Essential R: What you need to get started

Eric Nord

August 18 2017

Contents

R is the open-source statistical language that seems poised to "take over the world" of statistics and data science. R is really more than a statistical package - it is a language or an environment designed to potentiate statistical analysis and production of high quality graphics (for more on information see www.r-project.org/about/html).

Originally developed by two statisticians at the University of Auckland as a dialect of the S statistical language, since 1997 the development of R has been overseen by a core team of some 20 professional statisticians (for more on information see www.r-project.org/contributors/html).

Many new users find that R is initially hard to use. One needs to learn to write code, and there are no (or few) menu tools to make the process easier. In fact, when a grad school friend first excitedly described R to me in 2004 my first thought was "Why would I want to learn that?". I dabbled in R several times following that, but was put off by the difficulties I encountered. I finally began using R in earnest as an environment for some simulation modeling, and then later for statistical and graphical work.

These notes were developed for a short introduction to R for students who already understand basic statistical theory and methods, so the focus is mostly on R itself. Much of the inspiration for these notes comes from "SimpleR" [1] by John Verzani. ("SimpleR" was written to teach both R and introductory statistics together, but I successfully used it as the basis for an introduction to R for several years). As my course has evolved, I felt the need to develop my own materials, but the debt to John Verzani is substantial - many of the good bits in what follows are probably inspired by his work, particularly the didactic style, and the errors are certainly all mine.

A note about formatting in this document: To keep things clear, in this document, R output is shown in a black console (fixed width) font preceded by "#", like this:

```
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#   12.00   28.50   51.50   47.90   62.75   83.00
```

while R code is shown in a console font in colored text without the preceding #, like this:

```
summary(c(23, 45, 67, 46, 57, 23, 83, 59, 12, 64))
```

Where we mention R functions or arguments by name, they will appear in the console font, and the names of functions will be followed by parentheses, e.g. `mean()`.

You should be able to download these notes in a Zip file with associated folders. Unzip this to a convenient location and you will have an directory (folder) called "EssentialR". These notes assume that you are using the "EssentialR" directory as the working directory (see Ch 5), and examples point to files in the "Data" directory in "EssentialR".

Getting R

You can download R from CRAN (http://cran.r-project.org/). I also recommend that you then install the excellent RStudio IDE (http://www.rstudio.com/ide/download/) - while not strictly necessary, it makes working in R so much easier that it is worth using.

Other Resources

For an introduction to statistics using R (or a basic R reference), I recommend the following books:

Using R for Introductory Statistics. 2004. John Verzani. Chapman & Hall/CRC. (an extension of SimpleR)

Statistics: An introduction using R. 2005. Michael J. Crawley. Wiley and Sons. (This was useful enough to me when I began learning R that I bought a copy.).

Quick-R (http://www.statmethods.net) is a nice online overview of basic R functions and methods. Useful reference.

[1] "SimpleR" is also still available at http://www.math.csi.cuny.edu/Statistics/R/simpleR/.

Gardner's own (http://www.gardenersown.co.uk/Education/Lectures/R): a nice look at using R for analysis.

R Wikibook (http://en.wikibooks.org/wiki/Statistical_Analysis:_an_Introduction_using_R): an online book for a course like this.

IcebreakeR (http://cran.r-project.org/doc/contrib/Robinson-icebreaker.pdf): another PDF book for a course like this.

Also see the "Contributed Documentation" tab at CRAN (http://cran.r-project.org/doc/contrib) for links to more resources.

The citation for R (type `citation()` to get it) is as follows:

```
R Core Team (2013). R: A language and environment for statistical
  computing. R Foundation for Statistical Computing, Vienna, Austria.
  URL http://www.R-project.org/.
```

These notes were written in Markdown, and complied using the excellent R package "knitr" by Yihui Xie - for more information see: http://yihui.name/knitr/.

Of course, I would be remiss not to acknowledge the R core team and the other members of the R user community whose efforts have combined to make R such a powerful tool.

Eric Nord, Assistant Professor of Biology, Greenville College
Greenville, IL, August 2017
Version 1.39

Chapter 1: Basics

The console, the editor, and basic syntax in R

I: The Terminal

One of the things new users find strange about R is that all you have to interact with is a terminal (aka console) and (if you are using a GUI or IDE) an editor. This is very different from Excel or Minitab or similar applications. As we'll see, this has some advantages, but ease of learning might not be one of them. In the meantime, you might find it helpful to imagine that your *workspace* is a bit like a spreadsheet, and the terminal a bit like the "formula bar", where you have to type input to the spreadsheet, or see what is in the spreadsheet.

The R "terminal" is where commands can be entered and results are displayed. When you start R on your computer, it looks something like this:

```
R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.52 (6188) i386-apple-darwin9.8.0]

[Workspace restored from /Users/yourname/.RData]
[History restored from /Users/yourname/.Rhistory]

>
```

Notice the "`>`" at the end of all this? "`>`" is the R prompt - it means R is waiting for you to do something. If you are using RStudio, the terminal is on the left or lower left pane. Since R is waiting for us, let's try some things. Type (or copy & paste) the following lines (one at a time) into the console. When you press "Enter" you tell R to evaluate each expression.

```
2 + 2
```

```
# [1] 4
```

```
4 * 5
```

```
# [1] 20
```

```
6^2
```

```
# [1] 36
```

```
3 + 5 * 2
```

```
# [1] 13
```

```
(3 + 5) * 2
```

```
# [1] 16
```

```
# 2+2
```

Notice a couple of things:
1. spaces don't matter - 2+2 is the same as 2 + 2
2. the standard operators +,-,*,/,and ^ all function as expected
3. standard order of operations is respected, and can be altered using parentheses
4. all answers are preceded by [1] - we'll see why in a bit
5. an expression preceded by "`#`" is not evaluated – "`#`" is the *comment character* in R. Anything between a "`#`" and the end of a line is treated as a comment and is not evaluated.

Let's create a variable (also called an *object* in R), "a" and give it a value of 5.

```
a = 5  # =
```

Here we used the `=` to *assign* the value `5` to `a`. Those of you with some programming experience may wonder about data types. R can handle many types of data (including numerical, character, logical, factors, matrices, arrays, and lists), and we'll discuss them in more detail later. Now we have created the object a, we can see what it is:

```
a
```

```
# [1] 5
```

Notice that the value of `a`, `5`, is returned. We can also calculate with an object:

```
a * 2
```

```
# [1] 10
```

```
a
```

```
# [1] 5
```

```
a = a * 2
```

Notice that when we tell R `a*2` we get the result, but then when we type `a` we can see `a` is not changed. To *change an object* in R we must *assign* it a new value. What do you think we will get if we type `a` now? What if we type `A`? Remember that R is *case sensitive*! (We've just highlighted two of more common errors I see and make with R: 1. forgetting to assign some output that I intended to save to an object and 2. case errors.)

A note about `=` and `<-`. Assignment in R can be done using `=` or using the *assignment operator*: `<-` or `->`. The assignment operator (`<-`) is directional, and the leftward assign is *far* more common than the right. In the case of `=` the "name" is assumed to be on the left - i.e. `=` is equivalent to `<-`. In these notes I generally use `<-`, so that `=` is reserved for passing arguments to functions. (The keyboard shortcut for `<-` in RStudio is "alt+-".)

When you create an object in R it exists in your *workspace*. You can see what is in your workspace by typing `ls()`, which is short for "list". If you are using RStudio, you can also go to the "Environment" tab which is usually in the upper right pane.

You can remove things from your workspace using `rm()` - for example `rm(a)` will remove `a` from your workspace. Try `ls()` now. If you type `a` you'll get an error: `Error: object 'a' not found`. This makes sense, since we've removed (deleted) `a`. [2]

[2]Object names in R: In brief they consist of letters, numbers, and the dot and underscore characters. Names must begin with a letter or a dot followed by a letter. The dot has no special meaning in object names in R.

A note about the *console* and the *editor*. In RStudio you can go to "File>New>R script" and new pane will open above your console. This is your "Editor", and you can actually have multiple files open here (as tabs). An *R script* file contains R code and comments (see above re: comments) , but not output (though you can paste output into a script file, but you should put it behind comments so R doesn't try to run it). You can easily run code from the editor. Open an new script file and type "# editor demo". Now press Ctrl+Enter, and R will run that command (since it is a comment, it will just be written to the console). Now type "a*4" and press Ctrl+Enter.
Scripts can easily be saved for future reference or to continue your work later. The console *can't* easily be saved, but it contains the output from your R code. (There is a logic to this - if you have all the code saved, the console output is easily recreated - just run all the commands.) In the console the *up arrow* will retrieve previous commands, which can save you a fair amount of typing (often it is faster to edit a previous command than to type it over).

II: Working with Vectors

One of the ways R makes working with data easy is that *R natively handles vectors*, meaning that (almost) *anything you can do to a value in R you can do to a vector of values*. As we'll see, this becomes very useful.

Imagine that I count the number of SMS text messages I receive each day for a week and enter them in R as `sms`:

```
sms <- c(0, 1, 2, 0, 0, 0, 1)
```

Notice:
1. I get very few SMS messages - I like it this way!
2. We use a new *function* (command) here -`c`- it *concatenates* values together into a vector.
3. The function `c` is followed by parentheses `()`. All functions in R are followed by parentheses that contain the *arguments* to the function, like this: `function(argument1,argument2)`.
4. Within the parentheses the different values are separated by commas (`,`) - this is also standard in R - the arguments to a function are separated by commas, and here the values are the arguments to the function `c()`.

R can do many things easily with vectors: mathematical operations, sorting, and sub-setting.

```
sms + 5
```

```
# [1] 5 6 7 5 5 5 6
```

```
sms * 5
```

```
# [1]  0  5 10  0  0  0  5
```

```
sms/2
```

```
# [1] 0.0 0.5 1.0 0.0 0.0 0.0 0.5
```

```
sort(sms)
```

```
# [1] 0 0 0 0 1 1 2
```

Notice that if you type `sms` the original data has not been changed - to change the original data you **always need to assign the result** [3]. This is design principle in R - a function should *never change anything in your workspace*; objects in the workspace are only changed by assignment. If you want a function to change an object you must *assign the result* of the function to that object.

[3] This is a basic point that is very important.

III. Sub-setting Vectors - the magic "[]"

It is often the case when working with data that we want to select only specific parts of the data (think "filtering" in Excel). In R we do this by "sub-setting" vectors. In R the square brackets `[]` are used for *indexing* vectors, matrices, data tables, and arrays. Here we'll just consider vectors. The more you gain familiarity with R, the more you learn the power of the `[]`.

```
sms[3]
```

```
# [1] 2
```

```
sms[2:4]
```

```
# [1] 1 2 0
```

```
sms[-3]
```

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

```
sms[-(2:3)]
```

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

```
sms[-c(2, 3, 7)]
```

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

Here we've told R to give us the 3rd element of sms and the 2nd-4th elements of sms - the `:` symbol means "through" - you can test this by typing `5:9`:

```
5:9
```

```
# [1] 5 6 7 8 9
```

A minus sign "`-`" in the `[]`, `-` means "all elements except", and this can be used with a range (`2:3`) or a more complex list (`c(2,3,7)`), though it will be necessary to add parentheses as we have done here. This is because `-2:3` returns `-2 -1  0  1  2  3`, and the `-2th` element of a vector does not make sense!

In fact, a more general and useful approach is *logical extraction* - selecting parts of a vector based on some logical test. For example, if we wanted to know the average (mean) number of sms messages I received *on days that I received any sms messages*, we could do this easily with logical extraction. `sms>0` applies the *logical test* `>0` to `sms` and returns a *logical vector* (`TRUE` or `FALSE` for each element of the vector) of the result. This can be used to specify which elements of the vector we want:

```
sms > 0
```

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

```
sms[sms > 0]
```

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

It is worth noting that R treats logical values as `FALSE`=0 and `TRUE`=1. Thus `sum(sms)` will give us the total number of sms messages, but `sum(sms>0)` will give us the number of `TRUE` values in the logical vector created by the logical test `sms>0`. This is equivalent to the number of days where I received any sms messages.

```
sum(sms > 0)
```

```
# [1] 3
```

We can also use the function `which()` for logical extraction, and we can then use the function `mean()` to get the average number of sms messages on days where I received messages:

```
which(sms > 0)
```

```
# [1] 2 3 7
```

```
sms[which(sms > 0)]
```

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

```
mean(sms[which(sms > 0)])
```

```
# [1] 1.333333
```

Notice:
1. The function `which()` returns a *vector of indices* rather than a logical vector.
2. The final line here shows something that is very typical in R - we used one function, `which()` as an argument to another function, `mean()`. This type of programming can be confusing as there are square brackets and parentheses all over the place. I find that I need to build it up step by step, as you can see we've done here.
3. It is a good practice to develop the skill of reading such a statement from the inside out rather than left to right. `mean(sms[which(sms>0)])` could be read as "use only the values of sms that are greater than 0 and take the mean". 4. A small command like this already has 3 nested sets of parentheses and brackets - it is really easy to drop one by mistake [4]. If you entered `mean(sms[which(sms>0)]` (omitting the closing "`)`") R will notice that your command is incomplete, and instead of a result and a new line with the prompt (`>`) it will give no answer (as the command is not complete) and will give a `+` - this just means that R is waiting for the command to be complete.

You can also use the indexing operator to *change part of vector*:

```
sms
```

```
# [1] 0 1 2 0 0 0 1
```

```
sms[1] <- 1
sms
```

```
# [1] 1 1 2 0 0 0 1
```

This approach could be combined with logical extraction; for example if you wanted to replace all the zero values with 1: `sms[which(sms==0)]<-1`. Notice that *when making a logical test R expects the double equal sign* `==`; this is to differentiate between the assignment or argument use of the equal sign `=` and the logical use `==`. This is a common source of errors.

Recall the mysterious `[1]` that is returned with our results? This just means the first element of the result is shown. If your vector is long enough to go to a second (or third) line, each line will begin showing the element number that begins the line:

```
1:50
```

```
#  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
# [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
# [47] 47 48 49 50
```

Notice that each line begins with the index within square brackets (`[]`) for that element.

We've already seen how we can change elements of a vector using the indexing operator. There are other tools in R that allow us to edit data. The most useful in RStudio is `edit()`. For example, `edit(sms)` will bring up a small edit window in R studio. If you change the first element from 1 back to zero and press save, the console will show the edited value of `sms`.

However if you type `sms` again, you'll find that the original data hasn't been altered. As we said before, if you want to change values in R *you have to assign the result of you change* - so `sms<-edit(sms)` would actually

[4] Tip: Many text editing programs (including the editor in RStudio) will balance parentheses and brackets - when you type `(` the matching `)` is added, and the cursor is placed within them. Also, when you place your cursor on a parenthesis or bracket the matching on will be highlighted. If your text editor does not do this, find one that does - it will save you many headaches.

save the changes. Of course, you might want to make a copy of the data and change that - simple, just assign the result of edit to another name: `sms.2<-edit(sms)`.

It is also worth noting that you can use one variable to index another. First we'll make some vectors.

```
x <- c("a", "b", "c", "d", "e", "f", "g", "h", "i")
y <- 21:30
z <- c(2, 4, 6)
```

Now we can try various uses of `[]`:

```
x[y]
```

```
#  [1] NA NA NA NA NA NA NA NA NA NA
```

What is going on here? What does the `NA` mean?

```
x[z]
```

```
# [1] "b" "d" "f"
```

The `z-th` element of `x`.

```
y[z]
```

```
# [1] 22 24 26
```

```
z[y]
```

```
#  [1] NA NA NA NA NA NA NA NA NA NA
```

The second line here gives many NA's since `z` doesn't have enough elements to match some values of `y`.

```
x[rev(z)]
```

```
# [1] "f" "d" "b"
```

```
y[x]
```

```
# [1] NA NA NA NA NA NA NA NA NA
```

`rev()` just reverses its argument. The `x-th` elements of `y` fails because `x` can't be "coerced"" to numeric.

```
y * z
```

```
# Warning in y * z: longer object length is not a multiple of shorter object
# length
```

```
#  [1]  42  88 138  48 100 156  54 112 174  60
```

This warning is important. One might naively assume that R won't do the element-wise addition or multiplication of two vectors of different lengths, but it *does* by *recycling* the shorter vector as necessary, with a warning if the length of longer one is not a multiple of the length of the shorter. *Pay attention* to your vector lengths!

IV. Other Useful Functions

The R functions `de()` and `data.entry()` are similar to `edit()`, but aren't supported in RStudio, so we'll skip them here. It is worth noting that these functions (`edit()`,`de()`,and `data.entry()`) are *interactive* - that is they require user input outside of the terminal. There are 2 important things to know about interactive functions in R:
1. When you invoke an interactive function, *R waits for you to be done with it* before R will continue. In RStudio you will see a small red "stop sign" icon just above the console pane. This means R is waiting for

you do do something. (The first time this happened to me I thought R had frozen and I forced R to quit).
2. When you use an interactive function any changes you make to data this way are *not recorded in your code* - if you make your changes to your data via the code you write, then saving your code *preserves a record of what you have done*. If you are like me, and have ever found yourself looking at a spreadsheet that you (or someone else) made some time ago, and wondering "what was happening here?", you will see the benefit of having everything you do recorded in your code. It increases the *transparency* of your analysis.

Some other useful functions for working with vectors of data are:

Function	Description
`sd()`	standard deviation
`median()`	median
`max()`	maximum
`min()`	minimum
`range()`	maximum and minimum
`length()`	number of elements of a vector
`cummin()`	cumulative min (or max `cummax()`)
`diff()`	differences between successive elements of a vector

A couple of additional hints that you'll find helpful:
1. If you type some function in R and press `Return`, but R gives you `+` instead of the answer, it means you have not completed something - most often this means a parenthesis has not been closed.
2. Related to 1.: in RStudio when you type "(" the ")" is automatically placed - this is to help you keep track of your "()"s. Also when your cursor is beside a "(" the matching ")" is highlighted, and vice-versa.
3. You will occasionally see a semicolon (`;`) used - this simply allows two functions to be submitted on one line. This is generally rather rare, since R functions tend to be longer rather than shorter when doing anything complex. 4. The comment character in R is the hash (`#`) - anything between `#` and the end of the line is treated as a comment and is not evaluated. Adding comments in your R code is a good idea - it helps you remember where you are and what you are doing (transparency!)

V. Loops in R.

Often we want to repeat something many times. Sometimes a loop is a good way to do that, although in R the way vectors are handled natively obviates the need for many loops (e.g. no need to write a loop to add a two vectors), and code that uses vectorization is usually much faster than code that uses loops.

Loops are initiated with the function `for()`, with an index a the argument. The loop is usually enclosed in curly braces "{}", but if it all fits on one line it doesn't need the braces.

```
for (i in 1:27) {
    cat(letters[i])
}
```

```
# abcdefghijklmnopqrstuvwxyzNA
```

```
for (i in sample(x = 1:26, size = 14)) cat(LETTERS[i])
```

```
# HVXFGYNTMSUOED
```

Notice - these loops are both the sort that could be avoided. `letters[1:27]` would work in place of the first. Also note that `letters` and `LETTERS` are built in. Also note that the index does not need to be a number - it can be an object or something that evaluates to a number. Lastly, note that values aren't written to the console when a loop is run unless we specify that they should be using a function such as `cat()` or `print()`.

```
x <- c("Most", "loops", "in", "R", "can", "be", "avoided")
for (i in x) {
    cat(paste(i, "!"))
}
```

```
# Most !loops !in !R !can !be !avoided !
```

VI. Exercises.

1) Enter the following data in R and call it `P1`:
`23,45,67,46,57,23,83,59,12,64`
What is the maximum value? What is the minimum value? What is the mean value?

2) Oh no! The next to last (9th) value was mis-typed - it should be 42. Change it, and see how the mean has changed. How many values are greater than 40? What is the mean of values over 40? *Hint*: how do you see the 9th element of the vector `P1`?

3) Using the data from problem **2** find:
a) the sum of `P1`
b) the mean (using the sum and `length(P1)`)
c) the log(base10) of `P1` - use `log(base=10)`
d) the difference between each element of `P1` and the mean of `P1`

4) If we have two vectors, `a<-11:20` and `b<-c(2,4,6,8)` predict (*without running the code*) the outcome of the following (Write out your predictions as comments in your HW. After you make your predictions, you can check yourself by running the code). Were you correct? a) `a*2`
b) `a[b]`
c) `b[a]`
d) `c(a,b)`
e) `a+b`

Chapter 2: Qualitative Variables

Creating and using categorical variables in R

I. Introduction

In the last chapter we saw how we can work with vectors in R. Data that we work with in R is generally stored as vectors (though these vectors are usually part of a *data frame*, which we'll discuss in the next chapter). Of course there are several types of data that we might need to work with - minimally we need qualitative data (categorical data - *factors* in R-speak) and quantitative data (continuous or numerical data - *numeric* in R-speak), but strings (*character* in R) are often useful also. Today we'll consider the first two of these data types, and learn how to work with them in R.

Since there are different types of data, it is very important to know:
A. What type of data you have (by this I mean what it *represents*).

As our focus here is R, I won't dwell on this except to say that it is worth taking time to be clear about this when designing your data. For example, if four replicates are labelled "1, 2, 3, 4", then R is likely to treat replicate is a numerical variable. If they are labelled "A, B, C, D", or "I, II, III, IV", this can be avoided.

B. What R thinks the data is (or how it is *encoded* in R).

The most common data types for vectors in R are: `"logical","integer","double", and "character"`. There are several other types that you may never encounter and several types that apply to more complex structures that we'll explore later.

There are a couple of ways to find out how R is storing data. The function `str()` ("structure") will give you the basic data type. The function `summary()` gives summary statistics for numeric variables, but number of levels for factors. This works well because it also lets you quickly see if you have miss-coded data (e.g typos like "IIi" in place of "III") or extreme outliers.

C. How to ensure that the answers to **A.** and **B.** are the same!

II. The Function `factor()`.

Qualitative data, or categorical data is stored in R as *factors*. We can use the function `factor()` to *coerce* (convert) a vector to a factor, as demonstrated here:

```
cols <- c("Blue", "Blue", "Red", "Red", "Blue", "Yellow", "Green")
summary(cols)
```

```
#    Length     Class      Mode
#         7 character character
```

```
cols[2]
```

```
# [1] "Blue"
```

```
cols <- factor(cols)
cols[2]
```

```
# [1] Blue
# Levels: Blue Green Red Yellow
```

Notice that this factor was created as a character variable - the elements still have quotes around them. After we convert it to a factor, even returning one element (`cols[2]`) we can see that there are no quotes and we get the levels reported. The structure (`str(cols)`) reports that it is factor, and shows us the numeric representation of it (we'll discuss this more in a bit). The summary (`summary(cols)`) shows us the frequency

for (some of) the levels [5]. Now that we have `cols` as a factor, we can investigate its properties. The function `levels()` shows us all the levels for a factor.

```
str(cols)
```

```
#  Factor w/ 4 levels "Blue","Green",..: 1 1 3 3 1 4 2
```

```
summary(cols)
```

```
#   Blue  Green    Red Yellow
#      3      1      2      1
```

```
levels(cols)
```

```
# [1] "Blue"   "Green"  "Red"    "Yellow"
```

We can use the function `table()` to see a frequency table for our factor. Note: We can use `table()` on character or numeric vectors also - `table()` will coerce its argument(s) to factor if possible (though of course it doesn't *store* the factor - objects are only stored if you explicitly call for them to be stored).

```
table(cols)
```

```
# cols
#   Blue  Green    Red Yellow
#      3      1      2      1
```

```
b <- table(cols)
b[3]
```

```
# Red
#   2
```

```
b[3] * 4
```

```
# Red
#   8
```

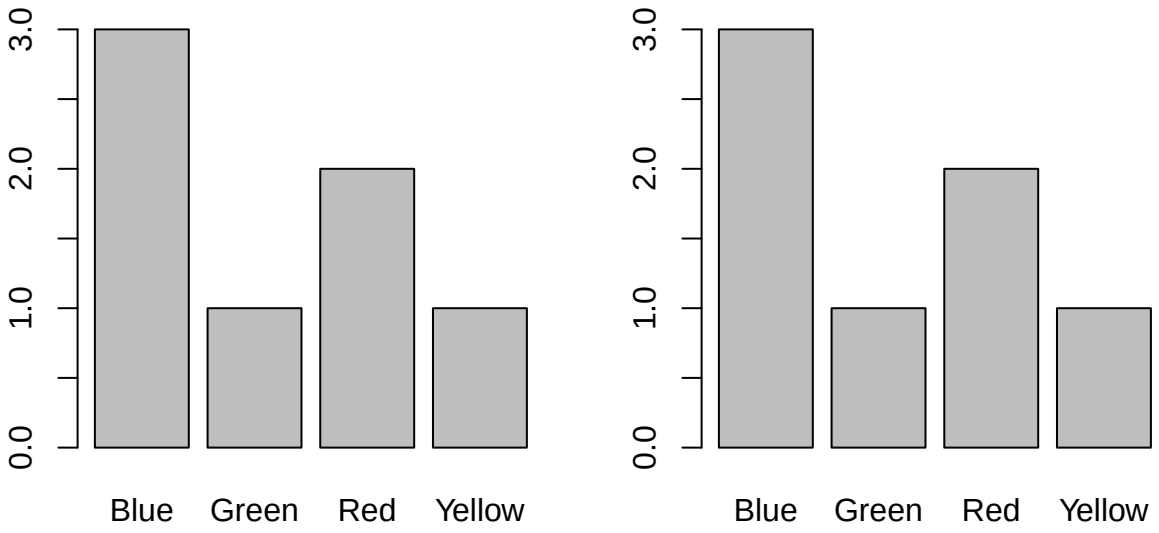
Notice that the frequency table created by `table()` is itself an R *object*, meaning that we can assign it to another name (`b` in this example), and we can access parts of it in the normal way, and use it in further calculations. Using functions to return or store (save) objects is a very common task in R, as you will see, and *many functions return objects*.

III. Visualizing Qualitative Variables.

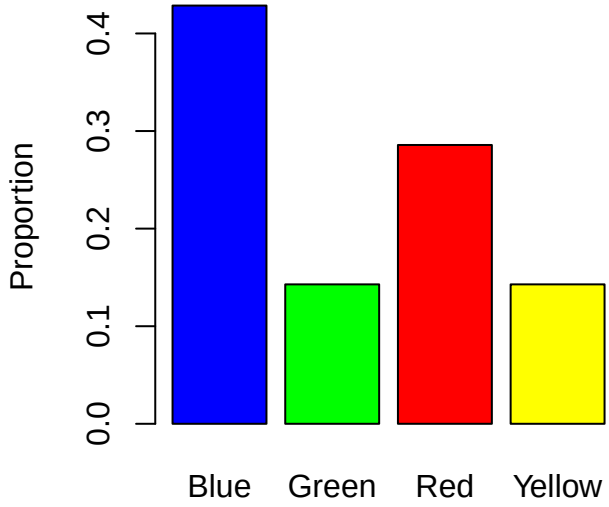
If we want a graphical summary of a factor, we can make a barplot (`barplot()`). However, we need to use `table()` with `barplot()`, since `barplot()` requires the values it will plot (its `height` argument) in a numeric form (i.e. a vector or a matrix; see `?barplot` for more detail).

```
barplot(table(cols))
plot(cols)
```

[5] If there were many levels, only the first 5 or 6 would be shown.

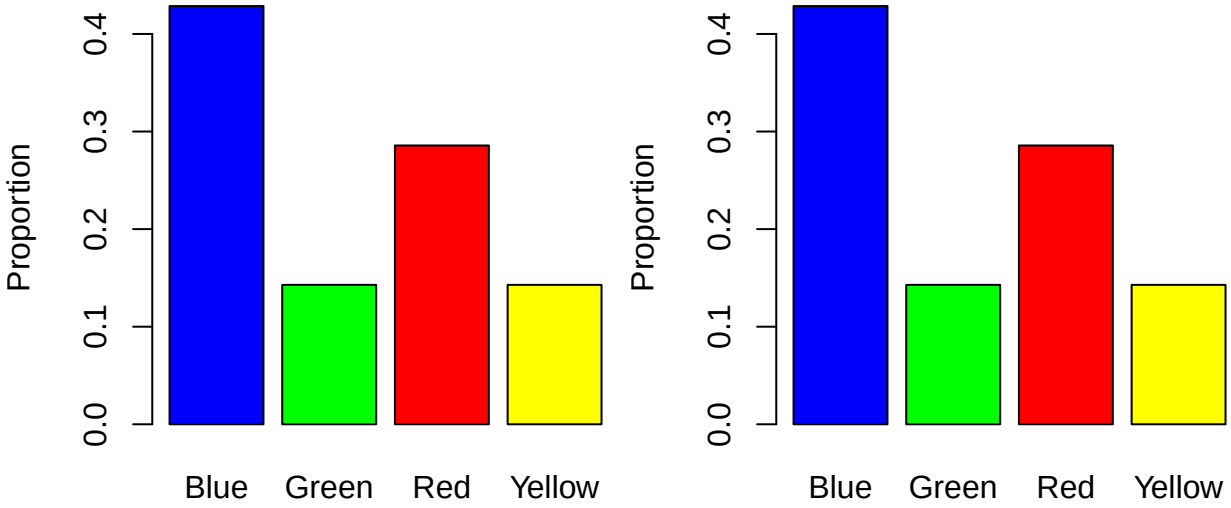



```
barplot(b/length(cols), col = c("blue", "green", "red", "yellow"), ylab = "Proportion")
```



Note that `plot(cols)` gives a barplot in the second plot - this is because the data is categorical. In the third plot we used `b` (recall that earlier we assigned `table(cols)` to `b`) either way works.

```
barplot(b/length(cols), col = c("blue", "green", "red", "yellow"), ylab = "Proportion")
barplot(table(cols)/length(cols), col = levels(cols), ylab = "Proportion")
```



The first plot here demonstrates how we can easily add color to plots and that we can carry out calculations within the call to `barplot()` (e.g. to calculate proportions). We can also specify the y-axis label (the argument is `ylab`). The second plot demonstrates how we can use the output of `levels()` to specify our colors. This only makes sense in a minority of cases, but it is an example of nesting functions - `table()`, `length()`, and `levels()` are all used to supply arguments to `barplot()`.

> Notice that the `col` argument to `barplot()` is *optional* - barplot works fine if we don't specify `col`, but we have the option to do so if need be. This is a common feature of R functions - minimal arguments are *required*, but there are often many optional arguments, which often have well chosen default values.

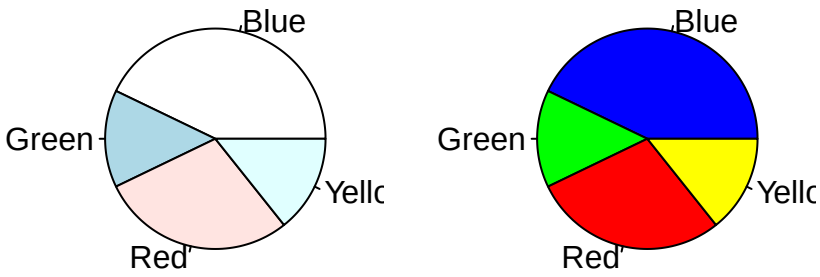
We can also convert a factor to a logical vector (by using a logical test) should we need to for sub-setting:

```
cols == "Red"
```

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

We can also create a pie chart quite easily (though see `?pie` for why this might not be a good idea).

```
pie(table(cols))
pie(b, col = c("blue", "green", "red", "yellow"))
```



Another example, this time with a numerically coded factor.

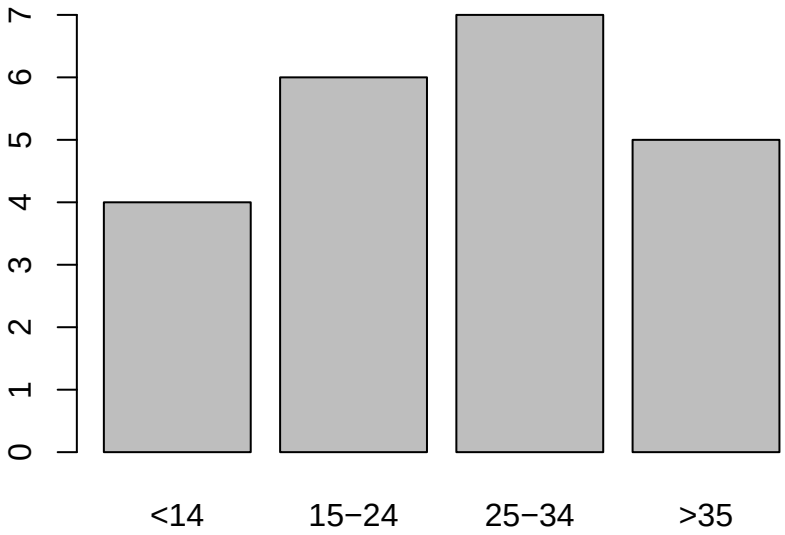
```
a <- factor(scan(text = "2 4 3 3 2 1 1 2 3 4 2 3 3 4 1 3 2 1 4 3 2 4"))
# scan(text='some text string with spaces separating value')
table(a)
```

```
# a
# 1 2 3 4
# 4 6 7 5
```

```
levels(a) <- c("<14", "15-24", "25-34", ">35")
table(a)
```

```
# a
#   <14 15-24 25-34   >35
#     4     6     7     5
```

```
barplot(table(a))
```



First notice that we introduce the function `scan()` here - when entering a longer list like this it may be easier than using `c()`, as commas don't need to be entered. Second note that we can use the function `levels()` to *set* levels as well as to return them (a number of other R functions display this dual utility as well). For a longer factor variable, it might be faster to enter it this way than by repeatedly entering all the factor levels [6] .

IV. How Factors are Stored in R

R stores factors as a list of levels and an integer vector representing the level of each element of the factor. So our factor `a`, with values `1,1,4,5` has three levels: `1 4 5`.

```
a <- c(1, 1, 4, 5)
str(a)
```

```
#  num [1:4] 1 1 4 5
```

```
(a <- as.factor(a))
```

```
# [1] 1 1 4 5
# Levels: 1 4 5
```

```
str(a)
```

```
#  Factor w/ 3 levels "1","4","5": 1 1 2 3
```

```
levels(a)
```

```
# [1] "1" "4" "5"
```

[6] In practice I enter rather little data when using R for analysis - mostly I import e data, as we'll see later.

Notice that `str(a)` shows that the original values have been replaced by the level numbers, which are `1,2,3`. This can create an unwelcome surprise if you are trying to use values from a factor variable in a calculation! For this reason, it is probably best to avoid using the integer values from a factor in calculations. Note that while factors levels are stored as a vector of numbers, it does not make sense to treat these as numbers - in this example the level "4" is represented by the value "2". If our levels were "blue","red","green", it clearly makes no sense to assume that because "blue" is level 1, and "green" is level 2 that "green" = twice "blue".

Note that in the third line we put parentheses around the assignment - this is equivalent to `a<-as.factor(a);a` - it both carries out the assignment and shows us the new value of `a`. This is occasionally useful. The function `levels()` returns the levels for a factor (what do you think it does for a non-factor variable?)

We already saw how we can use `factor()` and `as.factor()` will convert character or numeric data to factor data, and `as.numeric()` will (sort of) do the opposite.

```
as.numeric(a)
```

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

As you can see in this case we don't get the original values, we get the integer representation. We can also see this in the output from `str(a)`. If necessary, this can be solved by first converting to character and then to numeric [7] - `as.numeric(as.character(a))` returns: 1, 1, 4, 5.

```
as.numeric(as.character(a))
```

```
# [1] 1 1 4 5
```

Since factor levels are characters, they can be a mix of alphabetic and numeric characters, but that will clearly cause problems if we want to coerce the factor to a numeric vector.

```
(b <- c("-.1", " 2.7 ", "B"))
```

```
# [1] "-.1"   " 2.7 " "B"
```

```
str(b)
```

```
#  chr [1:3] "-.1" " 2.7 " "B"
```

```
as.numeric(b)
```

```
# Warning: NAs introduced by coercion
```

```
# [1] -0.1  2.7   NA
```

Here we entered the values with quotes, which created a character variable, (as shown by the `chr` returned by `str()`). When we convert the vector to numeric, the non-numeric value ("B"), can't be coerced to a number, so it is replaced by `NA`, thus the warning `NAs introduced by coercion`. This warning will occasionally show up when a function coerces one of its arguments.

V. Changing Factor Levels

Occasionally we need to change the levels of a factor - either to collapse groups together or to correct typos in the data.

```
cols <- factor(c("Blue", "Blue", "Red", "Red", "Bleu", "Yellow", "Green"))
levels(cols)
```

```
# [1] "Bleu"   "Blue"   "Green"  "Red"    "Yellow"
```

[7] Note that this is not the most computationally efficient way to accomplish this, but is the easiest to remember. The recommended approach is `as.numeric(levels(a)[a])`, which showcases the power of the `[]` in R.

Here we have mis-typed "Blue" as "Bleu" (I do this kind of thing all the time). We can use the function `levels()` to *set* levels as well as to query them. The key is that since there are currently 5 levels, we must specify 5 levels that correspond to the 5 current levels, but we don't have to specify *unique* levels.

```
levels(cols) <- c("B", "B", "G", "R", "Y")
levels(cols)
```

```
# [1] "B" "G" "R" "Y"
```

Now we have only four levels - by assigning "B" to both "Blue" and "Bleu" we collapsed them together. This is not reversible - if we wanted to get "Bleu" back, we'd have to reload the data. This is where making a copy of the vector before you start tweaking it may be a good idea (though if you are writing all your code in an R script, it is quite easy to get back to where you started - just run all the commands again).

Note that the new levels can be the same as the old levels - in the example above I avoided that just for clarity.

```
cols <- factor(c("Blue", "Blue", "Red", "Red", "Bleu", "Yellow", "Green"))
levels(cols) <- c("Blue", "Blue", "Green", "Red", "Yellow")
levels(cols)
```

```
# [1] "Blue"   "Green"  "Red"    "Yellow"
```

In fact we can supply the same levels in a different order, though that probably doesn't make much sense.

```
cols <- factor(c("Blue", "Blue", "Red", "Red", "Blue", "Yellow", "Green"))
levels(cols)
```

```
# [1] "Blue"   "Green"  "Red"    "Yellow"
```

```
levels(cols) <- c("Yellow", "Blue", "Green", "Red")
levels(cols)
```

```
# [1] "Yellow" "Blue"   "Green"  "Red"
```

Since there are four levels, we must supply a vector of four levels, but they don't need to all be different:

> Note: We could also use the function `replace()` to change factor levels, but to do this we first have to convert the factor to character using `as.character()`, so this method is not generally as useful.

It is worth noting that once a level is created for a factor, the level persists even if the corresponding data is removed.

```
cols
```

```
# [1] Yellow Yellow Green  Green  Yellow Red    Blue
# Levels: Yellow Blue Green Red
```

```
cols[-6]
```

```
# [1] Yellow Yellow Green  Green  Yellow Blue
# Levels: Yellow Blue Green Red
```

Notice the level `Red` is still present, even though there are no values with that level. This is occasionally annoying. The function `droplevels()` can be used to drop unused factor levels you can check this by running `droplevels(cols[-6])`.

Occasionally we want to impose our own order on a factor rather than accept the default (alphanumeric) order given by R. For example, a factor with levels "Low", "Medium" and "High" would be default ordered as "High", "Low", "Medium", which doesn't make sense. We can use the use the `levels` argument of `factor()` to set the orders as we'd like them to be.

```r
x <- factor(c("L", "M", "H"))
y <- factor(x, levels = c("L", "M", "H"))
x
```

```
# [1] L M H
# Levels: H L M
```

```r
y
```

```
# [1] L M H
# Levels: L M H
```

In such instances it may be useful to create an *ordered* factor.

```r
z <- factor(x, levels = c("L", "M", "H"), ordered = TRUE)
z
```

```
# [1] L M H
# Levels: L < M < H
```

Notice that the levels are listed from lowest to highest, and are shown with the "<" indicating the order. We can also see this when we call `str()` on an ordered factor. Also notice that when we create the ordered factor `z` we begin with `x` that already has the levels `"L","M","H"`. If `x` had other levels we'd need to set the correct levels first via `levels()`, or use the argument `labels` for `factor()`.

```r
str(y)
```

```
#  Factor w/ 3 levels "L","M","H": 1 2 3
```

```r
str(z)
```

```
#  Ord.factor w/ 3 levels "L"<"M"<"H": 1 2 3
```

Another potential advantage of an ordered factor is the ability to perform logical tests on the factor.

```r
y > "L"
```

```
# Warning in Ops.factor(y, "L"): '>' not meaningful for factors
```

```
# [1] NA NA NA
```

```r
z > "L"
```

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

VI. Hypothesis Testing for Factors

We may want to test hypotheses about a qualitative variable. For example, if we roll a die 50 times and get "6" 12 times how likely is it that the die is fair? (This really is a factor - it just have numeric levels.)

We can use the proportion test in R (`prop.test()`) to compare an observed frequency against a hypothesized frequency and calculate a p-value for the difference. Here our observed frequency is 12 out of 50, and the theoretical probability is 1/6. Our alternative hypothesis is that the probability is greater than 1/6.

```r
prop.test(x = 12, n = 50, p = 1/6, alt = "greater")
```

```
#
#   1-sample proportions test with continuity correction
#
# data:  12 out of 50, null probability 1/6
# X-squared = 1.444, df = 1, p-value = 0.1147
# alternative hypothesis: true p is greater than 0.1666667
```

```
# 95 percent confidence interval:
#  0.1475106 1.0000000
# sample estimates:
#    p
# 0.24
```

The p-value here is 0.115, so we don't have very strong evidence of an unfair die.

EXTRA: Simulating a hypothesis test for a qualitative variable

Another way to approach this question is to simulate the problem in R. The function `sample()` will randomly choose values, so `sample(1:6)` would be like rolling a fair die, and `sample(1:6,size=50,replace=TRUE)` like rolling the die 50 times. Adding the logical test `==6` asks how many 6's come up, and calling `sum()` on the logical test adds up the number of `TRUE`s (recall from Chapter 1 that logical values can be interpreted as 0 or 1).

```
sample(x = 1:6, size = 50, replace = TRUE)  # rolling a die 50 times
```

```
#  [1] 6 3 6 3 1 6 4 3 2 4 6 4 3 6 6 4 6 1 5 5 5 1 2 1 4 5 6 2 5 3 5 5 6 3 6
# [36] 5 1 2 4 2 4 2 6 1 4 3 3 3 2 4
```

```
sum(sample(1:6, 50, TRUE) == 6)  # how many times is it 6?
```

```
# [1] 5
```

You can easily use the up arrow in the console to repeat this - you'll see that the number of 6's varies. If we repeated this 100 times we could see how frequent a value of 12 or greater is. To do this we'll use a *loop*. First we'll create a vector of `NA`s to store the data, then we'll use a loop to run `sum(sample(1:6,50,TRUE)==6)` 100 times.

```
die <- rep(NA, 100)  # vector to store results
for (i in 1:100) {
    die[i] <- sum(sample(1:6, 50, TRUE) == 6)
}
table(die)
```

```
# die
#  2  3  4  5  6  7  8  9 10 11 12 13 14 15
#  2  2  2  4  9 10 16 14 16  8  9  4  2  2
```

```
sum(die >= 12)
```

```
# [1] 17
```

So a value of 12 or greater comes up 17% of the time, which is a bit different from the p-value we got from `prop.test()`. To get a more stable p-value we need to try this 1000 time rather than a hundred (go ahead and try it if you like) We don't have strong enough evidence to conclude that the die is *not* fair. This is much faster than rolling a die 5000 times and recording the results!

> Note: here we created a vector to store the results before running the loop. This is recommended, as it is more efficient, but you *can* "grow" a vector from inside a loop.

VII. Exercises.

1) The function `rep()` makes repeated series - for example try `rep(4,times=5)` and `rep(c(1,2),each=3)`. Use `rep()` to enter the sequence `1 1 1 1 2 2 2 2 3 3 3 3` repeated 3 times. Now convert it to a factor with the levels "Low","Medium", and "High". Can you change the levels to "Never","Rarely","Sometimes"?

2) Convert the factor from Problem 1 into a character vector, and save it (assign it a name). Can you convert the character vector to a numeric vector?

3) Earlier we used the factor variable `a` (created by `a<-factor(scan(text="2 4 3 3 2 1 1 2 3 4 2 3 3 4 1 3 2 1 4 3 2 4"))`). Convert `a` into an ordered factor with levels `"Sm","Med","Lg","X-Lg"` (with `1` for "Sm"). How many values are equal to or larger than "Lg"?

4) We can use the output of `table()` with `barplot()` to view the frequency of levels in a factor. Extending our discussion of rolling die, we can use this to view the likelihood of rolling any particular value on one die using `barplot(table(sample(x=1:6,size=1000,replace=TRUE)))`. How does this change if we add the rolls of 2 dice together? (*Hint:* recall that vectors are added in an element-wise fashion, and that `sample()` returns a vector).

Extra What happens if the two dice have different numbers of sides?

Chapter 3: Quantitative Variables

Creating and using continuous variables in R

I. Introduction

In the last chapter, we began working with qualitative data. Now we'll look at how to handle quantitative (continuous, or numerical) data.

II. Working with Numeric Data

In the first chapter we saw some functions that can be used with numeric vectors - here we'll demonstrate a bit more. We'll begin with some data that represents the size of a group of mp3 files (in MB), and get some summary statistics:

```
mp3 <- scan(text = "5.3 3.6 5.5 4.7 6.7 4.3 4.3 8.9 5.1 5.8 4.4")
mean(mp3)
```

```
# [1] 5.327273
```

```
var(mp3)
```

```
# [1] 2.130182
```

```
sd(mp3)
```

```
# [1] 1.459514
```

```
median(mp3)
```

```
# [1] 5.1
```

```
summary(mp3)
```

```
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#   3.600   4.350   5.100   5.327   5.650   8.900
```

These functions mostly do what we'd expect. The function `fivenum()` gives similar output to `summary()`, but differ slightly, since `fivenum()` returns the upper and lower hinges [8], while `summary()` returns the 1st and 3rd quartiles, and these can differ slightly depending on the number of data points.

```
quantile(mp3, c(0.25, 0.75))
```

```
#  25%  75%
# 4.35 5.65
```

```
quantile(mp3, c(0.18, 0.36, 0.54, 0.72, 0.9))
```

```
#  18%  36%  54%  72%  90%
# 4.30 4.58 5.18 5.56 6.70
```

Notice that the function `quantile()` can return any desired quantile.

[8] The "upper hinge" is the median of the data points above the median. Depending on the number of data points, this may differ from the 3rd quartile.

III. Hypothesis Testing

As we did for the qualitative data we can test hypotheses about quantitative data. For example, if we thought the mean was 4.5, we could test if the data support this by making a t-test. > Recall that `t` is the difference between observed and hypothesized means in units of the standard error, and standard error of the mean is standard deviation divided by the square root of n, and note that we can use `pt()` to calculate probabilities for a t-distribution. See "?Distributions" for more distributions.

```
t <- (mean(mp3) - 4.5)/(sd(mp3)/sqrt(length(mp3)))
pt(t, df = length(mp3) - 1, lower.tail = FALSE) * 2  # *2 for 2 sided test
```

```
# [1] 0.08953719
```

Recall that `length(mp3)^0.5` is the square root of n; we could also use `sqrt(length(mp3))`.

Of course, R has a built in t-test function that saves us the work:

```
t.test(mp3, mu = 4.5)
```

```
#
#   One Sample t-test
#
# data:  mp3
# t = 1.8799, df = 10, p-value = 0.08954
# alternative hypothesis: true mean is not equal to 4.5
# 95 percent confidence interval:
#  4.346758 6.307788
# sample estimates:
# mean of x
#  5.327273
```

We provide the null value of the mean with the argument `mu`.

IV. Resistant measures of center and spread

Since the mean and standard deviation can be quite sensitive to outliers, it is occasionally useful to consider some *resistant* measures of center and spread, so-called because they resist the influence of outliers. We'll add an outlier to our `mp3` data and experiment.

```
mp3[8] <- 10.8
mean(mp3)
```

```
# [1] 5.5
```

```
median(mp3)
```

```
# [1] 5.1
```

```
mean(mp3, trim = 0.1)
```

```
# [1] 5.122222
```

The median is substantially lower than the mean, but trimmed mean [9] is much nearer the median. Trimming more of the data will get still closer to the median.

For resistant measures of spread, one candidate is the "Interquartile range" or IQR, defined as the difference between the 3rd and 1st quartiles. Another candidate is the "median absolute deviation" or MAD, defined as

[9] The trimmed mean is taken after removing ("trimming") the upper and lower ends of the data, in this case we specified 10% via the `trim=0.1` argument.

the median of the absolute differences from the median, scaled by a constant [10]. If that sounds complex, it is simple in R, since R works easily with vectors.

```
IQR(mp3)
```

```
# [1] 1.3
```

```
median(abs(mp3 - median(mp3))) * 1.4826
```

```
# [1] 1.03782
```

```
mad(mp3)
```

```
# [1] 1.03782
```

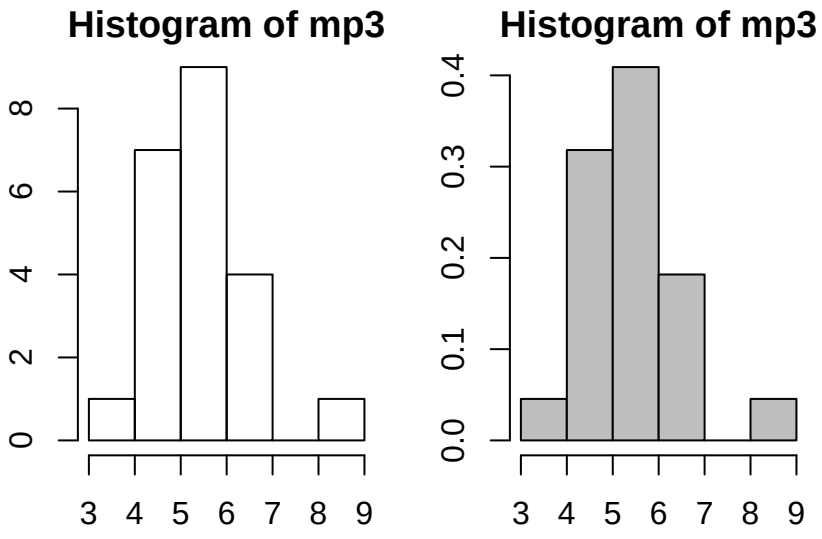
Of course, there is already a function for MAD, so we don't need to do it "by hand".

V. Visualizing Quantitative Data

One of the things we often want to do with qualitative data is "have a look". There are several ways to do this in R, and we'll review them here. The first and most common is the **histogram**. First we'll add another album's mp3 file sizes to our data `mp3` - note that `c()` can be used to combine vectors also.

```
mp3[8] <- 8.9
mp3 <- c(mp3, scan(text = "4.9 5 4.9 5.4 6.2 5.6 5.1 5.8 5.5 6.7 7"))
```

```
par(mfrow = c(1, 2))  # split the plot
hist(mp3)
hist(mp3, prob = TRUE, col = "grey")
```

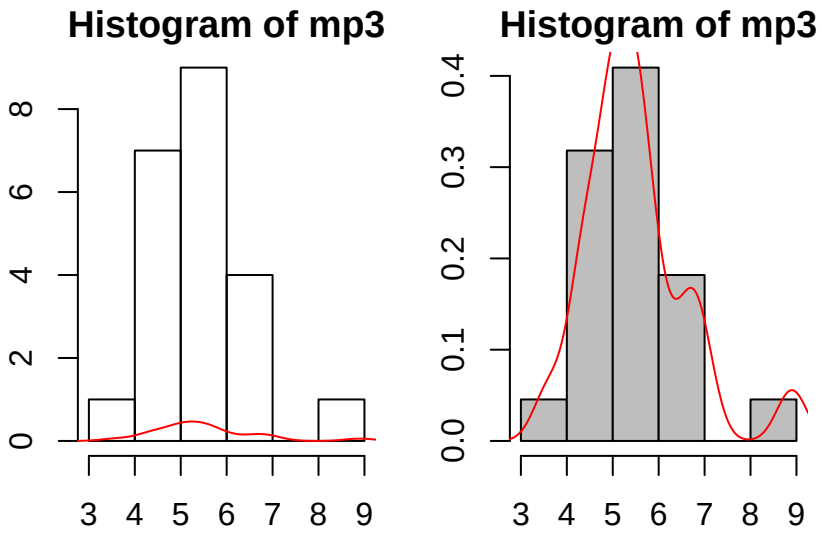


We have 2 versions of the histogram here - in the first, the y-axis is in units of *frequency*, so the scale changes for differing `n`, while the second is in units of *probability*, so distributions with differing `n` can be compared. Another useful visualization is the **kernel density estimate** (KDE), or density estimate, which approximates a probability density function.

```
par(mfrow = c(1, 2))  # split the plot
hist(mp3)
lines(density(mp3), col = "red")
```

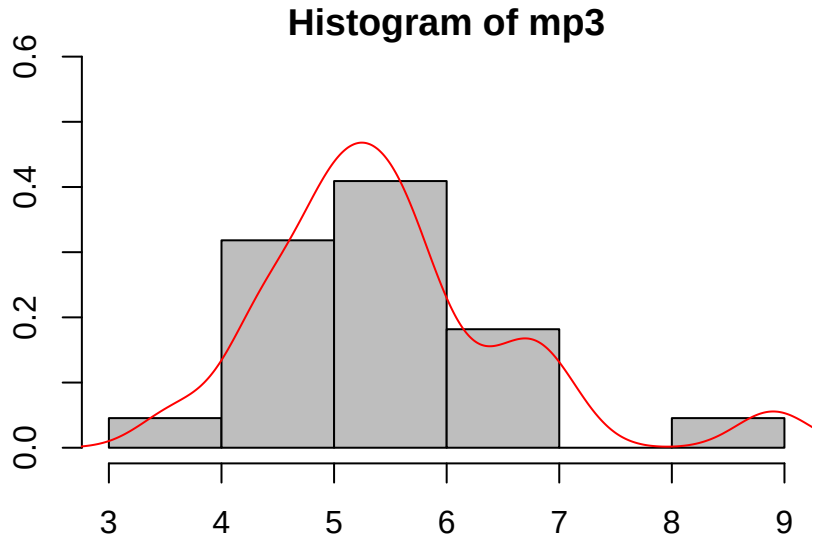
[10] The default value of the constant is 1.4826, as this gives a value comparable to standard deviation for normally distributed data.

```
hist(mp3, probability = TRUE, col = "grey")
lines(density(mp3), col = "red")
```



Note that the KDE approximates the histogram (and should have the same area), but for over-plotting on the histogram, the histogram must be in units of probability. In a case like this where our density function is off the scale, we might need to force the histogram to use a longer y-axis, which we can do using the `ylim` argument to specify the y-axis limits. (We'll use this optional argument with many plotting functions)

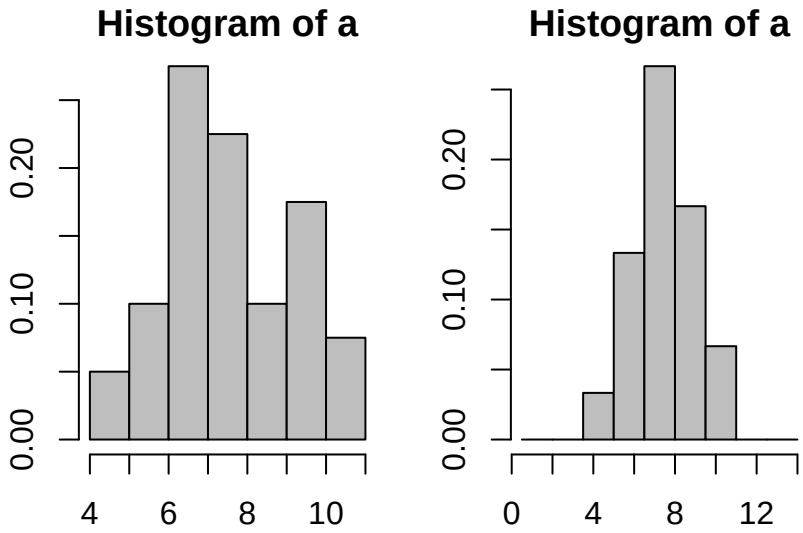
```
hist(mp3, probability = TRUE, col = "grey", ylim = c(0, 0.6))
lines(density(mp3), col = "red")
```



Here's another example, using `rnorm()`[11] to generate some random data from the normal distribution.

```
par(mfrow = c(1, 2))  # split the plot
a <- rnorm(n = 40, mean = 7, sd = 2)
hist(a, prob = T, col = "grey")
hist(a, prob = T, col = "grey", breaks = seq(0.5, 14, 1.5))
```

[11] The `r` in `rnorm()` means "random"; `runif()` generates some uniformly distributed random data, and many others are included - see `?Distributions`



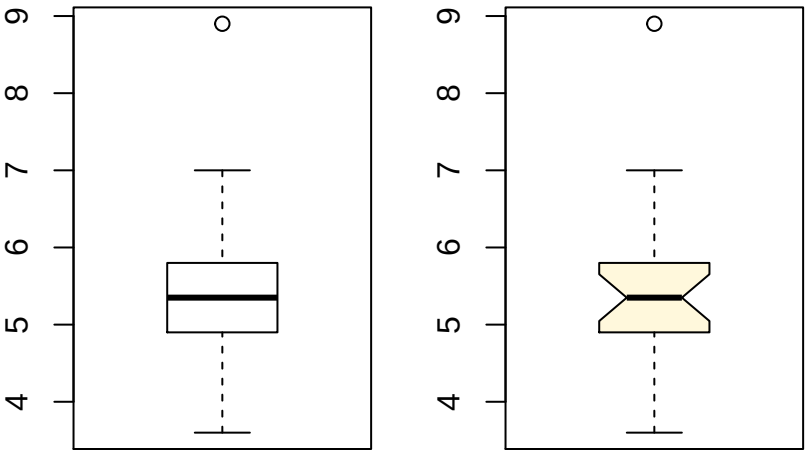
Notice that these two histograms represent the same data - this is one of the weaknesses of histograms: the idea they give us about the data depends on the bins used. This example shows how the `breaks` argument can be used to specify where the "bins" are in the histogram. Here we used the function `seq()` to create a sequence with lower and upper bounds and step size (0.5, 14, and 1.5 in our example). Breaks can also be an arbitrary sequence - try `breaks=c(0,4,5,5.5,6,6.5,7,7.5,8.5,14)`) and see what happens!

Note:

About arguments: Notice that here the argument `probability=TRUE` has been abbreviated as `prob=T`. R is happy to accept *unambiguous* abbreviations for arguments. R is also happy to accept un-named arguments *in the order they are entered* - we did this in our call to `seq()` in the last example - we could have specified `seq(from=0.5,to=14.0,by=1.5)`. For the simpler functions that I use frequently I don't usually spell out the arguments, though here I will tend to spell them out more frequently.

Boxplots are another useful visualization of quantitative data which show the median, lower and upper "hinges" and the upper and lower whiskers. They can also be "notched" to show a confidence interval about the median. Values beyond the whiskers are possible outliers.

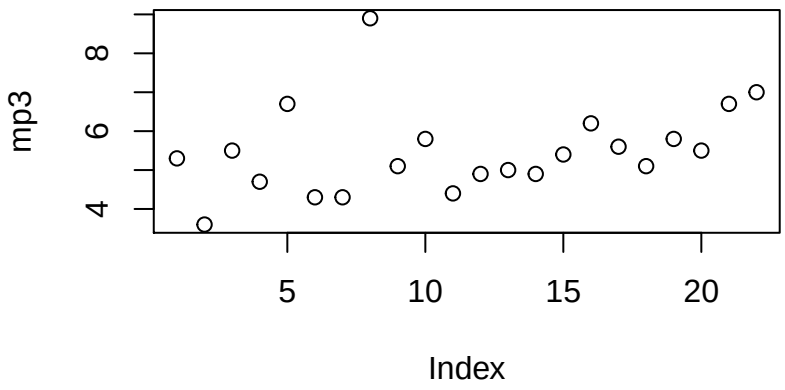
```
par(mfrow = c(1, 2))
boxplot(mp3)
boxplot(mp3, notch = TRUE, col = "cornsilk")
```



The value of `8.9` seems rather suspicious doesn't it?

We can visualize the "raw" data using `plot()`. Since `plot()` requires arguments for both x and y, but we are only providing x, the indices of x will be used for x, and the values of x for y.

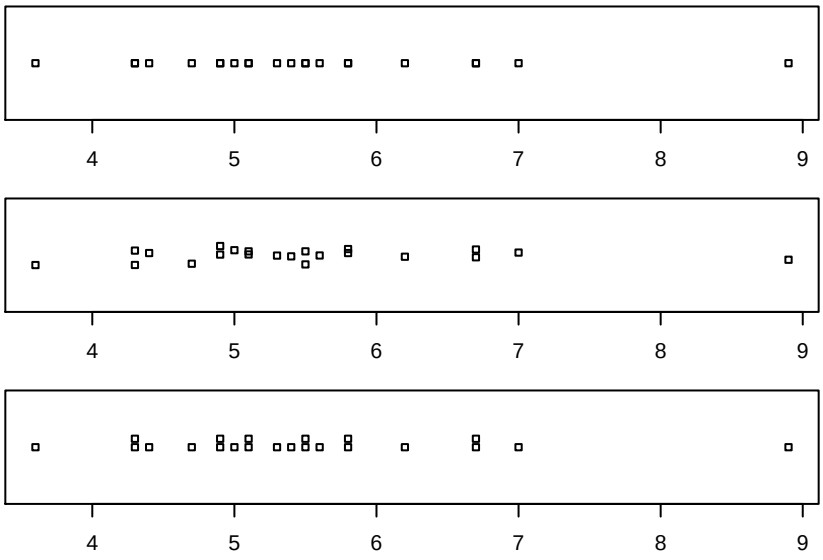
```
plot(mp3)
```



This method doesn't give us summary values like a boxplot does, but it has the advantage of letting us look for structure in the data (though it won't work for very large datasets). For example, here it is evident that the first half of the data set is more variable than the second half. Whether this is important or not depends on nature of the data.

Two other tools that are sometimes useful are the **stripchart** and the **stem and leaf** plot. The "stripchart" is a sort of "one-dimensional scatterplot". The argument `method` tells R how to display values that would over plot.

```
par(mfrow = c(3, 1))
stripchart(mp3)
stripchart(mp3, method = "jitter")
stripchart(mp3, method = "stack")
```



A final trick is the stem and leaf plot, which was originally developed because it could quickly be created with pencil and paper. While it looks simple and a bit crude, it has the advantages that it preserves the original data - from a stem and leaf plot you can reconstruct the actual values in the data, which you can't do with most of the other visualization tools we've looked at here.

```
stem(mp3)
```

```
#
```

```
#   The decimal point is at the |
#
#   2 | 6
#   4 | 3347990113455688
#   6 | 2770
#   8 | 9
```

```
stem(mp3, scale = 2)
```

```
#
#   The decimal point is at the |
#
#   3 | 6
#   4 | 334799
#   5 | 0113455688
#   6 | 277
#   7 | 0
#   8 | 9
```

The stem and leaf chart shows that the lowest value is 3.6, and occurs once, while the maximum value is 8.9, which may be an outlier. Stem and leaf plots are useful for exploratory data analysis (EDA), but I've rarely seen them published; histograms are much more commonly used.

V. Converting Quantitative Data to Qualitative

Sometimes we need to take a quantitative variable and "simplify" it by reducing it to categories. The function `cut()` can do this.

```
m.r <- cut(mp3, breaks = c(3:9))  # specify the breaks
m.r
```

```
#  [1] (5,6] (3,4] (5,6] (4,5] (6,7] (4,5] (4,5] (8,9] (5,6] (5,6] (4,5]
# [12] (4,5] (4,5] (4,5] (5,6] (6,7] (5,6] (5,6] (5,6] (5,6] (6,7] (6,7]
# Levels: (3,4] (4,5] (5,6] (6,7] (7,8] (8,9]
```

```
m.r[which(mp3 == 5)]  # values of 5.0 coded as (4,5]
```

```
# [1] (4,5]
# Levels: (3,4] (4,5] (5,6] (6,7] (7,8] (8,9]
```

Note non-matching brackets here: `(4,5]` - this means "greater than 4 and less than or equal to 5", so 4.0 is *not* included, but 5.0 *is* included in the interval `(4,5]`. We can demonstrate that this is so: `m.r[which(mid.rib==5.0)]` returns `(4,5]`.

We can now treat the factor `m.r` like any other factor variable, and assign other names to the levels as we see fit.

```
table(m.r)
```

```
# m.r
# (3,4] (4,5] (5,6] (6,7] (7,8] (8,9]
#     1     7     9     4     0     1
```

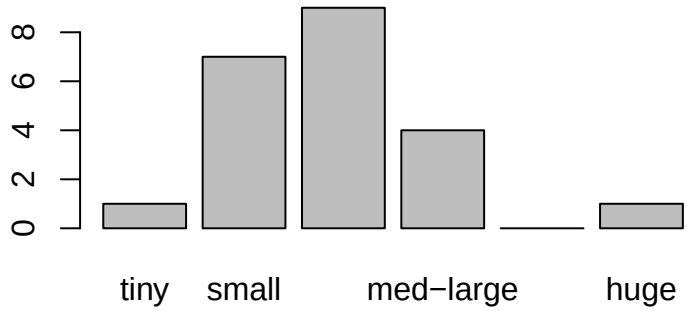
```
levels(m.r)
```

```
# [1] "(3,4]" "(4,5]" "(5,6]" "(6,7]" "(7,8]" "(8,9]"
```

```
levels(m.r) <- c("tiny", "small", "medium", "med-large", "large", "huge")
table(m.r)
```

```
# m.r
#      tiny     small    medium med-large     large      huge
#         1         7         9         4         0         1
```

```
plot(m.r)
```



Note that we could use any grouping we want to for `breaks`, just as we saw with `hist()`. Finally notice that in *this* case the function `plot()` creates a barplot. `plot()` is what is known as a *generic function*, meaning that what it does depends on the type of input it is given. For a qualitative variable it will return a barplot. As we progress, we'll see more kinds of output from `plot()`. The use of generic functions in R reduces the number of commands one needs to learn.

VI. Fitting and Modeling Distributions

t-distribution can be examined with a group of functions: `dx()`, `px()`, `qx()`, and `rx()` giving (respectively) the density, probabilities, quantiles, and random samples from the x distribution; arguments include parameters specific to the distributions. Most common distributions are included, see `?Distributions` for a full listing.

We'll demonstrate some of these functions for the exponential distribution. For example, what would be the probability of value of 3 or more from an exponential distribution with a rate parameter of 1?

```
pexp(q = 3, rate = 1, lower.tail = FALSE)
```

```
# [1] 0.04978707
```

The p-value is pretty close to 0.05, so about 1 of 20 random values from this distribution would be greater than 3. Let's generate 100 values and see how many are greater than or equal to 3.

```
x.exp <- rexp(n = 100, rate = 1)
sum(x.exp >= 3)
```

```
# [1] 3
```

Fairly close to 4.97, and a larger sample size would get even closer:

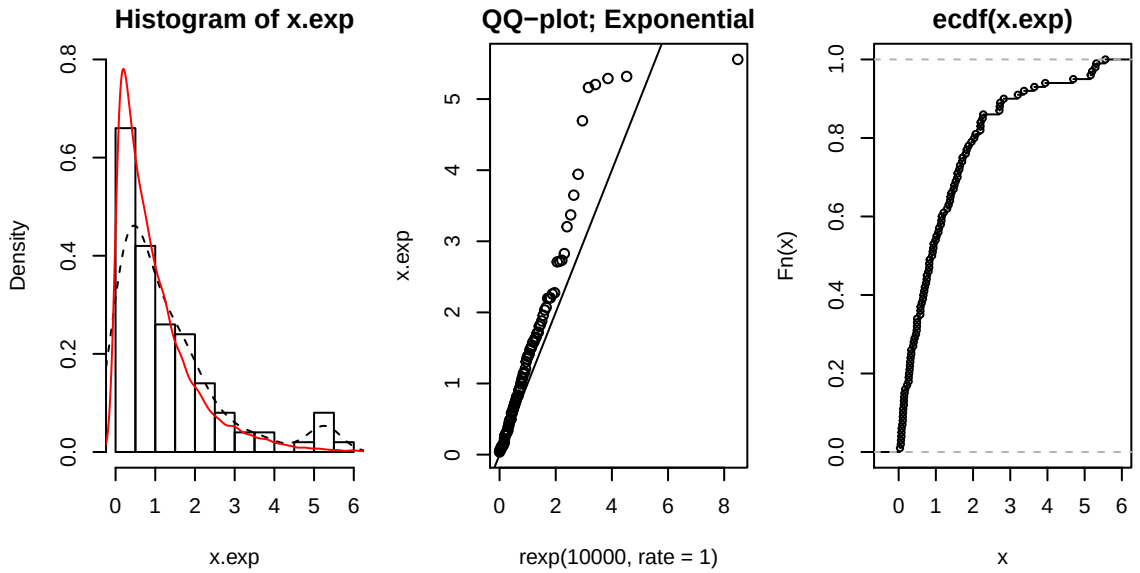
```
sum(rexp(n = 1e+05, rate = 1) >= 3)/1000
```

```
# [1] 4.929
```

Let's look at how we'd investigate the fit of a distribution. Imagine we have a sample of 100 values, and we think they come from an exponential distribution with a rate of 1.

```
x.exp <- rexp(n = 100, rate = 0.7)
hist(x.exp, prob = TRUE, ylim = c(0, 0.8))
lines(density(x.exp), lty = 2)
lines(density(rexp(10000, rate = 1)), col = "red")
qqplot(x = rexp(10000, rate = 1), y = x.exp, main = "QQ-plot; Exponential")
```

```
abline(a = 0, b = 1)
plot(ecdf(x.exp), pch = 21)
```



The first two plots here suggest that the distribution isn't what we hypothesize - rate=1 - (of course in this example we *know* the rate is not 1, our code that generates it shows the value is 0.7). For more ideas on modeling distributions in R see [Fitting Distributions with R] (http://cran.r-project.org/doc/contrib/Ricci-distributions-en.pdf)

VII. Exercises.

1) The R function `rnorm(n,mean,sd)` generates random numbers from a normal distribution. Use `rnorm(100)` to generate 100 values and make a histogram. Repeat two or three times. Are the histograms the same?

2) Make a histogram from the following data, and add a density estimate line to it. (use `scan()` to enter the data). Try changing the bandwidth parameter for the density estimate (use the argument "adj" for density( ); 1 is the default, 2 means double the bandwidth). How does this change the density estimate?
26 30 54 25 70 52 51 26 67 18 21 29 17 12 18 35 30 36 36 21 24 18 10 43 28 15 26 27 Note: If you are submitting this HW for class in a .txt file, you needn't include the plot, just include a brief description of how changing the bandwidth parameter for `density()` alters the shape of the curve. Note that the bandwith argument here can be a string that matches on of several methods for bandwith calculation or a numeric value for the bandwidth.

3) Using the data above compare the mean and median and the standard deviation, IQR and the MAD.

4) Use the function `boxplot()` to find possible outliers in the data from problem 2 (outliers shown as individual points; you don't need to show the boxplot, but you need to show the values that are possible outliers). Compare the mean and median of the data with the outliers removed. Compare the standard deviation and MAD for the data with the outliers removed.

Chapter 4: Documenting Your Work

How not to forget what you were doing

I. Introduction

One of the big challenges in using any analytical software tool is how to keep track of what you are doing, or how to document your work. In the real world, where the time needed to complete complex (or even supposedly "simple") projects often comes in short chunks with many interruptions, what this really means is not just documenting your work, but documenting your *thought process about your work*.

Before I used R I did all my data manipulation, etc in spreadsheets, and dumped data into stats software only for things like fitting models. I learned to do a wide range of things in spreadsheets (though often in rather kludgy ways!). But I often had the experience of coming back to an old (in months or even days) spreadsheet and being rather bewildered because I couldn't figure out what I had been doing with it. [12]

When I started using R, this habit carried over - I just pulled data into R for complicated stuff, and did everything else in spreadsheets. I copied and pasted my R code and output and figures into a word processor document to keep running notes on what I was doing.

As I discovered (and you may also know) this pattern leads to a *lot* of repeated copy/paste activity - for example when you discover that you have some mis-entered data and have to rerun the analysis, you have to repeat the copy/paste for all output and graphs.

To fully document statistical analysis, we need to have 4 components:
1. the exact method used to do the analysis (the R code)
2. the results of the analysis (statistical tables, etc)
3. any figures needed to illustrate the analysis
4. the appropriate commentary and narrative

There have been various tools that have attempted to simplify this process. Here we'll introduce the function `spin()` in the package `knitr` - it does everything on this list, (and is perfect for homework assignments!). In Chapter 14 we'll explore some more advanced tools.

II. Using knitr

The `knitr` package [13] provides tools to generate reports from R scripts. In R studio if you go to `File>New File>R script` you will open a .R file in your editor (really this is just a .txt file with .R as the file extension, so you could edit it with any text editor).

Conceptually how this works is you write comments and code in a .R file. Then you run the `spin()` function on the .R file. R runs the code, and puts together the comments, the code, and (this is the good part) the *output* and *figures* from the code in a single file.

If you have never used LaTeX or a similar writing tool or written html, this may seem odd to you. When using most word processors, you see what the output will look like "in real time". Compiled documents (like those we'll explore here) contain the *instructions* for making the final document.

III. An Example

Open an R script file. Copy the following into it:

```
#' A test of knitr
#' Some important text describing what this data is about
```

[12] The difficulty in annotating and narrating the logic behind spreadsheets is probably their greatest shortcoming.
[13] The package `knitr` was created by Yihui Xie while he was a PhD student in statistics.

```
# This is a comment
a<-1:25
b<-sort(rnorm(25)) # by default mean=0 and sd=1
mean(b)
sd(b)
#' Note mean and sd match the default values of sd=1 and mean=0
plot(a,b,type="l",col="red",lwd=2)
#' figures are included
```

The package `knitr` is nicely integrated into RStudio, making it very easy to use. We can compile a notebook from this R script file very easily. First we should save it. Let's call it "Ch4Demo1.R". You might as well save it in "Code Files" folder in your "Essential R" folder.

Note that the .R file contains 2 kinds of content only - R code and comments (preceded by `#`). If you look closely you will see that there are also two sub-types of comments - those that begin with `"#"` and those that begin with `"#'"`. Comments that begin with `#'` will be end up formatted as text, not as code comments.

Here is where **RStudio makes this super easy** - the "compile notebook" button appears at the top of the editor pane when a .R file is active:

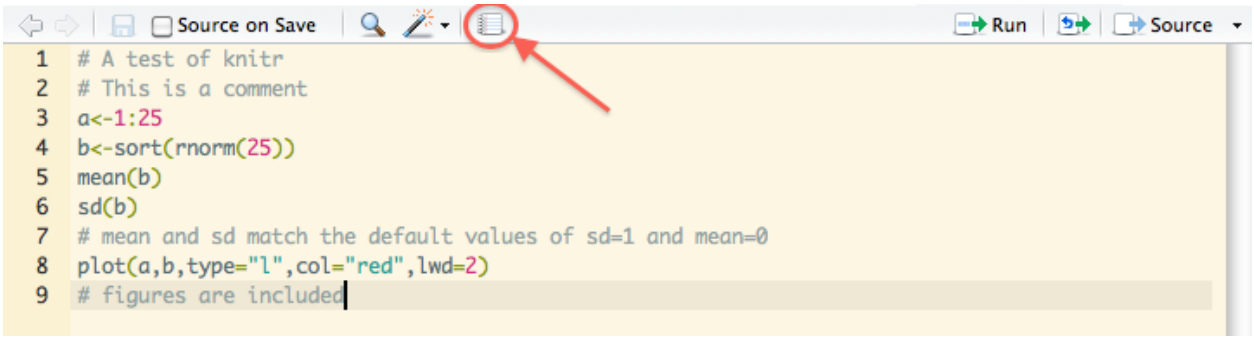


Figure 1: NotebookButton

When you click this button, it will ask you what output format you want. You can choose "Html", "MSWord", and "Pdf" (to use pdf you will need to install LaTex on your computer, check which version is appropriate for your OS). I suggest using this method rather than the console method (see below).

If you click the button shown above and get a message that you need to install `knitr`, it should be no problem [14]. You can go to the "Packages" tab in the upper right pane in RStudio and click "Install" and the type "knitr". Alternately just type (in the console) `install.packages ("knitr")`[15]. If this is the first time you have installed packages, you'll be prompted to select a mirror - just choose one nearer your location. R should work for a minute and you should get a preview of your html file. Easy!

This can all be done from the console also if you need to (but honestly, I mostly just use the button in RStudio!). Assuming the file is saved as "Ch4Demo1.R" in the "Code Files" folder in your "Essential R" folder, make sure that the "Code Files" folder is set as your *working directory*. Now we can load `knitr` and compile the document.

```
library(knitr)  # load the package
spin("Ch4Demo1.R")  # compile your document to Rmd
spin("Ch4Demo1.R", format = "Rhtml")  # compile your document to html
```

[14] As long as you have an internet connection. If you don't have an internet connection, it is harder. You have to find some way to get the source version of the package file onto your machine, and you can then install and compile the package from source as follows: `install.packages(repos=NULL,type="source",pkgs=file.choose())`, and point your file browser to the package file. If the file has a .tar.gz extension, use `type="source"`, if the extension is .zip use `type="win.binary"`.

[15] You should only have to install a package once on a given machine - this downloads it and copies it into the R library folder. Packages need to be loaded in every R session where they need to be used. When `knitr` is used via the RStudio button, you don't need to load it - that happens automagically.

If you didn't want to do this in 2 steps, you could simplify it:

```
knit(spin("Ch4Demo1.R"), output = "Ch4Demo1.html")
```

If you don't specify the `output=` argument here you will still get an .html file but the extension will be .txt.

NOTE: *If you choose to call* ***spin()*** *from the console,* **you probably don't want to include the call to spin() in you .R file** If it is in your .R file when you call `spin()` on the file it will recursively call `spin()` again, which you probably don't want. In my experience it is much simpler to use the "compile notebook" button, but I include the command line so you aren't completely reliant on RStudio. (If you prefer to call knitr from the console and don't want to have to remember the command, you can include it in your .R file but comment it out so it won't run recursively).

IV. Help! I'm Getting an Error!

The most common source of errors using `knitr` are actually not errors in knitr, but errors in the .R file being compiled. However, the error messages generated from compiling a file may be more cryptic. Here we'll demonstrate a common problem - a missing variable. In your "Ch4Demo1.R" file, comment out the line `a<-1:25` (put "#" in front of it). Now let's remove `a` in the workspace by typing (in the console):

```
rm(a)
```

Now if you run the code in "Ch4Demo1.R" line by line, you will get an obvious error when you get to `plot(a,b)` - there is no `a`, and R complains:

```
Error in plot(a, b, type = "l", col = "red", lwd = 2) :
  object 'a' not found
```

As you might expect, you will get an error when you try to compile the file with the line `a<-1:25` commented out. However, this error manifests itself differently depending on how you are using knitr. If you run the command from the console to knit the file, you don't get any warning errors, but the .html file that is created (in your working directory) has an `object 'a' not found` error, that you wouldn't notice if you didn't check it over, since when run from the console you don't get an automatic preview of the file.

However, when you knit the file by clicking the "notebook" icon in RStudio, the error message you get in the "R Markdown" console is a bit different:

```
Line 3 Error in plot(a,b,type="l",col="red",lwd=2): object 'a' not found Calls: <Anonymous>
... withCallingHandlers -> withVisisble -> eval -> eval -> plot Execution halted
```

The confusing bit here is that the error message points to Line 3, while the `plot()` command is in line 8 of the .R file - the line numbers referred to in these error messages don't seem to match the line numbers in the file being compiled.

On occasion it may be helpful to examine the intermediate (.md) file that is created. This is a file made by `spin()` and the error is generated when that file is compiled by the function `knit()`. When we compile by using the "Compile Notebook" button in RStudio this file is hidden somewhere, but we can see it if we re-run the command from the console:

```
spin("Ch4Demo1.R")
```

When we open the file "Ch4Demo1.md" in a text editor we can see the error "object 'a' not found", but we don't see why the line number was listed as line 3.

This is an extended explanation, but important because: *Missing objects are (in my experience) the* ***most common*** *cause of errors with knitr.*

Missing objects can cause errors when compiling your document even if the code worked in the console because when knitr evaluates the code to compile a document *it does so in a separate workspace*, not in your

workspace. You can confirm this by clearing your workspace (The broom icon in the environment pane in RStudio, or `rm(list=ls())`) and compiling the "Ch4Demo1.R" file (uncomment the `a<-1:25` line). Now look at your workspace in the environment pane or type `ls()` in the console. Notice that neither `a` nor `b` exist in your workspace, yet your document compiled without trouble.

This is not a bug, it is a feature. The intent of `knitr` is to enable "reproducible research". This means that *everything* required for the analysis should be in the script, nothing should happen that isn't documented. This ends up creating errors when you write code in the console and don't include it in your script. The solution is simple - *write all code in the editor* and test it in the console (recall control + enter will run the current line in the editor).

A final note: Plots are included in the .html file only when the plotting window is complete. So if the plotting window is split (e.g using `par(mfrow=c(1,2))`) the plot won't be inserted until both panes have been plotted, even if the second plot isn't called until many lines later. The file "Notebook and par.R" in the code files directory of EssentialR can be modified and compiled to see how this works.

VI. Exercises.

1) Take your code from the Ch2 and Ch3 HW and paste it into a single .R file. Test it to make sure it runs. Add comments where needed in the file to discuss/describe the results. Run the code from the console to check it (select all and Ctrl+enter). Now compile the file. (*Hint:* Before testing the code clear your workspace (Session>Clear Workspace) to be sure the code doesn't rely on anything not in your workspace.)

Chapter 5: Help in R

What to do when you need help.

I. Introduction.

When using R, one needs help *often*. Unfortunately, many users find R's extensive built in help files to be confusing, especially when you aren't really sure which function you need to use, or when you are a new R user.

Actually, when I was a new R user I probably would have said the help files were "nearly useless", because I didn't understand them. In this chapter I hope to give you a handle on how to make use of them.

II. R's Help System

For any function you can type `?` followed by the function name (e.g `?mean`). You can also search for similar functions using `??` - for example `??mean` will return a list of functions that include the word 'mean' in the name of short description[16].

In RStudio you can also search the help functions from the "search" field in the "Help" tab. Finally, RStudio has very good "function hinting" to give short reminders of function arguments. In the RStudio editor or console type `mean()` and with the cursor inside the `()`, press "tab". A window will pop up showing you arguments for the function. You can move through the list with the mouse or arrow keys, and use "tab" to select one from the list.

All R help files include several parts: a *Description* of the function, a typical *Usage* of the function, and a list of *Arguments* and their brief descriptions. Sometimes a section called *Details* provides additional details about the arguments. Usually there is a section called *Value* where the value returned by the function is explained. There are often also *References* and a list of similar or related functions under *See Also*. Finally, (almost) all help files include *Examples* of how the function can be used.

The *Examples* section is often one of the more useful features of the R help system. Example code is meant to "stand alone" - any data needed for the example code should be provided - so that you can copy and paste the code into your console and run it, and play with arguments to help you figure out how it works. In my experience this is one of the best ways to learn about a function.

On occasion the function you are looking for help on actually calls

Note that help file examples generally make very few assumptions about the state of your R workspace. For this reason they may include lines like `require(grDevices)` which loads the `grDevices` package, or `stats::rnomr()`, which specifies that rnorm is in the `stats` package. Since both of these packages are loaded by default on startup, these details wouldn't be necessary unless you had removed them, but the examples don't *assume* that they will be there.

III. Help! I Don't Know Which Function to Use.

R's help system is most useful when you know what function you need (or if you can at least guess a function that might be linked to the one you need). The more frustrating quandary is *when you don't even know the function you should use*. This is even worse when you have an idea in your head for what you want, but aren't even sure what the correct words for it are.

At times like this I use [RSeek] (http://www.rseek.org), which is an R specific search engine. This gets around the problem of searching for 'R + question', since R is just a letter. RSeek helpfully organizes search results into categories such as "Articles", "For Beginners", "Support", etc (Note that these tabs don't appear until *after* you have begun a search).

[16] As we'll see later, searching online may be more profitable until you have a function name.

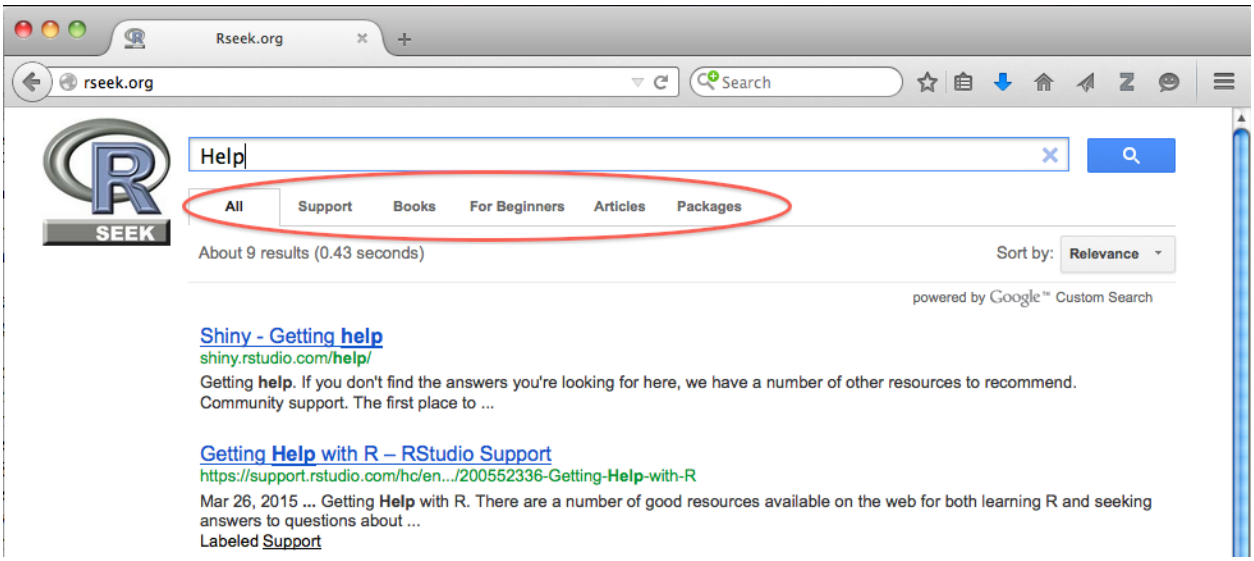


Figure 2: Rseek

When you don't know what function to use, the "Support" tab is your friend. I usually find that if I describe what I am trying to do and look under the "Support" tab, someone else has asked the same or similar question, and I can get an answer.

Another useful resource is the R archive "CRAN" (https://cran.r-project.org). The "Task Views" tab on the left points to overviews of analytical areas. These are often useful if you are trying to figure out a big question like "How do I fit mixed effects models"?

The "Documentation" tab on the CRAN site also contains many official manuals and contributed documents (in a wide array of languages). You may find what you are looking for here, but it would not be the first place I would look.

IV. I Need More Help

You can also create an account and post questions to the "R help mailing list" (see the "Mailing Lists" link at www.r-project.org/mail.html). However, don't post a question until you are sure it hasn't been answered already [17] - in other words "search first, search second, ask third". Also, be sure to follow the [posting guidelines] (http://www.r-project.org/posting-guide.html) - they may vary from site to site, the link is for the R help forums.

Some thoughts about posting to the R mailing lists (and this applies in general to all online help questions about R). The main idea here is to make it easy for someone to help you.
1. Remember no one gets paid to answer your questions - questions are answered by other R users (occasionally even by people who wrote parts of R). The R user community is generous with its time and expertise - don't abuse that. My mantra is "search first, search second, ask third".
2. State your question as clearly as possible.
3. Make sure you include a *reproducible example* in your post.
4. If you are getting error messages, include the error message and the code that generated it.
5. Allow time for responses.

Here is an example of how an example could be posted (from the posting guide, you'll need to read the posting guide for the answer).

[17] I have never needed to post a question - someone else has always asked my questions before me. This either means that there are a lot of R users asking questions or that my questions are not very original.

```
# If I have a matrix x as follows:
  x <- matrix(1:8, nrow=4, ncol=2,
              dimnames=list(c("A","B","C","D"), c("x","y"))
#  x
#    x y
#  A 1 5
#  B 2 6
#  C 3 7
#  D 4 8

# how can I turn it into a dataframe with 8 rows, and three
  columns named `row', `col', and `value', which have the
  dimension names as the values of `row' and `col', like this:
#    x.df
#     row col value
#  1    A   x      1
```

Notice that the example includes the code to make the data needed for the example. This makes it easy for someone to help you.

V. Understanding Errors

One of the skills that takes time to develop is understanding error messages. In some cases error messages can be a bit cryptic. In Essential R I will point out common errors along the way to assist in recognizing them.

As we've noted unbalanced parentheses will usually prompt a `+` instead of a prompt in the console. While this seems obvious, when nesting functions it is quite easy to misplace a ")".

VI. Exercises

To reinforce what you learned in Chapter 4, write up your answers and some of the example code in an R script file and compile it to html using `knitr`. Be sure to review the .html file for errors.

1) Practice running help examples with `?hist`, and `?plot` (in {graphics}): What can you learn about these functions?

2) Follow some links from `?hist`, and `?plot` and run some examples - what did you learn from any of these linked files?

3) Use www.RSeek.org to find a method to calculate the "outer product" of 2,4,6,8 and -2,-1,0,1,2,3.

Chapter 6: Bivariate Data

Basic apporaches to dealing with two variables.

I. Introduction

A great deal of statistical analysis is based on describing the relationship between two variables. For example - how does planting density (high, medium, or low) alter crop yield? How is home price related to lot size? How are height and foot size related? Is the incidence of heart disease different for men and women? Here we'll consider working with two qualitative variables, one qualitative and one quantitative variable, and two quantitative variables.

II. Two Qualitative Variables

We sometimes want to see how two qualitative (factor) variables are related. Here we'll work with some data for number of cylinders [18] (`cyl`) and transmission type (`am`) from 32 models of cars reported in *Motor Trend* magazine in 1974 [19].

```
cyl<-factor(scan(text=
    "6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4"))
am<-factor(scan(text=
    "1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1"))
levels(am)<-c("auto","manual")
table(cyl,am)
```

```
#    am
# cyl auto manual
#   4    3      8
#   6    4      3
#   8   12      2
```

```
table(am, cyl)
```

```
#         cyl
# am        4  6  8
#   auto    3  4 12
#   manual  8  3  2
```

It appears that manual transmissions were more common with smaller numbers of cylinders, while cars with 8 cylinders were far more likely to have automatic transmissions. Notice that our old friend `table()` can be used to give a two-way frequency table as well. Also note that as we discussed in Chapter 2, it is simpler to enter a long factor as level numbers and assign the levels later.

Sometimes we would rather see tables like this expressed as proportions. R can easily do this via the function `prop.table()`.

```
tab <- table(cyl, am)
prop.table(tab, margin = 1)
```

```
#    am
# cyl      auto    manual
```

[18] Number of engine cylinders is a nice example of a *numeric factor*. Not only are the values constrained to integer values, but there are only a few values that are common, although there have been a few 3 or 5 cylinder engines. Such a factor could be treated as "ordered", but that is beyond the scope of these notes. This variable might also be treated as numeric - the analyst woudl have to decide what makes the most sense here.

[19] This data is found in the `mtcars` data set that is included with R: as we'll see later you can access the whole data set using `data(mtcars)`.

```
#   4 0.2727273 0.7272727
#   6 0.5714286 0.4285714
#   8 0.8571429 0.1428571
```

The `margin=1` tells R that we want the proportions within *rows*. We can see that 85% of 8 cylinder cars have an automatic compared to 27% of four cylinder cars. We can also use `margin=2` to have proportions within *columns*. If we don't include the `margin` argument, the default it to proportions of the entire table.

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

```
#    am
# cyl      auto    manual
#   4 0.1578947 0.6153846
#   6 0.2105263 0.2307692
#   8 0.6315789 0.1538462
```

```
prop.table(tab)
```

```
#    am
# cyl    auto  manual
#   4 0.09375 0.25000
#   6 0.12500 0.09375
#   8 0.37500 0.06250
```

From the first we can see that 63% of cars with automatic transmissions had 8 cylinders. From the second we can see that 38% of cars had both automatic transmission and 8 cylinders.

Finally, note that in these proportion tables R is giving us a larger number of decimal places than we might really want. This can be controlled in several ways - the simplest is via the function `signif()`, which control the number of significant digits printed [20].

```
signif(prop.table(tab), 2)
```

```
#    am
# cyl  auto manual
#   4 0.094  0.250
#   6 0.120  0.094
#   8 0.380  0.062
```

We may want to test whether there is any association between categorical variables. The simplest approach is often to use the χ^2 (Chi^2) test with the null hypothesis that the variables are independent.

```
chisq.test(tab)
```

```
# Warning in chisq.test(tab): Chi-squared approximation may be incorrect
```

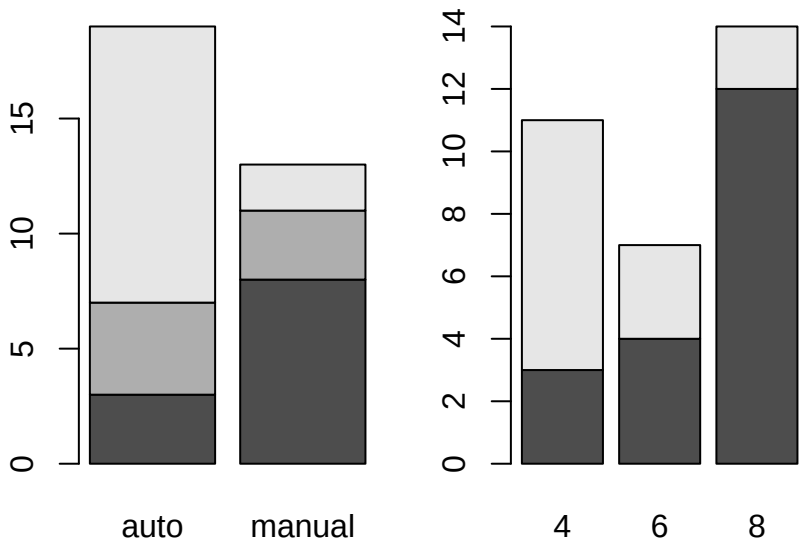
```
#
#   Pearson's Chi-squared test
#
# data:  tab
# X-squared = 8.7407, df = 2, p-value = 0.01265
```

In this case we have evidence of association between more cylinders and automatic transmissions, but we have a warning - this is because there are too few values in some of our groups - Chi^2 is not valid if any group has <5 members.

There are a couple of possible ways to visualize such data. One option is using a barplot.

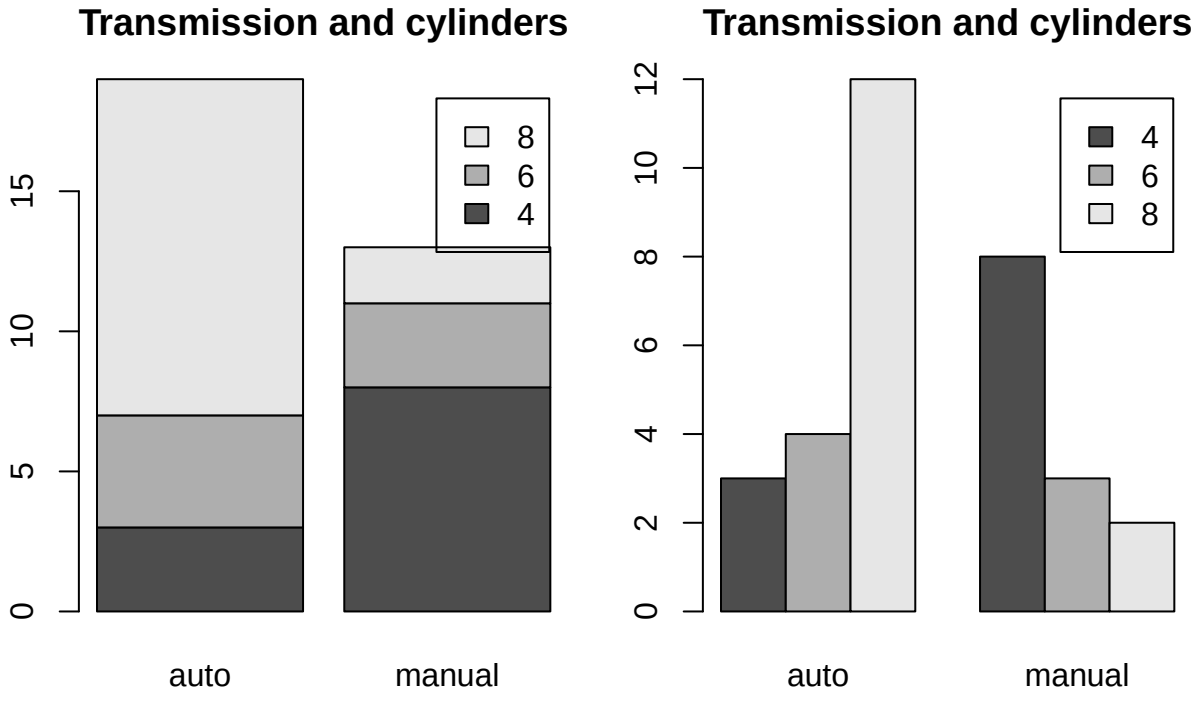
[20] You can also change this by changing R's options - see `?options`. The advantage of using `signif()` is that it is temporary and specific to the current command.

```
op = par(mfrow = c(1, 2))
barplot(table(cyl, am))
barplot(table(am, cyl))
```



Note: the function `par()` is used to set graphical *parameters* - in this case we're specifying that the plotting window will be divided into 1 row and 2 columns. We've simultaneously saved the old par settings as `op`. There are a few more options we can use to dress this up. The confusing thing here is that it is the *old* settings that are saved, not the new ones. Also note that the function `options()` we discussed above functions in a similar way with assignment of old options. It is possible that you may never need to set options - it just depends on how you use R.

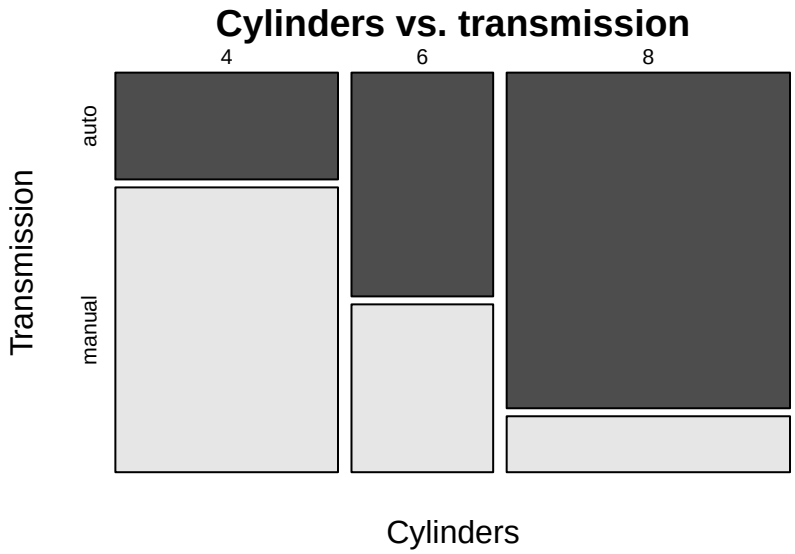
```
par(mfrow=c(1,2),mar=c(3,3,3,0.5))
barplot(table(cyl,am),legend.text=TRUE,main="Transmission and cylinders")
barplot(table(cyl,am),beside=TRUE,legend.text=TRUE,
        main="Transmission and cylinders")
```



```
par(op)
```

Here in addition to dividing the plotting window we've used `par()` to reduce the plot margins. The final line restores the old `par` settings we saved earlier. [21] Another option which is less familiar is the *mosaic plot*, which shows the proportions of each combination of factors.

```
mosaicplot(table(cyl, am), color = T, main = "Cylinders vs. transmission", ylab = "Transmission",
    xlab = "Cylinders")
```



Note that many of the arguments here are optional. You can try leaving them out to see what they do; the minimum is `mosaicplot(table(cyl,am))`.

III. Two Quantitative Variables

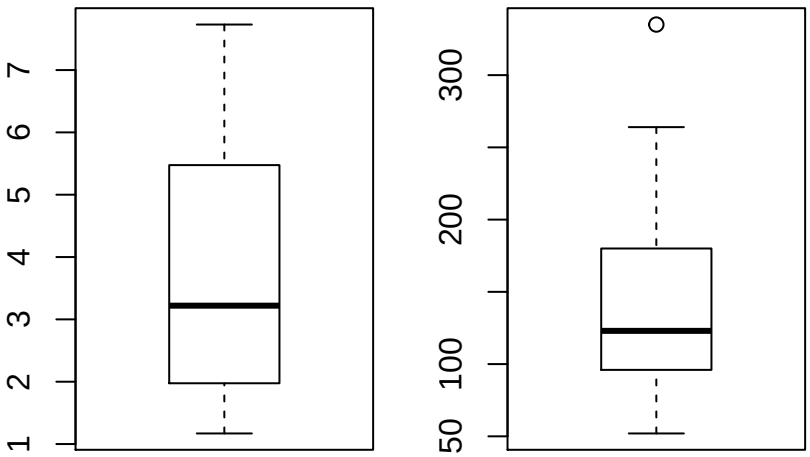
We frequently find that we are looking for association between two quantitative variables. For example, using the motor trend cars data we might wish to look at the association between engine displacement (here in liters) and power output (horsepower).

```
disp=scan(text=
    "2.62 2.62 1.77 4.23 5.90 3.69 5.90 2.40 2.31 2.75 2.75 4.52 4.52 4.52
    7.73 7.54 7.21 1.29 1.24 1.17 1.97 5.21 4.98 5.74 6.55 1.29 1.97 1.56
    5.75 2.38 4.93 1.98")
hp=scan(text=
    "110 110 93 110 175 105 245 62 95 123 123 180 180 180 205 215 230
    66 52 65 97 150 150 245 175 66 91 113 264 175 335 109")
```

Exploring the data

```
op = par(mfrow = c(1, 2))
boxplot(disp)
boxplot(hp)
```

[21] Restoring the old `par` settings is sometimes important - once we split the plotting window it stays split, and we might not want it to.




```
par(op)
```

Both variables show a bit of skew, with a larger number of low values. The plot of horsepower shows one possible outlier. We can find which it is using logical extraction:

```
data(mtcars)  # load the whole data set
mtcars[which(mtcars$hp > 250), ]
```

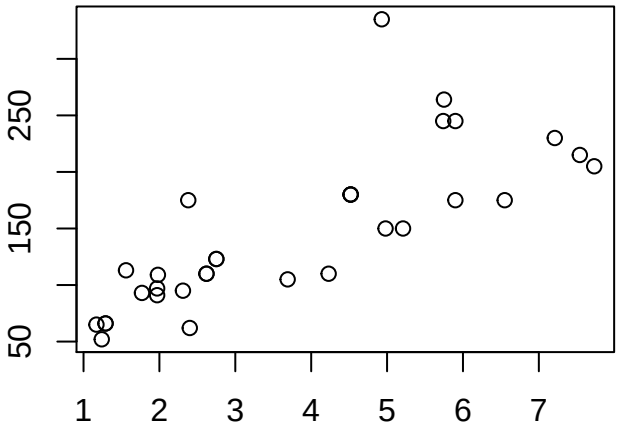
```
#                mpg cyl disp  hp drat   wt qsec vs am gear carb
# Ford Pantera L 15.8   8  351 264 4.22 3.17 14.5  0  1    5    4
# Maserati Bora  15.0   8  301 335 3.54 3.57 14.6  0  1    5    8
```

This shows only 2 cars with horsepower greater than 250. Notice that here we used the function `data()` to load one of the built-in data sets, and that we used the `$` to specify a variable within a dataset. We'll discuss this in more detail soon. Also notice that within the `[]` we have a comma - the format is `[rownumber,columnnumber]`, and we want the rows with `hp>250`.

Correlation

We might guess that there is some correlation between displacement and power. A simple scatter plot will confirm this:

```
plot(x = disp, y = hp)
```




```
cor(disp, hp)
```

```
# [1] 0.7910567
```

Notice that `plot()` here gives us a scatter plot [22]. The correlation coefficient r is reasonably high at 0.7910567.

By default `cor()` gives us the *pearson* correlation. By setting the `method` argument to `method="spearman"` we can get the *spearman rank* correlation (which is more robust to outliers). It should be noted that the `cor()` function needs to be told how to deal with missing values (`NA`) - this is done via the argument `use`, which tells R which values to use. A setting of `use="complete.obs"` will often work (see `?cor` for more information).

Regression

Often we want to go a step further and *model* one variable *as a function* of another. With two quantitative variables this is known as linear regression (regression for short). In this case, we might well suspect that larger displacement engines should be more powerful. In R such models are fit using `lm()` (for "linear model"):

```
model = lm(hp ~ disp)
model
```

```
#
# Call:
# lm(formula = hp ~ disp)
#
# Coefficients:
# (Intercept)         disp
#       45.69        26.71
```

```
summary(model)
```

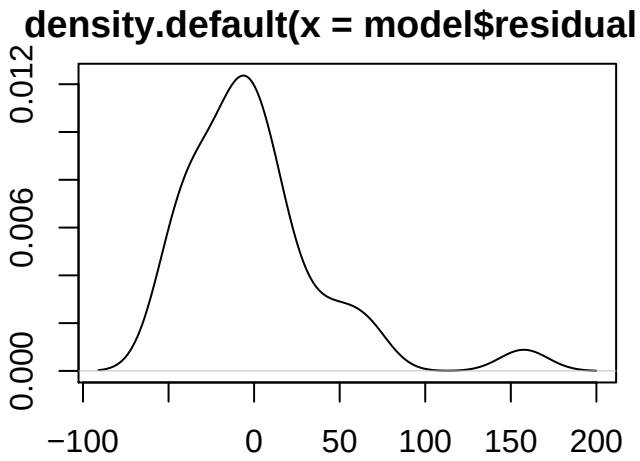
```
#
# Call:
# lm(formula = hp ~ disp)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -48.682 -28.396  -6.497  13.571 157.620
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   45.695     16.128   2.833  0.00816 **
# disp          26.711      3.771   7.083 7.09e-08 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 42.64 on 30 degrees of freedom
# Multiple R-squared:  0.6258,  Adjusted R-squared:  0.6133
# F-statistic: 50.16 on 1 and 30 DF,  p-value: 7.093e-08
```

Notice a couple of things here:
1. in our call to `lm()` we specified `hp~disp` - this means hp *as a function of* disp. This type of model notation is used by a number of functions in R.
2. `lm(hp~disp)` returns only the intercept and slope for the model.
3. `lm()` has actually done *much more* - it has created an *"lm object"* that we have named `model`. Type `names(model)` to see what all is there - you can access all of these - for example `model$residuals` will return the residuals from the model. 4. The function `summary()` when called on an lm object, gives a very helpful summary of the regression. This shows that our model is highly significant, with p-value = 7.093 x 10^-8 .

[22] We could omit the names of the `x` and `y` arguments, the first will be taken as `x` and the second as `y`. Also `plot(hp~disp)` would work.

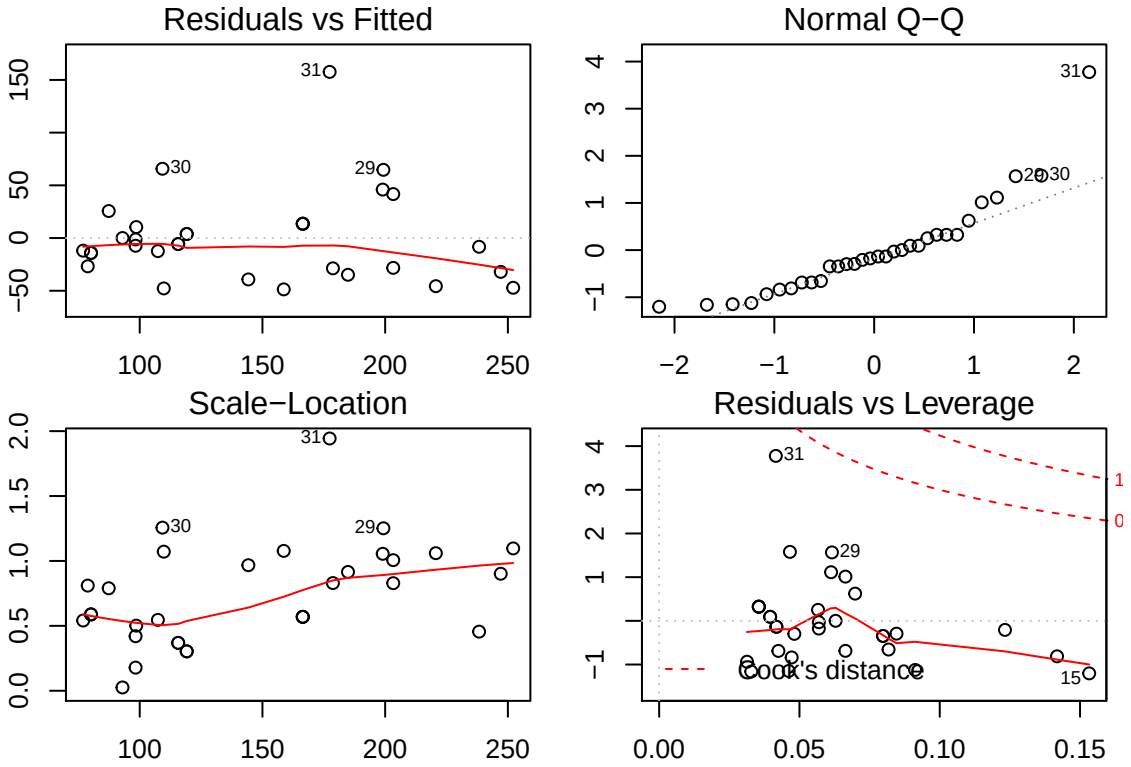
If you recall one of the assumptions in regression is that the residuals are normally distributed. We can check to see if this is true:

```
plot(density(model$residuals))
```



Overall, the residuals are not really normally distributed, but they are probably normal enough for the regression to be valid. Of course, checking model assumptions is a common (and *necessary*) task, so R makes it easy to do.

```
op = par(mfrow = c(2, 2))
plot(model)
```



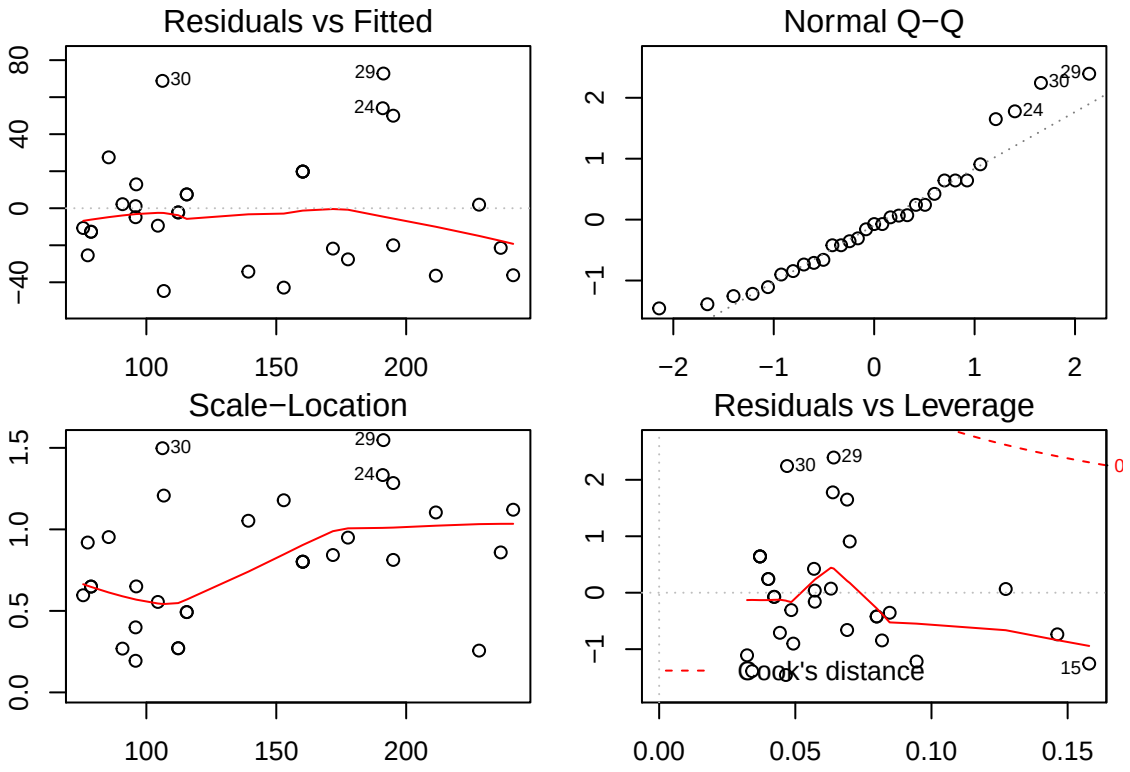
The normal Q-Q plot does show that we may have one outlier, point 31 (The Maserati Bora). We could refit the model without it to see if it fits better.

```
op = par(mfrow = c(2, 2))
model2 <- lm(hp[-31] ~ disp[-31])
```

```
summary(model2)
```

```
#
# Call:
# lm(formula = hp[-31] ~ disp[-31])
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -44.704 -21.601  -2.255  16.349  72.767
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   46.146     11.883   3.883 0.000548 ***
# disp[-31]     25.232      2.793   9.033 6.29e-10 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 31.41 on 29 degrees of freedom
# Multiple R-squared:  0.7378,  Adjusted R-squared:  0.7287
# F-statistic:  81.6 on 1 and 29 DF,  p-value: 6.291e-10
```

```
plot(model2)
```



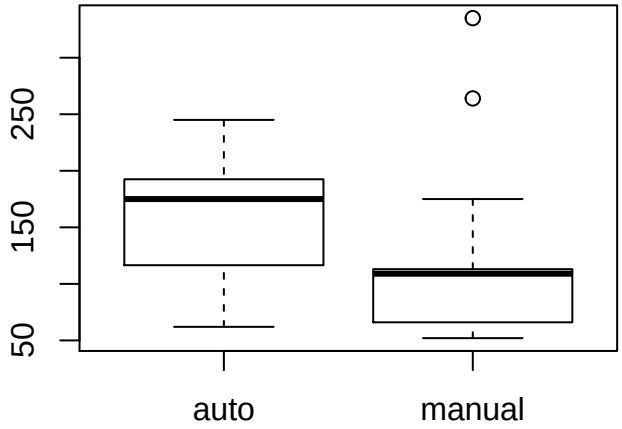

```
par(op)
```

Removing the outlier really improves the model fit - the R^2 increases to 0.729, and the residuals look much more normal (the Q-Q plot is more linear).

V. Qualitative and Quantitative Variables

When we have a quantitative and a qualitative variable, we can use similar tools to what we would use for two quantitative variables. Consider the data on cars - do we expect a difference in horsepower between cars with automatic and manual transmissions?

```
plot(am, hp)
```



It appears that more cars with automatic transmissions are generally more powerful, though the two most powerful cars have manual transmissions - we saw these earlier. We can use a two-sample t-test to see if these groups are different.

```
t.test(hp ~ am)
```

```
#
#   Welch Two Sample t-test
#
# data:  hp by am
# t = 1.2662, df = 18.715, p-value = 0.221
# alternative hypothesis: true difference in means is not equal to 0
# 95 percent confidence interval:
#  -21.87858  88.71259
# sample estimates:
#   mean in group auto mean in group manual
#             160.2632             126.8462
```

This show that the means are not different - likely the influence of the two "super-cars" with manual transmissions pulls the mean up enough to mask the difference.

ANOVA

Note that if we had more than two groups, we'd need a different approach - we can use `oneway.test()` to do a simple ANOVA. For two groups this is equivalent to the t-test, but it will work for more than two groups also. > Since R uses the function `lm()` for both regression and ANOVA, you may find it helpful to think about ANOVA as a kind of regression, where the predictor variable (x-axis) is *categorical*.
> NOTE: `lm()` and `oneway.test()` will return errors if you use a factor as the response variable, so recall that "~" should be read as "as a function of", so that `cyl~hp` is "cylinders (factor in our case) ~ horsepower" would not work here.

```
oneway.test(hp ~ am)
```

```
#
#   One-way analysis of means (not assuming equal variances)
```

```
#
# data:  hp and am
# F = 1.6032, num df = 1.000, denom df = 18.715, p-value = 0.221
```

```
oneway.test(hp ~ cyl)
```

```
#
#   One-way analysis of means (not assuming equal variances)
#
# data:  hp and cyl
# F = 35.381, num df = 2.000, denom df = 16.514, p-value = 1.071e-06
```

```
summary(lm(hp ~ cyl))
```

```
#
# Call:
# lm(formula = hp ~ cyl)
#
# Residuals:
#    Min     1Q Median     3Q    Max
# -59.21 -22.78  -8.25  15.97 125.79
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)    82.64      11.43   7.228 5.86e-08 ***
# cyl6           39.65      18.33   2.163   0.0389 *
# cyl8          126.58      15.28   8.285 3.92e-09 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 37.92 on 29 degrees of freedom
# Multiple R-squared:  0.7139,  Adjusted R-squared:  0.6941
# F-statistic: 36.18 on 2 and 29 DF,  p-value: 1.319e-08
```

We'll dig into ANOVA in more depth in a later chapter.

VI. Exercises

1) Using the data `cyl` and `am` (transmission type) from Part II, group vehicles based into 8 cylinder and less than 8 cyl. Test whether there is evidence of association between many cylinders and automatic transmissions. (*Hint:* use `levels()` to re-level `cyl` and then use `chisq.test()`).

2) The built in dataset `faithful` records the time between eruptions and the length of the prior eruption for 272 inter-eruption intervals (load the data with `data(faithful)`). Examine the distribution of each of these variables with `stem()` or `hist()`. Plot these variables against each other with the length of each eruption (`eruptions`) on the x axis. How would you describe the relationship? Recall that you can use `faithful$eruptions` to access `eruptions` within `faithful`.

3) Fit a regression of `waiting` as a function of `eruptions`. What can we say about this regression? Compare the distribution of the residuals (`model$resid` where `model` is your lm object) to the distribution of the variables.

4) Is this data well suited to regression? Create a categorical variable from `eruptions` to separate long eruptions from short eruptions (2 groups) and fit a model (ANOVA) of `waiting` based on this. (*Hint:* use `cut()` to make the categorical variable, and `lm()` to fit the model). How does this model compare with that from Ex 3? How did you choose the point at which to cut the data? How might changing the cut-point change the results?

Chapter 7: The Data Frame

The R equivalent of the spreadsheet.

I. Introduction

Most analytical work involves importing data from outside of R and carrying out various manipulations, tests, and visualizations. In order to complete these tasks, we need to understand how data is stored in R and how it can be accessed. Once we have a grasp of this we can consider how it can be imported (see Chapter 8).

II. Data Frames

We've already seen how R can store various kinds of data in vectors. But what happens if we have a mix of numeric and character values? One option is a *list*

```
a <- list(c(1, 2, 3), "Blue", factor(c("A", "B", "A", "B", "B")))
a
```

```
# [[1]]
# [1] 1 2 3
#
# [[2]]
# [1] "Blue"
#
# [[3]]
# [1] A B A B B
# Levels: A B
```

Notice the `[[]]` here - this is the *list element* operator. A list in R can contain an arbitrary number of items (which can be vectors) which can be of different forms - here we have one numeric, one character, an one factor, and they are all of different lengths.

A list like this may not be something you are likely to want to use often, but in most of the work you will do in R, you will be working with data that is stored as a *data frame* - this is R's most common data structure. A data frame is a special type of list - it is a list of vectors that have the same length, and whose elements correspond to one another - i.e. the 4th element of each vector correspond. Think of it like a small table in a spreadsheet, with the columns corresponding to each vector, and the rows to each record.

There are several different ways to interact with data frames in R. "Built in" data sets are stored as data frames and can be loaded with the function `data()`. External data can be read into data frames with the function `read.table()` and its relative (as we'll see in the next chapter). Existing data can be converted into a data frame using the function `data.frame()`.

```
cyl<-factor(scan(text=
    "6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4"))
am<-factor(scan(text=
    "1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1"))
levels(am)<-c("auto","manual")
disp<-scan(text=
    "2.62 2.62 1.77 4.23 5.90 3.69 5.90 2.40 2.31 2.75 2.75 4.52 4.52 4.52
    7.73 7.54 7.21 1.29 1.24 1.17 1.97 5.21 4.98 5.74 6.55 1.29 1.97 1.56
    5.75 2.38 4.93 1.98")
hp<-scan(text=
    "110 110  93 110 175 105 245 62 95 123 123 180 180 180 205 215 230 66
    52 65 97 150 150 245 175 66 91 113 264 175 335 109")
```

Here we've re-created the data on cars that we used in the last chapter.

```
car <- data.frame(cyl, disp, hp, am)
head(car)
```

```
#   cyl disp  hp     am
# 1   6 2.62 110 manual
# 2   6 2.62 110 manual
# 3   4 1.77  93 manual
# 4   6 4.23 110   auto
# 5   8 5.90 175   auto
# 6   6 3.69 105   auto
```

```
summary(car)
```

```
#  cyl         disp             hp             am
#  4:11   Min.   :1.170   Min.   : 52.0   auto  :19
#  6: 7   1st Qu.:1.978   1st Qu.: 96.5   manual:13
#  8:14   Median :3.220   Median :123.0
#         Mean   :3.781   Mean   :146.7
#         3rd Qu.:5.343   3rd Qu.:180.0
#         Max.   :7.730   Max.   :335.0
```

Now we've created a data frame named `car`. The function `head()` shows us the first 6 rows (by default). Here we see that `summary()`, when called on a data frame, gives the appropriate type of summary for each variable. The variables within the data frame have names, and we can use the function `names()` to retrieve or change these.

```
names(car)
```

```
# [1] "cyl"  "disp" "hp"   "am"
```

```
names(car)[4] <- "trans"
names(car)
```

```
# [1] "cyl"   "disp"  "hp"    "trans"
```

```
car$am
```

```
# NULL
```

```
car$trans
```

```
#  [1] manual manual manual auto   auto   auto   auto   auto   auto   auto
# [11] auto   auto   auto   auto   auto   auto   auto   manual manual manual
# [21] auto   auto   auto   auto   auto   manual manual manual manual manual
# [31] manual manual
# Levels: auto manual
```

Data in data frames can be accessed in several ways. We can use the indexing operator `[]` to access parts of a data frame by rows and columns. We can also call variables in a data frame by name using the `$` operator.

```
car[1:3, ]
```

```
#   cyl disp  hp  trans
# 1   6 2.62 110 manual
# 2   6 2.62 110 manual
# 3   4 1.77  93 manual
```

```
car[, 3]
```

```
#  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
# [18]  66  52  65  97 150 150 245 175  66  91 113 264 175 335 109
```

```
car$hp
```

```
#  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
# [18]  66  52  65  97 150 150 245 175  66  91 113 264 175 335 109
```

Note that when indexing a data frame we use 2 indices, separated by a comma (e.g. `[2,3]`). Leaving one value blank implies "all rows" or "all columns". Here the first line gives us rows 1:3, the second and third both give us the `hp` variable.

Where we've created a new data frame in this way it is important to note that *R has copied the vectors that make up the data frame*. So now we have `hp` and `car$hp`. It is important to know this because if we change one, the other is not changed.

```
hp[1] == car$hp[1]
```

```
# [1] TRUE
```

```
hp[1] <- 112
hp[1] == car$hp[1]
```

```
# [1] FALSE
```

In a case like this, it might be a good idea to *remove* the vectors we used to make the data frame, just to reduce the possibility of confusion. We can do this using the function `rm()`.

```
ls()
```

```
#  [1] "a"      "am"     "b"      "car"    "cols"   "cyl"    "die"
#  [8] "disp"   "hp"     "i"      "m.r"    "model"  "model2" "mp3"
# [15] "mtcars" "op"     "sms"    "t"      "tab"    "x"      "x.exp"
# [22] "y"      "z"
```

```
rm(cyl, disp, hp, am)
ls()
```

```
#  [1] "a"      "b"      "car"    "cols"   "die"    "i"      "m.r"
#  [8] "model"  "model2" "mp3"    "mtcars" "op"     "sms"    "t"
# [15] "tab"    "x"      "x.exp"  "y"      "z"
```

Now these vectors are no longer present in our workspace.

It is useful to know that many R functions (`lm()` for one) will accept a `data` argument - so rather than `lm(car$hp~car$cyl)` we can use `lm(hp~cyl,data=car)`. When we specify more complex models, this is very useful. Another approach is to use the function `with()` - the basic syntax is `with(some-data-frame, do-something)` - e.g. `with(car,plot(cyl,hp))`.

Indexing Data Frames

Since our data `car` is a *2-dimensional* object, we ought to use 2 indices. Using the incorrect number of indices can either cause errors or cause unpleasant surprises. For example, `car[,4]` will return the 4th column, as will `car$am` or `car[[4]]`. However `car[4]` will also return the 4th column. If you had intended the 4th row (`car[4,]`) and forgotten the comma, this could cause some surprises.

```
car[[4]]
```

```
#  [1] manual manual manual auto   auto   auto   auto   auto   auto   auto
# [11] auto   auto   auto   auto   auto   auto   auto   manual manual manual
# [21] auto   auto   auto   auto   auto   manual manual manual manual manual
```

```
# [31] manual manual
# Levels: auto manual
```

```
head(car[4])
```

```
#    trans
# 1 manual
# 2 manual
# 3 manual
# 4   auto
# 5   auto
# 6   auto
```

However, if we use a single index greater than the number of columns in a data frame, R will throw an error that suggests we have selected *rows* but not columns.

```
car[5]
```

```
# Error in `[.data.frame`(car, 5): undefined columns selected
```

Similarly, if we try to call for 2 indices on a one-dimensional object (vector) we get an "incorrect number of dimensions".

```
car$hp[2, 3]
```

```
# Error in car$hp[2, 3]: incorrect number of dimensions
```

In my experience, these are rather common errors (at least for me!), and you should recognize them.

The function `subset()` is very useful for working with dataframes, since it allows you to extract data from the dataframe based on multiple conditions, and it has an easy to read syntax. For example, we can extract all the records of the `faithful` data with eruptions less than 3 minutes long (`summary()` used here to avoid spewing data over the page).

```
data(faithful)
summary(subset(faithful, eruptions <= 3))
```

```
#    eruptions        waiting
#  Min.   :1.600   Min.   :43.00
#  1st Qu.:1.833   1st Qu.:50.00
#  Median :1.983   Median :54.00
#  Mean   :2.038   Mean   :54.49
#  3rd Qu.:2.200   3rd Qu.:59.00
#  Max.   :2.900   Max.   :71.00
```

III. Attaching data

Many R tutorials will use the function `attach()` to *attach* data to the search path in R. This allows us to call variables by name. For example, in this case we have our data frame `car`, but to get the data in `hp` we need to use `car$hp` - any function that calls `hp` directly won't work - try `mean(hp)`. If we use `attach(car)` then typing `hp` gets us the data, and function calls like `mean(hp)` will now work. There are (in my experience) 2 problems with this:

A) When attaching data, R makes copies of it, so if we change `hp`, the *copy* is changed, but the original data, `car$hp` isn't changed unless we explicitly assign it to be changed - i.e. `hp[2]=NA` is *not the same* as `car$hp[2]=NA`. Read that again - `hp` is *not necessarily* the same as `car$hp`! THIS IS A VERY GOOD REASON NOT TO ATTACH DATA.

```
attach(car)
mean(hp)
```

```
# [1] 146.6875
```

```
hp[1] <- 500
hp[1] == car$hp[1]
```

```
# [1] FALSE
```

B) In my experience it is not uncommon to have multiple data sets that have many of the same variable names (e.g. `biomass`). When attaching, these conflicting names cause even more confusion. For example, if we had *not* removed the vectors `cyl, disp, and hp` above, then when we try `attach(car)` R will give us this message:

```
The following object is masked _by_ .GlobalEnv:

    cyl, disp, hp
```

For these reasons I view `attach()` as a convenience for *demonstration* of R use, and not as a "production" tool. I do not use (or only very rarely) `attach()`, and *when I do I am sure to use* ***`detach()`*** as soon as I am done with the data.

IV. Changing Data Frames

Having imported or created a data frame it is likely that we may want to *alter* it in some way. It is rather simple to remove rows or columns by indexing - `car<-car[-31,]` will *remove* the 31st row of the data and assign the data to its previous name. Similarly `car[,-4]` would remove the 4th column (though here the changed data was not assigned).

It is also very simple to add new columns (or rows) to a data frame - simply index the row (or column) `n+1`, where `n` is the number of rows (or columns). Alternately, just specifying a new name for a variable can create a new column. Here we'll demonstrate both - to calculate a new variable, displacement per cylinder, we first need cylinders as numeric. We'll use the 'approved' method of converting a factor to numeric - indexing the levels (see Chapter 2).

```
car[, 5] <- as.numeric(levels(car$cyl)[car$cyl])
names(car)
```

```
# [1] "cyl"   "disp"  "hp"    "trans" "V5"
```

```
names(car)[5] <- "cyl.numeric"
```

Our data set now has 5 columns, but until we give the new variable a name it is just `"V5"`, for 'Variable 5'. Let's calculate displacement per cylinder:

```
car$disp.per.cyl <- car$disp/car$cyl.numeric
names(car)
```

```
# [1] "cyl"          "disp"         "hp"           "trans"
# [5] "cyl.numeric"  "disp.per.cyl"
```

This method of creating a new variable is easier because we don't have to bother about the variable name, or about which column in will occupy. Had we used a numeric index of 5, we would overwrite the value in that column.

Sometimes we might wish to combine 2 data frames together. We can do this using `cbind()` and `rbind()` (for *column* -wise and *row* -wise binding respectively). The dataset `mtcars` contains several variables that are not in our data frame `cars`. We'll use `cbind()` to combine the 2 data sets.

```
data(mtcars)  # load the data
names(mtcars)  # cols 1,5:8,10:11 not in our data
```

```
#  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
# [11] "carb"
```

```
dim(car)
```

```
# [1] 32  6
```

```
car <- cbind(car, mtcars[, c(1, 5:8, 10:11)])
dim(car)
```

```
# [1] 32 13
```

```
head(car)
```

```
#                   cyl disp  hp  trans cyl.numeric disp.per.cyl  mpg drat
# Mazda RX4           6 2.62 110 manual           6    0.4366667 21.0 3.90
# Mazda RX4 Wag       6 2.62 110 manual           6    0.4366667 21.0 3.90
# Datsun 710          4 1.77  93 manual           4    0.4425000 22.8 3.85
# Hornet 4 Drive      6 4.23 110   auto           6    0.7050000 21.4 3.08
# Hornet Sportabout   8 5.90 175   auto           8    0.7375000 18.7 3.15
# Valiant             6 3.69 105   auto           6    0.6150000 18.1 2.76
#                     wt  qsec vs gear carb
# Mazda RX4         2.620 16.46  0    4    4
# Mazda RX4 Wag     2.875 17.02  0    4    4
# Datsun 710        2.320 18.61  1    4    1
# Hornet 4 Drive    3.215 19.44  1    3    1
# Hornet Sportabout 3.440 17.02  0    3    2
# Valiant           3.460 20.22  1    3    1
```

Note that the row names from `mtcars` have followed the variables from that data frame. A couple of observations about using adding to data frames: * *Don't use `cbind()` if the rows don't correspond!*- we'll see how to use `merge()` (Chapter 10) which is the right tool for this situation. (Similarly don't use `rbind()` if the columns don't correspond). * `cbind()` and `rbind()` are rather slow - don't use them inside a loop! * If you are writing a loop it is far more efficient to make space for your output (whether in a new data frame or by adding to one) *before* the loop begins, adding a row to your data frame in each iteration of a loop will slow your code down.

EXTRA: Comments

There is an attribute of data frames that is reserved for comments. The function `comment()` allows one to set this. `comment(car)` will return `NULL` because no comment has been set, but we can use the same function to set comments.

```
comment(car) <- "A data set derived from the mtcars dataset. Displacement is in liters"
```

Now we have added a comment to this dataset, and `comment(car)` will retrieve it.

V. Exercises

1) Use the `mtcars` data (`data(mtcars)`) to answer these questions (if you get confused, review the bit on logical extraction in Chapter 1):
a) Which rows of the data frame contain cars that weigh more than 4000 pounds (the variable is `wt`, units are 1000 pounds).
b) Which cars are these? (*Hint:* since rows are named by car name, use `row.names()`).

c) What is the mean displacement (in inches^3) for cars with at least 200 horsepower (`hp`).
d) Which car has the highest fuel economy (`mpg`)?
e) What was the fuel economy for the Honda Civic?

2) Using the `mtcars` data create a new variable for horsepower per unit weight (`hp/wt`). Is this a better predictor of acceleration (`qsec`; seconds to complete a quarter mile) than raw horsepower? (Hint - check out correlations between these variables and acceleration, or fit regressions for both models).

3) Use the function `subset()` to return the cars with 4 cylinders and automatic transmissions (`am = 0`). (*Hint:* use "&" for logical "AND"; see `?Logic` and select `Logical Operators`).

Chapter 8: Importing Data

Getting your data into R

I. Introduction

Now that we understand *how* data frames function in R we'll see how they can be created by importing data. Importing data into R can be a difficult process, and many R learners get stuck here and give up (I quit here several times myself!). In order to avoid getting stuck, we'll consider some of the problems that can arise. While this is the shortest chapter in this book, I believe it is among the most important.

II. Importing Data

The most common way to import data into R is by using the function `read.table()` or one of it's derivatives (`read.delim()` and `read.csv()`) [23] . R can easily read three types of data files - *comma separated values* (.csv), *tab delimited text* (.txt), and *space delimited text* (.txt). (I've order them here in my order of preference - .csv files are usually the least troublesome). There are other ways to import data - reading data from the web, from the clipboard, or from spreadsheet files. These are more advanced or limited to specific installations of R - for now we'll focus on the most common and useful tools.

Typical workflow:
1) clean up data in spreadsheet.
* a) replace empty cells with NA (or other consistent string).
* b) fix column names (no spaces, no special characters, '.' and '_' are OK).
* c) save as *.csv (or* .txt).
2) Try to import the data `read.csv(...)`.
* a) if errors occur, open the .csv file in a text editor and find and remove problems (missing new lines, tabs, spaces or spurious quotation marks are common culprits) and repeat 1c) and 2). For this step is especially helpful to have the ability to see "invisible" characters - good text editors [24] will do that.
3) Check the data - `names()`, `dim()`, and `summary()` are my favorite tools, but `str()` works also.

On Using the "Working directory"

R wants to know where to save files and where to find files. When saving or opening files you can specify the complete file path (e.g "C:/Documents and Settings/EssentialR/my-file.R"), but this is not usually necessary. R will look for (or create) files by name in the *working directory*, which is just R speak for the folder you choose to have R use. Use `getwd()` to see what your working directory is. To change it use `setwd("file/path/here")`. Alternately, most R GUIs have a tool to do this - in RStudio go to "Session>Set Working Directory" and select the folder you want to use. The full command will show up in the console - it is a good idea to copy and paste this into your editor - now when you save your code, you have a note of where the working directory for that code is.

> **Note** From here on, these notes assume that you are using the "Code Files" folder in your "EssentialR" directory as the working directory. Examples point to files in the "Data" directory in "EssentialR", so file paths will resemble with "../Data/some-file.csv"; the "../Data/" means "go up what level and find the folder"Data".

[23] If you look at `?read.table` you'll see that `read.delim()` and `read.csv()` are just versions of `read.table()` with different default values. They may be referred to as 'convenience functions'.

[24] A good text editor is a wonderful tool. For Windows Notepad++ is the best I know of, but there may be better. On OSX I like Text Wrangler, on Linux Gedit or Geany work pretty well.

III. An Example

First we'll examine this data in Excel. Open the file "W101-2010.xls". First we'll make sure there are no empty cells (use `NA`) for empty cells. Next we'll delete unnecessary or partial columns. Finally we'll save it as a .csv file. Now we can import it:

```
my.data<-read.csv("C:/file-path/W101-2010.csv",comm="#")
my.data <- read.table("W101-2010.csv",comm="#",header=TRUE,sep=",")
my.data <- read.table(file.choose(),header=TRUE)
```

These are (possibly) three nearly equivalent ways to read the same data! Note that while the first way is a bit more cumbersome, it does have the advantage of keeping a full record of where the file came from. If you use the second form, it might be a good idea to include a note of the working directory (`getwd()` works for this) in your code. If you use the third, you really should note what the file was, or else when you come back to this in a year you won't know what file you read in (ask me how I know!).

Data Import from the clipboard

You *can* read data from the clipboard, but I only recommend this for quick and dirty jobs, since not only do you not have a record of what file the data came from, you also don't know what *part* of the file...
First Select and copy data from excel (or open office). This may cause problems with missing values, so be sure there are no empty cells! *Next choose from the following:*
`my.data <- read.delim("clipboard")` ## Windows
`my.data <- read.table("clipboard",header=T)` ## linux
`my.data <- read.table(pipe("pbpaste"),header=TRUE)` ## Mac
Since this makes your code less clear later I don't really recommend this, but it is sometimes useful in the experimenting phase.

Other Data Import Types

It is possible to import data files (.csv, etc) from websites, either via ftp or http, simply by replacing the file name with the URL. Note that this will not work with https sites, limiting its usefulness somewhat.

The package `xlsx` allows for data to be imported from Microsoft Excel files, though it is worth noting that now one also has to take note of which *worksheet* is imported as well as which file. There are also tools to allow data files from SAS, SPSS, etc - you can find information via Rseek.org.

There are also tools to import data from databases - see this page on Quick R for more information.

Finally note that `write.table()` functions to write R objects to text files, so your can export data as well.

Some Typical Problems

Here I have a couple of examples to show what to look for when data import doesn't work.

> **Note** Make sure your working directory is set to your "Code Files" folder.

```
setwd("~/Dropbox/R class/EssentialR/Code Files")
```

```
my.data <- read.table("../Data/Ex1.txt", header = TRUE)
dim(my.data)
```

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

```
summary(my.data)
```

```
#      MeanDM           DMn          Jday            DMse
#  Min.   :380.0   Min.   :8   Min.   :117.0   Min.   :27.66
```

```
#  1st Qu.:610.8   1st Qu.:8   1st Qu.:130.2   1st Qu.:29.31
#  Median :673.8   Median :8   Median :137.0   Median :31.65
#  Mean   :689.2   Mean   :8   Mean   :137.2   Mean   :39.50
#  3rd Qu.:794.1   3rd Qu.:8   3rd Qu.:146.0   3rd Qu.:42.78
#  Max.   :984.0   Max.   :8   Max.   :155.0   Max.   :71.01
```

```
my.data <- read.delim("../Data/Ex1.txt", header = TRUE)
dim(my.data)
```

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

```
summary(my.data)
```

```
#             MeanDM        DMn             Jday              DMse
#  # mean biomass:1   8       :6   117        :1   27.664       :1
#  380            :1   sample n:1   128        :1   28.663       :1
#  590            :1                137        :2   31.262       :1
#  673.25         :1                149        :1   32.033       :1
#  674.25         :1                155        :1   46.363       :1
#  834            :1                day of year:1   71.014       :1
#  984            :1                                SE of biomass:1
```

These two are different even though the same data was read - why? **Hint** Look at ?read.table and note the default settings for `comment.character`.

Occasionally there are small problems with the file that you can't see in Excel. The file "Ex2.txt" has a missing value with no NA.

```
## my.data <- read.table("../Data/Ex2.txt", header = TRUE)  # produces an error
my.data <- read.table("../Data/Ex2.txt", header = TRUE, sep = "\t")  # no error message
dim(my.data)
```

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

```
summary(my.data)  # but an NA was introduced
```

```
#      MeanDM           DMn          Jday          DMse
#  Min.   :380.0   Min.   :8   Min.   :117   Min.   :27.66
#  1st Qu.:610.8   1st Qu.:8   1st Qu.:137   1st Qu.:29.31
#  Median :673.8   Median :8   Median :137   Median :31.65
#  Mean   :689.2   Mean   :8   Mean   :139   Mean   :39.50
#  3rd Qu.:794.1   3rd Qu.:8   3rd Qu.:149   3rd Qu.:42.78
#  Max.   :984.0   Max.   :8   Max.   :155   Max.   :71.01
#                              NA's   :1
```

Telling R to use a tab (`sep="/t"`) cures this problem because the missing value can now be detected (`read.table()` defaults to `sep=" "`, which can't detect missing values). We could just use `read.delim()` since it looks for tab-delimiters, but we may need to specify the `comment.char` argument.

```
my.data <- read.delim("../Data/Ex2.txt", header = TRUE)
head(my.data)
```

```
#           MeanDM      DMn        Jday          DMse
# 1 # mean biomass sample n day of year SE of biomass
# 2            380        8         117        28.663
# 3         674.25        8         137        31.262
# 4            590        8                   27.664
# 5            834        8         149        46.363
# 6         673.25        8         137        32.033
```

```
my.data <- read.delim("../Data/Ex2.txt", header = TRUE, comment.char = "#")
my.data <- read.delim("../Data/Ex2.txt", header = TRUE, comm = "#")
dim(my.data)
```

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

```
summary(my.data)
```

```
#      MeanDM           DMn          Jday          DMse
#  Min.   :380.0   Min.   :8   Min.   :117   Min.   :27.66
#  1st Qu.:610.8   1st Qu.:8   1st Qu.:137   1st Qu.:29.31
#  Median :673.8   Median :8   Median :137   Median :31.65
#  Mean   :689.2   Mean   :8   Mean   :139   Mean   :39.50
#  3rd Qu.:794.1   3rd Qu.:8   3rd Qu.:149   3rd Qu.:42.78
#  Max.   :984.0   Max.   :8   Max.   :155   Max.   :71.01
#                              NA's   :1
```

Can you see why the `comm="#"` was added?

Another common problem is adjacent empty cells in Excel - an empty cell has had a value in it that has been deleted, and is not the same as a blank cell [25] .These will create an extra delimiter (tab or comma) in the text file, so all rows won't have the same number of values. A really good text editor will show you this kind of error, but when in doubt reopen your .csv file in Excel, copy *only the data* and paste it into a new tab and re-save it as a .csv file. The file "Ex3.txt" includes an error like this. Note that when it is saved as a .csv file this error is visible - this is why .csv files are preferred.

Another problem to watch for is leading quotes - in some cases Excel decides that a line that is commented is actually text and wraps it in quotes. This is invisible in Excel, so you don't know it has happened. When read into R, the leading `"` causes the comment character (by default `#`) to be ignored. You can usually diagnose this if you open the .csv or .txt file in a text editor rather than in Excel.

```
my.data <- read.table("../Data/Ex3.txt", header = TRUE)
## my.data <- read.table("../Data/Ex3.txt", header = TRUE, sep = "\t")  # produces an error
my.data <- read.delim("../Data/Ex3.txt", header = TRUE, comment.char = "#")  # no error
head(my.data)  # but wrong
```

```
#        MeanDM DMn   Jday DMse
# 380         8 117 28.663   NA
# 674.25      8 137 31.262   NA
# 590         8 128 27.664   NA
# 834         8 149 46.363   NA
# 673.25      8 137 32.033   NA
# 984         8 155 71.014   NA
```

The first line did not produce an error, since the separator is `" "` (white space). The second line produced an error because of the extra tab. The third line did not give an error message but the data was not imported correctly - the extra tab created a 5th column (with all NAs), but the 4 names were assigned to the last 4 columns, and the values of MeanDM were used for row.names.

> **Note** If you have numeric data that has thousands separators (e.g. 12,345,678) then you will run into trouble using .csv files (with comma separators). There are several ways to address this problem, but I think the easiest is to change the number format in Excel before creating the .csv file. To do this highlight the cells and choose "number" as the format.

[25] This is one example of why you should use R whenever possible!

IV. A (Possibly) Easier Way

RStudio has a nice built in data import tool. In the "Environment" browser toolbar (upper right pane) there is an icon for "Import Dataset" that features a rather nice dialog for importing data. Importantly this GUI tool writes the code for the action it completes, so it is trivial to copy the code into your document in the editor (so that you can document your work) and add any further arguments or code comments. I held off presenting it until the end for two reasons. First because I want to be sure you understand how `read.table()` works, because the troubleshooting described above may apply when using the GUI also. Second because later versions of RStudio actually use the function `read_table()` (from the package `readr`) rather than `read.table()`. This can cause some unexpected results in the way factors are imported (`read_table()` imports to character rather than factor). If you use the GUI to import data and you have factors in your data, you have a couple of possible ways to proceed. 1) You can just convert your character variables to factors with `as.factor()` - as long as there area small number of variables to convert this is not too unhandy. 2) You can edit the code created by the GUI to use `read.table()` rather than `read_table()`. This will work unless you have massive datasets - for very large data, `read_table()` may be prefereable.

V. Exercises

1) Find (or invent) some data (not from the "Data" directory supplied with EssentialR) and import it into R. (It is not a bad idea to include a commented line with units for each variable in your .txt or.csv file). a) What did you have to do to "clean it up" so it would read in? b) Are you satisfied with the console output of `summary(yourdata)`? Did all the variables import in the way (format) you thought they should? c) Include the output of `summary(yourdata)` and `head(yourdata)`.

2) The spreadsheet "StatesData.xls" located in the `Data` directory in your EssentialR folder contains some (old) data about the 50 US states, and includes a plot with a regression line. Clean this data up and import it into R. You should be able to fit a regression that mimics the plot in the spreadsheet. What is the p-value for the slope in this regression? **********

Chapter 9: Manipulating Data

An introduction to data wrangling

I. Introduction

Often you find that your data needs to be reconfigured in some way. Perhaps you recorded some measurement in two columns, but you realize it really should be a single variable, maybe the total or mean of the two, or a single value conditioned on another value. Or perhaps you want to make some summary figures, like barplots, that need group means. Often you might be tempted to do this kind of work in Excel because you already know how to do it. However, if you already have the data in R, it is probably faster to do it in R! Even better, when you do it in R it is really reproducible in a way that it is not in Excel.

II. Summarizing Data

Frequently we want to calculate data summaries - for example we want to plot means and standard errors for several subsets of a data set [26]. R has useful tools that make this quite simple. We'll look first that the `apply()` family of functions, and then at the very useful `aggregate()`. Fist we'll demonstrate with some data from a study where beans were grown in different size containers [27]. In this study bean plants were grown in varying sized pots (`pot.size`) and either one or two doses of phosphorus (`P.lev`), resulting in varying concentrations of phosphorus (`phos`). Root length and root and shoot biomass were measured.

```
beans <- read.csv("../Data/BeansData.csv", header = TRUE, comm = "#")
dim(beans)
```

```
# [1] 24  8
```

```
summary(beans)
```

```
#     pot.size       phos       P.lev  rep   trt       rt.len
#  Min.   : 4   Min.   : 70.0   H:12   A:6   a:4   Min.   :146.2
#  1st Qu.: 4   1st Qu.:105.0   L:12   B:6   b:4   1st Qu.:243.3
#  Median : 8   Median :175.0          C:6   c:4   Median :280.4
#  Mean   : 8   Mean   :192.5          D:6   d:4   Mean   :301.3
#  3rd Qu.:12   3rd Qu.:210.0                e:4   3rd Qu.:360.3
#  Max.   :12   Max.   :420.0                f:4   Max.   :521.7
#      ShtDM             RtDM
#  Min.   :0.5130   Min.   :0.4712
#  1st Qu.:0.8065   1st Qu.:0.6439
#  Median :1.0579   Median :0.7837
#  Mean   :1.1775   Mean   :0.8669
#  3rd Qu.:1.3159   3rd Qu.:0.9789
#  Max.   :2.7627   Max.   :1.7510
```

The "apply" family of functions are used to "do something over and over again to a subset of some data" (*apply* a function to the data in R-speak). For example we can get means for columns of data:

```
apply(X = beans[, 6:8], MARGIN = 2, FUN = mean, na.rm = TRUE)
```

```
#      rt.len       ShtDM        RtDM
# 301.3179167   1.1774750   0.8668583
```

[26] Excel users will think "Pivot Table".

[27] The data are a simplified form of that reported in: Nord, E. A., Zhang, C., and Lynch, J. P. (2011). Root responses to neighboring plants in common bean are mediated by nutrient concentration rather than self/non-self recognition. Funct. Plant Biol. 38, 941–952.

Here we have applied the function `mean()` to the columns (`MARGIN=2`) 6,7,8 (`6:8`)of `beans` (columns 3,4, and 5 are factors, so`mean()` will give an error, so `apply(X=beans,MARGIN=2,FUN=mean,na.rm=TRUE)` will fail).

Note that we've also specified `na.rm=TRUE` - this is actually an argument to `mean()`, not to `apply()`. If you look at `?apply` you'll find `...` among the arguments. This refers to arguments that are passed to the function specified by `FUN`. Many R functions allow `...` arguments, but they are initially confusing to new users.

In this case there is no missing data, but it is a worthwhile exercise to make a copy of the beans data and introduce an NA so you know what happens when there are missing values in the data.

Often we want to summarize data to get group means, for example we want the means for each treatment type.

```
tapply(beans$rt.len, INDEX = list(beans$trt), FUN = mean, na.rm = TRUE)
```

```
#        a        b        c        d        e        f
# 255.2925 400.2000 178.8750 436.2800 226.7575 310.5025
```

In this case, it was easy because there was a variable (trt) that coded all the treatments together, but we don't really need it:

```
tapply(beans$rt.len, list(beans$pot.size, beans$P.lev), mean, na.rm = TRUE)
```

```
#           H        L
# 4  400.2000 255.2925
# 8  436.2800 178.8750
# 12 310.5025 226.7575
```

This gives us a tidy little table of means [28]. If we just wanted a more straightforward list we can use `paste()` to make a *combined factor*.

```
tapply(beans$rt.len, list(paste(beans$pot.size, beans$P.lev)), mean, na.rm = TRUE)
```

```
#     12 H     12 L      4 H      4 L      8 H      8 L
# 310.5025 226.7575 400.2000 255.2925 436.2800 178.8750
```

Often we really want to get summary data for multiple columns. The function `aggregate()` is a convenience form of apply that makes this trivially easy.

```
aggregate(x = beans[, 6:8], by = list(beans$pot.size, beans$phos), FUN = mean,
    na.rm = TRUE)
```

```
#   Group.1 Group.2   rt.len    ShtDM     RtDM
# 1      12      70 226.7575 0.823000 0.683700
# 2       8     105 178.8750 0.797575 0.599950
# 3      12     140 310.5025 1.265075 0.958225
# 4       4     210 255.2925 0.944375 0.773750
# 5       8     210 436.2800 1.301950 1.000125
# 6       4     420 400.2000 1.932875 1.185400
```

Notice that `by` must be specified as a list when using variable names, and the output lists `Group.1` and `Group.2` rather than the variable names. If we use column numbers we get nicer output and avoid the need to use `list()`. We'll also extract the standard deviations for each group.

```
aggregate(x = beans[, 6:8], by = beans[c(1, 2)], FUN = mean, na.rm = TRUE)
```

```
#   pot.size phos   rt.len    ShtDM     RtDM
# 1       12   70 226.7575 0.823000 0.683700
# 2        8  105 178.8750 0.797575 0.599950
# 3       12  140 310.5025 1.265075 0.958225
```

[28] If we were summarizing by three variable rather than two we would get 3-d matrix.

```
# 4        4  210 255.2925 0.944375 0.773750
# 5        8  210 436.2800 1.301950 1.000125
# 6        4  420 400.2000 1.932875 1.185400
```

```
aggregate(beans[, 6:8], beans[c(1, 2)], sd, na.rm = TRUE)
```

```
#   pot.size phos    rt.len     ShtDM      RtDM
# 1       12   70  51.51115 0.2491654 0.2204654
# 2        8  105  50.64085 0.2039078 0.1021530
# 3       12  140  34.80957 0.3835852 0.2774636
# 4        4  210  32.37071 0.2066067 0.1503608
# 5        8  210 100.21786 0.3904458 0.2689229
# 6        4  420  86.66397 0.8402876 0.5187955
```

What if we want to specify a `FUN` that doesn't exist in R? Simple - we write our own.

```
aggregate(beans[, 6:8], beans[c(1, 2)], function(x) (sd(x, na.rm = TRUE)/(length(x) -
    sum(is.na(x)))^0.5))
```

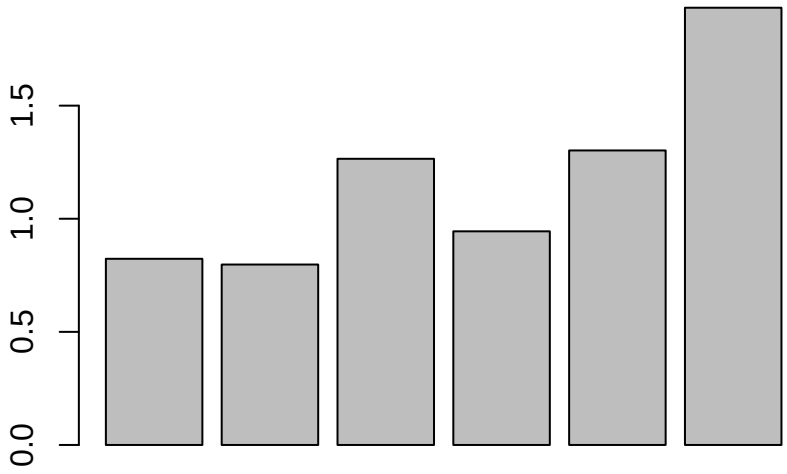
```
#   pot.size phos   rt.len     ShtDM       RtDM
# 1       12   70 25.75557 0.1245827 0.11023268
# 2        8  105 25.32042 0.1019539 0.05107649
# 3       12  140 17.40478 0.1917926 0.13873179
# 4        4  210 16.18536 0.1033033 0.07518041
# 5        8  210 50.10893 0.1952229 0.13446147
# 6        4  420 43.33198 0.4201438 0.25939775
```

Here we've defined the function (`function(x)`) on the fly. We could also begin by first defining a function: `SE=function(x)` followed by the definition of the function, and then just use `FUN=SE` in our call to aggregate.

This is a good example of something I have to look up in my "R Tricks"" file. It is also an example of how lines of R code can get really long (and why auto balancing of parentheses is really nice)! Adding spaces is OK

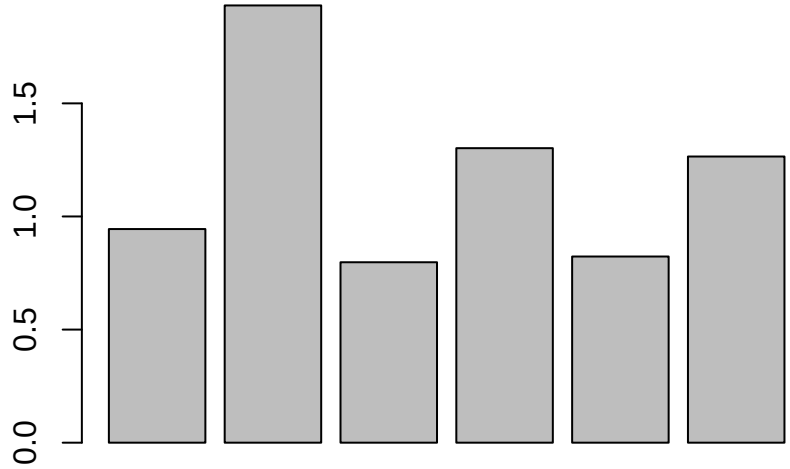
It is easy to see how useful summaries like this could be - let's make a plot from this.

```
beans.means <- aggregate(beans[, 6:8], beans[c(1, 2)], mean, na.rm = TRUE)
barplot(beans.means$ShtDM)
```



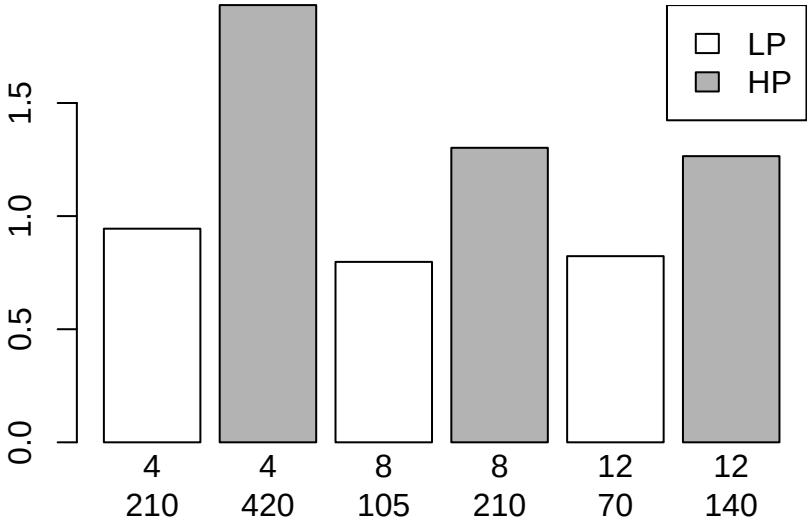
If we don't like the order of the bars here we can use the function `order()` to sort our data (`beans.means`) for nicer plotting.

```
beans.means <- beans.means[order(beans.means$pot.size, beans.means$phos), ]
barplot(beans.means$ShtDM)
```



Now we'll add the labels and legend - we'll discuss these in more detail in later lessons.

```
barplot(beans.means$ShtDM,col=c("white","grey70"),names.arg=
    paste(beans.means$pot.size,beans.means$phos,sep="\n"),ylab=
    "Shoot biomass (g)")
legend("topright",fill=c("white","grey70","grey50"),legend=c("LP","HP"))
```



You can see that R gives us powerful tools for data manipulation.

III. Reformatting Data from "Wide" to "Long"

Sometimes we record and enter data in a different format than that in which we wish to analyze it. For example, I might record the biomass from each of several replicates of an experiment in separate columns for convenience. But when I want to analyze it, I need biomass as a single column, with another column for replicate. Or perhaps I have biomass as a single column and want to break it into separate columns. The R functions `stack()` and `unstack()` are a good place to begin.

```
data(PlantGrowth)
head(PlantGrowth)
```

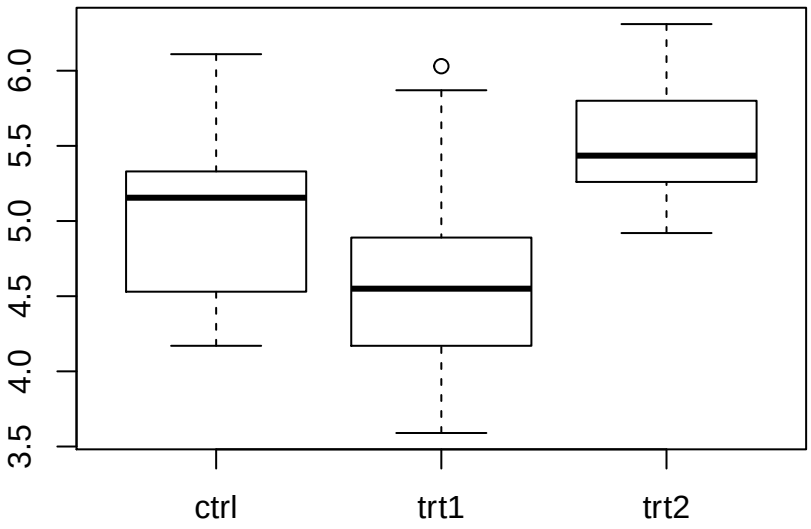
```
#   weight group
# 1   4.17  ctrl
# 2   5.58  ctrl
# 3   5.18  ctrl
# 4   6.11  ctrl
# 5   4.50  ctrl
# 6   4.61  ctrl
```

We have 2 variables: weight and group. We can use `unstack()` to make one column for each level of group.

```
unstack(PlantGrowth)
```

```
#    ctrl trt1 trt2
# 1  4.17 4.81 6.31
# 2  5.58 4.17 5.12
# 3  5.18 4.41 5.54
# 4  6.11 3.59 5.50
# 5  4.50 5.87 5.37
# 6  4.61 3.83 5.29
# 7  5.17 6.03 4.92
# 8  4.53 4.89 6.15
# 9  5.33 4.32 5.80
# 10 5.14 4.69 5.26
```

```
pg <- unstack(PlantGrowth, weight ~ group)
boxplot(pg)
```



This can be useful for plotting (though in this case `boxplot(weight~group,data=PlantGrowth)` would work equally well). We can use `stack()` to go the other way.

```
summary(stack(pg))
```

```
#      values        ind
#  Min.   :3.590   ctrl:10
#  1st Qu.:4.550   trt1:10
#  Median :5.155   trt2:10
#  Mean   :5.073
#  3rd Qu.:5.530
#  Max.   :6.310
```

```
summary(stack(pg, select = c(trt1, trt2)))
```

```
#      values        ind
#  Min.   :3.590   trt1:10
#  1st Qu.:4.620   trt2:10
#  Median :5.190
#  Mean   :5.093
#  3rd Qu.:5.605
#  Max.   :6.310
```

```
summary(stack(pg, select = -ctrl))
```

```
#      values        ind
#  Min.   :3.590   trt1:10
#  1st Qu.:4.620   trt2:10
#  Median :5.190
#  Mean   :5.093
#  3rd Qu.:5.605
#  Max.   :6.310
```

Notice that we can use the `select` argument to specify or exclude columns when we stack.

Suppose we're interested in comparing any treatment against the control with the PlantGrowth data. We've already seen how this can be done using the function `levels()`. There are three levels, but if we reassign one of them we can make 2 levels.

```
levels(PlantGrowth$group)
```

```
# [1] "ctrl" "trt1" "trt2"
```

```
PlantGrowth$group2 <- factor(PlantGrowth$group)
levels(PlantGrowth$group2) <- c("Ctrl", "Trt", "Trt")
summary(PlantGrowth)
```

```
#      weight       group     group2
#  Min.   :3.590   ctrl:10   Ctrl:10
#  1st Qu.:4.550   trt1:10   Trt :20
#  Median :5.155   trt2:10
#  Mean   :5.073
#  3rd Qu.:5.530
#  Max.   :6.310
```

```
unstack(PlantGrowth, weight ~ group2)
```

```
# $Ctrl
#  [1] 4.17 5.58 5.18 6.11 4.50 4.61 5.17 4.53 5.33 5.14
#
# $Trt
#  [1] 4.81 4.17 4.41 3.59 5.87 3.83 6.03 4.89 4.32 4.69 6.31 5.12 5.54 5.50
# [15] 5.37 5.29 4.92 6.15 5.80 5.26
```

We can even use `unstack()` to split `weight` based on `group2`, but the output is different as the groups aren't balanced.

IV. Reshape

For more sophisticated reshaping of data the package "reshape" has powerful tools to reshape data in many ways, but it takes some time to read the documentation and wrap your head around how it works and what it can do.

Basically there are 2 functions here - `melt()` and `cast()` (think working with metal) - once you 'melt' the data you can 'cast' it into any shape you want. `melt()` turns a data set into a series of observations which consist of a variable and value, and `cast()` reshapes melted data, 'casting' it into a new form. You will probably need to install `reshape` - either from the 'Packages' tab or via `install.packages("reshape")`[29]. We'll demonstrate with the built in data set `iris`, and we'll need to create a unique identifier for each case in the data set.

```
require(reshape)
data(iris)
iris$id <- row.names(iris)
head(melt(iris, id = "id"))
```

```
#   id     variable value
# 1  1 Sepal.Length   5.1
# 2  2 Sepal.Length   4.9
# 3  3 Sepal.Length   4.7
# 4  4 Sepal.Length   4.6
# 5  5 Sepal.Length     5
# 6  6 Sepal.Length   5.4
```

```
tail(melt(iris, id = "id"))
```

```
#      id variable     value
# 745 145  Species virginica
# 746 146  Species virginica
# 747 147  Species virginica
# 748 148  Species virginica
# 749 149  Species virginica
# 750 150  Species virginica
```

```
melt.iris <- melt(iris, id = c("id", "Species"))
dim(melt.iris)
```

```
# [1] 600   4
```

```
head(melt.iris)
```

```
#   id Species     variable value
# 1  1  setosa Sepal.Length   5.1
# 2  2  setosa Sepal.Length   4.9
# 3  3  setosa Sepal.Length   4.7
# 4  4  setosa Sepal.Length   4.6
# 5  5  setosa Sepal.Length   5.0
# 6  6  setosa Sepal.Length   5.4
```

```
tail(melt.iris)
```

```
#      id   Species    variable value
# 595 145 virginica Petal.Width   2.5
```

[29] Do recall that you need to then *load* the package either via `library(reshape)`, `require(reshape)`, or via RStudio's packages pane. Note that if you are using knitr, you will need to include the code to load the package (e.g. `require(reshape)`) in you file, since it will need to be loaded into the clean workspace where knitr evaluates the code.

```
# 596 146 virginica Petal.Width   2.3
# 597 147 virginica Petal.Width   1.9
# 598 148 virginica Petal.Width   2.0
# 599 149 virginica Petal.Width   2.3
# 600 150 virginica Petal.Width   1.8
```

Now instead of 150 observations with 6 variables we have 600 observations with 4 variables. We can cast this data using `cast()`. If we specify enough columns from our melted data to account for all the data, then we don't need to specify a `fun.aggregate` - a function with which to aggregate, but we can aggregate the data easily,and with more flexibility than by using aggregate:

```
cast(melt.iris, Species ~ variable, mean)
```

```
#      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
# 1     setosa        5.006       3.428        1.462       0.246
# 2 versicolor        5.936       2.770        4.260       1.326
# 3  virginica        6.588       2.974        5.552       2.026
```

```
cast(melt.iris, Species ~ variable, range)
```

```
#      Species Sepal.Length_X1 Sepal.Length_X2 Sepal.Width_X1 Sepal.Width_X2
# 1     setosa             4.3             5.8            2.3            4.4
# 2 versicolor             4.9             7.0            2.0            3.4
# 3  virginica             4.9             7.9            2.2            3.8
#   Petal.Length_X1 Petal.Length_X2 Petal.Width_X1 Petal.Width_X2
# 1             1.0             1.9            0.1            0.6
# 2             3.0             5.1            1.0            1.8
# 3             4.5             6.9            1.4            2.5
```

We can get our original data back.

```
head(cast(melt.iris, Species + id ~ variable))
```

```
#   Species id Sepal.Length Sepal.Width Petal.Length Petal.Width
# 1  setosa  1          5.1         3.5          1.4         0.2
# 2  setosa 10          4.9         3.1          1.5         0.1
# 3  setosa 11          5.4         3.7          1.5         0.2
# 4  setosa 12          4.8         3.4          1.6         0.2
# 5  setosa 13          4.8         3.0          1.4         0.1
# 6  setosa 14          4.3         3.0          1.1         0.1
```

But we can also do other types of reshaping. For example, what if we wanted to separate out Sepal and Petal variables for each record? We can use `strplit()` to split the stings that represent variables, e.g. "Sepal.Width".

```
head(strsplit(as.character(melt.iris$variable), split = ".", fixed = TRUE))
```

```
# [[1]]
# [1] "Sepal"  "Length"
#
# [[2]]
# [1] "Sepal"  "Length"
#
# [[3]]
# [1] "Sepal"  "Length"
#
# [[4]]
# [1] "Sepal"  "Length"
```

```
#
# [[5]]
# [1] "Sepal"  "Length"
#
# [[6]]
# [1] "Sepal"  "Length"
```

Notice that this returns a list. We'll have to use `do.call()` to call `rbind()` on the list to bind the list elements into a data frame.

```
head(do.call(rbind, strsplit(as.character(melt.iris$variable), split = ".",
    fixed = TRUE)))
```

```
#      [,1]    [,2]
# [1,] "Sepal" "Length"
# [2,] "Sepal" "Length"
# [3,] "Sepal" "Length"
# [4,] "Sepal" "Length"
# [5,] "Sepal" "Length"
# [6,] "Sepal" "Length"
```

Now this seems a bit esoteric, but we can use `cbind()` to bind these values to our melted iris data

```
melt.iris <- cbind(melt.iris, do.call(rbind, strsplit(as.character(melt.iris$variable),
    split = ".", fixed = TRUE)))
names(melt.iris)[5:6] <- c("Part", "Dimension")
head(melt.iris)
```

```
#   id Species     variable value  Part Dimension
# 1  1  setosa Sepal.Length   5.1 Sepal    Length
# 2  2  setosa Sepal.Length   4.9 Sepal    Length
# 3  3  setosa Sepal.Length   4.7 Sepal    Length
# 4  4  setosa Sepal.Length   4.6 Sepal    Length
# 5  5  setosa Sepal.Length   5.0 Sepal    Length
# 6  6  setosa Sepal.Length   5.4 Sepal    Length
```

Now we can see that we have separated the flower parts (Sepal or Petal) from the dimensions.

```
cast(melt.iris, Species ~ Dimension | Part, mean)
```

```
# $Petal
#      Species Length Width
# 1     setosa  1.462 0.246
# 2 versicolor  4.260 1.326
# 3  virginica  5.552 2.026
#
# $Sepal
#      Species Length Width
# 1     setosa  5.006 3.428
# 2 versicolor  5.936 2.770
# 3  virginica  6.588 2.974
```

```
cast(melt.iris, Species ~ Dimension + Part, mean)
```

```
#      Species Length_Petal Length_Sepal Width_Petal Width_Sepal
# 1     setosa        1.462        5.006       0.246       3.428
# 2 versicolor        4.260        5.936       1.326       2.770
# 3  virginica        5.552        6.588       2.026       2.974
```

So we can talk about the mean Length and Width, averaged over floral parts. In this case it may not make sense to do so, but it demonstrates the type of data reconfiguration that is possible.

```
cast(melt.iris, Species ~ Dimension, mean)
```

```
#      Species Length Width
# 1     setosa  3.234 1.837
# 2 versicolor  5.098 2.048
# 3  virginica  6.070 2.500
```

We can still go back to the original data by just casting the data in a different form:

```
head(cast(melt.iris, Species + id ~ Part + Dimension))
```

```
#   Species id Petal_Length Petal_Width Sepal_Length Sepal_Width
# 1  setosa  1          1.4         0.2          5.1         3.5
# 2  setosa 10          1.5         0.1          4.9         3.1
# 3  setosa 11          1.5         0.2          5.4         3.7
# 4  setosa 12          1.6         0.2          4.8         3.4
# 5  setosa 13          1.4         0.1          4.8         3.0
# 6  setosa 14          1.1         0.1          4.3         3.0
```

The package **reshape** adds important tools to the R data manipulation toolkit. While they may be a bit tricky to learn, they are very powerful. See ?cast for more examples.

V. Merging Data Sets

Another data manipulation task is merging two data sets together. Perhaps you have field and laboratory results in different files, and you want to merge them into one file. Here we'll use an example from the `merge()` help file.

```
authors <- data.frame(surname = c("Tukey", "Venables", "Tierney", "Ripley",
    "McNeil"), nationality = c("US", "Australia", "US", "UK", "Australia"),
    deceased = c("yes", rep("no", 4)))
books <- data.frame(name = c("Tukey", "Venables", "Tierney", "Ripley", "Ripley",
    "McNeil", "R Core"), title = c("Expl. Data Anal.", "Mod. Appl. Stat ...",
    "LISP-STAT", "Spatial Stat.", "Stoch. Simu.", "Inter. Data Anal.", "An Intro. to R"),
    other.author = c(NA, "Ripley", NA, NA, NA, NA, "Venables & Smith"))
authors
```

```
#    surname nationality deceased
# 1    Tukey          US      yes
# 2 Venables   Australia       no
# 3  Tierney          US       no
# 4   Ripley          UK       no
# 5   McNeil   Australia       no
```

```
books
```

```
#       name               title     other.author
# 1    Tukey    Expl. Data Anal.             <NA>
# 2 Venables Mod. Appl. Stat ...           Ripley
# 3  Tierney           LISP-STAT             <NA>
# 4   Ripley       Spatial Stat.             <NA>
# 5   Ripley        Stoch. Simu.             <NA>
# 6   McNeil   Inter. Data Anal.             <NA>
# 7   R Core      An Intro. to R Venables & Smith
```

We've created 2 small data frames for demonstration purposes here. Now we can use `merge()` to merge them.

```
(m1 <- merge(authors, books, by.x = "surname", by.y = "name"))
```

```
#    surname nationality deceased               title other.author
# 1   McNeil   Australia       no   Inter. Data Anal.         <NA>
# 2   Ripley          UK       no       Spatial Stat.         <NA>
# 3   Ripley          UK       no        Stoch. Simu.         <NA>
# 4  Tierney          US       no           LISP-STAT         <NA>
# 5    Tukey          US      yes    Expl. Data Anal.         <NA>
# 6 Venables   Australia       no Mod. Appl. Stat ...       Ripley
```

```
(m2 <- merge(books, authors, by.x = "name", by.y = "surname"))
```

```
#       name               title other.author nationality deceased
# 1   McNeil   Inter. Data Anal.         <NA>   Australia       no
# 2   Ripley       Spatial Stat.         <NA>          UK       no
# 3   Ripley        Stoch. Simu.         <NA>          UK       no
# 4  Tierney           LISP-STAT         <NA>          US       no
# 5    Tukey    Expl. Data Anal.         <NA>          US      yes
# 6 Venables Mod. Appl. Stat ...       Ripley   Australia       no
```

Notice that the order of the columns mirrors the order of columns in the function call - in the first line we asked for `authors,books` and the columns are the three columns of `authors` and the all but the first column of `books`, because that column (`name`) is the `by.y` column. If both data frames had the column `surname` we could just say `by=surname`. Notice that "R core" (`books[7,]`) is not included in the combined data frame - this is because it does not exist in *both* of them. We can override this, but `NA`s will be introduced. Also note that `by`, `by.x` and `by.y` can be vectors of more than one variable -useful for complex data sets.

```
merge(authors, books, by.x = "surname", by.y = "name", all = TRUE)
```

```
#    surname nationality deceased               title     other.author
# 1   McNeil   Australia       no   Inter. Data Anal.             <NA>
# 2   Ripley          UK       no       Spatial Stat.             <NA>
# 3   Ripley          UK       no        Stoch. Simu.             <NA>
# 4  Tierney          US       no           LISP-STAT             <NA>
# 5    Tukey          US      yes    Expl. Data Anal.             <NA>
# 6 Venables   Australia       no Mod. Appl. Stat ...           Ripley
# 7   R Core        <NA>     <NA>      An Intro. to R Venables & Smith
```

This is a good example of something that is much easier to do in R than in Excel.

VI. More about Loops

Sometimes we want to have R do something over and over again. Often a *loop* is the simplest [30] way to do this. AS we saw earlier the general syntax for a loop in R is: `for(index) {do something}`.

The curly braces are optional if it fits on one line, but required if it goes over one line.

Here are a couple of examples:

```
for (i in 1:5) print(i)
```

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

[30] Though possibly not the fastest - advanced R users will sometimes say that you should use an `apply()` function rather than a loop because it will run faster. This is probably only a real concern if you have very large data sets.

```
# [1] 4
# [1] 5
```

```
x <- c(2, 5, 7, 23, 6)
for (i in x) {
    cat(paste("i^2=", i^2, "\n"))
}
```

```
# i^2= 4
# i^2= 25
# i^2= 49
# i^2= 529
# i^2= 36
```

Another example might be converting multiple numeric columns to factors. Imagine we wanted to convert columns 3,5,and 6 of a data frame from numeric to factor. We could run (nearly) the same command three times:

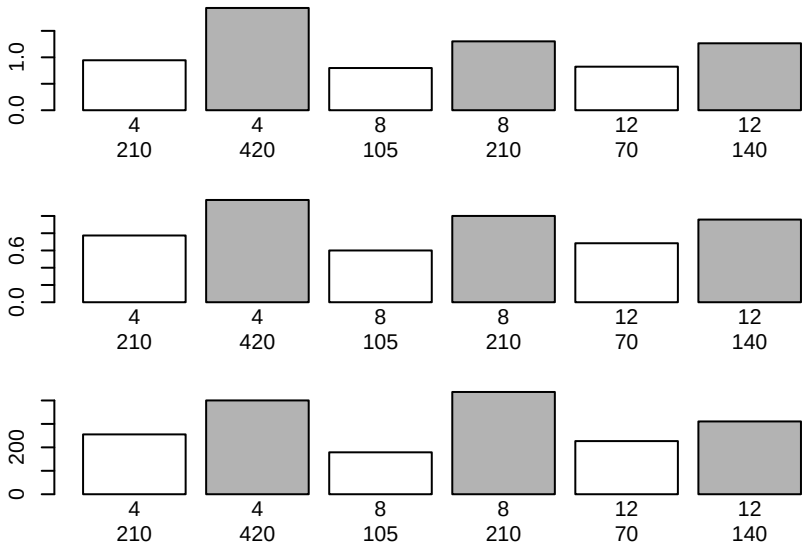
```
# df[,3]=factor(df[,3]); df[,5]=factor(df[,5]); df[,6]=factor(df[,6]);
```

However, particularly if we have more than two columns that need to be converted it may be easier to use a loop - just use the vector of the columns to be converted as the index:

```
# for(i in c(3,5,6)) df[,i]<-factor(df[,i])
```

Loops can be used in many ways - for example the code we used to plot shoot biomass could be put in a loop to plot all the response variables in the data.

```
par(mfrow=c(3,1),mar=c(3,3,0.5,0.5))
for (p in c(4,5,3)){
  barplot(beans.means[,p],col=c("white","grey70"),
    names.arg=paste(beans.means$pot.size,beans.means$phos,sep="\n"))
 # plot the pth column of beans.means
  }
```



Obviously, this plot isn't perfect yet, but it is good enough to be useful. Note - by convention, the code *inside* a loop is indented to make it easier to see where a loop begins and ends.

VII. Exercises

1) Load the data set "ufc" (the file is ufc.csv). This data shows diameter at breast height (Dbh) and Height for forest trees. Can you use `unstack()` to get the diameter data for white pine (`WP`)? Start by unstacking all the diameter data. Can you also get this data by logical extraction? (*Hint:* use the function `which()`. If you really only wanted the data for one species logical extraction would probably be better.)

2) For the data set ufc find the mean Dbh and Height for each species. (*Hint:* aggregate is your friend for more than one response variable.)

3) Make a barplot showing these mean values for each species. Use beside =TRUE (stacking two different variables wouldn't make sense...). (*Hint:* this will be easier if you make a new variable for the means from Q2. Look at ?barplot for the data type "height" must have - `as.matrix()` can be used to make something a matrix.)

4) The barplot in Q3 suggests a fair correlation between Dbh and height. Plot Height~DBH, fit a regression, and plot the line. What is the R^2?

Chapter 10: Working with multiple variables

Some basic tools for multivariate data

I. Introduction

Now that we've discussed the data frame in R, and seen how data can be imported, we can begin to practice working with multiple variables. Rather than a full introduction to multivariate methods, here we'll cover some basic tools.

II. Working with Multivariate Data

Working with more than two variables becomes more complex, but many of the tools we've already learned can also help us here. We'll use the `mtcars` data we referred to in the Chapters 4 & 5, but now we'll load it directly.

```
data(mtcars)
summary(mtcars)
```

```
#       mpg             cyl             disp             hp
#  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
#  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
#  Median :19.20   Median :6.000   Median :196.3   Median :123.0
#  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
#  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
#  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
#       drat             wt             qsec             vs
#  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
#  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
#  Median :3.695   Median :3.325   Median :17.71   Median :0.0000
#  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
#  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
#  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
#       am              gear            carb
#  Min.   :0.0000   Min.   :3.000   Min.   :1.000
#  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
#  Median :0.0000   Median :4.000   Median :2.000
#  Mean   :0.4062   Mean   :3.688   Mean   :2.812
#  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
#  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

If you look at the "Environment" tab in RStudio you should now see `mtcars` under "Data". We'll convert some of this data to factors, since as we discussed before, the number of cylinders, transmission type (and number of carburetors, V/S, and number of forward gears) aren't really continuous.

```
for (i in c(2, 8, 9, 10, 11)) {
    mtcars[, i] = factor(mtcars[, i])
}
names(mtcars)[9] <- "trans"
levels(mtcars$trans) <- c("auto", "manual")
summary(mtcars)
```

```
#       mpg        cyl         disp             hp             drat
#  Min.   :10.40   4:11   Min.   : 71.1   Min.   : 52.0   Min.   :2.760
#  1st Qu.:15.43   6: 7   1st Qu.:120.8   1st Qu.: 96.5   1st Qu.:3.080
```

```
#  Median :19.20   8:14   Median :196.3   Median :123.0   Median :3.695
#  Mean   :20.09          Mean   :230.7   Mean   :146.7   Mean   :3.597
#  3rd Qu.:22.80          3rd Qu.:326.0   3rd Qu.:180.0   3rd Qu.:3.920
#  Max.   :33.90          Max.   :472.0   Max.   :335.0   Max.   :4.930
#        wt             qsec       vs        trans    gear   carb
#  Min.   :1.513   Min.   :14.50   0:18   auto  :19   3:15   1: 7
#  1st Qu.:2.581   1st Qu.:16.89   1:14   manual:13   4:12   2:10
#  Median :3.325   Median :17.71                      5: 5   3: 3
#  Mean   :3.217   Mean   :17.85                             4:10
#  3rd Qu.:3.610   3rd Qu.:18.90                             6: 1
#  Max.   :5.424   Max.   :22.90                             8: 1
```

Notice the `for(){}` loop here; just as we saw in the last chapter this is faster than writing 5 lines of code. I prefer to have informative factor names, so we'll change that, and then check `summary()` to make sure it is all OK.

Note that we could achieve the same result using the list version of `apply()`, `lapply()`. This will run faster, but the code is slightly longer than the loop version above.

```
mtcars[, c(2, 8, 9, 10, 11)] <- lapply(mtcars[, c(2, 8, 9, 10, 11)], FUN = function(x) (factor(x)))
```

In the last lesson we used `table()` for two-way tables, but we can also use it for three-way tables.

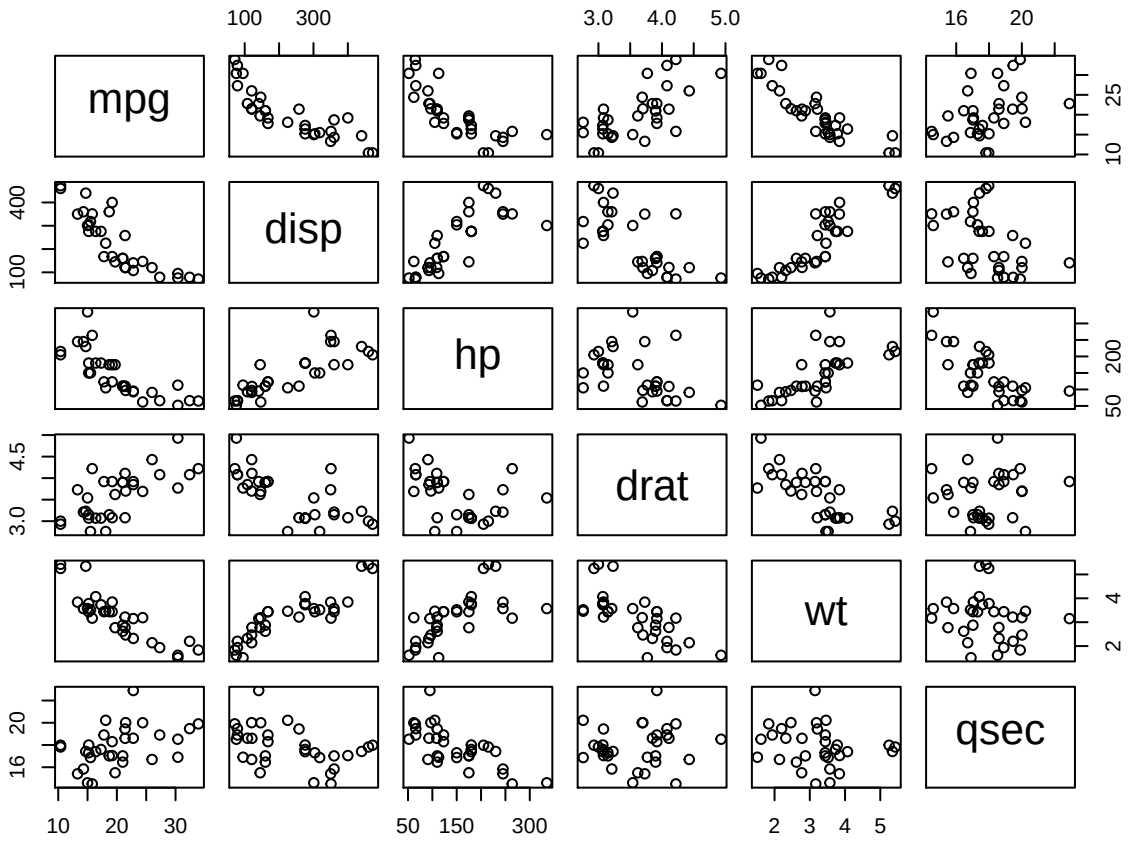
```
with(mtcars, table(carb, cyl, trans))
```

```
# , , trans = auto
#
#     cyl
# carb 4 6 8
#    1 1 2 0
#    2 2 0 4
#    3 0 0 3
#    4 0 2 5
#    6 0 0 0
#    8 0 0 0
#
# , , trans = manual
#
#     cyl
# carb 4 6 8
#    1 4 0 0
#    2 4 0 0
#    3 0 0 0
#    4 0 2 1
#    6 0 1 0
#    8 0 0 1
```

This give us bit more insight - cars with automatic transmissions don't seem to have more than 4 carburetors, but cars with manual transmissions might have as many as 8, but not more carburetors than cylinders.

Another tool that is often useful to explore multivariate data is the "pairs plot", which shows correlations for all pairs. Since we have 3 factors here, lets exclude them so we can focus on the numeric variables.

```
pairs(mtcars[, -c(2, 8, 9, 10, 11)])
```



Now we can see what patterns of correlation exist in this dataset. Fuel economy ("mpg") is relatively well correlated (negatively) with displacement, though perhaps not in a linear fashion. Quarter-mile time (qsec) is not strongly correlated with any of the other variables (but there may be a weak negative correlation with hp, which stands to reason).

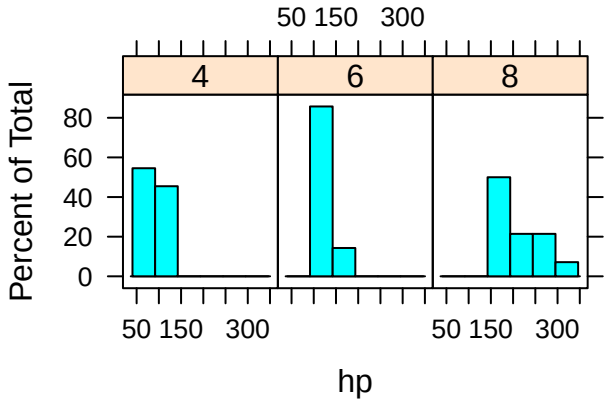
Lattice Graphics

The "Lattice" package includes nice functionality for making plots conditioned on a third variable. Either use `install.packages("lattice")` if your are connected to the internet or `install.packages(repos=NULL,type="source",pkg` and select the package file for "lattice". Remember that to use it you have to load it: `library(lattice)`

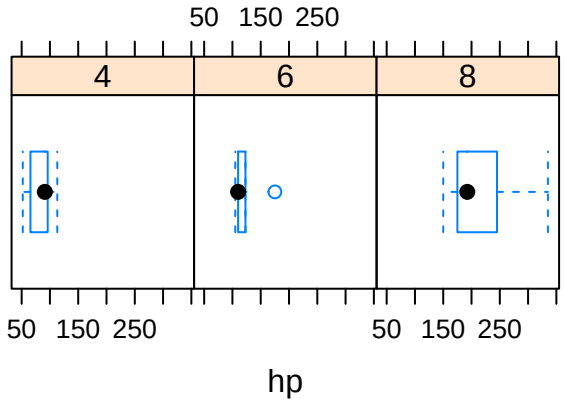
Alternately, you can go to the "Packages" tab in RStudio and click "Install Packages". Note that while you should only need to *install* a package once, you will need to *load* it (via the function `library()`) each session that you wish to use it - this is to keep R from getting to bloated.

The `lattice` package has functions that plot data *conditioned* on another variable, which is specified by the `|` operator.

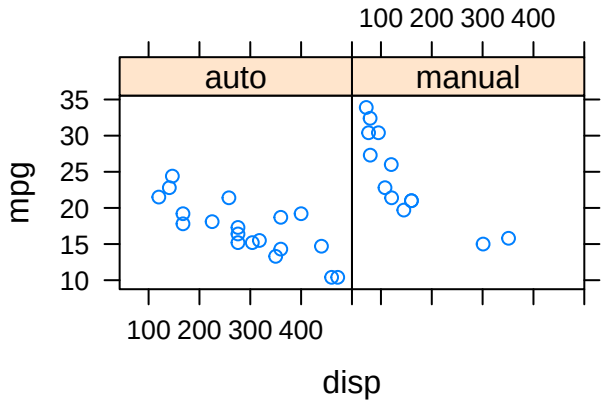
```
histogram(~hp | cyl, data = mtcars)
```




```
bwplot(~hp | cyl, data = mtcars)
```



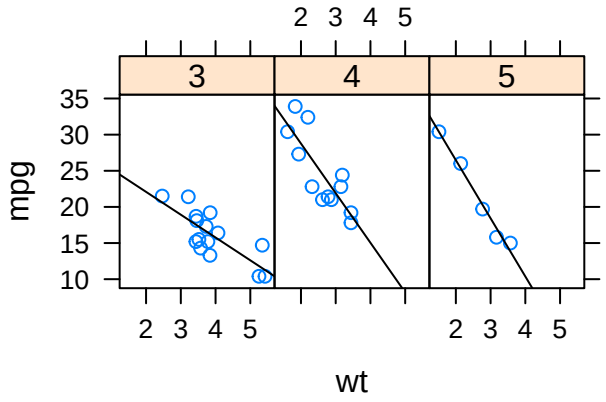

```
xyplot(mpg ~ disp | trans, data = mtcars)
```



EXTRA: Customizing Lattice Plots

By defining custom functions, we can customize lattice plots. Here we'll define a custom function using `panel.` functions from `lattice`. The actual definition of the function is simple, knowing the pieces needed to define this particular function is less so.

```
plot.regression = function(x, y) {
    panel.xyplot(x, y)
    panel.abline(lm(y ~ x))
}
xyplot(mpg ~ wt | gear, panel = plot.regression, data = mtcars)
```



III. An Example

For example let us consider some data from the pilot study on root anatomy that we looked at in the last chapter. 123 root cross sections were analyzed for a suite of 9 anatomical traits. There are several sources of variation here - samples are from different genotypes (12), grown in different media (2), and from different locations in the root system (2).

In the last chapter we saw how to covert several columns to factors, how to change factor levels, and how to calculate new variables.

```
getwd()  ## should be 'Code Files'
```

```
# [1] "/Users/enord/Dropbox/R_Class/EssentialR/Chapters"
```

```
anat <- read.table("../Data/anatomy-pilot-simple.txt", header = TRUE, sep = "\t")
summary(anat)
```

```
#      Gtype    media.rep     sample      Loc        RSXA.mm2
#  C      :12   r1:31     Min.   :1.000   L1:60   Min.   :0.0681
#  D      :12   R1:29     1st Qu.:1.000   L2:63   1st Qu.:0.9682
#  F      :12   r2:32     Median :2.000           Median :1.1354
#  G      :12   R2:31     Mean   :1.943           Mean   :1.1225
#  I      :12             3rd Qu.:3.000           3rd Qu.:1.2789
#  B      :11             Max.   :3.000           Max.   :1.6347
#  (Other):52
#     TCA.mm2           AA.mm2        Cort.Cell.Num     XVA.mm2
#  Min.   :0.0545   Min.   :0.0057   Min.   : 510   Min.   :0.00240
#  1st Qu.:0.7560   1st Qu.:0.1153   1st Qu.:1542   1st Qu.:0.04080
#  Median :0.9045   Median :0.2073   Median :1817   Median :0.04960
#  Mean   :0.8881   Mean   :0.2098   Mean   :1857   Mean   :0.05079
#  3rd Qu.:1.0176   3rd Qu.:0.2837   3rd Qu.:2136   3rd Qu.:0.06070
#  Max.   :1.3882   Max.   :0.5084   Max.   :3331   Max.   :0.08990
#
#      Per.A          CellSize.1          CellSize.2
#  Min.   : 1.359   Min.   :0.0000610   Min.   :0.0000680
#  1st Qu.:15.428   1st Qu.:0.0003300   1st Qu.:0.0003025
#  Median :23.258   Median :0.0004830   Median :0.0005520
#  Mean   :22.886   Mean   :0.0006254   Mean   :0.0007299
#  3rd Qu.:29.313   3rd Qu.:0.0008675   3rd Qu.:0.0009215
#  Max.   :46.262   Max.   :0.0017030   Max.   :0.0036640
#
#    CellSize.3          CellSize.4                 Comments
```

```
#  Min.   :0.0000470   Min.   :0.0000280                :98
#  1st Qu.:0.0001610   1st Qu.:0.0001150    mc          :13
#  Median :0.0001950   Median :0.0001390    wried anatomy: 3
#  Mean   :0.0002052   Mean   :0.0001400    blur image   : 2
#  3rd Qu.:0.0002380   3rd Qu.:0.0001615    a little blur: 1
#  Max.   :0.0004380   Max.   :0.0002710    dull blade   : 1
#                                          (Other)      : 5
```

```
for (i in c(1, 3)) anat[, i] <- factor(anat[, i])  # cols 1,3 to factors
levels(anat$media.rep) <- c("R1", "R1", "R2", "R2")  # r1,2 to R1,2
anat$CellSize.avg <- rowMeans(anat[, 11:14])  # avgCellSize
anat$m.Loc <- factor(paste(anat$media.rep, anat$Loc))  # combined levels
```

We've fixed that. Now that we have estimates for average CellSize let's remove the original cell size values as well as the comments column.

```
anat <- anat[, -(11:15)]
names(anat)
```

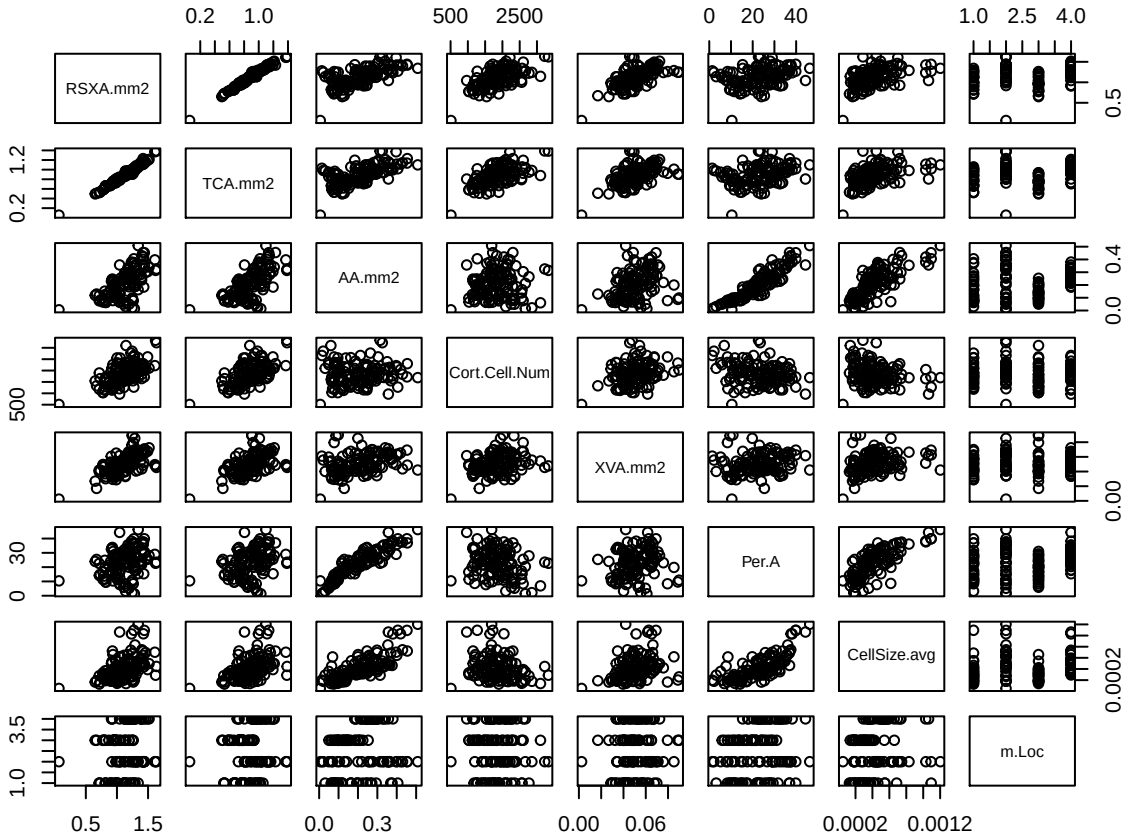
```
#  [1] "Gtype"         "media.rep"     "sample"        "Loc"
#  [5] "RSXA.mm2"      "TCA.mm2"       "AA.mm2"        "Cort.Cell.Num"
#  [9] "XVA.mm2"       "Per.A"         "CellSize.avg"  "m.Loc"
```

```
summary(anat)
```

```
#      Gtype    media.rep sample Loc        RSXA.mm2         TCA.mm2
#  C      :12   R1:60     1:43   L1:60   Min.   :0.0681   Min.   :0.0545
#  D      :12   R2:63     2:44   L2:63   1st Qu.:0.9682   1st Qu.:0.7560
#  F      :12             3:36           Median :1.1354   Median :0.9045
#  G      :12                            Mean   :1.1225   Mean   :0.8881
#  I      :12                            3rd Qu.:1.2789   3rd Qu.:1.0176
#  B      :11                            Max.   :1.6347   Max.   :1.3882
#  (Other):52
#      AA.mm2       Cort.Cell.Num     XVA.mm2            Per.A
#  Min.   :0.0057   Min.   : 510   Min.   :0.00240   Min.   : 1.359
#  1st Qu.:0.1153   1st Qu.:1542   1st Qu.:0.04080   1st Qu.:15.428
#  Median :0.2073   Median :1817   Median :0.04960   Median :23.258
#  Mean   :0.2098   Mean   :1857   Mean   :0.05079   Mean   :22.886
#  3rd Qu.:0.2837   3rd Qu.:2136   3rd Qu.:0.06070   3rd Qu.:29.313
#  Max.   :0.5084   Max.   :3331   Max.   :0.08990   Max.   :46.262
#
#   CellSize.avg         m.Loc
#  Min.   :0.0000510   R1 L1:29
#  1st Qu.:0.0002739   R1 L2:31
#  Median :0.0003802   R2 L1:31
#  Mean   :0.0004251   R2 L2:32
#  3rd Qu.:0.0005346
#  Max.   :0.0012030
#
```

We're down to 12 variables, but this is probably too many for a pairs plot. We'll exclude the first 4 variables as they are factors.

```
pairs(anat[, -(1:4)])
```



That is a useful figure that permits us to quickly see how seven variables are related to each other. What would happen if we used `pairs(anat)`? For more variables than 7 or 8 the utility of such a figure probably declines. We can see that RXSA (cross section area) is tightly correlated with TCA (cortical area), and that AA (aerenchyma area) is correlated with Per.A (aerenchyma area as percent of cortex). These are not surprising, as they are mathematically related. XVA (xylem vessel area) does not seem to be strongly correlated with any other measure. Per.A (Cortical aerenchyma as percent of root cortex) is correlated with average cell size (which is interesting if you are a root biologist!). > Sometimes it is useful to enlarge a plot to see it better - especially if it is a complex one like a pairs plot. In RStudio, you can click the "zoom" button above the plots to enlarge a plot for viewing.

We can also use the function `by()` for multi-way data summaries over factor variables.

```
by(data = anat[, 7:10], INDICES = list(anat$media.rep, anat$Loc), FUN = colMeans)
```

```
# : R1
# : L1
#        AA.mm2 Cort.Cell.Num       XVA.mm2         Per.A
#  1.712069e-01  1.808207e+03  4.722414e-02  2.038428e+01
# ------------------------------------------------------------
# : R2
# : L1
#        AA.mm2 Cort.Cell.Num       XVA.mm2         Per.A
#  1.368774e-01  1.740452e+03  4.993548e-02  1.864896e+01
# ------------------------------------------------------------
# : R1
# : L2
```

```
#        AA.mm2 Cort.Cell.Num       XVA.mm2         Per.A
#  2.577355e-01  2.009871e+03  5.344839e-02  2.447344e+01
# ------------------------------------------------------------
# : R2
# : L2
#        AA.mm2 Cort.Cell.Num       XVA.mm2         Per.A
#  2.688344e-01  1.865281e+03  5.228438e-02  2.772134e+01
```

Note that if we are including more than one factor in the argument `INDICES` they must be in a `list()` - the syntax is similar to that for `aggregate()` that we saw in chapter 9.

IV. PCA

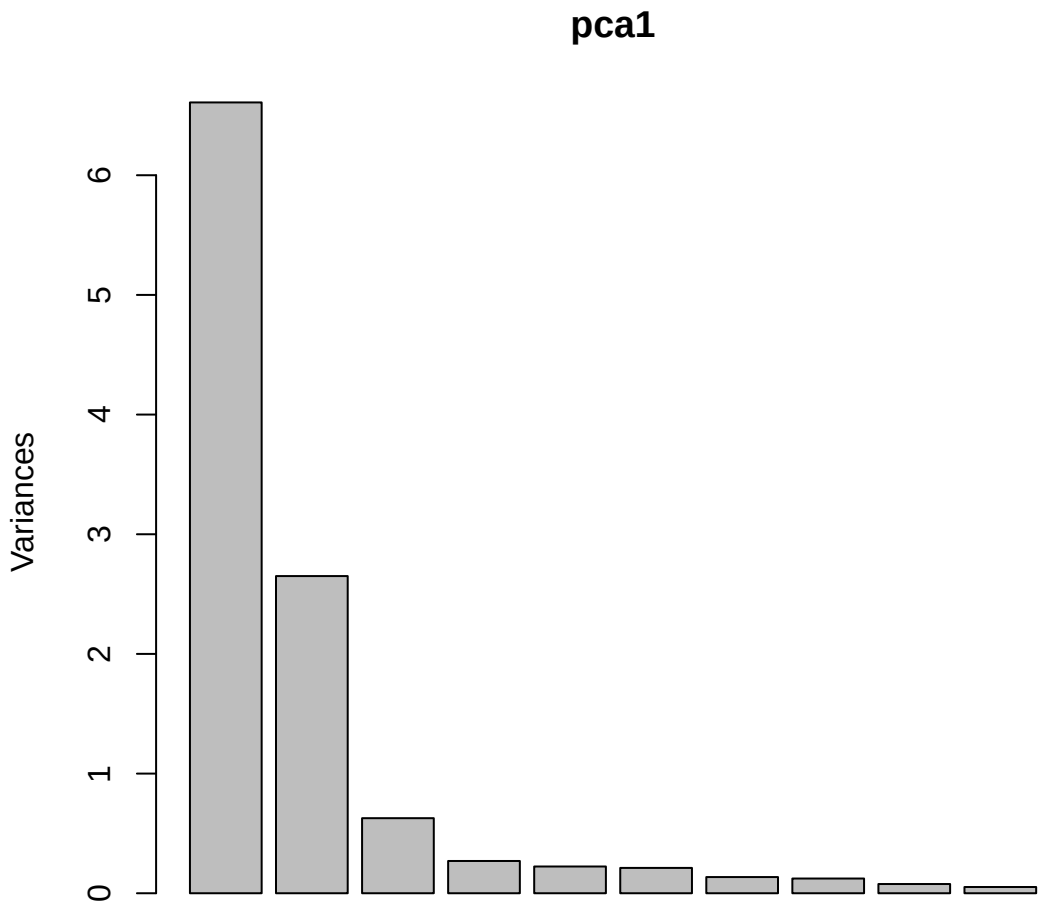
Since we live in a three dimensional world, our perceptual ability can generally cope with three dimensions, but we often have difficult time visualizing or understanding higher dimensional problems. Principal Components Analysis, or PCA, is a tool for reducing the dimensions of a data set.

In the most basic terms, PCA rotates the data cloud to produce new axes, or dimensions, that maximize the variability. These are the main axes of variation in the data, or the "Principal Components". They can be related to the original variables only by rotation. Here is an example with the `mtcars` data:

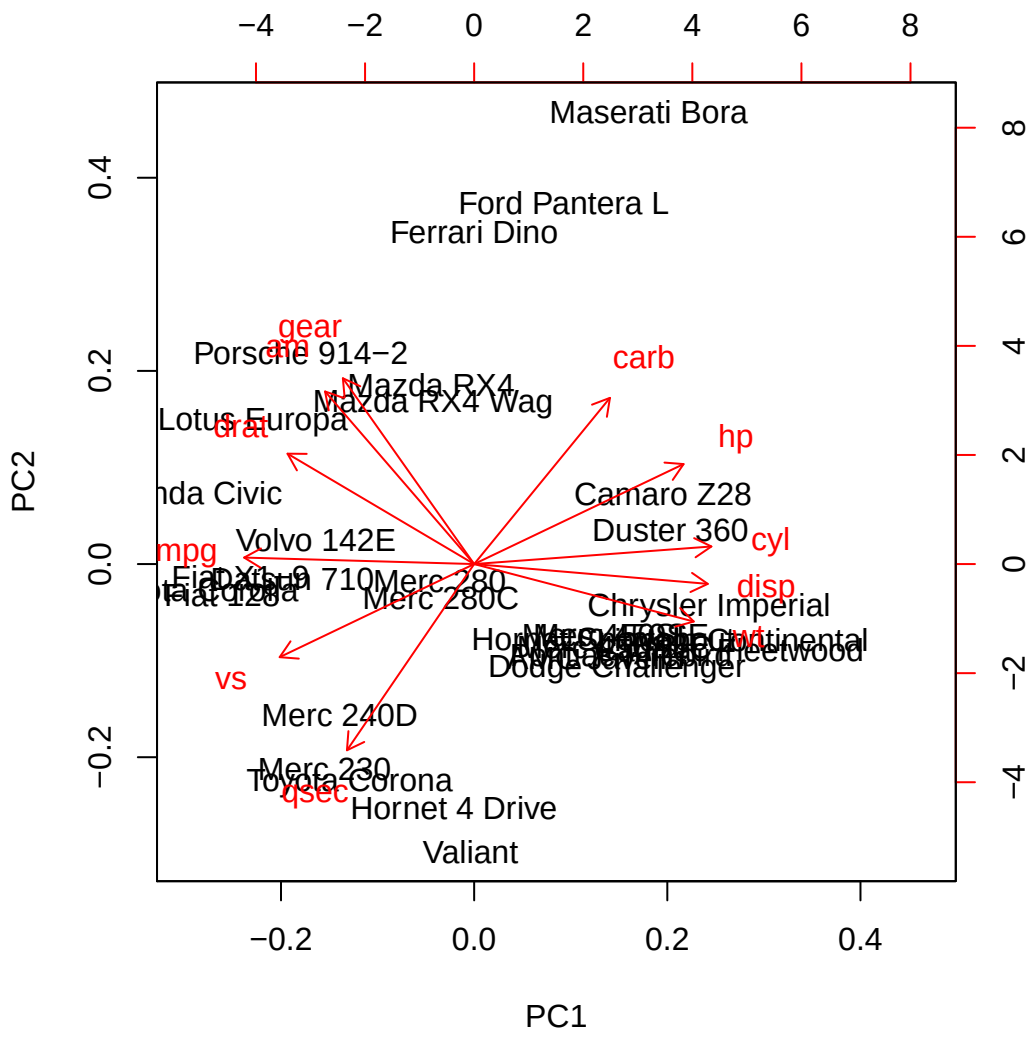
```
data(mtcars)
pca1 <- prcomp(mtcars, center = TRUE, scale. = TRUE)
summary(pca1)
```

```
# Importance of components%s:
#                           PC1    PC2     PC3     PC4     PC5     PC6
# Standard deviation     2.5707 1.6280 0.79196 0.51923 0.47271 0.46000
# Proportion of Variance 0.6008 0.2409 0.05702 0.02451 0.02031 0.01924
# Cumulative Proportion  0.6008 0.8417 0.89873 0.92324 0.94356 0.96279
#                           PC7     PC8    PC9    PC10   PC11
# Standard deviation     0.3678 0.35057 0.2776 0.22811 0.1485
# Proportion of Variance 0.0123 0.01117 0.0070 0.00473 0.0020
# Cumulative Proportion  0.9751 0.98626 0.9933 0.99800 1.0000
```

```
screeplot(pca1)
```




```
biplot(pca1)
```




```
pca1$rotation
```

```
#             PC1         PC2         PC3          PC4         PC5
# mpg  -0.3625305  0.01612440 -0.22574419 -0.022540255  0.10284468
# cyl   0.3739160  0.04374371 -0.17531118 -0.002591838  0.05848381
# disp  0.3681852 -0.04932413 -0.06148414  0.256607885  0.39399530
# hp    0.3300569  0.24878402  0.14001476 -0.067676157  0.54004744
# drat -0.2941514  0.27469408  0.16118879  0.854828743  0.07732727
# wt    0.3461033 -0.14303825  0.34181851  0.245899314 -0.07502912
# qsec -0.2004563 -0.46337482  0.40316904  0.068076532 -0.16466591
# vs   -0.3065113 -0.23164699  0.42881517 -0.214848616  0.59953955
# am   -0.2349429  0.42941765 -0.20576657 -0.030462908  0.08978128
# gear -0.2069162  0.46234863  0.28977993 -0.264690521  0.04832960
# carb  0.2140177  0.41357106  0.52854459 -0.126789179 -0.36131875
#              PC6          PC7          PC8          PC9         PC10
# mpg  -0.10879743  0.367723810 -0.754091423  0.235701617  0.13928524
# cyl   0.16855369  0.057277736 -0.230824925  0.054035270 -0.84641949
# disp -0.33616451  0.214303077  0.001142134  0.198427848  0.04937979
# hp    0.07143563 -0.001495989 -0.222358441 -0.575830072  0.24782351
# drat  0.24449705  0.021119857  0.032193501 -0.046901228 -0.10149369
# wt   -0.46493964 -0.020668302 -0.008571929  0.359498251  0.09439426
# qsec -0.33048032  0.050010522 -0.231840021 -0.528377185 -0.27067295
```

```
# vs    0.19401702 -0.265780836  0.025935128  0.358582624 -0.15903909
# am   -0.57081745 -0.587305101 -0.059746952 -0.047403982 -0.17778541
# gear -0.24356284  0.605097617  0.336150240 -0.001735039 -0.21382515
# carb  0.18352168 -0.174603192 -0.395629107  0.170640677  0.07225950
#              PC11
# mpg  -0.124895628
# cyl  -0.140695441
# disp  0.660606481
# hp   -0.256492062
# drat -0.039530246
# wt   -0.567448697
# qsec  0.181361780
# vs    0.008414634
# am    0.029823537
# gear -0.053507085
# carb  0.319594676
```

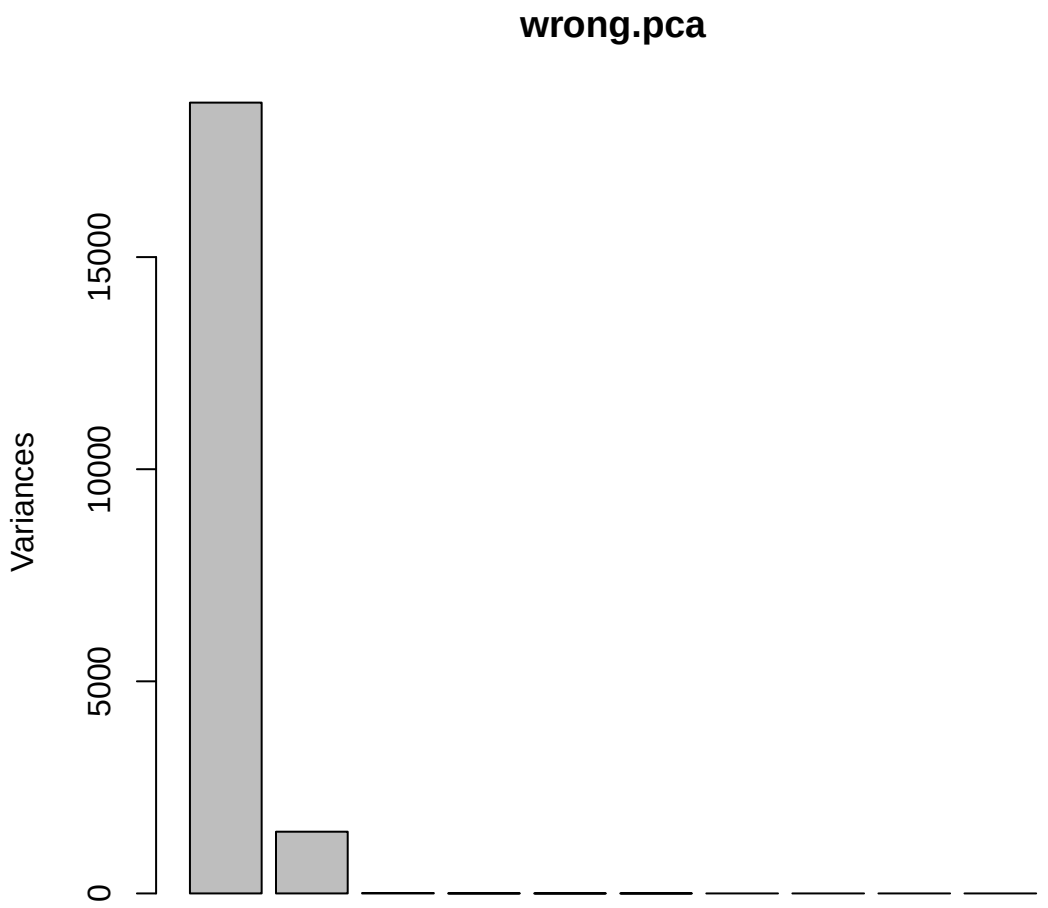
Notice:

1. `prcomp()` requires numeric variables only; we must exclude any non-numeric variables in the data.
2. Because the variables may be scaled differently, it is almost always necessary to subtract the mean and scale by the standard deviation, so all variables share the same scale, hence the arguments `center` and `scale.` For more evidence see the next code block.
3. `summary()` on an object of type `prcomp` gives us a brief description of the principal components. Here we see that there are 11 PCs - the full number of PCs is always the number of dimensions in the data, in this case 11.
4. The "reduction of dimensions" can be seen in the "Cumulative Proportion" row - the first three PCs explain about 90% of the variation in the data, so we can consider three rather than 11 variables (although those three represent a synthesis of all 11). The `screeplot()` is basically a barplot of the the proportion of variance explained. This data set is rather nicely behaved, the first 2 or 3 PCs capture most of the variation. This is not always the case.
5. The `rotation`, sometimes called "loadings" shows the contribution of each variable to each PC. Note that the signs (directions) are arbitrary here.
6. PCA is often visualized with a "biplot", in which the data points are visualized in a scatterplot of the first two PCs, and the original variables are also drawn in relation to the first two PCs. Here one can see that vehicle weight (`wt`) is rather well correlated with number of cylinders and displacement, but that fuel economy (`mpg`) in inversely correlated with number of cylinders and displacement. Variables that are orthogonal are not well correlated - forward gears (`gear`) and horsepower (`hp`). In this case the first 2 PCs explain about 85% of the variation in the data, so these relationships between variables are not absolute.

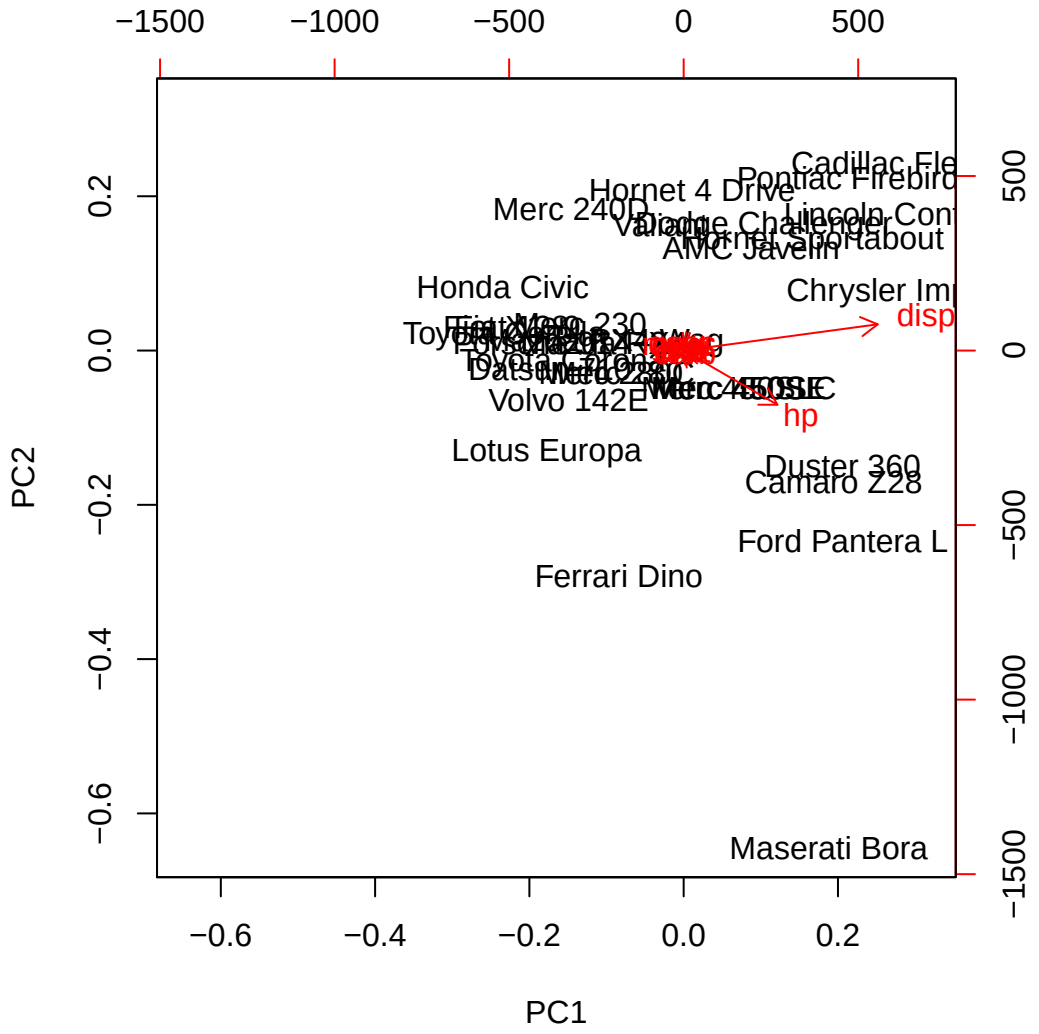
```
wrong.pca <- prcomp(mtcars)
summary(wrong.pca)
```

```
# Importance of components%s:
#                             PC1      PC2     PC3     PC4     PC5     PC6
# Standard deviation     136.533 38.14808 3.07102 1.30665 0.90649 0.66354
# Proportion of Variance   0.927  0.07237 0.00047 0.00008 0.00004 0.00002
# Cumulative Proportion    0.927  0.99937 0.99984 0.99992 0.99996 0.99998
#                            PC7   PC8    PC9   PC10   PC11
# Standard deviation     0.3086 0.286 0.2507 0.2107 0.1984
# Proportion of Variance 0.0000 0.000 0.0000 0.0000 0.0000
# Cumulative Proportion  1.0000 1.000 1.0000 1.0000 1.0000
```

```
screeplot(wrong.pca)
```




```
biplot(wrong.pca)
```



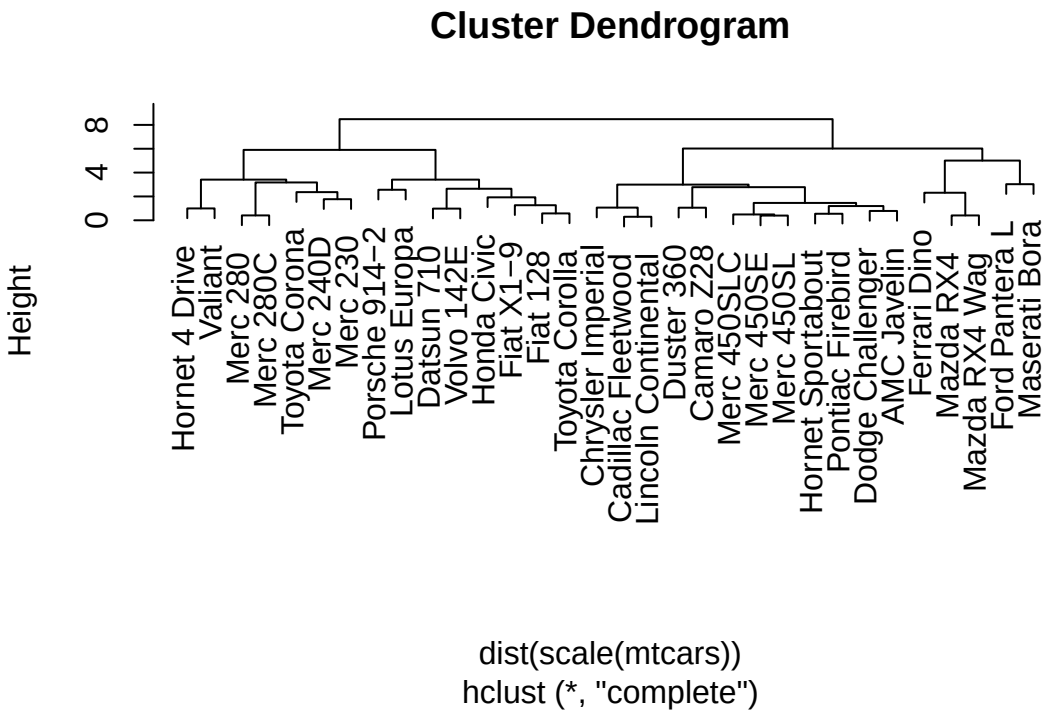
To highlight the importance of scaling, we demonstrate here the unscaled pca of the mtcars data. Note that for the *unscaled* `wrong.pca` the first 2 PCs explain nearly all the variance - this is simply because a small number (2) of variables are scaled much "larger" than the others and so contribute a much larger share of the variance. A look at `summary(mtcars)` should show you that re-scaling `mpg`, `disp`, and `hp` (simply dividing by 10, 100, and 100, respectively) would yield a different pca than the original scaled pca - I leave it for you to test.

V. Clustering

Another way to look at multivariate data is to ask how data points may be related to each other. A glance at the biplot above shows that there are some small clusters of data points. There are a number of ways to approach this question. Here we'll demonstrate hierarchical clustering. This begins by finding the points that are "closest" to each other. In R "Closeness" is actually calculated by the function `dist()` and can be defined various ways - the simplest is the euclidean distance, but there are other options also (see `?dist`). Data points are then grouped into clusters based on their "closeness" (or distance), and (as you might expect by now) there are different methods of grouping ("agglomeration") supported - see `?hclust` for more.

Now we can use `dist()` and `hclust()` to cluster the data. However, as we saw with PCA, differences in variable scaling can be very important. We can use the function `scale()` to scale and center the data before clustering. As with PCA, it is instructive to compare the this dendrogram with that derived from unscaled data (again, I leave this to the reader).

```r
plot(hclust(dist(scale(mtcars))))
```

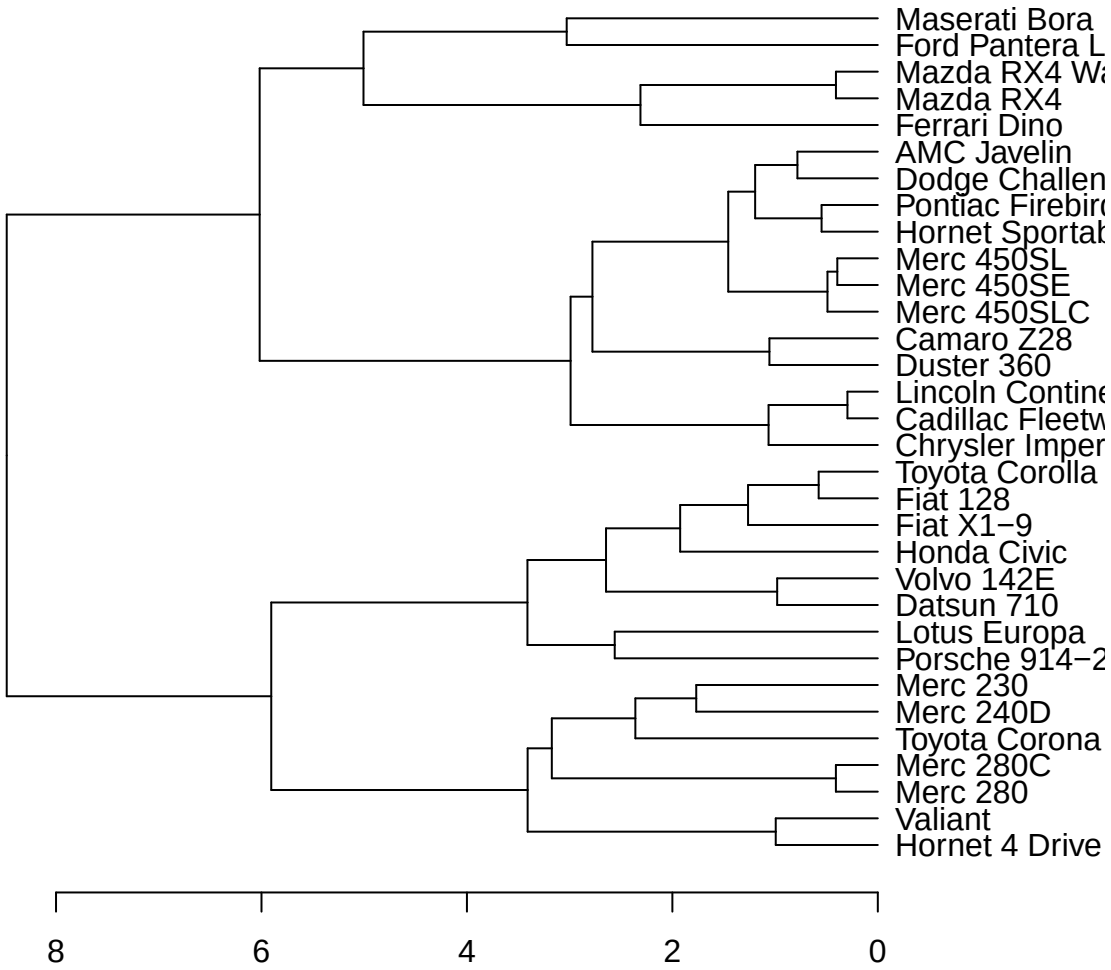


Note that the three Merc 450 models cluster together, and the Cadillac Fleetwood and Lincoln Continental also cluster. The dendrogram for the unscaled data makes less sense (though if you don't happen to remember cars from 1973 it is possible that *none* of these clusters look "right" to you).

EXTRA: Plotting a dendrogram horizontally.

If we want to view the dendrogram printed horizontally, we have to save our `hclust` object and create a `dendrogram` object from it. We'd also need to tweak the plot margins a bit, but it might make the dendrogram easier to read:

```r
hc <- hclust(dist(scale(mtcars)))
dendro <- as.dendrogram(hc)
par(mar = c(3, 0.5, 0.5, 5))
plot(dendro, horiz = TRUE)
```



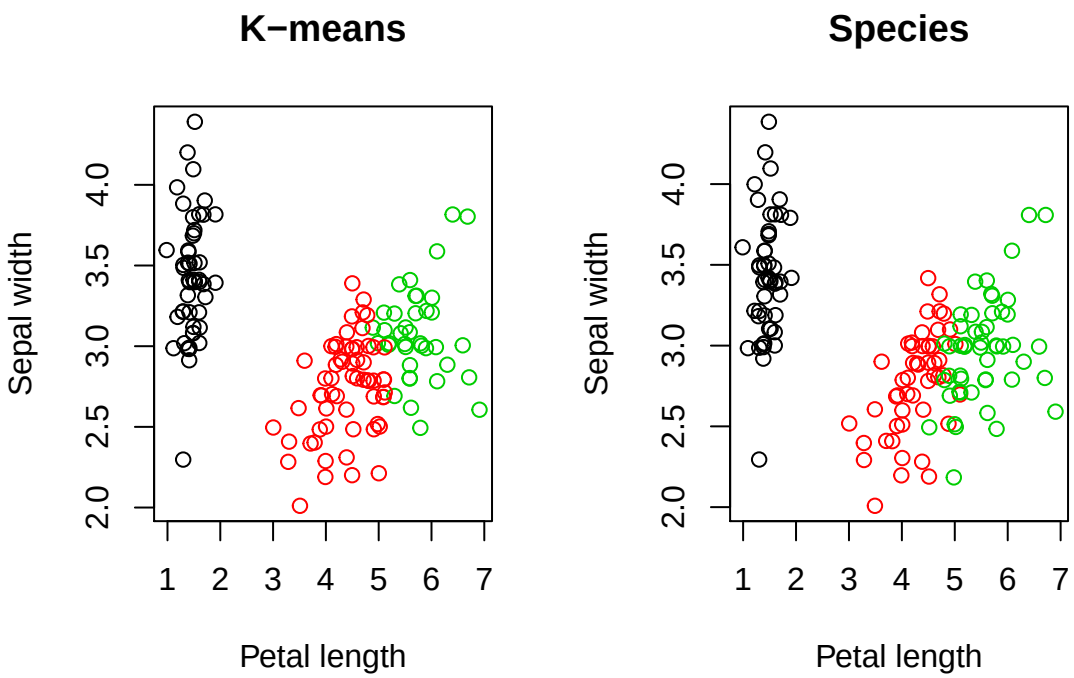
Another clustering tool is "k-means clustering". K-means clustering does require that we provide the number of clusters, and it is sensitive to the starting location of the clusters (which can be user specified or automatic). This means that K-means will not always return exactly the same clustering. Here we'll demonstrate using the `iris` data, which has floral dimensions for three species of iris. We'll specify three clusters to see how well we can separate the three species.

```
data(iris)
kcl <- kmeans(iris[, -5], 3, nstart = 3)
table(kcl$cluster, iris$Species)
```

```
#
#     setosa versicolor virginica
#   1     50          0         0
#   2      0         48        14
#   3      0          2        36
```

Based on the table, it seems like "setosa" can be reliably separated but "versicolor" and "virginica" are somewhat more difficult to separate, although even "virginica" is identified 72% of the time. Let's examine this data a bit more. Since the data is rounded to the nearest 0.1 cm, we'll add a bit of noise to the data, using `jitter()`

```
plot(jitter(iris$Petal.Length), jitter(iris$Sepal.Width), col = kcl$cluster,
    main = "K-means", xlab = "Petal length", ylab = "Sepal width")
plot(jitter(iris$Petal.Length), jitter(iris$Sepal.Width), col = iris$Species,
    main = "Species", xlab = "Petal length", ylab = "Sepal width")
```



Note that the colors in the above plots are essentially arbitrary. To better see where the `kmeans()` result doesn't match the original species, it is worth "zooming in" on the are where there is some confusion. Try running the above code and adding "`xlim=c(4,6),ylim=c(2.3,3.4)`" to both `plot()` calls. It is also worth repeating these plots without the use of `jitter()` to better appreciate how it helps in this case.

For more useful descriptions of functions, see the CRAN Task View on multivariate analysis.

VI. Exercises

1) The built in dataset `iris` has data on four floral measurements for three different species of iris. Make a pairs plot of the data. What relationships (correlations) look strongest?

2) The grouping evident with the `Species` variable in the last plot should make you curious. Add the argument `col=iris$Species` to the last plot you made. Does this change your conclusions about correlations between any of the relationships? Can you make a lattice plot (`xyplot()`) showing `Sepal.Length` as a function of `Sepal.Width` for the different species?

3) The built in data `state.x77` (which can be loaded via `data(state)`) has data for the 50 US states. Fit a principal components analysis to this data. What proportion of variation is explained by the first three principal components? What variable has the greatest (absolute value) loading value on each of the first three principal components? (*Note:* the dataset `state` is a list of datasets one of which is a matrix named `state.x77`)

4) The `state.x77` can also be approached using hierarchical clustering. Create a cluster dendrogram of the first 20 states (in alphabetical order, as presented in the data) using `hclust()`. (*Hint:* given the length of the names, it might be worth plotting the dendrogram horizontally). Do any clusters stand out as surprising?

Chapter 11: Linear Models I.

Linear regression

I. Introduction

Regression is one of the most basic but fundamental statistical tools. We have a *response* ("dependent") variable - y that we want to model, or *predict* as a function of the *predictor* ("independent") variable - x. (y~x in R-speak).

We assume a linear model of the form $y = B_0 + B_1 * x + e$ where B_0 is the intercept, B_1 is the slope, and e is the *error*. Mathematically we can estimate B_0 and B_1 from the data. Statistical inference requires that we assume that: *the errors are independent & normal with mean 0 and var* $= s^2$.

Notice that we **don't** have to assume that y or x are normally distributed, only that the **error** (or *residuals*) are normally distributed [31].

II. Violation of Assumptions and Transformation of Data

We'll begin with some data showing body and brain weights for 62 species of mammals (if you haven't installed MASS, do `install.packages("MASS")`). We'll load the data and fit a regression.

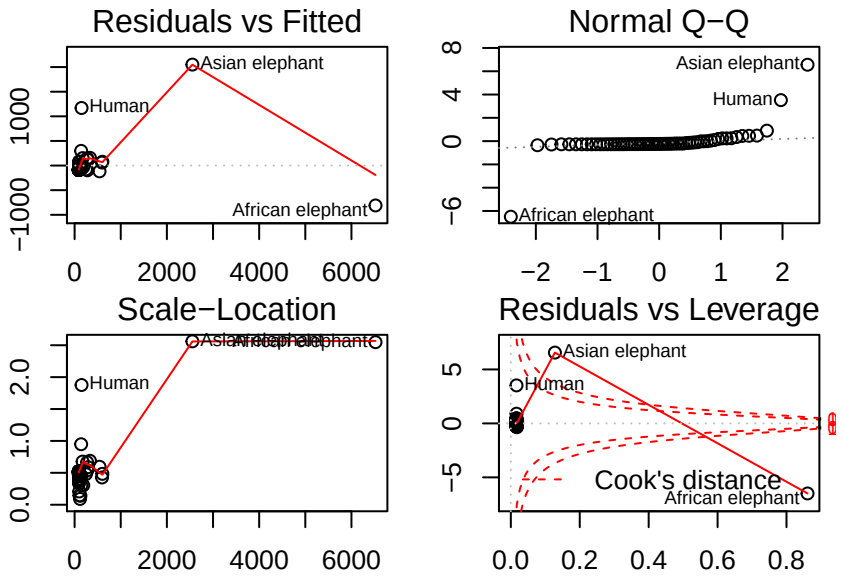
```
library(MASS)
data(mammals)
model1 <- lm(brain ~ body, data = mammals)
summary(model1)
```

```
#
# Call:
# lm(formula = brain ~ body, data = mammals)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -810.07  -88.52  -79.64  -13.02 2050.33
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept) 91.00440   43.55258    2.09   0.0409 *
# body         0.96650    0.04766   20.28   <2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 334.7 on 60 degrees of freedom
# Multiple R-squared:  0.8727,  Adjusted R-squared:  0.8705
# F-statistic: 411.2 on 1 and 60 DF,  p-value: < 2.2e-16
```

So far this looks pretty satisfactory - R^2 is , and the p-value is vanishingly small ($< 2.2*10^{-16}$). But let's have a look at the distribution of the residuals to see if we're violating any assumptions:

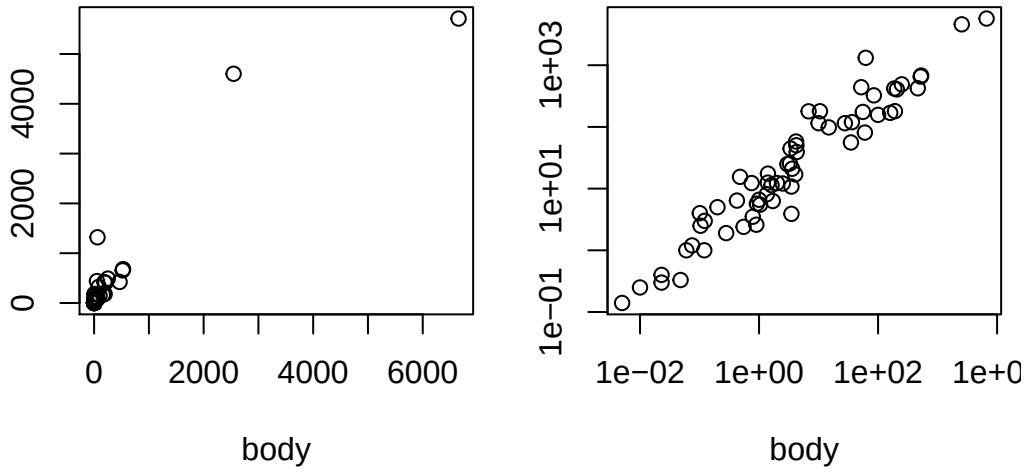
```
plot(model1)
```

[31] In practice, regression and ANOVA methods are fairly robust to some degree of non-normality in the residuals, but substantial non-random patterning in the residuals is cause for concern as it indicates either need for transformation of the response *or* an inadequate model (or both).



The diagnostic plots show some problems - the residuals don't seem to be normally distributed (Normal Q-Q plot shows large departure from linear). This suggests that transformation may be needed. Let's look at the data, before and after log transformation:

```
with(mammals, plot(body, brain))
with(mammals, plot(body, brain, log = "xy"))
```



Here it is clear that this data may need *log transformation* (as is often the case when the values occur over several orders of magnitude). The log/log plot of body and brain mass shows a much stronger linear association. Let's refit the model.

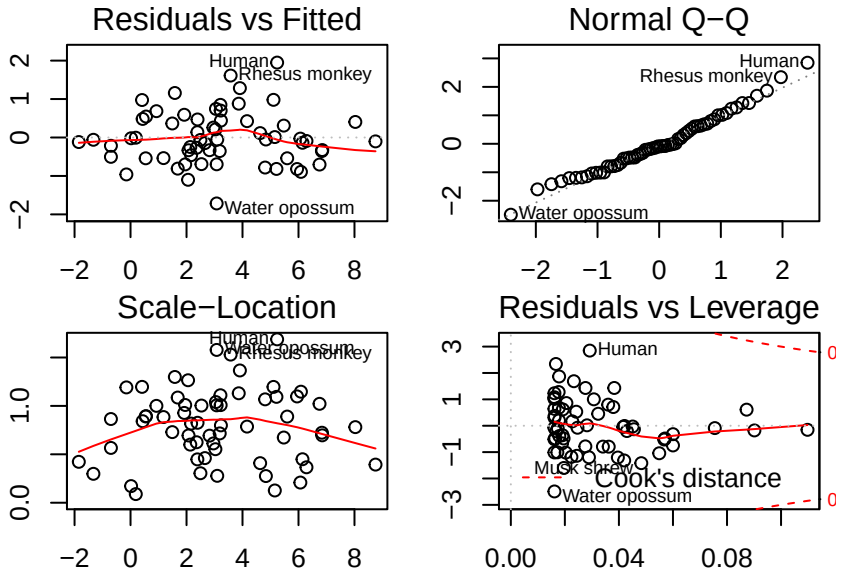
```
model2 <- lm(log(brain) ~ log(body), data = mammals)
summary(model2)
```

```
#
# Call:
# lm(formula = log(brain) ~ log(body), data = mammals)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -1.71550 -0.49228 -0.06162  0.43597  1.94829
#
```

```
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  2.13479    0.09604   22.23   <2e-16 ***
# log(body)    0.75169    0.02846   26.41   <2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.6943 on 60 degrees of freedom
# Multiple R-squared:  0.9208,  Adjusted R-squared:  0.9195
# F-statistic: 697.4 on 1 and 60 DF,  p-value: < 2.2e-16
```

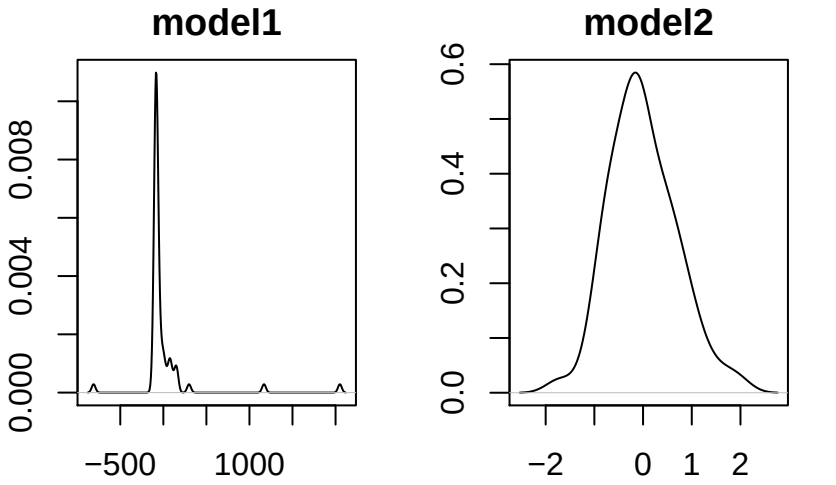
We see a bump in the R^2. The p-value isn't appreciably improved (but it was already *very* significant).

```
plot(model2)
```



This is much more satisfactory - the residuals are much nearer normal distribution than in the raw data. We can confirm this by inspecting the residuals. From the summary we can see that `log(brain)` is predicted to be near `2.13 + 0.752 * log(body)`, so each unit increase in `log(body)` yields an increase of `0.752` in `log(brain)`.

```
op <- par(mfrow = c(1, 2), mar = c(4, 3, 2, 1))
plot(density(model1$resid), main = "model1")
plot(density(model2$resid), main = "model2")
```



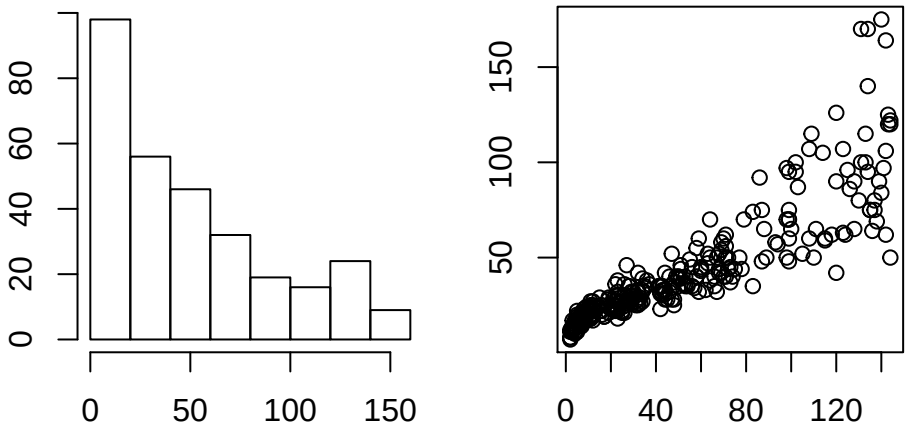

```
par(op)
```

Note that we *must refit the model* to check this - the residuals from the log transformed model *are not the same* as the log of the residuals! They can't be - the residuals are centered about zero (or should be), so there are many negative values, an the log of a negative number can't be computed.

Notice that `lm()` creates an "lm object" from which we can extract things, such as residuals. Try `coef(model1)` and `summary(model2)$coeff`.

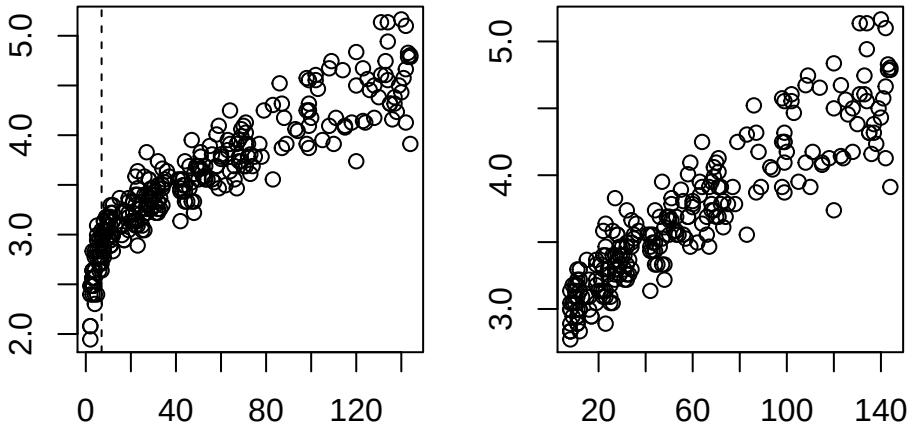
Another example of violation of assumptions. This is weight and age (in months) data for 5337 children <12 years old.

```
Weights <- read.csv("../Data/WeightData.csv", comm = "#")
hist(Weights$age, main = "")
plot(weight ~ age, data = Weights)
```



We can see that there appears to be increasing *variation in weight* as children age. This is pretty much what we'd expect - there is more space for variation (in absolute terms) in the weight of 12 year-olds than in 12 week-olds. However this suggests that for a valid regression model transformation of the weight might be needed.

```
op <- par(mfrow = c(1, 2), mar = c(2, 3, 1.5, 0.5))
plot(log(weight) ~ age, data = Weights)
abline(v = 7, lty = 2)
plot(log(weight) ~ age, data = subset(Weights, age > 7))
```

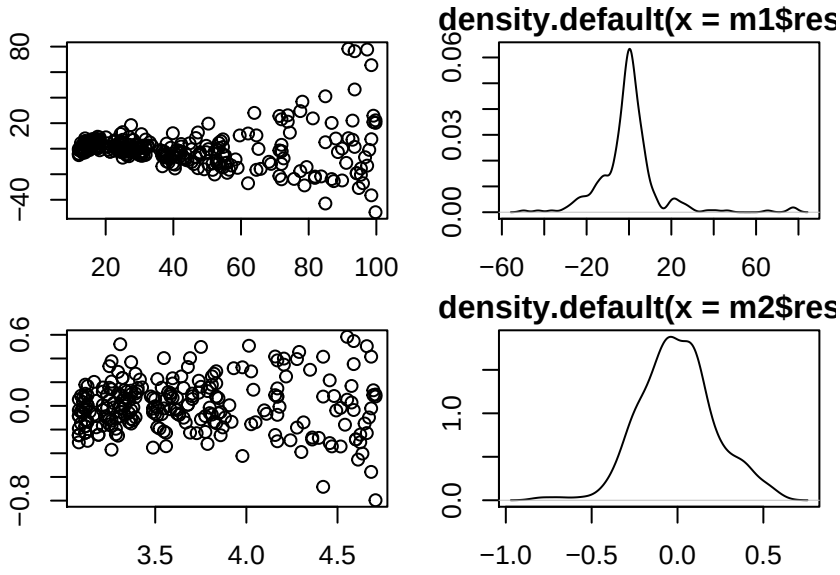


The variation appears more consistent in the first plot, apart from some reduced variation in weights (and a steeper slope) at the very low end of the age range (<7 months, indicated by the dashed line). In the second plot we've excluded this data. Let's fit a regression to both raw and transformed data and look at the residuals.

```
m1 <- lm(weight ~ age, data = Weights)
m2 <- lm(log(weight) ~ age, data = subset(Weights, age > 7))
```

You can use `plot(m1)` to look at the diagnostic plots. Let's compare the distribution of the residuals, though here we'll only look at the first two plots from each model.

```
op <- par(mfrow = c(2, 2), mar = c(2, 3, 1.5, 0.5))
plot(m1$fitted.values, m1$resid)
plot(density(m1$resid))
plot(m2$fitted.values, m2$resid)
plot(density(m2$resid))
```

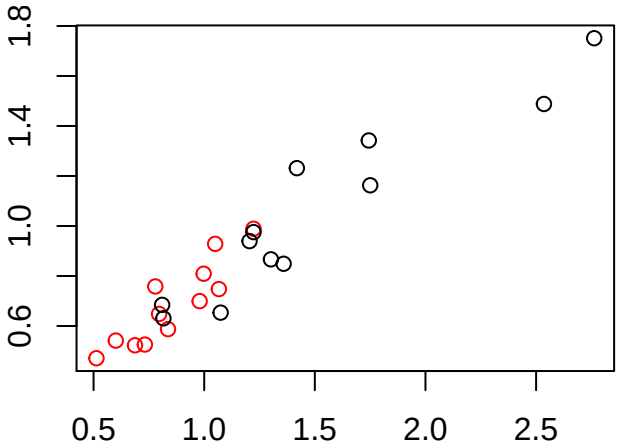



```
par(op)
```

Both of the plots of residuals vs fitted values and the kernel density estimates of the residuals show that the transformed data (with the <7 month subjects excluded) more nearly meets the regression assumptions, though there is still a bit of change in variation, they are certainly sufficiently close to normal and constant to meet regression assumptions.

III. Hypothesis Testing

```
beans <- read.csv("../Data/BeansData.csv", comm = "#")
plot(RtDM ~ ShtDM, data = beans, col = P.lev)
```



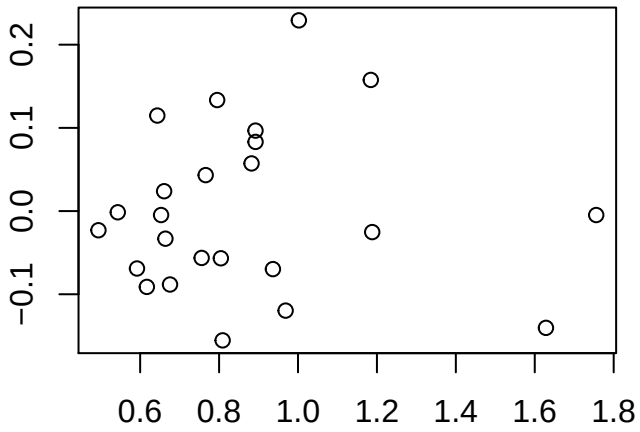
There was a strong correlation between Shoot and Root biomass [32], and it seems to apply to both the High P and Low P plants. Let's have a look at the regression fit to this data. Note that a factor (like `P.lev`) can be used directly to color points. They are colored with the colors from the color palette that correspond to the factor levels (see `palette()` to view or change the default color palette).

```
m1 <- lm(RtDM ~ ShtDM, data = beans)
summary(m1)
```

```
#
# Call:
# lm(formula = RtDM ~ ShtDM, data = beans)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -0.15540 -0.06922 -0.01391  0.06373  0.22922
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  0.20659    0.04892   4.223  0.00035 ***
# ShtDM        0.56075    0.03771  14.872 5.84e-13 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.1007 on 22 degrees of freedom
# Multiple R-squared:  0.9095,  Adjusted R-squared:  0.9054
# F-statistic: 221.2 on 1 and 22 DF,  p-value: 5.838e-13
```

[32] This data shows strong "root:shoot allometry" - the slope of log(Root)~log(Shoot) is constant across treatments, indicating that while a treatment might reduce overall *size*, it doesn't alter the fundamental growth pattern. See work by Karl Niklas and others on this topic.

```
plot(m1$resid ~ m1$fitted.values)
```



The residuals don't show any particular pattern, and they are approximately normal (see `plot(density(m1$resid))`). The summary shows an intercept of 0.2427 and slope (the `ShtDM` term) is 0.537. Standard errors are given for each parameter, as well as a t-value and a p-value (columns 3 and 4).

The t-value and p-values given by `summary(m1)`[33] are for the null hypothesis that B0 (or B1) is equal to 0. In this case the p-values are both very small, indicating a high degree of certainty that both parameters are *different* from 0 (note that this is a 2-sided test).

In some cases, we want to know more. For example, we might have the hypothesis that the *slope* of the root:shoot relationship should be 1. We can test this by calculating the t-value and p-value for this test. Recall that t is calculated as the difference between the observed and hypothesized values scaled by (divided by) the standard error.

```
B1 <- summary(m1)$coeff[2, 1:2]
t <- (B1[1] - 1)/B1[2]
t
```

```
#  Estimate
# -11.64947
```

For convenience we've stored the estimate and SE for B_1 (slope) as `B1` - we could just as well have used `t=(summary(m1)$coeff[2,1]-1)/summary(m1)$coeff[2,2]`. The t value is very large and negative (we can read this value as "B1 is 11.65 standard errors smaller than the hypothesized value"), so the slope estimate is smaller than the hypothesized value. If we want to get a p-value for this, we use the function `pt()`, which give probabilities for the t distribution.

```
pt(t, df = 22, lower.tail = TRUE)
```

```
#     Estimate
# 3.505015e-11
```

You can see we specify 22 degrees of freedom - this is the same as error degrees of freedom shown in `summary(m1)`. We specified `lower.tail=TRUE` because we have a large negative value of `t` and we want to know how likely a lower (more negative) value would be - in this case it is pretty unlikely! Since our hypothesis is a 2-sided hypothesis, we'll need to multiply the p-value by 2.

We can use the same approach to test hypothesized values of B_0 - but be sure to use the SE for B_0, since the estimates of SE are specific to the parameter. Note: Here is one way to check your work on a test like this: Calculate a new y-value that incorporates the hypothesis. For example - here our hypothesis is B_1=1, mathematically `1 * beans$ShtDM`. So if we *subtract* that from the root biomass, what we have left is *the difference between our hypothesized slope and the observed slope*. In this case we calculate it as `RtDM-1*ShtDM`.

[33] This applies to the t- and p-values shown in the summary for any lm object.

If the hypothesis was *correct* then the slope of a regression of this new value on the predictor would be zero. Have a look at `summary(lm(RtDM-1*ShtDM~ShtDM,data=beans))` - the slope will be quite negative, showing that our hypothesis is not true, and the p-value very low. Note though that this p-value is 2-sided, and in fact is twice the p-value we calculated above.

R actually has a built-in function to do such tests - `offset()` can be used to specify offsets within the formula in a call to `lm()`.

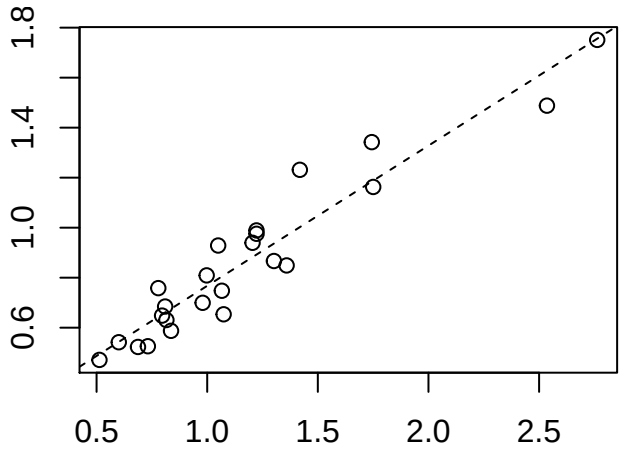
```
summary(lm(RtDM ~ ShtDM + offset(1 * ShtDM), data = beans))
```

```
#
# Call:
# lm(formula = RtDM ~ ShtDM + offset(1 * ShtDM), data = beans)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -0.15540 -0.06922 -0.01391  0.06373  0.22922
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  0.20659    0.04892   4.223  0.00035 ***
# ShtDM       -0.43925    0.03771 -11.649 7.01e-11 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.1007 on 22 degrees of freedom
# Multiple R-squared:  0.9095,  Adjusted R-squared:  0.9054
# F-statistic: 221.2 on 1 and 22 DF,  p-value: 5.838e-13
```

The syntax is a bit odd, since we have both `ShtDM` and `1*ShtDM` in our model, but this is how we specify a fixed hypothetical value.

It is useful to know that the line plotting function `abline()` can take an lm object as an argument (The argument `lty` specifies line type - 2 is a dotted line).

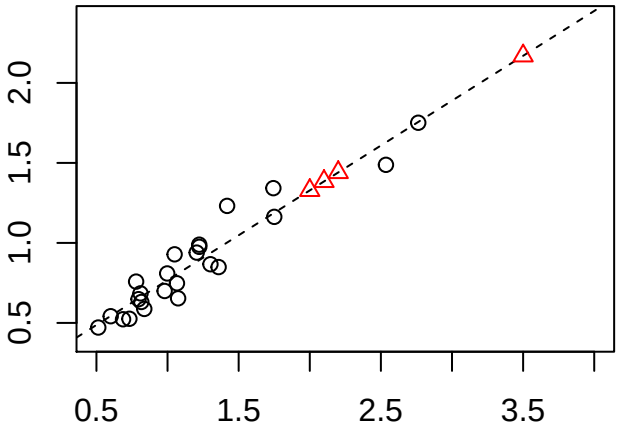
```
plot(RtDM ~ ShtDM, data = beans)
abline(m1, lty = 2)
```



IV. Predictions and Confidence Intervals from Regression Models

In some cases we want to use a regression model to predict values we haven't (or can't) measure, or we would like to know how confident we are about a regression line. The function `predict()` can make predictions from lm objects.

```
new.vals <- c(2, 2.1, 2.2, 3.5)
preds = predict(m1, newdata = data.frame(ShtDM = new.vals))
points(new.vals, preds, col = "red", pch = 24)
```



A key detail to notice: `predict()` requires the `newdata` as a data frame - this is to allow prediction from more complex models. The predicted values should (and do) fall right on the line. Also notice that the final value for which we wanted a prediction is well beyond the range of the data. This is not wise, and such predictions should not be trusted (but R will not warn you - always engage the brain when analyzing!).

We can think about two types of "confidence intervals" for regressions. The first can be thought of as describing the certainty about the location of the regression line (the average location of `y` given `x`). R can calculate this with the `predict()` function if we ask for the "confidence" interval.

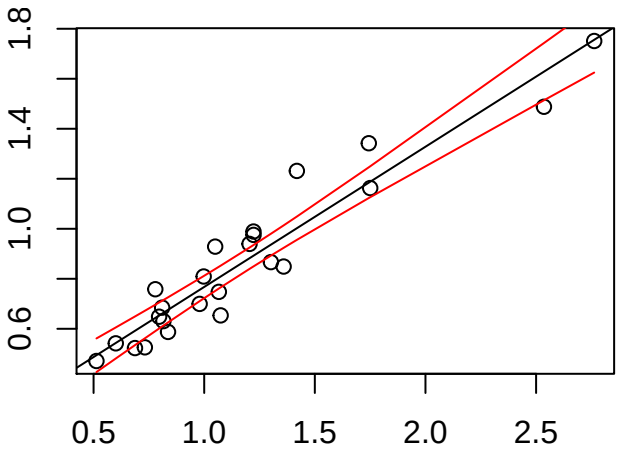
```
ci <- predict(m1, data.frame(ShtDM = sort(beans$ShtDM)), level = 0.95, interval = "confidence")
head(ci)
```

```
#         fit       lwr       upr
# 1 0.4942547 0.4270534 0.5614560
# 2 0.5432641 0.4811956 0.6053326
# 3 0.5919371 0.5346135 0.6492607
# 4 0.6169465 0.5618943 0.6719988
# 5 0.6434139 0.5906208 0.6962070
# 6 0.6530588 0.6010482 0.7050694
```

Notice there are three values - the "fitted" value, and the lower and upper CI. Also notice that we can specify a confidence level, and that we used our predictor variable (`beans$ShtDM`) as the `new` data, but *we used `sort()` on it* - this is to aid in plotting the interval.

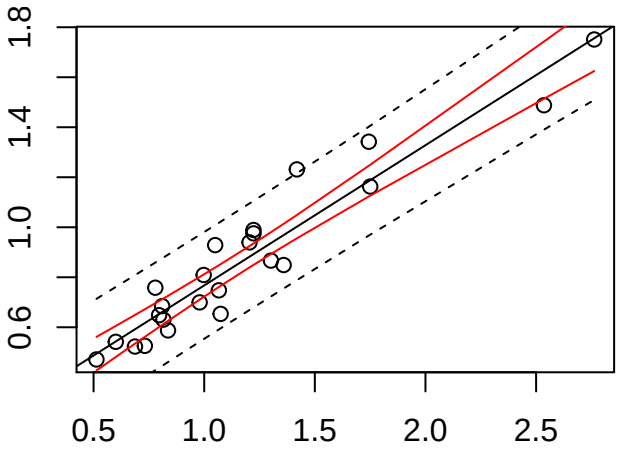
We can plot this interval on our scatterplot using the function `lines()`. Since we've sorted the data in `ci` (y-axis), we need to sort the x-axis values also. (Note: `sort()` also removes any `NA` values.)

```
plot(RtDM ~ ShtDM, data = beans)
abline(m1, lty = 1)
lines(sort(beans$ShtDM), ci[, 2], col = "red")
lines(sort(beans$ShtDM), ci[, 3], col = "red")
```



It shouldn't surprise us that the confidence interval here is quite narrow - the p-values for B0 and B1 are very small. Notice that a fair number of the data points are outside the bands. This is because this "confidence interval" applies *to the regression line as a whole*. If we want to predict individual values, the uncertainty is a bit greater - this is called a "prediction interval".

```
lines(sort(beans$ShtDM), ci[, 2], col = "red")
lines(sort(beans$ShtDM), ci[, 3], col = "red")
pri <- predict(m1, data.frame(ShtDM = sort(beans$ShtDM)), level = 0.95, interval = "prediction")
lines(sort(beans$ShtDM), pri[, 2], lty = 2)
lines(sort(beans$ShtDM), pri[, 3], lty = 2)
```



When we plot this notice that ~95% of our data points are within this interval - this is consistent with the meaning of this interval.

VI. Exercises

1) For the `beans` data test how effective root biomass (`RtDM`) is as a predictor of root length (`rt.len`).

2) For the `beans` data, test the hypothesis that the slope of the relationship of root biomass ~ shoot biomass ($B1$) is 0.5.

3) We worked with the dataset `mammals` earlier in this chapter, and concluded that it needed to be log-transformed to meet regression assumptions. Use `predict()` to calculate the confidence interval and regression line for this regression and plot it on both the log/log plot and on the un-transformed data (this will require that you back-transform the coordinates for the line and confidence intervals).

Chapter 12: Linear Models II.

ANOVA

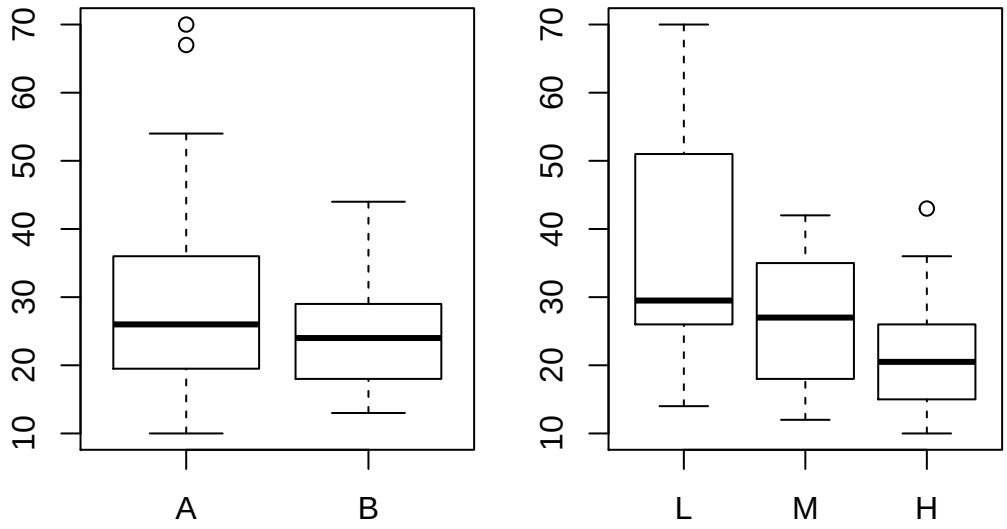
I. Introduction

It is often the case that we have one or more factors that we'd like to use to model some response. This is where we turn to Analysis of Variance, or ANOVA. Remember that, despite the fancy name, one-way ANOVA is basically just another form of regression - the continuous predictor variable is replaced by a factor. Since this is the case, it should not be surprising that the function `lm()` can be used for this type of analysis also.

II. One-way ANOVA

To explore this type of model we'll load some data on how the type of wool and the loom tension affects the number of breaks in wool yarn being woven.

```
data(warpbreaks)
boxplot(breaks ~ wool, data = warpbreaks)
boxplot(breaks ~ tension, data = warpbreaks)
```



Do the two types of wool have differing average numbers of breaks? The boxplot does not suggest much difference. One way to check would be to use a two-sample t-test.

```
t.test(breaks ~ wool, data = warpbreaks)
```

```
#
#   Welch Two Sample t-test
#
# data:  breaks by wool
# t = 1.6335, df = 42.006, p-value = 0.1098
# alternative hypothesis: true difference in means is not equal to 0
# 95 percent confidence interval:
#  -1.360096 12.915652
# sample estimates:
# mean in group A mean in group B
#        31.03704        25.25926
```

Of course, as we saw in Lesson 3, if we have more than two groups, we need something different.

```
oneway.test(breaks ~ tension, data = warpbreaks)
```

```
#
#   One-way analysis of means (not assuming equal variances)
#
# data:  breaks and tension
# F = 5.8018, num df = 2.00, denom df = 32.32, p-value = 0.007032
```

This strongly suggests that `tension` affects `breaks`. We can also use `lm()` to fit a linear model.

```
summary(lm(breaks ~ tension, data = warpbreaks))
```

```
#
# Call:
# lm(formula = breaks ~ tension, data = warpbreaks)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -22.389  -8.139  -2.667   6.333  33.611
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)    36.39       2.80  12.995  < 2e-16 ***
# tensionM      -10.00       3.96  -2.525 0.014717 *
# tensionH      -14.72       3.96  -3.718 0.000501 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 11.88 on 51 degrees of freedom
# Multiple R-squared:  0.2203,  Adjusted R-squared:  0.1898
# F-statistic: 7.206 on 2 and 51 DF,  p-value: 0.001753
```

The coefficients shown in the output from summary begin with the "Intercept". This is the mean for the first level of the factor variable [34] - in this case `tension="L"`. The coefficient given for the next level is for the difference between the second and the first level, and that for the third is for the difference between first and third.

We can use the function `anova()` on an lm object to see an analysis of variance table for the model.

```
anova(lm(breaks ~ tension, data = warpbreaks))
```

```
# Analysis of Variance Table
#
# Response: breaks
#           Df Sum Sq Mean Sq F value   Pr(>F)
# tension    2 2034.3 1017.13  7.2061 0.001753 **
# Residuals 51 7198.6  141.15
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This shows very strong support (p=0.0018) for a significant effect of `tension` on `breaks`. Note that `summary(aov(breaks~tension,data=warpbreaks))`[35] will give the same result with slightly different format.

[34] By default factor levels are assigned by alpha-numeric order, so the default order for levels "H", "M", and "L" would be "H=1; L=2, M=3". This doesn't make sense in this case (though it wouldn't change the estimates of group means or differences between them). We saw how to fix this in Chapter 7.

[35] The syntax is potentially confusing – unfortunately `anova(lm(y~x, data=df))`, `summary(lm(y~x, data=df))`, and `aov(y~x,data=df)` are so confusingly similar.

If we want to test all the differences between groups, we can use `TukeyHSD()` to do so - but we must use it on an an object created by `aov()`, it won't work with an `lm()` object.

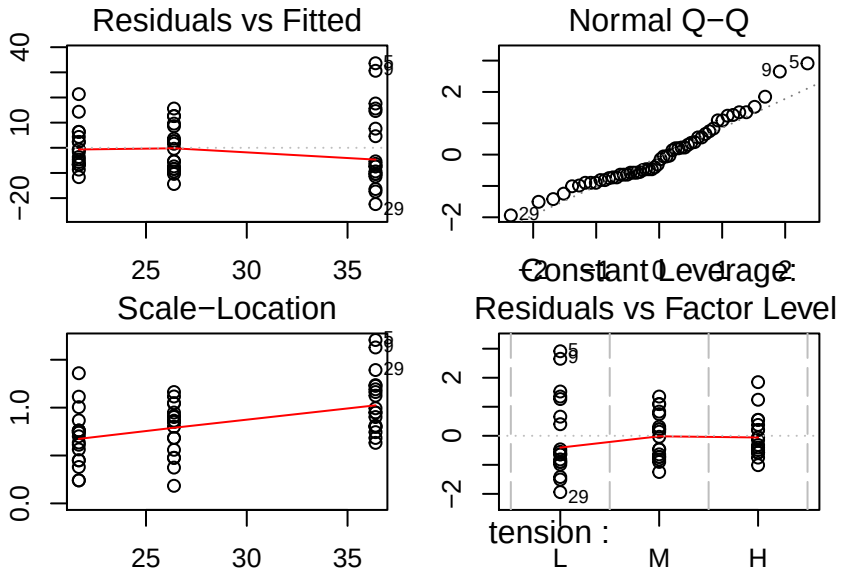
```
TukeyHSD(aov(breaks ~ tension, data = warpbreaks))
```

```
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
#
# Fit: aov(formula = breaks ~ tension, data = warpbreaks)
#
# $tension
#           diff       lwr        upr     p adj
# M-L -10.000000 -19.55982 -0.4401756 0.0384598
# H-L -14.722222 -24.28205 -5.1623978 0.0014315
# H-M  -4.722222 -14.28205  4.8376022 0.4630831
```

This lists all the pairwise differences between groups showing the estimated difference between groups, the lower and upper confidence limits for the difference, and the p-value for the difference - a significant p-value means the difference is real. We can see that both `H` and `M` are different from `L`, but not from each other.

Of course, just as in regression it is good practice to check our diagnostic plots for violations of assumptions.

```
plot(lm(breaks ~ tension, data = warpbreaks))
```



There is evidence of modest difference in variance between groups (though not enough to cause concern), and the normal QQ plot shows that the residuals are near enough to normal.

III. For violations of assumptions.

For a one-way ANOVA (i.e. a single factor) where assumptions are violated, we do have a few options. The function `oneway.test()` we used above does not assume equal variance, so it can be used with unequal variance. If the residuals are strongly non-normal, the Kruskal-Wallace test is a non-parametric alternative.

```
kruskal.test(breaks ~ wool, data = warpbreaks)
```

```
#
#   Kruskal-Wallis rank sum test
#
```

```
# data:  breaks by wool
# Kruskal-Wallis chi-squared = 1.3261, df = 1, p-value = 0.2495
```

Another option is transformation of the data. However (as we'll see) a common cause of violation of regression assumptions is that there are sources of variation not included in the model. One of the brilliant features of the linear model is that it can accommodate multiple predictors, and the inclusion of the right predictors sometimes allows the regression assumptions to be met [36].

IV. Multi-Way ANOVA - Understanding `summary(lm())`

Particularly in designed experiments we often do have more than one factor that we need to include in our model. For example in the `warpbreaks` data we looked at both `tension` and `wool` separately, but we might need to *combine* them to understand what is going on. The *formula interface* lets us tell R *how* to use multiple predictors.

R formula	Y as a function of:
`Y ~ X1`	X1
`Y ~ X1 + X2`	X1 and X2
`Y ~ X1 * X2`	X1 and X2 and the X1xX2 interaction
`Y ~ X1 + X2 + X1:X2`	Same as above, but the interaction is explicit
`(Y ~ (X1 + X2 +X3)^2)`	X1, X2, and X3, with only 2-way interactions
`Y ~ X1 +I(X1^2)`	X1 and X1 squared (use `I()`for a literal power)
`Y ~ X1|X2`	X1 for each level of X2

```
summary(lm(breaks ~ wool + tension, data = warpbreaks))
```

```
#
# Call:
# lm(formula = breaks ~ wool + tension, data = warpbreaks)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -19.500  -8.083  -2.139   6.472  30.722
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   39.278      3.162  12.423  < 2e-16 ***
# woolB         -5.778      3.162  -1.827 0.073614 .
# tensionM     -10.000      3.872  -2.582 0.012787 *
# tensionH     -14.722      3.872  -3.802 0.000391 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 11.62 on 50 degrees of freedom
# Multiple R-squared:  0.2691,  Adjusted R-squared:  0.2253
# F-statistic: 6.138 on 3 and 50 DF,  p-value: 0.00123
```

The model coefficients here are understood nearly as before - the intercept now is the first level of *each* factor (e.g. `wool=A` & `tension=L`). The `woolB` estimate is the difference between `wool=A` and `wool=B`. Because we have not included the `wool x tension` interaction here, we assume that the influence of `wool` is the same for all levels of `tension`. As before we can use `anova()` to see an ANOVA table showing the estimate of the effect for each factor, and `TukeyHSD()` on an `aov()` fit to test group-wise differences.

[36] Of course, this assumes that you knew or guessed what the "right" predictors might be and measured them.

```r
anova(lm(breaks ~ wool + tension, data = warpbreaks))
```

```
# Analysis of Variance Table
#
# Response: breaks
#           Df Sum Sq Mean Sq F value   Pr(>F)
# wool       1  450.7  450.67  3.3393 0.073614 .
# tension    2 2034.3 1017.13  7.5367 0.001378 **
# Residuals 50 6747.9  134.96
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
TukeyHSD(aov(breaks ~ wool + tension, data = warpbreaks))
```

```
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
#
# Fit: aov(formula = breaks ~ wool + tension, data = warpbreaks)
#
# $wool
#          diff       lwr       upr     p adj
# B-A -5.777778 -12.12841 0.5728505 0.0736137
#
# $tension
#           diff       lwr        upr     p adj
# M-L -10.000000 -19.35342 -0.6465793 0.0336262
# H-L -14.722222 -24.07564 -5.3688015 0.0011218
# H-M  -4.722222 -14.07564  4.6311985 0.4474210
```

Notice that the p-value for the ANOVA and the Tukey comparisons are the same for the factor `wool` but not for `tension` - that is because there are only two levels of `wool` but 3 levels of `tension`.

With more than one factor we also need to think about *interactions* between them, and what they mean. I this case we can understand the interaction as asking:
Does changing the tension have the same effect on breaks for both wool A and wool B?

```r
summary(lm(breaks ~ wool + tension + wool:tension, data = warpbreaks))
```

```
#
# Call:
# lm(formula = breaks ~ wool + tension + wool:tension, data = warpbreaks)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -19.5556  -6.8889  -0.6667   7.1944  25.4444
#
# Coefficients:
#                Estimate Std. Error t value Pr(>|t|)
# (Intercept)      44.556      3.647  12.218 2.43e-16 ***
# woolB           -16.333      5.157  -3.167 0.002677 **
# tensionM        -20.556      5.157  -3.986 0.000228 ***
# tensionH        -20.000      5.157  -3.878 0.000320 ***
# woolB:tensionM   21.111      7.294   2.895 0.005698 **
# woolB:tensionH   10.556      7.294   1.447 0.154327
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
#
# Residual standard error: 10.94 on 48 degrees of freedom
# Multiple R-squared:  0.3778,  Adjusted R-squared:  0.3129
# F-statistic: 5.828 on 5 and 48 DF,  p-value: 0.0002772
```

The output of `summary()` is similar to the above. The `woolB` estimate now is only for `tension=L`, since the last two estimates show the effect of wool on breaks for the other tensions. Since this estimate is negative for `tension=L` and positive for the higher levels of `tension` this is likely to be a significant difference.

V. Multi-Way ANOVA - Calculating group means

If we wanted to calculate the means for each for the groups we could do so by adding coefficients together - for example the estimate for wool=B; tension=H would be

`Intercept (wool=A;tension=L)+woolB+tensionH+woolB:tensionH`

It is not too complicated to calculate this from the coefficients.

```
m1 <- lm(breaks ~ wool * tension, data = warpbreaks)
sum(summary(m1)$coeff[c(1, 2, 4, 6), 1])
```

```
# [1] 18.77778
```

But it would be tedious (and error-prone) to do this for all factor levels. Fortunately we don't need to - we can use the function `predict()` to do it for us. We just need to give it all the possible factor levels as `"newdata"`. We can use the function `unique()` to give us the unique combinations of factor levels:

```
unique(warpbreaks[, 2:3])
```

```
#    wool tension
# 1     A       L
# 10    A       M
# 19    A       H
# 28    B       L
# 37    B       M
# 46    B       H
```

We can then use these as the `"newdata"` argument to `predict()` to get our predicted values. First we'll create the lm object, and then we'll create an object for this `"newdata"`, and then we'll calculate the predicted values.

```
m1 <- lm(breaks ~ wool + tension + wool:tension, data = warpbreaks)
m1.pv = unique(warpbreaks[, 2:3])
m1.pv$predicted = predict(m1, newdata = unique(warpbreaks[, 2:3]))
m1.pv
```

```
#    wool tension predicted
# 1     A       L  44.55556
# 10    A       M  24.00000
# 19    A       H  24.55556
# 28    B       L  28.22222
# 37    B       M  28.77778
# 46    B       H  18.77778
```

In some cases [37] we could also get these means is by using `tapply()` to apply the function `mean()`.

[37] Using `tapply()` in this way will only produce the same values as `predict()` for a *saturated* model, i.e. one that contains *all factors and interactions*! Also note that if there were any NA values in the data they would propagate as we haven't added `na.rm=TRUE`.

```
tapply(m1$fitted, list(warpbreaks$wool, warpbreaks$tension), mean)
```

```
#          L        M        H
# A 44.55556 24.00000 24.55556
# B 28.22222 28.77778 18.77778
```

VI. Multi-Way ANOVA - Getting a handle on interactions

It can be hard (or nearly impossible) to understand what an interaction really means. In this example it means that the effect of changing tension of rate of breakage differs for the two types of wool.

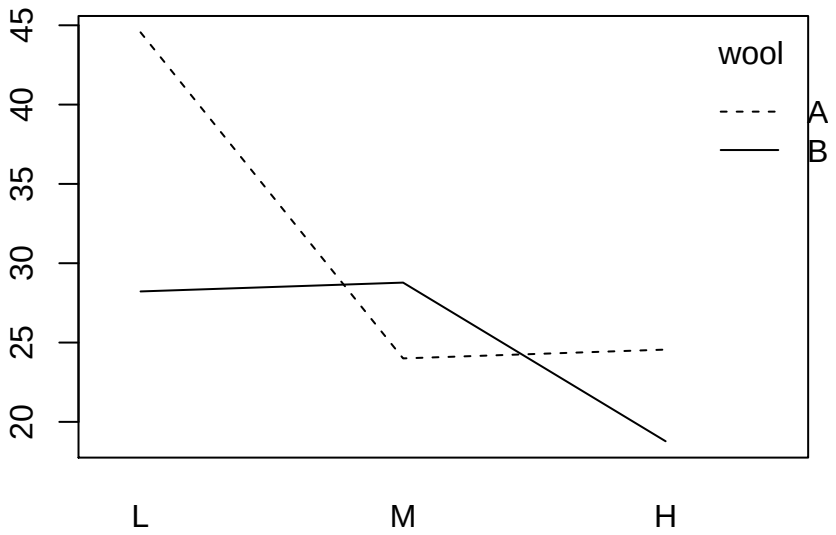
```
anova(lm(breaks ~ wool + tension + wool:tension, data = warpbreaks))
```

```
# Analysis of Variance Table
#
# Response: breaks
#              Df Sum Sq Mean Sq F value    Pr(>F)
# wool          1  450.7  450.67  3.7653 0.0582130 .
# tension       2 2034.3 1017.13  8.4980 0.0006926 ***
# wool:tension  2 1002.8  501.39  4.1891 0.0210442 *
# Residuals    48 5745.1  119.69
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Notice that here adding the interaction moved the effect of `wool` from p=0.074 to p=0.058. The interaction was strong enough that ignoring it increased the size of the residuals, and so reduced the magnitude of the F value.

Often it is helpful to visualize interactions to better understand them, and the function `interaction.plot()` gives us a quick way to do this.

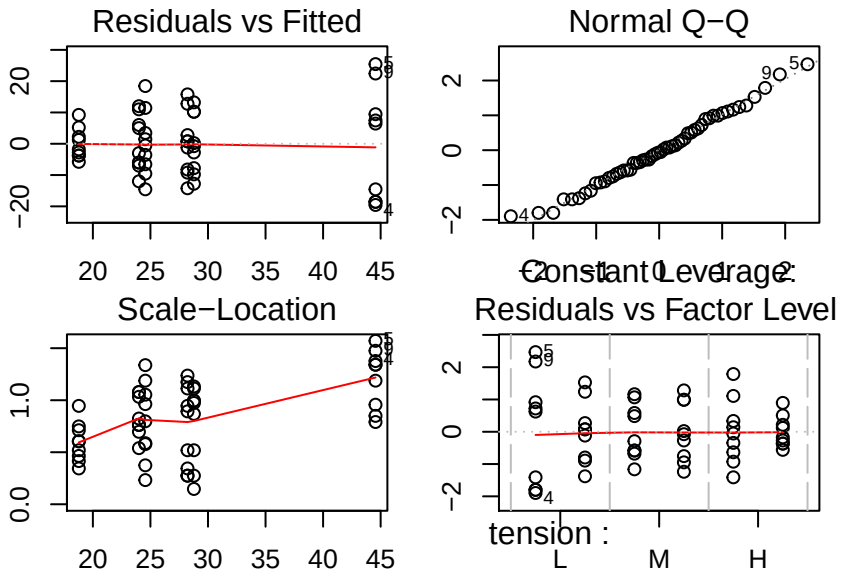
```
with(warpbreaks, interaction.plot(x.factor = tension, wool, response = breaks))
```



The syntax here is a bit different: the first and third arguments are the x and y axes, and the second is the grouping factor (`trace.factor`). This clearly and quickly shows us that the biggest difference between wools is at low tension.[38]

[38] A more complete interpretation is that increasing tension reduces the number of breaks, but this occurs at a lower tension for wool A than it does for wool B.

```
plot(lm(breaks ~ tension * wool, data = warpbreaks))
```



A check the diagnostic plots shows minimal evidence of unequal variance ("heteroscedasticity") or departure from normal distribution. As hinted above note how improving the model by accounting for more sources of variation (adding the second factor and the interaction) improved the agreement with regression assumptions.

VII. Multi-Way ANOVA - Tukey HSD and family-wise error

When we introduce an interaction we now have many more groups. In this example we have 6 groups (2 wools * 3 tensions). This gives many more pairwise comparisons.

```
TukeyHSD(aov(breaks ~ wool + tension + wool:tension, data = warpbreaks))
```

```
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
#
# Fit: aov(formula = breaks ~ wool + tension + wool:tension, data = warpbreaks)
#
# $wool
#          diff       lwr       upr    p adj
# B-A -5.777778 -11.76458 0.2090243 0.058213
#
# $tension
#           diff       lwr       upr     p adj
# M-L -10.000000 -18.81965 -1.180353 0.0228554
# H-L -14.722222 -23.54187 -5.902575 0.0005595
# H-M  -4.722222 -13.54187  4.097425 0.4049442
#
# $`wool:tension`
#                 diff       lwr        upr     p adj
# B:L-A:L -16.3333333 -31.63966  -1.027012 0.0302143
# A:M-A:L -20.5555556 -35.86188  -5.249234 0.0029580
# B:M-A:L -15.7777778 -31.08410  -0.471456 0.0398172
# A:H-A:L -20.0000000 -35.30632  -4.693678 0.0040955
# B:H-A:L -25.7777778 -41.08410 -10.471456 0.0001136
```

```
# A:M-B:L  -4.2222222 -19.52854  11.084100 0.9626541
# B:M-B:L   0.5555556 -14.75077  15.861877 0.9999978
# A:H-B:L  -3.6666667 -18.97299  11.639655 0.9797123
# B:H-B:L  -9.4444444 -24.75077   5.861877 0.4560950
# B:M-A:M   4.7777778 -10.52854  20.084100 0.9377205
# A:H-A:M   0.5555556 -14.75077  15.861877 0.9999978
# B:H-A:M  -5.2222222 -20.52854  10.084100 0.9114780
# A:H-B:M  -4.2222222 -19.52854  11.084100 0.9626541
# B:H-B:M -10.0000000 -25.30632   5.306322 0.3918767
# B:H-A:H  -5.7777778 -21.08410   9.528544 0.8705572
```

Tukey's HSD shows us all 15 pairwise differences between the 6 combinations of `wool` and `tension`. This a situation where Fishers LSD does not perform well. The probability of detecting at least *one* difference where none exist is $(1-(1-\alpha)^n)$, with $\alpha = 0.05$ and n=15 this is 0.537. This shows why Tukey's multiple comparisons is important – this "family-wise" error rate can get quite high quickly.

Sorting out which groups really differ is not always simple, though in this case we can rather quickly see that the other five groups differ from woolA:tensionL, but those 5 groups don't differ from each other.

VIII. HSD.test - a useful tool for ANOVA

The package `agricolae` has some nice tools to make multiple comparisons a bit easier. Install the package to follow along.

```
library(agricolae)
data(sweetpotato)
```

Now we can fit a model to the data and make comparisons.

```
model <- aov(yield ~ virus, data = sweetpotato)
out <- HSD.test(model, "virus", group = TRUE)
out$means
```

```
#       yield      std r  Min  Max
# cc 24.40000 3.609709 3 21.7 28.5
# fc 12.86667 2.159475 3 10.6 14.9
# ff 36.33333 7.333030 3 28.0 41.8
# oo 36.90000 4.300000 3 32.1 40.4
```

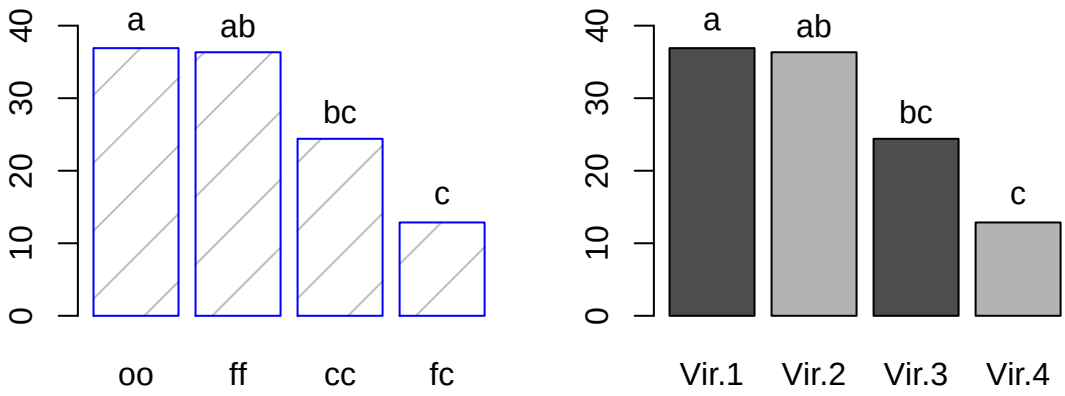
```
out$groups
```

```
#   trt    means  M
# 1  oo 36.90000  a
# 2  ff 36.33333 ab
# 3  cc 24.40000 bc
# 4  fc 12.86667  c
```

`HSD.test()` has nicely calculated the mean and standard deviations for each group and a description of how the means are grouped, with the groups denoted by letter such that groups that share a letter aren't different. In this case `oo` differs from `cc` and `fc`, and `fc` differs from `oo` and `ff`.

A convenience function is provided which we can use to make a barplot of such an analysis, and all the arguments to `barplot()` can be used to modify this plot.

```
bar.group(out$groups, ylim = c(0, 45), density = 4, border = "blue")
bar.group(out$groups, ylim = c(0, 45), col = c("grey30", "grey70"), names.arg = c("Vir.1",
    "Vir.2", "Vir.3", "Vir.4"), ylab = "some units")
```



Here we demonstrate again on the `warpbreaks` data:

```
model2 <- (lm(breaks ~ wool + tension + wool:tension, data = warpbreaks))
HSD.test(model2, trt = "tension", group = TRUE, main = "Wool x Tension")$groups
```

```
#   trt    means M
# 1   L 36.38889 a
# 2   M 26.38889 b
# 3   H 21.66667 b
```

Note that while the `HSD.test()` function does not work for interactions, we can create an interaction variable using `intearction()` and then use `HSD.test()`.

```
txw <- with(warpbreaks, interaction(wool, tension))
model3 <- aov(breaks ~ txw, data = warpbreaks)
library(agricolae)
HSD.test(model3, "txw", group = TRUE)$groups
```

```
#   trt    means M
# 1 A.L 44.55556 a
# 2 B.M 28.77778 b
# 3 B.L 28.22222 b
# 4 A.H 24.55556 b
# 5 A.M 24.00000 b
# 6 B.H 18.77778 b
```

Note that this output is much more compact than the output from `TukeyHSD(aov(breaks~wool+tension,data=warpbreaks))` shown earlier, but the conclusion is identical.

IX. Exercises

1) For the `beans` data we used in Chapter 11 model shoot biomass as a function of phosphorus (phos). Make sure `phos` is coded *as a factor*. What does the output of `summary` suggest?

2) Use Tukey to compare the different levels of `phos` for the model in Problem 1. Does this same pattern hold for Root biomass?

3) I kept track of my electric bill every month for over 7 years. The data set ("electric bill.txt") is located in the "Data" directory in essential R, and includes variable for `month`, `year`, the amount of electricity used in kilowatt hours (`kwh`), the number of days in the billing cycle (`days`), and the average temperature during the billing cycle (`avgT`). There are also variables that describe whether the bill was based on an estimated reading or actual reading (`est`, with levels `e` and `a` for estimated or actual), `cost` (in dollars), and energy use per day (`kWhd.1`).

Fit a model of `kwhd.1` as a function of `avgT`. What is the R^2? Test the hypothesis that the slope is -0.25 kwh per day per degree F increase in average monthly temperature. What is the p-value for this test?

4) My old house did not have AC, but we ran several fans in the summer, and the refrigerator and freezer certainly worked harder during the warmer months, so there could be a minimum in energy use at moderate temperatures. Include a quadratic (squared) term for average temperature. How does this change the R^2 value of the model? What is the p-value for the quadratic term? Do the residuals suggest that one model should be favored?

EXTRA) Plot the linear model (a line) and the quadratic model (a curve) over the data. *Hint* you can use the function `curve()` to add curves to plots.

Chapter 13: Linear Models III.

More Linear Models

I. Introduction

In the last chapter we looked at extending linear regression to prediction from one or more categorical variables (factors). Here we'll explore a few more extensions of the linear model. First we'll look at using multiple continuous predictors. We'll also see how we can mix continuous and categorical predictors (often known as ANCOVA - Analysis of Co-variance). We'll also briefly consider how we can make specified comparisons between groups, and look at some more complex experimental designs.

A word about model selection – whenever we make statistical models we make decisions about what to include and what not to include. Essential R is focused on the mechanics of using R, and does not treat the issue of model selection, beyond noting that looking at whether regression assumptions are met is an important part of model selection. There are many different tools for testing different combinations of variables in models (forward selection, reverse selection,...), and there are (sometimes very strong) critiques of all these methods. The most defensible practice is to build a model based on hypothesis.

II.Multiple Regression

In some situations there is more than one predictor variable that needs to be included in our model. Here we'll use the "stackloss" data to demonstrate some simple multiple regressions[39]. This data concerns the oxidation of ammonia to nitric acid based on three variables: Air flow, Water temperature, and Acid Concentration. Let's begin by looking at all the variables individually.

```
data(stackloss)
summary(lm(stack.loss ~ Air.Flow, data = stackloss))$coef
```

```
#               Estimate Std. Error   t value     Pr(>|t|)
# (Intercept) -44.132025 6.10585762 -7.227818 7.314184e-07
# Air.Flow      1.020309 0.09995287 10.207905 3.774296e-09
```

```
summary(lm(stack.loss ~ Water.Temp, data = stackloss))$coef
```

```
#               Estimate Std. Error   t value     Pr(>|t|)
# (Intercept) -41.910867  7.6056213 -5.510512 2.575258e-05
# Water.Temp    2.817445  0.3567438  7.897672 2.028017e-07
```

```
summary(lm(stack.loss ~ Acid.Conc., data = stackloss))$coef
```

```
#                Estimate Std. Error   t value  Pr(>|t|)
# (Intercept) -47.9631841 34.5043888 -1.390060 0.1805827
# Acid.Conc.    0.7589552  0.3991529  1.901415 0.0725230
```

There is some relationship with all three variables, though the p-value for `Acid.Conc.` is not quite low enough to be considered significant. With multiple regression we can combine multiple predictors, using the formula interface introduced in Chapter 12.

So we can use `stack.loss~Air.Flow + Water.Temp` to combine the two.

```
summary(lm(stack.loss ~ Air.Flow + Water.Temp, data = stackloss))$coef
```

```
#               Estimate Std. Error   t value     Pr(>|t|)
# (Intercept) -50.3588401  5.1383281 -9.800628 1.216471e-08
```

[39]For a slightly longer introduction to multiple regression, see Section 14 in "SimpleR" by John Verzani. For a longer, more in-depth introduction see Chapter 11 in "Statistics: An Introduction using R" by Michael Crawley.

```
# Air.Flow      0.6711544  0.1266910  5.297568 4.897970e-05
# Water.Temp    1.2953514  0.3674854  3.524905 2.419146e-03
```

Note that we've displayed only the coefficients here (`summary(...)$coef`) for brevity. We can see that both variables have a highly significant effect on the response. We haven't yet tested to see if they *interact*.

```
summary(lm(stack.loss ~ Air.Flow * Water.Temp, data = stackloss))$coef
```

```
#                       Estimate  Std. Error    t value   Pr(>|t|)
# (Intercept)         22.29030069 35.09601861  0.6351233 0.53380403
# Air.Flow            -0.51550885  0.57984497 -0.8890460 0.38638596
# Water.Temp          -1.93005822  1.58044278 -1.2212136 0.23867188
# Air.Flow:Water.Temp  0.05176139  0.02477854  2.0889604 0.05205945
```

This is an interesting situation. The interaction term of these variables is *marginally* significant, and each individual effect isn't significant apart from the interaction. Let's see how the third variable influences the response.

```
summary(lm(stack.loss ~ Air.Flow * Water.Temp * Acid.Conc., data = stackloss))$coef
```

```
#                                     Estimate   Std. Error   t value
# (Intercept)                     -3.835095e+03 1.771119e+03 -2.165351
# Air.Flow                         6.414455e+01 2.935552e+01  2.185093
# Water.Temp                       1.864374e+02 8.989859e+01  2.073864
# Acid.Conc.                       4.367229e+01 2.006597e+01  2.176435
# Air.Flow:Water.Temp             -3.074347e+00 1.479097e+00 -2.078530
# Air.Flow:Acid.Conc.             -7.321000e-01 3.324893e-01 -2.201875
# Water.Temp:Acid.Conc.           -2.131307e+00 1.017941e+00 -2.093743
# Air.Flow:Water.Temp:Acid.Conc.   3.537184e-02 1.674228e-02  2.112725
#                                    Pr(>|t|)
# (Intercept)                     0.04954646
# Air.Flow                        0.04778701
# Water.Temp                      0.05851935
# Acid.Conc.                      0.04855129
# Air.Flow:Water.Temp             0.05802734
# Air.Flow:Acid.Conc.             0.04633763
# Water.Temp:Acid.Conc.           0.05644977
# Air.Flow:Water.Temp:Acid.Conc. 0.05453733
```

This is an interesting and complex case as we have a marginally significant three way interaction, and all other effects are at least marginally significant. We can exclude the three-way interaction to see if that clarifies the situation.

```
summary(lm((stack.loss ~ (Air.Flow + Water.Temp + Acid.Conc.)^2), data = stackloss))$coef
```

```
#                           Estimate   Std. Error    t value  Pr(>|t|)
# (Intercept)           -108.13059141 176.49073390 -0.6126701 0.5499176
# Air.Flow                 2.32653047   2.64482119  0.8796551 0.3938952
# Water.Temp              -2.78892775   8.64066216 -0.3227678 0.7516411
# Acid.Conc.               1.44055540   1.95825282  0.7356330 0.4740992
# Air.Flow:Water.Temp      0.05012641   0.02807059  1.7857268 0.0958144
# Air.Flow:Acid.Conc.     -0.03188801   0.02967467 -1.0745868 0.3007319
# Water.Temp:Acid.Conc.    0.01184376   0.09472983  0.1250267 0.9022800
```

But in this case it does not - reality seems to require the three way interaction - all factors and all interactions would be significant at `p=0.06`. Fortunately, we *don't* need to interpret this model verbally[40] for it to be

[40]Three way interactions (higher order interactions in general) are very difficult to interpret - in this case, the effect of Acid concentration on stackloss depends on the combination of airflow and water temperature.

useful - we can predict which conditions will optimize the outcome, which *would* be useful in a process control context, even though those of us coming from analytical backgrounds may shudder at three-way interactions.

III. ANCOVA

In some cases we have both numeric and categorical predictors. For example, the weed suppressiveness of a rye cover-crop depends partly on the biomass of the rye, which is strongly affected by planting and termination dates of the rye. In one study rye biomass was modeled as a function of planting date (early or late) and terminated at one of five harvest dates (numeric). We'll load some data and have a look.

```
Rye <- read.csv("../Data/Rye-ANCOVA-2008.csv", comm = "#")
head(Rye)  # examine the data
```

```
#   Plant Term Rep RyeDMg Pdate Tdate GrDay Tdate2
# 1    P1   T1   I  443.0   257  20.0 233.0   20.0
# 2    P2   T1   I  287.0   288  20.0 202.0   20.0
# 3    P1   T2   I  631.5   257  32.5 245.5   32.5
# 4    P2   T2   I  513.0   288  32.5 214.5   32.5
# 5    P1   T3   I  771.5   257  39.5 252.5   39.5
# 6    P2   T3   I  570.5   288  39.5 221.5   39.5
```

The variable `Tdate` is days from the beginning of the growing season (April 15). We'd expect that rye that is terminated later will produce greater biomass.

```
summary(lm(RyeDMg ~ Tdate, data = Rye))
```

```
#
# Call:
# lm(formula = RyeDMg ~ Tdate, data = Rye)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -195.779  -87.402    2.592   82.281  164.221
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   96.232     86.083   1.118    0.276
# Tdate         16.963      2.715   6.249 2.74e-06 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 107.3 on 22 degrees of freedom
# Multiple R-squared:  0.6396,  Adjusted R-squared:  0.6232
# F-statistic: 39.04 on 1 and 22 DF,  p-value: 2.739e-06
```

Sure enough, we see an influence of termination date (`Tdate`). The model suggests that rye terminated at the very beginning of the growing season would only produce 96 g m^{-2} which is a reasonable value, and that each day of delay in termination increases rye biomass by 17 g. But there could still be more going on here. The rye was planted in the autumn, and the later planting missed more of the warm autumn weather during it's early growth phase.

```
summary(lm(RyeDMg ~ Plant, data = Rye))
```

```
#
# Call:
# lm(formula = RyeDMg ~ Plant, data = Rye)
```

```
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -256.63 -115.16   24.06   83.00  332.88
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   689.25      46.68  14.766 6.73e-13 ***
# PlantP2      -145.63      66.01  -2.206   0.0381 *
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 161.7 on 22 degrees of freedom
# Multiple R-squared:  0.1811,  Adjusted R-squared:  0.1439
# F-statistic: 4.867 on 1 and 22 DF,  p-value: 0.03812
```

This suggests that later planting in the autumn reduced biomass by 145 g m^{-2} . We should be able to combine these two effects with an ANCOVA.

```
summary(lm(RyeDMg ~ Tdate + Plant, data = Rye))
```

```
#
# Call:
# lm(formula = RyeDMg ~ Tdate + Plant, data = Rye)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -122.967  -65.880   -5.537   58.567  183.033
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   169.04      64.12   2.636 0.015429 *
# Tdate          16.96       1.96   8.656 2.27e-08 ***
# PlantP2      -145.62      31.61  -4.607 0.000152 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 77.43 on 21 degrees of freedom
# Multiple R-squared:  0.8208,  Adjusted R-squared:  0.8037
# F-statistic: 48.08 on 2 and 21 DF,  p-value: 1.45e-08
```

In the ANCOVA model we can see that the both the planting date and growing time effects are very significant, and the R^2 of this model is much better than either of the separate models. The early planted rye begins the growing season with about 170 g m^{-2} of biomass, while the later planted rye only has about 26 g m^{-2}. It is possible that the longer growth time is more beneficial in one group compared to the other. This would show in the model as a different slope (`Tdate` coefficient) for the groups.

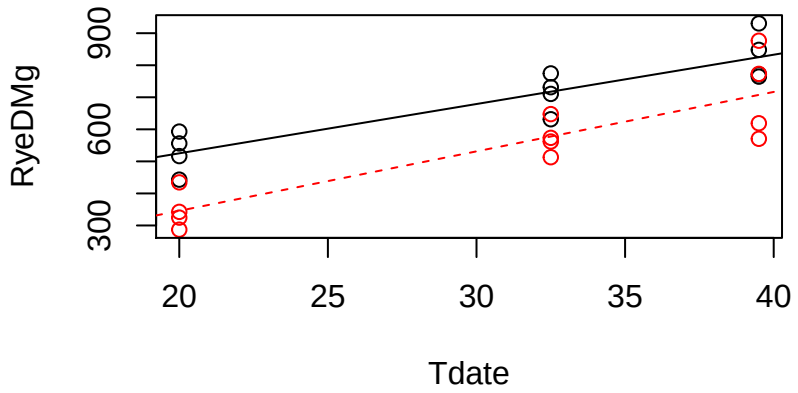
```
model1 <- lm(RyeDMg ~ Tdate + Plant + Tdate:Plant, data = Rye)
summary(model1)
```

```
#
# Call:
# lm(formula = RyeDMg ~ Tdate + Plant + Tdate:Plant, data = Rye)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
```

```
# -136.928  -59.276   -5.537   59.179  169.072
#
# Coefficients:
#               Estimate Std. Error t value Pr(>|t|)
# (Intercept)    217.514     88.641   2.454   0.0234 *
# Tdate           15.383      2.795   5.503 2.19e-05 ***
# PlantP2       -242.565    125.358  -1.935   0.0673 .
# Tdate:PlantP2    3.161      3.953   0.800   0.4333
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 78.1 on 20 degrees of freedom
# Multiple R-squared:  0.8263,  Adjusted R-squared:  0.8003
# F-statistic: 31.71 on 3 and 20 DF,  p-value: 8.487e-08
```

Here we see that the interaction term is *not* significant, so the slope (the effect of harvesting date) is the same for both groups, or (more accurately) the difference between the slope for the two groups is *not different* from zero (p=0.433.

```
plot(RyeDMg ~ Tdate, data = Rye, col = Plant)
mcf <- summary(model1)$coef[, 1]  #coefficient estimates only
abline(a = mcf[1], b = mcf[2])
abline(a = mcf[1] + mcf[3], b = mcf[2] + mcf[4], lty = 2, col = "red")
```



For a more detailed treatment of multi-way ANOVA, see Chapter 9 of "Statistics: An Introduction Using R" by Michael J. Crawley. Also see Chapter 16 of [Practical Regression and Anova using R] (http://www.ats.ucla.edu/stat/r/sk/books_pra.htm) by Julian J. Faraway, which is available as a pdf.

IV. About Sums of Squares in R

Linear models in R are fit using the classic method of minimizing the sum of the squared residuals. This is simple enough and can be solved analytically. For more detail see SimpleR pg 77, or see [this page] (http://www.wired.com/wiredscience/2011/01/linear-regression-by-hand/).

However, the F-tests used in ANOVA tests are also calculated using sums of squares - the F-test for an effect is the mean square for the effect divided by the mean square of the error, where the mean square is the sum of squares for that effect divided by the degrees of freedom for the effect

This is all pretty straightforward when considering a one-way ANOVA, or a two- or more way ANOVA with balanced data (no missing values). However, in the real world of imperfect experiments and missing data it gets more complicated. R by default calculates what are known as *sequential* sums of squares, or *Type I SS*. Many other software packages provide what are known as *Type III SS* (There are also Type II SS).

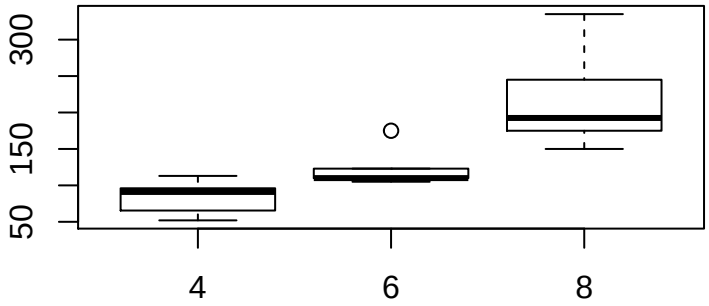
The differences in which sums of squares are calculated explains why a model fit in R will not necessarily return the same results as if it were fit in SAS, SPSS, or Minitab. Note that this **does not mean that R is wrong**. Type I SS are *sequential*, meaning the *order of terms in the model* matters. In many cases it can be argued that there is a logical order to the terms that reflects the design of the experiment (e.g. block comes first, etc.).

See [this article] (http://goanna.cs.rmit.edu.au/%7Efscholer/anova.php) for a more complete explanation of Type I, Type II, and Type III SS, and how they can be calculated in R.

V. Resistant Linear Models

There are resistant regression methods that are less susceptible to outliers. For this step we'll need to install the MASS package (`install.packages("MASS")` will download and install the package). We'll explore it a bit using the `mtcars` data.

```
library(MASS)  # load the package 'MASS'
data(mtcars)  # load the data
boxplot(hp ~ cyl, data = mtcars)
```



This boxplot should convice us that we may have a problem with heterogeneity of variance which might invalidate a linear model. Let's see how the resistant model compares. Since cyl is coded as numeric but is really a factor (taking only values 4, 6, or 8 in this data set), we'll model it as a factor.

```
summary(lm(hp ~ factor(cyl), data = mtcars))
```

```
#
# Call:
# lm(formula = hp ~ factor(cyl), data = mtcars)
#
# Residuals:
#    Min     1Q Median     3Q    Max
# -59.21 -22.78  -8.25  15.97 125.79
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)     82.64      11.43   7.228 5.86e-08 ***
# factor(cyl)6    39.65      18.33   2.163   0.0389 *
# factor(cyl)8   126.58      15.28   8.285 3.92e-09 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 37.92 on 29 degrees of freedom
# Multiple R-squared:  0.7139,  Adjusted R-squared:  0.6941
# F-statistic: 36.18 on 2 and 29 DF,  p-value: 1.319e-08
```

```
summary(rlm(hp ~ factor(cyl), data = mtcars))
```

```
#
# Call: rlm(formula = hp ~ factor(cyl), data = mtcars)
# Residuals:
#     Min      1Q  Median      3Q     Max
# -53.001 -21.227  -4.376  17.364 131.999
#
# Coefficients:
#              Value    Std. Error t value
# (Intercept)   82.6364  10.0218     8.2457
# factor(cyl)6  38.1156  16.0706     2.3718
# factor(cyl)8 120.3642  13.3922     8.9877
#
# Residual standard error: 32.35 on 29 degrees of freedom
```

The t-values are a bit larger for the resistant model. Note the resistant model does not supply p-values. This is because calculating p-values depends on making assumptions about the t-distribution which might not be met when regression assumptions are violated.

VI. Specifying Contrasts.

In some cases we approach a data set with very specific questions, that we want to answer. Consider the warpbreaks data: two types of wool and three levels of tension, for 6 treatment groups:

```
data(warpbreaks)
unique(warpbreaks[, 2:3])
```

```
#    wool tension
# 1     A       L
# 10    A       M
# 19    A       H
# 28    B       L
# 37    B       M
# 46    B       H
```

With 6 groups there are 15 possible comparisons, but the default treatment coding used in `summary(lm)` only tests 5 of these:

```
summary(lm(breaks ~ wool * tension, data = warpbreaks))$coef
```

```
#                Estimate Std. Error   t value     Pr(>|t|)
# (Intercept)     44.55556   3.646761 12.217842 2.425903e-16
# woolB          -16.33333   5.157299 -3.167032 2.676803e-03
# tensionM       -20.55556   5.157299 -3.985721 2.280796e-04
# tensionH       -20.00000   5.157299 -3.877999 3.199282e-04
# woolB:tensionM  21.11111   7.293523  2.894501 5.698287e-03
# woolB:tensionH  10.55556   7.293523  1.447251 1.543266e-01
```

We're in luck if we want to compare: woolA:tensionL with woolB:tensionL (row 2), woolA:tensionL with woolA:tensionM (row3), woolA:tensionL with woolA:tensionH (row4). Row 5 represents the difference between the estimate of woolB:tensionM generated by adding estimates 1,2, and 3 (the additive estimate) and the actual mean of woolB:tensionM (if this difference is small, we say there is not an interaction of the 2 terms). Row 6 is analogous to row 5.

But what if we want to compare WoolA vs WoolB *at each level of tension*? We need to specify our own contrasts. We can easily see that these differences may be important by looking at the means:

```
aggregate(warpbreaks$breaks, by = warpbreaks[, 2:3], mean)
```

```
#   wool tension        x
# 1    A       L 44.55556
# 2    B       L 28.22222
# 3    A       M 24.00000
# 4    B       M 28.77778
# 5    A       H 24.55556
# 6    B       H 18.77778
```

The difference between rows 1 and 2, 3 and 4, and 5 and 6 show our differences, but don't test them. To test these we need to define a *matrix of contrasts*. This recreates the means for each contrast from the coefficients in the model.

```
m1 <- (lm(breaks ~ wool * tension, data = warpbreaks))
summary(m1)$coef
```

```
#                 Estimate Std. Error   t value     Pr(>|t|)
# (Intercept)     44.55556   3.646761 12.217842 2.425903e-16
# woolB          -16.33333   5.157299 -3.167032 2.676803e-03
# tensionM       -20.55556   5.157299 -3.985721 2.280796e-04
# tensionH       -20.00000   5.157299 -3.877999 3.199282e-04
# woolB:tensionM  21.11111   7.293523  2.894501 5.698287e-03
# woolB:tensionH  10.55556   7.293523  1.447251 1.543266e-01
```

So woolA:tensionL would be:`AL=c(1,0,0,0,0,0)`, and woolB:tensionL would be: `BL=c(1,1,0,0,0,0)`. The difference between them is `AL-BL` or `c(0,-1,0,0,0,0)`. WoolA:tensionM is: `AM=c(1,0,1,0,0,0)` and woolB:tensionM is `BM=c(1,1,1,0,1,0)` and the difference is `c(0,-1,0,0,-1,0)`. By extension, the final difference is `c(0,-1,0,0,0,-1)`.

Constructing these contrasts is the most confusing part of this procedure. You can confirm that the last one is correct by taking the sum of product of the first column of the coefficients and e.g `c(0,-1,0,0,0,-1)` (the tension H difference). It will be the same as the wool difference for tensionH.

```
sum(summary(m1)$coef[, 1] * c(0, -1, 0, 0, 0, -1))
```

```
# [1] 5.777778
```

```
means <- aggregate(warpbreaks$breaks, by = warpbreaks[, 3:2], mean)
means[3, 3] - means[6, 3]
```

```
# [1] 5.777778
```

We can put these three contrasts together into a matrix like this (note there is one row of the matrix per contrast):

```
cont.mat = matrix(c(0, -1, 0, 0, 0, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, 0, -1),
    byrow = TRUE, nrow = 3)
cont.mat
```

```
#      [,1] [,2] [,3] [,4] [,5] [,6]
# [1,]    0   -1    0    0    0    0
# [2,]    0   -1    0    0   -1    0
# [3,]    0   -1    0    0    0   -1
```

Now we can calculate our comparisons using the function `glht()` in the package `multcomp` (you will probably need to install it unless you already have done so - `install.packages("multcomp")`.

```
library(multcomp)
tests <- glht(m1, linfct = cont.mat)
summary(tests)
```

```
#
#    Simultaneous Tests for General Linear Hypotheses
#
# Fit: lm(formula = breaks ~ wool * tension, data = warpbreaks)
#
# Linear Hypotheses:
#        Estimate Std. Error t value Pr(>|t|)
# 1 == 0   16.333      5.157   3.167    0.008 **
# 2 == 0   -4.778      5.157  -0.926    0.732
# 3 == 0    5.778      5.157   1.120    0.603
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
# (Adjusted p values reported -- single-step method)
```

And there you have it. The first test (wool at tension L) is significant, but the others aren't. Notice that here we are only making 3 comparisons rather than all 15, and so we conserve our statistical power.

As noted above, making the contrast matrix is the key (and confusing) step here. Be aware that subsequent changes to the model will change the contrast matrix.

For more information:

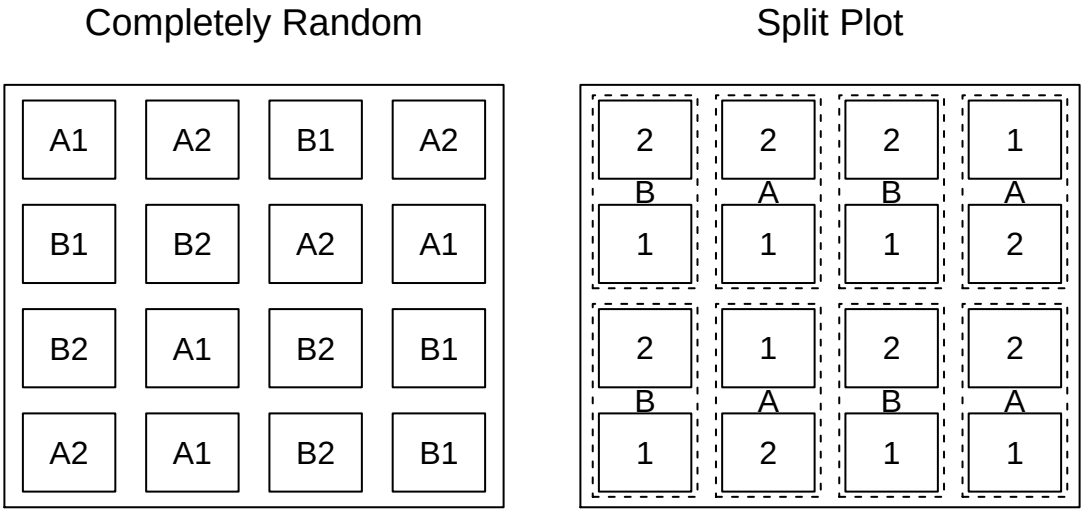
Chapter 9 ("Statistical Modeling") in "The R Book" by Michael Crawley has a very good introduction to this topic beginning at Section 9.25. He includes details on several other approaches and goes into more detail.

A good introduction to testing contrasts using the package "multcomp" can be found here

VII: More Complex Designs

It is beyond the scope of these notes to fully consider more complex models such as split-plot designs, nested designs, and mixed effects models. The chapter on "Analysis of Variance" (Ch11) in "The R Book", by Michael Crawley, has some very good examples, and several are to be had by searching the "R-help" mailing lists (See the "Support Lists" tab in the search results at Rseek).

A split-plot design means that one treatment or factor is applied to *larger* units ("whole plot" or "main plot"), and subunits of those larger units ("subplot") receive other treatments. Here we'll compare two factors - "A" vs "B" and "1" vs "2". In a completely random design (CRD) all combinations are randomly assigned to locations (First figure). In a split plot one factor (in this example, "A" vs "B") is assigned to larger plots, and the subplots are randomly assigned to the other factor (in this case "1" or "2")



By way of an introductory example, we'll consider a rather simple split-plot design here. The data is from the study on cereal rye winter cover-crops introduced above. Cereal rye was sown at two dates in the autumn (`PD`, a factor) and was killed with a roller-crimper at 5 dates in the spring (`Tdate`, days after April 15), when biomass (`RyeDM`, g m^{-2}) was measured. Planting date was the main plot (2 per block) with the subplots randomly assigned to combinations of termination date (5 per main plot).

```
# plot.
Rye <- read.csv("../Data/RyeSplitPlot.csv", comm = "#")
summary(Rye)
```

```
#  Plant   Tdate  WdCon    Rep         RyeDM            Pdate
#  P1:20   T1:8   SC:40   I  :10   Min.   : 104.0   Min.   :257.0
#  P2:20   T2:8           II :10   1st Qu.: 454.8   1st Qu.:257.0
#          T3:8           III:10   Median : 636.0   Median :272.5
#          T4:8           IV :10   Mean   : 630.0   Mean   :272.5
#          T5:8                    3rd Qu.: 816.0   3rd Qu.:288.0
#                                  Max.   :1256.0   Max.   :288.0
#     Tdate.1
#  Min.   :480.0
#  1st Qu.:492.0
#  Median :500.0
#  Mean   :500.8
#  3rd Qu.:513.0
#  Max.   :519.0
```

We can see we have 2 levels of `PD`, 5 of `Tdate`, and 4 for `Rep` (block).

The simplest approach to analyzing this as a split plot is to use the function `aov()` and specify the error structure using the argument `Error`. First we'll ignore the split-plot:

```
summary(aov(RyeDM ~ Rep + Plant + Tdate, data = Rye))
```

```
#             Df  Sum Sq Mean Sq F value   Pr(>F)
# Rep          3   72833   24278   1.938 0.144014
# Plant        1  216531  216531  17.283 0.000235 ***
# Tdate        4 2043518  510879  40.778 6.28e-12 ***
# Residuals   31  388379   12528
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

It is easy to fit this model and forget that it is wrong, since the split plot design has not been included. Now

we'll fit the split-plot model.

```
summary(aov(RyeDM ~ Rep + Plant + Tdate + Error(Rep/Plant), data = Rye))
```

```
#
# Error: Rep
#     Df Sum Sq Mean Sq
# Rep  3  72833   24278
#
# Error: Rep:Plant
#           Df Sum Sq Mean Sq F value Pr(>F)
# Plant      1 216531  216531   8.152 0.0648 .
# Residuals  3  79686   26562
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Error: Within
#           Df  Sum Sq Mean Sq F value   Pr(>F)
# Tdate      4 2043518  510879   46.34 5.92e-12 ***
# Residuals 28  308693   11025
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The `Error(Rep/Plant)`tells us that the units of the variable `Plant` are within the units of `Rep`. The units of `Tdate` are within `Plant`, but this doesn't need to be specified as this is the smallest unit. This gives us an ANOVA table split according to the error term used in the F test, and we can see that the main plot (`Plant`) effect is tested against the interaction of block and main plots (`Rep:Plant`). This results in a larger (in this case marginally significant) p-value. The larger p-value is because the correct error term is used here. A split-plot design has more power to detect differences in the split-plot factor and less power to detect those in the whole plot factor. Incorrect analysis of these designs is surprisingly common.

Notice that the predictor variables (effects) in the right-hand side of the formula are listed from physically largest to smallest rather than in order of our interest - this probably makes sense given that we are working with Type I sums of squares here - there is a natural ordering of the effects based on the construction of the experiment.

Later we'll consider another way to test this is as a linear mixed-effects model using the function `lme()` from the package "nlme".

VIII. Exercises

1) The built in data set `trees` has data for diameter, height, and timber volume for 31 trees. How well does the volume of a cone (0.333 x height x radius^2 x π) predict the timber volume (fit a regression between the measured and calculated volume). Note that volume is in cubic feet, height is in feet, and diameter is in inches (divide by 12 for feet). How does this differ from a model of measured volume as a function of height and diameter squared (which is mathematically incorrect)? *Note on units* The variable named "Girth" is actually diameter according to the help file for the data set. See `? trees`.

2) For the `beans` data test whether shoot biomass alters the relationship between root biomass and root length we explored in exercise 1 of Chapter 11. (That is, is there an interaction between shoot biomass and root biomass in modeling root length)

3) In exercise 3 and 4 of Chapter 12 we began looking at the "electric bill" data. Here is an ANCOVA problem. In April of 2008 (month 57 of occupancy) we added insulation to the roof of the house. How did this affect the relationship between daily energy use and average temperature? Add a factor for with and

without insulation, and add this factor to the quadratic model. How does this affect the fit of the model (consider residual distributions and R^2)

4) Does the effect of adding insulation alter the coefficients of the relationship between temperature and daily electric consumption? (Do the `avgT` and `I(avgT^2)` terms interact with the insulation factor?)

EXTRA What is wrong with this model?

Chapter 14: Productivity Tools in R

More advanced documentation and managing projects.

I. Introduction

Since version 0.96.122, RStudio has included tools to allow easy use of markdown to produce dynamic reports. Here we'll briefly explore how we can use this to document analytical work in R. (Markdown is a simple markup language created by people with an appreciation for irony). There are many parallels between using markdown and compiling notebooks from R scripts (Chapter 4) - markdown just gives a slightly higher level of control.

As noted in Chapter 4, to fully document statistical analysis, we need to have: 1) the R code; 2) the output from R code (statistical tables, etc); 3) the figures we generate; and 4) the appropriate commentary and narrative.

I've noted earlier the importance of keeping well commented code files, which can cover 1) and 4), but the code has to be run to create 2) and 3). In Chapter 4 we saw how the "compile notebook" functionality in `knitr` allows us to create html documents from our R script files. While generally adequate for the purposes of homework assignments, one might find that they don't offer all the features that one might want.

The combination of `knitr` and *markdown* solve this problem – we can have all four components in *one file* [41]. The highlights are: 1. Code and commentary stored in one file.
a. both data manipulation and analysis code stored
b. ability to have extensive commentary w/o commenting out every line
c. ability to apply basic formatting to non-code commentary
d. ability to keep code that is not run for reference
e. ability to produce figures without showing the code that produces them
2. Statistical output and figures easily kept with the code that produced them.
3. Easy to go back and re-run or update analyses.

Using markdown and `knitr` in RStudio it is almost trivially easy to put this all in **one** file. The markdown file is easily readable plain text and permits simple text formatting. R code is included in discrete *"chunks"*. Knitr will process ("knit" or "compile") the markdown file into HTML (or pdf or .docx) that includes the code and the output from that code.

Knitr "chunk options"" allow the user to specify whether code chunks are evaluated, (if something didn't work but you want to keep a record of how you tried it, a chunk that isn't evaluated is perfect), and whether the code itself is shown in the HTML file (perfect for figures). Chunk options also allow size of figures to be customized.

II. Getting stated with markdown

In the RStudio editor pane choose *File>New>R Markdown*. You'll be prompted to select type - at this point select "Document". If you haven't used knitr already you may be prompted to install the knitr package, in which case type `install.packages("knitr",dep=T)`. RStudio will open a new tab in the editor that is a basic template.

Examining this template will reveal some special features:
1. Unlike an R script file which can only contain comments and R code, a markdown file contains plain text and some special "chunks" delimited by special "fences".
2. Code chunks are fenced by lines containing "‘ ‘ ‘", and contain R code and comments.
3. For the "opening fence" the "‘ ‘ ‘" is followed by "{r}"
4. In some chunks there are "chunk options" following the "r" in "{r}. For example {r, echo=FALSE}" tells knitr to execute the code in this chunk but not to include the code itself in the output. This can be used to

[41] Sort of - as we'll see it may be thought of as 2 files.

show only the plots and note the code that produced them.[42]
5. Some areas within the plain text part of the file are fenced with "‘", such as "‘echo=FALSE‘". These areas are formatted as code in the compiled file. This is known as *inline code*.
6. There is a header that is fenced by lines containing "—". This contains details such as the title, the author and date, and the type of output. Other features can be added here, such as table of contents, but that is beyond this basic intro.[43]

Markdown is a simple markup language, and text not within code chunks is formatted according to markdown rules. Up to six levels of headers are possible, as well as italics, boldface, super and subscript, and simple tables. LaTeX style equations can also be included. More document customizations incuding tables of contents are possible, but are not discussed here [44].

Key Features

Inline code such as `\ mean(some.data)‘ will not be evaluated, but adding an ‘ ̊ ‘` followed by a space at the beginning of the online code will flag that inline code to be evaluated. This is useful when discussing results of an analysis, e.g. inserting a p-value into your text by linking to the analysis.

The "..." in `{r ...}` stand for "chunk options". The *first* option is preceded by a space, and all others are separated by commas, without any spaces. The first option is a "chunk name". You can navigate to named chunks from the pull-down menu on the lower left margin of the editor tab. The options commonly used options are:

- `eval=FALSE` - means don't evaluate this chunk - useful for keeping code you don't want to run, but don't want to delete.
- `echo=FALSE` - means run the code but don't print it - useful for figures where you want to show the figure not the code you used to make it. If I'm preparing a report for my boss, I use this a lot.
- `fig.width=x`; `fig.height=y` - controls width and height of figures (dimensions in inches).
- Note that the `echo` and `eval` chunk options can specify line numbers, so `echo=-1` means echo all the code in the chuck except line 1.

Other useful information:

- `Ctrl+Alt+I` inserts a new chunk (Mac: `Opt+Cmd+I`)
- `Ctrl+Alt+C` runs the current chunk
- `Ctrl+Alt+N` runs the next chunk
- `Ctrl+Enter` runs the current line just like in an R script file
- a line like:

  ```
  `knitr::opts_chunk$set(fig.width=4.5,fig.height=3,tidy=TRUE,cache=TRUE)`
  ```

  early after the title block sets *global options* for a document. The `cache=TRUE` option speeds up compilation for the second and following times a document in knit, but can occasionally cause surprise errors - use at your own risk
- As with the *notebook* feature, R code run when knitting a document is evaluated in a separate environment. This means that your R workspace is not accessible by this code, so you have to make sure ALL the necessary code is in your chunks. This most often causes errors when data or packages

[42] More information of advanced document features can be found [here] (http://rmarkdown.rstudio.com/html_document_format.html)
[43] The header is sometimes referred to as the "YAML metadata block".
[44] More information of advanced document features can be found [here] (http://rmarkdown.rstudio.com/html_document_format.html)

that the code needs haven't been loaded. Another possible source of errors is not having the CRAN mirror set correctly. [45]

Click the "?" button on the editor toolbar and select "Markdown Quick Reference" to bring up a short reference in the help viewer. A more extensive description is available online - selecting "Using R Markdown" from the "?" will open this link in your browser.

Finally, the full list of chunk options is detailed at the knitr website. Note that not all of the options listed here function in markdown - some only operate in "sweave" files (.rnw), which allow R code to be embedded and evaluated within LaTeX documents[^14-4]. [^14-4]: If you are comfortable with LaTeX you can use File>New>RSweave to open a LaTeX file that you can insert code chunks into, but the syntax is different than markdown.

One of the great advantages of this method of documenting your work is that you can easily update an analysis by updating the data file and running the analysis again. Important conclusions (e.g. p-values) can be in inline code chunks, so R just updates them too.

The file "knitr-markdown demo.rmd" in your "Essential R" folder is a demonstration of many features of R markdown.

III. Managing projects

Using a tool like markdown makes it easier to keep your thinking documented. One of the challenges of *any* kind of analytical work is keeping track of many different files. Typically most users use directories (folders) to group documents related to particular projects. RStudio makes this easy with *project* feature. The project menu is located in the upper right corner of the RStudio window.

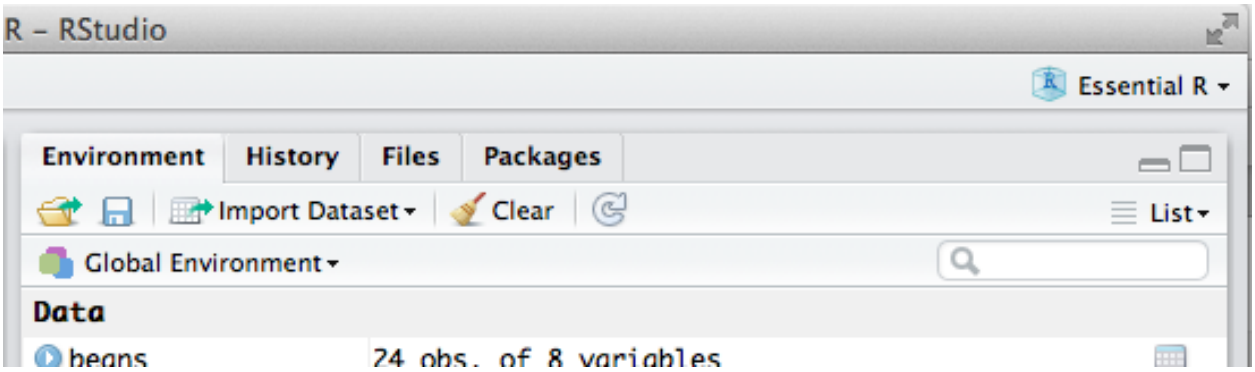


Figure 3: Project Menu

If you click on the project menu you see options to create a new project, to open a project, and there are shortcuts to the recent projects.

When a project is created it is associated with a directory, and a file with extension .Rproj is created in the folder. The workspace and the R Studio window state is saved. That means that if you open a saved project it opens with the same files open in the editor and the same objects in the workspace as when it was closed. This is very handy. For those (like me) who don't want to be completely reliant on RStudio, it is nice to know that your script files are saved normally and the workspace is saved as a hidden file (*.Rdata), so you're not "locked in" to using RStudio when you use the project feature.

[45] To set the CRAN mirror go to "Tools>Global Options" (Mac: "RSTudio>Preferences") and select the "Packages" tab and select a CRAN mirror from the menu - the nearest one is probably best.

IV. Using R and LaTeX via Knitr

Those who need much more control in formatting, or who already use LaTeX will be glad to know that `knitr` supports the inclusion and execution of R code in the context of LaTeX documents. To explore the features of LaTeX is beyond the scope of this class, but we will demonstrate the basic features.

You can open a new .rnw file from the file menu: `File > New > Rsweave`. Or open the file "RNW-test.Rnw" in the "Code Files" directory. This is a minimal example.

Notice a few features: Code blocks begin with `<<>>=` and end with `@`. Chunk options can be specified between the `<<` and `>>=` that precede the code block. There are more chunk options for .rnw files than for .rmd files - the one that is most of interest to us is the `fig.cap`, which allows us to specify a caption for figures.

To compile this to .pdf, you need to configure RStudio a bit. Open RStudio Preferences, and select the "Sweave" tab. Under "Weave Rnw files using:" and select "knitr". You should now be able to click "Compile PDF" from the top of the editor pane.

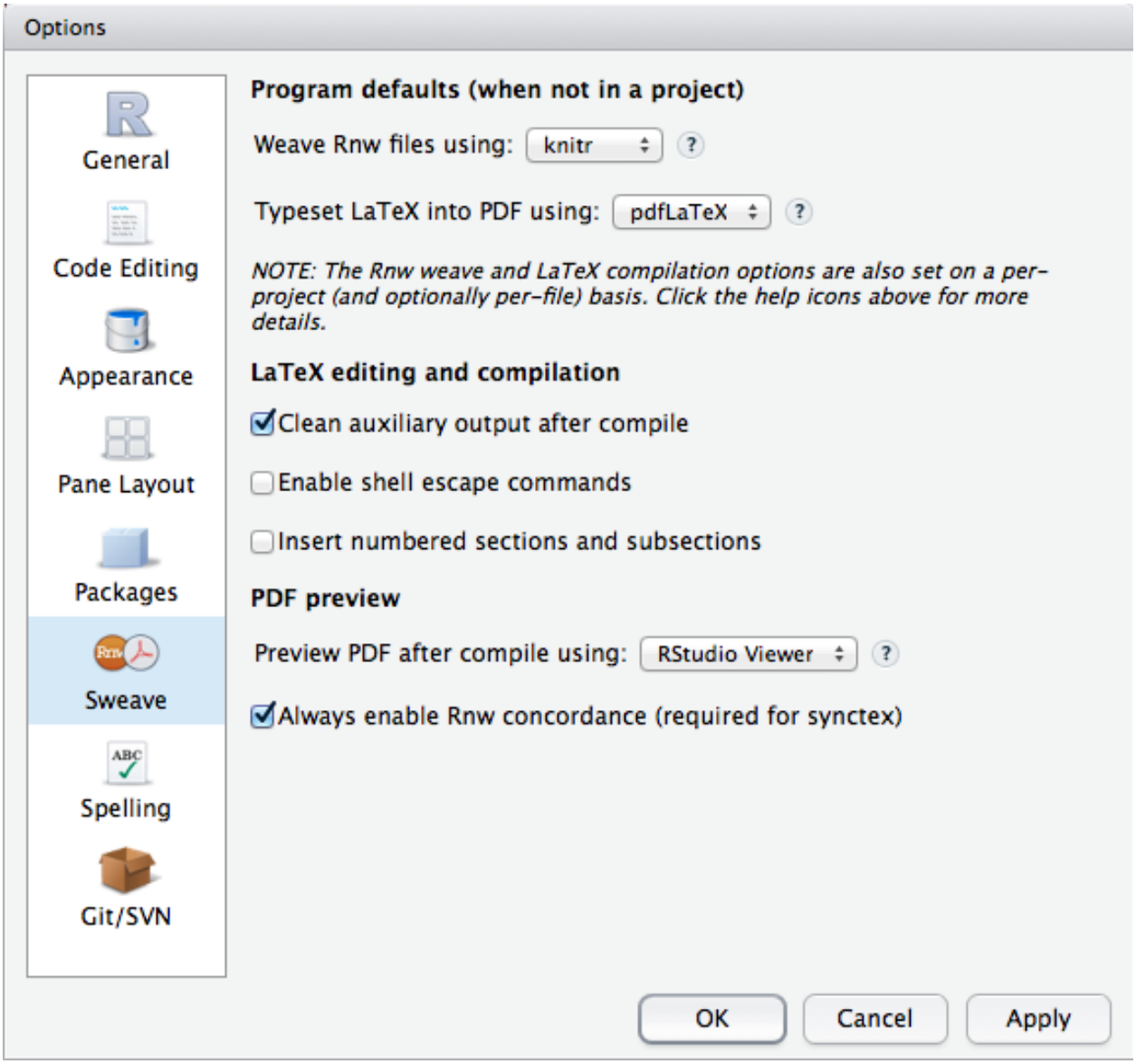


Figure 4: Preferences

There are many examples such as this online.

V. Exercises

1) Take the homework assignment from last chapter and reformat it as an .rmd document (this means convert all the commentary to text, using appropriate headings; code comments can be left as comments). Choose one graph and hide the code that produces it (only for this assignment!), and use inline code to show results from some model or calculation.

Chapter 15: Visualizing Data I

Enhancing scatter plots

I. Introduction

So far we have used R's graphics in fairly straightforward ways to examine data or look for model violations. But sometimes we need to do more. While creating publication quality plots is not something *everyone* will do, being able to create more complex graphics can be a great assist in data analysis, and the ability to customize graphics to visualize data is one of the strengths of R [46].

In this chapter and the next we build up some pretty complex figures from basic building blocks. Some of the examples are fairly complex - but my hope is that you can follow along to see what is possible by combining basic building blocks.

I chose the title "visualizing data" here because while one goal of making figures is communication, I often find (especially with more complex data) that I can understand the data better when I find the right way to visualize it. A resource that may be useful when considering how to visualize some data is the R graph gallery.

There are some R packages that provide more "advanced"" plotting interfaces (e.g. `ggplot2` and `lattice`), and you may want to have a look at these. Here, in the spirit of learning "essential" R, we'll focus on base R. Learning to use the basic plotting functions and graphical parameters can provide great flexibility in visualizing data. In this session we'll focus on scatter plots.

II. Basic Scatter Plots

A simple scatter plot

Here we'll look at the relationship between weed seed-bank density (number of weed seeds in soil) and weed density (number of growing weed plants) in conventional and herbicide-free plots. (This data set only contains the means).

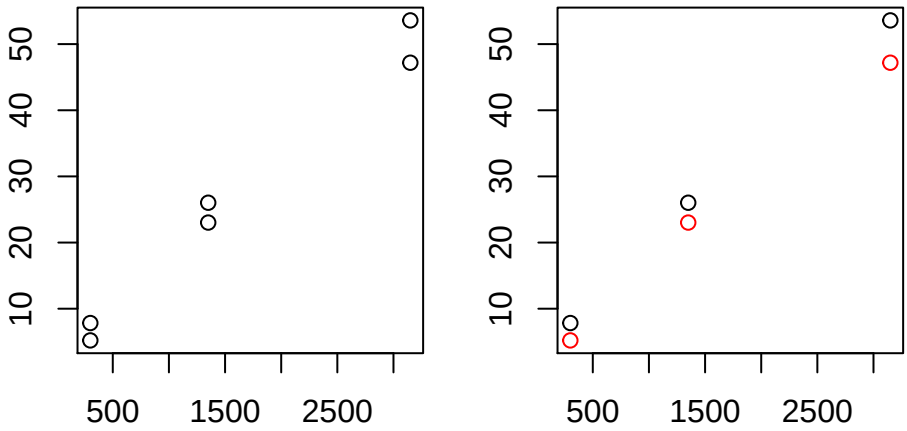
```
DT <- read.delim("../Data/DataVizEx1.txt")
summary(DT)
```

```
#  Den   Manag    SeedDen           TotDM             Den1
#  H:2   C:3   Min.   : 300.0   Min.   :  4.84   Min.   : 5.208
#  L:2   O:3   1st Qu.: 562.5   1st Qu.: 26.15   1st Qu.:11.630
#  M:2         Median :1350.0   Median : 45.05   Median :24.521
#              Mean   :1600.0   Mean   : 66.91   Mean   :27.142
#              3rd Qu.:2700.0   3rd Qu.: 85.81   3rd Qu.:41.896
#              Max.   :3150.0   Max.   :187.29   Max.   :53.583
#       Den2             Den3             DenF
#  Min.   : 4.292   Min.   : 0.800   Min.   : 2.120
#  1st Qu.: 7.375   1st Qu.: 3.799   1st Qu.: 4.942
#  Median :15.333   Median : 6.868   Median :10.197
#  Mean   :17.733   Mean   : 7.959   Mean   :11.297
#  3rd Qu.:26.604   3rd Qu.:11.674   3rd Qu.:17.609
#  Max.   :36.354   Max.   :17.167   Max.   :22.000
```

We have seed-bank density as a factor (`Den`) and as a continuous variable (`SeedDen`), and weed density (weeds per meter^2) at four time points (`Den1` to `DenF`). We'll look at the relationship between seed-bank density and weed density at the first count. We'll use `with()` to avoid attaching the data.

[46] In fact, I often see infographics online or in print publications that are almost certainly made with R

```
with(DT, plot(SeedDen, Den1))  # make the plot
plot(Den1 ~ SeedDen, data = DT, col = Manag)
```

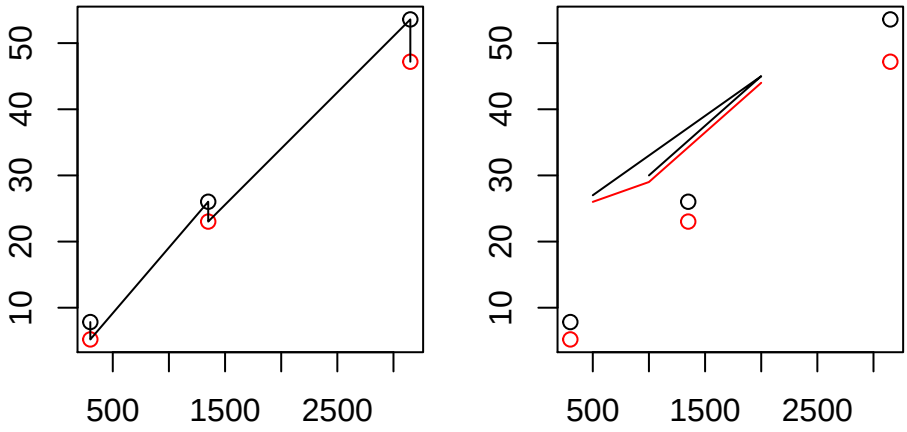


We have two points at each of three levels of weed seed-bank density. Is there a consistent difference between our two treatments - O(rganic) vs. C(onventional)? In the second plot we coded point color by treatment.

We can see that one set of points is consistently above the other. Which is which? They are coded as `C` and `O`, and the level that is first alphabetically (`C` in this case) will be represented in black, the next in red, and so on through all the colors in the palette (use `palette()` to view or change the palette).

Let's try adding some lines here - we'll use `lines()` draws lines (defined by vectors `x` and `y`) on an existing plot.

```
with(DT, plot(SeedDen, Den1, col = Manag))
with(DT, lines(SeedDen, Den1))
with(DT, plot(SeedDen, Den1, col = Manag))  # second plot
lines(x = c(1000, 2000, 500), y = c(30, 45, 27))  # not same as
lines(x = c(500, 1000, 2000), y = c(27, 30, 45) - 1, col = "red")  # -1 offset
```

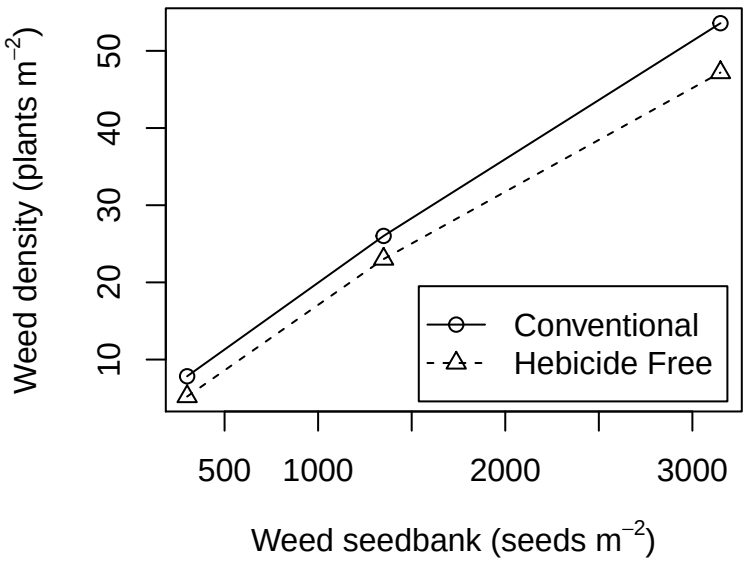


This line was not exactly what we had in mind. A quick look at the data is instructive - 300, 300, 1350, 1350, 3150, 3150 - there are 2 values for each x value. The line created by `lines()` follows the `x` vector literally - 2 at 300, 2 at 1350, etc. The *order of elements* in the `x` and `y` arguments matters. We can confirm this by plotting some arbitrary lines on the second plot (the second is offset -1 in the y direction for clarity). The same three points are plotted but in different order.

A simple scatter plot revisited

Let's try again with the same data. We'll use different symbols for our treatments, and we'll add proper axis labels.

```
op <- par(mar = c(4, 4.5, 0.5, 0.5))
sym <- c(21, 24)  # define vector of symbols
with(DT, plot(SeedDen, Den1, pch = sym[Manag], ylab = expression(paste("Weed density",
    " (plants ", m^-2, ")")), xlab = expression(paste("Weed seedbank (seeds ",
    m^-2, ")"))))  # plot with nice axis labels
lines(DT$SeedDen[DT$Manag == "C"], DT$Den1[DT$Manag == "C"], lty = 1)
# add line for trt C
lines(DT$SeedDen[DT$Manag == "O"], DT$Den1[DT$Manag == "O"], lty = 2)
# add line for trt O
legend("bottomright", inset = 0.025, pch = sym, lty = 1:2, c("Conventional",
    "Hebicide Free"))  # add legend
```



That is a basic, but clean plot that clearly shows the differences. There are quite a number of things to note here since we are including many common elements of quality plots (most of which we'll use repeatedly in this chapter and the next): 1. We selected 2 plotting symbols in our vector `sym`, and indexed these based on levels of the variable `Manag` (the `pch=sym[Manag]` in the plot command). "`pch`" means "plotting character"; see `?points` for a list of the plotting characters.
2. We used `expression()` to allow a superscript in our axis label, and `paste` to combine the mathematical expression (`m^-2`) with the plain text (`"Weed density (plants "` and `")"`) parts of the label [47].
3. We used logical extraction (`[DT$Manag=="C"]`) to specify which part of the data to plot for each of the two lines.
4. `lines()` creates a line from point to point, not a trend-line - we could use `abline(lm(De1~Den2,data=DT)` with the appropriate logical extraction for that).
5. while `legend()` can be complex, in many cases (like this one) it is pretty simple - we just specify line type (`lty`), plotting symbol (`pch`), and the legend text.
6. We used `par(mar=c(bottom,left,top,right))` to set the *margin* around each plot. `mar` is expressed in *lines of text*, so it changes as `cex` is changed (this assures that margins will be large enough for axis labels). Here because we want to put the plots near each other, we use low values for `top` and `bottom`, but for the first and last plot (respectively) where we need to have labels, we use a larger value for these. We could also use the `par` argument `oma` to set the outer margins - see `?par` for more detail.

Note: It is a worthwhile excercise is to take the code for this last plot and go through it line by

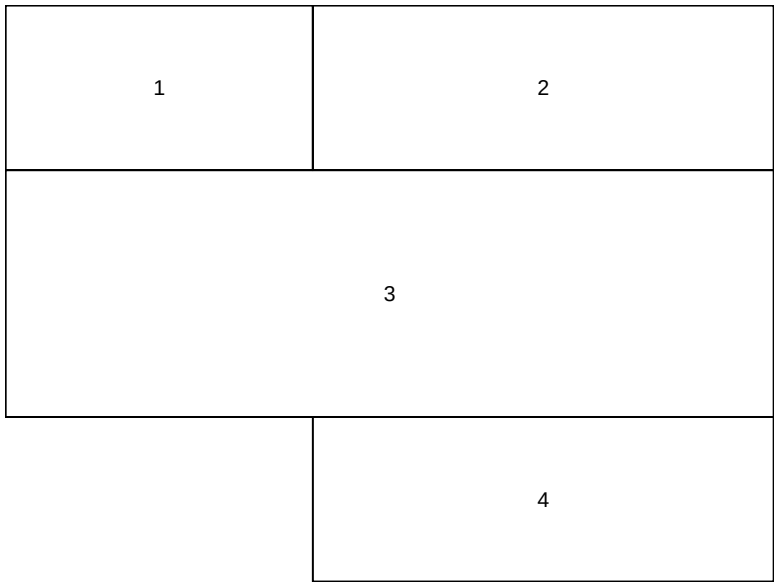
[47] Note that the text string `Weed density","  (plants "` is broken into 2 pieces - this is not necessary for R, but results in neater line wrapping in these notes.

line, commenting out lines and changing arguments to see how they work.

III. Multi-Panel Plots I: Layout

It is often the case that we want to show multiple graphs at once in a combined layout. We've seen how `par(mfrow=c(rows,columns))` can be used to split the plotting window. However, the function `layout()` provides a much more flexible tool to do this.

```
layout(matrix(c(1, 2, 3, 3, 0, 4), nrow = 3, byrow = TRUE), heights = c(1, 1.5,
    1), widths = c(1, 1.5))
layout.show(4)  # show the first 4 plots in the layout
```



A couple of things to notice here: 1. `layout()` takes as it's first argument a matrix of plot numbers
2. Plot widths and heights can be manipulated
3. Not all parts of the plotting window need to contain plots
4. `layout.show()` lets us see the layout
Since this layout contains 4 plots, the next 4 times the plot command is called (unless `new=TRUE` is used to force over-plotting - see `?par` or the discussion of adding a second y axis near the end of this chapter for more).

I often find it very useful to force my plot to a specific size. This lets me be sure that symbol and text size are as I want them to be. There are several ways to do this. For developing a plot I use the function `quartz(title,height,width)` (quartz is OSX only) or `x11(title,height,width)` (Linux or Windows) to open a new plotting window whose size I can control. Alternately, in RStudio you can choose "Save plot as image" from the "Export" menu on the "Plots" tab, and then specify the dimensions you want for the plot. If using markdown the chunk options `fig.width` and `fig.height` allow you to control plot size.

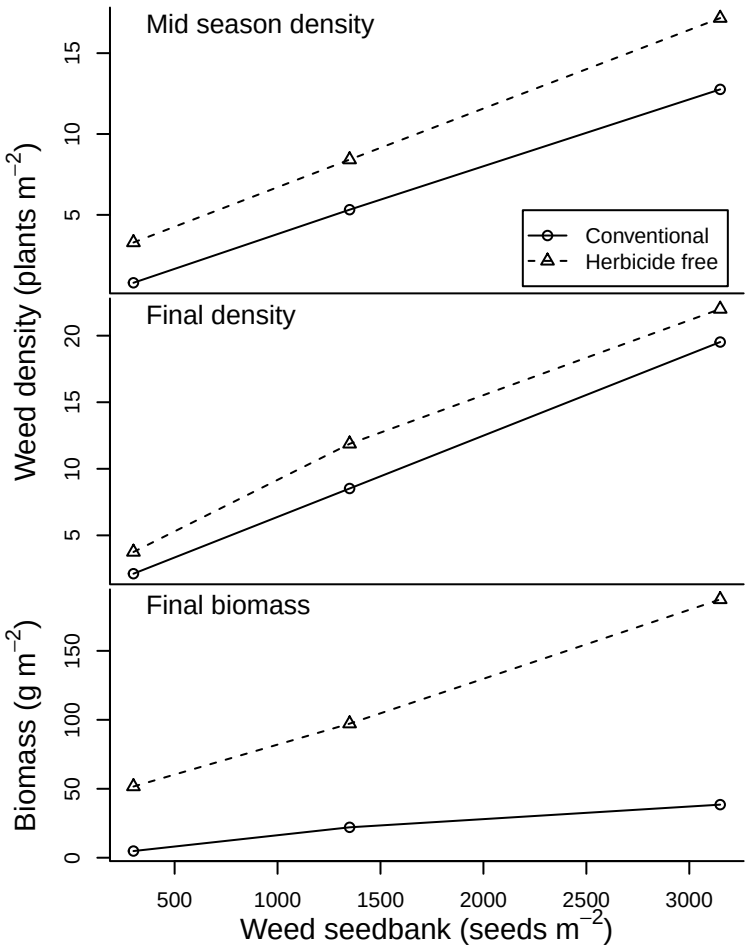
Here we'll make a multi-panel figure with 3 plots. Each panel will be similar to the scatter plot we made in part 2, but will show other response variables, and the response variables are called by column number rather than by name.

Note: When developing complex graphics, particular with multiple panels, it is very useful to control the size of your graphics window, as the interaction of text size and size of the graphics device can change the appearance of figures. By forcing R to open a new graphics device you can specify the size of the device. This can be done using the commands `quartz()` or `x11()` on Mac and Windows/Linux respectively. For this example you'll want to use `quartz(title="Density Dependence",height=8,width=4)` or `x11(title="Density Dependence",height=8,width=4)` to create the appropriately sized plotting window (sizes specified in

inches not cm).[48] This relates to another advantage of using R markdown vs. notebook (.Rmd vs. .R) for documenting work - in markdown there are chunk options that let you control figure size.

```
layout(matrix(c(1, 2, 3), nrow = 3), heights = c(1, 1, 1.3))
sym <- c(21, 24)  # plotting characters to use
par(mar = c(0.1, 4.3, 0.1, 1), bty = "l")  # set margins and plot frame type
## plot 1
with(DT, plot(SeedDen, DT[, 7], pch = sym[Manag], xaxt = "n", xlab = "", ylab = ""))
lines(DT$SeedDen[DT$Manag == "C"], DT[DT$Manag == "C", 7], lty = 1)
lines(DT$SeedDen[DT$Manag == "O"], DT[DT$Manag == "O", 7], lty = 2)
text(300, max(DT[, 7]) * 0.97, "Mid season density", pos = 4, cex = 1.2)
mtext(side = 2, line = 2.5, at = -1, text = expression(paste("Weed density (plants ",
    m^-2, ")")), cex = 0.9)
legend("bottomright", inset = 0.025, pch = sym, lty = 1:2, c("Conventional",
    "Herbicide free"))
## plot 2
with(DT, plot(SeedDen, DT[, 8], pch = sym[Manag], xaxt = "n", xlab = "", ylab = ""))
lines(DT$SeedDen[DT$Manag == "C"], DT[DT$Manag == "C", 8], lty = 1)
lines(DT$SeedDen[DT$Manag == "O"], DT[DT$Manag == "O", 8], lty = 2)
text(300, max(DT[, 8]) * 0.97, "Final density", pos = 4, cex = 1.2)
## plot 3
par(mar = c(4.1, 4.3, 0.1, 1))  # change margins
with(DT, plot(SeedDen, DT[, 4], pch = sym[Manag], xaxt = "n", xlab = "", ylab = ""))
lines(DT$SeedDen[DT$Manag == "C"], DT[DT$Manag == "C", 4], lty = 1)
lines(DT$SeedDen[DT$Manag == "O"], DT[DT$Manag == "O", 4], lty = 2)
text(300, max(DT[, 4]) * 0.97, "Final biomass", pos = 4, cex = 1.2)
axis(side = 1, at = seq(500, 3000, 500))
mtext(side = 1, line = 2.5, text = expression(paste("Weed seedbank (seeds ",
    m^-2, ")")), cex = 0.9)
mtext(side = 2, line = 2.5, text = expression(paste("Biomass (g ", m^-2, ")")),
    cex = 0.9)
```

[48] I have no idea why inches and not cm.



This is a fair amount of code for just one figure (though a lot of it is repeated). Things to notice: 1. `legend()` was only needed once because all the plots have the same legend.
2. the graphical argument `xaxt="n"` was used to suppress the plotting of the x axis in all the plots, and we used `axis()` to add the x axis only to the bottom plot (#5).
3. To keep x and y axis labels from printing we used `xlab=""` and `ylab=""`.
4. The "Weed density" y axis label was added in plot 2 using `mtext()` - in principle it could have been added in any of the plots, since the `at` argument allows us to specify where on the axis it will be centered. `at=2` specifies centering at y=2. (Panels 1 & 2 have the same y axis units so we've centered the label over both of them).
5. The height of the last plot was greater - this is to allow space for the x axis and labels. Alternately we could use `par(mfrow=c(3,1),oma=c(4.1,0,0,0),mar=c(0.1,4.3,0.1,1),bty="l")` to create space in the "outer margin", and omit the calls to `layout()` and the other calls to `par()`.

Loops for multi-panel figures.

The preceding plot required ~30 lines of code. This is typical for multi-panel plots, at least using base R. `ggplot2` and `lattice` have functions that make something like this simpler, but I usually stick with base R, partly because I know I can add elements to the plot. Notice that when you look at the code, most of it appears repeatedly with only minor variations. This might suggest using a loop. We won't re-create the plot here for brevity's sake, as it is essentially identical to the above plot . But I encourage you to run this code and look at how it works - you'll want to size the plotting window as discussed above.

```
par(mfrow = c(3, 1), oma = c(4.1, 0, 3, 0), mar = c(0.1, 4.3, 0.1, 1), bty = "l")
vars <- c(7, 8, 4)  # the column numbers for each panel's response variable
```

```
sym <- c(21, 24)  # plotting characters to use
labs <- c("", "", "", "Final biomass", "Rye", "Rye termination", "Mid season density",
    "Final density")  # plot labels
for (i in vars) {
    # begin loop for panels
    with(DT, plot(SeedDen, DT[, i], pch = sym[Manag], xaxt = "n", xlab = "",
        ylab = ""))  # plot the ith column
    lines(DT$SeedDen[DT$Manag == "C"], DT[DT$Manag == "C", i], lty = 1)  # add lines
    lines(DT$SeedDen[DT$Manag == "O"], DT[DT$Manag == "O", i], lty = 2)
    text(300, max(DT[, i]) * 0.97, labs[i], pos = 4, cex = 1.2)
    if (i == 4) {
        # add x axis for the last plot only (i==4)
        axis(side = 1, at = seq(500, 3000, 500))
        mtext(side = 1, line = 2.5, text = expression(paste("Weed seedbank (seeds ",
            m^-2, ")")), cex = 0.9)
        mtext(side = 2, line = 2.5, text = expression(paste("Biomass (g ", m^-2,
            ")")), cex = 0.9)
    }
    if (i == 7)
        {
            mtext(side = 2, line = 2.5, at = -1, text = expression(paste("Weed density",
                "(plants ", m^-2, ")")), cex = 0.9)  # y axis label for first 4 plots
            legend(1100, 22, legend = c("Conventional", "Herbicide free"), pch = sym,
                lty = c(1, 2), ncol = 2, xpd = NA)
        }  # y axis for last plot
}  # end loop for panels
```

Things to notice: 1. the loop is indexed by the vector `vars` which refers to the column numbers of the response variables.
2. the legend in the first plot is outside the plot area - the argument `xpd=NA` allows this, and the upper outer margin (`oma`) is set to give space for the legend.
3. the code to create the x-axis for the final plot could have been moved outside the loop, avoiding another `if()`.
Was it worth it? The code was somewhat reduced - 18 vs 24 lines of code. I'm not sure that justifies the added complexity. The main benefit (in my experience) is that the "guts" of the plot are only here once - this makes it *much* simpler to change something (plotting character or line type) and keep all the panels consistent. This suggests that it is more likely to be worth using a loop for a figure if all panels are very similar and when there are many panels.

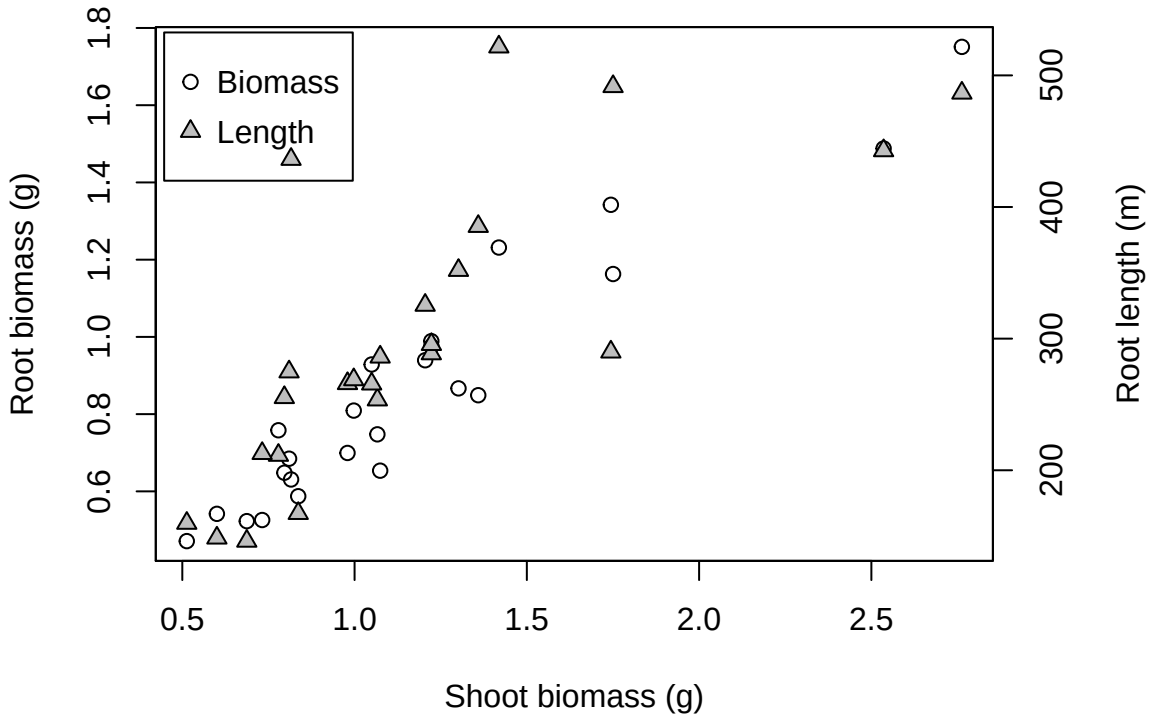
VI. Adding a Secondary y-axis

Occasionally we want to add a second y-axis to a plot to plot more than one response variable. Since `plot()` also creates a new coordinate space with appropriate units, this does not seem that it would help us.

One way this can be done is using `points()` to add the second variable, but first we'd need to convert the second response variable to the same scale as the first, and we might need to fool around with the `ylim` argument in the initial plot command. Workable, but probably a hassle.

There is an easier way, though it may seem counter-intuitive. One of the arguments to `par()` is `new`. A value of `new=TRUE` (counter-intuitively) tells R to treat the current graphics device *as if it were a new device*. This means that a new coordinate space is calculated, and a second plot can be made without the first having been erased.

```
op <- par(mar = c(4.1, 4.1, 2, 4.1))
beans <- read.csv("../Data/BeansData.csv", comm = "#")  # load data
with(beans, plot(ShtDM, RtDM, xlab = "Shoot biomass (g)", ylab = "Root biomass (g)"))
par(new = TRUE)
with(beans, plot(ShtDM, rt.len, xaxt = "n", yaxt = "n", ylab = "", xlab = "",
    pch = 24, bg = "grey"))
axis(side = 4)
mtext(side = 4, line = 3, "Root length (m)")
par(op)
legend("topleft", inset = 0.01, pch = c(21, 24), pt.bg = c("white", "grey"),
    legend = c("Biomass", "Length"), xpd = NA)
```



Notice that the initial `par()` statement includes an assignment: this saves the *original* settings, which are restored with the final `par(op)` statement. Also note the `xpd=NA` in the `legend()` command - the boundaries of the second plot are not exactly the same as the first. If you run this command without the `xpd=NA` you will see why it was added. NOTE: These two y variables may not really good candidates for this type of presentation!

In the last two sessions we have covered the most commonly used graphical tools in R (well, in base-R anyway). These are most of the tools you need to make most of the figures you might need to make, and enough base to learn how to make others.

VII. Summary

In this chapter we've looked at several scatterplots and at how lines, points, and secondary axes can be added to plots. In addition we've explored creating multi-panel plots. These are pretty basic tools which can be applied to a wide range of graphics.

VIII. Exercises

1) We'll revisit the electric bill data once more. In the Chapter 13 exercises we fit an ANCOVA to this data. Plot this model (not the residuals), showing the two curves and two parts of the data with distinct symbols, and with properly formatted axes and labels (i.e., Kilowatt hours per day should be shown as "KWHd^{-1}".)

2) Using the electric bill data, plot daily energy use (`kWhd.1`) as a function of average temperature (`avgT`). Add a second y axis to show cost. Include a legend.

Chapter 16: Visualizing Data II

Errorbars and polygons

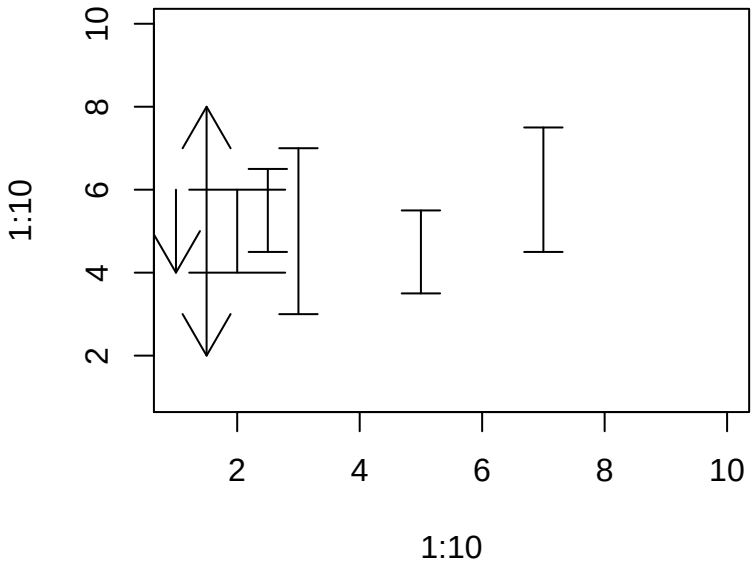
I. Introduction

In the last chapter we built some rather complex multiple-panel scatter plots. Here we'll consider some additional ways that scatter plots (x,y plots) can be enhanced, principally by adding error bars and polygons or ribbons.

II. Scatter Plot with Error Bars

This figure will illustrate an ANCOVA between the biomass of a rye crop and the the date at which it was cut (covariate) for two different rye planting dates (factor variable). In this figure we'll show the variability of biomass at each cutting date using error bars, and show the ANCOVA line for each of the two levels of planting date in two years. First we'll introduce the function `arrows()` which we'll use to make error bars.

```
op <- par(mar = c(4, 4, 0.5, 0.5))
plot(1:10, 1:10, type = "n")
arrows(x0 = 1, y0 = 6, x1 = 1, y1 = 4)
arrows(1.5, 8, 1.5, 2, code = 3)
arrows(2, 6, 2, 4, code = 3, angle = 90)
arrows(2.5, 6.5, 2.5, 4.5, code = 3, angle = 90, length = 0.1)
x <- c(3, 5, 7)
y <- c(5, 4.5, 6)
z <- c(2, 1, 1.5)
arrows(x, y - z, x, y + z, code = 3, angle = 90, length = 0.1)
```



Things to notice: 1. `arrows()` takes four points, `x0,y0,x1,y1` that define the ends of the arrows.
2. `code=3` puts arrows at both ends of the arrow
3. `angle=90` makes the arrow look like an error bar
4. `length` controls the length of the arrowhead or crossbar
5. `arrows()` is happy to take a vector as an argument, allowing one call to `arrows()` to create multiple error bars.

To make the plot we first need to load the data and examine it.

```
RyeMeans <- read.delim("../Data/Rye ANCOVA.txt", comment = "#")
head(RyeMeans)  # examine the data
```

```
#   Term.DOY    YrPd MeanDM   DMsd  DMse n
# 1      117 2008 P1  380.0  81.07 28.66 8
# 2      137 2008 P1  674.3  88.42 31.26 8
# 3      128 2008 P1  590.0  78.25 27.66 8
# 4      149 2008 P1  834.0 131.10 46.36 8
# 5      137 2008 P1  673.3  90.60 32.03 8
# 6      155 2008 P1  984.0 200.90 71.01 8
```

```
RyeMeans$Term.DOY  # not in order
```

```
#  [1] 117 137 128 149 137 155 117 137 128 149 137 155 118 142 128 152 140
# [18] 160 118 142 128 152 140 160
```

```
RyeMeans <- RyeMeans[order(RyeMeans$Term.DOY), ]  # sort the data by Term.DOY
## PLOT
range(RyeMeans$MeanDM + RyeMeans$DMse)  # ~110 to ~1100)
```

```
# [1]  111.885 1115.230
```

```
range(RyeMeans$Term.DOY)  # find x range (~115-160)
```

```
# [1] 117 160
```

```
levels(RyeMeans$YrPd)
```

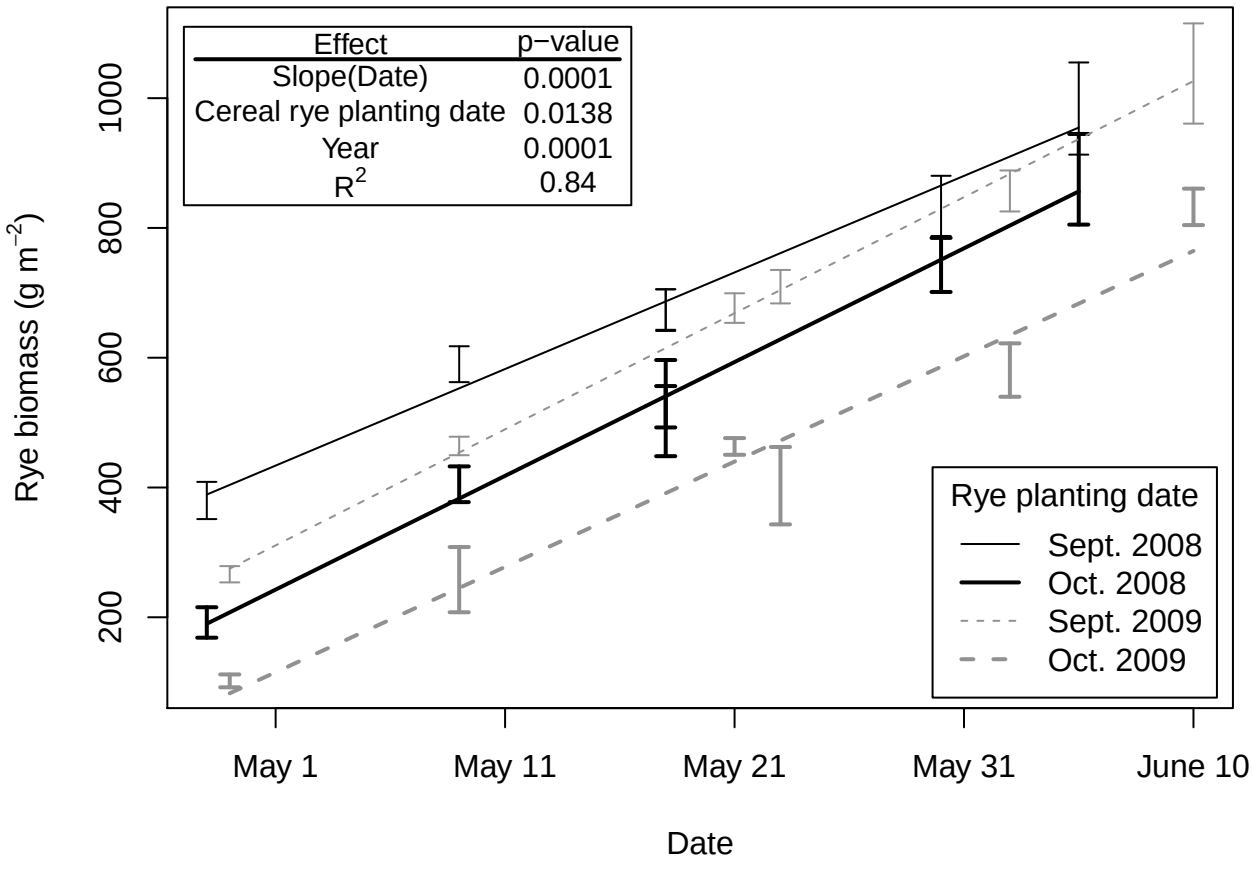
```
# [1] "2008 P1" "2008 P2" "2009 P1" "2009 P2"
```

Things to notice: 1. We sorted the data frame using `RyeMeans<-RyeMeans[order(RyeMeans$Term.DOY),]`. This is just standard R indexing: before the comma we specify the rows; here we just say the rows given by the function `order()` applied to `RyeMeans$Term.DOY`.
2. The variable `YrPd` lists the year and the planting date together as a single factor. This is a convenience for plotting. The original ANCOVA was fit with the year and rye planting date as separate factors, and included their interaction.

Now we'll plot the data. As before, if you run this code you may want to set your figure size to `width=6.75,height=5.25`.

```
### pdf(height=6.25,width=8.5)
op <- par(mar = c(5, 5, 3, 1))  # save the par settings and set margin settings.
with(subset(RyeMeans, YrPd == "2008 P1"), plot(Term.DOY, MeanDM, ylim = c(100,
    1100), xlim = range(RyeMeans$Term.DOY), ylab = expression(paste("Rye biomass (g ",
    m^{
        -2
    }, ")")), xlab = "Date", type = "n", xaxt = "n"))  # subest not necessary here
## for dates on X axis, use this, and add xaxt='n' to plot cmd above
axis(side = 1, at = seq(120, 160, by = 10), labels = c("May 1", "May 11", "May 21",
    "May 31", "June 10"))
## add error bars for termination date (Term.DOY) in each treatment group
## (YrPd).
with(subset(RyeMeans, YrPd == "2008 P1"), arrows(Term.DOY, MeanDM + DMse, Term.DOY,
    MeanDM - DMse, length = 0.05, angle = 90, code = 3, lwd = 1, col = "black"))
with(subset(RyeMeans, YrPd == "2008 P2"), arrows(Term.DOY, MeanDM + DMse, Term.DOY,
    MeanDM - DMse, length = 0.05, angle = 90, code = 3, lwd = 2, col = "black"))
with(subset(RyeMeans, YrPd == "2009 P1"), arrows(Term.DOY, MeanDM + DMse, Term.DOY,
    MeanDM - DMse, length = 0.05, angle = 90, code = 3, lwd = 1, col = "grey57"))
```

```
with(subset(RyeMeans, YrPd == "2009 P2"), arrows(Term.DOY, MeanDM + DMse, Term.DOY,
    MeanDM - DMse, length = 0.05, angle = 90, code = 3, lwd = 2, col = "grey57"))
legend("bottomright", inset = 0.015, legend = c("Sept. 2008", "Oct. 2008", "Sept. 2009",
    "Oct. 2009"), lwd = c(1, 2, 1, 2), col = c("black", "black", "grey57", "grey57"),
    lty = c(1, 1, 2, 2), title = "Rye planting date")
## ADD lines
endpoints <- data.frame(Term.DOY = c(117, 155, 117, 155, 118, 160, 118, 160),
    YrPd = rep(c("2008 P1", "2008 P2", "2009 P1", "2009 P2"), each = 2))
endpoints <- cbind(endpoints, predict(lm(MeanDM ~ Term.DOY * YrPd, data = RyeMeans),
    newdata = endpoints))
lines(endpoints[1:2, 1], endpoints[1:2, 3], lwd = 1, col = "black")
lines(endpoints[3:4, 1], endpoints[3:4, 3], lwd = 2, col = "black")
lines(endpoints[5:6, 1], endpoints[5:6, 3], lwd = 1, col = "grey57", lty = 2)
lines(endpoints[7:8, 1], endpoints[7:8, 3], lwd = 2, col = "grey57", lty = 2)
## ADD ANOVA table
legend(121, 1137, legend = c("Effect", "Slope(Date)", "Cereal rye planting date",
    "Year", expression(R^2)), bty = "n", adj = 0.5, cex = 0.9)  # adj=0.5:centered text
legend(130.5, 1137, legend = c("p-value", "0.0001", "0.0138", "0.0001", "0.84"),
    bty = "n", adj = 0.5, cex = 0.9)
rect(116, 835, 135.5, 1110)
lines(x = c(116.5, 135), y = c(1060, 1060), lwd = 2)
```




```
### dev.off()
```

There is an informative plot that shows the variability in the y-axis for each group at each poit in the x-axis. While you never may need to make a plot like this, some elements of this are likely to be useful - for example a regression line that doesn't extend beyond the x-axis range of the data would be generally useful.

Things to notice: 1. the use of `type="n"` to create a blank plot - even though nothing has been plotted, the coordinate space has been created, so `points()`, `lines()`, and `arrows()` can be used.
2. The best fit lines were added by calculating the starting and ending x and y values using `predict()` on the ANCOVA model rather than the (much simpler) alternative of using `abline()`. This was done to avoid having the line extrapolate beyond the data.
3. `"grey57"` means a grey that is 57% white, so `"grey90"` is a light grey and `"grey10"` a dark grey.
4. Adding the ANCOVA table to the plot with `legend()` required a bit of trial en error to get the spacing right. The function `addtable2plot()` in the package "plotrix" might make this easier.[49]
5. There are two lines commented out with ###: pdf(...) and dev.off(). If they were run they would open a pdf graphic device (`pdf()`) of specified size and close that device (`dev.off()`), and all code between them would be sent to that device. Rather than create a plot you can view while making it, they would create a .pdf file. RStudio has nice tools for exporting plots, but is is good to know how to write directly to pdf.

III. Scatter Plots with Confidence Ribbons

One of the wonderful things about R graphics is how anything is possible. Here we'll make a scatter plot of some data and visualize the confidence interval for around that data with a ribbon, which we'll make using `polygon()` for drawing arbitrary polygons. This example is a plot showing the effect of limestone gravel on the pH of acidic forest soils in PA. Soil was sampled at varying distances (`dist`) from the road[50] on several forest roads in PA, which were surfaced either with shale or limestone gravel.

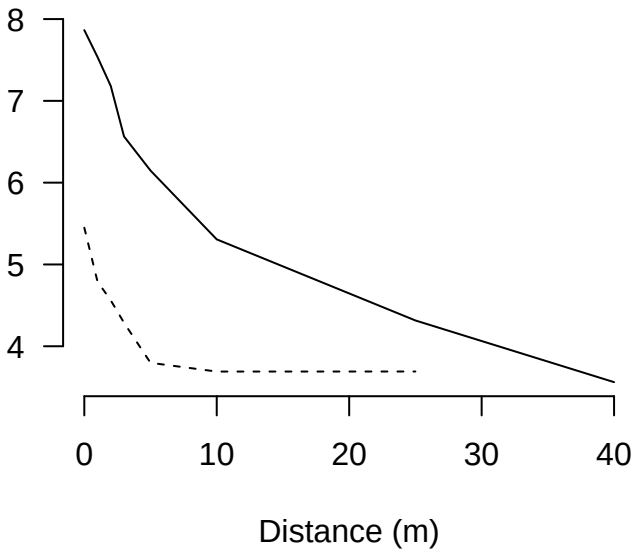
```
pHmeans <- read.table("../Data/ph data.txt", header = TRUE, sep = "\t")
pHmeans <- pHmeans[pHmeans$side == "down", -3]  # simplify the data
head(pHmeans)
```

```
#    trt dist     mean     stdev count      sterr      CI95
# 1    L    0 7.864583 0.2946167    12 0.08504851 0.1666951
# 3    L    1 7.536000 0.4277008    12 0.12346659 0.2419945
# 5    L    2 7.180917 0.4435590    12 0.12804447 0.2509672
# 7    L    3 6.564250 0.5775116    12 0.16671324 0.3267580
# 9    L    5 6.147750 0.7776220    12 0.22448014 0.4399811
# 11   L   10 5.306583 1.0053788    12 0.29022787 0.5688466
```

The data table here includes mean, standard deviation, standard error, and the 95% confidence interval for the mean. First we'll make a basic version of the plot. As above you'll want to set height of the graphics device to 5 inches and the width to 4.

```
xl <- range(pHmeans$dist)  # x limits
op <- par(mar = c(4.1, 4.1, 1, 1))  # set par()
with(subset(pHmeans, trt == "L"), plot(dist, mean, xlab = "Distance (m)", ylab = "",
    type = "l", ylim = range(pHmeans$mean), frame = FALSE, las = 1))  # plot
with(subset(pHmeans, trt == "S"), lines(dist, mean, lty = 2))
```

[49] Though using `legend()` is an improvement over just using `text()`, which is how I did it at first.
[50] The measurements were made on both sides of the road (up and down hill), but here we'll use just one.

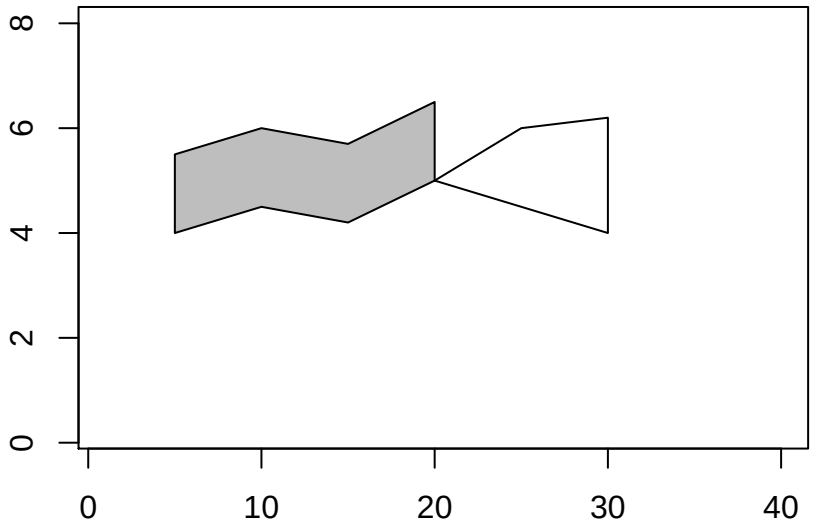



```
par(op)
```

Some things to notice: 1. The argument `las` forces axis tick labels to be horizontal. 2. the `frame=FALSE` argument suppresses the "box" around the plot.

In order to show the variability around the mean we'll use the function `polygon()`.

```
plot(1:40, (1:40)/5, type = "n")
polygon(x = c(20, 25, 30, 30, 20), y = c(5, 6, 6.2, 4, 5))
x <- c(5, 10, 15, 20)
y <- c(4, 4.5, 4.2, 5)
polygon(x = c(x, rev(x)), y = c(y, rev(y + 1.5)), col = "grey")
```



Notice that `polygon()` can create any arbitrary polygon as long as the number of x and y values are the same. The second `polygon` command shows how we can create a "ribbon" using `c()` and `rev()`. We'll use the same approach to calculate the ribbons for the limestone and shale roads on the up-slope and down-slope sides of the road.

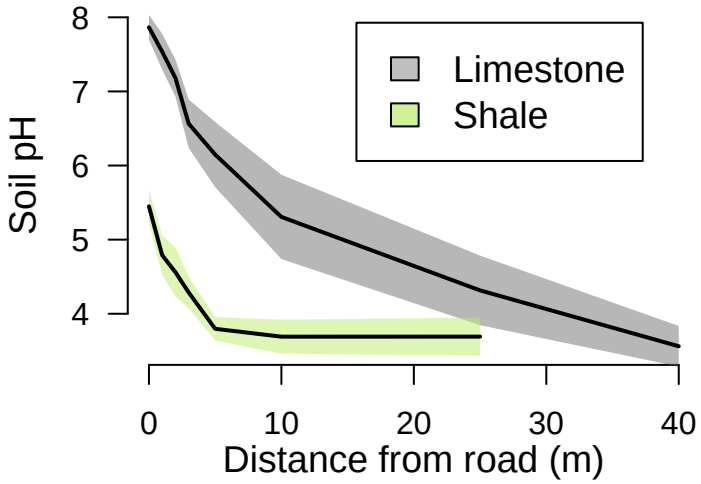
```
L <- with(subset(pHmeans, trt == "L"), c(mean + sterr * 1.96, rev(mean - sterr *
    1.96)))
S <- with(subset(pHmeans, trt == "S"), c(mean + sterr * 1.96, rev(mean - sterr *
```

```
    1.96)))
dds <- with(subset(pHmeans, trt == "S"), c(dist, rev(dist)))
ddl <- with(subset(pHmeans, trt == "L"), c(dist, rev(dist)))
```

Notice that we're using `subset()` to select the relevant part of the data, and then using `c()` and `rev()` to put together the top and the bottom y values (lines 2-5) and to create the x values (last 2 lines). In my head "L" stands for limestone and "S" for shale, and "dds" for distances for shale. The value of 1.96 in the first two lines here is the Z value for a 95% confidence interval.

```
lime = rgb(t(col2rgb("grey44")), alpha = 128, max = 255)  # lime='#70707080'
shal = rgb(t(col2rgb("darkolivegreen2")), alpha = 128, max = 255)  # '#BCEE6880'
xl <- range(pHmeans$dist)
op <- par(mar = c(4.1, 4.1, 1, 1))

with(subset(pHmeans, trt == "L"), plot(dist, mean, xlab = "", ylab = "", type = "n",
    ylim = c(3.5, 8.25), xlim = xl, las = 1, frame = FALSE))
polygon(ddl, L, col = lime, border = NA)
polygon(dds, S, col = shal, border = NA)
with(subset(pHmeans, trt == "L"), lines(dist, mean, xlab = "", lty = 1, lwd = 2))
with(subset(pHmeans, trt == "S"), lines(dist, mean, xlab = "", lty = 1, lwd = 2))
legend("topright", inset = 0.1, fill = c(rgb(r = 0.5, g = 0.5, b = 0.5, alpha = 0.5),
    rgb(r = 0.73, g = 0.93, b = 0.41, alpha = 0.7)), legend = c("Limestone",
    "Shale"), cex = 1.2)
mtext(text = "Soil pH", side = 2, cex = 1.2, line = 2.2)
mtext(text = "Distance from road (m)", side = 1, line = 2, cex = 1.2)
```



Things to notice here: 1. We want semi-transparent colors in case our ribbons overlap. We use `col2rgb()` to get the rgb colors that correspond to the R colors, and `rgb()` to specify colors with transparency (`alpha<1`). 2. Since `col2rgb` returns rows and `rgb()` requires columns, `t()` transposes the rows of `col2rgb()` to columns. 3. `col2rgb` returns values from 0-255, so we tell `rgb()` that `max=255` - the default for `rgb()` is a 0-1 scale. 4. We created the polygons first and plotted the lines on top.

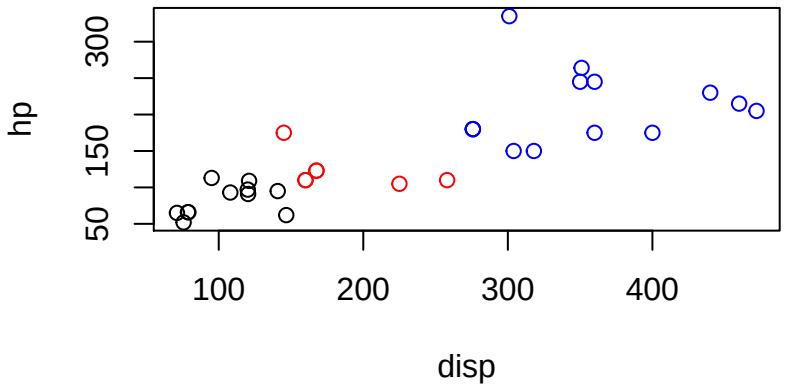
The package `ggplot2` has tools for automatically adding ribbons, but now you know how to manipulate arbitrary polygons.

IV. Error Bars in 2 Dimensions

Sometimes we might want to show errorbars in two dimensions. This is not particularly difficult, it just uses more of what we learned last chapter. We'll demonstrate with the `mtcars` data, and look at horsepower (`hp`)

and displacement (`disp`) for cars with differing number of cylinders (`cyl`).

```
data(mtcars)
cols = c("black", "red", "blue")
with(mtcars, plot(disp, hp, col = cols[cyl/2 - 1]))
```

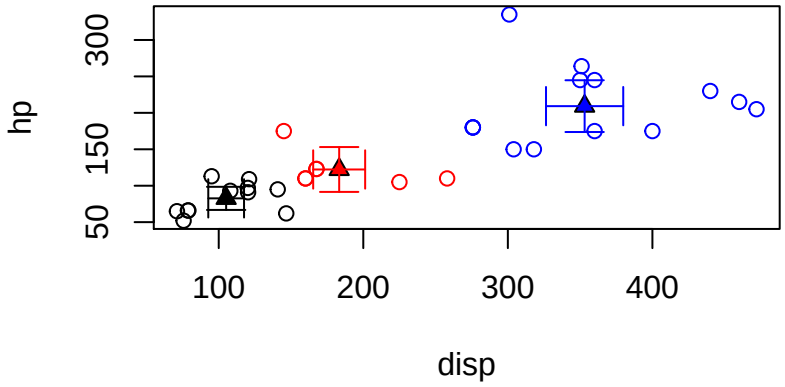


As a quick check we can plot the data and see we have three clusters. How different are they - are their means "significantly different"? First we'll calculate means and standard errors for each group (see Chapter 9).

```
car.means <- aggregate(mtcars[, 3:4], by = mtcars[2], mean)
car.means[, 4:5] <- aggregate(mtcars[, 3:4], by = mtcars[2], function(x) 1.96 *
    sd(x)/sqrt(length(x)))[2:3]
names(car.means)[4:5] <- c("hp.CI", "disp.CI")
```

Now we can plot this data and add errorbars. For interest, let's plot it on top of our previous plot.

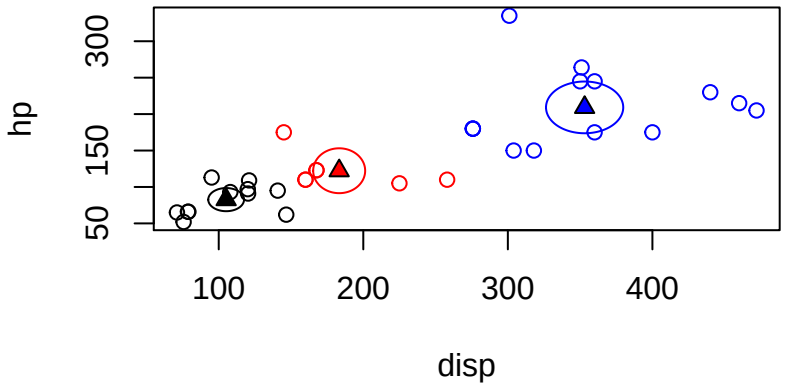
```
with(mtcars, plot(disp, hp, col = cols[cyl/2 - 1]))
with(car.means, points(disp, hp, pch = 24, bg = cols[cyl/2 - 1]))
with(car.means, arrows(disp, hp - hp.CI, disp, hp + hp.CI, code = 3, length = 0.1,
    angle = 90, col = cols[cyl/2 - 1]))
with(car.means, arrows(disp - disp.CI, hp, disp + disp.CI, hp, code = 3, length = 0.1,
    angle = 90, col = cols[cyl/2 - 1]))
```



An alternate way to show the errorbars in 2 dimensions is using an ellipse. The package `shape` has a function for plotting ellipses that we can use. You'll need to run `install.packages("shape")` if you haven't already done so.

```
library(shape)
with(mtcars, plot(disp, hp, col = cols[cyl/2 - 1]))
with(car.means, points(disp, hp, pch = 24, bg = cols[cyl/2 - 1]))
for (i in 1:3) {
    with(car.means, plotellipse(rx = disp.CI[i], ry = hp.CI[i], mid = c(disp[i],
```

```
        hp[i]), lcol = cols[i], lwd = 1))
}
```



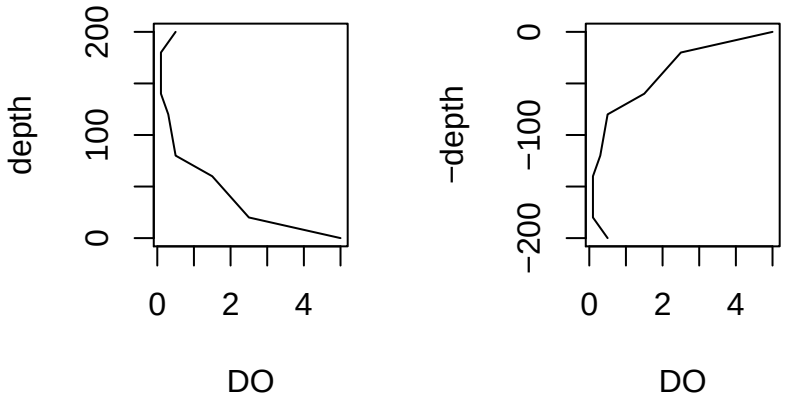
Unlike most R functions, `plotellipse()` doesn't seem to be happy with vectors for the arguments (at least the argument `mid`), thus the loop. There are other ways to do this, like directly coding an ellipse.

It is important to recall that the ellipses or error bars in these last figures are for the *means*. We don't expect that new samples drawn from these populations will fall within these bounds - in fact, few of our individual samples fall within them.

V. Reversing Axes

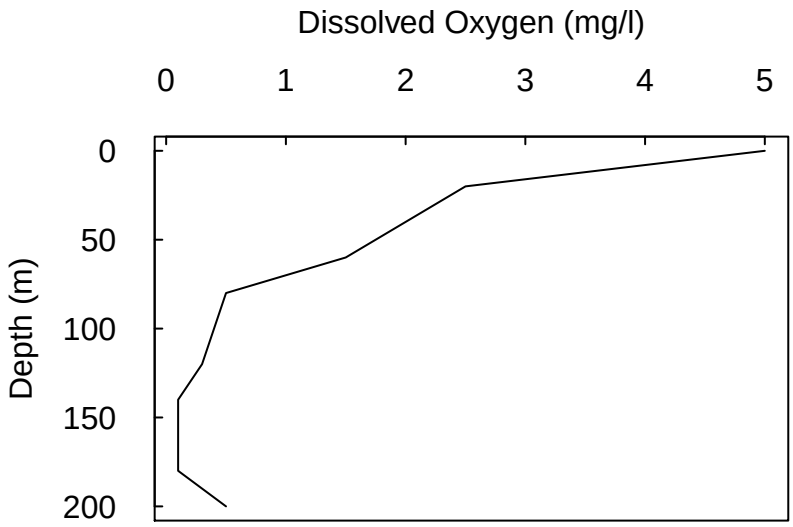
On occasion it make sense to reverse an axis for some reason - maybe to plot something that varies with depth (dissolved oxygen in ocean water for example). We'll produce a (crude) facsimile of this figure to demonstrate how to approach this. We'll begin by making up some data.

```
depth <- seq(0, 200, 20)
DO <- c(5, 2.5, 2, 1.5, 0.5, 0.4, 0.3, 0.1, 0.1, 0.1, 0.5)
plot(DO, depth, type = "l")
plot(DO, -depth, type = "l")
```



A simple call to plot is fine for basic visualization, but it is "upside down", rather than what we really want to see. Even plotting the negative of depth only gets us partway to our goal - the axis labels on the y axis should be positive, and the x-axis should be on top.

```
plot(DO, depth, type = "l", xaxt = "n", xlab = "", ylab = "Depth (m)", ylim = c(200,
    0), las = 1)
axis(side = 3)
mtext(side = 3, line = 2.5, "Dissolved Oxygen (mg/l)")
```



Note that here we reversed the default outside placement of tick marks using the argument `tck=0.02`. The value 0.02 is fraction of the plot width. We also placed the tick mark labels horizontally using the argument `las=1`.Also note that `mtext()` can take vectors as inputs (hardly surprising since this is R). One would use the same approach we used here for reversing the y-axis to reverse the x-axis.

VI. Summary

In this chapter we've looked at several scatterplots and at how lines, error bars, and polygons can be added to plots. We've also learned how to create semitransparent colors. These are pretty basic tools which can be applied to a wide range of graphics. In the next chapter we'll continue with visualizing data. We've also looked at how we can create multi-panel figures.

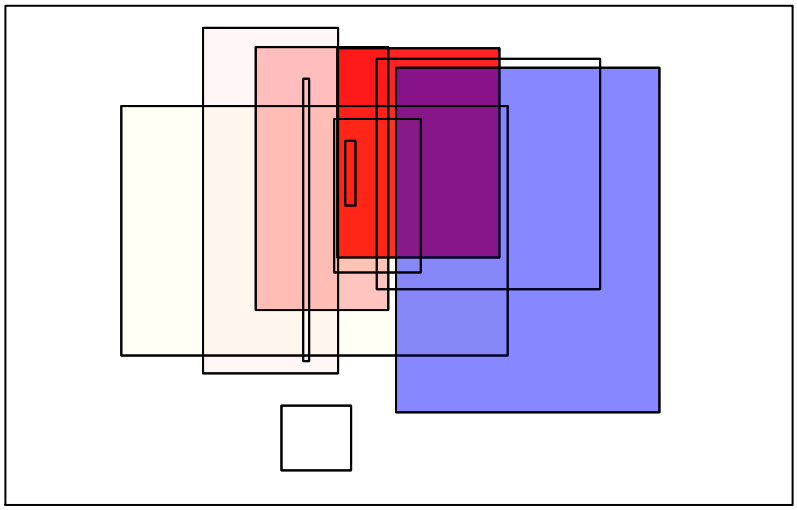
VII. Fun with R graphics

I created this at one point to see what kind of "art" (or perhaps I should say "aRt"?) I could generate in R with random rectangles and random (semitransparent) fill. It mostly uses things we've looked at in this lesson. Here I've defined it as a function because that makes it easier to run it repeatedly until I like the output. We'll look more at writing functions later.

```
random.color.boxes = function(n = 10) {
    cols <- colors()[c(1, 26, 552, 652, 454)]
    cols <- col2rgb(cols)  # to find out the rgb code for a color
    als <- c(rep(0, 5), rep(10, 5), seq(0, 220, 20))
    # rgb(red,green,blue,alpha,max) # to specify color

    par(mar = c(1, 1, 1, 1))
    plot(0:10, 0:10, type = "n", xaxt = "n", yaxt = "n", xlab = "", ylab = "")
    cs <- sample(1:5, n, rep = T)
    as <- sample(als, n, rep = T)
    a <- runif(n) * 10
    b <- runif(n) * 10
    c <- runif(n) * 10
    d <- runif(n) * 10
    rect(a, b, c, d, border = "black", col = rgb(cols[1, cs], cols[2, cs], cols[3,
        cs], as, max = 255))
    rect(a, b, c, d, border = "black")  # replot borders
```

```
}
random.color.boxes()
```



VIII. Exercises

1) In Chapter 12 we fit a simpler model (only the average temperature and the quadratic term, no factor for insulation). Use `polygon()` to plot the confidence interval for this model and then plot the points over it. *Hint* Use `predict()` to generate the confidence interval.

2) The "ufc" data we examined in last chapter's exercises can be loaded from ufc.csv (in the "data" directory of EssentialR). This contains data on forest trees, including `Species`, diameter (in cm, measured 4.5 feet above ground and known as "diameter at breast height" or `Dbh`), and `height` (in decimeters). Make a scatterplot showing average `Height` as a function of average `Dbh` for each species. Include x and y errorbars on the plot showing the standard errors.

Chapter 17: Visualizing Data III

Boxplots and barplots.

I. Introduction

In the chapters we've covered basic scatter plotting, and then looked at how we can use several graphical functions to control spacing of margins around plots, multiple panels in a figure, and the addition of error bars, and polygons on the plot and in the margins. Here we'll build on what we learned last time and apply it to boxplots and barplots.

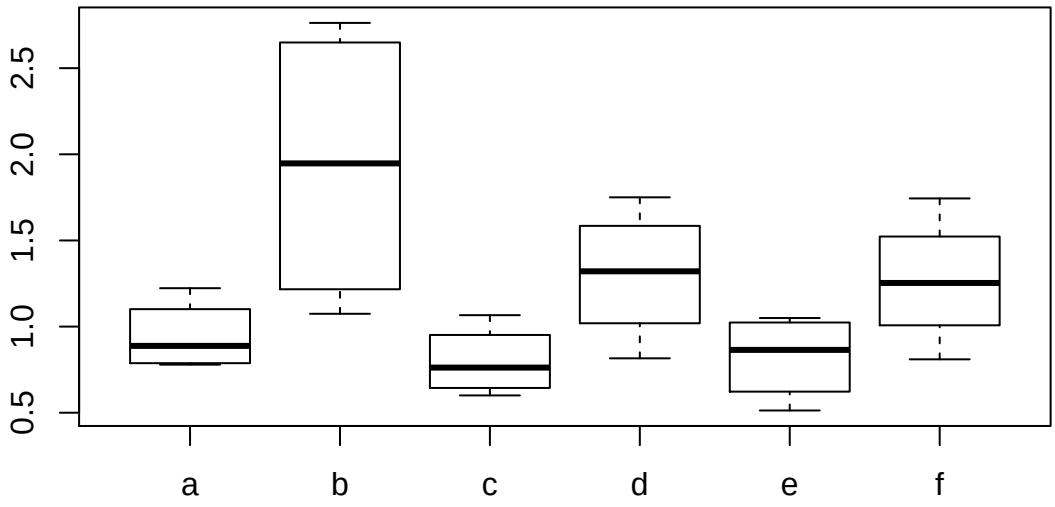
II. Boxplots

Boxplots are very useful tools for visualizing and comparing distributions. We'll use boxplots to explore some data on the growth of beans in nine different treatments (combinations of soil fertility, pot size, and presence/absence of competitors) [51]

```
beans <- read.csv("../Data/BeansData.csv", header = TRUE, comm = "#")  # get data
names(beans)
```

```
# [1] "pot.size" "phos"     "P.lev"    "rep"      "trt"      "rt.len"
# [7] "ShtDM"    "RtDM"
```

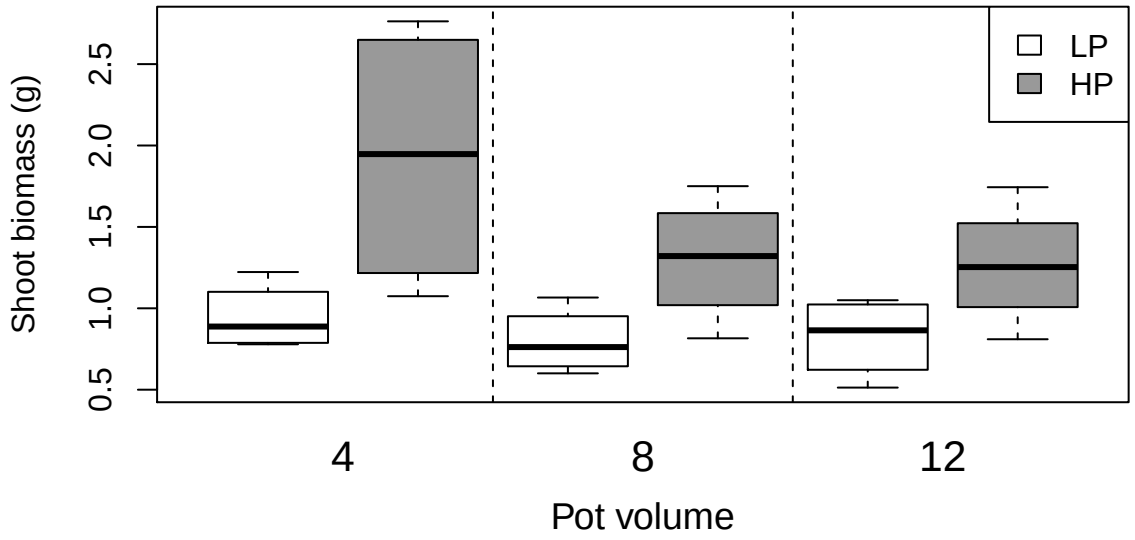
```
par(mar = c(3.5, 4.1, 0.6, 0.6))
with(beans, plot(trt, ShtDM))  # make plot
```



This is a pretty basic plot. Let's fix it up with treatment labels and a legend:

```
par(mar = c(4.1, 4.1, 0.6, 0.6))
with(beans, plot(trt, ShtDM, col = c("white", "grey60"), ylab = "Shoot biomass (g)",
    xlab = "", xaxt = "n"))
abline(v = c(2.5, 4.5), lty = 2)
axis(side = 1, at = c(1.5, 3.5, 5.5), labels = c(4, 8, 12), tick = FALSE, cex.axis = 1.3)
legend("topright", legend = c("LP", "HP"), fill = c("white", "grey60"), bg = "white")
mtext("Pot volume", side = 1, line = 2.5, cex = 1.2)
```

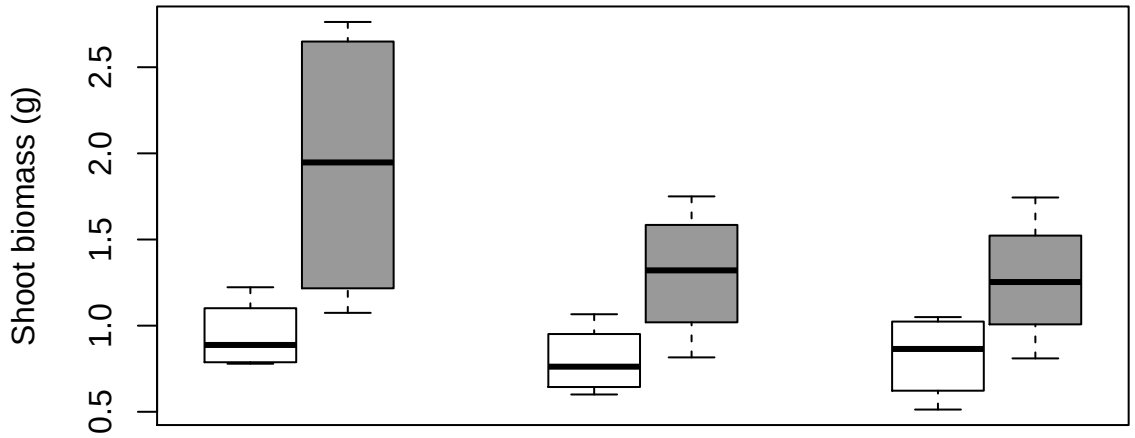
[51] You can find more than you want to know about this data here.



Most of the 'tricks' we used here were introduced in the last chapter. Things to notice here: 1. The vector of colors for filling the boxes is only 3 elements long - it is recycled over the 9 boxes. 2. `abline()` can be used to draw vertical (`v=...`) or horizontal lines - useful for gridlines or separating lines. 3. Using `axis()` with `tick=FALSE` allows us to put labels on the x axis wherever we want.

Here is an alternate (basic) form of the graph where we use the `at` argument in `boxplot()` to group the boxes. The other elements (axis labels, legend) haven't been included here, but can be added, though the spacing of added elements may need to be adjusted.

```
par(mar = c(3.5, 4.1, 0.6, 0.6))
with(beans, plot(trt, ShtDM, col = c("white", "grey60"), ylab = "Shoot biomass (g)",
    xlab = "", xaxt = "n", at = c(1.15, 2, 4.15, 5, 7.15, 8)))
```



Adding means to boxplots

Barplots (see below) are often used instead of boxplots, but barplots generally only show mean and variation. There are a number of very good reasons to prefer boxplots with added means over barplots. Here we'll demonstrate how to construct such a plot.

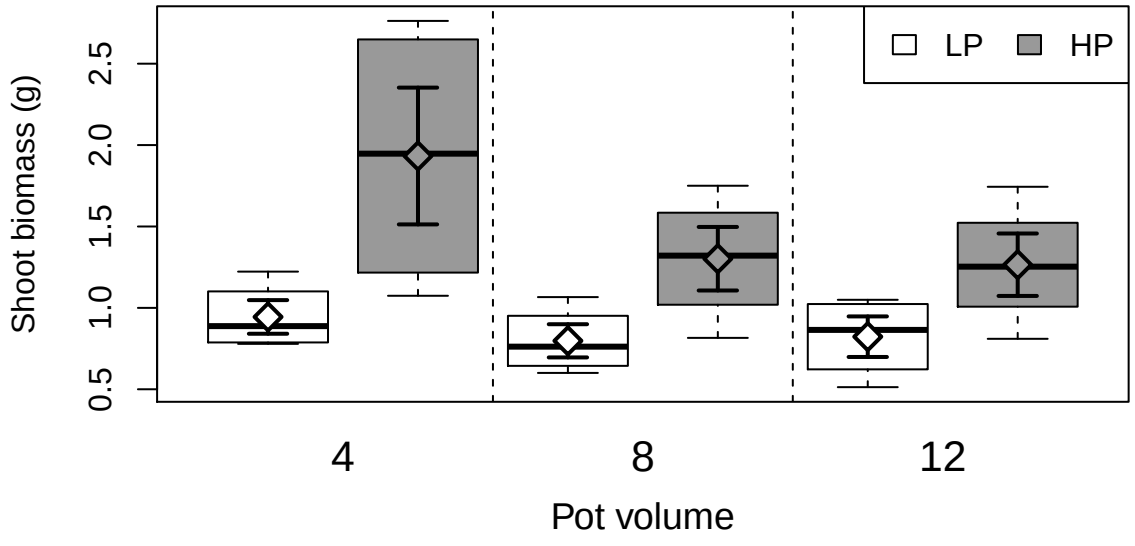
The first step is to extract means and standard errors from the data set. This should ring a bell - we used `apply()` functions and `aggregate()` for this in Chapter 9. Here is an example of defining a function "on the fly" to calculate the SE by treatment. It calculates the number of values by subtracting the number of `NA` values (`sum(is.na(x))`) from the total number of values (`length(x)`). In a perfect world, we don't have to worry about missing data, but real data is rarely that nice.

```
beans2 <- aggregate(beans[, 6:8], by = list(beans$phos, beans$pot.size), mean,
    na.rm = TRUE)  # get means
beans2 <- cbind(beans2, aggregate(beans[, 6:8], by = list(beans$phos, beans$pot.size),
    function(x) (sd(x, na.rm = TRUE)/((length(x) - sum(is.na(x)))^0.5)))[, 3:5])
names(beans2) <- c("phos", "pot.size", "rt.len", "ShtDM", "RtDM", "rt.lense",
    "ShtDMse", "RtDMse")
beans2$type <- letters[1:6]  # create the trt type variable
beans2  # check
```

```
#   phos pot.size   rt.len    ShtDM     RtDM rt.lense   ShtDMse     RtDMse
# 1  210        4 255.2925 0.944375 0.773750 16.18536 0.1033033 0.07518041
# 2  420        4 400.2000 1.932875 1.185400 43.33198 0.4201438 0.25939775
# 3  105        8 178.8750 0.797575 0.599950 25.32042 0.1019539 0.05107649
# 4  210        8 436.2800 1.301950 1.000125 50.10893 0.1952229 0.13446147
# 5   70       12 226.7575 0.823000 0.683700 25.75557 0.1245827 0.11023268
# 6  140       12 310.5025 1.265075 0.958225 17.40478 0.1917926 0.13873179
#   type
# 1    a
# 2    b
# 3    c
# 4    d
# 5    e
# 6    f
```

Now that we have the means and standard errors for the groups we can add them to the boxplot we just made.

```
par(mar = c(4.1, 4.1, 0.6, 0.6))
with(beans, plot(trt, ShtDM, col = c("white", "grey60"), ylab = "Shoot biomass (g)",
    xlab = "", xaxt = "n"))
arrows(1:6, beans2$ShtDM - beans2$ShtDMse, 1:6, beans2$ShtDM + beans2$ShtDMse,
    code = 3, angle = 90, length = 0.1, lwd = 2)
points(1:6, beans2$ShtDM, pch = 23, cex = 1.5, lwd = 2, bg = c("white", "grey60"))
abline(v = c(2.5, 4.5), lty = 2)
axis(side = 1, at = c(1.5, 3.5, 5.5), labels = c(4, 8, 12), tick = FALSE, cex.axis = 1.3)
legend("topright", legend = c("LP", "HP"), fill = c("white", "grey60"), bg = "white",
    ncol = 2)
mtext("Pot volume", side = 1, line = 2.5, cex = 1.2)
```

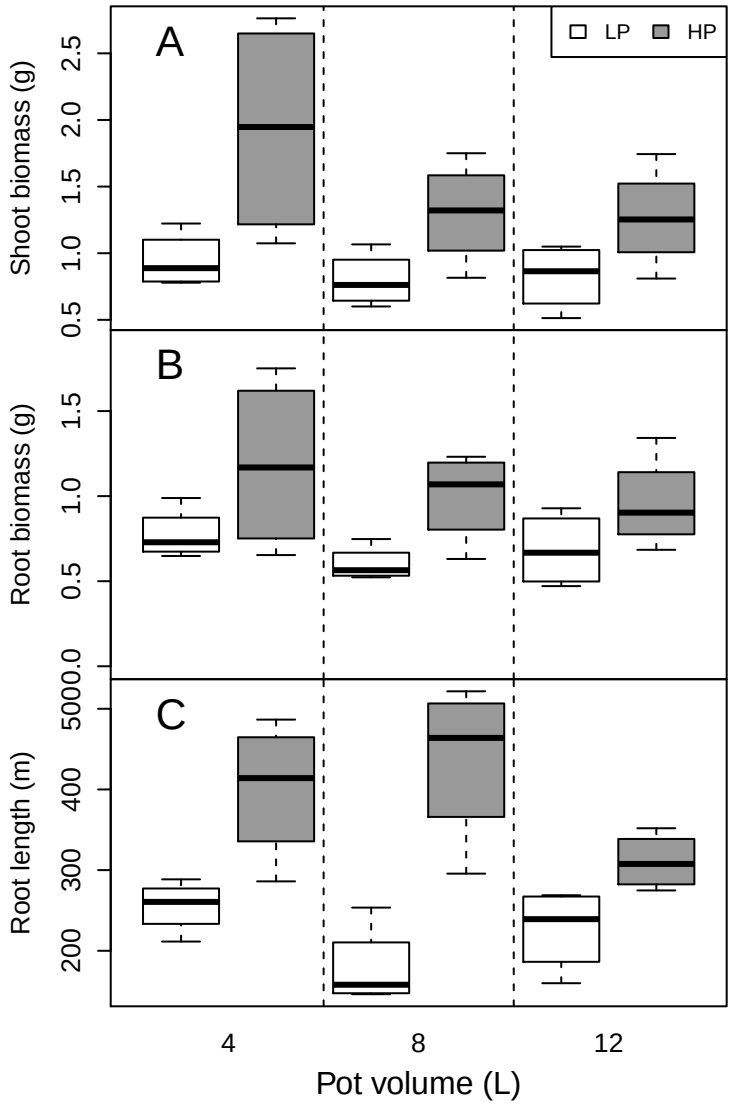


This is a graphic that clearly shows the distribution of the variables. The choice of how to indicate the means and standard errors has some room for variation. Here we've used a solid fill so the median line is obscured where overplotted by the mean - this is one way to ensure that it is clearly visible even if the mean happens to coincide with the median.

A Multi-panel Boxplot

Let's adapt this code to make a multi-panel boxplot. As we discussed in the last session, you may want to force a plotting window of specific size using: `quartz(title= "Density Dependence", height=6,width=4)` or `x11(title="Density Dependence",height=6,width=4)` to create the appropriately sized plotting window.

```
# quartz(width=5,height=7)
layout(matrix(c(1, 2, 3), nrow = 3), heights = c(1, 1, 1.3))  #; layout.show(3)
par(mar = c(0, 5, 1, 1), cex.axis = 1.3, cex.lab = 1.3)
# shoot biomass
with(beans, plot(trt, ShtDM, col = c("white", "grey60"), ylab = "Shoot biomass (g)",
    xlab = "", xaxt = "n", cex = 1.3))
abline(v = c(2.5, 4.5), lty = 2)
legend("topleft", inset = c(-0.05, -0.05), "A", cex = 2, bty = "n")
legend("topright", legend = c("LP", "HP"), fill = c("white", "grey60"), bg = "white",
    ncol = 2)
par(mar = c(0, 5, 0, 1))
# root biomass
with(beans, plot(trt, RtDM, col = c("white", "grey60"), ylab = "Root biomass (g)",
    xlab = "", xaxt = "n", cex = 1.2, ylim = c(0, 1.9)))
abline(v = c(2.5, 4.5), lty = 2)
legend("topleft", inset = c(-0.05, -0.05), "B", cex = 2, bty = "n")
par(mar = c(5, 5, 0, 1))
# root length
with(beans, plot(trt, rt.len, col = c("white", "grey60"), ylab = "Root length (m)",
    xlab = "", xaxt = "n", cex = 1.2))
abline(v = c(2.5, 4.5), lty = 2)
legend("topleft", inset = c(-0.05, -0.05), "C", cex = 2, bty = "n")
axis(side = 1, at = c(1.5, 3.5, 5.5), labels = c(4, 8, 12), tick = FALSE, cex.axis = 1.3)
mtext("Pot volume (L)", side = 1, line = 2.65)
```



Notice the use of `legend()` instead of `text()` to place the panel labels (A-C). Since a location like `"topleft"` can be specified and the `inset` specified (in fraction of the plot) this is an easier way to get these labels in the same location on all panels.

III. Barplots

Now we're ready to make barplots (though as noted above, you should probably ask yourself if a boxplot would be a better choice). We've used `barplot()` in a few examples previously, but we'll dig in more here. NOTE: `barplot()` does not accept data in a formula (y~x) like `plot()` or `boxplot()`. It also requires that the argument `height`, which specifies the height of the bars, be a vector or a matrix. We'll use `arrows()` to make error bars as we did previously. We already calculated the means and standard errors for this data for overplotting means on boxplots.

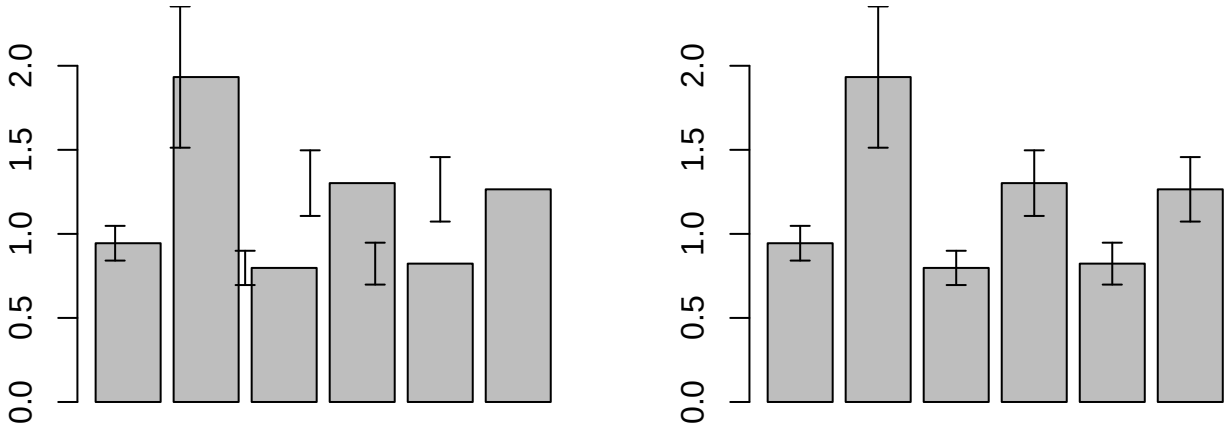
```
with(beans2, barplot(ShtDM, ylim = c(0, max(ShtDM + ShtDMse))))
with(beans2, arrows(0.5:5.5, ShtDM + ShtDMse, 0.5:5.5, ShtDM - ShtDMse, length = 0.05,
    angle = 90, code = 3))
```

The basic barplot (below left) is OK, but the error bars ended up in strange locations. In the previous example with `boxplot()` the boxes were centered on the integer values on the x-axis - we used this in placing

our separating lines and x-axis labels. Here something else is going on.

A closer reading of `?barplot` yields this useful information: in addition to making a barplot, `barplot()` also **invisibly returns the midpoints of the bars**. All we need to do is assign this output from barplot and we can use it (below right).

```
mp <- with(beans2, barplot(ShtDM, width = 0.8, ylim = c(0, max(ShtDM + ShtDMse))))
with(beans2, arrows(mp, ShtDM + ShtDMse, mp, ShtDM - ShtDMse, length = 0.05,
    angle = 90, code = 3))
```

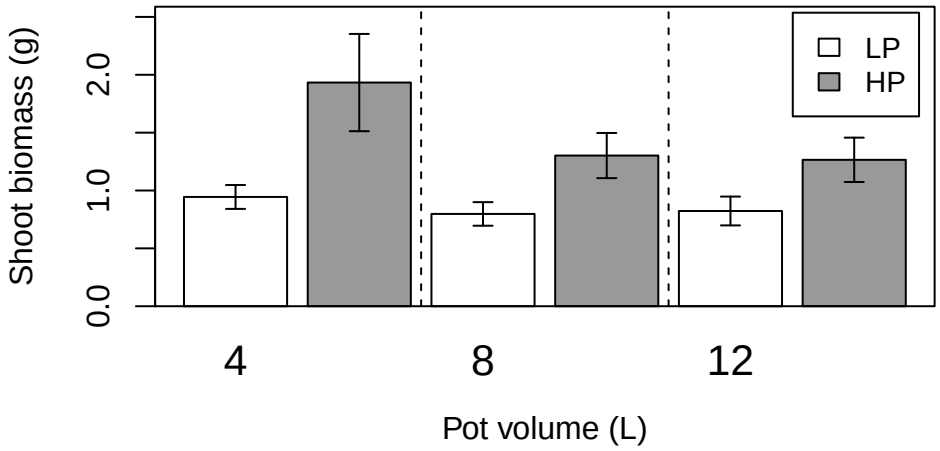


Can bar position be specified? Of course! A careful reading of `?barplot` is instructive: the argument that controls bar width is `width` and it's default value is 1. The spacing is controlled by `space`, which defaults to 0.2 (where bars are not grouped); `space` is specified in fractions of `width`. Both `space` and `width` can be specified as vectors for the ability to control bar placement.

Creating a Polished Barplot

Putting all of this together we can make a high-quality barplot.

```
par(mar = c(4.1, 4.1, 0.6, 0.6))
mp <- with(beans2, barplot(ShtDM, col = c("white", "grey60"), width = 0.8, ylim = c(0,
    max(ShtDM + ShtDMse) * 1.1), ylab = "Shoot biomass (g)", xlab = "", xaxt = "n"))
# make the plot
box(which = "plot")  # put a box around it
with(beans2, arrows(mp, ShtDM + ShtDMse, mp, ShtDM - ShtDMse, length = 0.05,
    angle = 90, code = 3))  # add error bars
abline(v = c(mean(mp[2:3]), mean(mp[4:5])), lty = 2)  #add dashed separator lines
legend("topright", inset = 0.02, legend = c("LP", "HP"), fill = c("white", "grey60"),
    bg = "white", cex = 0.9)  # legend
axis(side = 1, at = mp[c(1.5, 3.5, 5.5)], labels = c(4, 8, 12), tick = FALSE,
    cex.axis = 1.3)
mtext("Pot volume (L)", side = 1, line = 2.65)
```



Notice that we didn't have to hard-code legend or label locations: `legend()` can take `"topright"` as a location and we can specify `inset` also. The x-axis labels and vertical separators are based on the vector of bar midpoints also. This is generally the best approach: 1) if you change the size of the plotting window or the character size in the legend, things will still end up in the right location 2) since it makes your code more general, it is easier to adapt for the next plot you have to make.

Creating a Multi-panel Barplot

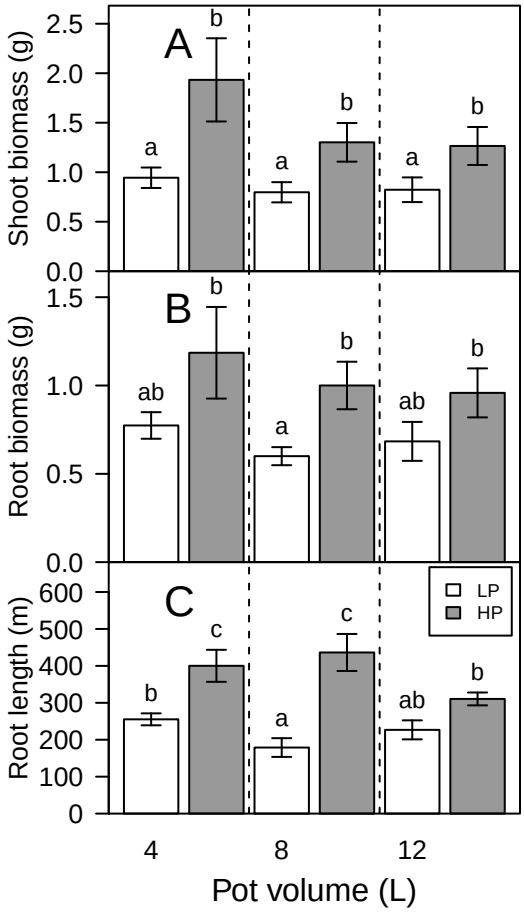
Now we'll put it all together for multiple panels, and include significance letters such as might be given by a Tukey test (not shown here). (As above, height and width want to be forced to 6" and 4").

```
# quartz(width=5,height=7)
layout(matrix(c(1, 2, 3), nrow = 3), heights = c(1, 1, 1.3))  # make layout
par(mar = c(0, 5.5, 1, 1), cex.axis = 1.3, cex.lab = 1.3, las = 1)  # par settings
mp <- with(beans2, barplot(ShtDM, col = c("white", "grey60"), ylim = c(0, max(ShtDM +
    ShtDMse) * 1.14), ylab = "Shoot biomass (g)", xlab = "", xaxt = "n"))  #plot 1
box(which = "plot")
with(beans2, arrows(mp, ShtDM + ShtDMse, mp, ShtDM - ShtDMse, length = 0.05,
    angle = 90, code = 3))  # err bars
divs <- c(mean(mp[2:3]), mean(mp[4:5]))  # calculate divider locations
abline(v = divs, lty = 2)
text(mp, (beans2$ShtDM + beans2$ShtDMse), c("a", "b", "a", "b", "a", "b"), pos = 3,
    cex = 1.15)  # add letters
legend("topleft", inset = c(-0.05, -0.05), "A", cex = 2, bty = "n")

par(mar = c(0, 5.5, 0, 1))
with(beans2, barplot(RtDM, col = c("white", "grey60"), ylim = c(0, max(RtDM +
    RtDMse) * 1.14), ylab = "Root biomass (g)", xlab = "", xaxt = "n"))  # plot 2
box(which = "plot")
with(beans2, arrows(mp, RtDM + RtDMse, mp, RtDM - RtDMse, length = 0.05, angle = 90,
    code = 3))  # err bars
abline(v = divs, lty = 2)
text(mp, (beans2$RtDM + beans2$RtDMse), c("ab", "b", "a", "b", "ab", "b"), pos = 3,
    cex = 1.15)  # add letters
legend("topleft", inset = c(-0.05, -0.05), "B", cex = 2, bty = "n")

par(mar = c(5, 5.5, 0, 1))
with(beans2, barplot(rt.len, col = c("white", "grey60"), ylim = c(0, max(rt.len +
    rt.lense) * 1.4), ylab = "Root length (m)", xlab = "", xaxt = "n"))  # plot 3
box(which = "plot")
```

```
with(beans2, arrows(mp, rt.len + rt.lense, mp, rt.len - rt.lense, length = 0.05,
    angle = 90, code = 3))  # err bars
abline(v = divs, lty = 2)
text(mp, (beans2$rt.len + beans2$rt.lense), c("b", "c", "a", "c", "ab", "b"),
    pos = 3, cex = 1.15)  # add letters
legend("topleft", inset = c(-0.05, -0.05), "C", cex = 2, bty = "n")
axis(side = 1, at = mp[c(1.5, 3.5, 5.5)], labels = c(4, 8, 12), tick = FALSE,
    cex.axis = 1.3)
mtext("Pot volume (L)", side = 1, line = 2.65)
legend("topright", inset = 0.02, legend = c("LP", "HP"), fill = c("white", "grey60"),
    bg = "white", cex = 0.9)
```



Notice that locations of the added elements mostly are not hard-coded. They `y` location for the panel labels is calculated the same way as the upper `ylim` value.

V. Summary

The last three chapters have introduced most of the tools required for presenting data in graphical form. One of the great challenges is figuring out what the clearest and most informative way to present a given set of data is. As noted above, barplots are commonly used when they probably should not be used. When thinking about presenting data, it may be useful to review the work of Cleveland and McGill, as this helps us think about how to clearly present data.

V. Exercises

1) The "ufc" data we examined in last chapter's exercises can be loaded from ufc.csv (in the "data" directory of EssentialR). This contains data on forest trees, including `Species`, diameter (in cm, measured 4.5 feet above ground and known as "diameter at breast height" or `Dbh`), and `height` (in decimeters). Make a single barplot showing both variables (`Dbh` and `Height`) for each `Species` and include errorbars (either SE of the mean or 95% CI) and a legend. *Hint*: In Ch 15 we saw how to add a secondary y-axis - with a barplot you need to specify the x-axis limits and change the `width` and `space` arguments to make the bars narrower and to get the bars to plot side-by-side.

2) Using the ufc data create a single boxplot that also shows the mean and standard error for both variables for the 5 most common (greatest frequency) species. *Hint*: create a new dataframe with only the 5 most common species - and be sure to check the levels for `Species` in the new dataframe - you may need to use `droplevels()` to get rid of unused levels.

Chapter 18: Mixed Effects Models

A brief overview of fitting mixed models

I. Introduction

I will begin by saying that this should be considered a *minimal* introduction to mixed effects models, and the goal here is point you toward a few functions and packages likely to be useful. There are many, many possible applications of mixed models, and I am not an expert. Having said that, I will share what I have learned.

The regression assumptions we examined so far (Chapters 6 and 11) deal with *normality* of the residuals and *constant variance* in the residuals. There is another assumption - namely that the residuals are *independant*[52].

In many cases this assumption causes difficulties in analysis. For example in a drug metabolism trial a group of subjects may be given a fixed dose of medication and the persistence of the medication, or its metabolites (breakdown products) in the blood measured over some length of time. We can imagine that concentration of a metabolite in blood might be related to time, but there may be individual differences between subjects. Furthermore, the levels from one subject are likely to be correlated with each other, because they are generated by the same organism.

Another possible example might be measuring the changes in student scores on a math test in response to one of three different curricula, with each curriculum presented by one of four different teachers. We might expect that each of the 12 teachers (four per curriculum x 3 curricula) would present the curriculum in a slightly different way, so there might be a lack of independence between residuals within a classroom. We can't really include teacher as a fixed effect here, becuase each teacher only teaches one curriculum. But we can treat teacher as a random effect.

We might be able to control for the within subject variation by adding a term for subject in the model, but this has a major drawback: we have to use a lot of degrees of freedom estimating coefficients for each subject, and this reduces our error degrees of freedom, which reduces our ability to detect the thing we are really interested in (in this case the persistence of the medication).

In linear models we are using a model like $y = B_0 + B_1 x + \varepsilon$ to model the effect of x on y. We refer to the parameters we estimate here as *fixed effects*. In the mixed effects model we partition the error portion of the model (ε) into *random effects* and error. The random effect are used to account for the variability between subjects. This allows our model to account for this source of variation without using degrees of freedom to estimate the coefficients (parameters) for each.

This is analogous to the split-plot design we discussed in Chapter 13. Instead of correlation (dependency) between measurements coming from a single subject, here we expect some correlation of residuals from subplots within a larger plot (main plot or whole plot) – correlation driven by spatial proximity.

How do you know if an effect is fixed or random? Part of the answer lies in what you want to estimate. In the drug trial example above our goal is to understand the way the drug works for a whole population, from which we have drawn a (hopefully!) representative sample. So we would consider the subjects as random, but the drug treatments or time after treatment to be fixed. Another way to ask the question might be "If someone else repeated my experiment what would be the same and what would be different?" In the example above the drug and time after treatment variables could be exactly repeated, but the subject would not. This suggests that subjects is a random effect.[53]

[52] The assumption of constant variance and independence of the residuals are not really separate assumptions - a pattern in the residuals indicates some dependence in them.

[53] Though it is worth bearing in mind that unless you have a minimum of 4 or 5 levels for a factor you are not likely to have sufficient ability to estimate random effects, so effects that are theoretically random may in some cases be analyzed as fixed.

II. A basic example

Let's begin with an example. We'll load the package `nlme` (Non Linear Mixed Effects) - if you haven't done so you will need to install it (`install.packages("nlme")`). We'll then load the dataset `Machines` and inspect it.

```
require(nlme)
data(Machines)  # 6 randomly selected workers tested on 3 types of machine
summary(Machines)
```

```
#  Worker Machine     score
#  6:9    A:18    Min.   :43.00
#  2:9    B:18    1st Qu.:52.42
#  4:9    C:18    Median :61.60
#  1:9            Mean   :59.65
#  3:9            3rd Qu.:65.30
#  5:9            Max.   :72.10
```

This dataset comes from a trial of three brands of machines used in an industrial process. Six workers in the factory were selected at random and each operated each machine three times. The response variable is "score" which is a productivity score - for more information use `? Machines` (Note: this will not work if `nlme` has not been loaded). If our primary goal is to ask "Which machine maximizes productivity?" we might start this way:

```
fm1 <- lm(score ~ Machine, data = Machines)
anova(fm1)
```

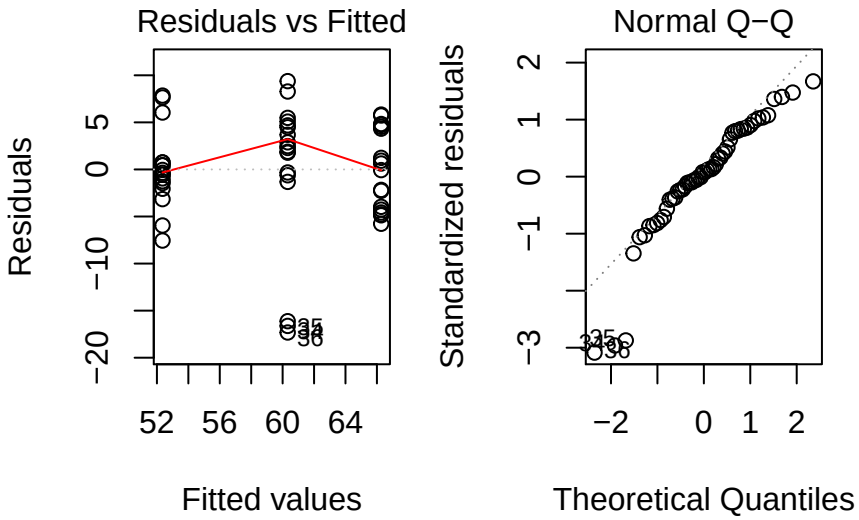
```
# Analysis of Variance Table
#
# Response: score
#           Df Sum Sq Mean Sq F value    Pr(>F)
# Machine    2 1755.3  877.63  26.302 1.415e-08 ***
# Residuals 51 1701.7   33.37
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
summary(fm1)
```

```
#
# Call:
# lm(formula = score ~ Machine, data = Machines)
#
# Residuals:
#      Min       1Q   Median       3Q      Max
# -17.3222  -2.1431   0.4444   4.4403   9.3778
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)   52.356      1.362  38.454  < 2e-16 ***
# MachineB       7.967      1.925   4.138 0.000131 ***
# MachineC      13.917      1.925   7.228 2.38e-09 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 5.776 on 51 degrees of freedom
# Multiple R-squared:  0.5077,  Adjusted R-squared:  0.4884
# F-statistic:  26.3 on 2 and 51 DF,  p-value: 1.415e-08
```

We can see that we have a significant difference associated with brand of machine (p much less than 0.0001), and it appears that all three machines are likely different from each other in their productivity. However an examination of the residuals reveals trouble – evidence of both departure from normality and unequal variance.

```
plot(fm1, which = 1:2)  # but residuals look a bit strange
```



A bit of reflection might convince us that different workers might also have different levels of productivity (`score`). A quick `oneway.test()` confirms this.

```
oneway.test(score ~ Worker, data = Machines)
```

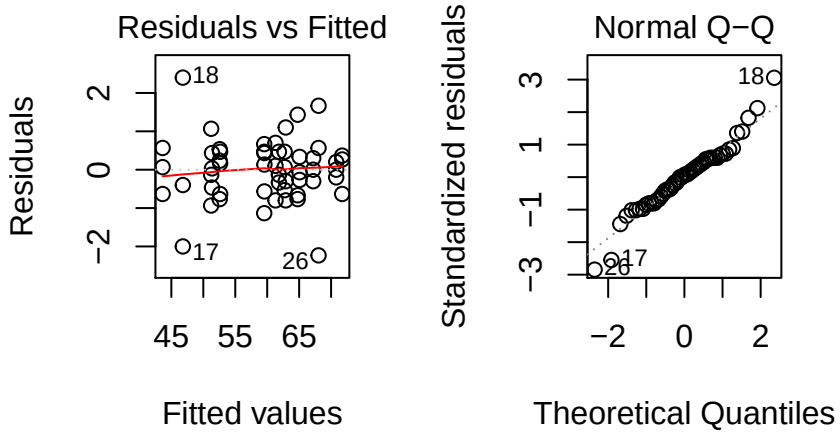
```
#
#   One-way analysis of means (not assuming equal variances)
#
# data:  score and Worker
# F = 4.866, num df = 5.000, denom df = 22.149, p-value = 0.003747
```

The fourth panel of the above figure also suggests that machines may affect productivity differently for different workers, which suggests an interaction.

```
fm2 <- lm(score ~ Machine * Worker, data = Machines)
anova(fm2)
```

```
# Analysis of Variance Table
#
# Response: score
#                Df  Sum Sq Mean Sq F value    Pr(>F)
# Machine         2 1755.26  877.63  949.17 < 2.2e-16 ***
# Worker          5 1241.89  248.38  268.63 < 2.2e-16 ***
# Machine:Worker 10  426.53   42.65   46.13 < 2.2e-16 ***
# Residuals      36   33.29    0.92
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
plot(fm2, which = 1:2)
```




```
par(op)
```

The ANOVA above shows that when we account for the dependence of machine productivity on worker both machine and worker differences are very significant. However the model (coefficients given by `summary(fm2)`) is difficult to interpret - the interaction means that different workers get different results from the machines. What we really want to know is how the machines would work for the *average* worker. To work toward this we'll use the function `lme()` from `nlme`.

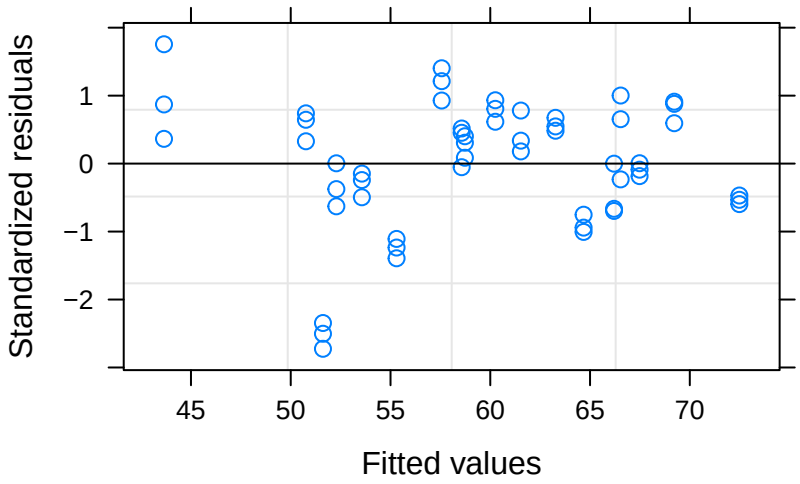
```
fm3 <- lme(score ~ Machine, random = ~1 | Worker, data = Machines)
```

Here we've created a mixed effects model (an "lme object") with a random intercept for each worker. We can query this object with various methods, which will be rather familiar. A quick review of `summary(fm3)` (not shown here) shows estimates for our Fixed effects - the three levels of `Machine`, and an estimate of the variation associated with workers.

```
anova(fm3)  #  larger F-value than with lm()
```

```
#             numDF denDF  F-value p-value
# (Intercept)     1    46 773.5703  <.0001
# Machine         2    46  87.7982  <.0001
```

```
plot(fm3)  # residuals still odd
```



This is the main benefit of mixed effects models - here we are using the variation among 6 workers to estimate

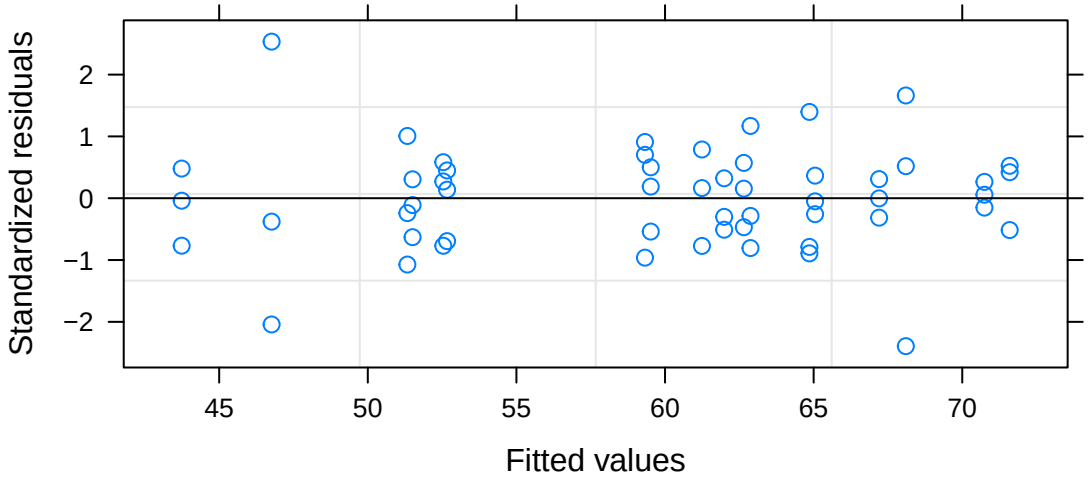
the general variability associated with workers, as opposed to trying to estimate the *fixed* effects for 6 specific workers. Note that the standard deviation of the intercept for `Worker` is still similar in size to inter-machine differences.

Note that the output from `anova()` is a bit different - we have an F test for the intercept as well as for the effect `Machine`. Note that the F-value for `Machine` is much greater here than in the first model that we fit with `lm()` - accounting for worker differences shrinks the residual variance and makes our F test more sensitive.

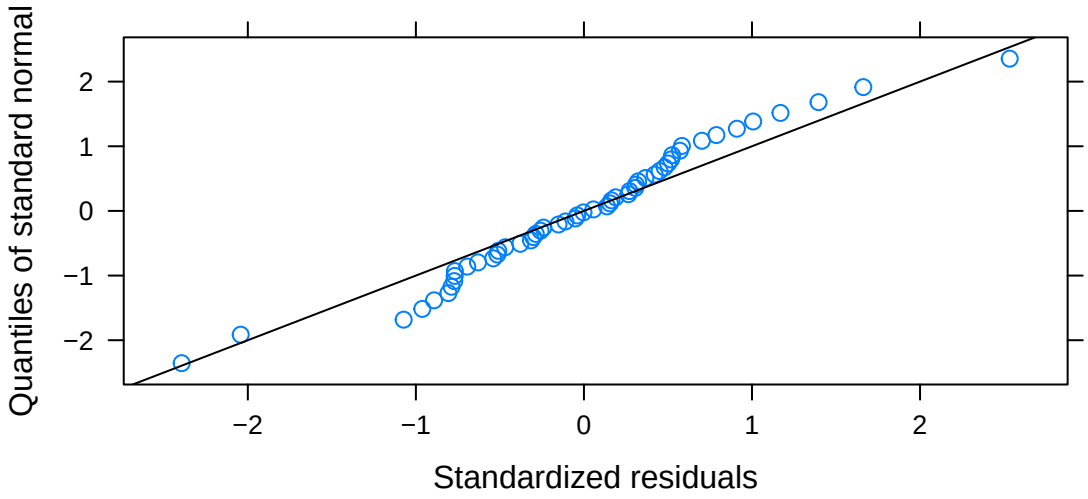
However the residuals suggest that we have some trouble - we seem to have more negative residuals for intermediate fitted values. This may not be a surprise, we have not accounted for differences between workers in how machines affect productivity.

We can fit a model with both random intercept and random effect of machine for each worker. Let's check the residuals first.

```
fm4 <- lme(score ~ Machine, random = ~1 + Machine | Worker, data = Machines)
plot(fm4)
```




```
qqnorm(fm4, abline = c(0, 1))
```




```
par(op)
```

The residuals are behaving a bit better here, and look rather similar to those of our second model. The qqplot shows they are sufficiently near normal.

```
anova(fm4)  # F value lower, but still signif.
```

```
#             numDF denDF   F-value p-value
# (Intercept)     1    46 2351.8063  <.0001
# Machine         2    46   41.0038  <.0001
```

```
summary(fm4)
```

```
# Linear mixed-effects model fit by REML
#  Data: Machines
#        AIC      BIC    logLik
#   228.3112 247.6295 -104.1556
#
# Random effects:
#  Formula: ~1 + Machine | Worker
#  Structure: General positive-definite, Log-Cholesky parametrization
#             StdDev    Corr
# (Intercept) 4.0792806 (Intr) MachnB
# MachineB    5.8776433  0.484
# MachineC    3.6898543 -0.365  0.297
# Residual    0.9615766
#
# Fixed effects: score ~ Machine
#                Value Std.Error DF   t-value p-value
# (Intercept) 52.35556  1.680711 46 31.150834  0.0000
# MachineB     7.96667  2.420851 46  3.290854  0.0019
# MachineC    13.91667  1.540100 46  9.036211  0.0000
#  Correlation:
#          (Intr) MachnB
# MachineB  0.463
# MachineC -0.374  0.301
#
# Standardized Within-Group Residuals:
#         Min          Q1         Med          Q3         Max
# -2.39354008 -0.51377574  0.02690829  0.47245471  2.53338699
#
# Number of Observations: 54
# Number of Groups: 6
```

The ANOVA shows that our F-value is still larger than in our first model (with `lm()`), but not as large as in our second model (`lm()` with interaction). However we now have estimates of differences between our machines that should better reflect our expectation for the *population* of workers, rather than for only *our sample* of workers. This is one of the main advantages of mixed effects models.

This is an example of *repeated measures* - each worker was measured 9 times, and so we'd expect some correlation between measurements from an individual worker.

III. Split-plots

In Chapter 13 we considered one example of a split-plot analysis. We'll revisit this analysis here, beginning with the analysis we introduced in Chapter 13. Review the last section of Chapter 13 if you do not recall the diagnostic features of a split plot. You can see that there is similarity between the correlation of residuals in the repeated measures analysis above (correlation of residuals within subject) and what we might find in a split-plot (correlation of residuals within main plots).

```
Rye <- read.csv("../Data/RyeSplitPlot.csv", comm = "#")
summary(Rye)
```

```
#  Plant   Tdate  WdCon    Rep         RyeDM            Pdate
#  P1:20   T1:8   SC:40   I  :10   Min.   : 104.0   Min.   :257.0
#  P2:20   T2:8           II :10   1st Qu.: 454.8   1st Qu.:257.0
#          T3:8           III:10   Median : 636.0   Median :272.5
#          T4:8           IV :10   Mean   : 630.0   Mean   :272.5
#          T5:8                    3rd Qu.: 816.0   3rd Qu.:288.0
#                                  Max.   :1256.0   Max.   :288.0
#     Tdate.1
#  Min.   :480.0
#  1st Qu.:492.0
#  Median :500.0
#  Mean   :500.8
#  3rd Qu.:513.0
#  Max.   :519.0
```

```
summary(aov(RyeDM ~ Rep + Plant + Tdate + Error(Rep/Plant), data = Rye))
```

```
#
# Error: Rep
#     Df Sum Sq Mean Sq
# Rep  3  72833   24278
#
# Error: Rep:Plant
#           Df Sum Sq Mean Sq F value Pr(>F)
# Plant      1 216531  216531   8.152 0.0648 .
# Residuals  3  79686   26562
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Error: Within
#           Df  Sum Sq Mean Sq F value   Pr(>F)
# Tdate      4 2043518  510879   46.34 5.92e-12 ***
# Residuals 28  308693   11025
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The function `aov()` used here does not create an object that includes residuals, so analysis of the residuals is not straightforward. Let's compare this with the same analysis done using `lme()`

```
fm1 <- lme(RyeDM ~ Plant + Tdate, random = ~1 | Rep/Plant, data = Rye)
anova(fm1)
```

```
#             numDF denDF  F-value p-value
# (Intercept)     1    28 624.5005  <.0001
# Plant           1     3   8.5182  0.0616
# Tdate           4    28  46.3393  <.0001
```

The syntax of the argument `random` is confusing. We've said here that the *intercept* (`1`) is random, for each level of `Rep` and `Plant` "within" `Rep`. This is how R knows the error structure. Just as the smallest plots (`Tdate` didn't need to be included in the `Error()` statement in the formula in `aov()`, it needn't be included here. Note that `Plant` is also included as a fixed effect.

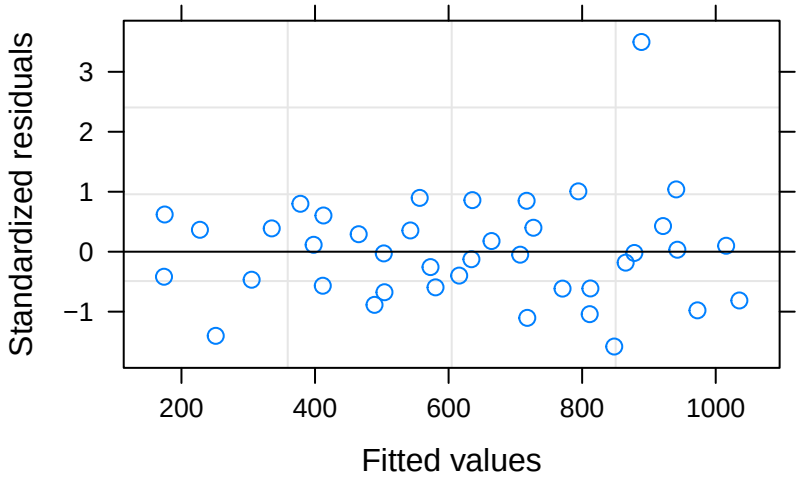
Note that the F-values and p-values are very similar. As noted in Chapter 13, all F-tests are based on *Type I SS*. If we want something like a *Type II SS* we can specify `anova(fm1, type="marginal")`. In this case it

makes no difference because the data is balanced, but of course one can't count on that!

```
summary(fm1)
```

```
# Linear mixed-effects model fit by REML
#  Data: Rye
#        AIC      BIC    logLik
#   448.6685 462.4057 -215.3342
#
# Random effects:
#  Formula: ~1 | Rep
#         (Intercept)
# StdDev:  0.03048796
#
#  Formula: ~1 | Plant %in% Rep
#         (Intercept) Residual
# StdDev:    53.65659 104.9988
#
# Fixed effects: RyeDM ~ Plant + Tdate
#                Value Std.Error DF   t-value p-value
# (Intercept)  354.075  48.71826 28  7.267809  0.0000
# PlantP2     -147.150  50.41814  3 -2.918592  0.0616
# TdateT2      237.750  52.49939 28  4.528624  0.0001
# TdateT3      329.125  52.49939 28  6.269121  0.0000
# TdateT4      543.000  52.49939 28 10.342978  0.0000
# TdateT5      637.500  52.49939 28 12.142999  0.0000
#  Correlation:
#         (Intr) PlntP2 TdatT2 TdatT3 TdatT4
# PlantP2 -0.517
# TdateT2 -0.539  0.000
# TdateT3 -0.539  0.000  0.500
# TdateT4 -0.539  0.000  0.500  0.500
# TdateT5 -0.539  0.000  0.500  0.500  0.500
#
# Standardized Within-Group Residuals:
#         Min          Q1         Med          Q3         Max
# -1.58156310 -0.59932896 -0.02460729  0.40645051  3.49696936
#
# Number of Observations: 40
# Number of Groups:
#            Rep Plant %in% Rep
#              4              8
```

```
plot(fm1)
```



The output from `summary()` gives us estimates for the variation associated with the random effects, and estimates for the fixed effects. In this case the residuals look quite good, although there is one value that seems to be an outlier – I leave it to the reader to find it.

Another useful feature of `anova()` is that it can be used to compare models. For example consider that our model above could include an interaction between `Plant` and `Tdate`.

```
fm2 <- lme(RyeDM ~ Plant * Tdate, random = ~1 | Rep/Plant, data = Rye)
anova(fm2)
```

```
#             numDF denDF  F-value p-value
# (Intercept)     1    24 624.5005  <.0001
# Plant           1     3   8.5182  0.0616
# Tdate           4    24  42.7303  <.0001
# Plant:Tdate     4    24   0.4548  0.7679
```

We could directly compare the two models with a call to `anova()`, but before doing so we need to refit them both using *ML* rather than *REML* (Maximum likelihood vs restricted maximum likelihood).[54]

```
fm1.ML <- lme(RyeDM ~ Plant + Tdate, random = ~1 | Rep/Plant, data = Rye, method = "ML")
fm2.ML <- lme(RyeDM ~ Plant * Tdate, random = ~1 | Rep/Plant, data = Rye, method = "ML")
anova(fm1.ML, fm2.ML)
```

```
#        Model df      AIC      BIC    logLik   Test  L.Ratio p-value
# fm1.ML     1  9 503.9396 519.1396 -242.9698
# fm2.ML     2 13 509.6014 531.5569 -241.8007 1 vs 2 2.338195  0.6738
```

Here the comparison shows that adding the interaction effect increases both AIC and BIC and gives a very non-significant p-value, strongly suggesting the simpler model is best. (Note that this is different than the result we get if we call `anova()` on the models fitted with REML).

A final (but important) note about `lme()` – the default behavior when `NA` values are encountered is `"na.fail"`, meaning the call to `lme()` will return an error. This can be over-ridden using the argument `na.action="na.omit"`.

IV. Summary

As noted in the introduction, mixed effects models is a huge topic, and there are many different types of models that can be fit. A good resource is Mixed Effects Models and Extensions in Ecology with R by Zuur

[54] Mixed effects models can't be solved analytically like simple linear models - they are solved numerically.

et al. There are also many other good resources available on the web, though many seem to be somewhat discipline-specific, so a bit of digging will probably be required.

The newer package `lme4` provides a similar function `lmer()`. At this point `lmer()` involves some extra steps as p-values are not directly calculated for the fixed effects[55] but can be estimated via a Markov Chain Monte Carlo approach, and there is a function provided for this.

V. Exercises

1) Refit the split-plot model for the Rye data from above without the outlier. Do the residuals suggest that the model is valid? How did removing the outlier affect the estimation of the fixed and random effects?

2) Another example of a nested model that may be similar to a repeated measures or a split plot can be seen in the dataset `RatPupWeight` that is included in `nlme`. Here are weights for 322 rat pups exposed to one of three experimental treatments. We'd expect that weight might be influenced by sex, size of litter and the experimental treatments, but there is likely to be an effect of the mother (`litter`) also that could mask these others. Fit a mixed effects model of weight as a function of treatment, litter size, and sex with litter as a random effect. How well does this appear to fit the data based on the residuals? Are there significant interactions between the fixed effects?

---

[55] The authors provide some good reasons for this - basically that "decomposition of the error term into random effects and error changes the error degrees of freedom and there is no consensus among statistical theoreticians on what the correct error df to use for the F-tests is."

Chapter 19: Fitting Other Models.

Non-linear least squared models, logistic regession

I. Introduction

The model fitting we have done so far has largely been confined to linear models. But there is a wide array of models that we can fit. Here again, as in the last chapter, we will not go deep into these topics - to do so would involve an entire semester. Instead we'll consider two fairly common cases of fitting non-linear models - Logistic regression and fitting arbitrary equations.

II. Logistic regression

Like the t-test, logistic regression is used when there is one binary response variable (e.g. a factor with two levels [56]) and one continuous predictor variable. The difference is that a logistic regression model allows us to predict the probability of the state of the binary response based on the continuous predictor, which the t-test does not allow.

This is sometimes called *logit* regression because the *logit* transformation on the data turns this into a linear regression. The logit is the natural log of the odds, where the odds are $p/(1-p)$ and p is the probability of some event occurring.

The logistic curve should be familiar to anyone who has studied much about population growth, though the formulation used in that context is generally a bit different:

$$dN/dt = rN(1 - N/K)$$

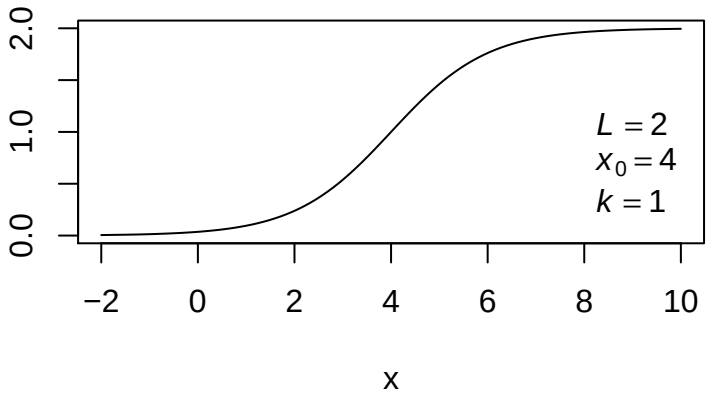
where N is the population size and$K$ is the maximum value N can take - the *Carrying capacity*.

A more general formulation of the logistic curve is:

$$f(x) = L/(1 + exp(-k * (x - x_0)))$$

where L is the maximum possible value of y, x_0 is the midpoint of the curve, and k is the steepness of the curve.

```
curve(2/(1 + exp(-1 * (x - 4))), from = -2, to = 10, ylab = "")
legend("bottomright", inset = 0.02, c(expression(italic(L) == 2), expression(italic(x)[0] ==
    4), expression(italic(k) == 1)), bty = "n")
```



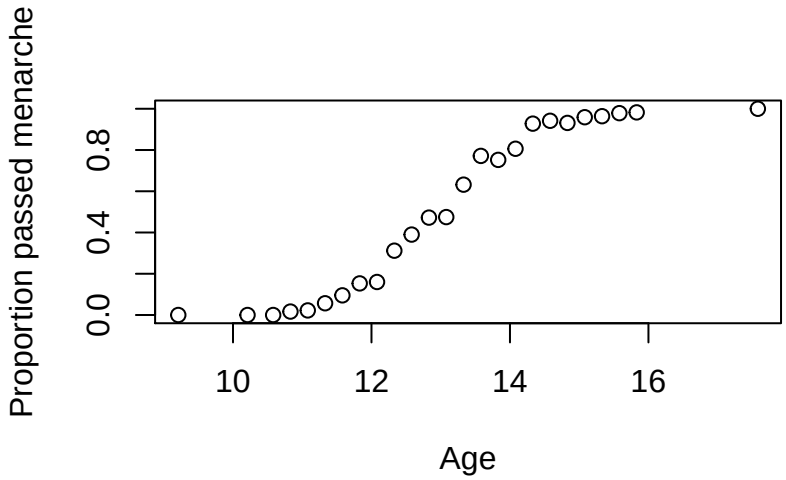
[56] A multiple level factor can be used as the response in multinomial logistic regression - RSeek can point you to more information.

The assumptions here are that the observations are independent and that the relationship between the logit (natural log of the odds ratio) and the predictor variable is linear. Note that the second assumption here is just expressing what we noted above where we introduce the term *logit*.

In R we'll use the function `glm()` to fit logistic regressions by specifying `family="binomial"` in the call to `glm()`.

We'll begin with an example. The package `MASS` includes the data set `menarche`, which describes the age at menarche of 3918 girls in Poland in 1965 [57]. The data includes mean age for a group of girls of the same age (`Age`), the number in each group (`Total`) and the number in each group who had reached Menarche (`Menarche`).

```
library(MASS)
data(menarche)
plot(Menarche/Total ~ Age, data = menarche, ylab = "Proportion passed menarche")
```



This is a perfect data set to demonstrate logistic regression as the plot shows - for an individual the response can be only *no* (0) or *yes* (1), but for the groups proportions can vary between 0 and 1. Since the response variable here is the number of *yesses* and number of *nos* we'll code the response as a data frame using `cbind()`.

```
lr1 <- glm(cbind(Menarche, Total - Menarche) ~ Age, data = menarche, family = "binomial")
summary(lr1)
```

```
#
# Call:
# glm(formula = cbind(Menarche, Total - Menarche) ~ Age, family = "binomial",
#     data = menarche)
#
# Deviance Residuals:
#     Min       1Q   Median       3Q      Max
# -2.0363  -0.9953  -0.4900   0.7780   1.3675
#
# Coefficients:
#              Estimate Std. Error z value Pr(>|z|)
# (Intercept) -21.22639    0.77068  -27.54   <2e-16 ***
# Age           1.63197    0.05895   27.68   <2e-16 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
```

[57] Data from: Milicer, H. and Szczotka, F., 1966, Age at Menarche in Warsaw girls in 1965, Human Biology, 38, 199-203

```
# (Dispersion parameter for binomial family taken to be 1)
#
#     Null deviance: 3693.884  on 24  degrees of freedom
# Residual deviance:   26.703  on 23  degrees of freedom
# AIC: 114.76
#
# Number of Fisher Scoring iterations: 4
```

We can interpret the slope of the model as log of the change in the odds ratio. In this case, the slope (1.632) tells us that each year of age increases the odds of menarche by 5.114-fold.

Furthermore (though not so interesting) the odds are massively in favor of a "no" for a newborn girl (odds ratio for age $= 0$ is $e^{-21.22}$ or $6.051 * 10^{-10}$, about 1.6 billion:1 odds. This highlights one feature of of logistic regression - the models are often only meaningful over a limited range of the predictor variable - in this case about 10-16 years [58].

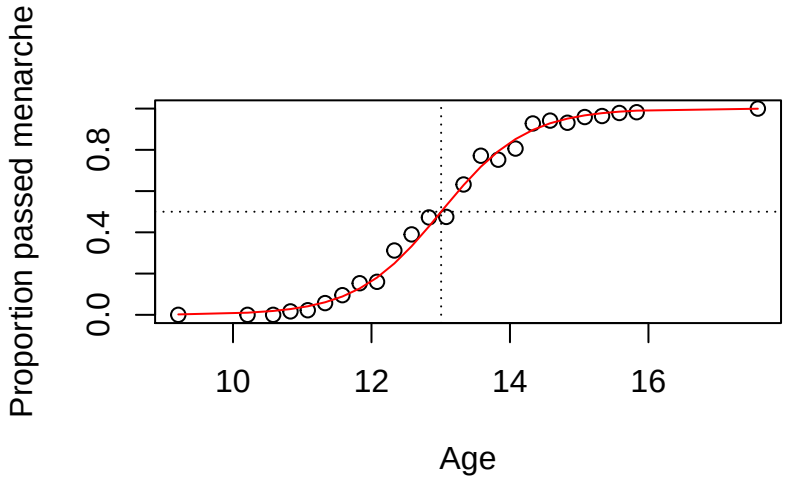
What is more interesting is to use the model to characterize the median age at menarche, or the age at which the probability is 0.5. At this point the odds ratio is 1, so we can find this by dividing the intercept estimate by the slope estimate.

```
-summary(lr1)$coef[1, 1]/summary(lr1)$coef[2, 1]
```

```
# [1] 13.00662
```

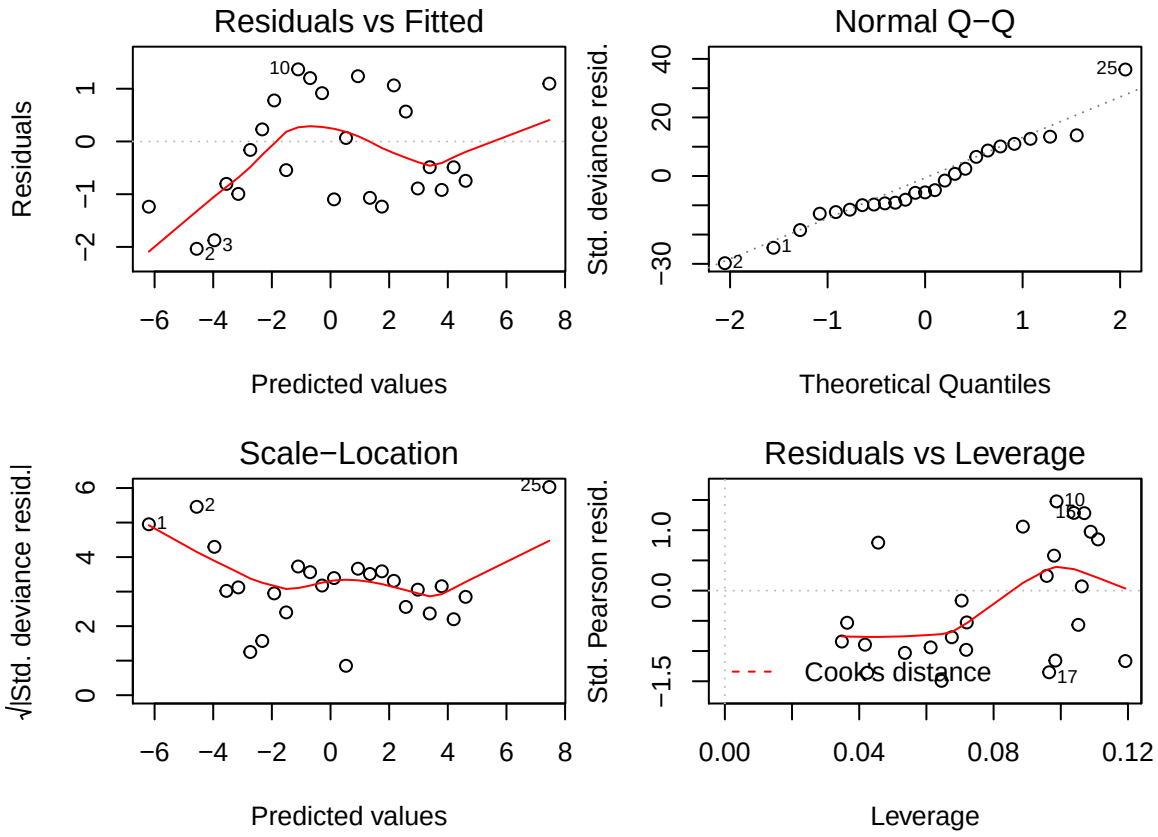
The model tell us that at 13 years of age the probability is 50%.

```
plot(Menarche/Total ~ Age, data = menarche, ylab = "Proportion passed menarche")
lines(menarche$Age, lr1$fitted, type = "l", col = "red")
abline(v = 13.006, h = 0.5, lty = 3)
```




```
par(mfrow = c(2, 2), mar = c(4, 4, 2.5, 0.5))
plot(lr1)
```

[58] This applies much more broadly to linear models of all kinds - they *probably don't* make sense outside of the range of the data from which they were characterized.



We don't need to assume normality of the residuals here, but a lack of pattern in the residuals does suggest that the model fits well - there does not seem to be any latent variability that would be explained by adding more terms.

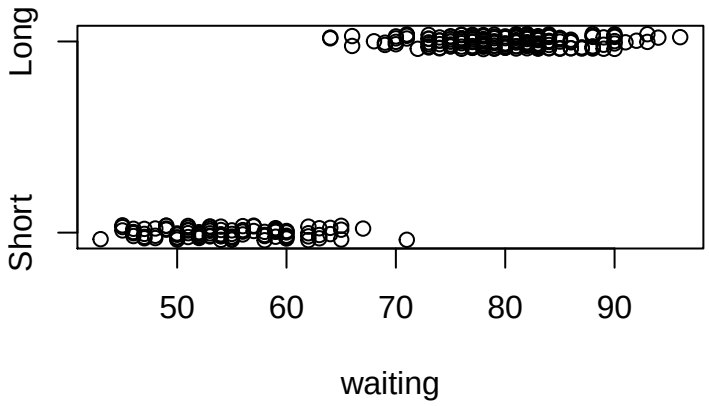
Overall performance of the model can be evaluated by comparing the Null and Residual deviance (see output from `summary(lr1)`, above). The Null deviance represents deviance of a model with only the grand mean, so the reduction in deviance represents the deviance explained by the model - about 3667 on one degree of freedom. In this case the p-value is not different from zero.

```
with(lr1, pchisq(null.deviance - deviance, df.null - df.residual, lower.tail = FALSE))
```

```
# [1] 0
```

For a second example let's consider the `faithful` data we examined in Chapter 6 - we have the length of the eruption (`eruptions`) and the length of time between one eruption and the next (`waiting`) for 272 eruptions of the geyser "Old Faithful". For this example we'll create a factor for eruption length

```
data(faithful)
faithful$length <- cut(faithful$eruptions, breaks = c(0, 3, 6), labels = c("S",
    "L"))
with(faithful, plot(waiting, jitter(as.numeric(length) - 1, amount = 0.04),
    ylab = "", yaxt = "n"))
axis(side = 2, at = 0:1, labels = c("Short", "Long"))
```



Here you can see we have plotted the data with a bit of jitter in the y-axis to help us better see points that would otherwise over-plot. Notice that here we are dealing with the individual events, so the response is truly binary - either an eruption is "Short" or "Long". We can fit a logistic regression as we did before.

```
lr2 <- glm(length ~ waiting, family = "binomial", data = faithful)
summary(lr2)
```

```
#
# Call:
# glm(formula = length ~ waiting, family = "binomial", data = faithful)
#
# Deviance Residuals:
#      Min        1Q    Median        3Q       Max
# -2.60716  -0.00778   0.00299   0.02362   1.82481
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept) -45.5228    10.7540  -4.233 2.30e-05 ***
# waiting       0.6886     0.1632   4.220 2.44e-05 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# (Dispersion parameter for binomial family taken to be 1)
#
#     Null deviance: 354.387  on 271  degrees of freedom
# Residual deviance:  25.163  on 270  degrees of freedom
# AIC: 29.163
#
# Number of Fisher Scoring iterations: 9
```

We'll interpret the coefficients as before - each increase of one minute in the waiting time increases the odds ratio by `exp(0.6886)` or 1.99 that the next eruption will be longer than 3 minutes.

For a waiting interval of 63 minutes we can calculate the odds as the exponent of the sum of the intercept and 63 times the slope (`exp(summary(lr2)$coef[1,1]+63*summary(lr2)$coef[2,1])`) or 0.117.

As above we can find the point with a 50% probability by setting the odds ratio equal to 1 or logit=0 and solving $a + bx$ for x; the ratio of the intercept to the slope. This gives a waiting time of 66.11 minutes

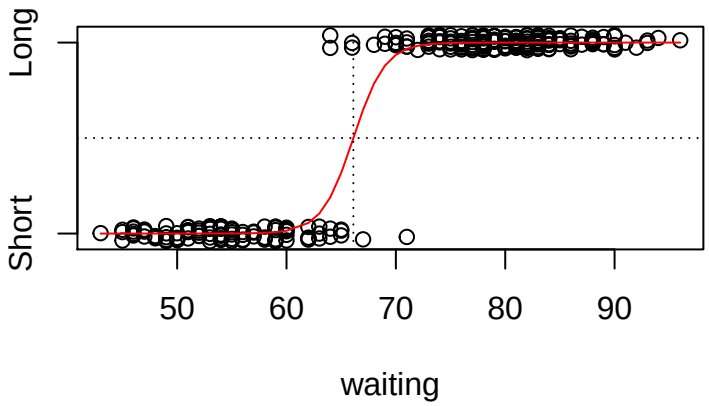
```
-summary(lr2)$coef[1, 1]/summary(lr2)$coef[2, 1]
```

```
# [1] 66.11345
```

Plotting this model shows that the fit isn't as perfect as the first example, but we're plotting the individual

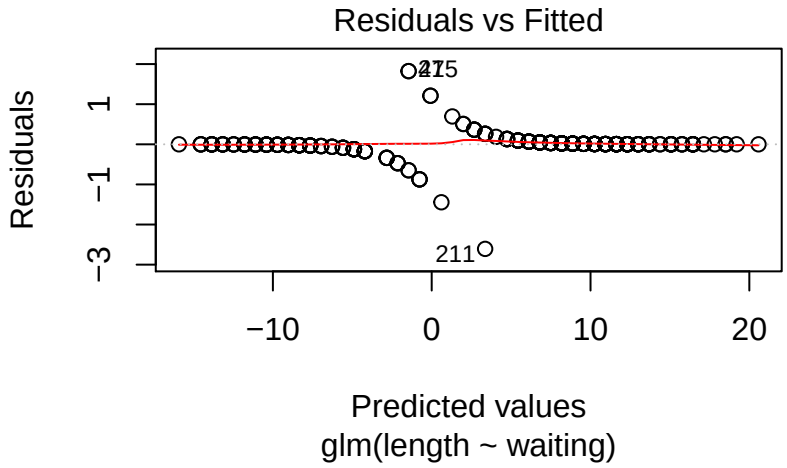
data points were which can't take a value between 0 and 1.

```
with(faithful, plot(waiting, jitter(as.numeric(length) - 1, amount = 0.04),
    ylab = "", yaxt = "n"))
axis(side = 2, at = 0:1, labels = c("Short", "Long"))
lines(cbind(faithful$waiting, lr2$fitted)[order(faithful$waiting), ], col = "red")
abline(h = 0.5, v = 66.113, lty = 3)
```



One feature of a model of this sort is there is some pattern in the residuals - these represent values of `waiting` for which fitted values are less than about 0.9 and greater than about 0.1 - since the variable "length" can only take a value of "long" or "short", the model can't predict an intermediate value.

```
plot(lr2, which = 1)
```



Logistic regression can be extended with multiple predictors and multinomial responses, but that is beyond the scope of these notes.

III. Fitting other non-linear models

Sometimes the model that we want to fit is not linear, but we know (or at least hypothesize) what it should be. For example, enzyme kinetics are often characterized by Michaelis-Menten kinetics, where reaction rate is affected by enzyme concentration in the following way:

$$f(x, (K, V_m)) = \frac{V_m x}{K + x}$$

where V_m is the maximum reaction rate and K is the concentration that yields half of V_m. We can fit such

an equation to data using the function `nls()`. We'll demonstrate with some data on the kinetics of Pyridoxal phosphate (PP) which is involved in catalyzing the breakdown of Glutamate.

```
ppg <- read.csv("../Data/PP-Glutamate.csv", comm = "#")
head(ppg)
```

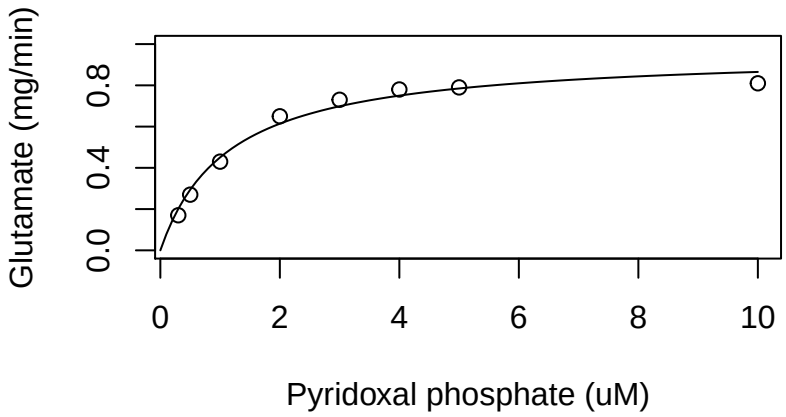
```
#    PP Glutam
# 1 0.3   0.17
# 2 0.5   0.27
# 3 1.0   0.43
# 4 2.0   0.65
# 5 3.0   0.73
# 6 4.0   0.78
```

We have concentration of PP (`PP`) and the reduction in concentration of Glutamate (`Glutam`). `nls()` may work better if we give starting values (guesses) from which it will begin optimizing. Here we'll specify values near the max and half that value.

```
ek1 <- nls(Glutam ~ Vm * PP/(K + PP), data = ppg, start = list(K = 0.5, Vm = 1))
```

If we had a large data set in might be simpler to specify `start` this way: `list(K = max(ppg$Glutam)/2, Vm = max(ppg$Glutam))`. But this is specific to this particular functional form (Michaelis-Menten), and might not work on types of models. Sometime we can get away with a naive call to `nls()` without specifying `start`, as below - we may get a warning, but the coefficients are still calculated correctly. But we can't count on that happening all the time.

```
plot(Glutam ~ PP, data = ppg, ylim = c(0, 1), ylab = "Glutamate (mg/min)", xlab = "Pyridoxal phosphate
paras <- summary(ek1)$coeff[, 1]
curve((paras[2] * x)/(paras[1] + x), from = 0, to = 10, add = TRUE)
```




```
summary(nls(Glutam ~ Vm * PP/(K + PP), data = ppg))$coeff
```

```
# Warning in nls(Glutam ~ Vm * PP/(K + PP), data = ppg): No starting values specified for some paramete
# Initializing 'Vm', 'K' to '1.'.
# Consider specifying 'start' or using a selfStart model
```

```
#    Estimate Std. Error   t value     Pr(>|t|)
# Vm 0.963516 0.03953373 24.372000 3.136878e-07
# K  1.137725 0.15943859  7.135821 3.814812e-04
```

```
summary(ek1)
```

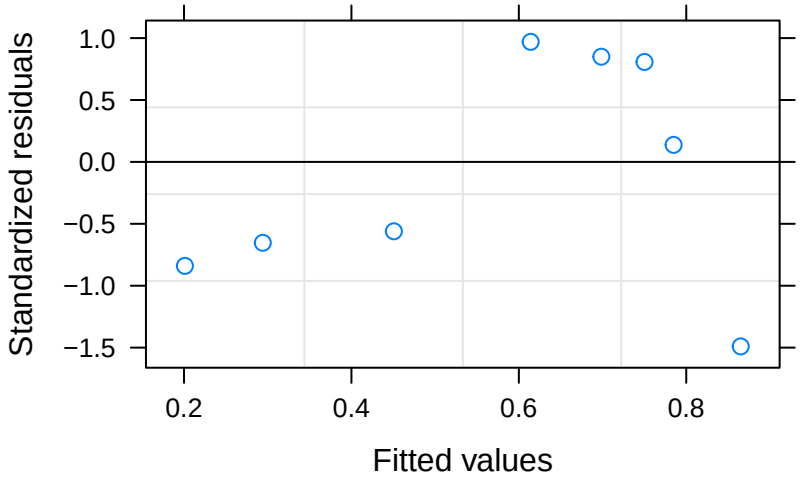
```
#
# Formula: Glutam ~ Vm * PP/(K + PP)
#
```

```
# Parameters:
#    Estimate Std. Error t value Pr(>|t|)
# K   1.13772    0.15944   7.136 0.000381 ***
# Vm  0.96351    0.03953  24.372 3.14e-07 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.03697 on 6 degrees of freedom
#
# Number of iterations to convergence: 6
# Achieved convergence tolerance: 8.036e-06
```

We have our estimates for the parameters K and V_m and t-tests for difference from zero. As we saw in Chapter 11 we can use the Standard errors of these estimates to test hypotheses about them. For example, if we hypothesized that $V_m = 1$, a quick look at the output from `summary(ek1)` should convince us that we can't reject that hypothesis because the hypothesized value (1) is within 2 standard errors (2*0.0395) of the estimate (0.964). We could also calculate a p-value as in Chapter 11.

Note that `nls()` can handle many types of functions, and we may be trying to fit the wrong model to the data. Checking the residuals is a good way to see how well the model fits. Note that the package `nlme` is required to plot `nls` objects.

```
require(nlme)
plot(ek1)
```



In this case there is a hint of structure here (low residuals at low fitted values, higher at high fitted values), but it is harder to be certain with a small number of data points.

Using `nls()` only makes sense if we have some hypothesized functional form for the relationship - there are an infinite number of possible functional forms, and all `nls()` can do is tell us how well any one of them fits your data.

If we're comparing models fit by `nls()` (*nls objects*) we can use the function `anova()` to compare 2 models *as long as they are nested*. Lets fit a 2nd order and 3rd order polynomials to our `ppg` data to demonstrate.

```
ek2 <- nls(Glutam ~ a + b * PP, data = ppg, start = list(a = 1, b = 1))
ek3 <- nls(Glutam ~ a + b * PP + c * PP^2, data = ppg, start = list(a = 1, b = 1,
    c = 1))
```

Note that here in the calls to `nls()` we are specifycing our coefficients with `a`, `b`, `c` as names. You can use any names you want for the parameters, but the variable names must match variable names in the data argument.

```
anova(ek2, ek3)
```

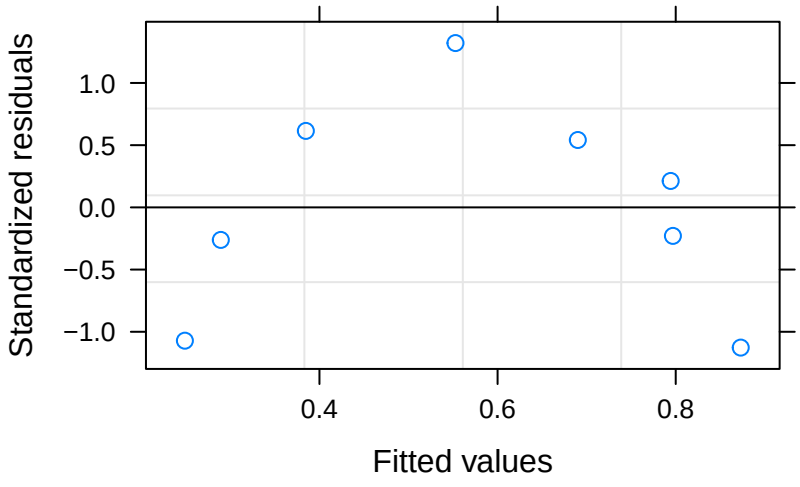
```
# Analysis of Variance Table
#
# Model 1: Glutam ~ a + b * PP
# Model 2: Glutam ~ a + b * PP + c * PP^2
#   Res.Df Res.Sum Sq Df  Sum Sq F value   Pr(>F)
# 1      6   0.189460
# 2      5   0.027118  1 0.16234  29.933 0.002779 **
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
summary(ek3)
```

```
#
# Formula: Glutam ~ a + b * PP + c * PP^2
#
# Parameters:
#    Estimate Std. Error t value Pr(>|t|)
# a  0.186116   0.052679   3.533 0.016686 *
# b  0.213932   0.029402   7.276 0.000767 ***
# c -0.015311   0.002798  -5.471 0.002779 **
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 0.07364 on 5 degrees of freedom
#
# Number of iterations to convergence: 1
# Achieved convergence tolerance: 2.223e-08
```

This indicates that the 3rd order polynomial is a better fit than the 2nd order, and it is significant in all parameters. But the residuals plot shows much more evidence of structure than the plot of `ek1` residuals and the residual standard error is better. More importantly, we have theoretical reason to prefer the first model.

```
plot(ek3)
```



Note: `nls()` uses an iterative numerical solver. In some instances it may fail to converge on a solution, and you will get an error message telling you this. You can set the max iterations higher using `control = list(maxiter=200, warnOnly=TRUE)`.

VI. Exercises

1) Using the built-in data set `mtcars` fit a logistic regression of `am` (transmission type – a factor coded as 0 & 1) as a function of `wt` (weight in thousands of pounds). At what weight is an automatic transmission three times as likely as a manual transmission?

2) The dataset given below [59]comes from a third order polynomial, and includes some random noise. Use `nls()` to fit both a 1st order ($a + bx$) and a 3rd order ($a + bx + cx^2 + dx^3$) and compare the models - how does the residual error compare? Does the distribution of the residuals give any reason to prefer one over the other?

```
x=c(1.08,4.55,2.92,5.74,8.36,5.84,5.08,8.05,9.74,1.26,4.63,3.43,8.22,5.52,9.30,7.70,4.10,
1.16,7.89,6.40,7.29,1.96,6.59,4.35,4.65)
y=c(8.42,14.10,15.80,15.60,16.60,17.10,14.80,16.40,21.80,11.50,14.20,18.90,16.20,15.40,
20.60,13.40,16.70,15.20,18.30,14.00,13.20,15.20,13.30,14.40,15.40)
```

3) Plot the functions given by the two models you created above over the range -3:13.

---

[59]If you copy and paste this data into the console you may have to delete some returns and add ";" before the y, or just paste the x and y data separately.

Chapter 20: Writing functions

Customizing R with functions and packages

I. Introduction

Two of the greatest strengths of R are *adpatability* and *extensibilty* – R can be customized to do almost anything (of course this may require time). In many cases, the tool you want may already exist in a package, and the tools we discussed in Chapter 5 should help you find it. In this chapter we'll consider two of the most important tools that support adaptability and extensibility - creating new functions and creating packages.

II. Creating a function

Defining a function

It is dead simple to define a function in R - simply use the function `function()` to define the new function. Some of what follows here is specific to R, but some is "common sense" programming advice.

```
fn.demo <- function(x, y) x^2 - y
fn.demo(4, 5)
```

```
# [1] 11
```

Some important detail to note about the use of `function()`
* The new function (`fn.demo()` in this case) is created in the user's workspace. It will only persist between R sessions if the workspace is loaded *or* if an R script that defines the function is *sourced* (the code in the file is run). * The syntax here is idiosyncratic – the parentheses contain the *arguments* for the new functions. The code that is run when the function is called is given *after* the parentheses. For a simple function such as we have here, when everything fits on one line, no braces are necessary. If the code spans line breaks it must be wrapped in braces (`{ }`).

```
fn.demo <- function(x, y = 5) {
    x^2 - y
}
fn.demo(4)
```

```
# [1] 11
```

```
fn.demo(4, y = 2)
```

```
# [1] 14
```

```
fn.demo(y = 2, x = 4)
```

```
# [1] 14
```

In this example we show how default values can be specified in the function definition. As long as no value is specified for `y`, then the default value is used.

```
##     z <- x^2 - y
##     if (verbose == TRUE) {
##         cat(paste(x, " squared minus ", y, " = ", z, sep = ""))
##     } else {
##         return(z)
##     }
## }
fn.demo(5)
```

```
# [1] 20
```

```
fn.demo(3, ver = TRUE)
```

```
# Error in fn.demo(3, ver = TRUE): unused argument (ver = TRUE)
```

```
z
```

```
# [1] 2.0 1.0 1.5
```

Here we demonstrate an important point (first remove `z` (`rm(z)`) in case there was something called `z` in the workspace). After running our `fn.demo()` function a couple of times we call `z` but it does not exist, though it is defined in the function. This highlights that functions *are not evaluated in the user workspace*, rather they are evaluated in a separate environment. This means that `z` is created in a different environment where the function is evaluated. So if you want to get a value out of a function you need to make the function *return* the value using `return()`. Note that if the end of the function is reached and `return()` has not been called the *last* expression evaluated will be returned.

```
fn.demo <- function(x, y = 5, verbose = FALSE) {
    z <- x^2 - y
    if (verbose == TRUE) {
        cat(paste(x, " squared minus ", y, " = ", z, sep = ""))
    } else {
        return(z)
    }
    z  ## <- added
}
fn.demo(5)
```

```
# [1] 20
```

```
fn.demo(5, ver = TRUE)
```

```
# 5 squared minus 5 = 20
```

```
# [1] 20
```

We can demonstrate this by adding a line `z` (see `## added`, above). If we call `fn.demo()` with `verbose=T` the `if()` expression avoids the call to `return()` and so `z` is returned. With `verbose=F` (default) the `return()` is evaluated and so `z` is not returned a second time.

If you want to return more than one value you can return a vector, list, or data frame (depending on what makes sense), and such an object can be assigned or indexed like any other vector in R.

```
fn.demo <- function(x, y = 5) {
    c(x, y, x^2 - y)
}
fn.demo(6)
```

```
# [1]  6  5 31
```

```
fn.demo(6)[3]
```

```
# [1] 31
```

```
a <- fn.demo(3.7)
a
```

```
# [1] 3.70 5.00 8.69
```

In general functions are expected to return objects, and to have *no other effects*. However there may be occasions where it might make sense to have a function that modifies an object that exists in the user's

workspace. In this case we use the `<<-` operator [60].

```
a <- 5
bad.fn <- function(b = 7) {
    a <<- b
}
a
```

```
# [1] 5
```

```
bad.fn(14)
a
```

```
# [1] 14
```

This example is ridiculously simplistic, and it doesn't make sense to do this - you could just assign the output of the function to a variable. However, there may be situations where it *might* make sense to do this - for example if one were working with large arrays of data the copying of the data that would be required to return the entire matrix might slow down processing enough to make it worthwhile to write a function using the `<<-` operator.

However, if that seemed to be needed, it might be worth writing the function so as to also print a message about the object that was changed. What we want to avoid is creating functions that have *invisible side effects*. Recall that in general the only way to change an object in R is via explicit assignment - functions written with `<<-` violate this situation, and so should only be undertaken with great care.

Finding and preventing errors

This brings us to the topic of finding and preventing errors. Lets see what happens when we call our function with the wrong type of data as an argument.

```
fn.demo <- function(x, y = 5) {
    z <- x + 10 - y
    return(z)
}
```

```
fn.demo(x = "black")
```

```
# Error in x + 10: non-numeric argument to binary operator
```

Here we've fed our function a character value when it was expecting a numeric value. In this case it is rather easy to find the error, but in a longer function in might be rather difficult. The function `debug()` can help us track down problems in functions. We'd call the function like this:

```
debug(fn.demo)
fn.demo(x = "black")
```

Use of `debug()` can't be easily demonstrated in a document like this - you'll need to try it yourself. Note that we actually run 2 lines to get `debug()` working - the first launches `debug()` on the next expression. Using `debug()` gives us a view into the environment where the function is evaluated - in RStudio we see the *function* environment in the environment browser rather than our workspace when we are running `debug()`.

Of course, *preventing* errors is even better than finding them when they occur. We can use the function `stop()` to halt execution and print an error message.

```
fn.demo <- function(x, y = 5) {
    if (is.numeric(x) == FALSE | is.numeric(y) == FALSE)
```

[60] Actually it is a bit more subtle - when `<<-` is used in an assignment the assignment occurs *in the environment from which the function was called*. This could be the workspace, but if the function is called by another function it could be the environment where the calling function is evaluated.

```
        stop("don't be ridiculous \n x & y must be numeric")
    z <- x + 10 - y
    return(z)
}
fn.demo(x = "black")
```

```
# Error in fn.demo(x = "black"): don't be ridiculous
#  x & y must be numeric
```

The most insidious errors are those in which a function returns a value but returns the *wrong* value. A common cause of such is missing values.

```
mean.se <- function(x) {
    m <- mean(x)
    se <- sd(x)/(sqrt(length(x)))
    cat(paste("mean = ", m, "; se = ", se, "\n"))
}
a <- c(2, 3, 4, 5, 6)
mean.se(a)
```

```
# mean =  4 ; se =  0.707106781186548
```

```
b <- a
b[3] <- NA
mean.se(b)
```

```
# mean =  NA ; se =  NA
```

Here we get a correct result when there are no missing values and the `NA` value propagates if there is an `NA`. This is not too bad because we know we have a problem (though if the function was more complex *finding* the problem might not be tough). A more serious problem would arise if we incompletely fixed the `NA` problem.

```
mean.se <- function(x) {
    m <- mean(x, na.rm = TRUE)
    se <- sd(x, na.rm = TRUE)/(sqrt(length(x)))
    cat(paste("mean = ", m, "; se = ", se, "\n"))
}
mean.se(a)
```

```
# mean =  4 ; se =  0.707106781186548
```

```
mean.se(b)
```

```
# mean =  4 ; se =  0.816496580927726
```

The function runs and returns values, but does not return the value we intended because we are including `NA` values in our calculation of n (`length(x)`).

```
mean.se <- function(x) {
    m <- mean(x, na.rm = TRUE)
    se <- sd(x, na.rm = TRUE)/(sqrt(sum(is.na(x) == FALSE)))
    cat(paste("mean = ", m, "; se = ", se, "\n"))
}
mean.se(a)
```

```
# mean =  4 ; se =  0.707106781186548
```

```
mean.se(b)
```

```
# mean =  4 ; se =  0.912870929175277
```

Of course, this example is obvious as so many are - it is hard to make a simple demonstration of a subtle problem.

Tips for writing functions

- Break the problem down into steps and write several functions - this modularity makes debugging easier and allows you to reuse more of your code.
- Name arguments as clearly as possible.
- Comment your code liberally.
- Think imaginatively about what could go wrong, check inputs, and provide intelligible error messages.

Developing custom functions could save a lot of time and make your work simpler. This blog post (http://rforpublichealth.blogspot.com/2014/07/3-ways-that-functions-can-improve-your.html) provides some examples.

If you develop functions that are useful for your work you could store them in a special R script file. It is possible to modify your R environment to source this file on start-up - see (http://www.statmethods.net/interface/customizing.html). But if you get to that point you might want to consider creating a package.

III. Creating a package

If we think of R packages as items that are available from the repositories we may miss some of their utility. A package could be a useful tool for loading custom functions and including some documentation on their use. It could be a tool for sharing tools, routines, and even data among collaborators. While the R packages that we access from CRAN have been approved, we can rather easily create our own packages.

First we'll install some packages.

```
install.packages("devtools")
install.packages("roxygen2")
library(devtools)
library(roxygen2)
```

Now we can go ahead and create the package.

```
create("easy.demo")
```

Your working directory will now contain a folder called `easy.demo`. This includes several files. The first one you should attend to is the "DESCRIPTION" file. Basically provide your contact information and a short description of the package contents.

Now we'll copy the function `mean.se()` from our last example above, paste it into a new R script file and save it as "meanANDse.R" in the "R" directory in the "easy.demo" folder.

Now we need to add documentation. This will be done mostly automatically by `roxygen2`. We'll paste these specially formatted comments into the beginning of the "mean_se.R" file, and then save. Obviously, for your own function you would add the relevant documentation here.

```
#' Mean and Standard Error
#'
#' This function returns mean and standard error of the mean.
#' @param x An object whose mean and se will be returned.
#' @keywords summary
#' @export
#' @examples
```

```
#' a <- rnorm(n=15,mean=6,sd=4)
#' mean_se(a)
```

Now all we need to do is set our working directory to the the "easy.demo" folder and invoke the function `document()`

```
setwd("easy.demo")
document()
setwd("..")
```

All that is left to do is install it.

```
install("easy.demo")
```

Type `? meanANDse` and see if it worked.

```
meanANDse(a)
```

```
# mean =  4 ; se =  0.707106781186548
```

Not too difficult. Notice that `easy.demo` does not appear in your "Packages" tab. It was installed using `install()` from the `devtools` package. A full installation would be done like this: `install.packages("easy.demo",repos=NULL,type="source")`.

Note that `_` and any mathematical operators (`+`, `-`, `*`, `/`, and `^`) can't be used in function names in packages. The period (`.`) can be used in function names but it has special meaning - if we named this function within the package `mean.se` it would be interpreted as a function to determine the mean of objects of type `se` - this is how generic function are denoted (E.G. `plot.lm()` is the function that plots a regression line from an `lm` object, when we call `plot()` on an `lm` object, `plot.lm()` actually does the work). This is useful if you develop a function that creates an object of type `mess-` you can develop methods like `summary.mess()` and `plot.mess()` to extend generic R functions to deal with your new object type.

A comprehensive guide to developing R packages is beyond these notes, and several exist already [61]. Instead I will include a couple of helpful links here should you want to investigate this further. I used several resources while developing this demo, this page was the most useful. There is also an RStudio page that give more information.

VI. Exercises

1) Write a function that does something interesting or amusing. The goal is not massive complexity (but it should require a few lines of code), just that you play around a bit with something that is a somewhat more complex than the trivial examples we've included here. Your function should include some basic error checking (for example making sure arguments are of the correct type).

If you're at a loss for ideas here are a few: 1) calculate the difference between row and column sums of a square matrix (see `?colSums`); 2) take a text string and extract numbers from it to generate several random polygons; 3) extract coefficients and calculate their 95% confidence intervals and plot them.

Note: You will need to add the chunk option `error=TRUE` to R code chunks that might produce errors (or as a document option) if you want errors to show up in your knitr output - otherwise errors stop compilation.

[61] The definitive guide can be found at (http://cran.r-project.org/doc/manuals/R-exts.html).