean2: num 34 NA 34.2 47 NA ... $ o3tmean2 : num 4.25 3.3 3.33 4.38 4.75 ... $ no2tmean2 : num 20 23.2 23.8 30.4 30.3 ...

The select() function can be used to select columns of a data frame that you want to focus on. Often you'll have a large data frame containing "all" of the data, but any given analysis might only use a subset of variables or observations. The select() function allows you to get the few columns you might need. Suppose we wanted to take the first 3 columns only. There are a few ways to do this. We could for example use numerical indices. But we can also use the names directly. > names(chicago)[1:3] [1] "city" "tmpd" "dptp" > subset <- select(chicago, city:dptp) > head(subset) city tmpd dptp 1 chic 31.5 31.500 2 chic 33.0 29.875 3 chic 33.0 27.375 4 chic 29.0 28.625 5 chic 32.0 28.875 6 chic 40.0 35.125 Note that the : normally cannot be used with names or strings, but inside the select() function you can use it to specify a range of variable names. You can also omit variables using the select() function by using the negative sign. With select() you can do > select(chicago, -(city:dptp)) which indicates that we should include every variable except the variables city through dptp. The equivalent code in base R would be Managing Data Frames with the dplyr package 53 > i <- match("city", names(chicago)) > j <- match("dptp", names(chicago)) > head(chicago[, -(i:j)]) Not super intuitive, right? The select() function also allows a special syntax that allows you to specify variable names based on patterns. So, for example, if you wanted to keep every variable that ends with a "2", we could do > subset <- select(chicago, ends_with("2")) > str(subset) 'data.frame': 6940 obs. of 4 variables: $ pm25tmean2: num NA NA NA NA NA NA NA NA NA NA ... $ pm10tmean2: num 34 NA 34.2 47 NA ... $ o3tmean2 : num 4.25 3.3 3.33 4.38 4.75 ... $ no2tmean2 : num 20 23.2 23.8 30.4 30.3 ... Or if we wanted to keep every variable that starts with a "d", we could do > subset <- select(chicago, starts_with("d")) > str(subset) 'data.frame': 6940 obs. of 2 variables: $ dptp: num 31.5 29.9 27.4 28.6 28.9 ... $ date: Date, format: "1987-01-01" "1987-01-02" ... You can also use more general regular expressions if necessary. See the help page (?select) for more details.

filter() The filter() function is used to extract subsets of rows from a data frame. This function is similar to the existing subset() function in R but is quite a bit faster in my experience. Suppose we wanted to extract the rows of the chicago data frame where the levels of PM2.5 are greater than 30 (which is a reasonably high level), we could do Managing Data Frames with the dplyr package 54 > chic.f <- filter(chicago, pm25tmean2 > 30) > str(chic.f) 'data.frame': 194 obs. of 8 variables: $ city : chr "chic" "chic" "chic" "chic" ... $ tmpd : num 23 28 55 59 57 57 75 61 73 78 ... $ dptp : num 21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ... $ date : Date, format: "1998-01-17" "1998-01-23" ... $ pm25tmean2: num 38.1 34

39.4 35.4 33.3 … $ pm10tmean2: num 32.5 38.7 34 28.5 35 … $ o3tmean2 : num 3.18 1.75 10.79 14.3 20.66 … $ no2tmean2 : num 25.3 29.4 25.3 31.4 26.8 … You can see that there are now only 194 rows in the data frame and the distribution of the pm25tmean2 values is. > summary(chic.f$pm25tmean2) Min. 1st Qu. Median Mean 3rd Qu. Max. 30.05 32.12 35.04 36.63 39.53 61.50 We can place an arbitrarily complex logical sequence inside of filter(), so we could for example extract the rows where PM2.5 is greater than 30 and temperature is greater than 80 degrees Fahrenheit. > chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80) > select(chic.f, date, tmpd, pm25tmean2) date tmpd pm25tmean2 1 1998-08-23 81 39.60000 2 1998-09-06 81 31.50000 3 2001-07-20 82 32.30000 4 2001-08-01 84 43.70000 5 2001-08-08 85 38.83750 6 2001-08-09 84 38.20000 7 2002-06-20 82 33.00000 8 2002-06-23 82 42.50000 9 2002-07-08 81 33.10000 10 2002-07-18 82 38.85000 11 2003-06-25 82 33.90000 12 2003-07-04 84 32.90000 13 2005-06-24 86 31.85714 14 2005-06-27 82 51.53750 15 2005-06-28 85 31.20000 16 2005-07-17 84 32.70000 17 2005-08-03 84 37.90000 Managing Data Frames with the dplyr package 55 Now there are only 17 observations where both of those conditions are met.

arrange() The arrange() function is used to reorder rows of a data frame according to one of the variables/- columns. Reordering rows of a data frame (while preserving corresponding order of other columns) is normally a pain to do in R. The arrange() function simplifies the process quite a bit. Here we can order the rows of the data frame by date, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation. > chicago <- arrange(chicago, date)

We can now check the first few rows > head(select(chicago, date, pm25tmean2), 3) date pm25tmean2 1 1987-01-01 NA 2 1987-01-02 NA 3 1987-01-03 NA and the last few rows. > tail(select(chicago, date, pm25tmean2), 3) date pm25tmean2 6938 2005-12-29 7.45000 6939 2005-12-30 15.05714 6940 2005-12-31 15.00000 Columns can be arranged in descending order too by useing the special desc() operator. > chicago <- arrange(chicago, desc(date)) Looking at the first three and last three rows shows the dates in descending order. Managing Data Frames with the dplyr package 56 > head(select(chicago, date, pm25tmean2), 3) date pm25tmean2 1 2005-12-31 15.00000 2 2005-12-30 15.05714 3 2005-12-29 7.45000 > tail(select(chicago, date, pm25tmean2), 3) date pm25tmean2 6938 1987-01-03 NA 6939 1987-01-02 NA 6940 1987-01-01 NA rename()

Renaming a variable in a data frame in R is surprisingly hard to do! The rename() function is designed to make this process easier. Here you can see the names of the first five variables in the chicago data frame. > head(chicago[, 1:5], 3) city tmpd dptp date pm25tmean2 1 chic 35 30.1 2005-12-31 15.00000 2 chic 36 31.0 2005-12-30 15.05714 3 chic 35 29.4 2005-12-29 7.45000 The dptp column is supposed to represent the dew point temperature adn the pm25tmean2 column provides the PM2.5 data. However, these names are pretty obscure or awkward and probably be renamed to something more sensible. > chicago <- rename(chicago, dewpoint = dptp, pm25 = pm25tmean2) > head(chicago[, 1:5],

3) city tmpd dewpoint date pm25 1 chic 35 30.1 2005-12-31 15.00000 2 chic 36 31.0 2005-12-30 15.05714 3 chic 35 29.4 2005-12-29 7.45000 The syntax inside the rename() function is to have the new name on the left-hand side of the = sign and the old name on the right-hand side. I leave it as an exercise for the reader to figure how you do this in base R without dplyr.

mutate() The mutate() function exists to compute transformations of variables in a data frame. Often, you want to create new variables that are derived from existing variables and mutate() provides a clean interface for doing that. For example, with air pollution data, we often want to detrend the data by subtracting the mean from the data. That way we can look at whether a given day's air pollution level is higher than or less than average (as opposed to looking at its absolute level). Here we create a pm25detrend variable that subtracts the mean from the pm25 variable. > chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE)) > head(chicago) city tmpd dewpoint date pm25 pm10tmean2 o3tmean2 no2tmean2 1 chic 35 30.1 2005-12-31 15.00000 23.5 2.531250 13.25000 2 chic 36 31.0 2005-12-30 15.05714 19.2 3.034420 22.80556 3 chic 35 29.4 2005-12-29 7.45000 23.5 6.794837 19.97222 4 chic 37 34.5 2005-12-28 17.75000 27.5 3.260417 19.28563 5 chic 40 33.6 2005-12-27 23.56000 27.0 4.468750 23.50000 6 chic 35 29.6 2005-12-26 8.40000 8.5 14.041667 16.81944 pm25detrend 1 -1.230958 2 -1.173815 3 -8.780958 4 1.519042 5 7.329042 6 -7.830958 There is also the related transmute() function, which does the same thing as mutate() but then drops all non-transformed variables. Here we detrend the PM10 and ozone (O3) variables. > head(transmute(chicago, + pm10detrend = pm10tmean2 - mean(pm10tmean2, na.rm = TRUE), + o3detrend = o3tmean2 - mean(o3tmean2, na.rm = TRUE))) pm10detrend o3detrend 1 -10.395206 -16.904263 2 -14.695206 -16.401093 3 -10.395206 -12.640676 4 -6.395206 -16.175096 5 -6.895206 -14.966763 6 -25.395206 -5.393846

Note that there are only two columns in the transmuted data frame. group_by() The group_by() function is used to generate summary statistics from the data frame within strata defined by a variable. For example, in this air pollution dataset, you might want to know what the average annual level of PM2.5 is. So the stratum is the year, and that is something we can derive from the date variable. In conjunction with the group_by() function we often use the summarize() function (or summarise() for some parts of the world). The general operation here is a combination of splitting a data frame into separate pieces defined by a variable or group of variables (group_by()), and then applying a summary function across those subsets (summarize()). First, we can create a year varible using as.POSIXlt(). > chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900) Now we can create a separate data frame that splits the original data frame by year. > $years <- group_by(chicago, year) Finally, we compute summary statistics for each year in the data framew summarize(years, pm25 = mean(pm25, na.rm = TRUE), +o3 = max(o3tmean2, na.rm = TRUE), +no2 = median(no2tmean2, na.rm = TRUE)) Source : local data frame [19 x 4] year pm25 o3 no2 qq <- quantile(chicago$pm25, seq(0, 1, 0.2), na.rm = TRUE > chicago <- mutate(chicago, pm25.quint = cut(pm25, qq))

Now we can group the data frame by the pm25.quint variable. > quint <- group_by(chicago, pm25.quint)

Finally, we can compute the mean of o3 and no2 within quintiles of pm25. > summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE), + no2 = mean(no2tmean2, na.rm = TRUE)) Source: local data frame [6 x 3] pm25.quint o3 no2 1 (1.7,8.7] 21.66401 17.99129 2 (8.7,12.4] 20.38248 22.13004 3 (12.4,16.7] 20.66160 24.35708 4 (16.7,22.6] 19.88122 27.27132 5 (22.6,61.5] 20.31775 29.64427 6 NA 18.79044 25.77585

From the table, it seems there isn't a strong relationship between pm25 and o3, but there appears to be a positive correlation between pm25 and no2. More sophisticated statistical modeling can help to provide precise answers to these questions, but a simple application of dplyr functions can often get you most of the way there.

%>% The pipeline operator %>% is very handy for stringing together multiple dplyr functions in a sequence of operations. Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e. > third(second(first(x))) This nesting is not a natural way to think about a sequence of operations. The %>% operator allows you to string operations in a left-to-right fashion, i.e. > first(x) %>% second %>% third Take the example that we just did in the last section where we computed the mean of o3 and no2 within quintiles of pm25. There we had to 1. create a new variable pm25.quint 2. split the data frame by that new variable 3. compute the mean of o3 and no2 in the sub-groups defined by pm25.quint That can be done with the following sequence in a single R expression. > mutate(chicago, pm25.quint = cut(pm25, qq)) %>% + group_by(pm25.quint) %>% + summarize(o3 = mean(o3tmean2, na.rm = TRUE), + no2 = mean(no2tmean2, na.rm = TRUE)) Source: local data frame [6 x 3] pm25.quint o3 no2 1 (1.7,8.7] 21.66401 17.99129 2 (8.7,12.4] 20.38248 22.13004 3 (12.4,16.7] 20.66160 24.35708 4 (16.7,22.6] 19.88122 27.27132 5 (22.6,61.5] 20.31775 29.64427 6 NA 18.79044 25.77585 This way we don't have to create a set of temporary variables along the way or create a massive nested sequence of function calls. Notice in the above code that I pass the chicago data frame to the first call to mutate(), but then afterwards I do not have to pass the first argument to group_by() or summarize(). Once you travel down the pipeline with %>%, the first argument is taken to be the output of the previous element in the pipeline. Another example might be computing the average pollutant level by month. This could be useful to see if there are any seasonal trends in the data. Managing Data Frames with the dplyr package 61 > mutate(chicago, month = as.POSIXlt(date)$mon + 1) %>% + group_by(month) %>% + summarize(pm25 = mean(pm25, na.rm = TRUE), + o3 = max(o3tmean2, na.rm = TRUE), + no2 = median(no2tmean2, na.rm = TRUE)) Source: local data frame [12 x 4] month pm25 o3 no2 1 1 17.76996 28.22222 25.35417 2 2 20.37513 37.37500 26.78034 3 3 17.40818 39.05000 26.76984 4 4 13.85879 47.94907 25.03125 5 5 14.07420 52.75000 24.22222 6

6 15.86461 66.58750 25.01140 7 7 16.57087 59.54167 22.38442 8 8 16.93380 53.96701 22.98333 9 9 15.91279 57.48864 24.47917 10 10 14.23557 47.09275 24.15217 11 11 15.15794 29.45833 23.56537 12 12 17.52221 27.70833 24.45773 Here we can see that o3 tends to be low in the winter months and high in the summer while no2 is higher in the winter and lower in the summer. Summary The dplyr package provides a concise set of operations for managing data frames. With these functions we can do a number of complex operations in just a few lines of code. In particular, we can often conduct the beginnings of an exploratory analysis with the powerful combination of group_by() and summarize(). Once you learn the dplyr grammar there are a few additional benefits • dplyr can work with other data frame "backends" such as SQL databases. There is an SQL interface for relational databases via the DBI package • dplyr can be integrated with the data.table package for large fast tables

The dplyr package is handy way to both simplify and speed up your data frame management code. It's rare that you get such a combination at the same time!

Chapter 10

Introduction to dplyr for Faster Data Manipulation in R"

Source: https://github.com/justmarkham/dplyr-tutorial

Note: There is a 40-minute video tutorial on YouTube that walks through this document in detail.

10.1 Why do I use dplyr?

- Great for data exploration and transformation
- Intuitive to write and easy to read, especially when using the "chaining" syntax (covered below)
- Fast on data frames

10.2 dplyr functionality

- Five basic verbs: `filter`, `select`, `arrange`, `mutate`, `summarise` (plus `group_by`)
- Can work with data stored in databases and data tables
- Joins: inner join, left join, semi-join, anti-join (not covered below)
- Window functions for calculating ranking, offsets, and more
- Better than plyr if you're only working with data frames (though it doesn't yet duplicate all of the plyr functionality)

- Examples below are based upon the latest release, version 0.2 (released May 2014)

10.3 Loading dplyr and an example dataset

- dplyr will mask a few base functions
- If you also use plyr, load plyr first
- hflights is flights departing from two Houston airports in 2011

```
# load packages
suppressMessages(library(dplyr))
library(hflights)

# explore data
data(hflights)
head(hflights)
```

```
##      Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
## 5424 2011     1          1         6    1400    1500            AA       428
## 5425 2011     1          2         7    1401    1501            AA       428
## 5426 2011     1          3         1    1352    1502            AA       428
## 5427 2011     1          4         2    1403    1513            AA       428
## 5428 2011     1          5         3    1405    1507            AA       428
## 5429 2011     1          6         4    1359    1503            AA       428
##      TailNum ActualElapsedTime AirTime ArrDelay DepDelay Origin Dest Distance
## 5424  N576AA                60      40      -10        0    IAH  DFW      224
## 5425  N557AA                60      45       -9        1    IAH  DFW      224
## 5426  N541AA                70      48       -8       -8    IAH  DFW      224
## 5427  N403AA                70      39        3        3    IAH  DFW      224
## 5428  N492AA                62      44       -3        5    IAH  DFW      224
## 5429  N262AA                64      45       -7       -1    IAH  DFW      224
##      TaxiIn TaxiOut Cancelled CancellationCode Diverted
## 5424      7      13         0                         0
## 5425      6       9         0                         0
## 5426      5      17         0                         0
## 5427      9      22         0                         0
## 5428      9       9         0                         0
## 5429      6      13         0                         0
```

- `tbl_df` creates a "local data frame"
- Local data frame is simply a wrapper for a data frame that prints nicely

```
# convert to local data frame
flights <- tbl_df(hflights)
```

```
## Warning: `tbl_df()` is deprecated as of dplyr 1.0.0.
## Please use `tibble::as_tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
```

```
# printing only shows 10 rows and as many columns as can fit on your screen
flights
```

```
## # A tibble: 227,496 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
##    <int> <int>      <int>     <int>   <int>   <int> <chr>             <int>
##  1  2011     1          1         6    1400    1500 AA                  428
##  2  2011     1          2         7    1401    1501 AA                  428
##  3  2011     1          3         1    1352    1502 AA                  428
##  4  2011     1          4         2    1403    1513 AA                  428
##  5  2011     1          5         3    1405    1507 AA                  428
##  6  2011     1          6         4    1359    1503 AA                  428
##  7  2011     1          7         5    1359    1509 AA                  428
##  8  2011     1          8         6    1355    1454 AA                  428
##  9  2011     1          9         7    1443    1554 AA                  428
## 10  2011     1         10         1    1443    1553 AA                  428
## # ... with 227,486 more rows, and 13 more variables: TailNum <chr>,
## #   ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>, DepDelay <int>,
## #   Origin <chr>, Dest <chr>, Distance <int>, TaxiIn <int>, TaxiOut <int>,
## #   Cancelled <int>, CancellationCode <chr>, Diverted <int>
```

```
# you can specify that you want to see more rows
print(flights, n=20)

# convert to a normal data frame to see all of the columns
data.frame(head(flights))
```

10.4 filter: Keep rows matching criteria

- Base R approach to filtering forces you to repeat the data frame's name
- dplyr approach is simpler to write and read
- Command structure (for all dplyr verbs):
  - first argument is a data frame
  - return value is a data frame

  - nothing is modified in place

- Note: dplyr generally does not preserve row names

```
# base R approach to view all flights on January 1
flights[flights$Month==1 & flights$DayofMonth==1, ]
```

```
# dplyr approach
# note: you can use comma or ampersand to represent AND condition
filter(flights, Month==1, DayofMonth==1)
```

```
## # A tibble: 552 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
##    <int> <int>      <int>     <int>   <int>   <int> <chr>             <int>
##  1  2011     1          1         6    1400    1500 AA                  428
##  2  2011     1          1         6     728     840 AA                  460
##  3  2011     1          1         6    1631    1736 AA                 1121
##  4  2011     1          1         6    1756    2112 AA                 1294
##  5  2011     1          1         6    1012    1347 AA                 1700
##  6  2011     1          1         6    1211    1325 AA                 1820
##  7  2011     1          1         6     557     906 AA                 1994
##  8  2011     1          1         6    1824    2106 AS                  731
##  9  2011     1          1         6     654    1124 B6                  620
## 10  2011     1          1         6    1639    2110 B6                  622
## # ... with 542 more rows, and 13 more variables: TailNum <chr>,
## #   ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>, DepDelay <int>,
## #   Origin <chr>, Dest <chr>, Distance <int>, TaxiIn <int>, TaxiOut <int>,
## #   Cancelled <int>, CancellationCode <chr>, Diverted <int>
```

```
# use pipe for OR condition
filter(flights, UniqueCarrier=="AA" | UniqueCarrier=="UA")
```

```
## # A tibble: 5,316 x 21
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
##    <int> <int>      <int>     <int>   <int>   <int> <chr>             <int>
##  1  2011     1          1         6    1400    1500 AA                  428
##  2  2011     1          2         7    1401    1501 AA                  428
##  3  2011     1          3         1    1352    1502 AA                  428
##  4  2011     1          4         2    1403    1513 AA                  428
##  5  2011     1          5         3    1405    1507 AA                  428
##  6  2011     1          6         4    1359    1503 AA                  428
##  7  2011     1          7         5    1359    1509 AA                  428
##  8  2011     1          8         6    1355    1454 AA                  428
##  9  2011     1          9         7    1443    1554 AA                  428
```

```
## 10  2011     1         10         1    1443    1553 AA                  428
## # ... with 5,306 more rows, and 13 more variables: TailNum <chr>,
## #   ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>, DepDelay <int>,
## #   Origin <chr>, Dest <chr>, Distance <int>, TaxiIn <int>, TaxiOut <int>,
## #   Cancelled <int>, CancellationCode <chr>, Diverted <int>
```

```
# you can also use %in% operator
filter(flights, UniqueCarrier %in% c("AA", "UA"))
```

10.5 select: Pick columns by name

- Base R approach is awkward to type and to read
- dplyr approach uses similar syntax to `filter`
- Like a SELECT in SQL

```
# base R approach to select DepTime, ArrTime, and FlightNum columns
flights[, c("DepTime", "ArrTime", "FlightNum")]
```

```
# dplyr approach
select(flights, DepTime, ArrTime, FlightNum)
```

```
## # A tibble: 227,496 x 3
##    DepTime ArrTime FlightNum
##      <int>   <int>     <int>
##  1    1400    1500       428
##  2    1401    1501       428
##  3    1352    1502       428
##  4    1403    1513       428
##  5    1405    1507       428
##  6    1359    1503       428
##  7    1359    1509       428
##  8    1355    1454       428
##  9    1443    1554       428
## 10    1443    1553       428
## # ... with 227,486 more rows
```

```
# use colon to select multiple contiguous columns, and use `contains` to match columns by name
# note: `starts_with`, `ends_with`, and `matches` (for regular expressions) can also be used to m
select(flights, Year:DayofMonth, contains("Taxi"), contains("Delay"))
```

```
## # A tibble: 227,496 x 7
##     Year Month DayofMonth TaxiIn TaxiOut ArrDelay DepDelay
```

```
##    <int> <int>      <int>  <int>   <int>    <int>    <int>
##  1  2011     1          1      7      13      -10        0
##  2  2011     1          2      6       9       -9        1
##  3  2011     1          3      5      17       -8       -8
##  4  2011     1          4      9      22        3        3
##  5  2011     1          5      9       9       -3        5
##  6  2011     1          6      6      13       -7       -1
##  7  2011     1          7     12      15       -1       -1
##  8  2011     1          8      7      12      -16       -5
##  9  2011     1          9      8      22       44       43
## 10  2011     1         10      6      19       43       43
## # ... with 227,486 more rows
```

10.6 "Chaining" or "Pipelining"

- Usual way to perform multiple operations in one line is by nesting
- Can write commands in a natural order by using the %>% infix operator (which can be pronounced as "then")

```
# nesting method to select UniqueCarrier and DepDelay columns and filter for delays ov
filter(select(flights, UniqueCarrier, DepDelay), DepDelay > 60)
```

```
# chaining method
flights %>%
    select(UniqueCarrier, DepDelay) %>%
    filter(DepDelay > 60)
```

```
## # A tibble: 10,242 x 2
##    UniqueCarrier DepDelay
##    <chr>            <int>
##  1 AA                  90
##  2 AA                  67
##  3 AA                  74
##  4 AA                 125
##  5 AA                  82
##  6 AA                  99
##  7 AA                  70
##  8 AA                  61
##  9 AA                  74
## 10 AS                  73
## # ... with 10,232 more rows
```

- Chaining increases readability significantly when there are many commands

- Operator is automatically imported from the magrittr package
- Can be used to replace nesting in R commands outside of dplyr

```
# create two vectors and calculate Euclidian distance between them
x1 <- 1:5; x2 <- 2:6
sqrt(sum((x1-x2)^2))
```

```
# chaining method
(x1-x2)^2 %>% sum() %>% sqrt()
```

```
## [1] 2.236068
```

10.7 arrange: Reorder rows

```
# base R approach to select UniqueCarrier and DepDelay columns and sort by DepDelay
flights[order(flights$DepDelay), c("UniqueCarrier", "DepDelay")]
```

```
# dplyr approach
flights %>%
    select(UniqueCarrier, DepDelay) %>%
    arrange(DepDelay)
```

```
## # A tibble: 227,496 x 2
##    UniqueCarrier DepDelay
##    <chr>            <int>
##  1 OO                 -33
##  2 MQ                 -23
##  3 XE                 -19
##  4 XE                 -19
##  5 CO                 -18
##  6 EV                 -18
##  7 XE                 -17
##  8 CO                 -17
##  9 XE                 -17
## 10 MQ                 -17
## # ... with 227,486 more rows
```

```
# use `desc` for descending
flights %>%
    select(UniqueCarrier, DepDelay) %>%
    arrange(desc(DepDelay))
```

10.8 mutate: Add new variables

- Create new variables that are functions of existing variables

```
# base R approach to create a new variable Speed (in mph)
flights$Speed <- flights$Distance / flights$AirTime*60
flights[, c("Distance", "AirTime", "Speed")]
```

```
# dplyr approach (prints the new variable but does not store it)
flights %>%
    select(Distance, AirTime) %>%
    mutate(Speed = Distance/AirTime*60)
```

```
## # A tibble: 227,496 x 3
##    Distance AirTime Speed
##       <int>   <int> <dbl>
##  1      224      40  336
##  2      224      45  299.
##  3      224      48  280
##  4      224      39  345.
##  5      224      44  305.
##  6      224      45  299.
##  7      224      43  313.
##  8      224      40  336
##  9      224      41  328.
## 10      224      45  299.
## # ... with 227,486 more rows
```

```
# store the new variable
flights <- flights %>% mutate(Speed = Distance/AirTime*60)
```

10.9 summarise: Reduce variables to values

- Primarily useful with data that has been grouped by one or more variables
- `group_by` creates the groups that will be operated on
- `summarise` uses the provided aggregation function to summarise each group

```
# base R approaches to calculate the average arrival delay to each destination
head(with(flights, tapply(ArrDelay, Dest, mean, na.rm=TRUE)))
head(aggregate(ArrDelay ~ Dest, flights, mean))
```

```
# dplyr approach: create a table grouped by Dest, and then summarise each group by taking the mea
flights %>%
    group_by(Dest) %>%
    summarise(avg_delay = mean(ArrDelay, na.rm=TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 116 x 2
##    Dest  avg_delay
##    <chr>     <dbl>
##  1 ABQ        7.23
##  2 AEX        5.84
##  3 AGS        4
##  4 AMA        6.84
##  5 ANC       26.1
##  6 ASE        6.79
##  7 ATL        8.23
##  8 AUS        7.45
##  9 AVL        9.97
## 10 BFL      -13.2
## # ... with 106 more rows
```

- `summarise_each` allows you to apply the same summary function to multiple columns at once
- Note: `mutate_each` is also available

```
# for each carrier, calculate the percentage of flights cancelled or diverted
flights %>%
    group_by(UniqueCarrier) %>%
    summarise_each(funs(mean), Cancelled, Diverted)
```

```
## Warning: `summarise_each_()` is deprecated as of dplyr 0.7.0.
## Please use `across()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
```

```
## Warning: `funs()` is deprecated as of dplyr 0.8.0.
## Please use a list of either functions or lambdas:
##
##   # Simple named list:
##   list(mean = mean, median = median)
##
##   # Auto named with `tibble::lst()`:
```

```
##   tibble::lst(mean, median)
##
##   # Using lambdas
##   list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
```

```
## # A tibble: 15 x 3
##    UniqueCarrier Cancelled Diverted
##    <chr>             <dbl>    <dbl>
##  1 AA              0.0185   0.00185
##  2 AS              0        0.00274
##  3 B6              0.0259   0.00576
##  4 CO              0.00678  0.00263
##  5 DL              0.0159   0.00303
##  6 EV              0.0345   0.00318
##  7 F9              0.00716  0
##  8 FL              0.00982  0.00327
##  9 MQ              0.0290   0.00194
## 10 OO              0.0139   0.00349
## 11 UA              0.0164   0.00241
## 12 US              0.0113   0.00147
## 13 WN              0.0155   0.00229
## 14 XE              0.0155   0.00345
## 15 YV              0.0127   0
```

```
# for each carrier, calculate the minimum and maximum arrival and departure delays
flights %>%
    group_by(UniqueCarrier) %>%
    summarise_each(funs(min(., na.rm=TRUE), max(., na.rm=TRUE)), matches("Delay"))
```

```
## # A tibble: 15 x 5
##    UniqueCarrier ArrDelay_min DepDelay_min ArrDelay_max DepDelay_max
##    <chr>               <int>        <int>        <int>        <int>
##  1 AA                    -39          -15          978          970
##  2 AS                    -43          -15          183          172
##  3 B6                    -44          -14          335          310
##  4 CO                    -55          -18          957          981
##  5 DL                    -32          -17          701          730
##  6 EV                    -40          -18          469          479
##  7 F9                    -24          -15          277          275
##  8 FL                    -30          -14          500          507
##  9 MQ                    -38          -23          918          931
## 10 OO                    -57          -33          380          360
## 11 UA                    -47          -11          861          869
```

```
## 12 US                    -42         -17         433         425
## 13 WN                    -44         -10         499         548
## 14 XE                    -70         -19         634         628
## 15 YV                    -32         -11          72          54
```

- Helper function `n()` counts the number of rows in a group
- Helper function `n_distinct(vector)` counts the number of unique items in that vector

```
# for each day of the year, count the total number of flights and sort in descending order
flights %>%
    group_by(Month, DayofMonth) %>%
    summarise(flight_count = n()) %>%
    arrange(desc(flight_count))
```

```
## `summarise()` regrouping output by 'Month' (override with `.groups` argument)
```

```
## # A tibble: 365 x 3
## # Groups:   Month [12]
##    Month DayofMonth flight_count
##    <int>      <int>        <int>
##  1     8          4          706
##  2     8         11          706
##  3     8         12          706
##  4     8          5          705
##  5     8          3          704
##  6     8         10          704
##  7     1          3          702
##  8     7          7          702
##  9     7         14          702
## 10     7         28          701
## # ... with 355 more rows
```

```
# rewrite more simply with the `tally` function
flights %>%
    group_by(Month, DayofMonth) %>%
    tally(sort = TRUE)
```

```
## # A tibble: 365 x 3
## # Groups:   Month [12]
##    Month DayofMonth     n
##    <int>      <int> <int>
##  1     8          4   706
##  2     8         11   706
```

```
##  3     8         12   706
##  4     8          5   705
##  5     8          3   704
##  6     8         10   704
##  7     1          3   702
##  8     7          7   702
##  9     7         14   702
## 10     7         28   701
## # ... with 355 more rows
```

```
# for each destination, count the total number of flights and the number of distinct p
flights %>%
    group_by(Dest) %>%
    summarise(flight_count = n(), plane_count = n_distinct(TailNum))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 116 x 3
##    Dest  flight_count plane_count
##    <chr>        <int>       <int>
##  1 ABQ           2812         716
##  2 AEX            724         215
##  3 AGS              1           1
##  4 AMA           1297         158
##  5 ANC            125          38
##  6 ASE            125          60
##  7 ATL           7886         983
##  8 AUS           5022        1015
##  9 AVL            350         142
## 10 BFL            504          70
## # ... with 106 more rows
```

- Grouping can sometimes be useful without summarising

```
# for each destination, show the number of cancelled and not cancelled flights
flights %>%
    group_by(Dest) %>%
    select(Cancelled) %>%
    table() %>%
    head()
```

```
## Adding missing grouping variables: `Dest`
```

```
##      Cancelled
## Dest     0    1
##   ABQ 2787   25
##   AEX  712   12
##   AGS    1    0
##   AMA 1265   32
##   ANC  125    0
##   ASE  120    5
```

10.10 Window Functions

- Aggregation function (like `mean`) takes n inputs and returns 1 value
- Window function takes n inputs and returns n values
  - Includes ranking and ordering functions (like `min_rank`), offset functions (`lead` and `lag`), and cumulative aggregates (like `cummean`).

```
# for each carrier, calculate which two days of the year they had their longest departure delays
# note: smallest (not largest) value is ranked as 1, so you have to use `desc` to rank by largest
flights %>%
    group_by(UniqueCarrier) %>%
    select(Month, DayofMonth, DepDelay) %>%
    filter(min_rank(desc(DepDelay)) <= 2) %>%
    arrange(UniqueCarrier, desc(DepDelay))
```

```
## Adding missing grouping variables: `UniqueCarrier`
```

```
# rewrite more simply with the `top_n` function
flights %>%
    group_by(UniqueCarrier) %>%
    select(Month, DayofMonth, DepDelay) %>%
    top_n(2) %>%
    arrange(UniqueCarrier, desc(DepDelay))
```

```
## Adding missing grouping variables: `UniqueCarrier`

## Selecting by DepDelay

## # A tibble: 30 x 4
## # Groups:   UniqueCarrier [15]
##    UniqueCarrier Month DayofMonth DepDelay
##    <chr>         <int>      <int>    <int>
##  1 AA               12         12      970
```

```
##  2 AA               11         19      677
##  3 AS                2         28      172
##  4 AS                7          6      138
##  5 B6               10         29      310
##  6 B6                8         19      283
##  7 CO                8          1      981
##  8 CO                1         20      780
##  9 DL               10         25      730
## 10 DL                4          5      497
## # ... with 20 more rows
```

```
# for each month, calculate the number of flights and the change from the previous mon
flights %>%
    group_by(Month) %>%
    summarise(flight_count = n()) %>%
    mutate(change = flight_count - lag(flight_count))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 12 x 3
##    Month flight_count change
##    <int>        <int>  <int>
##  1     1        18910     NA
##  2     2        17128  -1782
##  3     3        19470   2342
##  4     4        18593   -877
##  5     5        19172    579
##  6     6        19600    428
##  7     7        20548    948
##  8     8        20176   -372
##  9     9        18065  -2111
## 10    10        18696    631
## 11    11        18021   -675
## 12    12        19117   1096
```

```
# rewrite more simply with the `tally` function
flights %>%
    group_by(Month) %>%
    tally() %>%
    mutate(change = n - lag(n))
```

```
## # A tibble: 12 x 3
##    Month     n change
##    <int> <int>  <int>
```

```
##  1     1 18910     NA
##  2     2 17128  -1782
##  3     3 19470   2342
##  4     4 18593   -877
##  5     5 19172    579
##  6     6 19600    428
##  7     7 20548    948
##  8     8 20176   -372
##  9     9 18065  -2111
## 10    10 18696    631
## 11    11 18021   -675
## 12    12 19117   1096
```

10.11 Other Useful Convenience Functions

```
# randomly sample a fixed number of rows, without replacement
flights %>% sample_n(5)
```

```
## # A tibble: 5 x 22
##    Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
##   <int> <int>      <int>     <int>   <int>   <int> <chr>             <int>
## 1  2011     2         19         6     718     821 CO                  128
## 2  2011     1         11         2     846    1114 UA                  104
## 3  2011     8          5         5     729     822 WN                    6
## 4  2011    10          4         2     702    1007 DL                  810
## 5  2011     6         20         1    1328    1648 CO                 1548
## # ... with 14 more variables: TailNum <chr>, ActualElapsedTime <int>,
## #   AirTime <int>, ArrDelay <int>, DepDelay <int>, Origin <chr>, Dest <chr>,
## #   Distance <int>, TaxiIn <int>, TaxiOut <int>, Cancelled <int>,
## #   CancellationCode <chr>, Diverted <int>, Speed <dbl>
```

```
# randomly sample a fraction of rows, with replacement
flights %>% sample_frac(0.25, replace=TRUE)
```

```
## # A tibble: 56,874 x 22
##     Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
##    <int> <int>      <int>     <int>   <int>   <int> <chr>             <int>
##  1  2011     6         13         1    1905    2012 MQ                2870
##  2  2011    12         29         4    1726    1817 WN                  46
##  3  2011     4          3         7    1617    1729 XE                2936
##  4  2011    11          3         4    1550    1851 DL                   8
##  5  2011     9         29         4    1841    2021 CO                1741
```

```
##  6  2011     3         19         6     930    1151 OO                5825
##  7  2011    10          3         1    1858    2321 CO                1476
##  8  2011     9         13         2    1719    1817 WN                 492
##  9  2011     2         14         1    1437    1601 CO                 697
## 10  2011     4         21         4    1635    1745 XE                2128
## # ... with 56,864 more rows, and 14 more variables: TailNum <chr>,
## #   ActualElapsedTime <int>, AirTime <int>, ArrDelay <int>, DepDelay <int>,
## #   Origin <chr>, Dest <chr>, Distance <int>, TaxiIn <int>, TaxiOut <int>,
## #   Cancelled <int>, CancellationCode <chr>, Diverted <int>, Speed <dbl>
```

```
# base R approach to view the structure of an object
str(flights)
```

```
## tibble [227,496 x 22] (S3: tbl_df/tbl/data.frame)
##  $ Year             : int [1:227496] 2011 2011 2011 2011 2011 2011 2011 2011 2011 20
##  $ Month            : int [1:227496] 1 1 1 1 1 1 1 1 1 ...
##  $ DayofMonth       : int [1:227496] 1 2 3 4 5 6 7 8 9 10 ...
##  $ DayOfWeek        : int [1:227496] 6 7 1 2 3 4 5 6 7 1 ...
##  $ DepTime          : int [1:227496] 1400 1401 1352 1403 1405 1359 1359 1355 1443 14
##  $ ArrTime          : int [1:227496] 1500 1501 1502 1513 1507 1503 1509 1454 1554 15
##  $ UniqueCarrier    : chr [1:227496] "AA" "AA" "AA" "AA" ...
##  $ FlightNum        : int [1:227496] 428 428 428 428 428 428 428 428 428 428 ...
##  $ TailNum          : chr [1:227496] "N576AA" "N557AA" "N541AA" "N403AA" ...
##  $ ActualElapsedTime: int [1:227496] 60 60 70 70 62 64 70 59 71 70 ...
##  $ AirTime          : int [1:227496] 40 45 48 39 44 45 43 40 41 45 ...
##  $ ArrDelay         : int [1:227496] -10 -9 -8 3 -3 -7 -1 -16 44 43 ...
##  $ DepDelay         : int [1:227496] 0 1 -8 3 5 -1 -1 -5 43 43 ...
##  $ Origin           : chr [1:227496] "IAH" "IAH" "IAH" "IAH" ...
##  $ Dest             : chr [1:227496] "DFW" "DFW" "DFW" "DFW" ...
##  $ Distance         : int [1:227496] 224 224 224 224 224 224 224 224 224 224 ...
##  $ TaxiIn           : int [1:227496] 7 6 5 9 9 6 12 7 8 6 ...
##  $ TaxiOut          : int [1:227496] 13 9 17 22 9 13 15 12 22 19 ...
##  $ Cancelled        : int [1:227496] 0 0 0 0 0 0 0 0 0 0 ...
##  $ CancellationCode : chr [1:227496] "" "" "" "" ...
##  $ Diverted         : int [1:227496] 0 0 0 0 0 0 0 0 0 0 ...
##  $ Speed            : num [1:227496] 336 299 280 345 305 ...
```

```
# dplyr approach: better formatting, and adapts to your screen width
glimpse(flights)
```

```
## Rows: 227,496
## Columns: 22
## $ Year              <int> 2011, 2011, 2011, 2011, 2011, 2011, 2011, 2011, 2...
## $ Month             <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
```

```
## $ DayofMonth        <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15...
## $ DayOfWeek         <int> 6, 7, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7, 1...
## $ DepTime           <int> 1400, 1401, 1352, 1403, 1405, 1359, 1359, 1355, 1...
## $ ArrTime           <int> 1500, 1501, 1502, 1513, 1507, 1503, 1509, 1454, 1...
## $ UniqueCarrier     <chr> "AA", "AA", "AA", "AA", "AA", "AA", "AA", "AA", "...
## $ FlightNum         <int> 428, 428, 428, 428, 428, 428, 428, 428, 428, 428,...
## $ TailNum           <chr> "N576AA", "N557AA", "N541AA", "N403AA", "N492AA",...
## $ ActualElapsedTime <int> 60, 60, 70, 70, 62, 64, 70, 59, 71, 70, 70, 56, 6...
## $ AirTime           <int> 40, 45, 48, 39, 44, 45, 43, 40, 41, 45, 42, 41, 4...
## $ ArrDelay          <int> -10, -9, -8, 3, -3, -7, -1, -16, 44, 43, 29, 5, -...
## $ DepDelay          <int> 0, 1, -8, 3, 5, -1, -1, -5, 43, 43, 29, 19, -2, -...
## $ Origin            <chr> "IAH", "IAH", "IAH", "IAH", "IAH", "IAH", "IAH", ...
## $ Dest              <chr> "DFW", "DFW", "DFW", "DFW", "DFW", "DFW", "DFW", ...
## $ Distance          <int> 224, 224, 224, 224, 224, 224, 224, 224, 224, 224,...
## $ TaxiIn            <int> 7, 6, 5, 9, 9, 6, 12, 7, 8, 6, 8, 4, 6, 5, 6, 12,...
## $ TaxiOut           <int> 13, 9, 17, 22, 9, 13, 15, 12, 22, 19, 20, 11, 13,...
## $ Cancelled         <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ CancellationCode  <chr> "", "", "", "", "", "", "", "", "", "", "", "", "...
## $ Diverted          <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ Speed             <dbl> 336.0000, 298.6667, 280.0000, 344.6154, 305.4545,...
```

10.12 Resources

- Official dplyr reference manual and vignettes on CRAN: vignettes are well-written and cover many aspects of dplyr
- July 2014 webinar about dplyr (and ggvis) by Hadley Wickham and related slides/code: mostly conceptual, with a bit of code
- dplyr tutorial by Hadley Wickham at the useR! 2014 conference: excellent, in-depth tutorial with lots of example code (Dropbox link includes slides, code files, and data files)
- dplyr GitHub repo and list of releases

Chapter 11

Logistic regression: a yes/no model

https://stats.idre.ucla.edu/r/dae/logit-regression/

Logit Regression | R Data Analysis Examples

Logistic regression, also called a logit model, is used to model dichotomous outcome variables. In the logit model the log odds of the outcome is modeled as a linear combination of the predictor variables.

This page uses the following packages. Make sure that you can load them before trying to run the examples on this page. If you do not have a package installed, run: install.packages("packagename"), or if you see the version is out of date, run: update.packages().

```
library(aod)
library(ggplot2)
```

Please note: The purpose of this page is to show how to use various data analysis commands. It does not cover all aspects of the research process which researchers are expected to do. In particular, it does not cover data cleaning and checking, verification of assumptions, model diagnostics and potential follow-up analyses. Examples

11.0.1 Example 1. Suppose that we are interested in the factors that influence whether a political candidate wins an election. The outcome (response) variable is binary (0/1); win or lose. The predictor variables of interest are the amount of money spent on the campaign, the amount of time spent campaigning negatively and whether or not the candidate is an incumbent.

11.0.2 Example 2. A researcher is interested in how variables, such as GRE (Graduate Record Exam scores), GPA (grade point average) and prestige of the undergraduate institution, effect admission into graduate school. The response variable, admit/don't admit, is a binary variable.

11.0.3 Description of the data

For our data analysis below, we are going to expand on Example 2 about getting into graduate school. We have generated hypothetical data, which can be obtained from our website from within R. Note that R requires forward slashes (/) not back slashes () when specifying a file location even if the file is on your hard drive.

```
mydata <- read.csv("https://stats.idre.ucla.edu/stat/data/binary.csv")
head(mydata)
```

```
##   admit gre  gpa rank
## 1     0 380 3.61    3
## 2     1 660 3.67    3
## 3     1 800 4.00    1
## 4     1 640 3.19    4
## 5     0 520 2.93    4
## 6     1 760 3.00    2
```

This dataset has a binary response (outcome, dependent) variable called admit. There are three predictor variables: gre, gpa and rank. We will treat the variables gre and gpa as continuous. The variable rank takes on the values 1 through 4. Institutions with a rank of 1 have the highest prestige, while those with a rank of 4 have the lowest. We can get basic descriptives for the entire data set by using summary. To get the standard deviations, we use sapply to apply the sd function to each variable in the dataset.

```
summary(mydata)
```

```
##      admit             gre             gpa             rank
##  Min.   :0.0000   Min.   :220.0   Min.   :2.260   Min.   :1.000
##  1st Qu.:0.0000   1st Qu.:520.0   1st Qu.:3.130   1st Qu.:2.000
##  Median :0.0000   Median :580.0   Median :3.395   Median :2.000
##  Mean   :0.3175   Mean   :587.7   Mean   :3.390   Mean   :2.485
##  3rd Qu.:1.0000   3rd Qu.:660.0   3rd Qu.:3.670   3rd Qu.:3.000
##  Max.   :1.0000   Max.   :800.0   Max.   :4.000   Max.   :4.000
```

```
sapply(mydata, sd)
```

```
##       admit         gre         gpa        rank
##   0.4660867 115.5165364   0.3805668   0.9444602
```

```
xtabs(~admit + rank, data = mydata)
```

```
##      rank
## admit  1  2  3  4
##     0 28 97 93 55
##     1 33 54 28 12
```

Analysis methods you might consider

Below is a list of some analysis methods you may have encountered. Some of the methods listed are quite reasonable while others have either fallen out of favor or have limitations.

```
Logistic regression, the focus of this page.
Probit regression. Probit analysis will produce results similar logistic regression. The choice o
OLS regression. When used with a binary response variable, this model is known as a linear probab
Two-group discriminant function analysis. A multivariate method for dichotomous outcome variables
Hotelling's T2. The 0/1 outcome is turned into the grouping variable, and the former predictors a
```

Using the logit model

The code below estimates a logistic regression model using the glm (generalized linear model) function. First, we convert rank to a factor to indicate that rank should be treated as a categorical variable.

```
mydata$rank <- factor(mydata$rank)
mylogit <- glm(admit ~ gre + gpa + rank, data = mydata, family = "binomial")
```

Since we gave our model a name (mylogit), R will not produce any output from our regression. In order to get the results we use the summary command:

```
summary(mylogit)
```

```
##
## Call:
## glm(formula = admit ~ gre + gpa + rank, family = "binomial",
##     data = mydata)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.6268  -0.8662  -0.6388   1.1490   2.0790
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.989979   1.139951  -3.500 0.000465 ***
## gre          0.002264   0.001094   2.070 0.038465 *
## gpa          0.804038   0.331819   2.423 0.015388 *
## rank2       -0.675443   0.316490  -2.134 0.032829 *
## rank3       -1.340204   0.345306  -3.881 0.000104 ***
## rank4       -1.551464   0.417832  -3.713 0.000205 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 499.98  on 399  degrees of freedom
## Residual deviance: 458.52  on 394  degrees of freedom
## AIC: 470.52
##
## Number of Fisher Scoring iterations: 4
```

```
In the output above, the first thing we see is the call, this is R reminding us what tl
Next we see the deviance residuals, which are a measure of model fit. This part of outp
The next part of the output shows the coefficients, their standard errors, the z-statis

    For every one unit change in gre, the log odds of admission (versus non-admission)
    For a one unit increase in gpa, the log odds of being admitted to graduate school i
    The indicator variables for rank have a slightly different interpretation. For exam
Below the table of coefficients are fit indices, including the null and deviance residu
```

We can use the confint function to obtain confidence intervals for the coefficient estimates. Note that for logistic models, confidence intervals are based on the profiled log-likelihood function. We can also get CIs based on just the standard errors by using the default method.

11.1 CIs using profiled log-likelihood

confint(mylogit)

11.2 Waiting for profiling to be done...

11.3 2.5 % 97.5 %

11.4 (Intercept) -6.271620 -1.79255

11.5 gre 0.000138 0.00444

11.6 gpa 0.160296 1.46414

11.7 rank2 -1.300889 -0.05675

11.8 rank3 -2.027671 -0.67037

11.9 rank4 -2.400027 -0.75354

11.10 CIs using standard errors

confint.default(mylogit)

11.11 2.5 % 97.5 %

11.12 (Intercept) -6.22424 -1.75572

11.13 gre 0.00012 0.00441

11.14 gpa 0.15368 1.45439

11.15 rank2 -1.29575 -0.05513

11.16 rank3 -2.01699 -0.66342

11.17 rank4 -2.37040 -0.73253

We can test for an overall effect of rank using the wald.test function of the aod library. The order in which the coefficients are given in the table of coefficients is the same as the order of the terms in the model. This is important because the wald.test function refers to the coefficients by their order in the model. We use the wald.test function. b supplies the coefficients, while Sigma supplies the variance covariance matrix of the error terms, finally Terms tells R which terms in the model are to be tested, in this case, terms 4, 5, and 6, are the three terms for the levels of rank.

wald.test(b = coef(mylogit), Sigma = vcov(mylogit), Terms = 4:6)

11.18 Wald test:

11.19 ————-

11.20

11.21 Chi-squared test:

11.22 X2 = 20.9, df = 3, P(> X2) = 0.00011

The chi-squared test statistic of 20.9, with three degrees of freedom is associated with a p-value of 0.00011 indicating that the overall effect of rank is statistically

significant.

We can also test additional hypotheses about the differences in the coefficients for the different levels of rank. Below we test that the coefficient for rank=2 is equal to the coefficient for rank=3. The first line of code below creates a vector l that defines the test we want to perform. In this case, we want to test the difference (subtraction) of the terms for rank=2 and rank=3 (i.e., the 4th and 5th terms in the model). To contrast these two terms, we multiply one of them by 1, and the other by -1. The other terms in the model are not involved in the test, so they are multiplied by 0. The second line of code below uses L=l to tell R that we wish to base the test on the vector l (rather than using the Terms option as we did above).

l <- cbind(0, 0, 0, 1, -1, 0) wald.test(b = coef(mylogit), Sigma = vcov(mylogit), L = l)

11.23 Wald test:

11.24 ————-

11.25

11.26 Chi-squared test:

11.27 X2 = 5.5, df = 1, P(> X2) = 0.019

The chi-squared test statistic of 5.5 with 1 degree of freedom is associated with a p-value of 0.019, indicating that the difference between the coefficient for rank=2 and the coefficient for rank=3 is statistically significant.

You can also exponentiate the coefficients and interpret them as odds-ratios. R will do this computation for you. To get the exponentiated coefficients, you tell R that you want to exponentiate (exp), and that the object you want to exponentiate is called coefficients and it is part of mylogit (coef(mylogit)). We can use the same logic to get odds ratios and their confidence intervals, by exponentiating the confidence intervals from before. To put it all in one table, we use cbind to bind the coefficients and confidence intervals column-wise.

11.28 odds ratios only

exp(coef(mylogit))

11.29 (Intercept) gre gpa rank2 rank3 rank4

11.30 0.0185 1.0023 2.2345 0.5089 0.2618 0.2119

11.31 odds ratios and 95% CI

exp(cbind(OR = coef(mylogit), confint(mylogit)))

11.32 Waiting for profiling to be done...

11.33 OR 2.5 % 97.5 %

11.34 (Intercept) 0.0185 0.00189 0.167

11.35 gre 1.0023 1.00014 1.004

11.36 gpa 2.2345 1.17386 4.324

11.37 rank2 0.5089 0.27229 0.945

11.38 rank3 0.2618 0.13164 0.512

11.39 rank4 0.2119 0.09072 0.471

Now we can say that for a one unit increase in gpa, the odds of being admitted to graduate school (versus not being admitted) increase by a factor of 2.23. For more information on interpreting odds ratios see our FAQ page How do I interpret odds ratios in logistic regression? . Note that while R produces it, the odds ratio for the intercept is not generally interpreted.

You can also use predicted probabilities to help you understand the model. Predicted probabilities can be computed for both categorical and continuous predictor variables. In order to create predicted probabilities we first need to create a new data frame with the values we want the independent variables to take on to create our predictions.

We will start by calculating the predicted probability of admission at each value of rank, holding gre and gpa at their means. First we create and view the data frame.

newdata1 <- with(mydata, data.frame(gre = mean(gre), gpa = mean(gpa), rank = factor(1:4)))

11.40 view data frame

newdata1

11.41 gre gpa rank

11.42 1 588 3.39 1

11.43 2 588 3.39 2

11.44 3 588 3.39 3

11.45 4 588 3.39 4

These objects must have the same names as the variables in your logistic regression above (e.g. in this example the mean for gre must be named gre). Now that we have the data frame we want to use to calculate the predicted probabilities, we can tell R to create the predicted probabilities. The first line of code below is quite compact, we will break it apart to discuss what various components do. The newdata1$rankP tells R that we want to create a new variable in the dataset (data frame) newdata1 called rankP, the rest of the command tells R that the values of rankP should be predictions made using the predict( ) function. The options within the parentheses tell R that the predictions should be based on the analysis mylogit with values of the predictor variables coming from newdata1 and that the type of prediction is a predicted probability (type="response"). The second line of the code lists the values in the data frame newdata1. Although not particularly pretty, this is a table of predicted probabilities.

newdata1$rankP <- predict(mylogit, newdata = newdata1, type = "response")
newdata1

11.46 gre gpa rank rankP

11.47 1 588 3.39 1 0.517

11.48 2 588 3.39 2 0.352

11.49 3 588 3.39 3 0.219

11.50 4 588 3.39 4 0.185

In the above output we see that the predicted probability of being accepted into a graduate program is 0.52 for students from the highest prestige undergraduate institutions (rank=1), and 0.18 for students from the lowest ranked institutions (rank=4), holding gre and gpa at their means. We can do something very similar to create a table of predicted probabilities varying the value of gre and rank. We are going to plot these, so we will create 100 values of gre between 200 and 800, at each value of rank (i.e., 1, 2, 3, and 4).

newdata2 <- with(mydata, data.frame(gre = rep(seq(from = 200, to = 800, length.out = 100), 4), gpa = mean(gpa), rank = factor(rep(1:4, each = 100))))

The code to generate the predicted probabilities (the first line below) is the same as before, except we are also going to ask for standard errors so we can plot a confidence interval. We get the estimates on the link scale and back transform both the predicted values and confidence limits into probabilities.

newdata3 <- cbind(newdata2, predict(mylogit, newdata = newdata2, type = "link", se = TRUE)) newdata3 <- within(newdata3, { PredictedProb <- plogis(fit) LL <- plogis(fit - (1.96 * se.fit)) UL <- plogis(fit + (1.96 * se.fit)) })

11.51 view first few rows of final dataset

head(newdata3)

11.52 gre gpa rank fit se.fit residual.scale UL LL PredictedProb

11.53 1 200 3.39 1 -0.811 0.515 1 0.549 0.139 0.308

11.54 2 206 3.39 1 -0.798 0.509 1 0.550 0.142 0.311

11.55 3 212 3.39 1 -0.784 0.503 1 0.551 0.145 0.313

11.56 4 218 3.39 1 -0.770 0.498 1 0.551 0.149 0.316

11.57 5 224 3.39 1 -0.757 0.492 1 0.552 0.152 0.319

11.58 6 230 3.39 1 -0.743 0.487 1 0.553 0.155 0.322

It can also be helpful to use graphs of predicted probabilities to understand and/or present the model. We will use the ggplot2 package for graphing. Below we make a plot with the predicted probabilities, and 95% confidence intervals.

ggplot(newdata3, aes(x = gre, y = PredictedProb)) + geom_ribbon(aes(ymin = LL, ymax = UL, fill = rank), alpha = 0.2) + geom_line(aes(colour = rank), size = 1)

Predicted probabilities plot

We may also wish to see measures of how well our model fits. This can be particularly useful when comparing competing models. The output produced by summary(mylogit) included indices of fit (shown below the coefficients), including the null and deviance residuals and the AIC. One measure of model fit is the significance of the overall model. This test asks whether the model with predictors fits significantly better than a model with just an intercept (i.e., a null model). The test statistic is the difference between the residual deviance for the

model with predictors and the null model. The test statistic is distributed chi-squared with degrees of freedom equal to the differences in degrees of freedom between the current and the null model (i.e., the number of predictor variables in the model). To find the difference in deviance for the two models (i.e., the test statistic) we can use the command:

with(mylogit, null.deviance - deviance)

11.59 [1] 41.5

The degrees of freedom for the difference between the two models is equal to the number of predictor variables in the mode, and can be obtained using:

with(mylogit, df.null - df.residual)

11.60 [1] 5

Finally, the p-value can be obtained using:

with(mylogit, pchisq(null.deviance - deviance, df.null - df.residual, lower.tail = FALSE))

11.61 [1] 7.58e-08

The chi-square of 41.46 with 5 degrees of freedom and an associated p-value of less than 0.001 tells us that our model as a whole fits significantly better than an empty model. This is sometimes called a likelihood ratio test (the deviance residual is -2*log likelihood). To see the model's log likelihood, we type:

logLik(mylogit)

11.62 'log Lik.' -229 (df=6)

Things to consider

```
Empty cells or small cells: You should check for empty or small cells by doing a crosst
Separation or quasi-separation (also called perfect prediction), a condition in which t
Sample size: Both logit and probit models require more cases than OLS regression becaus
Pseudo-R-squared: Many different measures of psuedo-R-squared exist. They all attempt t
Diagnostics: The diagnostics for logistic regression are different from those for OLS r
```

References

Hosmer, D. & Lemeshow, S. (2000). Applied Logistic Regression (Second Edition). New York: John Wiley & Sons, Inc.

Long, J. Scott (1997). Regression Models for Categorical and Limited Dependent Variables. Thousand Oaks, CA: Sage Publications. See also

```
R Online Manual glm
Stat Books for Loan, Logistic Regression and Limited Dependent Variables
Everitt, B. S. and Hothorn, T. A Handbook of Statistical Analyses Using R
```