

# Stat 133 Class Notes - Spring, 2011

Phil Spector

May 31, 2011

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	What's this course about? . . . . .	6
1.2	Some Basic Concepts in Computing with Data . . . . .	7
1.3	A Short Note on Academic Integrity . . . . .	7
1.4	Introduction to R . . . . .	8
<b>2</b>	<b>The R Language</b>	<b>10</b>
2.1	Data in R . . . . .	11
2.2	Vectors . . . . .	11
2.3	Modes and Classes . . . . .	17
2.4	Reading Vectors . . . . .	17
2.5	Missing Values . . . . .	18
2.6	Matrices . . . . .	19
2.7	Data Frames . . . . .	21
2.8	More on Data Frames . . . . .	24
2.9	Reading Data Frames from Files and URLs . . . . .	25
2.10	Working with Multiple Data Frames . . . . .	32
2.11	Adding Color to Plots . . . . .	36
2.12	Using Dates in R . . . . .	39
2.13	Data Summaries . . . . .	43
2.14	Functions . . . . .	45
2.15	Functions . . . . .	48
2.16	Sizes of Objects . . . . .	55
2.17	Character Manipulation . . . . .	55
2.18	Working with Characters . . . . .	60
<b>3</b>	<b>Unix</b>	<b>63</b>
3.1	Software for Remote Access . . . . .	64
3.2	Basics of Unix . . . . .	64
3.3	Command Path . . . . .	64
3.4	Basic Commands . . . . .	65
3.5	Command History . . . . .	65
3.6	Editors . . . . .	66

3.7	Wildcards . . . . .	66
3.8	Redirection . . . . .	67
3.9	Job Control . . . . .	67
<b>4</b>	<b>Regular Expressions</b>	<b>70</b>
4.1	Regular Expressions . . . . .	71
4.2	Regular Expressions . . . . .	73
4.3	How matches are found . . . . .	76
4.4	Tagging and Backreferences . . . . .	79
4.5	Getting Text into R . . . . .	79
4.6	Examples of Reading Web Pages with R . . . . .	81
4.7	Reading a web page into R . . . . .	81
4.8	Another Example . . . . .	82
4.9	Dynamic Web Pages . . . . .	84
4.10	Writing a Function . . . . .	88
4.11	Another Example . . . . .	88
<b>5</b>	<b>Graphics</b>	<b>92</b>
5.1	More on Plotting . . . . .	93
5.2	Multiple Plots on a Page . . . . .	99
5.3	More on Barplots . . . . .	103
5.4	Mapping . . . . .	106
5.5	The Lattice Plotting Library . . . . .	116
5.6	Customizing the Panel Function . . . . .	119
5.7	Univariate Displays . . . . .	125
5.7.1	<code>dotplot</code> . . . . .	125
5.7.2	<code>bwplot</code> . . . . .	126
5.7.3	<code>densityplot</code> . . . . .	128
5.8	<code>barchart</code> . . . . .	130
5.9	3-D Plots: <code>cloud</code> . . . . .	132
<b>6</b>	<b>Spreadsheets and Databases</b>	<b>134</b>
6.1	Spreadsheets . . . . .	134
6.2	Writing Spreadsheets . . . . .	139
6.3	Databases . . . . .	140
6.4	Working with Databases . . . . .	142
6.5	Regular Expressions in SQL . . . . .	149
6.6	Accessing databases in R . . . . .	151
6.7	Using SQL in R . . . . .	155
6.8	Reading Spreadsheets with the <code>RODBC</code> Library . . . . .	155

<b>7 Cluster Analysis</b>	<b>158</b>
7.1 Introduction to Cluster Analysis . . . . .	159
7.2 Standardization . . . . .	159
7.3 Distance Measures . . . . .	160
7.4 Clustering Techniques . . . . .	161
7.5 Hierachial Clustering . . . . .	163
7.6 PAM: Partitioning Around Medoids . . . . .	170
7.7 AGNES: Agglomerative Nesting . . . . .	173
<b>8 XML</b>	<b>177</b>
8.1 What is XML? . . . . .	178
8.2 A Simple Example . . . . .	180
8.3 More Complex Example . . . . .	183
<b>9 Programming</b>	<b>189</b>
9.1 Operating on Groups of Data . . . . .	190
<b>10 Classification Analysis</b>	<b>203</b>
10.1 Introduction to Classification Methods . . . . .	204
10.2 kth Nearest Neighbor Classification . . . . .	204
10.3 Cross Validation . . . . .	205
10.4 Linear Discriminant Analysis . . . . .	206
10.5 Recursive Partitioning . . . . .	214
<b>11 Random Numbers and Simulations</b>	<b>219</b>
11.1 Hypothesis Testing . . . . .	220
11.2 Determining Power . . . . .	223
11.3 Probability Distributions . . . . .	226
11.4 A Note about Random Numbers in R . . . . .	227
11.5 t-tests . . . . .	229
11.6 Power of the t-test . . . . .	236
<b>12 Graphical User Interfaces (GUIs)</b>	<b>238</b>
12.1 Graphical User Interfaces . . . . .	239
12.2 Making a Calculator . . . . .	243
12.3 Improving the Calculator . . . . .	246
12.4 Appearance of Widgets . . . . .	246
12.5 Fonts and Colors . . . . .	249
12.6 Plotting . . . . .	252
12.7 Binding . . . . .	256
12.8 Checkbuttons . . . . .	258
12.9 Opening Files and Displaying Text . . . . .	260
12.10 Using Images with the <code>tcltk</code> package . . . . .	264

<b>13 CGI Programming with R</b>	<b>266</b>
13.1 Web Servers . . . . .	266
13.2 CGI Scripting . . . . .	271
13.3 A First CGI program with R . . . . .	272
13.4 Data . . . . .	276
13.5 Combo Forms . . . . .	277
13.6 Graphs . . . . .	279
13.7 Hidden Variables . . . . .	280
13.8 Outgoing HTTP Headers . . . . .	281
13.9 Creating Pretty Output . . . . .	283
13.10 File Upload . . . . .	284
13.11 Debugging CGI Programs . . . . .	284
<b>14 Smoothers</b>	<b>286</b>
14.1 Smoothers . . . . .	287
14.2 Kernel Smoothers . . . . .	287
14.3 Locally Weighted Regression Smoothers . . . . .	291
14.4 Spline Smoothers . . . . .	294
14.5 Supersmooth . . . . .	295
14.6 Smoothers with Lattice Plots . . . . .	296
<b>15 Linear Regression</b>	<b>299</b>
15.1 Linear Regression . . . . .	300
15.2 The <code>lm</code> command . . . . .	300
15.3 Using the model object . . . . .	303
15.4 Regression Diagnostics . . . . .	304
15.5 Collinearity . . . . .	307
15.6 Generalized Additive Models ( <code>gam</code> ) . . . . .	312
15.7 Recursive Partitioning . . . . .	316
15.8 Comparison of the 3 Methods . . . . .	317
<b>16 Analysis of Variance</b>	<b>319</b>
16.1 Analysis of Variance . . . . .	320
16.2 Multiple Comparisons . . . . .	325
16.3 Two-Way ANOVA . . . . .	331
16.4 Another Example . . . . .	335
16.5 More Complex Models . . . . .	340
16.6 Constructing Formulas . . . . .	344
16.7 Alternatives for ANOVA . . . . .	347

# Chapter 1

## Introduction

## 1.1 What's this course about?

The goal of this course is to introduce you to a variety of concepts and methods that are useful when dealing with data. You can think of data as any information that could potentially be studied to learn more about something.

Some simple examples:

1. Sales records for a company
2. Won/Lost records for a sports team
3. Web log listings
4. Email messages
5. Demographic Information found on the web

What may be surprising is that some of the data (for example web log listings or email) consists of text, not numbers. It's increasingly important to be able to deal with text in order to look at the wide variety of information that is available.

In most statistics courses, a lot of time is spent working on the preliminaries (formulas, algorithms, statistical concepts) in order to prepare the student for the interesting part of statistics, which is studying data for a problem that matters to you. Unfortunately, by the time these preliminaries are covered, many students are bored or frustrated, and they leave the course with a distorted view of statistics. In this class, we're going to do things differently. We will concentrate on:

1. Computer Languages – which will let us read in and manipulate data.
2. Graphical Techniques – which will allow us to display data in a way that makes it easier to understand and easier to make decisions based on the data.
3. Technologies – so that we can present these techniques to someone who's not as knowledgeable as us without too much misery.

The main computer language that we will be using in the course is a statistical programming environment known as R (<http://r-project.org>). R is freely downloadable and copyable, and you are strongly encouraged to install R on your own computer for use in this course and beyond. We will use this language for both data acquisition, data manipulation and producing graphical output. R is not the ideal language for all of the tasks we're going to do, but in the interest of efficiency, we'll try to use it for most things, and point you in the direction of other languages that you might want to explore sometime in the future.

Another tool that we'll use are UNIX shell commands, which allow you to store, copy and otherwise manipulate files which contain data, documents or programs. The computer accounts for this course will allow you to access computers running a version of the UNIX operating system

More and more commonly, data is stored on database servers, not in ordinary files. Most database servers use some version of a language known as the Structured Query Language (SQL). We'll use the open-source MySQL database as an example of an SQL database.

While you shouldn't have any problems understanding any of the statistical techniques we study in this class, a certain level of complexity arises when various techniques are combined together and presented as a start-to-finish solution. To learn how we can make techniques we've developed to others, we'll look at graphical user interfaces (GUIs) as well as using web servers to get information from users and display results.

## 1.2 Some Basic Concepts in Computing with Data

Most real-life projects that involve data can be broken down into several steps:

1. Data Acquisition - we need to find (or collect) the data, and get some representation of it into the computer
2. Data Cleaning - Inevitably, there will be errors in the data, either because they were entered incorrectly, we misunderstood the nature of the data, records were duplicated or omitted. Many times data is presented for viewing, and extracting the data in some other form becomes a challenge.
3. Data Organization - Depending on what you want to do, you may need to reorganize your data. This is especially true when you need to produce graphical representations of the data. Naturally, we need the appropriate tools to do these tasks.
4. Data Modeling and Presentation - We may fit a statistical model to our data, or we may just produce a graph that shows what we think is important. Often, a variety of models or graphs needs to be considered. It's important to know what techniques are available and whether they are accepted within a particular user community.

Further, techniques such as simulation and resampling have been developed to allow us to get information even in cases where we don't have exactly the data that we'd like to have.

## 1.3 A Short Note on Academic Integrity

One of the main ways you are going to learn in this course is by discussing the material in class and in lab sections, and I feel that this is one of the most important aspects of the course. However, the programs you write are like an essay or term paper, and you should not share them with others, or ask others to give their programs to you.

The University has a very detailed code of student conduct, available at <http://students.berkeley.edu/uga/conductiii-vii.asp>. Please refer to that document or talk to me if there are any questions.

## 1.4 Introduction to R

Here are some basic concepts about R that we'll return to later in more detail:

- R can be used as a calculator. Any statements that you type at the R prompt will be executed, and the answer printed:

```
> 12 + 9  
[1] 21  
> 17.1* 13  
[1] 222.3  
> 554 /3  
[1] 184.6667
```

Notice that spaces can be placed wherever you'd like, as long as they're not in the middle of a number.

- R is just as happy to work with vectors (more than one value) as it is with scalars (single values). The function to create a vector from individual values is `c()`:

```
> c(10,12,19) + c(8,5,9)  
[1] 18 17 28  
> c(1,2,3) * c(3,2,1)  
[1] 3 4 3
```

When you use operators like `*(multiplication)` or `+(addition)`, R does the operations element by element

- There are three ways to get help in R:

1. The `help()` command, which can be abbreviated by `?`, will open a help page in a browser with information about a particular command. For example, to get help on the `c` function mentioned above, you could type:

```
> help(c)
```

or

```
> ?c
```

2. The `help.search()` function, which can be abbreviated by `??`, will show other functions already installed in your version of R that relate to a particular topic. For example, to see about functions that combine things in R, you could type

```
> ??combine
```

`help.search()` only looks for functions already installed on your computer.

3. The `RSiteSearch()` command will open a browser to a searchable database of questions and answers posted on the R-help mailing list. (See <http://www.r-project.org/mail.html> for more information on the R-help mailing list.)
- When you type the name of an object into R, it will display that object. This can be frustrating when you want to actually run the object. For example, if you type `q` at the R prompt, you'll see:

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

To actually execute the `q` command, type

```
> q()
```

# Chapter 2

## The R Language

## 2.1 Data in R

While R can handle many types of data, the three main varieties that we'll be using are numeric, character and logical. In R, you can identify what type of object you're dealing with by using the `mode` function. For example:

```
> name = 'phil'  
> number = 495  
> happy = TRUE  
> mode(name)  
[1] "character"  
> mode(number)  
[1] "numeric"  
> mode(happy)  
[1] "logical"
```

Note that when we enter character data, it needs to be surrounded by quotes (either double or single), but the symbols `TRUE` and `FALSE` (without quotes) are recognized as values of a logical variable.

Another important characteristic of an object in R is its class, because many functions know how to treat objects of different classes in a special way. You can find the class of an object with the `class` function.

## 2.2 Vectors

Occasionally it may be useful to have a variable (like `name` or `happy` in the above example) to have only a single value (like `'phil'` or `TRUE`), but usually we'll want to store more than a single value (sometimes referred to as a scalar) in a variable. A vector is a collection of objects, all of the same mode, that can be stored in a single variable, and accessed through subscripts. For example, consider the minimum temperature in Berkeley for the first 10 days of January, 2006:

```
50.7 52.8 48.6 53.0 49.9 47.9 54.1 47.6 43.6 45.5
```

We could create a variable called `mintemp` as follows:

```
> mintemp = c(50.7,52.8,48.6,53.0,49.9,47.9,54.1,47.6,43.6,45.5)
```

The `c` function is short for catenate or combine, and it's used to put individual values together into vectors. You can find the number of elements in a vector using the `length` function.

Once you've created a vector, you can refer to the elements of the vector using subscripts. Numerical subscripts in R start at 1, and continue up to the length of the vector. Subscripts of 0 are silently ignored. To refer to multiple elements in a vector, simply use the `c` function to create a vector of the indexes you're interested in. So to extract the first, third, and fifth values of the `mintemp` vector, we could use:

```
> mintemp[c(1,3,5)]
[1] 50.7 48.6 49.9
```

If all of the subscripts of an object are negative, R will ignore those values, and use just the remaining elements. To extract all of the elements of `mintemp` except the first and last (tenth), use:

```
> mintemp[-c(1,10)]
[1] 52.8 48.6 53.0 49.9 47.9 54.1 47.6 43.6
```

In most programming languages, once you're dealing with vectors, you also have to start worrying about loops and other programming problems. Not so in R! For example, suppose we want to use the conversion formula

$$C = \frac{5}{9}(F - 32)$$

to convert our Fahrenheit temperatures into Celsius. We can act as if `mintemp` is just a single number, and R will do the hard part:

```
> mintempC = 5/9 * (mintemp - 32)
> mintempC
[1] 10.388889 11.555556 9.222222 11.666667 9.944444 8.833333 12.277778
[8] 8.666667 6.444444 7.500000
```

In fact, most similar operations in R are vectorized; that is they operate on entire vectors at once, without the need for loops or other programming.

There are some shortcuts to generate vectors. The colon operator lets you generate sequences of integers from one value to another. For example,

```
> x = 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
```

For more control, see the help page for the `seq` function.

You can repeat values using the `rep` function. This function is very flexible; if called with scalars, it does the obvious:

```
> rep(5,3)
[1] 5 5 5
```

with a vector and a scalar, it creates a new vector by repeating the old one:

```
> y = 3:7
> rep(y,3)
[1] 3 4 5 6 7 3 4 5 6 7 3 4 5 6 7
```

Finally, if you call `rep` with two equal length vectors, it repeats the elements of the first vector as many times as the corresponding element of the second vector:

```
> rep(1:4,c(2,3,3,4))
[1] 1 1 2 2 2 3 3 3 4 4 4 4
```

One surprising thing about vectors in R is that many times it will carry out an operation with two vectors that aren't the same size by simply recycling the values in the shorter vector. For example, suppose we try to add a vector with four numbers to one with just two numbers:

```
> c(1,2,3,4) + c(1,2)
[1] 2 4 4 6
```

Notice that, for the two last elements, it simply recycled the 1 and 2 from the second vector to perform the addition. R will be silent when things like this happen, but if the length of the larger vector isn't an even multiple of the length of the smaller vector, R will print a warning:

```
> c(1,2,3,4) + c(1,2,3)
[1] 2 4 6 5
Warning message:
longer object length is not a multiple of shorter object
length in: c(1, 2, 3, 4) + c(1, 2, 3)
```

It's possible to provide names for the elements of a vector. Suppose we were working with purchases in a number of states, and we needed to know the sales tax rate for a given state. We could create a named vector as follows:

```
> taxrate = c(AL=4,CA=7.25,IL=6.25,KS=5.3,NY=4.25,TN=7)
> taxrate
AL    CA    IL    KS    NY    TN
4.00 7.25 6.25 5.30 4.25 7.00
```

To add names to a vector after the fact, you can use the `names` function:

```
> taxrate = c(4,7.25,6.25,5.3,4.25,7)
> taxrate
[1] 4.00 7.25 6.25 5.30 4.25 7.00
> names(taxrate) = c('AL','CA','IL','KS','NY','TN')
> taxrate
AL    CA    IL    KS    NY    TN
4.00 7.25 6.25 5.30 4.25 7.00
```

If you have a named vector, you can access the elements with either numeric subscripts or by using the name of the element you want:

```
> taxrate[3]
IL
```

6.25

```
> taxrate['KS']
  KS
5.3
```

One of the most powerful tools in R is the ability to use logical expressions to extract or modify elements in the way that numeric subscripts are traditionally used. While there are (of course) many cases where we're interested in accessing information based on the numeric or character subscript of an object, being able to use logical expressions gives us a much wider choice in the way we can study our data. For example, suppose we want to find all of observations in `taxrate` with a taxrate less than 6. First, let's look at the result of just asking whether `taxrate` is less than 6:

```
> taxrate < 6
  AL    CA    IL    KS    NY    TN
TRUE FALSE FALSE  TRUE  TRUE FALSE
```

The result is a logical vector of the same length as the vector we were asking about. If we use such a vector to extract values from the `taxrate` vector, it will give us all the ones that correspond to `TRUE` values, discarding the ones that correspond to `FALSE`.

```
> taxrate[taxrate > 6]
  CA    IL    TN
7.25 6.25 7.00
```

Another important use of logical variables is counting the number of elements of a vector meet a particular condition. When a logical vector is passed to the `sum` function, `TRUE`s count as one and `FALSE`s count as 0. So we can count the number of `TRUE`s in a logical expression by passing it to `sum`:

```
> sum(taxrate > 6)
[1] 3
```

This tells us three observations in the `taxrate` vector had values greater than 6.

As another example, suppose we want to find which of the states we have information about has the highest sales tax. The `max` function will find the largest value in a vector. (Once again, note that we don't have to worry about the size of the vector or looping over individual elements.)

```
> max(taxrate)
[1] 7.25
```

We can find the state which has the highest tax rate as follows:

```
> taxrate[taxrate == max(taxrate)]
  CA
7.25
```

Notice that we use two equal signs when testing for equality, and one equal sign when we are assigning an object to a variable.

Another useful tool for these kinds of queries is the `which` function. It converts between logical subscripts and numeric ones. For example, if we wanted to know the index of the element in the `taxrate` vector that was the biggest, we could use:

```
> which(taxrate == max(taxrate))
CA
2
```

In fact, this is such a common operation that R provides two functions called `which.min` and `which.max` which will return the index of the minimum or maximum element of a vector:

```
> which.max(taxrate)
CA
2
```

While it's certainly not necessary to examine every function that we use in R, it might be interesting to see what `which.max` is doing beyond our straight-forward solution. As always, we can type the name of the function to see what it does:

```
> which.max
function (x)
.Internal(which.max(x))
<environment: namespace:base>
```

`.Internal` means that the function that actually finds the index of the maximum value is compiled inside of R. Generally functions like this will be faster than pure R solutions like the first one we tried. We can use the `system.time` function to see how much faster `which.max` will be. Because functions use the equal sign (=) to name their arguments, we'll use the alternative assignment operator, <- in our call to `system.time`:

```
> system.time(one <- which(taxrate == max(taxrate)))
 user  system elapsed
      0        0        0
```

It's not surprising to see a time of 0 when operating on such a small vector. It doesn't mean that it required no time to do the operation, just that the amount of time it required was smaller than the granularity of the system clock. (The granularity of the clock is simply the smallest interval of time that can be measured by the computer.) To get a good comparison, we'll need to create a larger vector. To do this, we'll use the `rnorm` function, which generates random numbers from the normal distribution with mean 0 and standard deviation 1. To get times that we can trust, I'll use a vector with 10 million elements:

```
> x = rnorm(10000000)
> system.time(one <- which(x == max(x)))
  user  system elapsed
 0.276   0.016   0.292
> system.time(two <- which.max(x))
  user  system elapsed
 0.068   0.000   0.071
```

While the pure R solution seems pretty fast (0.292 seconds to find the index of the largest element in a vector of 10 million numbers), the compiled (internal) version is actually around 4 times faster!

Of course none of this matters if they don't get the same answers:

```
> one
[1] 8232773
> two
[1] 8232773
```

The two methods do agree.

If you try this example on your own computer, you'll see a different value for the index of the maximum. This is due to the way random numbers are generated in R, and we'll see how to take more control of this later in the semester.

## 2.3 Modes and Classes

It was mentioned earlier that all the elements of a vector must be of the same mode. To see the mode of an object, you can use the `mode` function. What happens if we try to combine objects of different modes using the `c` function? The answer is that R will find a common mode that can accomodate all the objects, resulting in the mode of some of the objects changing. For example, let's try combining some numbers and some character strings:

```
> both = c('dog', 3, 'cat', 'mouse', 7, 12, 9, 'chicken')
> both
[1] "dog"      "3"       "cat"      "mouse"    "7"       "12"      "9"
[8] "chicken"
> mode(both)
[1] "character"
```

You can see that the numbers have been changed to characters because they are now displayed surrounded by quotes. They also will no longer behave like numbers:

```
> both[2] + both[5]
Error in both[2] + both[5] : non-numeric argument to binary operator
```

The error message means that the two values can no longer be added. If you really need to treat character strings like numbers, you can use the `as.numeric` function:

```
> as.numeric(both[2]) + as.numeric(both[5])
[1] 10
```

Of course, the best thing is to avoid combining objects of different modes with the `c` function. We'll see later that R provides an object known as a list that can store different types of objects without having to change their modes.

## 2.4 Reading Vectors

Once you start working with larger amounts of data, it becomes very tedious to enter data into the `c` function, especially considering the need to put quotes around character values and commas between values. To read data from a file or from the terminal without the need for quotes and commas, you can use the `scan` function. To read from a file (or a URL), pass it a quoted string with the name of the file or URL you wish to read; to read from the terminal, call `scan()` with no arguments, and enter a completely blank line when you're done entering your data. Additionally, on Windows or Mac OS X, you can substitute a call to the `file.choose()` function for the quoted string with the file name, and you'll be presented with the familiar file chooser used by most programs on those platforms.

Suppose there's a file called `numbers` in your working directory. (You can get your working directory by calling the `getwd()` function, or set it using the `setwd` function or **File -> Change dir** selection in the R console.) Let's say the contents of this file looks like this:

```
12 7  
9 8 14 10  
17
```

The `scan` function can be used to read these numbers as follows:

```
> nums = scan('numbers')  
Read 7 items  
> nums  
[1] 12 7 9 8 14 10 17
```

The optional `what=` argument to `scan` can be used to read vectors of character or logical values, but remember a vector can only hold objects all of which are of the same mode.

## 2.5 Missing Values

No matter how carefully we collect our data, there will always be situations where we don't know the value of a particular variable. For example, we might conduct a survey where we ask people 10 questions, and occasionally we forget to ask one, or people don't know the proper answer. We don't want values like this to enter into calculations, but we can't just eliminate them because then observations that have missing values won't "fit in" with the rest of the data.

In R, missing values are represented by the string `NA`. For example, suppose we have a vector of 10 values, but the fourth one is missing. I can enter a missing value by passing `NA` to the `c` function just as if it was a number (no quotes needed):

```
x = c(1,4,7,NA,12,19,15,21,20)
```

R will also recognize the unquoted string `NA` as a missing value when data is read from a file or URL.

Missing values are different from other values in R in two ways:

1. Any computation involving a missing value will return a missing value.
2. Unlike other quantities in R, we can't directly test to see if something is equal to a missing value with the equality operator (`==`). We must use the builtin `is.na` function, which will return `TRUE` if a value is missing and `FALSE` otherwise.

Here are some simple R statements that illustrate these points:

```
> x = c(1,4,7,NA,12,19,15,21,20)  
> mean(x)  
[1] NA  
> x == NA  
[1] NA NA NA NA NA NA NA NA NA
```

Fortunately, these problems are fairly easy to solve. In the first case, many functions (like `mean`, `min`, `max`, `sd`, `quantile`, etc.) accept an `na.rm=TRUE` argument, that tells the function to remove any missing values before performing the computation:

```
> mean(x,na.rm=TRUE)
[1] 12.375
```

In the second case, we just need to remember to always use `is.na` whenever we are testing to see if a value is a missing value.

```
> is.na(x)
[1] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
```

By combining a call to `is.na` to the logical “not” operator (!) we can filter out missing values in cases where no `na.rm=` argument is available:

```
> x[!is.na(x)]
[1] 1 4 7 12 19 15 21 20
```

## 2.6 Matrices

A very common way of storing data is in a matrix, which is basically a two-way generalization of a vector. Instead of a single index, we can use two indexes, one representing a row and the second representing a column. The `matrix` function takes a vector and makes it into a matrix in a column-wise fashion. For example,

```
> mymat = matrix(1:12,4,3)
> mymat
 [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

The last two arguments to `matrix` tell it the number of rows and columns the matrix should have. If you used a named argument, you can specify just one dimension, and R will figure out the other:

```
> mymat = matrix(1:12,ncol=3)
> mymat
 [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

To create a matrix by rows instead of by columns, the `byrow=TRUE` argument can be used:

```
> mymat = matrix(1:12, ncol=3, byrow=TRUE)
> mymat
 [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

When data is being read from a file, you can simply imbed a call to scan into a call to matrix. Suppose we have a file called `matrix.dat` with the following contents:

```
7 12 19 4
18 7 12 3
9 5 8 42
```

We could create a  $3 \times 4$  matrix, read in by rows, with the following command:

```
matrix(scan('matrix.dat'), nrow=3, byrow=TRUE)
```

To access a single element of a matrix, we need to specify both the row and the column we're interested in. Consider the following matrix, containing the numbers from 1 to 10:

```
> m = matrix(1:10, 5, 2)
> m
 [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

Now suppose we want the element in row 4 and column 1:

```
> m[4,1]
[1] 4
```

If we leave out either one of the subscripts, we'll get the entire row or column of the matrix, depending on which subscript we leave out:

```
> m[4,]
[1] 4 9
> m[,1]
[1] 1 2 3 4 5
```

## 2.7 Data Frames

One shortcoming of vectors and matrices is that they can only hold one mode of data; they don't allow us to mix, say, numbers and character strings. If we try to do so, it will change the mode of the other elements in the vector to conform. For example:

```
> c(12,9,"dog",7,5)
[1] "12"  "9"   "dog"  "7"   "5"
```

Notice that the numbers got changed to character values so that the vector could accomodate all the elements we passed to the `c` function. In R, a special object known as a data frame resolves this problem. A data frame is like a matrix in that it represents a rectangular array of data, but each column in a data frame can be of a different mode, allowing numbers, character strings and logical values to coincide in a single object in their original forms. Since most interesting data problems involve a mixture of character variables and numeric variables, data frames are usually the best way to store information in R. (It should be mentioned that if you're dealing with data of a single mode, a matrix may be more efficient than a data frame.) Data frames correspond to the traditional “observations and variables” model that most statistical software uses, and they are also similar to database tables. Each row of a data frame represents an observation; the elements in a given row represent information about that observation. Each column, taken as a whole, has all the information about a particular variable for the data set.

For small datasets, you can enter each of the columns (variables) of your data frame using the `data.frame` function. For example, let's extend our temperature example by creating a data frame that has the day of the month, the minimum temperature and the maximum temperature:

```
> temps = data.frame(day=1:10,
+                      min = c(50.7,52.8,48.6,53.0,49.9,47.9,54.1,47.6,43.6,45.5),
+                      max = c(59.5,55.7,57.3,71.5,69.8,68.8,67.5,66.0,66.1,61.7))
> head(temps)
  day  min  max
1   1 50.7 59.5
2   2 52.8 55.7
3   3 48.6 57.3
4   4 53.0 71.5
5   5 49.9 69.8
6   6 47.9 68.8
```

Note that the names we used when we created the data frame are displayed with the data. (You can add names after the fact with the `names` function.) Also, instead of typing the name `temps` to see the data frame, we used a call the the `head` function instead. This will show me just the first six observations (by default) of the data frame, and is very handy to check to make sure a large `data.frame` really looks the way you think. (There's a function called `tail` that shows the last lines in an object as well.)

If we try to look at the class or mode of a data frame, it's not that informative:

```
> class(temp)
[1] "data.frame"
> mode(temp)
[1] "list"
```

We'll see the same results for every data frame we use. To look at the modes of the individual columns of a data frame, we can use the `sapply` function. This function simplifies operations that would require loops in other languages, and automatically returns the appropriate results for the operation it performs. To use `sapply` on a data frame, pass the data frame as the first argument to `sapply`, and the function you wish to use as the second argument. So to find the modes of the individual columns of the `temp` data frame, we could use

```
> sapply(temp, mode)
      date      min      maximum
"numeric" "numeric" "numeric"
```

Notice that `sapply` even labeled the result with the name of each column.

Suppose we want to concentrate on the maximum daily temperature (which we've called `max` in our data frame) among the days recorded. There are several ways we can refer to the columns of a data frame:

1. Probably the easiest way to refer to this column is to use a special notation that eliminates the need to put quotes around the variable names (unless they contain blanks or other special characters). Separate the data frame name from the variable name with a dollar sign (\$):

```
> temp$max
[1] 59.5 55.7 57.3 71.5 69.8 68.8 67.5 66.0 66.1 61.7
```

2. We can treat the data frame like it was a matrix. Since the maximum temperature is in the third column, we could say

```
> temp[, 3]
[1] 59.5 55.7 57.3 71.5 69.8 68.8 67.5 66.0 66.1 61.7
```

3. Since we named the columns of `temp` we can use a character subscript:

```
> temp[, "max"]
[1] 59.5 55.7 57.3 71.5 69.8 68.8 67.5 66.0 66.1 61.7
```

4. When you use a single subscript with a data frame, it refers to a data frame consisting of just that column. R also provides a special subscripting method (double brackets) to extract the actual data (in this case a vector) from the data frame:

```
> temps['max']
  max
1 59.5
2 55.7
3 57.3
4 71.5
5 69.8
6 68.8
7 67.5
8 66.0
9 66.1
10 61.7
> temps[['max']]
[1] 59.5 55.7 57.3 71.5 69.8 68.8 67.5 66.0 66.1 61.7
```

Notice that this second form is identical to `temps$max`. We could also use the equivalent numerical subscript (in this case 3) with single or double brackets.

## 2.8 More on Data Frames

1. Notice that if you want to extract more than one column of a data frame, you need to use single brackets, not double:

```
> temps[c('min','max')]
   min   max
1 50.7 59.5
2 52.8 55.7
3 48.6 57.3
4 53.0 71.5
5 49.9 69.8
6 47.9 68.8
7 54.1 67.5
8 47.6 66.0
9 43.6 66.1
10 45.5 61.7
> temps[[c('min','max')]]
Error in .subset2(x, i, exact = exact) : subscript out of bounds
```

2. If you want to work with a data frame without having to constantly retype the data frame's name, you can use the `with` function. Suppose we want to convert our minimum and maximum temperatures to centigrade, and then calculate the difference between them. Using `with`, we can write:

```
> with(temps,5/9*(max-32) - 5/9*(min-32))
[1] 4.888889 1.611111 4.833333 10.277778 11.055556 11.611111 7.444444
[8] 10.222222 12.500000 9.000000
```

which may be more convenient than typing out the data frame name repeatedly:

```
> 5/9*(temps$max-32) - 5/9*(temps$min-32)
[1] 4.888889 1.611111 4.833333 10.277778 11.055556 11.611111 7.444444
[8] 10.222222 12.500000 9.000000
```

3. Finally, if the goal is to add one or more new columns to a data frame, you can combine a few operations into one using the `transform` function. The first argument to `transform` is the name of the data frame that will be used to construct the new columns. The remaining arguments to `transform` are name/value pairs describing the new columns. For example, suppose we wanted to create a new variable in the `temps` data frame called `range`, representing the difference between the `min` and `max` values for each day. We could use `transform` as follows:

```

> temps = transform(temps, range = max - min)
> head(temps)
  day  min  max range
1 1 50.7 59.5 8.8
2 2 52.8 55.7 2.9
3 3 48.6 57.3 8.7
4 4 53.0 71.5 18.5
5 5 49.9 69.8 19.9
6 6 47.9 68.8 20.9

```

As can be seen, `transform` returns a new data frame like the original one, but with one or more new columns added.

## 2.9 Reading Data Frames from Files and URLs

While creating a data frame the way we just did is very handy for quick examples, it's actually pretty rare to enter a data frame in that way; usually we'll be reading data from a file or possibly a URL. In these cases, the `read.table` function (or one of its' closely related variations described below) can be used. `read.table` tries to be clever about figuring out what type of data you'll be using, and automatically determines how each column of the data frame should be stored. One problem with this scheme is has to do with a special type of variable known as a factor. A factor in R is a variable that is stored as an integer, but displayed as a character string. By default, `read.table` will automatically turn all the character variables that it reads into factors. You can recognize factors by using either the `is.factor` function or by examining the object's class, using the `class` function. Factors are very useful for storing large data sets compactly, as well as for statistical modeling and other tasks, but when you're first working with R they'll most likely just get in the way. To avoid `read.table` from doing any factor conversions, pass the `stringsAsFactors=FALSE` argument as shown in the examples below.

By default, R expects there to be at least one space or tab between each of the data values in your input file; if you're using a different character to separate your values, you can specify it with the `sep=` argument. Two special versions of `read.table` are provided to handle two common cases: `read.csv` for files where the data is separated by commas, and `read.delim` when a tab character is used to separate values. On the other hand, if the variables in your input data occupy the same columns for every line in the file, the `read.fwf` can be used to turn your data into a data frame.

If the first line of your input file contains the names of the variables in your data separated with the same separator used for the rest of the data, you can pass the `header=TRUE` argument to `read.table` and its variants, and the variables (columns) of your data frame will be named accordingly. Otherwise, names like V1, V2, etc. will be used.

As an example of how to read data into a data frame, the URL  
<http://www.stat.berkeley.edu/classes/s133/data/world.txt> contains information about liter-

acy, gross domestic product, income and military expenditures for about 150 countries. Here are the first few lines of the file:

```
country,gdp,income,literacy,military
Albania,4500,4937,98.7,56500000
Algeria,5900,6799,69.8,2.48e+09
Angola,1900,2457,66.8,183580000
Argentina,11200,12468,97.2,4.3e+09
Armenia,3900,3806,99.4,1.35e+08
```

(You can use your favorite browser to examine a file like this, or you can use R's `download.file` and `file.edit` functions to download a copy to your computer and examine it locally.)

Since the values are separated by commas, and the variable names can be found in the first line of the file, we can read the data into a data frame as follows:

```
world = read.csv('http://www.stat.berkeley.edu/classes/s133/data/world.txt', header=TRUE, stringsAsFactors=FALSE)
```

Now that we've created the data frame, we need to look at some ways to understand what our data is like. The class and mode of objects in R is very important, but if we query them for our data frame, they're not very interesting:

```
> mode(world)
[1] "list"
> class(world)
[1] "data.frame"
```

Note that a data frame is also a list. We'll look at lists in more detail later. As we've seen, we can use the `sapply` function to see the modes of the individual columns. This function will apply a function to each element of a list; for a data frame these elements represent the columns (variables), so it will do exactly what we want:

```
> sapply(world, mode)
  country      gdp     income   literacy   military
"character"  "numeric"  "numeric"  "numeric"  "numeric"
> sapply(world, class)
  country      gdp     income   literacy   military
"character"  "integer"  "integer"  "numeric"  "numeric"
```

You might want to experiment with `sapply` using other functions to get familiar with some strategies for dealing with data frames.

You can always view the names of the variables in a data frame by using the `names` function, and the size (number of observations and number of variables) using the `dim` function:

```
> names(world)
[1] "country"  "gdp"       "income"    "literacy"   "military"
> dim(world)
[1] 154      5
```

Suppose we want to see the country for which military spending is the highest. We can use the `which.max` function that we used before but extra care is needed to make sure we get the piece of the data frame we want. Since each country occupies one row in the data frame, we want all of the columns in that row, and we can leave the second index of the data frame blank:

```
>
> world[which.max(world$military),]
  country   gdp income literacy  military
142     USA 37800 39496    99.9 3.707e+11
```

The 142 at the beginning of the line is the row number of the observation. If you'd like to use a more informative label for the rows, look at the `row.names=` argument in `read.table` and `data.frame`, or use the assignment form of the `row.names` function if the data frame already exists.

These types of queries, where we want to find observations from a data frame that have certain properties, are so common that R provides a function called `subset` to make them easier and more readable. The `subset` function requires two arguments: the first is a data frame, and the second is the condition that you want to use to create the subset. An optional third argument called `select=` allows you to specify which of the variables in the data frame you're interested in. The return value from `subset` is a data frame, so you can use it anywhere that you'd normally use a data frame. A very attractive feature of `subset` is that you can refer to the columns of a data frame directly in the second or third arguments; you don't need to keep retyping the data frame's name, or surround all the variable names with quotes. Suppose we want to find those countries whose literacy rate is below 20%. We could use the `subset` function like this:

```
> subset(world,literacy < 20)
  country   gdp income literacy  military
22 Burkina Faso 1100 1258    12.8 64200000
88      Mali  900 1024    19.0 22400000
102     Niger  800  865    14.4 33300000
```

One other nice feature of the `select=` argument is that it converts variable names to numbers before extracting the requested variables, so you can use "ranges" of variable names to specify contiguous columns in a data frame. For example, here are the names for the `world` data frame:

```
> names(world)
[1] "country"   "gdp"        "income"      "literacy"    "military"
```

To create a data frame with just the last three variables, we could use

```
> subset(world,select=income:military)
```

If we were interested in a particular variable, it would be useful to reorder the rows of our data frame so that they were arranged in descending or ascending order of that variable. It's easy enough to sort a variable in R; using `literacy` as an example, we simply call the `sort` routine:

```
> sort(world$literacy)
[1] 12.8 14.4 19.0 25.5 29.6 33.6 39.3 39.6 41.0 41.1 41.5 46.5 47.0 48.6 48.6
[16] 48.7 49.0 50.7 51.2 51.9 53.0 54.1 55.6 56.2 56.7 57.3 58.9 59.0 61.0 64.0
[31] 64.1 65.5 66.8 66.8 67.9 67.9 68.7 68.9 69.1 69.4 69.8 70.6 71.0 73.6 73.6
[46] 74.3 74.4 75.7 76.7 76.9 77.0 77.3 78.9 79.2 79.4 79.7 80.0 81.4 81.7 82.4
[61] 82.8 82.9 82.9 84.2 84.3 85.0 86.5 86.5 87.6 87.7 87.7 87.7 87.9 87.9 88.0
[76] 88.3 88.4 88.7 89.2 89.9 90.0 90.3 90.3 90.4 90.9 91.0 91.0 91.6 91.9 91.9
[91] 92.5 92.5 92.6 92.6 92.7 92.9 93.0 94.2 94.6 95.7 95.8 96.2 96.5 96.8 96.8
[106] 96.9 96.9 97.2 97.2 97.3 97.7 97.7 97.8 98.1 98.2 98.5 98.5 98.7 98.7 98.8
[121] 98.8 99.3 99.3 99.4 99.4 99.5 99.5 99.6 99.6 99.6 99.7 99.7 99.7 99.8 99.9
[136] 99.9 99.9 99.9 99.9 99.9 99.9 99.9 99.9 99.9 99.9 99.9 99.9 99.9 99.9 99.9
[151] 99.9 99.9 99.9 99.9
```

To reorder the rows of a data frame to correspond to the sorted order of one of the variables in the data frame, the `order` function can be used. This function returns a set of indices which are in the proper order to rearrange the data frame appropriately. (Perhaps the easiest way to understand what the `order` function is to realize that `x[order(x)]` is the same as `sort(x)`.)

```
> sworld = world[order(world$literacy),]
> head(sworld)
  country   gdp income literacy military
22 Burkina Faso 1100    1258     12.8 64200000
103      Niger  800     865     14.4 33300000
89       Mali  900    1024     19.0 22400000
29       Chad 1200    1555     25.5 101300000
121 Sierra Leone  500     842     29.6 13200000
14       Benin 1100    1094     33.6 96500000
```

To sort by descending values of a variable, pass the `decreasing=TRUE` argument to `sort` or `order`.

When you're first working with a data frame, it can be helpful to get some preliminary information about the variables. One easy way to do this is to pass the data frame to the `summary` function, which understands what a data frame is, and will give separate summaries for each of the variables:

```
> summary(world)
  country           gdp          income        literacy
Length:154      Min.   : 500   Min.   : 569   Min.   :12.80
  Class :character  1st Qu.: 1825  1st Qu.: 2176  1st Qu.:69.17
```

```

Mode : character   Median : 4900   Median : 5930   Median :88.55
                  Mean   : 9031   Mean   :10319   Mean   :81.05
                  3rd Qu.:11700  3rd Qu.:15066  3rd Qu.:98.42
                  Max.   :55100   Max.   :63609   Max.   :99.90
                               NA's    :      1

military
Min.   :6.500e+06
1st Qu.:5.655e+07
Median :2.436e+08
Mean   :5.645e+09
3rd Qu.:1.754e+09
Max.   :3.707e+11

```

Another useful way to view the properties of a variable is with the `stem` function, which produces a text-base stem-and-leaf diagram. Each observation for the variable is represented by a number in the diagram showing that observation's value:

```
> stem(world$gdp)
```

The decimal point is 4 digit(s) to the right of the |

```

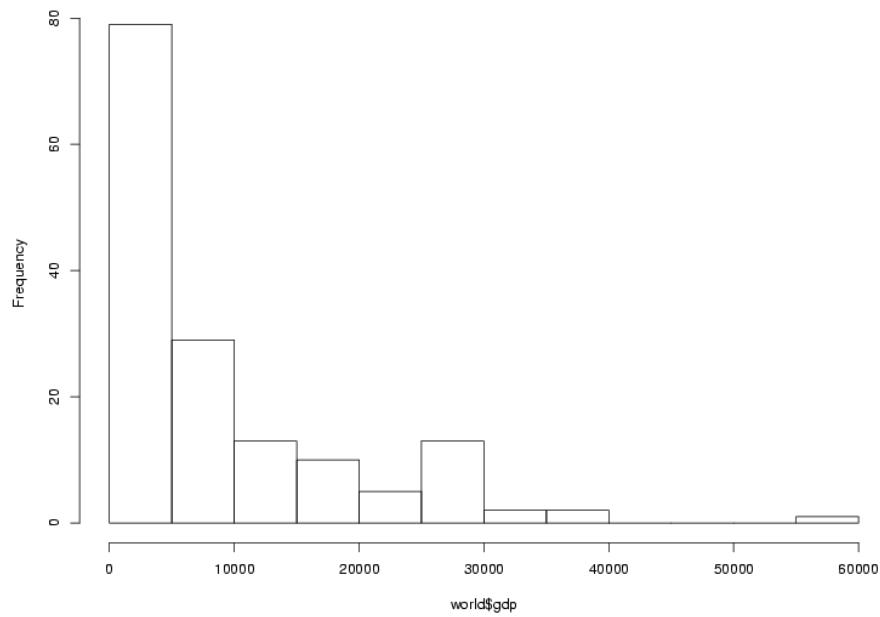
0 | 11111111111111111111111111111112222222222222223333333344444444
0 | 55555555556666666667777778889999
1 | 00011111223334
1 | 66788889
2 | 0022234
2 | 7778888999
3 | 00013
3 | 88
4 |
4 |
5 |
5 | 5

```

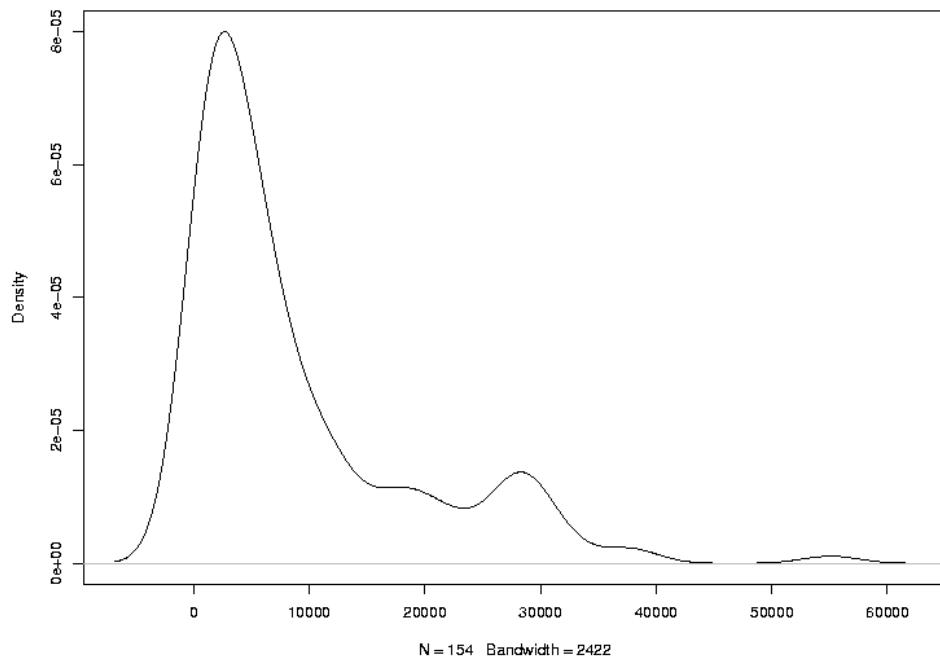
Graphical techniques are often useful when exploring a data frame. While we'll look at graphics in more detail later, the functions `boxplot`, `hist`, and `plot` combined with the `density` function are often good choices. Here are examples:

```
> boxplot(world$gdp,main='Boxplot of GDP')
> hist(world$gdp,main='Histogram of GDP')
> plot(density(world$gdp),main='Density of GDP')
```

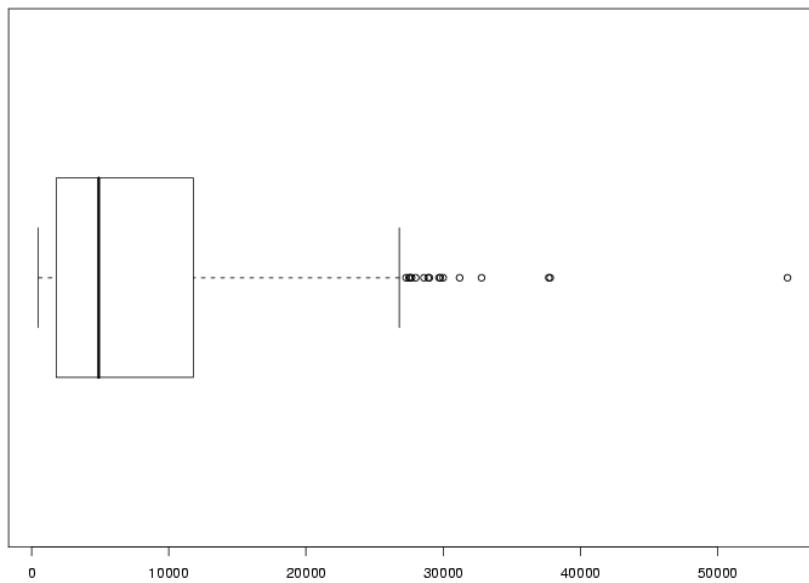
Histogram of GDP



Density of GDP



**Boxplot of GDP**



## 2.10 Working with Multiple Data Frames

Suppose we want to add some additional information to our data frame, for example the continents in which the countries can be found. Very often we have information from different sources and it's very important to combine it correctly. The URL <http://www.stat.berkeley.edu/s133/data/conts.txt> contains the information about the continents. Here are the first few lines of that file:

```
country,cont
Afghanistan,AS
Albania,EU
Algeria,AF
American Samoa,OC
Andorra,EU
```

In R, the `merge` function allows you to combine two data frames based on the value of a variable that's common to both of them. The new data frame will have all of the variables from both of the original data frames. First, we'll read in the continent values into a data frame called `conts`:

```
conts = read.csv('http://www.stat.berkeley.edu/classes/s133/data/conts.txt',na.string='.',stringAsFactors=F)
```

To merge two data frames, we simply need to tell the `merge` function which variable(s) the two data frames have in common, in this case `country`:

```
world1 = merge(world,conts,by='country')
```

Notice that we pass the name of the variable that we want to merge by, not the actual value of the variable itself. The first few records of the merged data set look like this:

```
> head(world1)
  country    gdp income literacy   military cont
1  Albania  4500    4937     98.7 5.6500e+07   EU
2  Algeria  5900    6799     69.8 2.4800e+09   AF
3  Angola   1900    2457     66.8 1.8358e+08   AF
4 Argentina 11200   12468     97.2 4.3000e+09   SA
5 Armenia   3900    3806     99.4 1.3500e+08   AS
6 Australia 28900   29893     99.9 1.6650e+10   OC
```

We've already seen how to count specific conditions, like how many countries in our data frame are in Europe:

```
> sum(world1$cont == 'EU')
[1] 34
```

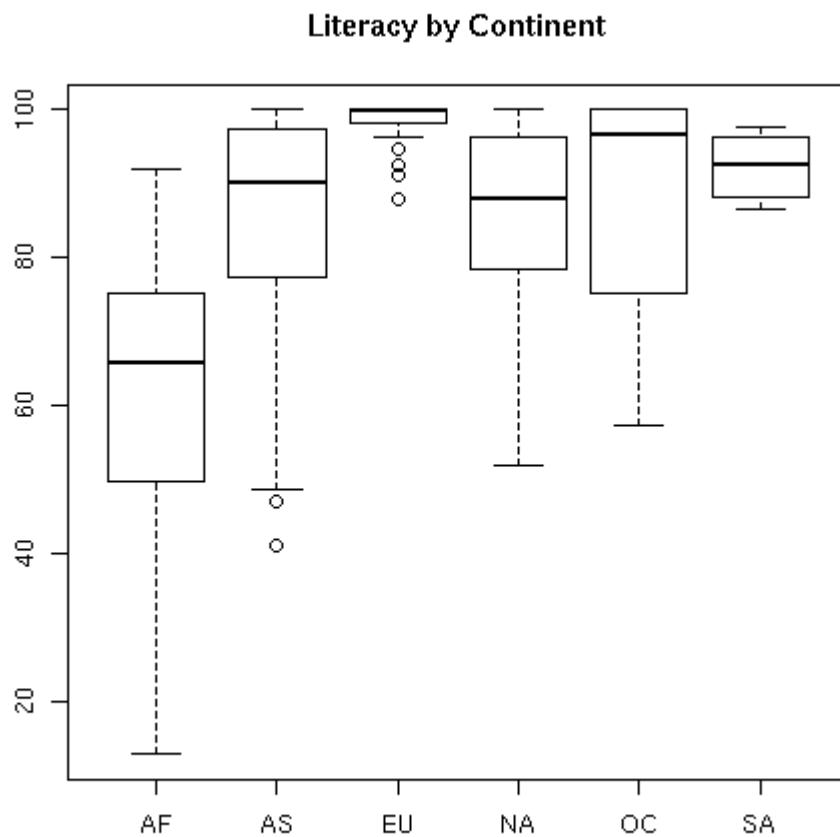
It would be tedious to have to repeat this for each of the continents. Instead, we can use the `table` function:

```
> table(world1$cont)
```

Continent	Count
AF	47
AS	41
EU	34
NA	15
OC	4
SA	12

We can now examine the variables taking into account the continent that they're in. For example, suppose we wanted to view the literacy rates of countries in the different continents. We can produce side-by-side boxplots like this:

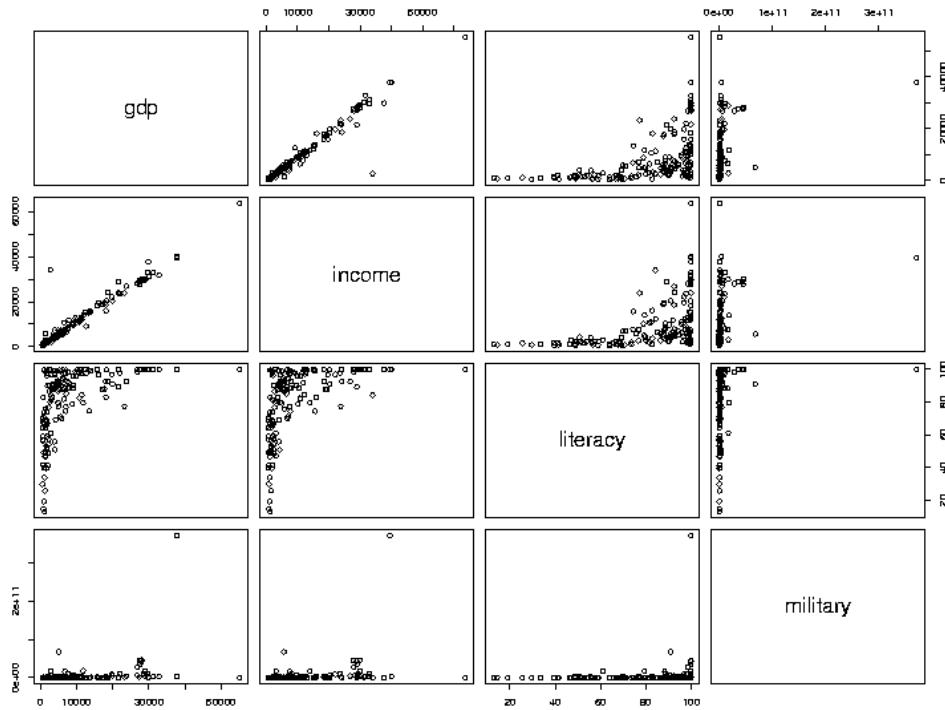
```
> boxplot(split(world1$literacy,world1$cont),main='Literacy by Continent')
```



Now let's concentrate on plots involving two variables. It may be surprising, but R is smart enough to know how to “plot” a data frame. It actually calls the `pairs` function, which will produce what's called a scatterplot matrix. This is a display with many little graphs showing the relationships between each pair of variables in the data frame. Before we can call `plot`, we need to remove the character variables (`country` and `cont`) from the data using negative subscripts:

```
> plot(world1[,-c(1,6)])
```

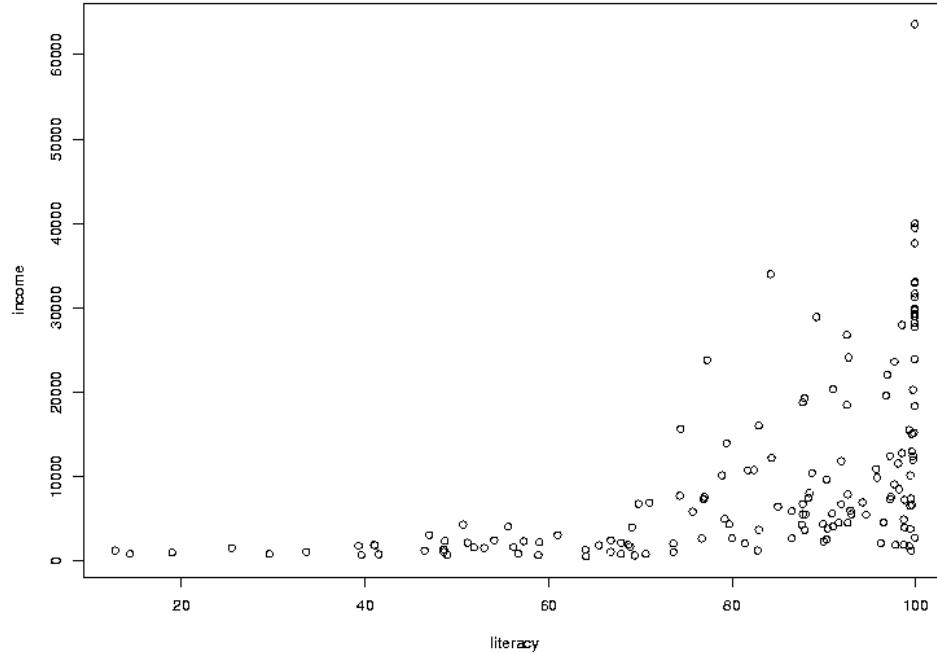
The resulting plot looks like this:



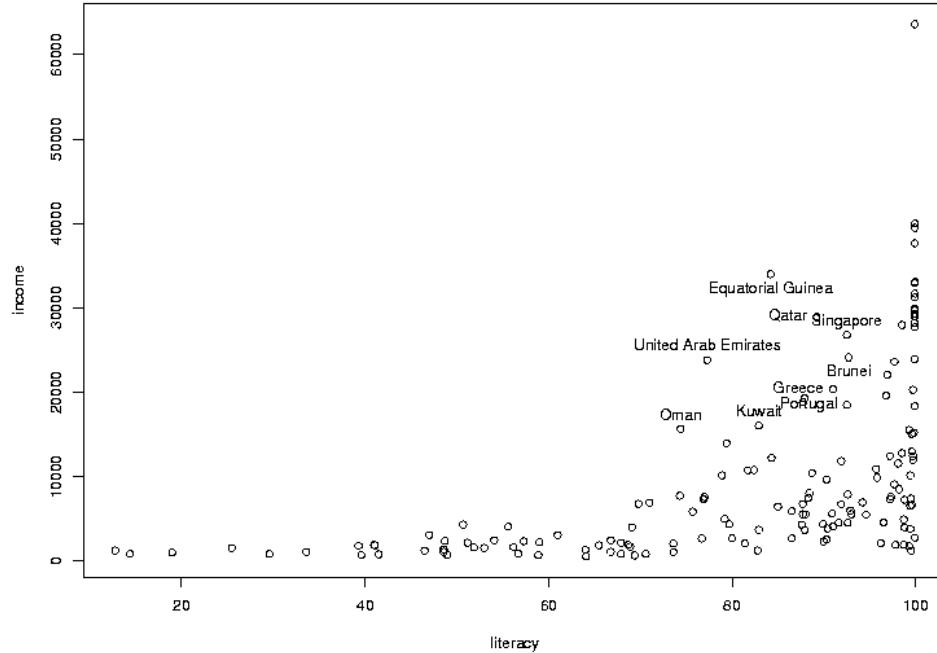
As we'd expect, `gdp` (Gross Domestic Product) and `income` seem to have a very consistent relationship. The relation between `literacy` and `income` appears to be interesting, so we'll examine it in more detail, by making a separate plot for it:

```
> with(world,plot(literacy,income))
```

The first variable we pass to plot (`literacy` in this example) will be used for the x-axis, and the second (`income`) will be used on the y-axis. The plot looks like this:



In many cases, the most interesting points on a graph are the ones that don't follow the usual relationships. In this case, there are a few points where the income is a bit higher than we'd expect based on the other countries, considering the rate of literacy. To see which countries they represent, we can use the `identify` function. You call `identify` with the same arguments as you passed to `plot`; then when you click on a point on the graph with the left mouse button, its row number will be printed on the graph. It's usually helpful to have more than just the row number, so `identify` is usually called with a `labels=` argument. In this case, the obvious choice is the country name. The way to stop identifying points depends on your operating system; on Windows, right click on the plot and choose "Stop"; on Unix/Linux click on the plot window with the middle button. Here's the previous graph after some of the outlier points are identified:



## 2.11 Adding Color to Plots

Color is often referred to as the third dimension of a 2-dimensional plot, because it allows us to add extra information to an ordinary scatterplot. Consider the graph of literacy and income. By examining boxplots, we can see that there are differences among the distributions of income (and literacy) for the different continents, and it would be nice to display some of that information on a scatterplot. This is one situation where factors come in very handy. Since factors are stored internally as numbers (starting at 1 and going up to the number of unique levels of the factor), it's very easy to assign different observations different colors based on the value of a factor variable.

To illustrate, let's replot the income vs. literacy graph, but this time we'll convert the continent into a factor and use it to decide on the color of the points that will be used for each country. First, consider the `world1` data frame. In that data frame, the continent is stored in the column (variable) called `cont`. We convert this variable to a factor with the `factor` function. First, let's look at the mode and class of the variable before we convert it to a factor:

```
> mode(world1$cont)
[1] "character"
> class(world1$cont)
[1] "character"
> world1$cont = factor(world1$cont)
```

In many situations, the `cont` variable will behave the same as it did when it was a simple character variable, but notice that its mode and class have changed:

```
> mode(world1$cont)
[1] "numeric"
> class(world1$cont)
[1] "factor"
```

Having made `cont` into a factor, we need to choose some colors to represent the different continents. There are a few ways to tell R what colors you want to use. The easiest is to just use a color's name. Most colors you think of will work, but you can run the `colors` function without an argument to see the official list. You can also use the method that's commonly used by web designers, where colors are specified as a pound sign (#) followed by 3 sets of hexadecimal digits providing the levels of red, green and blue, respectively. Using this scheme, red is represented as '#FF0000', green as '#00FF00', and blue as '#0000FF'. To see how many unique values of `cont` there are, we can use the `levels` function, since it's a factor. (For non-factors, the `unique` function is available, but it may give the levels in an unexpected order.)

```
> levels(world1$cont)
[1] "AF" "AS" "EU" "NA" "OC" "SA"
```

There are six levels. The first step is to create a vector of color values:

```
mycolors = c('red','yellow','blue','green','orange','violet')
```

To make the best possible graph, you should probably be more careful when choosing the colors, but this will serve as a simple example.

Now, when we make the scatterplot, we add an additional argument, `col=`, which is a vector of the same length as the number of pairs of points that we're plotting – the color in each position corresponds to the color that will be used to draw that point on the graph. Probably the easiest way to do that is to use the value of the factor `cont` as a subscript to the `mycolors` vector that we created earlier. (If you don't see why this does what we want, please take a look at the result of `mycolors[world1$cont]`).

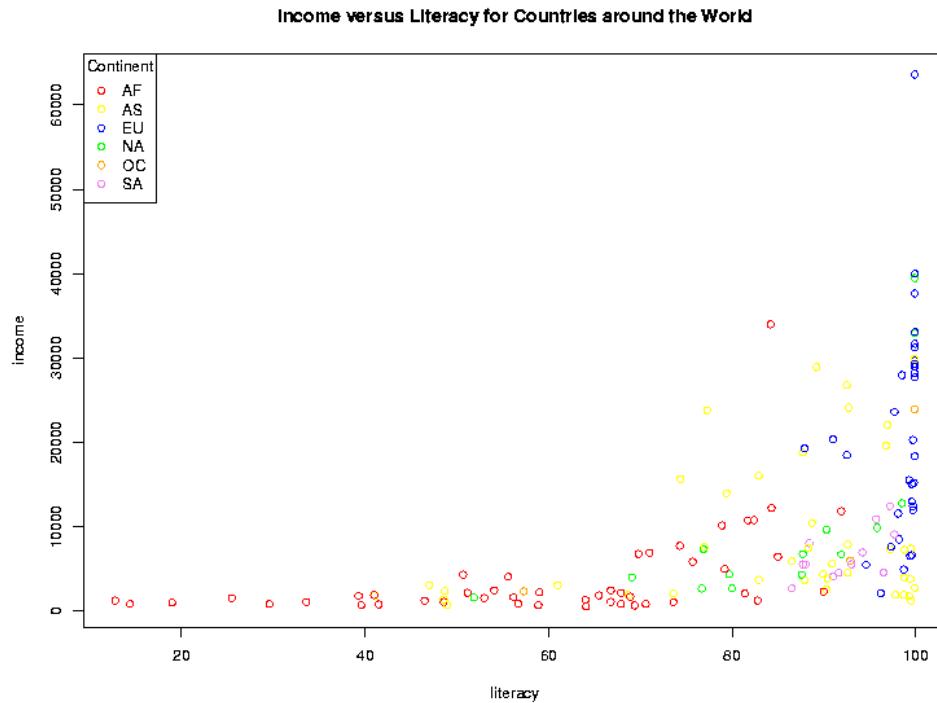
```
with(world1, plot(literacy, income, col=mycolors[cont]))
```

There's one more detail that we need to take care of. Since we're using color on the graph, we have to provide some way that someone viewing the graph can tell which color represents which continent, i.e. we need to add a legend to the graph. In R, this is done with the `legend` command. There are many options to this command, but in its simplest form we just tell R where to put the legend, whether we should show points or lines, and what colors they should be. A title for the legend can also be added, which is a good idea in this example, because the meaning of the continent abbreviations may not be immediately apparent. You can specify x- and y-coordinates for the legend location or you can use one of several shortcuts like "topleft" to do things automatically. (You may also want to look at the `locator` command, that lets you decide where to place your legends interactively). For our example, the following will place a legend in an appropriate place; the `title` command is also used to add a title to the plot:

```
with(world1,legend('topleft',legend=levels(cont),col=mycolors,pch=1,title='Continent'))
title('Income versus Literacy for Countries around the World')
```

Notice how the `title` function can be used to add a title to a plot after it's displayed if you forget to provide a `main=` argument to `plot`.

The `pch=` argument to the `legend` function is a graphics parameter representing the *plotting character*. While the `plot` function uses a value of `pch=1` by default, the `legend` function won't display anything if you don't provide a `pch=` argument. (You might want to experiment with different values for the `pch=` argument in the `plot` function.) Here's what the plot looks like:



Code	Value
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated)
%B	Month (full name)
%y	Year (2 digit)
%Y	Year (4 digit)

## 2.12 Using Dates in R

Dates on computers have been the source of much anxiety, especially at the turn of the century, when people felt that many computers wouldn't understand the new millenium. These fears were based on the fact that certain programs would store the value of the year in just 2 digits, causing great confusion when the century "turned over". In R, dates are stored as they have traditionally been stored on Unix computers – as the number of days from a reference date, in this case January 1, 1970, with earlier days being represented by negative numbers. When dates are stored this way, they can be manipulated like any other numeric variable (as far as it makes sense). In particular, you can compare or sort dates, take the difference between two dates, or add an increment of days, weeks, months or years to a date. The class of such dates is `Date` and their mode is numeric. Dates are created with `as.Date`, and formatted for printing with `format` (which will recognize dates and do the right thing.)

Because dates can be written in so many different formats, R uses a standard way of providing flexibility when reading or displaying dates. A set of format codes, some of which are shown in the table below, is used to describe what the input or output form of the date looks like. The default format for `as.Date` is a four digit year, followed by a month, then a day, separated by either dashes or slashes. So conversions like this happen automatically:

```
> as.Date('1915-6-16')
[1] "1915-06-16"
> as.Date('1890/2/17')
[1] "1890-02-17"
```

The formatting codes are as follows:

(For a complete list of the format codes, see the R help page for the `strptime` function.)

As an example of reading dates, the URL <http://www.stat.berkeley.edu/classes/s133/data/movies.txt> contains the names, release dates, and box office earnings for around 700 of the most popular movies of all time. The first few lines of the input file look like this:

```
Rank|name|box|date
1|Avatar|$759.563|December 18, 2009
2|Titanic|$600.788|December 19, 1997
3|The Dark Knight|$533.184|July 18, 2008
```

As can be seen, the fields are separated by vertical bars, so we can use `read.delim` with the appropriate `sep=` argument.

```

> movies = read.delim('http://www.stat.berkeley.edu/classes/s133/data/movies.txt',
+ sep='|', stringsAsFactors=FALSE)
> head(movies)
   rank          name    box      date
1    1           Avatar $759.563 December 18, 2009
2    2           Titanic $600.788 December 19, 1997
3    3       The Dark Knight $533.184 July 18, 2008
4    4 Star Wars: Episode IV - A New Hope $460.998 May 25, 1977
5    5            Shrek 2 $437.212 May 19, 2004
6    6 E.T. the Extra-Terrestrial $434.975 June 11, 1982

```

The first step in using a data frame is making sure that we know what we're dealing with. A good first step is to use the `sapply` function to look at the mode of each of the variables:

```

> sapply(movies, mode)
  rank        name        box      date
"numeric" "character" "character" "character"

```

Unfortunately, the box office receipts (`box`) are character, not numeric. That's because R doesn't recognize a dollar sign (\$) as being part of a number. (R has the same problem with commas.) We can remove the dollar sign with the `sub` function, and then use `as.numeric` to make the result into a number:

```
> movies$box = as.numeric(sub('\\$', '', movies$box))
```

To convert the character `date` values to R Date objects, we can use `as.Date` with the appropriate format: in this case it's the month name followed by the day of the month, a comma and the four digit year. Consulting the table of format codes, this translates to '`%B %d, %Y`':

```

> movies$date = as.Date(movies$date, '%B %d, %Y')
> head(movies$date)
[1] "2009-12-18" "1997-12-19" "2008-07-18" "1977-05-25" "2004-05-19"
[6] "1982-06-11"

```

The format that R now uses to print the dates is the standard `Date` format, letting us know that we've done the conversion correctly. (If we wanted to recover the original format, we could use the `format` function with a format similar to the one we used to read the data.)

Now we can perform calculations using the date. For example, to see the difference in time between the release of *Titanic* and *Avatar* (2 very popular movies directed by James Cameron), we could use:

```

> movies$date[movies$name == 'Avatar'] - movies$date[movies$name == 'Titanic']
Time difference of 4382 days

```

Even though the result prints out as a character string, it's actually just a number which can be used any way a number could be used.

Now suppose we want to see the time difference in years. To convert days to years, we can divide by 365.25. (The .25 tries to account for leap years.):

```
> diff = movies$date[movies$name == 'Avatar'] - movies$date[movies$name == 'Titanic']
> diff / 365.25
Time difference of 11.99726 days
```

We could either adjust the `units` attribute of this value or use `as.numeric` to convert it to an ordinary number. (In R, an attribute is additional information stored along with a variable.)

```
> diff = diff / 365.25
> attr(diff,'units') = 'years'
> diff
Time difference of 11.99726 years
> as.numeric(diff)
[1] 11.99726
```

Either way, it will be treated as an ordinary number when used in a calculation.

The `Sys.Date` function can be used to return the current date, so R can calculate the time until any date you choose. For example, the midterm for this class is March 2, 2011:

```
> as.Date('2011-03-02') - Sys.Date()
Time difference of 28 days
```

Another way to create dates is with the `ISOdate` function. This function accepts three numbers representing the year, month and day of the date that is desired. So to reproduce the midterm date we could use

```
> midterm = ISOdate(2011,3,2)
> midterm
[1] "2011-03-02 12:00:00 GMT"
```

Notice that, along with the date, a time is printed. That's because `ISOdate` returns an object of class `POSIXt`, not `Date`. To make a date like this work properly with objects of class `Date`, you can use the `as.Date` function.

Once we've created an R `Date` value, we can use the functions `months`, `weekdays` or `quarters` to extract those parts of the date. For example, to see which day of the week these very popular movies were released, we could use the `table` function combined with `weekdays`:

```
> table(weekdays(movies$date))
  Friday   Monday   Saturday   Sunday   Thursday   Tuesday   Wednesday 
     753       10         7        11        39        22        164
```

Notice that the ordering of the days is not what we'd normally expect. This problem can be solved by creating a factor that has the levels in the correct order:

```
> movies$weekday = factor(weekdays(movies$date),  
+   levels = c('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'))
```

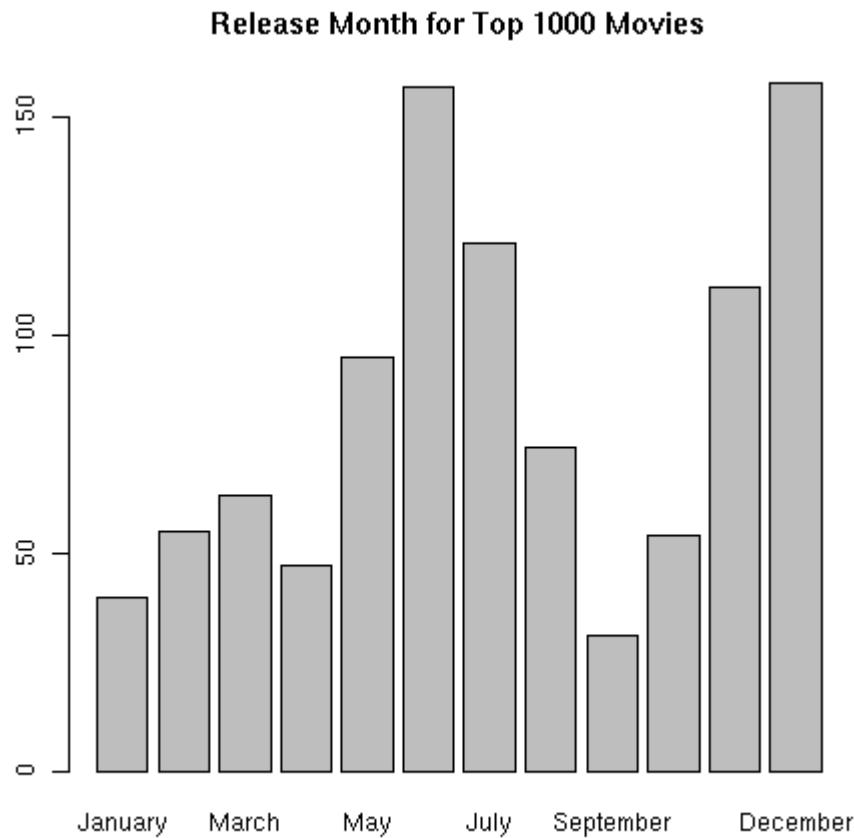
Now we can use `weekday` to get a nicer table:

```
> table(movies$weekday)
```

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
10	22	164	39	753	7	11

Similarly, if we wanted to graphically display a chart showing which month of the year the popular movies were released in, we could first create an ordered factor, then use the `barplot` function:

```
> movies$month = factor(months(movies$date),levels=c('January', 'February', 'March', 'April',  
> barplot(table(movies$month),main='Release Month for Top 1000 Movies')
```



To do a similar thing with years, we'd have to create a new variable that represented the year using the `format` function. For a four digit year the format code is `%Y`, so we could make a table of the hit movies by year like this:

```
> table(format(movies$date, '%Y'))
1938 1939 1940 1942 1946 1950 1953 1955 1956 1959 1961 1963 1964 1965 1967 1968
    1     1     1     2     1     1     1     1     1     1     2     1     2     3     2     2
1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984
    1     4     2     3     2     8     3     4     4     5     12    9     7     10    12    12
1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000
   10    12    13    13    23    19    21    26    19    20    32    26    29    37    41    44
2001 2002 2003 2004 2005 2006 2007 2008 2009 2010
   43    47    53    55    48    54    43    53    57    47
```

## 2.13 Data Summaries

We've looked at a few graphical techniques for exploring data, and now we're going to turn to a numerical one. Consider the question "Which day of the week has the highest average box office for hit movies released on that day?". As a first step in answering that question, it would be helpful to look at the mean box office receipts for each of the days. If you look for a function to do that specific task, you probably wouldn't find one, because R takes the more general approach of providing a function that will allow you to calculate anything you want from vectors of values broken down by groups. In fact, there are a variety of ways to do this. The one we're going to look at is called `aggregate`. You pass `aggregate` a vector or data frame containing the variables you want to summarize, a list of the groups to summarize by, and the function you'd like to use for your summaries. That way, a single function can perform many tasks, and, as we'll see when we learn to write functions, it even allows R to do things that the developers of R never imagined. For now, we'll stick to some built in functions, like `mean`. To find the means for the box office receipts for each day of the week, we could use a call to `aggregate` like this:

```
> aggregate(movies$box,movies['weekday'],mean)
  weekday      x
1 Monday 148.04620
2 Tuesday 110.42391
3 Wednesday 139.50965
4 Thursday 117.89700
5 Friday 112.24642
6 Saturday 91.18714
7 Sunday 140.45618
```

The same thing could be done to calculate other statistics, like `median`, `min`, `max`, or any statistic that returns a single scalar value for each group. Another nice feature of `aggregate`

is that if the first argument is a data frame, it will calculate the statistic for each column of the data frame. If we passed `aggregate` both the `rank` and `box`, we'd get two columns of summaries:

```
> aggregate(movies[,c('rank','box')],movies['weekday'],mean)
   weekday      rank      box
> aggregate(movies[,c('Rank','box')],movies['weekday'],mean)
   weekday      rank      box
1   Monday 354.5000 148.04620
2   Tuesday 498.9545 110.42391
3 Wednesday 423.2561 139.50965
4 Thursday 493.7692 117.89700
5     Friday 521.7384 112.24642
6 Saturday 577.5714  91.18714
7   Sunday 338.1818 140.45618
```

To add a column of counts to the table, we can create a data frame from the `table` function, and merge it with the aggregated results:

```
> dat = aggregate(movies[,c('rank','box')],movies['weekday'],mean)
> cts = as.data.frame(table(movies$weekday))
> head(cts)
  Var1 Freq
1   Monday   10
2   Tuesday   22
3 Wednesday 164
4 Thursday   39
5     Friday 753
6 Saturday    7
```

To make the merge simpler, we rename the first column of `cts` to `weekday`.

```
> names(cts)[1] = 'weekday'
> res = merge(cts,dat)
> head(res)
  weekday Freq      Rank      box
1   Friday  753 521.7384 112.24642
2   Monday   10 354.5000 148.04620
3 Saturday    7 577.5714  91.18714
4   Sunday   11 338.1818 140.45618
5 Thursday   39 493.7692 117.89700
6   Tuesday   22 498.9545 110.42391
```

Notice that the default behaviour of `merge` is to sort the columns before merging, so that we've lost the order that the `levels=` argument prescribed. The `sort=FALSE` argument to `merge` can be used to prevent that:

```

> res = merge(cts,dat,sort=FALSE)
> head(res)
  weekday Freq      rank      box
1   Monday  10 354.5000 148.04620
2  Tuesday  22 498.9545 110.42391
3 Wednesday 164 423.2561 139.50965
4 Thursday  39 493.7692 117.89700
5   Friday 753 521.7384 112.24642
6 Saturday  7 577.5714  91.18714

```

## 2.14 Functions

As you've already noticed, functions play an important role in R. A very attractive feature of R is that you can write your own functions which work exactly the same as the ones that are part of the official R release. In fact, if you create a function with the same name as one that's already part of R, it will override the built-in function, and possibly cause problems. For that reason, it's a good idea to make sure that there's not already another function with the name you want to use. If you type the name you're thinking of, and R responds with a message like "object "xyz" not found" you're probably safe.

There are several reasons why creating your own functions is a good idea.

1. If you find yourself writing the same code over and over again as you work on different problems, you can write a function that incorporates whatever it is you're doing and call the function, instead of rewriting the code over and over.
2. All of the functions you create are saved in your workspace along with your data. So if you put the bulk of your work into functions that you create, R will automatically save them for you (if you tell R to save your workspace when you quit.)
3. It's very easy to write "wrappers" around existing functions to make a custom version that sets the arguments to another function to be just what you want. R provides a special mechanism to "pass along" any extra arguments the other function might need.
4. You can pass your own functions to built-in R functions like `aggregate`, `by`, `apply`, `sapply`, `lapply`, `mapply`, `sweep` and other functions to efficiently and easily perform customized tasks.

Before getting down to the details of writing your own functions, it's a good idea to understand how functions in R work. Every function in R has a set of arguments that it accepts. You can see the arguments that built-in functions take in a number of ways: viewing the help page, typing the name of the function in the interpreter, or using the `args` function. When you call a function, you can simply pass it arguments, in which case they must line up exactly with the way the function is designed, or you can specifically pass particular arguments in whatever order you like by providing the with names using the `name=value`

syntax. You also can combine the two, passing unnamed arguments (which have to match the function's definition exactly), followed by named arguments in whatever order you like. For example, consider the function `read.table`. We can view its argument list with the command:

```
> args(read.table)
function (file, header = FALSE, sep = "", quote = "\"\"", dec = ".",
  row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",
  colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
  fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
  comment.char = "#", allowEscapes = FALSE, flush = FALSE,
  stringsAsFactors = default.stringsAsFactors(), encoding = "unknown")
NULL
```

This argument list tells us that, if we pass unnamed arguments to `read.table`, it will interpret the first as `file`, the next as `header`, then `sep`, and so on. Thus if we wanted to read the file `my.data`, with `header` set to `TRUE` and `sep` set to `,`, any of the following calls would be equivalent:

```
read.table('my.data',TRUE,',')
read.table(sep=',',TRUE,file='my.data')
read.table(file='my.data',sep=',',header=TRUE)
read.table('my.data',sep=',',header=TRUE)
```

Notice that all of the arguments in the argument list for `read.table` have values after the name of the argument, except for the `file` argument. This means that `file` is the only required argument to `read.table`; any of the other arguments are optional, and if we don't specify them the default values that appear in the argument list will be used. Most R functions are written so the the first few arguments will be the ones that will usually be used so that their values can be entered without providing names, with the other arguments being optional. Optional arguments can be passed to a function by position, but are much more commonly passed using the `name=value` syntax, as in the last example of calling `read.table`.

Now let's take a look at the function `read.csv`. You may recall that this function simply calls `read.table` with a set of parameters that makes sense for reading comma separated files. Here's `read.csv`'s function definition, produced by simply typing the function's name at the R prompt:

```
function (file, header = TRUE, sep = ",", quote = "\"\"", dec = ".",
  fill = TRUE, comment.char = "", ...)
read.table(file = file, header = header, sep = sep, quote = quote,
  dec = dec, fill = fill, comment.char = comment.char, ...)
<environment: namespace:utils>
```

Pay special attention to the three periods (...) in the argument list. Notice that they also appear in the call to `read.table` inside the function's body. The three dots mean all the

arguments that were passed to the function that didn't match any of the previous arguments in the argument list. So if you pass anything other than `file`, `header`, `sep`, `quote`, `dec`, or `fill` to `read.csv`, it will be part of the three dots; by putting the three dots at the end of the argument list in the call to `read.table`, all those unmatched arguments are simply passed along to `read.table`. So if you make a call to `read.csv` like this:

```
read.csv(filename,stringsAsFactors=FALSE)
```

the `stringsAsFactors=FALSE` will get passed to `read.table`, even though it wasn't explicitly named in the argument list. Without the three dots, R will not accept any arguments that aren't explicitly named in the argument list of the function definition. If you want to intercept the extra arguments yourself, you can include the three dots at the end of the argument list when you define your function, and create a list of those arguments inside the function body by referring to `list(...)`.

Suppose you want to create a function that will call `read.csv` with a filename, but which will automatically set the `stringsAsFactors=FALSE` parameter. For maximum flexibility, we'd want to be able to pass other arguments (like `na.strings=`, or `quote=`) to `read.csv`, so we'll include the three dots at the end of the argument list. We could name the function `read.csv` and overwrite the built-in version, but that's not a good idea, if for no other reason than the confusion it would cause if someone else tried to understand your programs! Suppose we call the function `myread.csv`. We could write a function definition as follows:

```
> myread.csv = function(file,stringsAsFactors=FALSE,...){  
+   read.csv(file,stringsAsFactors=stringsAsFactors,...)  
+ }
```

Now, we could simply use

```
thedata = myread.csv(filename)
```

to read a comma-separated file with `stringsAsFactors=FALSE`. You could still pass any of `read.table`'s arguments to the function (including `stringsAsFactors=TRUE` if you wanted), and, if you ask R to save your workspace when you quit, the function will be available to you next time you start R in the same directory.

## 2.15 Functions

When you create a function, it defines a separate environment and the variables you create inside your function only exist in that function environment; when you return to where you called the function from, those variables no longer exist. You can refer to other objects that are in the calling environment, but if you make any changes to them, the changes will only take place in the function environment. To get information back to the calling environment, you must pass a return value, which will be available through the functions name. R will automatically return the last unassigned value it encounters in your function, or you can place the object you want to return in a call to the `return` function. You can only return a single object from a function in R; if you need to return multiple objects, you need to return a list containing those objects, and extract them from the list when you return to the calling environment.

As a simple example of a function that returns a value, suppose we want to calculate the ratio of the maximum value of a vector to the minimum value of the vector. Here's a function definition that will do the job:

```
maxminratio = function(x)max(x)/min(x)
```

Notice for a single line function you don't need to use brackets (`{}`) around the function body, but you are free to do so if you like. Since the final statement wasn't assigned to a variable, it will be used as a return value when the function is called. Alternatively, the value could be placed in a call to the `return` function. If we wanted to find the max to min ratio for all the columns of the matrix, we could use our function with the `apply` function:

```
apply(mymat,2,maxminratio)
```

The 2 in the call to `apply` tells it to operate on the columns of the matrix; a 1 would be used to work on the rows.

Before we leave this example, it should be pointed out that this function has a weakness – what if we pass it a vector that has missing values? Since we're calling `min` and `max` without the `na.rm=TRUE` argument, we'll always get a missing value if our input data has any missing values. One way to solve the problem is to just put the `na.rm=TRUE` argument into the calls to `min` and `max`. A better way would be to create a new argument with a default value. That way, we still only have to pass one argument to our function, but we can modify the `na.rm=` argument if we need to.

```
maxminratio = function(x,na.rm=TRUE)max(x,na.rm=na.rm)/min(x,na.rm=na.rm)
```

If you look at the function definitions for functions in R, you'll see that many of them use this method of setting defaults in the argument list.

As your functions get longer and more complex, it becomes more difficult to simply type them into an interactive R session. To make it easy to edit functions, R provides the `edit` command, which will open an editor appropriate to your operating system. When you close the editor, the `edit` function will return the edited copy of your function, so it's important to remember to assign the return value from `edit` to the function's name. If you've already defined a function, you can edit it by simply passing it to `edit`, as in

```
minmaxratio = edit(minmaxratio)
```

You may also want to consider the `fix` function, which automates the process slightly.

To start from scratch, you can use a call to `edit` like this:

```
newfunction = edit(function(){})
```

Suppose we want to write a function that will allow us to calculate the mean of all the appropriate columns of a data frame, broken down by a grouping variable, and including the counts for the grouping variables in the output. When you're working on developing a function, it's usually easier to solve the problem with a sample data set, and then generalize it to a function. We'll use the `movies` data frame as an example, with both `weekday` and `month` as potential grouping variables. First, let's go over the steps to create the `movies` data frame with both grouping variables:

```
> movies = read.delim('http://www.stat.berkeley.edu/classes/s133/data/movies.txt',
+ sep='|',stringsAsFactors=FALSE)
> movies$box = as.numeric(sub('\\\\$','',movies$box))
> movies$date = as.Date(movies$date,'%B %d, %Y')
> movies$weekday = weekdays(movies$date)
> movies$weekday = factor(weekdays(movies$date),
+   levels = c('Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'))
> movies$month = months(movies$date)
> movies$month = factor(months(movies$date),levels=c('January','February','March',
+   'April','May','June','July','August','September','October','November','December'))
```

Since I've done a fair amount of processing to this data set, and since I'm going to want to use it later for testing my function, I'm going to use the `save` function to write a copy of the data frame to a file. This function writes out R objects in R's internal format, just like the workspace is saved at the end of an R session. You can also transfer a file produced by `save` to a different computer, because R uses the same format for its saved objects on all operating systems. Since `save` accepts a variable number of arguments, we need to specify the `file=` argument when we call it:

```
> save(movies,file='movies.rda')
```

You can use whatever extension you want, but `.rda` or `.Rdata` are common choices.

It's often useful to breakdown the steps of a problem like this, and solve each one before going on to the next. Here are the steps we'll need to go through to create our function.

1. Find the appropriate columns of the data frame for the aggregate function.
2. Write the call to the `aggregate` function that will give us the mean for each group.
3. Write the call to the `function` to get the counts and convert it to a data frame.
4. Merge together the results from `aggregate` and `table` to give us our result.

To find the appropriate variables, we can examine the `class` and `mode` of each column of our data frame:

```
> sapply(movies, class)
  rank      name       box      date   weekday     month
"integer" "character" "numeric" "Date"   "factor"   "factor"
> sapply(movies, mode)
  rank      name       box      date   weekday     month
"numeric" "character" "numeric" "numeric" "numeric" "numeric"
```

For this data frame, the appropriate variables for aggregation would be `rank` and `box`, so we have to come up with some logic that would select only those columns. One easy way is to select those columns whose class is either `numeric` or `integer`. We can use the `|` operator which represents a logical “or” to create a logical vector that will let us select the columns we want. (There’s also the `&` operator which is used to test for a logical “and”.)

```
> classes = sapply(movies, class)
> numcols = classes == 'integer' | classes == 'numeric'
```

While this will certainly work, R provides an operator that makes expressions like this easier to write. The `%in%` operator allows us to test for equality to more than one value at a time, without having to do multiple tests. In this example we can use it as follows:

```
> numcols = sapply(movies, class) %in% c('integer', 'numeric')
```

Now we need to write a call to the `aggregate` function that will find the means for each variable based on a grouping variable. To develop the appropriate call, we’ll use `weekday` as a grouping variable:

```
> result = aggregate(movies[, numcols], movies['weekday'], mean)
> result
  weekday      rank       box
1 Monday 354.5000 148.04620
2 Tuesday 498.9545 110.42391
3 Wednesday 427.1863 139.38540
4 Thursday 493.7692 117.89700
5 Friday 520.2413 112.44878
6 Saturday 577.5714  91.18714
7 Sunday 338.1818 140.45618
```

Similarly, we need to create a data frame of counts that can be merged with the result of `aggregate`:

```
> counts = as.data.frame(table(movies['weekday']))
> counts
  Var1 Freq
```

```

1   Monday    10
2   Tuesday   22
3 Wednesday 161
4 Thursday   39
5 Friday    750
6 Saturday    7
7 Sunday    11

```

Unfortunately, this doesn't name the first column appropriately for the `merge` function. The best way to solve this problem is to change the name of the first column of the `counts` data frame to the name of the grouping variable. Recall that using the `sort=FALSE` argument to `merge` will retain the order of the grouping variable that we specified with the `levels=` argument to `factor`

```

> names(counts)[1] = 'weekday'
> merge(counts,result,sort=FALSE)
  weekday Freq      rank      box
1   Monday    10 354.5000 148.04620
2   Tuesday   22 498.9545 110.42391
3 Wednesday 161 427.1863 139.38540
4 Thursday   39 493.7692 117.89700
5 Friday    750 520.2413 112.44878
6 Saturday    7 577.5714  91.18714
7 Sunday    11 338.1818 140.45618

```

This gives us exactly the result we want, with the columns labeled appropriately.

To convert this to a function, let's put together all the steps we just performed:

```

> load('movies.rda')
> numcols = sapply(movies,class) %in% c('integer','numeric')
> result = aggregate(movies[,numcols],movies['weekday'],mean)
> counts = as.data.frame(table(movies['weekday']))
> names(counts)[1] = 'weekday'
> merge(counts,result,sort=FALSE)
  weekday Freq      rank      box
1   Monday    10 354.5000 148.04620
2   Tuesday   22 498.9545 110.42391
3 Wednesday 161 427.1863 139.38540
4 Thursday   39 493.7692 117.89700
5 Friday    750 520.2413 112.44878
6 Saturday    7 577.5714  91.18714
7 Sunday    11 338.1818 140.45618

```

To convert these steps into a function that we could use with any data frame, we need to identify the parts of these statements that would change with different data. In this case,

there are two variables that we'd have to change: `movies` which represents the data frame we're using, and '`weekday`' which represents the grouping variable we're using. Here's a function that will perform these operations for any combination of data frame and grouping variable. (I'll change `movies` to `df` and '`weekday`' to `grp` to make the names more general, and name the function `aggall`:

```
> aggall = function(df,grp){
+   numcols = sapply(df,class) %in% c('integer','numeric')
+   result = aggregate(df[,numcols],df[grp],mean)
+   counts = as.data.frame(table(df[grp]))
+   names(counts)[1] = grp
+   merge(counts,result,sort=FALSE)
+ }
```

I'm taking advantage of the fact the R functions will return the result of the last statement in the function that's not assigned to a variable, which in this case is the result of the `merge` function. Alternatively, I could assign the result of the `merge` function to a variable and use the `return` function to pass back the result. At this point it would be a good idea to copy our function into a text file so that we can re-enter it into our R session whenever we need it. You could copy the definition from the console, and remove the `>` prompts, or you could use the `history` command to cut and paste the function without the prompts. (Other ways of saving the text of the function include the `dput` and `dump` functions, or simply making sure you save the workspace before quitting R.) I'll call the text file that I paste the definition into `aggall.r`.

Now that we have the function written, we need to test it. It's often a good idea to test your functions in a "fresh" R session. We can get the `movies` data frame back into our workspace by using the `load` command, passing it the name of the file that we created earlier with the `save` command, and we can use the `source` function to read back the definition of the `aggall` function:

```
> rm(list=objects()) # removes everything from your workspace !!
> source('aggall.r')
> load('movies.rda')
```

The first test would be to use the `movies` data frame, but with a different grouping variable:

```
> aggall(movies,'month')
    month Freq      rank      box
1   January  40 605.0750  92.41378
2   February  55 621.5818  90.98182
3     March  63 516.1111 107.73638
4     April  47 712.9362  77.45891
5      May  95 338.7474 168.77575
6     June 157 449.3885 125.98888
```

```

7      July  121 454.4298 129.67063
8     August   74 562.4865 100.53996
9 September   31 553.2581  99.29284
10 October   54 623.6667  86.12557
11 November  111 441.3423 124.22192
12 December  158 507.9051 117.10242

```

It seems to work well.

But the real test is to use the function on a different data frame, like the `world1` data frame. Let's recreate it:

```

> world = read.csv('http://www.stat.berkeley.edu/classes/s133/data/world.txt',
+   header=TRUE)
> conts = read.csv('http://www.stat.berkeley.edu/classes/s133/data/conts.txt',
+   na.string='.')
> world1 = merge(world,conts)
> aggall(world1,'cont')
  cont Freq      gdp    income literacy    military
1   AF   47 2723.404 3901.191 60.52979 356440000
2   AS   41 7778.049 8868.098 84.25122 5006536341
3   EU   34 19711.765 21314.324 98.40294 6311138235
4   NA   15 8946.667        NA 85.52000 25919931267
5   OC     4 14625.000 15547.500 87.50000 4462475000
6   SA   12 6283.333 6673.083 92.29167 2137341667

```

We're close, but notice that there's an `NA` for income for one of the continents. Since the `movies` data frame didn't have any missing values, we overlooked that detail. The most important thing about writing functions is to remember that you can always modify them using either the `edit` or `fix` function. For example, if I type

```
> aggall = edit(aggall)
```

I'll see the body of the `aggall` function in a text editor where I can modify it. In this case, I simply need to add the `na.rm=TRUE` argument to my call to `aggregate`, resulting in this new version of the function:

```

function(df,grp){
  numcols = sapply(df,class) %in% c('integer','numeric')
  result = aggregate(df[,numcols],df[grp],mean,na.rm=TRUE)
  counts = as.data.frame(table(df[grp]))
  names(counts)[1] = grp
  merge(counts,result,sort=FALSE)
}

```

Testing it again on the `world1` data frame shows that we have solved the problem:

```
> aggall(world1,'cont')
  cont Freq      gdp    income literacy   military
1   AF   47 2723.404 3901.191 60.52979 356440000
2   AS   41 7778.049 8868.098 84.25122 5006536341
3   EU   34 19711.765 21314.324 98.40294 6311138235
4   NA   15 8946.667 10379.143 85.52000 25919931267
5   OC     4 14625.000 15547.500 87.50000 4462475000
6   SA   12 6283.333 6673.083 92.29167 2137341667
```

## 2.16 Sizes of Objects

Before we start looking at character manipulation, this is a good time to review the different functions that give us the size of an object.

1. `length` - returns the length of a vector, or the total number of elements in a matrix (number of rows times number of columns). For a data frame, returns the number of columns.
2. `dim` - for matrices and data frames, returns a vector of length 2 containing the number of rows and the number of columns. For a vector, returns `NULL`. The convenience functions `nrow` and `ncol` return the individual values that would be returned by `dim`.
3. `nchar` - for a character string, returns the number of characters in the string. Returns a vector of values when applied to a vector of character strings. For a numeric value, `nchar` returns the number of characters in the printed representation of the number.

## 2.17 Character Manipulation

While it's quite natural to think of data as being numbers, manipulating character strings is also an important skill when working with data. We've already seen a few simple examples, such as choosing the right format for a character variable that represents a date, or using `table` to tabulate the occurrences of different character values for a variable. Now we're going to look at some functions in R that let us break apart, rearrange and put together character data.

One of the most important uses of character manipulation is "massaging" data into shape. Many times the data that is available to us, for example on a web page or as output from another program, isn't in a form that a program like R can easily interpret. In cases like that, we'll need to remove the parts that R can't understand, and organize the remaining parts so that R can read them efficiently.

Let's take a look at some of the functions that R offers for working with character variables:

- `paste` The `paste` function converts its arguments to character before operating on them, so you can pass both numbers and strings to the function. It concatenates the arguments passed to it, to create new strings that are combinations of other strings. `paste` accepts an unlimited number of unnamed arguments, which will be pasted together, and one or both of the arguments `sep=` and `collapse=`. Depending on whether the arguments are scalars or vectors, and which of `sep=` and `collapse=` are used, a variety of different tasks can be performed.

1. If you pass a single argument to `paste`, it will return a character representation:

```
> paste('cat')
```

```
[1] "cat"
> paste(14)
[1] "14"
```

2. If you pass more than one scalar argument to `paste`, it will put them together in a single string, using the `sep=` argument to separate the pieces:

```
> paste('stat',133,'assignment')
[1] "stat 133 assignment"
```

3. If you pass a vector of character values to `paste`, and the `collapse=` argument is not `NULL`, it pastes together the elements of the vector, using the `collapse=` argument as a separator:

```
> paste(c('stat',133,'assignment'),collapse=' ')
[1] "stat 133 assignment"
```

4. If you pass more than one argument to `paste`, and any of those arguments is a vector, `paste` will return a vector as long as its' longest argument, produced by pasting together corresponding pieces of the arguments. (Remember the recycling rule which will be used if the vector arguments are of different lengths.) Here are a few examples:

```
> paste('x',1:10,sep='')
[1] "x1"  "x2"  "x3"  "x4"  "x5"  "x6"  "x7"  "x8"  "x9"  "x10"
> paste(c('x','y'),1:10,sep='')
[1] "x1"  "y2"  "x3"  "y4"  "x5"  "y6"  "x7"  "y8"  "x9"  "y10"
```

- `grep` The `grep` function searches for patterns in text. The first argument to `grep` is a text string or regular expression that you're looking for, and the second argument is usually a vector of character values. `grep` returns the indices of those elements of the vector of character strings that contain the text string. Right now we'll limit ourselves to simple patterns, but later we'll explore the full strength of commands like this with regular expressions.

`grep` can be used in a number of ways. Suppose we want to see the countries of the world that have the word 'United' in their names.

```
> grep('United',world1$country)
[1] 144 145
```

`grep` returns the indices of the observations that have 'United' in their names. If we wanted to see the values of `country` that had 'United' in their names, we can use the `value=TRUE` argument:

```
> grep('United',world1$country,value=TRUE)
[1] "United Arab Emirates" "United Kingdom"
```

Notice that, since the first form of `grep` returns a vector of indices, we can use it as a subscript to get all the information about the countries that have 'United' in their names:

```
> world1[grep('United',world1$country),]
      country   gdp income literacy   military cont
144 United Arab Emirates 23200  23818     77.3 16000000000   AS
145       United Kingdom 27700  28938     99.9 42836500000   EU
```

`grep` has a few optional arguments, some of which we'll look at later. One convenient argument is `ignore.case=TRUE`, which, as the name implies will look for the pattern we specified without regard to case.

- **strsplit** `strsplit` takes a character vector, and breaks each element up into pieces, based on the value of the `split=` argument. This argument can be an ordinary text string, or a regular expression. Since the different elements of the vector may have different numbers of "pieces", the results from `strsplit` are always returned in a list. Here's a simple example:

```
> mystrings = c('the cat in the hat','green eggs and ham','fox in socks')
> parts = strsplit(mystrings, ' ')
> parts
[[1]]
[1] "the" "cat" "in" "the" "hat"

[[2]]
[1] "green" "eggs" "and" "ham"

[[3]]
[1] "fox" "in" "socks"
```

While we haven't dealt much with lists before, one function that can be very useful is `sapply`; you can use `sapply` to operate on each element of a list, and it will, if possible, return the result as a vector. So to find the number of words in each of the character strings in `mystrings`, we could use:

```
> sapply(parts,length)
[1] 5 4 3
```

- **substring** The `substring` function allows you to extract portions of a character string. Its first argument is a character string, or vector of character strings, and its second argument is the index (starting with 1) of the beginning of the desired substring. With no third argument, `substring` returns the string starting at the specified index and

continuing to the end of the string; if a third argument is given, it represents the last index of the original string that will be included in the returned substring. Like many functions in R, its true value is that it is fully vectorized: you can extract substrings of a vector of character values in a single call. Here's an example of a simple use of `substring`

```
> strings = c('elephant', 'aardvark', 'chicken', 'dog', 'duck', 'frog')
> substring(strings, 1, 5)
[1] "eleph" "aardv" "chick" "dog"    "duck"   "frog"
```

Notice that, when a string is too short to fully meet a substringing request, no error or warning is raised, and `substring` returns as much os the string as is there.

Consider the following example, extracted from a web page. Each element of the character vector `data` consists of a name followed by five numbers. Extracting an individual field, say the field with the state names is straight forward:

```
> data = c("Lyndhurst      Ohio      199.02  15,074  30  5  25",
          "Southport Town New York  217.69  11,025  24  4  20",
          "Bedford       Massachusetts 221.20  12,658  28  0  28")
> states = substring(data, 16, 28)
> states
[1] "Ohio"           "New York"        "Massachusetts"
```

It is possible to extract all the fields at once, at the cost of a considerably more complex call to `substring`:

```
> starts = c(1, 16, 30, 38, 46, 50, 54)
> ends   = c(14, 28, 35, 43, 47, 50, 55)
> ldata = length(data)
> lstarts = length(starts)
> x = substring(data, rep(starts, rep(ldata, lstarts)), rep(ends, rep(ldata, lstarts)))
> matrix(x, ncol=lstarts)
 [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
[1,] "Lyndhurst" "Ohio"    "199.02" "15,074" "30"    "5"     "25"
[2,] "Southport" "Town"   "217.69" "11,025" "24"    "4"     "20"
[3,] "Bedford"   "Massachusetts" "221.20" "12,658" "28"    "0"     "28"
```

Like many functions in R, `substring` can appear on the left hand side of an assignment statement, making it easy to change parts of a character string based on the positions they're in. To change the third through fifth digits of a set of character strings representing numbers to 99, we could use:

```
> nums = c('12553','73911','842099','203','10')
> substring(nums,3,5) = '99'
> nums
[1] "12993"  "73991"  "849999" "209"    "10"
```

- **tolower, toupper** These functions convert their arguments to all upper-case characters or all lower-case characters, respectively
- **sub, gsub** These functions change a regular expression or text pattern to a different set of characters. They differ in that **sub** only changes the first occurrence of the specified pattern, while **gsub** changes all of the occurrences. Since numeric values in R cannot contain dollar signs or commas, one important use of **gsub** is to create numeric variables from text variables that represent numbers but contain commas or dollars. For example, in gathering the data for the world dataset that we've been using, I extracted the information about military spending from [http://en.wikipedia.org/wiki/List\\_of\\_countries\\_by\\_military\\_expenditures](http://en.wikipedia.org/wiki/List_of_countries_by_military_expenditures). Here's an excerpt of some of the values from that page:

```
> values = c('370,700,000,000','205,326,700,000','67,490,000,000')
> as.numeric(values)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

The presence of the commas is preventing R from being able to convert the values into actual numbers. **gsub** easily solves the problem:

```
> as.numeric(gsub(',','',values))
[1] 370700000000 205326700000 67490000000
```

## 2.18 Working with Characters

As you probably noticed when looking at the above functions, they are very simple, and, quite frankly, it's hard to see how they could really do anything complex on their own. In fact, that's just the point of these functions – they can be combined together to do just about anything you would want to do. As an example, consider the task of capitalizing the first character of each word in a string. The `toupper` function can change the case of *all* the characters in a string, but we'll need to do something to separate out the characters so we can get the first one. If we call `strsplit` with an empty string for the splitting character, we'll get back a vector of the individual characters:

```
> str = 'sherlock holmes'  
> letters = strsplit(str, '')  
> letters  
[[1]]  
[1] "s" "h" "e" "r" "l" "o" "c" "k" " " "h" "o" "l" "m" "e" "s"  
> theletters = letters[[1]]
```

Notice that `strsplit` always returns a list. This will be very useful later, but for now we'll extract the first element before we try to work with its output.

The places that we'll need to capitalize things are the first position in the vector of letters, and any letter that comes after a blank. We can find those positions very easily:

```
> wh = c(1,which(theletters == ' ') + 1)  
> wh  
[1] 1 10
```

We can change the case of the letters whose indexes are in `wh`, then use `paste` to put the string back together.

```
> theletters[wh] = toupper(theletters[wh])  
> paste(theletters,collapse='')  
[1] "Sherlock Holmes"
```

Things have gotten complicated enough that we could probably stand to write a function:

```
maketitle = function(txt){  
  theletters = strsplit(txt, '')[[1]]  
  wh = c(1,which(theletters == ' ') + 1)  
  theletters[wh] = toupper(theletters[wh])  
  paste(theletters,collapse='')  
}
```

Of course, we should always test our functions:

```
> maketitle('some crazy title')  
[1] "Some Crazy Title"
```

Now suppose we have a vector of strings:

```
> titls = c('sherlock holmes','avatar','book of eli','up in the air')
```

We can always hope that we'll get the right answer if we just use our function:

```
> maketitle(titls)
[1] "Sherlock Holmes"
```

Unfortunately, it didn't work in this case. Whenever that happens, `sapply` will operate on all the elements in the vector:

```
> sapply(titls,maketitle)
      sherlock holmes          avatar        book of eli      up in the air
"Sherlock Holmes"       "Avatar"     "Book Of Eli"    "Up In The Air"
```

Of course, this isn't the only way to solve the problem. Rather than break up the string into individual letters, we can break it up into words, and capitalize the first letter of each, then combine them back together. Let's explore that approach:

```
> str = 'sherlock holmes'
> words = strsplit(str, ' ')
> words
[[1]]
[1] "sherlock" "holmes"
```

Now we can use the assignment form of the `substring` function to change the first letter of each word to a capital. Note that we have to make sure to actually return the modified string from our call to `sapply`, so we insure that the last statement in our function returns the string:

```
> sapply(words[[1]],function(w){substring(w,1,1) = toupper(substring(w,1,1));w})
      sherlock      holmes
"Sherlock"    "Holmes"
```

Now we can paste the pieces back together to get our answer:

```
> res = sapply(words[[1]],function(w){substring(w,1,1) = toupper(substring(w,1,1));w})
> paste(res,collapse=' ')
[1] "Sherlock Holmes"
```

To operate on a vector of strings, we'll need to incorporate these steps into a function, and then call `sapply`:

```
mktitl = function(str){
  words = strsplit(str, ' ')
  res = sapply(words[[1]],function(w){substring(w,1,1) = toupper(substring(w,1,1));w})
  paste(res,collapse=' ')
}
```

We can test the function, making sure to use a string different than the one we used in our initial test:

```
> mktitl('some silly string')
[1] "Some Silly String"
```

And now we can test it on the vector of strings:

```
> titls = c('sherlock holmes','avatar','book of eli','up in the air')
> sapply(titls,mktitl)
  sherlock holmes      avatar      book of eli      up in the air
"Sherlock Holmes"     "Avatar"    "Book Of Eli"   "Up In The Air"
```

How can we compare the two methods? The R function `system.time` will report the amount of time any operation in R uses. One important caveat – if you wish to assign an expression to a value in the `system.time` call, you must use the “`<-`” assignment operator, because the equal sign will confuse the function into thinking you’re specifying a named parameter in the function call. Let’s try `system.time` on our two functions:

```
> system.time(one <- sapply(titls,maketitle))
  user  system elapsed
  0.000  0.000  0.001
> system.time(two <- sapply(titls,mktitl))
  user  system elapsed
  0.000  0.000  0.002
```

For such a tiny example, we can’t really trust that the difference we see is real. Let’s use the movie names from a previous example:

```
> movies = read.delim('http://www.stat.berkeley.edu/classes/s133/data/movies.txt',
+ sep='|',stringsAsFactors=FALSE)
> nms = tolower(movies$name)
> system.time(one <- sapply(nms,maketitle))
  user  system elapsed
  0.044  0.000  0.045
> system.time(two <- sapply(nms,mktitl))
  user  system elapsed
  0.256  0.000  0.258
```

It looks like the first method is better than the second. Of course, if they don’t get the same answer, it doesn’t really matter how fast they are. In R, the `all.equal` function can be used to see if things are the same:

```
> all.equal(one,two)
[1] TRUE
```

# **Chapter 3**

## **Unix**

## 3.1 Software for Remote Access

To learn about the software you'll need to access the SCF UNIX machines remotely see Accessing the SCF remotely. To see a list of all the SCF computers go to the Computer Grid page.

## 3.2 Basics of Unix

On a UNIX system, most commands are entered into a shell. There is a large amount (sometimes too much) of online documentation for all UNIX commands, through the `man` command. For example, to find out more about the `ls` command, which lists the files in a directory, type

```
man ls
```

at the UNIX prompt.

Another way to use the `man` command is to use keywords; type either

```
man -k keyword
```

or

```
apropos keyword
```

at the UNIX prompt to find a list of commands that involve “keyword”.

One attractive feature of UNIX shells is tab completion; if you type only the first few letters of a command or file name, and then hit the tab key, the shell will complete the name you started to type provided that there is only one match. If there's more than one match, hitting tab twice will list all the names that match.

A properly configured UNIX file system has permission controls to prevent unauthorized access to files, and to make sure that users do not accidentally remove or modify key files. If you want to adjust the permissions on the files you own, take a look at the man page for the `chmod` command.

I'm not going to make a big distinction between UNIX, Linux and the UNIX core of Mac OSX, so the commands we're going to look at here should work under any of these operating systems.

## 3.3 Command Path

When you type a command into the shell, it will search through a number of directories looking for a command that matches the name you typed. The directories it looks through are called the search path. You can display your search path by typing

```
echo $PATH
```

Command	Description	Examples
<code>ls</code>	Lists files in a given directory	<code>ls /some/directory</code> <code>ls # with no args, lists current dir</code>
<code>cd</code>	Change Working Directory	<code>cd /some/directory</code> <code>cd #with no args, cd to home dir</code>
<code>pwd</code>	Print Working Directory	<code>pwd</code>
<code>mkdir</code>	Create New Directory	<code>mkdir subdirectory</code>
<code>less</code>	Display file one screen at a time	<code>less filename</code>
<code>cp</code>	Copy files	<code>cp file1 newfile1</code> <code>cp file1 file2 file3 somedirectory</code>
<code>mv</code>	Move or rename a file	<code>mv oldfile newfile</code> <code>mv file1 file2 file3 somedirectory</code>
<code>rm</code>	Remove a file	<code>rm file1 file2</code> <code>rm -r dir #removes all directories and subdirectories</code>
<code>rmdir</code>	Remove a (empty) directory	<code>rmdir mydir</code>
<code>history</code>	Display previously typed commands	<code>history</code>
<code>grep</code>	Find strings in files	<code>grep Error file.out</code>
<code>head</code>	Show the first few lines of a file	<code>head myfile</code> <code>head -20 myfile</code>
<code>tail</code>	Show the last few lines of a file	<code>tail myfile</code> <code>tail -20 myfile</code>
<code>file</code>	Identify the type of a file	<code>file myfile</code>

To see the complete list of commands that are available on a given computer, you could look at all the files (commands) in all of the directories on your search path. (There are well over 2000 commands on most UNIX systems.)

## 3.4 Basic Commands

The table below shows some of the most commonly used UNIX commands:

Each of these commands has many options which you can learn about by viewing their man page. For example, to get more information about the `ls` command, type

```
man ls
```

## 3.5 Command History

Another useful feature of most UNIX shells is the ability to retrieve and review the commands you've previously typed into that instance of the shell. The arrow keys can be used to scroll up or down through previous commands. Once a command appears on the command line (where you would normally type a command), you can use the arrow and/or backspace keys

Command control-p	Meaning Previous line	Command control-n	Meaning Next line
control-f	One character forward	control-b	One character backward
control-a	Beginning of line	control-e	End of line
control-d	Delete one character	control-k	Delete to end of line

Wildcard	Meaning
*	Zero or more of any character
?	Any single character
[...]	Any of the characters between the brackets
[^...]	Any characters except those between the brackets
[x-y]	Any character in the range x to y [0-9] [a-z]
string-1,string-2,string-3	Each of the strings in turn

to edit the command. A number of control key combinations can also be used to navigate your command history and are displayed in the table below; these same control sequences will work in the Emacs editor.

## 3.6 Editors

Most programming tasks involve a stored copy of a program in a file somewhere, along with a file which could contain necessary data. On a UNIX system, it becomes very important to be able to view and modify files, and the program that does that is known as an editor. The kind of editor you use to work with files on a UNIX system is very different than a word processor, or other document handling program, as it's not concerned with formatting, fonts, or other issues of appearance – it basically just holds the bit patterns that represent the commands in your program or the information in your data. There are many editors available on UNIX systems, but the two most popular are `emacs` and `vi`. Some of the other editors you might encounter on a UNIX system are `pico`, `nano`, `vim`, `xemacs`, `gedit`, and `kate`.

## 3.7 Wildcards

Along with file completion, UNIX shells offer another way to save typing when entering filenames. Certain characters (known as wildcards) have special meaning when included as a filename, and the shell will expand these characters to represent multiple filenames. The most commonly used wildcard is the asterisk (\*), which will match anything in a file name; other possibilities are in the table below. To see what will be matched for any string containing wildcards, use the UNIX `echo` command.

In addition, the shell will recognize the tilda (~) as representing your home directory, and `~user` as `user`'s home directory.

## 3.8 Redirection

Unix systems manage their input and output through three so-called streams, known as standard input, standard output and standard error. Normally, you don't have to do anything – standard input will be whatever you type, and standard output and error will be your computer screen. However, sometimes it's useful to redirect input from some other source, or to redirect output to some other source. For example, suppose you wanted a list of files in your current directory. You could run the `ls` command and redirect the output to a file, say `myfile`s as follows

```
ls > myfile
```

Such a command will overwrite the file `myfile` if it already existed. To redirect output to the end of a file, leaving any content intact, use

```
ls >> myfile
```

To have a command read its input from a file instead from standard input the less-than sign (<) can be used.

Another useful form of redirection is known as a pipe. In this case, the standard output of one program is used as the standard input to another program. A very common use of pipes is to view the output of a command through a pager, like `less`. This allows you to view a screen at a time by pressing the space bar, to move up in the file by using control-u, and to move down using control-d. For example, suppose that you are listing the files in a directory that contains many, many files. If you simply type `ls`, the output will go streaming by too fast to read. But by typing

```
ls | less
```

the output will be displayed one screen at a time, and you can navigate using the commands described above. As another example, suppose we want to find which files we've modified recently. The `-lt` option of the `ls` command will provide a long display of files in reverse chronological order; to display, say the five most recently modified files we could use

```
ls -lt | head -5
```

## 3.9 Job Control

Normally, you type a command into the shell and, while the command is running, you can't type any more commands into that shell. This is known as running in the foreground. When a job is running in the foreground, you can signal it to stop with control-C, which is known as an interrupt signal. Some programs, like R, will catch this signal and do something useful. For example, R will stop working and return to its normal prompt when it receives an interrupt signal. Other programs, (like `ls`) will simply terminate when they receive that signal. Another signal you can send to a foreground program is control-\, which is known

as a kill signal. Programs are not capable of catching this signal, so when you use control-\ it will terminate your foreground program unless there is some larger problem.

When you are sitting in front of the computer, it's not too much of a burden to open a second window, but if you're computing remotely it's nice to be able to run a command and get back the shell prompt to continue working. In addition, when you run jobs in the foreground, they will always terminate when you log out of the computer. The alternative to running in the foreground is running in the background. You can run any command in the background by putting an ampersand (&) at the end of your command. If a job is running in the background and you log off from the computer, it will continue to run, but any output that is not redirected will be lost.

If you've got a job running in the foreground, and you'd like to put it in the background, you need to carry out two separate steps. First, signal the program with control-Z which will suspend the program. Once the program is suspended, and the shell prompt returns, type

**bg**

to put it in the background. Notice that jobs that are suspended with control-Z continue to use resources even if they are not running, so when you really want to stop a job you should use control-C or control-\|. If you want to put a suspended job in the foreground, use the **fg** command.

We've seen how to manage jobs when we still have access to the shell from which we started them, but sometimes you need to find out about jobs for which we don't have access to their originating shells. For example, you may remotely log in to a computer, start a job, log out and then want to find out if it's running, or to stop it. The **ps** command can be used to find out about all the programs that are running on a computer. Remember that for a networked system of computers, the **ps** command will only display information about commands that are running on the particular computer you're using, so if you put a job in the background and log off, it's a very good idea to remember the name of the computer you were using so that you can check on its progress later. To find all the commands you're running on a particular computer type:

```
ps -aux | grep username
```

where **username** is the account name you logged in with. A single line of **ps** output looks like this:

```
spector 1325 10380 0 09:48 pts/10 00:00:00 /bin/bash /usr/local/linux/bin/R
```

Notice that the name of the program that's running is shown at the end of the line. You can identify a particular program by looking for its name there, or by using a second grep command when invoking **ps**. The second column of the output contains the process id or PID of the process; this number uniquely identifies your job on a particular computer. If you want to terminate a job whose pid you know, type

```
kill pid
```

This is similar to control-C. To get the same affect as control-\, type

```
kill -9 pid
```

where `pid` is the number from the `ps` output.

# Chapter 4

## Regular Expressions

## 4.1 Regular Expressions

Regular expressions are a method of describing patterns in text that's far more flexible than using ordinary character strings. While an ordinary text string is only matched by an exact copy of the string, regular expressions give us the ability to describe what we want in more general terms. For example, while we couldn't search for email addresses in a text file using normal searches (unless we knew every possible email address), we can describe the general form of an email address (some characters followed by an "@" sign, followed by some more characters, a period, and a few more characters) through regular expressions, and then find all the email addresses in the document very easily.

Another handy feature of regular expressions is that we can "tag" parts of an expression for extraction. If you look at the HTML source of a web page (for example, by using View-iSource in a browser, or using `download.file` in R to make a local copy), you'll notice that all the clickable links are represented by HTML like:

```
<a href="http://someurl.com/somewhere">
```

It would be easy to search for the string `href=` to find the links, but what if some webmasters used something like

```
<a href = 'http://someurl.com/somewhere'>
```

Now a search for `href=` won't help us, but it's easy to express those sorts of choices using regular expressions.

There are a lot of different versions of regular expressions in the world of computers, and while they share the same basic concepts and much of the same syntax, there are irritating differences among the different versions. If you're looking for additional information about regular expressions in books or on the web, you should know that, in addition to basic regular expressions, recent versions of R also support perl-style regular expressions. (perl is a scripting language whose creator, Larry Wall, developed some attractive extensions to the basic regular expression syntax.) Some of the rules of regular expressions are laid out in very terse language on the R help page for `regex` and `regexp`. Since regular expressions are a somewhat challenging topic, there are many valuable resources on the internet.

Before we start, one word of caution. We'll see that the way that regular expressions work is that they take many of the common punctuation symbols and give them special meanings. Because of this, when you want to refer to one of these symbols literally (that is, as simply a character like other characters), you need to precede those symbols with a backslash (\). But backslashes already have a special meaning in character strings; they are used to indicate control characters, like tab (\t), and newline (\n). The upshot of this is that when you want to type a backslash to keep R from misinterpreting certain symbols, you need to precede it with *two* backslashes in the input. By the way, the characters for which this needs to be done are:

```
. ^ $ + ? * ( ) [ ] { } | \
```

Symbol	Matches	Symbol	Matches
\ w	Alphanumerics and _	\ W	Non-alphanumerics
\ d	Digits	\ D	Non-digits
\ s	Whitespace	\ S	Non-whitespace

To reiterate, when any of these characters is to be interpreted literally in a regular expression, they must be preceded by two backslashes when they are passed to functions in R. If you're not sure, it's almost always safe to add a backslash (by typing two backslashes) in front of a character – if it's not needed, it will be ignored.

Regular expressions are constructed from three different components: literal characters, which will only be matched by the identical literal character, character classes, which are matched by more than one characters, and modifiers which operate on characters, character classes, or combinations of the two. A character class consists of an opening square bracket ([), one or more characters, and a closing square bracket (]), and is matched by any of the characters between the brackets. If the first character inside the brackets is the caret (^), then the character class is matched by anything *except* the other characters between the brackets. Ranges are allowed in character classes and one of the most important examples is [0-9], a character class matched by any of the digits from 0 to 9. Other ranges of characters include [a-z], all the lower case letters, and [A-Z], all the uppercase letters.

There are also some shortcuts for certain character classes that you may or may not find useful. They're summarized in the following table.

Like other cases in R where backslashes need to be interpreted literally, we have to include two backslashes to use the above shortcuts.

Modifier	Meaning
*	zero or more
?	zero or one
+	one or more
{n}	exactly n occurrences
{n,}	at least n occurrences
{n,m}	between n and m occurrences

## 4.2 Regular Expressions

The period (.) represents the wildcard character. Any character (except for the newline character) will be matched by a period in a regular expression; when you literally want a period in a regular expression you need to precede it with a backslash.

Many times you'll need to express the idea of the beginning or end of a line or word in a regular expression. For example, you may be looking for a line number at the beginning of a line, or only be interested in searching for a string if it's at the end of a line. The carat (^) represents the beginning of a line in a regular expression, and the dollar sign (\$) represents the end of a line. To specify that the regular expression you're looking for is a word (i.e. it's surrounded by white space, punctuation or line beginnings and endings), surround the regular expression with escaped angle brackets (<\> and <\>).

One of the most powerful features in regular expressions is being able to specify that you're interested in patterns which may be variable. In the previous example regarding extracting links through the href= argument, we'd like to be able to find a reference no matter how many spaces there are between the href and the =. We also might want to account for the fact that quotes around the url are sometimes omitted. There are a few characters that act as modifiers to the part of the regular expression that precedes them, summarized in the table below.

The first three modifiers in the above table are the most important ones; the others are used much less often. Let's return to the idea of looking for email addresses. Here's a regular expression that will match most email addresses: [-A-Za-z0-9\_.%]+@[ -A-Za-z0-9\_.%]+\\.[A-Za-z]+. In words we could read the regular expression as "one or more occurrences of the characters between the brackets literally followed by an @-sign, followed by one or more characters between the brackets, literally followed by a period, and completed by one or more letters from among A-Z and a-z. You can simplify things by specifying ignore.case=TRUE to most R functions that deal with regular expressions; in that case you would only have to put either A-Z or a-z in the character class, and it would still match both upper or lower case letters.

If we use the regular expression in a call to grep, it will find all of the elements in a character vector that contain an email address:

```
> chk = c('abc noboby@stat.berkeley.edu', 'text with no email',
+ 'first me@mything.com also you@yourspace.com')
> grep('[-A-Za-z0-9_.%]+@[ -A-Za-z0-9_.%]+\\.[A-Za-z]+', chk)
[1] 1 3
```

Since regular expressions in R are simply character strings, we can save typing by storing regular expressions in variables. For example, if we say:

```
> emailpat = '[-A-Za-z0-9_.%]+@[ -A-Za-z0-9_.%]+\\.[A-Za-z]+'
```

then we can use the R variable `emailpat` in place of the full regular expression. (If you use this technique, be sure to modify your stored variable when you change your regular expression.)

To actually get to the regular expressions, we can use the `gregexpr` function, which provides more information about regular expression matches. First, let's see what the output from `gregexpr` looks like:

```
> gregout = gregexpr(emailpat,chk)
> gregout
[[1]]
[1] 5
attr("match.length")
[1] 24

[[2]]
[1] -1
attr("match.length")
[1] -1

[[3]]
[1] 7 27
attr("match.length")
[1] 14 17
```

First, notice that, since there may be a different number of regular expressions found in different lines, `gregexpr` returns a list. In each list element is a vector of starting points where regular expressions were found in the corresponding input string. In addition, there is additional information stored as an attribute, which is part of the value, but which doesn't interfere if we try to treat the value as if it was simply a vector. The `match.length` attribute is another vector, of the same length as the vector of starting points, telling us how long each match was. Concentrating on the first element, we can use the `substring` function to extract the actual address as follows:

```
> substring(chk[1],gregout[[1]],gregout[[1]] + attr(gregout[[1]],'match.length') - 1)
[1] "noboby@stat.berkeley.edu"
```

To make it a little easier to use, let's make a function that will do the extraction for us:

```
getexpr = function(s,g)substring(s,g,g + attr(g,'match.length') - 1)
```

Now it's a little easier to get what we're looking for:

```

> getexpr(chk[2],gregout[[2]])
[1] ""
> getexpr(chk[3],gregout[[3]])
[1] "me@mything.com" "you@yourspace.com"

```

To use the same idea on an entire vector of character strings, we could either write a loop, or use the `mapply` function. The `mapply` function will repeatedly call a function of your choice, cycling through the elements in as many vectors as you provide to the function. To use our `getexpr` function with `mapply` to extract all of the email addresses in the `chk` vector, we could write:

```

> emails = mapply(getexpr,chk,gregout)
> emails
$"abc noboby@stat.berkeley.edu"
[1] "noboby@stat.berkeley.edu"

$"text with no email"
[1] ""

$"first me@mything.com also you@yourspace.com"
[1] "me@mything.com" "you@yourspace.com"

```

Notice that `mapply` uses the text of the original character strings as names for the list it returns; this may or may not be useful. To remove the names, use the assignment form of the `names` function to set the names to `NULL`

```

> names(emails) = NULL
> emails
[[1]]
[1] "noboby@stat.berkeley.edu"

[[2]]
[1] ""

[[3]]
[1] "me@mything.com" "you@yourspace.com"

```

The value that `mapply` returns is a list, the same length as the vector of input strings (`chk` in this example), with an empty string where there were no matches. If all you wanted was a vector of all the email addresses, you could use the `unlist` function:

```

> unlist(emails)
[1] "noboby@stat.berkeley.edu" ""
[3] "me@mything.com"           "you@yourspace.com"

```

The empty strings can be removed in the usual way:

```
emails = emails[emails != '']
```

or

```
emails = subset(emails, emails != '')
```

Suppose we wanted to know how many emails there were in each line of the input text (chk). One idea that might make sense is to find the length of each element of the list that the `getexpr` function returned:

```
> emails = mapply(getexpr, chk, gregout)
> names(emails) = NULL
> sapply(emails, length)
```

The problem is that, in order to maintain the structure of the output list, `mapply` put an empty (length 0) string in the second position of the list, so that `length` sees at least one string in each element of the list. The solution is to write a function that modifies the `length` function so that it only returns the length if there are some characters in the strings for a particular list element. (We can safely do this since we've already seen that there will always be at least one element in the list. We can use the `if` statement to do this:

```
> sapply(emails, function(e) if(nchar(e[1]) > 0) length(e) else 0)
[1] 1 0 2
```

## 4.3 How matches are found

Regular expressions are matched by starting at the beginning of a string and seeing if a possible match might begin there. If not, the next character in the string is examined, and so on; if the end of the string is reached, then no match is reported.

Let's consider the case where there is a potential match. The regular expression program remembers where the beginning of the match was and starts checking the characters to the right of that location. As long as the expression continues to be matched, it will continue, adding more characters to the matched pattern until it reaches a point in the string where the regular expression is no longer matched. At that point, it backs up until things match again, and it checks to see if the entire regular expression has been matched. If it has, it reports a match; otherwise it reports no match.

While the specifics of this mechanism will rarely concern you when you're doing regular expression matches, there is one important point that you should be aware of. The regular expression program is always going to try to find the longest match possible. This means if you use the wildcard character, `.`, with the “one or more” modifier, `*`, you may get more than you expected.

Suppose we wish to remove HTML markup from a web page, in order to extract the information that's on the page. All HTML markup begins with a left angle bracket (`<`) and ends with a right angle bracket (`>`), and has markup information in between. Before diving

in and trying to remove the markup, let's make sure we can find it correctly. An obvious (but ill-advised) choice for the regular expression is `<.*>`. In words it means “a literal left angle bracket, followed by one or more of any character, followed by a literal right angle bracket”. Let's see how effective it is:

```
> htmls = c('This is an image: <IMG SRC="img/rxb.gif">',  
+           '<a href="somepage.html">click here</a>')  
> grep('<.*>',htmls)  
[1] 1 2
```

We've matched both pieces, but what have we really got? We can apply our `mapply`-based solution to see the actual strings that were matched:

```
> getexpr = function(s,g)substring(s,g,g + attr(g,'match.length') - 1)  
> matches = mapply(getexpr,htmls,gregexpr('<.*>',htmls))  
> names(matches) = NULL  
> matches  
[1] "<IMG SRC=\"img/rxb.gif\">"  
[2] "<a href=\"somepage.html\">click here</a>"
```

The first match worked fine, but the second match went right past the end of the first piece of markup, and included the second piece in it. For this reason, regular expressions are said to be **greedy** – they will always try to find the **longest pattern** that satisfies the match. Fixing problems like this is pretty easy – we just have to think about what we really want to match. In this case, we don't want zero or more occurrences of anything; what we want is zero or more occurrences of anything *except* a right angle bracket. As soon as we see the first right angle bracket after the left angle bracket we want to stop. Fortunately, it's very easy to express ideas like this with negated character classes. **In this case, we simply replace the period with `[^>]`:**

```
> matches = mapply(getexpr,htmls,gregexpr('<[^>]*>',htmls))  
> names(matches) = NULL  
> matches  
[[1]]  
[1] "<IMG SRC=\"img/rxb.gif\">"  
  
[[2]]  
[1] "<a href=\"somepage.html\">" "</a>"
```

The two pieces of markup in the second element of `htmls` are now correctly found.

Another way to solve the problem of greedy regular expressions is to use a feature of regular expressions invented by Larry Wall, the creator of perl. His idea was to use the question mark (?) after an asterisk (\*) or plus sign (+) to indicate that you want a **non-greedy** match, that is, to search for the smallest string which will match the pattern. Recent versions of R have incorporated these features into their regular expression support. So an alternative to the solution shown above is:

```
> matches = mapply(getexpr,htmls,gregexpr('<.*?>',htmls))
```

## 4.4 Tagging and Backreferences

Consider again the problem of looking for email addresses. The regular expression that we wrote is exactly what we want, because we don't care what's surrounding the email address. But in many cases, the only way we can find what we want is to specify the surroundings of what we're looking for. Suppose we wish to write a program that will find all of the links (URLs that can be reached by clicking some text on the page) of a web page. A line containing a link may look something like this:

```
<a href="http://www.stat.berkeley.edu">UC Berkeley Stat Dept Home Page</a><br />
```

Finding the links is very easy; but our goal here is to extract the links themselves. Notice that there's no regular expression that can match just the link; we need to use some information about the context in which it's found, and when we extract the matched expression there will be extra characters that we really don't want. To handle this problem, parentheses can be used to surround parts of a regular expression that we're really interested in, and tools exist to help us get those parts separated from the overall expression. In R, the only functions that can deal with these tagged expressions are `sub` and `gsub`, so to take advantage of them, you may have to first extract the regular expressions with the methods we've already seen, and then apply `sub` or `gsub`. To illustrate, let's compose a simple regular expression to find links. I don't need to worry about the case of the regular expression, because the `grep`, `sub`, `gsub` and `gregexpr` functions all support the `ignore.case=` argument. Notice that I've surrounded the part we want (`[^\>]+`) in parentheses. This will allow me to refer to this tagged expression as `\1` in a call to `gsub`. (Additional tagged expressions will be referred to as `\2`, `\3`, etc.) Using this pattern, we can first find all the chunks of text that have our links embedded in them, and then use `gsub` to change the entire piece to just the part we want:

```
> link = '<a href="http://www.stat.berkeley.edu">UC Berkeley Stat Dept Home Page</a><br />'  
> gregout = gregexpr('href *= *["\']?([^\>]+)["\']? *>', link, ignore.case=TRUE)  
> thematch = mapply(getexpr, link, gregout)  
> answer = gsub('href *= *["\']?([^\>]+)["\']? *>', '\1', thematch, ignore.case=TRUE)  
> names(answer) = NULL  
> answer  
[1] "http://www.stat.berkeley.edu"
```

## 4.5 Getting Text into R

Up until now, we've been working with text that's already been formatted to make it easy for R to read, and `scan` or `read.table` (and its associated wrapper functions) have been sufficient to take care of things. Now we want to treat our input as raw text; in other words, we don't want R to assume that the data is in any particular form. The main function for taking care of this in R is `readLines`. In the simplest form, you pass `readLines` the name of a URL or file that you want to read, and it returns a character vector with one element for each line of the file or url, containing the contents of each line. An optional argument to `readLines` specifies the number of lines you want to read; by default it's set to `-1`, which

means to read all available lines. `readLines` removes the newline at the end of each line, but otherwise returns the text exactly the way it was found in the file or URL.

`readLines` also accepts connections, which are objects in R that represent an alternative form of input, such as a pipe or a zipped file. Take a look at the help file for `connections` for more information on this capability.

For moderate-sized problems that aren't too complex, using `readLines` in its default mode (reading all the lines of input into a vector of character strings) will usually be the best way to solve your problem, because most of the functions you'll use with such data are vectorized, and can operate on every line at once. As a simple example, suppose we wanted to get the names of all the files containing notes for this class. A glance at the page <http://www.stat.berkeley.edu/classes/s133/schedule.html> indicates that all the online notes can be found on lines like this one:

```
<tr><td> Jan 23 </td><td> <a href="Unix.html">Introduction to UNIX</a></td></tr>
```

We can easily extract the names of the note files using the `sub` function (since there is only one link per line, we don't need to use `gsub`, although we could).

The first step is to create a vector of character strings that will represents the lines of the URL we are trying to read. We can simply pass the URL name to `readLines`:

```
> x = readLines('http://www.stat.berkeley.edu/classes/s133/schedule.html')
```

Next, we can write a regular expression that can find the links. Note that the pattern that we want (i.e. the name of the file referenced in the link) has been tagged with parentheses for later extraction. By using the caret (^) and dollar sign (\$) we can describe our pattern as an entire line – when we substitute the tagged expression for the pattern we'll have just what we want. I'll also add the basename of the files so that they could be, for example, entered into a browser.

```
> baseurl = 'http://www.stat.berkeley.edu/classes/s133'
> linkpat = '^.*<td> *<a href=[\"\\'](.*)[\"\\']>.*$'
> x = readLines('http://www.stat.berkeley.edu/classes/s133/schedule.html')
> y = grep(linkpat,x,value=TRUE)
> paste(baseurl,sub(linkpat,'\\1',y),sep='/')
[1] "http://www.stat.berkeley.edu/classes/s133/Intro.html"
[2] "http://www.stat.berkeley.edu/classes/s133/OS.html"
[3] "http://www.stat.berkeley.edu/classes/s133/Unix.html"
[4] "http://www.stat.berkeley.edu/classes/s133/R-1a.html"
[5] "http://www.stat.berkeley.edu/classes/s133/R-2a.html"
[6] "http://www.stat.berkeley.edu/classes/s133/R-3a.html"
. . .
```

## 4.6 Examples of Reading Web Pages with R

As an example of how to extract information from a web page, consider the task of extracting the spring baseball schedule for the Cal Bears from <http://calbears.cstv.com/sports/m-basebl/sched/cal-m-basebl-sched.html>.

## 4.7 Reading a web page into R

Read the contents of the page into a vector of character strings with the `readLines` function:

```
> thepage = readLines('http://calbears.cstv.com/sports/m-basebl/sched/cal-m-basebl-sched.html')
Warning message:
In readLines("http://calbears.cstv.com/sports/m-basebl/sched/cal-m-basebl-sched.html") :
  incomplete final line found on 'http://calbears.cstv.com/sports/m-basebl/sched/cal-m-basebl-
```

The warning messages simply means that the last line of the web page didn't contain a newline character. This is actually a good thing, since it usually indicates that the page was generated by a program, which generally makes it easier to extract information from it.

*Note:* When you're reading a web page, make a local copy for testing; as a courtesy to the owner of the web site whose pages you're using, don't overload their server by constantly rereading the page. To make a copy from inside of R, look at the `download.file` function. You could also save a copy of the result of using `readLines`, and practice on that until you've got everything working correctly.

Now we have to focus in on what we're trying to extract. The first step is finding where it is. If you look at the web page, you'll see that the title "Opponent / Event" is right above the data we want. We can locate this line using the `grep` function:

```
> grep('Opponent / Event',thepage)
[1] 513
```

If we look at the lines following this marker, we'll notice that the first date on the schedule can be found in line 536, with the other information following after:

```
> thepage[536:545]
[1] "<td class=\"row-text\">02/20/11</td>" 
[2] "
[3] "<td class=\"row-text\">vs. Utah</td>" 
[4] "
[5] "<td class=\"row-text\">Berkeley, Calif.</td>" 
[6] "
[7] "<td class=\"row-text\">W, 7-0</td>" 
[8] "
[9] "</tr>" 
[10] "
```

Based on the previous step, the data that we want is always preceded by the HTML tag “`<td class="row-text">`”, and followed by “`</td>`”. Let’s grab all the lines that have that pattern:

```
> mypattern = '<td class="row-text">([^<]*)</td>'  
> datalines = grep(mypattern, thepage[536:length(thepage)], value=TRUE)
```

I used `value=TRUE`, so I wouldn’t have to worry about the indexing when I restricted myself to the lines from 550 on. Also notice that I’ve already tagged the part that I want, in preparation to the final call to `gsub`.

Now that I’ve got the lines where my data is, I can use `gregexpr`, then `getexpr` (from the previous lecture), and `gsub` to extract the information without the HTML tags:

```
> getexpr = function(s,g) substring(s,g,g+attr(g,'match.length')-1)  
> gg = gregexpr(mypattern,datalines)  
> matches = mapply(getexpr,datalines,gg)  
> result = gsub(mypattern,'\\1',matches)  
> names(result) = NULL  
> result[1:10]  
[1] "02/19/11"      "vs. Utah"       "Evans Diamond" "1:00 p.m. PT"  
[5] "02/20/11"      "vs. Utah"       "Evans Diamond" "1:00 p.m. PT"  
[9] "02/22/11"      "at Stanford"
```

It seems pretty clear that we’ve extracted just what we wanted – to make it more usable, we’ll convert it to a data frame and provide some titles. Since it’s hard to describe how to convert a vector to a data frame, we’ll use a matrix as an intermediate step. Since there are four pieces of information (columns) for each game (row), a matrix is a natural choice:

```
> schedule = as.data.frame(matrix(result,ncol=4,byrow=TRUE))  
> names(schedule) = c('Date','Opponent','Location','Result')  
> head(schedule)  
    Date          Opponent        Location      Result  
1 02/19/11      vs. Utah  Evans Diamond  1:00 p.m. PT  
2 02/20/11      vs. Utah  Evans Diamond  1:00 p.m. PT  
3 02/22/11      at Stanford Stanford, Calif. 5:30 p.m. PT  
4 02/25/11      at Coastal Carolina Conway, S.C. 4:00 p.m. ET  
5 02/26/11      vs. Kansas State Conway, S.C. 11:00 a.m. ET  
6   &nbsp; vs. North Carolina State Conway, S.C. 11:30 a.m. ET
```

## 4.8 Another Example

At <http://www.imdb.com/chart> is a box-office summary of the ten top movies, along with their gross profits for the current weekend, and their total gross profits. We would like to make a data frame with that information. As always, the first part of the solution is to read

the page into R, and use an anchor to find the part of the data that we want. In this case, the table has column headings, including one for “Rank”.

```
> x = readLines('http://www.imdb.com/chart/')
> grep('Rank',x)
[1] 1294 1546 1804
```

Starting with line 1294 we can look at the data to see where the information is. A little experimentation shows that the useful data starts on line 1310:

```
> x[1310:1318]
[1] "      <td class=\"chart_even_row\">
[2] "          <a href=\"/title/tt0990407/\">The Green Hornet</a> (2011)"
[3] "      </td>"
[4] "      <td class=\"chart_even_row\" style=\"text-align: right; padding-right: 20px\"
[5] "          $33.5M"
[6] "      </td>"
[7] "      <td class=\"chart_even_row\" style=\"text-align: right\">>"
[8] "          $40M"
[9] "      </td>"
[10] "     </tr>"
```

There are two types of lines that contain useful data: the ones with the title, and the ones that begin with some blanks followed by a dollar sign. Here's a regular expression that will pull out both those lines:

```
> goodlines = '<a href="/title[^>]*>(.*)</a>.*$|^ *\\$'
> try = grep(goodlines,x,value=TRUE)
```

Looking at the beginning of `try`, it seems like we got what we want:

```
> try[1:10]
[1] "          <a href=\"/title/tt1564367/\">Just Go with It</a> (2011)"
[2] "          $30.5M"
[3] "          $30.5M"
[4] "          <a href=\"/title/tt1702443/\">Justin Bieber: Never Say Never</a> (2011)"
[5] "          $29.5M"
[6] "          $30.3M"
[7] "          <a href=\"/title/tt0377981/\">Gnomeo & Juliet</a> (2011)"
[8] "          $25.4M"
[9] "          $25.4M"
[10] "         <a href=\"/title/tt1034389/\">The Eagle</a> (2011)"
```

Sometimes the trickiest part of getting the data off a webpage is figuring out exactly the part you need. In this case, there is a lot of extraneous information after the table we want. By examining the output, we can see that we only want the first 30 entries. We also need to remove the extra information from the title line. We can use the `sub` function with a modified version of our regular expression:

```

> try = try[1:30]
> try = sub('<a href="/title[^>]*>(.*)</a>.*$', '\1', try)
> head(try)
[1] "                Just Go with It"
[2] "                $30.5M"
[3] "                $30.5M"
[4] "                Justin Bieber: Never Say Never"
[5] "                $29.5M"
[6] "                $30.3M"

```

Once the spaces at the beginning of each line are removed, we can rearrange the data into a 3-column data frame:

```

> try = sub('^\s+', '', try)
> movies = data.frame(matrix(try, ncol=3, byrow=TRUE))
> names(movies) = c('Name', 'Wkend Gross', 'Total Gross')
> head(movies)

```

	Name	Wkend Gross	Total Gross
1	Just Go with It	\$30.5M	\$30.5M
2	Justin Bieber: Never Say Never	\$29.5M	\$30.3M
3	Gnomeo & Juliet	\$25.4M	\$25.4M
4	The Eagle	\$8.68M	\$8.68M
5	The Roommate	\$8.13M	\$25.8M
6	The King's Speech	\$7.23M	\$93.7M

We can replace the special characters with the following code:

```

> movies$name = sub('&#x26;', '&', movies$name)
> movies$name = sub('&#x27;', '\'', movies$name)

```

## 4.9 Dynamic Web Pages

While reading data from static web pages as in the previous examples can be very useful (especially if you're extracting data from many pages), the real power of techniques like this has to do with dynamic pages, which accept queries from users and return results based on those queries. For example, an enormous amount of information about genes and proteins can be found at the National Center of Biotechnology Information website (<http://www.ncbi.nlm.nih.gov/>), much of it available through query forms. If you're only performing a few queries, it's no problem using the web page, but for many queries, it's beneficial to automate the process.

Here is a simple example that illustrate the concept of accessing dynamic information from web pages. The page <http://finance.yahoo.com> provides information about stocks; if you enter a stock symbol on the page, (for example `aapl` for Apple Computer), you will be directed to a page whose URL (as it appears in the browser address bar) is

```
http://finance.yahoo.com/q?s=aapl&x=0&y=0
```

The way that stock symbols are mapped to this URL is pretty obvious. We'll write an R function that will extract the current price of whatever stock we're interested in.

The first step in working with a page like this is to download a local copy to play with, and to read the page into a vector of character strings:

```
> download.file('http://finance.yahoo.com/q?s=aapl&x=0&y=0', 'quote.html')
trying URL 'http://finance.yahoo.com/q?s=aapl&x=0&y=0'
Content type 'text/html; charset=utf-8' length unknown
opened URL
..... .
downloaded 39Kb
> x = readLines('quote.html')
```

To get a feel for what we're looking for, notice that the words "Last Trade:" appear before the current quote. Let's look at the line containing this string:

```
> grep('Last Trade:', x)
45
> nchar(x[45])
[1] 3587
```

Since there are over 3500 characters in the line, we don't want to view it directly. Let's use `gregexpr` to narrow down the search:

```
> gregexpr('Last Trade:', x[45])
[[1]]
[1] 3125
attr(", "match.length")
[1] 11
```

This shows that the string "Last Trade:" starts at character 3125. We can use `substr` to see the relevant part of the line:

```
> substring(x[45], 3125, 3220)
[1] "Last Trade:</th><td class=\"yfnc_tabledata1\"><big><b><span id=\"yfs_l10_aapl\">363
```

There's plenty of context – we want the part surrounded by `<big><b><span ...>` and `</span>`. One easy way to grab that part is to use a tagged regular expression with `gsub`:

```
> gsub('^.*<big><b><span [^>]*>([<^>]*)</span>.*$', '\1', x[45])
[1] "363.50"
```

This suggests the following function:

```

> getquote = function(sym){
+   baseurl = 'http://finance.yahoo.com/q?s='
+   myurl = paste(baseurl,sym,'&x=0&y=0',sep=' ')
+   x = readLines(myurl)
+   q = gsub('^.*<b><big><span [^>]*>([<^>]*)</span>.*$', '\1', grep('Last Trade:',x,value=TRUE))
+   as.numeric(q)
+}

```

As always, functions like this should be tested:

```

> getquote('aapl')
[1] 196.19
> getquote('ibm')
[1] 123.21
> getquote('nok')
[1] 13.35

```

These functions provide only a single quote; a little exploration of the yahoo finance website shows that we can get CSV files with historical data by using a URL of the form:

<http://ichart.finance.yahoo.com/table.csv?s=xxx>

where `xxx` is the symbol of interest. Since it's a comma-separated file, We can use `read.csv` to read the chart.

```

gethistory = function(symbol)
  read.csv(paste('http://ichart.finance.yahoo.com/table.csv?s=',symbol,sep=' '))

```

Here's a simple test:

```

> aapl = gethistory('aapl')
> head(aapl)
  Date  Open  High  Low Close Volume Adj.Close
1 2011-02-15 359.19 359.97 357.55 359.90 10126300    359.90
2 2011-02-14 356.79 359.48 356.71 359.18 11073100    359.18
3 2011-02-11 354.75 357.80 353.54 356.85 13114400    356.85
4 2011-02-10 357.39 360.00 348.00 354.54 33126100    354.54
5 2011-02-09 355.19 359.00 354.87 358.16 17222400    358.16
6 2011-02-08 353.68 355.52 352.15 355.20 13579500    355.20

```

Unfortunately, if we try to use the `Date` column in plots, it will not work properly, since R has stored it as a factor. The format of the date is the default for the `as.Date` function, so we can modify our function as follows:

```

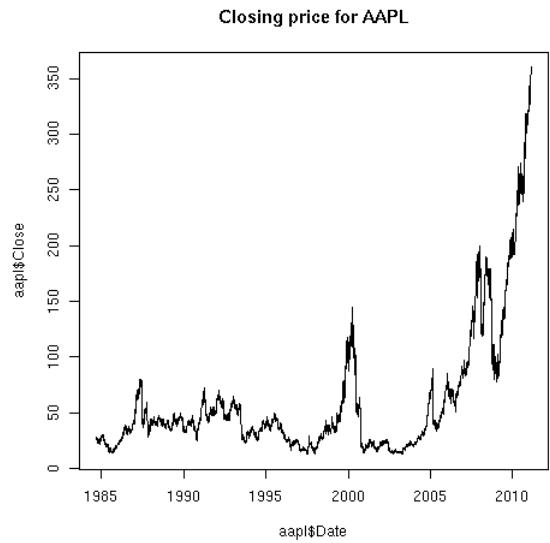
gethistory = function(symbol){
  data = read.csv(paste('http://ichart.finance.yahoo.com/table.csv?s=',symbol,sep=' '))
  data$date = as.Date(data$date)
  data
}

```

Now, we can produce plots with no problems:

```
> aapl = gethistory('aapl')
> plot(aapl$Date,aapl$Close,main='Closing price for AAPL',type='l')
```

The plot is shown below:



## 4.10 Writing a Function

```
plothistory = function(symbol,what){  
  match.arg(what,c('Open','High','Low','Close','Volume','Adj.Close'))  
  data = gethistory(symbol)  
  plot(data$date,data[,what],main=paste(what,'price for ',symbol),type='l')  
  invisible(data)  
}
```

This function introduces several features of functions that we haven't seen yet.

- The `match.arg` function lets us specify a list of acceptable values for a parameter being passed to a function, so that an error will occur if you use a non-existent variable:

```
> plothistory('aapl','Last')  
Error in match.arg("what", c("Open", "High", "Low", "Close", "Volume", :  
  'arg' should be one of Open, High, Low, Close, Volume, Adj.Close
```

- The `invisible` function prevents the returned value (`data` in this case) from being printed when the output of the function is not assigned, but also allows us to assign the output to a variable if we want to use it latter:

```
> plothistory('aapl','Close')
```

will produce a plot without displaying the data to the screen, but

```
> aapl.close = plothistory('aapl','Close')
```

will produce the plot, *and* store the data in `aapl.close`.

## 4.11 Another Example

When using google, it's sometimes inconvenient to have to click through all the pages. Let's write a function that will return the web links from a google search. If you type a search term into google, for example a search for 'introduction to r', you'll notice that the address bar of your browser looks something like this:

```
http://www.google.com/search?q=introduction+to+r&ie=utf-8&oe=utf-8&aq=t&rls=com.ubuntu:en
```

For our purposes, we only need the "q=" part of the search. For our current example, that would be the URL

```
http://www.google.com/search?q=introduction+to+r
```

Note that, since blanks aren't allowed in URLs, plus signs are used in place of spaces. If we were to click on the "Next" link at the bottom of the page, the URL changes to something like

```
http://www.google.com/search?hl=en&safe=active&client=firefox-a&rls=com.ubuntu:en-US:uno
```

For our purposes, we only need to add the `&start=` argument to the web page. Since google displays 10 results per page, the second page will have `start=10`, the next page will have `start=20`, and so on. Let's read in the first page of this search into R:

```
z = readLines('http://www.google.com/search?q=introduction+to+r')
Warning message:
In readLines("http://www.google.com/search?q=introduction+to+r") :
  incomplete final line found on 'http://www.google.com/search?q=introduction+to+r'
```

As always, you can safely ignore the message about the incomplete final line.

Since we're interested in the web links, we only want lines with "`href=`" in them. Let's check how many lines we've got, how long they are, and which ones contain the href string:

```
> length(z)
[1] 17
> nchar(z)
 [1] 369   208   284    26   505    39 40605 1590   460   291   152   248
[13] 317   513   507     5     9
> grep('href=' ,z)
[1] 5 7
```

It's pretty clear that all of the links are on the seventh line.

Now we can construct a tagged regular expression to grab all the links.

```
> hrefpat = 'href *= *"( [^"]*)"'
> getexpr = function(s,g)substring(s,g,g+attr(g,'match.length')-1)
> gg = gregexpr(hrefpat,z[[7]])
> res = mapply(getexpr,z[[7]],gg)
> res = sub(hrefpat,'\\1',res)
> res[1:10]
[1] "http://images.google.com/images?q=introduction+to+r&um=1&ie=UTF-8&sa=N&hl=en&tab=w"
[2] "http://video.google.com/videosearch?q=introduction+to+r&um=1&ie=UTF-8&sa=N&hl=en&tab=v"
[3] "http://maps.google.com/maps?q=introduction+to+r&um=1&ie=UTF-8&sa=N&hl=en&tab=wl"
[4] "http://news.google.com/news?q=introduction+to+r&um=1&ie=UTF-8&sa=N&hl=en&tab=wn"
[5] "http://www.google.com/products?q=introduction+to+r&um=1&ie=UTF-8&sa=N&hl=en&tab=wf"
[6] "http://mail.google.com/mail/?hl=en&tab=wm"
[7] "http://www.google.com/intl/en/options/"
[8] "/preferences?hl=en"
[9] "https://www.google.com/accounts/Login?hl=en&continue=http://www.google.com/search%20"
[10] "http://www.google.com/webhp?hl=en"
```

We don't want the internal (google) links – we want external links which will begin with "http://". Let's extract all the external links, and then eliminate the ones that just go back to google:

```
> refs = res[grep('^https?:',res)]
> refs = refs[-grep('google.com/',refs)]
> refs[1:3]
[1] "http://cran.r-project.org/doc/manuals/R-intro.pdf"
[2] "http://74.125.155.132/search?q=cache:d4-KmcWVA-oJ:cran.r-project.org/doc/manuals/R-intro.pdf"
[3] "http://74.125.155.132/search?q=cache:d4-KmcWVA-oJ:cran.r-project.org/doc/manuals/R-intro.pdf"
```

If you're familiar with google, you may recognize these as the links to google's cached results. We can easily eliminate them:

```
> refs = refs[-grep('cache:',refs)]
> length(refs)
[1] 10
```

We can test these same steps with some of the other pages from this query:

```
> z = readLines('http://www.google.com/search?q=introduction+to+r&start=10')
Warning message:
In readLines("http://www.google.com/search?q=introduction+to+r&start=10") :
  incomplete final line found on 'http://www.google.com/search?q=introduction+to+r&start=10'
> hrefpat = 'href *= *"(["]*")"'
> getexpr = function(s,g)substring(s,g,g+attr(g,'match.length')-1)
> gg = gregexpr(hrefpat,z[[7]])
> res = mapply(getexpr,z[[7]],gg)
Error in substring(s, g, g + attr(g, "match.length") - 1) :
  invalid multibyte string at '<93>GNU S'
```

Unfortunately, there seems to be a problem. Fortunately, it's easy to fix. What the message is telling us is that there's a character in one of the results that doesn't make sense in the language (English) that we're using. We can solve this by typing:

```
> Sys.setlocale('LC_ALL','C')
> res = mapply(getexpr,z[[7]],gg)
```

Since we no longer get the error, we can continue

```
> res = sub(hrefpat,'\\1',res)
> refs = res[grep('^https?:',res)]
> refs = refs[-grep('google.com/',refs)]
> refs = refs[-grep('cache:',refs)]
> length(refs)
[1] 10
```

Once again, it found all ten links. This obviously suggests a function:

```
googlerefs = function(term,pg=0){  
  getexpr = function(s,g)substring(s,g,g+attr(g,'match.length')-1)  
  qurl = paste('http://www.google.com/search?q=',term,sep=' ')  
  if(pg > 0)qurl = paste(qurl,'&start=',pg * 10,sep=' ')  
  qurl = gsub(' ','+',qurl)  
  z = readLines(qurl)  
  hrefpat = 'href *= *"(["]*")"'  
  wh = grep(hrefpat,z)[2]  
  gg = gregexpr(hrefpat,z[[wh]])  
  res = mapply(getexpr,z[[wh]],gg)  
  res = sub(hrefpat,'\\1',res)  
  refs = res[grep('^https?:',res)]  
  refs = refs[-grep('google.com/|cache:',refs)]  
  names(refs) = NULL  
  refs[!is.na(refs)]  
}
```

Now suppose that we want to retrieve the links for the first ten pages of query results:

```
> links = sapply(0:9,function(pg)googlerefs('introduction to r',pg))  
> links = unlist(links)  
> head(links)  
[1] "http://cran.r-project.org/doc/manuals/R-intro.pdf"  
[2] "http://cran.r-project.org/manuals.html"  
[3] "http://www.biostat.wisc.edu/~kbroman/Rintro/"  
[4] "http://faculty.washington.edu/tlumley/Rcourse/"  
[5] "http://www.stat.cmu.edu/~larry/all-of-statistics/=R/Rintro.pdf"  
[6] "http://www.stat.berkeley.edu/~spector/R.pdf"
```

# Chapter 5

## Graphics

## 5.1 More on Plotting

We've seen the usage of some basic graphics functions in previous lectures, but there are still a few points that should be covered before we study more advanced graphics commands. One issue has to do with multiple lines on a plot. In addition to the high-level `plot` command, R provides the `points` and `lines` functions, which can add new data to a plot. Consider the nationwide Community Health Data, available at the website <http://communityhealth.hhs.gov>. The data is available in a zip file, which contains a number of CSV files containing information recorded for each county in the USA. One such file, containing information about various risk factors and access to health care in the counties can be found on the class website: <http://www.stat.berkeley.edu/classes/s133/data/RISKFACTORSANDACCESSTOCARE.csv>. First, we'll read in the data, and look at the first few records:

```
> risk = read.csv('http://www.stat.berkeley.edu/classes/s133/data/RISKFACTORSANDACCESSTOCARE.csv')
> head(risk)
  State_FIPS_Code County_FIPS_Code CHSI_County_Name CHSI_State_Name
1                  1                  1        Autauga      Alabama
2                  1                  3       Baldwin      Alabama
3                  1                  5     Barbour      Alabama
4                  1                  7        Bibb      Alabama
5                  1                  9      Blount      Alabama
6                  1                 11    Bullock      Alabama
  CHSI_State_Abbr Strata_ID_Number No_Exercise CI_Min_No_Exercise
1             AL              29        27.8            20.7
2             AL              16        27.2            23.2
3             AL              51     -1111.1           -1111.1
4             AL              42     -1111.1           -1111.1
5             AL              28        33.5            26.3
6             AL              75     -1111.1           -1111.1
  CI_Max_No_Exercise Few_Fruit_Veg CI_Min_Fruit_Veg CI_Max_Fruit_Veg Obesity
1            34.9          78.6          69.4          87.8      24.5
2            31.2          76.2          71.2          81.3      23.6
3          -1111.1         -1111.1         -1111.1         -1111.1      25.6
4          -1111.1          86.6          77.8          95.4 -1111.1
5            40.6          74.6          66.1          83.0      24.2
6          -1111.1         -1111.1         -1111.1         -1111.1 -1111.1
  CI_Min_Obesity CI_Max_Obesity High_Blood_Pres CI_Min_High_Blood_Pres
1            17.3          31.7          29.1          19.2
2            19.5          27.6          30.5          24.5
3            16.2          35.0         -1111.1         -1111.1
4          -1111.1         -1111.1         -1111.1         -1111.1
5            17.2          31.2         -1111.1         -1111.1
6          -1111.1         -1111.1         -1111.1         -1111.1
```

	CI_Max_High_Blood_Pres	Smoker	CI_Min_Smoker	CI_Max_Smoker	Diabetes
1	39.0	26.6	19.1	34.0	14.2
2	36.6	24.6	20.3	28.8	7.2
3	-1111.1	17.7	10.2	25.1	6.6
4	-1111.1	-1111.1	-1111.1	-1111.1	13.1
5	-1111.1	23.6	16.7	30.4	8.4
6	-1111.1	-1111.1	-1111.1	-1111.1	-1111.1
	CI_Min_Diabetes	CI_Max_Diabetes	Uninsured	Elderly_Medicare	Disabled_Medicare
1	9.1	19.3	5690	4762	1209
2	5.2	9.3	19798	22635	3839
3	2.0	11.3	5126	3288	1092
4	4.7	21.5	3315	2390	974
5	4.4	12.4	8131	5019	1300
6	-1111.1	-1111.1	2295	1433	504
	Prim_Care_Phys_Rate	Dentist_Rate	Community_Health_Center_Ind	HPSA_Ind	
1	45.3	22.6		1	2
2	67.0	30.8		1	2
3	45.8	24.6		1	2
4	41.8	18.6		1	1
5	16.2	10.8		2	1
6	54.3	18.1		1	1

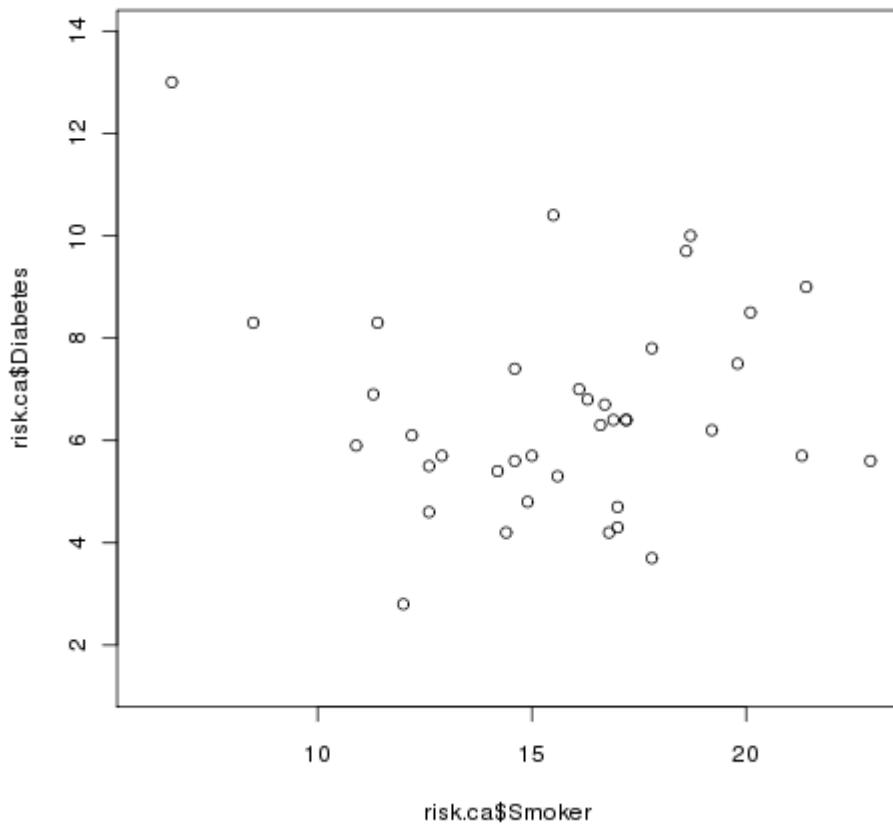
It's clear that the value `-1111.1` is being used for missing values, and we'll need to fix that before we work with the data:

```
> risk[risk == -1111.1] = NA
```

Suppose we want to investigate the relationship between Diabetes and Smoking in each of the counties in California. We could create a data set with only California using the `subset` command, and then plot the two variables of interest:

```
> risk.ca = subset(risk, CHSI_State_Name == 'California')
> plot(risk.ca$Smoker,risk.ca$Diabetes)
```

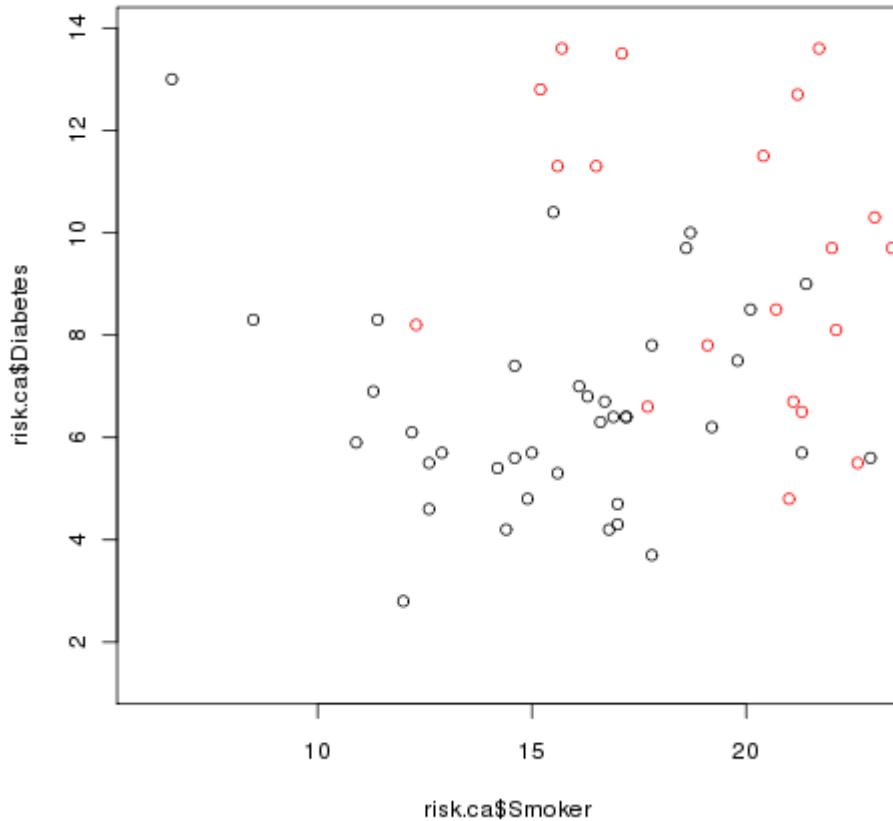
Here's what the plot looks like:



Now let's say that we wanted to examine the relationship between smoking and diabetes for some other state, say Alabama. We can extract the Alabama data using subset, and then use the `points` command to add that data to the existing plot. (Unlike `plot`, `points` doesn't produce a new plot, it adds to the currently active plot.)

```
> risk.al = subset(risk, CHSI_State_Name == 'Alabama')
> points(risk.al$Smoker,risk.al$Diabetes,col='red')
```

The plot now looks like this:

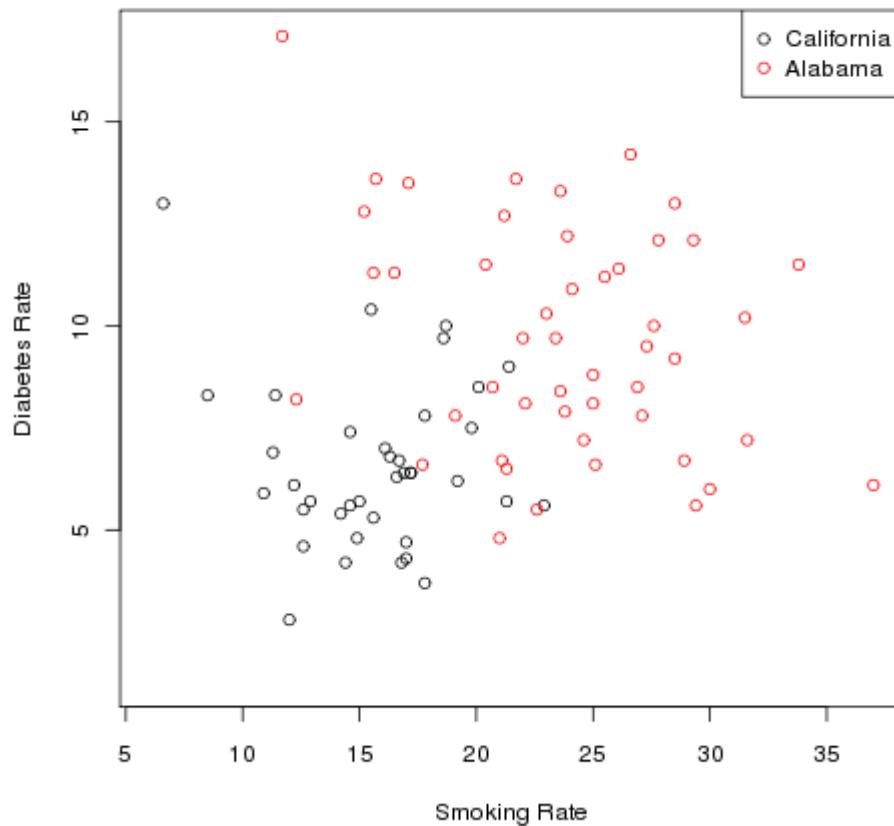


Clearly there's a problem: some of the Alabama points are off the scale. This demonstrates that when you wish to plot multiple sets of points on the same graph that you have to make sure that the axes are big enough to accomodate all of the data. One very easy way to do this is to call the `plot` function with the minimums and maximums of all the data using `type='n'` as one of the arguments. This tells R to set up the axes, but not to actually plot anything. So a better way to plot the two sets of points would be as follows:

```
> plot(range(c(risk.ca$Smoker,risk.al$Smoker),na.rm=TRUE),
+      range(c(risk.ca$Diabetes,risk.al$Diabetes),na.rm=TRUE),
+      type='n',xlab='Smoking Rate',ylab='Diabetes Rate')
> points(risk.ca$Smoker,risk.ca$Diabetes)
> points(risk.al$Smoker,risk.al$Diabetes,col='red')
> legend('topright',c('California','Alabama'),col=c('black','red'),pch=1)
> title('Diabetes Rate vs. Smoking by County')
```

The completed plot looks like this:

### Diabetes Rate vs. Smoking by County



To add a line to a plot, the `lines` function can be used in place of the `points` function. The built-in data set `Puromycin` provides data on concentration and reaction rate of Puromycin on two types of cells, treated and untreated. You can learn more about R's builtin data sets by using the `data()` function, and then using the usual help facility within R. Since this is a small data set (only 23 rows), we can examine it directly:

```
> Puromycin
  conc rate      state
1  0.02   76  treated
2  0.02   47  treated
3  0.06   97  treated
4  0.06  107  treated
5  0.11  123  treated
6  0.11  139  treated
7  0.22  159  treated
8  0.22  152  treated
9  0.56  191  treated
10 0.56  201  treated
```

```

11 1.10 207   treated
12 1.10 200   treated
13 0.02  67   untreated
14 0.02  51   untreated
15 0.06  84   untreated
16 0.06  86   untreated
17 0.11  98   untreated
18 0.11  115  untreated
19 0.22  131  untreated
20 0.22  124  untreated
21 0.56  144  untreated
22 0.56  158  untreated
23 1.10  160  untreated

```

Since there are two observations at each concentration, we can use the `aggregate` function to calculate the mean before plotting the data:

```
> Puro = aggregate(list(rate=Puromycin$rate),list(conc=Puromycin$conc,state=Puromycin$st
```

By putting the `rate` variable in a named list, I insure that the column with the mean will be called “`rate`”. Alternatively, I could rename the column later:

```
> Puro = aggregate(Puromycin$rate,list(conc=Puromycin$conc,state=Puromycin$state),mean)
> names(Puro)[3] = 'rate'
```

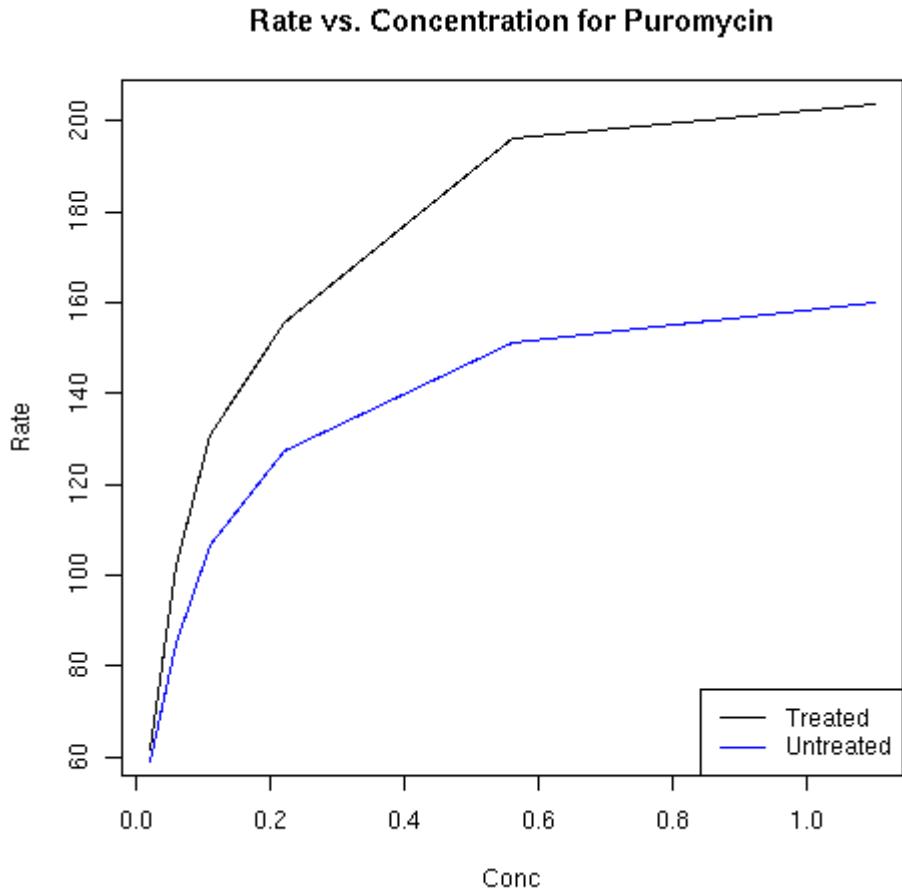
Now we can create two data frames, one for the treated cells, and one for the untreated ones:

```
> Puro.treated = subset(Puro,state=='treated')
> Puro.untreated = subset(Puro,state=='untreated')
```

Examination of the data shows that the rate for the treated cells are higher than the untreated cells. Thus, instead of creating an empty plot as in the previous example, I'll just plot the line for the treated cells first:

```
> plot(Puro.treated$conc,Puro.treated$rate,xlab='Conc',ylab='Rate',main='Rate vs. Concentration')
> lines(Puro.untreated$conc,Puro.untreated$rate,col='blue')
> legend('bottomright',c('Treated','Untreated'),col=c('black','blue'),lty=1)
```

The plot appears below:



## 5.2 Multiple Plots on a Page

The `mfrow` or `mfcol` arguments to the `par` function allow you to divide the plotting page into a grid, to accomodate multiple plots on a single page. The layout of the plot is determined by a vector of length 2, specifying the number of rows and columns in the grid. The difference between `mfrow` and `mfcol` concerns the order in which the plots are drawn. When using `mfrow`, plots are drawn across the rows, while `mfcol` draws the plot down the columns.

As an example, consider the builtin `airquality` data set, which contains various air quality measures for a five month period in New York. We can get an idea of what the data is like by using the `summary` function:

```
> summary(airquality)
   Ozone          Solar.R          Wind          Temp
Min. : 1.00  Min. : 7.0  Min. : 1.700  Min. :56.00
1st Qu.:18.00 1st Qu.:115.8 1st Qu.: 7.400 1st Qu.:72.00
Median :31.50 Median :205.0 Median : 9.700 Median :79.00
```

```

Mean      : 42.13   Mean     :185.9    Mean     : 9.958   Mean     :77.88
3rd Qu.: 63.25   3rd Qu.:258.8    3rd Qu.:11.500   3rd Qu.:85.00
Max.     :168.00   Max.     :334.0    Max.     :20.700   Max.     :97.00
NA's     : 37.00   NA's     :    7.0

      Month           Day
Min.    :5.000   Min.    : 1.00
1st Qu.:6.000   1st Qu.: 8.00
Median  :7.000   Median  :16.00
Mean    :6.993   Mean    :15.80
3rd Qu.:8.000   3rd Qu.:23.00
Max.    :9.000   Max.    :31.00

```

Suppose we want to plot histograms of each of the first four variables, side by side with a plot of the variable over time. The first step is to convert the `Month` and `Date` variables into R `Date` variables:

```
> airquality$date = as.Date(ISOdate(1973,airquality$Month,airquality$Day))
```

Rather than typing in the plotting commands repeatedly, we can write a function to make the plots for each variable:

```
twoplot = function(var){
  plot(density(airquality[,var],na.rm=TRUE),main=paste('Density of ',var))
  plot(airquality$date,airquality[,var],type='l',main=paste(var,'vs. time'),ylab=var)
}
```

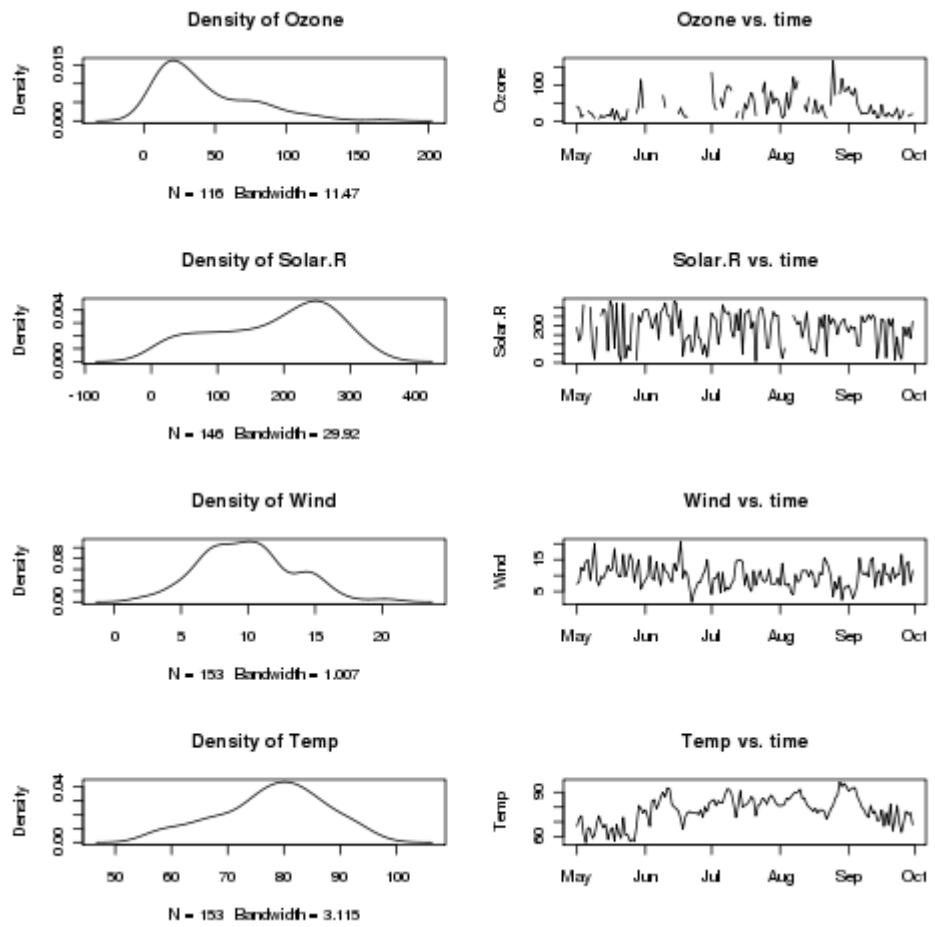
Since we'll be plotting the first four variables, we'll want a 4x2 grid, and we'll want to plot by rows. Thus, we issue the following command:

```
> par(mfrow=c(4,2))
```

We can use `sapply` to call our `twoplot` function for each of the variables:

```
> sapply(names(airquality)[1:4],twoplot)
      Ozone Solar.R Wind Temp
[1,] 1216    1216 1216 1216
[2,] 1247    1247 1247 1247
[3,] 1277    1277 1277 1277
[4,] 1308    1308 1308 1308
[5,] 1339    1339 1339 1339
[6,] 1369    1369 1369 1369
```

Here's the plot:



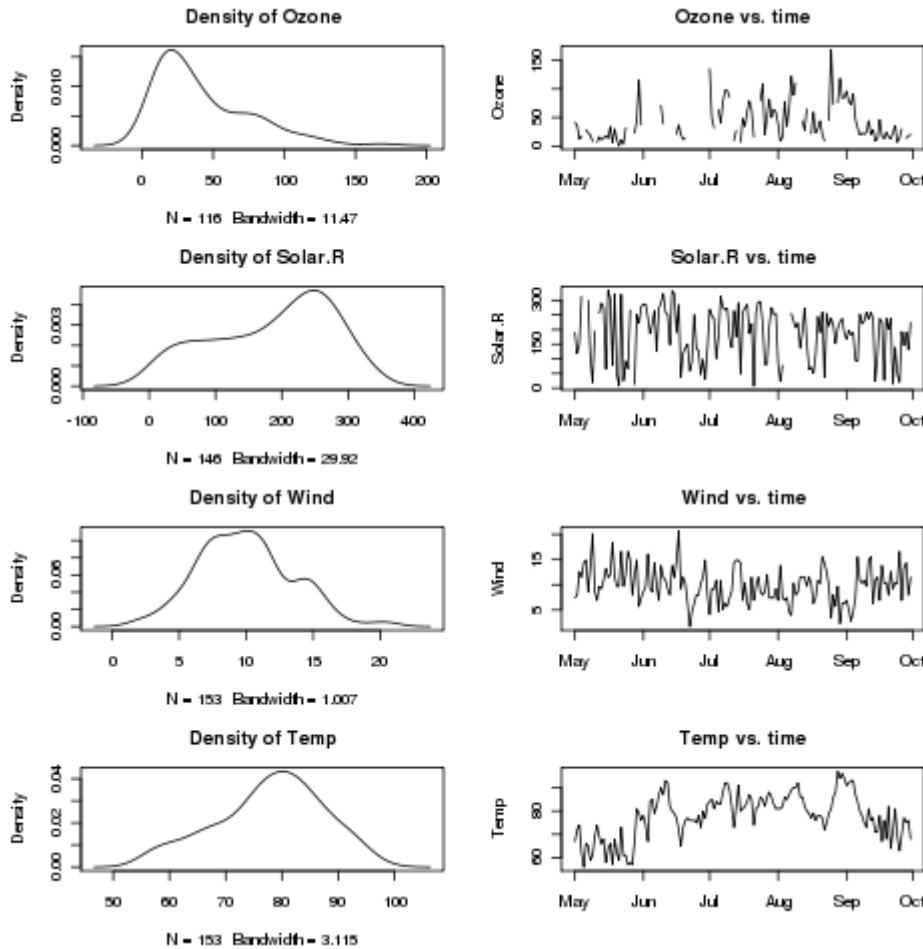
As can be seen, the default for multiple plots leaves quite a bit of space between the plots. The graphics parameter that controls this is called `mar`. The value that determines the spacing is a vector of length 4, with the number of lines of space on the bottom, left, top, and right of the plots. We can find the current value of graphics parameters by passing the name of the parameter to `par`:

```
> par('mar')
[1] 5.1 4.1 4.1 2.1
```

Let's remove one line from each of the top and bottom, and replot the series of graphs:

```
> par(mar=c(4.1,4.1,3.1,2.1),mfrow=c(4,2))
> sapply(names(airquality)[1:4],twoplot)
      Ozone Solar.R Wind Temp
[1,] 1216    1216 1216 1216
[2,] 1247    1247 1247 1247
[3,] 1277    1277 1277 1277
[4,] 1308    1308 1308 1308
[5,] 1339    1339 1339 1339
[6,] 1369    1369 1369 1369
```

The plot is shown below:



After plotting multiple plots, you can restore R's normal behaviour of plotting each plot on a separate page by resetting `mfrow` as follows:

```
> par(mfrow=c(1,1))
```

## 5.3 More on Barplots

We briefly looked at barplots when we examined the day of the week on which popular movies opened. You may recall that we passed the result of a call from the `table` function to `barplot`. This idea can be extended to create side-by-side or stacked barplots. As an alternative to using `table`, we can produce a matrix with the values we wish to plot. By providing `dimnames` for this matrix, `barplot` will be able to appropriately label the plot.

Once again consider the `risk` data set we used earlier. To simplify the plot, the different states in the data set can be grouped into regions, using the built-in `state.name` and `state.region` objects:

```
> risk = merge(risk,data.frame(state=state.name,region=state.region),
+               by.x='CHSI_State_Name',by.y='state')
```

Suppose we're interested in comparing the average smoking, diabetes, and obesity rates for the different regions. First, we use the `aggregate` function to find the averages of the variables, broken down by regions:

```
> mns = aggregate(risk[,c('Smoker','Diabetes','Obesity')],risk['region'],  
+                  mean,na.rm=TRUE)  
> mns  
    region   Smoker Diabetes Obesity  
1   Northeast 22.62537 7.161972 22.13430  
2      South 25.15847 8.754302 25.71493  
3 North Central 22.28592 7.318565 24.16283  
4       West 19.50740 6.349167 21.01306
```

If we try to pass this data frame to `barplot`, we'll get an error, since `barplot` needs a matrix. In addition, the column that identifies the different groups (in this case `region`), needs to be part of the row names, not a variable in the data frame itself. These two steps create the row names, and eliminate the `region` column from the data frame:

```
> row.names(mns) = mns$region  
> mns$region = NULL
```

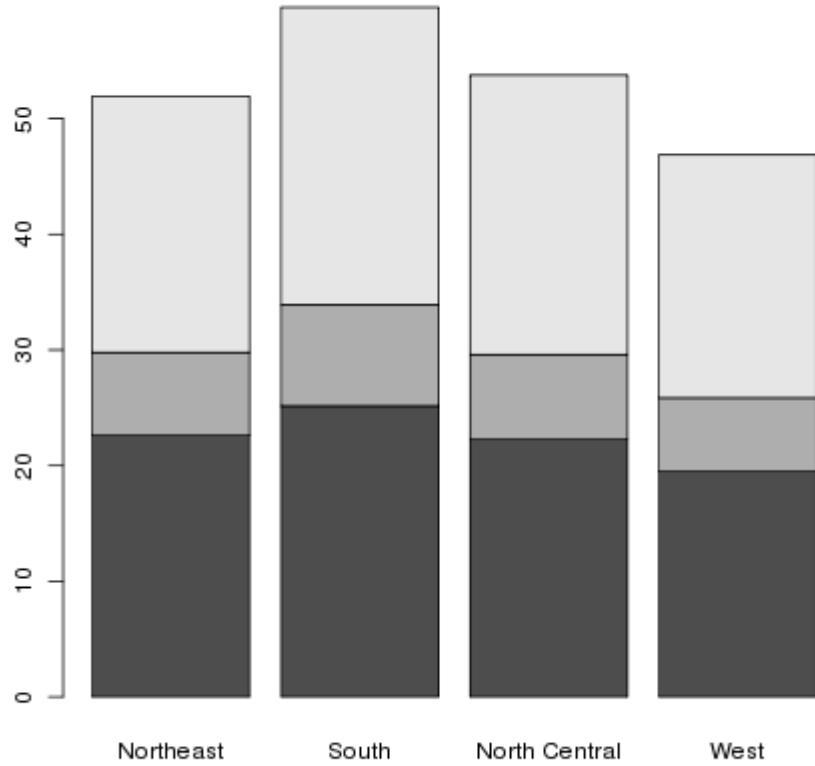
If we were to use the following statement to create the barplot:

```
> barplot(as.matrix(mns))
```

we'd find that the bars were representing the variables, not the regions. To fix this, we simply pass the transpose of the matrix to `barplot`:

```
> barplot(t(as.matrix(mns)))
```

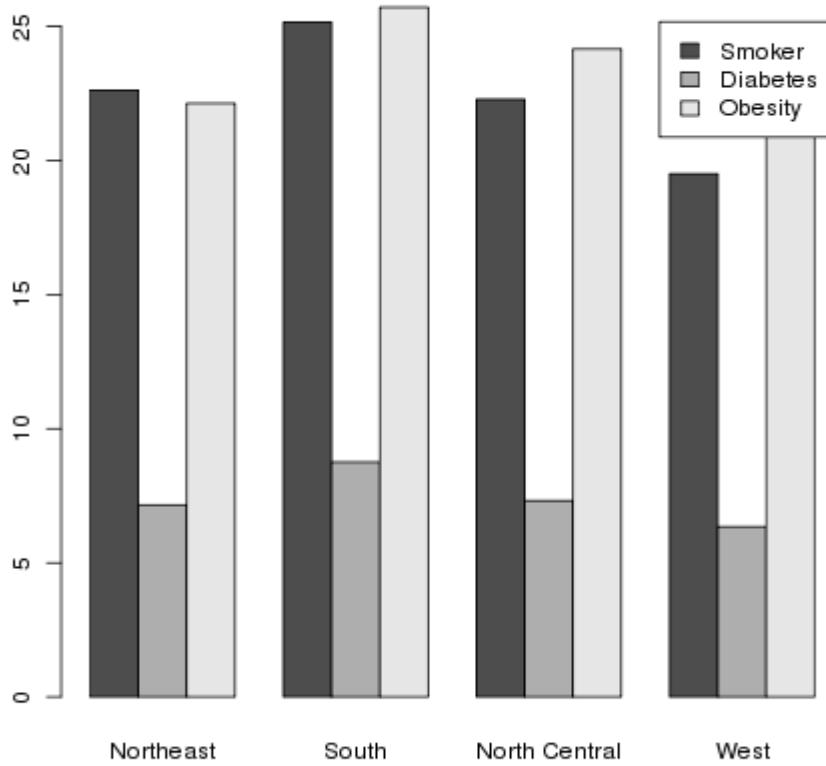
Here's what the plot looks like:



This is known as a stacked barplot, and while it's useful for some kinds of data, I don't find it appropriate for this type of data. By passing the `beside=TRUE` argument to `barplot`, we can produce a side-by-side barchart, which I find more useful. The `legend=TRUE` argument helps to identify the bars:

```
> barplot(t(as.matrix(mns)), beside=TRUE, legend=TRUE)
```

Here's the result:



## 5.4 Mapping

We've looked at some data about different countries around the world, but so far we haven't taken advantage of the fact that the data has a geographic origin (other than trying to see if relationships were different among different continents by using color to represent continent in scatterplots.) The R library `maps` gives us the ability to use different colors to represent values of variables on actual maps of the world or the United States. A current concern in the United States has to do with the number of people without health insurance. Since this continues to be of great concern, it would be interesting to see if there are geographic patterns to the rate of uninsured adults in the different states.

The data for this example comes from the [swivel.com](http://swivel.com) website, and can be downloaded from the class website from <http://www.stat.berkeley.edu/classes/s133/data/insurance.csv>.

```
> library(maps)
> ins = read.csv('http://www.stat.berkeley.edu/classes/s133/data/insurance.csv',
+ stringsAsFactors=FALSE)
```

```
> head(ins)
   State Employer Individual Medicaid Medicare Other.Public Uninsured
1  Alabama      0.54       0.03     0.15     0.13      0.01     0.14
2  Alaska       0.52       0.04     0.16     0.06      0.05     0.17
3 Arizona       0.47       0.05     0.16     0.13      0.01     0.18
4 Arkansas      0.47       0.06     0.15     0.14      0.02     0.17
5 California    0.48       0.07     0.16     0.09      0.01     0.18
6 Colorado      0.59       0.07     0.07     0.08      0.02     0.16
```

The first step in preparing a map (after loading the `maps` library) is determining a variable to use to provide color to the different regions. In this case we'll use the `Uninsured` variable. Our goal will be to break the data up into four groups, based on the value of the `Uninsured` variable, and then to color in each state in the map using a color based on the groups.

To create the four groups based on values of `Uninsured`, we can use the `cut` function. If we pass `cut` a variable and a number of groups, it will divide the range of the variable by that number, and assign each variable to one of the groups. Depending on how the variable is distributed, there may be more observations in one group than the other. Alternatively, we can provide `cut` with a `breaks=` argument, giving one more value than the number of groups we want, and it will assign the values to groups based on the ranges that the breaks define. Thus, if different values of the variable being used to determine the ranges have special meanings, the grouping can be customized using this argument. A final alternative is to guarantee nearly equal-sized groups by using quantiles of the variable in question as the breaks. The `include.lowest=TRUE` argument should be included to make sure that the smallest value gets properly classified. To create four equal-sized groups for the `Uninsured` variable, we could use the following call to `cut`:

```
> ugroups = cut(ins$Uninsured,quantile(ins$Uninsured,(0:4)/4),include.lowest=TRUE)
> head(ugroups)
[1] (0.13,0.17] (0.13,0.17] (0.17,0.24] (0.13,0.17] (0.17,0.24] (0.13,0.17]
Levels: [0.08,0.11] (0.11,0.13] (0.13,0.17] (0.17,0.24]
```

Notice that `cut` produces a factor by default; to suppress this, use the `labels=FALSE` argument. The factor, since it contains information about the actual values, will turn out to be quite useful.

The `maps` library provides three databases: "`world`", "`state`", and "`county`". For each identifier in the database, there is information on what polygons need to be drawn to create an outline of the area in question. For example, the entry identified by `california` in the `state` database would contain the information about California's borders in the form of polygon coordinates which the `polygon` function will draw when we ask for a map of California. While the help page for the `map` function implies that we can pass a vector of region names and colors directly to `map`, things are complicated by the fact that some states can't be plotted by a single polygon, and `map` gets confused about the proper colors to use when it needs to draw more than one polygon for a state.

One way around the problem is to create multiple entries in a vector of colors for those states that have more than one polygon. To do this, we need to look at the region names stored inside the database. We can get a vector of region names by calling `map` with the `names=TRUE` and `plot=FALSE` arguments:

```
> map.names = map('state', names=TRUE, plot=FALSE)
```

The regions which represent multiple polygons for a single state will always contain semi-colons:

```
> grep(':', map.names, value=TRUE)
[1] "massachusetts:martha's vineyard" "massachusetts:main"
[3] "massachusetts:nantucket"          "michigan:north"
[5] "michigan:south"                  "new york:manhattan"
[7] "new york:main"                   "new york:staten island"
[9] "new york:long island"            "north carolina:knotts"
[11] "north carolina:main"            "north carolina:spit"
[13] "virginia:chesapeake"           "virginia:chincoteague"
[15] "virginia:main"                 "washington:san juan island"
[17] "washington:lopez island"        "washington:orcadas island"
[19] "washington:whidbey island"      "washington:main"
```

To properly connect our data with these region names, we first create a vector of state names corresponding to the regions in the data base:

```
> map.states = sub('^([:]*):.*$', '\\1', map.names)
```

Now we can use the `match` function to see which observations in the `ins` dataframe correspond to the regions in the database. Since the database uses all lower case, I'll use the `tolower` function to convert the state names in the `ins` data frame to lower case:

```
> which.state = match(map.states, tolower(ins$State))
```

Now that we know how the state names line up with the region names, we can create a vector of colors to properly create our map:

```
> mycolors = c('gray', 'white', 'pink', 'red')
> thecolors = mycolors[ugroups[which.state]]
```

This process is complicated enough that it might be worth making a function to give us the proper vector of groupings. Here's such a function, created by putting together the steps we just followed:

```
mapgroups = function(db, myregions, mygroups, tolower=TRUE){
  map.names = map(db, names=TRUE, plot=FALSE)
  map.regions = gsub('^([:]*):.*$', '\\1', map.names)
  if(tolower)myregions = tolower(myregions)
  which.region = match(map.regions, myregions)
  mygroups[which.region]
}
```

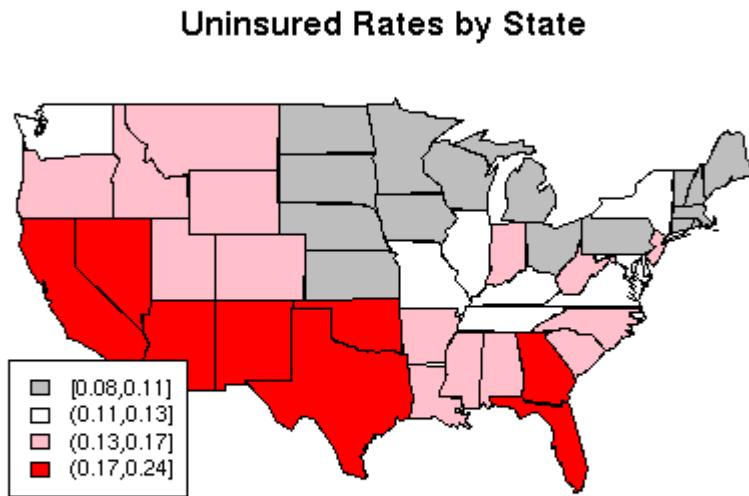
(I've included the `tolower=` argument because not all the map databases use lower case.) Using this function (or the step-by-step approach that led to it), we can now make our plot:

```
> mycolors = c('gray','white','pink','red')
> thecolors = mycolors[mapgroups('state',ins$State,ugroups)]
> map('state',col=thecolors,fill=TRUE)
```

Plots like this should always have a legend:

```
> title('Uninsured Rates by State')
> legend('bottomleft',legend=levels(ugroups),fill=mycolors,cex=.8)
```

The map is pictured below.



There are a variety of functions in R which will help us choose colors that will be useful for plotting. Some of the functions you might want to investigate include `rainbow`, `heat.colors`, `topo.colors`, as well as the `color.gradient` function in the `plotrix` library, the `colorpanel` function in the `gplots` library, and the functions in the `RColorBrewer` library. Let's look at another example, also from [swivel.com](http://swivel.com), concerning rates of childhood obesity. A copy of the dataset is available on the class website.

```
> obesity = read.csv('http://www.stat.berkeley.edu/classes/s133/data/obesity.csv')
```

Before preceding, it turns out that the state names in this data set have some embedded blanks:

```
> head(as.character(obesity$State))
[1] " Alabama "      " Alaska "       " Arizona "      " Arkansas "     " California "
[6] " Colorado "
```

Left uncorrected, this would make our `mapgroups` function fail to find any matching states in the map data base, and when we plot the map, there would be no colors. If you think you're doing everything right, but there are no colors on the map, check to make sure that the state names are correct.

Naturally fixing something like this is not a problem:

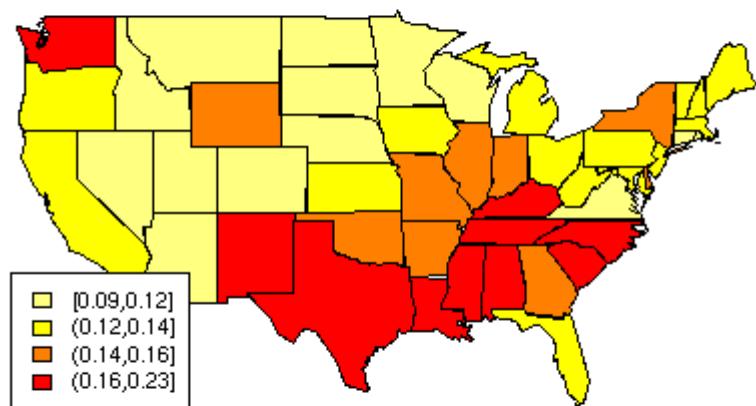
```
> obesity$State = gsub('^ +', '', obesity$State)
> obesity$State = gsub(' +$', '', obesity$State)
```

Now we can proceed with producing the map. We'll use the `Childhood.Obesity.Rate` variable for the example, and a color scheme generated by the `heat.colors` function:

```
> mycolors = rev(heat.colors(4))
> ogroups = cut(obesity$Childhood.Obesity.Rate,
+                 quantile(obesity$Childhood.Obesity.Rate,(0:4)/4),include.lowest=TRUE)
> thecolors = mycolors[mapgroups('state',obesity$State,ogroups)]
> map('state',col=thecolors,fill=TRUE)
> title('Childhood Obesity Rates by State')
> legend('bottomleft',legend=levels(ogroups),fill=mycolors,cex=.8)
```

The map appears below:

### Childhood Obesity Rates by State



As an example of using the county maps, we can use another one of the Community Health data sets, this one concerned with demographics. It's located at <http://www.stat.berkeley.edu/classes/s133/data/DEMOGRAPHICS.csv>. The first step is to read the data into R. We'll concentrate on the variable `Hispanic`, which gives the percentage of Hispanics in each county.

```
> demo = read.csv('http://www.stat.berkeley.edu/classes/s133/data/DEMOGRAPHICS.csv')
> summary(demo$Hispanic)
   Min. 1st Qu. Median    Mean 3rd Qu.    Max.
0.000  1.100  2.300  7.018  6.300  97.500
```

Next we need to see how the county names are stored in the county map database:

```
> nms = map('county',names=TRUE,plot=FALSE)
> head(nms)
[1] "alabama,autauga" "alabama,baldwin" "alabama,barbour" "alabama,bibb"
[5] "alabama,blount"   "alabama,bullock"
```

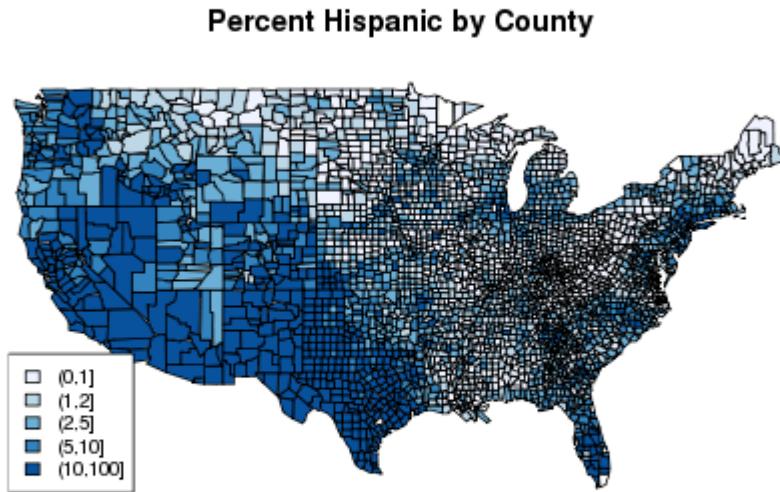
We need to combine the state and county information in our data set so that it matches up with the format used in the database. We could either call `tolower` now, or leave it to our `mapgroups` function:

```
> head(subset(demo,select=c(CHSI_State_Name,CHSI_County_Name)))
  CHSI_State_Name CHSI_County_Name
1      Alabama        Autauga
2      Alabama        Baldwin
3      Alabama       Barbour
4      Alabama         Bibb
5      Alabama        Blount
6      Alabama       Bullock
> thecounties = paste(demo$CHSI_State_Name,demo$CHSI_County_Name,sep=',')
```

Based on the summary information above, we can cut the `Hispanic` variable at 1,2,5, and 10. Then we can create a palette of blues using the `brewer.pal` function from the `RColorBrewer` package.

```
> hgroups = cut(demo$Hispanic,c(0,1,2,5,10,100))
> table(hgroups)
hgroups
  (0,1]   (1,2]   (2,5]   (5,10]  (10,100]
    678     770     778     356      558
> library(RColorBrewer)
> mycolors = brewer.pal(5,'Blues')
> thecolors = mycolors[mapgroups('county',thecounties,hgroups)]
> map('county',col=thecolors,fill=TRUE)
> legend('bottomleft',levels(hgroups),fill=mycolors,cex=.8)
> title('Percent Hispanic by County')
```

The `cex=.8` argument reduces the text size to 80% of the default size, to prevent the legend from running into the map. The map looks like this:



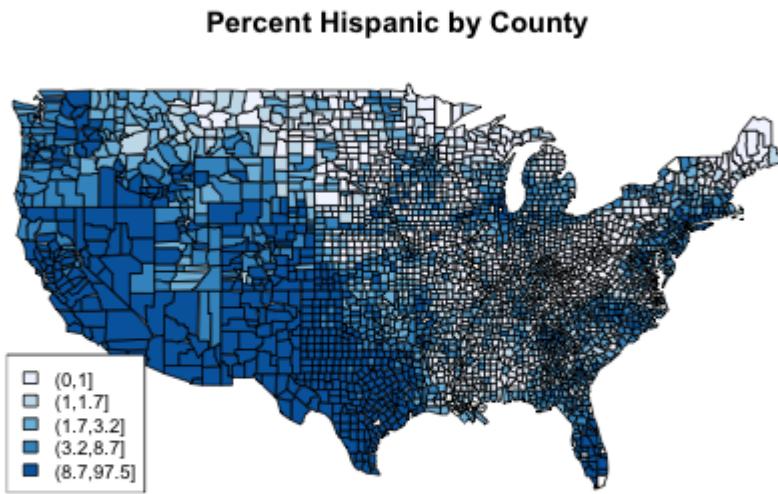
In previous examples, we created equal sized groups using the `quantile` function. Let's see if it will make a difference for this example:

```
> hgroups1 = cut(demo$Hispanic,quantile(demo$Hispanic,(0:5)/5))
> table(hgroups1)
hgroups1
(0,1]   (1,1.7]   (1.7,3.2]   (3.2,8.7]   (8.7,97.5]
678       597       627       618       620
```

The rest of the steps are the same as before:

```
> mycolors = brewer.pal(5,'Blues')
> thecolors = mycolors[mapgroups('county',thecounties,hgroups1)]
> map('county',col=thecolors,fill=TRUE)
> legend('bottomleft',levels(hgroups1),fill=mycolors,cex=.8)
> title('Percent Hispanic by County')
```

The plot, which is very similar to the previous plot, appears below:



As a final example of working with maps, let's revisit the world data set that we've used in other examples. Suppose we want to create a map showing literacy rates around the world. First we need to decide on a grouping. The `summary` function is useful in helping us decide:

```
> world = read.csv('http://www.stat.berkeley.edu/classes/s133/data/world2.txt',
+ na.strings='.',comment='#')
> summary(world$literacy)
   Min. 1st Qu. Median    Mean 3rd Qu.    Max.
 12.80   69.10  88.40  80.95  98.50  99.90
```

Let's create four levels: less than 50 use `cut` to create these levels and label them at the same time:

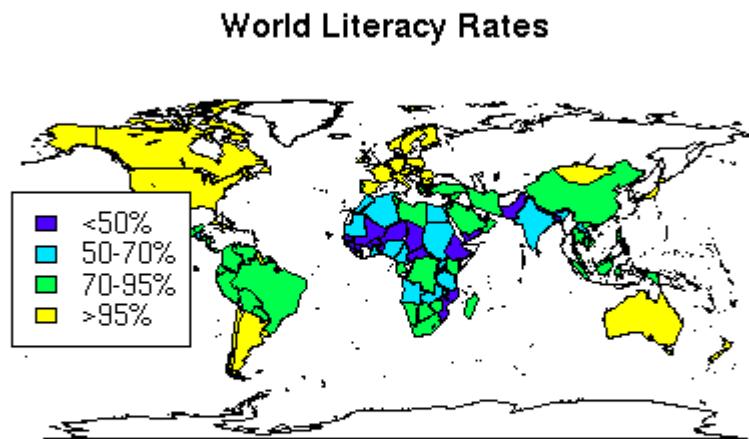
```
litgroups = cut(world$literacy,breaks=c(0,50,70,95,100),include.lowest=TRUE,
               labels=c('<50%','50-70%','70-95%','>95%'))
```

We can use the `mapgroups` function to come up with the correct colors, noticing that the region names in the `world` database are not in lower case:

```

> mycolors = topo.colors(4)
> litcolors = mycolors[mapgroups('world',world$country,litgroups,tolower=FALSE)]
> map('world',col=litcolors,fill=TRUE)
> title('World Literacy Rates')
> legend('left',legend=levels(litgroups),fill=mycolors,cex=.9)

```



The map appears below

An alternative which is useful when you only want to use part of a database is to eliminate missing values from the vector of colors and the corresponding regions from the full list of regions, and pass those vectors directly to `map`. Applying this idea to the previous example, we could have gotten the same plot with statements like this:

```

> omit = is.na(litcolors)
> useregions = map('world',names=TRUE,plot=FALSE)[!omit]
> map('world',regions=useregions,col=litcolors[!omit],fill=TRUE)

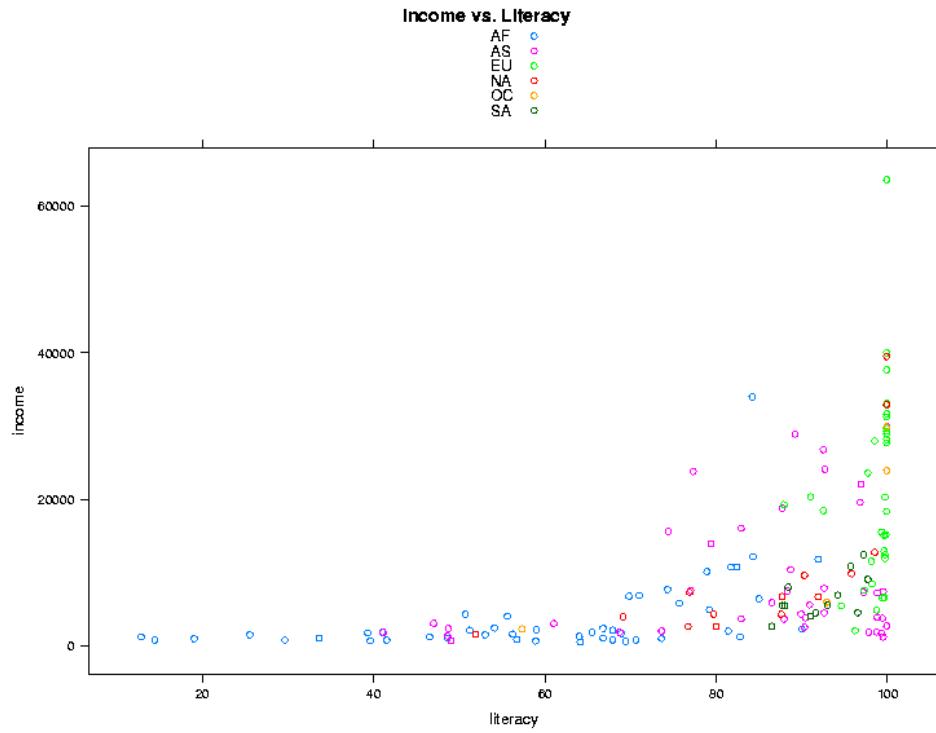
```

## 5.5 The Lattice Plotting Library

The graphics we've been using up until now are sometimes known as "traditional" graphics in the R community, because they are the basic graphics components that have been part of the S language since its inception. To review, this refers to the high-level functions like `plot`, `hist`, `barplot`, `plot`, `pairs`, and `boxplot`, the low-level functions like `points`, `lines`, and `legend`, and the graphics parameter system accessed through the `par` function. There's another entire set of graphics tools available through the `lattice` library. Originally developed as the `trellis` library, the implementation in R is known as `lattice`, although the name "trellis" persists in some functions and documentation. All of the functions in the `lattice` library use the formula interface that we've seen in classification and modeling functions instead of the usual list of `x` and `y` values. Along with other useful features unique to each function, all of the `lattice` functions accept a `data=` argument, making it convenient to work with dataframes; by specifying the data frame name via this argument, you can refer to the variables in the data frame without the data frame name. They also accept a `subset=` argument, similar to the second argument to the `subset` function, to allow selection of only certain cases when you are creating a plot. Finally, the `lattice` plotting functions can produce what are known as conditioning plots. In a conditioning plot, several graphs, all with common scaling, are presented in a single display. Each of the individual plots is constructed using observations that have a particular value of a variable (known as the conditioning variable), allowing complex relationships to be viewed more easily. To illustrate the idea, let's revisit the income versus literacy graph that we looked at when we first started studying graphics. The `lattice` equivalent of the traditional `plot` command is `xyplot`. This function accepts a plotting formula, and has some nice convenience functions not available in the regular `plot` command. For example, the `groups=` argument allows specifying a grouping variable; observations with different levels of this variable will be plotted with different colors. The argument `auto.key=TRUE` automatically shows which colors represent which groups. So we could create our graph with the single statement:

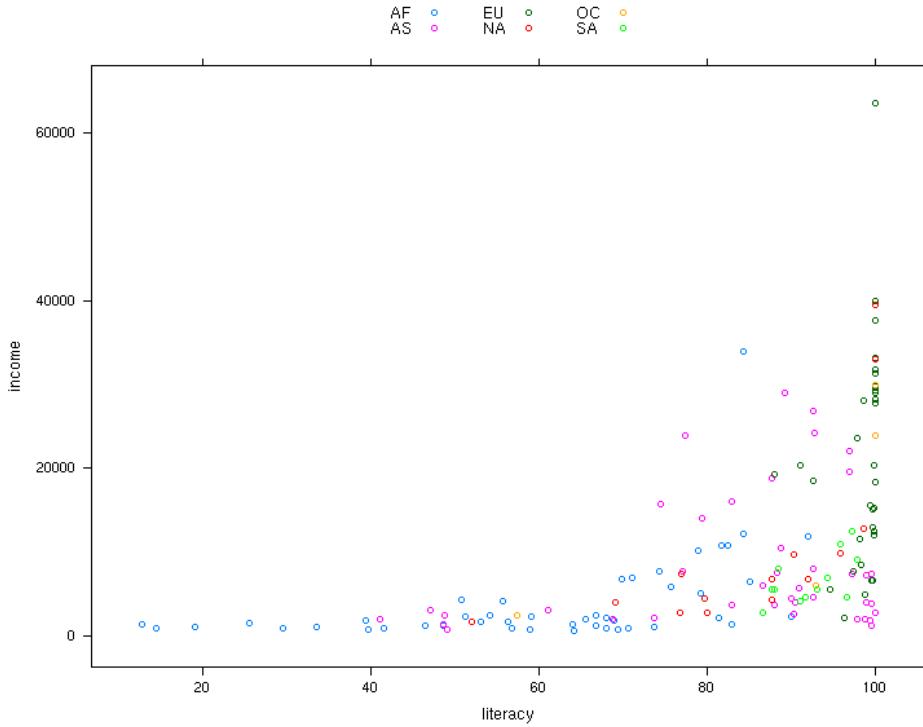
```
> library(lattice)
> world = read.csv('http://www.stat.berkeley.edu/classes/s133/data/world2.txt',
+   comment='#',na.strings='.')
> xyplot(income ~ literacy,data=world,groups=cont,auto.key=TRUE,
+   main='Income vs. Literacy'))
```

The plot is shown below.



One simple change we could make is to display the legend (created by `auto.key=TRUE`) in 3 columns. This can be achieved by changing the value of `auto.key` to `auto.key=list(columns=3)`. Many of the parameters to lattice functions can be changed by passing a list of named parameters. Here's the updated call to `lattice`, and the result:

```
> xyplot(income ~ literacy, data=world, groups=cont, auto.key=list(columns=3))
```



If you wish to finetune the appearance of lattice plots, you can modify most aspects of lattice plots through the command `trellis.par.set`, and you can display the current values of options with the command `trellis.par.get`. To explore possible modifications of the trellis (lattice) environment, take a look at the output from

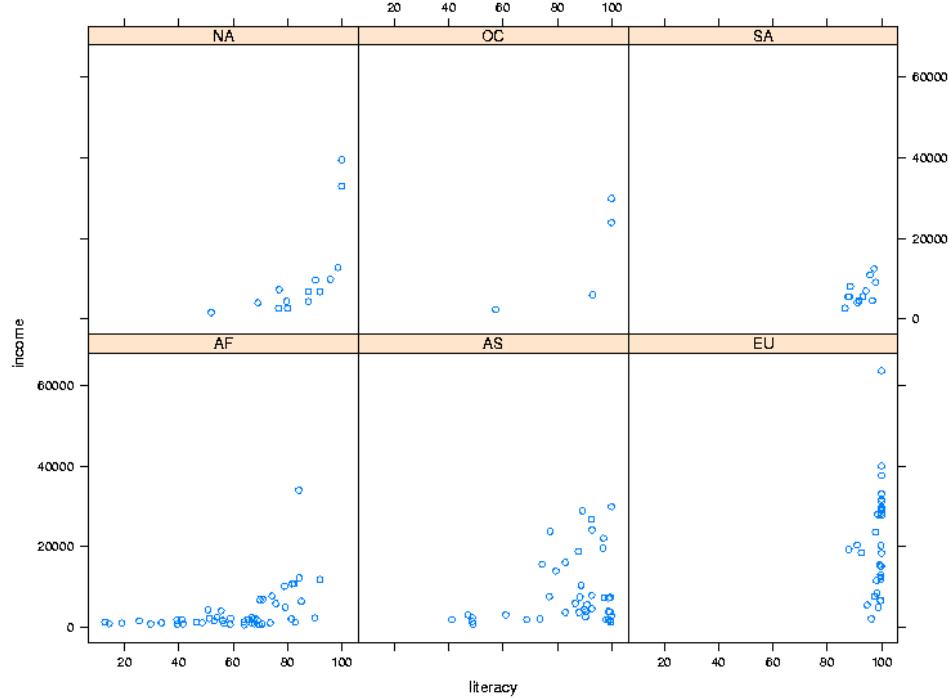
```
> names(trellis.par.get())
```

Any of the listed parameters can be changed through the `trellis.par.set()` command.

To illustrate the idea of a conditioning plot, let's create a scatter plot like the previous one, but, instead of using color to distinguish among the continents, we'll use the continent as a conditioning variable, resulting in a separate scatter plot for each continent. To use a conditioning variable in any of the `lattice` commands, follow the formula with a vertical bar (`|`) and the name of the conditioning variable. To get `xypplot` to display the value of the conditioning variable, it helps if it's a factor:

```
> world$cont = factor(world$cont)
> xyplot(income ~ literacy | cont, data=world)
```

The plot is shown below:

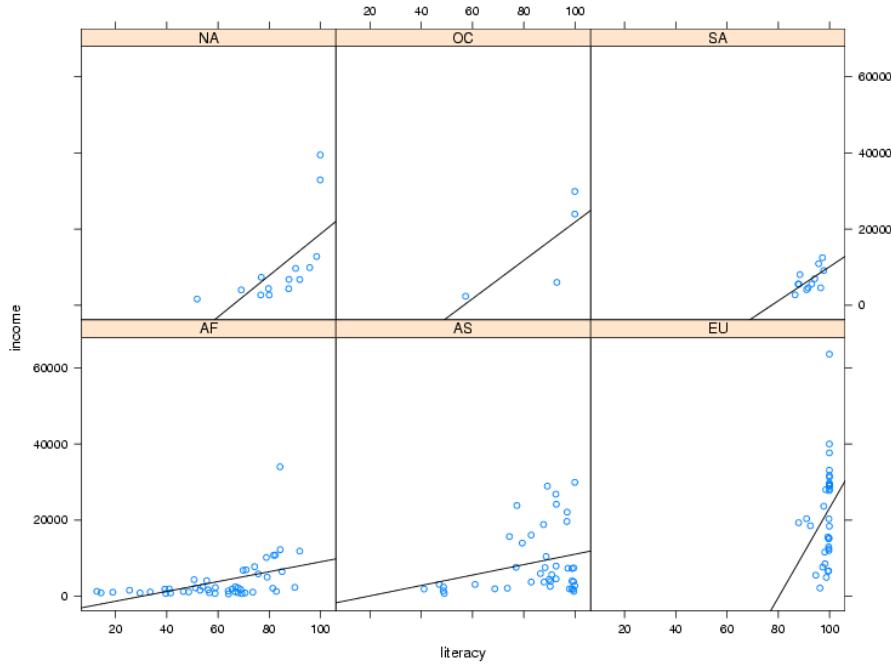


## 5.6 Customizing the Panel Function

One of the basic concepts of lattice plots is the idea of a panel. Each separate graph that is displayed in a multi-plot lattice graph is known as a panel, and for each of the basic types of lattice plots, there's a function called `panel.plottype`, where `plottype` is the type of plot in question. For example, the function that actually produces the individual plots for `xyplot` is called `panel.xyplot`. To do something special inside the panels, you can pass your own panel function to the lattice plotting routines using the `panel=` argument. Generally, the first thing such a function would do is to call the default panel plotting routine; then additional operations can be performed with functions like `panel.points`, `panel.lines`, `panel.text`. (See the help page for `panel.functions` to see some other possibilities.) For example, in the income versus literacy plot, we might want to show the best regression line that goes through the points for each continent, using the `panel.lmline` function. Here's how we could construct and call a custom panel function:

```
> mypanel = function(x,y,...){
+   panel.xyplot(x,y,...)
+   panel.lmline(x,y)
+ }
xyplot(income ~ literacy | cont,data=world,panel=mypanel)
```

The plot is shown below.



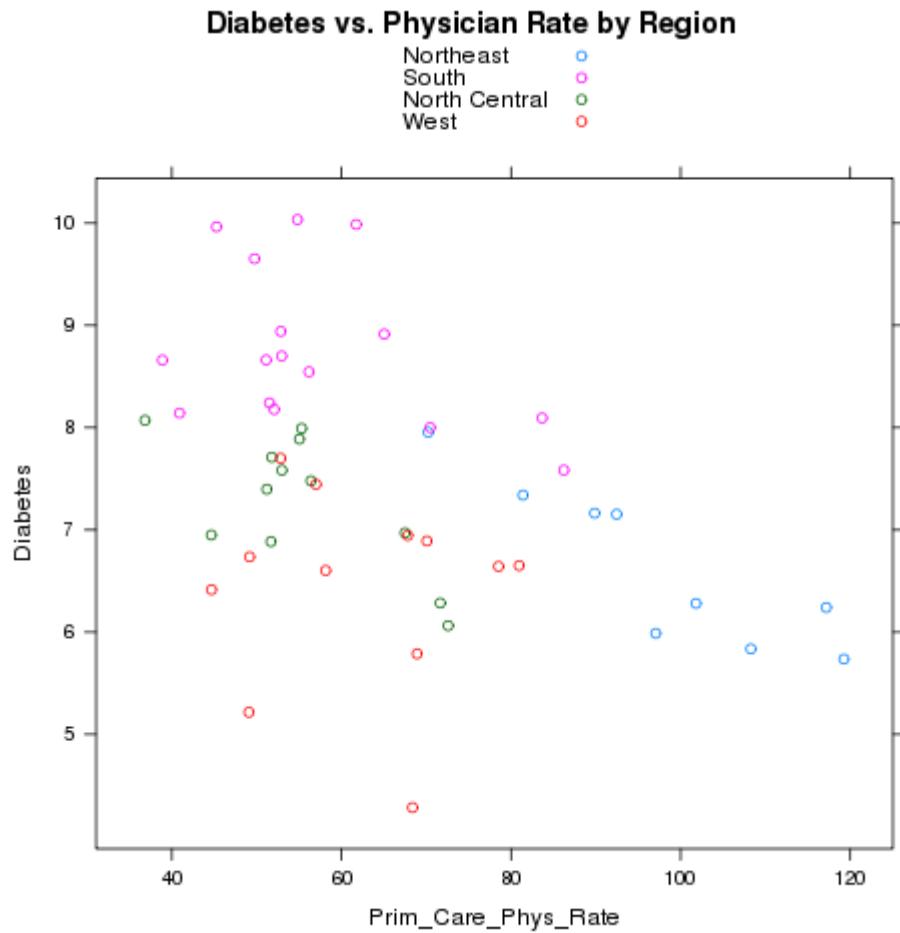
As another example, consider again the Community Health Data risk and access to care data set. We want to see if there is a relationship between the number of physicians in a state, and the rate of Diabetes in that state. We'll read it in as before, aggregate it, and merge it with the state regions.

```
> risk = read.csv('http://www.stat.berkeley.edu/classes/s133/data/RISKFACTORSANDACCESSTOCARE.csv')
> risk[risk== -1111.1] = NA
> avgs = aggregate(risk[,c('Diabetes','Prim_Care_Phys_Rate')],risk['CHSI_State_Name'],
+ ,mean,na.rm=TRUE)
> avgs = merge(avgs,data.frame(state.name,region=state.region),by.x='CHSI_State_Name',by.y=1)
```

Notice that I used the variable number instead of the name in the call to `merge`. Let's first use color to represent the different regions.

```
> xyplot(Diabetes~Prim_Care_Phys_Rate,groups=region,data=avgs,auto.key=TRUE,
+ main='Diabetes vs. Physician Rate by Region')
```

Here's the plot:

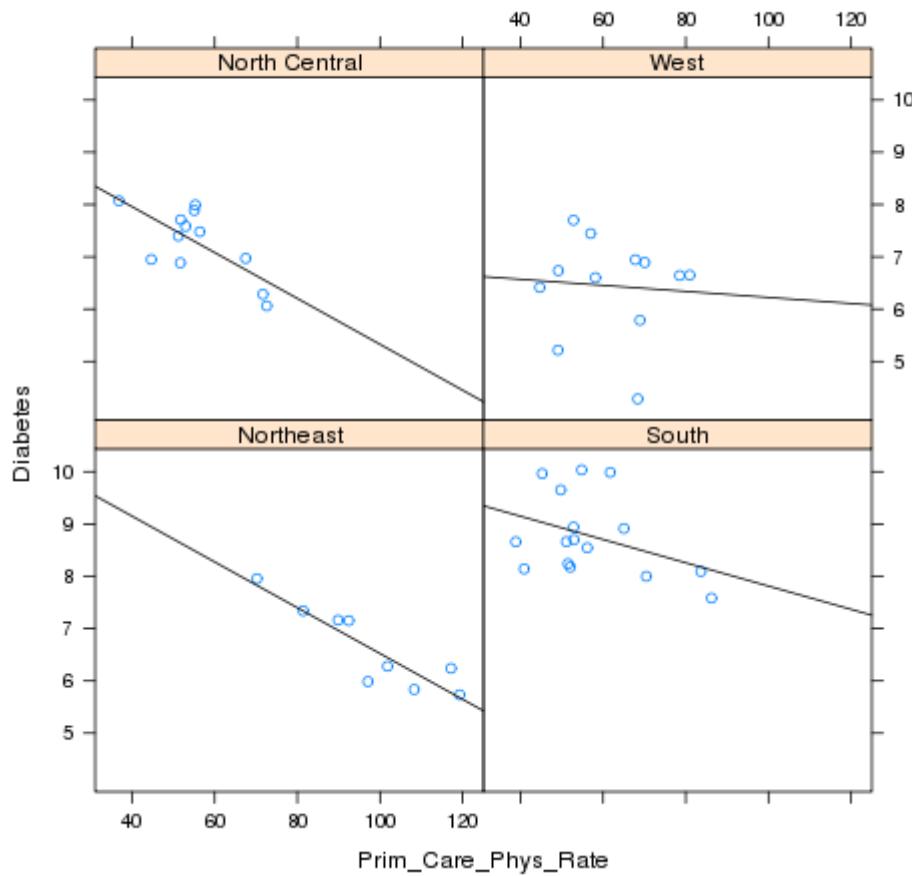


Alternatively, we could place each region in a separate panel, and display the best fit regression line:

```
> mypanel = function(x,y,...){
+   panel.xyplot(x,y,...)
+   panel.lmline(x,y)
+ }
> xyplot(Diabetes~Prim_Care_Phys_Rate|region,data=avgs,
+         main='Diabetes vs. Physician Rate by Region',panel=mypanel)
```

The plot appears below:

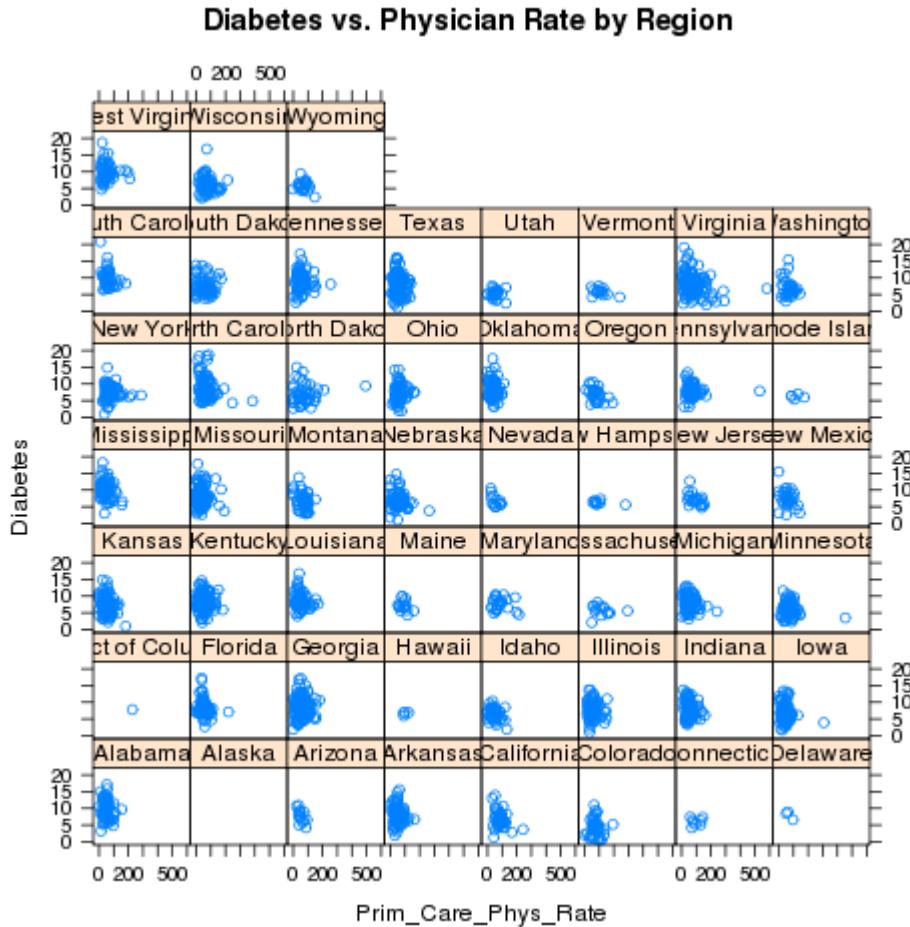
### Diabetes vs. Physician Rate by Region



By default, the lattice functions display their panels from bottom to top and left to right, similar to the way points are drawn on a scatterplot. If you'd like the plots to be displaying going from top to bottom, use the `as.table=TRUE` argument to any of the lattice plotting functions.

The lattice library is actually flexible enough to produce separate plots for each state:

```
> xyplot(Diabetes~Prim_Care_Phys_Rate|CHSI_State_Name,data=risk,
+         main='Diabetes vs. Physician Rate by Region')
```

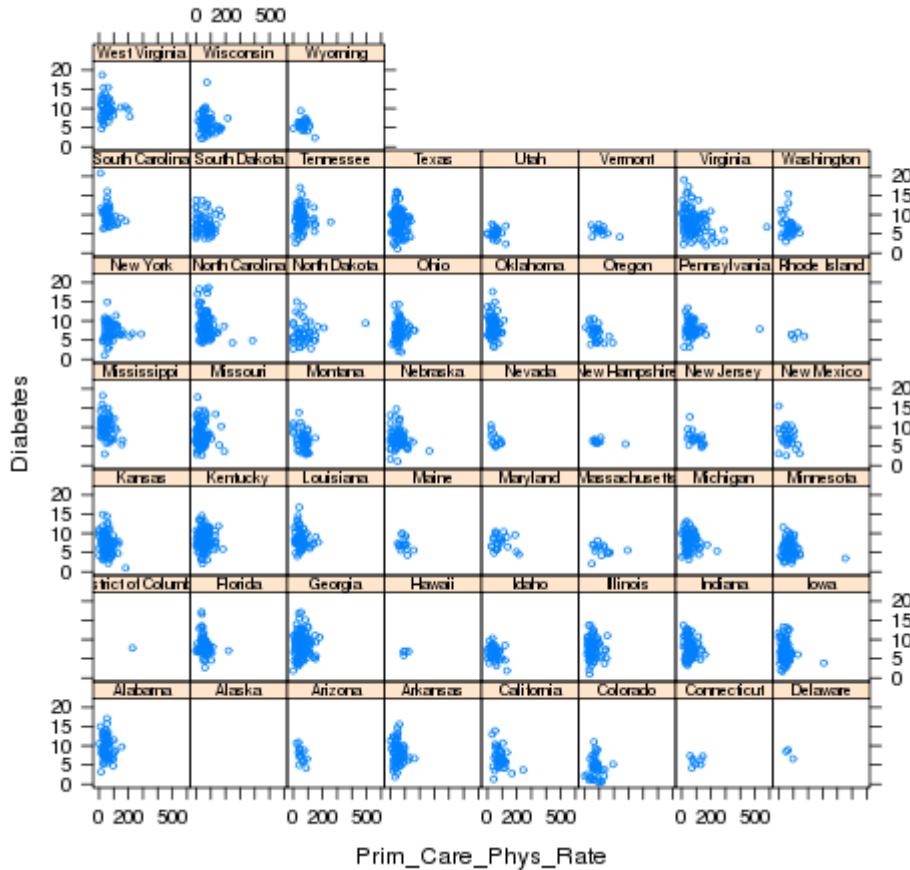


Looking at the plot, we should make some elements of the plot smaller, namely the titles on the strips and the points themselves:

```
> xyplot(Diabetes~Prim_Care_Phys_Rate|CHSI_State_Name,data=risk,
+         main='Diabetes vs. Physician Rate by Region',cex=.5,
+         par.strip.text=list(cex=.6))
```

This results in the following plot:

## Diabetes vs. Physician Rate by Region



Finally, let's suppose that we want to break up the plots into two pages, each with 25 plots in a 5x5 arrangement. To get exactly 50 states, we'll use the `subset=` argument of the lattice functions to remove Alaska (for which there's no data), and the `layout=` argument to arrange the plots the way we want, and the `page=` argument to call a function at the end of each page:

```
> xyplot(Diabetes~Prim_Care_Phys_Rate|CHSI_State_Name,data=risk,
+         main='Diabetes vs. Physician Rate by Region',
+         subset=CHSI_State_Name != 'Alaska',
+         layout=c(5,5,2),page=readline)
```

You can run this example to see the two pages.

Now that we've seen some of the basics of how the lattice library routines work, we'll take a look at some of the functions that are available. Remember that there are usually similar alternatives available among the traditional graphics functions, so you can think of these as additional choices that are available, and not necessarily the only possibility for producing a particular type of plot.

## 5.7 Univariate Displays

Univariate displays are plots that are concerned with the distribution of a single variable, possibly comparing the distribution among several subsamples of the data. They are especially useful when you are first getting acquainted with a data set, since you may be able to identify outliers or other problems that could get masked by more complex displays or analyses.

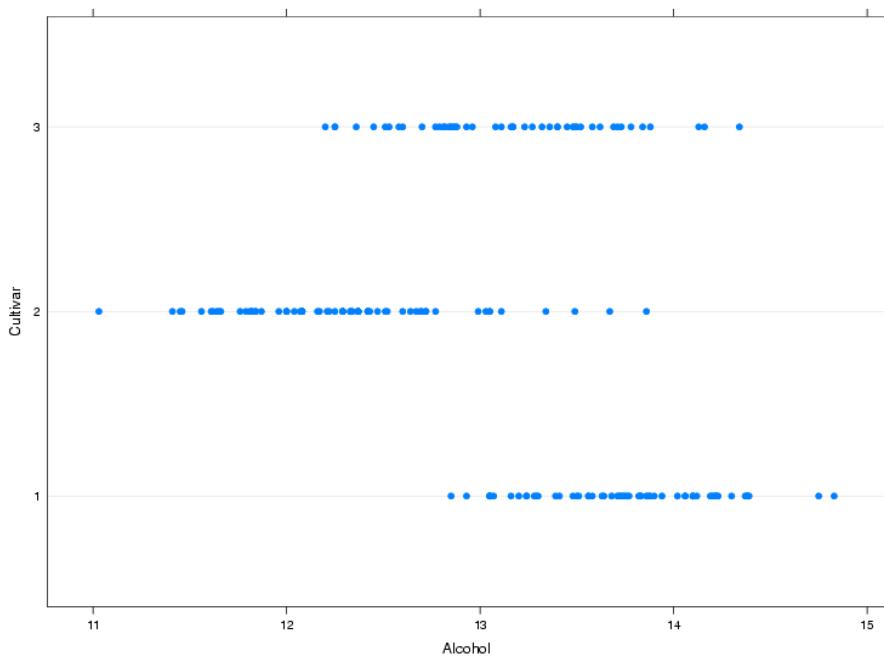
### 5.7.1 dotplot

A simple but surprisingly useful display for small to moderate amounts of univariate data is the dotplot. Each observation's value for a variable is plotted as a dot along a line that spans the range of the variable's value. In the usual case, there will be several such lines, one for each level of a grouping variable, making it very easy to spot differences in the variable's distribution for different groups.

To illustrate, we'll use a data set from a wine recognition experiment where a number of chemical and other measurements were taken on wines from three cultivars. The data is available at <http://www.stat.berkeley.edu/classes/s133/data/wine.data>; information about the variables is at <http://www.stat.berkeley.edu/classes/s133/data/wine.names>. Suppose we are interested in comparing the alcohol level of wines from the three different cultivars:

```
> wine = read.csv('http://www.stat.berkeley.edu/classes/s133/data/wine.data',header=FALSE)
> names(wine) = c("Cultivar", "Alcohol", "Malic.acid", "Ash", "Alkalinity.ash",
+                 "Magnesium", "Phenols", "Flavanoids", "NF.phenols", "Proanthocyanins",
+                 "Color.intensity","Hue","OD.Ratio","Proline")
> wine$Cultivar = factor(wine$Cultivar)
> dotplot(Cultivar~Alcohol,data=wine,ylab='Cultivar')
```

The plot is shown below.

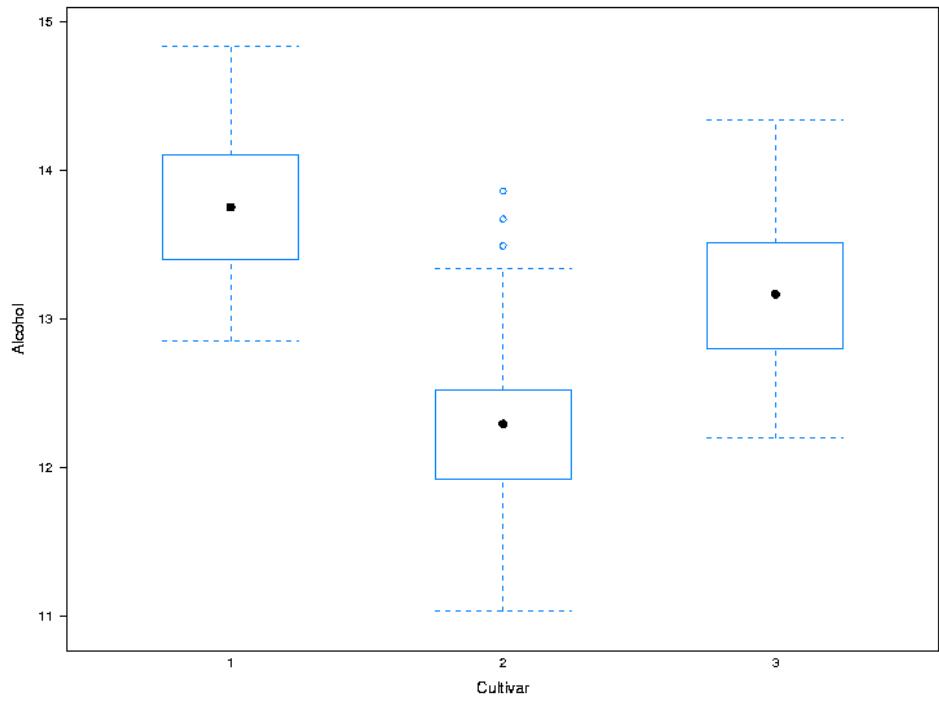


### 5.7.2 bwplot

The `bwplot` produces box/whisker plots. Unfortunately, notched boxplots are not currently available using `bwplot`. To create a box/whisker plot of Alcohol for the three cultivars, we can use the same formula we passed to `dotplot`:

```
> bwplot(Alcohol~Cultivar,data=wine,xlab='Cultivar',ylab='Alcohol')
```

The plot is shown below.

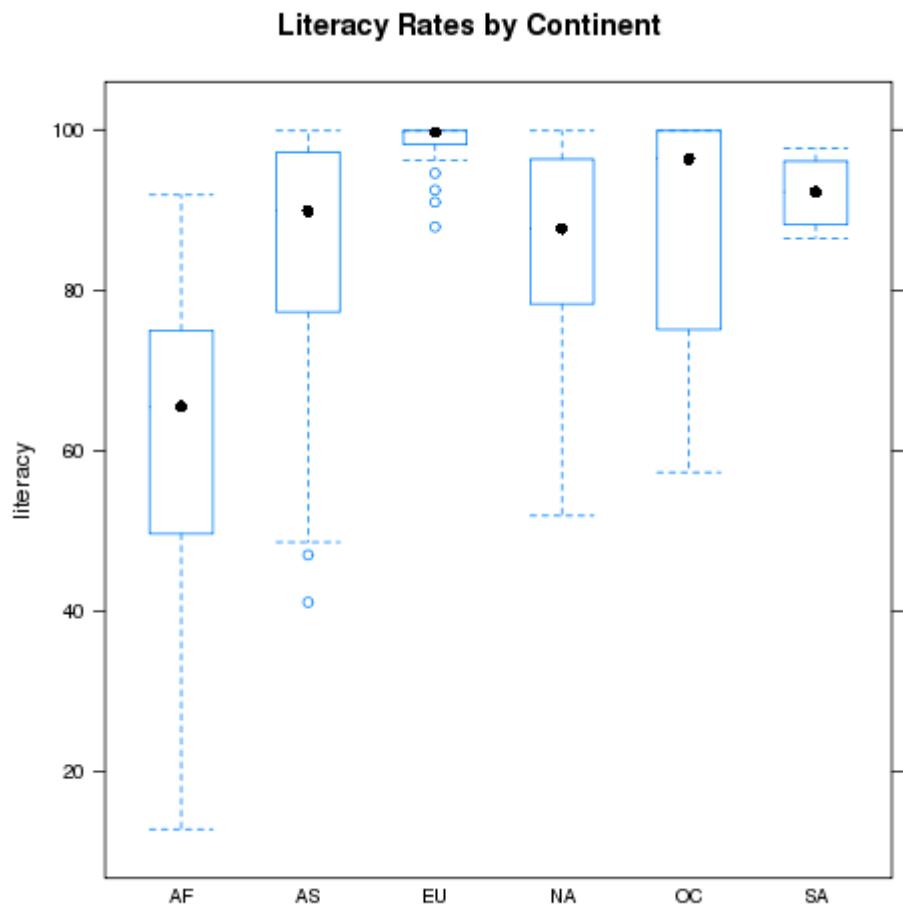


For both `dotplot` and `bwplot`, if you switch the roles of the variables in the formula, the orientation of the plot will change. In other words, the lines in the dotplot will be displayed vertically instead of horizontally, and the boxplots will be displayed horizontally instead of vertically.

As a second example, consider the literacy rates for the different continents in the `world` data set. We can compare these rates using the following command:

```
> bwplot(literacy~cont,data=world,main='Literacy Rates by Continent')
```

The plot is shown below

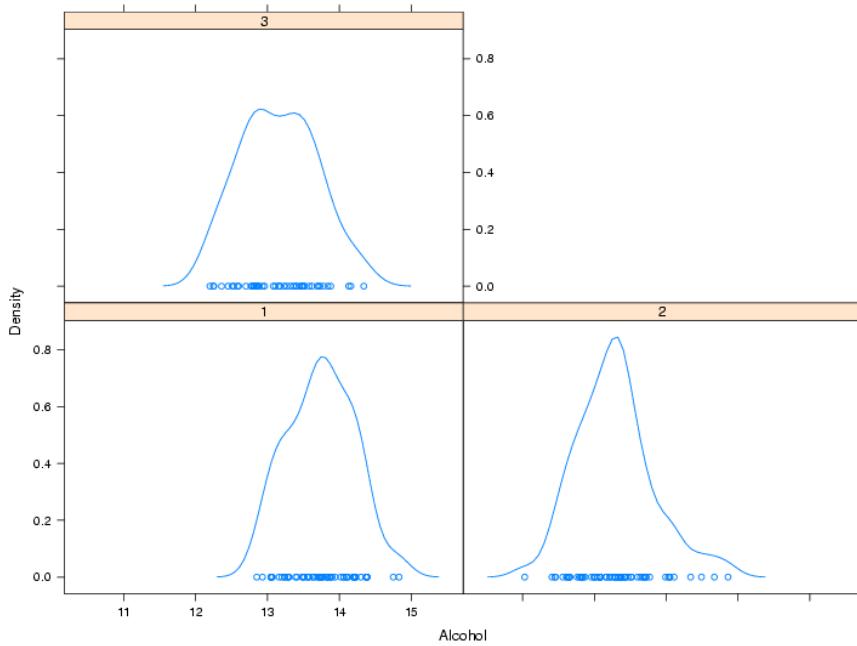


### 5.7.3 densityplot

As its name implies, this function produces smoothed plots of densities, similar to passing a call to the `density` function to the `plot` function. To compare multiple groups, it's best to create a conditioning plot:

```
> densityplot(~Alcohol|Cultivar,data=wine)
```

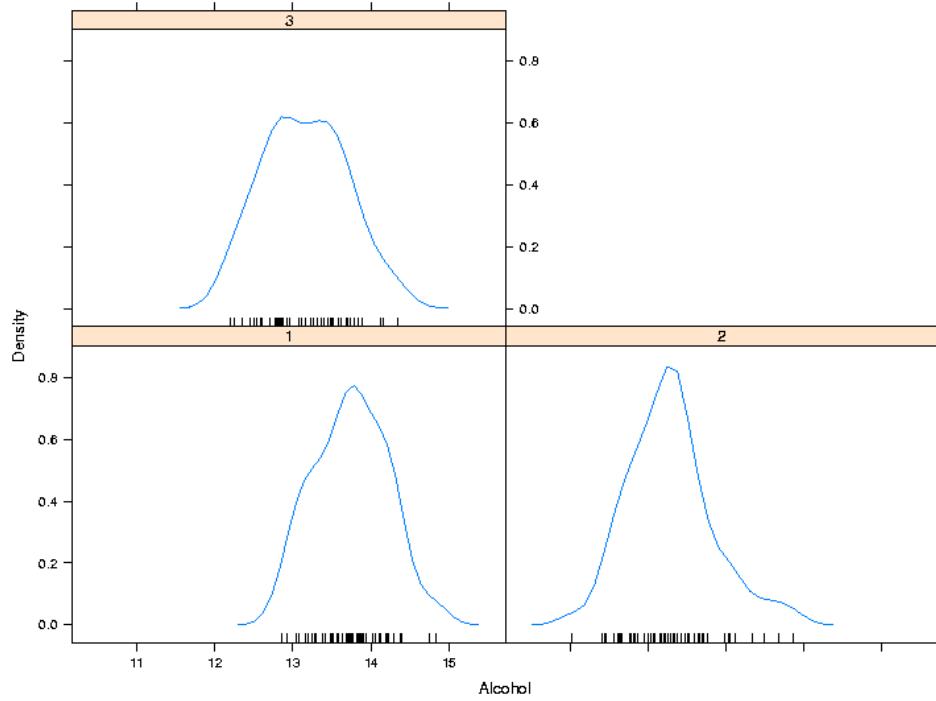
Notice that, for plots like this, the formula doesn't have a left hand side. The plot is shown below:



As another example of a custom panel function, suppose we wanted to eliminate the points that are plotted near the x-axis and replace them with what is known as a rug - a set of tickmarks pointing up from the x-axis that show where the observations were. In practice, many people simply define panel functions like this on the fly. After consulting the help pages for `panel.densityplot` and `panel.rug`, we could replace the points with a rug as follows:

```
> densityplot(~Alcohol|Cultivar,data=wine,panel=function(x,...){
+   panel.densityplot(x,plot.points=FALSE)
+   panel.rug(x=x)
+ })
```

Of course, if you find it easier or more convenient to define a custom panel function separate from the call to the plotting function, you can use that method. Here's the result:

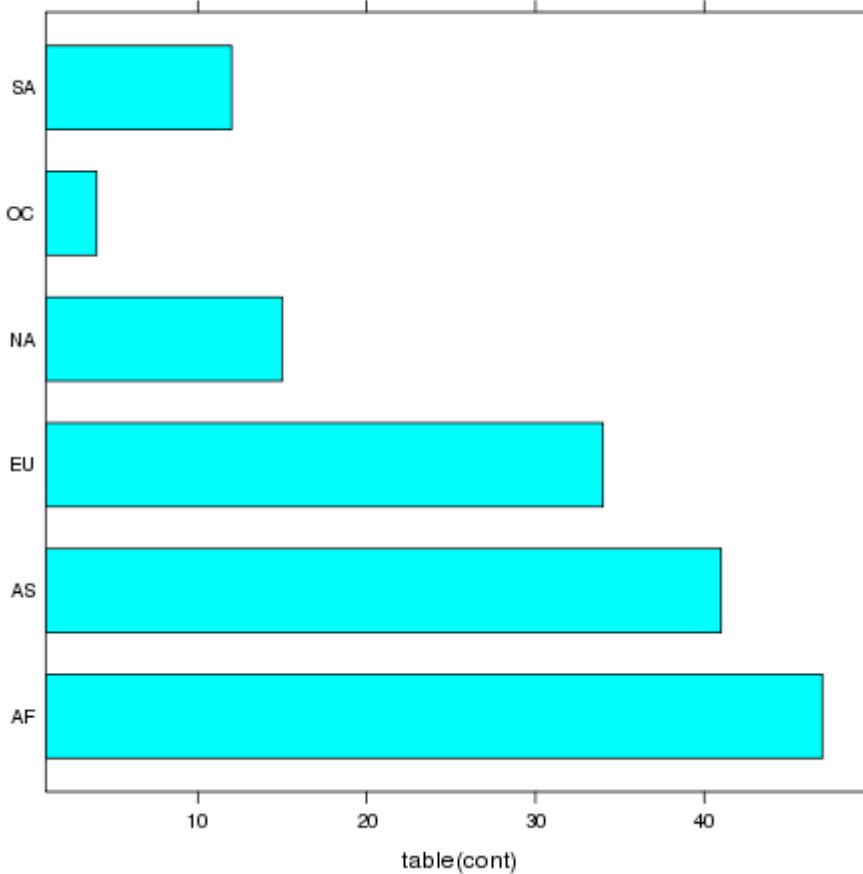


## 5.8 barchart

A bar chart is like a histogram for categorical data. The **barchart** function expects its input data frame to already have the numbers of observations for each grouping tabulated. For the simplest case of a single variable with no conditioning variable, you can use a call to **table** on the right hand side of the tilda to produce a vertical bar chart:

```
> barchart(~table(cont), data=world)
```

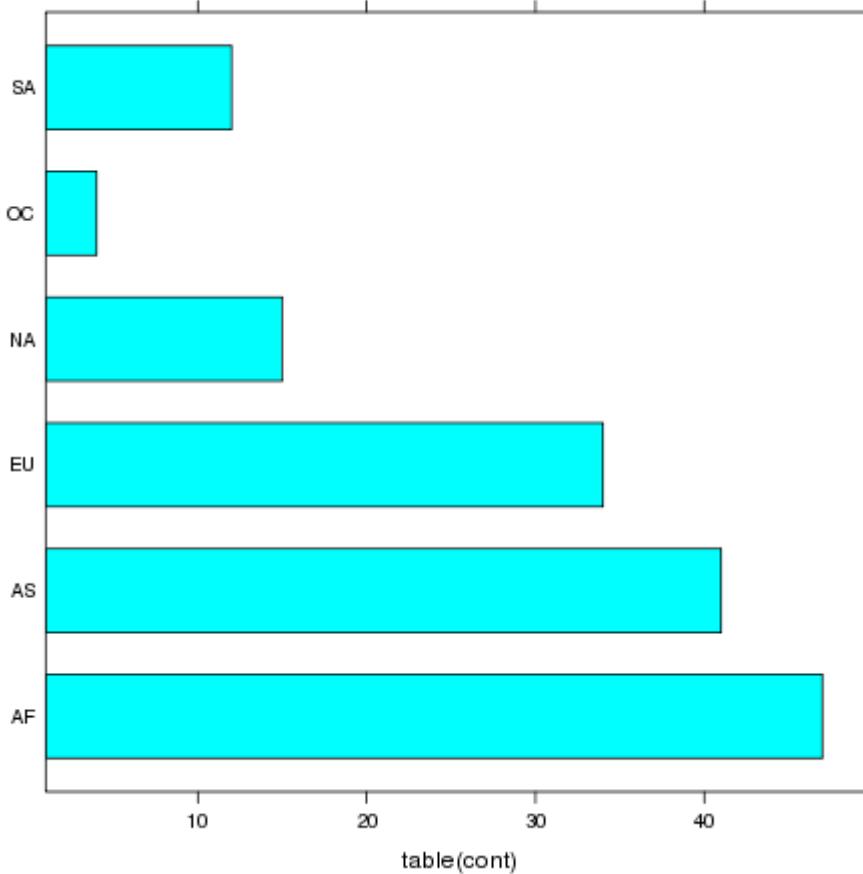
The plot is shown below.



For more complex barcharts, a data frame containing the counts to be plotted needs to be constructed. This can be done easily using the `table` in conjunction with `as.data.frame`. To illustrate, we'll return to the movies data set which has the release dates and box office receipts for some of the all-time most popular movies. Suppose we want to see if the distribution of the day of the week the movies opened on has changed over time. First, we'll read the data and create a grouping variable for different time periods:

```
> movies = read.delim('http://www.stat.berkeley.edu/classes/s133/data/movies.txt',as.is=TRUE,sep=' ')
> movies$box = as.numeric(sub('^\$', '',movies$box))
> movies$date = as.Date(movies$date,'%B %d, %Y')
> movies$year = as.numeric(format(movies$date, '%Y'))
> movies$weekday = factor(weekdays(movies$date),
+                         levels=c('Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'))
> movies$yrgrp = cut(movies$year,c(1936,1980,1985,1990,1995,2000,2006,2011),
+                      labels=c('1937-1979','1980-1984','1985-1990','1991-1995','1996-2000','2001-2006','2007-2011'))
> counts = as.data.frame(table(yrgrp=movies$yrgrp,weekday=movies$weekday))
> barchart(Freq~weekday|yrgrp,data=counts,scales=list(x=list(rot=90)),as.table=TRUE)
```

The plot is shown below.



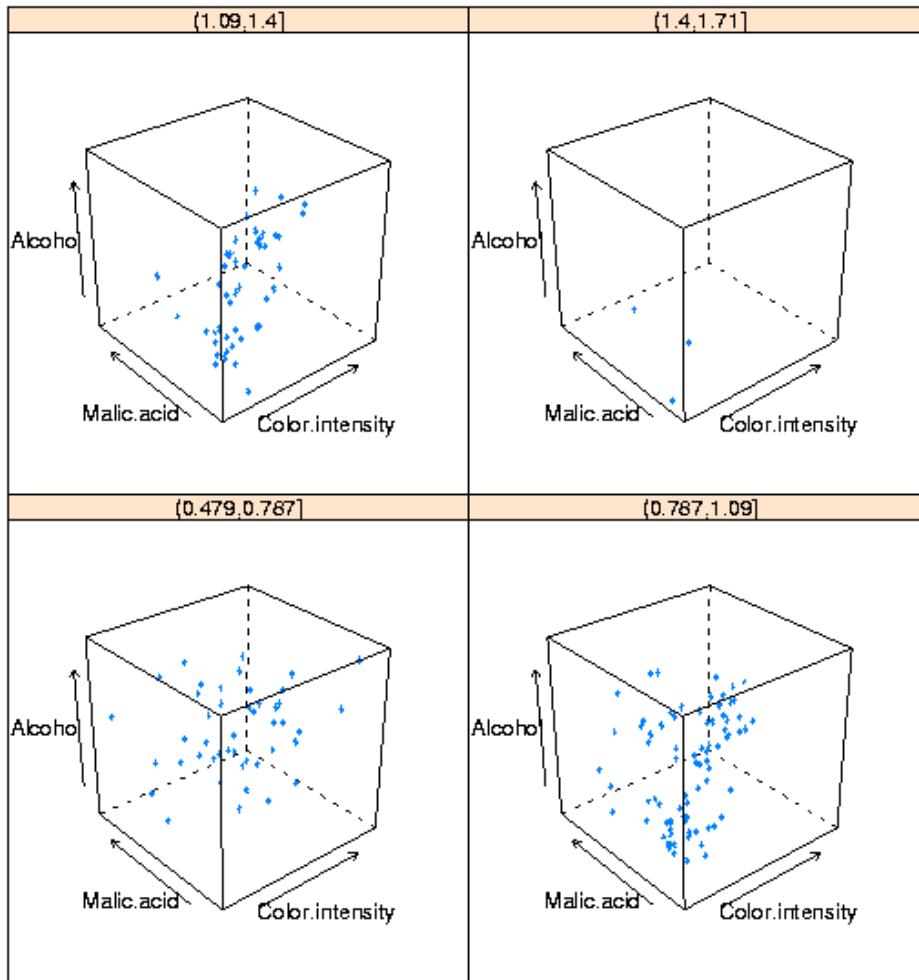
If the roles of `Freq` and `weekday` were reversed in the previous call to `barchart`, the bars would be drawn horizontally instead of vertically.

## 5.9 3-D Plots: `cloud`

The three-dimensional analog of the two-dimensional `xyplot` in the lattice library is `cloud`. By using a conditioning variable, `cloud` can allow us to consider the relationship of four variables at once. To illustrate, here's conditioning plot showing the relationship among four variables from the `wine` data frame:

```
cloud(Alcohol ~ Color.intensity + Malic.acid | cut(Hue, 4), data=wine)
```

The plot is shown below:



One important point about lattice graphics is that they operate slightly differently from R's traditional graphics. When you type a lattice command at the top level R session, it's R's automatic printing that actually displays the plot. So if you're calling a lattice function in a function or source file, you'll need to surround the call with a call to the `print` function, or store the result of the lattice plot in a variable, and print that variable.

# Chapter 6

## Spreadsheets and Databases

### 6.1 Spreadsheets

Spreadsheets, in particular the "xls" format used by Microsoft Excel, are one of the most common (if not the most common) methods for transferring data from one organization to another. When you download or receive a spreadsheet, it's often a good idea to make sure that it really is a spreadsheet. Comma-separated files, which are easily read into spreadsheet programs, are often erroneously labeled "spreadsheets" or "Excel files", and may even have an incorrect extension of ".xls". In addition, Excel spreadsheets are often used to hold largely textual information, so simply finding a spreadsheet that appears to contain data can be misleading. Furthermore, many systems are configured to use a spreadsheet program to automatically open files with an extension of ".csv", further adding to the confusion.

For many years, the only reliable way to read an Excel spreadsheet was with Microsoft Excel, which is only available on Windows and Mac OS computers; users on UNIX were left with no option other than finding a different computer. In the last few years, a number of programs, notably `gnumeric` and OpenOffice.org (usually available through the `ooffice` command) have been developed through careful reverse engineering to allow Unix users the ability to work with these files. To insure its advantage in the marketplace, Microsoft doesn't publish a detailed description of exactly how it creates its spreadsheet files. There are often "secret" ways of doing things that are only known to the developers within Microsoft. Reverse engineering means looking at the way a program handles different kinds of files, and then trying to write a program that imitates what the other program does. The end result of all this is that there are several ways to get the data from spreadsheet files into R.

Spreadsheets are organized as a collection of one or more sheets, stored in a single file. Each of the sheets represents a rectangular display of rows and columns, not unlike a data frame. Unfortunately, people often embed text, graphics and pictures into spreadsheets, so it's not a good idea to assume that a spreadsheet has the consistent structure of a data frame or matrix, even if portions of it do appear to have that structure. In addition, spreadsheets often have a large amount of header information (more than just a list of variable names), sometimes making it challenging to figure out the correct variable names to use for the

different columns in the spreadsheet.

Our main concern will be in getting the information from the spreadsheet into R: we won't look at how to work with data within Excel.

If you are simply working with a single Excel spreadsheet, the easiest way to get it into R is to open the spreadsheet with any compatible program, go to the File menu, and select Save As. When the file selector dialog appears, notice that you will be given an option of different file format choices, usually through a drop down menu. Choose something like "Tab delimited text", or "CSV (Comma separated values)". If any of the fields in the spreadsheet contain commas, tab delimited is a better choice, but generally it doesn't matter which one you use. Now, the file is (more or less) ready to be read into R, using `read.csv` or `read.delim`. A very common occurrence is for fields stored in a spreadsheet to contain either single or double quotes, in which case the `quote=` argument can be used to make sure R understands what you want. Spreadsheets often arrange their data so that there are a different number of valid entries on different lines, which will confuse `read.table`; the `fill=` argument may be of use in these cases. Finally, as mentioned previously, there are often multiple descriptive lines of text at the top of a spreadsheet, so some experimentation with the `skip=` argument to `read.table` may be necessary. Alternatively, you can simply delete those lines from the file into which you save the tab- or comma-delimited data.

As you can see, this is a somewhat time-consuming process which needs to be customized for each spreadsheet. It's further complicated by the fact that it only handles one sheet of a multi-sheet spreadsheet; in those cases the process would need to be repeated for each sheet. If you need to read a number of spreadsheets, especially ones where you need to access the data from more than one sheet, you need to use a more programmable solution.

R provides several methods to read spreadsheets without having to save them as tab- or comma-delimited files. The first is the `read.xls` function in the `gdata` library. This function uses a Perl program (included as part of the library), that converts the spreadsheet to a comma-separated file, and then uses `read.csv` to convert it to a data frame. In order for this library to work, perl must also be installed on the computer that's running R. perl will be installed on virtually any Unix-based computer (including Mac OSX), but will most likely not be on most Windows computers (although once perl is installed on a Windows system, the `read.xls` function will work with no problems). However there is a Windows-only package , `xlsReadWrite`, available from CRAN which can both read and write Excel files. (On other platforms, the `dataframes2xls` package provides a `write.xls` function.)

Another method utilizes a facility known as ODBC (Open DataBase Connectivity), which is a method that allows different programs that handle data to communicate with each other. The ODBC interface uses a query language known as SQL, which we'll study later, so for now I'll just describe the use of `read.xls`. (Remember, for just one or two data tables from a spreadsheet, saving as a tab- or comma-delimited file and using the correct `read.table` variant will usually be the easiest route. )

Since `read.xls` does nothing more than convert the spreadsheet to comma-separated form and then call `read.csv`, the guidelines for using it are very similar to the normal use of `read.csv`. The most useful feature of `read.xls` is that it accepts an optional `sheet=`

argument, allowing multiple sheets from a spreadsheet file to be automatically read.

As a simple example, consider a spreadsheet with listings for the top 200 educational institutions in the US with respect to the number of postdoctoral students, which I found at [http://mup.asu.edu/Top200-III/2\\_2005\\_top200\\_postdoc.xls](http://mup.asu.edu/Top200-III/2_2005_top200_postdoc.xls)

Here's a view of how the data looks in the UNIX gnumeric spreadsheet:

The screenshot shows a gnumeric spreadsheet window. The title bar includes 'File Edit View Insert Format Tools Data Help'. The menu bar has 'Verdana' selected. The toolbar includes icons for file operations, zoom, and cell selection. The formula bar shows 'B4 = Harvard University'. The main area displays a table titled 'The Top 200 Institutions--Postdoctoral Appointees (2005)'. The table has columns: 'Top 50 Institutions in Postdoctoral Appointees (2005)', 'Number of Postdocs', 'National Rank', 'Control Rank', and 'Institutional Control'. The data starts with Harvard University at rank 1. The 'Institutional Control' column shows 'Private' for most entries and 'Public' for a few. The bottom of the table shows a footer row with 'Sum=0'.

<b>The Top 200 Institutions--Postdoctoral Appointees (2005)</b>				
Top 50 Institutions in Postdoctoral Appointees (2005)	Number of Postdocs	National Rank	Control Rank	Institutional Control
Harvard University	4384	1	1	Private
Johns Hopkins University	1442	2	2	Private
Stanford University	1259	3	3	Private
University of California - Los Angeles	1094	4	1	Public
Yale University	1012	5	4	Private
University of California - San Francisco	1003	6	2	Public
University of Washington - Seattle	963	7	3	Public
University of California - San Diego	886	8	4	Public
Massachusetts Institute of Technology	851	9	5	Private
University of Pennsylvania	815	10	6	Private
Columbia University	793	11	7	Private
University of California - Berkeley	774	12	5	Public
Duke University	755	13	8	Private
University of Pittsburgh - Pittsburgh	722	14	6	Public
University of Minnesota - Twin Cities	669	15	7	Public
University of California - Davis	659	16	8	Public
University of Colorado - Boulder	651	17	9	Public
University of North Carolina - Chapel Hill	627	18	10	Public
Emory University	606	19	9	Private
University of Michigan - Ann Arbor	603	20	11	Public

Here are the first few lines of the file created when I use File->Save As, and choose the CSV format:

```
"The Top 200 Institutions--Postdoctoral Appointees
(2005)",,,,,
,,,,,
"Top 50 Institutions
in Postdoctoral Appointees
(2005)","Number of Postdocs","National Rank","Control Rank","","Institutional
Control"
"Harvard University",4384,1,1,,Private
"Johns Hopkins University",1442,2,2,,Private
"Stanford University",1259,3,3,,Private
"University of California - Los Angeles",1094,4,1,,Public
"Yale University",1012,5,4,,Private
"University of California - San Francisco",1003,6,2,,Public
"University of Washington - Seattle",963,7,3,,Public
"University of California - San Diego",886,8,4,,Public
"Massachusetts Institute of Technology",851,9,5,,Private
"University of Pennsylvania",815,10,6,,Private
"Columbia University",793,11,7,,Private
```

```
"University of California - Berkeley",774,12,5,,Public
```

While the data itself looks fine, it's unlikely that the header information will be of much use. Since there are 7 header lines, I could use the following statements to read the data into R:

```
> fromcsv = read.csv('2_2005_top200_postdoc.csv',header=FALSE,  
+ skip=7,stringsAsFactors=FALSE)  
> dim(fromcsv)  
[1] 206   6  
> head(fromcsv)
```

		V1	V2	V3	V4	V5	V6
1		Harvard University	4384	1	1	NA	Private
2		Johns Hopkins University	1442	2	2	NA	Private
3		Stanford University	1259	3	3	NA	Private
4	University of California - Los Angeles	1094	4	1	NA	Public	
5		Yale University	1012	5	4	NA	Private
6	University of California - San Francisco	1003	6	2	NA	Public	

The fifth column can be removed (take a look at the spreadsheet to see why), and we should supply some names to the spreadsheet. One problem with this spreadsheet is that it repeats its header information several times part way through the spreadsheet. These lines will have to be removed manually. This can be identified by the fact that the sixth variable should be either Public or Private, and could be eliminated as follows:

```
> fromcsv = fromcsv[fromcsv$V6 %in% c('Public','Private'),]
```

The fifth variable can be removed by setting it to NULL, and the columns can be named.

```
> fromcsv$V5 = NULL  
> names(fromcsv) = c('Institution','NPostdocs','Rank','ControlRank','Control')
```

Finally, a check of the columns shows that, because of the header problem, the numeric variables explicitly need to be converted:

```
> sapply(fromcsv,class)  
Institution      NPostdocs          Rank ControlRank      Control  
"character" "character" "character" "character" "character"  
> fromcsv[,c(2,3,4)] = sapply(fromcsv[,c(2,3,4)],as.numeric)
```

A similar procedure could be carried out with `read.xls`, although it might take some experimentation with `skip=` to make things work properly. The only other difference between `read.xls` and `read.csv` for this example is that a space mysteriously appeared at the end of one of the variables when using `read.xls`:

```
library(gdata)  
fromreadxls = read.xls('2_2005_top200_postdoc.xls',stringsAsFactors=FALSE,
```

```

header=FALSE,skip=2)
fromreadxls = fromreadxls[-1,-c(1,6)]
fromreadxls$V7 = sub(' ', '', fromreadxls$V7)
fromreadxls = fromreadxls[fromreadxls$V7 %in% c('Public','Private'),]
fromreadxls[,c(2,3,4)] = sapply(fromreadxls[,c(2,3,4)],as.numeric)
names(fromreadxls) = c('Institution','NPostdocs','Rank','ControlRank','Control')

```

## 6.2 Writing Spreadsheets

As with reading a spreadsheet, one option for writing a spreadsheet is to write out a comma- or tab-separated file, and to use a spreadsheet program to read it and then save it as a “.xls” file. The `write.csv`, and `write.table` functions are useful in this regard. For example, we could create a comma-separated version of the `world` data frame as follows:

```
> write.csv(world,file='world.csv')
```

Here's the result of reading the csv file with a spreadsheet program:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	country	cont	gdp	income	literacy	military	phys							
2	1 Albania	EU	4500	4937	98.7	5.6E+07	138.929							
3	2 Algeria	AF	5900	6799	69.8	2.5E+09	84.6							
4	3 Angola	AF	1900	2457	66.8	1.8E+08	7.7							
5	4 Argentina	SA	11200	12468	97.2	4.3E+09	300.947							
6	5 Armenia	AS	3900	3806	99.4	1.4E+08	352.586							
7	6 Australia	OC	28900	29893	99.9	1.7E+10	249.13							
8	7 Austria	EU	30000	31254	99.9	1.5E+09	324.285							
9	8 Azerbaijan	AS	3400	3968	98.8	1.2E+08	353.562							
10	9 Bahrain	AS	17100	18817	87.7	6.3E+08	159.570							
11	10 Bangladesh	AS	1900	1875	41.1	1.0E+09	23.0678							
12	11 Belarus	EU	6000	6646	99.6	1.8E+08	449.651							
13	12 Belgium	EU	29000	29707	99.9	4E+09	418.360							
14	13 Belize	NA	4900	7339	76.9	1.8E+07	104.502							
15	14 Benin	AF	1100	1094	33.6	9.6E+07	5.7591							
16	15 Bhutan	AS	1300	3095	47	1.4E+07	5.1405							
17	16 Bolivia	SA	2400	2710	86.5	1.3E+08	73.3371							
18	17 Bosnia	ar EU	6100	5504	94.6	2.3E+08	133.840							
19	18 Botswana	AF	8800	10169	78.9	3.4E+08	28.7644							
20	19 Brazil	SA	7600	8049	88.4	1.1E+10	205.648							
21	20 Brunei	AS	18600	24143	92.7	2.9E+08	100.589							

The `File->Save As` menu would allow saving the spreadsheet in an appropriate format.

To automate the process of writing spreadsheets, the Windows-only `xlsReadWrite` package provides a `write.xls` function; for other operating systems, this function is provided by the `dataframe2xls` package.

## 6.3 Databases

A database is a collection of data, usually with some information (sometimes called metadata) about how the data is organized. But many times when people refer to a database, they mean a database server, which is similar to a web server, but responds to requests for data instead of web pages.

By far the most common type of database server is known as a relational database management system (RDBMS), and the most common way of communicating with such a database is a language known as SQL, which is an acronym for Structured Query Language. Some examples of database systems that use SQL to communicate with an RDBMS include Oracle, Sybase, Microsoft SQL Server, SQLite, MySQL and Postgres. While there is an SQL standard, each database manufacturer provides some additional features, so SQL that works with one database is not guaranteed to work on another. We'll try to stick to aspects of SQL that should be available on most SQL based systems.

A Database consists of one or more tables, which are generally stored as files on the computer that's running the DBMS. A table is a rectangular array, where each row represents an observation, and each column represents a variable. Most databases consist of several tables, each containing information about one aspect of the data. For example, consider a database to hold information about the parts needed to build a variety of different products. One way to store the information would be to have a data table that had the part id, its description, the supplier's name, and the cost of the part. An immediate problem with this scheme concerns how we could store information about the relation between products and parts. Furthermore, if one supplier provided many different parts, redundant information about suppliers would be repeated many times in the data table. In the 1970s when database technology was first being developed, disk and memory space were limited and expensive, and organizing data in this way was not very efficient, especially as the size of the data base increased. In the late 1970s, the idea of a relational database was developed by IBM, with the first commercial offering coming from a company which is now known as Oracle. Relational database design is governed by a principle known as normalization. While entire books are devoted to the subject, the basic idea of normalization is to try and remove redundancy as much as possible when creating the tables that make up a data base. Continuing the parts example, a properly normalized database to hold the parts information would consist of four tables. The first would contain a part id to uniquely identify the part, its description, price and an id to identify the supplier. A second table would contain the supplier codes and any other information about the supplier. A third table would contain product ids and descriptions, while the final table would have one record for each part used in each product, stored as pairs of product id and part id. The variables that link together the different databases are referred to as keys, or sometimes foreign keys. Clearly, making sure that there are keys to link information from one table to another is critical to the idea of a normalized data base. This allows large amounts of information to be stored in manageable sized tables which can be modified and updated without having to change all of the information in all of the tables. Such tables will be efficient in the amount of disk and memory resources that they need, which was critical at the time such databases were developed. Also critical to

this scheme is a fast and efficient way of joining together these tables so that queries like "Which suppliers are used for product xyz?" or "What parts from Acme Machine Works cost between \$2 and \$4" or "What is the total cost of the parts needed to make product xyz?". In fact, for many years the only programs that were capable of combining data from multiple sources were RDBMSs.

We're going to look at the way the SQL language is used to extract information from RDBMSs, specifically the open source SQLite data base (<http://sqlite.org>) (which doesn't require a database server), and the open source MySQL data base (<http://www.mysql.com>) which does. In addition we'll see how to do typical database operations in R, and how to access a database using SQL from inside of R.

One of the first questions that arises when thinking about databases is "When should I use a database?" There are several cases where a database makes sense:

1. The only source for the data you need may be an existing database.
2. The data you're working with changes on a regular basis, especially if it can potentially be changed by many people. Database servers (like most other servers) provide concurrency, which takes care of problems that might otherwise arise if more than one person tries to access or modify a database at the same time.
3. Your data is simply too big to fit into the computer's memory, but if you could conveniently extract just the parts you need, you'd be able to work with the data. For example, you may have hundreds of variables, but only need to work with a small number at a time, or you have tens of thousands of observations, but only need to work with a few thousand at a time.

One of the most important uses of databases in the last ten years has been as back ends to dynamic web sites. If you've ever placed an order online, joined an online discussion community, or gotten a username/password combination to access web resources, you've most likely interacted with a database. In addition both the Firefox and Safari browsers use SQLite databases to store cookies, certificates, and other information.

There are actually three parts to a RDBMS system: data definition, data access, and privilege management. We're only going to look at the data access aspect of databases; we'll assume that a database is already available, and that the administrator of the database has provided you with access to the resources you need. You can communicate with a database in at least three different ways:

1. You can use a command line program that will display its results in a way similar to the way R works: you type in a query, and the results are displayed.
2. You can use a graphical interface (GUI) that will display the results and give you the option to export them in different forms, such as a comma-separated data file. One advantage to these clients is that they usually provide a display of the available databases, along with the names of the tables in the database and the names of the columns within the tables.

3. You can use a library within R to query the database and return the results of the query into an R data frame. For use with MySQL library, the `RMySQL` library is available for download from CRAN; for SQLite, the `RSQlite` package is available. When you install either, the `DBI` library, which provides a consistent interface to different databases, will also be installed.

It's very important to understand that SQL is not a programming language – it's said to be a declarative language. Instead of solving problems by writing a series of instructions or putting together programming statements into functions, with SQL you write a single statement (query) that will return some or all of the records from a database. SQL was designed to be easy to use for non-programmers, allowing them to present queries to the database in a language that is more similar to a spoken language than a programming language. The main tool for data access is the select statement. Since everything needs to be done with this single statement, descriptions of its syntax can be quite daunting. For example, here's an example that shows some of the important features of the select statement:

```
SELECT columns or computations
      FROM table
      WHERE condition
      GROUP BY columns
      HAVING condition
      ORDER BY column [ASC | DESC]
      LIMIT offset, count;
```

(The keywords in the above statement can be typed in upper or lower case, but I'll generally use upper case to remind us that they are keywords.) MySQL is case sensitive to table and column names, but not for commands. SQLite is not case sensitive at all. SQL statements always end with a semi-colon, but graphical clients and interfaces in other languages sometimes don't require the semi-colon. Fortunately, you rarely need to use all of these commands in a single query. In fact, you may be able to get by knowing just the command that returns all of the information from a table in a database:

```
SELECT * from table;
```

## 6.4 Working with Databases

When you're not using a graphical interface, it's easy to get lost in a database considering that each database can hold multiple tables, and each table can have multiple columns. Three commands that are useful in finding your way around a database are:

MySQL	SQLite
SHOW DATABASES;	.databases
SHOW TABLES IN database;	.tables
SHOW COLUMNS IN table;	pragma table.info(table)
DESCRIBE table;	.schema table

For use as an example, consider a database to keep track of a music collection, stored as an SQLite database in the file `albums.db`. We can use the commands just shown to see how the database is organized. In class, I sometimes used some of the graphical SQLite clients that are mentioned on the Class Resources page. Here, I'm using the UNIX command-line client called `sqlite3`:

```
springer.spector$ sqlite3 albums.db
SQLite version 3.4.2
Enter ".help" for instructions
sqlite> .mode column
sqlite> .header on
sqlite> .tables
Album      Artist      Track
title      TEXT
);
sqlite> .schema Album
CREATE TABLE Album
( aid INTEGER,
alid INTEGER,
title TEXT
);
CREATE INDEX jj on Album(alid);
sqlite> .schema Artist
CREATE TABLE Artist
( aid INTEGER,
name TEXT
);
CREATE INDEX ii on Artist(aid);
```

Studying these tables we can see that the organization is as follows: each track in the `Track` table is identified by an artist id (`aid`) and an album id (`alid`). To get the actual names of the artists, we can link to the `Artist` database through the `aid` key; to get the names of the albums, we can link to the `Album` table through the `alid` key. These keys are integers, which don't take up very much space; the actual text of the album title or artist's name is stored only once in the table devoted to that purpose. (This is the basic principle of normalization that drives database design.) The first thing to notice in schemes like this is that the individual tables won't allow us to really see anything interesting, since they just contain keys, not actual values; to get interesting information, we need to join together multiple tables. For example, let's say we want a list of album titles and artist names. We want to extract the `Name` from the `Artist` table and combine it with `Title` from the `Album` table, based on a matching value of the `aid` variable:

```
sqlite> .width 25 50
sqlite> SELECT Artist.name, Album.title FROM Artist, Album
```

```

...> WHERE Artist.aid = Album.aid;
Gloria Wood           Wood by the Fire
Bobby Hackett        Jazz Session
Bobby Hackett        1943-1947
Bobby Hackett        Sextet
Bobby Hackett        Blues With A Kick
Bobby Hackett        Coast Concert
Bobby Hackett        Creole Cookin'
Bobby Hackett        In Concert
George Wettling's Jazz Ba Jazz Session
Jimmy Forrest         most much!
Jimmy Forrest         Night Train
Jimmy Forrest         Sit Down And Relax With Jimmy Forrest
Jimmy Forrest         Soul Street
Jimmy Forrest         Out of the Forrest (1961)
Jimmy Forrest         Forrest Fire
Jimmy Forrest         Black Forrest
Mary Lou Williams     1927-1940
Mary Lou Williams     Mary Lou Williams Presents Black Christ of the And
.
.
```

Queries like this are surprisingly fast – they are exactly what the database has been designed and optimized to achieve.

There are a few conveniences that you can use in SQL to save typing and improve the appearance of the output. You can use the AS keyword to provide an alternative name for a database for use within a query, or to rename a column so that it will display with a different heading. The previous query could have been written:

```

sqlite> SELECT ar.name AS Artist, al.title AS 'Album Name'
...>      FROM Artist AS ar, Album AS al
...>      WHERE ar.aid = al.aid LIMIT 200,10;
Artist                  Album Name
-----
Mundell Lowe            TransitWest
Mundell Lowe            A Grand Night For Swinging
Mundell Lowe            The Mundell Lowe Quartet
Duke Ellington          Jazz Violin Session
Duke Ellington          Happy-Go-Lucky Local
Duke Ellington          The Great Paris Concert - Disc 2
Duke Ellington          The Great Paris Concert
Duke Ellington          Piano Reflections
Duke Ellington          Continuum
Duke Ellington          At Newport

```

Note the new headings that appear in the output. The **LIMIT** keyword allows you to display just some of the results. Given a single number, it prints that many lines starting from the beginning; given two numbers it starts at the first number and prints as many records as specified in the second number. Combining two data sets in this way is a very common operation, known as an inner join. An alternative way of expressing the query we're looking at would be as follows:

```
SELECT ar.name AS Artist, al.title AS 'Album Name'  
      FROM Artist AS ar inner join Album AS al  
        using(aid) limit 200,10;
```

When the two tables being joined share exactly one common column (key), the operation is sometimes known as a natural join. Since this is the case with the **Album** and **Artist** tables, yet another way to express the query is

```
SELECT ar.name AS Artist, al.title AS 'Album Name'  
      FROM Artist AS ar natural join Album AS al limit 200,10;
```

Note that if the key used to join two tables does not have the same name in each of the tables, you must use the previous method that specifies the matching condition in the **WHERE** clause.

SQL provides some convenient facilities for aggregating values based on counts, sums, averages and minimums and maximums, through the **COUNT**, **SUM**, **AVG**, **MIN** and **MAX** functions. To find the number of records that match a certain condition, the expression **COUNT(\*)** can be used. For example, to find the number of observations in the **Track** data set, we could use:

```
sqlite> SELECT COUNT(*) from Track;  
COUNT(*)  
-----  
54396
```

To count how many tracks were longer than 1000, we could use

```
sqlite> SELECT COUNT(*) from Track WHERE length > 1000;  
COUNT(*)  
-----  
129
```

To find the track with the longest length in the **Track** table, we could try:

```
sqlite> SELECT MAX(length) from Track;  
MAX(length)  
-----  
2049
```

This gives us the length of the maximum track, but not its title. The easiest way to get around this is to use a subquery. In SQL, a subquery is a SELECT statement surrounded by parentheses. You can use the values returned by the subquery in WHERE or HAVING clauses. So to find the track with the maximum length, we could use

```
sqlite> SELECT title,length FROM Track WHERE length = (SELECT max(length) FROM Track);
title           length
-----
Hues Of Melanin      2049
```

Of course, this query would be more useful if we knew the name of the album and artist that the track came from:

```
sqlite> .width 20 25 25 5
sqlite> SELECT tr.title,al.title,ar.name,tr.length
...>     FROM Track as tr, Album as al, Artist as ar
...>     WHERE tr.alid = al.alid and tr.aid = al.aid and ar.aid = al.aid
...>     AND length = (SELECT max(length) FROM Track);
title           title           name           length
-----
Hues Of Melanin      Live          Sam Rivers Trio      2049
```

When you construct queries like this, it's extremely important to list *all* of the WHERE conditions that need to be satisfied to avoid getting meaningless duplicate records.

As another example of a subquery, suppose we want to count how many albums we have for each artist, and then tabulate how many artists there are with one album, two albums, three albums, and so on. The first part is fairly easy:

```
sqlite> SELECT aid,count(aid) FROM Album GROUP BY aid limit 100;
aid           count(aid)
-----
1              1
2              1
3              1
4              1
5              1
6              1
. . .
```

We can use this result as a subquery to see the distribution of albums per artist:

```
mysql> SELECT ct,count(ct) FROM
->     (SELECT aid,count(aid) AS ct FROM Album GROUP BY aid)
->     GROUP by ct ORDER by ct;
ct           count(ct)
```

```

-----
1          2303
2          302
3          137
4          72
5          46
6          36
7          25
8          25
9          15
. . .

```

When using subqueries in this way, some databases may require that the subquery is given an alias. In such a case, a query like the following could be used:

```

SELECT ct, count(ct) FROM
    (SELECT aid, count(aid) AS ct FROM Album GROUP BY aid) AS x
GROUP by ct ORDER by ct;

```

One of the most useful features of SQL is that it can summarize large data sets, producing a smaller result set that can be more easily handled by other programs. For example, suppose we are interested in the total length (in seconds) of the tracks on each album. Instead of processing all the tracks outside of the database, we can create a summarized version of the database that has the lengths already summarized. Very few programs can operate on databases as fast as SQL, so operations like this can save lots of processing time. To perform operations like this we can use the `GROUP BY` clause:

```

sqlite> SELECT alid, SUM(length) AS TotalTime FROM Track GROUP BY alid limit 50;
alid           TotalTime
-----
1              2551
2              3710
3              2402
4              3339
5              3588
6              3207
7              4268
8              4409
9              4120
10             4249
. . .

```

Once again, the results would probably be more useful with the title of the album and the artist included. In this example, we can also sort them by the total time of the album:

```

sqlite> .header on
sqlite> .mode column
sqlite> .width 20 25 25 5
sqlite> SELECT ar.name,al.title,SUM(tr.length) AS TotalTime
...>      FROM Track as tr, Artist as ar,Album as al
...>      WHERE ar.aid = al.aid AND al.alid = tr.alid
...>      GROUP BY tr.alid
...>      ORDER by TotalTime DESC limit 100;
ar.name          al.title          TotalTime
-----
Jimmy Giuffre/Marty    West Coast Scene    13856
Curtis Fuller        Complete Blue Note UA Ses 11529
Clifford Brown       At The Cotton Club (1956) 11282
Count Basie          American Decca Recordings 11261
Onzy Matthews         Mosaic Select: Onzy Matth 9624
Cannonball Adderley   Sophisticated Swing    9486
Thelonius Monk        Live at the It Club    9289
. . .

```

In this particular example, instead of processing the 53000 or so tracks outside of SQL, we've reduced what we have to work with to around 5000 records, which could make the overall processing much faster. Exactly which aggregation functions will be available depends on the database you're using.

## 6.5 Regular Expressions in SQL

For things like partial matching of strings, all flavors of SQL provide the `LIKE` operator, which allows limited wildcard matching. With the `LIKE` operator the percent sign (%) stands for 0 or more characters, and the underscore (\_) stands for exactly one. Many modern databases (MySQL included) provide the `RLIKE` operator, which uses regular expressions. For the remainder of these examples, we'll use the MySQL database running on `springer.berkeley.edu`. We'll write a regular expression to find all the artists who have exactly three words in their names.

```
springer.spector$ mysql -u stat133 -p albums
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 5.0.51a-3ubuntu5.4-log (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SELECT name FROM Artist
->      WHERE name RLIKE '^[A-Za-z]+ [A-Za-z]+ [A-Za-z]+$';
+-----+
| name           |
+-----+
| Mary Lou Williams          |
| Dexter Gordon Quartet      |
| Modern Jazz Septet          |
| Eddie South Trio            |
| Gerry Mulligan Tentette     |
| Bill Harris rarities       |
| Jimmy Heath Sextet          |
| Horae Parlan Trio           |
+-----+
```

Note that in order to access the MySQL database, a username and password needed to be used.

If you look at the original definition of the `SELECT` statement, you'll notice the `HAVING` clause which we have not yet covered. The basic difference between `HAVING` and `WHERE` is that, due to the timing of operations in the database, we can't use computed variables (like `SUM` and `AVG`) in the `WHERE` clause, but you can in the `HAVING` clause. Suppose we wanted to find the artists for which there were more than five albums in the database. A first try might be as follows:

```

mysql > SELECT aid,COUNT(aid) as count FROM Album
      -> WHERE count > 5
      -> GROUP BY aid;
ERROR 1054 (42S22): Unknown column 'count' in 'where clause'

```

Since the WHERE clause doesn't have access to columns you've created through computations, MySQL can't find the computed variable (`count`). To fix this, add a HAVING clause after the GROUP BY clause:

```

mysql> SELECT aid,COUNT(aid) as count FROM Album
      ->     GROUP BY aid
      ->     HAVING count > 5;
+-----+
| aid | count |
+-----+
|   0 |    6 |
|  14 |    6 |
|  20 |    9 |
|  24 |    6 |
|  40 |   15 |
|  74 |    6 |
|  92 |    6 |
| 101 |    6 |
| 107 |    9 |

```

. . .

Of course, queries like this are infinitely more useful if they bring in the relevant information from other tables:

```

mysql> SELECT ar.name as Artist,COUNT(al.alid) AS count
      ->     FROM Artist as ar, Album as al
      ->     WHERE ar.aid = al.aid
      ->     GROUP BY al.aid HAVING count > 5;
| Abdullah Ibrahim          |    6 |
| Ahmad Jamal                |    9 |
| Al Cohn                   |    6 |
| Al Haig                   |   15 |
| Andre Previn               |    6 |
| Anita O'Day                |    6 |
| Archie Shepp               |    6 |

```

. . .

## 6.6 Accessing databases in R

To provide consistent access to different databases, R uses an intermediate layer of communication known as the DBI (Data Base Interface) library. As we've seen before, different commands are necessary to get information about different databases, so the DBI layer provides a set of routines that will work identically for all databases. Some of these routines are described in the following table.

<code>dbConnect</code>	connect to a database
<code>dbDisconnect</code>	close the connection to a database
<code>dbExistsTable</code>	returns TRUE or FALSE
<code>dbGetQuery</code>	send a query to the database and get the results
<code>dbListTables</code>	shows all the tables in a database
<code>dbListFields</code>	shows the names of columns in a database
<code>dbSendQuery</code>	send a query to the database and use <code>fetch</code> to get results
<code>dbWriteTable</code>	stores a data frame in a database

To let the DBI layer know what type of database you are using, you pass a string (like '`MySQL`' or '`SQLite`') to the `dbDriver` function. This will return an object which can be passed to `dbConnect` to make a connection to the database. `dbConnect` returns a connection object which is passed to all the other routines so that they can communicate with the database you are using.

For example, to connect to the `albums` SQLite database, we'd use the following statements:

```
library(RSQLite)
drv = dbDriver('SQLite')
con = dbConnect(drv, db='albums.db')
```

Since SQLite doesn't require a database server, you can easily create SQLite databases using `dbWriteTable`. For very large data sets, you can process the data in chunks, using the `append=` option of `dbWriteTable`.

To use MySQL, you must have a valid username and password. For the MySQL database running on `springer.berkeley.edu`, you'd need to use the following statements:

```
drv = dbDriver('MySQL')
con = dbConnect(drv, dbname='albums', user='stat133', pass='T0pSecr3t')
```

If you're using an SCF machine, you'd need to specify a hostname (`springer`) when connecting; for non-SCF machines, you would set up an SSH tunnel and use a hostname of '`127.0.0.1`'.

Once you get the connection object from `dbConnect` you can use it repeatedly to make as many queries as you need, and the functions you use are the same, regardless of the nature of the underlying database.

It's up to you to decide how much of the work you want to do through the database, and how much you'll do in R. For example, suppose we want to find albums in the `albums`

database whose total time (sum of the `length` variable in the `Track` table) is between 2400 and 2700 seconds, displayed in descending order of total time.

One approach (after loading the library and making the connection to the database) would be to do all the work in the database. Notice that this may be your only option if the database you're dealing with is too large to fit into R.

```
> albums = dbGetQuery(con,statement='SELECT al.title,ar.name,SUM(tr.length) AS tot
+                               FROM Album AS al,Artist AS ar,Track AS tr
+                               WHERE tr.alid = al.alid AND tr.aid = ar.aid AND tr.aid = al.aid
+                               GROUP BY tr.alid
+                               HAVING tot BETWEEN 2400 AND 2700 ORDER BY tot DESC')
> head(albums)
      title          name   tot
1 Blues Groove Tiny Grimes & Coleman Hawkins 2699
2 I Need Some Money          Eddie Harris    2699
3 Young Chet                 Chet Baker    2699
4 Up & Down                  Horace Parlan 2698
5 Arcadia Shuffle            Roy Eldridge 2697
6 Night Dance                Jimmy Giuffre 2697
```

It's usually most convenient to spread out the query over multiple lines, but the entire query could be passed as one long string. The `dbGetQuery` function gets the results of the query and puts them into a data frame, in this case called `albums`.

Another way to find the albums with total time between 2400 and 2700 would be to use SQL to get the total time of all the albums, and then use the `subset` function in R to get the ones in the desired time range:

```
> albums = dbGetQuery(con,statement='SELECT al.title,ar.name,
+                               SUM(tr.length) AS "Total Time"
+                               FROM Album AS al,Artist AS ar,Track AS tr
+                               WHERE tr.alid = al.alid AND tr.aid = ar.aid AND tr.aid = al.aid
+                               GROUP BY tr.alid')
> myalbums = subset(albums,'Total Time' > 2400 & 'Total Time' < 2700)
> myalbums = myalbums[order(myalbums$"Total Time",decreasing=TRUE),]
> head(myalbums)
      title          name Total Time
821 Blues Groove Tiny Grimes & Coleman Hawkins 2699
2206 I Need Some Money          Eddie Harris    2699
5632 Young Chet                 Chet Baker    2699
5337 Up & Down                  Horace Parlan 2698
418 Arcadia Shuffle            Roy Eldridge 2697
3390 Night Dance                Jimmy Giuffre 2697
```

Here I used a more descriptive name for the total time, namely `Total Time`. Note that, since there's a space in the variable name, I need to surround it with backquotes (`) in the

`subset` function, but when I refer to the variable in the `order` function, I used ordinary quotes. Backquotes will work in either case.

At the other end of the extreme, you could read in the entire database and work with it entirely in R. This may not be the wisest approach, because if the database is too large, you may have problems. Working with the data in this way, however will allow us to see the R equivalent of some of the database operations we've been studying. Once again, I'm assuming that the `RMySQL` library has been loaded, and a connection object has been obtained through a call to `dbConnect`.

```
> album = dbGetQuery(con,statement="select * from Album")
> track = dbGetQuery(con,statement="select * from Track")
> artist = dbGetQuery(con,statement="select * from Artist")
```

We now have three data frames in R corresponding to the three data tables in the database. In R, the equivalent of an inner join can be carried out with the `merge` function. You provide `merge` with the data frames you wish to join, and a character string or vector of character strings with the name(s) of the variables that you wish to join them by. In database terminology, these names are the keys for the join. The `merge` function will only accept two data frames, but it can be called repeatedly in case there are more than two data frames that need to be merged.

If there are variables with the same name (other than the keys) in the two data frames to be merged, `merge` will rename them by adding either `.x` or `.y` to the end of the name to tell you which data frame that variable came from. An often easier approach is to rename variables before merging so that all the names (other than the keys) will be unique. In the current example, the variable `title` is in both the `album` data frame and the `track` data frame. Let's rename the `title` variable in the `album` data frame to `albumname`:

```
> names(album)
[1] "alid"   "aid"    "title"  "year"
> names(album)[3] = 'albumname'
```

Now we can merge the `album` data frame with the `artist` data frame using `aid` as a key, and then merge the `track` data frame with the `album` data frame using `alid` as a key.

```
> album = merge(album,artist,by='aid')
> track = merge(track,album,by='alid')
> head(track)
  alid aid.x          title filesize bitrate length aid.y albumname
1     1  2836 Sonnymoon For Two      6459     189     279  2836 100% Proof
2     1  2836             Nutty      6062     191     259  2836 100% Proof
3     1  2836        100% Proof    19877     190     853  2836 100% Proof
4     1  2836       Bluesology     7101     189     306  2836 100% Proof
5     1  2836  Night In Tunisia    9102     190     391  2836 100% Proof
6     1  2836      Milestones    10774     190     463  2836 100% Proof
```

```

      name
1 Tubby Hayes Orchestra
2 Tubby Hayes Orchestra
3 Tubby Hayes Orchestra
4 Tubby Hayes Orchestra
5 Tubby Hayes Orchestra
6 Tubby Hayes Orchestra

```

To calculate the total length of tracks, we only need the album and artist names along with the length of each track:

```
> track = track[,c('albumname','name','length')]
```

The rest of the processing is pretty straightforward now that everything is in R:

```

> tot = aggregate(track$length,list(albumname=track$albumname,artist=track$name),sum)
> use = tot[tot$x > 2400 & tot$x <= 2700,]
> use = use[order(use$x,decreasing=TRUE),]
> head(use)

      albumname          artist    x
1105   Young Chet        Chet Baker 2699
1748 I Need Some Money  Eddie Harris 2699
5245   Blues Groove Tiny Grimes & Coleman Hawkins 2699
2520       Up & Down      Horace Parlan 2698
2848     Night Dance     Jimmy Giuffre 2697
4559   Arcadia Shuffle   Roy Eldridge 2697

```

Because of the way `merge` operates, ties are resolved differently than in previous examples, but the results are essentially the same.

A different approach illustrates the relationship between factors and the principles of normalization. Since a factor stores each of its levels only once, normalized tables can often be used to form factors in R. Once again, let's assume that we've loaded the `RMySQL` library and obtained a connection object. This time, I'll trim off the unneeded variables from the `track` data frame before processing:

```

> album = dbGetQuery(con,statement="select * from Album")
> track = dbGetQuery(con,statement="select * from Track")
> artist = dbGetQuery(con,statement="select * from Artist")
> track = track[,c('aid','alid','length')]
> names(track) = c('artist','album','length')
> track$artist = factor(track$artist,levels=artist$aid,labels=artist$name)
> track$album = factor(track$album,levels=album$alid,labels=album$title)

```

The tables formed in the process of normalizing the database essentially contain the levels and labels of a factor variable in R. Processing the data set is the same as in the previous example:

```

> track$albart = paste(track$album,track$artist,sep='\t')
> tot = aggregate(track$length,track['albart'],sum)
> aa = strsplit(tot$albart,'\\t')
> tot$album = sapply(aa, '[',1)
> tot$artist = sapply(aa, '[',2)
> tot$albart = NULL
> use = tot[tot$x > 2400 & tot$x <= 2700,]
> use = use[order(use$x,decreasing=TRUE),]
> head(use)

      x                      album            artist
821 2699    Blues Groove Tiny Grimes & Coleman Hawkins
2206 2699 I Need Some Money           Eddie Harris
5632 2699     Young Chet             Chet Baker
5337 2698       Up & Down          Horace Parlan
418 2697   Arcadia Shuffle        Roy Eldridge
3390 2697     Night Dance         Jimmy Giuffre

```

In practice, the most useful solutions would probably fall in between the extreme of doing everything on the database and doing everything in R.

## 6.7 Using SQL in R

If you like using SQL, or you see a handy way to get something done using SQL, you can use the `sqldf` package to operate on R data frames using SQL. When you pass a query to the `sqldf` function, it will identify the data frames involved in the query, create a temporary SQLite database, perform the query, and return the results. For example, consider the task of merging two data frames, based on the value of a variable `id`:

```

> data1 = data.frame(id=c(1,4,2,3,5),x=c(7,12,19,15,9))
> data2 = data.frame(id=c(2,3,4,1,5),y=c(21,32,29,35,19))
> sqldf('select * from data1 natural join data2')
  id  x  y
1  1  7 35
2  4 12 29
3  2 19 21
4  3 15 32
5  5  9 19

```

## 6.8 Reading Spreadsheets with the RODBC Library

Now that we've got the basics of SQL down, we can look at a third option for reading spreadsheets into R, the `RODBC` library. Note that this functionality is only available on

Windows and you don't need a copy of Excel on your computer in order for it to work. The ODBC interface provides access to spreadsheets using SQL as a query language. Here's a sample session showing how we could read the spreadsheet from the previous example using RODBC:

```

> library(RODBC)
> con = odbcConnectExcel('2_2005_top200_postdoc.xls')
> tbls = sqlTables(con)
> tbls
      TABLE_CAT
1 C:\\\\Documents and Settings\\\\spector\\\\Desktop\\\\2_2005_top200_postdoc
2 C:\\\\Documents and Settings\\\\spector\\\\Desktop\\\\2_2005_top200_postdoc
3 C:\\\\Documents and Settings\\\\spector\\\\Desktop\\\\2_2005_top200_postdoc
4 C:\\\\Documents and Settings\\\\spector\\\\Desktop\\\\2_2005_top200_postdoc
5 C:\\\\Documents and Settings\\\\spector\\\\Desktop\\\\2_2005_top200_postdoc

      TABLE_SCHEM   TABLE_NAME   TABLE_TYPE REMARKS
1 <NA>           Sheet1$ SYSTEM TABLE  <NA>
2 <NA>           Sheet2$ SYSTEM TABLE  <NA>
3 <NA>           Sheet3$ SYSTEM TABLE  <NA>
4 <NA>   Sheet1$Print_Area     TABLE  <NA>
5 <NA>   Sheet1$Print_Titles    TABLE  <NA>

> qry = paste('select * from ', '[', tbls$TABLE_NAME[1], ']', sep=' ')
> postdocs = sqlQuery(con, qry, stringsAsFactors=FALSE)
> head(postdocs)
      F1 The Top 200 Institutions--Postdoctoral Appointees_(2005)   F3 F4 F5 F6
1 NA                                         <NA>   NA NA NA NA
2 NA Top 50 Institutions\\nin Postdoctoral Appointees \\n(2005)   NA NA NA NA
3 NA                                         Harvard University 4384  1  1 NA
4 NA                                         Johns Hopkins University 1442  2  2 NA
5 NA                                         Stanford University 1259  3  3 NA
6 NA                                         University of California - Los Angeles 1094  4  1 NA
      F7
1 <NA>
2 Institutional\\nControl
3 Private
4 Private
5 Private
6 Public
> sapply(postdocs, class)
      F1
      "logical"
The Top 200 Institutions--Postdoctoral Appointees_(2005)
      "character"
```

```
          F3
"numeric"
          F4
"numeric"
          F5
"numeric"
          F6
"logical"
          F7
"character"

> dim(postdocs)
[1] 208    7
```

The RODBC interface was able to correctly classify the numeric columns.

# Chapter 7

## Cluster Analysis

## 7.1 Introduction to Cluster Analysis

While we often think of statistics as giving definitive answers to well-posed questions, there are some statistical techniques that are used simply to gain further insight into a group of observations. One such technique (which encompasses lots of different methods) is cluster analysis. The idea of cluster analysis is that we have a set of observations, on which we have available several measurements. Using these measurements, we want to find out if the observations naturally group together in some predictable way. For example, we may have recorded physical measurements on many animals, and we want to know if there's a natural grouping (based, perhaps on species) that distinguishes the animals from another. (This use of cluster analysis is sometimes called "numerical taxonomy"). As another example, suppose we have information on the demographics and buying habits of many consumers. We could use cluster analysis on the data to see if there are distinct groups of consumers with similar demographics and buying habits (market segmentation).

It's important to remember that cluster analysis isn't about finding the right answer – it's about finding ways to look at data that allow us to understand the data better. For example, suppose we have a deck of playing cards, and we want to see if they form some natural groupings. One person may separate the black cards from the red; another may break the cards up into hearts, clubs, diamonds and spades; a third person might separate cards with pictures from cards with no pictures, and a fourth might make one pile of aces, one of twos, and so on. Each person is right in their own way, but in cluster analysis, there's really not a single "correct" answer.

Another aspect of cluster analysis is that there are an enormous number of possible ways of dividing a set of observations into groups. Even if we specify the number of groups, the number of possibilities is still enormous. For example, consider the task of dividing 25 observations into 5 groups. (25 observations is considered very small in the world of cluster analysis). It turns out there are  $2.4 * 10^{15}$  different ways to arrange those observations into 5 groups. If, as is often the case, we don't know the number of groups ahead of time, and we need to consider all possible numbers of groups (from 1 to 25), the number is more than  $4 * 10^{18}$ ! So any technique that simply tries all the different possibilities is doomed to failure.

## 7.2 Standardization

There are two very important decisions that need to be made whenever you are carrying out a cluster analysis. The first regards the relative scales of the variables being measured. We'll see that the available cluster analysis algorithms all depend on the concept of measuring the distance (or some other measure of similarity) between the different observations we're trying to cluster. If one of the variables is measured on a much larger scale than the other variables, then whatever measure we use will be overly influenced by that variable. For example, recall the `world` data set that we used earlier in the semester. Here's a quick summary of the mean values of the variables in that data set:

```
> apply(world1[-c(1,6)], 2, mean, na.rm=TRUE)
```

gdp	income	literacy	military
9.053595e+03	1.025796e+04	8.094902e+01	5.679061e+09

Without some sort of standardization, a variable like literacy, measured on a scale of 0 to 100 has no chance of influencing our solution when the other variables are so much larger.

The traditional way of standardizing variables is to subtract their mean, and divide by their standard deviation. Variables standardized this way are sometimes referred to as z-scores, and always have a mean of zero and variance of one. In the case of variables that contain outliers (observations that are much bigger or smaller than the vast majority of the data), this sort of standardization may be too severe, scaling down the outlying observations so that they appear to be closer to the others. One alternative is to use the median absolute deviation in place of the standard deviation; another possibility is to subtract the median and divide by either the interquartile range or the median absolute deviation. For the common methods of measuring distances (discussed below), centering the data by subtracting the mean or median is not really critical; it's the division by an appropriate scaling factor that's important.

## 7.3 Distance Measures

The most common distance measure, and the default for most programs that perform cluster analysis is the Euclidean distance, which is an extension of the usual notion of the distance between two points in a plane. The Euclidean distance between two observations is calculated as the square root of the sum of the squares of the distances between corresponding variables in the two observations being considered. Another widely used measure is the Manhattan distance, so named because it is similar to the distance between two points in a city, where you can only travel along a grid of streets. It's calculated by adding up the absolute value of the differences of the corresponding variables, and is less likely to be influenced by a very large difference between just one of the variables. The Canberra distance is interesting in that it performs its own standardization; absolute values of differences are divided by the absolute value of the sum of the corresponding variables in the two observations. Depending on the values and distributions of the variables in the data set being clustered, these different distance measures may point out different aspects of the structure of the data set.

Special consideration needs to be given to binary variables, that is, variables that take on only one of two values like TRUE or FALSE, especially when they are used in conjunction with continuous variables. Generally there are two types of measures that are used with binary data. Symmetric measures view two observations as being close together if the binary feature is either absent in both or present in both, while asymmetric measures only view the observations as being close if the feature is present for both.

For some clustering methods, the entire distance matrix must be calculated; for other methods, distances are only calculated as needed.

## 7.4 Clustering Techniques

Much of the history of cluster analysis is concerned with developing algorithms that were not too computer intensive, since early computers were not nearly as powerful as they are today. Accordingly, computational shortcuts have traditionally been used in many cluster analysis algorithms. These algorithms have proven to be very useful, and can be found in most computer software.

More recently, many of these older methods have been revisited and updated to reflect the fact that certain computations that once would have overwhelmed the available computers can now be performed routinely. In R, a number of these updated versions of cluster analysis algorithms are available through the `cluster` library, providing us with a large selection of methods to perform cluster analysis, and the possibility of comparing the old methods with the new to see if they really provide an advantage.

One of the oldest methods of cluster analysis is known as k-means cluster analysis, and is available in R through the `kmeans` function. The first step (and certainly not a trivial one) when using k-means cluster analysis is to specify the number of clusters ( $k$ ) that will be formed in the final solution. The process begins by choosing  $k$  observations to serve as centers for the clusters. Then, the distance from each of the other observations is calculated for each of the  $k$  clusters, and observations are put in the cluster to which they are the closest. After each observation has been put in a cluster, the center of the clusters is recalculated, and every observation is checked to see if it might be closer to a different cluster, now that the centers have been recalculated. The process continues until no observations switch clusters.

Looking on the good side, the k-means technique is fast, and doesn't require calculating all of the distances between each observation and every other observation. It can be written to efficiently deal with very large data sets, so it may be useful in cases where other methods fail. On the down side, if you rearrange your data, it's very possible that you'll get a different solution every time you change the ordering of your data. Another criticism of this technique is that you may try, for example, a 3 cluster solution that seems to work pretty well, but when you look for the 4 cluster solution, all of the structure that the 3 cluster solution revealed is gone. This makes the procedure somewhat unattractive if you don't know exactly how many clusters you should have in the first place.

The R `cluster` library provides a modern alternative to k-means clustering, known as `pam`, which is an acronym for "Partitioning around Medoids". The term medoid refers to an observation within a cluster for which the sum of the distances between it and all the other members of the cluster is a minimum. `pam` requires that you know the number of clusters that you want (like k-means clustering), but it does more computation than k-means in order to insure that the medoids it finds are truly representative of the observations within a given cluster. Recall that in the k-means method the centers of the clusters (which might or might not actually correspond to a particular observation) are only recalculated after all of the observations have had a chance to move from one cluster to another. With `pam`, the sums of the distances between objects within a cluster are constantly recalculated as observations move around, which will hopefully provide a more reliable solution. Furthermore, as a by-product of the clustering operation it identifies the observations that represent the medoids,

and these observations (one per cluster) can be considered a representative example of the members of that cluster which may be useful in some situations. `pam` does require that the entire distance matrix is calculated to facilitate the recalculation of the medoids, and it does involve considerably more computation than k-means, but with modern computers this may not be an important consideration. As with k-means, there's no guarantee that the structure that's revealed with a small number of clusters will be retained when you increase the number of clusters.

Another class of clustering methods, known as hierarchical agglomerative clustering methods, starts out by putting each observation into its own separate cluster. It then examines all the distances between all the observations and pairs together the two closest ones to form a new cluster. This is a simple operation, since hierarchical methods require a distance matrix, and it represents exactly what we want – the distances between individual observations. So finding the first cluster to form simply means looking for the smallest number in the distance matrix and joining the two observations that the distance corresponds to into a new cluster. Now there is one less cluster than there are observations. To determine which observations will form the next cluster, we need to come up with a method for finding the distance between an existing cluster and individual observations, since once a cluster has been formed, we'll determine which observation will join it based on the distance between the cluster and the observation. Some of the methods that have been proposed to do this are to take the minimum distance between an observation and any member of the cluster, to take the maximum distance, to take the average distance, or to use some kind of measure that minimizes the distances between observations within the cluster. Each of these methods will reveal certain types of structure within the data. Using the minimum tends to find clusters that are drawn out and “snake”-like, while using the maximum tends to find compact clusters. Using the mean is a compromise between those methods. One method that tends to produce clusters of more equal size is known as Ward's method. It attempts to form clusters keeping the distances within the clusters as small as possible, and is often useful when the other methods find clusters with only a few observations. Agglomerative Hierarchical cluster analysis is provided in R through the `hclust` function.

Notice that, by its very nature, solutions with many clusters are nested within the solutions that have fewer clusters, so observations don't “jump ship” as they do in k-means or the `pam` methods. Furthermore, we don't need to tell these procedures how many clusters we want – we get a complete set of solutions starting from the trivial case of each observation in a separate cluster all the way to the other trivial case where we say all the observations are in a single cluster.

Traditionally, hierarchical cluster analysis has taken computational shortcuts when updating the distance matrix to reflect new clusters. In particular, when a new cluster is formed and the distance matrix is updated, all the information about the individual members of the cluster is discarded in order to make the computations faster. The `cluster` library provides the `agnes` function which uses essentially the same technique as `hclust`, but which uses fewer shortcuts when updating the distance matrix. For example, when the mean method of calculating the distance between observations and clusters is used, `hclust` only uses the

two observations and/or clusters which were recently merged when updating the distance matrix, while `agnes` calculates those distances as the average of all the distances between all the observations in the two clusters. While the two techniques will usually agree quite closely when minimum or maximum updating methods are used, there may be noticeable differences when updating using the average distance or Ward's method.

## 7.5 Hierarchical Clustering

For the hierarchical clustering methods, the dendrogram is the main graphical tool for getting insight into a cluster solution. When you use `hclust` or `agnes` to perform a cluster analysis, you can see the dendrogram by passing the result of the clustering to the `plot` function.

To illustrate interpretation of the dendrogram, we'll look at a cluster analysis performed on a set of cars from 1978-1979; the data can be found at <http://www.stat.berkeley.edu/classes/s133/data/cars.tab>. Since the data is a tab-delimited file, we use `read.delim`:

```
> cars = read.delim('cars.tab', stringsAsFactors=FALSE)
```

To get an idea of what information we have, let's look at the first few records;

```
> head(cars)
   Country          Car   MPG Weight Drive_Ratio Horsepower
1   U.S.      Buick Estate Wagon 16.9  4.360      2.73       155
2   U.S.    Ford Country Squire Wagon 15.5  4.054      2.26       142
3   U.S.      Chevy Malibu Wagon 19.2  3.605      2.56       125
4   U.S.    Chrysler LeBaron Wagon 18.5  3.940      2.45       150
5   U.S.        Chevette 30.0  2.155      3.70        68
6 Japan      Toyota Corona 27.5  2.560      3.05        95
Displacement Cylinders
1           350      8
2           351      8
3           267      8
4           360      8
5            98      4
6          134      4
```

It looks like the variables are measured on different scales, so we will likely want to standardize the data before proceeding. The `daisy` function in the `cluster` library will automatically perform standardization, but it doesn't give you complete control. If you have a particular method of standardization in mind, you can use the `scale` function. You pass `scale` a matrix or data frame to be standardized, and two optional vectors. The first, called `center`, is a vector of values, one for each column of the matrix or data frame to be standardized, which will be subtracted from every entry in that column. The second, called `scale`, is

similar to `center`, but is used to divide the values in each column. Thus, to get z-scores, you could pass `scale` a vector of means for `center`, and a vector of standard deviations for `scale`. These vectors can be created with the `apply` function, that performs the same operation on each row or column of a matrix. Suppose we want to standardize by subtracting the median and dividing by the mean average deviation:

```
> cars.use = cars[,-c(1,2)]
> medians = apply(cars.use,2,median)
> mads = apply(cars.use,2,mad)
> cars.use = scale(cars.use,center=medians,scale=mads)
```

(The 2 used as the second argument to `apply` means to apply the function to the columns of the matrix or data frame; a value of 1 means to use the rows.) The country of origin and name of the car will not be useful in the cluster analysis, so they have been removed. Notice that the `scale` function doesn't change the order of the rows of the data frame, so it will be easy to identify observations using the omitted columns from the original data.

First, we'll take a look at a hierarchical method, since it will provide information about solutions with different numbers of clusters. The first step is calculating a distance matrix. For a data set with `n` observations, the distance matrix will have `n` rows and `n` columns; the  $(i,j)$ th element of the distance matrix will be the difference between observation `i` and observation `j`. There are two functions that can be used to calculate distance matrices in R; the `dist` function, which is included in every version of R, and the `daisy` function, which is part of the `cluster` library. We'll use the `dist` function in this example, but you should familiarize yourself with the `daisy` function (by reading its help page), since it offers some capabilities that `dist` does not. Each function provides a choice of distance metrics; in this example, we'll use the default of Euclidean distance, but you may find that using other metrics will give different insights into the structure of your data.

```
cars.dist = dist(cars.use)
```

If you display the distance matrix in R (for example, by typing its name), you'll notice that only the lower triangle of the matrix is displayed. This is to remind us that the distance matrix is symmetric, since it doesn't matter which observation we consider first when we calculate a distance. R takes advantage of this fact by only storing the lower triangle of the distance matrix. All of the clustering functions will recognize this and have no problems, but if you try to access the distance matrix in the usual way (for example, with subscripting), you'll see an error message. Thus, if you need to use the distance matrix with anything other than the clustering functions, you'll need to use `as.matrix` to convert it to a regular matrix.

To get started, we'll use the `hclust` method; the `cluster` library provides a similar function, called `agnes` to perform hierarchical cluster analysis.

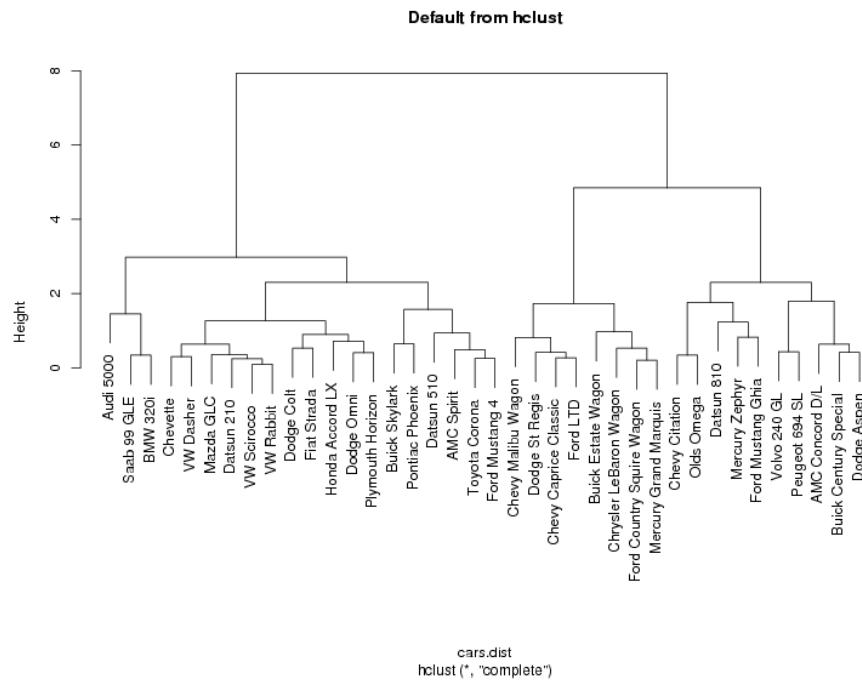
```
> cars.hclust = hclust(cars.dist)
```

Once again, we're using the default method of `hclust`, which is to update the distance matrix using what R calls "complete" linkage. Using this method, when a cluster is formed,

its distance to other objects is computed as the maximum distance between any object in the cluster and the other object. Other linkage methods will provide different solutions, and should not be ignored. For example, using `method=ward` tends to produce clusters of fairly equal size, and can be useful when other methods find clusters that contain just a few observations.

Now that we've got a cluster solution (actually a collection of cluster solutions), how can we examine the results? The main graphical tool for looking at a hierarchical cluster solution is known as a dendrogram. This is a tree-like display that lists the objects which are clustered along the x-axis, and the distance at which the cluster was formed along the y-axis. (Distances along the x-axis are meaningless in a dendrogram; the observations are equally spaced to make the dendrogram easier to read.) To create a dendrogram from a cluster solution, simply pass it to the `plot` function. The result is displayed below.

```
plot(cars.hclust, labels=cars$Car, main='Default from hclust')
```



If you choose any height along the y-axis of the dendrogram, and move across the dendrogram counting the number of lines that you cross, each line represents a group that was identified when objects were joined together into clusters. The observations in that group are represented by the branches of the dendrogram that spread out below the line. For example, if we look at a height of 6, and move across the x-axis at that height, we'll cross two lines. That defines a two-cluster solution; by following the line down through all its branches, we can see the names of the cars that are included in these two clusters. Since the y-axis represents how close together observations were when they were merged into clusters, clusters whose branches are very close together (in terms of the heights at which they were merged)

probably aren't very reliable. But if there's a big difference along the y-axis between the last merged cluster and the currently merged one, that indicates that the clusters formed are probably doing a good job in showing us the structure of the data. Looking at the dendrogram for the car data, there are clearly two very distinct groups; the right hand group seems to consist of two more distinct cluster, while most of the observations in the left hand group are clustering together at about the same height. For this data set, it looks like either two or three groups might be an interesting place to start investigating. This is not to imply that looking at solutions with more clusters would be meaningless, but the data seems to suggest that two or three clusters might be a good start. For a problem of this size, we can see the names of the cars, so we could start interpreting the results immediately from the dendrogram, but when there are larger numbers of observations, this won't be possible.

One of the first things we can look at is how many cars are in each of the groups. We'd like to do this for both the two cluster and three cluster solutions. You can create a vector showing the cluster membership of each observation by using the `cutree` function. Since the object returned by a hierarchical cluster analysis contains information about solutions with different numbers of clusters, we pass the `cutree` function the cluster object and the number of clusters we're interested in. So to get cluster memberships for the three cluster solution, we could use:

```
> groups.3 = cutree(cars.hclust,3)
```

Simply displaying the group memberships isn't that revealing. A good first step is to use the `table` function to see how many observations are in each cluster. We'd like a solution where there aren't too many clusters with just a few observations, because it may make it difficult to interpret our results. For the three cluster solution, the distribution among the clusters looks good:

```
> table(groups.3)
groups.3
 1  2  3
 8 20 10
```

Notice that you can get this information for many different groupings at once by combining the calls to `cutree` and `table` in a call to `sapply`. For example, to see the sizes of the clusters for solutions ranging from 2 to 6 clusters, we could use:

```
> counts = sapply(2:6,function(ncl)table(cutree(cars.hclust,ncl)))
> names(counts) = 2:6
> counts
$"2"
 1  2
18 20

$"3"
```

```
1 2 3  
8 20 10
```

\$"4"

```
1 2 3 4  
8 17 3 10
```

\$"5"

```
1 2 3 4 5  
8 11 6 3 10
```

\$"6"

```
1 2 3 4 5 6  
8 11 6 3 5 5
```

To see which cars are in which clusters, we can use subscripting on the vector of car names to choose just the observations from a particular cluster. Since we used all of the observations in the data set to form the distance matrix, the ordering of the names in the original data will coincide with the values returned by `cutree`. If observations were removed from the data before the distance matrix is computed, it's important to remember to make the same deletions in the vector from the original data set that will be used to identify observations. So, to see which cars were in the first cluster for the four cluster solution, we can use:

```
> cars$Car [groups.3 == 1]  
[1] Buick Estate Wagon      Ford Country Squire Wagon  
[3] Chevy Malibu Wagon     Chrysler LeBaron Wagon  
[5] Chevy Caprice Classic   Ford LTD  
[7] Mercury Grand Marquis  Dodge St Regis
```

As usual, if we want to do the same thing for all the groups at once, we can use `sapply`:

```
> sapply(unique(groups.3), function(g) cars$Car [groups.3 == g])  
[[1]]  
[1] Buick Estate Wagon      Ford Country Squire Wagon  
[3] Chevy Malibu Wagon     Chrysler LeBaron Wagon  
[5] Chevy Caprice Classic   Ford LTD  
[7] Mercury Grand Marquis  Dodge St Regis
```

```

[[2]]
[1] Chevette          Toyota Corona    Datsun 510      Dodge Omni
[5] Audi 5000         Saab 99 GLE     Ford Mustang 4  Mazda GLC
[9] Dodge Colt        AMC Spirit       VW Scirocco   Honda Accord LX
[13] Buick Skylark   Pontiac Phoenix  Plymouth Horizon Datsun 210
[17] Fiat Strada     VW Dasher       BMW 320i      VW Rabbit

```

```

[[3]]
[1] Volvo 240 GL      Peugeot 694 SL    Buick Century Special
[4] Mercury Zephyr    Dodge Aspen      AMC Concord D/L
[7] Ford Mustang Ghia Chevy Citation   Olds Omega
[10] Datsun 810

```

We could also see what happens when we use the four cluster solution

```

> groups.4 = cutree(cars.hclust,4)
> sapply(unique(groups.4),function(g)cars$Car[groups.4 == g])
[[1]]
[1] Buick Estate Wagon      Ford Country Squire Wagon
[3] Chevy Malibu Wagon     Chrysler LeBaron Wagon
[5] Chevy Caprice Classic  Ford LTD
[7] Mercury Grand Marquis Dodge St Regis

[[2]]
[1] Chevette          Toyota Corona    Datsun 510      Dodge Omni
[5] Ford Mustang 4    Mazda GLC       Dodge Colt     AMC Spirit
[9] VW Scirocco       Honda Accord LX Buick Skylark  Pontiac Phoenix
[13] Plymouth Horizon Datsun 210    Fiat Strada    VW Dasher
[17] VW Rabbit

[[3]]
[1] Audi 5000   Saab 99 GLE BMW 320i

[[4]]
[1] Volvo 240 GL      Peugeot 694 SL    Buick Century Special
[4] Mercury Zephyr    Dodge Aspen      AMC Concord D/L
[7] Ford Mustang Ghia Chevy Citation   Olds Omega
[10] Datsun 810

```

The new cluster can be recognized as the third group in the above output.

Often there is an auxiliary variable in the original data set that was not included in the cluster analysis, but may be of interest. In fact, cluster analysis is sometimes performed

to see if observations naturally group themselves in accord with some already measured variable. For this data set, we could ask whether the clusters reflect the country of origin of the cars, stored in the variable `Country` in the original data set. The `table` function can be used, this time passing two arguments, to produce a cross-tabulation of cluster group membership and country of origin:

```
> table(groups.3,cars$Country)
```

groups.3	France	Germany	Italy	Japan	Sweden	U.S.
1	0	0	0	0	0	8
2	0	5	1	6	1	7
3	1	0	0	1	1	7

>

Of interest is the fact that all of the cars in cluster 1 were manufactured in the US. Considering the state of the automobile industry in 1978, and the cars that were identified in cluster 1, this is not surprising.

In an example like this, with a small number of observations, we can often interpret the cluster solution directly by looking at the labels of the observations that are in each cluster. Of course, for larger data sets, this will be impossible or meaningless. A very useful method for characterizing clusters is to look at some sort of summary statistic, like the median, of the variables that were used to perform the cluster analysis, broken down by the groups that the cluster analysis identified. The `aggregate` function is well suited for this task, since it will perform summaries on many variables simultaneously. Let's look at the median values for the variables we've used in the cluster analysis, broken up by the cluster groups. One oddity of the `aggregate` function is that it demands that the variable(s) used to divide up the data are passed to it in a list, even if there's only one variable:

```
> aggregate(cars.use,list(groups.3),median)
  Group.1      MPG     Weight Drive_Ratio Horsepower Displacement Cylinders
1       1 -0.7945273  1.5051136 -0.9133729  1.0476133    2.4775849  4.7214353
2       2  0.6859228 -0.5870568   0.5269459 -0.6027364   -0.5809970 -0.6744908
3       3 -0.4058377  0.5246039 -0.1686227   0.3587717    0.3272282  2.0234723
```

If the ranges of these numbers seem strange, it's because we standardized the data before performing the cluster analysis. While it is usually more meaningful to look at the variables in their original scales, when data is centered, negative values mean "lower than most" and positive values mean "higher than most". Thus, group 1 is cars with relatively low MPG, high weight, low drive ratios, high horsepower and displacement, and more than average number of cylinders. Group 2 are cars with high gas mileage, and low weight and horsepower; and group 3 is similar to group 1. It may be easier to understand the groupings if we look at the variables in their original scales:

```

> aggregate(cars[,-c(1,2)],list(groups.3),median)
  Group.1   MPG Weight Drive_Ratio Horsepower Displacement Cylinders
1       1 17.30  3.890      2.430     136.5        334          8
2       2 30.25  2.215      3.455      79.0        105          4
3       3 20.70  3.105      2.960     112.5        173          6

```

It may also be useful to add the numbers of observations in each group to the above display. Since `aggregate` returns a data frame, we can manipulate it in any way we want:

```

> a3 = aggregate(cars[,-c(1,2)],list(groups.3),median)
> data.frame(Cluster=a3[,1],Freq=as.vector(table(groups.3)),a3[,-1])
  Cluster Freq   MPG Weight Drive_Ratio Horsepower Displacement Cylinders
1       1    8 17.30  3.890      2.430     136.5        334          8
2       2   20 30.25  2.215      3.455      79.0        105          4
3       3   10 20.70  3.105      2.960     112.5        173          6

```

To see how the four cluster solution differed from the three cluster solution, we can perform the same analysis for that solution:

```

> a4 = aggregate(cars[,-c(1,2)],list(groups.4),median)
> data.frame(Cluster=a4[,1],Freq=as.vector(table(groups.4)),a4[,-1])
  Cluster Freq   MPG Weight Drive_Ratio Horsepower Displacement Cylinders
1       1    8 17.3  3.890      2.43     136.5        334          8
2       2   17 30.9  2.190      3.37     75.0         98          4
3       3    3 21.5  2.795      3.77    110.0        121          4
4       4   10 20.7  3.105      2.96    112.5        173          6

```

The main difference seems to be that the four cluster solution recognized a group of cars that have higher horsepower and drive ratios than the other cars in the cluster they came from.

## 7.6 PAM: Partitioning Around Medoids

Unlike the hierarchical clustering methods, techniques like k-means cluster analysis (available through the `kmeans` function) or partitioning around mediods (available through the `pam` function in the `cluster` library) require that we specify the number of clusters that will be formed in advance. `pam` offers some additional diagnostic information about a clustering solution, and provides a nice example of an alternative technique to hierarchical clustering. To use `pam`, you must first load the `cluster` library. You can pass `pam` a data frame or a distance matrix; since we've already formed the distance matrix, we'll use that. `pam` also needs the number of clusters you wish to form. Let's look at the three cluster solution produced by `pam`:

```
> library(cluster)
> cars.pam = pam(cars.dist,3)
```

First of all, let's see if the `pam` solution agrees with the `hclust` solution. Since `pam` only looks at one cluster solution at a time, we don't need to use the `cutree` function as we did with `hclust`; the cluster memberships are stored in the `clustering` component of the `pam` object; like most R objects, you can use the `names` function to see what else is available. Further information can be found in the help page for `pam.object`.

```
> names(cars.pam)
[1] "medoids"      "id.med"       "clustering"   "objective"   "isolation"
[6] "clusinfo"     "silinfo"     "diss"         "call"
```

We can use `table` to compare the results of the `hclust` and `pam` solutions:

```
> table(groups.3,cars.pam$clustering)
groups.3  1  2  3
      1  8  0  0
      2  0 19  1
      3  0  0 10
```

The solutions seem to agree, except for 1 observations that `hclust` put in group 2 and `pam` put in group 3. Which observations was it?

```
> cars$Car[groups.3 != cars.pam$clustering]
[1] Audi 5000
```

Notice how easy it is to get information like this due to the power of R's subscripting operations.

One novel feature of `pam` is that it finds observations from the original data that are typical of each cluster in the sense that they are closest to the center of the cluster. The indexes of the medoids are stored in the `id.med` component of the `pam` object, so we can use that component as a subscript into the vector of car names to see which ones were selected:

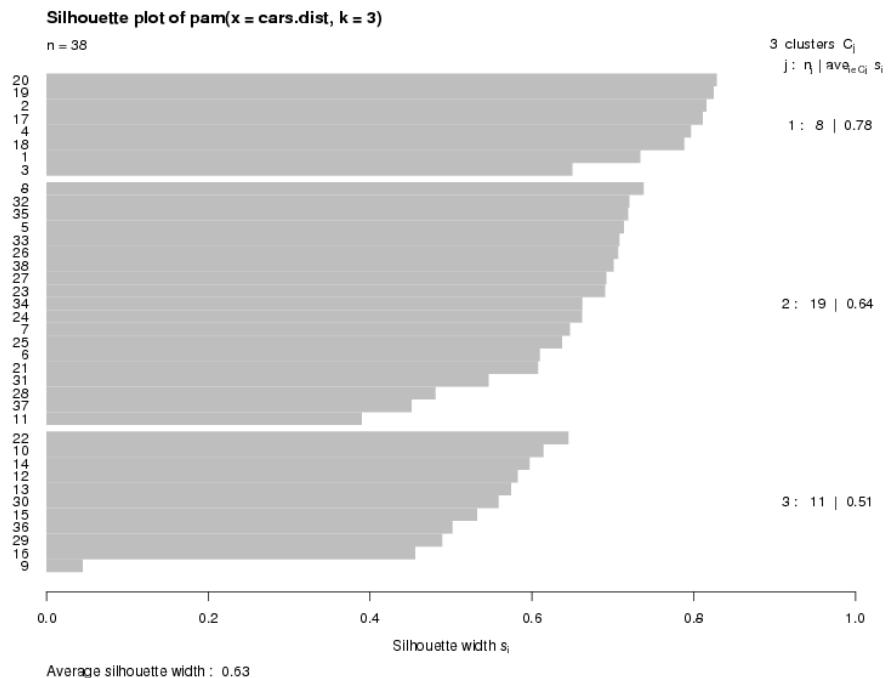
```
> cars$Car[cars.pam$id.med]
> cars$Car[cars.pam$id.med]
[1] Dodge St Regis      Dodge Omni        Ford Mustang Ghia
```

Another feature available with `pam` is a plot known as a silhouette plot. First, a measure is calculated for each observation to see how well it fits into the cluster that it's been assigned to. This is done by comparing how close the object is to other objects in its own cluster with how close it is to objects in other clusters. (A complete description can be found in the help page for `silhouette`.) Values near one mean that the observation is well placed in its cluster; values near 0 mean that it's likely that an observation might really belong in some other cluster. Within each cluster, the value for this measure is displayed from smallest to largest. If the silhouette plot shows values close to one for each observation, the fit was

Range of SC	Interpretation
0.71-1.0	A strong structure has been found
0.51-0.70	A reasonable structure has been found
0.26-0.50	The structure is weak and could be artificial
<0.25	No substantial structure has been found

good; if there are many observations closer to zero, it's an indication that the fit was not good. The silhouette plot is very useful in locating groups in a cluster analysis that may not be doing a good job; in turn this information can be used to help select the proper number of clusters. For the current example, here's the silhouette plot for the three cluster `pam` solution, produced by the command

```
> plot(cars.pam)
```



The plot indicates that there is a good structure to the clusters, with most observations seeming to belong to the cluster that they're in. There is a summary measure at the bottom of the plot labeled "Average Silhouette Width". This table shows how to use the value:

To create a silhouette plot for a particular solution derived from a hierarchical cluster analysis, the `silhouette` function can be used. This function takes the appropriate output from `cutree` along with the distance matrix used for the clustering. So to produce a silhouette plot for our 4 group hierarchical cluster (not shown), we could use the following statements:

```
plot(silhouette(cutree(cars.hclust,4),cars.dist))
```

## 7.7 AGNES: Agglomerative Nesting

As an example of using the `agnes` function from the `cluster` package, consider the famous Fisher iris data, available as the dataframe `iris` in R. First let's look at some of the data:

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2   setosa
2          4.9         3.0          1.4         0.2   setosa
3          4.7         3.2          1.3         0.2   setosa
4          4.6         3.1          1.5         0.2   setosa
5          5.0         3.6          1.4         0.2   setosa
6          5.4         3.9          1.7         0.4   setosa
```

We will only consider the numeric variables in the cluster analysis. As mentioned previously, there are two functions to compute the distance matrix: `dist` and `daisy`. It should be mentioned that for data that's all numeric, using the function's defaults, the two methods will give the same answers. We can demonstrate this as follows:

```
> iris.use = subset(iris,select=-Species)
> d = dist(iris.use)
> library(cluster)
> d1 = daisy(iris.use)
> sum(abs(d - d1))
[1] 1.072170e-12
```

Of course, if we choose a non-default metric for `dist`, the answers will be different:

```
> dd = dist(iris.use,method='manhattan')
> sum(abs(as.matrix(dd) - as.matrix(d1)))
[1] 38773.86
```

The values are very different!

Continuing with the cluster example, we can calculate the cluster solution as follows:

```
> z = agnes(d)
```

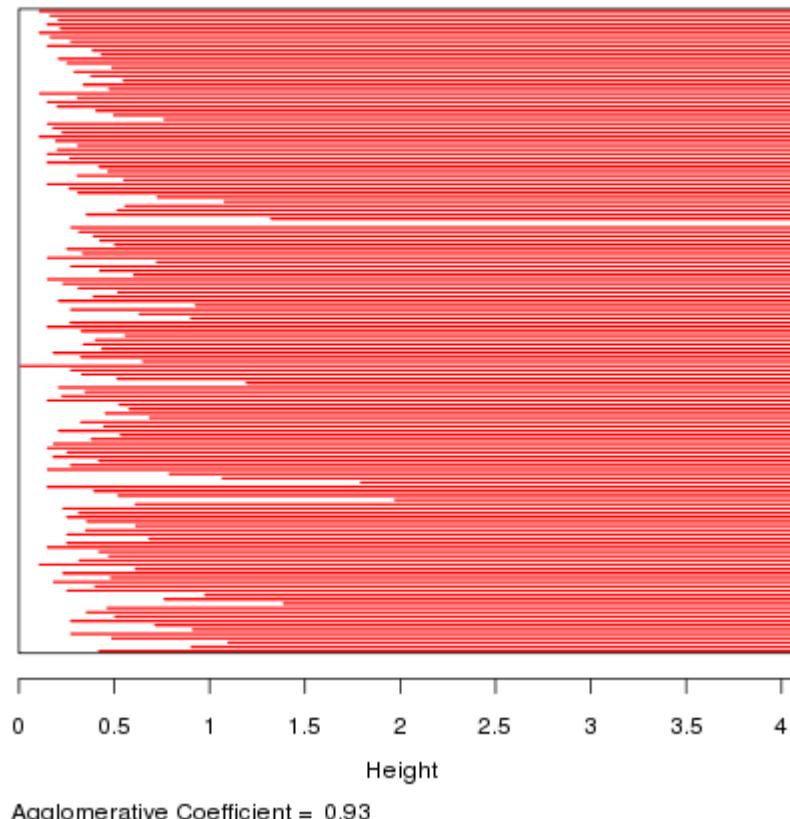
The plotting method for `agnes` objects presents two different views of the cluster solution. When we plot such an object, the plotting function sets the graphics parameter `ask=TRUE`, and the following appears in your R session each time a plot is to be drawn:

Hit <Return> to see next plot:

If you know you want a particular plot, you can pass the `which.plots=` argument an integer telling which plot you want.

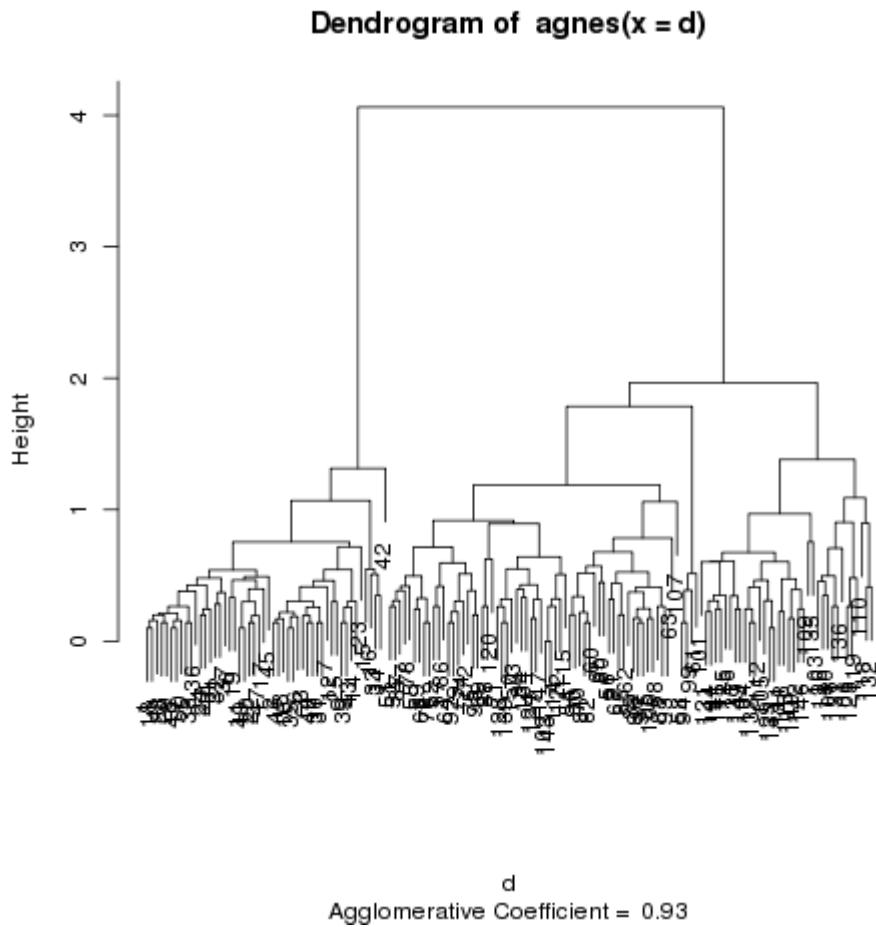
The first plot that is displayed is known as a banner plot. The banner plot for the iris data is shown below:

**Banner of agnes(x = d)**



The white area on the left of the banner plot represents the unclustered data while the white lines that stick into the red are show the heights at which the clusters were formed. Since we don't want to include too many clusters that joined together at similar heights, it looks like three clusters, at a height of about 2 is a good solution. It's clear from the banner plot that if we lowered the height to, say 1.5, we'd create a fourth cluster with only a few observations.

The banner plot is just an alternative to the dendrogram, which is the second plot that's produced from an `agnes` object:



The dendrogram shows the same relationships, and it's a matter of individual preference as to which one is easier to use.

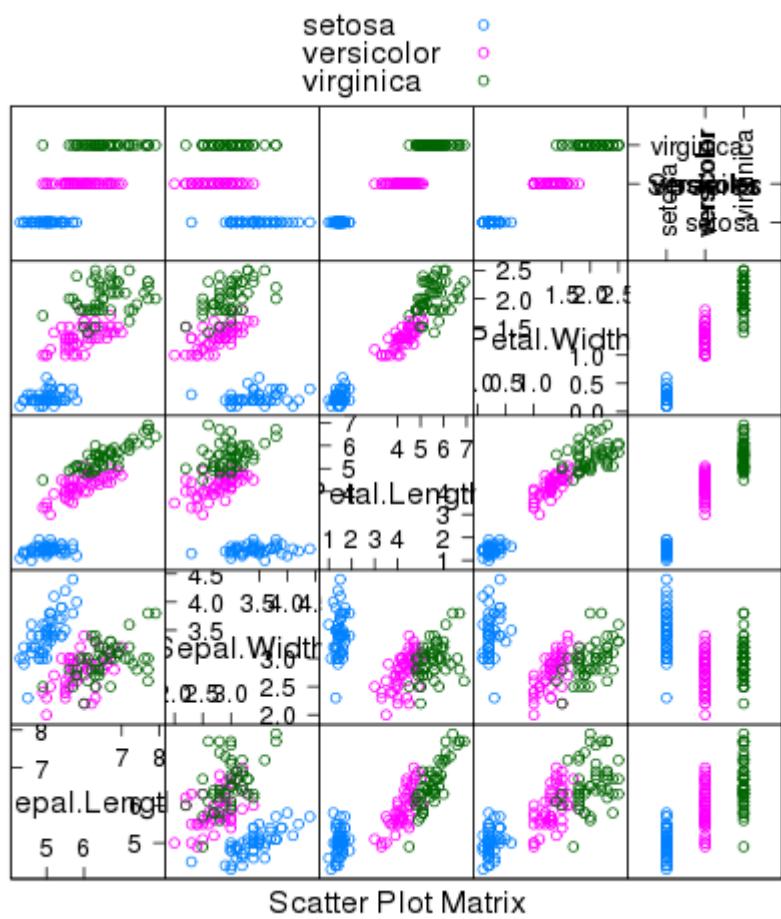
Let's see how well the clusters do in grouping the irises by species:

```
> table(cutree(z,3),iris$Species)
```

	setosa	versicolor	virginica
1	50	0	0
2	0	50	14
3	0	0	36

We were able to classify all the setosa and versicolor varieties correctly. The following plot gives some insight into why we were so successful:

```
> splom(~iris,groups=iris$Species,auto.key=TRUE)
```



# Chapter 8

## XML

## 8.1 What is XML?

XML, which is an acronym for extensible markup language, is a way of creating documents that contain structured information. Up until now, most of the data sets that we've considered were easily accommodated by a observations and variables rectangular data structure (like a comma separated file, which translates neatly to a data frame in R), where the number of variables for each observations is the same. Outside of variable names and the actual data values, there was no additional information stored in the data, either in its text form or once it was read into R.

One type of data that we've seen that accommodates extra information is the spreadsheet; people often annotate their data will all sorts of useful and maybe-not-so-useful information. But as we've seen, there's no standard structure to these non-data rows of the spreadsheet, so they really just get in our way and require customized processing.

Another example of non-regular data would be the baseball database; players played for varying numbers of seasons, and for varying numbers of teams; they may or may not have attended college, or played in a world series, etc. Databases have traditionally been used for data like this, but they certainly have their disadvantages: there are many tables, making it necessary to perform lots of joins, and it's easy to lose track of all that information.

One frustrating aspect regarding designing formats for data is that if you go into the design with a specific type of data in mind, you'll build in features that are wonderful for that data, but may be very lacking for other sorts of data. For example, statisticians would want a rich syntax to describe things like repeated measures, missing values, allowable ranges of variables, etc., but these features would not be of much use to a general user. So in some ways, we can think of XML as a compromise that will allow a wide variety of different information to be stored and distributed in a similar way without favoring any particular structure or type of data. The features that I've described can easily be accommodated in an XML document by simply defining new tags to contain this information. Furthermore, XML is designed so that it's alright if there is no extra information in the document other than the data and its' associated tags. This sort of flexibility is one of the most powerful forces behind the development of XML. To see a list of some of the applications for which XML document styles have been proposed, go to <http://xml.coverpages.org/xml.html#applications>.

At first glance, XML looks very similar to HTML, but there are important differences. First, HTML is used to describe the desired appearance of information in the browser, not the structure of that information. Second, the set of allowable tags in HTML is fixed; we can't just redefine them at our convenience. HTML is also inconsistent. Tags like `h1` and `td` must have accompanying closing tags (`</h1>` and `</td>`), but tags like `<br>` and `<p>` don't. Finally, to accommodate all the people creating web pages and all the browsers trying to display them, HTML doesn't lay down strict rules about the way information is put into a web page. These facts about HTML should not be interpreted as saying the HTML is not useful. It's very useful for what it does, but it doesn't give us a consistent, extensible format for accommodating data that preserves its structure.

Here are some of the things that we'll see in XML documents:

- XML documents always begin with

```
<?xml ....?>
```

- Information is identified by tags (i.e. an identifier surrounded by angle brackets, like

```
<coursename>
    Statistics 133
</coursename>
```

Every opening tag has a corresponding tag, except for the case where there is no information; then a single tag like `<sometag/>` can be used. The specific names of these tags will vary with each XML document.

- Tags are case sensitive, and opening and closing tags must match exactly.
- No spaces are allowed between the opening angle bracket and the tag name
- Tag names must begin with a letter, and can contain only letters and numbers.
- As in HTML, angle brackets will always be interpreted as parts of tags; to get literal angle brackets in data we need to use `&lt;` and `&gt;`.
- Additional information can be stored in the tags in the form of attributes. For example, in the previous example, we could have specified the department as:

```
<coursename department='Statistics'>
    Statistics 133
</coursename>
```

or

```
<course>
    <name>
        Statistics 133
    </name>
    <department>
        Statistics
    </department>
</course>
```

The decision is up to the designer of the particular document.

- Comments can be placed in an XML document by starting with `<!--` and ending with `-->`, and they may appear anywhere in the document.

Documents that conform to these rules are said to be well-formed, and this is the minimum requirement for any XML document. But to be useful, they should also conform to additional rules described in what is known as the Document Type Definitions for a particular document, or the DTD. The DTD describes the allowable tags, what kind of data can be stored within those tags, and what attributes are allowed in the tags. The DTD for a particular document can be provided in the document itself; when a document uses multiple DTDs, the tag name may be prefaced with an identifier followed by a colon to make clear which DTD the tag comes from. The programs that read XML are known as parsers; without a DTD they simply make sure that the document is well-formed; with a DTD they can additionally check to make sure that the data is valid.

## 8.2 A Simple Example

Let's start with a very simple example to introduce the XML library and some of the strategies we'll need to use when working with XML. The example we'll use is a catalog of plants. (You can find the XML document at [http://www.stat.berkeley.edu/classes/s133/data/plant\\_catalog.xml](http://www.stat.berkeley.edu/classes/s133/data/plant_catalog.xml).) Here's what the beginning of the file looks like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>^M
<!-- Edited with XML Spy v2006 (http://www.altova.com) -->^M
<CATALOG>^M
    <PLANT>^M
        <COMMON>Bloodroot</COMMON>^M
        <BOTANICAL>Sanguinaria canadensis</BOTANICAL>^M
        <ZONE>4</ZONE>^M
        <LIGHT>Mostly Shady</LIGHT>^M
        <PRICE>$2.44</PRICE>^M
        <AVAILABILITY>031599</AVAILABILITY>^M
    </PLANT>^M
    <PLANT>^M
        <COMMON>Columbine</COMMON>^M
        <BOTANICAL>Aquilegia canadensis</BOTANICAL>^M
        <ZONE>3</ZONE>^M
        <LIGHT>Mostly Shady</LIGHT>^M
        <PRICE>$9.37</PRICE>^M
        <AVAILABILITY>030699</AVAILABILITY>^M
    </PLANT>^M
    . . .

```

The main body of the data is enclosed by the CATALOG tag; we'll sometimes refer to this as a node. Within the CATALOG node are several PLANT nodes - each of them contains several pieces of data which are the information we wish to extract. We're going to use the R XML library with what is known as the Document Object Model (DOM), in which R will read the

entire XML file into memory, and internally convert it to a tree structure. (There is another model, called SAX (Simple API for XML), which only reads part of the data at a time, but it's more complicated than the DOM model.) All the information about the data and its structure will be stored in a list inside of R, but when we print it, it will look very much like the document that we started with. The basic strategy of working with XML files under this model is to keep “tunneling down” into the document structure without disturbing its structure until we come to the end of the node. For example, with the plant catalog, we want to extract the PLANT nodes as XML structures, and then extract the values of the terminal branches of the tree (like COMMON, BOTANICAL, and ZONE) using the `xmlValue` function. As we explore the structure of the data, we can use the `names` function to see what nodes exist; at any time, we can print a node, and it will show us a representation like the original XML document so we'll know where to go next.

The first step in reading an XML document into R is loading the `XML` library. Next, the `xmlTreeParse` function is called to read the document into memory, and the root of the XML tree is extracted using `xmlRoot`:

```
> library(XML)
> doc = xmlTreeParse('plant_catalog.xml')
> root = xmlRoot(doc)
```

Let's look at some properties of this root node:

```
> class(root)
[1] "XMLNode"           "RXMLAbstractNode" "XMLAbstractNode"   "oldClass"
> names(root)
> names(root)
[1] "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"
[10] "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"
[19] "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"
[28] "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"  "PLANT"
```

In this example, `xmlTreeParse` read its input from a local file. Other choices are local gzipped files and URLs representing XML documents or gzipped XML documents.

The class of `XMLNode` indicates that internally, R is storing the object as part of an XML tree. This will be important later when we decide how to process the entire tree. The result of the `names` function call tells us that the structure of this document is quite simple. Inside the root node, there are 36 `PLANT` nodes. (For larger documents, it might be prudent to make a table of the names instead of displaying the names directly.) As we've already seen from looking at the document, this is where our data lies, so we'll want to examine one of the plant nodes a little more carefully to figure out how to extract the actual data.

```
> oneplant = root[[1]]
> class(oneplant)
[1] "XMLNode"           "RXMLAbstractNode" "XMLAbstractNode"   "oldClass"
> oneplant
```

```

<PLANT>
  <COMMON>
    Bloodroot
  </COMMON>
  <BOTANICAL>
    Sanguinaria canadensis
  </BOTANICAL>
  <ZONE>
    4
  </ZONE>
  <LIGHT>
    Mostly Shady
  </LIGHT>
  <PRICE>
    $2.44
  </PRICE>
  <AVAILABILITY>
    031599
  </AVAILABILITY>
</PLANT>

```

We can see that this single PLANT object is still an `XMLnode`; its printed representation shows us exactly what we've got. Notice that the individual elements don't have any further tree structure; this means we can use `xmlValue` to extract the values:

```

> xmlValue(oneplant[['COMMON']])
[1] "Bloodroot"
> xmlValue(oneplant[['BOTANICAL']])
[1] "Sanguinaria canadensis"

```

Of course, we don't want to perform these tasks one by one for each node of the tree. You may recall that when we were confronted with problems like this before, we used the `sapply` function, which operates on every element of a list. If the object we want to process is an `xmlNode`, the corresponding function to use is `xmlSApply`. For example, to get all the common names of all the plants, we can use `xmlValue` on all the PLANT nodes like this:

```

> commons = xmlSApply(root,function(x)xmlValue(x[['COMMON']]))

> head(commons)
      PLANT          PLANT          PLANT
    "Bloodroot"     "Columbine"   "Marsh Marigold"
      PLANT          PLANT          PLANT
    "Cowslip"   "Dutchman's-Breeches"   "Ginger, Wild"

```

We could repeat the process for each column manually, and then combine things into a data frame, or we can automate the process using `lapply` and the names of the objects within the plants:

```

> getvar = function(x,var)xmlValue(x[[var]])
> res = lapply(names(root[[1]]),function(var)xmlSApply(root,getvar,var))
> plants = data.frame(res)
> names(plants) = names(root[[1]])
> head(plants)

      COMMON           BOTANICAL ZONE    LIGHT PRICE
1   Bloodroot  Sanguinaria canadensis     4 Mostly Shady $2.44
2   Columbine   Aquilegia canadensis     3 Mostly Shady $9.37
3 Marsh Marigold   Caltha palustris     4 Mostly Sunny $6.81
4   Cowslip       Caltha palustris     4 Mostly Shady $9.90
5 Dutchman's-Breeches Dicentra cucullaria     3 Mostly Shady $6.44
6   Ginger, Wild   Asarum canadense     3 Mostly Shady $9.03

  AVAILABILITY
1      031599
2      030699
3      051799
4      030699
5      012099
6      041899

```

## 8.3 More Complex Example

For more complex documents, a few other tools are useful. To illustrate, we'll look at a file that uses Geographic Markup Language, or GML. This file (which you can find at <http://www.stat.berkeley.edu/classes/s133/data/counties.gml>) contains the x- and y-coordinates of the county centers for each state in the United States. Information like this would be difficult to store in a less structured environment, because each state has a different number of counties. If we were going to read it into a database, we might want to have a separate table for each state; in some programs, we'd have to force it into a rectangular form, even though it wouldn't be that efficient. If we were using a spreadsheet, we might want to put all the information in a single spreadsheet, with a separate heading for each state. In R, a reasonable way to store the data would be in a list, with a separate data frame for each state. Thus, providing the data in a form that could be easily converted to any of those formats is essential, and that's just what XML does for us.

The first steps are the same as the previous example; loading the library, parsing the tree, and extracting the root.

```

> doc = xmlTreeParse('counties.gml')
> root = xmlRoot(doc)

```

To see what's in the root, we can get a table of the names found there:

```
> table(names(root))
```

```
state
51
```

Let's extract the first state node for further study:

```
> onestate = root[[1]]
> class(onestate)
[1] "XMLNode"           "RXMLAbstractNode" "XMLAbstractNode"  "oldClass"
> table(names(onestate))

county    name
 67        1
```

Here's what the onestate object looks like – I've truncated it so that it only displays a single county, but we can still see the general structure:

```
<state>
<gml:name abbreviation="AL">ALABAMA</gml:name>
<county>
  <gml:name>Autauga County</gml:name>
  <gml:location>
    <gml:coord>
      <gml:X>-86641472</gml:X>
      <gml:Y>32542207</gml:Y>
    </gml:coord>
  </gml:location>
</county>
. . .
```

The name element (labeled as gml:name) is just the name of the state. We can extract them from all of the states using `xmlSApply`:

```
> statenames = xmlSApply(root,function(x)xmlValue(x[['name']]))

> head(statenames)
  state      state      state      state      state      state
"ALABAMA"   "ALASKA"   "ARIZONA"  "ARKANSAS" "CALIFORNIA" "COLORADO"
```

Note that in this example there is an attribute in the name tag, namely the state abbreviation. To access these attributes we can use the `xmlAttrs` function in a fashion similar to `xmlValue`:

```
> stateabbs = xmlSApply(root,function(x)xmlAttrs(x[['name']]))

> head(stateabbs)
state.abbreviation state.abbreviation state.abbreviation state.abbreviation
                  "AL"                 "AK"                 "AZ"                 "AR"
state.abbreviation state.abbreviation
                  "CA"                 "CO"
```

Since there was only one attribute, `xmlSApply` was used to extract directly into a vector. If there were multiple attributes, then `xmlApply` would need to be used, since it will return a list of attributes, preserving the structure of the data.

To process the county data further, we need to extract just the `county` nodes. The `xmlElementsByTagName` function will extract just the elements that we want:

```
> counties = xmlElementsByTagName(onestate, 'county')
> class(counties)
[1] "list"
> length(counties)
[1] 67
```

Notice that extracting the elements in this way results in a list, not an `xmlNode`. But what's in the list?

```
> onecounty = counties[[1]]
> class(onecounty)
[1] "XMLNode"           "RXMLAbstractNode" "XMLAbstractNode"   "oldClass"
> names(onecounty)
[1] "name"      "location"
> onecounty
<county>
  <gml:name>
    Autauga County
  </gml:name>
  <gml:location>
    <gml:coord>
      <gml:X>
        -86641472
      </gml:X>
      <gml:Y>
        32542207
      </gml:Y>
    </gml:coord>
  </gml:location>
</county>
```

The elements inside the `counties` list are still `xmlNodes`. The fact that they are contained in a list simply means that we'll use `sapply` or `lapply` to process them instead of `xmlSApply` or `xmlApply`. What we really want are the X and Y values within the `coord` nodes, so let's extract out those nodes from the full list of county nodes:

```
> coords = lapply(counties, function(x)x[['location']] [['coord']])
> class(coords)
[1] "list"
```

```

> class(coords[[1]])
[1] "XMLNode"           "RXMLAbstractNode" "XMLAbstractNode" "oldClass"
> coords[[1]]
<gml:coord>
<gml:X>
-86641472
</gml:X>
<gml:Y>
32542207
</gml:Y>
</gml:coord>

```

Since there is only one `coord` node within the `location` nodes, I extracted it directly. I could also have used `xmlGetElementsByTagName`, and used the first element of the resulting list:

```
coords = lapply(counties, function(x) xmlElementsByTagName(x[['location']], 'coord')[[1]])
```

Notice that I used `lapply` to extract the `coord` nodes. Since `xmlNodes` are represented internally as lists, I would have lost the structure of the data if I used `sapply` in this case.

Now we can extract the x and y values using `xmlValue`:

```

> x = as.numeric(sapply(coords, function(x) xmlValue(x[['X']])))
> y = as.numeric(sapply(coords, function(x) xmlValue(x[['Y']])))

```

That shows the process for extracting the county names and x- and y-coordinates for a single state. Let's summarize the steps in a function, which we can then apply to the root of the tree to get a separate list for each state:

```

onestate = function(state){
  counties = xmlElementsByTagName(state, 'county')
  countynames = sapply(counties, function(x) xmlValue(x[['name']]))

  coords = lapply(counties, function(x)x[['location']][['coord']])
  x = as.numeric(sapply(coords, function(x) xmlValue(x[['X']])))
  y = as.numeric(sapply(coords, function(x) xmlValue(x[['Y']])))
  data.frame(county=countynames, x=x, y=y)
}

```

To combine everything together, and create a list with one data frame per state, we can do the following:

```

> res = xmlApply(root, onestate)
> names(res) = xmlSApply(root, function(x) xmlValue(x[['name']]))


```

Although these examples may seem simplistic, there are usable XML formats that are this simple, or even simpler. For example, at the web site [http://www.weather.gov/data/current\\_obs/](http://www.weather.gov/data/current_obs/) there is a description of an XML format used to distribute weather information. Basically, the XML document is available at [http://www.weather.gov/data/current\\_obs/XXXX.xml](http://www.weather.gov/data/current_obs/XXXX.xml) where XXXX represents the four letter weather station code, usually related to a nearby airport.

A quick look at the data reveals that the format is very simple:

```
> library(XML)
> doc = xmlTreeParse('http://www.weather.gov/data/current_obs/KOAK.xml')
> root = xmlRoot(doc)
> names(root)
[1] "credit"                  "credit_URL"
[3] "image"                   "suggested_pickup"
[5] "suggested_pickup_period" "location"
[7] "station_id"              "latitude"
[9] "longitude"               "observation_time"
[11] "observation_time_rfc822" "weather"
[13] "temperature_string"     "temp_f"
[15] "temp_c"                  "relative_humidity"
[17] "wind_string"             "wind_dir"
[19] "wind_degrees"            "wind_mph"
[21] "wind_gust_mph"           "pressure_string"
[23] "pressure_mb"             "pressure_in"
[25] "dewpoint_string"         "dewpoint_f"
[27] "dewpoint_c"               "heat_index_string"
[29] "heat_index_f"             "heat_index_c"
[31] "windchill_string"        "windchill_f"
[33] "windchill_c"              "visibility_mi"
[35] "icon_url_base"            "icon_url_name"
[37] "two_day_history_url"     "ob_url"
[39] "disclaimer_url"          "copyright_url"
[41] "privacy_policy_url"
```

Essentially, all the information is in the root node, so it can be extracted with `xmlValue`:

```
> xmlValue(root[['temp_f']])
[1] "49.0"
> xmlValue(root[['wind_mph']])
[1] "12.7"
```

To make this easy to use, we can write a function that will allow us to specify a location and some variables that we want information about:

```

getweather = function(loc='KOAK', vars='temp_f'){
  require(XML)
  url = paste('http://www.weather.gov/data/current_obs/', loc, '.xml', sep=' ')
  doc = xmlTreeParse(url)
  root = xmlRoot(doc)
  sapply(vars, function(x) xmlValue(root[[x]]))
}

```

Let's check to make sure it works with the defaults:

```

> getweather()
temp_f
"49.0"

```

That seems to be ok. To make it even more useful, we can create a vector of station names, and use sapply to find weather information for each station:

```

> result = sapply(c('KOAK', 'KACV', 'KSDM'), getweather, var=c('temp_f', 'wind_mph', 'wind_dir'))
> data.frame(t(result))
   temp_f wind_mph wind_dir relative_humidity
KOAK    49.0     12.7      South            93
KACV    45.0      6.9      South            86
KSDM    66.0      6.9  Northwest           42

```

sapply has properly labeled all the stations and variables.

If information was recorded at different times, the `observation_time_rfc822` variable could be converted to an R `POSIXct` object as follows:

```

> as.POSIXct(getweather('KOAK', 'observation_time_rfc822'), format='%a, %d %b %Y %H:%M:%S')
[1] "2011-03-18 14:53:00 PDT"

```

# Chapter 9

## Programming

## 9.1 Operating on Groups of Data

We've already seen how the `aggregate` function let's us calculate data summaries for groups in our data. For example, using the `world` data set, we could find the average value of each variable, broken down by continent:

```
> world = read.csv('http://www.stat.berkeley.edu/classes/s133/data/world2.txt',na.string=)
> names(world)
[1] "country"    "cont"        "gdp"         "income"       "literacy"     "military"    "phys"
[8] "un.date"
> aggregate(world[,-c(1,2,8)],world['cont'],mean,na.rm=TRUE)
  cont      gdp      income literacy      military      phys
1   AF  2723.404  3901.191 60.52979  356440000  26.31129
2   AS  7778.049  8868.098 84.25122  5006536341 161.83381
3   EU 19711.765 21314.324 98.40294  6311138235 311.18482
4   NA  8946.667 10379.143 85.52000  25919931267 187.08178
5   OC 14625.000 15547.500 87.50000  4462475000 127.63105
6   SA  6283.333  6673.083 92.29167  2137341667 154.81088
```

Since the `aggregate` function requires the grouping variables to be in a list, I took advantage of the fact that data frames are also lists, and passed a single element (using single brackets) to `aggregate`. Alternatively, I could have used a list directly:

```
> aggregate(world[,-c(1,2,8)],list(world$cont),mean,na.rm=TRUE)
  Group.1      gdp      income literacy      military      phys
1   AF  2723.404  3901.191 60.52979  356440000  26.31129
2   AS  7778.049  8868.098 84.25122  5006536341 161.83381
3   EU 19711.765 21314.324 98.40294  6311138235 311.18482
4   NA  8946.667 10379.143 85.52000  25919931267 187.08178
5   OC 14625.000 15547.500 87.50000  4462475000 127.63105
6   SA  6283.333  6673.083 92.29167  2137341667 154.81088
```

I would need to name the list element in order for its column in the output to be properly labeled.

Now suppose I'd like to calculate both the mean and the median for each continent:

```
> answer = aggregate(world[,-c(1,2,8)],world['cont'],function(x)c(mean(x,na.rm=TRUE),
+               median(x,na.rm=TRUE)))
> answer
  cont      gdp.1      gdp.2  income.1  income.2 literacy.1 literacy.2
1   AF  2723.404  1400.000  3901.191  1813.000 60.52979  65.50000
2   AS  7778.049  4300.000  8868.098  4561.000 84.25122  89.90000
3   EU 19711.765 18150.000 21314.324 19804.000 98.40294  99.70000
4   NA  8946.667  4900.000          NA          NA 85.52000  87.70000
5   OC 14625.000 13700.000 15547.500 14970.000 87.50000  96.40000
```

```

6   SA  6283.333  5000.000  6673.083  5563.500    92.29167  92.30000
    military.1  military.2    phys.1    phys.2
1   356440000    101300000  26.31129    8.68680
2   5006536341    650000000 161.83381 139.92540
3   6311138235   1288500000 311.18482 317.88580
4   25919931267   147000000 187.08178 164.40130
5   4462475000   591500000 127.63105 128.11915
6   2137341667   742150000 154.81088 126.00035

```

There are eleven columns in the data, but when we look at the dimensions of the result, things seem strange:

```

> dim(answer)
[1] 6 6

```

That's because `aggregate` stores each set of the two values returned from the function we passed to it in matrices inside the `answer` data frame:

```

> sapply(answer, class)
      cont          gdp        income     literacy     military       phys
"character"    "matrix"    "matrix"    "matrix"    "matrix"    "matrix"
> names(answer)
[1] "cont"        "gdp"        "income"      "literacy"    "military"   "phys"
> answer[, 2]
      [,1]     [,2]
[1,] 2723.404 1400
[2,] 7778.049 4300
[3,] 19711.765 18150
[4,] 8946.667 4900
[5,] 14625.000 13700
[6,] 6283.333 5000

```

This actually makes it pretty easy to access the values that we want. For example, suppose we want the mean for the variable `military`. Since the mean was the first element returned by the function we passed to `aggregate`, it's the first column in the `military` matrix:

```

> data.frame(answer$cont, military.mean=answer$military[, 1])
  answer.cont military.mean
1           AF      356440000
2           AS      5006536341
3           EU      6311138235
4           NA      25919931267
5           OC      4462475000
6           SA      2137341667

```

The output from `aggregate` is often easier to use if we give names to the values calculated by the function we pass to `aggregate`. For example, if we call `aggregate` like this:

```
> answer = aggregate(world[,-c(1,2,8)],world['cont'],function(x)c(mean=mean(x,na.rm=TRUE),
+                           median=median(x,na.rm=TRUE)))
> answer
   cont    gdp.mean    gdp.median    income.mean    income.median    literacy.mean
1   AF    2723.404    1400.000    3901.191    1813.000    60.52979
2   AS    7778.049    4300.000    8868.098    4561.000    84.25122
3   EU   19711.765   18150.000   21314.324   19804.000    98.40294
4   NA   8946.667    4900.000    10379.143    6753.000    85.52000
5   OC  14625.000   13700.000   15547.500   14970.000    87.50000
6   SA   6283.333    5000.000    6673.083    5563.500    92.29167
   literacy.median    military.mean    military.median    phys.mean    phys.median
1           65.50000    356440000    101300000    26.31129     8.68680
2           89.90000    5006536341    650000000   161.83381    139.92540
3           99.70000    6311138235   1288500000   311.18482    317.88580
4           87.70000    25919931267   147000000   187.08178    164.40130
5           96.40000    4462475000   591500000   127.63105    128.11915
6           92.30000    2137341667   742150000   154.81088    126.00035
```

Now we can say, for example:

```
> data.frame(cont=answer$cont,literacy.median=answer$literacy[, 'median'])
   cont    literacy.median
1   AF          65.5
2   AS          89.9
3   EU          99.7
4   NA          87.7
5   OC          96.4
6   SA          92.3
```

Another way to manage the output from `aggregate` takes advantage of the fact that when the `data.frame` function is called with multiple data frames, it doesn't store the data frames as separate columns, but rather makes one big data frame with lots of columns. So if we convert each of the matrices to data frames we can combine all of them into one big data frame:

```
> answer1 = data.frame(answer[,1],data.frame(answer[,2]),data.frame(answer[,3]),
+                       data.frame(answer[,4]),data.frame(answer[,5]),data.frame(answer[,6]))
> answer1
   answer...1.      mean median      mean.1 median.1      mean.2 median.2      mean.3
1       AF  2723.404  1400  3901.191  1813.0 60.52979      65.5 356440000
2       AS  7778.049  4300  8868.098  4561.0 84.25122      89.9 5006536341
3       EU 19711.765 18150 21314.324 19804.0 98.40294      99.7 6311138235
```

```

4      NA 8946.667   4900 10379.143   6753.0 85.52000    87.7 25919931267
5      OC 14625.000   13700 15547.500   14970.0 87.50000    96.4 4462475000
6      SA 6283.333    5000 6673.083    5563.5 92.29167    92.3 2137341667
median.3 mean.4 median.4
1 101300000 26.31129   8.6868
2 650000000 161.83381 139.9254
3 1288500000 311.18482 317.8858
4 147000000 187.08178 164.4013
5 591500000 127.63105 128.1191
6 742150000 154.81088 126.0003
> dim(answer1)
[1] 6 11

```

We'd need to rename the columns, but there are now actually eleven columns as we would expect. Of course, we wouldn't want to do all that typing in practice. We would use some of R's powerful tools to automate the process. For example, we could try to use `sapply` to convert all the matrices to data frames

```

> sapply(answer[,-1],data.frame)
      gdp      income      literacy      military      phys
mean Numeric,6 Numeric,6 Numeric,6 Numeric,6 Numeric,6
median Numeric,6 Numeric,6 Numeric,6 Numeric,6 Numeric,6

```

As we've seen before this is usually an indication that `sapply` has simplified things too much. If we try `lapply`, this is what we get:

```

> lapply(answer[,-1],data.frame)
$gdp
      mean median
1 2723.404   1400
2 7778.049   4300
3 19711.765  18150
4 8946.667   4900
5 14625.000  13700
6 6283.333   5000

$income
      mean median
1 3901.191 1813.0
2 8868.098 4561.0
3 21314.324 19804.0
4 10379.143 6753.0
5 15547.500 14970.0
6 6673.083  5563.5

```

We can convert the list to a data frame as follows:

```
> answer1= data.frame(answer[1],lapply(answer[,-1],data.frame))
> answer1
  cont gdp.mean gdp.median income.mean income.median literacy.mean
1 AF 2723.404      1400    3901.191      1813.0    60.52979
2 AS 7778.049      4300    8868.098      4561.0    84.25122
3 EU 19711.765     18150   21314.324      19804.0    98.40294
4 NA 8946.667      4900    10379.143      6753.0    85.52000
5 OC 14625.000     13700   15547.500      14970.0    87.50000
6 SA 6283.333      5000    6673.083      5563.5    92.29167
  literacy.median military.mean military.median phys.mean phys.median
1           65.5    356440000      101300000    26.31129    8.6868
2           89.9    5006536341      650000000    161.83381   139.9254
3          99.7    6311138235      1288500000   311.18482   317.8858
4          87.7    25919931267      147000000    187.08178   164.4013
5          96.4    4462475000      591500000    127.63105   128.1191
6          92.3    2137341667      742150000    154.81088   126.0003
```

As is often the case, R takes care of the names more effectively than if we labouriously combined everything by hand. All that would be left to make this a truly useful display is to add the number of countries in each continent. Since this will be the same for each variable, it's usually not a good idea to return this value in the function you pass to aggregate. Instead, we can use the `table`, `as.data.frame`, and `merge` functions as follows:

```
> counts = as.data.frame(table(world$cont))
> names(counts) = c('cont','N')
> merge(counts,answer1)
  cont N gdp.mean gdp.median income.mean income.median literacy.mean
1 AF 47 2723.404      1400    3901.191      1813.0    60.52979
2 AS 41 7778.049      4300    8868.098      4561.0    84.25122
3 EU 34 19711.765     18150   21314.324      19804.0    98.40294
4 NA 15 8946.667      4900    10379.143      6753.0    85.52000
5 OC  4 14625.000     13700   15547.500      14970.0    87.50000
6 SA 12 6283.333      5000    6673.083      5563.5    92.29167
  literacy.median military.mean military.median phys.mean phys.median
1           65.5    356440000      101300000    26.31129    8.6868
2           89.9    5006536341      650000000    161.83381   139.9254
3          99.7    6311138235      1288500000   311.18482   317.8858
4          87.7    25919931267      147000000    187.08178   164.4013
5          96.4    4462475000      591500000    127.63105   128.1191
6          92.3    2137341667      742150000    154.81088   126.0003
```

Even though `aggregate` can handle cases with more than one value being calculated, it still can only work with one variable at a time. There are two approaches to solve the cases

where you need to work with more than one variable. The first, uses the `split` function that we've seen for making boxplots. Suppose we want to find the two variables with the highest correlation within each continent. First, we can write a function that will find the variables for with the highest correlation in a data frame:

```
hicorr = function(x){
  x = x[,sapply(x,function(col)mode(col) == 'numeric' & class(col) != 'factor')]
  cc = cor(x,use='complete.obs')
  diag(cc) = 0
  wh = which(cc == max(cc),arr.ind=TRUE)
  c(rownames(cc)[wh[1]],colnames(cc)[wh[2]])
}
```

Now we can use `sapply` and `split` to find the variables with the highest correlation in each continent:

```
> as.data.frame(t(sapply(split(world,world$cont),hicorr)))
      V1      V2
AF  phys  military
AS income     gdp
EU income     gdp
NA income     gdp
OC income     gdp
SA income     gdp
```

The other way to solve problems like this is with the `by` function. The `by` function produces nicely formatted output, but its output can be manipulated if necessary. Here's the basic approach for a problem like this.

```
> byvals = by(world,world$cont,hicorr)
> byvals
world$cont: AF
[1] "phys"     "military"
-----
world$cont: AS
[1] "income"   "gdp"
-----
world$cont: EU
[1] "income"   "gdp"
-----
world$cont: NA
[1] "income"   "gdp"
-----
world$cont: OC
[1] "income"   "gdp"
```

```
-----  
world$cont: SA  
[1] "income" "gdp"
```

Since each piece of the `byvals` object looks like part of a data frame we want, we can use `do.call` with `rbind`:

```
> ans = do.call(rbind, byvals)  
> ans  
 [,1]      [,2]  
AF "phys"    "military"  
AS "income"   "gdp"  
EU "income"   "gdp"  
NA "income"   "gdp"  
OC "income"   "gdp"  
SA "income"   "gdp"
```

As another example of `do.call`, consider the task of sorting a data frame by its first column, then using the second column to break any ties, and so on for each of the columns of the data frame. For example:

```
> x = data.frame(matrix(sample(1:5, 100, replace=TRUE), ncol=5))  
> head(x)  
  X1 X2 X3 X4 X5  
1  4  4  1  3  4  
2  2  4  2  4  1  
3  1  1  1  3  2  
4  4  3  3  2  2  
5  5  2  2  4  3  
6  4  2  5  4  5
```

Since a data frame is a list, with each element representing a column, we can order the data frame as follows:

```
> x[do.call(order, x),]  
  X1 X2 X3 X4 X5  
3  1  1  1  3  2  
17 1  2  3  5  5  
10 1  4  5  1  5  
20 2  4  2  1  1  
2  2  4  2  4  1  
11 3  1  4  4  2  
16 3  4  1  1  3  
8  4  2  1  1  1  
14 4  2  2  3  5
```

```

18  4  2  3  4  2
13  4  2  5  1  1
6   4  2  5  4  5
7   4  3  1  4  2
4   4  3  3  2  2
1   4  4  1  3  4
9   4  5  3  1  2
5   5  2  2  4  3
12  5  3  4  2  4
15  5  3  5  1  5
19  5  4  5  1  3

```

Another interesting R function is `Reduce`. This function works with any binary function, that is, one which accepts two arguments, and repeatedly applies the binary function until the list is reduced to a single answer. One of the most important uses of `Reduce` is to merge more than two data frames at a time. Suppose we have three data frames that we wish to merge:

```

> one = data.frame(id=sample(LETTERS[1:10]),a=rnorm(10))
> two = data.frame(id=sample(LETTERS[1:10]),b=rnorm(10))
> three = data.frame(id=sample(LETTERS[1:10]),c=rnorm(10))

```

Of course, we could call `merge` twice:

```

> both = merge(one,two)
> both = merge(both,three)

```

But what if we had 20 such data frames:

```

> twenty = lapply(letters[1:20],function(i){
+           z = data.frame(id=sample(LETTERS[1:10]),rnorm(10))
+           names(z)[2] = i
+           z})
> twenty[[1]]
  id      a
1 D  1.9112203
2 F -0.7198699
3 J -0.4290842
4 E  0.7849467
5 B  1.5758446
6 I  0.4998762
7 H  0.3568294
8 A -0.8567896
9 G -0.7182695
10 C -0.6894396

```

```
> twenty[[2]]
  id      b
1  C  2.03416915
2  J  0.69535618
3  E -0.66387813
4  H -0.25343952
5  D -1.04782198
6  G  1.56147090
7  B -0.28126409
8  F -0.30900749
9  I  0.52277648
10 A  0.03841634
```

We simply pass `Reduce` the `merge` function and the list of data frames, and it does the rest:

```
> all = Reduce(merge,twenty)
   id      a      b      c      d      e      f
1  A -0.8567896  0.03841634  1.1137667 -0.6139491 -1.2893453  0.5760507
2  B  1.5758446 -0.28126409  0.2777182 -0.1615362  0.8221153 -0.3328126
3  C -0.6894396  2.03416915  0.6990830  0.9430438 -0.7059770 -0.1309771
4  D  1.9112203 -1.04782198 -0.1040303  0.2433795  0.7645007 -0.3161749
5  E  0.7849467 -0.66387813  0.0683588 -0.5752617 -0.4059950 -1.2280617
6  F -0.7198699 -0.30900749 -1.0955231 -0.8593922  0.1585112  0.5434146

      g      h      i      j      k      l      m
1 1.1469512  1.8675396 -1.0730001 -0.6306937  2.5853058  0.2524264  0.67496861
2 0.3483675  0.6846907 -1.7896035  0.7710611  1.9890408  0.6174135  0.16124957
3 -0.9972951 -0.7140730  0.7919375  0.3678690  1.2336345 -0.2323708 -0.52465997
4 0.1179205 -0.1865189 -1.0364221  0.6013162  1.3426701 -0.3513651  0.48844598
5 -0.1171988 -0.1514659  0.2650891  0.1813734  0.9233442 -1.7945477 -0.07855065
6 -0.2897749 -0.5888773  1.8274135 -0.3195992  0.6230858 -0.4897521 -0.49559174

      n      o      p      q      r      s      t
1 0.5551094  1.6335976 -1.6149968 -0.0601484  0.1622304  0.1624207 -1.1408366
2 1.3832478 -0.8993505 -0.3657969 -0.2817061  0.2501859 -0.2096964 -0.5082768
3 -0.7369804  0.2675346 -0.3801290  2.2369038  1.7701125  0.6809073 -1.1597869
4 1.4028788 -0.9317347  0.1623700 -2.8401085 -2.0066582 -0.5500940  1.9968541
5 -0.2516113 -1.1269027  2.2224559 -0.2109566  1.6218192 -0.7154724  1.8204627
6 2.0342595 -0.1864191 -0.7241448  1.4295025 -2.1206024  0.4883462 -0.1468977
```

Another useful R function is `reshape`. This function reorganizes data in either the so-called wide or long format. In the wide format, all observations for a single individual are recorded in one row of a data frame, while in the long view, each observation is a separate row, with an additional variable to identify the observation. For example, here is a data set, in long format, representing test scores on several different observations:

```

> data = data.frame(id=rep(1:3,rep(3,3)),time=rep(1:3,3),x=rnorm(9))
> data
  id time         x
1  1    1 -0.85519374
2  1    2 -0.60227681
3  1    3  0.47671168
4  2    1  0.28097796
5  2    2 -2.20903570
6  2    3  0.08311228
7  3    1 -0.56753562
8  3    2  0.65588709
9  3    3  0.42419294

```

To convert to wide format, the following statements could be used:

```

> reshape(data,idvar='id',timevar='time',v.names='x',direction='wide')
  id      x.1      x.2      x.3
1  1 -0.8551937 -0.6022768 0.47671168
4  2  0.2809780 -2.2090357 0.08311228
7  3 -0.5675356  0.6558871 0.42419294

```

Notice that not all possible combinations of observations need to be present:

```

> data1 = data[-7,]
> reshape(data1,idvar='id',timevar='time',v.names='x',direction='wide')
  id      x.1      x.2      x.3
1  1 -0.8551937 -0.6022768 0.47671168
4  2  0.2809780 -2.2090357 0.08311228
8  3       NA  0.6558871 0.42419294

```

The missing observations are represented as NA in the output data frame.

To illustrate the wide-to-long transformation, consider the following data:

```

> data = data.frame(id = rep(1:3),x1=c(1,2,3),x2=c(10,20,30),x3=c(100,200,300))
> data
  id x1 x2  x3
1  1  1 10 100
2  2  2 20 200
3  3  3 30 300
> reshape(data,idvar='id',varying=list(c('x1','x2','x3')),direction='long')
  id time  x1
1.1  1    1   1
2.1  2    1   2
3.1  3    1   3
1.2  1    2  10

```

```

2.2 2      2  20
3.2 3      2  30
1.3 1      3 100
2.3 2      3 200
3.3 3      3 300

```

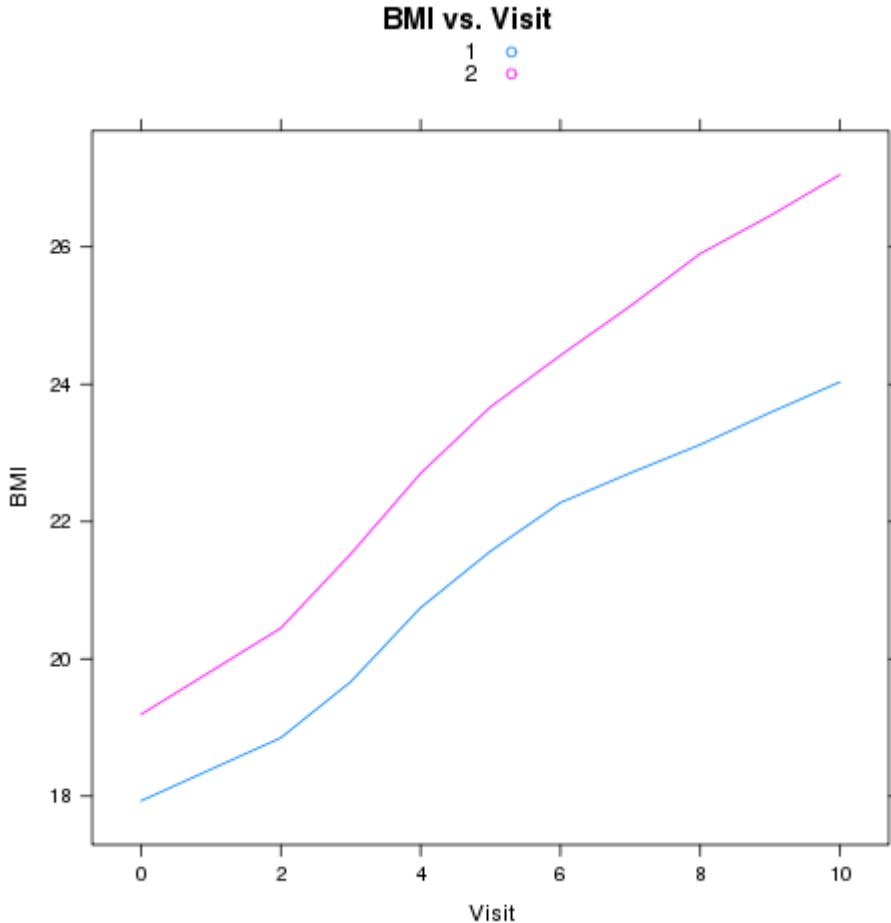
The trickiest part is noticing that the names of the three varying variables have to be placed in a list.

As another example of the use of reshape, consider a study looking at the BMI (Body Mass Index) of girls over time, measured for two different races. This data can be found in a tab-delimited file at <http://www.stat.berkeley.edu/classes/s133/data/bmi.tab>. The variables in the file are ID, RACE, BMI, and VISIT. Suppose we wish to produce a plot of the average BMI versus visit, with separate lines for the two races. We could use `xyplot` from the `lattice` package as follows:

```

> bmi = read.delim('http://www.stat.berkeley.edu/classes/s133/data/bmi.tab')
> avgbmi = aggregate(list(BMI=bmi$BMI),list(Race=bmi$RACE,Visit=bmi$VISIT),
+                      mean,na.rm=TRUE)
> library(lattice)
> xyplot(BMI~Visit,group=Race,type='l',auto.key=TRUE,
+         main='BMI vs. Visit',data=avgbmi)

```



The plot appears below:

Now suppose we wish to calculate the correlations between the BMIs at the different visits. The `cor` function expects all of the variables for which we want the correlations to be separate columns in the data frame we pass to the function. We can rearrange the data to make it suitable for calculating the correlations by calling `reshape` as follows:

```
> bmi1 = reshape(bmi,idvar=c('ID','RACE'),timevar='VISIT',v.names='BMI',
+                 direction='wide')
```

We can now calculate the correlations by race:

```
> by(bmi1,bmi1$RACE,function(x)cor(x[,-c(1,2)],use='complete.obs'))
bmi1$RACE: 1
      BMI.0     BMI.2     BMI.3     BMI.4     BMI.5     BMI.6     BMI.7
BMI.0  1.0000000 0.9513181 0.9238520 0.8871666 0.8489529 0.8227992 0.8018502
BMI.2  0.9513181 1.0000000 0.9490917 0.9144459 0.8625492 0.8339581 0.8110637
BMI.3  0.9238520 0.9490917 1.0000000 0.9517413 0.9025264 0.8612542 0.8327787
BMI.4  0.8871666 0.9144459 0.9517413 1.0000000 0.9458218 0.9054674 0.8781602
BMI.5  0.8489529 0.8625492 0.9025264 0.9458218 1.0000000 0.9530917 0.9105118
BMI.6  0.8227992 0.8339581 0.8612542 0.9054674 0.9530917 1.0000000 0.9568980
```

	BMI.7	0.8018502	0.8110637	0.8327787	0.8781602	0.9105118	0.9568980	1.0000000
BMI.8	0.7968433	0.8026011	0.8166798	0.8540803	0.8859762	0.9319712	0.9630340	
BMI.9	0.7888784	0.7931164	0.8042629	0.8351485	0.8690846	0.9107269	0.9316619	
BMI.10	0.7630397	0.7690169	0.7777677	0.8131666	0.8430350	0.8803034	0.8952306	
	BMI.8	BMI.9	BMI.10					
BMI.0	0.7968433	0.7888784	0.7630397					
BMI.2	0.8026011	0.7931164	0.7690169					
BMI.3	0.8166798	0.8042629	0.7777677					
BMI.4	0.8540803	0.8351485	0.8131666					
BMI.5	0.8859762	0.8690846	0.8430350					
BMI.6	0.9319712	0.9107269	0.8803034					
BMI.7	0.9630340	0.9316619	0.8952306					
BMI.8	1.0000000	0.9596225	0.9209648					
BMI.9	0.9596225	1.0000000	0.9488572					
BMI.10	0.9209648	0.9488572	1.0000000					

---

bmi1\$RACE: 2

	BMI.0	BMI.2	BMI.3	BMI.4	BMI.5	BMI.6	BMI.7
BMI.0	1.0000000	0.9628273	0.9281780	0.9010607	0.8646773	0.8410384	0.8173900
BMI.2	0.9628273	1.0000000	0.9607964	0.9334043	0.8940498	0.8730059	0.8543804
BMI.3	0.9281780	0.9607964	1.0000000	0.9656133	0.9300707	0.9081140	0.8886918
BMI.4	0.9010607	0.9334043	0.9656133	1.0000000	0.9605275	0.9378850	0.9130036
BMI.5	0.8646773	0.8940498	0.9300707	0.9605275	1.0000000	0.9722481	0.9465896
BMI.6	0.8410384	0.8730059	0.9081140	0.9378850	0.9722481	1.0000000	0.9751832
BMI.7	0.8173900	0.8543804	0.8886918	0.9130036	0.9465896	0.9751832	1.0000000
BMI.8	0.8051828	0.8423513	0.8746659	0.8979719	0.9293276	0.9561339	0.9769345
BMI.9	0.8003754	0.8389726	0.8689569	0.8868970	0.9147281	0.9410288	0.9582904
BMI.10	0.7862279	0.8224567	0.8464351	0.8640517	0.8894121	0.9157546	0.9316408
	BMI.8	BMI.9	BMI.10				
BMI.0	0.8051828	0.8003754	0.7862279				
BMI.2	0.8423513	0.8389726	0.8224567				
BMI.3	0.8746659	0.8689569	0.8464351				
BMI.4	0.8979719	0.8868970	0.8640517				
BMI.5	0.9293276	0.9147281	0.8894121				
BMI.6	0.9561339	0.9410288	0.9157546				
BMI.7	0.9769345	0.9582904	0.9316408				
BMI.8	1.0000000	0.9767038	0.9460700				
BMI.9	0.9767038	1.0000000	0.9675484				
BMI.10	0.9460700	0.9675484	1.0000000				

Not surprisingly, there's a very high correlation between the BMI at the different visits for both races.

# Chapter 10

## Classification Analysis

## 10.1 Introduction to Classification Methods

When we apply cluster analysis to a dataset, we let the values of the variables that were measured tell us if there is any structure to the observations in the data set, by choosing a suitable metric and seeing if groups of observations that are all close together can be found. If we have an auxilliary variable (like the country of origin from the cars example), it may be interesting to see if the natural clustering of the data corresponds to this variable, but it's important to remember that the idea of clustering is just to see if any groups form naturally, not to see if we can actually figure out which group an observation belongs to based on the values of the variables that we have.

When the true goal of our data analysis is to be able to predict which of several non-overlapping groups an observation belongs to, the techniques we use are known as classification techniques. We'll take a look at three classification techniques: kth nearest neighbor classification, linear discriminant analysis, and recursive partitioning.

## 10.2 kth Nearest Neighbor Classification

The idea behind nearest neighbor classification is simple and somewhat intuitive – find other observations in the data that are close to an observation we're interested, and classify that observation based on the class of its neighbors. The number of neighbors that we consider is where the "k" comes in – usually we'll have to look at several different values of k to determine which ones work well with a particular data set. Values in the range of one to ten are usually reasonable choices.

Since we need to look at all of the distances between one observation and all the others in order to find the neighbors, it makes sense to form a distance matrix before starting a nearest neighbor classification. Each row of the distance matrix tells us the distances to all the other observations, so we need to find the k smallest values in each row of the distance matrix. Once we find those smallest values, we determine which observations they belong to, and look at how those observations were classified. We assign whichever value of the classification that was most common among the k nearest neighbors as our guess (predicted value) for the current observation, and then move on to the next row of the distance matrix. Once we've looked at every row of the distance matrix, we'll have classified every observation, and can compare the predicted classification with the actual classification. To see how well we've done, various error rates can be examined. If the observations are classified as TRUE / FALSE, for example disease or no disease, then we can look at two types of error rates. The first type of error rate, known as Type I error, occurs when we say that an observation should be classified as TRUE when it really should have been FALSE. The other type of error (Type II) occurs when we say that an observation should be classified as FALSE when it should have been TRUE. When the classification is something other than TRUE/FALSE, we can report an overall error rate, that is, the fraction of observations for which our prediction was not correct. In either case, the error rates can be calculated in R by using the `table` function. As a simple example, suppose we have two vectors: `actualvalues`, which contains

the actual values of a classification variable, and `predvalues`, the value that our classification predicted:

```
> actualvalues = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE)
> predvalues = c(TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, TRUE)
> tt = table(actualvalues,predvalues)
> tt
      predvalues
actualvalues FALSE TRUE
  FALSE     3    2
  TRUE      1    4
```

The observations that contribute to Type I error (the actual value is false but we predicted true) can be found in the first row and second column; those that contribute to Type II error can be found in the second row and first column. Since the `table` function returns a matrix, we can calculate the rows as follows:

```
> tot = sum(tt)
> type1 = tt['FALSE', 'TRUE'] / tot
> type2 = tt['TRUE', 'FALSE'] / tot
> type1
[1] 0.2
> type2
[1] 0.1
```

## 10.3 Cross Validation

There's one problem with the above scheme. We used the data that we're making predictions about in the process of making those predictions. In other words, the data that we're making predictions for is not independent of the data that we're using to make the predictions. As might be expected, it's been shown in practice that calculating error rates this way will almost always make our classification method look better than it should be. If the data can be naturally (or even artificially) divided into two groups, then one can be used as a training set, and the other can be used as a test set – we'd calculate our error rates only from the classification of the test set using the training set to make our predictions. Many statisticians don't like the idea of having to "hold back" some of their data when building models, so an alternative way to bring some independence to our predictions known as v-fold cross validation has been devised. The idea is to first divide the entire data set into v groups. To classify objects in the first group, we don't use any of the first group to make our predictions; in the case of k-th nearest neighbor classification, that would mean that when we're looking for the smallest distances in order to classify an observation, we don't consider any of the distances corresponding to other members of the same group that the current one belongs to. The basic idea is that we want to make the prediction for an observation as

independent from that observation as we can. We continue through each of the  $v$  groups, classifying observations in each group using only observations from the other groups. When we're done we'll have a prediction for each observation, and can compare them to the actual values as in the previous example.

Another example of cross-validation is leave-out-one cross-validation. With this method, we predict the classification of an observation without using the observation itself. In other words, for each observation, we perform the analysis without using that observation, and then predict where that observation would be classified using that analysis.

## 10.4 Linear Discriminant Analysis

One of the oldest forms of classification is known as linear discriminant analysis. The idea is to form linear combinations of predictor variables (similar to a linear regression model) in such a way that the average value of these linear combinations will be as different as possible for the different levels of the classification variable. Based on the values of the linear combinations, linear discriminant analysis reports a set of posterior probabilities for every level of the classification, for each observation, along with the level of the classification variable that the analysis predicted. Suppose we have a classification variable that can take one of three values: after a linear discriminant analysis, we will have three probabilities (adding up to one) for each variable that tell how likely it is that the observation be categorized into each of the three categories; the predicted classification is the one that had the highest probability, and we can get insight into the quality of the classification by looking at the values of the probabilities.

To study the different classification methods, we'll use a data set about different wines. This data set contains various measures regarding chemical and other properties of the wines, along with a variable identifying the Cultivar (the particular variety of the grape from which the wine was produced). We'll try to classify the observations based on the Cultivar, using the other variables. The data is available at <http://www.stat.berkeley.edu/classes/s133/data/wine.data>; information about the variables is at <http://www.stat.berkeley.edu/classes/s133/data/wine.names>

First, we'll read in the `wine` dataset:

```
wine = read.csv('http://www.stat.berkeley.edu/classes/s133/data/wine.data', header=FALSE)
names(wine) = c("Cultivar", "Alcohol", "Malic.acid", "Ash", "Alkalinity.ash",
              "Magnesium", "Phenols", "Flavanoids", "NF.phenols", "Proanthocyanins",
              "Color.intensity", "Hue", "OD.Ratio", "Proline")
wine$Cultivar = factor(wine$Cultivar)
```

Notice that I set `wine$Cultivar` to be a factor. Factors are very important and useful in modeling functions because categorical variables almost always have to be treated differently than numeric variables, and turning a categorical variable into a factor will insure that they are always used properly in modeling functions. Not surprisingly, the dependent variable for `lda` must be a factor.

The `class` library of R provides two functions for nearest neighbor classification. The first, `knn`, takes the approach of using a training set and a test set, so it would require holding back some of the data. The other function, `knn.cv` uses leave-out-one cross-validation, so it's more suitable to use on an entire data set.

Let's use `knn.cv` on the wine data set. Since, like cluster analysis, this technique is based on distances, the same considerations regarding standardization as we saw with cluster analysis apply. Let's examine a summary for the data frame:

```
> summary(wine)
   Cultivar      Alcohol       Malic.acid        Ash      Alkalinity.ash
1:59     Min.    :11.03     Min.    :0.740     Min.    :1.360     Min.    :10.60
2:71     1st Qu.:12.36    1st Qu.:1.603    1st Qu.:2.210    1st Qu.:17.20
3:48     Median  :13.05    Median  :1.865    Median  :2.360    Median  :19.50
          Mean    :13.00    Mean    :2.336    Mean    :2.367    Mean    :19.49
          3rd Qu.:13.68    3rd Qu.:3.083    3rd Qu.:2.558    3rd Qu.:21.50
          Max.    :14.83    Max.    :5.800    Max.    :3.230    Max.    :30.00
   Magnesium      Phenols      Flavanoids      NF.phenols
Min.    : 70.00     Min.    :0.980     Min.    :0.340     Min.    :0.1300
1st Qu.: 88.00     1st Qu.:1.742     1st Qu.:1.205     1st Qu.:0.2700
Median  : 98.00     Median  :2.355     Median  :2.135     Median  :0.3400
Mean    : 99.74     Mean    :2.295     Mean    :2.029     Mean    :0.3619
3rd Qu.:107.00    3rd Qu.:2.800     3rd Qu.:2.875     3rd Qu.:0.4375
Max.    :162.00    Max.    :3.880     Max.    :5.080     Max.    :0.6600
   Proanthocyanins Color.intensity        Hue      OD.Ratio
Min.    :0.410      Min.    : 1.280     Min.    :0.4800     Min.    :1.270
1st Qu.:1.250      1st Qu.: 3.220     1st Qu.:0.7825     1st Qu.:1.938
Median  :1.555      Median  : 4.690     Median  :0.9650     Median  :2.780
Mean    :1.591      Mean    : 5.058     Mean    :0.9574     Mean    :2.612
3rd Qu.:1.950      3rd Qu.: 6.200     3rd Qu.:1.1200     3rd Qu.:3.170
Max.    :3.580      Max.    :13.000     Max.    :1.7100     Max.    :4.000
   Proline
Min.    : 278.0
1st Qu.: 500.5
Median  : 673.5
Mean    : 746.9
3rd Qu.: 985.0
Max.    :1680.0
```

Since the scale of the variables differ widely, standardization is probably a good idea. We'll divide each variable by its standard deviation to try to give each variable more equal weight in determining the distances:

```
> wine.use = scale(wine[,-1], scale=apply(wine[,-1], 2, sd))
> library(class)
```

```

> res = knn.cv(wine.use,wine$Cultivar,k=3)
> names(res)
NULL
> length(res)
[1] 178

```

Since there are no names, and the length of `res` is the same as the number of observations, `knn.cv` is simply returning the classifications that the method predicted for each observation using leave-out-one cross validation. This means we can compare the predicted values to the true values using `table`:

```

> table(res,wine$Cultivar)

res  1   2   3
  1 59   4   0
  2   0 63   0
  3   0   4 48

```

To calculate the proportion of incorrect classifications, we can use the `row` and `col` functions. These unusual functions don't seem to do anything very useful when we simply call them:

```

> tt = table(res,wine$Cultivar)
> row(tt)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3
> col(tt)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3

```

However, if you recall that the misclassified observations are those that are off the diagonal, we can find those observations as follows:

```

> tt[row(tt) != col(tt)]
[1] 0 0 4 4 0 0

```

and the proportion of misclassified observations can be calculated as:

```

> sum(tt[row(tt) != col(tt)]) / sum(tt)
[1] 0.04494382

```

or a missclassification rate of about 4.5

Could we have done better if we used 5 nearest neighbors instead of 3?

```
> res = knn.cv(wine.use,wine$Cultivar,k=5)
> tt = table(res,wine$Cultivar)
> sum(tt[row(tt) != col(tt)]) / sum(tt)
[1] 0.02808989
```

How about using just the single nearest neighbor?

```
> res = knn.cv(wine.use,wine$Cultivar,k=1)
> tt = table(res,wine$Cultivar)
> sum(tt[row(tt) != col(tt)]) / sum(tt)
[1] 0.04494382
```

For this data set, using k=5 did slightly better than 1 or 3.

In R, linear discriminant analysis is provided by the `lda` function from the `MASS` library, which is part of the base R distribution.

Like many modeling and analysis functions in R, `lda` takes a formula as its first argument. A formula in R is a way of describing a set of relationships that are being studied. The dependent variable, or the variable to be predicted, is put on the left hand side of a tilda (`~`) and the variables that will be used to model or predict it are placed on the right hand side of the tilda, joined together by plus signs (`+`). To save typing, you can provide the name of a data frame through the `data=` argument, and use the name of the variables in the data frame in your formula without retyping the data frame name or using the `with` function.

A convenience offered by the modeling functions is that a period (`.`) on the right-hand side of the tilda in a formula is interpreted as meaning “all the other variables in the data frame, except the dependent variable”. So a very popular method of specifying a formula is to use the period, and then use subscripting to limit the `data=` argument to just the variables you want to fit. In this example, we don’t need to do that, because we really do want to use all the variables in the data set.

```
> wine.lda = lda(Cultivar ~ ., data=wine)
```

We’ll see that most of the modeling functions in R share many things in common. For example, to predict values based on a model, we pass the model object to the `predict` function along with a data frame containing the observations for which we want predictions:

```
> pred = predict(wine.lda, wine)
```

To see what’s available from the call to `predict`, we can look at the names of the `pred` object:

```
> names(pred)
[1] "class"      "posterior"   "x"
```

The predicted classification is stored in the `class` component of the object returned by `predict`. Now that we’ve got the predicted classes, we can see how well the classification went by making a cross-tabulation of the real Cultivar with our prediction, using the `table` function:

```
> table(wine$Cultivar, pred$class)
    predclass
      1  2  3
  1 59  0  0
  2  0 71  0
  3  0  0 48
```

Before we get too excited about these results, remember the caution about predicting values based on models that were built using those values. The error rate we see in the table (0) is probably an overestimate of how good the classification rule is. We can use v-fold cross validation on the data, by using the `lda` command repeatedly to classify groups of observations (folds) using the rest of the data to build the model. We could write this out “by hand”, but it would be useful to have a function that could do this for us. Here’s such a function:

```

vlda = function(v,formula,data,cl){
  require(MASS)
  grps = cut(1:nrow(data),v,labels=FALSE)[sample(1:nrow(data))]
  pred = lapply(1:v,function(i,formula,data){
    omit = which(grps == i)
    z = lda(formula,data=data[-omit,])
    predict(z,data[omit,])
  },formula,data)

  wh = unlist(lapply(pred,function(pp)pp$class))
  table(wh,cl[order(grps)])
}

```

This function accepts four arguments: `v`, the number of folds in the cross classification, `formula` which is the formula used in the linear discriminant analysis, `data` which is the data frame to use, and `cl`, the classification variable (`wine$Cultivar` in this case).

By using the `sample` function, we make sure that the groups that are used for cross-validation aren't influenced by the ordering of the data – notice how the classification variable (`cl`) is indexed by `order(grps)` to make sure that the predicted and actual values line up properly.

Applying this function to the `wine` data will give us a better idea of the actual error rate of the classifier:

```

> vlda(5,Cultivar~.,wine,wine$Cultivar)
wh   1   2   3
1 59   1   0
2 0 69   1
3 0   1 47

```

While the error rate is still very good, it's not quite perfect:

```

> error = sum(tt[row(tt) != col(tt)]) / sum(tt)
> error
[1] 0.01685393

```

Note that because of the way we randomly divide the observations, you'll see a slightly different table every time you run the `vlda` function.

We could use a similar method to apply v-fold cross-validation to the kth nearest neighbor classification. Since the `knn` function accepts a training set and a test set, we can make each fold a test set, using the remainder of the data as a training set. Here's a function to apply this idea:

```

vknn = function(v,data,cl,k){
  grps = cut(1:nrow(data),v,labels=FALSE)[sample(1:nrow(data))]
  pred = lapply(1:v,function(i,data,cl,k){

```

```

        omit = which(grps == i)
        pcl = knn(data[-omit,],data[omit,],cl[-omit],k=k)
    },data,cl,k)

wh = unlist(pred)
table(wh,cl[order(grps)])
}

```

Let's apply the function to the standardized wine data:

```

> tt = vknn(5,wine.use,wine$Cultivar,5)
> tt

```

```

wh   1   2   3
1 59  2  0
2  0 66  0
3  0  3 48
> sum(tt[row(tt) != col(tt)]) / sum(tt)
[1] 0.02808989

```

Note that this is the same misclassification rate as achieved by the “leave-out-one” cross validation provided by `knn.cv`.

Both the nearest neighbor and linear discriminant methods make it possible to classify new observations, but they don't give much insight into what variables are important in the classification. The `scaling` element of the object returned by `lda` shows the linear combinations of the original variables that were created to distinguish between the groups:

```

> wine.lda$scaling
          LD1           LD2
Alcohol      -0.403399781  0.8717930699
Malic.acid    0.165254596  0.3053797325
Ash         -0.369075256  2.3458497486
Alkalinity.ash 0.154797889 -0.1463807654
Magnesium    -0.002163496 -0.0004627565
Phenols       0.618052068 -0.0322128171
Flavanoids    -1.661191235 -0.4919980543
NF.phenols    -1.495818440 -1.6309537953
Proanthocyanins 0.134092628 -0.3070875776
Color.intensity 0.355055710  0.2532306865
Hue          -0.818036073 -1.5156344987
OD.Ratio     -1.157559376  0.0511839665
Proline      -0.002691206  0.0028529846

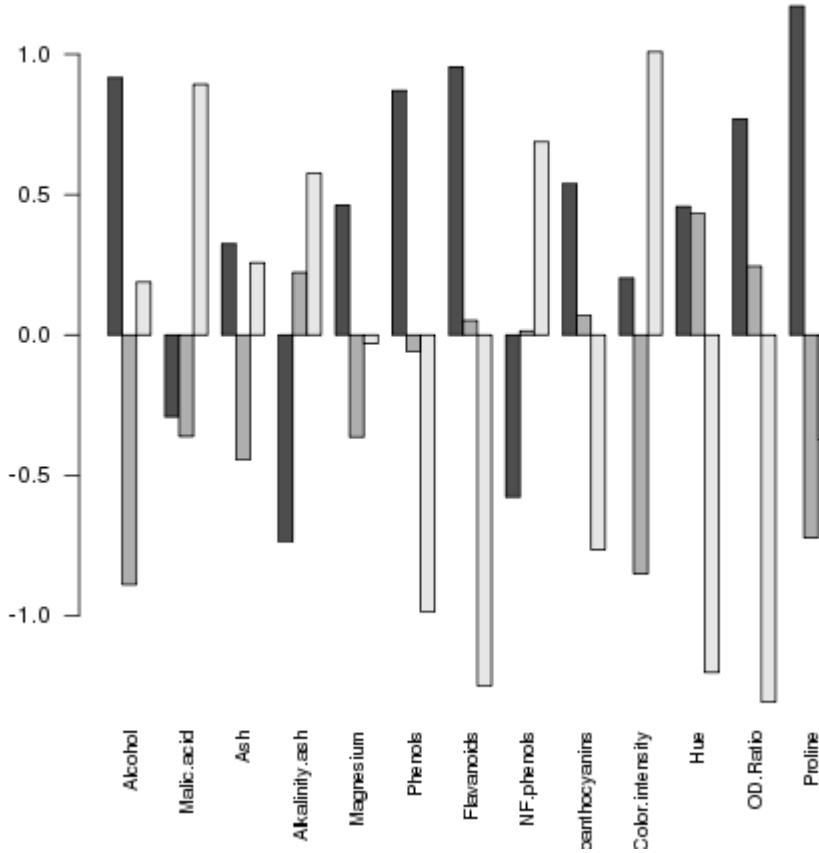
```

It's really not that easy to interpret them, but variables with large absolute values in the scalings are more likely to influence the process. For this data, `Flavanoids`, `NF.phenols`, `Ash`, and `Hue` seem to be among the important variables.

A different way to get some insight into this would be to examine the means of each of the variables broken down by the classification variable. Variables which show a large difference among the groups would most likely be the ones that are useful in predicting which group an observation belongs in. One graphical way of displaying this information is with a barplot. To make sure we can see differences for all the variables, we'll use the standardized version of the data:

```
> mns = aggregate(wine.use,wine['Cultivar'],mean)
> rownames(mns) = mns$Cultivar
> mns$Cultivar = NULL
> barplot(as.matrix(mns),beside=TRUE,cex.names=.8,las=2)
```

The `las` parameter rotates the labels on the x-axis so that we can see them all. Here's the plot:



It seems like the big differences are found in Flavanoids, Hue and OD.Ratio

Now let's take a look at a method that makes it very clear which variables are useful in distinguishing among the groups.

## 10.5 Recursive Partitioning

An alternative classification method, developed in the 1980s, has attracted a lot of attention in a variety of different fields. The technique, known as recursive partitioning or CART (Classification and Regression Trees), can be used for either classification or regression – here we'll concentrate on its use for classification. The basic idea is to examine, for each of the variables that we're using in our classification model, the results of splitting the data set based on one of the values of that variable, and then examining how that split helps us distinguish between the different classes of interest. For example, suppose we're using just one variable to help distinguish between two classes:

```
> mydata = data.frame(x=c(7,12,8,3,4,9,2,19),grp=c(1,2,1,1,1,2,2,2))
> mydata
  x grp
1 7  1
2 12 2
3 8  1
4 3  1
5 4  1
6 9  2
7 2  2
8 19 2
```

We'd consider each value of `x` in the data, and split the data into two groups: one where `x` was less than or equal to the value and the other where `x` is greater than the value. We then look at how our classification variable (`grp` in this example) breaks down when the data is split this way. For this example, we can look at all the possible cross-tabulations:

```
> tbls = sapply(mydata$x,function(val)table(mydata$x <= val,mydata$grp))
> names(tbls) = mydata$x
> tbls
> tbls
$"7"
  1 2
FALSE 1 3
TRUE  3 1

$"12"
  1 2
FALSE 0 1
```

TRUE 4 3  
\$"8"

1 2  
FALSE 0 3  
TRUE 4 1

\$"3"

1 2  
FALSE 3 3  
TRUE 1 1

\$"4"

1 2  
FALSE 2 3  
TRUE 2 1

\$"9"

1 2  
FALSE 0 2  
TRUE 4 2

\$"2"

1 2  
FALSE 4 3  
TRUE 0 1

\$"19"

1 2  
TRUE 4 4

If you look at each of these tables, you can see that when we split with the rule  $x \leq 8$ , we get the best separation; all three of the cases where  $x$  is greater than 8 are classified as 2, and four out of five of the cases where  $x$  is less than 8 are classified as 1. For this phase of a recursive partitioning, the rule  $x \leq 8$  would be chosen to split the data.

In real life situations, where there will always be more than one variable, the process would be repeated for each of the variables, and the chosen split would be the one from among all the variables that resulted in the best separation or node purity as it is sometimes known. Now that we've found the single best split, our data is divided into two groups based on that split. Here's where the recursive part comes in: we keep doing the same thing to each side of the split data, searching through all values of all variables until we find the one that gives us the best separation, splitting the data by that value and then continuing. As implemented in R through the `rpart` function in the `rpart` library, cross validation is used internally to determine when we should stop splitting the data, and present a final tree as the output. There are also options to the `rpart` function specifying the minimum number of observations allowed in a split, the minimum number of observations allowed in one of the final nodes and the maximum number of splits allowed, which can be set through the `control=` argument to `rpart`. See the help page for `rpart.control` for more information.

To show how information from recursive partitioning is displayed, we'll use the same data that we used with `lda`. Here are the commands to run the analysis:

```
> library(rpart)
> wine.rpart = rpart(Cultivar~., data=wine)
> wine.rpart
n= 178

node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 178 107 2 (0.33146067 0.39887640 0.26966292)
  2) Proline>=755 67  10 1 (0.85074627 0.05970149 0.08955224)
    4) Flavanoids>=2.165 59   2 1 (0.96610169 0.03389831 0.00000000) *
    5) Flavanoids< 2.165 8   2 3 (0.00000000 0.25000000 0.75000000) *
  3) Proline< 755 111  44 2 (0.01801802 0.60360360 0.37837838)
    6) OD.Ratio>=2.115 65   4 2 (0.03076923 0.93846154 0.03076923) *
    7) OD.Ratio< 2.115 46   6 3 (0.00000000 0.13043478 0.86956522)
    14) Hue>=0.9 7   2 2 (0.00000000 0.71428571 0.28571429) *
    15) Hue< 0.9 39   1 3 (0.00000000 0.02564103 0.97435897) *
```

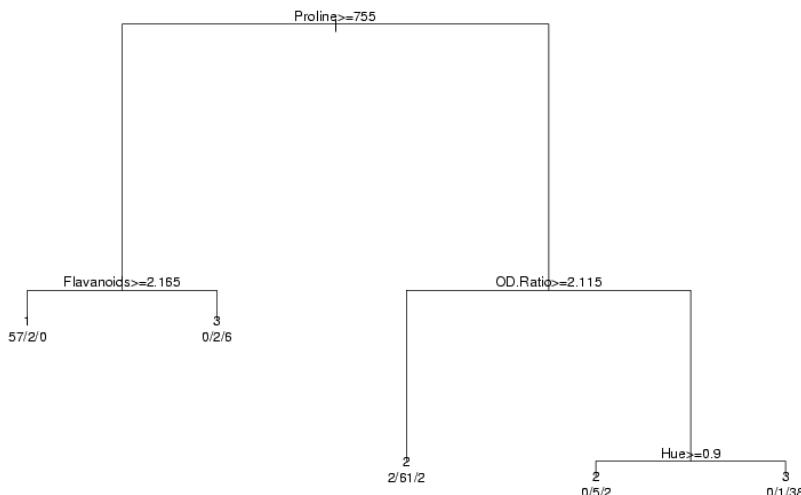
The display always starts at the root (all of the data), and reports the splits in the order they occurred. So after examining all possible values of all the variables, `rpart` found that the `Proline` variable did the best job of dividing up the observations into the different cultivars; in particular of the 67 observations for which `Proline` was greater than 755, the fraction that had `Cultivar == 1` was .8507. Since there's no asterisk after that line, it indicates that `rpart` could do a better job by considering other variables. In particular if `Proline` was  $\geq 755$ , and `Flavanoids` was  $\geq 2.165$ , the fraction of Cultivar 1 increases to .9666; the asterisk at the end of a line indicates that this is a terminal node, and, when classifying an observation if `Proline` was  $\geq 755$  and `Flavanoids` was  $\geq 2.165$ , `rpart`

would immediately assign it to Cultivar 1 without having to consider any other variables. The rest of the output can be determined in a similar fashion.

An alternative way of viewing the output from `rpart` is a tree diagram, available through the `plot` function. In order to identify the parts of the diagram, the `text` function also needs to be called.

```
> plot(wine.rpart)
> text(wine.rpart,use.n=TRUE,xpd=TRUE)
```

The `xpd=TRUE` is a graphics parameter that is useful when a plot gets truncated, as sometimes happens with `rpart` plots. There are other options to `plot` and `text` which will change the appearance of the output; you can find out more by looking at the help pages for `plot.rpart` and `text.rpart`. The graph appears below.



In order to calculate the error rate for an `rpart` analysis, we once again use the `predict` function:

```
> pred.rpart = predict(wine.rpart,wine)
```

As usual we can check for names to see how to access the predicted values:

```
> names(pred.rpart)
NULL
```

Since there are no names, we'll examine the object directly:

```
> head(pred.rpart)
  1          2          3
```

```
1 0.96610169 0.03389831 0.00000000
2 0.96610169 0.03389831 0.00000000
3 0.96610169 0.03389831 0.00000000
4 0.96610169 0.03389831 0.00000000
5 0.03076923 0.93846154 0.03076923
6 0.96610169 0.03389831 0.00000000
```

All that `predict` returns in this case is a matrix with estimated probabilities for each cultivar for each observation; we'll need to find which one is highest for each observation. Doing it for one row is easy:

```
> which.max(rped.rpart[1,])
1
1
```

To repeat this for each row, we can pass `which.max` to `apply`, with a second argument of 1 to indicate we want to apply the function to each row:

```
> table(apply(pred.rpart, 1, which.max), wine$Cultivar)
```

	1	2	3
1	57	2	0
2	2	66	4
3	0	3	44

To compare this to other methods, we can again use the `row` and `col` functions:

```
> sum(tt[row(tt) != col(tt)]) / sum(tt)
[1] 0.06179775
```

Since `rpart` uses cross validation internally to build its decision rules, we can probably trust the error rates implied by the table.

# Chapter 11

## Random Numbers and Simulations

## 11.1 Hypothesis Testing

Much of classical statistics is concerned with the idea of hypothesis testing. This is a formal framework that we can use to pose questions about a variety of topics in a consistent form that lets us apply statistical techniques to make statements about how results that we've gathered relate to questions that we're interested in. If we carefully follow the rules of hypothesis testing, then we can confidently talk about the probability of events that we've observed under a variety of hypotheses, and put our faith in those hypotheses that seem the most reasonable. Central to the idea of hypothesis testing is the notion of null and alternative hypotheses. We want to present our question in the form of a statement that assumes that nothing has happened (the null hypothesis) versus a statement that describes a specific relation or condition that might better explain our data.

For example, suppose we've got a coin, and we want to find out if it's true, that is, if, when we flip the coin, are we as likely to see heads as we are to see tails. A null hypothesis for this situation could be

$H_0$ : We're just as likely to get heads as tails when we flip the coin.

A suitable alternative hypothesis might be:

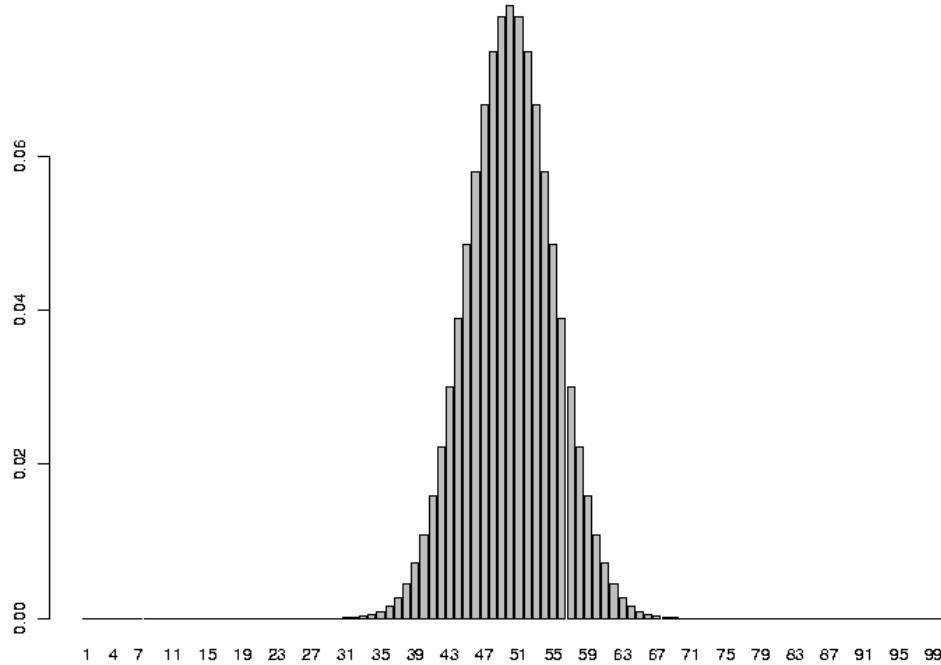
$H_a$ : We're more likely to see either heads or tails when we flip the coin.

An alternative hypothesis such as this is called two-sided, since we'll reject the null hypothesis if heads are more likely or if tails are more likely. We could use a one-sided alternative hypothesis like "We're more likely to see tails than heads," but unless you've got a good reason to believe that it's absolutely impossible to see more heads than tails, we usually stick to two-sided alternative hypotheses.

Now we need to perform an experiment in order to help test our hypothesis. Obviously, tossing the coin once or twice and counting up the results won't help very much. Suppose I say that in order to test the null hypothesis that heads are just as likely as tails, I'm going to toss the coin 100 times and record the results. Carrying out this experiment, I find that I saw 55 heads and 45 tails. Can I safely say that the coin is true? Without some further guidelines, it would be very hard to say if this deviation from 50/50 really provides much evidence one way or the other. To proceed any further, we have to have some notion of what we'd expect to see in the long run if the null hypothesis was true. Then, we can come up with some rule to use in deciding if the coin is true or not, depending on how willing we are to make a mistake. Sadly, we can't make statements with complete certainty, because there's always a chance that, even if the coin was true, we'd happen to see more heads than tails, or, conversely, if the coin was weighted, we might just happen to see nearly equal numbers of heads and tails when we tossed the coin many times. The way we come up with our rule is by stating some assumptions about what we'd expect to see if the null hypothesis was true, and then making a decision rule based on those assumptions. For tossing a fair coin (which is what the null hypothesis states), most statisticians agree that the number of heads (or tails) that we would expect follows what is called a binomial distribution. This distribution takes

two parameters: the theoretical probability of the event in question (let's say the event of getting a head when we toss the coin), and the number of times we toss the coin. Under the null hypothesis, the probability is .5. In R, we can see the expected probability of getting any particular number of heads if we tossed a coin 100 times by plotting the density function for the binomial distribution with parameters 100 and .5.

```
> barplot(dbinom(1:100, 100, .5), names.arg=1:100)
```



As you'd probably expect, the most common value we'd see would be 50, but there are lots of reasonable values that aren't exactly 50 that would also occur pretty often. So to make a decision rule for when we will reject the null hypothesis, we would first decide how often we're willing to reject the null hypothesis when it was actually true. For example, looking at the graph, there's a tiny probability that we'd see as few as 30 heads or as many as 70 heads when we tossed a true coin 100 times. But if we tossed a coin 100 times and only saw 30 heads, we'd be wise if we said the coin wasn't true because 30 heads out of 100 is such a rare event. On the other hand, we can see on the graph that the probability of getting, say 45 heads is still pretty high, and we might very well want to accept the null hypothesis in this case. Without any compelling reason to choose otherwise, people usually will accept an error of 5% when performing a hypothesis test, although you might want to alter that value in situations when making such an error is more or less of a problem than usual. With a two-sided hypothesis like this one, to come up with a decision rule that would be wrong only 5% of the time, we'd need to find the number of heads for which we'd see fewer heads 2.5% of the time, and the number of heads for which we'd see more heads 2.5% of the time. To find quantities like this for the binomial distribution, we can use the `qbinom` function. The lower limit we want is the result of calling `qbinom` with an argument of 0.025, and the upper limit is the result of calling `qbinom` with an argument of 0.975:

```
> qbinom(.025,100,.5)
[1] 40
> qbinom(.975,100,.5)
[1] 60
```

So in this experiment, if the number of heads we saw was between 40 and 60 (out of 100 tosses), we would tentatively accept the null hypothesis; formally we would say that there's not enough evidence to reject the null hypothesis. If we saw fewer than 40 or more than 60 tosses, we would say we had enough evidence to reject the null hypothesis, knowing that we would be wrong only 5% of the time. When we reject the null hypothesis when it was actually true, it's said to be a Type I error. So in this example, we're setting the Type I error rate to 5%.

To summarize, the steps for performing a hypothesis test are:

1. Describe a null hypothesis and an alternative hypothesis.
2. Specify a significance or alpha level for the hypothesis test. This is the percent of the time that you're willing to be wrong when you reject the null hypothesis.
3. Formulate some assumptions about the distribution of the statistic that's involved in the hypothesis test. In this example we made the assumption that a fair coin follows a binomial distribution with  $p=0.50$ .
4. Using the assumptions you made and the alpha level you decided on, construct the rejection region for the test, that is, the values of the statistic for which you'll be willing to reject the null hypothesis. In this example, the rejection region is broken up into two sections: less than 40 heads and more than 60 heads.

Although it's not always presented in this way, one useful way to think about hypothesis testing is that if you follow the above steps, and your test statistic is in the rejection region then either the null hypothesis is not reasonable or your assumptions are wrong. By following this framework, we can test a wide variety of hypotheses, and it's always clear exactly what we're assuming about the data.

Most of hypothesis testing is focused on the null hypothesis, and on possibly rejecting it. There are two main reasons for this. First, there is only one null hypothesis, so we don't have to consider a variety of possibilities. Second, as we've seen, it's very easy to control the error rate when rejecting the null hypothesis. The other side of the coin has to do with the case where the null hypothesis is not true. This case is much more complicated because there are many different ways to reject the null hypothesis, and it would be naive to believe that they'd all behave the same way. When the null hypothesis is not true, we might mistakenly accept the null hypothesis anyway. Just as we might see an unusually high number of heads for a true coin, we might coincidentally see nearly equal numbers of heads and tails with a coin that was actually weighted. The case where the null hypothesis is not true, but we fail to reject it results in a Type II error. Type II error is usually expressed as power, which is 1 minus the probability of making an error. Power is the proportion of the time that you

correctly reject the null hypothesis when it isn't true. Notice that the power of a hypothesis test changes depending on what the alternative is. In other words, it makes sense that if we had a coin that was so heavily weighted that it usually had a probability of 0.8 of getting heads, we'd have more power in detecting the discrepancy from the null distribution than if our coin was only weighted to give a 0.55 probability of heads. Also, unlike Type I error, we can't set a Type II error rate before we start the analysis – the only way to change the Type II error rate is to get more samples or redesign the experiment.

## 11.2 Determining Power

For many years, the only way to answer questions about power required complex mathematical computations, and power was often ignored because sometimes it was too difficult or even impossible to calculate. Even if it was easy to calculate, the fact that power varies depending on the alternative hypothesis would make considering power a much more difficult task than thinking about Type I error. But more recently, statisticians have come to realize that it's often a good idea to estimate power for alternative hypotheses of interest.

Using computers, we can calculate the power of any alternative hypothesis we please by doing the following:

1. Follow the general hypothesis testing procedure outlined above.
2. Repeatedly generate simulated data which corresponds to an alternative hypothesis that you're interested in. For each set of data that you simulate (sometimes referred to as a trial), calculate the statistic, and decide whether to reject the null hypothesis or not.
3. When you're done with all the trials, count how often you (correctly) rejected the null hypothesis, and divide by the total number of trials to obtain the power of the experiment for that specific alternative distribution.

To generate random values from the binomial distribution, to simulate a coin toss, we can use the `rbinom` function. You pass this function the number of samples you'd like, the size of the sample, and the probability. Before diving into the power computation, let's verify that, when the null hypothesis is true, using the rejection region we developed previously will actually result in a Type I error rate of 5%. First, let's simulate a single set of 100 tosses a few times to see if it makes sense

```
> rbinom(1,100,.5)
[1] 45
> rbinom(1,100,.5)
[1] 52
> rbinom(1,100,.5)
[1] 52
> rbinom(1,100,.5)
[1] 46
```

That seems reasonable. Now let's generate 1000 such samples, and count how many of them had less than 40 heads or more than 60 heads:

```
> rr = rbinom(1000,100,.5);sum(rr<40|rr>60)/1000
[1] 0.042
> rr = rbinom(1000,100,.5);sum(rr<40|rr>60)/1000
[1] 0.03
> rr = rbinom(1000,100,.5);sum(rr<40|rr>60)/1000
[1] 0.043
> rr = rbinom(1000,100,.5);sum(rr<40|rr>60)/1000
[1] 0.029
```

The values are a little low, but it's because we've got a discrete distribution and we had to choose the nearest integer value of heads. To see the exact probabilities that the values 40 and 60 represent, we can use the `pbinom` function:

```
> pbinom(40,100,.5)
[1] 0.02844397
> pbinom(60,100,.5)
[1] 0.9823999
```

If we make the rejection region larger, then the probability will become too high:

```
> rr = rbinom(1000,100,.5);sum(rr<41|rr>59)/1000
[1] 0.054
> rr = rbinom(1000,100,.5);sum(rr<41|rr>59)/1000
[1] 0.064
> rr = rbinom(1000,100,.5);sum(rr<41|rr>59)/1000
[1] 0.056
> rr = rbinom(1000,100,.5);sum(rr<41|rr>59)/1000
[1] 0.068
> rr = rbinom(1000,100,.5);sum(rr<41|rr>59)/1000
[1] 0.064
```

So it seems that the idea of using less than 40 or greater than 60 as a rejection region will give us a Type I error as close to 5% as is possible.

Now let's ask a much more interesting question. If I had a coin that really had a 55% chance of heads instead of a 50% chance, and I tossed the coin 100 times, how often would I correctly reject the null hypothesis? In other words, what is the power of the test against an alternative that the actual chances of getting heads is 55%? I can do exactly the same thing:

```
> rr = rbinom(1000,100,.55);sum(rr<40|rr>60)/1000
[1] 0.139
> rr = rbinom(1000,100,.55);sum(rr<40|rr>60)/1000
```

```
[1] 0.112
> rr = rbinom(1000,100,.55);sum(rr<40|rr>60)/1000
[1] 0.134
> rr = rbinom(1000,100,.55);sum(rr<40|rr>60)/1000
[1] 0.133
```

We'd only correctly identify the coin as being untrue in about 13% of the cases. It's better than the 5% we'd expect when the coin was true, but not very impressive. Maybe tossing more coins would help. Let's see what happens to the power when we do 500 tosses instead of just 100. First we need to find the correct rejection region:

```
> qbinom(.975,500,.5)
[1] 272
> qbinom(.025,500,.5)
[1] 228
```

Now we can try the simulation:

```
> rr = rbinom(1000,500,.55);sum(rr<228|rr>272)/1000
[1] 0.595
> rr = rbinom(1000,500,.55);sum(rr<228|rr>272)/1000
[1] 0.587
> rr = rbinom(1000,500,.55);sum(rr<228|rr>272)/1000
[1] 0.577
> rr = rbinom(1000,500,.55);sum(rr<228|rr>272)/1000
[1] 0.61
> rr = rbinom(1000,500,.55);sum(rr<228|rr>272)/1000
[1] 0.611
```

That's quite a bit better. But suppose we wanted to find out how big of a sample size we'd need to be able to detect the 55% coin 95% of the time. Probably the easiest way is to try a bunch of different numbers in the simulation. It's getting complicated enough that we should probably write a function:

```
> coin.power = function(ntoss=100,nsim=1000,prob=.5){
+   lower = qbinom(.025,ntoss,.5)
+   upper = qbinom(.975,ntoss,.5)
+   rr = rbinom(nsim,ntoss,prob)
+   sum(rr < lower | rr > upper) / nsim
+ }
```

We can use `sapply` to call the `coin.power` function for a variety of number of tosses:

```
> ntosses = c(10,100,200,500,600,800,1000,1500,2000,2500)
> res = sapply(ntosses,coin.power,prob=.55)
> names(res) = ntosses
```

```

> res
  10   100   200   500   600   800   1000   1500   2000   2500
0.040 0.135 0.289 0.583 0.697 0.773 0.884 0.967 0.989 0.999

```

Intuitively, if the coin was more dishonest, we'd expect our test to have better power. Let's try the same thing with a coin that's weighted to give 70% heads:

```

> res = sapply(ntosses,coin.power,prob=.70)
> names(res) = ntosses
> res
  10   100   200   500   600   800   1000   1500   2000   2500
0.153 0.986 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000

```

With just 100 tosses, we're pretty much guaranteed to be able to recognize the coin as weighted. What if the coin was just barely weighted, say 51% instead of 50%:

```

> res = sapply(ntosses,coin.power,prob=.51)
> names(res) = ntosses
> res
  10   100   200   500   600   800   1000   1500   2000   2500
0.019 0.035 0.051 0.067 0.071 0.087 0.084 0.129 0.125 0.158

```

Even after 2500 tosses, we'd only have about a 16% chance of detecting the weighted coin.

## 11.3 Probability Distributions

The naming conventions that were used for the binomial probability functions in the previous section are used consistently with all of the different distributions that are supported by R. To summarize:

1. To get an overall idea of how values are distributed, we can examine (usually graphically), the probability density function for a distribution. For example, the familiar bell curve is the density function for a normal distribution. In R, density functions begin with the letter “d”.
2. To find out what value corresponds to a particular probability (like 0.025 and 0.975 in the previous example), we can call the quantile function for the distribution. These functions accept an argument between 0 and 1 (i.e. a probability) and return the value for that distribution where the probability that an observation would be smaller than that value is equal to the probability passed to the function. When you perform a statistical test, choose a significance level, and look up the statistic's value in a table, you're actually using the quantile function. Quantile functions in R begin with the letter “q”.

beta	Beta	binom	Binomial
cauchy	Cauchy	chisq	Chi-squared
exp	Exponential	exp	Exponential
f	F	gamma	Gamma
geom	Geometric	hyper	Hypergeometric
lnorm	Lognormal	logis	Logistic
nbinom	Negative Binomial	norm	Normal
pois	Poisson	signrank	Signed Rank
t	Student's t	unif	Uniform
weibull	Weibull	wilcox	Wilcoxon's Rank Sum

3. If you've got a value for an observation that comes from a particular distribution, and you want to know the probability that observations from the distribution would be less than that value, you can pass the value to a probability function (like `pbinom`) and it will return the probability. In R, these functions begin with the letter "p".
4. When you want to generate random numbers from a distribution, you can call the appropriate random number generator. Random number generators in R begin with the letter "r".

The following table shows the distributions for which these four types of functions are available in R. In each case, precede the name in the table with a d, q, p, or r to get the function you want.

## 11.4 A Note about Random Numbers in R

Some people like to make a distinction between truly random numbers (like those formed from physically tossing dice or counting the number of clicks of a Geiger counter in a given time interval) and the kinds of random numbers that are generated on a computer. (If you think the second example is impractical, take a look at <http://www.fourmilab.ch/hotbits/>. Another source of truly random numbers can be found at <http://www.lavarnd.org/>.)

Some chip manufacturers are developing chips which allow noise in the circuitry (which is truly random) to be measured and used as a source of random numbers. But the programs in R that generate random numbers produce them by applying a reproducible set of rules (a computer program), so they are not really random, and some people refer to them as psuedo-random numbers for that reason. In general, we won't make any such distinction between truly random numbers and the ones we get from the computer.

In R and most other computer programs, all of the random number generators are based on the uniform distribution. By developing programs that are guaranteed to give random numbers with the right distribution if they are provided by a uniform random number generator, only one distribution (the uniform) needs to be studied to insure that it's truly random. The uniform random number generator in R has been extensively tested, so you can have confidence that all the random number generators in R are reliable.

While it's obviously a useful feature that R's default behaviour is to give you a different set of random numbers each time that you call any of the random number generators, sometimes it creates problems or misunderstandings. Because of this, R provides a simple way to set the random number generators to a reproducible state, so that the exact same sequence of random numbers will be generated every time a program is run. By passing an integer to the `set.seed` function, you can insure that the sequence of random numbers that are generated will be the same everytime you run an identical program.

## 11.5 t-tests

One of the most common tests in statistics is the t-test, used to determine whether the means of two groups are equal to each other. The assumption for the test is that both groups are sampled from normal distributions with equal variances. The null hypothesis is that the two means are equal, and the alternative is that they are not. It is known that under the null hypothesis, we can calculate a t-statistic that will follow a t-distribution with  $n_1 + n_2 - 2$  degrees of freedom. There is also a widely used modification of the t-test, known as Welch's t-test that adjusts the number of degrees of freedom when the variances are thought not to be equal to each other. Before we can explore the test much further, we need to find an easy way to calculate the t-statistic.

The function `t.test` is available in R for performing t-tests. Let's test it out on a simple example, using data simulated from a normal distribution.

```
> x = rnorm(10)
> y = rnorm(10)
> t.test(x,y)
```

```
Welch Two Sample t-test

data: x and y
t = 1.4896, df = 15.481, p-value = 0.1564
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.3221869 1.8310421
sample estimates:
mean of x mean of y
0.1944866 -0.5599410
```

Before we can use this function in a simulation, we need to find out how to extract the t-statistic (or some other quantity of interest) from the output of the `t.test` function. For this function, the R help page has a detailed list of what the object returned by the function contains. A general method for a situation like this is to use the `class` and `names` functions to find where the quantity of interest is. In addition, for some hypothesis tests, you may need to pass the object from the hypothesis test to the `summary` function and examine its contents. For `t.test` it's easy to figure out what we want:

```
> ttest = t.test(x,y)
> names(ttest)
[1] "statistic"    "parameter"    "p.value"      "conf.int"      "estimate"
[6] "null.value"   "alternative"  "method"       "data.name"
```

The value we want is named “`statistic`”. To extract it, we can use the dollar sign notation, or double square brackets:

```

> ttest$statistic
      t
1.489560
> ttest[['statistic']]
      t
1.489560

```

Of course, just one value doesn't let us do very much – we need to generate many such statistics before we can look at their properties. In R, the `replicate` function makes this very simple. The first argument to `replicate` is the number of samples you want, and the second argument is an expression (not a function name or definition!) that will generate one of the samples you want. To generate 1000 t-statistics from testing two groups of 10 standard random normal numbers, we can use:

```
> ts = replicate(1000,t.test(rnorm(10),rnorm(10))$statistic)
```

Under the assumptions of normality and equal variance, we're assuming that the statistic will have a t-distribution with  $10 + 10 - 2 = 18$  degrees of freedom. (Each observation contributes a degree of freedom, but we lose two because we have to estimate the mean of each group.) How can we test if that is true?

One way is to plot the theoretical density of the t-statistic we should be seeing, and superimposing the density of our sample on top of it. To get an idea of what range of x values we should use for the theoretical density, we can view the range of our simulated data:

```

> range(ts)
> range(ts)
[1] -4.564359  4.111245

```

Since the distribution is supposed to be symmetric, we'll use a range from -4.5 to 4.5. We can generate equally spaced x-values in this range with `seq`:

```

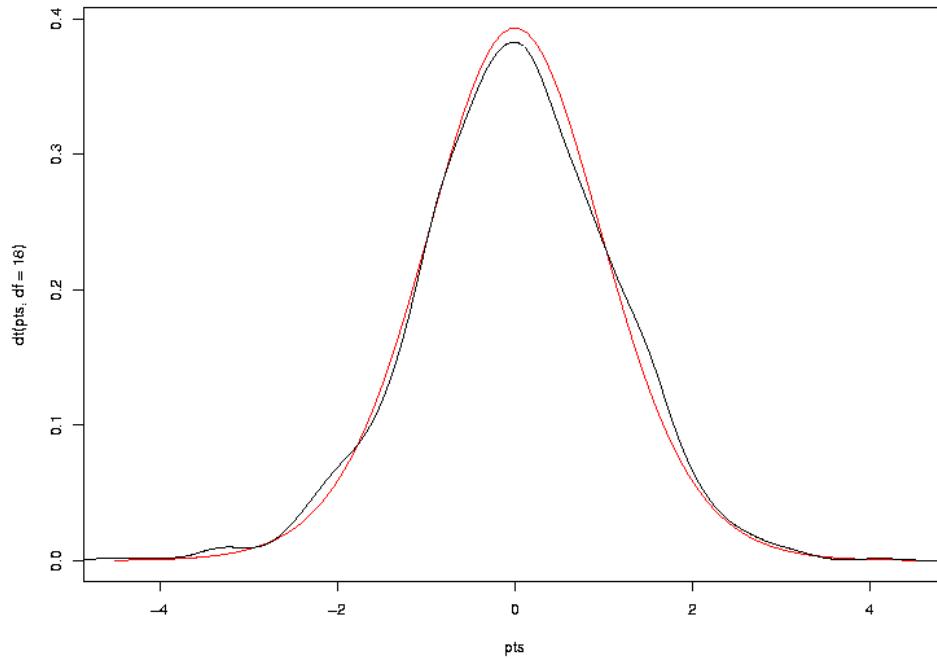
> pts = seq(-4.5,4.5,length=100)
> plot(pts,dt(pts,df=18),col='red',type='l')

```

Now we can add a line to the plot showing the density for our simulated sample:

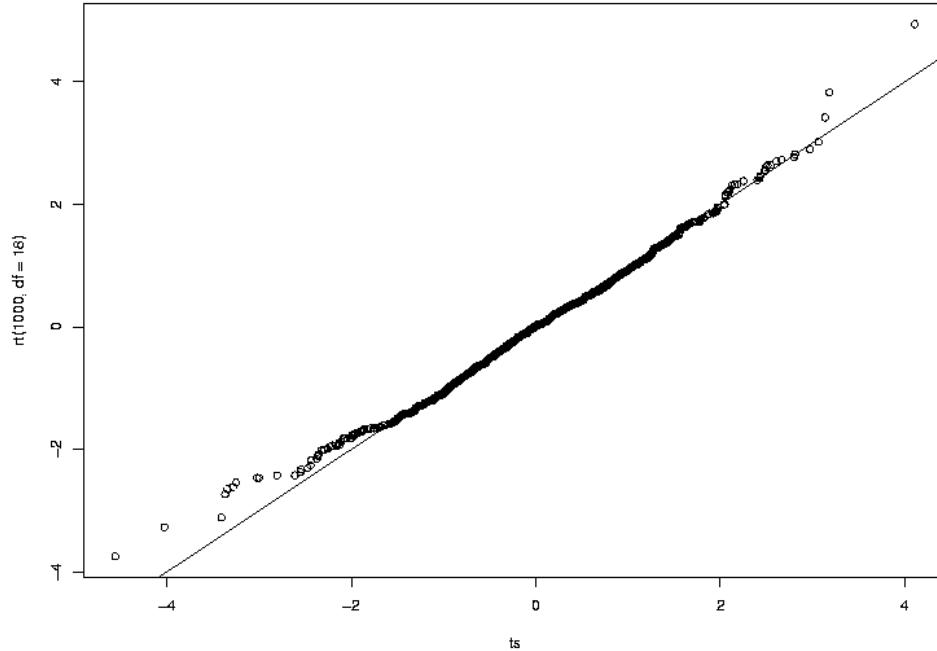
```
> lines(density(ts))
```

The plot appears below.



Another way to compare two densities is with a quantile-quantile plot. In this type of plot, the quantiles of two samples are calculated at a variety of points in the range of 0 to 1, and then are plotted against each other. If the two samples came from the same distribution with the same parameters, we'd see a straight line through the origin with a slope of 1; in other words, we're testing to see if various quantiles of the data are identical in the two samples. If the two samples came from similar distributions, but their parameters were different, we'd still see a straight line, but not through the origin. For this reason, it's very common to draw a straight line through the origin with a slope of 1 on plots like this. We can produce a quantile-quantile plot (or QQ plot as they are commonly known), using the `qqplot` function. To use `qqplot`, pass it two vectors that contain the samples that you want to compare. When comparing to a theoretical distribution, you can pass a random sample from that distribution. Here's a QQ plot for the simulated t-test data:

```
> qqplot(ts, rt(1000,df=18))
> abline(0,1)
```



We can see that the central points of the graph seems to agree fairly well, but there are some discrepancies near the tails (the extreme values on either end of the distribution). The tails of a distribution are the most difficult part to accurately measure, which is unfortunate, since those are often the values that interest us most, that is, the ones which will provide us with enough evidence to reject a null hypothesis. Because the tails of a distribution are so important, another way to test to see if a distribution of a sample follows some hypothesized distribution is to calculate the quantiles of some tail probabilities (using the `quantile` function) and compare them to the theoretical probabilities from the distribution (obtained from the function for that distribution whose first letter is “q”). Here’s such a comparison for our simulated data:

```
> probs = c(.9,.95,.99)
> quantile(ts,probs)
  90%    95%    99%
1.427233 1.704769 2.513755
> qt(probs,df=18)
[1] 1.330391 1.734064 2.552380
```

The quantiles agree fairly well, especially at the .95 and .99 quantiles. Performing more simulations, or using a large sample size for the two groups would probably result in values even closer to what we have theoretically predicted.

One final method for comparing distributions is worth mentioning. We noted previously that one of the assumptions for the t-test is that the variances of the two samples are equal. However, a modification of the t-test known as Welch’s test is said to correct for this problem by estimating the variances, and adjusting the degrees of freedom to use in the test.

This correction is performed by default, but can be shut off by using the `var.equal=TRUE` argument. Let's see how it works:

```
> t.test(x,y)
```

Welch Two Sample t-test

```
data: x and y
t = -0.8103, df = 17.277, p-value = 0.4288
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-1.0012220 0.4450895
sample estimates:
mean of x mean of y
0.2216045 0.4996707
```

```
> t.test(x,y,var.equal=TRUE)
```

Two Sample t-test

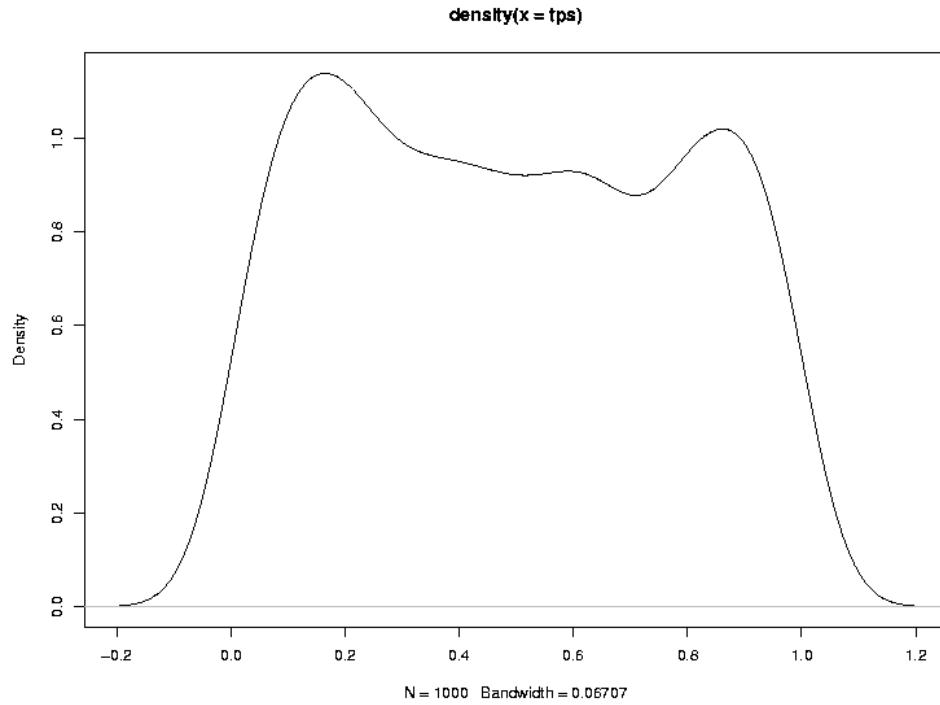
```
data: x and y
t = -0.8103, df = 18, p-value = 0.4284
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.9990520 0.4429196
sample estimates:
mean of x mean of y
0.2216045 0.4996707
```

Since the statistic is the same in both cases, it doesn't matter whether we use the correction or not; either way we'll see identical results when we compare the two methods using the techniques we've already described. Since the degree of freedom correction changes depending on the data, we can't simply perform the simulation and compare it to a different number of degrees of freedom. The other thing that changes when we apply the correction is the p-value that we would use to decide if there's enough evidence to reject the null hypothesis. What is the behaviour of the p-values? While not necessarily immediately obvious, under the null hypothesis, the p-values for any statistical test should form a uniform distribution between 0 and 1; that is, any value in the interval 0 to 1 is just as likely to occur as any other value. For a uniform distribution, the quantile function is just the identity function. A value of .5 is greater than 50% of the data; a value of .95 is greater than 95% of the data. As a quick check of this notion, let's look at the density of probability values when the null hypothesis is true:

```
> tps = replicate(1000,t.test(rnorm(10),rnorm(10))$p.value)
```

```
> plot(density(tps))
```

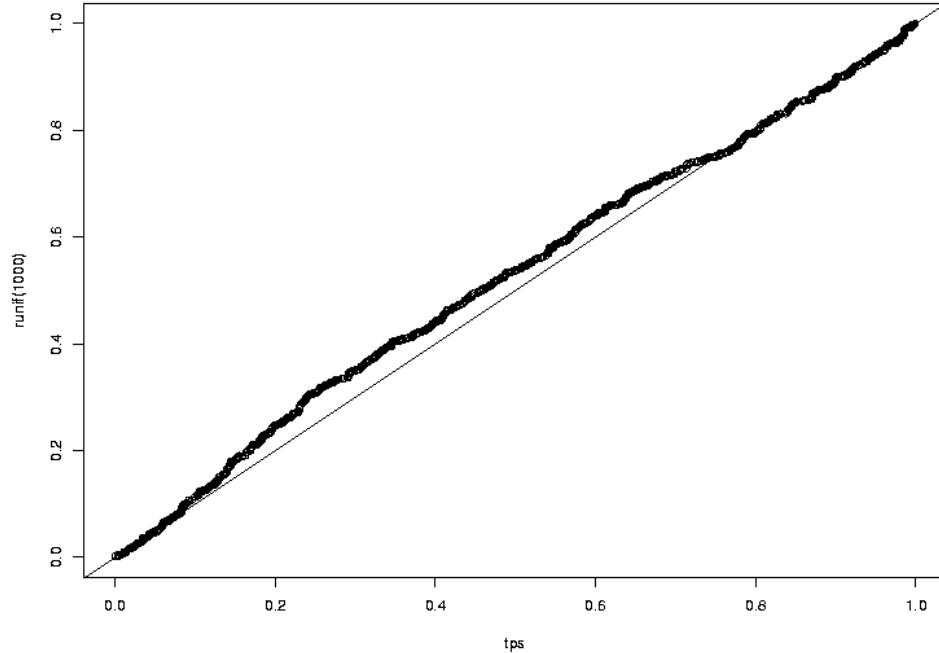
The graph appears below.



Another way to check to see if the probabilities follow a uniform distribution is with a QQ plot:

```
> qqplot(tps,runif(1000))
> abline(0,1)
```

The graph appears below.



The idea that the probabilities follow a uniform distribution seems reasonable.

Now, let's look at some of the quantiles of the p-values when we force the `t.test` function to use `var.equal=TRUE`:

```
> tps = replicate(1000,t.test(rnorm(10),rnorm(10),var.equal=TRUE)$p.value)
> probs = c(.5,.7,.9,.95,.99)
> quantile(tps,probs)
  50%      70%      90%      95%      99%
0.4873799 0.7094591 0.9043601 0.9501658 0.9927435
```

The agreement actually looks very good. What about when we let `t.test` decide whether to make the correction or not?

```
> tps = replicate(1000,t.test(rnorm(10),rnorm(10))$p.value)
> quantile(tps,probs)
  50%      70%      90%      95%      99%
0.4932319 0.7084562 0.9036533 0.9518775 0.9889234
```

There's not that much of a difference, but, of course, the variances in this example were equal. How does the correction work when the variances are not equal?

```
> tps = replicate(1000,t.test(rnorm(10),rnorm(10,sd=5),var.equal=TRUE)$p.value)
> quantile(tps,probs)
  50%      70%      90%      95%      99%
0.5221698 0.6926466 0.8859266 0.9490947 0.9935562
> tps = replicate(1000,t.test(rnorm(10),rnorm(10,sd=5))$p.value)
> quantile(tps,probs)
```

50%	70%	90%	95%	99%
0.4880855	0.7049834	0.8973062	0.9494358	0.9907219

There is an improvement, but not so dramatic.

## 11.6 Power of the t-test

Of course, all of this is concerned with the null hypothesis. Now let's start to investigate the power of the t-test. With a sample size of 10, we obviously aren't going to expect truly great performance, so let's consider a case that's not too subtle. When we don't specify a standard deviation for `rnorm` it uses a standard deviation of 1. That means about 68% of the data will fall in the range of -1 to 1. Suppose we have a difference in means equal to just one standard deviation, and we want to calculate the power for detecting that difference. We can follow the same procedure as the coin tossing experiment: specify an alpha level, calculate the rejection region, simulate data under the alternative hypothesis, and see how many times we'd reject the null hypothesis. As in the coin toss example, a function will make things much easier:

```
t.power = function(nsamp=c(10,10),nsim=1000,means=c(0,0),sds=c(1,1)){
  lower = qt(.025,df=sum(nsamp) - 2)
  upper = qt(.975,df=sum(nsamp) - 2)
  ts = replicate(nsim,
    t.test(rnorm(nsamp[1],mean=means[1],sd=sds[1]),
           rnorm(nsamp[2],mean=means[2],sd=sds[2]))$statistic)

  sum(ts < lower | ts > upper) / nsim
}
```

Let's try it with our simple example:

```
> t.power(means=c(0,1))
[1] 0.555
```

Not bad for a sample size of 10!

Of course, if the differences in means are smaller, it's going to be harder to reject the null hypothesis:

```
> t.power(means=c(0,.3))
[1] 0.104
```

How large a sample size would we need to detect that difference of .3 with 95% power?

```

> samps = c(100,200,300,400,500)
> res = sapply(samps,function(n)t.power(means=c(0,.3),nsamp=c(n,n)))
> names(res) = samps
> res
  100   200   300   400   500
0.567 0.841 0.947 0.992 0.999

```

It would take over 300 samples in each group to be able to detect such a difference.

Now we can return to the issue of unequal variances. We saw that Welch's adjustment to the degrees of freedom helped a little bit under the null hypothesis. Now let's see if the power of the test is improved using Welch's test when the variances are unequal. To do this, we'll need to modify our `t.power` function a little:

```

t.power1 = function(nsamp=c(10,10),nsim=1000,means=c(0,0),sds=c(1,1),var.equal=TRUE){
  tps = replicate(nsim,
    t.test(rnorm(nsamp[1],mean=means[1],sd=sds[1]),
           rnorm(nsamp[2],mean=means[2],sd=sds[2]))$p.value)

  sum(tps < .025 | tps > .975) / nsim
}

```

Since I set `var.equal=TRUE` by default, Welch's adjustment will not be used unless we specify `var.equal=FALSE`. Let's see what the power is for a sample of size 10, assuming the mean of one of the groups is 1, and its standard deviation is 2, while the other group is left at the default of mean=0 and sd=1:

```

> t.power1(nsim=10000,sds=c(1,2),mean=c(1,2))
[1] 0.1767
> t.power1(nsim=10000,sds=c(1,2),mean=c(1,2),var.equal=FALSE)
[1] 0.1833

```

There does seem to be an improvement, but not so dramatic.

We can look at the same thing for a variety of sample sizes:

```

> res1 = sapply(sizes,function(n)t.power1(nsim=10000,sds=c(1,2),
+                           mean=c(1,2),nsamp=c(n,n)))
> names(res1) = sizes
> res1
  10      20      50     100
0.1792 0.3723 0.8044 0.9830
> res2 = sapply(sizes,function(n)t.power1(nsim=10000,sds=c(1,2),
+                           mean=c(1,2),nsamp=c(n,n),var.equal=FALSE))
> names(res2) = sizes
> res2
  10      20      50     100
0.1853 0.3741 0.8188 0.9868

```

# Chapter 12

## Graphical User Interfaces (GUIs)

## 12.1 Graphical User Interfaces

While using R through the command line provides a great deal of flexibility, there are times when a simpler interface would be useful. For example, you may develop an R program that performs a complex analysis, and you'd like others to be able to run the program by, say just specifying a file to be read, and choosing a couple of options through checkboxes or a drop down list. There may be a set of operations that your routinely do when first studying a data set, and you'd like to be presented with a list of available data, and just click on the one that you want.

There are several different options available in R to produce graphical user interfaces (GUIs); we are going to focus on a simple, portable one based on the Tcl/Tk library originally developed by John Ousterhout when he was in the Computer Science department here at Berkeley. Tcl is a scripting language (like perl or python), and Tk is a graphical toolkit designed to work with that language. Since the Tk toolkit is portable, easy-to-use, and provides most of the basic elements needed to create GUIs, interfaces to it are available in many languages, so if you understand the basics of Tk GUIs, you may see the same techniques used later on in your career by some other program. To use the Tcl/Tk library in R, you must load the library with the `library(tcltk)` command. This library is part of the base R distribution, so you don't have to download anything in order to use it (although you may choose to download some optional components later on.)

If you look at the help pages for the functions in the `tcltk` library, you'll probably be surprised to see how little information or guidance there is. However, once you work with the library for a while, you'll get accustomed to the way Tk functions are mapped to R functions, and you'll be able to use any documentation about Tk that you can find on the web or elsewhere to get more information. There are a wide variety of sample programs at [http://www.sciviews.org/\\_rgui/tcltk/](http://www.sciviews.org/_rgui/tcltk/), but they use a different geometry method and some require external packages. I'll try to provide examples here to help you get started, but don't hesitate to look around the web for more examples and ideas.

The process of creating a GUI can be broken down into two parts: choosing the elements (often referred to as widgets) that will make up the GUI, and arranging for the elements to be placed in the appropriate positions and displayed. This second step is sometimes known as geometry management. Tk actually offers several different geometry management methods; we are going to focus on one known as packing. In the packing geometry model, the GUI is considered a collection of separate frames, each containing some of the widgets that make up the final GUI. The widgets are loaded into the frames in a carefully chosen order, and they "spread out" to fill the frame. When you pack a widget into a frame, you can specify the side to which to pack it, choosing among `left`, `right`, `top` or `bottom`. With a little practice and experimentation, you can usually get the widgets to line up the way you want them to.

Most of the widgets that are available through the Tk toolbox should look familiar, as common applications like email clients and web browser use these same elements in their GUIs. Some examples of available widgets are:

- text entry fields

- drop down menus
- check buttons
- radio buttons
- file selection browsers
- sliding scales

Since the Tk toolbox is not integrated entirely into R, it's necessary to use special techniques to get information out of the Tcl/Tk environment and into R. Many of the widgets, for example a text entry widget, will need a Tcl variable associated with them to hold necessary information. All of the variables that will be captured by the GUI will be character variables. The `tclVar` function can be used to create an empty tcl variable which can then be used in call to functions in the `tcltk` library as follows:

```
myvar = tclVar('')
```

Later, when you need to get the information that was put into the Tk variable back into the R environment, you can use the `tclvalue` function:

```
rmyvar = tclvalue(myvar)
```

Another aspect of GUI programming that is different from regular programs has to do with associating actions with certain GUI elements. An obvious example is a button, which when pressed needs to do something. Widgets like this will have an argument, usually called `command=` which will accept the name of a function that will be called when the button (or other action) is recognized. Such functions, often called callbacks, should be written using the `...` notation for their argument lists, and should not have any returned values. It sometimes becomes necessary to “remember” things over the course of a GUI’s execution. For example, you may want to count the number of times a button is pressed, or display a different message based on the value of a previous response. Since all of the actions which the GUI will initiate are carried out in functions, any modifications to variables that are made in those functions will be lost when the function returns. While not usually a good practice, you can use the special assignment operator `<<-` inside a function to modify global variables, which will retain their values after the function exits.

To make some of these concepts clearer, let's consider the construction of a GUI to accept input to the `coin.power` program that we developed earlier. Recall that the program accepts the number of simulations, number of coin tosses, probability of getting heads for the coin, and returns the power. We want to make a GUI with three entries, one for each of the parameters, a place to display the answer, and buttons to run the simulation or close the window. The first step in creating a GUI with the `tcltk` library in R is loading the library and calling the `tktoplevel` function, which creates an initial window into which we will pack the frames containing the widgets. Since we can control the order of packing into individual frames, each set of related elements can be put into a separate frame. For this

example, we'll create a separate frame for each entry and its associated label, as well as a frame to hold those entries and another frame for the two buttons; the area for the answer can be packed into the toplevel frame.

In the code that follows, you'll notice that usually widgets are packed directly into their enclosing frames, but the label widget that will hold the answer is assigned to a variable before being packed. If you'll need to communicate with the widget at some point in the running of the program, it's vital to store the widget in a variable. In this example, we'll create a blank label, and use the `tkconfigure` function when we have an answer to change what the label displays. Here's the code to create the GUI:

```

require(tcltk)
coin.power = function(ntoss=100,nsim=1000,prob=.5){
    lower = qbinom(.025,ntoss,.5)
    upper = qbinom(.975,ntoss,.5)
    rr = rbinom(nsim,ntoss,prob)
    sum(rr < lower | rr > upper) / nsim
}

destroy = function(...)tkdestroy(base)

tst = function(...){
    nsim = as.numeric(tclvalue(nsim_))
    ntoss = as.numeric(tclvalue(ntoss_))
    prob = as.numeric(tclvalue(prob_))
    res = coin.power(ntoss=as.numeric(ntoss),nsim=as.numeric(nsim),prob=as.numeric(prob))
    tkconfigure(ans,text=paste(res))
}

base = tktoplevel()
tkwm.title(base,'Coin Toss')

# create a frame to hold the three entries
nfrm = tkframe(base)

# create tcl variables to associate with the
# entry fields -- to include starting values
# replace the '' with the desired value
nsim_ = tclVar('')
ntoss_ = tclVar('')
prob_ = tclVar('')

f1 = tkframe(nfrm)
tkpack(tklabel(f1,text='Nsim',width=8),side='left')

```

```

tkpack(tkentry(f1,width=10,textvariable=nsim_),side='left')

f2 = tkframe(nfrm)
tkpack(tklabel(f2,text='Ntoss',width=8),side='left')
tkpack(tkentry(f2,width=10,textvariable=ntoss_),side='left')

f3 = tkframe(nfrm)
tkpack(tklabel(f3,text='Prob',width=8),side='left')
tkpack(tkentry(f3,width=10,textvariable=prob_),side='left')

# the widgets were packed in the frames, but the frames have
# not yet been packed. Remember that nothing can actually be
# displayed unless it's been packed.
tkpack(f1,side='top')
tkpack(f2,side='top')
tkpack(f3,side='top')

tkpack(nfrm)

# now we can repeat the process for the label to hold the
# answer, and the frame containing the buttons
ans = tklabel(base,text=' ')
tkpack(ans,side='top')

bfrm = tkframe(base)
tkpack(tkbutton(bfrm,text='Run',command=tst),side='left')
tkpack(tkbutton(bfrm,text='Quit',command=destroy),side='right')

tkpack(bfrm,side='bottom')

```

When we run the program (for example by using the `source` command inside of R), we see the left hand image; after making some entries and hitting the Run button, we see the image on the right.



## 12.2 Making a Calculator

The basic idea behind writing a calculator in R is very simple: create a set of buttons which will build up a text string that represents the calculation we wish to perform, and then use the `eval` and `parse` functions in R to perform the calculation. These two functions allow us to evaluate a character string as if we typed it into the R console. For example, consider the string "29\*43". We could calculate the value of that expression as follows:

```
> txt = "29*43"
> eval(parse(text=txt))
[1] 1247
```

Using this as the core of the calculator, we can set up rows of buttons arranged in the usual calculator fashion, each of which will add a character to a character string through the `command=` option, and then call `parse` and `eval` to do the calculation when the “=” key is pressed. We need to use a few of the concepts we’ve already seen to make the calculator work. First, we will need to create a global variable that represents the input to the calculator, and have each of the keys add its input using the “<<-” operator. To change the value of the calculator’s display, we’ll use the `tkconfigure` function as in the previous example.

We could define the callback functions separately for each key, but most of the keys will do exactly the same thing – add a character to the string that we’ll eventually calculate. The only difference is that different keys will naturally add different characters. So it would make sense to write a function to generate the different callbacks. Let’s call the calculator’s input `calcinp`. We initialize `calcinp` to an empty string, and then define a function which will take a symbol and add it to the calculator’s input:

```
calcinp = ''
```

```

mkput = function(sym){
  function(...){
    calcinp <- paste(calcinp,sym,sep=' ')
    tkconfigure(display,text=calcinp)
  }
}

```

Notice that we're referring to an object called `display`, even though we haven't defined it yet. R uses a technique called lazy evaluation which means that an object doesn't have to exist when we use it in a function definition – it only needs to be available when the function is actually called. Keeping this in mind, we can define two more functions, one to clear the calculator's display and one to do the actual calculation:

```

clearit = function(...){
  tkconfigure(display,text=' ')
  calcinp <- ''
}

docalc = function(...){
  result = try(eval(parse(text=calcinp)))
  if(class(result) == 'try-error') calcinp <- 'Error' else calcinp <- result
  tkconfigure(display,text=calcinp)
  calcinp <- ''
}

```

A new concept that's introduced in the `docalc` function is the `try` function. This function allows us to try any operation in R, without worrying that an error will stop our program. If there's an error when we pass an R statement to the `try` function, it will return an object of class `try-error`. Thus we can decide what to do when an error occurs by examining the class of the returned value.

Now we can build the calculator itself. The approach is to first pack the label which will serve as the calculator's display, then to create a separate frame for each row of keys, packing them as they are created:

```

base = tktoplevel()
tkwm.title(base,'Calculator')

display = tklabel(base,justify='right')
tkpack(display,side='top')
row1 = tkframe(base)
tkpack(tkbutton(row1,text='7',command=mkput('7'),width=3),side='left')
tkpack(tkbutton(row1,text='8',command=mkput('8'),width=3),side='left')
tkpack(tkbutton(row1,text='9',command=mkput('9'),width=3),side='left')

```

```

tkpack(tkbutton(row1,text='+',command=mkput('+'),width=3),side='left')
tkpack(row1,side='top')

row2 = tkframe(base)
tkpack(tkbutton(row2,text='4',command=mkput('4'),width=3),side='left')
tkpack(tkbutton(row2,text='5',command=mkput('5'),width=3),side='left')
tkpack(tkbutton(row2,text='6',command=mkput('6'),width=3),side='left')
tkpack(tkbutton(row2,text='-',command=mkput('-'),width=3),side='left')
tkpack(row2,side='top')

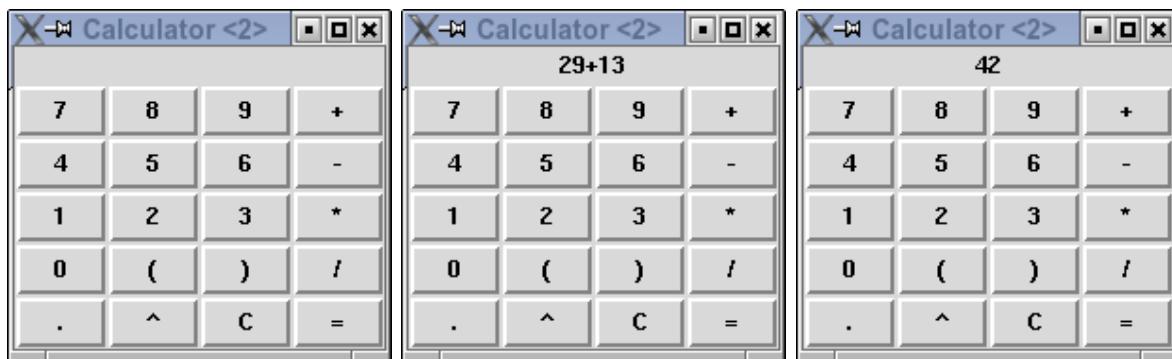
row3 = tkframe(base)
tkpack(tkbutton(row3,text='1',command=mkput('1'),width=3),side='left')
tkpack(tkbutton(row3,text='2',command=mkput('2'),width=3),side='left')
tkpack(tkbutton(row3,text='3',command=mkput('3'),width=3),side='left')
tkpack(tkbutton(row3,text='*',command=mkput('*'),width=3),side='left')
tkpack(row3,side='top')

row4 = tkframe(base)
tkpack(tkbutton(row4,text='0',command=mkput('0'),width=3),side='left')
tkpack(tkbutton(row4,text='(',command=mkput('('),width=3),side='left')
tkpack(tkbutton(row4,text=')',command=mkput(')'),width=3),side='left')
tkpack(tkbutton(row4,text='/',command=mkput('/'),width=3),side='left')
tkpack(row4,side='top')

row5 = tkframe(base)
tkpack(tkbutton(row5,text='.',command=mkput('.'),width=3),side='left')
tkpack(tkbutton(row5,text='^',command=mkput('^'),width=3),side='left')
tkpack(tkbutton(row5,text='C',command=clearit,width=3),side='left')
tkpack(tkbutton(row5,text='=',command=docalc,width=3),side='left')
tkpack(row5,side='top')

```

Here's what the calculator looks like in several stages of a computation:



## 12.3 Improving the Calculator

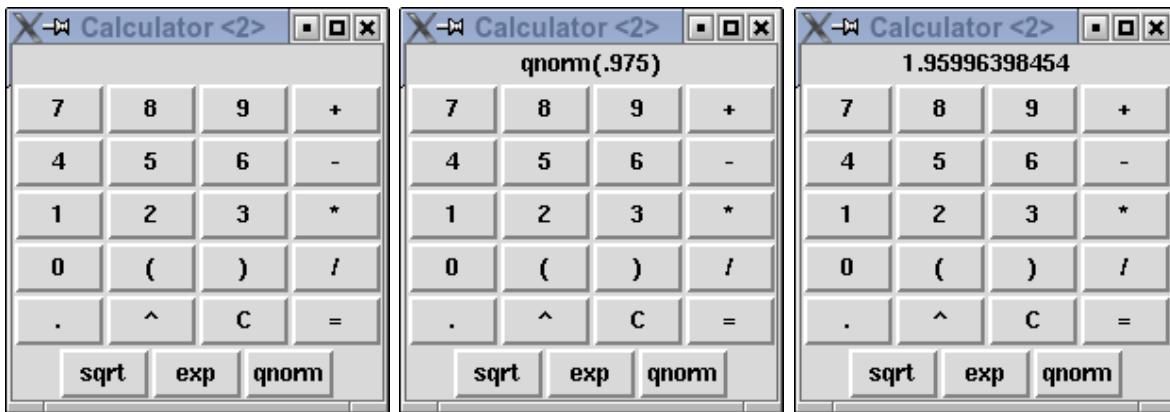
While the calculator is capable of doing basic operations, it's easy to incorporate any function in R which takes a single argument, by creating a callback using this function:

```
mkfun = function(fun){
  function(...){
    calcinp <- paste(fun, '(', calcinp, ')', sep=' ')
    tkconfigure(display, text=calcinp)
  }
}
```

So, for example, we could add the `sqrt`, `exp`, and `qnorm` functions to our calculator like this:

```
row6 = tkframe(base)
tkpack(tkbutton(row6, text='sqrt', command=mkfun('sqrt'), width=3), side='left')
tkpack(tkbutton(row6, text='exp', command=mkfun('exp'), width=3), side='left')
tkpack(tkbutton(row6, text='qnorm', command=mkfun('qnorm'), width=3), side='left')
tkpack(row6, side='top')
```

Here are screenshots of a `qnorm` calculation:



## 12.4 Appearance of Widgets

The previous example was a very minimalistic approach, with no concern with the actual appearance of the final product. One simple way to improve the appearance is to add a little space around the widgets when we pack them. The `tkpack` function accepts two arguments, `padx=` and `pady=` to control the amount of space around the widgets when they are packed. Each of these arguments takes a scalar or a vector of length 2. When a scalar is given, that much space (in pixels) is added on either side of the widget in the specified dimension. When a list is given the first number is the amount of space to the left (`padx`) or the top (`pady`),

and the second number is the amount of space to the right (`padx`) or the bottom (`pady`). Like most aspects of GUI design, experimentation is often necessary to find the best solution. The following code produces a version of the coin power GUI with additional padding:

```
base = tk.toplevel()
tkwm.title(base, 'Coin Toss')

nfrm = tkframe(base)

nsim_ = tclVar('')
ntoss_ = tclVar('')
prob_ = tclVar('')

f1 = tkframe(nfrm)
tkpack(tklabel(f1, text='Nsim', width=8), side='left', pady=c(5,10))
tkpack(tkentry(f1, width=10, textvariable=nsim_), side='left', padx=c(0,20), pady=c(5,10))

f2 = tkframe(nfrm)
tkpack(tklabel(f2, text='Ntoss', width=8), side='left', pady=c(5,10))
tkpack(tkentry(f2, width=10, textvariable=ntoss_), side='left', padx=c(0,20), pady=c(5,10))

f3 = tkframe(nfrm)
tkpack(tklabel(f3, text='Prob', width=8), side='left', pady=c(5,10))
tkpack(tkentry(f3, width=10, textvariable=prob_), side='left', padx=c(0,20), pady=c(5,10))

tkpack(f1, side='top')
tkpack(f2, side='top')
tkpack(f3, side='top')

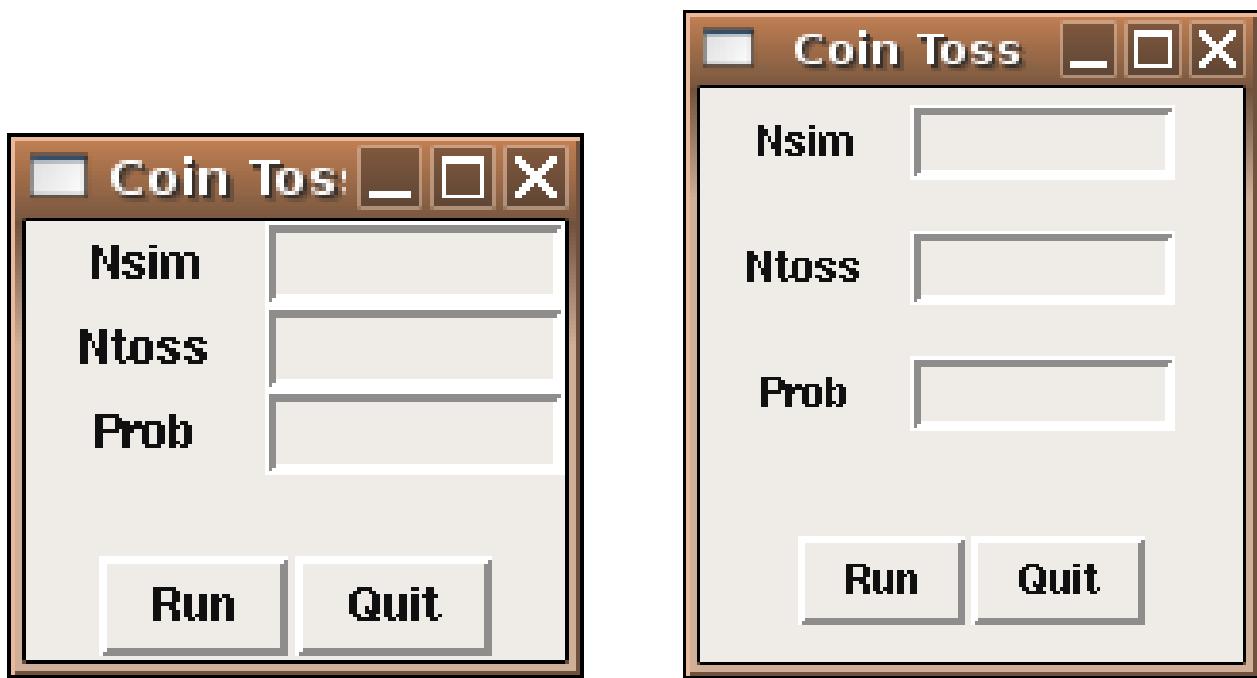
tkpack(nfrm)

ans = tklabel(base, text=' ')
tkpack(ans, side='top')

bfrm = tkframe(base)
tkpack(tkbutton(bfrm, text='Run', command=tst), side='left')
tkpack(tkbutton(bfrm, text='Quit', command=destroy), side='right')

tkpack(bfrm, side='bottom', pady=c(0,10))
```

Here are side-by-side pictures showing the difference between the two versions:



Another way to modify a GUI's appearance is through three-dimensional effects. These are controlled by an argument called `relief`=, which can take values of "flat", "groove", "raised", "ridge", "solid", or "sunken". The following code produces a (non-functioning) GUI showing the different relief styles applied to labels and buttons:

```
require(tcltk)
types = c('flat','groove','raised','ridge','solid','sunken')

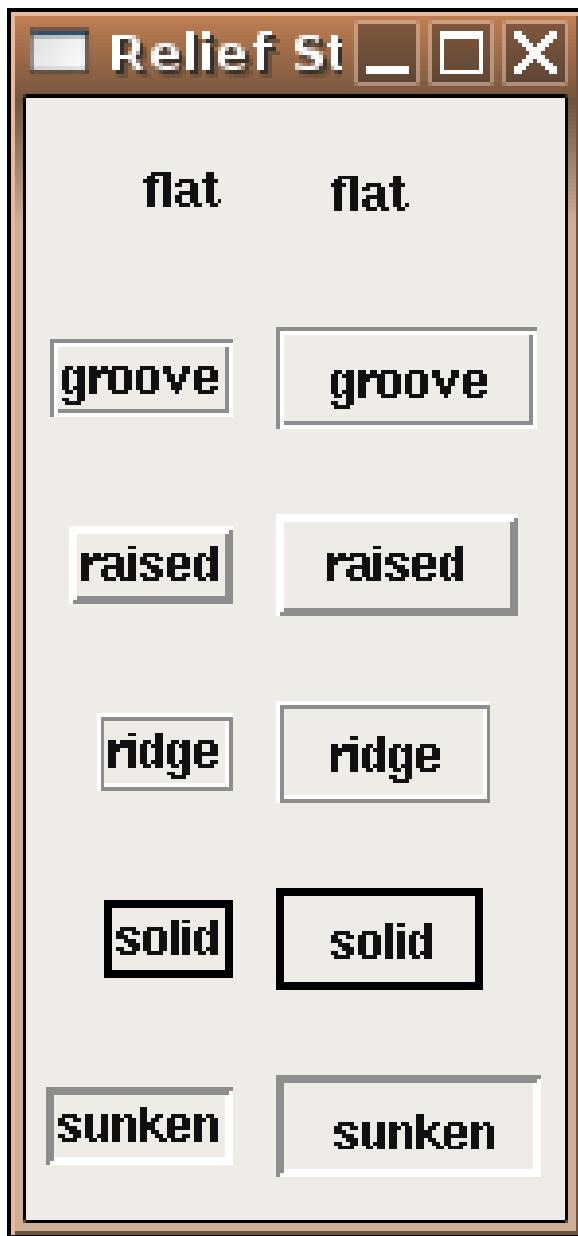
base = tkoplevel()
tkwm.title(base,'Relief Styles')

frms = list()

mkframe = function(type){
  fr = tkframe(base)
  tkpack(tklabel(fr,text=type,relief=type),side='left',padx=5)
  tkpack(tkbutton(fr,text=type,relief=type),side='left',padx=5)
  tkpack(fr,side='top',pady=10)
  fr
}

sapply(types,mkframe)
```

Here's a picture of how it looks:



## 12.5 Fonts and Colors

Several arguments can be used to modify color. The `background=` argument controls the overall color, while the `foreground=` argument controls the color of text appearing in the widget. (Remember that things like this can be changed while the GUI is running using the `tkconfigure` command. Each of these arguments accepts colors spelled out (like "lightblue", "red", or "green") as well as web-style hexadecimal values (like "#add8e6", "#ff0000" or "#00ff00").

To change from the default font (bold Helvetica), you must first create a Tk font, using

the `tkfont.create` function. This function takes a number of arguments, but only the font family is required. The available font families will vary from platform to platform; the `as.character(tkfont.families())` command will display a list of the available fonts. Some of the other possible arguments to `tkfont.create` include:

1. `size=`, the size of the font in points.
2. `weight=`, either "normal" or "bold"
3. `slant=`, either "roman" or "italic"
4. `underline=`, either TRUE or FALSE
5. `overstrike=`, either TRUE or FALSE

As mentioned earlier, colors can be specified as either names or web-style hexadecimal values. Generally, the argument to change the color of a widget is `background=`; the argument to change the color of any text appearing in the widget is `foreground=`. If these choices don't do what you expect, you may need to check the widget-specific documentation.

Here's a simple program that chooses some randomly selected colors and fonts, and displays them as labels.

```
require(tcltk)

allfonts = as.character(tkfont.families())
somefonts = sample(allfonts,9)

somecolors = c('red','blue','green','yellow','lightblue','tan',
'darkred','lightgreen','white')

txt = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

base = tkoplevel()
tkwm.title(base,'Fonts')
mkfonts = function(font,color){
  thefont = tkfont.create(family=font,size=14)
  tkpack(tklabel(base,text=paste(font,':',txt),
    font=thefont,background=color),
    side='top')
}

mapply(mkfonts,somefonts,somecolors)
```

Here's how it looks:



Note that only the specific widget for which `background=` was set changes – if you want to change the background for the entire GUI, you'll probably have to pass the `background=` argument to every widget you use.

## 12.6 Plotting

When developing a TK-based GUI for plotting, there are two possibilities available. The first, which is the only choice if you need to interact with the graph using `locator` or `identify`, is to create a GUI with the controls for the plot, and to let R open a normal plotting window. The second option utilizes the `tkrplot` library, available from CRAN, to create a label widget with an image of the plot; this label can be placed anywhere that any other widget can be placed.

To illustrate the use of a GUI with plotting, consider the problem of mixtures of normal distributions. Many times a sample will actually contain two separate distributions, even though it will be treated as a single distribution. Obviously data like this will create problems when we try to analyze it, so it's important to be able to recognize such data, using, say a density plot. To get an idea of how the density of mixtures of distributions would look, we can create a GUI using a scale or slider widget that will allow us to control the fraction of each of two distributions that will be combined. Many times the first step in making a GUI is writing the function that the GUI will eventually call to actually do the work. This helps to identify the parameters that need to be supplied to the function so that the GUI can be designed in such a way to get all the necessary information. For this case, we'll want to specify the mean and standard deviation of the first distribution, the mean and standard deviation of the second distribution, and the fraction of the second distribution to use. (The fraction for the first distribution will just be 1 minus the fraction for the first distribution.) Here's such a function:

```
genplot = function(m1,s1,m2,s2,frac,n=100){  
  dat = c(rnorm((1-frac)*n,m1,s1),rnorm(frac*n,m2,s2))  
  plot(density(dat),type='l',main='Density of Mixtures')  
}
```

Now we can create the interface. We'll make three frames: the first will accept the mean and standard deviation for the first distribution, the second will have the mean and standard deviation for the second distribution, and the third will have the slider to determine the fraction of each distribution to use. Recall that we need to create `tcl` variables, and then convert them into R variables before we can call our function, so I'll use an intermediate function which will do the translations and then call `genplot` as a callback. Although there is a fair amount of code, most of it is very similar.

```
require(tcltk)  
  
doplot = function(...){  
  m1 = as.numeric(tclvalue(mean1))  
  m2 = as.numeric(tclvalue(mean2))  
  s1 = as.numeric(tclvalue(sd1))  
  s2 = as.numeric(tclvalue(sd2))  
  fr = as.numeric(tclvalue(frac))
```

```

genplot(m1,s1,m2,s2,fr,n=100)
}

base = tk.toplevel()
tkwm.title(base,'Mixtures')
mainfrm = tkframe(base)

mean1 = tclVar(10)
mean2 = tclVar(10)
sd1   = tclVar(1)
sd2   = tclVar(1)
frac  = tclVar(.5)

m1 = tkframe(mainfrm)
tkpack(tklabel(m1,text='Mean1, SD1',width=10),side='left')
tkpack(tkentry(m1,width=5,textvariable=mean1),side='left')
tkpack(tkentry(m1,width=5,textvariable=sd1),side='left')
tkpack(m1,side='top')

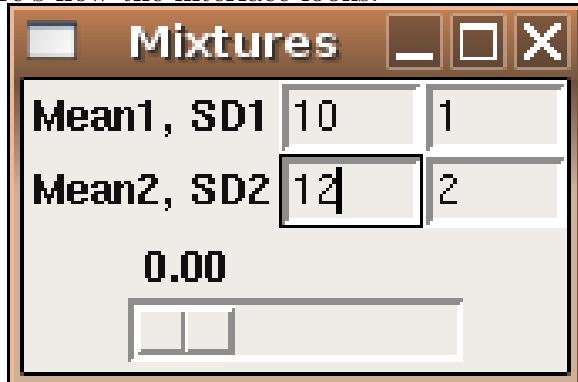
m2 = tkframe(mainfrm)
tkpack(tklabel(m2,text='Mean2, SD2',width=10),side='left')
tkpack(tkentry(m2,width=5,textvariable=mean2),side='left')
tkpack(tkentry(m2,width=5,textvariable=sd2),side='left')
tkpack(m2,side='top')

m3 = tkframe(mainfrm)
tkpack(tkscale(m3,command=doplot,from=0,to=1,showvalue=TRUE,
               variable=frac,resolution=.01,orient='horiz'))
tkpack(m3,side='top')

tkpack(mainfrm)

```

Here's how the interface looks:



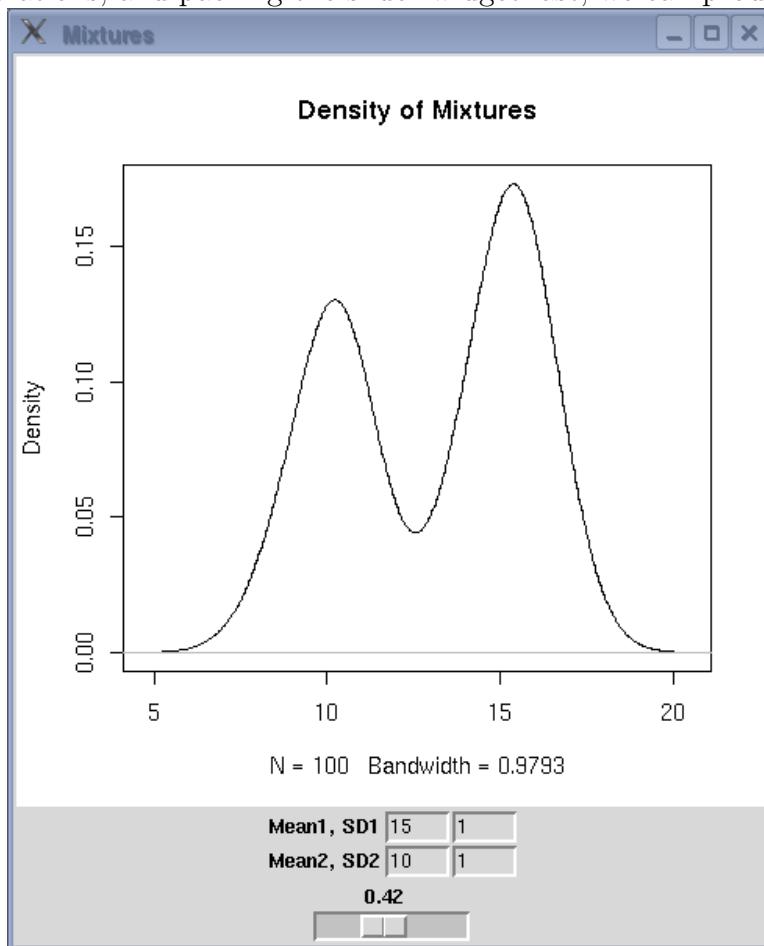
To produce the same sort of GUI, but with the plot in the same frame as the slider, we can use the `tkrplot` library. To place the plot in the same frame as the slider, we must first create a `tkrplot` widget, using the `tkrplot` function. After loading the `tkrplot` library, we call this function with two arguments; the frame in which the plot is to be displayed, and a callback function (using `...` as the only argument) that will draw the desired plot. In this example, we can use the same function (`doplot`) as in the standalone version:

```
img = tkrplot(mainfrm,doplot)
```

Since the `tkrplot` widget works by displaying an image of the plot, the only way to change the plot is to change this image, which is exactly what the `tkrreplot` function does. The only argument to `tkrreplot` is the `tkrplot` widget that will need to be redrawn. Thus, the slider can be constructed with the following statements:

```
scalefunc = function(...)tkrreplot(img)
s = tkScale(mainfrm,command=scalefunc,from=0,to=1,showvalue=TRUE,
            variable='frac',resolution=.01,orient='horiz')
```

By packing the `tkrplot` object first, followed by the frames for the mean and standard deviations, and packing the slider widget last, we can produce the GUI shown below:



The complete code for this GUI is as follows:

```
require(tcltk)
require(tkrplot)

genplot = function(m1,s1,m2,s2,frac,n=100){
    dat = c(rnorm((1-frac)*n,m1,s1),rnorm(frac*n,m2,s2))
    plot(density(dat),type='l',main='Density of Mixtures')
}

doplot = function(...){
    m1 = as.numeric(tclvalue(mean1))
    m2 = as.numeric(tclvalue(mean2))
    s1 = as.numeric(tclvalue(sd1))
    s2 = as.numeric(tclvalue(sd2))
    fr = as.numeric(tclvalue(frac))
    genplot(m1,s1,m2,s2,fr,n=100)
}

base = tktoplevel()
tkwm.title(base,'Mixtures')
mainfrm = tkframe(base)

mean1 = tclVar(10)
mean2 = tclVar(10)
sd1   = tclVar(1)
sd2   = tclVar(1)
frac  = tclVar(.5)

img = tkrplot(mainfrm,doplot)
scalefunc = function(...)tkrreplot(img)
s = tkscale(mainfrm,command=scalefunc,from=0,to=1,showvalue=TRUE,
            variable=frac,resolution=.01,orient='horiz')
tkpack(img)

m1 = tkframe(mainfrm)
tkpack(tklabel(m1,text='Mean1, SD1',width=10),side='left')
tkpack(tkentry(m1,width=5,textvariable=mean1),side='left')
tkpack(tkentry(m1,width=5,textvariable=sd1),side='left')
tkpack(m1,side='top')
m2 = tkframe(mainfrm)
tkpack(tklabel(m2,text='Mean2, SD2',width=10),side='left')
```

<Return>	<FocusIn>
<Key-x>	<FocusOut>
<Alt-x>	<Button-1>, <Button-2>, etc.
<Control-x>	<ButtonRelease-1>, <ButtonRelease-2>, etc.
<Destroy>	<Double-Button-1>, <Double-Button-2>, etc.

```

tkpack(tkentry(m2,width=5,textvariable=mean2),side='left')
tkpack(tkentry(m2,width=5,textvariable=sd2),side='left')
tkpack(m2,side='top')

tkpack(s)

tkpack(mainfrm)

```

## 12.7 Binding

For those widgets that naturally have an action or variable associated with them, the Tk interface provides arguments like `textvariable=`, or `command=` to make it easy to get these widgets working as they should. But we're not limited to associating actions with only those widgets that provide such arguments. By using the `tbind` command, we can associate a function (similar to those accepted by functions that use a `command=` argument) to any of a large number of possible events. To indicate an event, a character string with the event's name surrounded by angle brackets is used. The following table shows some of the actions that can have commands associated with them. In the table, a lower case x is used to represent any key on the keyboard.

As an example, suppose we wish to create a GUI that will allow the user to type in the name of a data frame, and when the user hits return in the entry field, a set of radio buttons, one for each variable in the data frame, will open up below the entry field. Finally, a button will allow for the creation of a histogram from the selected variable.

```

require(tcltk)

makehist = function(...){
  df = get(tclvalue(dfname))
  thevar = tclvalue(varname)
  var = as.numeric(df[[thevar]])
  hist(var,main=paste("Histogram of",thevar),xlab=thevar)
}

showb = function(...){
  df = get(tclvalue(dfname))
  vars = names(df)

```

```

frms = list()
k = 1

mkframe = function(var){
  fr = tkframe(base)
  tkpack(tklabel(fr,width=15,text=var),side='left')
  tkpack(tkradiobutton(fr,variable=varname,value=var),side='left')
  tkpack(fr,side='top')
  fr
}

frms = sapply(vars,mkframe)

tkpack(tkbutton(base,text='Run',command=makehist),side='top')
}

base = tktoplevel()
tkwm.title(base,'Chooser')

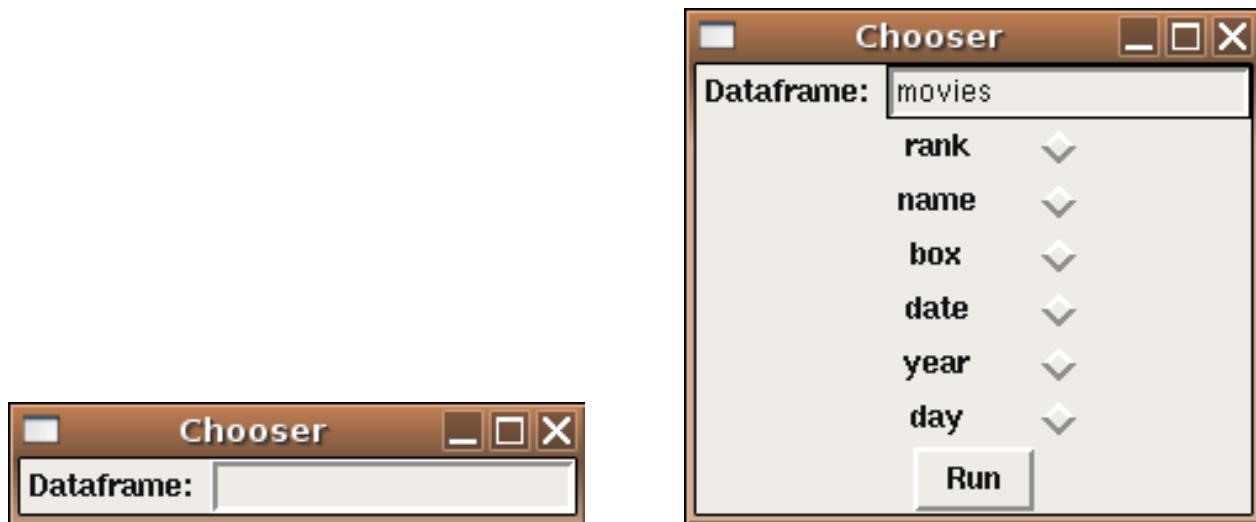
dfname = tclVar()
varname = tclVar()

efrm = tkframe(base)

tkpack(tklabel(efrm,text='Dataframe: '),side='left')
dfentry = tkentry(efrm,textvariable=dfname,width=20)
tkbind(dfentry,'<Return>',showb)
tkpack(dfentry,side='left')
tkpack(efrm)

```

Here's a picture of the widget in both the "closed" and "open" views:



As another example, recall the calculator we developed earlier. To allow us to enter values from the keyboard in addition to clicking on the calculator “key”, we could modify the `mkput` function from that example as follows:

```
mkput = function(sym){
  fn = function(...){
    calcinp <- paste(calcinp,sym,sep=' ')
    tkconfigure(display,text=calcinp)
  }
  tkbind(base,sym,fn)
  fn
}
```

When we define each key, we also bind that key to the same action in the `base` window. Similar actions could be done for the “C” and “=” keys:

```
tkbind(base,'c',clearit)
tkbind(base,'=',docalc)
```

Now input from the calculator can come from mouse clicks or the keyboard.

## 12.8 Checkbuttons

When we created radiobuttons, it was fairly simple, because we only needed to create one `tcl` variable, which would contain the value of the single chosen radiobutton. With checkbuttons, the user can choose as many selections as they want, so each button must have a separate `tcl` variable associated with it. While it’s possible to do this by hand for each choice provided, the following example shows how to use the `sapply` function to do this in a more general

way. Notice that special care must be taken when creating a list of `tcl` variables to insure that they are properly stored. In this simple example, we print the choices, by using `sapply` to convert the list of `tcl` variables to R variables. Remember that all the toplevel variables that you create in an R GUI are available in the console while the GUI is running, so you examine them in order to see how their information is stored.

```

require(tcltk)
tt <- tkoplevel()
choices = c('Red','Blue','Green')
bvars = lapply(choices,function(i)tclVar('0'))
# or bvars = rep(list(tclVar('0')),length(bvars))
names(bvars) = choices
mkframe = function(x){
  nn = tkframe(tt)
  tkpack(tklabel(nn,text=x,width=10),side='left')
  tkpack(tkcheckbutton(nn,variable=bvars[[x]]),side='right')
  tkpack(nn,side='top')
}

sapply(choices,mkframe)

showans = function(...){
  res = sapply(bvars,tclvalue)
  res = names(res)[which(res == '1')]
  tkconfigure(result,text = paste(res,collapse=' '))
}

result = tklabel(tt,text=' ')
tkpack(result,side='top')

bfrm = tkframe(tt)
tkpack(tkbutton(bfrm,command=showans,text='Select'),side='top')
tkpack(bfrm,side='top')

```

The figure below shows what the GUI looks like after two choices have been made, and the button is pressed.



## 12.9 Opening Files and Displaying Text

While it would be very simple (and not necessarily a bad idea) to use an entry field to specify a file name, the Tk toolkit provides a file browser which will return the name of a file which a user has chosen, and will usually be a better choice if you need to open a file.

Another issue that arises has to do with displaying output. If the output is a plot, it's simple to just open a graph window; if the output is text, you can just print it in the normal way, and it will display in the same place as you typed the R command to display the GUI. While this is useful while debugging your program, if there are printed results they should usually be displayed in a separate window. For this purpose, Tk provides the `tktext` object. To display output in this way, we can use the `capture.output` function, which takes the output of any R command and, instead of displaying it to the screen, stores the output in a vector of character values. Alternatively, the output could be written to a file, then read back into a character vector.

The following example program invokes `tkgetOpenFile` to get the name of a file to be opened; it then opens the file, reads it as a CSV file and displays the results in a text window with scrollbars. The `wrap='none'` argument tells the text window not to wrap the lines; this is appropriate when you have a scrollbar in the x-dimension. Finally, to give an idea of how the text window can be modified, a button is added to change the text window's display from the listing of the data frame to a data frame summary.

```
require(tcltk)

summ = function(...){
  thestring = capture.output(summary(myobj))
  thestring = paste(thestring, collapse='\n')
  tkdelete(txt,'@0,0','end')
  tkinsert(txt,'@0,0',thestring)
}
```

```

fileName<-tclvalue(tkgetOpenFile())

myobj = read.csv(fileName)

base = tktoplevel()
tkwm.title(base,'File Open')
yscr <- tkscrollbar(base,
                     command=function(...)tkyview(txt,...))
xscr <- tkscrollbar(base,
                     command=function(...)tkxview(txt,...),orient='horiz')
txt <- tktext(base,bg="white",font="courier",
              yscrollcommand=function(...)tkset(yscr,...),
              xscrollcommand=function(...)tkset(xscr,...),
              wrap='none',
              font=tkfont.create(family='lucidatypewriter',size=14))

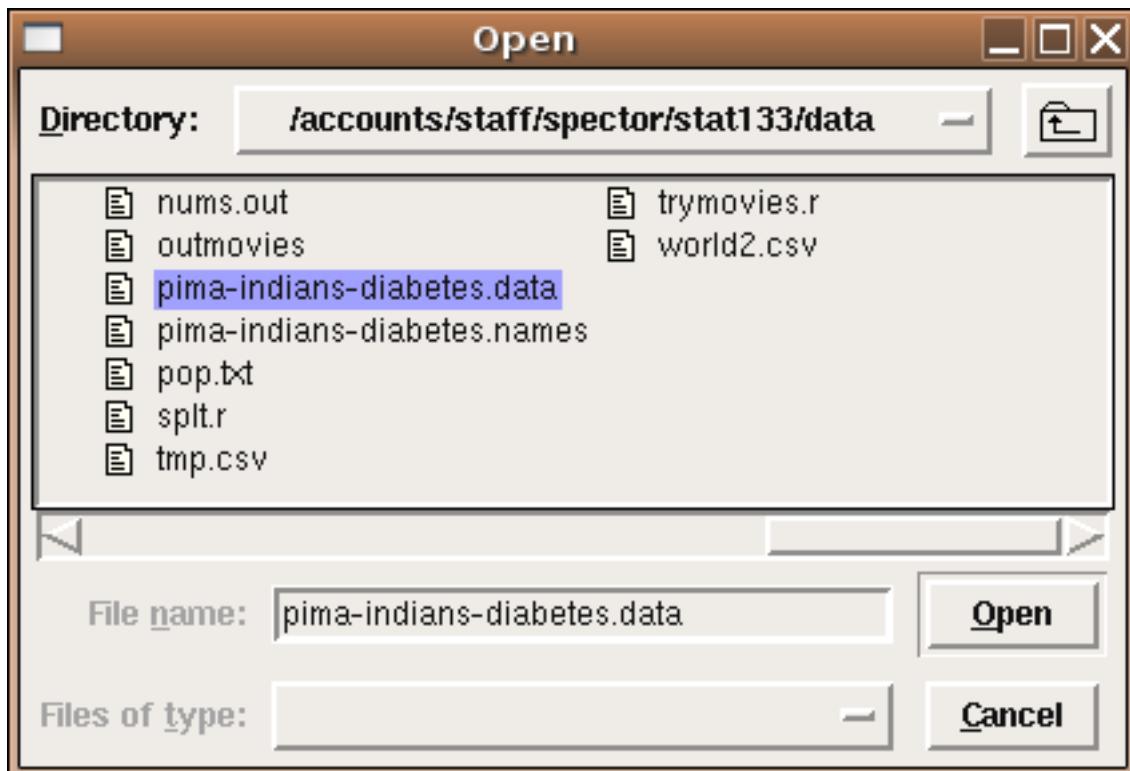
tkpack(yscr,side='right',fill='y')
tkpack(xscr,side='bottom',fill='x')
tkpack(txt,fill='both',expand=1)

tkpack(tkbutton(base,command=summ,text='Summary'),side='bottom')

thestring = capture.output(print(myobj))
thestring = paste(thestring,collapse='\n')
tkinsert(txt,"end",thestring)
tkfocus(txt)

```

The first picture shows the file chooser (as displayed on a Linux system); the second picture shows the diabetes data set displayed in a text window, and the third picture shows the result of pressing the summary button.



	npreg	glucose	bp	skinf	insulin	BMI	ped	age	diabetes
1	6	148	72	35	0	33.6	0.627	50	1
2	1	85	66	29	0	26.6	0.351	31	0
3	8	183	64	0	0	23.3	0.672	32	1
4	1	89	66	23	94	28.1	0.167	21	0
5	0	137	40	35	168	43.1	2.288	33	1
6	5	116	74	0	0	25.6	0.201	30	0
7	3	78	50	32	88	31.0	0.248	26	1
8	10	115	0	0	0	35.3	0.134	29	0
9	2	197	70	45	543	30.5	0.158	53	1
10	8	125	96	0	0	0.0	0.232	54	1
11	4	110	92	0	0	37.6	0.191	30	0
12	10	168	74	0	0	38.0	0.537	34	1
13	10	139	80	0	0	27.1	1.441	57	0
14	1	189	60	23	846	30.1	0.398	59	1
15	5	166	72	19	175	25.8	0.587	51	1
16	7	100	0	0	0	30.0	0.484	32	1
17	0	118	84	47	230	45.8	0.551	31	1
18	7	107	74	0	0	29.6	0.254	31	1
19	1	103	30	38	83	43.3	0.183	33	0
20	1	115	70	30	96	34.6	0.529	32	1
21	3	126	88	41	235	39.3	0.704	27	0
22	8	99	84	0	0	35.4	0.388	50	0
23	7	196	90	0	0	39.8	0.451	41	1

File Open

npreg	glucose	bp	skinf	insulin
Min. : 0.000	Min. : 0.0	Min. : 0.0	Min. : 0.00	Min. : 0.0
1st Qu.: 1.000	1st Qu.: 99.0	1st Qu.: 62.0	1st Qu.: 0.00	1st Qu.: 0.0
Median : 3.000	Median :117.0	Median : 72.0	Median :23.00	Median : 30.5
Mean : 3.845	Mean :120.9	Mean : 69.1	Mean :20.54	Mean : 79.8
3rd Qu.: 6.000	3rd Qu.:140.2	3rd Qu.: 80.0	3rd Qu.:32.00	3rd Qu.:127.2
Max. :17.000	Max. :199.0	Max. :122.0	Max. :99.00	Max. :846.0

Summary

## 12.10 Using Images with the `tcltk` package

The current version of tcl that ships with R allows you to display images in your GUIs, but only if they are in GIF format. Fortunately, it's very easy to change the format of images if they are, for example JPEG or PNG files. On any SCF machine, the command:

```
mogrify -format gif *.jpg
```

will convert all the files ending in `.jpg` to files with the same name, but in the GIF format. You can then use the `.gif` files that are created to display in your GUI.

To use images with the `tcltk` library, first create a Tcl variable representing the image with code like:

```
myimage = tclVar()  
tcl("image","create","photo",myimage,file="picture.gif")
```

Next, create a `tklabel` using the `image=` argument pointing to the tcl variable that holds the image:

```
img = tklabel(frame,image=myimage)
```

Like any other widget, you can change the image during the execution of your program using the `tkconfigure` function. Suppose we have another tcl variable called `otherimage` that has an image associated with it. To change the `img` widget to display that image, use

```
tkconfigure(img,image=otherimage)
```

If you have many pictures, it will be more convenient to store the tcl image variables in a list. Suppose the R list `pics` contains the full pathname of several GIF files. (It's usually simplest to use `setwd` to change to the directory where you're storing the GIF files, so that you only need to type their names, not their full paths. The command

```
setwd(file.choose())
```

allows you to navigate to the directory with the GIF files.)

The following example displays pictures of three fruits, chosen from a list of five. When the button is pressed, it uses `tkconfigure` to change each picture many times, to give the illusion of a slot machine. Note the use of `sapply` to create a list of images, regardless of the number of images used.

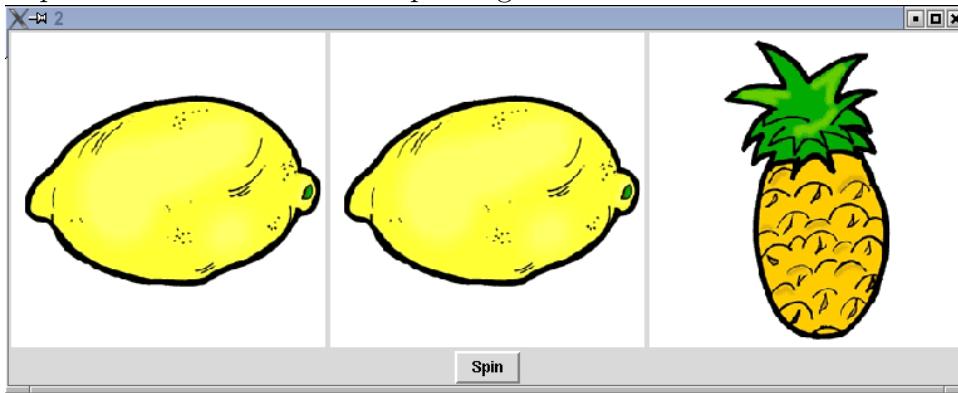
```
require(tcltk)  
pics = list.files('fruits',pattern='\\.gif$')  
pics = paste('fruits/',pics,sep='')  
n = length(pics)  
theimages = sapply(pics,function(pic)  
    tcl("image","create","photo",tclVar(),file=pic))
```

```

spinner = function(...){
  for(i in 1:50){
    r = sample(1:n,size=3,replace=TRUE)
    tkconfigure(img1,image=theimages[[r[1]]])
    tkconfigure(img2,image=theimages[[r[2]]])
    tkconfigure(img3,image=theimages[[r[3]]])
  tcl('update')
  Sys.sleep(.07)
  }
}
top = tkoplevel()
f1 = tkframe(top)
f2 = tkframe(top)
r = sample(1:n,size=3,replace=TRUE)
img1 = tklabel(f1,image=theimages[[r[1]]])
img2 = tklabel(f1,image=theimages[[r[2]]])
img3 = tklabel(f1,image=theimages[[r[3]]])
tkpack(img1,side='left')
tkpack(img2,side='left')
tkpack(img3,side='left')
tkpack(tkbutton(f2,text='Spin',command=spinner),side='bottom')
tkpack(f1,side='top')
tkpack(f2,side='bottom')

```

A picture of the GUI before spinning is shown below



# Chapter 13

## CGI Programming with R

### 13.1 Web Servers

We're all familiar with the use of a webserver - we type an address or click on a link, and then the page we want appears in our browser. But what actually happens when we type in a web address? Web servers are simply programs that listen for requests on a particular port (port 80 by default), and then send out information to the connection that requested it. Such a two-way connection is known as a socket. To see what really happens when a request is sent to a webserver, we can write an R program that creates a socket on an unused port, and then try to access that port through a web browser. Then we can have our program simply send back whatever it received in order to see what's really going on. The program is surprisingly simple:

```
webecho = function(){
  xx = make.socket('localhost',port=1888,server=TRUE)
  on.exit(close.socket(xx))
  stuff = read.socket(xx,maxlen=1024)
  write.socket(xx,stuff)
}
```

The function opens a socket to port 1888, reads whatever gets sent to it, and then sends it right back to the same place as the request came from, so it can be displayed in a browser. The call to `on.exit` makes sure that the socket is closed after the communication takes place. To run the program so that it will be able to respond to requests, we place it in a loop:

```
> while(1)webecho()
```

(If you run the program on a UNIX system, you'll have to quit your R session (with control-\) to stop it; on a Windows system you may be prompted to quit your R session when you try to exit.)

Now let's type the following address into a browser:

```
http://localhost:1888/something
```

Here's what appears in the browser:

```
GET /something HTTP/1.1
Host: localhost:1888
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.7.12) Gecko/20051010 Galeon/1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,
Accept-Language: en
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

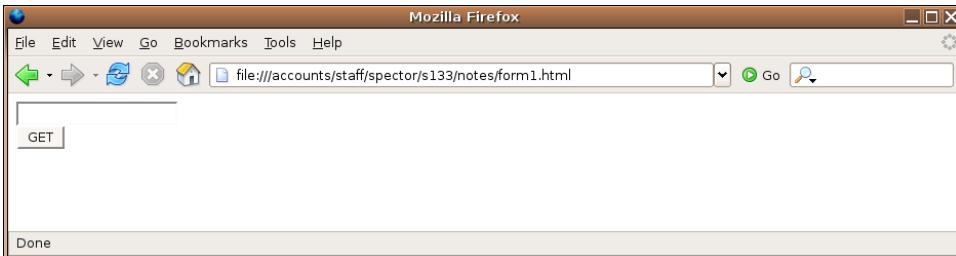
The output consists of the actual request (the line with the `GET`), and a series of headers. This is all the information that is transmitted from the browser to the webserver when a request is made for a particular page. Once a webserver receives a request like this, it returns the page that the user requested (as specified in the `GET` directive). If that page contains images, documents or additional programs, they would be sent to the browser through the same mechanism.

This is fine for ordinary, static pages, but what about the case where we accept input from the user? This would be the case when data is entered into a field, or checkboxes are checked on a page, and then a button is clicked to send the information from the browser to the web server. Web pages that accept information are known as forms. There are two mechanisms that can be used to send information from a web browser to a webserver; GET (which is the default, if you don't explicitly state a method) and POST. (We'll look at their differences shortly.) For example, the HTML that would be used on a web page to accept a text field and use the GET method to send the information would look like this:

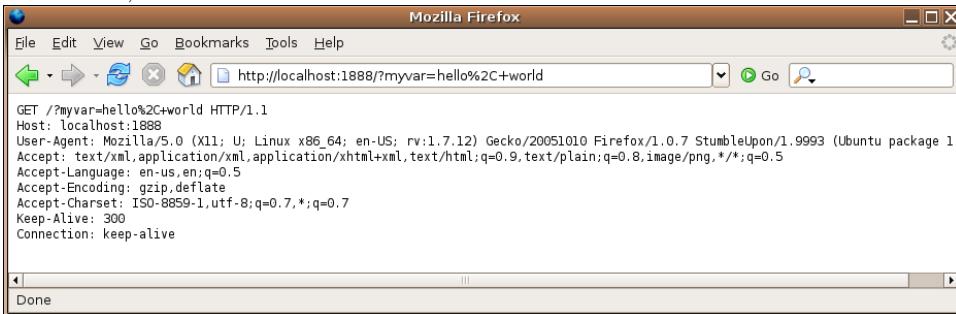
```
<form action='http://localhost:1888' method=GET>
<input type=text name=myvar><br>
<input type=submit value='GET'>
</form>
</html>
```

Notice the `action=` specification in the form tag; this specifies the program which will be executed once the information is sent from the browser to the web server. The `name=` specification, which must always be present in an HTML that will accept information from a user, defines the name of a CGI variable. CGI stands for "Common Gateway Interface", and is a catch-all term for any program which is executed by a web server in response to a request from a browser.

Here's how this html would display in a browser:



What happens if we enter something in the field, and click on the button? Let's put the words "hello, world" in the field and see:



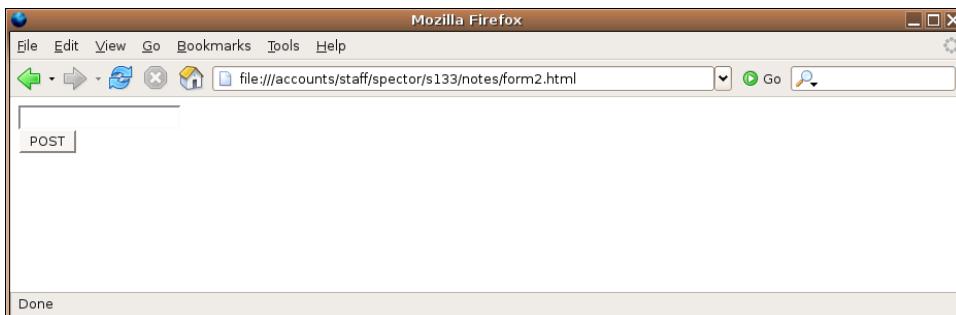
First, notice that the address has changed, reflecting the address given in the `action=` specification. Additionally, some text has been added to the end of the displayed URL, and we can see that this same text was sent to the web server as part of the `GET` specification.

The extra text that's in the URL and the `GET` specification is said to be urlencoded; non-alphanumeric characters (like the comma) are replaced with a percent sign followed by the two-digit hexadecimal representation of the character, and spaces are replaced with plus signs. (A complete list of the encodings can be found at [http://www.w3schools.com/tags/ref\\_urlencode.asp](http://www.w3schools.com/tags/ref_urlencode.asp))

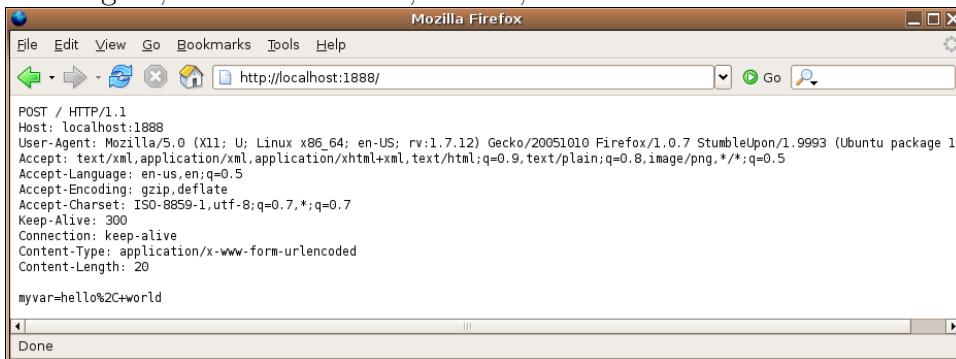
This method of transferring information has its limitations. First of all, it's obvious that it won't work for things like files, because there's just not enough room in a URL to accommodate files of arbitrary length. Secondly, it may not be prudent to display the information that you're transferring in the URL where it can easily be seen. An alternative to `GET` is known as `POST`. Let's see how it differs from `GET` by creating a form identical to the first one, but with the method specified as `POST`. The html looks like this:

```
<form action='http://localhost:1888' method=post>
<input type=text name=myvar><br>
<input type=submit value='POST'>
</form>
```

Here's how it displays in the browser:



Once again, we'll enter "hello, world", and click on the button:



Notice that the URL no longer has any information about the CGI variables; the urlencoded string with the values now appears after the header information that the web browser sent. (Remember that for protocols like HTTP, a single completely blank line indicates the end of the headers.) Since the information in the form is not placed in the URL, this should allow us to upload local files to a CGI program. Here's the HTML code for a form that will upload a file:

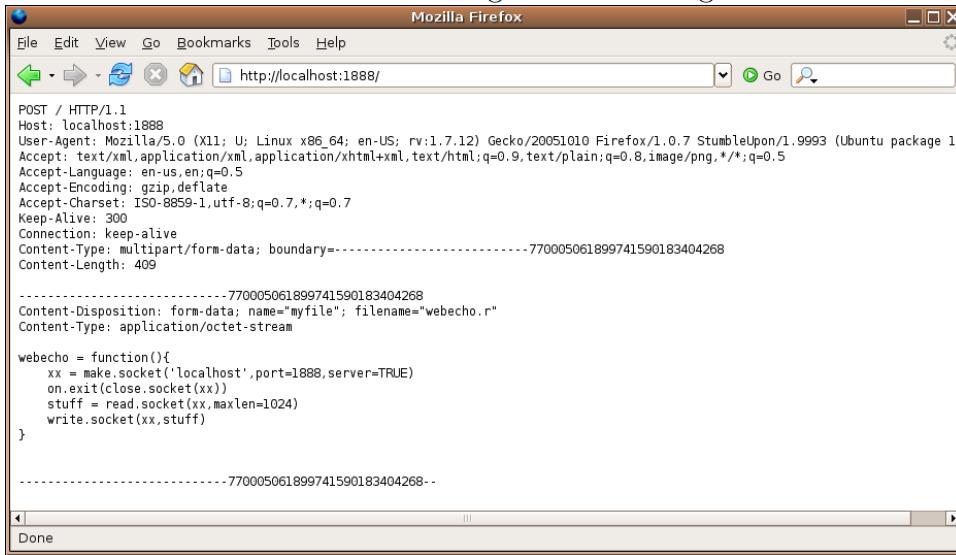
```
<form action='http://localhost:1888' method=post enctype="multipart/form-data">
<input type=file name='myfile'>
<input type=submit value='POST'>
</form>
```

Here it is displayed in a browser:



The `Browse...` button is generated automatically when `type=file` is specified for an input element. For file uploading, the method must be set to `post`; using `get` just won't

work. Now I'll fill out the form with the name of the file that holds the source for the webecho function that we've been using to see how it gets transferred.



```
POST / HTTP/1.1
Host: localhost:1888
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.7.12) Gecko/20051010 Firefox/1.0.7 StumbleUpon/1.9993 (Ubuntu package 1.0.7)
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Content-Type: multipart/form-data; boundary=-----770005061899741590183404268
Content-Length: 409

-----770005061899741590183404268-
Content-Disposition: form-data; name="myfile"; filename="webecho.r"
Content-Type: application/octet-stream

webecho = function(){
  xx = make.socket('localhost',port=1888,server=TRUE)
  on.exit(close.socket(xx))
  stuff = read.socket(xx,maxlen=1024)
  write.socket(xx,stuff)
}

-----770005061899741590183404268-
```

The file is sent (after the headers) in a form very similar to an email attachment. One big difference is that, if the file to be transmitted is a non-text file (like an image or document), no encoding needs to be done – the HTTP protocol utilizes all 8 bits of each byte. You may have noticed the `enctype="multipart/form-data"` specification in the previous form. This is what causes the web browser to send it's data in the “attachment” style, and is required when you're uploading a file. If you specify it in a normal form, it will send the CGI variable information in a similar fashion. With a form like this:

```
<form action='http://localhost:1888' method=post enctype="multipart/form-data">
<input type=text name=myvar><br>
<input type=submit value='POST'>
</form>
```

entering “hello, world” results in the following:

```
POST / HTTP/1.1
Host: localhost:1888
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.7.12) Gecko/20051010 Firefox/1.0.7 StumbleUpon/1.9993 (Ubuntu package 1.0.7)
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Content-Type: multipart/form-data; boundary=-----14394547745218385
Content-Length: 178
```

```
-----14394547745218385925125258
Content-Disposition: form-data; name="myvar"

hello, world
-----14394547745218385925125258--
```

Note that in this case, there was no urlencoding of the variable's value.

## 13.2 CGI Scripting

Before continuing, it's worth making note of some very practical details that you'll have to consider before beginning a project that uses CGI scripting. The programs which a web server will execute in response to a request need to be in specially designated directories, or the web server will treat them as ordinary documents, not programs. In general, access to such directories will be limited, both for financial and security reasons. In addition, a program which requires large amounts of computer resources may make it difficult for others who use the server to get anything done.

When you are writing CGI programs, it's very important to avoid allowing users manipulate any input which might find its way outside of the web server's slightly protected environment. What this means is that programs that call operating system programs, (for example calls to `system` in R), should generally be avoided.

Now that we've seen what's going on behind the scenes, we can start to look at the facilities in R that will allow us to write CGI scripts. The `CGIwithR` library provides a "wrapper" script that sets up an environment in which R can run. The most important function of the `CGIwithR` library is to gather all the information that the web browser has sent to the web server (regardless of how it was sent to the browser) and to make it easily available inside of R. The `CGIwithR` library takes all of the CGI variables (defined in the forms that a user has responded to), and places them in a list called `formData`, which can be manipulated like any other list in R. Note that the library resolves all issues regarding which method (get or post) was used in the form that generated the variables, and that the `formData` list is the only object you need to look at to get the values of the CGI variables defined in the form.

Once your program has received the information from a user's web browser, any printing statements that you make in your program will be directed back to the browser. This means that you need to follow the rules of HTML when you are displaying results, not the ordinary rules that would apply for an regular R session. The `cat` function is very useful in this regard, because it displays exactly what you tell it to, without any element numbering or formatting. If you know the basics of HTML, you may prefer to simply use `cat` to generate most or all of the HTML your CGI script creates. (There is a library called `R2HTML` that automates the "HTML-ization" of R objects, and you may want to study it on your own, but we won't be using it here.) Here are some ideas for generating appropriate output in CGI programs written with R.

- The `CGIwithR` library provides a few helper functions that make generating useful HTML much easier:
    - `HTMLheader` - This function, called with no arguments, prints the appropriate information at the beginning of your output so that all browsers will realize that they're receiving HTML. It's recommended that you call this function before you produce any output in an CGI script.
    - `tag` and `untag` - These functions accept unquoted arguments and produce either the opening or closing tag required for most HTML constructs. For example, a large title line can be specified in HTML by beginning with `<h1>` and ending with `</h1>`. You could generate a title using `tag` and `untag` as follows:

```
tag(h1)
cat('Here's my title')
untag(h1)
```

  - `br` - To indicate a line break in HTML, the `<br>` tag must be used; HTML pays no attention to newline characters. The `br` function will print such a tag, so it can be inserted into your CGI program wherever you want to insert line breaks. An optional argument (defaulting to 1) tells `br` how many lines you want inserted.
- The HTML `<pre>` tag specifies that you don't want the browser that displays the output to perform any formatting. Suppose you want your CGI program to display the results of the `summary` function for a variable `x` in a browser window just as it would appear in a command-line R session. You could use the following:

```
tag(pre)
print(summary(x))
untag(pre)
```

- The `<p>` tag is an alternative way of placing line breaks into your output.

### 13.3 A First CGI program with R

As a simple way of getting started with CGI programming, let's take the output from the first simple form that we created and see how the information gets translated from the CGI environment into the R environment. I'm assuming that your SCF account is `s133xx` and that you've followed the instructions at <http://www.stat.berkeley.edu/classes/s133/projects/tech/Cgiprogs.html> to prepare your account to use CGI scripting. Typically, you would put the page containing the html in your `public_html` directory. Suppose the following is placed in a file called `form1.html` in your `public_html` directory:

```

<html>
<form action='cgi-bin/R.cgi/test1.cgi' method=get>
<input type=text name=myvar><br>
<input type=submit value='GET'>
</form>
</html>

```

Since file names in HTML are interpreted relative to the directory that the HTML resides in, we can refer to the CGI program relative to the `cgi-bin` directory. Notice that this will allow you to use the same form whether you're running on the SCF network or through a tunnel. In addition to the `formData` list that holds the values of the CGI variables, information is also passed to your CGI program through environmental variables. These variables can be accessed with the `Sys.getenv` function, but for our purposes here, we can use the `showEnvironmentVariables` function that's part of the `CGIwithR` package. Here's a simple program that will just print out some information about the `formData` list and the environmental variables that are transferred into the environment:

Here's the `test1.cgi` program:

```

HTMLheader()

cat('Class of formData=', class(formData), ' Mode of formData=', mode(formData), '<br>')
cat('Names of formData=', names(formData), '<br>')

tag(pre)
print(formData)
untag(pre)

showEnvironmentVariables()

cat('</body></html>')

```

If we run the program (by pointing a browser at `http://springer/ s133xx/form1.html`, entering "hello, world" and pressing the button), here's what the output looks like:

```

Class of formData= list Mode of formData= list
Names of formData= myvar

$myvar
[1] "hello, world"

SERVER_SIGNATURE      Apache/2.0.54 (Ubuntu) Server at springer Port 80
R_INCLUDE_DIR        /usr/local/linux/R-2.2.1/include
HTTP_KEEP_ALIVE     300
HTTP_USER_AGENT      Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.7.12) Gecko/20051

```

```

SERVER_PORT          80
HTTP_HOST            springer
LD_LIBRARY_PATH      /usr/local/linux/R-2.2.1/lib:/usr/local/lib64:/usr/lib/gcc/x86_64-linux-gnu
DOCUMENT_ROOT        /mirror/data/pub/html/classes/s133
HTTP_ACCEPT_CHARSET ISO-8859-1,utf-8;q=0.7,*;q=0.7
SCRIPT_FILENAME      /class/u/s133/s133xx/public_html/cgi-bin/R.cgi
REQUEST_URI          /~s133xx/cgi-bin/R.cgi/test1.cgi?myvar=hello%2C+world
SCRIPT_NAME          /~s133xx/cgi-bin/R.cgi
R_GSCMD              /usr/bin/gs
HTTP_CONNECTION      keep-alive
PATH_INFO             /test1.cgi
REMOTE_PORT           48242
PATH                 /usr/local/bin:/usr/bin:/bin
R_LIBS
PWD                 /class/u/s133/s133xx/public_html/cgi-bin
SERVER_ADMIN          webmaster@localhost
R_SHARE_DIR           /usr/local/linux/R-2.2.1/share
LANG
HTTP_ACCEPT_LANGUAGE en
PATH_TRANSLATED       /class/u/s133/s133xx/public_html/cgi-bin/test1.cgi
HTTP_REFERER          http://springer/~s133xx/form1.html
HTTP_ACCEPT           text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,image/*,*/*;q=0.8
REMOTE_ADDR           128.32.135.22
SHLVL                1
SERVER_NAME           springer
FORM_DATA             myvar=hello%2C+world
SERVER_SOFTWARE        Apache/2.0.54 (Ubuntu)
QUERY_STRING           myvar=hello%2C+world
SERVER_ADDR            128.32.135.22
GATEWAY_INTERFACE     CGI/1.1
R_HOME                /usr/local/linux/R-2.2.1
SERVER_PROTOCOL        HTTP/1.1
HTTP_ACCEPT_ENCODING  gzip,deflate
R_DOC_DIR              /usr/local/linux/R-2.2.1/doc
REQUEST_METHOD         GET
R_SESSION_TMPDIR       /tmp/RtmpPPrmxy
R_PLATFORM             x86_64-unknown-linux-gnu
R_PAPERSIZE            letter
R_PRINTCMD             lpr
R_LATEXCMD             /usr/bin/latex
R_DVIPS_CMD             /usr/bin/dvips
R_MAKEINDEXCMD         /usr/bin/makeindex

```

R_RD4DVI	ae
R_RD4PDF	times,hyper
R_UNZIPCMD	/usr/bin/unzip
R_ZIPCMD	/usr/bin/zip
R_BROWSER	/usr/bin/firefox
EDITOR	vi
PAGER	/usr/bin/less
R_PDFVIEWER	/usr/local/linux/bin/acroread
AWK	gawk
EGREP	grep -E
MAKE	make
PERL	/usr/bin/perl
TAR	tar
LN_S	ln -s
R_USE_AQUA_SUBDIRS	no

You may have noticed that the URL we used for the `action=` specification makes it look like `R.cgi` is a directory, not a file. What happens is that the `R.cgi` program gets invoked by the web server, and it examines the `PATH_INFO` variable to find the name of your script (`test1.cgi` in this case.) It then loads up the `formData` list and calls your program.

If you examine the information that our simple web echoing program displayed, you'll see that each of the headers that were passed to the program have been transferred to environmental variables with a prefix of `HTTP_`. Although you most likely won't need them, you can examine the list of environmental variables to find information that you might want to use in your CGI programs and then access the information by using `Sys.getenv`.

The most important thing to notice is that the `myvar` CGI variable, defined in the input field of the HTML form, is available inside your R program, in unencoded form, in the element with the name `myvar`. This is the mechanism by which all the CGI variables are transferred into the R environment.

## 13.4 Data

The `R.cgi` script calls your R program in such a way that it doesn't automatically load any data into the R environment. So if you want to have data available to your CGI program, you'll need to explicitly get the data into R's environment. For reasons of efficiency, your program should always use the `load` function to load a previously saved binary version of the data you need. The most convenient place to store these objects is right in the `cgi-bin` directory from which your program will execute.

Suppose we wish to create a CGI program that will accept the name of one of the variables from the `wine` data frame, and then display a summary of the data. Before you plan to run the script, the `wine` data should be saved in a simple R session that's started after you've changed your working directory to be your cgi-bin directory. The command to do this is

```
save(wine,file='wine.rda')
```

Next, we can create a form, which would be saved in the `public_html` directory. Here's a simple example, which we'll save in the file `wine.html`:

```
<html><body>
<h1>Summary for Wine Variables</h1>
<form action='cgi-bin/R.cgi/dowine.cgi'>
Enter the name of the variable:
<input type=text name=winevar><br>
<center>
<input type=submit value='Run'>
</center>
</form>
</body></html>
```

The `dowine.cgi` program would look like this:

```
load('wine.rda')

HTMLheader()

winevar = formData$winevar
tag(h1)
cat('Summary for wine$',winevar,sep=' ')
untag(h1)
tag(h2)
tag(pre)
print(summary(wine[[winevar]]))
untag(pre)
untag(h2)
cat('</body></html>')
```

Here's the form:

The screenshot shows a Mozilla Firefox window with the title bar "Mozilla Firefox". The address bar contains the URL "http://springer/~s133ar/wine.html". The main content area displays a heading "Summary for Wine Variables" followed by a form input field containing "Alcohol" and a "Run" button. At the bottom of the window is a "Done" button.

Here's the result of submitting the form:

The screenshot shows a Mozilla Firefox window with the title bar "Mozilla Firefox". The address bar contains the URL "http://springer/~s133ar/cgi-bin/R.cgi?dowine.cgi?winevar=Alcohol". The main content area displays a heading "Summary for wine\$Alcohol" followed by a table of statistical summary values:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
11.03	12.36	13.05	13.00	13.68	14.83

At the bottom of the window is a "Done" button.

## 13.5 Combo Forms

Of course, having the user remember the name of the variable they're interested in isn't a very user-friendly strategy, but the thought of manually preparing a form that lists all the variables isn't very appealing either. The problem can be solved by having the CGI program generate the form the first time it's called, and then processing the form when it's submitted back to the web server. If we call the CGI program directly (not through a form submission), the `formData` list will be empty, and we can use that condition to tell whether we need to generate the form or respond to it. Since R will be generating the form, it's very easy to have it provide a choice for each variable. For this example, let's use a drop down menu that will display the names of each variable. Here's a program that will both generate the form and respond to it:

```
if(length(formData) == 0){
  HTMLheader()
  tag(h1)
  cat('Summary for Wine Variables')
  untag(h1)
  cat("<form action='dowine1.cgi'>")
  cat("Choose the variable:")
  cat("<select name='winevar'>")
  load("wine.rda")
  sapply(names(wine),function(x)cat("<option value='",x,'">",x,"<br>\n",sep=''))
```

```

cat("</select>")
cat('<input type="submit" value="Run">')
cat("</form></body></html>")
} else {
  load('wine.rda')
  HTMLheader()
  winevar = formData$winevar
  tag(h1)
  cat('Summary for wine$', winevar, sep=' ')
  untag(h1)
  tag(h2)
  tag(pre)
  print(summary(wine[[winevar]]))
  untag(pre)
  untag(h2)
  untag(h2)
  cat('</body></html>')
}

```

One very important thing to notice if you use this approach – the `action=` argument should specify only the name of the program, without the usual `R.cgi/`; since R is already calling your program, it thinks it's in the `R.cgi` “directory”.

Here's the result of calling the program directly:



and here's the result after making a choice:



## 13.6 Graphs

Certainly one of the most useful functions of CGI scripting with R is to display graphics generated by R, based on a user's choices as specified on a web form. This would provide a simple way to allow people unfamiliar with R to produce attractive graphs; if a means is provided for data input, and enough options are provided through checkboxes, drop-down menus, radiobuttons, etc, a complete web-based graphing solution could be developed.

To properly create and display graphics with a CGI program, it's necessary to understand the difference between the internal paths (which your R program will see) and the external paths (which are the addresses typed into the browser's address field.) For example, the way the class webserver is configured, the directory into which you would put HTML pages is (once again assuming your SCF id is s133xx):

```
/class/u/s133/s133xx/public_html/
```

This directory provides a convenient place to place graphics generated by your scripts. To the outside world, this directory would be indicated as:

```
http://springer/~s133xx/
```

or

```
http://localhost:8080/~s133xx/
```

So as far as the webserver is concerned (i.e. the way the outside world would find your files through a URL), the directory is known as

```
/~s133xx/
```

To create graphics from your CGI script, you first create a variable called `graphDir` and set it equal to the full internal name of the directory into which you'll write your graphs. In our example it would be `/class/u/s133/s133xx/public_html/`. Then use the `webPNG` function, specifying the name (without any leading directories) that you want to use for your graph. In order to generate the appropriate HTML so that your image will be displayed, you can use the `img` function of the `CGIwithR` library. This function takes two arguments. The first is the name of the graphic you produced via `webPNG`, and the second is called `graphURLroot`, and should be set to the "outside" view of your `public_html` directory, namely `/~s133xx/`. (Note the trailing slashes in both the `graphDir` and `graphURLroot`; they are required.)

To illustrate, let's create a simple CGI program that will generate some random data and create a conditioning plot containing histograms.

```
library(lattice)
HTMLheader()
x = data.frame(z = rnorm(1000), g = factor(sample(1:5,size=1000,replace=TRUE)))
graphDir='/class/u/s133/s133xx/public_html/'
```

```

cat("Now I'm going to plot some histograms:<br>")
webPNG(file='hist.png')
histogram(~z|g,data=x)
invisible(dev.off())
img(src='hist.png',graphURLroot='/~s133xx/')
cat("</body></html>")

```

The size of the plot can be controlled by passing `width=` and `height=` arguments to `webPNG`; the units for these arguments are pixels.

If you are using lattice graphics and your plot does not appear, try passing the call to the `lattice` function to the `print` function.

Notice the call to `dev.off`; without it, your graph may not be properly terminated, and only some (or possibly none) of the graph will be displayed.

## 13.7 Hidden Variables

Suppose we have a web page that displays a choice of data frames for possible further analysis. Once a data frame is chosen, another page could display the variables that are available from that data frame, and a choice of plots could be provided. Remember that each time our CGI program is called, a new R process begins. So how can we “remember” the data set name in between invocations of our program? The answer is to use hidden variables. Any time you create an HTML form, you can create as many hidden variables as you need to store information that needs to be available in the next step of processing. These hidden variables are exactly like any other CGI variable, but there is no visible sign of the variable on the form that is displayed in the user’s browser. To create a hidden variable, use code like the following:

```
<input type="hidden" name="varname" value="the value">
```

Here’s an implementation of a program that looks in the current directory for any files with an extension of `.rda`, provides a drop down menu of data set names, then allows a choice of variables, and finally produces the requested plot:

```

HTMLheader()
if(length(formData) == 0){
  datasources = list.files('.',pattern='\\.rda$')
  datasources = sub('\\.rda$', '',datasources)
  cat('<form action="doplot.cgi">')
  cat('<select name=dataset>\n')
  sapply(datasources,function(x)cat('<option value="' ,x,'">' ,x,'</option>\n'))
  cat('</select>\n')
  cat('<center><button type="submit">Run</button></center></form>')
} else if('dataset' %in% names(formData)){

```

```

dataset = formData$dataset
dataset = gsub(' ', '', dataset)
load(paste(dataset, '.rda', sep=''))
cat('<form action="doplot.cgi">\n')
cat('<p>X-variable:<br>\n')
sapply(names(get(dataset)), function(i)
cat('<input type="radio" name="xvar" value=""', i, '">', i, '<br>\n'))
cat('<p>Y-variable:<br>\n')
sapply(names(get(dataset)), function(i)
cat('<input type="radio" name="yvar" value=""', i, '">', i, '<br>\n'))
cat('<input type="hidden" name="set" value=""', dataset, '">\n')
cat('<center><button type="submit">Run</button></center></form>')
cat('</form>')
} else{
  dataset = gsub(' ', '', formData$set)
  load(paste(dataset, '.rda', sep=''))
  xvar=gsub(' ', '', formData$xvar)
  yvar=gsub(' ', '', formData$yvar)
  graphDir = '/home/spector/public_html/'
  webPNG(file='theplot.png', graphDir=graphDir)
  thedata = get(dataset)
  plot(thedata[[xvar]], thedata[[yvar]], xlab=xvar, ylab=yvar)
  img(src='theplot.png', graphURLroot='/~spector/')
  invisible(dev.off())
}
cat('</body></html>')

```

This program has three sections: the first displays the initial form showing the data frame names (invoked when no `formData` is available); the second displays the variable choices (invoked when a `dataframe` variable is specified), and the third, which is invoked when `formData` is available, but the `dataset` variable is not defined.

## 13.8 Outgoing HTTP Headers

We've already seen that when a web browser makes a request to a web server, it sends a series of headers before the actual content (if any). The web server also sends headers to the browser, but up until now we've let the `R.cgi` wrapper script take care of that detail for us. In most cases, the only header that needs to be sent to a browser is one that informs the browser that we're sending it HTML (as opposed to, say an image or other binary file). That header looks like this:

`Content-type: text/html`

and it would be followed by a completely blank line to signal the end of the headers. The `R.cgi` script examines what you're about to send to the web browser, and, if it doesn't find the "Content-type" line, it inserts it before it sends your output to the browser. Thus, if you do insert that line, you are taking responsibility for the outgoing headers, and, if desired, you can add additional ones.

Just about the only header line that you might consider adding is one that specifies the value of a cookie to the browser. Cookies are small pieces of text, associated with a particular website, that are stored by a browser, and sent to web servers if they match the domain and possibly the path of the URL that initially set them. There are two types of cookies: session cookies, which expire when a particular browser session is ended and the browser is shut down, and persistent cookies, that are stored in a text file on the computer on which the browser is running, and will expire at a date specified in the header that defined the cookie. For this example, we'll create a session cookie, and then access it through a different script. If every web transaction had a form associated with it, we could use hidden CGI variables to do much of the work that cookies do, but, since they're stored on the user's computer, they are more reliable, and don't require any special programming. Here's an example of a script that sets a cookie:

```
if(length(formData) == 0){
  HTMLheader()
  cat("What is your name?")
  cat('<form action="setcookie.cgi">\n')
  cat('<input type="entry" name="name"><br>\n')
  cat('<button type="submit">Run</button>\n')

} else if('name' %in% names(formData)){
  name = formData$name
  cat("Content-type: text/html\nSet-Cookie: thename=",name,
      "; path=/~s133xx/\n\n",sep=' ')
  cat("Hello there, ",name)
}

cat('</body></html>')
```

Since everyone in class is sharing the same webserver, I've added a `path=` specification to the cookie header. For this class, it probably is a good idea to prevent people from getting cookies set by other programs. Note the two newlines at the end of the header line – these are essentially to make sure that the browser understands that the headers are done and the content is following. If you want to create persistent headers, you need to add an `expires=` specification to the `Set-cookie` header. The format of the expiration time must be followed precisely; in particular, the parts of the date must be separated by dashes, and the only allowable time zone is `GMT`. Here's an example of a header containing a variable, path and expiration date:

```
Set-Cookie: thename(Somebody; path=/~s133xx/; expires=Monday, 09-May-10 00:00:00 GMT
```

Now let's look at a program that will retrieve an already set cookie. When a browser recognizes a domain/path/time combination for which it has an active cookie, it sends it back to the webserver in a `Cookie:` header, not as a CGI variable. The format of the cookie is `name=value`, similar to the format in which CGI variables are transmitted. This means that we'll need to use `Sys.getenv` to access the environmental variable called `HTTP_COOKIE`.

```
HTMLheader()
cookie = Sys.getenv('HTTP_COOKIE')
name = gsub('^ *thename=(.*)$', '\1', cookie)
cat('Welcome back, ', name)
cat('</body></html>')
```

Notice that you can only access the cookies in a CGI program, not in ordinary HTML, but you don't need any form elements to get the cookie.

## 13.9 Creating Pretty Output

Since the output created by CGI programs is interpreted by the browser as HTML, we can use any HTML commands by simply having our program generate the necessary HTML statements. One simple way of organizing output in HTML is the HTML table. A table begins with the string `<table>`, and ends with `</table>`. Each row of the table begins with `<tr>`, and ends with `</tr>`; each element within a row begins with `<td>` and ends with `</td>`. To specify headings, the `th` tag can be used in place of `td`. This suggests the following function to produce one row of an HTML table:

```
makerow = function(x, tag='td'){
  st = paste('<', tag, '>', sep=' ')
  end= paste('</', tag, '>', sep=' ')
  cat(paste(paste('<tr>', st, sep=' '),
            paste(x, collapse=paste(end, st, sep=' '))),
      paste(end, '</tr>', sep=' '), "\n")
}
```

To print an entire data frame, we can first print the names as a header line, then use `apply` to print the body of the data frame:

```
dftable = function(df){
  cat('<table border=1>')
  makerow(names(df), tag='th')
  apply(df, 1, makerow)
  cat('</table>')
}
```

An example of using these functions will be presented in the next section.

## 13.10 File Upload

We've already seen that that an `input` element with `type=file` will create an entry field and browse button to allow a user to specify a file for upload. In the following program, we'll create such a field to allow a user to specify a local comma-separated file which will then be read into R, and displayed as a table. File upload using the `CGIwithR` library simply places the content of the uploaded file into a character string in the `formData` list corresponding to the name of the CGI variable specified in the HTML form. To treat this character string as a file, we can use the `textConnection` function.

The following program will upload a comma-separated file, which will then be read by `read.csv`, and displayed using the `dftable` function from the previous section:

```
HTMLheader()
if(length(formData) == 0){
  cat('<form action="readcsv.cgi" method=post enctype="multipart/form-data">\n')
  cat('<input type=file name=thefile><br>')
  cat('<input type=submit value="Upload">')
  cat('</form>')
} else{
  makerow = function(x,tag='td'){
    st = paste('<',tag,'>',sep=' ')
    end= paste('</',tag,'>',sep=' ')
    cat(paste(paste('<tr>',st,sep=' '),
              paste(x,collapse=paste(end,st,sep=' ')),
              paste(end,'</tr>',sep=' ')), "\n")
  }
  dftable = function(df){
    cat('<table border=1>')
    makerow(names(df),tag='th')
    apply(df,1,makerow)
    cat('</table>')
  }
  txtcon = textConnection(formData$thefile)
  df = read.csv(txtcon)
  dftable(df)
}
cat('</body></html>')
```

## 13.11 Debugging CGI Programs

The first step in debugging an R program is to make sure that there are no obvious syntax errors. This can be done easily by changing directories to the location of your CGI programs, running R and using the `source` command to execute your program. If you see any syntax

errors, they should be corrected before attempting to run your program through the web server. You can simulate input from a form by creating a list called `formData` and loading appropriate elements into named values in that list.

When a program is run as a CGI program through a webserver, the program's standard error, which contains error messages, is normally routed to the webserver's error logs. Unless you are the owner of the webserver, these logs are usually not readable. To redirect error messages to your browser (which represents the standard output stream of a CGI program), you can use the following command inside your CGI program:

```
sink(file=stdout(), type='message')
```

Remember that the error messages will not be displayed using HTML, so they may be difficult to read.

If simulating your web data through an artificially constructed `formData` list is not sufficient to resolve problems in getting your program to run properly through the webserver, you can have a form generate the `formData` list, and save it to a file; then when you are testing the program offline, you can load that copy of `formData`, giving you an interactive session with the same data as would be generated by the web page.

# Chapter 14

## Smothers

## 14.1 Smoothers

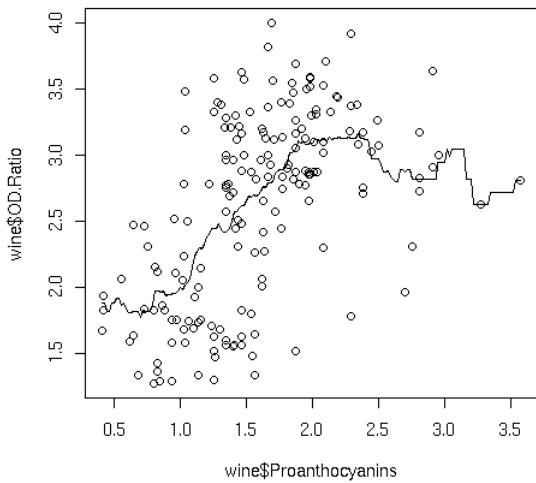
In many of the graphs we've looked at, we added a straight line representing the best linear regression line that went through the data we were plotting. Such lines can be very helpful when there are lots of points, or when outliers might detract from our seeing a relationship among the points. But plotting the best linear regression line has some limitations. For one thing, the regression has to fit all the data, so finding a good regression fit is often a compromise between being true to the data in any given range, and trying to come up with a single line that does reasonably well throughout the entire range. For some data, this may be appropriate. For example, if we know that two variables really follow a linear relationship, then we'd have to assume that deviations from that relationship are just noise, and the best straight line would be a meaningful way to display their relationship on a graph. However, situations like that are not that common. To come up with a way of visualizing relationships between two variables without resorting to a regression lines, statisticians and mathematicians have developed techniques for smoothing curves. Essentially this means drawing lines through the points based only on other points from the surrounding neighborhood, not from the entire set of points. There are many different types of smoothers available, and most of them offer an option that controls how much smoothing they will do as well as options to control the basic methods that they use, so it's usually possible to find a smoother that will work well for a particular set of data.

## 14.2 Kernel Smoothers

Kernel smoothers work by forming a weighted average of all the y-values corresponding to points whose x-values are close to the x-value of a point being plotted. The function that defines the weights is known as a kernel, and the number of points involved in the weighted average is based on a parameter known as the bandwidth. The default kernel is a box function; in other words, it simply averages together y-values which are within the specified bandwidth of a given x-value, and uses that average as the y-value for the x-value in question. With a very tiny bandwidth, this corresponds to a “connect-the-dots” type of drawing. With a very large bandwidth, it will basically estimate every y-value as the mean of all the y-values. However, even when the bandwidth is carefully chosen, using the box kernel rarely will result in a truly smooth graph. For example, consider a plot of `OD.Ratio` versus `Proanthocyanins` from the `wine` data set that we've used in previous examples. The following code produces a plot of the variables, and superimposes a line representing a box kernel smooth with the default bandwidth:

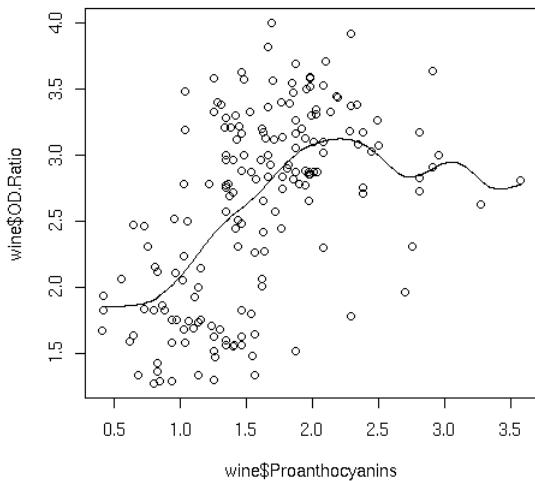
```
> plot(wine$Proanthocyanins,wine$OD.Ratio)
> lines(ksmooth(wine$Proanthocyanins,wine$OD.Ratio))
```

Here's the graph:



Notice how choppy the line is, especially where there isn't much data. That's because the box kernel is too extreme – it either adds in a point or not. So using the box kernel is like stacking up a bunch of square boxes around each point, and we don't really get a smooth result.

More commonly, kernels will have a maximum at distances that are very small, and will decrease gradually as the (absolute value) of the distance from the center of the kernel increases. This means that nearby points will have lots of influence on the weighted estimate that will be plotted, but as we move away from a particular point, the neighboring points will have less and less influence. We can modify how many points are considered through the bandwidth – including more points tends to give smoother curves that don't respond as well to local variation, while decreasing the bandwidth tends to make the curve look "choppier". One of the most common kernels used in smoothing is the Gaussian or normal kernel. This kernel is the familiar "bell curve" – largest in the middle (corresponding in this case to distances of zero from a particular point), and gradually decreasing over its supported range. The width of that range is determined by the bandwidth when using a kernel smoother. Here's the `Proanthocyanins` vs. `OD.Ratio` plot, smoothed with a normal kernel using the default bandwidth:

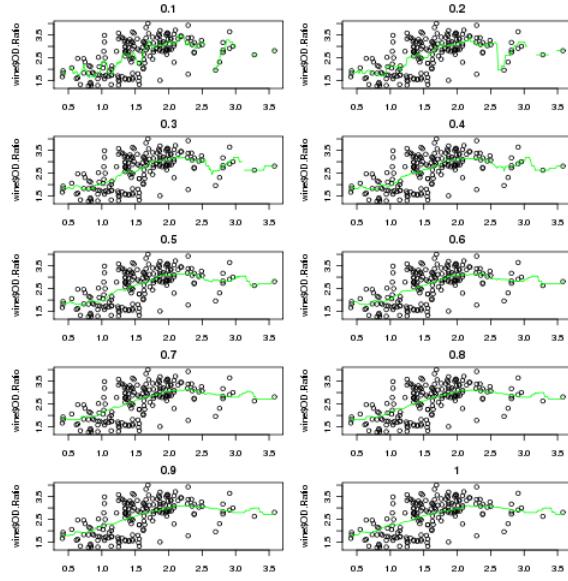


Notice the change in the line when switching to the `normal` kernel; the line is now smooth, and we can see that a linear relationship that holds up until around a Proanthocyanin concentration of about 2.

The argument that controls the size of the neighborhood that's used by `ksmooth` to estimate the smoothed value at each point is called `bandwidth`. We can examine the effect of changing the bandwidth as follows;

```
kplotfn = function(bw){
  plot(wine$Proanthocyanins,wine$OD.Ratio,main=bw)
  lines(ksmooth(wine$Proanthocyanins,wine$OD.Ratio,bandwidth=bw),col='green')
}

bws = seq(.1,1,by=.1)
par(mfrow=c(5,2),mar=c(2,4,2,1)+.1)
sapply(bws,kplotfn)
```



After adjusting the margins, the plot we got looks something like a lattice plot. But how could we produce an actual lattice plot with this data? We would need a data frame with `x` and `y` values for each bandwidth. What does `ksmooth` return?

```
> result = ksmooth(wine$Proanthocyanins,wine$OD.Ratio)
> class(result)
[1] "list"
> names(result)
[1] "x" "y"
```

Since the lengths of `x` and `y` must be equal, we can convert the output to a data frame directly. Let's write a function which will generate a data frame with `x`, `y` and `bandwidth` values for a single bandwidth:

```
kfun = function(bw)
  data.frame(bw=bw,as.data.frame(ksmooth(wine$Proanthocyanins,wine$OD.Ratio,bandwidth=bw)))
```

Notice that R realized it had to repeat the single `bw` value to make it compatible with `x` and `y`. As always, we should test the function:

```
> result = kfun(.5)
> head(result)
  bw      x      y
1 0.5 0.4100000 1.881429
2 0.5 0.4279096 1.881429
3 0.5 0.4458192 1.812500
4 0.5 0.4637288 1.812500
5 0.5 0.4816384 1.879000
6 0.5 0.4995480 1.879000
```

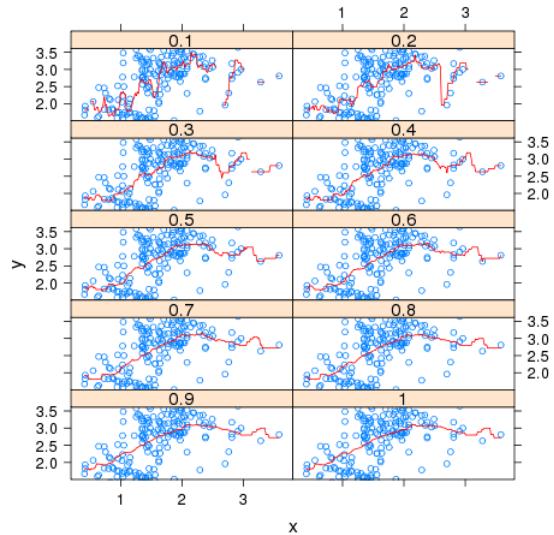
Now we can create a list of data frames, one for each bandwidth:

```
> frames = lapply(seq(.1,1,by=.1),kfun)
> allbws = do.call(rbind,frames)
> dim(allbws)
[1] 1780      3
```

Notice the use of the `do.call` function, which is used here to call `rbind` with the data frames for all of the different bandwidths.

To get both points and lines on each panel, we can create a custom panel function:

```
mypanel = function(x,y,...){
  panel.xyplot(wine$Proanthocyanins,wine$OD.Ratio);
  panel.xyplot(x,y,type='l',col='red')
}
xyplot(y~x|factor(bw),data=allbws,type='l',as.table=TRUE,layout=c(2,5),panel=mypanel)
```



The resulting plot appears below:

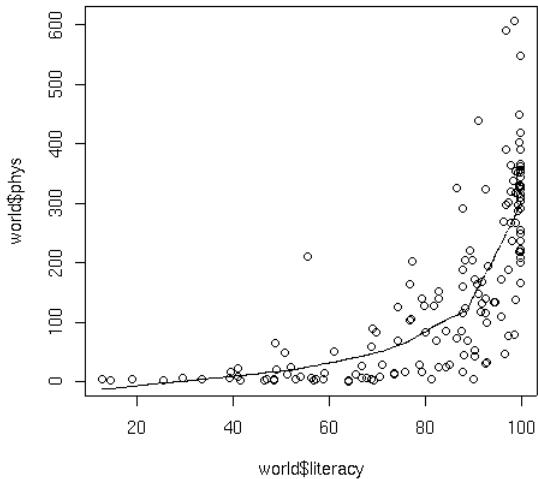
### 14.3 Locally Weighted Regression Smoothers

Another approach that is often used to smooth curves is locally weighted regression. Instead of taking a weighted average of y-values near the x-values we want to plot, the nearby points are used in a (usually quadratic) weighted regression, and predicted values from these local regressions are used as the y-values that are plotted. The `lowess` function in R implements this technique by using the reciprocal of the residuals of successive fits as the weights, downgrading those points that don't contribute to a smooth fit. In the `lowess` function, the argument `f=` specifies the fraction of the data to be used in the local regressions. Specifying a larger value results in a smoother curve.

To illustrate, consider a plot of `literacy` versus `phys`, the number of physicians per 100000 people, from the `world` data set that we've used in previous examples. The following code produces a plot of the data with a lowess smoothed curve superimposed:

```
> plot(world$literacy,world$phys)
> lines(lowess(world$literacy,world$phys))
```

The graph appears below:

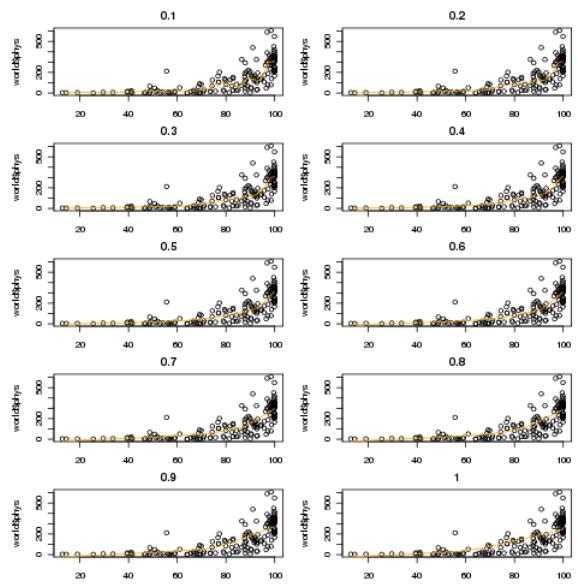


The argument to `lowess` that controls the level of smoothing is `f`, the fraction of the data which will be used in the local regressions. Let's compare the results of smoothing the literacy/physician curve using different values of `f`

```
lplotfn = function(f){
  plot(world$literacy,world$phys,main=f)
  lines(lowess(world$literacy,world$phys,f=f),col='orange')
}

fs = seq(.1,1,by=.1)
par(mfrow=c(5,2),mar=c(2,4,2,1)+.1)
sapply(fs,lplotfn)
```

For this particular data set, there's not much difference in the appearance of the smoothed curve, although there is a bit of noise when using the lowest value of `f`



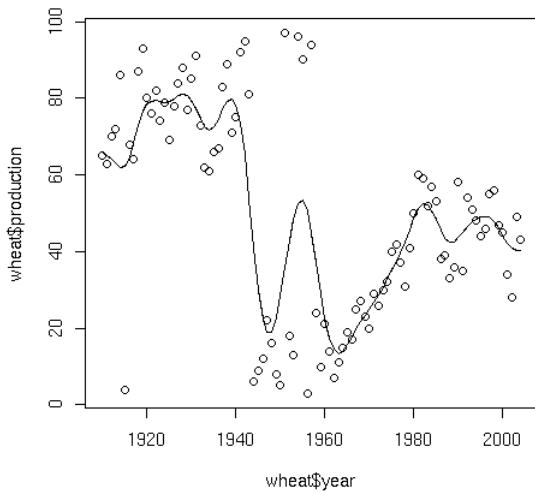
## 14.4 Spline Smoothers

Another type of smoothing is known as spline smoothing, named after a tool formerly used by draftsmen. A spline is a flexible piece of metal (usually lead) which could be used as a guide for drawing smooth curves. A set of points (known as knots) would be selected, and the spline would be held down at a particular x,y point, then bent to go through the next point, and so on. Due to the flexibility of the metal, this process would result in a smooth curve through the points.

Mathematically, the process can be reproduced by choosing the knot points and using (usually cubic) regression to estimate points in between the knots, and using calculus to make sure that the curve is smooth whenever the individual regression lines are joined together. The `smooth.spline` function in R performs these operations. The degree of smoothness is controlled by an argument called `spar=`, which usually ranges between 0 and 1. To illustrate, consider a data set consisting of the wheat production of the United States from 1910 to 2004. The data set can be found at <http://www.stat.berkeley.edu/classes/s133/data/wheat.txt>. The following lines will produce a plot of the data, and superimpose a spline smooth.

```
> wheat = read.table('http://www.stat.berkeley.edu/classes/s133/data/wheat.txt',
+                      header=TRUE)
> plot(wheat$year,wheat$production)
> lines(smooth.spline(wheat$year,wheat$production))
```

Here's the result:



The amount of smoothing that `smooth.spline` applies is controlled by the argument `spar`. We can examine the effect of `spar` by producing several plots, each with a different value of `spar`.

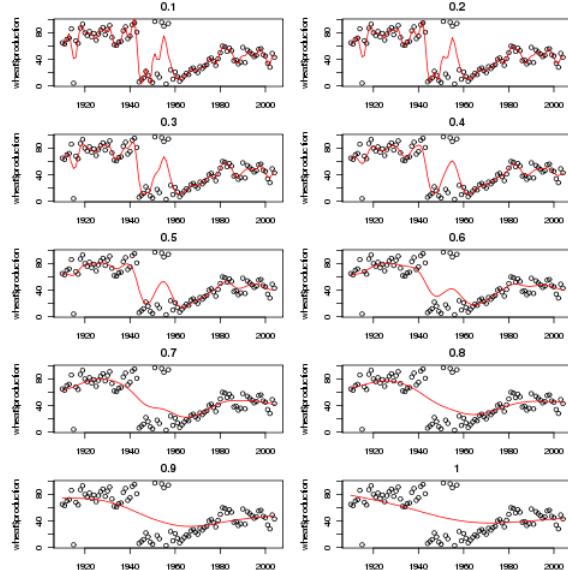
```
plotfn = function(spar){
  plot(wheat$year,wheat$production,main=spar)}
```

```

    lines(smooth.spline(wheat$year,wheat$production,spar=spar),col='red')
}
spars = seq(.1,1,by=.1)
par(mfrow=c(5,2),mar=c(2,4,2,1)+.1)
sapply(spars,plotfn)

```

The plot appears below:



## 14.5 Supersmooth

While most smoothers require specification of a bandwidth, fraction of data, or level of smoothing, supersmooth is different in that it figures these things out for itself. Thus, it's an excellent choice for situations where smoothing needs to be done without any user intervention. Supersmooth works by performing lots of simple local regression smooths, and, at each x-value it uses those smooths to decide the best y-value to use. In R, supersmooth is made available through the `supsmu` function.

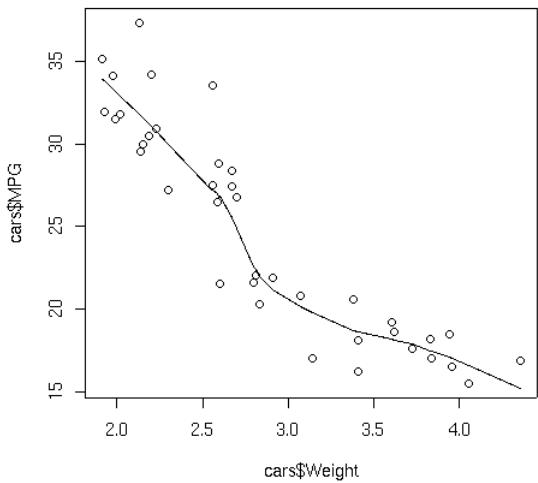
To illustrate, consider the car data which we used earlier when we were studying cluster analysis. The following lines produce a plot of weight versus MPG, and superimposes a supersmooth line.

```

> plot(cars$Weight,cars$MPG)
> lines(supsmu(cars$Weight,cars$MPG))

```

The plot appears below:

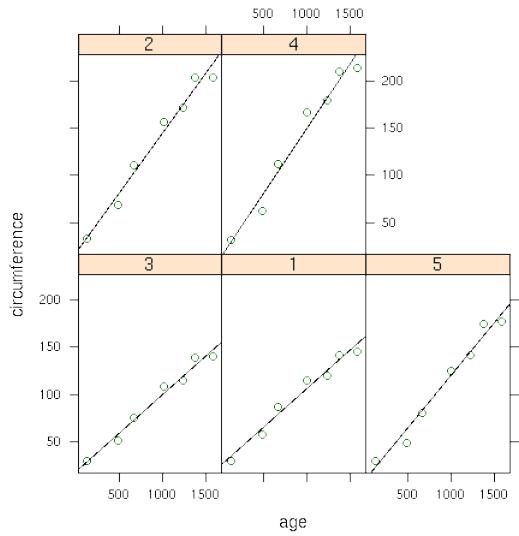


## 14.6 Smoothers with Lattice Plots

When working with lattice graphics, we've already seen the use of `panel.lmline`, which displays the best regression line in each panel of a lattice plot. A similar function, `panel.loess`, is available to superimpose a locally weighted regression smoother in each panel of a plot. As a simple illustration, consider the built-in `Orange` data set, which has information about the age and circumference of several orange trees. First, let's look at a plot with the best regression line smoother superimposed on the plot of `age` versus `circumference` for each `Tree`:

```
> library(lattice)
> xyplot(circumference~age|Tree,data=Orange,
+         panel=function(x,y,...){panel.xyplot(x,y,...);panel.lmline(x,y,...)})
```

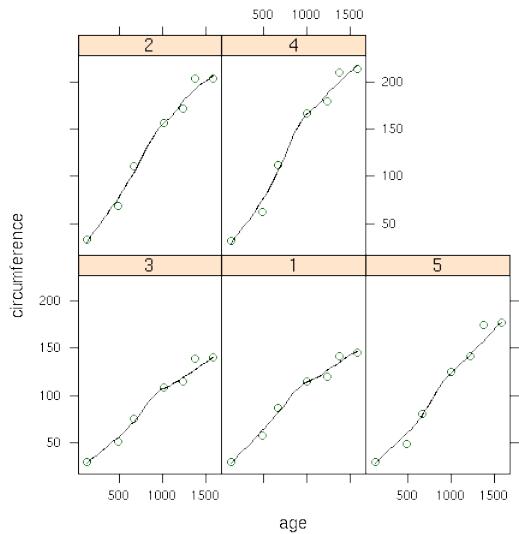
Here's the plot:



To create the same plot, but using the `panel.loess` function, we can use the following:

```
> xyplot(circumference~age|Tree,data=Orange,
+         panel=function(x,y,...){panel.xyplot(x,y,...);panel.loess(x,y,...)})
```

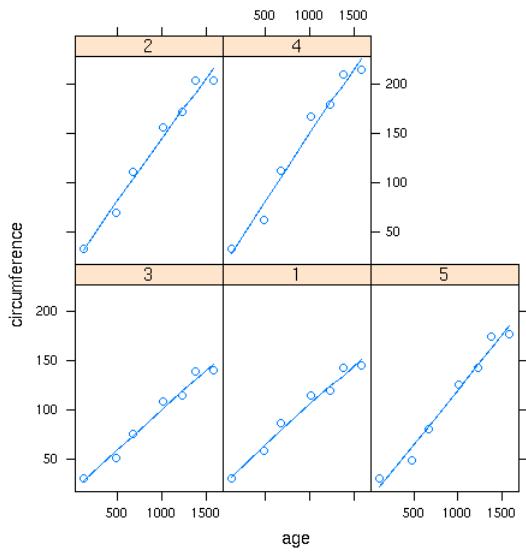
Here's how the plot looks:



If a `panel.` function doesn't exist for a smoother you'd like to use, you can use the `panel.lines` function to plot it directly:

```
> xyplot(circumference~age|Tree,data=Orange,
+         panel=function(x,y,...){panel.xyplot(x,y,...);
+                               z=supsmu(x,y);panel.lines(z$x,z$y,...)})
```

In this case supersmooth came closer to a straight line than lowess.



# Chapter 15

## Linear Regression

## 15.1 Linear Regression

Linear regression is a very popular procedure for modeling the value of one variable on the value(s) of one or more other variables. The variable that we're trying to model or predict is known as the dependent variable, and the variables that we use to make predictions are known as independent variables, or covariates. Linear regression makes the assumption that the changes in the dependent variable can be modeled as a monotonic linear function of the independent variables; that is, we assume that a change of a certain amount in the independent variables will result in a change in the dependent variable, and the amount of change in the dependent variable is constant across the range of the independent variables. As a simple example, suppose we're interested in the relationship between the horsepower of a car (the independent variable) and the miles per gallon of gas (MPG) of the car. When we fit a linear regression model, we're saying that a change of one horsepower will have the same effect on the MPG regardless of the value of horsepower that we're changing. In other words, a linear regression model would assume that if we had a car with 100 horsepower, and compared it to a car with 101 horsepower, we'd see the same difference in MPG as if we had a car with 300 horsepower and compared it to a car with 301 horsepower. Relationships like this often hold for a limited range of independent variable values, but the linear regression model assumes that it applies for the entire range of the independent variables.

Even with these limitations, linear regression has proven itself to be a very valuable tool for modeling, and it's widely used in many branches of research. There are also a variety of diagnostic measures, both numerical and graphical, that can help us to determine whether the regression is doing a good job, so it's not unreasonable that many people use linear regression as their first tool when trying to model a variable of interest.

## 15.2 The `lm` command

The `lm` command uses the model-based formula interface that we've already seen in the context of lattice graphics. The dependent variable is placed on the left-hand side of the tilde (~), and the independent variables are placed on the right-hand side, joined together with plus signs (+). When you want to use all the variables in a data frame (except for the dependent variable) as independent variables, you can use a period (.) for the right-hand side of the equation.

These models correspond to a mathematical model that looks like this:

$$y_i = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + e_i \quad \text{for } i = 1, \dots, n \quad (15.1)$$

The  $\beta$ s represent coefficients that measure how much the dependent variable changes for each unit of change in the independent variable, and are often referred to as the slopes. The term  $\beta_0$  is often known as the intercept. (To omit the intercept in a formula in R, insert the term `-1`.) The  $e$ 's represent the part of the observed dependent variable that can't be explained by the regression model, and in order to do hypothesis testing, we assume that these errors follow a normal distribution. For each term in the model, we test the hypothesis

that the  $\beta$  corresponding to that term is equal to 0, against the alternative that the  $\beta$  is different from 0.

To illustrate the use of the `lm` command, we'll construct a regression model to predict the level of `Alcohol` in the `wine` data set, using several of the other variables as independent variables. First, we'll run `lm` to create an `lm` object containing all the information about the regression:

```
> wine.lm = lm(Alcohol~Malic.acid+Alkalinity.ash+Proanthocyanins+Color.intensity+
+               OD.Ratio+Proline,data=wine[-1])
```

To see a very brief overview of the results, we can simply view the `lm` object:

```
> wine.lm
```

Call:

```
lm(formula = Alcohol ~ Malic.acid + Alkalinity.ash + Proanthocyanins +      Color.intensi
```

Coefficients:

	(Intercept)	Malic.acid	Alkalinity.ash	Proanthocyanins
Alcohol	11.333283	0.114313	-0.032440	-0.129636
Color.intensity	0.158520	0.225453	0.001136	

To get more information about the model, the `summary` function can be called:

```
> summary(wine.lm)
```

Call:

```
lm(formula = Alcohol ~ Malic.acid + Alkalinity.ash + Proanthocyanins +
    Color.intensity + OD.Ratio + Proline, data = wine[-1])
```

Residuals:

Min	1Q	Median	3Q	Max
-1.502326	-0.342254	0.001165	0.330049	1.693639

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	11.3332831	0.3943623	28.738	< 2e-16 ***
Malic.acid	0.1143127	0.0397878	2.873	0.00458 **
Alkalinity.ash	-0.0324405	0.0137533	-2.359	0.01947 *
Proanthocyanins	-0.1296362	0.0846088	-1.532	0.12732
Color.intensity	0.1585201	0.0231627	6.844	1.32e-10 ***
OD.Ratio	0.2254528	0.0834109	2.703	0.00757 **
Proline	0.0011358	0.0001708	6.651	3.76e-10 ***
---				

```
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 0.5295 on 171 degrees of freedom  
Multiple R-Squared: 0.5889, Adjusted R-squared: 0.5745  
F-statistic: 40.83 on 6 and 171 DF, p-value: < 2.2e-16
```

The probability levels in the last column of the bottom table are for testing the null hypotheses that the slopes for those variables are equal to 0; that is, we're testing the null hypothesis that changes in the independent variable will not result in a linear change in the dependent variable. We can see that most of the variables in the model do seem to have an effect on the dependent variable.

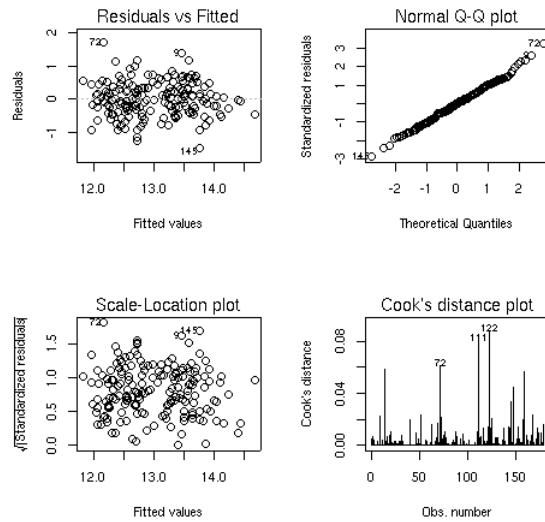
One useful measure of the efficacy of a regression model is the multiple R-Squared statistic. This essentially measures the squared correlation of the dependent variable values with the values that the model would predict. The usual interpretation of this statistic is that it measures the fraction of variability in the data that is explained by the model, so values approaching 1 mean that the model is very effective. Since adding more variables to a model will always inflate the R-Squared value, many people prefer using the adjusted R-Squared value, which has been adjusted to account for the number of variables included in the model.

When you pass a model object to the `plot` function, it will display one or more plots that the author of the model fitting function felt were appropriate for studying the effectiveness of the model. For the `lm` object, four plots are created by default:

1. A plot of residuals versus fitted (predicted) values - The residuals are the part of the dependent variable that the model couldn't explain, and they are our best available estimate of the error term from the regression model. They're calculated by subtracting the predicted value from the actual value of the dependent variable. Under the usual assumptions for the linear regression model, we don't expect the variability of the residuals to change over the range of the dependent variable, so there shouldn't be any discernable pattern to this plot. Note that outliers in the plot will be labeled by their observation number making it easy to track them down.
2. A normal quantile-quantile plot of the standardized residuals - For the probabilities we saw in the summary table to be accurate, we have assumed that the errors of the model follow a normal distribution. Thus, we'd expect a normal quantile-quantile plot of the residuals to follow a straight line. Deviations from a straight line could mean that the errors don't follow a normal distribution.
3. A scale-location plot - This plot is similar to the residuals versus fitted values plot, but it uses the square root of the standardized residuals. Like the first plot, there should be no discernable pattern to the plot.
4. A Cook's distance plot - Cook's distance is a statistic that tries to identify points which have more influence than other points. Generally these are points that are distant from other points in the data, either for the dependent variable or one or

more independent variables. Each observation is represented as a line whose height is indicative of the value of Cook's distance for that observation. There are no hard and fast rules for interpreting Cook's distance, but large values (which will be labeled with their observation numbers) represent points which might require further investigation.

Here are the four plots for the `wine.lm` object:



## 15.3 Using the model object

The design of the R modeling functions makes it very easy to do common tasks, regardless of the method that was used to model the data. We'll use `lm` as an example, but most of these techniques will work for other modeling functions in R. We've already seen that the `plot` function will produce useful plots after a model is fit. Here are some of the other functions that are available to work with modeling objects. In each case, the modeling object is passed to the function as its first argument.

1. Coefficients - The `coef` function will return a vector containing the coefficients that the model estimated, in this case, the intercept and the slopes for each of the variables:

```
> coef(wine.lm)
(Intercept)      Malic.acid   Alkalinity.ash Proanthocyanins Color.intensity
11.333283116    0.114312670   -0.032440473   -0.129636226    0.158520051
          OD.Ratio        Proline
          0.225452840     0.001135776
```

2. Predicted Values - The `predict` function, called with no additional arguments, will return a vector of predicted values for the observations that were used in the modeling process. To get predicted values for observations not used in building the model, a data

frame containing the values of the observations can be passed to `predict` through the `newdata=` argument. The variables in the data frame passed to `predict` must have the same names as the variables used to build the model. For example, to get a predicted value of `Alcohol` for a mythical wine, we could use a statement like this:

```
> predict(wine.lm,newdata=data.frame(Malic.acid=2.3,Alkalinity.ash=19,
+                                         Proanthocyanins=1.6,Color.intensity=5.1,OD.Ratio=2.6,Proline=746.9))
[1] 12.88008
```

3. Residuals - The `residuals` function will return a vector of the residuals from a model.

In addition, the `summary` function, which is usually used to display a printed summary of a model, often contains useful information. We can see what it contains by using the `names` function:

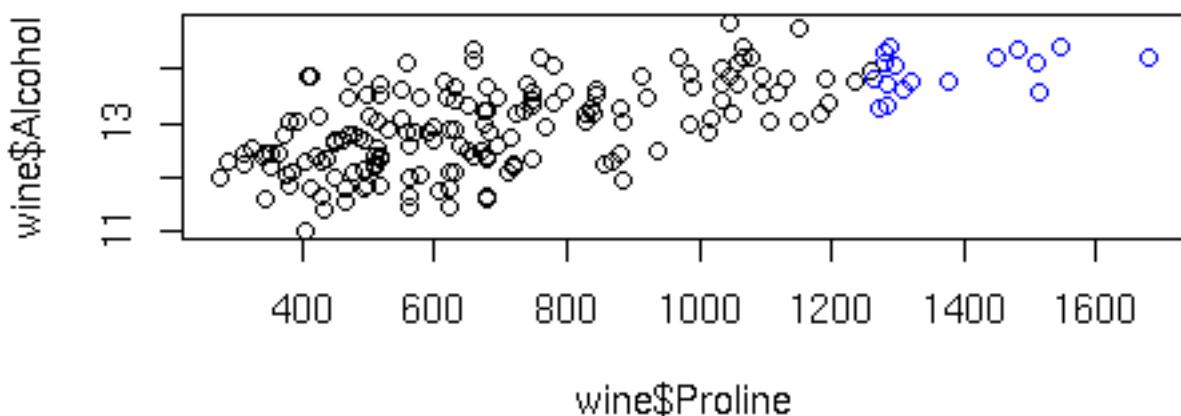
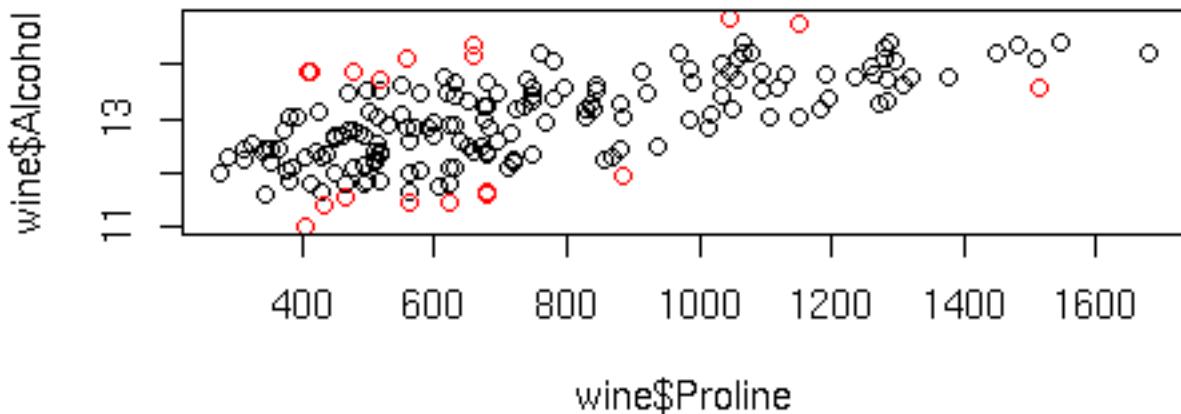
```
> names(summary(wine.lm))
[1] "call"           "terms"          "residuals"       "coefficients"
[5] "aliased"        "sigma"          "df"              "r.squared"
[9] "adj.r.squared"  "fstatistic"     "cov.unscaled"
```

## 15.4 Regression Diagnostics

Two statistics which have proven to be useful in identifying influential observations are Cook's distance and a statistic known as the hat statistic. Cook's distance is calculated for each observation by comparing the results of the regression with that observation included to the results when that particular observation is removed. Thus, it can find observations which are outliers with regard to both the dependent variables and the independent variables. In R, the `cooks.distance` function calculates the statistics given an `lm` model object. The hat statistic is based entirely on the independent variables of the model, so it focuses on observations which are distant from the others with regard to the independent variables in the model. In R, the `lm.influence` function will return a list, including a component named `hat`, which contains the hat statistic. For simple regressions, with just one independent variable, influential observations are usually at the far reaches of the range of either the dependent or independent variables. For example, for the `wine.lm` model, `Proline` was one of the independent variables which seemed effective in predicting alcohol. Let's perform a simple regression using this variable, and then plot the results, highlighting those points that had unusually high Cook's distances or hat statistic values:

```
> simple.lm = lm(Alcohol~Proline,data=wine)
> cooks = cooks.distance(simple.lm)
> hat = lm.influence(simple.lm)$hat
> par(mfrow=c(2,1))
> plot(wine$Proline,wine$Alcohol,col=ifelse(cooks > quantile(cooks,.90),'red','black'))
> plot(wine$Proline,wine$Alcohol,col=ifelse(hat > quantile(hat,.90),'blue','black'))
```

The plots are displayed below:

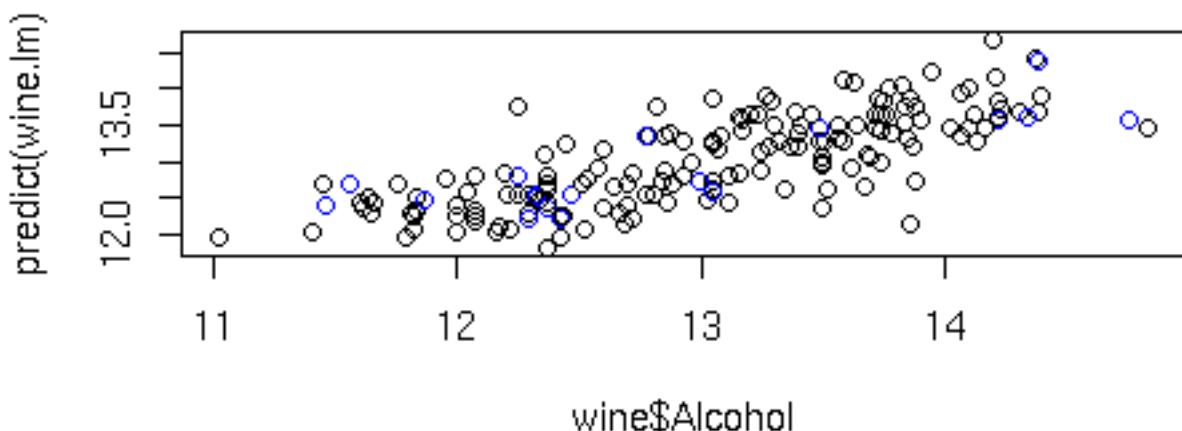
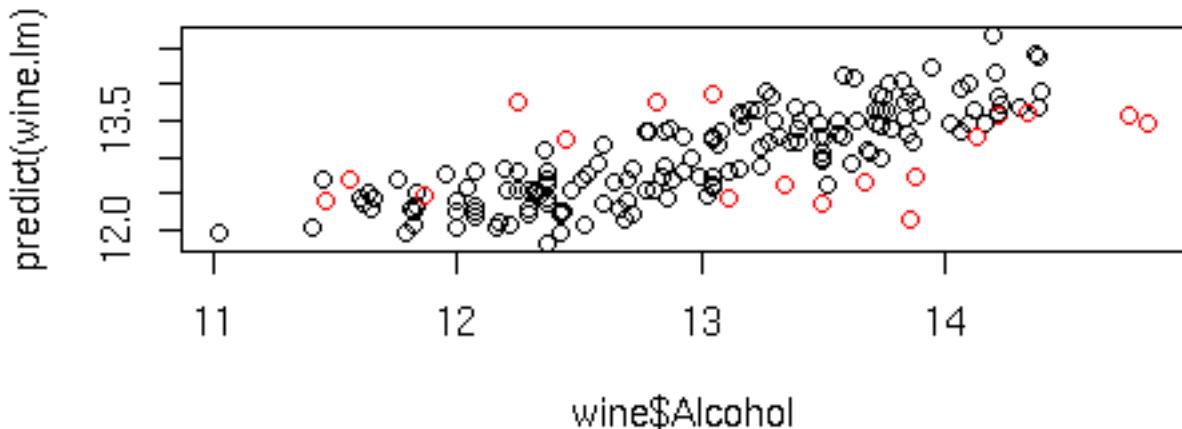


In the top graph, the points displayed in red represent the observations with large Cook's distances; in the bottom graph, the blue points are those with high hat statistic values. Of course, with more variables in the model, things are not so simple. Let's look at a plot of predicted values versus actual values for the full regression model for this data, using the same coloring conventions:

```
> cooks = cooks.distance(wine.lm)
> hat = lm.influence(wine.lm)$hat
> par(mfrow=c(2,1))
```

```
> plot(wine$Alcohol,predict(wine.lm),col=ifelse(cooks > quantile(cooks,.9),'red','black'))  
> plot(wine$Alcohol,predict(wine.lm),col=ifelse(hat > quantile(hat,.9),'blue','black'))
```

Here are the plots:



The extreme Cook's distance points seem to congregate at the outer edges of the predicted values, but the extreme hat statistics points don't follow a simple pattern. In practice, many statisticians use the rule of thumb that Cook's distances bigger than the 10th percentile of an F distribution with p and n-p degrees of freedom represent potential problems, where n is the number of observations, and p is the number of parameters estimated. For the wine data, that cutoff point can be calculated as follows:

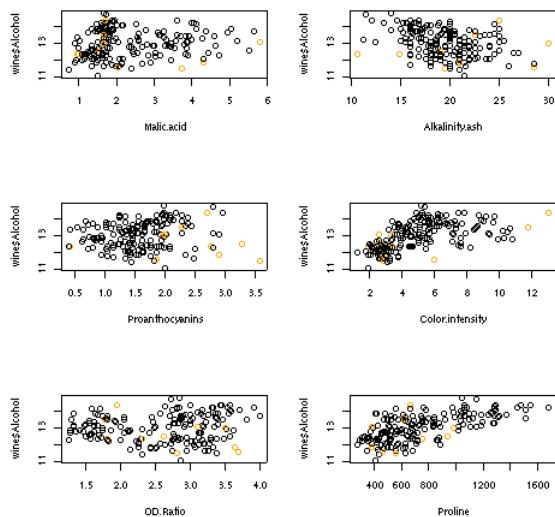
```
> qf(.1,7,178-7)
[1] 0.4022056
```

In fact, for this data set none of the Cook's distances are greater than this value.

For the hat statistic, a cutoff of  $2 * p/n$  has been proposed; for the wine data this corresponds to a value of 0.079. With the wine example, there are ten such points. Plotting each independent variable against the dependent variable, and highlighting the extreme points in orange helps to show where these points are:

```
> par(mfrow=c(3,2))
> sapply(names(coef(wine.lm)[-1]),
+         function(x)plot(wine[[x]],wine$Alcohol,
+                          col=ifelse(hat >.0786 , 'orange','black'),xlab=x))
```

Here are the plots:



## 15.5 Collinearity

Another problem which might occur when using linear regression is known as collinearity. This problem occurs when the independent variables are so highly correlated that they contain redundant information, which confuses the regression process. When data is collinear, the standard errors of the parameter estimates get very large, and removing one or two variables may make the coefficients change dramatically. In addition, collinearity can mask important relationships in the data. The classic data set to illustrate collinearity is known as the Longley data set, available in R under the name `longley`. This data set contains a variety of measurements about the population of the US in an attempt to predict employment. Let's take a look at the result of regressing `Employed` against the other variables in the Longley data set:

```

> lreg = lm(Employed ~ ., data=longley)
> summary(lreg)

Call:
lm(formula = Employed ~ ., data = longley)

Residuals:
    Min      1Q  Median      3Q     Max 
-0.41011 -0.15767 -0.02816  0.10155  0.45539 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -3.482e+03  8.904e+02 -3.911 0.003560 ** 
GNP.deflator 1.506e-02  8.492e-02  0.177 0.863141    
GNP          -3.582e-02  3.349e-02 -1.070 0.312681    
Unemployed   -2.020e-02  4.884e-03 -4.136 0.002535 ** 
Armed.Forces -1.033e-02  2.143e-03 -4.822 0.000944 ***  
Population   -5.110e-02  2.261e-01 -0.226 0.826212    
Year         1.829e+00  4.555e-01  4.016 0.003037 ** 
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1    1

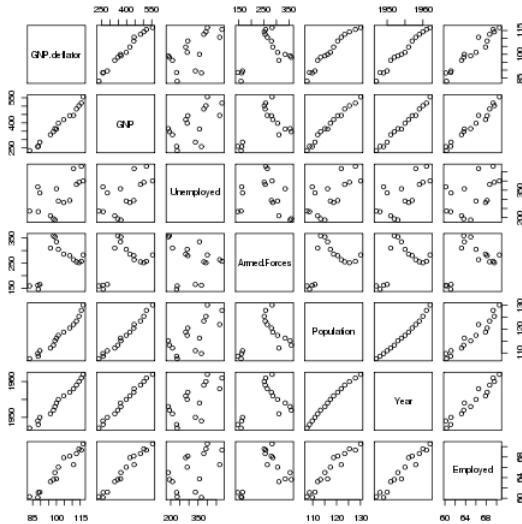
Residual standard error: 0.3049 on 9 degrees of freedom
Multiple R-squared:  0.9955, Adjusted R-squared:  0.9925 
F-statistic: 330.3 on 6 and 9 DF,  p-value: 4.984e-10

```

On the surface, nothing seems wrong – in fact, with an adjusted R-squared of .9925, it seems great. We can see the problem with the data by looking at the pairwise scatterplots, using the `pairs` function:

```
> pairs(longley)
```

Here's the plot:

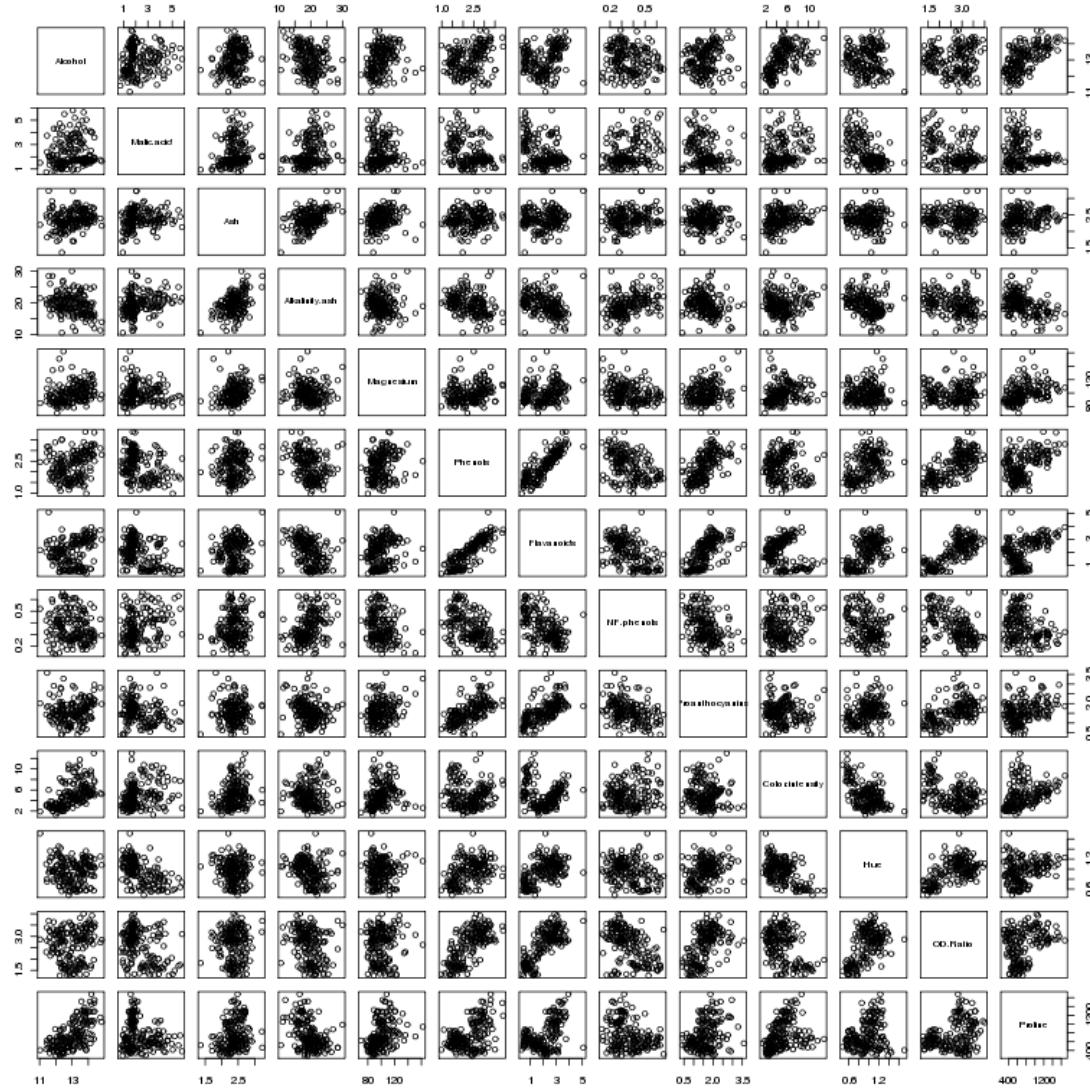


Many of the variables seem to be correlated with each other, so it's difficult to see which is causing the problem. A statistic known as VIF (Variance Inflation Factor) can be very useful in situations like this. In R, the `vif` function in the `car` package will provide this statistic. Before using `vif` on the Longley data, let's look at the wine data we used previously:

```
> wine.lm = lm(Alcohol~., data=subset(wine, select=-Cultivar))
> library(car)
> vif(wine.lm)

      Malic.acid          Ash  Alkalinity.ash        Magnesium        Phenols
    1.575916      2.180108      2.179282      1.417855     4.330552
  Flavanoids      NF.phenols Proanthocyanins Color.intensity        Hue
    7.029040      1.793883      1.947243      2.493007     2.542273
    OD.Ratio       Proline
    3.736818      2.441810
```

None of the inflation factors is bigger than 10, which indicates collinearity is not a problem with the data set, confirmed by looking at the pairs plot for the wine data set:



There does seem to be a linear relationship between Flavanoids and Phenols – not surprisingly those two variables have the highest VIFs.

Now let's return to the Longley data.

```
> vif(lreg)
   GNP.deflator          GNP  Unemployed Armed.Forces  Population      Year
   135.53244    1788.51348     33.61889      3.58893    399.15102    758.98060
```

The two largest VIFs are for GNP and Year. Let's eliminate them from the model, and see how the VIFs change:

```
> lreg1 = lm(Employed ~ ., data=subset(longley, select=-c(GNP,Year)))
> summary(lreg1)
```

```
Call:  
lm(formula = Employed ~ ., data = subset(longley, select = -c(GNP,  
Year)))
```

Residuals:

Min	1Q	Median	3Q	Max
-0.6561730	-0.2576601	-0.0008123	0.1213544	1.2225443

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )		
(Intercept)	13.781314	6.886470	2.001	0.070657 .		
GNP.deflator	0.207046	0.081376	2.544	0.027270 *		
Unemployed	-0.012412	0.002780	-4.465	0.000955 ***		
Armed.Forces	-0.005968	0.003325	-1.795	0.100170		
Population	0.306601	0.123795	2.477	0.030755 *		
---						
Signif. codes:	0 ***	0.001 **	0.01 *	0.05 .	0.1	1

Residual standard error: 0.5671 on 11 degrees of freedom  
Multiple R-squared: 0.9809, Adjusted R-squared: 0.9739  
F-statistic: 141.1 on 4 and 11 DF, p-value: 2.26e-09

```
> vif(lreg1)  
GNP.deflator Unemployed Armed.Forces Population  
35.970754 3.147600 2.497795 34.588299
```

The reduced model is probably more realistic than the full model, but GNP.deflator and Population are still highly correlated

```
> with(longley, cor(GNP.deflator,Population))  
[1] 0.9791634
```

Removing GNP.deflator results in a model that seems to make sense:

```
> lreg2 = lm(Employed ~ ., data=subset(longley, select=-c(GNP,Year,GNP.deflator)))  
> summary(lreg2)
```

Call:

```
lm(formula = Employed ~ ., data = subset(longley, select = -c(GNP,  
Year, GNP.deflator)))
```

Residuals:

Min	1Q	Median	3Q	Max
-----	----	--------	----	-----

```

-1.3835 -0.2868 -0.1353  0.3596  1.3382

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.323091   4.211566 -0.314  0.75880
Unemployed   -0.012292   0.003354 -3.665  0.00324 ** 
Armed.Forces -0.001893   0.003516 -0.538  0.60019
Population     0.605146   0.047617 12.709 2.55e-08 *** 
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

```

Residual standard error: 0.6843 on 12 degrees of freedom  
 Multiple R-squared: 0.9696, Adjusted R-squared: 0.962  
 F-statistic: 127.7 on 3 and 12 DF, p-value: 2.272e-09

```

> vif(lreg2)
      Unemployed Armed.Forces Population
      3.146686     1.918225    3.514335

```

Now let's look at some alternatives to ordinary linear regression.

## 15.6 Generalized Additive Models (gam)

One of the most useful alternative methods to regression is known as a generalized additive model. Instead of fitting a single linear parameter to try to explain the relationship between independent variables and dependent variables, GAM models perform spline smooths on selected variables, and use these smoothed versions of the independent variables to try to explain the values of the dependent variables. To try to make the information from the analysis similar to the familiar `lm` output, an estimated number of degrees of freedom is calculated for each variable, based on how different the fitted spline smooth for that variable is from the strictly linear relationship that `lm` uses for prediction, and an F-statistic is produced for each independent variable to test whether the smoothed version of the variable made a significant contribution to the predicted value of the dependent variable. In R, the `gam` function is provided by the `mgcv` library. This library also provides the `s` function, which is used by `gam` to identify the variables that should be smoothed before they are used to predict the dependent variable. Without prior knowledge, it's not unreasonable to try smoothing on all the variables. We can fit a `gam` model by using the same formula as we used with `lm`, passing each variable in the model to the `s` function:

```

> library(mgcv)
> wine.gam = gam(Alcohol~s(Malic.acid)+s(Alkalinity.ash)+ 
+                 s(Proanthocyanins)+s(Color.intensity)+ 
+                 s(OD.Ratio)+s(Proline), data=wine[-1])

```

```

> wine.gam

Family: gaussian
Link function: identity

Formula:
Alcohol ~ s(Malic.acid) + s(Alkalinity.ash) + s(Proanthocyanins) +
    s(Color.intensity) + s(OD.Ratio) + s(Proline)

```

Estimated degrees of freedom:  
 1 7.920717 3.492826 4.022189 1 3.567478 total = 22.00321

GCV score: 0.2314599

Like the `lm` function, `gam` provides a more familiar table when the `summary` method is invoked:

```

> summary(wine.gam)

Family: gaussian
Link function: identity

Formula:
Alcohol ~ s(Malic.acid) + s(Alkalinity.ash) + s(Proanthocyanins) +
    s(Color.intensity) + s(OD.Ratio) + s(Proline)

```

Parametric coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	13.00062	0.03376	385.1	<2e-16 ***

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Approximate significance of smooth terms:

	edf	Est.rank	F	p-value
s(Malic.acid)	1.000	1.000	14.133	0.000240 ***
s(Alkalinity.ash)	7.921	9.000	4.403	3.84e-05 ***
s(Proanthocyanins)	3.493	9.000	1.844	0.064387 .
s(Color.intensity)	4.022	9.000	8.391	3.66e-10 ***
s(OD.Ratio)	1.000	1.000	2.246	0.135990
s(Proline)	3.567	9.000	5.508	1.42e-06 ***

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

R-sq.(adj) = 0.692 Deviance explained = 72.9%

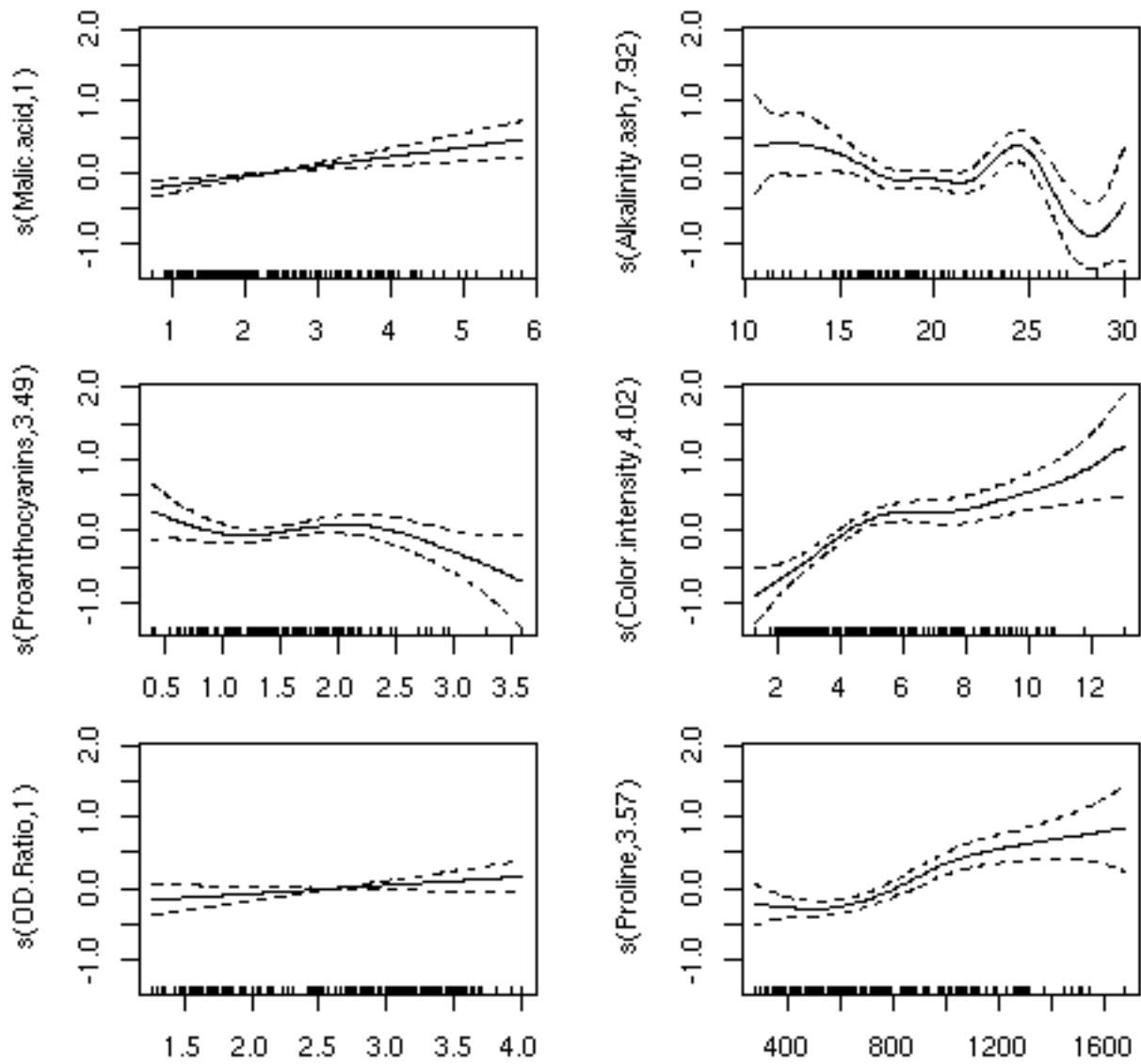
```
 GCV score = 0.23146  Scale est. = 0.20285  n = 178
```

The relative importance of the variables has changed somewhat from the linear regression results. Notice that the adjusted R-squared value is 0.692, as compared to 0.575 for linear regression, showing an improvement in prediction by using the smoothed versions of the independent variable.

Applying the `plot` function to a `gam` model is often the most useful part of the analysis. The plots produced show how the independent variable was smoothed before fitting, so a straight (or nearly straight) line for a particular variable means a truly linear relationship was found, while deviations from linearity describe the nature of non-linear relationships that exist. Here's the results of using `plot` on our `gam` model. For convenience, I've put all the plots in a single graphic; in practice, you might want to examine each plot separately. I've used the `par` function to adjust the margins so that the individual plots will be a little larger:

```
> par(mfrow=c(3,2),mar=c(2,4,1,2)+.1,oma=c(0,0,4,0),xpd=FALSE)
> plot(wine.gam)
```

Here's the plot:



For the variables `Malic.acid` and `OD.ratio`, the relationships do seem to be linear; this is supported by the fact that `gam` only used a single degree of freedom to fit these terms. For some of the other variables, it's clear that linear relationships hold only over a limited range of the data. The `Alkalinity.ash` plot is particularly interesting, but it may indicate oversmoothing.

## 15.7 Recursive Partitioning

We've already seen how recursive partitioning can be used for classification, but it can also be used for regression if the dependent variable passed to `rpart` is not a factor. When used for regression, `rpart` follows a similar strategy as for classification; each variable is tested for all possible splits, looking for large separation between the dependent variable values for one side of the split as opposed to the other. As is the case for classification, `rpart` presents its results as a tree, with terminal nodes representing the best prediction the model can provide. Here are the results of using recursive partitioning on the `wine` data frame to predict `Alcohol`. I'm using a period on the right hand side of the model to indicate that `rpart` should consider all of the variables in the data frame (except `Cultivar`):

```
> library(rpart)
> wine.rpart = rpart(Alcohol ~ . ,data=wine[-1])
> wine.rpart
n= 178

node), split, n, deviance, yval
  * denotes terminal node

1) root 178 116.654000 13.00062
  2) Color.intensity< 3.325 50  9.161498 12.13980
    4) Ash>=2.41 13  1.269369 11.86846 *
    5) Ash< 2.41 37  6.598724 12.23514 *
  3) Color.intensity>=3.325 128  55.969350 13.33687
  6) Proline< 900 79  28.974900 13.05013
    12) Color.intensity< 8.315 61  21.586760 12.93197
      24) Proline< 722.5 43  14.291710 12.80209
        48) Malic.acid< 3.1 27  8.142067 12.62889
          96) NF.phenols>=0.33 14  2.388493 12.30929 *
          97) NF.phenols< 0.33 13  2.783477 12.97308 *
        49) Malic.acid>=3.1 16  3.972794 13.09437 *
      25) Proline>=722.5 18  4.837111 13.24222 *
    13) Color.intensity>=8.315 18  3.650294 13.45056 *
  7) Proline>=900 49  10.025970 13.79918
  14) Color.intensity< 4.44 10  0.787410 13.27700 *
  15) Color.intensity>=4.44 39  5.812631 13.93308 *
```

Since `rpart` doesn't actually estimate any coefficients, we can't produce a table of hypothesis tests as we did for `lm` or `gam`, but we can get a sort of multiple R-squared value by squaring the correlation between the true value of the dependent variable and the value that `rpart` predicts:

```
> cor(wine$Alcohol,predict(wine.rpart))^2
```

```
[1] 0.7248247
```

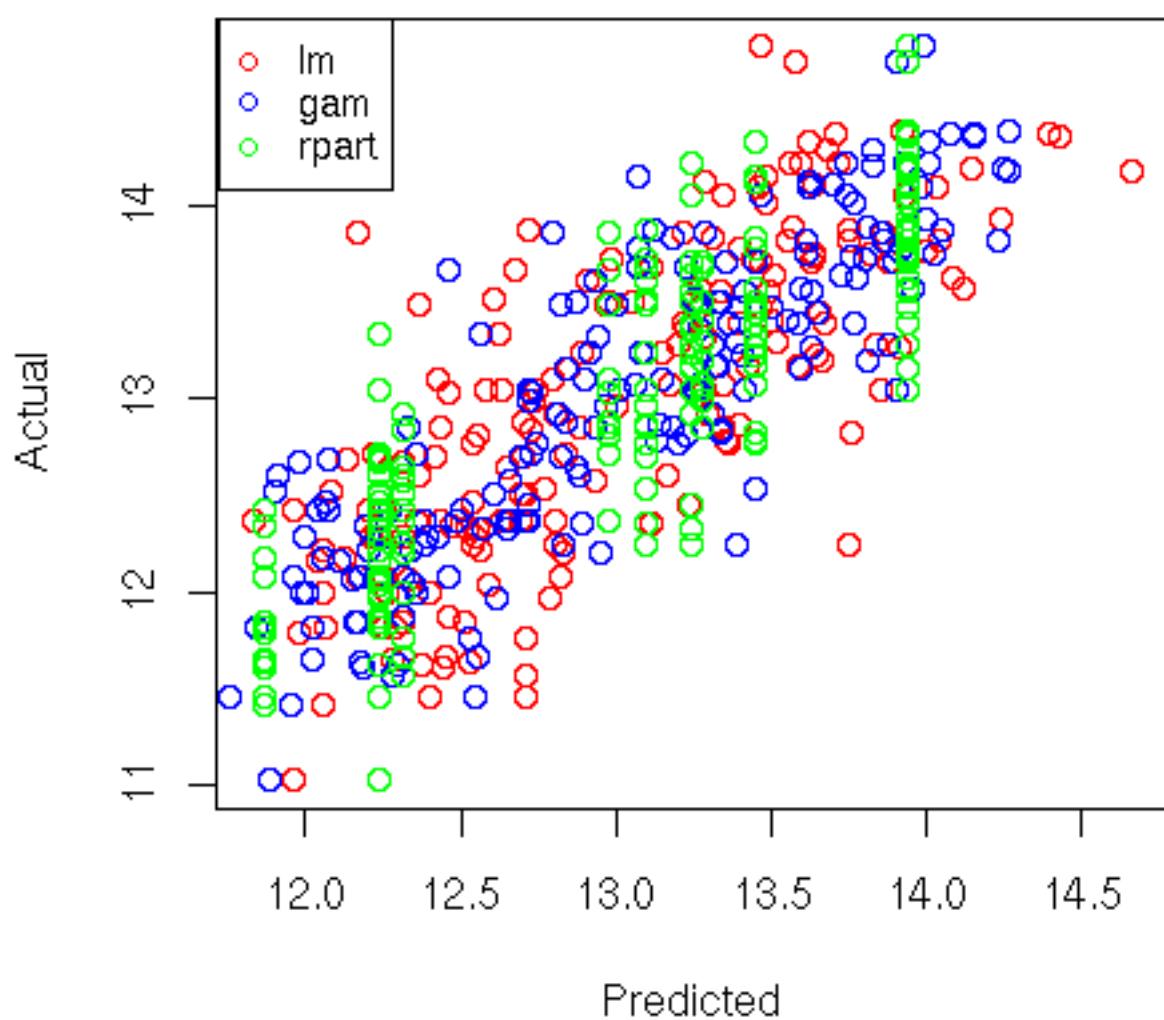
This unadjusted R-squared value is a little higher than the adjusted R-squared value from the `gam` model.

## 15.8 Comparison of the 3 Methods

A very simple way to get an idea of how the three methods compare with each other is to make a plot of predicted versus actual values for the three methods, using a different color for each:

```
> plot(predict(wine.lm),wine$Alcohol,col='red',xlab='Predicted',ylab='Actual')
> points(predict(wine.gam),wine$Alcohol,col='blue')
> points(predict(wine.rpart),wine$Alcohol,col='green')
> legend('topleft',legend=c('lm','gam','rpart'),col=c('red','blue','green'),
+         pch=1,cex=.8)
```

Here's the plot:



# Chapter 16

## Analysis of Variance

## 16.1 Analysis of Variance

In its simplest form, analysis of variance (often abbreviated as ANOVA), can be thought of as a generalization of the t-test, because it allows us to test the hypothesis that the means of a dependent variable are the same for several groups, not just two as would be the case when using a t-test. This type of ANOVA is known as a one-way ANOVA.

In cases where there are multiple classification variables, more complex ANOVAs are possible. For example, suppose we have data on test scores for students from four schools, where three different teaching methods were used. This would describe a two-way ANOVA. In addition to asking whether the means for the different schools were different from each other, and whether the means for the different teaching methods were different from each other, we could also investigate whether the differences in teaching methods were different depending on which school we looked at. This last comparison is known as an interaction, and testing for interactions is one of the most important uses of analysis of variance.

Before getting to the specifics of ANOVA, it may be useful to ask why we perform an analysis of variance if our interest lies in the differences between means. If we were to concentrate on the differences between the means, we would have many different comparisons to make, and the number of comparisons would increase as we increased the number of groups we considered. Thus, we'd need different tests depending on how many groups we were looking at. The reasoning behind using variance to test for differences in means is based on the following idea: Suppose we have several groups of data, and we calculate their variance in two different ways. First, we put together all the data, and simply calculate its variance disregarding the groups from which the data arose. In other words, we evaluate the deviations of the data relative to overall mean of the entire data set. Next, we calculate the variance by adding up the deviations around the mean of each of the groups. The idea of analysis of variance is that if the two variance calculations give us very similar results, then each of the group means must have been about the same, because using the group means to measure variation didn't result in a big change than from using the overall mean. But if the overall variance is bigger than the variance calculated using the group means, then at least one of the group means must have been different from the overall mean, so it's unlikely that the means of all the groups were the same. Using this approach, we only need to compare two values (the overall variance, and the variance calculated using each of the group means) to test if any of the means are different, regardless of how many groups we have.

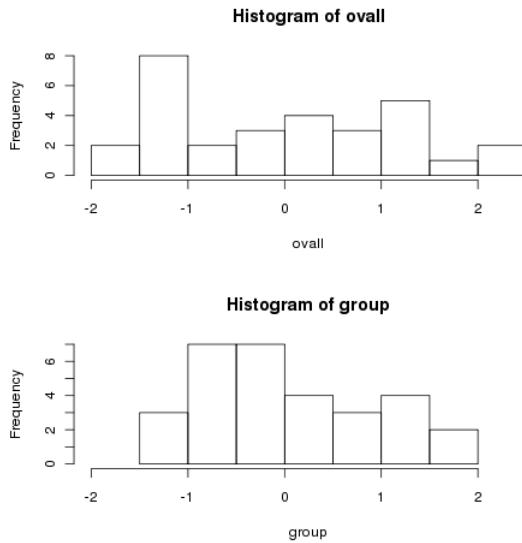
To illustrate how looking at variances can tell us about differences in means, consider a data set with three groups, where the mean of the first group is 3, and the mean for the other groups is 1. We can generate a sample as follows:

```
> mydf = data.frame(group=rep(1:3,rep(10,3)),x=rnorm(30,mean=c(rep(3,10),rep(1,20))))
```

Under the null hypothesis of no differences among the means, we can center each set of data by the appropriate group mean, and then compare the data to the same data centered by the overall mean. In R, the `ave` function will return a vector the same length as its' input, containing summary statistics calculated by grouping variables. Since `ave` accepts an unlimited number of grouping variables, we must identify the function that calculates the

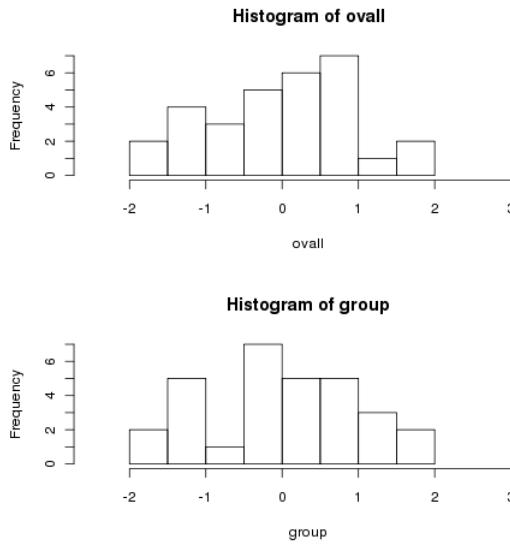
statistic as the `FUN=` argument. Let's look at two histograms of the data, first centered by the overall mean, and then by the group means. Recall that under the null hypothesis, there should be no difference.

```
> oval1 = mydf$x - mean(mydf$x)
> group = mydf$x - ave(mydf$x,mydf$group,FUN=mean)
> par(mfrow=c(2,1))
> hist(oval1,xlim=c(-2,2.5))
> hist(group,xlim=c(-2,2.5))
```



Notice how much more spread out the data is when we centered by the overall mean. To show that this isn't a trick, let's generate some data for which the means are all equal:

```
> mydf1 = data.frame(group=rep(1:3,rep(10,3)),x=rnorm(30))
> oval1 = mydf1$x - mean(mydf1$x)
> group = mydf1$x - ave(mydf1$x,mydf1$group,FUN=mean)
> par(mfrow=c(2,1))
> hist(oval1,xlim=c(-2.5,3.2))
> hist(group,xlim=c(-2.5,3.2))
```

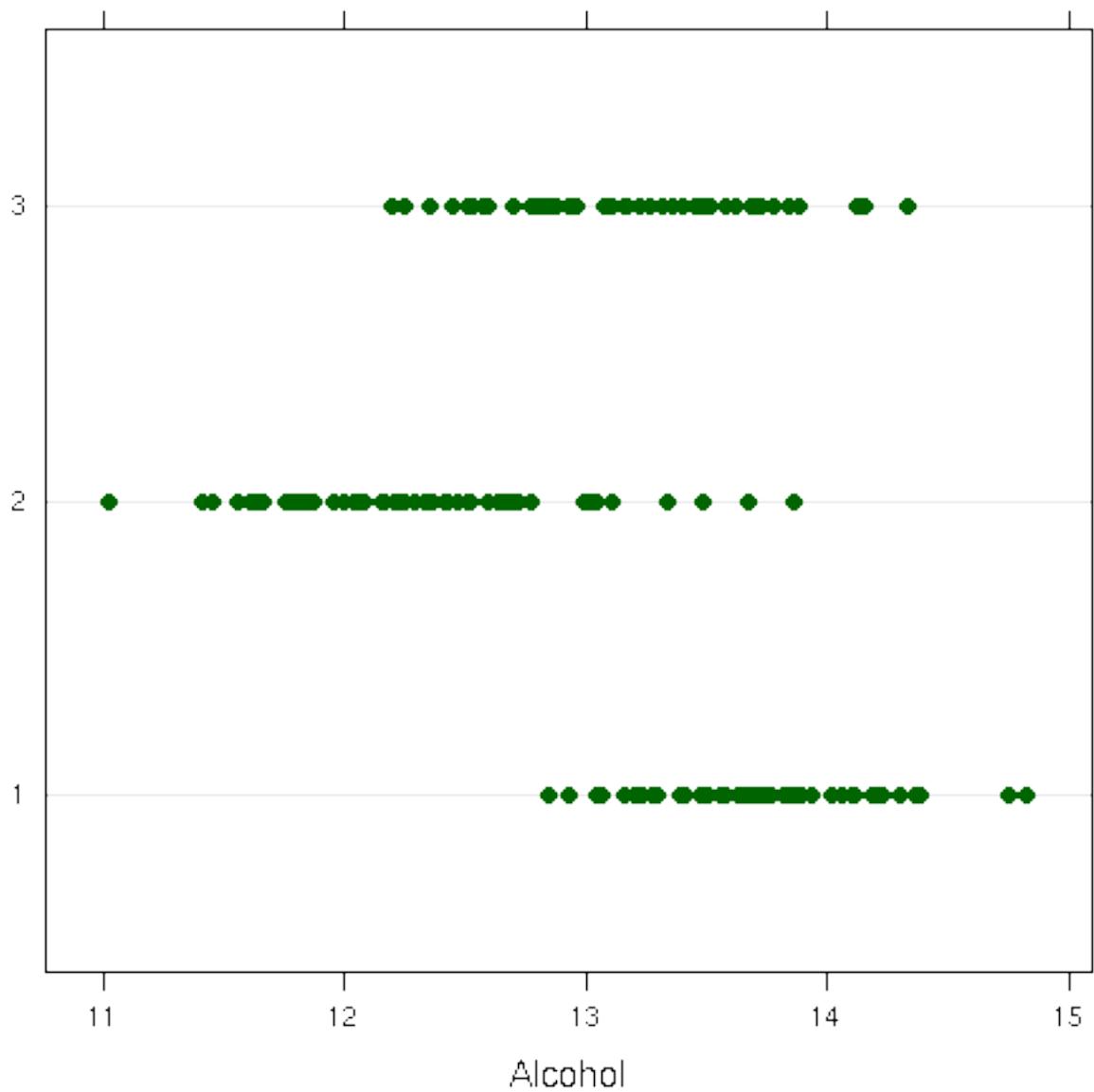


Notice how the two histograms are very similar.

To formalize the idea of a one-way ANOVA, we have a data set with a dependent variable and a grouping variable. We assume that the observations are independent of each other, and the errors (that part of the data not explained by an observation's group mean) follow a normal distribution with the same variance for all the observations. The null hypothesis states that the means of all the groups are equal, against an alternative that at least one of the means differs from the others. We can test the null hypothesis by taking the ratio of the variance calculated in the two ways described above, and comparing it to an F distribution with appropriate degrees of freedom (more on that later).

In R, ANOVAs can be performed with the `aov` command. When you are performing an ANOVA in R, it's very important that all of the grouping variables involved in the ANOVA are converted to factors, or R will treat them as if they were just independent variables in a linear regression.

As a first example, consider once again the `wine` data frame. The `Cultivar` variable represents one of three different varieties of wine that have been studied. As a quick preliminary test, we can examine a dotplot of `Alcohol` versus `Cultivar`:



It does appear that there are some differences, even though there is overlap. We can test for these differences with an ANOVA:

```
> wine.aov = aov(Alcohol~Cultivar,data=wine)
> wine.aov
Call:
aov(formula = Alcohol ~ Cultivar, data = wine)
```

Terms:  
Cultivar Residuals

```

Sum of Squares 70.79485 45.85918
Deg. of Freedom      2        175

Residual standard error: 0.5119106
Estimated effects may be unbalanced
> summary(wine.aov)
      Df Sum Sq Mean Sq F value    Pr(>F)
Cultivar     2 70.795 35.397 135.08 < 2.2e-16 ***
Residuals   175 45.859  0.262
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

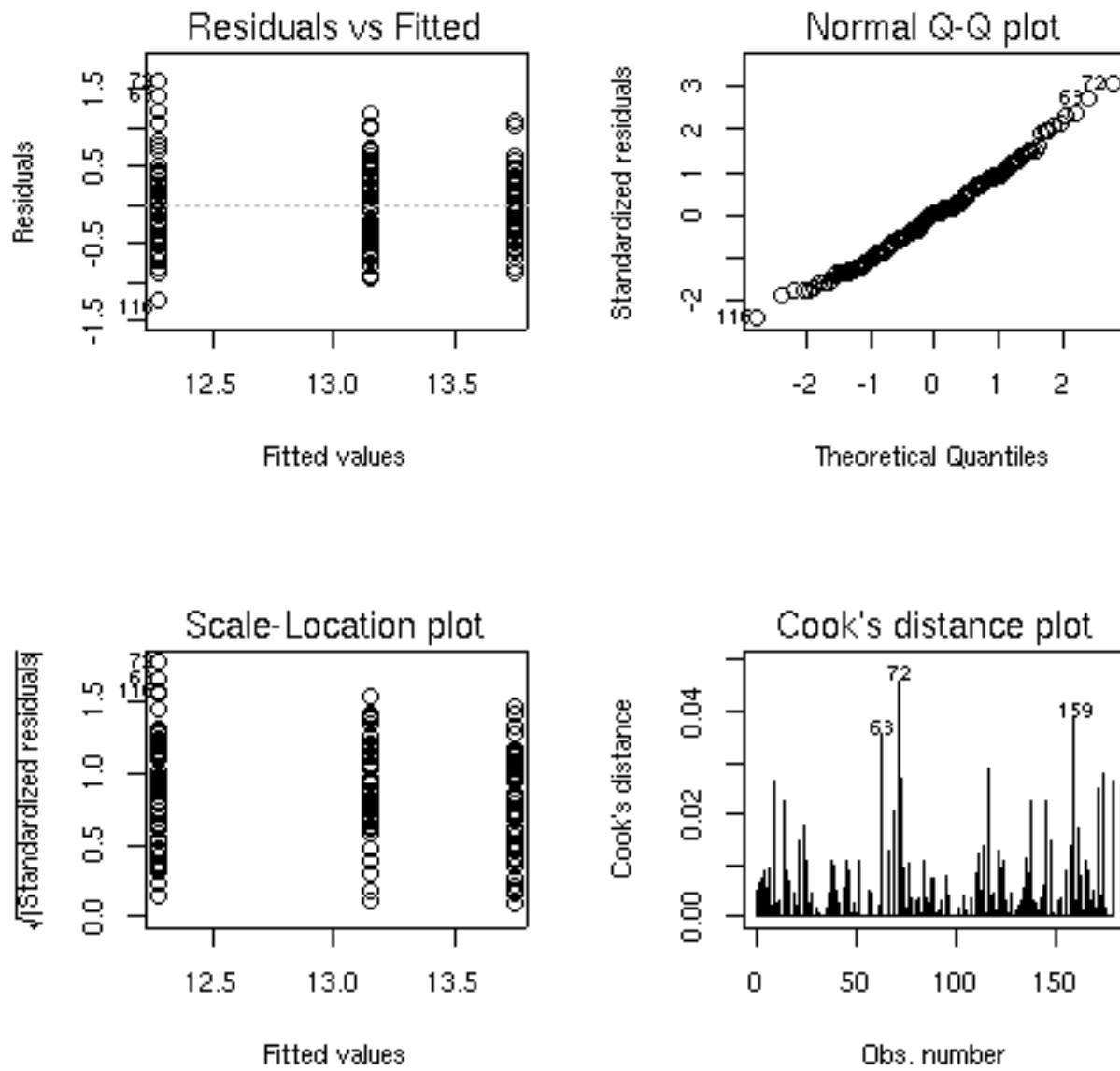
The `summary` function displays the ANOVA table, which is similar to that produced by most statistical software. It indicates that the differences among the means are statistically significant. To see the values for the means, we can use the `aggregate` function:

```

> aggregate(wine$Alcohol,wine['Cultivar'],mean)
  Cultivar      x
1       1 13.74475
2       2 12.27873
3       3 13.15375

```

The default plots from an `aov` object are the same as those for an `lm` object. They're displayed below for the `Alcohol/Cultivar` ANOVA we just calculated:



## 16.2 Multiple Comparisons

In the previous example, notice that the test for `Cultivar` is simply answering the question “Are there any significant differences among the cultivars?”. This is because the F-test which is used to determine significant is based on the two different ways of calculating the variance, not on any particular differences among the means. Having seen that there is a significant effect for `Cultivar` in the previous example, a natural question is “Which cultivars are different from each other”. One possibility would be to look at all possible t-tests between

the levels of the cultivar, i.e. do t-tests for 1 vs. 2, 1 vs. 3, and 2 vs. 3. This is a very bad idea for at least two reasons:

1. One of the main goals of ANOVA is to combine together all our data, so that we can more accurately estimate the residual variance of our model. In the previous example, notice that there were 175 degrees of freedom used to estimate the residual variance. Under the assumptions of ANOVA, the variance of the dependent variable doesn't change across the different levels of the independent variables, so we can (and should) use the estimate from the ANOVA for all our tests. When we use a t-test, we'll be estimating the residual variance only using the observations for the two groups we're comparing, so we'll have fewer degrees of freedom, and less power in determining differences.
2. When we're comparing several groups using t-tests, we have to look at all possible combinations among the groups. This will sometimes result in many tests, and we can no longer be confident that the probability level we use for the individual tests will hold up across all of those comparisons we're making. This is a well-known problem in statistics, and many techniques have been developed to adjust probability values to handle this case. However, these techniques tend to be quite conservative, and they may prevent us from seeing differences that really exist.

To see how probabilities get adjusted when many comparisons are made, consider a data set on the nitrogen levels in 5 varieties of clover. We wish to test the hypothesis that the nitrogen level of the different varieties of clover is the same.

```
> clover = read.table('http://www.stat.berkeley.edu/classes/s133/data/clover.txt',
+                      header=TRUE)
> clover.aov = aov(Nitrogen~Strain,data=clover)
> summary(clover.aov)
    Df Sum Sq Mean Sq F value    Pr(>F)
Strain      5 847.05 169.41 14.370 1.485e-06 ***
Residuals  24 282.93 11.79
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

Let's say that we want to look at all the possible t-tests among pairs of the 6 strains. First, we can use the `combn` function to show us all the possible 2-way combinations of the strains:

```
> combs = combn(as.character(unique(clover$Strain)),2)
> combs
     [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]
[1,] "3DOK1" "3DOK1" "3DOK1" "3DOK1" "3DOK1" "3DOK5" "3DOK5" "3DOK5"
[2,] "3DOK5" "3DOK4" "3DOK7" "3DOK13" "COMPOS" "3DOK4" "3DOK7" "3DOK13"
     [,9]   [,10]   [,11]   [,12]   [,13]   [,14]   [,15]
[1,] "3DOK5" "3DOK4" "3DOK4" "3DOK4" "3DOK7" "3DOK7" "3DOK13"
[2,] "COMPOS" "3DOK7" "3DOK13" "COMPOS" "3DOK13" "COMPOS" "COMPOS"
```

Let's focus on the first column:

```
> x = combs[,1]
> tt = t.test(Nitrogen~Strain,data=clover)
> names(tt)
[1] "statistic"    "parameter"    "p.value"      "conf.int"     "estimate"
[6] "null.value"   "alternative"  "method"       "data.name"
```

This suggests a function which would return the probability for each combination of strains:

```
> gettprob = function(x)t.test(Nitrogen~Strain,
                                data=clover[clover$Strain %in% x,])$p.value
```

We can get the probabilities for all the tests, and combine them with the country names for display:

```
> probs = data.frame(t(combs),probs=apply(combs,2,gettprob))
> probs
      X1      X2      probs
1 3DOK1 3DOK5 1.626608e-01
2 3DOK1 3DOK4 2.732478e-03
3 3DOK1 3DOK7 2.511696e-02
4 3DOK1 3DOK13 3.016445e-03
5 3DOK1 COMPOS 1.528480e-02
6 3DOK5 3DOK4 5.794178e-03
7 3DOK5 3DOK7 7.276336e-02
8 3DOK5 3DOK13 1.785048e-03
9 3DOK5 COMPOS 3.177169e-02
10 3DOK4 3DOK7 4.331464e-02
11 3DOK4 3DOK13 5.107291e-01
12 3DOK4 COMPOS 9.298460e-02
13 3DOK7 3DOK13 4.996374e-05
14 3DOK7 COMPOS 2.055216e-01
15 3DOK13 COMPOS 4.932466e-04
```

These probabilities are for the individual t-tests, each with an alpha level of 0.05, but that doesn't guarantee that the experiment-wise alpha will be .05. We can use the `p.adjust` function to adjust these probabilities:

```
> probs = data.frame(probs,adj.prob=p.adjust(probs$probs,method='bonferroni'))
> probs
      X1      X2      probs      adj.prob
1 3DOK1 3DOK5 1.626608e-01 1.0000000000
2 3DOK1 3DOK4 2.732478e-03 0.040987172
3 3DOK1 3DOK7 2.511696e-02 0.376754330
```

```

4 3DOK1 3DOK13 3.016445e-03 0.045246679
5 3DOK1 COMPOS 1.528480e-02 0.229272031
6 3DOK5 3DOK4 5.794178e-03 0.086912663
7 3DOK5 3DOK7 7.276336e-02 1.000000000
8 3DOK5 3DOK13 1.785048e-03 0.026775721
9 3DOK5 COMPOS 3.177169e-02 0.476575396
10 3DOK4 3DOK7 4.331464e-02 0.649719553
11 3DOK4 3DOK13 5.107291e-01 1.000000000
12 3DOK4 COMPOS 9.298460e-02 1.000000000
13 3DOK7 3DOK13 4.996374e-05 0.000749456
14 3DOK7 COMPOS 2.055216e-01 1.000000000
15 3DOK13 COMPOS 4.932466e-04 0.007398699

```

Notice that many of the comparisons that seemed significant when using the t-test are no longer significant. Plus, we didn't take advantage of the increased degrees of freedom. One technique that uses all the degrees of freedom of the combined test, while still correcting for the problem of multiple comparisons is known as Tukey's Honestly Significant Difference (HSD) test. The TukeyHSD function takes a model object and the name of a factor, and provides protected probability values for all the two-way comparisons of factor levels. Here's the output of TukeyHSD for the clover data:

```

> tclover = TukeyHSD(clover.aov, 'Strain')
> tclover
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = Nitrogen ~ Strain, data = clover)

$Strain
      diff      lwr      upr      p adj
3DOK13-3DOK1 -15.56 -22.27416704 -8.845833 0.0000029
3DOK4-3DOK1  -14.18 -20.89416704 -7.465833 0.0000128
3DOK5-3DOK1   -4.84 -11.55416704  1.874167 0.2617111
3DOK7-3DOK1   -8.90 -15.61416704 -2.185833 0.0048849
COMPOS-3DOK1  -10.12 -16.83416704 -3.405833 0.0012341
3DOK4-3DOK13   1.38  -5.33416704  8.094167 0.9870716
3DOK5-3DOK13   10.72  4.00583296 17.434167 0.0006233
3DOK7-3DOK13   6.66  -0.05416704 13.374167 0.0527514
COMPOS-3DOK13   5.44  -1.27416704 12.154167 0.1621550
3DOK5-3DOK4    9.34  2.62583296 16.054167 0.0029837
3DOK7-3DOK4    5.28  -1.43416704 11.994167 0.1852490
COMPOS-3DOK4    4.06  -2.65416704 10.774167 0.4434643
3DOK7-3DOK5   -4.06 -10.77416704  2.654167 0.4434643
COMPOS-3DOK5   -5.28 -11.99416704  1.434167 0.1852490

```

```
COMPOS-3DOK7 -1.22 -7.93416704 5.494167 0.9926132
```

```
> class(tclover)
[1] "multicomp" "TukeyHSD"
> names(tclover)
[1] "Strain"
> class(tclover$Strain)
[1] "matrix"
```

These probabilities seem more reasonable. To combine these results with the previous ones, notice that `tclover$Strain` is a matrix, with row names indicating the comparisons being made. We can put similar row names on our earlier results and then merge them:

```
> row.names(probs) = paste(probs$X2, probs$X1, sep=' - ')
> probs = merge(probs, tclover$Strain[, 'p adj', drop=FALSE], by=0)
> probs
   Row.names     X1     X2      probs    adj.prob      p adj
1 3DOK13-3DOK1 3DOK1 3DOK13 0.0030164452 0.045246679 2.888133e-06
2 3DOK4-3DOK1 3DOK1 3DOK4 0.0027324782 0.040987172 1.278706e-05
3 3DOK5-3DOK1 3DOK1 3DOK5 0.1626608271 1.000000000 2.617111e-01
4 3DOK7-3DOK1 3DOK1 3DOK7 0.0251169553 0.376754330 4.884864e-03
5 3DOK7-3DOK4 3DOK4 3DOK7 0.0433146369 0.649719553 1.852490e-01
6 3DOK7-3DOK5 3DOK5 3DOK7 0.0727633570 1.000000000 4.434643e-01
7 COMPOS-3DOK1 3DOK1 COMPOS 0.0152848021 0.229272031 1.234071e-03
8 COMPOS-3DOK13 3DOK13 COMPOS 0.0004932466 0.007398699 1.621550e-01
9 COMPOS-3DOK4 3DOK4 COMPOS 0.0929845957 1.000000000 4.434643e-01
10 COMPOS-3DOK5 3DOK5 COMPOS 0.0317716931 0.476575396 1.852490e-01
11 COMPOS-3DOK7 3DOK7 COMPOS 0.2055215679 1.000000000 9.926132e-01
```

Finally, we can display the probabilities without scientific notation as follows:

```
> format(probs, scientific=FALSE)
   Row.names     X1     X2      probs    adj.prob      p adj
1 3DOK13-3DOK1 3DOK1 3DOK13 0.0030164452 0.045246679 0.000002888133
2 3DOK4-3DOK1 3DOK1 3DOK4 0.0027324782 0.040987172 0.000012787061
3 3DOK5-3DOK1 3DOK1 3DOK5 0.1626608271 1.000000000 0.261711120046
4 3DOK7-3DOK1 3DOK1 3DOK7 0.0251169553 0.376754330 0.004884863746
5 3DOK7-3DOK4 3DOK4 3DOK7 0.0433146369 0.649719553 0.185248969392
6 3DOK7-3DOK5 3DOK5 3DOK7 0.0727633570 1.000000000 0.443464260597
7 COMPOS-3DOK1 3DOK1 COMPOS 0.0152848021 0.229272031 0.001234070633
8 COMPOS-3DOK13 3DOK13 COMPOS 0.0004932466 0.007398699 0.162154993324
9 COMPOS-3DOK4 3DOK4 COMPOS 0.0929845957 1.000000000 0.443464260597
10 COMPOS-3DOK5 3DOK5 COMPOS 0.0317716931 0.476575396 0.185248969392
11 COMPOS-3DOK7 3DOK7 COMPOS 0.2055215679 1.000000000 0.992613208547
```

By using all of the data to estimate the residual error, Tukey's HSD method actually reports some of the probabilities as even lower than the t-tests.

## 16.3 Two-Way ANOVA

To express the idea of an interaction in the R modeling language, we need to introduce two new operators. The colon (:) is used to indicate an interaction between two or more variables in model formula. The asterisk (\*) is used to indicate all main effects and interactions among the variables that it joins. So, for example the term A\*B would expand to the three terms A, B, and A:B. As an example of a two-way ANOVA, consider a study to determine the effects of physical activity on obesity. Subjects were rated for their physical activity on a three point scale with 1=not very active, 2=somewhat active, and 3=very active. In addition, the race (either 1 or 2) of the participant was recorded, along with their Body Mass Index (BMI). We want to answer the following three questions:

1. Were the means for BMI the same for the two races?
2. Were the means for BMI the same for the three activity levels?
3. Is the effect of activity level different depending on race?, or equivalently Is the effect of race different depending on activity level?

The first two questions can be answered by looking at the `race` and `activity` main effects, while the third question describes the `race` by `activity` interaction. The data can be found at <http://www.stat.berkeley.edu/classes/s133/data/activity.csv>. Here are the R statements to run the ANOVA:

```
> activity = read.csv('activity.csv')
> activity$race = factor(activity$race)
> activity$activity = factor(activity$activity)
> activity.aov = aov(bmi~race*activity,data=activity)
> summary(activity.aov)

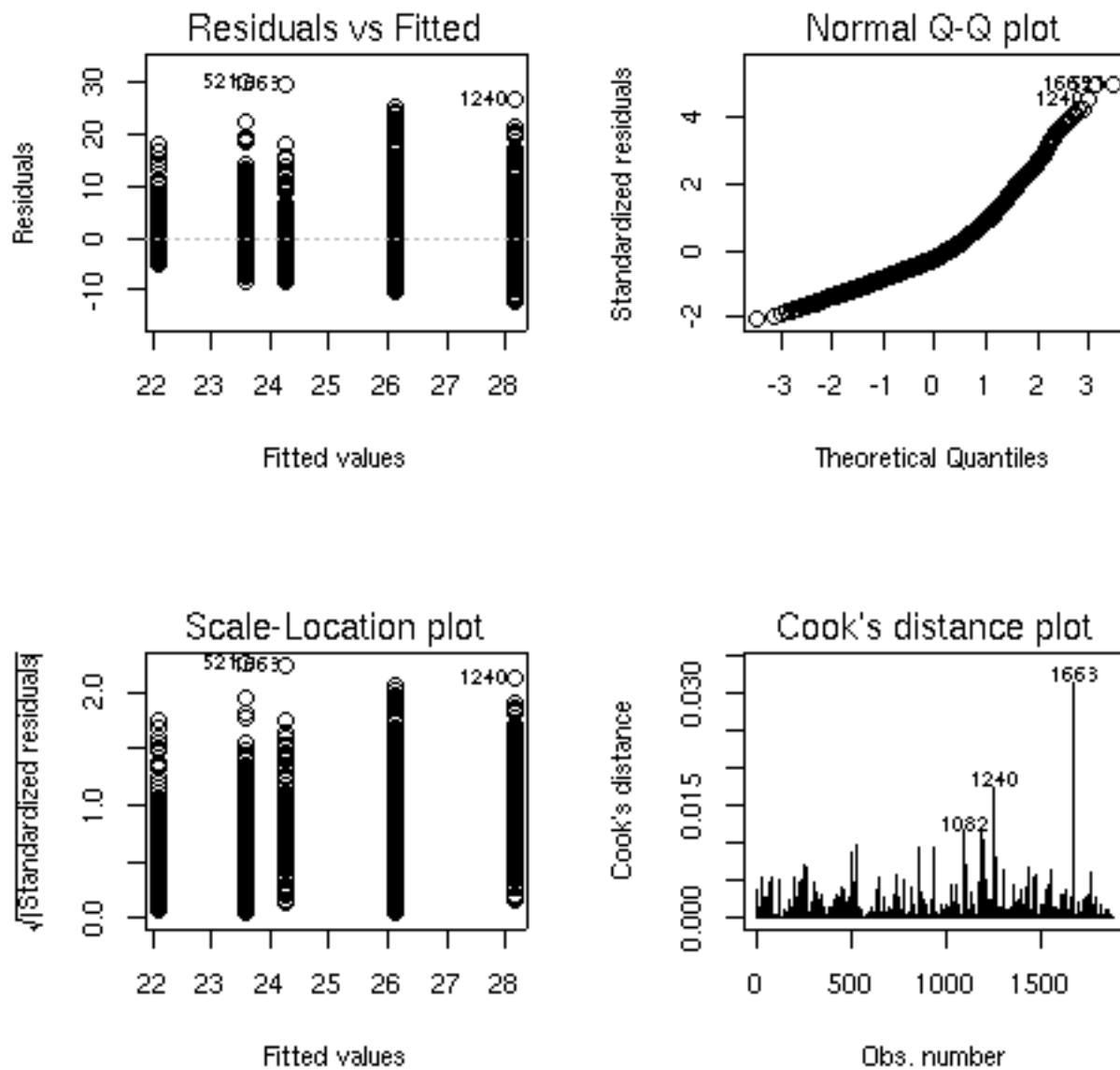
      Df  Sum Sq Mean Sq F value    Pr(>F)
race        1    3552    3552 102.5894 < 2e-16 ***
activity     2    2672    1336  38.5803 < 2e-16 ***
race:activity 2    301     151   4.3508 0.01303 *
Residuals   1865  64574      35
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Notice that there are two degrees of freedom for `activity` – this means two parameters will be estimated in order to explain `activity`'s effect on `bmi`. Unlike linear regression, where only a single parameter is estimated, and the only relationship that can be fit is a linear one, using two parameters (to account for the three levels of `activity`) provides more flexibility than would be possible with linear regression.

To see if the analysis was reasonable, we can look at the default plots:

```
> plot(activity.aov)
```

The graphs appear below:



There seems to be some deviation from normality when looking at the Normal Q-Q plot (recall that, if the residuals did follow a normal distribution, we would see a straight line.) When this situation arises, analyzing the logarithm of the dependent variable often helps. Here are the same results for the analysis of  $\log(\text{bmi})$ :

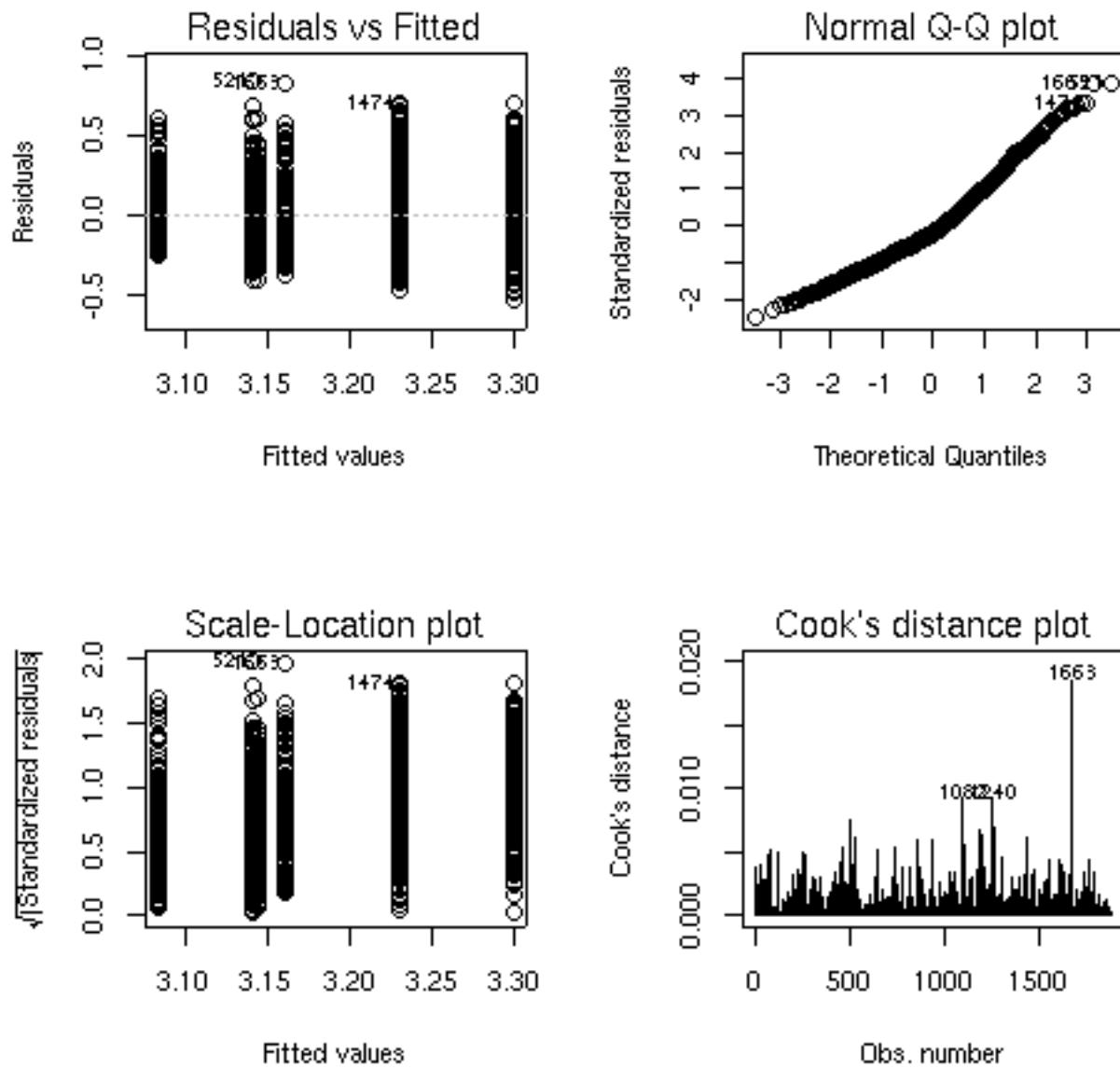
```
> activity1.aov = aov(log(bmi)~race*activity,data=activity)
> summary(activity1.aov)
      Df Sum Sq Mean Sq  F value    Pr(>F)
race          1  4.588   4.588 100.3741 < 2.2e-16 ***

```

```

activity           2   3.251    1.625   35.5596  6.98e-16 ***
race:activity     2   0.317    0.158   3.4625   0.03155 *
Residuals        1865  85.240    0.046
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> plot(activity1.aov)

```



The Q-Q plot looks better, so this model is probably more appropriate. We can see both main effects as well as the interaction are significant. To see what's happening with the main effects, we can use `aggregate`:

```

> aggregate(log(activity$bmi),activity['race'],mean)
  race      x
1    1 3.122940
2    2 3.222024
> aggregate(log(activity$bmi),activity['activity'],mean)
  activity      x
1          1 3.242682
2          2 3.189810
3          3 3.109518

```

Race 2 has higher values of BMI than race 1, and BMI decreases as the level of activity increases.

To study the interaction, we could use `aggregate`, passing both `race` and `activity` as the second argument:

```

> aggregate(log(activity$bmi),activity[c('race','activity')],mean)
  race activity      x
1    1      1 3.161119
2    2      1 3.298576
3    1      2 3.140970
4    2      2 3.230651
5    1      3 3.084426
6    2      3 3.143478

```

The arrangement of the output from `tapply` may be more helpful:

```

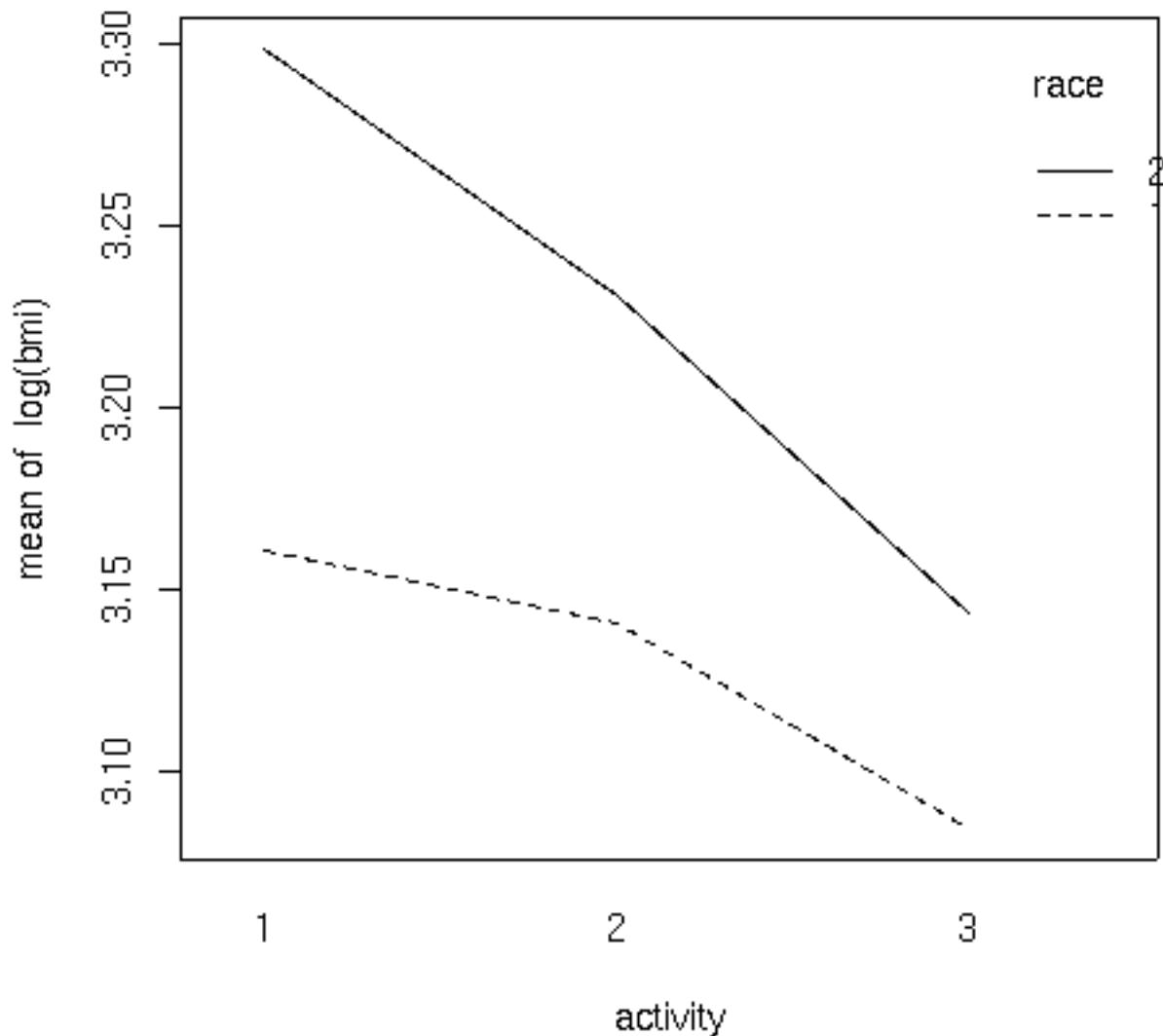
> tapply(log(activity$bmi),activity[c('race','activity')],mean)
  activity
race      1      2      3
1 3.161119 3.140970 3.084426
2 3.298576 3.230651 3.143478

```

It's usually difficult to judge relationships like this from a table. One useful tool in this case is an interaction plot. An interaction plot has one point for each combination of the factors defined by an interaction. The x-axis represents the levels of one of the factors, and the y-axis represents the mean of the dependent variable, and a separate line is drawn for each level of the factor not represented on the x-axis. While it wouldn't be too hard to produce such a plot with basic commands in R, the process is automated by the `interaction.plot` function. The first argument to this function is the factor to appear on the x-axis; the second is the factor which will define the multiple lines being drawn, and the third argument is the dependent variable. By default, `interaction.plot` uses the mean for its display, but you can provide a function of your own choosing through the `fun=` argument. For the activity data, we can produce an interaction plot with the following code:

```
> with(activity,interaction.plot(activity,race,log(bmi)))
```

Here's the plot:



It can be seen that the interaction is due to the fact that the slope of the line for race 2 is steeper than the line for race 1.

## 16.4 Another Example

This example has to do with iron retention in mice. Two different treatments, each at three different levels, were fed to mice. The treatments were tagged with radioactive iron, so that

the percentage of iron retained could be measured after a fixed period of time. The data is presented in a table as follows:

Fe2+			Fe3+		
high	medium	low	high	medium	low
0.71	2.20	2.25	2.20	4.04	2.71
1.66	2.93	3.93	2.69	4.16	5.43
2.01	3.08	5.08	3.54	4.42	6.38
2.16	3.49	5.82	3.75	4.93	6.38
2.42	4.11	5.84	3.83	5.49	8.32
2.42	4.95	6.89	4.08	5.77	9.04
2.56	5.16	8.50	4.27	5.86	9.56
2.60	5.54	8.56	4.53	6.28	10.01
3.31	5.68	9.44	5.32	6.97	10.08
3.64	6.25	10.52	6.18	7.06	10.62
3.74	7.25	13.46	6.22	7.78	13.80
3.74	7.90	13.57	6.33	9.23	15.99
4.39	8.85	14.76	6.97	9.34	17.90
4.50	11.96	16.41	6.97	9.91	18.25
5.07	15.54	16.96	7.52	13.46	19.32
5.26	15.89	17.56	8.36	18.40	19.87
8.15	18.30	22.82	11.65	23.89	21.60
8.24	18.59	29.13	12.45	26.39	22.25

Thus, before we can perform analysis on the data, it needs to be rearranged. To do this, we can use the `reshape` function. Since there are two different sets of variables that represent the change in the factors of the experiment, we first read in the data (skipping the header), and create two groups of variables in our call to reshape:

```
> iron0 = read.table('iron.txt',skip=5,nrows=18)
> names(iron0) = c('Fe2high','Fe2medium','Fe2low','Fe3high','Fe3medium','Fe3low')
> iron1 = reshape(iron0,varying=list(1:3,4:6),direction='long')
> head(iron1)
   time Fe2high Fe3high id
1.1    1    0.71   2.20  1
2.1    1    1.66   2.69  2
3.1    1    2.01   3.54  3
4.1    1    2.16   3.75  4
5.1    1    2.42   3.83  5
6.1    1    2.42   4.08  6
```

After examining the data, it can be seen that the low, medium, and high values have been translated into values 1, 2, and 3 in the variable `time`. The `id` variable is created to help us see which line each observation came from, which is not relevant in this case, since the table was just used to present the data, and the values in the table don't represent repeated measures on the same experimental unit.

Next, we eliminate the `id` column, rename the column named “`time`” and further reshape the data to represent the two treatments:

```
> iron1$id = NULL
> names(iron1)[1] = 'level'
> iron = reshape(iron1, varying=list(2:3), direction='long')
> head(iron)
  level time Fe2high id
1.1    1    1    0.71  1
2.1    1    1    1.66  2
3.1    1    1    2.01  3
4.1    1    1    2.16  4
5.1    1    1    2.42  5
6.1    1    1    2.42  6
```

All that's left is to remove the `id` column and to rename `time` and `Fe2high`:

```
> iron$id = NULL
> names(iron)[2:3] = c('treatment', 'retention')
> head(iron)
  level treatment retention
1.1    1          1      0.71
2.1    1          1      1.66
3.1    1          1      2.01
4.1    1          1      2.16
5.1    1          1      2.42
6.1    1          1      2.42
```

Once the data has been reshaped, it's essential to make sure that the independent variables are correctly stored as factors:

```
> sapply(iron, class)
  level treatment retention
"integer" "integer" "numeric"
```

Since `treatment` and `level` are not factors, we must convert them:

```
> iron$treatment = factor(iron$treatment, labels=c('Fe2+', 'Fe3+'))
> iron$level = factor(iron$level, labels=c('high', 'medium', 'low'))
> head(iron)
```

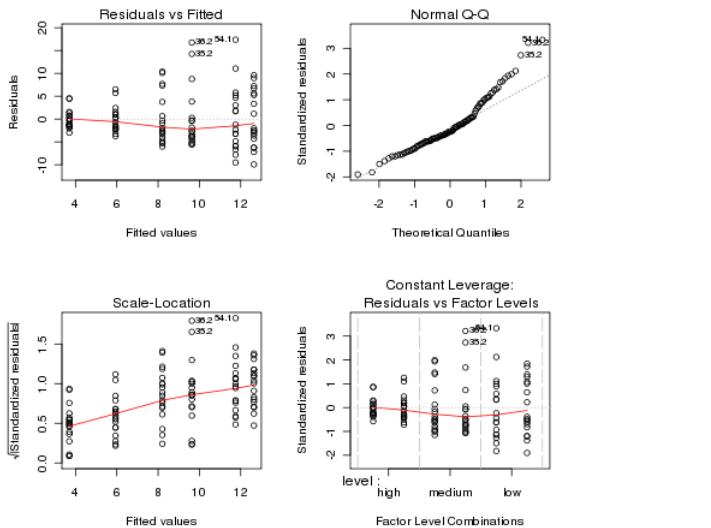
	level	treatment	retention	
1.1	high	Fe2+	0.71	
2.1	high	Fe2+	1.66	
3.1	high	Fe2+	2.01	
4.1	high	Fe2+	2.16	
5.1	high	Fe2+	2.42	
6.1	high	Fe2+	2.42	

Now we can perform the ANOVA:

```
> iron.aov = aov(retention ~ level*treatment,data=iron)
> summary(iron.aov)

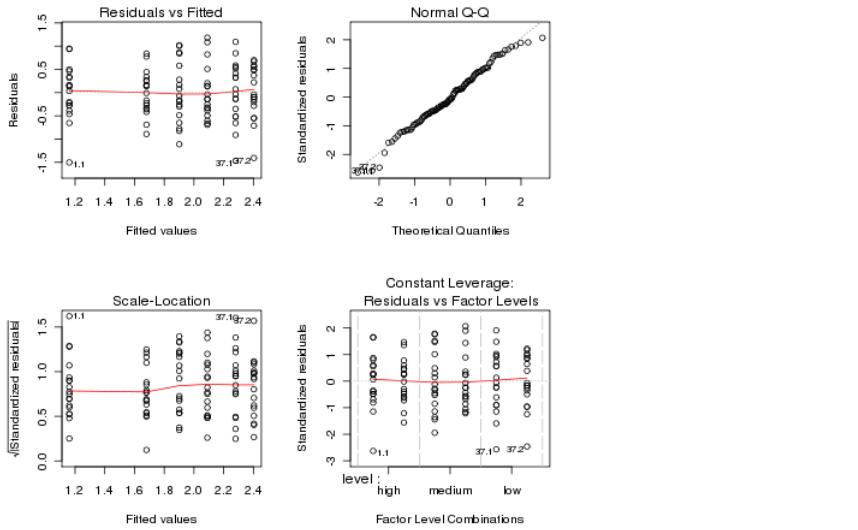
Df Sum Sq Mean Sq F value    Pr(>F)
level          2 983.62 491.81 17.0732 4.021e-07 ***
treatment      1   62.26   62.26  2.1613   0.1446
level:treatment 2   8.29    4.15  0.1439   0.8661
Residuals     102 2938.20   28.81
```

Before proceeding further, we should examine the ANOVA plots to see if the data meets the assumptions of ANOVA:



Both the normal Q-Q plot and the scale-location plot indicate problems similar to the previous example, and a log transformation is once again suggested. This is not unusual when data is measured as percentages or ratios.

```
> ironl.aov = aov(log(retention) ~ level*treatment,data=iron)
> par(mfrow=c(2,2))
> plot(ironl.aov)
```



The plots look much better, so we'll continue with the analysis of the log of retention.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
level	2	15.588	7.794	22.5241	7.91e-09 ***
treatment	1	2.074	2.074	5.9931	0.01607 *
level:treatment	2	0.810	0.405	1.1708	0.31426
Residuals	102	35.296	0.346		
<hr/>					
Signif. codes:	0	***	0.001	** 0.01	* 0.05 . 0.1 1

Since there were only two levels of treatment, the significant treatment effect means the two treatments were different. We can use the TukeyHSD function to see if the different levels of the treatment were different:

```
> TukeyHSD(iron1.aov, 'level')
Tukey multiple comparisons of means
 95% family-wise confidence level
```

```
Fit: aov(formula = log(retention) ~ level * treatment, data = iron)
```

\$level	diff	lwr	upr	p	adj
medium-high	0.5751084	0.24533774	0.9048791	0.0002042	
low-high	0.9211588	0.59138806	1.2509295	0.0000000	
low-medium	0.3460503	0.01627962	0.6758210	0.0373939	

It appears that the high level had much lower retention values than the other two levels:

```
> aggregate(log(iron$retention), iron['level'], mean)
```

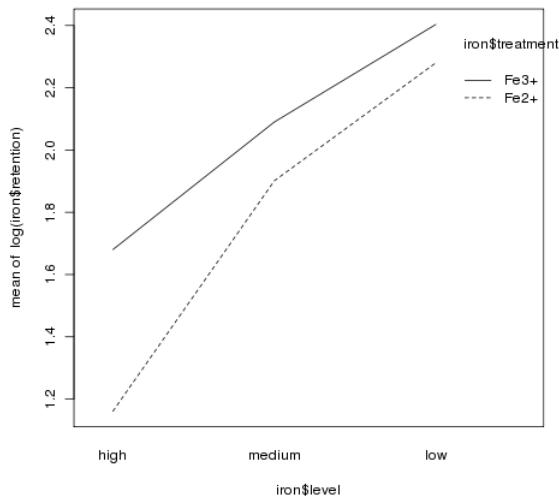
```

level      x
1  high 1.420526
2 medium 1.995635
3   low 2.341685

```

Although there was no significant interaction, an interaction plot can still be useful in visualizing what happens in an experiment:

```
> interaction.plot(iron$level, iron$treatment, log(iron$retention))
```



## 16.5 More Complex Models

When working with the `wine` data frame, we've separated the categorical variable (`Cultivar`) from the continuous variable for pedagogical reasons, but the `aov` function can accommodate both in the same model. Let's add the `Cultivar` variable to the regression model we've previously worked with:

```
> wine.new = lm(Alcohol~Cultivar+Malic.acid+Alkalinity.ash+Proanthocyanins+Color.intensi
> summary(wine.new)
```

Call:

```
lm(formula = Alcohol ~ Cultivar + Malic.acid + Alkalinity.ash +
    Proanthocyanins + Color.intensity + OD.Ratio + Proline, data = wine)
```

Residuals:

Min	1Q	Median	3Q	Max
-1.13591	-0.31737	-0.02623	0.33229	1.65633

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	12.9158487	0.4711149	27.415	< 2e-16 ***
Cultivar2	-0.9957910	0.1776136	-5.607	8.26e-08 ***
Cultivar3	-0.6714047	0.2396380	-2.802	0.00568 **
Malic.acid	0.0559472	0.0410860	1.362	0.17510
Alkalinity.ash	-0.0133598	0.0134499	-0.993	0.32198
Proanthocyanins	-0.0561493	0.0817366	-0.687	0.49305
Color.intensity	0.1135452	0.0270097	4.204	4.24e-05 ***
OD.Ratio	0.0494695	0.0987946	0.501	0.61721
Proline	0.0002391	0.0002282	1.048	0.29629
---				
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1				

Residual standard error: 0.4886 on 169 degrees of freedom

Multiple R-Squared: 0.6541, Adjusted R-squared: 0.6377

F-statistic: 39.95 on 8 and 169 DF, p-value: < 2.2e-16

One problem with the summary display for models like this is that it's treating our factor variable (Cultivar) as two separate variables. While that is the way it is fit in the model, it's usually more informative to combine the effects of the two variables as a single effect. The anova command will produce a more traditional ANOVA table:

```
> anova(wine.new)
```

Analysis of Variance Table

Response: Alcohol

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Cultivar	2	70.795	35.397	148.2546	< 2.2e-16 ***
Malic.acid	1	0.013	0.013	0.0552	0.8146
Alkalinity.ash	1	0.229	0.229	0.9577	0.3292
Proanthocyanins	1	0.224	0.224	0.9384	0.3341
Color.intensity	1	4.750	4.750	19.8942	1.488e-05 ***
OD.Ratio	1	0.031	0.031	0.1284	0.7206
Proline	1	0.262	0.262	1.0976	0.2963
Residuals	169	40.351	0.239		
---					

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

The summary display contained other useful information, so you shouldn't hesitate to look at both.

Comparing these results to our previous regression, we can see that only one variable (Color.intensity) is still significant, and the effect of Cultivar is very significant. For

this data set, it means that while we can use the chemical composition to help predict the `Alcohol` content of the wines, but that knowing the `Cultivar` will be more effective. Let's look at a reduced model that uses only `Cultivar` and `Color.intensity` to see how it compares with the model containing the extra variables:

```
> wine.new1 = lm(Alcohol~Cultivar+Color.intensity,data=wine)
> summary(wine.new1)

Call:
lm(formula = Alcohol ~ Cultivar + Color.intensity, data = wine)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.12074 -0.32721 -0.04133  0.34799  1.54962 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 13.14845   0.14871  88.417 < 2e-16 ***
Cultivar2   -1.20265   0.10431 -11.530 < 2e-16 ***
Cultivar3   -0.79248   0.10495  -7.551 2.33e-12 ***
Color.intensity 0.10786   0.02434   4.432 1.65e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4866 on 174 degrees of freedom
Multiple R-Squared: 0.6468,    Adjusted R-squared: 0.6407 
F-statistic: 106.2 on 3 and 174 DF,  p-value: < 2.2e-16
```

The adjusted R-squared for this model is better than that of the previous one, indicating that removing those extra variables didn't seem to cause any problems. To formally test to see if there is a difference between the two models, we can use the `anova` function. When passed a single model object, `anova` prints an ANOVA table, but when it's passed two model objects, it performs a test to compare the two models:

```
> anova(wine.new,wine.new1)
Analysis of Variance Table

Model 1: Alcohol ~ Cultivar + Malic.acid + Alkalinity.ash + Proanthocyanins +
          Color.intensity + OD.Ratio + Proline
Model 2: Alcohol ~ Cultivar + Color.intensity
  Res.Df   RSS Df Sum of Sq    F Pr(>F)    
  1     169 40.351                                 
  2     174 41.207 -5    -0.856 0.7174 0.6112
```

The test indicates that there's no significant difference between the two models.

When all the independent variables in our model were categorical model (the `race/activity` example), the interactions between the categorical variables was one of the most interesting parts of the analysis. What does an interaction between a categorical variable and a continuous variable represent? Such an interaction can tell us if the slope of the continuous variable is different for the different levels of the categorical variable. In the current model, we can test to see if the slopes are different by adding the term `Cultivar:Color.intensity` to the model:

```
> anova(wine.new2)
Analysis of Variance Table

Response: Alcohol
              Df Sum Sq Mean Sq F value    Pr(>F)
Cultivar          2 70.795 35.397 149.6001 < 2.2e-16 ***
Color.intensity   1  4.652   4.652  19.6613 1.644e-05 ***
Cultivar:Color.intensity 2  0.509   0.255   1.0766     0.343
Residuals        172 40.698   0.237
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

There doesn't seem to be a significant interaction.

As another example, consider a data set with information about experience, gender, and wages. Experience is recorded as number of years on the job, and gender is recorded as 0 or 1. To see if the slope of the line relating experience to wages is different for the two genders, we can proceed as follows:

```
> wages = read.delim('http://www.stat.berkeley.edu/classes/s133/data/wages.tab')
> wages$gender = factor(wages$gender)
> wages.aov = aov(wage ~ experience*gender,data=wages)
> anova(wages.aov)
Analysis of Variance Table

Response: wage
            Df  Sum Sq Mean Sq F value    Pr(>F)
experience      1   106.7  106.69  4.2821  0.03900 *
gender          1   635.8  635.78 25.5175 6.042e-07 ***
experience:gender 1   128.9  128.94  5.1752  0.02331 *
Residuals     530 13205.3   24.92
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The significant probability value for `experience:gender` indicates that the effect of experience on wages is different depending on the gender of the employee. By performing two separate regressions, we can see the values for the slopes for the different genders:

```
> coef(lm(wage~experience,data=subset(wages,gender == 0)))
(Intercept)  experience
8.62215280  0.08091533
> coef(lm(wage~experience,data=subset(wages,gender == 1)))
(Intercept)  experience
7.857197368 0.001150118
```

This indicates that there is a small increase in wages as experience increases for gender == 0, but virtually no increase for gender == 1.

## 16.6 Constructing Formulas

For the examples we've looked at, there weren't so many terms in the model that it became tedious entering them by hand, but in models with many interactions it can quickly become a nuisance to have to enter every term into the model. When the terms are all main effects, you can often save typing by using a `data=` argument specifying a data set with just the variables you are interested in and using the period (.) as the right-hand side of the model, but that will not automatically generate interactions.

The `formula` function will accept a text string containing a formula, and convert it to a formula that can be passed to any of the modeling functions. While it doesn't really make

sense for the `wine` data, suppose we wanted to add `Cultivar` and all the interactions between `Cultivar` and the independent variables to our original regression model. The first step is to create a vector of the variables we want to work with. This can usually be done pretty easily using the `names` of the data frame we're working with.

```
> vnames = names(wine)[c(3,5,10,11,13,14)]
```

For the main part of the model we need to join together these names with plus signs (+):

```
> main = paste(vnames,collapse=' + ')
```

The interactions can be created by pasting together `Cultivar` with each of the continuous variables, using a colon (:) as a separator, and then joining them together with plus signs:

```
> ints = paste(paste('Cultivar',vnames,sep=':'),collapse=" + ")
```

Finally, we put the dependent variable and `Cultivar` into the model, and paste everything together:

```
> mymodel = paste('Alcohol ~ Cultivar',main,ints,sep='+')
```

```
> mymodel
```

```
[1] "Alcohol ~ Cultivar+Malic.acid + Alkalinity.ash + Proanthocyanins + Color.intensity"
```

To run this, we need to pass it to a modeling function through the `formula` function:

```
> wine.big = aov(formula(mymodel),data=wine)
```

```
> anova(wine.big)
```

```
Analysis of Variance Table
```

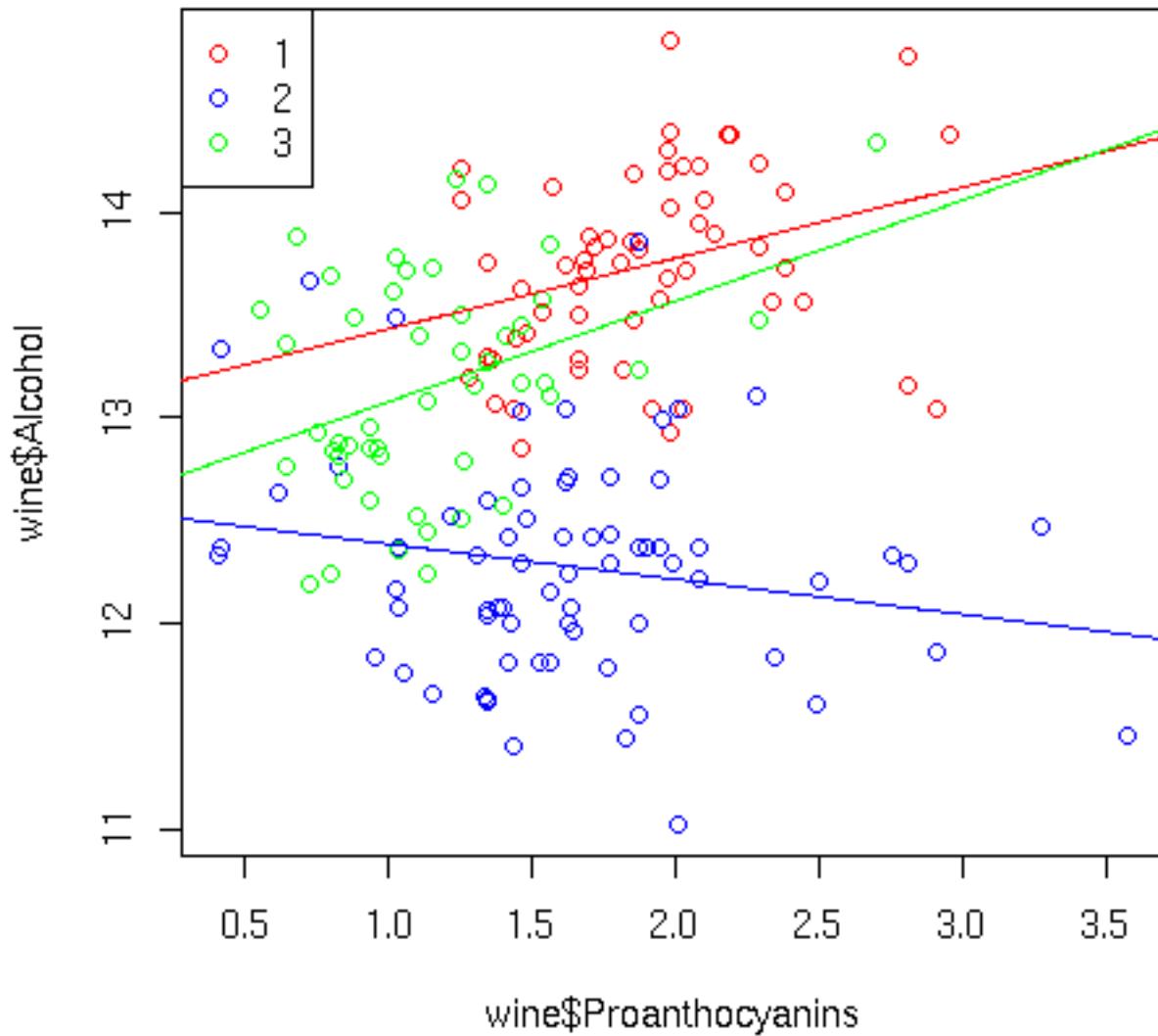
Response: Alcohol

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Cultivar	2	70.795	35.397	154.1166	< 2.2e-16 ***
Malic.acid	1	0.013	0.013	0.0573	0.81106
Alkalinity.ash	1	0.229	0.229	0.9955	0.31993
Proanthocyanins	1	0.224	0.224	0.9755	0.32483
Color.intensity	1	4.750	4.750	20.6808	1.079e-05 ***
OD.Ratio	1	0.031	0.031	0.1335	0.71536
Proline	1	0.262	0.262	1.1410	0.28708
Cultivar:Malic.acid	2	0.116	0.058	0.2524	0.77727
Cultivar:Alkalinity.ash	2	0.876	0.438	1.9071	0.15194
Cultivar:Proanthocyanins	2	1.176	0.588	2.5610	0.08045 .
Cultivar:Color.intensity	2	0.548	0.274	1.1931	0.30602
Cultivar:OD.Ratio	2	0.415	0.207	0.9024	0.40769
Cultivar:Proline	2	1.160	0.580	2.5253	0.08328 .
Residuals	157	36.060	0.230		
---					
Signif. codes:	0	'***'	0.001	'**'	0.01
	*	'*'	0.05	'.'	0.1
					' 1

As expected there isn't anything too startling. If we wanted to investigate, say, the `Cultivar:Proanthocyanins` interaction, we could look at a scatter plot using separate colors for the points and corresponding best regression lines for each `Cultivar`:

```
> plot(wine$Proanthocyanins,wine$Alcohol,col=c('red','blue','green')[wine$Cultivar])
> abline(lm(Alcohol~Proanthocyanins,data=wine,subset=Cultivar==1),col='red')
> abline(lm(Alcohol~Proanthocyanins,data=wine,subset=Cultivar==2),col='blue')
> abline(lm(Alcohol~Proanthocyanins,data=wine,subset=Cultivar==3),col='green')
> legend('topleft',legend=levels(wine$Cultivar),pch=1,col=c('red','blue','green'))
```

The plot appears below:



## 16.7 Alternatives for ANOVA

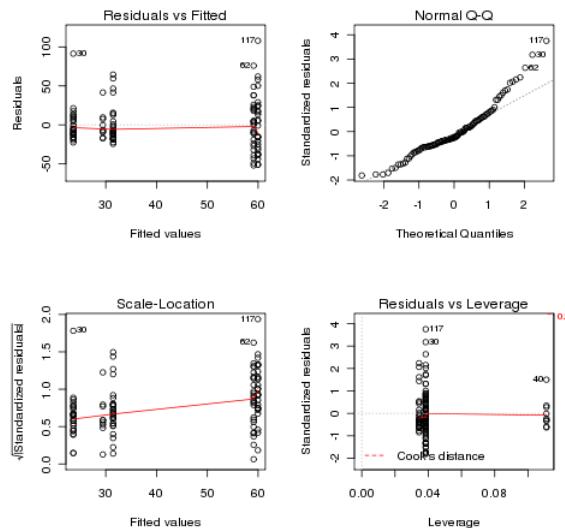
Not all data is suitable for ANOVA – in particular, if the variance varies dramatically between different groups, the assumption of equal variances is violated, and ANOVA results may not be valid. We've seen before that log transformations often help with ratios or percentages, but they may not always be effective.

As an example of a data set not suitable for ANOVA, consider the builtin data set `airquality` which has daily measurements of ozone and other quantities for a 153 day

period. The question to be answered is whether or not the average level of ozone is the same over the five months sampled.

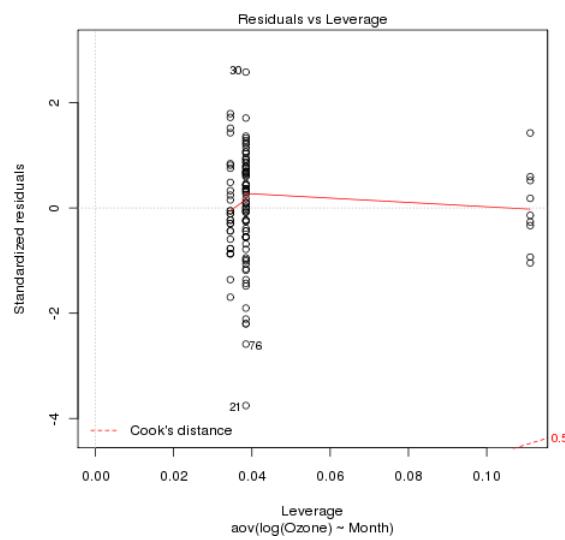
On the surface, this data seems suitable for ANOVA, so let's examine the diagnostic plots that would result from performing the ANOVA:

```
> airquality$Month = factor(airquality$Month)
> ozone.aov = aov(Ozone~Month,data=airquality)
> plot(ozone.aov)
```



There are deviations at both the low and high ends of the Q-Q plot, and some deviation from a constant in the Scale-Location plot. Will a log transformation help?

```
> ozonel.aov = aov(log(Ozone)~Month,data=airquality)
> plot(ozonel.aov)
```



In this case, the transformation didn't really help.

It might be possible to find a more suitable transformation, but we can also use a statistical test that makes fewer assumptions about our data. One such test is the Kruskal-Wallis test. Basically, the test replaces the data with the ranks of the data, and performs an ANOVA on those ranks. It assumes that observations are independent from each other, but doesn't demand equality of variance across the groups, or that the observations follow a normal distribution. The `kruskal.test` function in R performs the test, using the same formula interface as `aov`.

```
> ozone.kruskal = kruskal.test(Ozone~Month,data=airquality)
> ozone.kruskal
```

Kruskal-Wallis rank sum test

```
data: Ozone by Month
Kruskal-Wallis chi-squared = 29.2666, df = 4, p-value = 6.901e-06
```

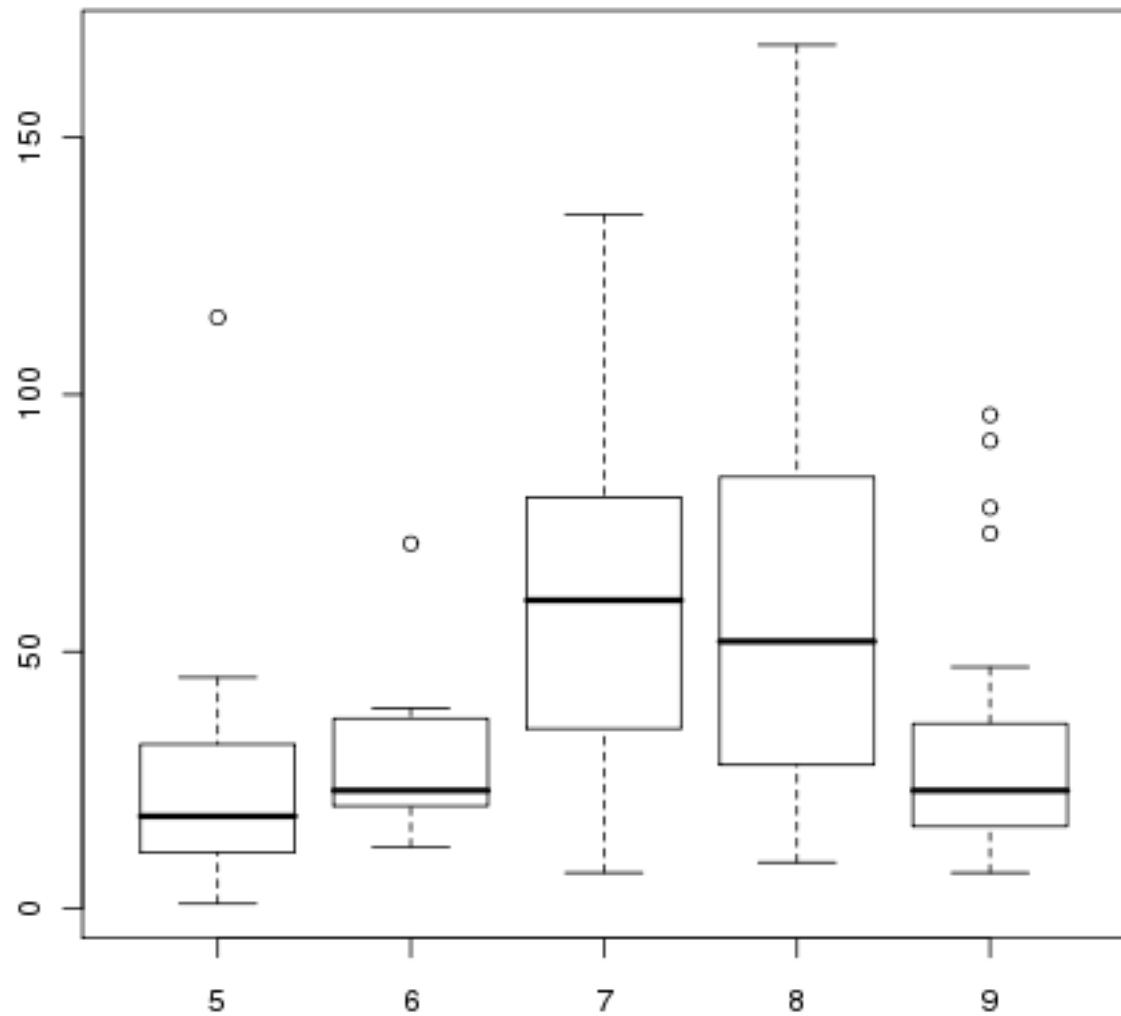
All that's reported is the significance level for the test, which tells us that the differences between the Ozone levels for different months is very significant. To see where the differences come from, we can use the `kruskalmc` function in the strangely-named `pgirmess` package. Unfortunately this function doesn't use the model interface – you simply provide the response variable and the (factor) grouping variable.

```
> library(pgirmess)
> kruskalmc(airquality$Ozone,airquality$Month)
Multiple comparison test after Kruskal-Wallis
p.value: 0.05
Comparisons
  obs.dif critical.dif difference
5-6 57.048925    31.85565      TRUE
5-7 38.758065    31.59346      TRUE
5-8 37.322581    31.59346      TRUE
5-9  2.198925    31.85565     FALSE
6-7 18.290860    31.85565     FALSE
6-8 19.726344    31.85565     FALSE
6-9 54.850000    32.11571      TRUE
7-8  1.435484    31.59346     FALSE
7-9 36.559140    31.85565      TRUE
8-9 35.123656    31.85565      TRUE
```

Studying these results shows that months 6,7, and 8 are very similar, but different from months 5 and 9.

To understand why this data is not suitable for ANOVA, we can look at boxplots of the Ozone levels for each month:

```
> boxplot(with(airquality,split(Ozone,Month)))
```



The variances are clearly not equal across months, and the lack of symmetry for months 5 and 6 brings the normal assumption into doubt.