Preparing Data for Analysis

Last Updated 2017-09-04 14:14:16

Contents

PMA5 Ch 3 notes

Reproducible Research

- You are your own collaborator 6 months from now. Make sure you will be able to understand what you were doing.
- Investing the time to do things clearly and in a reproducible manner will make your future self happy.
- Comment your code with explanations and instructions.
  - How did you get from point A to B?
  - Why did you recode this variable in this manner?
- We need to record those steps (not just for posterity).
- This means your code must be saved in a script file.
  - Include sufficient notes to yourself describing what you are doing and why.
  - For R, this can be in a `.R` or `.RMD` file. I always prefer the latter.
  - SPSS users can open a text file or something similar. At each step be sure to copy the code from the program into this document.

Figure Credits: Roger Peng

Data Management

Questions to ask yourself (and the data) while preparing a data management file.

1. Do you need to code out missing data?
2. Do you need to code out skip patterns?
3. Do you need to make response codes more logical?
4. Do you need to recode categorical variables to quantitative?
5. Do you need to create secondary variables?

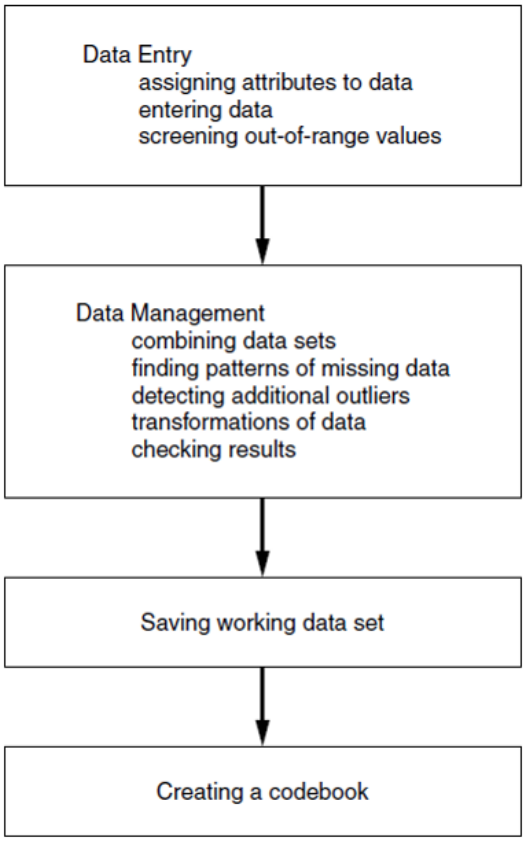


Figure 3.1: *Preparing Data for Statistical Analysis*

Figure 1:

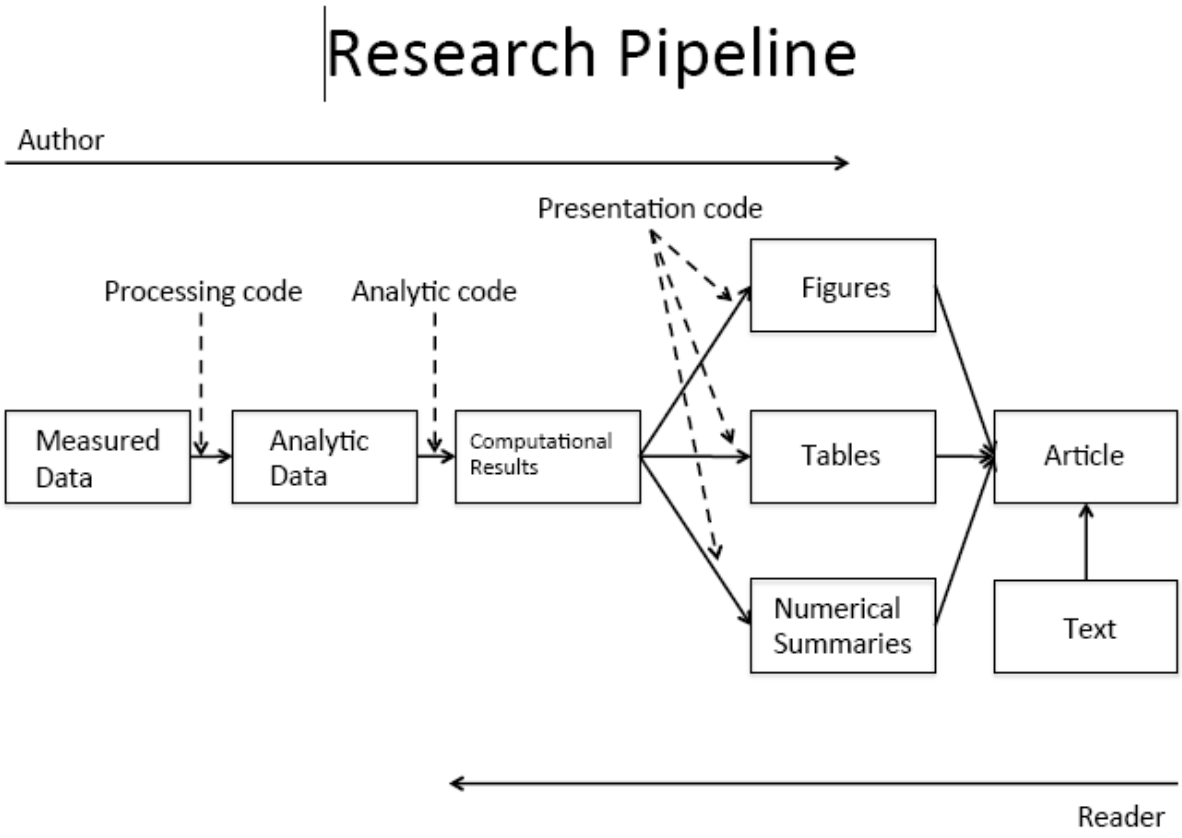

Research Pipeline
Author
Presentation code
Processing code
Analytic code
Figures
Measured Data
Analytic Data
Computational Results
Tables
Article
Numerical Summaries
Text
Reader

Figure 2:

Clean depression data

If you are using SPSS, be sure to save the code for each **successfull** step into a script file. R users should be following along using R Markdown.

Import data

```
library(ggplot2)
depress <- read.table("https://norcalbiostat.netlify.com/data/Depress.txt",
                      sep="\t", header=TRUE)
```

(*Read the data from your local machine, don't use the URL above. It'll ping my server to death*)

LOOK AT YOUR DATA

The absolute first thing you should do is to look at your raw data table. The `str` function is short for *structure*. This shows you the variable names, what data types R thinks each variable are, and some of the raw data. You can also use the `view()` function to open the data as a simliar spreadsheet format, or `head()` to see the top 6 rows of the data. The latter is sometimes less than helpful for a very large data set.

```
str(depress)
```

```
## 'data.frame':    294 obs. of  37 variables:
##  $ ID      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ SEX     : int  2 1 2 2 2 1 2 1 2 1 ...
##  $ AGE     : int  68 58 45 50 33 24 58 22 47 30 ...
##  $ MARITAL : int  5 3 2 3 4 2 2 1 2 2 ...
##  $ EDUCAT  : int  2 4 3 3 3 3 2 3 3 2 ...
##  $ EMPLOY  : int  4 1 1 3 1 1 5 1 4 1 ...
##  $ INCOME  : int  4 15 28 9 35 11 11 9 23 35 ...
##  $ RELIG   : int  1 1 1 1 1 1 1 1 2 4 ...
##  $ C1      : int  0 0 0 0 0 0 2 0 0 0 ...
##  $ C2      : int  0 0 0 0 0 0 1 1 1 0 ...
##  $ C3      : int  0 1 0 0 0 0 1 2 1 0 ...
##  $ C4      : int  0 0 0 0 0 0 2 0 0 0 ...
##  $ C5      : int  0 0 1 1 0 0 1 2 0 0 ...
##  $ C6      : int  0 0 0 1 0 0 0 1 3 0 ...
##  $ C7      : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ C8      : int  0 0 0 3 3 0 2 0 0 0 ...
##  $ C9      : int  0 0 0 0 3 1 2 0 0 0 ...
##  $ C10     : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ C11     : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ C12     : int  0 1 0 0 0 1 0 0 3 0 ...
##  $ C13     : int  0 0 0 0 0 2 0 0 0 0 ...
##  $ C14     : int  0 0 1 0 0 0 0 0 3 0 ...
##  $ C15     : int  0 1 1 0 0 0 3 0 2 0 ...
##  $ C16     : int  0 0 1 0 0 2 0 1 3 0 ...
##  $ C17     : int  0 1 0 0 0 1 0 1 0 0 ...
##  $ C18     : int  0 0 0 0 0 0 0 1 0 0 ...
##  $ C19     : int  0 0 0 0 0 0 0 1 0 0 ...
##  $ C20     : int  0 0 0 0 0 0 1 0 0 0 ...
##  $ CESD    : int  0 4 4 5 6 7 15 10 16 0 ...
```

```
##  $ CASES   : int  0 0 0 0 0 0 0 0 1 0 ...
##  $ DRINK   : int  2 1 1 2 1 1 2 2 1 1 ...
##  $ HEALTH  : int  2 1 2 1 1 1 3 1 4 1 ...
##  $ REGDOC  : int  1 1 1 1 1 1 1 2 1 1 ...
##  $ TREAT   : int  1 1 1 2 1 1 1 2 1 2 ...
##  $ BEDDAYS : int  0 0 0 0 1 0 0 0 1 0 ...
##  $ ACUTEILL: int  0 0 0 0 1 1 1 1 0 0 ...
##  $ CHRONILL: int  1 1 0 1 0 1 1 0 1 0 ...
```

Right away this tells me that **R** thinks all variables are numeric integers, not categorical variables. This will have to be changed. We'll get to that in a moment.

Missing data (DM Step 1)

In Excel, missing data can show up as a blank cell. In SPSS it is represented as a `.` period. R displays missing data as `NA` values.

Missing Data in SPSS: https://stats.idre.ucla.edu/spss/modules/missing-data/

Why would data be missing? Other than the obvious data entry errors, tech glitches or just non-cooperative plants or people, sometimes values are out of range and you would rather delete them than change their value (data edit).

Lets look at the religion variable in the depression data set.

```
table(depress$RELIG, useNA="always")
```

```
##
##    1    2    3    4    6 <NA>
##  155   51   30   56    2    0
```

Looking at the codebook, there is no category `6` for religion. Let's change all values to `NA`.

```
depress$RELIG[depress$RELIG==6] <- NA
```

This code says take all rows where RELIG is equal to 6, and change them to `NA`.

Confirm recode.

```
table(depress$RELIG, useNA="always")
```

```
##
##    1    2    3    4 <NA>
##  155   51   30   56    2
```

Notice the use of the `useNA="always"` argument. If we just looked at the base table without this argument, we would have never known there was missing data!

```
table(depress$RELIG)
```

```
##
##   1   2   3   4
## 155  51  30  56
```

What about continuous variables? Well there happens to be no other missing data in this data set, so let's make up a set of 7 data points stored in a variable named `y`.

```
y <- c(1, 2, 3, NA, 4, NA, 6)
y
```

```
## [1]  1  2  3 NA  4 NA  6
```

The #1 way to identify missing data in a continuous variable is by looking at the `summary()` values.

```
mean(y)
```

```
## [1] NA
```

```
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##     1.0     2.0     3.0     3.2     4.0     6.0       2
```

```
mean(y, na.rm=TRUE)
```

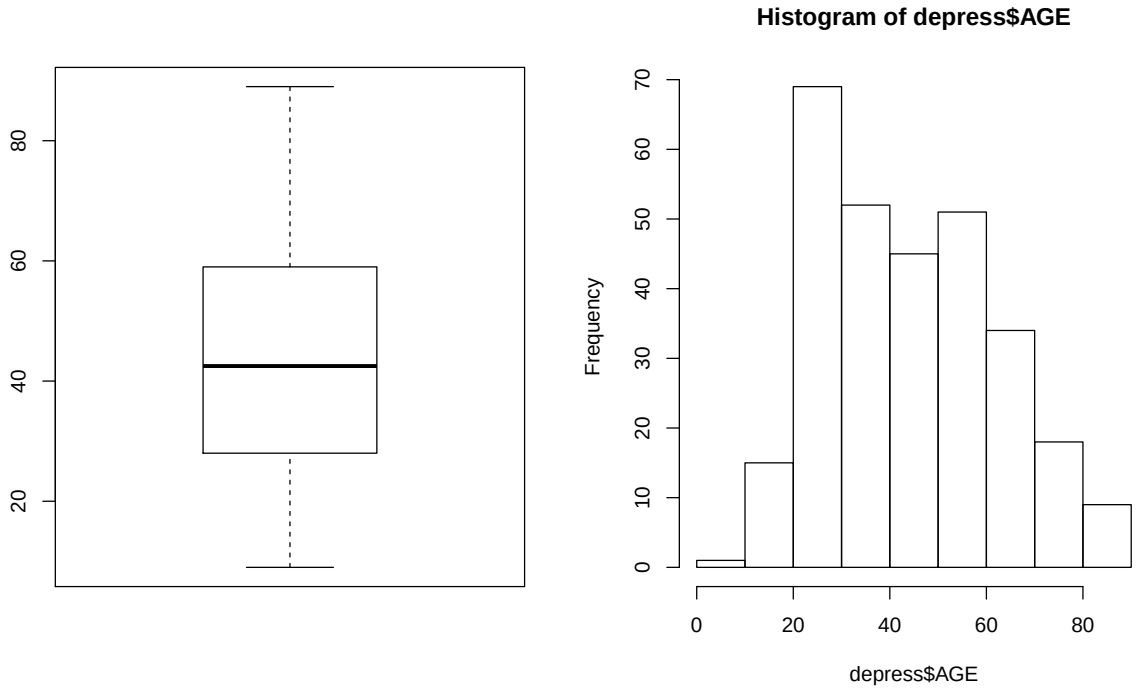
```
## [1] 3.2
```

In R, any arithmetic function (like addition, multipliation) on missing data results in a missing value. The `na.rm=TRUE` toggle tells R to calculate the *complete case* mean. This is a biased measure of the mean, but missing data is a topic worthy of it's own course.

Detecting and recoding outliers and/or errors

Let's look at the age variable in the depression data set.

```
par(mfrow=c(1,2)) # sets the grahpics grid 1 row by 2 columns
boxplot(depress$AGE)
hist(depress$AGE)
```



Just looking at the data graphically raises no red flags. The boxplot shows no outlying values and the histogram does not look wildly skewed. This is where knowledge about the data set is essential. The codebook does not provide a valid range for the data, but the description of the data starting on page 3 in the textbook clarifies that this data set is on adults. In the research world, this specifies 18 years or older.

Now look back at the graphics. See anything odd? It appears as if the data go pretty far below 20, possibly below 18. Let's check the numerical summary to get more details.

```
summary(depress$AGE)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    9.00   28.00   42.50   44.38   59.00   89.00
```

The minimum value is a 9, which is outside the range of valid values for this variable. This is where you, as a statistician, data analyst or researcher goes back to the PI and asks for advice. Should this data be set to missing, or edited in a way that changes this data point into a valid piece of data.

As an example of a common data entry error, and for demonstration purposes, I went in and changed a 19 to a 9. So the correct thing to do here is to change that 9, back to a 19. This is a very good use of the `ifelse()` function.

```
depress$AGE <- ifelse(depress$AGE==9, 19, depress$AGE)
```

The logical statement is `depress$AGE==9`. Wherever this is true, replace the value of `depress$AGE` with 19, wherever this is false then keep the value of `depress$AGE` unchanged (by "replacing" the new value with the same old value).

Confirm the recode.

```
summary(depress$AGE)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   18.00   28.00   42.50   44.41   59.00   89.00
```

Looks like it worked.

Identifying variable types (and fixing them) (Similar to DM #4)

- Consider the variable that measures marital status What data type does the codebook say this variable is?
- What data type does R see this variable as?

```
table(depress$MARITAL)
```

```
##
##   1   2   3   4   5
##  73 127  43  13  38
```

```
str(depress$MARITAL)
```

```
##  int [1:294] 5 3 2 3 4 2 2 1 2 2 ...
```

```
is(depress$MARITAL)
```

```
## [1] "integer"             "numeric"             "vector"
## [4] "data.frameRowLabels"
```

When variables have numerical levels it is necessary to ensure that R knows it is a factor variable.
The following code uses the `factor()` function to take the marital status variable and convert it into a factor variable with specified labels that match the codebook.

```
depress$MARITAL <- factor(depress$MARITAL,
                          labels = c("Never Married", "Married", "Divorced", "Separated", "Widowed"))
```

You should always confirm the recode worked. If it did not you will have to re-read in the raw data set again since the variable `MARITAL` was replaced.

```
table(depress$MARITAL)
```

```
##
## Never Married       Married      Divorced     Separated       Widowed
##            73           127            43            13            38
```

```
is(depress$MARITAL)
```

```
## [1] "factor"              "integer"             "oldClass"
## [4] "numeric"             "vector"              "data.frameRowLabels"
```

Creating secondary variables

Collapsing variables into fewer categories

For unbiased and accurate results of a statistical analysis, sufficient data has to be present. Often times once you start slicing and dicing the data to only look at certain groups, or if you are interested in the behavior of certain variables across levels of another variable, sometimes you start to run into small sample size problems.

For example, consider marital status again. There are only 13 people who report being separated. This could potentially be too small of a group size for valid statistical analysis.

One way to deal with insufficient data within a certain category is to collapse categories. The following code uses the `recode()` function from the `car` package to create a new variable that I am calling `MARITAL2` that combines the `Divorced` and `Separated` levels.

```
library(car)
MARITAL2 <- recode(depress$MARITAL, "'Divorced' = 'Sep/Div'; 'Separated' = 'Sep/Div'")
```

Always confirm your recodes.

```
table(depress$MARITAL, MARITAL2, useNA="always")
```

```
##                MARITAL2
##                 Married Never Married Sep/Div Widowed <NA>
##   Never Married       0            73       0       0    0
##   Married           127             0       0       0    0
##   Divorced            0             0      43       0    0
##   Separated           0             0      13       0    0
##   Widowed             0             0       0      38    0
##   <NA>                0             0       0       0    0
```

This confirms that records where `MARITAL` (rows) is `Divorced` or `Separated` have the value of `Sep/Div` for `MARITAL2` (columns). And that no missing data crept up in the process. Now I can drop the temporary `MARITAL2` variable and actually fix `MARITAL`. (keeping it clean)

```
depress$MARITAL <- recode(depress$MARITAL, "'Divorced' = 'Sep/Div'; 'Separated' = 'Sep/Div'")
rm(MARITAL2)
```

Binning a continuous variable into categorical ranges.

Let's create a new variable that categorizes income into the following ranges: <30, [30, 40), [40,50), [50, 60), 60+.
The easiest way is to use the `cut2` function in the package `Hmisc`. Note you don't have to load the package

fully to use a function from within that package. Useful for times when you only need to use a function once or twice.

```
depress$inc_cut <- Hmisc::cut2(depress$INCOME, cuts=c(30,40,50,60))
table(depress$inc_cut)
```

```
##
## [ 2,30) [30,40) [40,50) [50,60) [60,65]
##     231      28      16       9      10
```

Dichotomizing

Dichotomous variables tend to be binary indicator variables where a code of `1` is the level you're interested in.

For example, gender is coded as 2=Female and 1=Male. This is in the right direction but it needs to be 0/1.

```
depress$SEX <- depress$SEX -1
table(depress$SEX)
```

```
##
##   0   1
## 111 183
```

0/1 binary coding is mandatory for many analyses. One simple reason is that now you can calculate the mean and interpret it as a proportion.

```
mean(depress$SEX)
```

```
## [1] 0.622449
```

62% of individuals in this data set are female.

Sometimes the data is recorded as 1/2 (Yes/No), so just subtracting from 1 doesn't create a positive indicator of the variable. For example, `DRINK=1` if they are a regular drinker, and `DRINK=2` if they are not. We want not drinking to be coded as `0`, not `2`.

```
table(depress$DRINK)
```

```
##
##   1   2
## 234  60
```

The `ifelse()` function says that if `depress$DRINK` has a value equal to 2 `==2`, then change the value to 0. Otherwise leave it alone.

```
depress$DRINK <- ifelse(depress$DRINK==2, 0, depress$DRINK)
table(depress$DRINK)
```

```
##
##   0   1
##  60 234
```

Sum or Average values across multiple variables

The Center for Epidemiologic Studies Depression Scale (CESD) is series of questions asked to a person to measure their level of depression. `CESD` is calculated as the sum of all 20 component variables, and is already on this data set. Let's create a new variable named `sleep` as subscale for sleep quality by adding up question numbers 5, 11, and 19.

Reference: http://cesd-r.com/cesdr/

```
depress$sleep <- depress$C5 + depress$C11 + depress$C19
## depress <- depress %>% mutate(sleep = C5+C11+C19) # Not run. dplyr example
```

```
summary(depress$sleep)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   1.000   1.167   2.000   7.000
```

Renaming varible names for sanity sake

Turn all variable names to lower case. This is especially frustrating for R and STATA users where syntax is case sensitive.

```
names(depress) <- tolower(names(depress))
```

Saving your changes

You've just made a ton of changes!

- Save or export the new data set to your computer.
- Edit the codebook to reflect the changes that you made. Save this codebook with todays date as well.
- Keep the data, codebook and data management file in the same folder.

The `Sys.Date()` function takes the current date from your computer. The value is then formatted nicely for human consumption and added (pasted) to the file name before written to the working directory as a new text file.

```
date <- format(Sys.Date(), "%m%d%y")
filename <- paste("depress_", date, ".txt", sep="")
write.table(depress, filename, sep="\t", row.names=FALSE)
```