R Refresher

R. Todd Jobe

Draft: July 24, 2008

Contents

I July 25 **1**

1 Introduction to R **3**
1.1 R Basics . . . 3
1.1.1 How R Stores data . . . 3
1.1.2 A Typical R Session . . . 5
1.1.3 Getting help . . . 8
1.1.4 R Packages . . . 10
1.2 Objects . . . 11
1.2.1 Vectors . . . 12
1.2.2 Matrices . . . 14
1.2.3 Lists . . . 14
1.2.4 Dataframes . . . 15
1.2.5 Factors . . . 17

2 Function and Script Basics **21**
2.1 Functions . . . 21
2.1.1 Basics . . . 22
2.1.2 Brackets: { ([. . . 23
2.1.3 Nesting and Scope . . . 23
2.1.4 The most important data manipulation functions . . . 25
2.2 Scripts . . . 26
2.3 Loading Packages . . . 27
2.4 Exercises . . . 27

3 Getting Data into and out of R **29**
3.1 Importing data . . . 29
3.1.1 Importing text files . . . 29
3.1.2 readLines . . . 30
3.1.3 read.table and its variants . . . 31
3.1.4 Importing directly from databases . . . 33

4 Formatting and Output **35**
4.1 Well-formed data frames . . . 35
4.1.1 The Brute Force Method . . . 36
4.1.2 The Elegant Method . . . 39

4.2 Data output in Tables and Figures . . . 40
4.2.1 Tabular output . . . 40
4.3 Xtables . . . 41
4.3.1 Figure Output . . . 42
4.4 Encapsulating Input and Output . . . 42
4.5 Exercises . . . 44

II July 26 **45**

5 Scripting and Functions **47**
5.1 Text Editors . . . 47
5.1.1 MS Word . . . 47
5.1.2 The R GUI . . . 47
5.1.3 TextPad . . . 48
5.1.4 Emacs and ESS . . . 49
5.2 Executing Scripts in R . . . 50
5.3 Writing functions in R . . . 51
5.3.1 Building scripts from history files . . . 52
5.3.2 Building functions from scripts . . . 54

6 Loops and Scripting Best Practices **57**
6.1 Looping through code . . . 57
6.1.1 vectorization . . . 57
6.1.2 for loops . . . 58
6.1.3 apply functions . . . 60
6.2 4 R's for R Programming . . . 61
6.2.1 Right . . . 61
6.2.2 Repeatable . . . 62
6.2.3 Robust . . . 63
6.2.4 Readable . . . 64
6.3 Exercises . . . 65

7 Statistical Models **67**
7.1 Linear Models . . . 67
7.1.1 Standard Arguments . . . 68
7.1.2 Formulas . . . 73
7.1.3 Alternatives . . . 75
7.2 Model Selection . . . 79
7.3 Bootstrapping . . . 79

8 Plotting **81**
8.1 Basic Plotting . . . 81
8.1.1 Scatterplots . . . 82
8.1.2 Box Plots and Histograms . . . 82
8.1.3 Line Plots . . . 84

8.1.4 Bar Plots 88
8.1.5 Multiple Plots 89
8.2 Lattice Graphics 90
8.3 Execises 93

Day I

July 25

Session 1

Introduction to R

Commands

setwd	save.image	ls	quit	rm	load
?	example	help.search	help.start	class	->,<-
c	matrix	list	data.frame	as.*	attributes
runif	cbind	rbind	factor	levels	sample
boxplot	dist	function	print	assign	seq
rep	sample	return	rnorm	rpois	source
library	invisible				

1.1 R Basics

R is an open source statistical programming language. It is becoming a standard for statistical analysis in biology. R is primarily a programming language. The is one of the main advantages that it has over other statistical packages. It is rare for scientists to use off-the-shelf statistical tests today. Our data rarely conform to the assumptions of classical tests, and computer intensive methods allow for more complex inferential models. As a result, inferential models are often unique to the data. The flexibility of a programming language combined with a the availability of many different functions relating to probability and statistics makes it very easy to build such models in R.

1.1.1 How R Stores data

Working with R is unlike working with most software we use on a day to day basis. Modern software typically employs a multiple document interface to work with data objects. That is, data are saved on the hard disk and multiple pieces of data can be opened simultaneously in separated windows (think documents in MS Word or worksheets in MS Excel). Data are manipulated using mouse clicks and or short commands.

R was developed out of the S language which was originally written in the early 80's, long before windowed interfaces were available. It was developed to be used on a terminal that has only a single window and only accepts command typed onto the screen, rather than mouse clicks. Though Graphical Users Interfaces (GUIs) have been built for R today, it still it's terminal feel. A R session consists of commands entered onto the command line.

R Stores its workspace in RAM

It is important for R users to understand the different ways that computers interact with data, and where data is stored during an R session. Distinguishing between RAM and hard disk storage is vital.

Computers today contain between 512MB and a few GB of Random Access Memory (RAM). This memory typically resides on cards that plug directly into the motherboard of the computer. Random Access Memory is an extremely fast but temporary (volatile) mode of storage. When a program starts in your operating system, the commands necessary to make the program run are read from the hard disk into RAM so that the commands can be read quickly and the processes executed by the CPU. As particular bits of data are needed, they are read from the hard disk and placed in RAM. When these bits are no longer needed, they are either saved to the hard disk by the program or deleted from the RAM. When the program is terminated, all the RAM occupied by the program is released.

Most software today keeps a copy of the working data stored permanently on the hard disk and only places bits of data into RAM as they are needed. This minimizes data loss if the program crashes. R is different, however, in that all the commands and data are stored in RAM during the entire session unless the user explicitly exports the data to a hard storage location. The upside of this is that calculations performed on the data are extremely fast. There are two downsides to having data stored in RAM during a work session. First, if the program crashes all data that has not been saved explicitly is lost. The second downside is that the size of data that can be loaded by R is limited to the amount of RAM available on the computer. On 32-bit systems (e.g. Windows XP), this means that the maximum size of all the data in an R workspace is roughly 2GB. 64-bit systems overcome this limitations (e.g. Mac, *nix) because their upper limit of available RAM is on the order of terabytes. Even with 64-bit systems, however, you can't store more data than you have RAM. So, for large phylogenetic and spatial data-sets, the platform of choice is likely a Macintosh with as much RAM as you can afford.

The R working directory

When R starts, it chooses a working directory. It looks for any `.RData` file (see below) in this working directory and loads the data from this file into the workspace. The workspace is an abstract concept that refers the data stored in RAM for a particular R session. During an R session, the working directory is default path to which R looks for any external input. So, if you read in data from a text file, R looks for the named file in the working directory unless you have specified the full path to the file. By default the working directory in windows installations of R is the installation path of R (typically "`C:\Program Files\R\R` *version*`\bin`"). On Mac and *nix systems the default working directory is your home directory (`~`).

It is usually more convenient to set the working directory for a particular analysis to a unique folder. Create a folder for each analysis you perform. In Windows systems you can create a shortcut in this folder to the R executable. Set the Start In property of the shortcut to be the working folder. Then, R can be started in the working directory by double-clicking on the shortcut.

The .Rdata file

The permanent storage of workspaces is done with a binary file named `.RData`. When you quit an R session (`quit()`), you are prompted to save the workspace. If you reply `yes`, then R stores

its current workspace in a `.RData` file on the working directory. When you start R again in this directory, the data that was saved is automatically restored. If you want to restore a particular `.RData` file during a session, you can use the command `load("`*path to .RData*") There is nothing special about the filename except that R looks for the file named `.RData` by default when it starts. So, you could store workspaces with any name you choose and just use the `load` command to put them in the current R session.

The .RHistory file

When R exits, the user is prompted to save the data. This includes not only the `.RData` file, but also an `.RHistory` file. The `.RHistory` file is simple a text file containing all the commands that have been executed for that particular R workspace. Again, when R starts it looks for a `.RHistory` file to load into the session. The commands in this history can be accessed in the `File` menu on the GUI, or sequentially recalled by pressing the up-arrow at the command line. As with `.RData` a particular history file can be saved under a different name and recalled later during another R session using the commands `savehistory` and `loadhistory`.

1.1.2 A Typical R Session

As outlined above, you begin a typical R session by staring R in the working directory of the analysis you want to perform. This working directory should contain all the external data that you might need during the R session (e.g. Excel workbooks, shapefiles). When you finish your first session in this directory, a `.RData` and `.RHistory` file will be created. It is important to remember that while you are working in R, the objects you create are *not* stored permanently on the hard disk. This only occurs when the `.RData` is updated on exit or when you explicitly save the workspace from the command line.

The prompt and line continuation

The command line begins with the character `>`. You may type commands on this command line and execute them by pressing `Enter`. R ignores whitespace (e.g. tabs, spaces, newlines), so if the line you execute does not contain a full command (i.e. there is an open parenthesis or bracket) the command entry is continued on the next line. Each continuation line is prefixed with the character `+`.

Escaping command execution

At any time during the typing or execution of a command you may signal a break. In the GUI version of R this is done using the `Esc` key. In terminal versions of R, you use the standard escape sequence for the terminal. C-c in Mac and *nix, C-c C-c in Emacs.

Scripts and External Data

R can read commands not only from the keyboard but also from text documents. These documents simply contain the commands to execute (without the `>` character that is present at the command line) and they may be `source`d into the current session (See 2.2). Commands are executed sequentially, just as they are written. The standard extension for R scripts is `.r`, though R can read any type of standard text file regardless of the extension.

R can also read data from external files. These data will nearly always be in some tabular form with column headings and values in rows separated by delimiters. For people who don't use `,` as a delimiter for decimal places (e.g. most US researchers), the best format for data storage is as a comma-separated values file (`*.csv`). These files can be read using the command `read.csv`. Tab-delimited files are common as well, though there are frequently problems with this method because the default function for reading tab delimited files, `read.table`, actually recognizes any whitespace as a delimiter. So, if there are any strings, such as column headers or categorical variables that contain spaces, R attempts to recognize each word as a separate field. This usually results in an error. This behavior can be overcome by specifying delimiters explicitly with the argument `delim` in `read.table`. Any character can be a field delimiter.

Saving your work

R *does not* store your work as you go. If MS Word crashes while you are writing a document, you can usually recover it the next time you start Word. If R crashes while you are working on the command line, everything you have done for that session is lost. The only data that persist are stored in the `.RData` file from your previous session.

There are two methods for preventing this kind of data loss, and you should probably use both. The first is to frequently save your R workspace while you are in your R session using the command `saveWorkspace()`. The second is to write all of your commands, from importing the data to generating final output, in script files. If you ever lose your data, you can just execute the commands that you have saved to create your workspace anew. This latter method is by far the safest way to make sure you don't lose any of your work, and is just good programming practice in general. You should strive to write scripts that take you from raw data to output in a minimal number of steps.

An R Workspace Exercise

Below is a quick set of exercises to help you understand explicitly how an R session works and how R stores data.

1. Create a folder to store your analysis.

   ```
   mkdir -p ~/RCourse/day1hw
   ```

2. Start R

3. Change the working directory of R to your analysis folder using either the GUI or the command line (`setwd("`*pathtofolder*`")`)

   ```
   > setwd("~/RCourse/day1hw")
   ```

4. Create a vector containing the integers 1 through 5 and assign it to a symbol.

   ```
   > v1 <- 1:5
   > v1
   ```

```
[1] 1 2 3 4 5
```

5. Save your workspace

```
> save.image()
```

6. Create a vector containing the integers 6 through 10 and assign it to a symbol.

```
> v2 <- 6:10
> v2

[1]  6  7  8  9 10
```

7. Save a backup of your workspace in a different file

```
> save.image("Backup.RData")
```

8. List your workspace.

```
> ls()

 [1] "ctl"          "group"        "height"       "levs"         "light"
 [6] "light2"       "lm.D9"        "lm.D90"       "lv"           "mat1"
[11] "mat2"         "mat3"         "opar"         "out.var"      "popWeights"
[16] "resample"     "sample.df"    "sampleDf"     "smpWeights"   "smpWeightsDf"
[21] "tmp"          "tmp.fun2"     "tmp.fun3"     "tmp.fun4"     "tmpDist"
[26] "tmpNames"     "tmpWeights"   "tmplist"      "trt"          "v1"
[31] "v2"           "weight"       "x"            "y"
```

9. Create a vector containing the integers 11 through 15 and assign it to a symbol.

```
> v3 <- 11:15
> v3

[1] 11 12 13 14 15
```

10. Quit R without saving your workspace.

```
> quit("no")
```

11. Start R again.

12. List your workspace. Note what objects it contains.

```
> ls()

character(0)
```

13. Load your backup workspace into R.

```
> load("Backup.RData")
```

14. List your workspace. Note what objects it contains.

```
> ls()
```

```
 [1] "ctl"          "group"        "height"       "levs"         "light"
 [6] "light2"       "lm.D9"        "lm.D90"       "lv"           "mat1"
[11] "mat2"         "mat3"         "opar"         "out.var"      "popWeights"
[16] "resample"     "sample.df"    "sampleDf"     "smpWeights"   "smpWeightsDf"
[21] "tmp"          "tmp.fun2"     "tmp.fun3"     "tmp.fun4"     "tmpDist"
[26] "tmpNames"     "tmpWeights"   "tmplist"      "trt"          "v1"
[31] "v2"           "weight"       "x"            "y"
```

15. Quit R, saving your workspace.

```
> quit("yes")
```

16. Start a text editor and open the `.RHistory` file in your working directory.

1.1.3 Getting help

Once you master the basics of the R language, you open yourself to a wealth of information and statistical algorithms. The obstacle to efficient analysis, then, becomes finding the right information to match your analysis and massaging your data into a form that can be analyzed by the packages you downloaded. While it's not possible in every situation, you should do your best never to reinvent the wheel. Open source software such as R has a long history of providing extensive help online in the form or users groups and listservs. R also provides on-board help documentation for commands

?

If you want to look at the help pages for a particular command, you simply type `?`*command*.

example

At the bottom of most R help pages are a series of commands the provide examples of how to use the documented command. You can execute these examples from the R command line using `example(`*command*`)`.

```
> example(lm)
```

```
lm> require(graphics)

lm> ## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
lm> ## Page 9: Plant Weight Data.
lm> ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
```

```
lm> trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)

lm> group <- gl(2,10,20, labels=c("Ctl","Trt"))

lm> weight <- c(ctl, trt)

lm> anova(lm.D9 <- lm(weight ~ group))
Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value Pr(>F)
group      1 0.6882  0.6882  1.4191  0.249
Residuals 18 8.7292  0.4850

lm> summary(lm.D90 <- lm(weight ~ group - 1))# omitting intercept

Call:
lm(formula = weight ~ group - 1)

Residuals:
    Min      1Q  Median      3Q     Max
-1.0710 -0.4937  0.0685  0.2462  1.3690

Coefficients:
         Estimate Std. Error t value Pr(>|t|)
groupCtl   5.0320     0.2202   22.85 9.55e-15 ***
groupTrt   4.6610     0.2202   21.16 3.62e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared: 0.9818,        Adjusted R-squared: 0.9798
F-statistic: 485.1 on 2 and 18 DF,  p-value: < 2.2e-16


lm> summary(resid(lm.D9) - resid(lm.D90)) #- residuals almost identical
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-7.772e-16 -3.608e-16  0.000e+00 -1.110e-17  0.000e+00  3.220e-15

lm> opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))

lm> plot(lm.D9, las = 1)      # Residuals, Fitted, ...

lm> par(opar)
```

```
lm> ## model frame :
lm> stopifnot(identical(lm(weight ~ group, method = "model.frame"),
lm+                     model.frame(lm.D9)))
```

help.search

If you do not know the specific command name, but know the type of analysis you wish to so, you can try searching all of your installed R packages for a help file that contains the keywords of your analysis:

```
> help.search("Moran")
```

help.start()

If you would like to browse all the R documentation installed on your system you can open them in a web-browser using `help.start()`. To find a particular command you need to navigate to `Packages`, then look under the package you want to browse. You can also find general R information here like the R Programming Guide and the R Syntax Definition.

CRAN

The Comprehensive R Archive Network (CRAN: `cran.r-project.org`) is the project responsible for mirroring R binaries and packages at various locations throughout the world. They have links to a Wiki and user forums. Typically, if you have a problem getting something to work in R after you've looked at the standard documentation you should do a web search naming your problem and appending the keywords `CRAN` and `R`.

1.1.4 R Packages

R itself is simply a programming language. If that were all that R could do, it would just be a tiny speck in an ocean of similar programming languages. The real power of R is contained in the contributed packages written by people like you who want to make their hard work available to everyone else. There are over a thousand R packages, and one of the biggest challenges you will face in doing analysis in R is choosing the right package for the job.

You can see a list of all available packages and their descriptions at `cran.r-project.org`. You can install R packages from the GUI or simply using the command `install.packages("`*packagename*`")`. You will be asked to select a repository from which to download. Repositories closer to you will likely have faster downloads, though for US researchers the fasted downloads nearly always come from Berkley (repository `CA1`).

CRAN Task View

Wading through a thousand packages to find the one that suits your needs is an enormous time waster. Recently, CRAN has added some functionality to its website that provides a shortcut for choosing packages. The link from the website is `Task Views`. In task views the R packages are separated into disciplines such as Econometrics and Environmetrics. Each task view provides a vignette that summarizes the most important packages for standard analyses in that discipline.

1.2 Objects

Everything in R is an object. Data are objects. Functions are objects. This is a distinction from truly object-oriented programming languages, where objects contain methods, and functions may not have an object associated with them. You can view a list of all the objects in your workspace using the `ls` command.

```
> load("objectExample.RData")
> ls()
```

```
 [1] "ctl"          "group"        "height"       "levs"         "light"
 [6] "light2"       "lm.D9"        "lm.D90"       "lv"           "mat1"
[11] "mat2"         "mat3"         "opar"         "out.var"      "popWeights"
[16] "resample"     "sample.df"    "sampleDf"     "smpWeights"   "smpWeightsDf"
[21] "tmp"          "tmp.fun2"     "tmp.fun3"     "tmp.fun4"     "tmpDist"
[26] "tmpNames"     "tmpWeights"   "tmplist"      "trt"          "v1"
[31] "v2"           "weight"       "x"            "y"
```

You can see the values held in an object by typing the object name at the command line.

```
> x
```

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

In R, each object has an associated class. I describe the most important classes below. In the above example there are 5 objects in the workspace. They are not all of the same class.

```
> class(x)
```

```
[1] "integer"
```

```
> class(y)
```

```
[1] "character"
```

Think of classes as drawers of different sizes and styles. If you have an organization task (in this case placing data into objects) you look through the different drawers and pick the one that is the right size and shape for the things you want to store. Choosing the right drawer is important, because it controls how easy it is for you to get your data back, and how easy it is to work on it.

Objects that are stored in a workspace are associated with a symbol. A symbol can be thought of as the name of an object. The command to assign a symbol is a two character sequence that looks like an arrow, `->`. Assignment of an object to a symbol can occur in any order. *symbol*`<-`*object* and *object*`->`*symbol* are both valid statements.

1.2.1 Vectors

The simplest R object is a vector. It is a 1–dimensional, ordered container for one kind of data. These data could be numbers: (1,5,7,9,10). They could be text (”how”,”now”,”brown”,”c. They could be a series of binary (yes/no) values: (TRUE,FALSE,TRUE,TRUE). The command for entering (also known as concatenating) such a sequence is `c()`.

```
> c(T, F, TRUE, FALSE)

[1]  TRUE FALSE  TRUE FALSE

> c(1, 2, 3, 4)

[1] 1 2 3 4

> c("hello", "world")

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

We can perform elementary math operations like addition and subtraction on the variable.

```
> tmp <- c(1, 3, 4, 5, 4)
> tmp + 2

[1] 3 5 6 7 6
```

You’ll notice that 2 was added to each value in the vector. This is the main advantage of choosing the right ”drawer” for your data. You can perform operations simply and quickly across the entire dataset, (or a specific subset of it as we’ll see later).

The : operator generates an integer sequence from the first value to the second value. It works for negative values as well, and you can use parenthesis to control the output (in this case parentheses are used for evaluation order like in typical mathematics)

```
> 1:5

[1] 1 2 3 4 5

> -1:5

[1] -1  0  1  2  3  4  5

> -(1:5)

[1] -1 -2 -3 -4 -5
```

Subsetting

To get a specific element a vector we use square brackets with the index of the vector element

```
> tmp

[1] 1 3 4 5 4

> tmp[1]

[1] 1

> tmp[2]

[1] 3
```

We can retrieve multiple elements from a vector using a sequence of indices.

```
> tmp[c(1, 2, 3)]

[1] 1 3 4

> tmp[1:3]

[1] 1 3 4
```

We can also use logical vectors that are the same length as the sub-setting vector.

```
> tmp[c(T, T, T, T, F)]

[1] 1 3 4 5

> tmp > 2

[1] FALSE  TRUE  TRUE  TRUE  TRUE

> tmp[tmp > 2]

[1] 3 4 5 4

> tmp > 2 & tmp < 4

[1] FALSE  TRUE FALSE FALSE FALSE

> tmp[tmp > 2 & tmp < 4]

[1] 3
```

Logical sub-setting is a common task and it pays to spend a little extra time understanding exactly how it works.

1.2.2 Matrices

Most often, scientific data come in the form of multiple vectors of similar length that are related in some way. For instance, you may collect both temperature and growth rate for a series of plants at different locations. It is more appropriate to combine such vectors into a matrix in which the rows correspond to locations or replications and the columns represent the variables you collect. The `matrix` function is meant to do just that. It coerces a vector of data into a 2–dimensional container.

```
> matrix(1:10, nrow = 2)

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

The command above took the vector 1:10 and fit it into a matrix of 2 rows. You'll notice that the vector was placed into the matrix column by column. Another way to say this is that R by default always iterates across rows first, then columns. You can change that default behavior with the argument `byrow`.

```
> matrix(1:10, nrow = 2, byrow = T)

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

In the previous two commands I specified a matrix using one vector and one dimension of the vector. I could just have easily specified both dimensions and let R fill up any empty spaces.

```
> matrix(1:10, nrow = 2, ncol = 10)

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

You can extend matrices to more than two dimensions using the `array` function. We'll not go over that here because the operations on arrays of more than two dimensions are identical to those of matrices.

1.2.3 Lists

A list and a vector are virtually the same thing with one exception. A vector must contain all the same data type in every slot. A list can have any data type in each slot. The great thing about lists is that they can be containers for containers and be nested infinitely.

```
> tmplist <- list(c(2, 4, 6, 8), c(1, 1, 1, 1, 1), matrix(1:10,
+     nrow = 2), c("hello", "world"))
> tmplist
```

```
[[1]]
[1] 2 4 6 8

[[2]]
[1] 1 1 1 1 1

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

[[4]]
[1] "hello" "world"
```

You can access the components of a list using double square brackets `[[]]`. Unlike lists, you can't access multiple components using vectors of indices.

```
> tmplist[[1]]

[1] 2 4 6 8

> tmplist[[1:2]]

[1] 4
```

1.2.4 Dataframes

The most frequently used object type is a dataframe. It is a 2–dimensional container like a matrix, but unlike a matrix, each column can have a different data type. Most often the rows of a data frame are the individual observations of a study. The columns are the covariates and responses for each observation. Dataframes may be created from a matrix or may be created column by column.

```
> as.data.frame(matrix(1:10, nrow = 5))

  V1 V2
1  1  6
2  2  7
3  3  8
4  4  9
5  5 10

> data.frame(resp = 1:10, expl = sample(c(T, F), 10, replace = T))

   resp  expl
1     1  TRUE
2     2 FALSE
3     3 FALSE
4     4  TRUE
```

```
5     5 FALSE
6     6 FALSE
7     7  TRUE
8     8 FALSE
9     9 FALSE
10   10 FALSE
```

Dataframes have 3 attributes: `names` `rownames`, and `class`.

```
> tmp <- data.frame(resp = 1:10, expl = sample(c(T, F), 10, replace = T))
> attributes(tmp)

$names
[1] "resp" "expl"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10

$class
[1] "data.frame"
```

You can access the columns of a dataframe by name and by index.

```
> tmp$resp

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

> tmp[, 1]

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

> tmp[, "resp"]

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

Accessing objects by name also applies to any object that has a names attribute.

```
> tmp <- list(stuff1 = 1:5, nostuff = NA)
> tmp

$stuff1
[1] 1 2 3 4 5

$nostuff
[1] NA

> attributes(tmp)

$names
[1] "stuff1"  "nostuff"
```

```
> tmp$stuff1

[1] 1 2 3 4 5
```

It is worth mentioning here two alternative ways to build dataframes and matrices: `cbind` and `rbind`. Occasionally, you will have to piece dataframes together by binding columns which have the same number of rows, or binding rows that have an equal number of columns. `cbind` binds columns of a dataframe (or a vector) together that have the same number of rows. `rbind` binds rows together that have the same number of columns. By default `cbind` and `rbind` create matrices, but they have dataframe counterparts, `cbind.data.frame` and `rbind.data.frame`.

```
> mat1 <- matrix(1:10, nrow = 5)
> mat2 <- matrix(rnorm(15), nrow = 5)
> v1 <- runif(5)
> cbind(mat1, mat2, v1)

                                                    v1
[1,] 1  6  1.3760439 -0.9160760  1.7490382 0.1626130
[2,] 2  7  1.0254423  1.6323398  1.2115488 0.1959679
[3,] 3  8 -0.5859041 -1.1657121 -0.3591283 0.8027563
[4,] 4  9  0.6149840  1.5502385  0.2141133 0.6388274
[5,] 5 10 -0.4997532  0.9894213 -0.8717418 0.6971929

> mat3 <- matrix(rnorm(16), ncol = 2)
> rbind(mat1, mat3)

             [,1]       [,2]
 [1,]  1.00000000  6.0000000
 [2,]  2.00000000  7.0000000
 [3,]  3.00000000  8.0000000
 [4,]  4.00000000  9.0000000
 [5,]  5.00000000 10.0000000
 [6,]  2.12103668 -0.1092064
 [7,] -1.05106551  1.1122826
 [8,]  0.43801827 -0.6657890
 [9,] -0.41328658  0.2108364
[10,]  0.26347346 -0.1528133
[11,]  0.07776033 -0.8213993
[12,]  0.14395873  1.6855975
[13,]  1.19290836 -0.5053494
```

1.2.5 Factors

The final container that we'll discuss here is a factor. Factors are categorical variables. They are stored just like vectors, but with and additional `levels` attribute.

```
> c("light", "shade", "light", "shade")
```

```
[1] "light" "shade" "light" "shade"
```

```
> factor(c("light", "shade", "light", "shade"))
```

```
[1] light shade light shade
Levels: light shade
```

```
> tmp <- factor(c("light", "shade", "light", "shade"))
> levels(tmp)
```

```
[1] "light" "shade"
```

Factors can also be ordered. You should be careful to order your factors if an ordering exists in the levels. This will cause analysis (e.g. plotting and regression) to be done in the correct order rather than by the default which is alphabetical, and makes a difference for degrees of freedom in regression procedures.

```
> height <- rnorm(30, c(10, 8, 6), 5)
> levs <- c("L", "M", "H")
> lv <- sample(levs, 30, replace = T)
> height
```

```
 [1] 11.456598 13.282706  2.347772 14.508829  5.646261 13.707445  8.051463
 [8]  6.377971 14.619579  9.801535  6.383039 10.593779 14.223488  4.891057
[15]  1.035119 14.405061  9.499780  1.141328 17.330403  9.938224  6.280072
[22]  3.748102  7.309500  3.433915 12.903078  1.627278  5.114657  6.780878
[29] 11.266248  5.790863
```

```
> lv
```

```
 [1] "M" "M" "M" "M" "H" "L" "H" "M" "H" "M" "H" "L" "H" "L" "H" "L" "H" "H" "H"
[20] "M" "M" "M" "M" "L" "H" "M" "H" "L" "H" "L"
```

```
> light <- factor(lv)
> light
```

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

```
> light2 <- factor(lv, levels = levs, ordered = T)
> light2
```

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

One final note on factors is that the levels of the factor remain constant even if a particular level doesn't occur in a vector.

```
> light2[light2 == "L"]
```

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

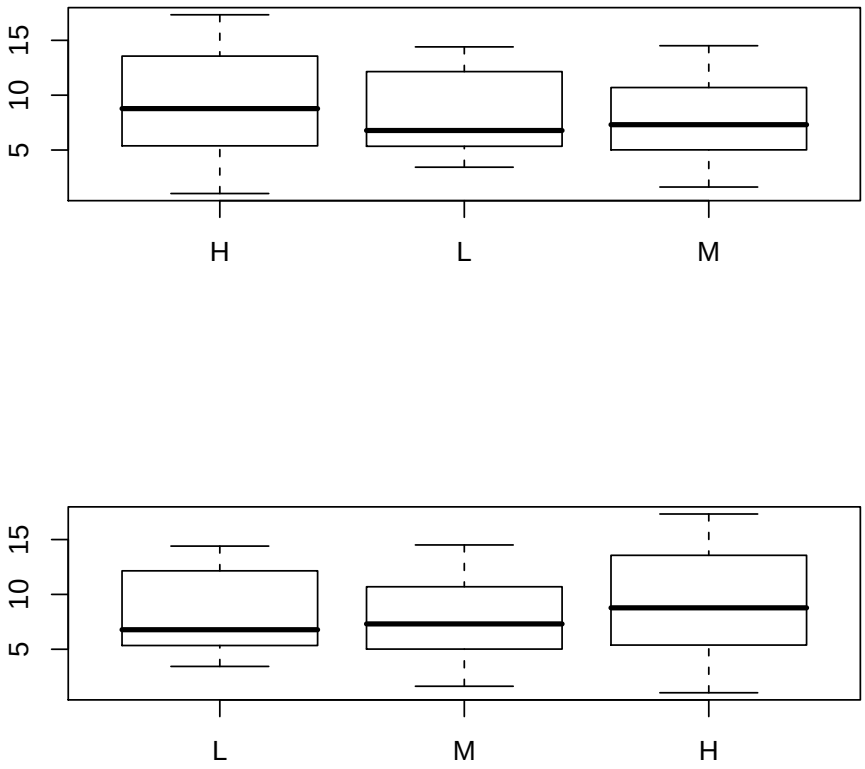

Figure 1.1: B
oxplot illustrating the difference between unordered factors (top, sorted alphabetically) and ordered factors (bottom).

Session 2

Function and Script Basics

2.1 Functions

R is function-based. This is a distinction from other truly object-oriented programming languages, which are class based. Nearly everything that is done in R is done through functions. Functions are meant to receive data (in the form of arguments) perform some calculation or data manipulation task, and return an output to the user. Often, we assign the output of a function to a symbol, though this is not absolutely necessary.

```
> tmpDist <- dist(1:10)
> tmpDist

   1 2 3 4 5 6 7 8 9
2  1
3  2 1
4  3 2 1
5  4 3 2 1
6  5 4 3 2 1
7  6 5 4 3 2 1
8  7 6 5 4 3 2 1
9  8 7 6 5 4 3 2 1
10 9 8 7 6 5 4 3 2 1
```

In the above example, `dist` is a function. We sent it input (the vector `1:10`) and it sent us output which we stored in the symbol `tmpDist`. Were we to look at the documentation for the `dist` function, we would discover that it calculates a pairwise distance matrix on the vector.

Functions are designed to be black boxes. You don't have to see functions working. You just call them up, send them data, and they work. This has two advantages for end users of R. First, it's easy for people to use code for different problems. For instance, if you created a function that calculated the sums of squares from a vector of data, you could use that function in many different contexts, from calculating variance to linear regression. Second, functions can be called from other functions, allowing you to reuse your own code, to say, calculate the sums of squares for many different pieces of data without having to type the code over and over.

2.1.1 Basics

Functions are objects just the same as data are objects. When you list the objects in your workspace there is no distinction between the two. If you want to see the definition of a function you enter the name of the function (without parentheses) at the command just as you would a data object.

```
> dist

function (x, method = "euclidean", diag = FALSE, upper = FALSE,
    p = 2)
{
    if (!is.na(pmatch(method, "euclidian")))
        method <- "euclidean"
    METHODS <- c("euclidean", "maximum", "manhattan", "canberra",
        "binary", "minkowski")
    method <- pmatch(method, METHODS)
    if (is.na(method))
        stop("invalid distance method")
    if (method == -1)
        stop("ambiguous distance method")
    N <- nrow(x <- as.matrix(x))
    d <- .C("R_distance", x = as.double(x), nr = N, nc = ncol(x),
        d = double(N * (N - 1)/2), diag = as.integer(FALSE),
        method = as.integer(method), p = as.double(p), DUP = FALSE,
        NAOK = TRUE, PACKAGE = "stats")$d
    attr(d, "Size") <- N
    attr(d, "Labels") <- dimnames(x)[[1]]
    attr(d, "Diag") <- diag
    attr(d, "Upper") <- upper
    attr(d, "method") <- METHODS[method]
    if (method == 6)
        attr(d, "p") <- p
    attr(d, "call") <- match.call()
    class(d) <- "dist"
    return(d)
}
<environment: namespace:stats>
```

To call a function you just type its name and follow it with parentheses. Parentheses delineate the function arguments from other parts of R code (Parentheses, then have both a mathematical and syntactical meaning in R). A function is defined to accept a sequence of arguments separated by commas when it is called. These, arguments go inside the parentheses and may be named explicitly using the syntax (*argument=value*) or are evaluated in the order in which the y appear in the function definition. In our distance example above:

```
> dist(x = 1:10, method = "euclidean")
> dist(x = 1:10, "euclidean")
```

```
> dist(1:10, method = "euclidean")
> dist(1:10, "euclidean")
```

all mean the same thing.

You can define your own functions using the function `function` (surprise!). In the function definition, arguments in parentheses represent definition of the arguments rather than an input. If you use the syntax *argument=value*, then *value* becomes the default value of the argument when the function is called. You'll notice above that the definition for the `dist` function has 4 arguments with default values: `method`,`diag`,`upper`, and `p`. So, when we made our first call to `dist`: `dist(1:10)`, the input vector was assigned to the `x` argument and every other argument was assigned the default.

The body of a function is delineated by curly brackets {}. These are not necessary when the function is only one statement long, but this rarely occurs. Inside the body of the function arguments that were defined between the parentheses become symbols that have the value given them in the function call. So, you can define a function with an argument `x` and then use `x` in the body just as another symbol. When the function is called, `x` takes on the value that you assigned it in the call.

Most function generate output. By default, R assigns the output of a function be the output of the last statement executed before the }. This is not safe, however, and it is better to use the `return(`*return value*`)` command to return objects from functions.

Occasionally, function are created for their side-effects rather than their return values. This often the case with plotting function, and data export functions. In these cases you can use the special function `invisible())` as the return value. This suppresses even the `NULL` return value that is the default for `return()`

2.1.2 Brackets: { ([

It is easy for beginners to get confused with the different types of brackets that are used in R. Here I provide a reference for distinguishing these delimiters.

parentheses: () Used in every function call (even if there are no arguments used). Function arguments go in between the parentheses. Also used to determine the order of mathematical operations.

square brackets: [] Used only for indexing and subsetting operations on containers.

double square brackets: [[]] Used for indexing lists.

curly brackets: {} Surrounds blocks of statements that should be executed together. These blocks occur as the statements within a function, the statements to be executed in a loop (using the command `for`), or conditionally executed statements (using the command `if...else`).

2.1.3 Nesting and Scope

Symbols in R are stored in environments. You main workspace environment has a special name, `.GlobalEnv`. When you type `ls()` at the command line, the symbols stored in `.GlobalEnv` are listed. When a function is called, a new environment is created specifically for this function. Symbols created in this environment (as part of the command list in the function) last only as long as the function is executed. Stated another way, the scope of symbols is specific to the environment in which they were created.

```
> tmp.fun <- function() {
+     x1 <- 2 + 4
+     y1 <- 10 + 6
+     print(x1)
+     return(y1)
+ }
> x1
```

```
Error:  chunk 41 (label=scope)
Error in eval.with.vis(expr, .GlobalEnv, baseenv()) :
object "x1" not found
```

In the function above, we cannot access the symbol `x` from the command line because it was released from memory when `tmp.fun` completed execution. The environment of the main workspace `.GlobalEnv` is special in that it can be accessed from within any function.

```
> x <- 10
> tmp.fun2 <- function() {
+     y <- 10 + x
+     return(y)
+ }
> tmp.fun2()
```

```
[1] 20
```

Symbols that are defined in both a function and the `.GlobalEnv` are overwritten by the function when the function is in scope.

```
> y <- 10
> tmp.fun3 <- function() {
+     y <- 11
+     print(y)
+     return(invisible())
+ }
> tmp.fun3()
```

```
[1] 11
```

If you wish for a symbol to be stored permanently in the `.GlobalEnv` from within a function you can use the `assign` command.

```
> tmp.fun4 <- function() {
+     x2 <- 10
+     assign("out.var", x2, envir = .GlobalEnv)
+     return(invisible())
+ }
> tmp.fun4()
> out.var
```

```
[1] 10
```

2.1.4 The most important data manipulation functions

There are a few function for generating vectors that are used very frequently. I outline them below.

seq Generate regular sequences. While the `:` is useful for generating integer sequences, there will be many times that you want to generate regular sequences that are not necessarily increasing integers.

```
> seq(1.5, 5, by = 0.5)

[1] 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

> seq(1.5, 5, length = 20)

 [1] 1.500000 1.684211 1.868421 2.052632 2.236842 2.421053 2.605263 2.789474
 [9] 2.973684 3.157895 3.342105 3.526316 3.710526 3.894737 4.078947 4.263158
[17] 4.447368 4.631579 4.815789 5.000000
```

rep Replicates values. You can generate vectors that are repetitions of particular numbers of vectors.

```
> rep(1, times = 5)

[1] 1 1 1 1 1

> rep(1:2, times = 5)

 [1] 1 2 1 2 1 2 1 2 1 2

> rep(1:2, each = 5)

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

sample It is often necessary to randomly sample from vectors. This can be a permutation, a subsample with replacement, or a sub- or super- sample with replacement.

```
> sample(1:10)

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

> sample(1:10, size = 3)

[1] 6 8 1

> sample(1:10, size = 12)

Error:  chunk 47 (label=sample)
Error in sample(length(x), size, replace, prob) :
cannot take a sample larger than the population
 when 'replace = FALSE'
```

```
> sample(1:10, size = 12, replace = T)

 [1] 10  7  6 10  8  1  3  4 10  1  6  4
```

It's easy enough to extend random sampling from vectors to random sampling of rows from a dataframe using the `...` argument, which we will discuss later, and the function `nrow` which returns the number of rows in the dataframe.

```
> sample.df <- function(df, ...) {
+     smp <- sample(1:nrow, ...)
+     return(df[smp, ])
+ }
```

r* In addition to generating random samples, we also often want to generate random variates from a specified distribution. R has many built-in probability distribution such as the Binomial, Geometric, Poisson, Exponential, Gamma, Beta and the classic Normal distribution. If you are unfamiliar with a particular distribution there is no better place to learn about it than Wikipedia. To generate random variables from a probability distribution you simply type the command `r`*distribution*(*size*,*param1*,*param2*). For the normal distribution the two parameters are the mean and the standard deviation. For the Poisson there is only one parameter (λ):

```
> rnorm(10, mean = 0, sd = 1)

 [1] -1.65722007 -1.60250942 -0.47008351 -0.25618195 -1.48299646 -1.23027924
 [7]  0.06924713  2.58444604 -0.18059833 -0.51707936

> rpois(10, lambda = 1)

 [1] 3 0 0 2 2 2 1 2 1 2
```

In addition to generating random variates from all these distributions, you can also get the probability density `d`*distribution*, cumulative distribution `p`*distribution* and the quantile function `q`*distribution* for each distribution as well.

2.2 Scripts

Typing on the command-line is error prone and can be frustrating. Most users of R only type on the command-line when they are testing short commands that they can easily type without error. Barring those cases, commands are generally placed into text files and then read into R. These R scripts can be read into R in at least 3 ways.

The first, and most straight–forward is simply to copy the lines of code from the text file and paste them directly into the command line. Many R users do all their analyses in this way. There are some weaknesses associated with this approach, though. First, you have to remember to execute the commands in your text file in the proper order. Second, what is copied and pasted into the command line is stored in the `.RHistory` file. This may not seem like a big deal, but if you were copying a 50 line function over and over again, it would be difficult to backtrack through your history to find other commands you might want to reference.

The second way to read commands into R is only available when R is run in a particular GUI interface. In R for both Mac and Windows you can open up scripts in a window of the R software. There are click-able options to send lines, regions or entire files to the R command line. This process does not send the commands into the history file.

The final way to send commands to R is through directly reading a text file at the command line. This is done via the `source` command. This is preferred method of debugging R functions, since it is the quickest to implement and the easiest to repeat. The example below sources a text file to the R command line.

```
> source("sumsq.r")
```

2.3 Loading Packages

We have already discussed that one of the advantages of R is the abundance of packages written for it, and that these packages may be downloaded through the GUI or by the command `install.packages`. Downloading a package places it in the R install folder. It must be loaded however, into a running R process in order to have access to its functionality. R only loads a few packages by default when it boots. So, every time you restart R you need to load your packages you again.

```
> library(MASS)
```

It is common to write `library` commands at the top of source files. Another option is to write a source file that loads initially when R boots. When R boots, it looks for a file called `.First` and sources this file. You can include your `library` commands in this file. As a side note, you may also include a `.Last` file that is run when R closes.

2.4 Exercises

In this exercise you will simulate some field collected data (weights of animals from 2 populations), and perform elementary descriptive analyses on the datasets.

1. Create a workspace for this analysis
2. Create a simulated datasets of weights from a normal distribution whose mean weight is 32g and whose standard deviation is 5g.
3. Create a similar dataset with mean 10g and standard deviation 5g.
4. Put these into a dataframe with 2 columns (population and weight).
5. Calculate the sample mean and standard deviation for the total population and each individual population.
6. Put all the data and outputs into a single list
7. Extra credit: Bootstrap confidence intervals sample means of each populations. Are they different?

Session 3

Getting Data into and out of R

About 80% of R analysis is getting data from a file on the computer into a format that is useful for analysis on R. Most statistical analyses require few lines of code. Data manipulation, on the other hand, represents the bulk of R code. In today's lesson we concentrate on how to get data into R, and the properties of well-formed dataframes. We also discuss different output tactics for generating manuscript-ready figures and tables in R, and how best to encapsulate such functionality in R scripts. There is an entire documentation module written on data import and export for R, which is available from the HTML help (`help.start()`).

3.1 Importing data

Most scientific data is stored in one of three formats: text files, spreadsheets, and databases. R can read and write data from any of these. Text files are often the easiest format to read and write from, but they have disadvantages. First, most researchers do no enter data directly into text files. They instead use forms on spreadsheets or databases to enter data. So, text files typically are exported data from some other format. Spreadsheets and databases, on the other hand, have the disadvantage that is not as straightforward to read and write from them as text files. In windows, Object Database Connectivity (ODBC) protocols make this process easier, and we will learn how to use these connections in R. But, for *nix and Mac systems text files are still the best way to go, unless you have installed 3rd party ODBC drivers.

3.1.1 Importing text files

There are three main commands used for importing data into R from text files: `scan`, `readLines`, and `read.table`. I address each of these in order.

scan

`scan` is used mainly to read in single vectors of data.

```
> cat("TITLE extra line", "2 3 5 7", "11 13 17", file = "vector.txt",
+     sep = "\n")
> scan("vector.txt", what = character(0))
```

```
 [1] "TITLE" "extra" "line"  "2"     "3"     "5"     "7"     "11"    "13"
[10] "17"
```

If no file is specified to scan it reads input from `stdin` which is your terminal. You can type in values until you hit `Enter` twice. Using this protocol, you could read in a single column from an spreadsheet by simply copying and pasting it into the terminal.

Scan has a few optional arguments that it shares with the more common `read.table`. The first is the `file` argument. This is a character string that is the path to the data file. By default, R looks in your working directory first for this file. You can also use connections (FTP and HTTP URLs) as file names. So, you can download data directly from the Internet or another computer.

```
> scan("http://www.unc.edu/~toddjobe/RHttpEx.txt")

 [1]   1   2   3   4  10  20  30  40 101 202 303 404
```

Reading data from the web is very handy. So, I have provided a function using some of the commands we learned in the previous chapter that will generate a HTTP URL from my website for downloading data.

```
> getHttp <- function(x, baseUrl = "www.unc.edu/~toddjobe") {
+     paste("http:/", baseUrl, x, sep = "/")
+ }
```

The second common argument is `sep`, which is a character vector of field delimiters. By default, `sep=""` which means that all whitespace is counted as a delimiter.

```
> scan("http://www.unc.edu/~toddjobe/RHttpEx.txt", sep = "\n")

[1]          1234     10203040 101202303404
```

The final common argument that we will discuss here is `skip` which tells the function to skip a certain number of lines at the beginning of the file. This is useful for files that contains multiple lines of header information.

```
> scan("vector.txt", skip = 1)

[1]  2  3  5  7 11 13 17
```

3.1.2 readLines

`readLines` is very similar to `scan`, but it reads the entire line without consideration for delimiters. Connections and `stdin` can also be used, so you can paste from Excel to `readLines` as well.

```
> readLines("vector.txt")

[1] "TITLE extra line" "2 3 5 7"          "11 13 17"
```

3.1.3 read.table and its variants

`read.table` is the most frequently used import command. It can read text files directly into dataframes. You can think of it as a combination of `scan` which can delineate columns on a single line, and `readLines` which distinguishes rows. As stated above, many of the optional arguments to scan like `file`, `sep`, and `skip`, are present in `read.table`.

For the `read.table` examples we'll be using a text file names "chickens.txt" which is space-delimited file of chick weights by a variety of treatments. Here's what it looks like.

```
> tmp <- read.table("http://www.unc.edu/~toddjobe/chickens.txt")
> tmp[1:10, ]

         V1      V2        V3    V4     V5
1  prtntype prtnamt fshsolamt house weight
2        GN       L         L     F   6559
3        GN       L         L     S   6292
4        GN       L         H     F   7075
5        GN       L         H     S   6779
6        GN       M         L     F   6564
7        GN       M         L     S   6622
8        GN       M         H     F   7528
9        GN       M         H     S   6856
10       GN       H         L     F   6738

> str(tmp)

'data.frame':        25 obs. of  5 variables:
 $ V1: Factor w/ 3 levels "GN","SB","prtntype": 3 1 1 1 1 1 1 1 1 1 ...
 $ V2: Factor w/ 4 levels "H","L","M","prtnamt": 4 2 2 2 2 3 3 3 3 1 ...
 $ V3: Factor w/ 3 levels "H","L","fshsolamt": 3 2 2 1 1 2 2 1 1 2 ...
 $ V4: Factor w/ 3 levels "F","S","house": 3 1 2 1 2 1 2 1 2 1 ...
 $ V5: Factor w/ 25 levels "6249","6292",..: 25 6 2 17 13 8 9 22 14 10 ...
```

Notice that all the columns in this imported dataset are factors and that the column headings were imported as rows in the dataframe. It is the default behavior of `read.table` to ignore column headers. You can cause `read.table` to recognize column headers by using the argument `header=T`.

```
> tmp <- read.table("http://www.unc.edu/~toddjobe/chickens.txt",
+     header = T)
> tmp[1:10, ]

   prtntype prtnamt fshsolamt house weight
1        GN       L         L     F   6559
2        GN       L         L     S   6292
3        GN       L         H     F   7075
4        GN       L         H     S   6779
5        GN       M         L     F   6564
```

```
6       GN      M       L     S   6622
7       GN      M       H     F   7528
8       GN      M       H     S   6856
9       GN      H       L     F   6738
10      GN      H       L     S   6444

> str(tmp)

'data.frame':        24 obs. of  5 variables:
 $ prtntype : Factor w/ 2 levels "GN","SB": 1 1 1 1 1 1 1 1 1 1 ...
 $ prtnamt  : Factor w/ 3 levels "H","L","M": 2 2 2 2 3 3 3 3 1 1 ...
 $ fshsolamt: Factor w/ 2 levels "H","L": 2 2 1 1 2 2 1 1 2 2 ...
 $ house    : Factor w/ 2 levels "F","S": 1 2 1 2 1 2 1 2 1 2 ...
 $ weight   : int  6559 6292 7075 6779 6564 6622 7528 6856 6738 6444 ...
```

This time the columns import as expected, numeric columns are seen as numeric and factor columns are seen as factors. Character strings within columns are automatically converted to factors. If you want to override this automatic conversion you can use the argument `as.is=T`.

There are a few other problems associated with importing data using the defaults of `read.table`. The first is what to do about missing data. Missing data in R is assigned a special symbol `NA`. Given the default behavior of `read.table`, there is no way for missing data to be read correctly unless that missing data has been assigned a specific value and the argument `na.strings` specified. We have already discussed the issue of character strings with spaces in them acting as multiple delimiters.

Fortunately, R provides higher-level functions that read specific types of text files, which overcome these problems to a great degree. The first and most important is `read.csv`. `read.csv` is identical to `read.table` except that `sep=","` and `header=T`. This is probably the safest way to export and read text files in R. Of course the problem of multiple delimiters can still arise if the character string that you are trying to import as factors contain `,` in them. Another common type of text file format is tab-delimited. The function to read such files is `read.delim`. This is a good alternative if you prefer not to use `*.csv` files.

One final extension to `read.table` is the ability to read directly from the clipboard. The wrapper function for this is `readClipboard`. Using `readClipboard` you can highlight multiple columns and rows in an Excel spreadsheet, copy them, and paste them into R.

Common problems reading text files

It is common for users to spend a lot of time looking at indecipherable errors when attempting read in data from text files. Below, I outline some of the common pitfalls and their solutions.

Forgetting to read the header If you use `read.table` to import your text file, you must explicitly set `header=T` in order for the data to be read correctly. Usually, forgetting to set `header=T` will not result in a run-time error when you import the file. Instead, it is likely to show up when you attempt to do some mathematical manipulation on what seems like a numeric column only to discover that it is considered a factor by R. To avoid this error always look at the first few lines of your imported dataframe and also look at its structure using `str`.

Extra delimiters Occasionally, when you export a file from Excel into a `*.csv` or a tab-delimited text file, Excel will add some delimiters for extra columns or rows. This can happen if you have a value that you can't see on one particular row, too many column headers, or a character string that contains a field delimiter. This can also happen if you've opened the text file, edited it directly, and you added some extra lines by accident. The error that you will receive in such cases could be:

```
Warning message:
In read.table(file = file, header = header, sep = sep, quote = quote,  :
  incomplete final line found by readTableHeader on 'tmp.csv'
```

To avoid this error, make sure that all cells in Excel which should not contain data are actually clear. Also, do not edit a text file directly if possible.

3.1.4 Importing directly from databases

One of the most frequently used programs for storing data is Microsoft Excel. Unfortunately, R does not natively read `*.xls` files, so I have added a section here on importing this type of file. Most of R users circumvent this problem by saving an Excel worksheet as a `*.csv` or tab-delimited file. It is quite easy, however, to import the raw `*.xls` file into R if you have ODBC enabled on your windows machine. Most windows machines do. If yours does not and you have administrator privileges you can add ODBC functionality by going to Control Panel->Add Remove Programs->Add Remove Windows Components. There are also ODBC extensions available for Mac and *nix if you use those platforms.

Getting ODBC up and running on Windows is simply a matter of activating it as a windows component. On Macs, however, you have to load the drivers for a particular data source (e.g. Excel, Access). Unfortunately, these drivers are not free. You can download them for a small fee ($29.95 at the time of this writing) from Actual Technologies. The nice thing about using ODBC is that users do not have to actually install the database software to read the database. So, assuming you have the drivers, it doesn't matter that MS Access isn't available for the Mac, you could still query an Access database if you had the ODBC driver.

You must have the RODBC package installed in R to import data via ODBC. Once installed the code to import an Excel file is this:

```
> library(RODBC)
> channel1 <- odbcConnectExcel("myexcelfilename.xls")
> dataset <- sqlFetch(channel1, "worksheet name")
> odbcCloseAll()
```

The command `odbcConnectExcel` opens up a connection to a Excel workbook. You can view the list of tables available on the connection using the `sqlTables` command. The `sqlFetch` command will read in a table from the open connection as a data frame. When you are done reading in tables, you can close the open connection using the `odbcCloseAll()` command.

Unfortunately, writing to Excel is a little more difficult than reading. R doesn't have the tabular formatting capabilities that Excel has. A work around for this problem is to use the ODBC connection in Excel in addition to using them in R. R will export to `*.csv` just fine. Excel can connect to `*.csv` data via ODBC. Using ODBC rather than just importing the `*.csv` means that as R updates the `*.csv` it will automatically be updated in the Excel workbook. To generate

formatted tables for publication you can use cell references that point from the `*.csv` file into a well-formatted table.

The above example is just one of the many types of connections that ODBC offers. Using ODBC you can connect to virtually any data storage format including all common database (e.g. Access, Oracle, SQL Server, etc.). The database formats have good write capabilities, unlike Excel, so you can send data to them directly from R. Also, the `sqlFetch` command allows for SQL statements to be executed within the command. So, you can create a database query in R to pull piece of data from a larger database that you are interested in.

Session 4

Formatting and Output

4.1 Well-formed data frames

Learning how to build a well-formed data frame is perhaps the most important skill to develop in R. Fortunately, there is a simple test in Excel to determine if your data frame is well-formed. If you can generate a pivot table for any combination of variables in your data set, then you will have a well-formed data frame in R.

This means that no columns headings should form a meaningful sequence. For instance, if you have nested samples, you should not have a response column for each nested subplot. Instead, you should have one column for the response variable and one column for the scale of observation. The same applies to multiple columns that might form a sequence of dates.

Frequently, the optimal format for recording data is not a well-formed data frame. Well-formed data frames tend to have few columns and many rows with information in some columns repeated frequently. If you can easily change your data into a data frame in Excel, do so. Most of the time, however, it will be easier to import the data "as-is" into R and change its format in R. R will allow automation of the process so that if your data ever changes (you should always assume that it will) you can perform the necessary formatting again without trying to remember how you did it the first time.

We'll go through a small example now of generating a poor data frame and converting it to a good data frame. Let's assume that we're measuring the height of plants in a greenhouse over 3 successive months. We generate a fake dataset such as this:

```
> dat <- data.frame(plantId = 1:5, Jan01 = rnorm(5, 10, 1), Feb01 = rnorm(5,
+     20, 2), Mar01 = rnorm(5, 30, 3))
> dat
```

```
  plantId     Jan01    Feb01    Mar01
1       1  9.190147 22.50304 25.55897
2       2 10.360700 21.92445 29.15972
3       3  9.697200 20.90764 33.64665
4       4 10.960069 23.06336 33.62743
5       5 10.558849 17.51010 22.73288
```

This is a poorly formed data frame because it is difficult to analyze the change in height through time in this form. Specifically, there is no way to perform a regression of height on time using the

standard formula format `y~x`, because the explanatory variables must be single columns in the dataset. We need date to be it's own column and height to be its own column. I show here two different ways to accomplish this task. The first I have designated as the brute force method. This is shown here mainly to give you understanding what a well-formed dataframe looks like, and to give you practice working with various vector manipulation functions such as `unlist` and `rep`. The second method is more elegant, and takes advantage of some higher level manipulation commands. In practice you will St the second method most often, but it is useful to know the brute force method, just in case you data structures are more complex.

4.1.1 The Brute Force Method

Let's focus on making the height column first. We want to turn the columns of height into a single vector. There are two commands to do this in R and it is not necessarily intuitive which one to use, so you have to experiment. The two options are `unlist()` and `c()`. `unlist()` is meant to do exactly what it sounds like, take a list and remove the element distinctions, turning the list into one big vector. In the case of this example `unlist()` works.

```
> unlist(dat[, 2:4])

   Jan011    Jan012    Jan013    Jan014    Jan015    Feb011    Feb012    Feb013
 9.190147 10.360700  9.697200 10.960069 10.558849 22.503036 21.924448 20.907641
   Feb014    Feb015    Mar011    Mar012    Mar013    Mar014    Mar015
23.063358 17.510099 25.558973 29.159722 33.646649 33.627426 22.732877
```

You will notice that `unlist` turned the three height columns of the data frame into a single vector with a `names` attribute which is just the column name of the data frame with a sequential index appended. A *very* important thing to note is that the columns have been "unwound" by row. The `unlist` command always iterates across rows first and then columns. This will be important when we create our date column.

The command `c()` does not work for these data:

```
> c(dat[, 2:4])

$Jan01
[1]  9.190147 10.360700  9.697200 10.960069 10.558849

$Feb01
[1] 22.50304 21.92445 20.90764 23.06336 17.51010

$Mar01
[1] 25.55897 29.15972 33.64665 33.62743 22.73288
```

Instead, `c` turns the three columns of the data frame into a list with three named elements and a vector of length 5 in each element. However, if the data had been in a matrix form, `c` would have been the correct command, and `unlist` would have done nothing:

```
> unlist(as.matrix(dat[, 2:4]))
```

```
         Jan01    Feb01    Mar01
[1,]  9.190147 22.50304 25.55897
[2,] 10.360700 21.92445 29.15972
[3,]  9.697200 20.90764 33.64665
[4,] 10.960069 23.06336 33.62743
[5,] 10.558849 17.51010 22.73288
```

```
> c(as.matrix(dat[, 2:4]))
```

```
 [1]  9.190147 10.360700  9.697200 10.960069 10.558849 22.503036 21.924448
 [8] 20.907641 23.063358 17.510099 25.558973 29.159722 33.646649 33.627426
[15] 22.732877
```

As a rule of thumb, data frames use `unlist` and matrices use `c`, but it is best to test each one out on a subset of your data first.

We can now store the unlisted vector of heights, and move toward creating a date column that matches correctly with this vector:

```
> height <- unlist(dat[, 2:4])
> height
```

```
   Jan011    Jan012    Jan013    Jan014    Jan015    Feb011    Feb012    Feb013
 9.190147 10.360700  9.697200 10.960069 10.558849 22.503036 21.924448 20.907641
   Feb014    Feb015    Mar011    Mar012    Mar013    Mar014    Mar015
23.063358 17.510099 25.558973 29.159722 33.646649 33.627426 22.732877
```

Since `height` iterates first across rows and second across columns, our date vector will need to repeat the date of the first column for each row in the original data frame. Next, the date vector will need to repeat the date of the second column for each row, and so on. To do this we use the `rep()` command. We can use the `rep` function to generate a vector in which *each* date is repeated 5 times:

```
> date.vec <- factor(names(dat)[2:4], ordered = T)
> date.vec
```

```
[1] Jan01 Feb01 Mar01
Levels: Feb01 < Jan01 < Mar01
```

```
> date <- rep(date.vec, each = nrow(dat))
> date
```

```
 [1] Jan01 Jan01 Jan01 Jan01 Jan01 Feb01 Feb01 Feb01 Feb01 Feb01 Mar01 Mar01
[13] Mar01 Mar01 Mar01
Levels: Feb01 < Jan01 < Mar01
```

We now have vectors `height` and `date` which are of equal length and we could `cbind` them together to make a new data frame, but we are still missing something—the plant id. Unlike date, `plantId` iterates across the rows rather than across the columns of the original data. So, we can create a vector of plant ids using the same technique we used to create the date vectors but using the `times` argument instead of the `each` argument.

```
> plantId <- rep(dat$plantId, times = 3)
> plantId

 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

We are now ready to create our new data frame that is well-formed:

```
> dat.good <- cbind.data.frame(plantId, date, height)
> dat.good

       plantId  date    height
Jan011       1 Jan01  9.190147
Jan012       2 Jan01 10.360700
Jan013       3 Jan01  9.697200
Jan014       4 Jan01 10.960069
Jan015       5 Jan01 10.558849
Feb011       1 Feb01 22.503036
Feb012       2 Feb01 21.924448
Feb013       3 Feb01 20.907641
Feb014       4 Feb01 23.063358
Feb015       5 Feb01 17.510099
Mar011       1 Mar01 25.558973
Mar012       2 Mar01 29.159722
Mar013       3 Mar01 33.646649
Mar014       4 Mar01 33.627426
Mar015       5 Mar01 22.732877
```

You will notice that the `row.names` attribute is defined based on the `heights` vector. This is sort of annoying, but we can fix it easily enough.

```
> dat.good <- data.frame(dat.good, row.names = NULL)
> dat.good

       plantId  date    height
Jan011       1 Jan01  9.190147
Jan012       2 Jan01 10.360700
Jan013       3 Jan01  9.697200
Jan014       4 Jan01 10.960069
Jan015       5 Jan01 10.558849
Feb011       1 Feb01 22.503036
Feb012       2 Feb01 21.924448
Feb013       3 Feb01 20.907641
Feb014       4 Feb01 23.063358
Feb015       5 Feb01 17.510099
Mar011       1 Mar01 25.558973
Mar012       2 Mar01 29.159722
Mar013       3 Mar01 33.646649
Mar014       4 Mar01 33.627426
Mar015       5 Mar01 22.732877
```

It is a pretty long process to create a well-formed data frame, but the effort up front is worth it. The final step is to automate what we've just done by putting everything into a function.

```
> convertDat <- function(d = dat) {
+     height <- unlist(d[, 2:4])
+     date.vec <- factor(c("Jan01", "Feb01", "Mar01"), ordered = T)
+     date <- rep(date.vec, each = nrow(d))
+     plantId <- rep(d$plantId, times = 3)
+     out <- data.frame(plantId, date, height)
+     row.names(out) <- NULL
+     return(out)
+ }
```

In summary, to create a well-formed data frame any columns that form meaningful sequences should be split into a response column and an iterating explanatory column. To build new variables that iterated across rows in the original dataset, use the `times` argument, for variables that repeated across columns, use the `each` argument. Put all the commands into one function. Then, when your data changes or you get a new dataset, you can simply execute the function.

4.1.2 The Elegant Method

We wish to "unwind" the height columns, to generate a single column that is height and a single column that is the date of the measurement. This is easily accomplished using the command `stack`. `stack` concatenates multiple vectors into a single vector and a factor which is the column name where each element of the vector originated. In the example above we want to stack the columns of heights across dates

```
> hgt <- stack(dat, select = -plantId)
> hgt

      values   ind
1   9.190147 Jan01
2  10.360700 Jan01
3   9.697200 Jan01
4  10.960069 Jan01
5  10.558849 Jan01
6  22.503036 Feb01
7  21.924448 Feb01
8  20.907641 Feb01
9  23.063358 Feb01
10 17.510099 Feb01
11 25.558973 Mar01
12 29.159722 Mar01
13 33.646649 Mar01
14 33.627426 Mar01
15 22.732877 Mar01
```

We now have two vectors that are the correct length for the final dataframe, but we need to duplicate the values in `plantId`. We do this by replicating the indices of the rows for the dataframe. This can be done as in the 4.1.1 via `rep`.

```
> id <- rep(1:nrow(dat), length = nrow(hgt))
```

Finally, we put it all together to create a new dataframe.

```
> dat2 <- cbind.data.frame(dat[id, 1], hgt)
```

As stated earlier, most data that you will work with can be manipulated using this method. There will be occasions, however, that you will have to use the brute force method.

4.2 Data output in Tables and Figures

Outputting data from R is much easier than getting data into R. That said, because R was developed for Unix systems, getting figures and tables that are manuscript-ready out of R can take some doing for Windows users or for journals that do not accept Postscript figures.

4.2.1 Tabular output

There are analogous `write.` functions for the `read.table` function at its derivatives (e.g. `write.csv`). The output produced by these functions looks just like the data that is read into R with the exception that row names are exported by default for all dataframes.

```
> df <- data.frame(resp = rnorm(10), trt = sample(c("trt1", "trt2",
+     "trt3"), 10, replace = T))
> write.csv(df, "df.csv")
```

"df.csv" looks like this:

```
"","resp","trt"
"1",0.0487173946817355,"trt1"
"2",-1.09039323012717,"trt1"
"3",1.15214344621636,"trt2"
"4",-0.0237947788086044,"trt1"
"5",-1.11909058034539,"trt3"
"6",1.02685427714149,"trt3"
"7",0.0226001537783687,"trt1"
"8",2.12493601241390,"trt1"
"9",1.73754976261647,"trt2"
"10",-2.18835432067972,"trt3"
```

We have already discussed how to generate well-formed dataframes for R. Dataframes, however, are not very compact data storage. They maximize the number of rows, minimize the number of columns, and often have lots of repeated data values. We can convert this *long* data format into a *compact* data format using the `unstack` command in a similar way to which we used the `stack`

command. `unstack` takes a formula as an argument. We have not discussed formulas yet, but in a nutshell they are specified as `response~predictor1 + predictor2 + predictor3...`. In the case of `unstack` the response variable is the column which should be compacted and the predictors are variables whose levels should be turned into column headings.

The statements below compact the `dat.good` dataframe that was created earlier using the elegant method:

```
> us <- unstack(dat.good, height ~ date)
> cbind.data.frame(plantId = dat.good$plantId[1:nrow(us)], us)
```

```
  plantId    Feb01     Jan01    Mar01
1       1 22.50304  9.190147 25.55897
2       2 21.92445 10.360700 29.15972
3       3 20.90764  9.697200 33.64665
4       4 23.06336 10.960069 33.62743
5       5 17.51010 10.558849 22.73288
```

4.3 Xtables

Content missing...will be repaired by 20080418.

```
> library(xtable)
> data(tli)
> fm1 <- aov(tlimth ~ sex + ethnicty + grade + disadvg, data = tli)
> fm1.table <- xtable(fm1)
> print(fm1.table)
```

```
% latex table generated in R 2.7.1 by xtable 1.5-2 package
% Thu Jul 24 19:04:02 2008
\begin{table}[ht]
\begin{center}
\begin{tabular}{lrrrrr}
  \hline
 & Df & Sum Sq & Mean Sq & F value & Pr($>$F) \\
  \hline
sex & 1 & 75.37 & 75.37 & 0.38 & 0.5417 \\
  ethnicty & 3 & 2572.15 & 857.38 & 4.27 & 0.0072 \\
  grade & 1 & 36.31 & 36.31 & 0.18 & 0.6717 \\
  disadvg & 1 & 59.30 & 59.30 & 0.30 & 0.5882 \\
  Residuals & 93 & 18682.87 & 200.89 &  &  \\
   \hline
\end{tabular}
\end{center}
\end{table}
```

The output from print(tli.table) is written in LaTeXmarkup language and can be pasted directly into a LaTeXdocument as I have done below.

`xtables` can also make pretty tables for web pages

```
> print(fm1.table, type = "html")

<!-- html table generated in R 2.7.1 by xtable 1.5-2 package -->
<!-- Thu Jul 24 19:04:02 2008 -->
<TABLE border=1>
<TR> <TH>  </TH> <TH> Df </TH> <TH> Sum Sq </TH> <TH> Mean Sq </TH> <TH> F value </TH> <TH> Pr(
  <TR> <TD> sex </TD> <TD align="right"> 1 </TD> <TD align="right"> 75.37 </TD> <TD align="righ
  <TR> <TD> ethnicty </TD> <TD align="right"> 3 </TD> <TD align="right"> 2572.15 </TD> <TD alig
  <TR> <TD> grade </TD> <TD align="right"> 1 </TD> <TD align="right"> 36.31 </TD> <TD align="ri
  <TR> <TD> disadvg </TD> <TD align="right"> 1 </TD> <TD align="right"> 59.30 </TD> <TD align="
  <TR> <TD> Residuals </TD> <TD align="right"> 93 </TD> <TD align="right"> 18682.87 </TD> <TD a
   </TABLE>
```

4.3.1 Figure Output

Figures can be saved in multiple formats directly from the GUI graphic window. It is also common to create a figure output directly to a file from the command line. You can do this by calling a new `device` of the figure format that you want. All figure output formats are available in the `grDevices` library. These include: `pdf`,`postscript`,`bmp`,`jpeg`, and `png`. Once these devices are started, you can call plotting functions as normal. Instead of being output to the screen, however, the plots are created in a file of the chosen format. To complete the plot, and close the file use, turn off the device using the command `dev.off`.

```
> library(grDevices)
> x <- seq(1, 20, length = 50)
> err <- rnorm(50)
> y <- 2 * x + 3 + err
> pdf("fig1.pdf", width = 7.5, height = 7.5, paper = "letter")
> plot(x, y)
> abline(3, 2, col = "red")
> dev.off()

X11
  2
```

4.4 Encapsulating Input and Output

Importing and Exporting data from R typically takes many lines of code. It is best to encapsulate data input and output into their own functions as part of a `*.r` script. Within the `input` function ODBC channels are opened, data are imported, and then reshaped into forms appropriate for analysis. Newly created data can be assigned to the global environment using the `assign` command with the argument `envir=.GlobalEnv`. The `output` function calls all the plotting and a table generating commands needed to export the figures and tables in formats fit for publication. This keeps final data output separate from generic plotting or printing to the screen. So, you can work on your output within the R native interfaces, but then easily export the results into manuscript formats. Here I provide a small example using some of the datasets we created above.

io.r

```
### Includes
library(RODBC)
library(grDevices)

### My importing function
import<-function(){
    ## import example.xlsx
    ch1<-odbcConnectExcel("example.xlsx")
    ex<-sqlFetch(ch1,"Sheet1")
    assign(ex,"ex",envir=.GlobalEnv)
    odbcCloseAll()

    ## import dat.csv
    dat<-read.csv("dat.csv")
    hgt<-stack(dat,select = -plantId)
    id<-rep(1:nrow(dat),length=nrow(hgt))
    dat<-cbind.data.frame(dat[id,1],hgt)
    assign(dat,"dat",envir=.GlobalEnv)

    return(invisible())
}

analyze.example<-function(x=example){aov(resp~trt,data=x)}

plot.dat<-function(x=dat){boxplot(hgt~date,data=x)}

output<-function(){
    ## output example
    mod<-analyze.example()
    mod.table<-xtable(mod)
    writeLines(print(mod.table))

    ## output dat
    pdf("fig1",width=7.5,height=7.5,paper="letter")
    plot.dat()
    dev.off()
}
```

4.5 Exercises

The following is a time series of weights for 3 indiviudals chosen from 2 populations.

Individual	Pop1Time1	Pop
A	25	34
B	33	12
C	16	17

1. Write this table as a .csv
2. Read the *.csv into R
3. Turn the data into a well-formed dataframe.
4. Calculate population means and variance by population and time
5. Output you results to a file
6. Extra Credit: Enclose all the input and output into two function

Day II

July 26

Session 5

Scripting and Functions

R scripts are simply text files that contain R commands. There are many different ways to build scripts in R and there are many different ways to interact with scripts in R. The best advice is to try as many different ways as you can and select find one that fits for you.

5.1 Text Editors

5.1.1 MS Word

Any text editor can create R script files. Many R users simply use Notepad. WYSIWYG text editors like Microsoft Word can also be use to store R commands. These documents allow for extensive documentation of the code, or for embedding figures by copying and pasting from an R graphics device. Since these files are not true text files, though, commands stored in MS Word format cannot use the `source` command to read an entire file. Instead, commands must always be copied and pasted into the R command line.

5.1.2 The R GUI

The R GUI also provides a limited text editor for creating script files. The Mac version of this text editor is far superior to the Windows version, and notably includes syntax highlighting and automatic indentation. Syntax highlighting means that R commands, comments, and strings, are colored differently than other text. Most programmer use text editors which have syntax highlighting because it makes the code much easier to read.

Automatic indentation is useful when writing functions. Code is more readable when nested blocks are indented. For example:

```
> tmp <- function() {
+     print(1:10)
+     x <- NULL
+     y <- NULL
+     for (i in 1:10) {
+         x <- i + 1
+         for (j in 1:20) {
+             y <- x + j
```

```
+         }
+     }
+ }
```

is less readable than

```
> tmp <- function() {
+     print(1:10)
+     x <- NULL
+     y <- NULL
+     for (i in 1:10) {
+         x <- i + 1
+         for (j in 1:20) {
+             y <- x + j
+         }
+     }
+ }
```

Editors in both operating systems allow users to execute code for a particular selection or source the entire document directly into the R command line. It is important to remember that code sourced in this way is seen on the R standard out screen, but is not report in the `.RHistory` file 5.2. This contrasts with code which is copied and pasted onto the command line, which does become part of the `.RHistory`. While writing the `.RHistory` file may be seen as an advantage, it isn't really because your `.RHistory` history can become clogged with incorrect code while you debug a script.

Neither editor allows anything but raw text in the file. So, plots and tables cannot be embedded.

5.1.3 TextPad

Since the editor provided by R is lacking syntax highlighting, the shareware TextPad (`http://www.textpad.com`) is a good alternative for Windows users. It provides both syntax highlighting and automatic indentation at the expense of only being able to source the file using the `source` command or copying and pasting commands. TextPad is very good text editor for programming in general, and provides styles for virtually any programming language.

To set up TextPad to work with R:

1. Install TextPad
2. Download the R syntax definition (`http://www.textpad.com/add-ons/files/syntax/r.zip`)
3. Unzip `r.zip` into the `Samples` the TextPad installation folder, typically, `C:\Program Files\Text Pad 5\Sam`
4. Start TextPad
5. From the menu: `Configure > New Document Class...`
6. Document Class Name: `R`

7. Class Members: `*.rhistory, *.r`
8. Check Enable Syntax Highlighting
9. Syntax definition file: `r.syn`
10. Finish
11. From the menu: `Configure > Preferences > Document Classes > R > Tabulation`
12. Default Spacing: 2 or 4
13. Indent Size: 2 or 4
14. Check Convert new tabs to spaces
15. Check Convert existing tabs to spaces when saving files

Once you've finished creating the new document class. Files with the extension `*.r` or `*.RHistory` will automatically be associated with the R class, and the proper highlighting and tabulation will be invoked. If you save your files with a different extension (e.g. `.txt`) then you will have to set the properties by hand from `View > Document Properties` in a similar manner to the sequence for creating the document class.

5.1.4 Emacs and ESS

Emacs is a venerable text editor developed for Unix environments. It has so many features, some have dubbed it a full fledged operating system. It is the *de facto* editor of R power users for generating scripts and packages, and for even running R. There are ports of emacs for Windows (`http://www.gnu.org/software/emacs/windows/`) and Mac (`http://aquamacs.org/`, though it is definitely more difficult to get the advanced features of emacs (like compiling R packages) working in the Windows port.

The learning curve for Emacs is steep, and we'll not take time here to go into how to use Emacs with its R interface package, Emacs Speaks Statistics (ESS). I will provide an overview, however, of all the things that Emacs can do with R:

- All of the standard features of a good text editor including syntax highlighting and auto-indentation
- Sourcing of individual lines, functions, code blocks, or files
- Running R directly within the editor so that the output of R can be edited directly.
- Extensively documented code using the `Rweave` commands and LaTeX. This allows users to generate a fully formatted document (even a scholarly article) with R code embedded in the text. R code is run within the text and the output (even figures and tables) are created directly in the document.
- Generate R packages which easily incorporate `C` or `Fortran` Code.
- Generate R documentation files.

If you plan on using `R` for all your future data analysis, learning Emacs is well worth the effort.

5.2 Executing Scripts in R

Given that your chosen text editor will allow it there are 5 ways to execute code stored in a file on the R command line:

1. by line
2. by function
3. by selection
4. by file and
5. copy and paste

. Execution by line will send the line on which your cursor currently resides to R and move the cursor to the next line. By function will source the entire function surrounding the cursor. By selection will send a highlighted region to R. Finally sourcing the entire file (either through a shortcut-key, a pressed button, or via the `source` command) reads every line into R just as though you had typed it directly. Finally, you can copy an paste sections of code directly onto the command line.

When code is sourced by copy and paste, the pasted code will appear in the `.RHistory`. Code sourced by any other method will not appear in the `.RHistory` file. While writing to the `.RHistory` file may be seen as an advantage, it typically is not because the `.RHistory` history can become clogged with incorrect code while you debug a script. For all methods except for `source` command the sourced code will appear in the R standard output.

Sourcing files directly by using the `source` command is typically the fastest way to debug scripts. Doing so allows you to re-run scripts with a minimum of keystrokes. To make sourcing the file efficient using the `source` command, however, most of the code in the script should be stored as functions. An example script might be:

```
### myfile.r
### Created: 2008.06.05

### Import my data
import<-function(){
  ## import code goes here....
}

analyze<-function(){
  ## analysis code goes here...
}

### This is the function to debug
makeNewData<-function(){
  ## do some stuff
}
```

To debug the function `makeNewData`. I would type the following on the R command line:

```
> source("myfile.r")
> tmp <- makeNewData
```

If an error results from the execution of this line of code, switch to the text editor (`Alt-Tab`). The offending statements in the text file would be corrected and saved (`Ctrl-s`). Switch back to R (`Alt-Tab`) and the exact procedure can be re-run on the R command line via `up-arrow, Enter`. This keystroke sequence becomes second nature once a few files are written and debugged.

5.3 Writing functions in R

Writing function is the most daunting task that new R users face. Many R users never climb this mountain and their scripts are merely a hodge-podge of commands. They must remember how to execute those commands in the correct order so the output is predictable and repeatable. They can not apply the code they've written do other projects without copying and pasting their code over and over. Finally, they end up with thousands of lines of code, where there may have only been a dozen if they had written a function and executed that function repeatedly.

Writing functions alleviates all of these problems, but many R users do not know where to begin. They can see how individual commands work together, but encapsulating their behavior into a more general function is difficult. Below I outline a method that I have found particularly useful in going from commands typed on the screen to scripts with functions. To illustrate the process, we'll use an example problem of doing a permutation test on a series of weights from 2 populations. We want to test whether populations differ significantly in weight. Let's assume that when fieldwork was started animals were sorted into pairs and the data look like this (as a `.csv`):

```
pop1, pop2
23,45
24,41
...
```

There are few things that we need to do with this data. First, it is not a well-formed dataframe. Species is an attribute and it would be better if species were a column and weight was a column. Second, we need to take the mean weight of each species. Third, we must randomly assign species to weights many times and calculate the mean. Finally, we need to compare the difference in means of the populations against the difference in means for the permuted samples. In the absence of real data we can simulate the dataset above (See 6.2.1). Let's assume that species weights are normally distributed with different means and standard deviations.

```
> popWeights <- data.frame(pop1 = rnorm(10000, 500, 20), pop2 = rnorm(10000,
+     400, 10))
> str(popWeights)

'data.frame':        10000 obs. of  2 variables:
 $ pop1: num  514 522 514 485 451 ...
 $ pop2: num  404 397 398 399 389 ...
```

This dataset consists of 10,000 observations. for each population.

5.3.1 Building scripts from history files

The early stages of any analysis in R are filled with merely trying to get statements to execute on the command line and making sure that the output is correct. If you have a large dataset it can be useful to do these preliminary steps on a subset of the data. An easy way to do this is to write a function that randomly samples rows from a data frame such as this:

```
> sampleDf <- function(df, ...) {
+     rw <- sample(1:nrow(df), ...)
+     return(df[rw, ])
+ }
```

Let's assume that my initial analysis steps on the command line look something like this:

```
> sampleDf(popWeights, 5)

         pop1     pop2
6236 471.1347 401.0590
9248 540.3618 402.4048
9790 493.9149 398.2597
6819 522.7567 400.7953
857  532.1074 402.8629

> sampleDf(popWeights, 5)

         pop1     pop2
3789 479.1615 389.6545
7770 527.7397 401.4634
5124 500.5769 390.5791
2154 520.6714 405.0478
7772 504.1651 389.8230

> smpWeights <- sampleDf(popWeights, 5)
> names(smpWeights)

[1] "pop1" "pop2"

> unlist(smpWeights)

   pop11    pop12    pop13    pop14    pop15    pop21    pop22    pop23
509.4116 526.1957 504.3259 499.7033 487.7454 401.2340 411.0782 398.6482
   pop24    pop25
410.1229 406.4585

> tmpWeights <- unlist(tmpWeights)
> tmpWeights <- unlist(smpWeights)
> rep(names(smpWeights), length = length(tmpWeights))
```

```
 [1] "pop1" "pop2" "pop1" "pop2" "pop1" "pop2" "pop1" "pop2" "pop1" "pop2"

> tmpNames <- rep(names(smpWeights), length = length(tmpWeights))
> smpWeightsDf <- cbind(pop = tmpNames, weight = tmpWeights)
> smpWeightsDf

      pop    weight
pop11 "pop1" "509.411617887864"
pop12 "pop2" "526.195683458395"
pop13 "pop1" "504.325936102086"
pop14 "pop2" "499.703315896955"
pop15 "pop1" "487.745439763608"
pop21 "pop2" "401.23398264769"
pop22 "pop1" "411.078170446368"
pop23 "pop2" "398.648161644099"
pop24 "pop1" "410.122880759933"
pop25 "pop2" "406.458525575216"
```

This series of commands simply goes through the process of creating a well-formed sample dataframe with 5 observations for each population.

Once you have the preliminary analysis steps executing correctly, you can begin to join your statements together using the history file. You can save a history file on the fly using `savehistory`. Save the history of the R session as a new file, and open this file in a text editor.

```
> savehistory("FormDataFrame.r")
```

The history file looks like this:

```
sampleDf(popWeights,5)
sampleDf(popWeights,5)
smpWeights<-sampleDf(popWeights,5)
names(smpWeights)
unlist(smpWeights)
tmpWeights<-unlist(tmpWeights)
tmpWeights<-unlist(smpWeights)
rep(names(smpWeights),length=length(tmpWeights))
tmpNames<-rep(names(smpWeights),length=length(tmpWeights))
smpWeightsDf<-cbind(pop=tmpNames,weight=tmpWeights)
smpWeightsDf
```

Next, go through each statement in the history file and delete statements that returned an error when you execute them or are superfluous in some way. Once you have completed this process, you can test your remaining code by opening a new R session with an empty workspace (or a workspace that only contains your data subset) and sourcing this code. If your output is the same as what you created on the command line, then you can have reasonable confidence that your new script is working correctly. For the example, the pruned history file looks like this:

```
smpWeights<-sampleDf(popWeights,5)
tmpWeights<-unlist(smpWeights)
tmpNames<-rep(names(smpWeights),length=length(tmpWeights))
smpWeightsDf<-cbind(pop=tmpNames,weight=tmpWeights)
```

The next step, if you had been working on a data subset is to do a search and replace on the script in the text editor replacing the symbol of the data subset with the symbol of the full dataset. For the example the new file would look like:

```
tmpWeights<-unlist(popWeights)
tmpNames<-rep(names(popWeights),length=length(tmpWeights))
weightsDf<-cbind(pop=tmpNames,weight=tmpWeights)
```

There is an easier way to do than simply searching and replacing, however, by encapsulating the entire script into a function.

5.3.2 Building functions from scripts

Building functions from scripts is as simple as placing brackets around the entire script and assigning it to a symbol. So, for instance

```
cleanData<-function(){
  tmpWeights<-unlist(popWeights)
  tmpNames<-rep(names(popWeights),length=length(tmpWeights))
  weightsDf<-cbind(pop=tmpNames,weight=tmpWeights)
}
```

Since `weightsDf` is assigned in the last line of the function, this function will return the value of `weightsDf`. We could make this more explicit by adding the statement: `return(weightsDf)`. Remembering the discussion on Scope (See **??**), we realize that `tmpWeights`, `tmpNames` and `weightsDf` are all scoped inside the function and will not be assigned permanently to the workspace. So, we would have to execute the following statements on the command line to assign `weightsDf` to the global workspace.

```
> source("FormDataFrame.r")
> weightsDf <- cleanData()
```

Right now, `cleanData` does not accept any arguments. It might be useful, however, to pass the original data `popWeights` as an argument to the function. Say, for instance, that a second dataset similar to `popWeights` is collected. If the function accepts an argument that has the same form as `popWeights`, we can clean the second dataset immediately. To minimize confusion I'll replace `popWeights` with `df` in the function. Adding a parameter is a simple matter of placing the symbol in the function definition. We can even assign a default value to the parameter so that it cleans `popWeights` when no arguments are passed.

```
cleanData<-function(df=popWeights){
  tmpWeights<-unlist(df)
```

```
  tmpNames<-rep(names(df,length=length(tmpWeights)))
  weightsDf<-cbind(pop=tmpNames,weight=tmpWeights)
  return(weightsDf)
}
```

The steps for taking a workspace session from the command line to well formed scripts and functions can be summarized as follows:

1. Save history as script file
2. Clean history of errors and unnecessary statements
3. Text Script
4. Enclose script with functions
5. Give the function a return value
6. Convert input data to parameters

While encapsulating cleaned history file in function may seem tedious and a bit unnecessary, as you analyses become more complex this procedure is very helpful. Section gives some additional reasons why writing in function is a good thing, in general.

Session 6

Loops and Scripting Best Practices

6.1 Looping through code

Often, there is reason to do calculations more than once. Perhaps you have multiple datasets of similar form that you must calculate statistics for, or you have to repeatedly sample from distributions. R has multiple built-in methods for calling functions multiple times. These methods vary in the speed. The slowest method, `for` loops, is quite transparent and the logic of `loops` is easy to follow. Faster methods such as `apply` functions are not as easy to read, but make your often make your code more succinct.

6.1.1 vectorization

All fundamental types in R are built upon the basic list and vector classes. It is relatively easy to perform simple mathematical and statistical operations for each value in an entire list or vector. We have already discussed this in 1.2.1. All elementary mathematical operations can be performed on vectors simply by treating the vector as an individual unit:

```
> h <- 1:5
> h + 2

[1] 3 4 5 6 7

> 2^h

[1]  2  4  8 16 32
```

Because they are built upon vector and list classes, matrices, arrays and dataframes can accept similar operators.

```
> k <- data.frame(1:5, rnorm(5))
> k * 2

  X1.5   rnorm.5.
1    2 -0.7847659
2    4 -0.5127008
```

```
3    6  0.5104317
4    8 -3.8656292
5   10  0.8054540
```

In addition to elementary operators, many other functions in R accept vectors as arguments. When sampling from a distribution, for example, you can pass a vector of means, and the function will sample successively from each mean, up to the defined sampled size.

```
> rnorm(50, mean = c(-100, 100))

 [1] -100.34072   99.17127 -100.08154  100.20972  -99.53559   98.81432
 [7] -100.20773  100.95001  -99.50621  100.42025 -100.99467  100.03054
[13] -100.61734   99.77686  -99.64446   99.24278  -99.67107  100.34850
[19]  -99.94083  100.37482  -98.68266  100.81757  -99.61577   99.46580
[25] -100.60288  100.98679 -100.25290  100.63571 -100.59128  101.09107
[31]  -98.53812  100.85844  -99.10016   99.40329 -101.02080  100.81769
[37]  -98.96822  101.87037 -100.74265   99.18763 -101.21869   99.40196
[43] -100.80644   99.47621  -99.68403   99.17581  -99.75691  100.59428
[49] -101.68857  100.66914
```

Performing operations on entire vectors, or their derived classes is the fastest method for performing multiple operations at once. The vector and matrix structure, in fact, one of the great benefits of R as a programming language for mathematics. However, the functions that support vector operations and those which do not are haphazard. So, for instance you can pass a vector of means to the `rnorm` function and the sample will be generated from all the means. But, if you pass a vector of sample sizes, the first argument, `rnorm` generates a sample of the same length as the vector of sample sizes, not multiple samples of the sizes assigned in the vector

```
> rnorm(c(5, 10, 20))

[1] -0.06516272 -0.23402256  1.35652543
```

Functions supporting vectorization are not well documented. You can look in the help and sometimes the particular parameter you can to vectorize will be defined specifically for vector, but often the people who developed the package just forgot to document this support. So, the best practice is to experiment with vectorization of operations before moving onto other methods such as `for` loops or `apply` functions. Just try it out on a subset of the data, and if it works, then great. If not, then you must move on to more pragmatically iintensive operations.

6.1.2 for loops

The `for` loop is the easiest method to understand, conceptually. Any calculations that you want to perform multiple times simply go between braces `{}` as part of a `for` command. The number of times the calculation is performed is set by an index vector which is passed to the for loop as an argument, and whose value at each iteration is passed to the loop. It is easiest to see by example.

```
> out <- vector(length = 10)
> for (i in 1:10) {
+     out[i] <- 2 * i
+ }
> out

 [1]  2  4  6  8 10 12 14 16 18 20
```

There are three major parts to a for loop: initialization, the call, and the procedure. Since the output is to be built sequentially, you must create an empty container that can hold each output in the `for` loop. In the example above, the initialization portion of the loop creates an empty vector named `out` whose length is 10. At that point, every value in `out` is set to `NA`.

The second part of the `for` loop is the call. This is the actual statement of the `for` command. There are 2 arguments to the `for` function. The second argument is a vector of indices that will be assigned to the symbol of the first argument for each execution of the loop. So, in the example above, each element of the vector `1:10` is assigned to the symbol `i`. The first time through the loop, `i` is equal to `1`. The second time through the loop `i` is equal to 2, and so on.

The third part of a `for` loop is the procedure. These are the statements inside the brackets that will be executed multiple times. Unlike other functions, `for` does not have a return value. So, the last statement before the `}` is treated simply as another statement, not a return value. That is why the output vector, if one exists, must be initialized beforehand and then assigned sequentially in the procedure.

Most often, the index vector will simply be a vector from 1 to the number of times you wish to execute the loop. In order to build the output sequentially, you simply assign the index position on the output. It is important to note, however, that the index vectors, can be a vector of any type. So, the following is a perfectly valid, and useful, `for` loop:

```
> filesToImport <- c("data1", "data2", "data3")
> for (f in filesToImport) {
+     tbl <- read.csv(f)
+     assign(f, tbl, env = .GlobalEnv)
+ }
```

The above code, initializes a vector of files to import. Then, it imports each file in a `for` loop and `assign`s them to the global workspace. Another instance where for loops are very useful is in creating multiple plots (See ???REF).

`for` loops are the slowest way to build up calculations in R. While, they are relatively straightforward to code, they do have some other weaknesses. First and foremost, is that using `for` loops keeps developers from thinking about objects in R as whole entities. Instead, `for` loop are designed to treat each individual value in an object as distinct. This prevents users and developers from thinking correctly about how to treat objects. Second, `for` loops often add unnecessary length to codes and are, in general, more error prone. Finally, `for` loops are in nearly every situation easily circumvented using `apply` function which are discussed below (6.1.3). A general rule might be to avoid `for` loops where possible. If using `for` loops helps you think about a problem better, then go ahead and build the `for` loop. After getting the loop to run it is typically very easy to translate the completed loop into an `apply` function.

6.1.3 apply functions

Writing R scripts using `for` loops is very slow because R is an interpreted rather than a compiled language. Most of the functions that cause R to run fast have a `C` backend. We can take advantage of this speed by replacing `for` loops with `*apply` functions.

There are 5 different types of `apply` functions: `sapply`,`lapply`,`apply`, `tapply`, and `by`. Each apply function works with a different class of data, but their general application is the same. That is, apply functions call a user-defined functions for each element in a sequence of elements. These elements may be single numbers, vectors, lists, or rows or columns of a data frame or matrix. Whatever the elements, `apply` functions perform the exact same procedure on every element of the function. The easiest to begin with is `sapply`. `sapply` is used to execute a particular function in each element of a vector.

```
> v <- 1:10
> sapply(v, function(x) {
+     x * 2
+ })

 [1]  2  4  6  8 10 12 14 16 18 20
```

In the above example, we fed `sapply` two arguments. The first was a vector. This is the object that for which we want a calculation done on each element. The second argument is a function. You'll notice that unlike functions we've written previously, the function definition is not assigned to a symbol. That is standard practice for apply functions, though you could easily have written it in this way:

```
> f <- function(x) {
+     x * 2
+ }
> sapply(v, f)

 [1]  2  4  6  8 10 12 14 16 18 20
```

You'll notice that the function takes one argument `x`. This argument (which should always be the first argument in your function), is where each element of the vector is passed to. So, in this case, `sapply` sends each element of `v` (e.g. 1,2,3,4...) successively to the function as `x`. You can specify additional arguments to the function as well. These additional arguments come after the function definition in the `sapply` call.

```
> sapply(v, function(x, y) {
+     x * y
+ }, 2)

 [1]  2  4  6  8 10 12 14 16 18 20
```

You can also use built-in R functions as the functions to `apply` to dataset.

```
> sapply(v, print)
```

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

All further `apply` functions operate similarly to `sapply`. They only differ in the type of data they accept as input and pass to the function. I go through them in order.

Function	Input	Pass
`lapply`	list	list component
`sapply`	vector	single element
`apply`	matrix,dataframe,or an array	single rows or columns
`tapply`	vector	vector
`by`	dataframe	rows

6.2 4 R's for R Programming

There are few things more frustrating than spending huge amounts of time figuring out what a piece of code is supposed to do. As researchers, we want to spend our time searching for patterns in data, ogling pretty graphs, and writing about our findings. In order to get to those fun parts, however, we have to analyze or data, generate those pretty graphs, and understand how we got there. Today, this process inevitably requires writing code for a statistical package. Unfortunately, it is at this stage that many of us get bogged down. Since most scientists are not computer programmers by training, we end up spending 80% of our time trying to get our code to work right, and 10% of our time interpreting the results. This paper is designed to help you reduce that 80% to 20%, by introducing you to some of the elements of good program design that might be taught to a undergraduate computer science major. None of the suggestions I propose here require advanced programming knowledge. Rather they require self-discipline and self-imposed structure in writing code. 5 minutes spent preparing good code will prevent days of work at a later stage in your research. The code you write needs to be right, repeatable, robust, and readable.

6.2.1 Right

Your code must run without errors and give you the correct answer. The first does not imply the second. For every output you must ask yourself, âĂIJDo I believe this?âĂİ The best way to guarantee correctness is to generate a simulated dataset with a known pattern similar in structure to your observed dataset, and see if your code can extract the pattern. For instance, if you want to do a linear regression you might generate the following dataset:

```
> lm.dataset <- function(slope = 2, intercept = 5, std.dev = 2,
+     sample.size = 1000, x.range = c(0, 10)) {
```

```
+     error <- rnorm(sample.size, 0, std.dev)
+     x <- seq(x.range[1], x.range[2], length.out = sample.size)
+     y <- (slope * x + intercept) + error
+     out <- cbind.data.frame(x, y)
+     return(out)
+ }
```

You could then use this dataset to assess whether your analysis (in this case itâĂŹs simply the command `lm`) extracts the correct slope and intercept.

```
> ds <- lm.dataset()
> ds[1:10, ]

            x        y
1  0.00000000 3.951035
2  0.01001001 5.655147
3  0.02002002 7.033094
4  0.03003003 6.029300
5  0.04004004 3.355816
6  0.05005005 4.936983
7  0.06006006 7.935332
8  0.07007007 7.995864
9  0.08008008 3.585052
10 0.09009009 6.374611

> lm(y ~ x, data = ds)

Call:
lm(formula = y ~ x, data = ds)

Coefficients:
(Intercept)            x
      4.854        2.021
```

6.2.2 Repeatable

You will have to do your analysis more than once. Most often you will spend some time at the beginning of your research project creating the code that extracts patterns from your data. Then, you will go away from it for a while, do some reading or write sections of the paper. After that, you will return to your analysis and try to generate the figures and tables for the paper. You might have collected more data by this time, or forgotten which functions did what in your code. To avoid problems at this late stage in your research, always build your R code in such a way that you can repeat your analysis beginning from your raw data and proceeding to the finished figures in just a few steps. Ideally, this could be one function (e.g. rebuild()) that calls other functions youâĂŹve written. A structure that I have adopted which works well is to have every project contain the following functions:

Function	Description
`rebuild()`	Clean out any old datasets, and rebuild the entire project.
`import()`	Import the raw data into R.
`analyze()`	Perform statistical analysis on the data.
`output()`	Generate tables and figures in a format acceptable for print.

6.2.3 Robust

The best programmers are the laziest. They write good code once, and then use it over and over again. To facilitate this when writing R code, you should always write in functions. Do not write source files that just execute lines of code. Rather source files should contain functions that can be called from the command line. Second, writing robust code that can be reused in different circumstances means learning how to intelligently design functions that use parameters. Take, for example the simulated dataset function from the Right section above. A poorly written version of this code with identical output is:

```
> lm.dataset <- function() {
+     error <- rnorm(1000, 0, 2)
+     x <- seq(0, 10, length.out = 1000)
+     y <- (2 * x + 5) + error
+     out <- cbind.data.frame(x, y)
+     return(out)
+ }
```

This function is only useful for generating a 1000 data-point sample in the x range (0,10) from a normal linear process with slope 2 and intercept 5, and a error standard deviation of 2. It would be difficult to use this code in any other context, but by defining all the âĂIJnumbersâĂİ in the function as parameters, we could generate any type normal linear process we wanted. Third, lazy programmers never write code until they determine whether the code they want already exists. You should frequently reference the list of available R libraries (`http://cran.r-project.org/web/packages/`) to see if thereâĂŹs good code already written for your problem. Then, use that code in a very general way, such that you could perform similar analyses with different data if you need to.

A second, but more advanced characteristic of robust R code is the use of generalized functions. Generalized functions are similar to overloaded functions in object-oriented programming, in that they perform similar tasks but with different classes of data. This helps the code perform as expected for users who may not familiar with your specific analysis (or for you when youâĂŹve forgotten what your code is supposed to do). The plot function is an example of a generalized function. The plot function may be called on a linear model object (the result of the lm command), a dataframe, or any number of objects. R knows which plot function to call based on the class of the object and the presence of a function named plot.class where class is the class of the object. We could write a plot function for the lm.dataset function by the following code:

```
> lm.dataset <- function(slope = 2, intercept = 5, std.dev = 2,
+     sample.size = 1000, x.range = c(0, 10)) {
+     class(out) <- c("lmsim", class(out))
+     return(out)
```

```
+ }
> plot.lmsim <- function(lmsim) {
+     fit <- lm(y ~ x, lmsim)
+     plot(y ~ x, data = lmsim, main = "Simulated linear process")
+     abline(fit$coef[1], fit$coef[2], col = "red")
+     return(invisible())
+ }
```

With the generalized function in place datasets of class âĂIJlmsimâĂİ will plot as we expect.

```
> ds <- lm.sim()
> plot(ds)
```

The generalized functions you will encounter most frequently are plot and print.

6.2.4 Readable

Just as you will always have to redo your analysis, you will always forget what your code does. So, you must be diligent to write code that you can pick up years later and jump right into. Here are a few main characteristics of readable code. First, readable code conforms to the 3 Rs listed above. Second, readable code has good variable names. Dataset names and column names should describe what sort of data they contain (`tree.dbh`). Function names should contain a verb (e.g. `plot.treeDbh()`) The length of variable names should match how often it is used in your code. For instance a main dataset used across many functions should have a long variable name (`smokiesSpOccur`) and a dataset used only a few times in a single function should have a short name (`sp`). Use single letters for index variables (e.g. `for(i in 1:10)`). Third, readable code must be commented. The comment character for R is #. Each comment line must have a comment character. At a bare minimum you should have a description of each function above the function definition. Additionally, you might include a table of parameter descriptions, a description of the output, and the date the function was last updated. Hold to a format similar to that of the R documentation (see `http://cran.r-project.org/doc/manuals/R-exts.html`) for the specification or just look at any function help file). Additionally you should add comments to code blocks within functions whose purpose may not be obvious at a glance. All that said, you should prefer well-named variables and concise code over heavily commented functions. Finally, readable code is short. Each line of a function should be no more than 80 characters long, and the number of lines in any one function should not exceed 50. If your function is longer than that, consider splitting it into 2 functions. Finally, an entire R source file should not contain more than 400 lines of code. If your code is more than 400 lines, consider splitting the major sections of code, like, plotting and analysis into separate files that are sourced by one main file. So, for instance in a file âĂIJproject.râĂİ you would have the following lines:

```
source("plotProject.r")
source("analysisProject.r")
```

It takes patience, and a little bit of effort up front to force your code to conform to the four RâĂŹs of R programming. Time on the front end of your research and analysis, however, will save you time in the long run. Never think to yourself, âĂIJOh, this will just be a short analysis. I donâĂŹt have to be so disciplined about my code.âĂİ There are no short analyses and you will always save time by writing well-formed code.

6.3 Exercises

We are interested in the stage structure of tree species in a forest. We want to be able to describe the allometric relationships between diameter and height of species, and the relationship between, age, diameter and a moisture index (categorical with levels “low” “med” and “high”). We haven't collected field data, yet, but we expect the relationship between diameter and height to follow a power law (height = a*exp(b*diameter), the relationship between age, diameter to be linear, and the relationship between age and diameter and wetness to be linear.

- Write a function which generates a simulated dataset
- Write functions that determine the relationships between the simulated variables. Make sure that you write them in such a way that when actually get around to collecting the field data, all you have to do is send your new data as arguments to the analysis functions
- Write a function that outputs the results your analysis to a file

Session 7

Statistical Models

So far, we have discuss mainly working with R as a programming language, learning how to work with the built-in objects, and how to maniupulate objects programmatically. In this session, we delve more deeply into the statistical problems that R was designed to address. You may ask why so much time was spent in data manipulation and programming topics instead of statistical tests. The answer is simple. Given a well-formed data frame (**??**), most statistical tests in R are one line of code. The bulk of the work of R data analysis is in actually getting your data into a form expected by the statistical functions.

The focus of this session is the end product of statistical analyses, figures and tables. As you will learn, most statistical tests take similar and predictable arguments. Plotting functions, likewise, are take similar arugments to their test counterparts. This unified framework makes analysis a simple matter of choosing the right test, and the right plot, and feeding the testing and plotting functions the predictor and response variables from your well-formed dataframe.

This is not meant to be a comprehensive treatment of all the different statistical tests and plots that standard R packages offer. One book couldn't hold all that information, and anyway the list of available tests is constantly growing. Instead it is meant to help you understand the basic rules that most statistical tests follow in R. Once you know the rules, you can search the R help for that test.

I do not cover complex statistical models that must be built from the ground up, here. For instance, the possible configurations of multi-level models are endless, and Bayesian inference on such models often requires sampling procedures built specifically for the model. Given sufficient knowledge of probability thoery on the reader's part, however, it is a relatively simple matter to build resampling methods for complex models from the basic commands given in chapter 1 (e.g. `sample`, `rnorm`, `seq`, `rep`), and the looping constructs outlined in chapter 7.

7.1 Linear Models

This simplest statistical model in R is a general linear model (not to be confused with a generalized linear model). A general linear model is simple least-squares linear regression. The function for linear models is `lm`. The minimal arguments to `lm` are a data frame containing all the data and a formula which specifies what are the predictor variables and what is the reseponse variable. The output from this function is a linear model object. The linear model object is a special type of list, with named components corresponding to the different attributes of the linear model like

coefficients or fitted values. This standard input (data frame and formula) and output (list with named components) is common to most statistical models in R (e.g generalized linear models, ANOVA, or Wilcoxan Rank-Sum tests). Below I discuss the inputs to linear models in detail.

7.1.1 Standard Arguments

The usage of `lm` is a follows:

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

The arguments `formula`, `data`, `subset`, `weights`, `na.action`, `model` and `contrasts` are common to many other statistical modelling functions. I have included the help text for each of these arguments here:

formula an object of class `formula` (or one that can be coerced to that class): a symbolic description of the model to be fitted...

data an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `lm` is called.

subset an optional vector specifying a subset of observations to be used in the fitting process.

weights an optional vector of weights to be used in the fitting process. Should be `NULL` or a numeric vector. If non-NULL, weighted least squares is used with weights...

na.action a function which indicates what should happen when the data contain `NA`s. The default is set by the `na.action` setting of `options`, and is `na.fail` if that is unset. The "factory-fresh" default is `na.omit`. Another possible value is `NULL`, no action. Value `na.exclude` can be useful.

contrasts an optional list. See the `contrasts.arg` of `model.matrix.default`.

The only required arguments for `lm` are a `formula` object (discussed below) and a `data.frame` object. A simple example might be:

```
> pred <- sample(1:50, 50, replace = T)
> err <- rnorm(50)
> resp <- 2 + 0.5 * pred + err
> dat <- data.frame(resp, pred)
> mod <- lm(resp ~ pred, data = dat)
> mod

Call:
lm(formula = resp ~ pred, data = dat)

Coefficients:
(Intercept)         pred
     1.7104       0.5099
```

In the snippet above, I have created a simulated dataset where the response variable (`resp`) is a linear function of the predictor (`pred`) plus some additional error (`err`). I combined the predictor and response into a single dataframe (`dat`). The call to `lm` says to used the data frame `dat` to find the columns `resp` and `pred`, and that `resp` is a function of `pred`.

Technically, the statment `lm(resp~pred)` would also work. The response and predictor variables do not have to be given as columns in a dataframe if they are part of the global workspace. But, it is best practice to store all part of a statistical model as a dataframe, simply because the dataframe object force the predictor and responses to have the same length.

There are some specific things that should be noticed about the output of the `lm` function. The first is that the output is of class `lm`, but it is really just a glorified list.

```
> class(mod)

[1] "lm"

> attributes(mod)

$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"
```

You should also notice when the name `mod` is typed on the command line, it is not the list which is printed to the screen. Rather the statement we used to create the model (the call) and the model coefficients are printed. This occurs because a special `print` function is written for objects of class `lm`. Most outputs from statistical models have special `print` methods, that print a brief summary of results, rather than outputting the entire list.

Additionally, most statistical model classes have a special `summary` function written for them that behaves differently than the `summary` function for dataframes or vectors.

```
> summary(dat)

      resp             pred
 Min.   : 1.700   Min.   : 1.00
 1st Qu.: 8.564   1st Qu.:14.25
 Median :16.024   Median :29.00
 Mean   :14.998   Mean   :26.06
 3rd Qu.:20.673   3rd Qu.:37.75
 Max.   :28.603   Max.   :50.00

> summary(mod)

Call:
lm(formula = resp ~ pred, data = dat)
```

```
Residuals:
     Min       1Q   Median       3Q      Max
-2.46046 -0.57122 -0.03414  0.64647  1.83264

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.710399   0.280548   6.097 1.78e-07 ***
pred        0.509888   0.009358  54.486  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9806 on 48 degrees of freedom
Multiple R-squared: 0.9841,        Adjusted R-squared: 0.9838
F-statistic:  2969 on 1 and 48 DF,  p-value: < 2.2e-16
```

The `summary` function for the `lm` class prints to the screen the output we would typically expect to see in a results section of paper, with a regression table and Perason R^2 statistics. You can expect the `summary` function for other statistical models to output similary results, namely what publishers expect to see in a summary table of a regression.

At it's core, however, objects of class `lm` are still justs lists, and you can access important components of the model output just as you would any other list

```
> mod$coefficients

(Intercept)        pred
  1.7103986   0.5098878

> mod$residuals[1:10]

         1          2          3          4          5          6          7
 1.5122023 -0.6982323 -2.4604560 -0.5205254  1.1446545 -0.1276449  1.0018029
         8          9         10
 0.5407996  0.9353999  1.3251650
```

Finally, the `lm` class has a special plotting function written for it. Plotting is discussed in the final chapter, but I give a brief synopsis of model plots here. A plot of the linear model can be called simply using the `plot` function with the model as the only argument.

Figure 7.1.1

```
> plot(mod)
```

Just as with the `print` and `summary` commands, objects of class `lm` have a special plotting function written for them. This special plotting function does not plot what you might expect, though. A reasonable expectation might be that the `plot` function would produce a scatter plot with the model data plotted as points, and the best fit regression line plotted on top of them. This is not the case, however. Instead, the default `plot` method for `lm` objects generates a series of 4 plots

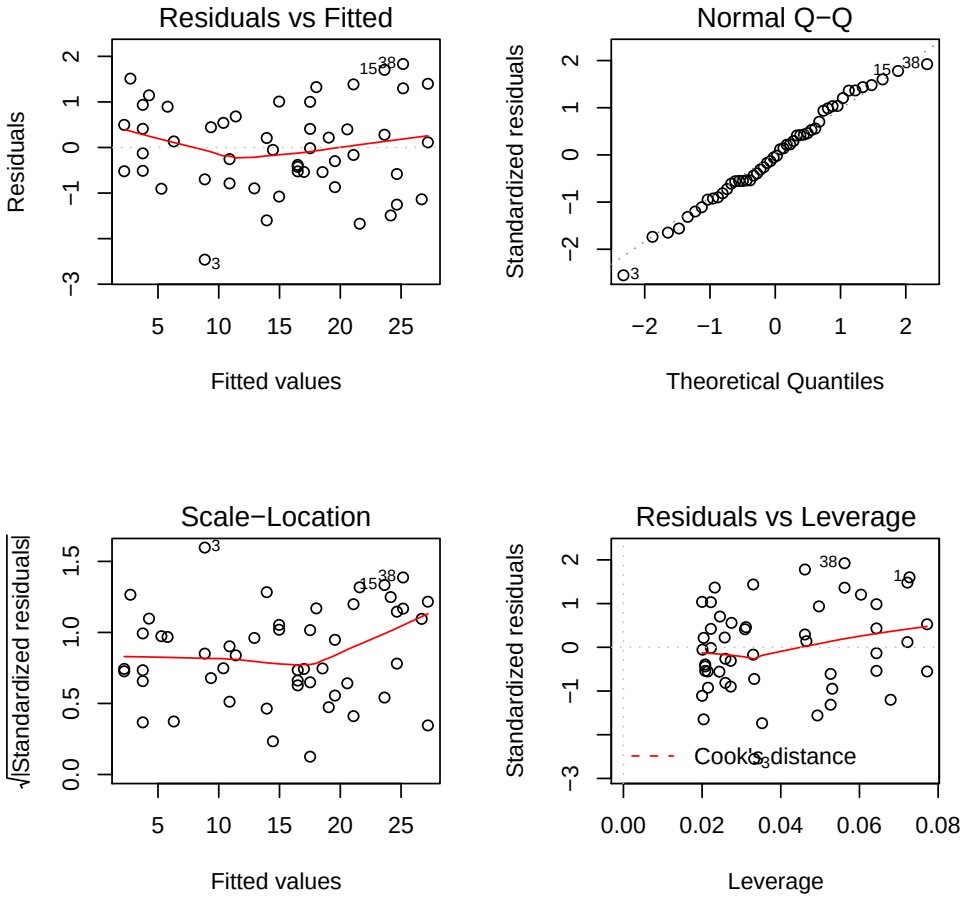

Figure 7.1: Graphical output of the plot function for lm objects.

meant to help you validate the *assumptions* of the linear model. For instance, the first plot of the series plots the residuals of the regression against the fitted values. The residuals should be evenly distributed as the fitted values increase. If the spread of residuals is greater for high fitted values than low fitted values, then the data is heteroscedastic and violates one of the most important assumptions of linear regression. Again, `plot` functions for otherstatistical models behave the same way. Their default plotting methods allow you to test assumptions of the model.

The arugments `subset` is used frequently to remove outliers from a regression. Say for instance, that the plot of leverage against the standardized residuals reveals that a few datapoints are strongly influencing the coefficient estimates (the lower left plot in Figure ?). I set up this situation here:

Figure 7.1.1

```
> outpred <- sample(1:50, 3, replace = T)
> err <- rnorm(3)
> outresp <- 8 + 2 * outpred + err
> outdat <- data.frame(resp = outresp, pred = outpred)
```

```
> dat2 <- rbind.data.frame(dat, outdat)
> mod2 <- lm(resp ~ pred, dat2)
> summary(mod2)

Call:
lm(formula = resp ~ pred, data = dat2)

Residuals:
   Min     1Q Median     3Q    Max
-7.264 -4.411 -3.332 -1.173 63.607

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.3384     3.8065   0.352    0.727
pred          0.6425     0.1259   5.104 4.97e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 13.46 on 51 degrees of freedom
Multiple R-squared: 0.3381,        Adjusted R-squared: 0.3251
F-statistic: 26.05 on 1 and 51 DF,  p-value: 4.972e-06

> plot(mod2, which = 5)
```

I created 3 outliers whose relationship between the response and predictor were very different from the rest of the data. These data are shown on the plot (Figure ?), where they are labelled as outliers. Also, notice the substantial decrease in the Pearson R^2 with the outliers in the model. To remove them from the regression simply use the `subset` argument to remove the rows.

```
> mod3 <- lm(resp ~ pred, data = dat2, subset = -c(51, 52, 53))
> summary(mod3)

Call:
lm(formula = resp ~ pred, data = dat2, subset = -c(51, 52, 53))

Residuals:
     Min       1Q   Median       3Q      Max
-2.46046 -0.57122 -0.03414  0.64647  1.83264

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.710399   0.280548   6.097 1.78e-07 ***
pred        0.509888   0.009358  54.486  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9806 on 48 degrees of freedom
```

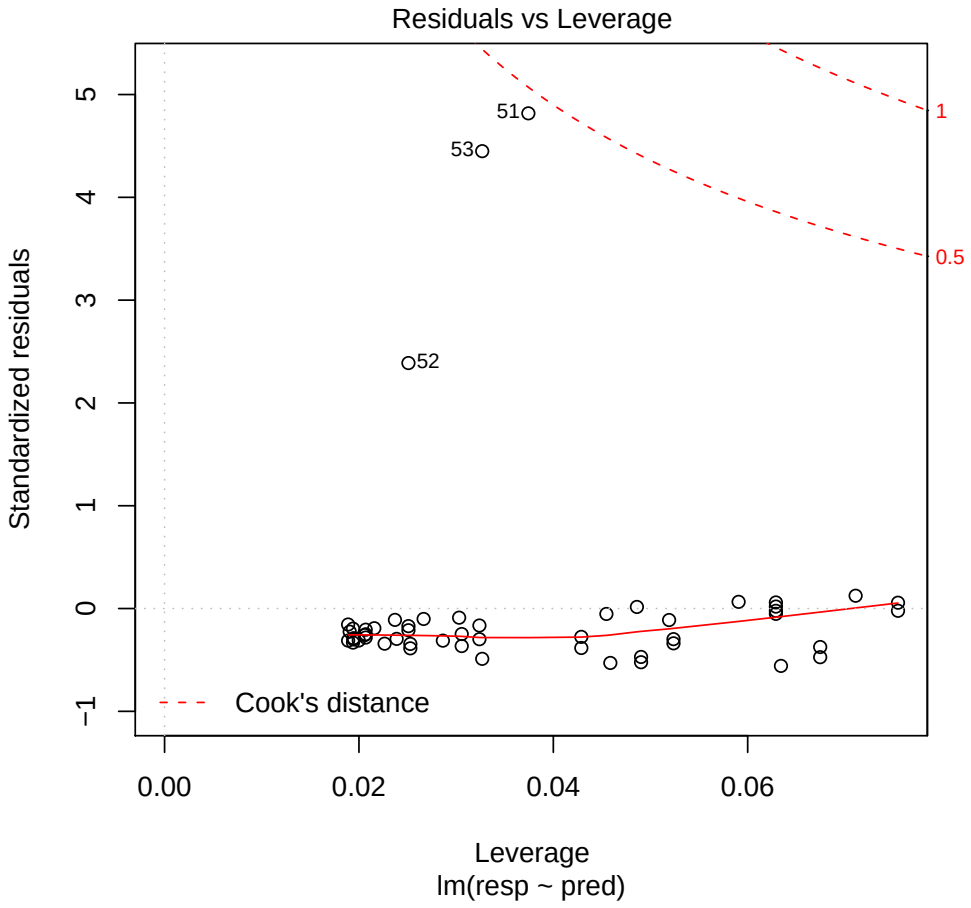

Figure 7.2: Leverage against standardized residuals plot for linear model with outlier. Note the row number labeled on each outlier

```
Multiple R-squared: 0.9841,        Adjusted R-squared: 0.9838
F-statistic:  2969 on 1 and 48 DF,  p-value: < 2.2e-16
```

The `weights` argument is often used in bootstrapping (See **??**). Random weights can be assigned to each observation so that they emulate random resampling of the data. Weights can also be used when there is a additional factor that affects the predictability of the model, but is not included explicitly in the model formula. In a clinical trial, for instance, subjects may be ranked according to the likelihood that their response is "valid". These rankings can be used as weights in the regression to emphasize "valid" responses over "invalid" ones.

7.1.2 Formulas

The models above illustrate the simplest `formula` object that can be passed to a model: one response and one predictor. A `formula` object is simply a representation of the relationship between

responses and predictors. The ~ character is used to separted responses from predictors and can be thought of as "...is a function of...". So, the formula `y~x` means "response y is a function of predictor x".

When there are multiple predictors, they can have additive effect or a multiplicative effect on the response. These effects are specified with `+` and `*`, respectively. The formula `weight~height+weight` can be traslated as "response weight is a function of the additive effects of height and weight". The formula `weight~height*width` can be translated as "response weight is a function of the multiplicative effects of height and weight". Another way to think about multiplicative effects is that they are the summation of all individuals effect of the predictor and the interaction effects of the predictors:

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \beta_2 X_{2,i} + \beta_3 X_{1,i} X_{2,i} + \epsilon_i \tag{7.1}$$

$$\epsilon \sim N\left(0, \sigma^2\right) \tag{7.2}$$

Interaction effect can also be written explicitly using a colon to separate predictors. So, `weight~height+width+height:width` is equivalent to `weight~height*width`.

Numeric constants can also be added to `formula` objects. The formula `y~(a+b)^2` specifies the model:

$$Y_i = \beta_0 + \left(\beta_1 a_i + \beta_2 b_i\right)^2 + \epsilon_i \tag{7.3}$$

$$\epsilon \sim N\left(0, \sigma^2\right) \tag{7.4}$$

Formulas can also contain a `-` to remove particular components from the model. `y~a*b-a:b` is eqivalent to `y~a+b`. While the usage `-` may not apparent immediately, it will become very important when suites models.

The intercept of a model has the special symbol `1` in formulas. An intercept is included by default in all formulas even though it is not written explicity in the formula. If for some reason, you need the intercept of a model to be 0, then you can simply remove the intercept from the model (`y~a+b-1`).

The one final character that has a special interpretation in formulas is the period. A period stands for "every other variable". So, if you have a dataframe with columns `y`, `a`, `b`, and `c`, the formula `y~.` when called with the `data` argument equal to the dataframe is equivalent to `y~a+b+c`. The formula `y~.*.` is equivalent to `y~a+b+c+a:b+b:c+a:c`, all 1st and 2nd order effects. The formula `y~.*.*.` includes all primary, secondary, and tertiary effects and is equivalent to `y~a+b+c+a:b+b:c+a:c+a:b:c` or `y~a*b*c`.

Formulas are used not only in statistical modesl in R, but in plotting as well. This makes it easy to go from tabular results to graphical ones without too much thought on the part of the user. In our `lm` example:

Figure 7.1.2

```
> lm(resp ~ pred, data = dat)

Call:
lm(formula = resp ~ pred, data = dat)

Coefficients:
```

```
(Intercept)         pred
     1.7104       0.5099

> plot(resp ~ pred, data = dat)
```

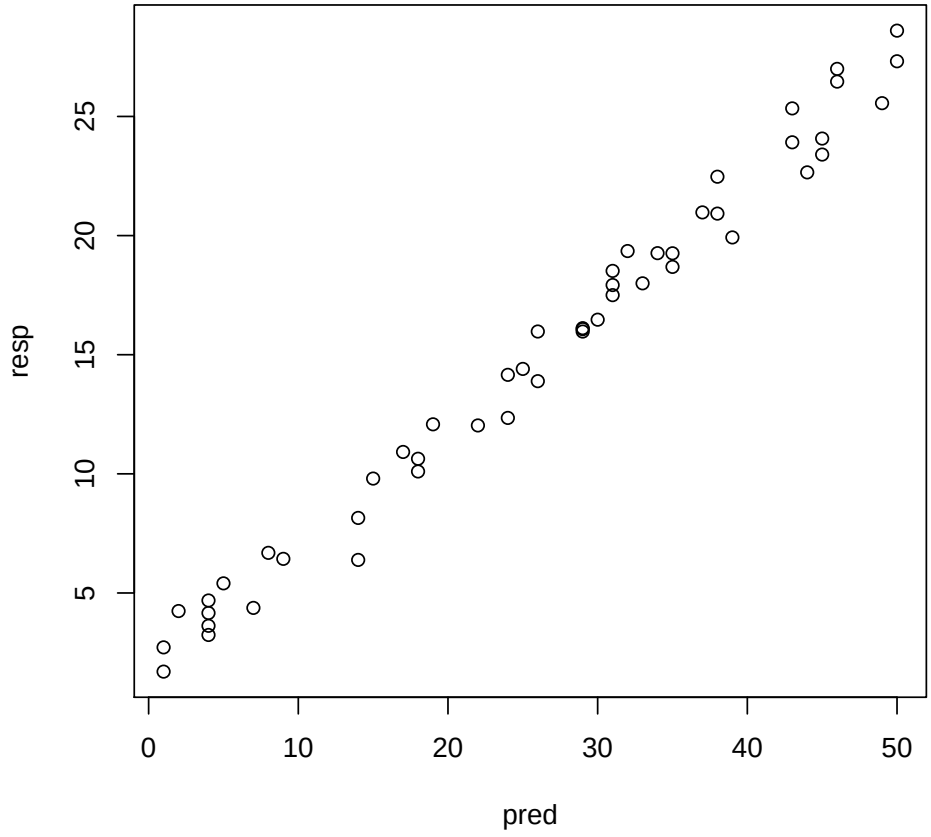

Figure 7.3: Scatterplot generated using a formula object

Because we have used the formula construct, the call to the model is identical to the scatter plot call.

7.1.3 Alternatives

It has been intimated that the most of the arguments to `lm` function are constant across many different statistical models. Here, I provide a brief overview of some commonly used models. It is by no means an exhastive list. There are thousands of R packages and most of them have a statistical model function of some sort in them. Good developers are aware of the transparency of the general `lm` arguments, and try to make their function definitions conform closely to these

specification with arugments like `formula`, `data`, and `weights`. So, you should be able to install virtually any package and develop models with that package very quickly. The most common statistical models are available in the `stats` package which is loaded by default when R starts.

glm This function is used for generalized linear models. These model are common for data that do not fit the assumptions of standard linear models. Foremost among these are data where the responses are binary (logistic regression) or counts (Poisson regression). `glm` accepts virtually the same arguments and `lm` with the addition of a `family` argument which specifies the structure of the errors and the transformation used. For logistic regression the family should be `binomial` (whose default tranformation is the logit). For poisson regression the family should be `poisson` (whose default transformation is the log function).

```
> logitInv <- binomial()$linkinv
> pred <- sample(1:50, 50, replace = T)
> err <- rnorm(50)
> resp1 <- round(exp(0.1 + 0.001 * pred + err))
> resp2 <- round(logitInv(0.2 * pred + err))
> dat <- data.frame(resp1, resp2, pred)
> dat[1:5, ]

  resp1 resp2 pred
1     0     1    6
2     8     1   22
3    49     1   16
4     1     1   41
5     1     1   49

> glm(resp1 ~ pred, family = poisson)

Call:  glm(formula = resp1 ~ pred, family = poisson)

Coefficients:
(Intercept)         pred
    1.34323     -0.02349

Degrees of Freedom: 49 Total (i.e. Null);  48 Residual
Null Deviance:            345
Residual Deviance: 333.4         AIC: 415.5

> glm(resp2 ~ pred, family = binomial)

Call:  glm(formula = resp2 ~ pred, family = binomial)

Coefficients:
(Intercept)         pred
     1.6903       0.1649
```

```
Degrees of Freedom: 49 Total (i.e. Null);  48 Residual
Null Deviance:            9.804
Residual Deviance: 7.762         AIC: 11.76
```

gam This function is used for generalized additive models (GAMs). It is avaiable in the package `mgcv` which comes with the standard R distribution but is not loaded by default. Rather than assuming a constant linear relationship between predictor and responses, GAMs fit a spline to one or more predictors. The formula for the `gam` function contains the special construct `s(`*pred*`)` for predictors to which a spline is fitted. GAMs are useful for distinguishing the relative importance of predictors, but are of little use for predictive purposed since GAMs are almost by definition overfitted.

```
> library(mgcv)
> pred1 <- sample(1:50, 50, replace = T)
> pred2 <- sample(1:50, 50, replace = T)
> resp <- rnorm(50)
> mod <- gam(resp ~ pred1 + s(pred2))
> summary(mod)

Family: gaussian
Link function: identity

Formula:
resp ~ pred1 + s(pred2)

Parametric coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.285850   0.257873   1.108    0.273
pred1       -0.007212   0.007797  -0.925    0.360

Approximate significance of smooth terms:
         edf Ref.df     F p-value
s(pred2)   1    1.5 0.184   0.769

R-sq.(adj) =  -0.0167   Deviance explained = 2.48%
GCV score = 0.70407   Scale est. = 0.66183   n = 50
```

anova This function performs a standard analysis of variance (ANOVA) for a continuous response against one or more categorical predictors (factors). It is included here because it takes as its argument the output of a linear model. It should be noted that `lm` works just fine with both factors and continuous variables. When used in combination with `lm` this is often referred to as analysis of covariance `ANVOCA`. The output of `anova` is slightly different than that of `lm`. `anova` output is focused on the standard ANOVA output, namely, the significance of differences between versus within groups, while `lm` output focuses on the contrasts of different factors levels.

```
> pred1 <- gl(5, 10)
> pred2 <- gl(2, 25, labels = c("a", "b"))[sample(1:50)]
> err <- rnorm(50)
> resp <- 10 + sapply(pred1, function(x) {
+     switch(x, 2, 10, 3, 4, 1)
+ }) + sapply(pred2, function(x) {
+     switch(x, 5, 3)
+ }) + err
> dat <- data.frame(resp, pred1, pred2)
> mod.lm <- lm(resp ~ pred1 + pred2, data = dat)
> anova(mod.lm)

Analysis of Variance Table

Response: resp
          Df Sum Sq Mean Sq F value    Pr(>F)
pred1      4 482.21  120.55 154.326 < 2.2e-16 ***
pred2      1  58.54   58.54  74.933 4.698e-11 ***
Residuals 44  34.37    0.78
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> summary(mod.lm)

Call:
lm(formula = resp ~ pred1 + pred2, data = dat)

Residuals:
     Min       1Q   Median       3Q      Max
-1.72886 -0.59850  0.09146  0.55377  1.54302

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  17.2203     0.3084  55.841  < 2e-16 ***
pred12        8.2269     0.3961  20.769  < 2e-16 ***
pred13        1.2940     0.3987   3.246  0.00224 **
pred14        1.7316     0.3987   4.343 8.14e-05 ***
pred15       -0.9890     0.3961  -2.497  0.01635 *
pred2b       -2.2561     0.2606  -8.656 4.70e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8838 on 44 degrees of freedom
Multiple R-squared: 0.9402,        Adjusted R-squared: 0.9334
F-statistic: 138.4 on 5 and 44 DF,  p-value: < 2.2e-16
```

7.2 Model Selection

We won't have time for this.

7.3 Bootstrapping

We won't have time for this.

Session 8

Plotting

One of the great advantages of R is the flexibility of plotting functions it offers. R provides a wide variety of plotting functions from high-level (e.g. plots using only formulas) to low-level (drawing individual shapes). A great reference for R plotting is: Murrell, P. 2005. R Graphics. . This book covers in much greater detail the concepts that I summarize here.

R plots are highly customizable. You can change every aspect of a given plot from the symbols used for plotting points to the length and position of tick marks on the axes. As a consequence, generating a manuscript-ready plot often requires more lines of code than a typical statistical analysis does. Customized plots are ideal candidates for encapsulation in functions. You pass your data as an argument to the function, and the function generates the specific plot you are trying to acheive.

8.1 Basic Plotting

All plots in R work are built on the transparency model. Imagine a plot being built by laying successive transparencies on an overhead projector (the old man's power-point). You start with a basic plot transparency, and once you lay it on the overhead it can no longer be edited. So, you might start by laying a sheet with the axes down. Then, you would add a sheet that just had the points. Finally you would add a best-fit line transperency to complete the plot. The finished plot looks like a scatterplot with a regression line, but it was built by successively adding the different pieces of the plot on top of each other. Any part of the transparency that has a graphic on it will cover up the graphics on the transparencies below it. So, in the example above, the best-fit line would appear on top of the points of the scatter plot, and cover any points that it intersected.

The transparency model is another reason why writing plots in functions is a good idea. In our scatterplot example, suppose that after creating the plot, we discover that our tick marks are not long enough. Well, the tick mark were the first tranpency we laid down. In order to change the length of the tick marks, we have to start all over again laying each transparency down in order.

The general command for making a high-level plot is `plot`. Unfortunately, the help files for the general plot function are somewhat confusing and scattered. For instance, the usage of according to the help file is `plot(x, y, ...)`. The ... arguement can be an enormous ammount of different arguments. The only ones listed on the `plot` help page are `type` (for choosing a line, point, step plot, etc.), and the plot labelling arugments (`main`,`sub`,`xlab`, and `ylab`. The help page redirects you to three other help pages: `par`, `plot.default`, and `plot.formula`. Of those three, the `par` page

is the most helpful. Using the `par` function you can set a wide variety of plot characteristics such as whether or not the axes should be log transformed or whether you have more than one plot on a page. I do not go over all the options for the `par` function here, but it should always be the first option when looking for help customizing plots. Many of the arguments to `par` can be passed to directly to a plot function. So, for instance, the argument `pch` which stands for plotting character is part of the `par` aruguments, but can be passed to a stanard plotting function `plot(y~x,pch=16)` (Note: you have to look at the help file for `points` to see all the available settings for `pch`).

We have already used the function `plot.formula` during our discussion on model formulas. The part of the function name after the period specifies the class of objects that can be plotted by that function. So, when a call is a made using `plot(y~x)`, R sees the first arugment is of class `formula` and chooses the correct plotting funcion `plot.formula` to make the plot. The same is true of linear models, whose class is `lm`. When you pass a linear model as the first argument to the `plot` function, R looks up the function `plot.lm` and makes the series of 4 plots that help verify the assumptions of linear models.

8.1.1 Scatterplots

The simplest plot in R is a scatterplot. Given a predictor and response variable, a scatterplot is simply the data points placed on the in the graphing region. We can make scatterplots using the stanard plot arguments (x and y) or using a formula object.

```
> pred <- sample(1:50, 50, replace = T)
> err <- rnorm(50)
> resp <- 1 + 2 * pred + err
> plot(pred, pred)
> plot(resp ~ pred)
```

To increase the size of points, or any part of the graph for that matter we use the `cex` arguments. The `cex` argument is set to 1 by default, and any change in its value results in a proportional increase or decrease in the size. So, `cex=2` makes the plotting symbols twice as big, and `cex=0.5` makes them half as big. There are counterparts to `cex` for axis (`cex.axis`), main title (`cex.main`), subtitle (`cex.sub`), and axis label sizes (`cex.lab`)

Figure 8.1.1

```
> plot(resp ~ pred, main = "scatterplot", sub = "a")
> plot(resp ~ pred, main = "scatterplot", sub = "b", cex = 2)
> plot(resp ~ pred, main = "scatterplot", sub = "c", cex = 0.5)
> plot(resp ~ pred, main = "scatterplot", sub = "d", cex.main = 2)
```

8.1.2 Box Plots and Histograms

Boxplots (`boxplot`) and histograms (`hist`) are the typical plots for visualizing one-dimensional data, or comparing the distribution between groups for analyses such as ANOVA.

Figure 8.1.2

```
> resp <- rlnorm(50)
> hist(resp)
> boxplot(resp)
```

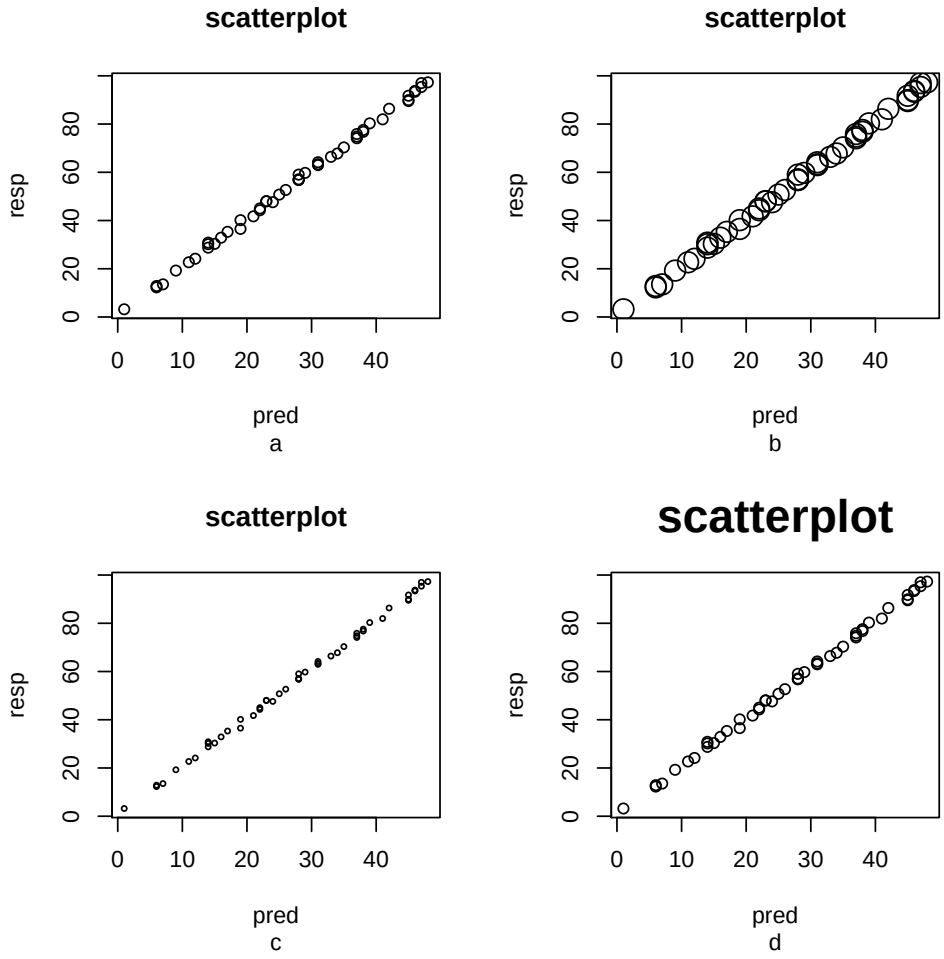

Figure 8.1: Example of using cex to make points larger (b) and smaller (c), and the main title larger (d)

The arguments to boxplot can also be a formula, where the predictor is a factor that will be used to generate multiple boxplots. Again, this is very useful for simple ANOVA analyses. The arugment `notch=T` is also useful in boxplots. The notches represent a reasonable estimation of the 95% confidence interval for a given factor level.

8.1.2

```
> pred <- gl(3, 50, labels = c("low", "med", "high"))
> err <- rnorm(3 * 50)
> resp <- 2 + sapply(pred, function(x) {
+     switch(x, low = 5, med = 1, high = 3)
+ }) + err
> boxplot(resp ~ pred, notch = T)
```

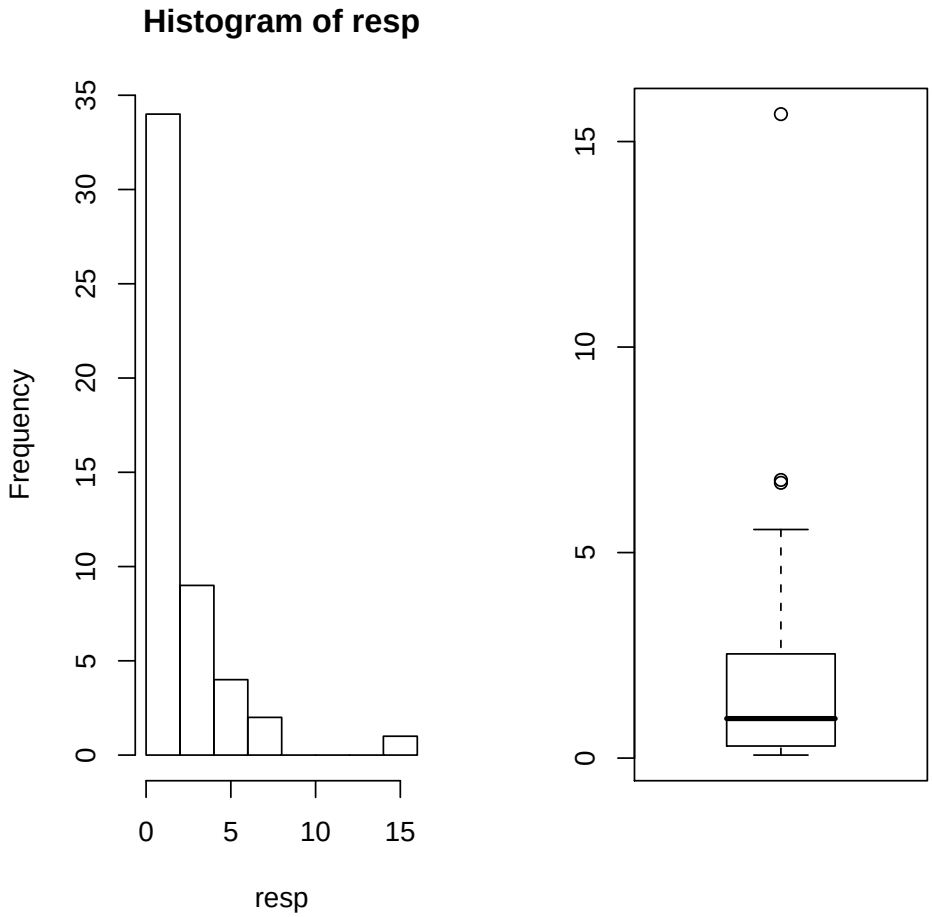

Figure 8.2: simple boxplot and histogram with normal data

8.1.3 Line Plots

There are two ways to generate lines plots in R. The first is to set the `type` argument to `"l"` in the high-level plot function. This usage is atypical. More often a high-level plot is created beforehand, for instance a scatter plot, and lines are added to it using the `lines` function. An important thing to remember is that lines are drawn using successive points. So, the points must be sorted in order along the x-axis if the line is to look correct. An easy way to do that is to generate a new sorted sequence of predictor values, and preict the response variable based on the model using `predict`.

Figre 8.1.3

```
> pred <- sample(1:50, 50, replace = T)
> err <- rnorm(50, sd = 5)
> resp <- 1 + 2 * pred + err
> mod <- lm(resp ~ pred)
> plot(resp ~ pred)
> xrng <- range(pred)
```

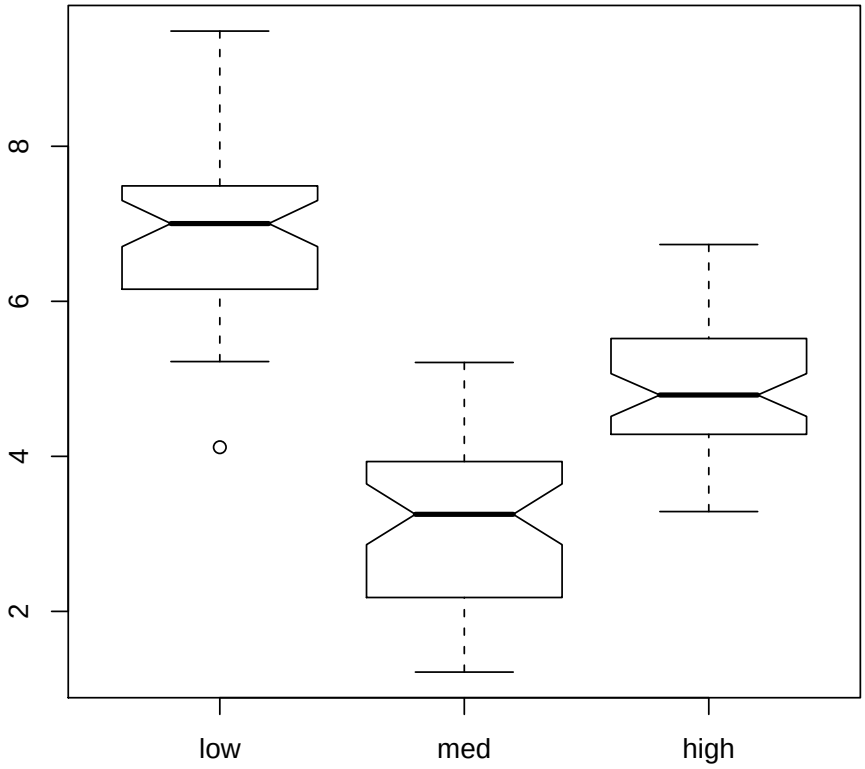

Figure 8.3: Simple boxplot and histogram with normal data. Boxplot of an ANOVA using notch=T

```
> xseq <- seq(xrng[1], xrng[2], length = 100)
> yseq <- predict(mod, data.frame(pred = xseq))
> lines(yseq ~ xseq, col = "red")
```

The line in the above plot also makes use of the `col` argument, which can be added to any plot and changes the plotting color of the data. `col` can take integers, hexdecimal RGB values, and string color names. The available string colors of which there are many can be seen using `colors()` function.

A common operation is to fit a spline to a scatterplot. This can be done using the `loess` function.

Figure 8.1.3

```
> pred <- sample(seq(-pi, pi, length = 500), 50, replace = T)
> err <- rnorm(50, sd = 0.1)
> resp <- sin(pred) + err
> plot(resp ~ pred)
```

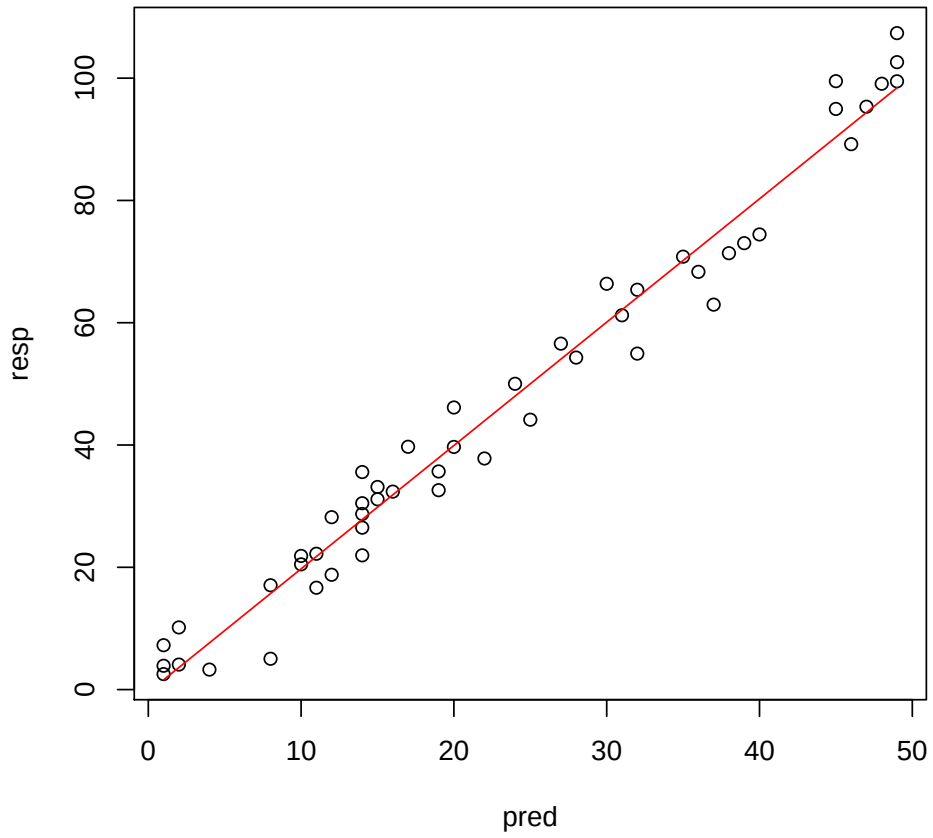

Figure 8.4: Scatter plot with regression line overlain

```
> mod <- loess(resp ~ pred)
> xrng <- range(pred)
> xseq <- seq(xrng[1], xrng[2], length = 500)
> yseq <- predict(mod, data.frame(pred = xseq))
> lines(yseq ~ xseq, col = "red")
```

The fitted lines from linear regressions can be built from predicted values as shown above, but they can also be built from the model coefficients. The function `abline` adds a line to a plot that has a given slope and intercept, or alternatively a straight vertical or horizontal line. The example below introduces another `par` paramters that is useful for lines: `lty` which stands for line type. The defaut is 1, a solid line, increasing integer values of `lty` cause various dashed and dotted lines to be generated. See `?lines` for more details.

8.1.3

```
> plot(resp ~ pred, mod$model)
> abline(mod$coef, col = "red")
```

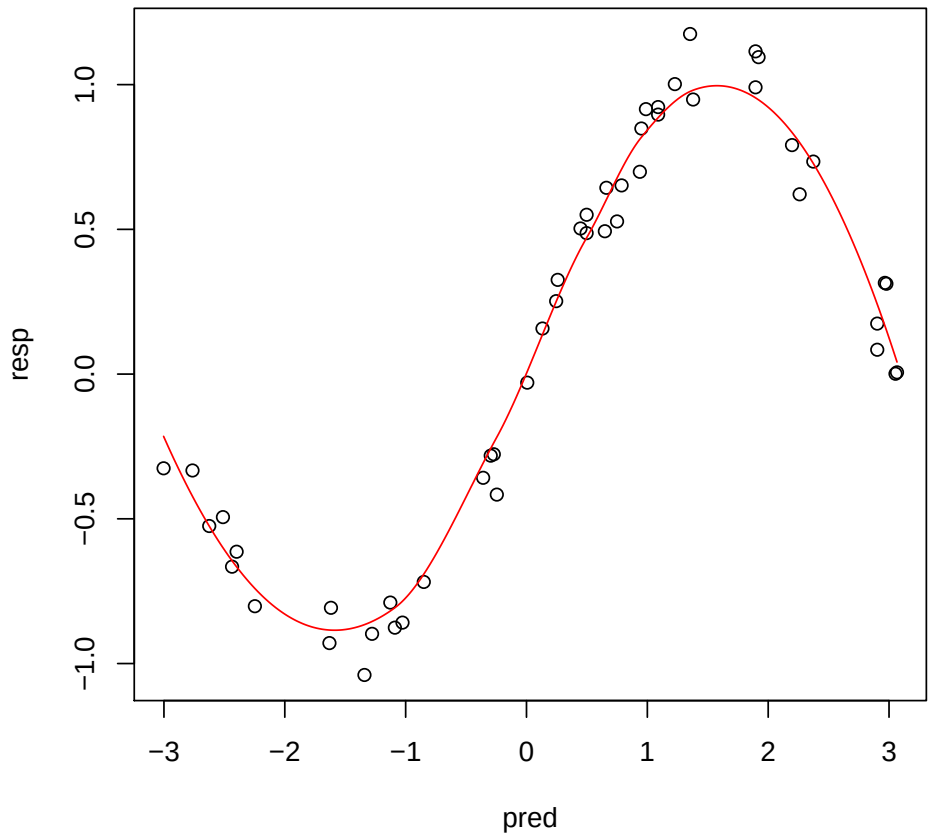

Figure 8.5: scatterplot with loess smoother

,

```
> plot(resp ~ pred, mod$model)
> abline(mod$coef, col = "red")
> abline(h = 20, lty = 2)
> abline(v = 20, lty = 3)
```

Figure 8.6: Scatterplot with a regression line and reference lines plotted with abline

```
> abline(h = 20, lty = 2)
> abline(v = 20, lty = 3)
```

8.1.4 Bar Plots

Many journals do not accept boxplots, and instead prefer grouped data to be shown as barplots with standard errors. There are 2 ways to generate barplots, one using standard graphics, and one using lattice graphics (See **??**). The one using lattice graphics (`barchart`) is slightly easier to use, but I show the standard graphics function (`barplot`) here to give an example which uses the `segments` command. `barplot` takes a vector of heights, which might the mean response of a group. Standard errors can be drawn using any line function, `segments`. `segments` draws line segments instead of a continuous line. It requires four parameters:

x0 a vector of segment starting positions on the x-axis

y0 a vector of segment starting positions on the y-axis

x1 a vector of segment ending positions on the x-axis

y1 a vector of segment ending positions on the y-axis

Using `barplot` also illustrates the fact that plotting functions, just like any other function can have a return value. In the case of `barplot`, the return value is a vector of locations on the x-axis where each bar was drawn. This makes choosing the x values for the `segments` easy, just pass the return value of bar plot the `x0` and `x1` arguments of `segments`.

Figure 8.1.4

```
> levs <- c("low", "med", "high")
> pred <- gl(3, 50, labels = levs)
> err <- rnorm(3 * 50, sd = 5)
> resp <- 2 + sapply(pred, function(x) {
+     switch(x, low = 5, med = 1, high = 3)
+ }) + err
> mu <- tapply(resp, pred, mean)
> se <- tapply(resp, pred, function(x) {
+     sd(x)/sqrt(length(x))
+ })
> ul <- mu + se
> ll <- mu - se
```

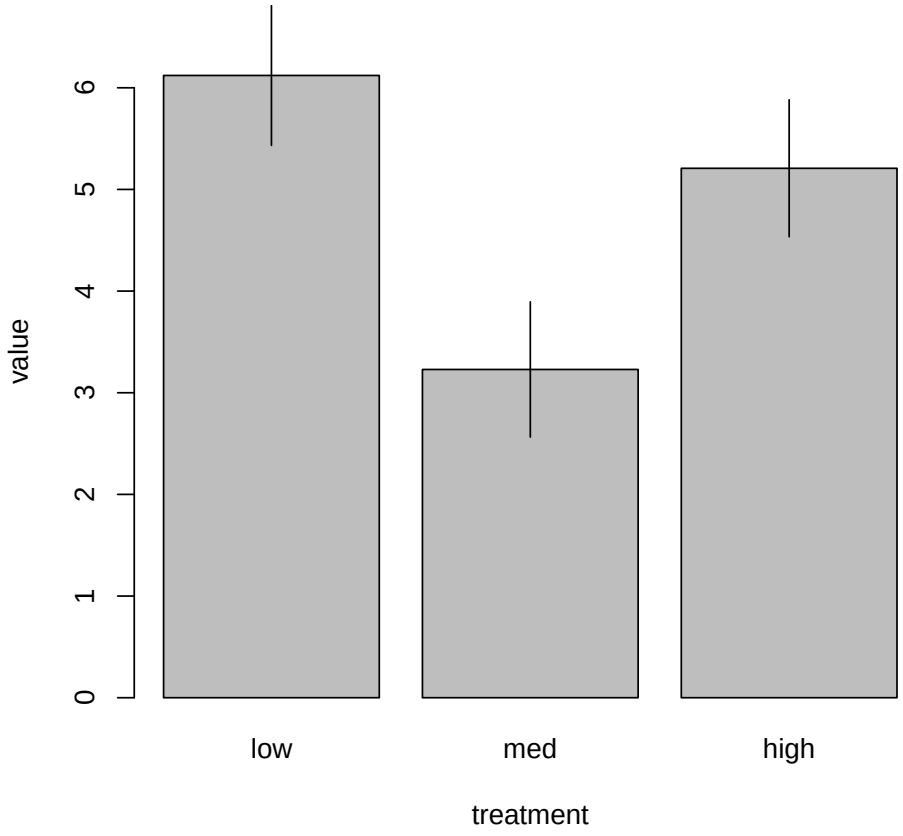

Figure 8.7: Barplot with standard errors

```
> xbars <- barplot(mu, names = levs, ylim = c(0, max(ul)), xlab = "treatment",
+     ylab = "value", )
> segments(xbars, ul, xbars, ll)
```

8.1.5 Multiple Plots

One common task in making graphics for display is putting more than one plot on a graphic device. This is accomplished by setting the `par` parameters `mfcol` or `mfrow` which stand for "multi-figure by column" and "multi-figure by row". The value of these arguments is a vector of length two which specifies (in the case of `mfrow`) the number of rows and number of columns of figure output. This argument is set before any plots are made. Then, when a new call is made to plot, plotting begins in the upper left had corner. The next call to a high-level plotting functions puts a figure in the next row or column depedning on whether you set `mfrow` or `mfcol`. Graphic panels are added until the entire matrix of graphics is filled. Then, when a new plot is called, the old grid is erased, and the new plot is placed in the upper left hand corner. Again, the `par` settings are held until the

graphic device is close, so it is especially important to resent the `par` arguments after creating a multiple graphic figure, or all subsequent calls to plot will continue to place plots in the graphic matrix. In the example below, I use `mfrow` to generate a bivariate density plot with marginal plots for each variable. I also use an example of 3d plot, `contour`.

Figure 8.1.5

```
> s1 <- seq(-3, 3, length = 100)
> s2 <- seq(0.1, 4, length = 100)
> d1 <- dnorm(s1)
> d2 <- dgamma(s2, 3, 2)
> d12 <- d1 %o% d2
> opar <- par(no.readonly = T)
> par(mfrow = c(2, 2))
> contour(x = s1, y = s2, z = d12, xlab = "V1", ylab = "V2")
> plot(d2, s2, type = "l", xlab = "density", ylab = "V2")
> plot(s1, d1, type = "l", ylab = "density", xlab = "V1")
> par(opar)
```

8.2 Lattice Graphics

There is a special graphics package in R called `lattice`. This package is used to produce trellis graphics. Trellis graphics are multi-panel graphs where each panel repesents the levels of one or more predictor variables. They are very useful for visualizing multivariate regression data and results. They also are built strongly upon the formula structure of statistical models. There is an additional symbol for use in lattice plot formulas, `|`. Variables after the bar, represent grouping variables. These are the variables that will be varied across the panels of the lattice plot.

Lattice graphics are not loaded by default in R. You must load the lattice library in each session where you will use it.

Figure 8.2

```
> library(lattice)
> pred1 <- gl(3, 50, labels = c("low", "med", "high"))
> pred2 <- sample(1:50, 150, replace = T)
> err <- rnorm(150)
> res <- 2 + 2 * pred2 + 3 * sapply(pred1, function(x) {
+     switch(x, 1, 2, 3)
+ }) * pred2 + err
> xyplot(resp ~ pred2 | pred1)
```

Adding elements to a lattice plot does not work just like adding elements to a standard graphic. Each panel in a lattice graphic is drawn using a panel function, which is one of the arguments to a lattice plot. To add an element to each panel, say a best-fit line over the scatter plot, you must add the line to the panel function. Selections the many built-in panel functions are listed below. You may also write your own customized panel function. Each panel function can be called as a lattice function in its own right. For instance, you can use `panel.densityplot` as part of the panel functionwithin a call to `xyplot`, or you can just call `density.plot` to have a lattice density plot only.

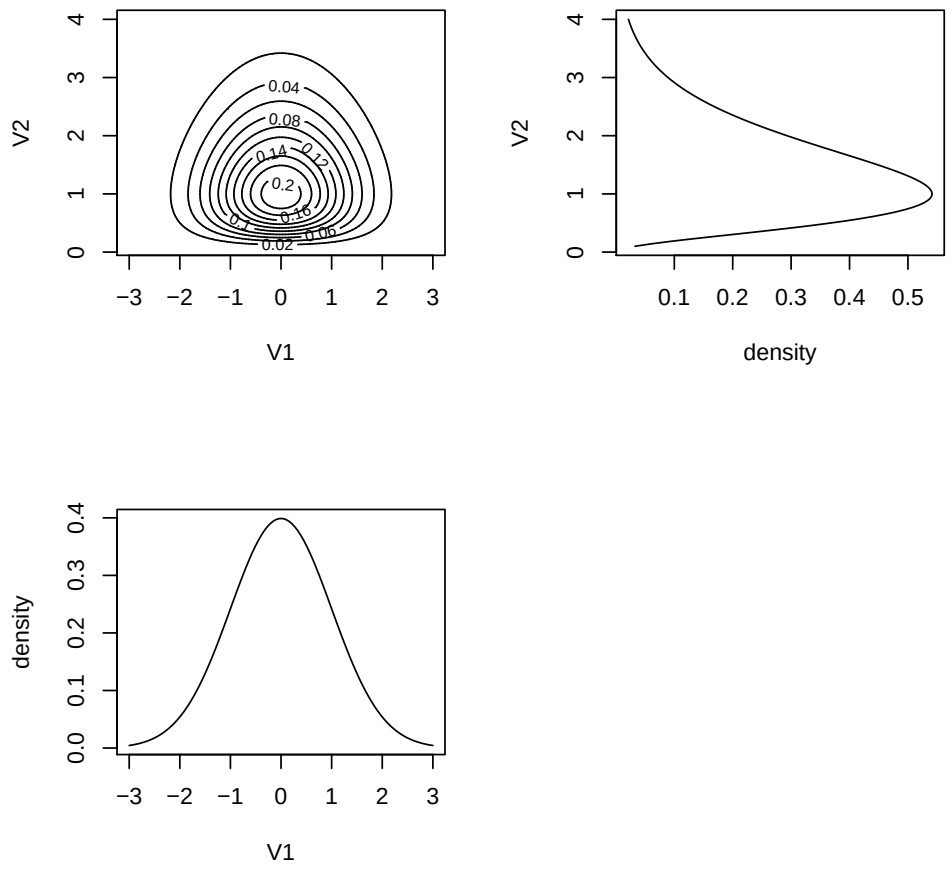

Figure 8.8: Multifigure plot showing multiple regression resuls, with marginals

panel.abline Add a line to a plot

panel.average Add a mean line to a plot

panel.barchart Default Panel Function for barchart

panel.bwplot Default Panel Function for bwplot

panel.cloud Default Panel Function for cloud

panel.curve Add a curve from an expression

panel.densityplot Default Panel Function for densityplot

panel.dotplot Default Panel Function for dotplot

panel.functions Useful Panel Functions

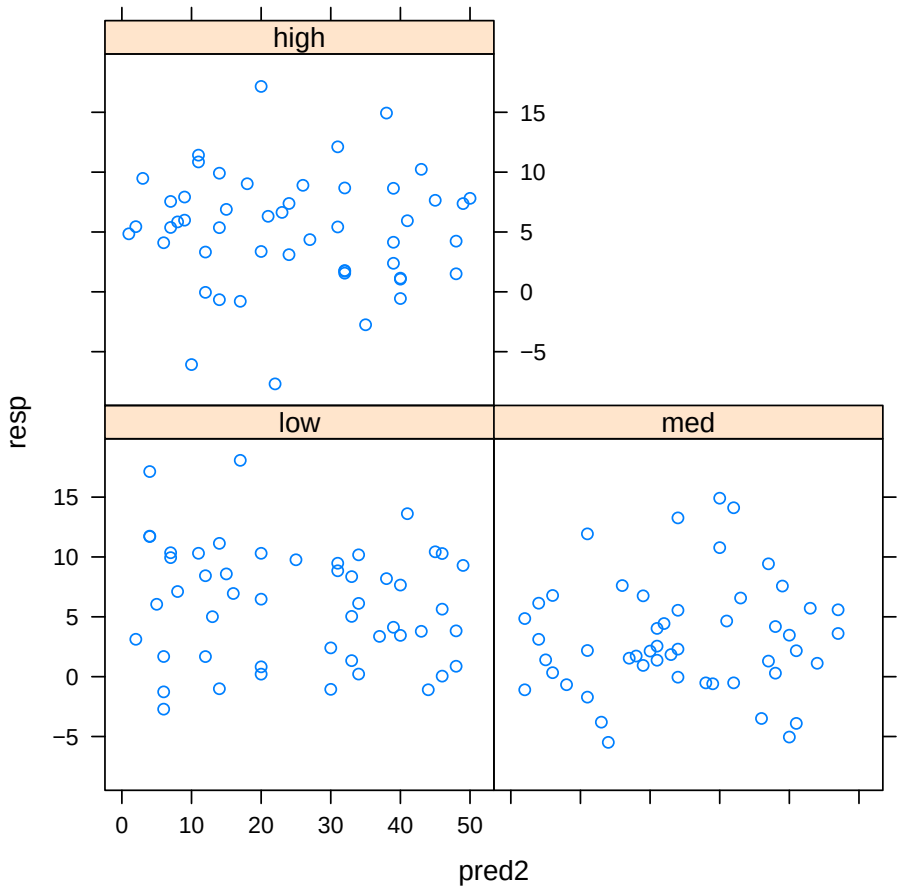

Figure 8.9: simple lattice plot of a response versus a predictor across a grouping variable

panel.histogram Default Panel Function for histogram

panel.levelplot Default Panel Function for levelplot

panel.loess Add a loess smoother to a pot

panel.lmline Add a best fit regression line

panel.pairs Default Superpanel Function for splom

panel.parallel Default Panel Function for parallel

panel.rug Add a rug plot to an axis

panel.qqmath Default Panel and Prepanel Function for qqmath

panel.qqmathline Useful panel function with qqmath

panel.stripplot Default Panel Function for stripplot

panel.superpose Panel Function for Display Marked by groups

panel.violin Panel Function to create Violin Plots

panel.xyplot Default Panel Function for xyplot

Below is an example of a lattice plot with multiple panel functions in a single plot, using the data generated in the previous code block.

Figure 8.2

```
> pan.fun <- function(x, y, ...) {
+     panel.xyplot(x, y, ...)
+     panel.lmline(x, y, col = "red", ...)
+     panel.abline(h = mean(y), lty = 3, ...)
+     panel.abline(v = mean(x), lty = 3, ...)
+ }
> xyplot(resp ~ pred2 | pred1, panel.function = pan.fun)
```

8.3 Execises

Using the code you created in the previous set of exercises:

1. Regress age against diameter and wetness.
2. Regress the number of stems against wetness.

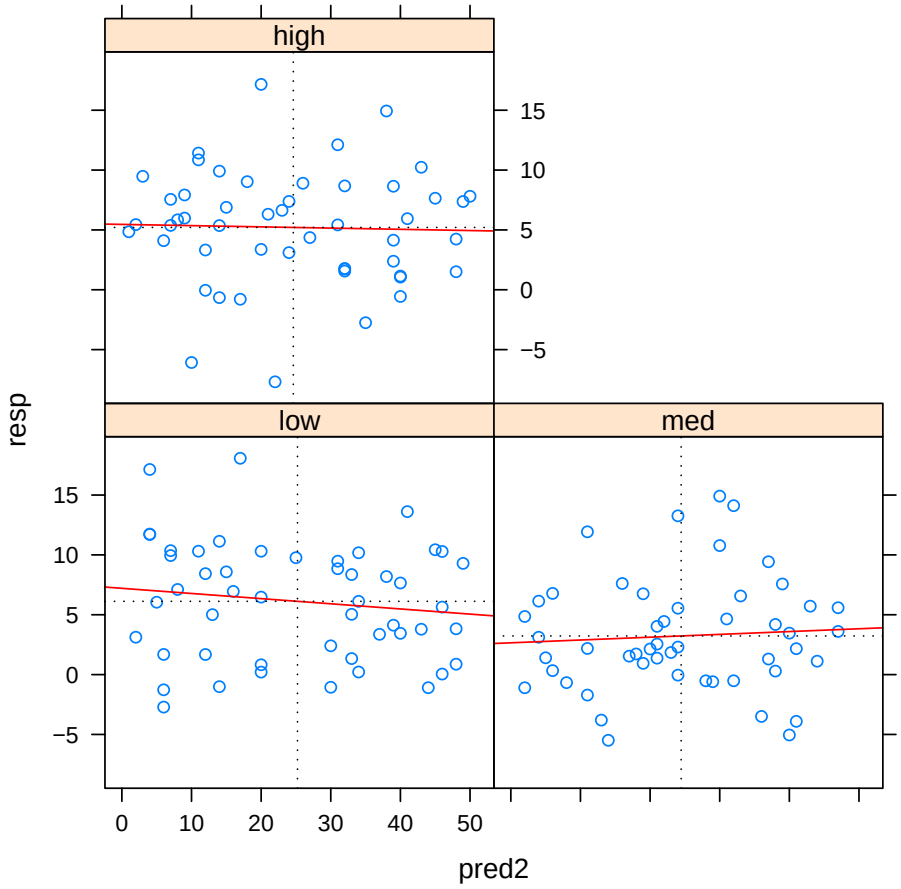

Figure 8.10: Lattice plot with a customized panel function which generates both a scatterplot,fitted line, and x,y averages.