P9489 Practicals and Exercises
Spring 2013

Charles DiMaggio, PhD

March 8, 2013

Contents

Chapter 1

Part I: Working in Base R

The best way to start getting comfortable with a new language is to use it. This series of exercises reviews some of the content we've discussed during lecture, and introduces some other basic concepts about working with data in R. It's important that you actively type in the commands and review the results rather than just read. Try to briefly answer the questions that come up along the way. Don't worry if everything doesn't make a lot of sense during the earlier exercises. It will. And there's an answer key available if you become too frustrated.

Start R and open a new script document. The approach may differ in minor details depending on whether you are on a Windows or Mac machine.

1.1 working with objects

1.1.1 vectors

A vector is one of the most basic R objects, and a good place to start using R for epidemiology. Consider the following epidemiologic scenario taken from Tomas Aragon's book "Applied Epdemiology Using R". In 2003, 111 airplane passengers were exposed to a fellow passenger who was subsequently diagnosed with Severe Acute Respiratory Syndrome (SARS). Eight of 23 passengers who sat "close" to the index case [1] developed SARS. Ten of 88 passengers who sat "far" from the index case also developed SARS. Work through the following example to get a feel for conducting simple epidemiological calculations in R using nothing but two vectors.

using the assignment and concatenation operators to create vectors

Use the assignment operator (<-), and the concatenation function (c()) to create two vectors.

Create a vector called *case* that consists of two numbers, the number of *exposed* people who developed SARS, and the number of *unexposed* people who developed SARS. [2]

Using the same approach, create a second vector called *noncase* that consists of the number of exposed people who did not develop disease, and the number of unexposed people who did not develop disease. Again, try to name each element, using the same names you used for the noncase vector.

Print each vector to your console (screen)

[1] We won't go into how *close* is "close".

[2] As an additional step, try to name each element in the vector by putting the name in quotation marks and using an equal sign for the number assigned to that name.

combining vectors to create tables

Combine these two vectors to create a standard 2x2 Epi table, where the upper left cell (cell *a*) is the number of exposed cases of disease. You will need to use *cbind()* or *rbind()*. If you want to see a help page on how they work, simply type and enter ?cbind or ?rbind.

Which function (cbind or rbind) results in the correct table?

creating vectors and tables from individual observations

You won't often work directly with totals like in the SARS example. Usually, you'll a data set of individual observations which you need to total up. Let's go through the process of creating a table from these kind of data.

In this first step, you'll create a data set of observations using the *rep()* function which *repeats* the first argument by the number of times specified in the second argument. Begin by trying this.

```
> rep("hello", 5)
```

To create the data set from which you'll make vectors and a table, we'll walk through the process of repeating the character strings "case" and "noncase" to match the numbers in the SARS example. [3]

First, create a vector called *outcome* that consists of a total of 111 elements for the 111 people in the SARS example. Use the concatenation function to create the vector by first repeating the word "case" 18 times (this is because there were 8 cases in the exposed group and 10 cases in the unexposed group), and then repeating the word "noncase" 93 times (because 15 of the exposed and 78 of the unexposed people did not develop SARS). View or print out to your screen the vector *outcome*.

It's a little trickier to create the vector for *exposures*. We have to ensure that we have a vector that aligns with the outcome vector so that each case or noncase of disease correctly matches whether someone was exposed. We know that 8 of the exposed and 10 of the unexposed people became cases, and that 15 of the exposed and 78 of the unexposed did not become cases. The following code takes advantage of the ability of the *rep()* function to take a vector as the first argument. We begin by creating a temporary vector of the words "exposed" and "unexposed". We then feed that to the *rep()* function, which will cycle through those words in precisely the fashion we need. Again, type and submit the "exposure" vector you created to see what R did.[4]

```
> tmp <- c("exposed", "unexposed")
> exposure <- c(rep(tmp, c(8, 10)), rep(tmp, c(15, 78)))
> exposure
```

Bind the two vectors to create a single data set of observations called *sars.obs*

Then look at the first 4 rows of this new data object. To do this you will *index* the first 4 rows of the data object called *sars.obs* by using brackets.

```
> sars.obs[1:4,]
```

The bracket notation notation *1:4* is shorthand for the range 1 to 4. The comma after the number 4 is very important. It defines the dimension of the table from which you want the observations. Here we are interested in seeing the first 4 rows. *Rows* are the first dimension in an index and come *before* the first comma. Columns are the second dimension in an index, and come *after* the first comma. Compare the results of the indexing operation above to this one.

```
> sars.obs[,2]
```

We'll spend a lot of time talking about indexing in R.

[3]This is not something you are going to do in practice, but it a way to introduce you to working with a couple of functions, while at the same time setting up a data set we can use to illustrate creating a table from observations. Also, if you submit questions to lists like StackOverflow (which I recommend), providing toy data sets like this facilitates answers.

[4]Don't worry if this doesn't make a whole lot of sense right now.

The *table()* function can be used to tally up the counts or frequencies of data elements. So, for example, the following command will return the number of times the character strings "case" and "noncase" occur in the vector *outcome*.

```
> table(outcome)
```

To cross-tabulate the counts of two sets of observations, submit both vectors to *table()*.

```
> table(exposure,outcome)
```

As a quick example of how combining simple R functions can facilitate analysis, let's cross tabulate the two columns of the *sars.obs* data object, use indexing. Here, we submit the first column of the *sars.obs* object, and the second column of the *sars.obs* object, to the table function. *table()* then cross tabulates the frequencies of those two vectors and returns a convenient table object.

```
> sars.tab<-table(sars.obs[,1], sars.obs[,2])
> sars.tab
```

Notice that we are assigning the results of the *table()* function to a new object called *sars.tab*. This is good practice and a common way of working in R, because it opens up the possibility of exploring and manipulating the results, for example to calculate odds ratios and their confidence intervals.

1.1.2 matrix

In the examples above, we created a matrix by combining two vectors. Here, we'll spend a little more time considering matrices in R.

In this next example, we consider the University Group Diabetes Program. This was a placebo-controlled, multi-center randomized clinical trial from the early 1960's intended to establish the efficacy of treatments for type 2 diabetes. These data are often used in epidemiology training programs. [5] There were a total of 409 patients. 204 patients received tolbutamide. 205 patients received placebo. 30 of the 204 tolbutamide patients died. 21 of the placebo patients died.

As opposed to combining vectors like we did above, we'll create a 2x2 table from the reported results directly using the *matrix()* function. Read the help file for *matrix* by submitting ?matrix. Print the arguments for *matrix()* by using the *args()* function.

```
> args(matrix)
```

To get a feel for how the *matrix()* function works, let's create the classic "a,b,c,d" 2x2 table.

	Disease	No Disease
Exposed	a	b
Not Exposed	c	d

We'll use the *rownames* and *colnames* functions to name the rows and columns.

```
> tab <- matrix(c("a", "b", "c", "d"), 2, 2)
> rownames(tab) <- c("Exposed", "unExposed")
> colnames(tab) <- c("Disease", "noDisease")
> tab
```

This is not what we wanted. The default behavior for *matrix()* is to fill the matrix by columns. To override this behavior, you have to specify *byrow=T*

```
> tab <- matrix(c("a", "b", "c", "d"), 2, 2, byrow=T)
> rownames(tab) <- c("Exposed", "unExposed")
> colnames(tab) <- c("Disease", "noDisease")
> tab
```

[5]There was a surprising and troubling excess of cardiac deaths in the treatment group that has yet to be completely explained.

At this point you have enough information to create a 2x2 table from the UGDP data using the *matrix()* function. The table should contain the following data:

	Disease	No Disease
Exposed	30	21
Not Exposed	174	184

Once you've successfully created a table, try using the *addmargins()* function on it.

Calculating marginal totals

In this next bit of code, we'll use a function called *apply()* to to create column totals (dimension 2) of the treatment and placebo groups. We'll then use this total to calculate the proportion of deaths in each group.

Logically enough, *apply()* applies a procedure or function to one of the dimensions of a matrix. We'll discuss it in more detail during class, but for now, take a look at the help file.

```
> ?apply
```

The function takes 3 arguments. First, we identify the matrix object that we want it to work on, second we specify the dimension across which we want to apply the procedure (1=rows, 2=columns), and third we define function to apply, e.g. sum, mean, max, min, etc... Take a look at this code, then copy and run it.

```
> coltot <- apply(dat, 2, sum)
> risks <- dat["Deaths",]/coltot
> odds <- risks/(1-risks)
```

```
> coltot <- apply(dat, 2, sum)
> risks <- dat[1,]/coltot
> odds <- risks/(1-risks)
```

Notice that in the second line of code, we are indexing the matrix by the *name* of the row rather than its number. We could also have used

```
risks <- dat[1,]/coltot
```

Note that because we are specifying a row, we still place the numbered index before the comma. In the third line of the code, we used the "risks" object we created, to calculate the *odds* of disease for the treatment and placebo groups.

In the following code, we calculate two measures effect, the risk ratio and the odds ratio. Then, we use the *rbind()* (r for row) function to display the results.[6]

```
> risks
> risk.ratio<-risks/risks[2]
> odds.ratio <- odds/odds[2]
> rbind(risks, risk.ratio, odds, odds.ratio)
```

These are the kinds of steps that make up the epidemiological functions you will find in packages like **epitools** or **epicalc**.

1.1.3 lists

R functions are the "procedures" you will use to analyze data. Unlike some other programming languages, R functions return only the most minimal results. Those results are, though, usually just the tip of the iceberg. Accessing the rich trove of R results is part of the process of learning R. The basic approach is to save the results of an R function as a named object. You can then use indexing (and other functions) to access and manipulate the contents of the function-results object you created.

[6]You may want to take a moment to consider the implications of these results.

Most folks who write functions for R use *lists* to collect up and store the results of their functions. This is because a list is a very flexible R object that can store all types of data. You will have to gain some facility with lists to get the most out of R results. Let's run through an example of working with a list object that arises from applying a Fisher's exact test.

We'll work with a subset of the UGDP data table we created above. Create a table by dividing the "dat" table by 10. To do this, simply divide the "dat" matrix object by the number 10. Use the *round()* function to return the rounded result in whole numbers. Use the assignment operator to save the operation as an object called "dat2"

Now use the function *fisher.text()* on the "dat2" data table to return a Fisher's exact test.

The result is (clearly) not statistically significant. Let's take a closer look. Save the results of the fisher exact test to an object named "fish" (we could have called it anything). You will have to re-run the function using an assignment operator.

Use the *str()* function to examine the *structure* of this object.

We see that it is a list consisting of 7 named elements. Using this information, we can extract any of these elements. Try the following.

```
> fish$estimate
```

Say you wanted (for some reason) to extract just the lower confidence limit? The following will accomplish just that.

```
> fish$conf.int[1]
```

In this case, the authors of the function were kind enough to store the results in a named list. You can always use numeric indexing if you need to.

```
> fish[[2]][1]
```

create your own function and store the results as a list

If you're feeling brave, try the following.

```
> orcalc <- function(x){
+        or <- (x[1,1]*x[2,2])/(x[1,2]*x[2,1])
+        pval <- fisher.test(x)$p.value
+        list(data = x, odds.ratio = or, p.value = pval)}
```

You just created your own function to calculate odds ratios. The function stores results as a list.

Let's step through what you just did. First, you used a function called *function()* that, well, creates functions. Notice that unlike other functions, the parentheses are followed by an additional set of (curly or squiggly) brackets. First you specify the *arguments* in the parentheses. Here there is a single argument called "x". Then, in the brackets, you define what happens to "x" when you invoke the function.

You created an object called "or" by cross-tabulating the elements of x. Notice that presupposes that "x" is a 2x2 matrix. If you try this function on something that is not a matrix, it will fail spectacularly.[7] Next you created an object called "pval" by applying the *fisher.test()* function to "x" and, using the named list that results from using *fisher.test()* to extract out just the p value. Lastly, you collect the results in a list.

Let's see if it worked by running it on the UGDP data table.

```
> test<-orcalc(dat)
> test
> str(test)
```

[7]There are error message functions you can write that will inform users that they need to provide a 2x2 matrix. Since you're just writing this for yourself this isn't necessary.

1.2 class and mode of an R object

The class and mode of an R object affects how it behaves and how we can work with it. We'll explore classes and modes by looking at one very special R object called a "factor". A factor in R corresponds to what you might consider a categorical variable to be in epidemiology, but with one important difference: factors are stored as numeric, not character, vectors. Here, we create a data object of factors.

```
> x<-factor(1:7)
> mode(x)
> class(x)
```

While the mode of x is numeric, it's class is *factor*. This means that functions will treat it based on it's attributes as a factor. One important way factors are treated differently is in statistical modeling. Modeling functions like *lm()* (for linear model) will treat factors as categorical variables rather than continuous or numeric.

One important place where you may encounter factors is when you read in data. By default, the *read.table()* function automatically converts any character or "string" variable it encounters into a factor. This can be a convenience in some settings. Or a colossal headache in others. Imagine every patient and street name in a data file being it's own numbered factor. If you don't know this is going on, it can cause otherwise inexplicable behavior. I recommend using the *read.table()* option "stringsAsFactors=FALSE" to turn off this behavior. You can always convert a variable to a factor if you need to.

R provides many ways to work with the class and mode of an object. We will explore some of them. Begin by creating a data object called "y" that consists of three characters: "1", "2" and "3". Placing what would otherwise be read as numbers in quotations makes them characters. Remember to use the concatenation operator and to separate each item by a comma.

Use the functions *mode()* and *class()* to explore the object "y"

You see that for simple objects, the class is usually the same as the mode. Try to use the *sum()* function on "y".

```
> sum(y)
```

Functions will only work on objects for which they are intended. It does not make sense to try to add up characters, and R tells you so in its own inimitable style.

The *as.xxx()* function allows you to change an object's mode.[8] Try the *sum()* function again, only this time provide *as.numeric(y)* as the argument.

Now, try to sum up the factor object called "x" that you created above.

```
> sum(x)
> sum(as.numeric(x))
```

1.3 ordered factors

It's been my experience that factors can cause unwanted and sometimes inexplicable behavior in R, and unless they are absolutely necessary, I avoid them. One place where factors *are* necessary is for ordinal variables where you will likely want an ordered factor. Here we step through some material to make sense of ordered factors.

1.3.1 months example

Begin by creating a vector of month names called "mths"

[8] xxx stands in for mode names like "numeric", "character", "factor", etc...

```
>  mths = c("March","April","January","November","January",
+ "September","October","September","November","August",
+ "January","November","November","February","May","August",
+ "July","December","August","August","September","November",
+ "February","April")
```

Check the class and mode of this vector.

Examine a frequency table of the vector "mths". What is the most commonly named month? In what order is the vector?

To arrange these month names in an order that makes more sense, we will need to convert the vector from character to factor, and then order the factors. Begin by reading the help page for the *factor()* function. Pay particular attention to the options for *"levels="* and *"ordered="*

```
> ?factor
```

Convert the character vector "mths" to a factor called "f.mths" by submitting the following code.

```
> f.mths = factor(mths,levels=c("January","February","March", "April","May","June","July",
+  "October","November","December"),ordered=TRUE)
```

Print out a frequency table of "f.mths"

What is the mode of "f.mths"? What is it's class?

This is important information. The mode of "f.mths" differs from its class. The mode of an R object (e.g. numeric, character, logical) tells us its underlying data type. The class of an R object tells us how the object will be treated by functions.

An ordered factor is ordered by number, and that number is meaningful in R operations. In the factor vector "f.mths", "January", because it is ordered first, carries a lower value than "February".

Let's test this. First, look at the first and second elements of the object "mths".

```
> mths[1]
> mths[2]
```

Try to compare them with the following logical statement.

```
> as.numeric(mths[1])<as.numeric(mths[2])
```

Now, repeat the two previous bits of code with "f.mths".

```
> f.mths[1]
> f.mths[2]
```

```
> as.numeric(f.mths[1])<as.numeric(f.mths[2])
```

Try "adding" January to February.

We can coerce the "f.mths" object to *behave* as numbers by using *as.numeric()*. [9]

```
> as.numeric(f.mths)[1] + as.numeric(f.mths)[2]
> mean(as.numeric(f.mths))
```

This isn't possible with the character-mode "mths" object.

1.3.2 ses example

As I mentioned, in general, I will reserve the use of factors to settings where order matters. Because such ordinal variables can be important to epidemiological analyses, let's take one more look at the use of ordered factors in

[9]Getting ordered factors to behave as numbers can sometimes be a bit tricker than just using *as.numeric()*. See this helpful post.

R. Consider the following data which consists of 100 observations of a three-level ordinal variable that represents socioeconomic status.

```
> ses<- sample(c("Low","Medium","High"), 100, replace=TRUE)
```

Print out a table of the variable "ses".

Using the approach we took with the months variable above, create an ordered factor called "f.ses". Print out a table of the ordered factor vector.

1.4 exploring data sets

1.4.1 the infertility data set

Rather than entering data manually, or creating your own vectors and matrices, you may be working with existing data sets. R comes with some data sets pre-loaded that are useful in learning how to navigate data frames To view all the data sets available to you, submit the command *data()*.

Scroll through and read the brief descriptions of the data sets. We will look at the "infert" data set as an example. (You can follow along or use a different data set of your choice). You can find out more about the data set by submitting *?infert*.

Read the description. To use a data set, you have to *load* or copy it as an *object* into your work space or active directory with the *data()* function.

```
> data(infert)
```

When first working with a data object, it helps to have some idea of its over all structure (*str()*). Use these this function on the "infert" data set. [10]

- What kind of object is the infertility data set?
- How many observations and how many variables are in the infertility data set?

It is also often helpful to look at the data set itself. If you're used to working with spreadsheet programs like Excel, where the data are displayed by default, transitioning to a program like R, where the data are tucked away in memory, can be difficult. In R (and other programs, like SAS) you have to issue explicit commands to display the data set. The simplest way to do that is to type the name of the data object. Try this with the "infert" data frame.

This quickly becomes unwieldy with large data sets. Usually it's sufficient, when first becoming acquainted with a data set, to display the first few rows of observations. Rows are the first dimension when indexing an R data frame. Try the following command (the comma is important).

As is often the case, someone came up with a shortcut. Try the *head()* function on the "infert" data set.

1.4.2 preliminary statistics

After familiarizing yourself with a data set, it's a good idea to tabulate some frequencies and calculate some descriptive statistics to begin to understand the information.

We may want to know the mean age of the women in the study.

```
> mean(infert$age)
```

[10] *str()* may be the single most useful function in R.

Notice what we did. The convention in R is to identify a variable by the dataset name followed by the dollar sign ($) and the variable name. [11] You may create or come across data sets that do not have named variables or columns. In that case you would use indexing. Calculate the mean parity for the infertility data set using the index for that column.

Verify that you correctly identified the "parity" column by using the dollar-sign naming convention to calculate the mean parity.

The *fivenum()* function returns more detailed information about continuous variables (minimum, first quartile, median, third quartile, maximum). Calculate the five-number summary for age and parity.

You will find that in R, there are multiple ways of getting the same information and that there are often convenience functions based on other, more elementary functions. For example, the *summary()* function returns information about a continuous variable that includes the 5-number summary. Compare the results of the *summary()* function for age to those of *mean* and *fivenum()* above.

1.4.3 frequency tables

Epidemiologists spend a lot of time counting things. The most elementary way to count up occurrences of some outcome in an R data set is with the *table()* function.

To specify a cross tabulation, simply separate the two variables by a comma. Try requesting a cross tabulation of parity by education.

1.4.4 installing and using a package

If you are coming from SAS (as I did), you can think of *table()* as similar to PROC FREQ. It is, though, rather a bare-bones experience. Usually this a strength, as R allows you to the flexibility to explore and manipulate results, rather than returning verbose output. In this case, I prefer the SAS output. And as is often the case, someone has already thought of that and created a function to return a more satisfying SAS-like cross tabulation. It's called *CrossTable()* and it's in the package *"gmodels"*. User-contributed functions and packages are one of the great strengths of R. Let's walk through how you install and use a new package. [12]

The first step is to install the *"gmodels"* package.

```
> install.packages("gmodels")
```

You may be prompted to choose a "CRAN mirror". "CRAN" stands for the Comprehensive R Archive Network. A "mirror" is a server that makes the package files available. The sites are mirror-like in that it should not matter which one you choose. There may be some (small) speed benefit to choosing a site geographically close to you. After you select a CRAN mirror, R will install the files.

Installing a package does not mean that it is ready for use. Installing a package means it is now available on your machine if and when you want to use it. You still need to bring the files or library that make up the package into your workspace. To bring the package into your workspace, you will use the *library()* function.

```
> library(gmodels)
```

A couple of important notes. When you install the package using *install.packages*, you need to put it in quotes. When you load the package into your workspace using *library()* you don't need quotes. You only need to *install* a package with *install.packages* once from CRAN. You need to *load* a package with *library()* every time you want to use it.[13]

Now let's see what a cross tabulation looks like using *CrossTable()*.

[11] You can *attach()* a data set to avoid having to type the dataset$ prefix, but can cause problems down the line, and you must remember to *detach()* it as you move along in your analyses

[12] This might not work on the classroom machines, which may block access to executable files.

[13] If there are packages that you use a lot, there are ways to automatically load them when you start up R. The details vary depending on your operating system, but they all involve editing a text file that R looks at every time you start up R.

```
> CrossTable(infert$parity, infert$education)
```

1.4.5 plots

A picture is worth a thousand words. The workhorse of R graphing is the *plot()* function. Often *plot* is smart enough to return the type of graph we want.

```
> plot(infert$education)
```

But often, it is not. Try using *plot()* to graph parity or age

```
> plot(infert$parity)
> plot(infert$age)
```

Using *table()* to prepare the data is a common approach to getting informative initial plots of count or integer data.

```
> plot(table(infert$parity))
```

Try plotting a table of age values.

Sometimes, *plot()* is just the wrong choice.

```
> hist(infert$age)
```

1.4.6 the US Arrest data set

Base R comes with a data set about violent crime rates in the United States called “USArrests”. Try the following:

1. Load the USArrests data set.
2. How many observations and how many variables are in the USArrests data set?
3. What was the average murder rate in the US? What was the highest murder rate and what was the lowest murder rate?[14]
4. Plot a histogram of assault arrest rates in the United States.

1.5 calculations

R is a remarkably powerful and robust statistical computing language. It is also a humble calculator and can perform spreadsheet calculation. Here are a couple of exercises to get you used to doing calculations in R.

1.5.1 temperature conversion

The following formula converts temperatures from the Fahrenheit scale to the Celsius scale: $C = (F - 32)x\frac{5}{9}$

Use the assignment operator to create a vector data object called "f" of Fahrenheit temperatures from 95 to 104, incrementing by 0.1[15] Create a vector of celsius values called "c", using the formula above to convert the Fahrenheit values in the vector "f".

Use the cbind() function to create a matrix based on the Fahrenheit and Celsius temperature vectors you created.[16] Search for help on cbind if you need to using *help(cbind)* or *?cbind*.

[14]Want to know which state had the lowest rate? Type and enter the following: USArrests[USArrests$Murder == min(USArrests$Murder),]

[15]Hint: Use the sequence function. *seq(95,104, 0.1)*

[16]extra credit if you can name the columns

1.5.2 body mass index

The formula for body mass index is $BMI = kg/m^2$ The conversions for weight and height are: $1kg = 2.2lb$ and $1m = 3.3ft$

Calculate your body mass index. What would your body mass index be if you lost 5%, 10% or 15% of your current weight?

1.5.3 AIDS transmission

In his book *Innumeracy*, John Allen Paulos states that the probability of transmitting the AIDS virus from an infected to an uninfected person in one heterosexual act is 1/500. Using the basic probability concept that the *complement* of an act *not* occurring is 1 minus the probability of an act occurring, what is the probability of *not* transmitting the AIDS virus in a single heterosexual act?

What is the probability of *not* transmitting the AIDS virus in a daily heterosexual act over the course of a year? [17]

Recalculate the above two probabilities with the risk decreased to 1/5000 through condom use.

1.5.4 cumulative risk

In epidemiology we can use the exponential formula to calculate the risk of an event:

$$R(0,t) = 1 - e^{-\lambda t} \tag{1.1}$$

where t is the time to the event and λ is the rate at which the event occurs.

Assuming that in the US influenza occurs at a rate of 10 infections per 100,000 person-years of observation, we can create a table of the cumulative risk of influenza in a population over 1, 5 and 10 years.

We begin by defining the variables in our calculation. First we define the rate at which influenza is transmitted. Then we create a vector of three elements for the three years in which we are interested.

```
> lambda<-10/100
> t<-c(1,5,10)
```

Substituting the variables we created above, write an equation to create a vector of risks called "risk" [18]

Now, use *cbind()* to create a matrix or table of the rate, years and cumulative risk.

1.5.5 attributable fractions

The attributable fraction is the proportion of disease among exposed people that is caused by an exposure. We can calculate it with the formula:

$$AF = (Risk_{exposed} - Risk_{unexposed})/Risk_{exposed} = 1 - \frac{1}{RR}$$

Assuming that the risk of some disease is 1 in 100,000 among unexposed people, create a table of attributable fractions for exposures that increase the risk to 1 in 50,000, 1 in 10,000 and 1 in 1000.

An important concept about attributable fractions is that the calculation applies to the *exposed* cases. There are, invariably, cases in a population among *unexposed* individuals. We can adjust for this and calculate a measure called

[17]hint: multiply individual probabilities to get an over all probability

[18]Hint: The exponential function is *exp()*

the *population* attributable fraction by multiplying the attributable fraction by the proportion of all cases represented by the exposed cases: $AF_{pop} = AF \cdot \frac{exposedcases}{totalcases}$

Let's say that 500 of a total of 1400 cases of our hypothetical disease occurred exposed individuals, with the remaining 900 cases occurring in unexposed individuals. Add a column to the matrix you created above for the population attributable risk.

1.6 rates, risks, odds and logits

Epidemiology often involves measuring rates and risks.

Rates are how frequently something occurs over some period of time. Rates can never be negative and (depending on how short a period of time we are considering), range from zero to infinity. A risk is a probability of an event occurring, and is classically calculated as the number of times something occurs over the number of times in which it *can* occur. Like all probabilities it is confined to values between 0 and 1.

Much of the statistical modeling done in epidemiology has involved transforming risks, which are constrained to 0 and 1, into outcomes that are amenable to linear models that can range from negative to positive infinity.

One useful approach to this problem has been the logistic transformation. It involves two steps. First, we take our risk measurement, and extend it to include values beyond 1. We do this by taking the *odds* of the probability. Odds are simply the probability of an event occurring over the probability of the event *not* occurring, or $P/1-P = Risk/1-Risk$. Second, we take the *log* of the odds. This effectively establishes a linear model that can range from minus to plus infinity.

We can use R to illustrate this process and how it affects the basic shape of a model. We'll use the *curve()* function which plots a formula. (Look at the help page of *curve()*). First we plot a series of *odds*

```
> curve(x/(1-x), 0, 1)
```

Second, we plot the log of the odds.

```
> curve(log(x/(1-x)), 0, 1)
```

1.6.1 HIV transmission

Let's explore risks, odds and log odds. The following table lists the rate of HIV transmission by type of exposure.

	Rate per 10,000 exposures
Blood transfusion (BT)	9,000
Needle-sharing injection-drug use (IDU)	67
Receptive anal intercourse (RAI)	50
Percutaneous needle stick (PNS)	30
Receptive penile-vaginal intercourse (RPVI)	10
Insertive anal intercourse (IAI)	6.5
Insertive penile-vaginal intercourse (IPVI)	5
Receptive oral intercourse on penis (ROI)	1
Insertive oral intercourse with penis (IOI)	0.5

For each exposure type, calculate the risk, the odds and the log odds. Combine them into a matrix. Using a simple plot() command with the option *type="l"* for line, graph the odds and the log odds.

```
> plot(odds, type="l")
```

```
> plot(logit, type="l")
```

1.6.2 Scottish Health Study

In the Scottish Health Study 85 of 1821 people who lived in rented apartments had coronary artery disease, compared to 77 of 2477 people who owned their homes.

- Create a named 2-by-2 matrix object from this data
- Calculate the row totals. [19]
- Calculate the risk ratio[20] and odds ratio for these data. [21]
- Create a matrix of the risks, relative risks, odds and odds ratios

1.7 cross tabulations and stratified analysis

1.7.1 UGDP

Let's return to the University Group Diabetes Program data. This time we'll work with a text file of individual observations. I downloaded these data from Tomas Aragon's site. There are three variables: Status (Survivor or Death), Treatment (Placebo or Tolbutamide) and Age Group (older or younger than 55).

Begin by using the *read.csv()* function to read the data into R. To do this, go to the course website and locate the file called "ugdp" under the right-hand resource list. Right click on the link to get the path information and paste this information into the *read.csv()* statement to read the data into R. Remember to put the path in quotes. Use the options *header=TRUE* and *stringsAsFactors=FALSE* options.[22] Remember to assign the function to an object. I will be using the name "ugdp".

- How many observations are in the data set?
- How many participants received Tolbutamide?
- How many participants were under the age of 55?

Calculate the odds ratio for the association of tolbutamide with death. There are a couple of ways you can do this. You can create a matrix by cross tabulating treatment with outcome and do a cross multiplication ($\frac{ad}{bc}$) by indexing the individual cells by row and column. Or you can take the approach from the Scottish Health Study exercise. With either approach, you may find that it will help to reverse the rows To reverse the rows of a matrix, index it with [2:1,].

Now calculate a confidence interval for the odds ratio using the following formula which comes from Kenneth Rothman's "Epidemiology: An Introduction":

$$OR_{CI} = e^{ln(OR) \pm 1.96\sqrt{\frac{1}{a}+\frac{1}{b}+\frac{1}{c}+\frac{1}{d}}} \tag{1.2}$$

Where a,b,c,d refer to the cell values in a cross-tabulation of exposure by outcome.

epitab()

Another nice thing about R for epidemiologists, is that many folks have taken the time to write the kinds of formulas we frequently use, and have been kind enough to share them in the form of R packages. One such package is "epitools".

- load the epitools package [23]

[19] Hint: Use apply()

[20] a risk is the number of occurrences over the population; a risk ratio is the risk in the exposed over the risk in the unexposed; odds are risk over 1 minus the risk; and an odds ratio is the odds in the exposed over the odds in the unexposed

[21] Hint: Use the row totals to calculate risks for each groups. Use these two risks to calculate a risk ratio. Use the risks (again) to calculate the odds for the two groups. Use these two odds to calculate and odds ratio

[22] if you don't know what those options do, you look at the help file for read.table by typing *help(read.table)*

[23] library(epitools)

- calculate the odds ratio for the ugdp data using the *epitab()* function. [24]

1.7.2 the cars data set

The text file "cars" contains the following variables:

safety:	safety score (1=Below Average, 0=Average or Above)
type:	type of vehicle (Sports, Small, Medium, Large, and Sport/Utility)
region:	manufacturing region (Asia, N America)
weight:	weight of the vehicle in thousands of pounds

unstratified analysis

- Go to the course website and locate the file under the right-hand resource list. Right click on the file to get the path information and paste this information into a read.table() statement to read the data into R. Remember to put the path in quotes. Use the header=TRUE and as.is=TRUE options.[25]
- Print out the first five observations using head() or indexing. You will have had to have created a data object when reading in the file. Look at the structure of the car data object using str().
- Use prop.table() to determine the proportion of cars manufactured in North America. [26]
- Use table() to create a matrix object cross tabulation of region by safety. Use [,2:1] indexing to reverse the columns of this matrix so unsafe cars are in the first column.
- What are your initial observations about the safety of vehicles manufactured in Asia vs. North America?
- Load the epitools library, and run epitab on the region by safety matrix you created. What is the odds ratio for the association of region and safety?[27]
- If you want, test the statistical significance of the association of region and safety by using chisq.test() to calculate a chi square statistic.

1.7.3 stratified analysis

There may be a third factor that accounts for the apparent association of one factor with another. In epidemiology, this is termed *confounding* and the third factor is called a *confounder*. In this example, the size of vehicles manufactured in Asia vs North America may be confounding the association of region with safety because the vehicles manufactured in North America may be larger. We can look at the effect of a third variable or possible confounder by *stratifying* our cross tabulations by levels of this third factor.

- Dichotmize the *type* variable into *big* and *small* cars, where *big* refers to *Large and Sport/Utility*, and *small* refers to all *others*. You will have to do some indexing.
  - Create a logical vector called “big” by using the assignment operator and a statement for Large or Sport/Utility vehicles. Remember to use the double equal sign (= =) equivalence operator and put the variable names in quotes.
  - Create a new variable, attached to the "car" data frame you read in, called *cat*. Assign the character "big" to those rows of observations that are indexed as TRUE for *big* and assign the character "small" to those rows that are indexed as FALSE for *big*.[28]

[24]Search for help on the epitab() function to learn a little more about it

[25]if you don’t know what those options do, you should look at the help file for read.table by typing help(read.table)

[26]hint: run prop.table on a table object created with table()

[27]as noted in previous exercises, an odds ratio may be considered a way to approximate the risk of one factor relative to another factor

[28]It’s important that you begin to get comfortable with these kinds of manipulations and indexing, but I understand if you feel the need to peek at the solution...

  - Print out the new variable and the first 5 observations of the data frame to make sure your variable is formatted correctly.

- Use xtabs() to create a 3-level array object of region by safety by this new cat variable you just created.[29] This is our stratified table. Look at the structure of this array object. Print out the array object and look at the cell numbers for the association of region with safety. Look at the structure of the array.

- We now will look at the association between region and safety within the two strata of car size category. Do do this, we will first create two separate 2-by-2 matrix objects, one for each level of car category. Then we will calculate an odds ratio for the association of region with safety for each table. The first step is to separate out the two levels of the array object, and this will (again) require indexing.

  - create a two-by-two matrix of the large vehicle strata of the array by indexing the array using the name of the level you want. If you need to, print out the array again to remind yourself of the names of the levels. [30]

  - Using the same approach, create a matrix object from the small vehicle strata of the array.

- Once you have two matrices based on the levels of the array, use epitabs() to calculate odds ratios. *You will have to reverse the columns* to put riskier vehicles in the first column. Use *args(epitabs)* to learn how to do this within the *epitabs()* function itself.

- Does the apparent association of safety by region change when you look at it within strata of car size? If it does, you may want to calculate a measure of association that controls for that third, possibly confounding, variable. One such statistic is called the Mantel-Haenszel Odds Ratio.

- There is an R function that will return a Mantel-Haenszel odds ratio. To search for it, type:

  ```
  ??mantel
  ```

  You will get a number of functions that include some variant of your search term. Click on the info for the mantelhaen.test() function. Read how it is used and the arguments it takes. Run the function on your 3-level array. You will have to *reverse the columns of the array dimension for safety* by using the 2:1 index trick for the correct dimension. [31]

- The results of the mantelhaen.test() will return an odds ratio when submitted a 2-by-2 object. This odds ratio is a measure of the association between the first two levels of the array (here region and safety) stratified by or *controlling for* the third level (here car size category). Compare this estimate to that for the unadjusted estimate from the first part of this exercise.

stratified analysis of UGDP data set

Let's return for a moment to the UGDP data set you created in the previous exercise. You will by know appreciate that there was an apparent and paradoxical association of tolbudatamide with death. Compare the unadjusted measure of effect with an adjusted measure of effect obtained with a Mantel-Haenszel odds ratio. Is there any evidence of confounding?

1.8 making sense of the **apply()* family of functions

Just like the popular Apple marketing phrase "There's an APP for that", you could say "There's an *APPly for that." If one exists, using one of the **apply()* functions is invariably more efficient and faster that coming up with a loop or other hack. There are enough of them, that you can get lost in the thicket of variants. It helps to work through some examples to see the differences. [32]

[29] Again, if you're unsure about how to use a function, use help.
[30] This is why I like to use xtabs() to create these arrays. It automatically names the levels making indexing a little easier
[31] Again, better that you try to work this out for yourself, but the answer is in the solutions section
[32] Much of this discussion comes from a very nice post on stack overflow

apply()

Use *apply()* to apply a function to the margins or dimensions of a matrix or array. In plain(er) English, to get things like totals for rows or columns. For example:

```
> m <- matrix(round(rnorm(16, 20, 10)),4,4)
> m
> apply(m,1,min)
```

Here's what happened. We applied the *min()* function to the first dimension of the matrix. The first dimension is the row. So, for the first row ([1,]) the minimum value is 21. For the second row ([2,]) the minimum value is 16. For the third row ([3,]) the minimum value is 14. And for the fourth row ([4,]) the minimum value is 8.

Let's try the sum.

```
> apply(m,1,sum)
```

These are the row totals. Your turn. Write a statement to calculate the column totals.

Now for a three-dimensional array. We have data on the ages of people in a placebo controlled study. We have information on the participants gender, race (white, african american, hispanic, other), and treatment status. We can imagine an array with three dimensions (age, race, treatment):

```
> a<-array(round(rnorm(16, 20, 10)), dim=c(2,4,2))
> a
```

How many rows are there in this data object? You might be tempted to say *4*, but there are, in fact, *2 rows*. One for each gender. We designed it that way. So, if we wanted to compare the ages of males and females, we would get two results.

```
> apply(a,1,mean)
```

Now, you try. What is the mean age of treated females vs. untreated females? Let's make life a little easier by naming the dimensions.

```
> sex<-c("male", "female")
> race<-c("w", "b", "h", "o")
> tx<-c("yes", "no")
> dimnames(a)<-list(gender=sex, race=race, treatement=tx)
> a
```

Now, let's use apply to get the mean age of treated vs. untreated females.

```
> apply(a,c(1,3),mean)
```

apply() is useful for these kinds of things. If you are just interested in marginals for matrices, you're better off using *colMeans, rowMeans, colSums, rowSums*, or the even more convenient *addmargins*

1.8.1 tapply()

You don't really see a lot about *tapply()* in the general R community, but it's a fairly straightforward way to group or stratify observations, so it's useful for epidemiologists. The key is the idea of an indexing or grouping variable. Say we have the ages of 10 patients drawn from 5 clinics and we want the mean age for each clinic. [33]

```
> dat<-data.frame(age=(round(rnorm(10,5,1))), clinic=sample(c("a","b","c","d","e")))
> tapply(dat$age, dat$clinic, mean)
```

We can get more involved, say in addition to age and clinic, we have treatment status, and let's up the number to 100 patients. Now let's get the mean ages for patients by clinic and treatment status.

[33]Note that I'm letting *data.frame()* recycle additional values for the clinic.

```
> dat<-data.frame(age=(round(rnorm(100,5,1))), clinic=sample(c("a","b","c","d","e")), trea
> b<-tapply(dat$age, dat[,c(2,3)], mean)
> b
```

There is another function, called *by()* that will accomplish essentially the same thing as *tapply()*.

```
> c<-by(dat$age, dat[,c(2,3)], mean)
> c
```

The main difference is that *tapply()* returns a matrix, and a matrix is easy to work with in R, i.e. to extract elements and conduct additional analyses. *by()* returns a "by" object, that might be less tractable to additional analyses.

```
> class(b)
> class(c)
```

1.8.2 lapply()

The "l" in *lapply()* refers to *list*. Use this function when you want to apply a function to each "bin" of a list in turn and get a list back. As I've mentioned, because it's so flexible, lists are used a lot in R, especially for the results or summaries of statistical tests and functions.[34] So having a way to operate directly on the elements of a list can be useful when extracting and working with the results of statistical models in R. Let's see how *lapply()* works.

First, we'll create a simple list of numeric objects:

```
> l<-list(a=5, b=1:5, c=round(rnorm(20,2,1)))
> l
```

Now, let's find the length and the sum of each bin.

```
> lapply(l,length)
> lapply(l,sum)
```

Notice that the results are lists themselves.

1.8.3 sapply()

sapply() is like lapply() but it returns a *simplified* (hence the "s") result, i.e. a vector rather than a list.

```
> sapply(l,length)
> sapply(l,sum)
```

sapply() can do some more advanced and tricky things, like coercing results into a matrix or into an array, but that is another subject, and we won't go into here. There is a version (essentially) of *sapply()* called *vapply()* that can be tweaked to run more quickly, but I've never used it.

1.8.4 mapply()

mapply() is one of those functions that doesn't seem to make sense until you have a use for it, and then you wonder how you could possibly have accomplished your task without it. In brief, it will apply a function to all the first elements of a list or set of vector, then to all the second elements of a list or set of vectors, then the third, etc... To get a sense of what it does, imagine a list object that contains three "bins", bin1=a,b,c; bin2=d,e,f; bin3=g,h,i. If you pass the *sum* function and this list to *mapply()*, it will add things up as follows: a+d+g b+e+h c+f+i

Let's see it in action.

[34] *lapply()* lurks beneath many other functions in R

```
> v1<-1:5
> v2<-6:10
> v3<-11:15
> v1;v2;v3
> mapply(sum, v1,v2,v3)
```

1.8.5 other *apply functions

There are other (less common) *apply functions. *rapply()* allows you to control how functions are applied to list bins. So you can, for example, specify that the function is applied to just the first element of each list bin. Then there is *eapply()* that allows you to apply functions to R environments. Clearly, you'd have to know what an R environment is, and have a reason to manipulate it. If not, you will not have use for *eapply()*.

1.9 indexing to manipulate data

Indexing is the key to working with and manipulating R data. There are three ways to index data in R:

- position
- logical vector
- name

Run the following to see an example of each type of indexing.

```
> x <- c(chol = 234, sbp = 148, dbp = 78, age = 54)
> x
> x[1] # by position
> x[x>150]# by logical
> x["chol"] # by name
```

You can use indexing to replace or change a data entry.

```
> x[1] <- 250 #by position
> x[x<100] <- NA # by logical
> x["sbp"] <- 150 # by name
> x
```

Let's look at the three approaches to indexing in a bit more detail.

1.9.1 indexing vectors

by position

```
> x<-1:40
> x[11] #only the 11th element
> x[-11] #exclude the 11th element
> x[11:20] #members 11 to 20
> x[-(11:100)] # all but members 11 to 20
```

by logical

R uses the following symbols to establish logical relationships between variables.

```
==   IS equivalent to
!    is NOT
&    AND
|    OR (if either or both comparison elements are TRUE)
xor  EITHER  (element-wise exclusive or operator, if either,
              but not both, comparison elements TRUE)
&& || special operators, control flow in "if" functions,  only the first
              element of logical is used.
```

In addition, the *which()* function returns an integer vector of positions from a Boolean operation, for example

```
> age <- c(8, NA, 7, 4)
> which(age<5 | age>=8)
```

Here, the positions 1 and 4 in the vector "age" meet the Boolean definition.

To use a logical expression to index R data:

1. create a logical vector
2. use the logical vector to index data

Let's take a look at an example. First create three vectors of related data.

```
> names<-c("dopey" , "grumpy" , "doc" , "happy" , "bashful" ,
+ "sneezy" , "sleepy" )
> ages<-c(142, 240, 232,  333,  132, 134,  127)
> sex<-c("m" , "m" , "f" , "f" , "f" , "m" , "m")
```

Now, do some indexing.

```
> young <- ages < 150 #create logical vector
> names[young] #index name vector using logical vector
> names[!young] # old dwarves
> male<- sex == "m" #logical vector male dwarves
> names[male] #index names using logical vector males
> names[young & male] # young male dwarves
```

One important use of logical indexing is to categorize a continuous variable.

```
> # simulate vector with 1000 age values
> age <- sample(0:100, 1000, replace = TRUE)
> mean(age) ;  sd(age)
> agecat <- age # make copy
> #replace elements agecat with strings for q category
> agecat[age<15] <- "<15" # creating character vector
> agecat[age>=15 & age<25] <- "15-24"
> agecat[age>=25 & age<45] <- "25-44"
> agecat[age>=45 & age<65] <- "45-64"
> agecat[age>=65] <- "65+"
> table(agecat) # get freqs
```

1.9.2 indexing matrices and arrays

A vector has only one dimension, so it is indexed by a single number in a bracket. To index matrices and arrays, you have account for their additional dimensions.

indexing matrices

Create the following matrix.

```
> m<-matrix(round(rnorm(16,50,5)),2,2)
> dimnames(m)<-list(behavior=c("type A", "type B"),
+ MI=c("yes", "no"))
> m
```

Now do some indexing.

1. by position

   ```
   m[1, ] #first row
   m[1, , drop = FALSE]
   m[1,2] # cell "d"
   ```

2. by name

   ```
   m["type A",]
   m[, "no"]
   ```

3. by logical

   ```
   m[, 2] < 45 # logical vector
   m[m[, 2] < 45] # data
   @
   ```

You can achieve increasing levels of precision and complexity with indexing. In the following statement, (don't submit it, it's just for illustration) the extra comma after 3 tells R to return all the rows in x for which the 1st column is <3.

```
x[x[,1]<3,]
```

the extra comma after 3 tells R to return all the rows in x for which the 1st column is <3

The functions *lower.tri()* and *upper.tri()* use indexing to return the positions below or above a matrix.

```
> m2<-matrix(round(rnorm(81,50,5)),3,3)
> m2
> lower.tri(m2)
> upper.tri(m2)
> m2[lower.tri(m2)]
```

indexing arrays

Create the following array.

```
>         a<-array(sample(10:70,8, rep=T),c(2,2,2))
>         dimnames(a)<-list(exposure=c("e", "E"), disease=c("d", "D"),
+         confounder=c("c", "C"))
> a
```

Now, index to return the cell count for *unexposed, diseased, confounder negative* individuals...

1. by position

   ```
   a[1,2,1]
   ```

2. by name

   ```
   a["e","D","c"]
   ```

3. by logical

   ```
   a[a==33]
   ```

1.9.3 indexing lists

Indexing lists can sometimes be challenging. Recall the bracket notation for lists, where double brackets refer to the "bin" of like objects, and a following single bracket refers to the contents of that bin.

```
l<- list(1:5, matrix(1:4,2,2),
    c("John Snow", "William Farr"))
```

1. by position

   ```
   l[[1]]
   l[[2]][2,1]
   l[[3]][2]
   ```

2. logical

   ```
   char <- sapply(l, is.character)
   char
   epi.folk<-l[char]
   epi.folk
   ```

indexing the results of modeling

Indexing lists comes in handy when working with the results of statistical models, which frequently return results in the form of lists. Fortunately, most package authors return the results as named lists.

Work through the following conditional logistic regression of abortion and infertility to see an example of extracting list elements from the results of a model.

```
>          data(infert)
>          library(survival) # package with clogit()
>          mod1 <- clogit(case ~ spontaneous + induced +
+                        strata(stratum),  data = infert)
>          mod1 # default results (7x risk c spont AB, 4x induced)
>          str(mod1)
>          names(mod1) #structure, names
>          mod1$coeff # name to index result (list element)
>          summod1<-summary(mod1) #more detailed results
>          names(summod1) #detailed list components
```

1.9.4 indexing dataframes

Data frames can (generally) be indexed like matrices, with the added advantage of being able to use column (variable) names.

Run through this code to get a sense of how dataframes can be indexed.

data(infert)

1. position

   ```
   infert[1:4, 1:2]
   infert[1:4, 2] <- c(NA, 45, NA, 23)
   infert[1:4, 1:2]
   ```

2. name

   ```
   names(infert)
   infert[1:4, c("education", "age")]
   infert[1:4, c("age")] <- c(NA, 45, NA, 23)
   ```

```
infert[1:4, c("education", "age")]
```

3. logical

```
table(infert$parity)
# change values of 5 or 6 to missing
infert$parity[infert$parity==5 | infert$parity==6] <- NA
table(infert$parity)
table(infert$parity, exclude=NULL)
```

In the following perhaps more realistic example you will read in a set of anonymized hospital discharge data, and then index it in various ways.

```
> sparcs<-read.csv(file="http://www.columbia.edu/~cjd11/charles_dimaggio/
+ DIRE/resources/R/sparcsShort.csv", stringsAsFactors=F)
```

- index rows

```
brooklyn<-sparcs[sparcs$county=="59",]
nyc<- sparcs$county=="58"| sparcs$county=="59"|
sparcs$county=="60"| sparcs$county=="61"| sparcs$county=="62"
nyc.sparcs<-sparcs[nyc,]
```

- index columns

```
dxs<-sparcs[,"pdx"]
vars<-c("date", "pdx", "disp")
my.vars<-sparcs[,vars]
```

- index rows and columns

```
sparcs2<-sparcs[nyc,vars]
```

- variables to include

```
brooklyn.sparcs<-subset(sparcs, county=="59",
select=c(date, pdx,disp))
```

- range of variables

```
nyc.sparcs<-subset(sparcs, county=="59":"62",
select=c(county, pdx,disp))
```

- excluding rows

```
nyc.sparcs<-subset(sparcs, county=="59":"62",
select=-c(county, pdx,disp))
```

1.10 logistic regression: the Titanic

The file titanic.csv is a comma-separated value data file that contains information about the passengers on the Titanic.[35]

- Go to the course website and locate the file under the right-hand resource list. Right click on the file to get the path information and use this information to read the data into R. In this case, because it is a .csv file, you have two possible approaches. You can use read.table() with the appropriate option to read in a comma-separated file. Or you can use read.csv().
- Look at the structure and the first 5 observations (rows) of the R data object you created. Look at the names of the variables [36]

[35]For more information on this data set, install the R package "PASWR" and search for help on "titanic3"

[36]If you don't know how to do that ?names is a good place to start looking for information

- The variable *survived* has two levels, with 1 meaning the person lived. The variable *sex* also has two levels. Create an object cross tabulating gender by survival status, and use epitabs to calculate an odds ratio for the risk of death by gender. Look at the help file for *epitab()*. Are the columns of the table object your created in the correct order? Are the rows in the correct order to demonstrate the risk associated with being a male? Correct the table, and re-run the analysis.
- We will now use logistic regression to do the same analysis. Use glm() to create an object from a model with survival as the dependent or outcome variable, and sex as the independent or explanatory variable.[37]
- Print the model object. Most often a summary() of the model object will be more informative. To view the confidence intervals for the model, use the *confint()* function. To exponentiate the results use the *exp()* function. You can exponentiate the model coefficients by appending $coefficients to the name of the model object and exponentiate an object to which you assign the confidence intervals.
- Are you modeling the event in which you are interested? You can address this in one of two ways. First, and most intuitively, you can recode the outcome variable. The following snippet of syntax will accomplish that.

  ```
  surv2<-(titanic$survived-1)*-1
  ```

  Substitute the recoded variable into your model statement and review the results, including exponentiated coefficients.
- A bit more unintuitively (at least to me) you can achieve the same result by recoding the exposure variable. To re-order the exposure variable to calculate the risk of male gender, you can use the relevel() function. Substitute the following for the sex variable in your glm model: [38]

  ```
  relevel(titanic$sex, ref="male")
  ```
- One of the strengths of a logistic regression approach compared to stratified analysis is that it will more easily accommodate controlling for multiple potentially confounding variables. All you have to do is add them to your model statement preceded by a plus sign. Create a new model controlling for age and controlling for passenger class [39]

1.11 survival analysis: papal longevity

Readers of Papal history may be struck by (among many things) how frequently Popes died of "fever". The word malaria, in fact, arose in reference to the "bad air" ("mal aria") surrounding Rome. Is it possible Popes who were born in or near Rome had some immunological advantage? We need wonder no more.

Go to the course R page here. On the sidebar locate and download the R data file called "popes". Use the R package "survival", as described here to answer the following questions:

1. Did Popes who were born in Rome survive longer from birth to death?
2. Did Popes who were born in Rome survive longer from election to death?

For each analysis, provide Kaplan-Meir curves and Log Rank tests.

[37]Hint: glm(outcome explanatory, family=binomial(link="logit"))

[38]This can be a bit confusing. As noted in the *relevel()* help page, "the (factor) level specified by ref is first and the others are moved down." In this case, this effectively makes males the exposure level.

[39]Use relevel(pclass, ref="3rd") for the class variable to demonstrate the increased risk for third class

Chapter 2

Part II: Bayesian Analysis in R

This series of exercises addresses simulation analyses like bootstrapping, and introduces Bayesian analysis. Again, try to actively type in and play with the commands rather than just cutting and pasting. Most of the material is tutorial. Questions are indicated by boldface caps.

2.1 loops

2.1.1 for loops

The basic form of a while loop is:

```
for (var in seq) expression
```

where var is the name of your iteration variable, and seq is a sequence of values.

The expression itself can be a single R command, or several commands or lines of code enclosed in curly brackets:

```
for (var in seq) {
    expression
    expression
    expression
}
```

Try the following example that prints the the square root of the integers one to ten:

```
for(x in 1:10) print(sqrt(x))
```

2.1.2 while loops

A while loop looks like this:

```
while (condition) expression
```

Where “condition” is some logical statement that evaluates to TRUE/FALSE.

Again, you can have more than one “expression” wrapped in curly brackets.

Let’s use a while loop to print out Fibonacci numbers. [1]

[1] A famous (to mathematicians and to folks who’ve read the DaVinci Code) series of numbers in which each number (Fibonacci number) is the sum of the two preceding numbers. The simplest is the series 1, 1, 2, 3, 5, 8, etc...

```
a <- 0
b <- 1
print(a)
while (b < 50) {
    print(b)
    temp <- a + b
    a <- b
    b <- temp
}
```

As with most things that involve writing computer code, there is usually more than one way to do things. This following code accomplishes the same thing as that above, but is more elegant in that it gradually builds a vector that it prints at the end:[2] it does it:

```
x <- c(0,1)
while (length(x) < 10) {
    position <- length(x)
    new <- x[position] + x[position-1]
    x <- c(x,new)
}
print(x)
```

2.2 writing functions

You write functions in R with (logically enough) the function() command. We can write a function to calculate Fibonacci numbers based on the code above.

```
Fibonacci <- function(n) {
    x <- c(0,1)
    while (length(x) < n) {
        position <- length(x)
        new <- x[position] + x[position-1]
        x <- c(x,new)
    }
    return(x)
}
```

Now let's test the function:

```
Fibonacci(10)

Fibonacci(3)

Fibonacci(2)

Fibonacci(1)
```

Note that for Fibonacci(1), the function returns the first two Fibonacci numbers, rather than only the first. We can fix that with an if statement.

```
Fibonacci <- function(n) {
    if (n==1) return(0)
    x <- c(0,1)
    while (length(x) < n) {
        position <- length(x)
        new <- x[position] + x[position-1]
        x <- c(x,new)
```

[2] You can try the following to help you understand how the code appends values to an existing vector: x = c(1,2,3,4); c(x,5);c(x,6)

```
    }
    return(x)
}
```

This is an example of a simple conditional statement:

```
if (condition) expression
```

You could write a more involved if statement by combining it with else:

```
if (condition) expression else expression
```

And like in the case of for and while loops, you can have multiple expressions within curly brackets. We will take advantage of this to fix our Fibonacci function:

```
Fibonacci <- function(n) {
    if (n==1) {
        x <- 0
    } else {
        x <- c(0,1)
        while (length(x) < n) {
            position <- length(x)
            new <- x[position] + x[position-1]
            x <- c(x,new)
        }
    }
    return(x)
}
```

2.2.1 an odds ratio function

The ability to write your own functions is one of the appeals of working with R. Here is a function that calculates odds ratios and their confidence intervals.[3] See if you understand all the steps, then test it out on some data, say a vector of the 4 numbers 64, 459, 83 and 1409. The result should be OR = 2.3 (95% CI 1.7, 3.3)

```
or.ci<-function(x){
data<-matrix(x,2,2, byrow=T)
rownames(data)<-c("e", "E")
colnames(data)<-c("d", "D")
a<-data["e","d"]
b<-data["e","D"]
c<-data["E","d"]
d<-data["E","D"]
or<-(a*d)/(b*c)
l.or<-log(or)
l.se<-sqrt(1/a+1/b+1/c+1/d)
l.low<-l.or-(1.96*l.se)
l.up<-l.or+(1.96*l.se)
low.ci<-exp(l.low)
up.ci<-exp(l.up)
list(or, low.ci, up.ci)
}
```

By the way, if you come across a function in R that you'd like to learn more about or adapt to you own use, just type it's name and hit enter. For example, load the "epitools" packages, then type the name "epitab" (without the quotes). Compare this function to the simple one I wrote.

[3]Imitation is the sincerest form of flattery, so fee free to take and use as the basis for your own functions

2.3 Sampling and Simulations

Let's start with some binomial simulations. [4] The following code is a single draw, from a sample of 10 with a probability of 0.4. Run it a few times.

```
> set.seed(100)
> rbinom(n=1,size=10,prob=0.4)
```

Think of this code as defining a binomial "experiment". Change the rules of the experiment so that the probability is 0.2, and run that a few times.

```
> set.seed(111)
> rbinom(n=1,size=10,prob=0.2)
```

Each run is a "simulation". Do 300 simulations of the same "experiment". (You just need to change one number in the code.)

```
> set.seed(121)
> rbinom(n=300,size=10,prob=0.2)
```

Now, run 3,000 simulations. Assign them to a variable called "sims". What is the mean and median of this series of simulations? Plot the frequency table of the object "sims".

```
> set.seed(131)
> sims<-rbinom(n=3000,size=10,prob=0.2)
> summary(sims)
> plot(table(sims))
```

Re-run the simulations setting the probability to 0.8.

```
> set.seed(141)
> sims<-rbinom(n=3000,size=10,prob=0.8)
> summary(sims)
> plot(table(sims))
```

In R, you can sample from most any probability distribution with the *rxxxx()* group of functions. For a simple random sample, you can use *sample()*. Note the specification for replacement.

2.3.1 Simulating Risk Ratios

We can use simulations to approximate results for many problems when there is no direct or closed solution, which is often the case in Bayesian analysis, when the underlying distribution is unknown, or for small sample sizes. It may also be helpful in power calculations. To put the process in a context that is more familiar to epidemiologists, let's look at risk ratios. The observed rates of disease in exposed and unexposed populations can serve as the probabilities in two sets of simulated experiments using *rbinom()*. Divide those results by the number of simulations and use the mean and 0.025 tails for the point estimate and confidence limits. The process is sometimes referred to as "bootstrapping". In outline it looks like this:

1. simulate 5000 replicate bernoulli trials in sample size n_1 = exposed
2. divide those results by n_1 to get 5000 simulated risk estimates for the exposed group
3. repeat that process for the unexposed group n_2
4. divide 5000 simulated risks in exposed by 5000 simulated risks in unexposed to get 5000 simulate relative risks
5. calcualate mean and 0.25 tails from that population

[4]If you're interested in a more thorough introduction to using probability distributions in R, look here

Let's walk through the process with an example. In a ground breaking 1987 study, Hennekens and colleagues reported on the protective benefits of aspirin. There were 104 myocardial infarctions (fatal and non-fatal) among 11,037 people in the treatment group, and 189 MI's among 11,034 people in the placebo group. We calculate the risk ratio and confidence interval using the usual, log-approximated approach found in the "epitools" package.

```
> library(epitools)
> asa.tab<- matrix(c(104,11037,189,11034),2,2)
> epitab(asa.tab, method="riskratio")
```

We'll compare these results to those from a simulation-based approach. Begin by "setting a seed" for the random draws, so you can replicate your results later on. Then use *rbinom()* to repeat 5,000 times an experiment where we count the number of outcomes (MI's) in two populations, with the probability of the outcome in a population defined by the results of the Hennekens study.

```
> set.seed(151)
> tx <- rbinom(5000, 11037, 104/11037)
> plac <- rbinom(5000, 11034, 189/11034)
```

In this next step, for each of the 5,000 experimental replicates, we divide the number of outcomes by the number of people in each population to get a 5,000 risk estimates for each group. Then, again for each of the 5,000 experiments or replicates, we divide the risk in the treatment group by the risk in the placebo group to get 5,000 estimates of the relative risk.

```
> r.tx<-tx/11037
> r.plac<-plac/11034
> rr.sim <- r.tx/r.plac
```

Now, we get the statistics in which we are interested.

```
> mean(rr.sim)
> quantile(rr.sim, c(0.025, 0.975))
```

The sample standard deviation is an estimate of the standard error.

```
> sd(rr.sim)
```

2.3.2 Bootstrap Function

TRY WRITING A FUNCTION TO CALCULATE BOOTSTRAP ESTIMATES OF RELATIVE RISKS. TEST THE FUNCTION USING THE ASPIRIN EXAMPLE ABOVE.

You may find the following function, written by Tomas Aragon useful, though you can probably write a simpler version for our purposes.

```
> rr.boot <- function(x, conf.level = 0.95, replicates = 5000){ ##prepare input
+ x1 <- x[1,1]; n1 <- sum(x[1,])
+ x0 <- x[2,1]; n0 <- sum(x[2,])
+ ##calculate
+ p1 <- x1/n1 ##risk among exposed
+ p0 <- x0/n0 ##risk among unexposed
+ RR <- p1/p0
+ r1 <- rbinom(replicates, n1, p1)/n1
+ x0.boot <- x0.boot2 <- rbinom(replicates, n0, p0)
+ x0.boot[x0.boot2==0] <- x0.boot2 + 1
+ n0.denom <- rep(n0, replicates)
+ n0.denom[x0.boot2==0] <- n0.denom + 1
+ r0 <- x0.boot/n0.denom
+ rrboot <- r1/r0
+ rrbar <- mean(rrboot)
+ alpha <- 1 - conf.level
+ ci <- quantile(rrboot, c(alpha/2, 1-alpha/2))
+ ##collect
+ list(x = x, risks = c(p1 = p1, p0 = p0), risk.ratio = RR,
+ rrboot.mean = rrbar, conf.int = unname(ci), conf.level =
+ conf.level, replicates = replicates) }
```

2.4 Bayesian Conjugate and MCMC Analysis of Airline Fatalities

The file airline contains 26 years worth of airline fatality data. We're interested in the average or mean number of yearly fatalities. We'll analyze the data with a Poisson model, comparing the exact conjugate results to the simulation-based MCMC results using the program JAGS. If you haven't already done so, take a moment to download and install JAGS. Then within R install R2jags using *install.package("R2jags")*.

Read in the airline fatality data.

```
> airline<-read.csv("http://www.columbia.edu/~cjd11/charles_dimaggio/DIRE/resources/Bayes
/data/airline.csv",header=T, stringsAsFactors=F)
```

HOW MANY OBSERVATIONS ARE IN THE DATA SET? WHAT IS THE TOTAL NUMBER OF FATALITIES? WHAT IS THE MEAN NUMBER OF FATALITIES?.

Obtain the exact results with a conjugate analysis. [5] The likelihood model for the data is $y_i|\mu \sim Poisson(\mu)$. We choose as an "uninformative" prior distribution $\mu \sim \Gamma(0.01, 0.01)$, which is nearly uniform on $(0, +\inf)$. The conjugate posterior for this model is then $\mu \sim \Gamma(0.01 + \Sigma y_i, 0.01 + n)$, where Σy_i is the total number of fatalities, and n is the number of observations. We're interested in the mean of this distribution.

USE THE *rgamma()* FUNCTION TO SIMULATE 10,000 DRAWS FROM A DISTRIBUTION BASED ON $\mu \sim \Gamma(0.01 + \Sigma y_i, 0.01 + n)$. WHAT IS THE MEAN OF THIS DISTRIBUTION?

PLOT THE DENSITY OF THE GAMMA DISTRIBUTION YOU JUST SIMULATED. USE THE *abline()* FUNCTION TO ADD A VERTICAL LINE FOR THE MEAN OF DISTRIBUTION.

Let's compare this result to that obtained through MCMC simulation. We begin by creating a list object to hold the data, and another list object to hold the initial values for the simulation (notice that we need three values because we will be running three chains). We then write a model statement to file using the *cat()* function. You can save the text file that holds the model anywhere you like. If you don't specify a path, it will save to your default home library or folder.

```
> set.seed(123)
> a.dat<-list(fatal=c(airline$fatal,NA), I=27)
> a.inits<-list(
+         list(mu=20),
+         list(mu=23),
+         list(mu=26)
+         )
> cat(
+ "model
+ {
+ for(i in 1:I)
+ {
+ fatal[i]~dpois(mu)
+ }
+ mu~dgamma(0.1,0.1)
```

[5]Look here for an introductory review of Bayesian and conjugate analysis.

```
+ }",
+ file="m1.txt")
```

We run the simulation in JAGS via the R2jags package function *jags()*. Type *?jags* to get more information on options and use. Note that we are only monitoring the parameter "mu" (*parameters.to.save="mu"*). The summary, print, plot and traceplot functions operate directly on the bugs object R2jags produces. To access the tools in in the coda package, we have to convert it to an mcmc object.

```
> library(R2jags)
> air.jag<-jags(data=a.dat,
+          inits=a.inits,
+          parameters.to.save="mu",
+          model.file="m1.txt")
> print(air.jag)
>

> plot(air.jag)

> traceplot(air.jag)

> air.jag.mcmc<-as.mcmc(air.jag)
> plot(air.jag.mcmc)
```

SUMMARIZE THE RESULTS OF THE MCMC RUN. WHAT ARE YOUR CONCLUSIONS ABOUT THE MEAN NUMBER OF YEARLY AIRLINE FATALITIES BASED ON EACH OF THE ANALYTIC APPROACHES?

2.5 Bayesian Approach to Drug Trials

Epidemiologists working in drug trials spend a lot of time thinking about binomial outcomes and proportions, where each person in a trial is essentially a Bernoulli trial, $y \in (0,1)$. The data, then, is binomially distributed with the likelihood for x events out of n trials:

$$p(y|\theta_j) = {\theta_j}^y(1-\theta_j)^{1-y},$$

Where θ is some health outcome that can take on a number of possible discrete values, i.e. $(\theta_1, \theta_2, \ldots, \theta_J)$.

To this point, we are working in the usual, frequentist, statistical realm. But, say, that we have some pre-existing or prior belief about the probability of each possible outcome, [6] $p(\theta_j)$, where (under the rule of total probability) $\sum_j p(\theta_j) = 1$.

A straightforward application of Bayes' theorem is to multiply the distribution of discrete prior probabilities by the likelihood of the observed data to arrive at posterior probabilities:

$$p(\theta_j|x) \propto \theta_j^x(1-\theta_j)^{n-x} \times p(\theta_j).$$

Note that we'll also need to divide by the sum to normalize the probabilities to add up to 1.

Let's run through an example. Say, a drug has an (unknown true) response rate θ, and that θ can only take one of the discrete values $\theta_1 = 0.2$, $\theta_2 = 0.4$, $\theta_3 = 0.6$ or $\theta_4 = 0.8$. Further assume that each value θ_j is equally likely, i.e. $p(\theta_j) = 0.25$ for each $j = 1, 2, 3, 4$.

IF WE ADMINISTER THE DRUG TO A SINGLE INDIVIDUAL, AND SHE RESPONDS, ($y = 1$), HOW WOULD WE UPDATE OUR BELIEFS ABOUT THE EFFECTIVENESS OF THE DRUG? CALCULATE THE POSTERIORS FOR THE FOLLOWING TABLE.

[6]This may be based on our clinical experience, on the literature, on prior experiments, etc...

Theta	Prior	Likelihood	Posterior
1	0.2	0.25	
2	0.4	0.25	
3	0.6	0.25	
4	0.8	0.25	

Begin by assigning values to the variables you will need for your calculations, i.e. *theta*, *y*, *n* and *prior*

```
> theta<-c(.2,.4,.6,.8)
> y<-1
> n<-1
> prior<-.25
```

Write a statement to calculate the likelihoods.

```
> likelihood<-(theta^y)*((1-theta)^(n-y))
```

Write another statement to calculate the posteriors. This proceeds in two steps. First multiply the likelihood by the prior. then divide each of these values by the sum of all the values to normalize them.

```
> likelihood.prior<-likelihood*prior
> posterior<-likelihood.prior/sum(likelihood.prior)
```

Finally, summarize the results by calculating the mean and plotting them. (You would normally also want the standard deviation and 95% credible intervals, but that doesn't make sense in this simple example.)

```
> mean(posterior)
> plot(posterior, type="h")
```

WHAT CAN YOU REASONABLY CONCLUDE FROM RUNNING A SINGLE EXPERIMENT? CAN YOU THINK OF AN ALTERNATE APPROACH TO THE CALCULATIONS THAT INVOLVES SAMPLING DIRECTLY FROM A PROBABILITY DISTRIBUTION USING *dbinom()*?

Now, let's say we administer the drug to 20 people, of whom 15 have a positive response.

RERUN YOUR CALCULATIONS TO ANALYZE THE NEW DATA. WHAT CHANGES DID YOU HAVE TO MAKE? WHAT ARE YOUR NEW CONCLUSIONS?

Next, imagine you are interested in the extreme scenarios that the drug cures everyone or no one at all. Update your calculations to include these two possibilities. Note, the prior distribution now consists of 6 values: $\theta_1 = 0$, $\theta_2 = 0.2$, $\theta_3 = 0.4$,$\theta_4 = 0.6$, $\theta_5 = 0.8$ or $\theta_6 = 1.0$.

RERUN YOUR CALCULATIONS TO INCLUDE THE NEW POSSIBLE OUTCOMES. WHAT DO YOU CONCLUDE?

It may be unreasonable to assume that θ can only take one discrete value. We may, rather, believe that response rates are most likely to be anywhere between 0.2 and 0.6. We can reflect that belief with a continuous prior distribution. Since we are working with total probability, the distribution should be on the interval $(0,1)$. We'll reflect our beliefs by choosing one with a mean 0.4 and 95% of the probability in the interval (0.2,0.6). We can construct a Beta(a,b) distribution to reflect this. [7]

The mean m and standard deviation s for a Beta distributed variable are:

$$m = \quad \frac{a}{a+b} s = \quad \sqrt{\frac{m(1-m)}{a+b+1}}$$

The expression in equation ([betavar]) can be rearranged to give $a+b = \big(m(1-m)/s^2\big) - 1$.

USE USE THE TARGET VALUES $m = 0.4$ and $s = 0.1$ TO OBTAIN A VALUE FOR $a+b$, AND THE FORMULA FOR m TO GET SEPARATE VALUES FOR a AND b.

[7]For a description of the Beta distribution and how it is used in Bayesian analysis, see here

PLOT THE PRIOR DISTRIBUTION FOR THE PROBABILITY OF SUCCESS, *p*. Hint: Create a sequence of 100 probability values from 0 to 1 and assign it to a variable. Use this variable as the quantile argument to the *dbeta()* probability density function, and the results for a and b above as the parameters. Then plot the sequence of probability values against the results of the density values.

PLOT THE LIKELIHOOD OF 15 RESPONSES OUT OF 20 TRIALS. Hint: The approach is the same as that for the prior above, only this time you will use *dbinom()*.

We will use the convenience of conjugacy to generate the posterior probability distribution. The posterior distribution for a beta prior and a binomial likelihood is a beta($a+x$, $b+(n-x)$).

PLOT A GRAPH OF THE POSTERIOR PROBABILITY DISTRIBUTION Hint: You will, again, use the *dbeta()* function, only this time substitute the updated values for a and b.

PLOT THE THREE GRAPHS TOGETHER. Hint: Use par(mfcol=c(3,1)) to set up the plot, then restate the three plots from above.

WHAT ARE YOUR CONCLUSIONS BASED ON THESE THREE PLOTS?

2.6 Bayesian Linear Regression

We'll use JAGS to demonstrate a Bayesian approach to a simple linear regression.

We start with a simple linear model using *lm()*. Create the following two variables.

```
> x <- c(1,2,3,4,5,6)
> y <- c(1,3,3,3,5,7)
```

PLOT THE VARIABLES AGAINST EACH OTHER. FIT A SIMPLE LINEAR REGRESSION MODEL OF y ON x. WHAT ARE THE COEFFICIENT ESTIMATES AND THEIR CONFIDENCE INTERVALS. SUMMARIZE YOUR RESULTS AND CONCLUSIONS Hint: Use *summary()* and *confint()* on the results of the regression.

Now we'll fit the same model using JAGS.

Use *list()* to create a variable called *dat* to hold the two variables and the number of observations as a list object.

```
> dat<-list(x=x, y=y, N=length(x) )
```

Create another list object called *inits* to hold the initial values for the parameters we will be simulating. We will be simulating three parameters, the two coefficients, *b.0*, *b.1*, and the precision variable, *tau*, we will be using to define the normally distributed variable *y*. [8] For each variable, assign three initial values. Based on the simple linear model above, we'll use *-2,0,2* for *b.0*, *-.4,.5,1.5* for *b.0* and *-.01, 0, .01* [9]

```
> inits <- list("b.0"=c(-2,0,2),"b.1"=c(-.4,.5,1.5), tau=c(-.01, 0, .01))
```

Then we write a model document out to file and run the JAGS simulation using the *jags()* function of the R2jags package. We then explore the object and get our results using summary, print, plot and traceplot. We then convert the bugs object to an mcmc object and get additional plots.

Write a model statement using *cat()* Note, that we are going to monitor the standard deviation (*sigma*) as a function of the precision variable *tau*.

```
> cat(
+         "model
+         {
+             for (i in 1:N) {
```

[8]In BUGS and JAGS, normally distributed variables are defined in terms of their precision, which is the inverse of the standard deviation

[9]Setting initial values can be a bit fiddly, but is important and can be the difference between a model that runs and one that doesn't. There are a number of suggestions and commentaries on the topic which you can access with Google. For now, suffice it to say, that initial values should be reasonable, and should be dispersed. You can do worse than what we've done here, which is basing them on an initial simpler model.

```
+                    y[i]    ~ dnorm(mu[i], tau)
+                    mu[i] <- b.0 + b.1 * (x[i])
+              }
+                  b.0   ~ dnorm(0.0, 1.0E-4)
+                  b.1 ~ dnorm(0.0, 1.0E-4)
+                  sigma <- 1.0/sqrt(tau)
+                  tau   ~ dgamma(1.0E-3, 1.0E-3)
+          }", file="mod2.txt"
+          )
```

Define the parameters you want to monitor, i.e. that you want results for.

```
> params<-c("b.0", "b.1", "sigma")
```

Write the *jags()* statement to run the model. Assign the process to a variable that you can use to extract results.

```
> library(R2jags)
> set.seed(456)
> res<-jags(data=dat,
+          inits=inits,
+          parameters.to.save=params,
+          model.file="mod2.txt")
```

Use *traceplot()* for a quick assessment of convergence.

```
> traceplot(res)
```

Use *print()* and *plot()* to review the results.

```
> print(res)
> plot(res)
```

You can combine the traceplot and plot functions, by converting the *jags* object to an *mcmc* object and run plot on it.

```
> plot(as.mcmc(res))
```

COMPARE THE SIMPLE LINEAR RUN TO THE BAYESIAN RUN. HOW ARE THEY SIMILAR? HOW ARE THEY DIFFERENT?

The *cars* data set contains 50 observations of vehicle speeds ("speed") and stopping distances ("dist").

REPEAT THE SIMPLE LINEAR REGRESSION ANALYSES ON THE "cars" DATASET.

```
> dat<-list(x=x, y=y, N=length(x) )
> inits <- list("b.0"=c(-30,-17,30),"b.1"=c(-2,.4,6), tau=c(-.01, 0, .01))
> cat(
+          "model
+          {
+              for (i in 1:N) {
+                    y[i]    ~ dnorm(mu[i], tau)
+                    mu[i] <- b.0 + b.1 * (x[i])
+              }
+                  b.0   ~ dnorm(0.0, 1.0E-4)
+                  b.1 ~ dnorm(0.0, 1.0E-4)
+                  sigma <- 1.0/sqrt(tau)
+                  tau   ~ dgamma(1.0E-3, 1.0E-3)
+          }", file="mod3.txt"
+          )
> params<-c("b.0", "b.1", "sigma")
> set.seed(789)
> res2<-jags(data=dat,
+           inits=inits,
+           parameters.to.save=params,
+           model.file="mod3.txt")
> print(res2)
> plot(as.mcmc(res2))
>
> # again, essentially same results as frequentist
```

2.7 Bayesian Analysis of Repeated Fetal Growth Measurements

The dataset fetal.csv contains measurements of head circumference, gestational age, and a transformation of gestational age for children enrolled in the Western Australia (Raine) Cohort Study. This is a repeated measures data set ranging from 1 to 7 measurements for each child. Most children had 4 or 5 measurements. We're interested in fetal growth as a function of time. A quadratic transformation ($tga = ga$

READ IN AND EXPLORE THE DATA SET.

TABULATE THE NUMBER OF CHILDREN AND THE NUMBER OF MEASUREMENTS FOR EACH CHILD. Hint: Use the *id* variable to calculate a table of a table. Use *addmargins()* to sum up.

PLOT THE RELATIONSHIP BETWEEN GESTATIONAL AGE AND HEAD CIRCUMFERENCE.

PLOT THE RELATIONSHIP BETWEEN TRANSFORMED GESTATIONAL AGE AND THE SQUARE ROOT OF HEAD CIRCUMFERENCE.

Fit a random effects linear model for each measurement for each fetus f at time$t$::

$$y_{ft} = \beta_0 + \beta_1 + u_{0f} + e_{ft} u_{0f}, u_{1f} \sim \mathcal{N}(0, \Sigma), \quad e_{ft} \sim \mathcal{N}(0, \sigma)$$

Specify a model with a random effect term (u_{0f}), where each child's measurements across time are parallel to each other, varying only by intercept. [10] Note that the β_0 is not really interpretable (no child has a zero head circumference), though we could set it to some baseline level, e.g. at age 18 weeks with (1|(tga-18)).

The effect of time in this model (β_1) is a *fixed effect*. The *random effect* term is for each child (id), which is allowed to vary relative to each other, (1|id), though as noted above it only varies by level, and is otherwise parallel to each other reflecting the fixed effect of time on head circumference. We fit the model with *lmer()* function from the "lme4" package.

```
> library( lme4 )
> m0 <- lmer(sqrt(hc) ~ tga + (1|id), data=fetal)
> summary(m0)
```

The intercept is -.08 but (as noted above) it is not really interpretable in this setting. The effect of time is "tga estimate = 0.086" or about a 1 cm increase in head circumference per week. The random effects result of interest is "id (Intercept) 0.059714 0.24437" which indicates that each child's series of results (regression line) varies from each other by about 0.24 cm. The residual result "Residual 0.062665 0.25033 " is interpreted as usual, i.e. the difference between an observation and that predicted from the regression line.

We can extract individual results. We will use this to set initial values in the Bayesian model.

```
> fixef(m0)
> VarCorr(m0)
> attr( VarCorr(m0)$id,"stddev")
> attr(VarCorr(m0),"sc")
```

[10]This is not very realistic...

The variance-covariance result is not very informative for this very simple model, since there is only one random effect term and (as expected) it correlates with itself as 1.

Now, we fit a Bayesian model in JAGS. In the model specification, the random effect for each child is represented as sigma.u, the error as sigma.e:

```
> cat("
+ model
+ {
+  # model for each observation
+  for( j in 1:N )
+ {
+  mu[j] <- beta[1] + beta[2] * ( tga[j]-18 )  + u[id[j]]
+  hc[j] ~ dnorm( mu[j], tau.e )
+ }
+
+  # Random effects for each person
+  for( i in 1:I )
+ {
+  u[i] ~ dnorm(0,tau.u)
+ }
+
+  # Priors:
+    # Fixed intercept and slope
+      beta[1] ~ dnorm(0.0,1.0E-5)
+      beta[2] ~ dnorm(0.0,1.0E-5)
+
+    # Residual variance
+      tau.e <- pow(sigma.e,-2)
+      sigma.e  ~ dunif(0,100)
+
+    # Between-person variation
+      tau.u <- pow(sigma.u,-2)
+      sigma.u  ~ dunif(0,100)
+ }",
+ file="fetal1.txt" )
```

In defining the data elements note that we need both *N* and *I*

```
> fetal.dat <- list( id = as.integer( factor(fetal$id) ),
+               hc = fetal$hc,
+               tga = fetal$tga,
+               N = nrow(fetal),
+               I = length( unique(fetal$id) ) )
```

Set the initial values using the results from the *lmer* run.

```
> ( sigma.e <- attr(VarCorr(m0),"sc") )
> ( sigma.u <- attr(VarCorr(m0)$id,"stddev") )
> ( beta    <- fixef( m0 ) )
> fetal.inits <- list( list( sigma.e = sigma.e/3,
+               sigma.u = sigma.u/3,
+            beta = beta   /3 ),
+   list( sigma.e = sigma.e*3,
+         sigma.u = sigma.u*3,
+            beta = beta   *3 ),
+   list( sigma.e = sigma.e/3,
+          sigma.u = sigma.u*3,
+             beta = beta   /3 ),
+   list( sigma.e = sigma.e*3,
+          sigma.u = sigma.u/3,
+             beta = beta   *3 ) )
```

Define the parameters to monitor.

```
>  fetal.params<-c("beta", "sigma.e", "sigma.u")
```

Run the model. [11]

```
> set.seed(965)
>  jag.fetal<-jags(data=fetal.dat,
+          inits=fetal.inits,
+          parameters.to.save=fetal.params, n.chains=4, n.iter=500, n.thin=20,
+          model.file="fetal1.txt")
```

Look at the results.

```
> print(jag.fetal)
> plot(as.mcmc(jag.fetal))
```

Note that these results differ results differ from the lmer run because we are fitting a different model, i.e. the JAGS model has untransformed head circumference as the dependent variable while the lmer model has the square root of head circumference as the dependent variable. And the bugs model has the transformed gestational age adjusted to the 18th week (tga[j]-18), while the lmer model uses an unadjusted tga.

If you are interested in working through a more complex (and more realistic) model of these data that includes both a random slope term and a random intercept term for fetal growth, see Dr. Bendix Carstensen's Sweave file.

2.8 Hierarchical Bayesian Model of Pelvic Inflammatory Disease

The file wart.csv contains data on visits to 23 physicians at the Melbourne,Sexual Health Centre, and how many visits resulted in a diagnosis of pelvic inflammatory disease (PID) and/or vaginal warts.

consults	PID	warts height80	1
1 816	41	46 726	12
37 2891	38	137 79	4
4 1876	34	73 469	8
27 1124	13	76 210	10
6 539	8	28 1950	22
101 1697	24	86 811	13
56 908	52	48 944	19
65 832	10	33 1482	10
62 456	0	20 420	0
8 1258	3	58 1101	1
22 109	0	3 1006	2
62 height			

We are interested in whether there is clinically important variation among physicians in how frequently physicians make these diagnoses.

READ IN THE DATA. SUMMARIZE AND PLOT THE PROPORTION OF CONSULTS THAT WERE PID DIAGNOSES, AND THE PROPORTION OF CONSULTS THAT WERE GENITAL WART DIAGNOSES

We are interested in how meaningful these differences are. The data themselves may be considered binomial:

[11] In this interests of time when compiling this latex document, I am running only 500 iterations...

$$y_i|\theta_i \sim \text{Bin}(n_i, \theta_i)$$

where,

- i indexes physicians from 1 to 23 - n is the number of consults with n_i consults per physician - y is the number of diagnoses, y_i per physician - with a physician-level probability of diagnosis θ_i, and

We can model the data as a logistic regression with α_i, the physician-level log odds of diagnosis:

$$\text{logit}(\theta_i) = \log(\theta_i/(1-\theta_i)) = \alpha_i$$

The *MLE estimates* are:

$$\hat{\theta}_i = y_i/n_i\hat{\alpha}_i$$

$$\log(y_i/(n_i - y_i))$$

We may assume that α_i is the same for *all* the physicians, i.e. this is the realization of a single process and identifying the physician making the diagnosis is irrelevant. In this case we can analyze the data as a single group. This may seem a bit extreme. And it is. Perhaps equally extreme, is to assume that α_i is different for each physician and unrelated to any other α_i. The concept of *exchangeability* is a compromise between these two extremes. α_i varies across physicians in a predictable fashion. Commonly, we assume the estimate for α for each physician varies across physicians in a normally distributed fashion. This is a random effect model (as opposed to a fixed effect model).

An extreme assumption is that $\alpha_i = \alpha$ for all physicians. Alternatively, we can assume a different, un-related α_i for each physician. A hierarchical model, replacing independence with exchangeability, provides a compromise. There is an estimate for α_i for each physician which varies across physicians in a normally distributed fashion,

$$\alpha_i \sim \text{N}(\mu, \tau^2).$$

The data or observations (y_i) are conditionally independent binomial variables given α_i This logistic-normal model, where the α_i's are given a distribution, and so are *random* rather than *fixed* effects, is an example of a *generalised linear mixed model* (GLMM).

The model is available in programs like Stata and SAS, and in R with packages like *lmer4*. We can implement in as a Bayesian model in JAGS by giving the data (y_i) a distribution and the parameters governing the data (hyperparameters) distributions of their own.

We start with a *fixed-effect logit model*, where we assume a different genital warts diagnosis proportion for each physician.

The model code is

```
> cat( "model
+     {
+     a[1] ~ dnorm(0,10000)
+     for(i in 2:23)
+     {
+     a[i] ~ dnorm(0,0.0001)
+     }
+     for(i in 1:23)
+     {
+     mean[i] <- mu + a[i]
+     logit(pr[i]) <-  mean[i]
+     warts[i] ~ dbin(pr[i],consults[i])
+     }
+
+     mu ~ dnorm(0,0.001);
+
+     }",file="wart1.txt" )
```

Specify a data list object, an initial values list object, and name the parameter we want to monitor.

```
> wart1<-pid[,c("consults", "warts")]
> wart1.dat<-as.list(wart1)
> wart1.ini<-list(mu = -3, a = c(0,rep(0,22)))
> wart1.par<-c("mean")
```

Run the model, sample from the posterior, and summarize it.

Model compilation and burn-in

```
> library(R2jags)
> wart1.mod <- jags.model( file = "wart1.txt",
+                      data = wart1.dat,
+                      n.chains = 3,
+                      inits = wart1.ini,
+                      n.adapt = 1500 )
> wart1.res <- coda.samples( model = wart1.mod,
+                        var = wart1.par,
+                        n.iter = 1500,
+                        thin = 1 )
> summary(wart1.res)
```

The following code (from Dr. Carstenen) extracts the point estimates and the lower and upper 95% credible interval values, calculates the exponentiated proportions, then graphs them as a caterpillar or forest plot.

```
> fixeffs <- cbind(as.vector(summary(wart1.res)$statistics[,1]),
as.vector(summary(wart1.res)$quantiles[,1]),
+              as.vector(summary(wart1.res)$quantiles[,5]))
> expit <- function(x) {exp(x)/(1 + exp(x))}
> plot(x = expit(fixeffs[,1]),y = 5*(1:23)-2, xlim = c(0,0.125), pch = 16,
 ylab = "Doctor", xlab = "Proportion diagnosed with warts (with 95% CI)",
yaxt = "n")
> segments(x0 = expit(fixeffs[,2]), y0 = 5*(1:23)-2,
x1 = expit(fixeffs[,3]), y1 = 5*(1:23)-2, lty = 1)
```

We we write a *random-effect logit model*, where we allow the physician-specific parameters to come from a normally-distributed population of all possible parameters.

```
> cat( "model
+     {
+     for(i in 1:23)
+     {
+     a[i] ~ dnorm(0,taua);
+     mean[i] <- mu + a[i];
+     logit(pr[i]) <-  mean[i];
+     warts[i] ~ dbin(pr[i],consults[i]);
+     }
+
+     taua ~ dpar(1,0.01);
+
+     sigma2a <- 1/taua;
+     sigmaa <- sqrt(sigma2a);
+
+     mu ~ dnorm(0,0.001);
+
+     }",file="wart2.txt" )
> wart2.dat <- as.list(wart1)
> wart2.ini <- list(mu = -3, taua = 1)
> wart2.par <-c("mean","mu","sigma2a","sigmaa","a")
> wart2.mod <- jags.model( file = "wart2.txt",
+                       data = wart2.dat,
+                       n.chains = 3,
+                       inits = wart2.ini,
+                       n.adapt = 1500 )
> wart2.res <- coda.samples( model = wart2.mod,
+                         var = wart2.par,
+                         n.iter = 1500,
+                         thin = 1 )
> summary(wart2.res)
>
```

```
> raneffs <- cbind(as.vector(summary(wart2.res)$statistics[(1:23)+23,1]),
as.vector(summary(wart2.res)$quantiles[(1:23)+23,1]),
+ as.vector(summary(wart2.res)$quantiles[(1:23)+23,5]))
> plot(x = expit(fixeffs[,1]),y = 5*(1:23)-2, xlim = c(0,0.125), pch = 16,
ylab = "Doctor", xlab = "Proportion diagnosed with warts (with 95% CI)",
yaxt = "n")
> segments(x0 = expit(fixeffs[,2]), y0 = 5*(1:23)-2,
x1 = expit(fixeffs[,3]), y1 = 5*(1:23)-2, lty = 1)
> points(x = expit(raneffs[,1]),y = 5*(1:23)-4, pch = 15)
> segments(x0 = expit(raneffs[,2]), y0 = 5*(1:23)-4,
x1 = expit(raneffs[,3]), y1 = 5*(1:23)-4, lty = 2)
> abline(v = expit(summary(wart2.res)$statistics["mu",1]),
col = "red", lty = 2, lwd = 2)
> legend(x = 0.09, y = 100,legend = c("Fixed","Random","Mean"),
lty = c(1,2,2), pch = c(16,15,NA), lwd = c(2,2,2),
col = c("black","black","red"))
```

WHAT ARE YOUR CONCLUSIONS ABOUT THESE RESULTS? REPEAT THE ANALYSES FOR PID

2.9 Meta-analysis

2.9.1 R Meta-Analysis Packages

A number of R packages are available to conduct meta-analysis [12]. The *rmeta* package was written by Thomas Lumley in 2009. It was written for categorical outcomes amenable to 2x2 tables and implements Mantel-Haenszel and DerSimonian and Laird approaches. The following code runs through a simple example using a data set included with the package called *cochrane*, which contains results of RCT's f corticosteroid therapy in premature labor and its effect on neonatal death. The function *meta.MH()* returns a Mantel-Haenszel summary estimate of the studies.

WHAT IS THE OVERALL ESTIMATE? HOW DO YOU INTERPRET IT? WHAT IS THE TEST OF HOMOGENEITY? WHAT DOES IT MEAN AND HOW DO YOU INTERPRET IT?

```
> # Mantel-Haenszel OR =0.53 95% CI ( 0.39,0.73 )
> # Test for heterogeneity: X^2( 6 ) = 6.9 ( p-value 0.3303 )
> # fail to reject the null hypothesis of homogeneity
> # variability across effect sizes does not exceed what would
> # be expected based on sampling error
```

Create a forest plot,

```
> plot(steroid)
```

And a funnel plot

```
> funnelplot(steroid)
```

HOW DO YOU INTERPRET THESE TWO PLOTS?

If we wanted to run a random effect model using DerSimonian-Laird methods to categorical data, we use the *meta.DSL()* function.

```
> steroid2<-meta.DSL(n.trt, n.ctrl, ev.trt, ev.ctrl, names=name, data=cochrane)
> summary(steroid2)
```

HOW DO THE RESULTS DIFFER FROM THE FIXED EFFECT MODEL?

Guido Schwarzer's 2010 *meta* package extends the capabilities of R to conduct meta-analysis. The *metabin()* function for binary outcomes allows a choice between a default Mantel-Haenszel approach and a log-transformed inverse variance approach (*method="Inverse"*). The function also automatically returns the results of both a fixed and random effect model. If you are interested in meta-analysis of continuous outcomes, you can use the *metacont()* function.

The data set *Fleiss93* contains the results of RCTS of the effect of aspirin on preventing death after myocardial infarction. Here we read in the data, then run both a Mantel-Haenszel and an inverse variance model.

```
> library(meta)
> data(Fleiss93) #ASA and MI
> head(Fleiss93)
> ASA<-metabin(event.e, n.e, event.c, n.c, data=Fleiss93,
studlab=paste(study,year), sm="OR")
> ASA2<-metabin(event.e, n.e, event.c, n.c, data=Fleiss93,
studlab=paste(study,year), sm="OR", method="Inverse")
> summary(ASA)
> summary(ASA2)
```

The forest plot comes with some additional bells and whistles.

```
> forest(ASA, fontsize=7)
```

[12]Refer to my slides on meta-analysis for a review of the basic principles of meta-analysis

As does the funnel plot. If you don't trust your visual assessment, the *metabias()* function will for funnel plot asymmetry, based on rank correlation or linear regression method.

```
> funnel(ASA)
```

The data set *Fleiss93cont* contains the mean or continuously distributed results of 5 studies of the effect of mental health treatment on medical utilization. We use the *metacont()* function to analyze these data.

```
> data(Fleiss93cont)
> metacont(n.e, mean.e, sd.e, n.c, mean.c, sd.c,
data=Fleiss93cont,studlab=paste(study,year) )
```

The *meta* package has a few additional tricks up its sleeve. *metagen()* can be used for generic inverse variance meta-analysis of treatment effects and their standard errors. *metaprop()* can be used for single proportions. *metainf()* is used analyze data for influential observations.

Finally, Wolfgang Viechtbauer's 2011 *metafor* package extends capabilities to conduct meta-regressions and assess moderator variables.

The file http://www.columbia.edu/ cjd11/charles$_d$*imaggio/DIRE/resources/Bayes/data/cholORDat.csv* contains data from 34 randomized clinical trials of cholesterol lowering interventions and total mortality. (Taken from Dr. Keith Abrams' 2008 Boston course on Bayesian Meta-analysis and Data Synthesis). *nC* is the number in the control arm, *nT* the number in the treatment arm, *rC* the number of deaths in the control arm, and *rT* the number of deaths in the treatment arm.

USE ONE OF THE R PACKAGES ABOVE TO CALCULATE A SUMMARY ESTIMATE FOR THE CHOLESTEROL DATA. USE THE INVERSE VARIANCE METHOD TO CONDUCT BOTH A FIXED AND RANDOM EFFECT META-ANALYSIS. INTERPRET THE RESULTS. REPORT AND INTERPRET THE Q STATISTIC. CREATE BOTH A FOREST AND FUNNEL PLOT. INTERPRET BOTH.

```
> forest(chol.meta, fontsize=7)
```

```
> funnel(chol.meta, fontsize=7)
```

2.9.2 Bayesian Meta-Analysis

A Bayesian approach to meta-analysis is essentially an extension of our discussion of the random effects model of physician diagnosis of PID and genital warts above.

USE THE NOTES ON CONDUCT A BAYESIAN HIERARCHICAL RANDOM EFFECT META-ANALYSIS OF THE CHOLESTEROL RCT'S. INTERPRET AND COMPARE THE RESULTS TO THOSE OBTAINED FROM THE INVERSE VARIANCE APPROACH ABOVE. Hint: Scroll down to "Example: Random Effects Meta-Analysis of Beta Blocker Studies", or look here for a real cheat.

2.10 Gibb's Sampler Code

This section is purely for your information. Topics like MCMC and Gibb's sampling can tend to be shrouded in mystery. This code walks through a simple implementation of a Gibb's sampler for a bivariate normally distributed with mean $\theta = (\theta_1, \theta_2)$ and known covariance matrix.

$$\begin{pmatrix} 1 & \rho\rho \\ 1 & \end{pmatrix}$$

.

It is another thing I've blatantly stolen from Bendix Carstensen. Taken with these notes it should help clear away the smoke and mirrors aspect of Bayesian simulation algorithms.

bivariate normal distribution, theta1, theta2 with known covariance matrix

```
> numsims<-1000
> rho<-0.8
> theta1<-numeric(numsims) # set up theta's to hold 10000 simulations
> theta2<-numeric(numsims)
>  # set first value theta1 to -3, use the definition of the conditional distribution
# for theta1
>  # to simulate a value for theta2
> theta1[1]<--3
> theta2[1] <- rnorm(1, mean=rho*theta1[1], sd=sqrt(1 - (rho^2)))
>  # set up loop to repeat this process again and again for theta1 and theta2
>  # note, first loop based on previous iteration, second based on current iteration
>
> for(i in 2:numsims)
+ {
+  theta1[i] <- rnorm(1,mean=rho*theta2[i-1],sd=sqrt(1 -(rho^2)))
+  theta2[i] <- rnorm(1,mean=rho*theta1[i],sd=sqrt(1 -(rho^2)))
+ }
> theta1[1:20]
> theta2[1:20]
>  # calculate mean and sd of last 500 samples
>
> mean(theta1[501:1000])
> mean(theta2[501:1000])
> sqrt(var(theta1[501:1000]))
> sqrt(var(theta2[501:1000]))
> cor(theta1[501:1000], theta2[501:1000]) # check correlation is same
> plot(density(theta1), ylim=c(0, 0.05))
> lines(density(theta2), add=T, col="blue")
>
```

2.11 acknowledgments

Except for the odds ratio function (which I wrote) the loop and function material came from `http://www2.warwick.ac.uk/fac/sci/moac/degrees/modules/ch923/r_introduction/r_programming/`

The aspirin bootstrap example comes from Efron and Tibshirani’s 1993 book "An Introduction to the Bootstrap" by way of Clifford Sutton.

The JAGS material came from Bendix Cartensen and Lyle Gurrin

The Bayesian meta-analysis code is from Keith Abrams

Chapter 3

Part III: Spatial Analysis in R

3.1 Introduction to the sp Package

The material in this section presents the *sp* package and some of the functions available for working with spatial data in R. I took this material from a much more extensive treatment I wrote, which is itself based extensively on Roger Bivand's work and the book he co-wrote, Applied Data Analysis with R Almost the entire text is a tutorial. The single question is at the end of the section in boldface caps.

First install the *sp* and *maps* package which consists of location coordinates and topsoil heavy metal concentrations collected from a flood plain of the river Meuse in the Netherlands.

```
> install.packages("sp")
> install.packages("maps")
```

Load *sp* and the *meuse* data set.

```
> library(sp)
> data(meuse)
```

The *meuse* data set is a simple data frame.

```
> str(meuse)
```

Say we wanted to plot the points that make up the *meuse* data frame. The usual *plot()* function returns the following, which is clearly note what we are after.

```
> plot(meuse)
```

The *x* and *y* variables are map coordinates, in meters keyed to the Netherlands RDH topographical convention. We can plot them as a simple scatterplot.

```
> plot(meuse$x, meuse$y)
```

Note that the *x* coordinate is the number of meters east of a reference point on the earth [1] and the *y* coordinate the number of meters north of the reference. [2] This is important to plot the data correctly. In general, *x* is assigned to longitude which must come before *y* (latitude) in plot statements.

```
> plot(meuse$y, meuse$x)
```

If all we were interested in was plotting the points, we could stop here. More often, the spatially-referenced points have some attribute or measurement associated with them. Converting a *spatial data frame* out of the *meuse* data

[1]There are different references, with the North American Datum (NAD) and the World Geodetic System (WGS) being the most commonly encountered in practice.

[2]This is often referred to as "Easting" and "Northing" in geographic documentation.

frame facilitates working with and analyzing these data. There are a number of ways to create or convert to spatial objects. The most straightforward approach is to use the *coordinates()* function in *sp*. This automatically converts the data frame to a 'SpatialPointsDataFrame' object, and when we can pass the entire data frame to the *plot()* function, we get the expected results.

```
> coordinates(meuse) <- ~x+y
> str(meuse)
> plot(meuse)
```

We can manipulate the plots of spatial objects as we can any object. Here we categorize and plot flood frequencies.

```
> meuse$ffreq <- as.numeric(meuse$ffreq)
> plot(meuse, col = meuse$ffreq, pch = 19)
> labs <- c("annual", "every 2-5 years", "> 5 years")
> cols <- 1:nlevels(meuse$ffreq)
> legend("topleft", legend = labs, col = cols, pch = 19, bty = "n")
```

Spatial objects are *S4* objects. Usually this will not affect how we work with them, but there are some differences. Note that the structure of the SpatialPointsDataFrame uses the @ convention to name and extract components.

```
> names(meuse@data)
```

But spatial objects come with special methods and tools. Say we are interested in identifying the three points on the lower right-hand size of the plot. We use the *select.spatial()*. Left click on some areas on the plot to surround the area in which we are interested (you don't need to close off the polygon) then right click to return the row numbers of those observations.

```
> select.spatial(meuse)
```

Spatial objects also come with special plotting methods. The *spplot()* method plots the attributes associated with the spatial points.

```
> spplot(meuse)
```

We can restrict to one attribute.

```
> spplot(meuse, "zinc")
```

The *bubble()* function returns (logically enough) a bubble plot of

```
> bubble(meuse, "lead")
```

As noted above, the *meuse* coordinates are referenced to Netherlands RDH topographical convention. If we want to combine our data with other maps or data sources (often the case in epidemiology), or need to be very precise in our locations we want to make sure they key up. The *proj4string()* function in *sp* uses the *rgdal* library to specify a coordinate reference system (CRS). Once you've applied a coordinate reference system, you can do things like convert from one system to another, e.g. with *spTransform()*, so you can combine data from different sources.

```
> proj4string(meuse) <- CRS("+init=epsg:28992")
```

The *maps* package is another useful too that is worth exploring.

```
> library(maps)
> data(usaMapEnv)
> map("county","new.york")
```

In the following three sections, you will walk through a realistic spatial epidemiologic analysis of traumatic brain injury in New York City.

3.2 About Areal data

More often than not, in the spatial epidemiology work I've done at least, I find myself working with these with so-called "areal" or ecological units rather than with discrete point processes. Areal units are characterized by irregular geographic categorizations like ZIP code tabulation areas (ZCTAs). They more properly be thought of as convenient "bins" into which spatial data are gathered, rather than as objects of interest in themselves. They also are likely to have a high degree of autocorrelation, with entities mirroring and influencing each other, for example by adopting similar policies and practices in response to similar issues as nearby areal units.

Areal units are prone to all the biases of other ecologic data, such as the well-known ecologic fallacy, as well as to some less well known biases peculiar to spatial data, such as the so-called modifiable areal unit problem, where a change in how we define an areal unit can affect and even change the direction of our results. [3]

3.3 New York City ZIP Code Neighbors

3.3.1 The New York City ZIP Code Tabulation Area Data Set

In the United States, a lot of health outcome data can be obtained for the ZIP Code Tabulation Area (ZCTA) level. I most often work with New York City data, so the first step in my spatial analyses is to create a geo-referenced New York City ZCTA map. [4] I did this in GRASS, and you can find the details in my comprehensive set of notes online. You don't have to worry about that for now, as I've provided you with an R data object that contains the SpatialPolygonsDataFrame. Download the file by clicking here. Note where folder where the download ended up, then use *load()* to load it into your working directory.

```
> load("~/spatialEpiNeighbors.RData")
```

Load *sp* and plot the file.

```
> library(sp)
> plot(nycZIPS)
```

It's always gratifying to see a procedure work, and this map of New York City ZCTAs' appears reasonably complete. To continue our due dilligence at this point, though, we must look at the dataframe object itself. We are interested at this point in the ZCTA's themselves, so let's start looking at them. Although it's a spatial object, we can treat it as dataframe object, which allows us to run some simple summary procedures.

In the next few lines of code, we determine that there are 295 ZCTA's in the spatial object. This does *not* coincide with one source that states there are either 176 ZCTA's in New York City (41 in Manhattan, 37 in Brooklyn, 61 in Queens, 25 in the Bronx, and 12 on Staten Island). To complicate matters even further, some buildings in Manhattan (like the Federal Reserve Bank at 33 Liberty Street) have their own ZIP Codes, some ZIP Codes are reserved for Post Office Boxes, and the World Trade Center former ZIP Code of 10048 (also reserved, for future use) is still in circulation in some data sources. As I said, areal data can be messy.

```
> length(nycZIPS$zcta5)
```

Next, we see that a number of ZIP Codes are repeated, some many times:

```
> table(nycZIPS$zcta5)
```

[3]A commonly known example of a modifiable area unit bias, is gerrymandering political districts to enhance the fortunes of a political party.

[4]There is, to my knowledge anyway, no standard such map. I have created a number of these ZCTA maps over the years with varying degrees of success. The approach I describe here is so far the most accurate.

There are a couple of approaches we can take to address these duplicate values. We might be tempted to use the unique() function,

```
> nycZIPSunique<-nycZIPS[unique(nycZIPS$zcta5),]
> plot(nycZIPSunique)
```

That is not at all what we want. [5] The unique() function *deletes*, all the duplicated values. Another approach could be to create a logical vector of duplicate values using the duplicated() function,

```
nycZIPS$dup<-duplicated(nycZIPS$zcta5)}
```

Or, more directly, restricting to non-duplicate values.

```
> nycZIPSnodup<- nycZIPS[ !duplicated(nycZIPS$zcta5),]
> length(nycZIPSnodup)
> plot(nycZIPSnodup)
```

Closer, but clearly unacceptable. We are at the correct number of 177 ZCTA's, but the plotted map doesn't look like New York City. R is now choosing the *first occurrence* of the duplicates. Some of these are smaller polygons contained within the same-named larger polygons, or small islands [6]. While these data are better than the two previous spatial data frame objects, they remain problematic. They do not represent the geography correctly, and as a result of this our neighbor relationships will be similarly incorrect.

We need a way to choose the larger polygons. Since we are working with a spatial object, we can use the *area* of the polygons, which are conveniently stored in one of the slots of an *sp* object. Getting at slot information is a bit fiddly. We can get at each individual area by specifying the item using slot notation. Here is the area of the first ZCTA.

```
> nycZIPS@polygons[[1]]@Polygons[[1]]@area
```

A better approach, since we are dealing with fairly simple polygon data, is to use sapply() to return a vector of areas and append them to the ZCTA data frame.

```
> areas<-sapply(slot(nycZIPS, "polygons"), slot, "area")
> nycZIPS$area<-areas
```

The next step involves creating a logical vector of TRUES and FALSES based on whether the area is the maximum area for the set of ZCTAs. We use the *ave()* function to generate a vector that contains the maximum of the area variable for every ZCTA. We set this equivalent to the area variable (using the == operator) to create the logical vector called *sel*. Then we use the logical vector to select rows from our spatial data frame. The plot shows that this is a much better approach.

```
> sel<-ave(nycZIPS$area, nycZIPS$zcta5, FUN=max)==nycZIPS$area
> nycZIPS2<-nycZIPS[sel,]
> plot(nycZIPS2)
> length(nycZIPS2)
```

The take-away message is that spatial data, like any data, requires care at the earliest stages to ensure valid analyses. Just because you get a map, doesn't mean the data are correct.

3.3.2 NYC ZCTA Contiguity Neighbors

As noted, areal units are similar and related to other nearby areal units. We need to establish these neighbor relationships. (Again, refer to the full set notes for a more comprehensive discussion)

To create neighbors, we use the *poly2nb()* function in the *spdep* package which outputs a list object to which we assign the name *nycZIPS.nb*. The object indexes each ZCTA and lists the other ZCTA's that neighbor it. We print the relationships (in red).

[5] Queens has effectively seceded from New York City. Those lone ZCTA's where Queens used to be are Riker's Island, which is administratively part of the Bronx.

[6] Of which there are surprisingly many in New York City, among my favorites being Rat Island, and Chimney Sweeps Island

```
> library(spdep)
> nycZIPS.nb<-poly2nb(nycZIPS2)
> head(nycZIPS.nb)
> plot(nycZIPS.nb, coordinates(nycZIPS2), col="red", pch=".")
```

Note that contiguity does not account for cross-county relationships that are separated by rivers. New York City is so well knit by bridges, roads and public transportation like the subway that such geographic boundaries are relatively unimportant and are likely not to define neighborhood links. Graph-based neighbor definitions may address this shortcoming.

3.3.3 NYC ZCTA Graph-Based Neighbors

In this next set of code, we apply graph-based neighbor definitions to our New York City ZIP Code tabulation areas. We begin by loading the tripack package and extracting the coordinates and row names for the coordinates. We then create and plot Delaunay, Sphere-of-Influence and Gabriel graphs. Again, refer to this document for definitions.

```
> library(tripack)
> coords<-coordinates(nycZIPS2)
> IDs<-row.names(coords)
> nycZIPS.nb4<-tri2nb(coords,row.names=IDs)
> plot(nycZIPS.nb4,coords, pch=".", col="red")
```

The Delaunay graph is indeed pretty, and it does ignore water boundaries, but it results in some odd neighbors. ZCTA's in northern Manhattan and the Bronx are "neighbors" of Staten Island, which is over 25 miles away, and differs in important geographic and social ways. A Gabriel graphs addresses this inconsistency:

```
> nycZIPS.nb6<-graph2nb(gabrielneigh(coords), row.names=IDs)
> plot(nycZIPS.nb6,coords, pch=".", col="red")
```

3.4 Spatial Correlation of New York City Pediatric Traumatic Brain Injury

In this example, we'll look at New York City traumatic brain injury diagnoses in a group of children younger than 9 years of age. The file tbizip has age-stratified gender-specific TBI and population counts at the New York City ZCTA level, as well as gender-standardized morbidity ratios.[7] Read in and explore the data frame.

```
#these data are restricted, speak to me...
> tbizip<-read.csv("~/tbizip.csv",header=T, stringsAsFactors=F)
> names(tbizip)
```

3.4.1 Merging the data file with the map file

We want to merge our disease data with the the map object nycZIPS2 we created in the previous section. We take a quick look at our disease data set, and realize that there are 195 rows of data in nyczip compared to 177 ZCTAs in nycZIPS2.

```
length(nycZIPS2$zcta5)
length(tbizip$zips)
```

The observations in 'tbizip' that do not correspond to valid ZCTA in our map object appear to be empty. [8]

```
tbizip$tbi_tot[!tbizip$zips %in% nycZIPS2$zcta5]
tbizip$smr[!tbizip$zips %in% nycZIPS2$zcta5]
```

[7]The data are aggregated and the years over which the injuries occurred are masked in order to maintain confidentiality.

[8]I used the *%in%* operator which returns a logical vector of TRUES and FALSES for whether one vector is in another to index one file with the other

We will merge the two data objects into a single dataframe object called nycZIPS.df, restricting the merge to only the observations that are present in the ZCTA map object. The ID variables are tbizip$zips and nycZIPS$zcta5. We perform a left merge that ensures the result is keyed to the map object.

```
library(sp)
nycZIPS.df<-merge(nycZIPS2,tbizip,all.x=T, all.y=F, by.x="zcta5", by.y="zips")
```

3.4.2 Plotting a choropleth

In preparation for plotting a choropleth of TBI rates, we will need to do some data manipulation. We begin by creating variables for counts of diagnoses and population at risk. We deal with missing data by assigning zero to areas with no diagnoses, and adding 0.1 (to avoid dividing by zero) to ZCTA's for which there are no records of children enrolled in Medicaid. NB: We assign these new variables to our *spatial polygons map object* nycZIPS2, so we can work with the sp object. Otherwise, methods like spplot will not work on the dataframe object.

```
cases<- nycZIPS.df$tbi_tot78 + nycZIPS.df$tbi_tot012 +nycZIPS.df$tbi_tot34 +  nycZIPS.df$tbi_tot56
cases[is.na(cases)]<-0

pop<-nycZIPS.df$pop_tot78 + nycZIPS.df$pop_tot012 +nycZIPS.df$pop_tot34 +  nycZIPS.df$pop_tot56
pop[is.na(pop)]<-.1

nycZIPS2$cases<-cases
nycZIPS2$pop<-pop
```

We calculate a rate per 10,000 population and again assign that variables to the map file.

```
> rate<-cases/pop*10000
> nycZIPS2$rate<-rate
```

Based on a summary of the rate outcome variable, we create four categories using the *cut()* function, and map the categories with colors chosen from *RColorBrewer*.

```
> summary(nycZIPS2$rate)
> nycZIPS2$rate.cat<-cut(nycZIPS2$rate, breaks=c(0, 30, 60, 100, 200),
labels=c("<30", "30-60", "60-100", ">100"))
```

```
> library(RColorBrewer)
> tbiplot1<-spplot(nycZIPS2, "rate.cat",col.regions = brewer.pal(4, "Reds"))
> print(tbiplot1)
```

After this preliminary work, we can turn our attention to statistical tests of global autocorrelation for the ZCTA-based pediatric TBI injury rates. [9]

3.4.3 Global tests of autocorrelation in the NYC TBI data

We now apply global tests of spatial autocorrelation to our TBI data. Here we use three variants of Moran's I as a way to test the sensitivity of our results and check for possible problems with model mispecification. We first use the *moran.test()* function with the Gabriel neighbor list we created above. We are using a binary definition for neighbors and must address the issue of zero neighbors. [10] The resulting weight list object includes a vector of "nieghbors" and their associated "weights".

```
> nycZIPS.wts<-nb2listw(nycZIPS.nb6, zero.policy=T, style="B")
> moran.cases<-moran.test(nycZIPS2$cases, listw=nycZIPS.wts, zero.policy=TRUE)
> moran.cases
> moran.cases$p.value
```

[9] Again, see the full notes for a description of these tests.
[10] Note that when we want to print this object, we have to re-invoke the zero.policy argument.

The test returns a statistically significant p-value indicating areas of autocorrelation or clustering in our data.

Next, we use a Monte Carlo simulation version of the Moran test on the TBI rate variable. This also returns a statistically significant result.

```
> set.seed(987)
> moran.mc(rate, listw=nycZIPS.wts, nsim=999, zero.policy=TRUE)
```

Lastly, we use an Empirical Bayes version, specifying both the count and the population at risk.

```
> EBImoran.mc(n=nycZIPS2$cases, x=nycZIPS2$pop, listw=nycZIPS.wts, nsim=999)
```

We see that all three tests returned results indicating similar levels of correlation. Clearly, areas with increased numbers or proportions of pediatric TBI are near each other.

Now that we have a sense of what spatial areal units are, how neighbor relationships are defined and weighted, and how we can test for spatial autocorrelation, we turn our attention to spatial modeling of health outcomes.

3.5 Modeling Pediatric TBI in New York City

Taking up where we left off, after determining that pediatric traumatic brain injury in New York City is spatially dependent, and plotting choropleths to identify areas of increase risk, we might reasonably be interested in analyzing the data to see how neighborhood variables might be related to risk. In this example, we'll look at housing characteristics. Our first step then is to get some housing data for our ZCTA map.

3.5.1 Entering and Cleaning the Data

The US Census Bureau is a compendious (if not always user friendly) source of such information. To create the kind of custom geographically-referenced data set we need, we can navigate from the Census Bureau homepage to the American Factfinder data tool. On that page, we choose the Dicenial Census from among the data sets under the "Data Sets" menu on the left and select "Custom Table" from the list of options for the "2000 Census Summary File 1". On the next page, under "Choose a selection method", click the "geo within geo" tab. [11] We then use the series of drop-down menus to select 5-digit ZIP Code tabulation areas within New York City, and on the resulting page, we select (and individually add) each variable we want. Finallly, we save the resulting table as a .csv file, which we read into R after doing some data cleaning, dropping some unnecessary variables, identifying odd or inappropriate ZCTA's and removing those observations.

```
> housing<-read.table(file="http://www.columbia.edu/~cjd11/charles_dimaggio/DIRE
/resources/spatial/censusPopHousing.txt", header=F, sep=",",
skip=2, col.names=c("x1", "zcta", "x2", "x3","x4", "house.tot", "pop.tot",
"house.rent", "house.occ"), colClasses=c("character",
 "character", "character", "character","character",
"numeric", "numeric","numeric", "numeric"))
> housing<-housing[,-c(1,3:5)]
> housing$zcta<-as.numeric(housing$zcta)
> complete<-complete.cases(housing)
> housing<-housing[complete,]
> head(housing)
```

In the section on spatial neighbors, we learned how to extract the area data from the spatial polygon object. If we did not have that data, we could get it as well from the US Census Bureau in their US Gazatteer files, and read it into an R data set:

[11] I found these links only after numerous frustrating and time consuming dead ends. The Census Bureau is "improving" their site and will be transitioning to a new American Factfinder2 tool, with which I have yet to be able to create these kinds of tables. Even more recently, I came across the R Census2000 and Census2012 packages which make working with census data much easier. I have information on these packages in the full set of spatial notes.

```
> area<-read.table(file="http://www.columbia.edu/~cjd11/charles_dimaggio/
DIRE/resources/spatial/censusSqMiles.txt", header=T)
> area<-area[,-c(2,3,5:7)]
> head(area)
```

We load our our NYC ZIP Codes map and weight objects which we created in our previous session. Recall that if we just use *merge()* the resulting simple dataframe object will not be an spatial polygon dataframe and will not work with sp methods. As in the previous set of notes, we add variables to the map object after indexing by a logical vector to restrict the entries to our 177 ZCTA's. [12]

```
> nycZIPS3<-nycZIPS2
> housing2<-housing[housing$zcta %in% nycZIPS3$zcta5,]
> housing2<-housing2[order(housing2$zcta),]
> nycZIPS3<-nycZIPS2[order(nycZIPS3$zcta5),]
> nycZIPS3$house.pop<-housing2$house.pop
> nycZIPS3$pop.tot<-housing2$pop.tot
> nycZIPS3$house.rent<-housing2$house.rent
> nycZIPS3$house.occ<-housing2$house.occ
```

Now we do the same thing with the square mile area variable from the US census. One important note, the US Census file of area by square miles does not include ZIP code 10048, the old World Trade Center ZIP Code. We add it to the areal sp object data as a value of 0.0001. [13] We also create a variable called "rent.dense". This is measure of rental housing density in a ZCTA and it is based on the number of rental units in a ZCTA divided by the area in square miles. The choice of this variable is motivated by the hypothesis that, in New York City at least, rental density is an indirect measure of the number of high-rise apartments in an area, which in turn may be related to TBI risk through falls.

```
> area2<-area[area$zcta %in% nycZIPS3$zcta5,]
> wtc<-data.frame(zcta="10048", sq.miles=0.0001)
> area2<-rbind(area2, wtc)
> area2<-area2[order(area2$zcta),]
> nycZIPS3$sq.miles<-area2$sq.miles
> nycZIPS3$rent.dense<-nycZIPS3$house.rent/nycZIPS3$sq.miles
```

One more bit of data housekeeping. We will be conducting analyses that will return errors if they lead to division by or a log transformation of a zero value. [14] In the previous bit of code, we added a 0.0001 to ZCTA 10048 to avoid a missing value. In the case of our outcome and predictor variables, we will convert zero values to missing (NA). We do this for two reasons. First, spatial areas with no people (or at least no people who meet our study criteria) are essentially undefined, so NA makes sense. And, R has many options for dealing with NA values which will allow us some flexibility down the line.

```
> nycZIPS3$rate[which(nycZIPS3$rate==0)]<-NA
> nycZIPS3$house.rent[which(nycZIPS3$house.rent==0)]<-NA
> nycZIPS3$house.occ[which(nycZIPS3$house.occ==0)]<-NA
>
```

3.5.2 Exploring the Data

We begin our analyses as we do with all data, whether they be spatial or not: exploration, cleaning, simple descriptive analyses. In the interest of time and space, I'll omit the initial analyses [15], but you should spend a little time going over the data.

Our outcome variable is the rate of pediatric TBI among the population of children enrolled in Medicaid. Here, we simply plot that rate

[12] I found this alternative approach online which seems like a much more elegant way to do the same thing.

[13] A bit more on this in the following paragraph

[14] Both of which are undefined or infinite

[15] I, in fact, wrote an entire paper just looking at the descriptive statistics of this data set.

```
> plot(density(na.omit(nycZIPS3$rate)))
```

As we noted above, if we plan to use autoregressive procedures, there is an underlying assumption of normality, which these data do not appear to meet. A simple log transformation may be helpful:

```
> plot(density(log(na.omit(nycZIPS3$rate))))
```

This is closer to what we need to meet our assumptions.

Next, we graph a simple matrix of scatterplots of the numeric variables in our spatial data frame. We first create a data set restricted to just the variables we are interested in. The we use the *pairs()* procedure to plot them.

```
> scatterData<-cbind(log(nycZIPS3$rate), nycZIPS3$house.rent,
nycZIPS3$house.occ, nycZIPS3$rent.dense)
```

```
> colnames(scatterData) <-c("log TBI rate", "rentals", "owned", "rental density")
```

```
> library(lattice)
> print(splom(~na.omit(scatterData), panel = function(x,y)
      {panel.xyplot(x,y); panel.lmline(x,y)}))
```

Turn your attention to the upper left (and conversely the lower right) scatterplots of the association between the log of the TBI rate and rental density. We will explore this possible association.

3.5.3 Modeling the Data

Non-Spatial Models

As I mentioned on the very first page of these sets of notes, whether a spatial component adds anything your analysis is not a given.[16] With that in mind, let's first model the ZCTA-based TBI rates with two models that do not take the spatial component into account.

We first perform a simple linear regression model of the log of the TBI rates by the rental density variable.

```
> linear.model<-lm(log(rate) ~ rent.dense, data=nycZIPS3)
> summary(linear.model)
> exp(linear.model$coefficients[2])
```

The model is statistically significant (p=0.004) with each one unit increase in the number of rental units per square mile, we can expect a 1 unit increase (notice we exponentiated back to our original scale) in the TBI rate. Although the adjusted R^2 reflects a poor model fit. Let's see what a Poisson model of the case count returns.

```
> poisson.model <- glm(cases ~ rent.dense + offset(log(pop)),
data = nycZIPS3, family="poisson")
> summary(poisson.model)
> exp(poisson.model$coefficients[2])
```

The results are not too very different from those of the linear model. Now let's see what, if anything, the spatial component adds.

3.5.4 Spatial Models

We need the packages *sp, spdep, tripack* to create spatial neighbor objects with *nb2listw()* and for the call to *spautolm()*.

```
> coords<-coordinates(nycZIPS3)
> IDs<-row.names(coords)
> nycZIPS.delauny<-tri2nb(coords,row.names=IDs)
```

[16]To be honest, I think its fairly clear that there is *some* spatial component to these data just based on our previous Moran's test. But bear with me.

```
> nycZIPS.soi<-graph2nb(soi.graph(nycZIPS.delauny, coords), row.names=IDs)
> nycZIPS.gabriel<-graph2nb(gabrielneigh(coords), row.names=IDs)
```

```
> tbi.sar.model <- spautolm(log(rate) ~ rent.dense,
data = nycZIPS3, nb2listw(nycZIPS.delauny), zero.policy=TRUE)
> summary(tbi.sar.model)
> exp(tbi.sar.model$fit$coefficients[2])
```

While the point estimate remains unchanged from our previous, non-spatial, models, we now see that taking spatial structure and non-independence of the observations into account results in a non-statistically significant p-value. This makes sense, since the spatial component essentially adds heterogeneity.

You may notice that I based the spatial structure on a Delauny graph. We should check the sensitivity of our results to the choice of neighbor structure. Let's see what the results would be with a sphere-of-influence set of neighbors.

```
> tbi.sar.model2 <- spautolm(log(rate) ~ rent.dense,
data = nycZIPS3, nb2listw(nycZIPS.soi), zero.policy=TRUE)
> summary(tbi.sar.model2)
> exp(tbi.sar.model2$fit$coefficients[2])
```

Clearly, the results are quite sensitive to our choice of spatial structure. Given the apparent clustering of TBI outcomes in our maps, and the fact that New York City neighborhoods tend to vary in similarly non-homogenous ways in terms of socioeconomics and housing, we could legitimately challenge the choice of a global or simultaneous autoregression model. Let's consider a conditional autoregression model.

```
> tbi.car.model <- spautolm(log(rate) ~ rent.dense,
data = nycZIPS3, nb2listw(nycZIPS.delauny), zero.policy=TRUE, family="CAR")
> summary(tbi.car.model)
> exp(tbi.car.model$fit$coefficients[2])
```

Modeling the data this way, returns a statistically significant result.

```
> tbi.car.model2 <- spautolm(log(rate) ~ rent.dense,
data = nycZIPS3, nb2listw(nycZIPS.soi), zero.policy=TRUE, family="CAR")
> summary(tbi.car.model2)
> exp(tbi.car.model2$fit$coefficients[2])
```

Again, our results are hardly robust to our assumptions.

REPEAT THESE ANALYSES FOR ADULTS. INTERPRET AND COMPARE THE RESULTS FOR CHILDREN.

Chapter 4

Part IV: ggplot

4.1 About ggplot2

R is justly admired for its graphics capabilities. R graphics comes in three flavors. The venerable *base graphics* which come with the base R installation, the *lattice* package which is the R implementation of the trellis data visualization system for multivariable data, and *ggplot2* which is Hadley Wickham's implementation of Leland Wilkenson's *Grammer of Graphics* (hence the "gg"), which has captured the imaginations of many R users because of its elegance, modularity and scalability. It also makes very nice images. Most of the following tutorial and exercise material was taken from Hadley Wickham's ggplot 2 book, though I have also "borrowed" liberally from other sources, notably a very good, very brief introduction to ggplot2 by Christophe Ladroue.

I find *ggplot2* to have a steep learning curve. Not because it is inherently difficult to use, but because the underlying concepts and nomenclature are unlike the way I'm used to working in R. But, I am a big fan and find myself using it more and more, almost to the exclusion of base graphics. I like being able to build up a graphic layer by layer, saving as I go along, and I think the faceting feature makes for particularly informative graphics. The benefits of *ggplot2* do not come without some cost. The program is quite noticeably slower than base graphics. Recent implementations have tried to some extend addressed this issue, but you will still notice the difference with large data sets.

One last piece of philosophy. Though it does produce some nice publication quality images, ggplot2 is very much intended as a tool to *explore* data, rather than a a production tool.

> *NB: A new version of ggplot2 was just released. I tried to incorporate the new language (which I am still learning myself), but you may get warnings about deprecated command for some of the examples.*

4.2 Quick Intro

There are 3 ways to invoke a plot in *ggplot2*.

1. *qplot* - a quick plot with defaults
2. *ggplot+layer* - build a plot layer by layer, full control

3. $ggplot{+}geom_xxx - goodwaytolearnhowtouse$ggplot2

Load *ggplot2* and the *diamonds* data set. Create a small version of the data to plot more quickly.

```
> library(ggplot2)
> data(diamonds)
> set.seed(1960)
> dsmall<-diamonds[sample(nrow(diamonds),999),]
> names(dsmall)
> head(dsmall)
```

Run the following plot.

```
> p1<-ggplot(dsmall)+geom_point(aes(x=carat,y=price,color=cut))
> print(p1)
```

There are two parts to the code, separated by a + sign. [1] First we identify the **data**. The initial part of the syntax (*ggplot(dsmall)*) creates a ggplot object out of a data frame. *ggplot()* will *only* accept a data frame as an argument. Think of it as "setting the stage". It does not plot anything, and in fact will not run (returning an error message) if submitted without the additional information *ggplot2* needs to set up a plot.

That additional information is contained in the second part of the statement (*geom_point(aes(xc̄arat,yp̄rice,colorc̄ut))*), which establishes the **geometry**, and itself consists of two parts. First you *select* a geometry. *geom_point()* is used to create scatterplots. It is one of *many* geometries you can invoke. *geom_smooth()* plots trend lines (using splines), *geom_density()* for density plots, *geom_histogram()* for (you guessed it) histograms.

Then you *define* the geometry *aesthetics* in an *aes()* statement. In this case, you are setting three aesthetics. The *x* and *y* variables are clearly required to define a scatterplot. The third aesthetic (*colorc̄ut*) introduces some of the multivariable power of *ggplot2*. It assigns a color to each category of the variable "cut", automatically creating a very nice legend in the process.

Another thing to notice in this brief bit of syntax is that once you've created the ggplot object by identifying a data frame, you no longer need to invoke that data frame name when defining geometries and aesthetics. So, you can refer to variable names on their own, rather than have to use the more verbose *dataframe$variable* format required in base (and lattice) graphics.

Notice also, that we assigned the plot to an object called *p1*. We can now start building up the graphic by adding to and saving additional layers.

ggplots can also be defined by the **scales** on which the variables are measured. Here, we see what the plot looks like with the price (y variable) on the log scale.

```
> p2<-p1+scale_y_log10()
> print(p2)
```

We can also easily stratify or **facet** the data. Perhaps we want to stratify the plots by the variable "color".

```
> p3<-p2+facet_wrap(~color)
> print(p3)
```

These, then, are the basic elements of plotting with *ggplot2*:

The basic idea is that we layer the various elements of a plot over each other. These elements consist of:

1. the **data** themselves.
2. the **geometry** or figures that represent the variables which are themselves mapped to attributes or *aesthetics*
3. **scales** which represent the size or measurement value of a plot object in relation to the size of a data point
4. **faceting**, stratifying or displaying multiple slices of the data (sometimes called "lattice" or "trellis" graphing and considered one of the strengths of a graphing in R)

There are additional elements we can control, including

1. **statistics** that we might want displayed
2. **coordinates** of a graph (though we usually use cartesian coordinates)
3. **options** like titles, axis statements and legends

[1]The use of the plus sign to string syntax is pretty much unique to *ggplot2* and not really seen elsewhere in R. It is one of those aspects of *ggplot2* that I found to be a conceptual hurdle.

There are a number of ways of writing a *ggplot* statement. The following three statements are equivalent. I find it easier to remember to establish the data first, then define the aesthetics as part of the geometry.

```
ggplot(dsmall)+geom_point(aes(x=carat,y=price,colour=cut))
ggplot(dsmall,aes(x=carat,y=price,colour=cut))+geom_point()
ggplot(dsmall,aes(x=carat,y=price))+geom_point(aes(colour=cut))
```

4.2.1 About aesthetics

There is a difference between *assigning* or mapping aesthetics to variables, and *setting* aesthetics. Assigning or mapping is done in the *aes()* statement. Assigning an aesthetic is done outside the aesthetic statement. Take a look at this statement and compare the plot to the first plot above.

```
> print(ggplot(dsmall)+geom_point(aes(x=carat,y=price),color="blue"))
```

If you try to *set* and aesthetic inside the *aes()* statement, you'll get weird results.

```
> print(ggplot(dsmall)+geom_point(aes(x=carat,y=price,color="blue")))
```

4.3 About facets

Faceting is a great feature. As we've seen *facet_wrap()*, stratifies the data by a variable. You can control some aspects of the faceting, e.g.

```
> print(p2+facet_wrap(~color))
> print(p2+facet_wrap(~color, ncol=1))
> print(p2+facet_wrap(~color))
```

You can cross classify the plots by two variables with *facet_grid()*.

```
> print(p2+facet_grid(cut~color))
```

4.3.1 Adding geometries

You can geometries together. Here, we add a trend line with *geom_smooth()* to a scatter plot. Note that in this case we must assign the aesthetics as part of the *ggplot()* statement.

```
> print(ggplot(dsmall,(aes(x=carat,y=price)))+geom_point()+geom_smooth())
```

4.4 qplot

As noted above, you can get the same plot with different syntax. *qplot()*, which stands for "quick plot", is the analog of the *plot()* function in base R. It's "quick" in that defaults are built in. In the statement below, [2] the first two arguments are the variables you want plotted, followed by the data set that they come from, then some options. Here, I'm using "alpha=" to blur the image so we can see overlying data points better. Play around with the "alpha" option by using a denominator of 10 or 100. Notice that I'm plotting the log of the variables. ggplot2 is very good about this kind of thing.

```
> print(qplot(log(carat), log(price), data=diamonds, alpha=I(1/200)))
```

Stratify the relationship of carat and price by price, reflected in the size of the geometry.

```
print(qplot(carat, price, data=dsmall, size=price))
```

[2]You may notice that I'm enclosing the statement in a *print()* statement. I do that only to Sweave this document. You don't have to.

So far *qplot* has assumed, because we identified two numeric variables, that we want a scatter plot. It's usually pretty good about that.

REQUEST A QPLOT OF PRICE USING THE SAMLL DIAMOND DATA SET

```
qplot(carat, data=dsmall)
```

You can request explicit geometries. Let's take a look at the "geom" option. Notice that "geom" can take multiple geometries. You can turn the confidence interval off with "se=F". The "span" parameter controls how smooth or "wiggly" the line is.

```
> print(qplot(carat, price, data = dsmall, geom = c("point", "smooth"), span=.2) )
```

The default method for the "smooth" line (for data sets less than 1000 points) is the loess method, but you can use "method=" to define other approaches such as a linear model (method="lm").

```
> print(qplot(carat, price, data = dsmall, geom = c("point", "smooth"), method="lm") )
```

Boxplots are useful geometry when you want to see how a continuous variable varies by the values of a categorical variable. Here we see price per carat conditional on color.

```
> print(qplot(color, price / carat, data = diamonds, geom = "boxplot"))
```

REWRTIE THE LAST TWO PLOT STATEMENTS USING THE *ggplot+geom_xxx* APPROACH

```
> print(ggplot(dsmall)+geom_boxplot(aes(color, price / carat)))
```

4.4.1 density plots and histograms

A density plot displays the cumulative distribution of a variable. In base R, you would write:

```
> plot(density(dsmall$carat))
```

Notice, you are first passing the variable to the density function, then plotting the resulting density object. In ggplot2, you get the same results with:[3]

```
> print(qplot(carat, data = dsmall, geom = "density"))
```

You can pass an "adjust" argument to control how smooth you want the graph to be, with higher values returning smoother plots. Here, we try a smaller value to illustrate some o of the variability in the data.

```
> print(qplot(carat, data = dsmall, geom = "density", adjust=.4))
```

A histogram presents information similar to a density plot, but note the data are "binned" into categories rather than continuous.

```
> print(qplot(carat, data = dsmall, geom = "histogram"))
```

You can control how the data are binned with the "binwidth" argument.

```
> print(qplot(carat, data = dsmall, geom = "histogram", binwidth=.01))
```

Clearly, the choice of binwidth affects our impression of the data, so it's a good idea to try out a few. The ease with which *ggplot2* allows you to investigate this kind of parameter is one of the strengths of the package.

As we did before, we can stratify the output by a third variable. Here, we set each color of a diamond a different color on the graph. [4]

```
> print(qplot(carat, data = dsmall, geom = "density", color = color))
```

WRTIE A STATEMENT USING THE *ggplot+geom_xxx* APPROACH FOR A DENSITY PLOT OF PRICE GROUPED BY CUT AND STRATIFIED BY DIAMOND COLOR.

[3]In this instance, I think I prefer the base graphics...

[4]Don't be confused. The parameter is called "color=", it just so happens in this data set there is a variable called color which refers to the color of the diamond.

bar charts

As opposed to histograms, which are bins of a quantitative variable, bar charts represent levels of an explicitly categorical variable. The geometry in the ggplot2 call, is "bar". Here are the counts of diamonds in each color category.

```
> print(ggplot(dsmall)+geom_bar(aes(color)))
```

REWRTIE THE SYNTAX AS A QPLOT STATEMENT.

4.4.2 line graphs and time series

The geometry for times series graphs is "line". We load 40 years worth of US economic, and create some plots.

```
data(economics)
head(economics)
class(economics$date)
```

First, we look at the unemployment rate which, again, ggplot2 will calculate on the fly.

```
> print(qplot(date, unemploy / pop, data = economics, geom = "line"))
```

Then we look at median number of weeks folks were unemployed.

```
> print(qplot(date, uempmed, data = economics, geom = "line"))
```

We see that length of unemployment increased over time in ways that the unemployment rate did not.

4.4.3 faceting

As we have seen, faceting refers to sub-setting or grouping data by creating panels, or multiple plots on the same page. The *qplot* option is "facets=" with which you define a row.variable column.variable. To get a single row based on one grouping variable, specify rowVar .

Here, we plot carat by density, subgrouping again with facets of color. Note, here we use ..density.. to tell ggplot2 to map density to the y-axis.

```
> print(qplot(carat, ..density.., data = dsmall,
> facets = color ~ ., geom = "histogram", binwidth = 0.1, xlim = c(0, 3)))
```

4.5 ggplot()

qplot() takes care of a lot of the preliminary work of setting up a plot, by using a bunch of defaults. You may find it is all you need. You will, though, most likely choose to unlock some of the additional capabilities of ggplot2. This involves *layering* the important elements of a plot i.e. *data, geometry and aesthetic mappings, statistics*

In its basic implementation, *ggplot()* takes two essential arguments for a plot: the data, and the aesthetic mappings. The data are a data frame. Here is an example:

```
> p<-ggplot(dsmall, aes(carat, price, color=cut))
```

Note that this does not produce a plot, it establishes the defaults. To produce a plot, we must define and add a *layer*. Adding a layer can be accomplished with a + sign and the *layer()* function, e.g.

```
> p<-p + layer(geom="point")
> p
```

Or by directly adding a geometry

```
> p<-p + geom_point()
> p
```

the *layer()* function can take a number of arguments: layer(geom, geom_params, stat, stat_params, data, mapping, position).

The *geom_XXXX()* function is essentially a shortcut, and accepts arguments for mapping aesthetics, stratification, etc... Because each geometry is associated with a default statistic,there is yet *another* basic way of specifying a layer, with the *stat_XXXX()* function. Finally, as seen above, the *aes()* function can be used to add, override, or remove aesthetics. You will settle into your own routine, I'm sure, but you should be aware of these different approaches so you can more effectively "borrow" code you find online.

Whichever approach you use (*qplot, or ggplot() with layer(), geom_xxxx(), stat_xxxx()*) you will want to save plots as objects as you build up your image. *ggplot* objects are like other R objects, and can be saved to a workspace file, explored, re-invoked.

4.5.1 About "aesthetics"

The aesthetics of a plot describe the variables you want mapped. They are controlled by the *aes()* function, which is passed as an argument.

aes() takes arguments for the x and y variables (at a minimum) and for possible third dimensions, e.g.

```
aes(x=weight, y=height, color=age)
```

Note, you *map* an aesthetic to a variable, e.g. aes(color=group), that will vary by value. You *set* and aesthetic to a constant, e.g. color=blue, that remains constant.

E.g.,

```
p <- ggplot(mtcars, aes(mpg, wt))
p + geom_point(color = "darkblue")
```

sets the color to dark blue instead of black.

```
p + geom_point(aes(color = "darkblue"))
```

maps a variable called "color" to varying shades of dark blue.

grouping

To illustrate how to group data in ggplot2, we'll work with a longitudinal data set of repeat observations. The "Oxboys" data set consists of repeated height and weight observations on 26 male Oxford students.

```
> library(nlme)
> data(Oxboys)
```

Say we want to plot the measurements over time for each boy. The variables we want to plot are age (on the x axis) and height (on the y axis). The appropriate geometry is a line or time series.

```
> ggplot(Oxboys, aes(age, height)) + geom_line()
```

Whoa. What's *that* all about? Well, the default value for the "group=" option in a ggplot statement is "group=1". So ggplot is connecting all the data points. We need to tell ggplot to *group* the observations by subject.

```
> print(ggplot(Oxboys, aes(age, height, group = Subject)) + geom_line())
```

Now, say we want to add a single regression line to the spaghetti plot of individual subject heights. A "geom_smooth()" layer accomplishes this. But, again, we have to be careful about defining the group layer. This code, adds a regression line for *each* subject.

```
> p <- ggplot(Oxboys, aes(age, height, group = Subject)) + geom_line()
> p + geom_smooth(aes(group = Subject), method="lm", se =F)
> print(p)
```

In this case, we *want* "group=1". (Notice I'm *setting* the color and size of the regression line outside the *aes() call*)

```
> p + geom_smooth(aes(group = 1), method="lm", se =F, color="red", lwd=2)
> print(p)
```

You can combine group observations with individual observation. Plot summaries (boxplots) of heights at each measurement date. (Note that by not specifying group, we are effectively defining "group=1"):

```
> p2 <- ggplot(Oxboys, aes(Occasion, height)) + geom_boxplot()
> print(p2)
```

Now, we can overlay the lines for each subject.

```
> print(p2 + geom_line(aes(group = Subject), colour = "#3366FF"))
```

4.5.2 About "geoms"

The geometries, or "geoms", define the actual plot. A "point" geometry defines a scatterplot, a "line" geometry a line graph, etc...

As noted above, each geometry has a default statistic and each statistic has a default geometry.

4.5.3 about "stats"

The "stat" or statistics element in ggplot2 refers, essentially, to how the variables summarized before being plotted. In the case of *stat="identity"*, the variables are untransformed and simply plotted as is.

Geometries have default stats. For example *geom_bar* bins data to crate bar charts, so the default stat is *"bin"*. This may not be the behavior you want. If, for example, you want to use *geom_bar* with data that is already summarized (e.g. say on a table object) you would have to specify "stat=identity" e.g.

```
ggplot(dat, aes(group, mean)) + geom_bar(stat = "identity")
```

Note that a "stat" returns a new dataset with new variables. You can use these new variables by placing them inside 2 sets of 2 dots, e.g. "..newstat.." For example, stat_bin() creates three new variables: "count" (the number of observations in each bin), "density" (the percentage of total/bar width), and "x" (the center of the bin). The help file for a "stat" gives information on the variables created. So, if you would like a histogram to present the density rather than the default count, you would write:

```
ggplot(diamonds, aes(carat))
+ geom_histogram(aes(y = ..density..), binwidth = 0.1)
```

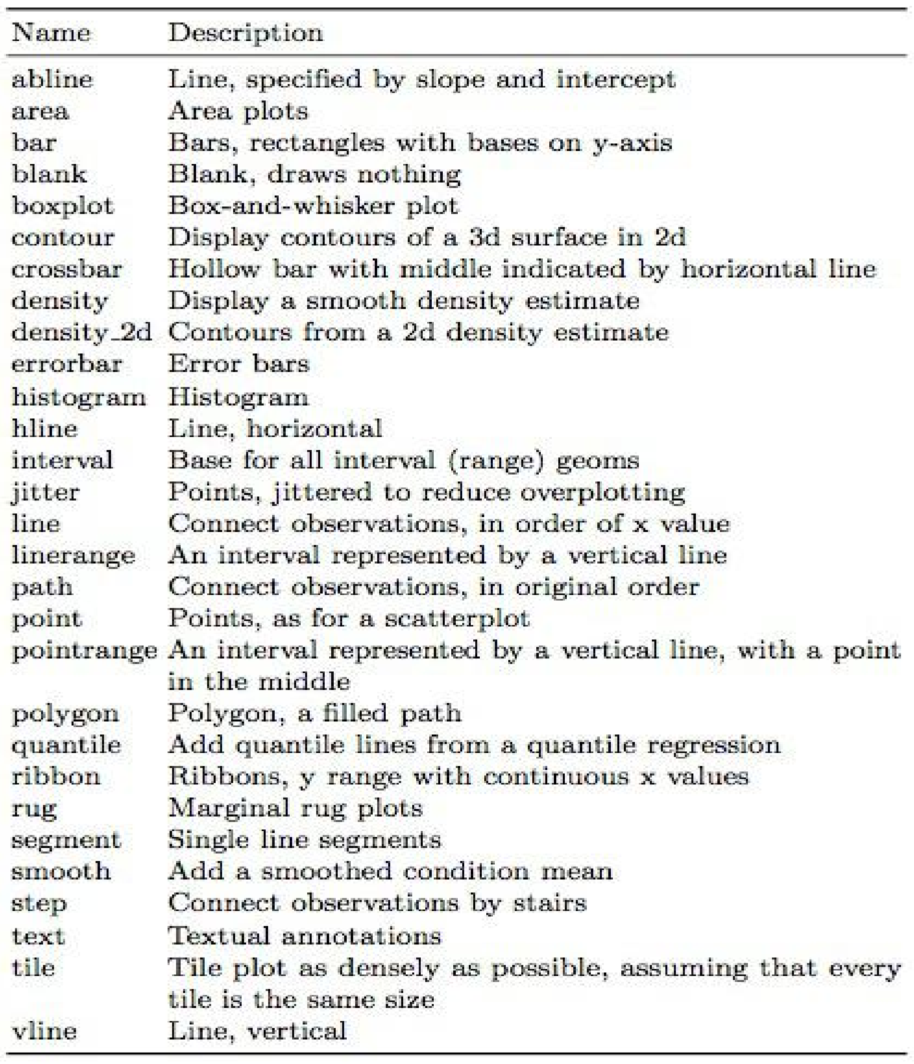
Or, with *qplot()*

```
qplot(carat, ..density.., data = diamonds, geom="histogram",
binwidth = 0.1)
```

Again, *stat_identity()* or specifying *stat="identity"* will allow you to plot pre-computed values, e.g. if you have a vector or list of rates or other statistics which you have already processed and just want to map them.

4.5.4 combining "geom"'s and "stats"

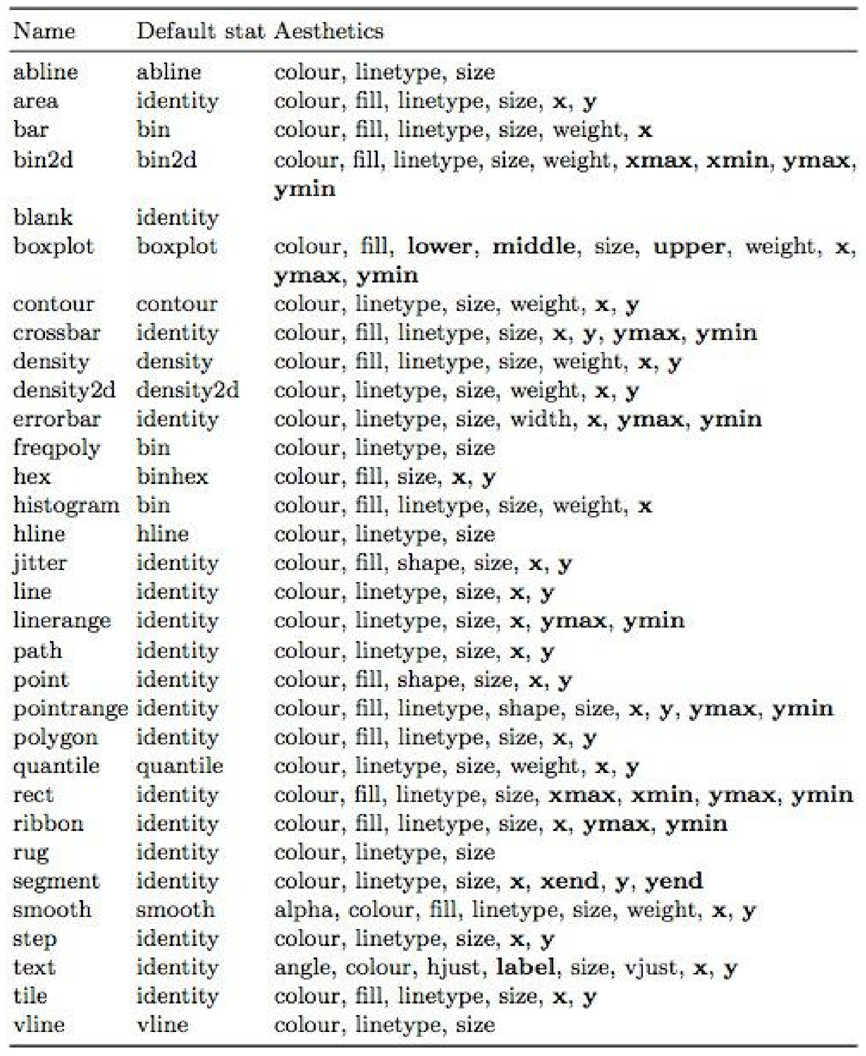
You can create new kinds of graphic images by combining non-default geom's and stats. You can, for example, create a frequency polygon by combining the "area" geometry with the "bin" stat. Start by defining the data and aesthetics (variable), limiting the x axis to between 1 and 3.



Name	Description
abline	Line, specified by slope and intercept
area	Area plots
bar	Bars, rectangles with bases on y-axis
blank	Blank, draws nothing
boxplot	Box-and-whisker plot
contour	Display contours of a 3d surface in 2d
crossbar	Hollow bar with middle indicated by horizontal line
density	Display a smooth density estimate
density_2d	Contours from a 2d density estimate
errorbar	Error bars
histogram	Histogram
hline	Line, horizontal
interval	Base for all interval (range) geoms
jitter	Points, jittered to reduce overplotting
line	Connect observations, in order of x value
linerange	An interval represented by a vertical line
path	Connect observations, in original order
point	Points, as for a scatterplot
pointrange	An interval represented by a vertical line, with a point in the middle
polygon	Polygon, a filled path
quantile	Add quantile lines from a quantile regression
ribbon	Ribbons, y range with continuous x values
rug	Marginal rug plots
segment	Single line segments
smooth	Add a smoothed condition mean
step	Connect observations by stairs
text	Textual annotations
tile	Tile plot as densely as possible, assuming that every tile is the same size
vline	Line, vertical

Table 4.2: Geoms in `ggplot2`

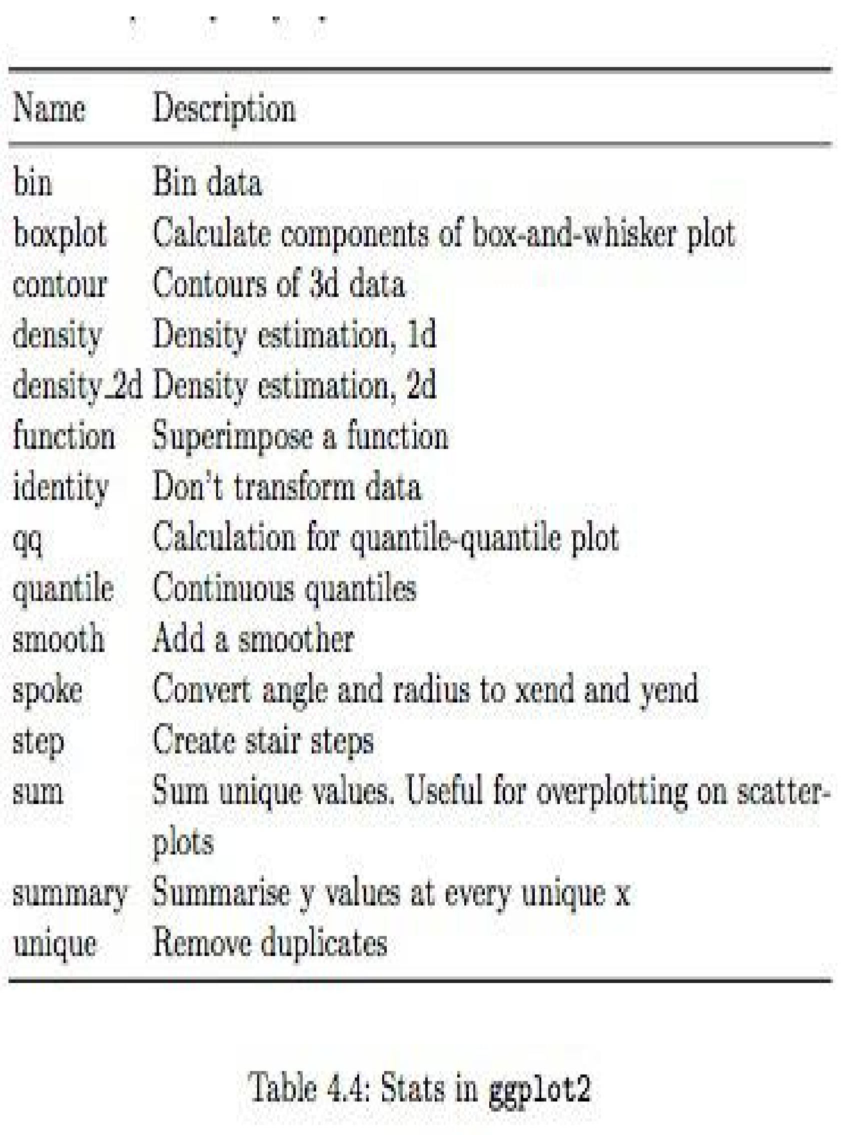
Figure 4.1: caption



Name	Default stat	Aesthetics
abline	abline	colour, linetype, size
area	identity	colour, fill, linetype, size, **x**, **y**
bar	bin	colour, fill, linetype, size, weight, **x**
bin2d	bin2d	colour, fill, linetype, size, weight, **xmax**, **xmin**, **ymax**, **ymin**
blank	identity	
boxplot	boxplot	colour, fill, **lower**, **middle**, size, **upper**, weight, **x**, **ymax**, **ymin**
contour	contour	colour, linetype, size, weight, **x**, **y**
crossbar	identity	colour, fill, linetype, size, **x**, **y**, **ymax**, **ymin**
density	density	colour, fill, linetype, size, weight, **x**, **y**
density2d	density2d	colour, linetype, size, weight, **x**, **y**
errorbar	identity	colour, linetype, size, width, **x**, **ymax**, **ymin**
freqpoly	bin	colour, linetype, size
hex	binhex	colour, fill, size, **x**, **y**
histogram	bin	colour, fill, linetype, size, weight, **x**
hline	hline	colour, linetype, size
jitter	identity	colour, fill, shape, size, **x**, **y**
line	identity	colour, linetype, size, **x**, **y**
linerange	identity	colour, linetype, size, **x**, **ymax**, **ymin**
path	identity	colour, linetype, size, **x**, **y**
point	identity	colour, fill, shape, size, **x**, **y**
pointrange	identity	colour, fill, linetype, shape, size, **x**, **y**, **ymax**, **ymin**
polygon	identity	colour, fill, linetype, size, **x**, **y**
quantile	quantile	colour, linetype, size, weight, **x**, **y**
rect	identity	colour, fill, linetype, size, **xmax**, **xmin**, **ymax**, **ymin**
ribbon	identity	colour, fill, linetype, size, **x**, **ymax**, **ymin**
rug	identity	colour, linetype, size
segment	identity	colour, linetype, size, **x**, **xend**, **y**, **yend**
smooth	smooth	alpha, colour, fill, linetype, size, weight, **x**, **y**
step	identity	colour, linetype, size, **x**, **y**
text	identity	angle, colour, hjust, **label**, size, vjust, **x**, **y**
tile	identity	colour, fill, linetype, size, **x**, **y**
vline	vline	colour, linetype, size

Table 4.3: Default statistics and aesthetics. Emboldened aesthetics are required.

Figure 4.2: caption



Name	Description
bin	Bin data
boxplot	Calculate components of box-and-whisker plot
contour	Contours of 3d data
density	Density estimation, 1d
density_2d	Density estimation, 2d
function	Superimpose a function
identity	Don't transform data
qq	Calculation for quantile-quantile plot
quantile	Continuous quantiles
smooth	Add a smoother
spoke	Convert angle and radius to xend and yend
step	Create stair steps
sum	Sum unique values. Useful for overplotting on scatter-plots
summary	Summarise y values at every unique x
unique	Remove duplicates

Table 4.4: Stats in `ggplot2`

Figure 4.3: caption

```
> d <- ggplot(dsmall, aes(carat)) + xlim(0, 3)
```

Now, combine the "bin" stat with the "area" geom to create a frequency polygon.

```
> print(d + stat_bin(aes(ymax = ..count..), binwidth = 0.1, geom = "area"))
```

Here, a scatterplot with both size and height mapped to frequency.

```
> d2<-d + stat_bin(
+ aes(size = ..density..), binwidth = 0.1,
+ geom = "point", position="identity"
+ )
> print(d2)
```

4.5.5 about "faceting"

We've already encountered faceting of stratifying by variables. There are two types of faceting, facet_wrap() which is 1 dimensional, and facet_grid, which is 2-dimensional.

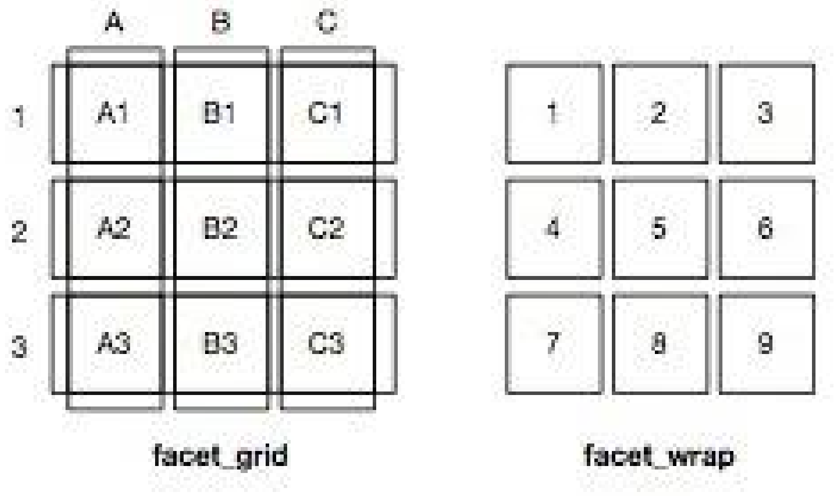


Figure 4.4: caption

In qplot(), a 2d faceting specification (e.g., x ỹ) will use facet_grid, while a 1d specification (e.g., x̃) will use facet_wrap.

To illustrate faceting, we first subset the mpg data set to three cylinders (4, 6, 8) and two types of drive train (4 and f).

```
> data(mpg)
> mpg2 <- subset(mpg, cyl != 5 & drv %in% c("4", "f"))
```

Use facet_grid, specify the rows and columns using the formula .̃ Here, we plot a single row with multiple columns stratified by cylinder.

```
> print(qplot(cty, hwy, data = mpg2) + facet_grid(. ~ cyl))
```

Now, look at the association of "cty" by "hwy" by combinations of "drv" and "cyl".

```
> print(qplot(cty, hwy, data = mpg2) + facet_grid(drv ~ cyl))
```

If you think of faceting as, essentially, a contingency table, it is useful to have the marginals. This is controlled with the "margins=T" argument. You can get more control over the margins by specifying the variable names, e.g. margins = c("sex", "age").

```
> print(qplot(cty, hwy, data = mpg2) + facet_grid(drv ~ cyl, margins=T))
```

Note that adding an additional component, like a regression line, reflects both facets in the margins.

```
> print(qplot(displ, hwy, data = mpg2)
+ + geom_smooth(aes(colour = drv), method = "lm", se = F)
+ + facet_grid(cyl ~ drv, margins=T))
```

By contrast, facet_wrap() creates a ribbon of plots stratified by a variable that may have many levels. Here we look at movie rating by decade.

```
> data(movies)
> library(plyr)
> movies$decade <- round_any(movies$year, 10, floor)
> qplot(rating, ..density.., data=subset(movies, decade > 1890),
+ geom="histogram", binwidth = 0.5) + facet_wrap(~ decade, ncol = 6)
```

The scales for the facets are controlled by the "scales=" parameter. They can vary from "fixed", where they are the same for each facet, to "free" where x and y are allowed to vary for each facet, with options for "free_x", and "free_y".

Fixed scales are useful to compare subsets on an equal scale. Free scales allow you to see more details. Free scales are particularly useful to display multiple times series that were measured on different scales.

Here we work again with the economics data. We first use the *melt()* function in the *reshape2* package (also written by Hadley Wickham) to change the data from "wide" to "long", stacking the variables to a single column.

```
> library(reshape2)
> data(economics)
> em <- melt(economics, id = "date")
```

Now we plot those variables across time.

```
> print(qplot(date, value, data = em, geom = "line", group = variable) +
+ facet_grid(variable ~ ., scale = "free_y"))
```

Faceting is nice, but is not useful in all cases. When you are interested in seeing small differences between groups, you might be better off going with grouping, which is invoked in the aesthetics.

4.6 misc

ggplot2 uses some of the same parameters as base graphics, e.g. "xlim", "ylim", "main", "xlab", "ylab". You can also choose to use the base graph names for aesthetics, e.g. "col", "cex", "pch", but the ggplot2 names ("color", "size", "shape") are probably easier to remember.

Note that because ggplot2 is not generic, you can't pass an object to it (like you can with plot()) and expect some default image.

Based on the work of folks like Tufte, I like a minimalist background. I can get this by appending the following code to a plot:

```
 + opts(panel.background = theme_blank())
```

Or, you can use:

```
+ opts(panel.background = theme_rect(colour = NA))
```

Or, you can use Finally, *theme_bw()* to remove the default grey background.

4.6.1 alpha blending for overplotting

Sometimes you have too many data points obscuring each other. The usual first approach to over plotting is to add some "jigger" with *geom_jigger()* For large data sets, you can also try the *alpha()* function for color. You specify a ratio where the denominator is the number of points that must be over plotted to give a solid color. [5]

```
> library(scales)
> df <- data.frame(x = rnorm(2000), y = rnorm(2000))
> norm <- ggplot(df, aes(x, y))
> print(norm + geom_point(colour = alpha("black", 1/3)))

> print(norm + geom_point(colour = alpha("black", 1/10)))
```

4.7 bubble plots

This code comes from Matt Maenner and illustrates how to use ggplot2 to create bubble plots.

```
> crime <- read.csv("http://www.columbia.edu/~cjd11/charles_dimaggio/DIRE
> /resources/R/crimeRatesByState2005.csv", header=TRUE)
> ggplot(crime, aes(x=murder, y=burglary, size=population, label=state),legend=FALSE)+
+ geom_point(colour="white", fill="red", shape=21)+ scale_area(range=c(1,25))+
+ scale_x_continuous(name="Murders per 1,000 population", limits=c(0,12))+
+ scale_y_continuous(name="Burglaries per 1,000 population", limits=c(0,1250))+
+ geom_text(size=4)+
+ theme_bw()
```

In this code, *scale_area()* automatically scales the bubbles to reflect differences in terms of areas (instead of radius), although the specified range of minimum and maximum sizes constrains the dimensions. The color specification in *geom_point()* produces red bubbles-without affecting the text color; shape 21 sets the outline of the shape to white, allowing the fill of the circle to be red. Finally, *theme_bw()* removes the default grey background.

4.8 mapping with ggplot

ggplot2 can also be use to map spatial data. I have some code and images here to illustrate the process.

[5] You used to be able to invoke *alpha()* directly from ggplot2, but after a recent revision you now have to explicitly call the *scales* package first.