Lab 8

Public Health 241: Statistical Analysis of Categorical Data

In this lab, we'll cover some commands you'll need to complete Homework 9 and introduce you to some commonly used R commands and concepts not mentioned previously. We'll also use this opportunity to expose you to some more advanced R features that are beyond the requirements of this course's assignments.

1 Tests for Trend

Today, we'll begin by using the Western Collaborative Group Study data. Open the WCGS dataset from the class website, and again generate the `wtcat` variable:

```
wcgs <- read.dta("data/wcgs.dta")
```

```
## Warning in read.dta("data/wcgs.dta"): cannot read factor labels from Stata
## 5 files
```

```
wcgs <- wcgs %>% mutate(wtcat = ifelse(weight0 <= 150, 0,
                        ifelse(weight0 > 150 & weight0 <= 160, 1,
                        ifelse(weight0 > 160 & weight0 <= 170, 2,
                        ifelse(weight0 > 170 & weight0 <= 180, 3, 4)))))
```

We've been relying heavily on `table()` and `epitab()` so far for data that is easily summarized into a 2 x 2 table. However, these functions can also be used for 2 x *k* tables. To produce tables and statistics from such data structures, we'll need to use the tabulate and tabodds commands. First, create the 2 x *k* table using `table()`. You can then use `chisq.test()` to get the overall test of association.

```
chd_wtcat <- table(wcgs$chd69, wcgs$wtcat)
chd_wtcat
```

```
##
##       0   1   2   3   4
##   0 558 505 594 501 739
##   1  32  31  50  66  78
```

```
chisq.test(chd_wtcat)
```

```
##
##  Pearson's Chi-squared test
##
## data:  chd_wtcat
## X-squared = 21.357, df = 4, p-value = 0.000269
```

You can use `prop.trend.test()` to perform a test for trend. This function require two parameters: a vector of events and a vector of trials. We can do this by subsetting our 2 x *k* table. To get the vector of positive `chd69`, subset the table by `c(TRUE, FALSE)`. This works by selecting every other element as R simply repeats the subset parameter if the parameter vector length is less than the object vector length. Thus, this expression is a shortcut that expands to `[c(TRUE, FALSE, TRUE, FALSE, TRUE)]` for a object vector of length 5. Then, to get the vector of negative `chd69`, subset the table by `c(FALSE, TRUE)`. To get the total number of events, we add `chd_pos` and `chd_neg`.

```
chd_pos <- table(wcgs$chd69, wcgs$wtcat)[c(FALSE, TRUE)]
chd_neg <- table(wcgs$chd69, wcgs$wtcat)[c(TRUE, FALSE)]
chd_tot <- chd_neg + chd_pos
```

```
prop.trend.test(chd_pos, chd_tot) # test for trend
```

```
##
##  Chi-squared Test for Trend in Proportions
##
## data:  chd_pos out of chd_tot ,
##  using scores: 1 2 3 4 5
## X-squared = 15.361, df = 1, p-value = 8.88e-05
```

The goodness of fit test statistic and p-value can easily be deduced by subtraction and `pchisq()`:

```
goodness_of_fit <- 21.35 - 15.36
goodness_of_fit
```

```
## [1] 5.99
```

```
pchisq(goodness_of_fit, df = 3, lower.tail = FALSE)
```

```
## [1] 0.1120978
```

It is a very good idea to plot the estimated odds (or perhaps, even better, the log odds) here against the values of wtcat to see the pattern of the growth in risk. The next section will show you how to do this.

This approach can also be used for case-control data although the actual values of the odds here are not immediately interpretable due to the over-sampling of cases. However, the pattern in the log odds plot mentioned above is still meaningful and this is a good plot to examine in this case.

2 Additional Basic Commands and Concepts

2.1 Graphs and Charts

2.1.1 Scatter Plots

To graph 2 variables that exist in your dataset (let's call them x_var and y_var), we will be using a package called `ggplot2`. To begin constructing a ggplot, first feed the data to `ggplot()` and specify the variables you are interested in. Then, add feature functions to the base ggplot to visualize different trends in that dataset.

```
p <- ggplot(data, aes(x_var, y_var))
s <- p + geom_point()          # scatterplot

p <- ggplot(data, aes(x_var))
s <- p + geom_jitter()         # jitterplot
h <- p + geom_histogram()      # histogram
```

To plot the WCGS `wtcat` variable against the log odds of CHD, mentioned above, we'll first need to calculate the following probabilities:

$$P(\text{chd69} = 1|\text{wtcat} = k)\text{for each}k\text{where}k\text{is an integer from 0 to 4}$$

In other words, we need the conditional probability of being a case within each exposure level. To do this, we simply divide the number of events by number of trials for each strata. Using our `chd_pos` and `chd_tot` objects from earlier:

```
p <- chd_pos/chd_tot
p
```

```
## [1] 0.05423729 0.05783582 0.07763975 0.11640212 0.09547124
```

Be sure that you can see the connection between this new dataset and your summary table above.

Now, we can create variables representing the odds and log odds:

```r
odds <- p/(1-p)
odds
```

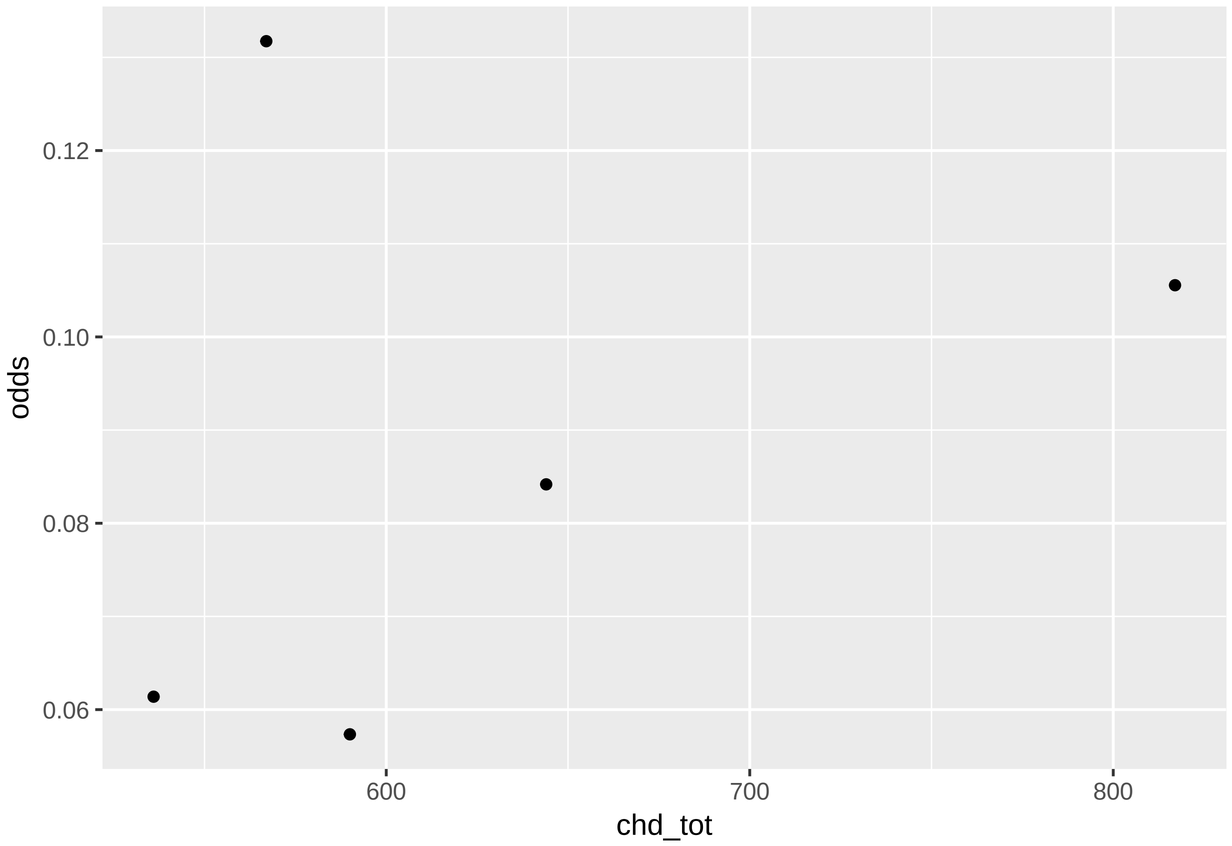
```
## [1] 0.05734767 0.06138614 0.08417508 0.13173653 0.10554804
```

```r
log_odds <- log(odds)
log_odds
```

```
## [1] -2.858623 -2.790571 -2.474856 -2.026951 -2.248589
```

Finally, let's create the two graphs mentioned in the last section.

```r
chd_odds <- cbind.data.frame(chd_tot, odds)
plotodds <- ggplot(chd_odds, aes(chd_tot, odds))
plotodds + geom_point()
```



The basic ggplot is very barebones. However, the beauty of R is that your plot is customizable in almost every way you can imagine. You can add labels and annotations, change the color or shape of your points, include a legend, etc. To see the full suite of capabilities, check out the full documentation online.

```r
plotodds <- ggplot(cbind.data.frame(chd_tot, log_odds), aes(chd_tot, log_odds))
plotodds <- plotodds + geom_point() +
                       labs(title = "CHD Log Odds", x = "Cases of CHD", y = "Log Odds")
```

2.1.2 Saving and Exporting Plots

To save your plot for use outside of your markdown file, you can use `ggsave()` for ggplots and other grid objects. The first argument to `ggsave()` is the desired file name, given as a file path relative to your current working directory; you can also specify the file type to be exported by the extension provided here (.pdf, .png, etc.). By default, `ggsave()` will save your most recently generated plot, but if this is not what you desire, you can specify the object to be saved with the `plot` parameter. To find out more about customizing the size of the plot, resolution, scale, etc., check the documentation here.

```
ggsave("chd_logodds.png")
ggsave("chd_odds.pdf", plot = plot_logodds)
```

2.1.3 Other Types of Graphs and Charts

If you wish to make other types of graphs and charts, you can do so by customizing your ggplot by adding additional functions. For example, for a histogram, you can append your ggplot with `geom_hist()`. This site lists the top 50 common ggplot visualizations including jitter plots, bubble plots, violin plots, etc.

3 Other Cool Tricks and Tips

3.1 Filtering Data

From this section of the lab forward, we'll be using the `animals2.dta` dataset, available on bCourses. This dataset is a little silly and contrived, but we'll be using it to demonstrate ways in which you can manipulate data in R. To limit the observations on which you perform calculations, you can first take a subset of your data using the `subset()` function. For example, to tabulate bunnies and cats across homes with a happiness level above 5 and at least 1 dog:

```
animals <- read.dta("data/animals2.dta")
homes_happy_dogs <- subset(animals, happiness > 5 & dogs != 0, select = c(bunnies, cats))
table(homes_happy_dogs$bunnies, homes_happy_dogs$cats)
```

```
##
##     0 1 2 3 4 5
##   0 2 3 1 1 0 1
##   1 0 0 1 1 2 1
##   2 0 1 0 0 0 0
##   3 1 0 0 0 0 0
```

Or if you only want to tabulate bunnies and cats across the first 5 observations, you can use the bracket operator to subset your data:

```
first_five <- animals[1:5,]
table(first_five$bunnies, first_five$cats)
```

```
##
##     0 1 4
##   0 2 1 0
##   1 0 0 1
##   2 0 1 0
```

3.2 Merging and Appending Data

You can merge existing datasets using joins. There are four options for joins: `left_join()`, `right_join()`, `inner_join()`, and `outer_join()`. Each option will give you a different result based on what variables you want to keep in your output if there happens to be an observation in dataset A in which there is no observation with a matching key in dataset B. Details here..

If you want to append observations to an existing dataset, you can use `rbind()` (short for row-bind), which takes two data frames as arguments. This appends the second dataset to the end of the first dataset. `rbind()` only works if the column names all match.

If you want to add additional variables of which you already have values for, you can use `cbind()` (short for column-bind). This requires the variable-to-be-appended to have the same number of observations as the existing dataset.

3.3 Data Types

Stata stores data values as either strings or numbers. There are different ways to store strings and different ways to store numbers. The describe command tells you the storage type for each variable in your dataset. You should know that numbers are designated by the following data types: byte, int, long, float, double. Strings are designated by these data types: str1, str2, . . . , str80. The number after str indicates the maximum number of characters in the string. The reason this information is relevant to even beginning Stata users is because anytime you want to refer to the value of a string variable, you will need to include quotation marks, whereas when you want to refer to the value of a numeric variable, you should not include quotation marks. For example:

Everything in R is an object. Fundamentally, these objects are broken down into *6 atomic types*: character, numeric (real or decimal), integer, logical, complex. From these atomic types, we can create atomic vectors, or vectors that hold only 1 type. To check what type of object we are working with, use `typeof()`. For example:

```r
x <- "This is a vector of character objects."
typeof(x)
```

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

When we combine many atomic data into one object (whether the data is one type or of different types), we call these *data structures*. There are many base data structures in R including atomic vectors, lists, matrices, data frames, factors. Data structures can also be user- and package-defined. Many times, errors in R will have to do with type mismatch and will require type conversion to resolve. To do so, there are functions that support conversion of atomic types to other atomic types. More details here. For example:

```r
two <- "2"
three <- two + 1 # error in two + 1 : non-numeric argument to binary operator
three <- as.numeric(two) + 1 # no error
```

3.4 Cleaning Messy Data

It's difficult to provide one concise method for cleaning data since data can be "messy" for a large variety of reasons. One common problem that data has, though, is inconsistent coding.

For example, you might want to group all observations together for people who live in an apartment (versus house), but the answers to your survey (which are now in your dataset) use different strings of characters to designate "apartment." For example, the variable "housing" in the `animals2.dta` dataset takes on several different following values:

```
table(animals$housing)
```

```
##
##      apartment      Apartment apartment unit            apt   apt building
##              4              3              2              7              1
##           Apt.          house
##              4             19
```

We can create a variable called housing_clean that takes only 2 values – “apartment” and “house” – by doing the following:

```
animals <- mutate(animals, housing_clean = ifelse(housing == "Apartment" |
                                                  housing == "Apt." |
                                                  housing == "apartment unit" |
                                                  housing == "apt" |
                                                  housing == "apt building" |
                                                  housing == "apartment", "apartment", "house"))
```

Typing `table(housing_clean)` will verify that housing_clean is as we want it. Note that since R is case-sensitive, it considers “Apartment” (with a capital ’A’) to be a different value as compared to “apartment.”

You’ll notice that the above method of creating a housing_clean variable necessitated specifying all of the “incorrect” values of the variable housing. Doing this can be incredibly tedious and time-consuming because there are sometimes tons of “incorrect” values. Fortunately, there are more automated ways to do the same thing using R’s string manipulation functions. For example, you can filter by entries in which the first two letters are “ap” using the `str_detect()` function from the `stringr` package and regular expressions (regex). You can use the `ignore_case = TRUE` option to do a case-insensitive match.

```
bananas <- c("banana", "Banana", "BANANA")
str_detect(bananas, "banana")
```

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

```
#> [1]  TRUE FALSE FALSE
str_detect(bananas, regex("banana", ignore_case = TRUE))
```

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

```
#> [1] TRUE TRUE TRUE
```

```
animals <- mutate(animals, housing_clean2 = ifelse(str_detect(animals$housing,
                                                       regex("ap.*", ignore_case = TRUE)), "apar
```

3.5 Importing/Exporting to Different File Types

For the most part, we’ve been importing data from .dta files with `read.dta()`. However, R can support many different file types including csv and xls. Simply search online for a package that supports importing your desired data type and go from there. For example, to import csv’s, we use `read.csv()` from the base utils package.

When exporting, for example by using `ggsave()`, you can also change the export filetype by changing the file path extension. Most file types are supported.

3.6 Loops

If you're interested, search online for R documentation on for-loops and while-loops and ask your GSI if you have any questions. These techniques can be used to make your workflows more efficient.