Self-Study Exercises I
Epidemiologic Analysis Using R
Columbia University EPIC Summer 2014

C DiMaggio, with edits from S Mooney

June 3, 2014

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 working with objects

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.

1.1.1 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. [3].

[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.

[3]Again, try to name each element, using the same names you used for the noncase vector

Print each vector to your console (screen)

1.1.2 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?

1.1.3 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. [4]

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

[4]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.

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.[5]

```
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.

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
```

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

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.

1.2 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. [6] 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")
```

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

```
tab
```

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	174
Not Exposed	21	184

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

1.2.1 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.[7]

```
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**.

1.3 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.

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.test()* 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.

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

```
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]
```

1.3.1 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.[8] 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)
```

[8]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.

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.[9] Try the *sum()* function again, only this time provide *as.numeric(y)* as the argument.

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

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

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

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.

3.1 months example

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

```r
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="*

```r
`?`(factor)
```

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

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

Print out a frequency table of "f.mths"

What is the mode of ”f.mths”? What is its 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()*. [10]

```
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.

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 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”.

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

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.

4 exploring data sets

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. [11]

- 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).

```
infert[1:5, ]
```

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

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

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)
```

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. [12] 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.

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.

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

[12]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

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. [13]

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

```r
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.

```r
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.[14]

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

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

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.

```r
plot(infert$education)
```

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

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

[14]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.

```
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)
```

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?[15]
4. Plot a histogram of assault arrest rates in the United States.

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.

5.1 temperature conversion

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

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

Use the assignment operator to create a vector data object called "f" of Fahrenheit temperatures from 95 to 104, incrementing by 0.1[16] 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.[17] Search for help on cbind if you need to using *help(cbind)* or *?cbind*.

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?

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? [18]

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

5.4 cumulative risk

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

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

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

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

[17]extra credit if you can name the columns

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

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" [19]

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

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 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.

6 rates, risks, odds and logits

Epidemiology often involves measuring rates and risks.

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

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*

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

Second, we plot the log of the odds.

```r
curve(log(x/(1 - x)), 0, 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.

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. [20]
- Calculate the risk ratio[21] and odds ratio for these data. [22]
- Create a matrix of the risks, relative risks, odds and odds ratios

7 cross tabulations and stratified analysis

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.[23] 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?

[20]Hint: Use apply()

[21]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

[22]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

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

- 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":

$$OR_{CI} = e^{ln(OR) \pm 1.96\sqrt{\frac{1}{a}+\frac{1}{b}+\frac{1}{c}+\frac{1}{d}}} \tag{2}$$

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

7.1.1 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 [24]
- calculate the odds ratio for the ugdp data using the *epitab()* function. [25]

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

7.2.1 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

[24]library(epitools)

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

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.[26]

- 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. [27]
- 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?[28]
- If you want, test the statistical significance of the association of region and safety by using chisq.test() to calculate a chi square statistic.

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.

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

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

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

- 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*.[29]
- 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.[30] 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. [31]
  - 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
  ```

[29] 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...

[30] Again, if you're unsure about how to use a function, use help.

[31] This is why I like to use xtabs() to create these arrays. It automatically names the levels making indexing a little easier

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. [32]

- 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.

7.3.1 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?

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. [33]

8.1 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)
```

[32]Again, better that you try to work this out for yourself, but the answer is in the solutions section

[33]Much of this discussion comes from a very nice post on stack overflow

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.

```r
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):

```r
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.

```r
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.

```r
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.

```r
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*

8.2 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. [34]

```
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.

```
dat <- data.frame(age = (round(rnorm(100, 5, 1))),
    clinic = sample(c("a", "b", "c", "d", "e")), treatment = sample(c("Tx",
        "noTx")))
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)
```

8.3 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.[35] So having a way to operate directly on the elements of a list

[34]Note that I'm letting *data.frame()* recycle additional values for the clinic.
[35]*lapply()* lurks beneath many other functions in R

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.

8.4 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.

8.5 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.

```
v1 <- 1:5
v2 <- 6:10
v3 <- 11:15
v1
v2
```

```
v3
mapply(sum, v1, v2, v3)
```

8.6 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()*.

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.

9.1 indexing vectors

9.1.1 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
```

9.1.2 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
```

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.

9.2.1 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)]
```

9.2.2 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]
```

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
```

9.3.1 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
```

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.

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

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.[36]

- 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 [37]
- 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.

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

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

- 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.[38]
- 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: [39]

  ```
  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 [40]

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

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

[39]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.

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

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.