# Class Notes for “Getting and Cleaning Data” – NOT PROJECT NOTES

# Getting and Cleaning Data Notes

RAW DATA is the original untouched data. Might be pressure samples that are averaged to report blood pressure, might be raw images from a gene sequencing machine.

## How do you know you have raw data?

* You ran no software on the data
* Did not manipulate any of the numbers in the data
* Did not remove any data from the dataset
* Did not summarize the data in any way

## Tidy Data:

* Each variable should be in one column
* Each different observation should be in a different row
* There should be one table for each kind of variable
* If there are multiple tables, they should include a column that allows them to be linked.
* Include a row at the top of each file with variable names
* Make the variable names human-readable
* Save data in one file per table.

## The Code Book

* Information about the variables including units that are not in the data file.
* Information about the summary choices you made.
* Information about the experimental design.

Commonly this is a text or Word-format file.

Include a section “Study Design” that thoroughly describes how you collected the data.

Include a section “Code Book” that describes each variable and its units.

## The Instruction List

A computer script in some language like R

Input for the script is the raw data; output is the processed tidy data.

NO PARAMETERS to the SCRIPT!

If you can’t automate everything, include explicit instructions for manual steps, including details like versions of software tools used to process the data.

## DOWNLOADING FILES

Use setwd() and getwd()

Remember relative vs. absolute paths (./data vs. /home/ehemdal/data)

Remember to double backslashes on Windows: setwd(“C:\\git\\datascience\\data”)

You can check to see if a file exists and create subdirectories:

if (!file.exists(“data”)) {

dir.create(“data”)

}

## Using download.file()

Downloads a file from the Internet. Parameters are:

url, destfile, method. Useful for tab-delimited, CSV, and other files.

The download.file() method works with HTTP and should work with HTTPS on Windows. Record the date downloaded with date().

# Reading Local Files

Look into unzip() to extract ZIP files.

Use read.table(): Parameters file, header, sep, row.names, nrows

Related functions are read.csv() and read.csv2()

The read.table() function defaults to reading a tab-delimited file. If you have a CSV, then use sep = “,”. Header=TRUE tells R that the first line is the names of the columns.

The read.csv() function works like read.table() with these defaults set.

Other parameters:

quote=”’” Data has single quotes in it. Try quote = “” if trouble reading.

Use na.strings to set the character(s) that mean “no data”. Could be “NA”, or “-1” or “99999” or whatever.

Use nrows to specify how many rows to read (might test with only a few rows).

Skip lines with skip = n.

# Reading Excel Files

Use the xlsx package – library(xlsx)

Then read.xlsx() arguments are filename first, sheetIndex = n (which sheet to read) header = TRUE (there’s a header row)

colIndex and rowIndex take vectors of column and row numbers so you can read only a subset of the columns and rows in the spreadsheet.

The package read.xlsx2() is faster than read.xlsx() but might be unstable.

You can write an Excel file with write.xlsx function --- similar arguments.

For heavy-duty Excel work, try XLConnect package.

Recommendation is flat files or tab-separated or CSV files; not everyone has an appropriate version of Excel.

# Examples of using dplyr package

These are from the swirl tutorial:

# Compute four values, in the following order, from

# the grouped data:

#

# 1. count = n()

# 2. unique = n\_distinct(ip\_id)

# 3. countries = n\_distinct(country)

# 4. avg\_bytes = mean(size)

#

# A few thing to be careful of:

#

# 1. Separate arguments by commas

# 2. Make sure you have a closing parenthesis

# 3. Check your spelling!

# 4. Store the result in pack\_sum (for 'package summary')

#

# You should also take a look at ?n and ?n\_distinct, so

# that you really understand what is going on.

pack\_sum <- summarize(by\_package,

count = n() ,

unique = n\_distinct(ip\_id),

countries = n\_distinct(country),

avg\_bytes = mean(size))

Now a little more:

# Don't change any of the code below. Just type submit()

# when you think you understand it.

# We've already done this part, but we're repeating it

# here for clarity.

by\_package <- group\_by(cran, package)

pack\_sum <- summarize(by\_package,

count = n(),

unique = n\_distinct(ip\_id),

countries = n\_distinct(country),

avg\_bytes = mean(size))

# Here's the new bit, but using the same approach we've

# been using this whole time.

top\_countries <- filter(pack\_sum, countries > 60)

result1 <- arrange(top\_countries, desc(countries), avg\_bytes)

# Print the results to the console.

print(result1)

Now do everything in one go, this is ugly:

# Don't change any of the code below. Just type submit()

# when you think you understand it. If you find it

# confusing, you're absolutely right!

result2 <-

arrange(

filter(

summarize(

group\_by(cran,

package

),

count = n(),

unique = n\_distinct(ip\_id),

countries = n\_distinct(country),

avg\_bytes = mean(size)

),

countries > 60

),

desc(countries),

avg\_bytes

)

print(result2)

Finally, with chaining operator: “then…” %>%

# Read the code below, but don't change anything. As

# you read it, you can pronounce the %>% operator as

# the word 'then'.

#

# Type submit() when you think you understand

# everything here.

result3 <-

cran %>%

group\_by(package) %>%

summarize(count = n(),

unique = n\_distinct(ip\_id),

countries = n\_distinct(country),

avg\_bytes = mean(size)

) %>%

filter(countries > 60) %>%

arrange(desc(countries), avg\_bytes)

# Print result to console

print(result3)

Final bit, pulls a lot of stuff together about how to use dplyr package.

cran %>%

select(ip\_id, country, package, size) %>%

mutate(size\_mb = size / 2^20) %>%

filter(size\_mb <= 0.5) %>%

arrange(desc(size\_mb)) %>%

print()

Messy Data:

> students

grade male female

1 A 1 5

2 B 5 0

3 C 5 2

4 D 5 5

5 E 7 4

gather(students, sex, count, -grade)

grade sex count

1 A male 1

2 B male 5

3 C male 5

4 D male 5

5 E male 7

6 A female 5

7 B female 0

8 C female 2

9 D female 5

10 E female 4

| |======================================= | 34%

| Print res to the console to see what we accomplished.

> res

grade sex\_class count

1 A male\_1 3

2 B male\_1 6

3 C male\_1 7

4 D male\_1 4

5 E male\_1 1

6 A female\_1 4

7 B female\_1 4

8 C female\_1 4

9 D female\_1 0

10 E female\_1 1

11 A male\_2 3

12 B male\_2 3

13 C male\_2 3

14 D male\_2 8

15 E male\_2 2

16 A female\_2 4

17 B female\_2 5

18 C female\_2 8

19 D female\_2 1

20 E female\_2 7

| You are amazing!

separate(data = res, col = sex\_class, into = c("sex","class"))

grade sex class count

1 A male 1 3

2 B male 1 6

3 C male 1 7

4 D male 1 4

5 E male 1 1

6 A female 1 4

7 B female 1 4

8 C female 1 4

9 D female 1 0

10 E female 1 1

11 A male 2 3

12 B male 2 3

13 C male 2 3

14 D male 2 8

15 E male 2 2

16 A female 2 4

17 B female 2 5

18 C female 2 8

19 D female 2 1

20 E female 2 7

| You are really on a roll!

| |============================================= | 40%

| Conveniently, separate() was able to figure out on its own how to separate the sex\_class column. Unless you request

| otherwise with the 'sep' argument, it splits on non-alphanumeric values. In other words, it assumes that the values are

| separated by something other than a letter or number (in this case, an underscore.)

Or, simpler with %>%:

students2 %>%

gather(sex\_class ,count ,-grade ) %>%

separate(sex\_class, c("sex", "class")) %>%

print

With data like this:

name test class grade

1 Sally midterm class1 A

2 Sally final class1 C

3 Brian midterm class1 B

4 Brian final class1 B

5 Jeff midterm class2 D

6 Jeff final class2 E

7 Roger midterm class2 C

8 Roger final class2 A

Code:

spread( test, grade) %>%

Gives:

name class final midterm

1 Brian class1 B B

2 Brian class5 C A

3 Jeff class2 E D

4 Jeff class4 C A

5 Karen class3 C C

6 Karen class4 A A

7 Roger class2 A C

In words, take the values in the “key” column (test in the example) and use them as columns (like midterm, final). Fill the new columns with data from the original “value” column (grade in the example.