Comments & guidance on computations for Unit 1 paper

Stats/DataSci 485, Winter 2021

Version of Mon 25 January, 2021; 18PM

Statistical Reproducibility Appendix

R markdown

R markdown files…

- record code and comments in manner designed for readility by both people and computers
- have a header at the top (in YAML) for title, author, date and metadata relevant to processing and formatting that script
- go on to interleave chunks of markdown with "fenced blocks" of R code
- have the file extension = `.Rmd`
- are easy to generate in RStudio: In new file dropdown at upper left of the RStudio window, select "R Notebook" or "R markdown…"
- are the **only** accepted format for assignments with "reproducibility appendix" in the title.

Style guidelines for code scripts in Stats 485

Structure of code script

Overall please adhere to:

1. Header information and metadata for rendering, as YAML block
2. Your R programs' external requirements, as R code block
3. Overview of goals of computations to follow, as markdown block
4. Code and computations for statistical calculations, organized using markdown sections

Do include each of these parts, beginning with (1) and with (1)–(3) preceding (4).

Code script header

- The YAML block
  - Make sure yours has an `author:` line (as well as `title:`, …)
  - Put your name there.
- R block(s) for R program's external requirements
  a. Any needed `library(foo)` statements. **All** package dependencies stated at top of file.
  b. Reading in of data, ideally from public URLs. **All** necessary data read-ins go here.
  c. Examples:

```
library(brglm)
econ_mobility <-
  read.csv("http://dept.stat.lsa.umich.edu/~bbh/s485/data/mobility0.csv")
americannationsdata <- read.csv("https://tinyurl.com/yb65jsq2")
```

```
positivity_fa20 <-
  read.csv("http://dept.stat.lsa.umich.edu/~bbh/s485/data/covidTestsFA2020.csv")
```

(For the full/actual URL of the American Nations spreadsheet, see README sheet of that Google Sheets file.)

The general principle behind the external requirements block is that if the script is going to die eventually because of a missing dependency, it's better to have this happen before computations get underway.

Code script outline and content

- Your audiences for code overview, code comments & code itself:
  a. you
  b. your future self
  c. instructors/GSIs
- The overview should:
  - be written before you start coding
  - say what type of confidence interval you intend to compute to answer (1)
  - help you stay on track as you develop your programs
  - be revised as your plans and coding strategy evolve
  - be updated/clarified once you're done
- Code itelf should
  - define and combine smaller functions, rather than do everything at once
  - interleave informal checks of your functions, along lines of checks on `sd.prop` in U1PS2.

Additional programming hints

Running example.

```
midata = data.frame(zone=c("L", "BR", "J"), n_l=c(85, 132, 88),
                    phat_u=c(0.059, 0.076, 0.068) )
```

D-R-Y

I.e., don't repeat yourself.

Basic R techniques for iterative computations

- The `for` loop
- Vectorized computations

Whatever you do, don't just repeat the same block of code 40 times, once for each row of your data set!

Vectorized computation example Suppose that `n_l` and `phat_u` are variables living in `midata`. To get upper and lower CI endpoints based on the "conservative margin of error":

```
midata$lower = with(midata, phat_u -1/sqrt(n_l))
midata$upper = with(midata, phat_u +1/sqrt(n_l))
```

(If `n_l` and `phat_u` are just free standing variables, not bound up in a data frame, then you don't need the `with()` part. If you want `lower` to be a free standing variable, omit the `midata$` piece at the front.)

The same operations can be expressed more concisely using the `dplyr` package.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
midata <- mutate(midata, lower=phat_u - 1/sqrt(n_l),
                 upper=phat_u + 1/sqrt(n_l)
         )
```

Here's how to pretty-print a data frame.

```
knitr::kable(midata)
```

zone	n_l	phat_u	lower	upper
L	85	0.059	-0.049	0.167
BR	132	0.076	-0.011	0.163
J	88	0.068	-0.039	0.175

(This relies on the `knitr` package, which you may have to install; see below. If you knit your Rmd to HTML, as opposed to PDF, you'll be able to copy-and-paste this table into a Word or Google document.)

```
midata = cbind(midata, lower = numeric( nrow(midata) ))
for (i in 1:nrow(midata) )
{
  midata[i, "lower"] = midata[i,"phat_u"] - 1/sqrt(midata[i, "n_l"])
}
```

Example with a `for` loop Note how `lower` was pre-allocated prior to invoking the loop (as a numeric vector of length `nrow(midata)`). In this example it's pre-allocated to be a new column of `midata`, but it could have pre-allocated as a free-standing vector: `lower = numeric(nrow(midata))`.

The loop can also be indexed over strings, rather than numbers: for example, the row names of midata.

```
if (is.null(rownames(midata)))
   rownames(midata) = as.character(1:nrow(midata))

midata = cbind(midata, lower = numeric( nrow(midata) )

for (rn in row.names(midata) )
{
  lower[rn, "lower"] = midata[rn,"phat_u"] - 1/sqrt(midata[rn, "n_l"])
}
```

Touching-up selected computations post hoc

Two of BCD's recommended methods call for modifications when the estimated proportion is sufficiently close to 0 or 1.

We recommend that you do the un-modified version of either of these before considering adding in the modification. If you need to modify: by comparing your data to the criteria for modification stated by BCD, you may find that with your data the modified interval is no different from the unmodified one, and requires no additional calculation.

If you do need to modify an existing set of confidence interval endpoints, there are two strategies.

1. Modify CI enpoints in a vectorized manner.
2. Write an explicit loop over the rows of your data frame, deciding whether to modify and calculating modification within the loop.

I'll refer to modifying, and deciding whether to modify, as "updating."

Vectorized update approach Here are some examples of the vectorized manner of computation, applied to updating or modifying existing vectors.

First suppose we just wanted to make sure our CI limits were within [0,1].

```
midata$lower[midata$lower<0] <- 0
midata$upper[midata$upper>1] <- 1
knitr::kable(midata)
```

zone	n_l	phat_u	lower	upper
L	85	0.059	0	0.167
BR	132	0.076	0	0.163
J	88	0.068	0	0.175

Now suppose that `lower` is a vector of unmodified lower endpoints. Suppose that the modification you want to implement is that any time the sample size is less than 10 and $\hat{p} < .2$, then your lower endpoint should be 0. Then you could do

```
lower[n_l<10 & phat_u <.2] = 0
```

If `lower` is the name of a variable inside of `midata` rather than a freestanding vector, then you could change this to

```
midata$lower[n_l<10 & phat_u <.2] = 0
```

or

```
midata[n_l<10 & phat_u <.2, "lower"] = 0
```

Unless `n_l` and/or `phat_u` are also variables inside of `midata`, in which case you'll need something like

```
midata[with(midata, n_l<10 & phat_u <.2), "lower"] = 0
```

If the value you're updating to also depends on variables inside of `midata`, you may need two separate `with()` statements.

```
midata[with(midata, n_l<10 & phat_u <.2), "lower"] = with(midata, phat_u - 1/sqrt(n_l))
```

this is starting to be a lot packed into a single line. Splitting into two:

```
lower.modified = with(midata, phat_u - 1/sqrt(n_l))
midata$lower = with(midata, ifelse(n_l<10 & phat_u <.2, lower.modified, lower) )
```

Explicit loop approach to updating An explicit loop will be more verbose, but gives you a little more flexibility about how to execute the computation at each stage.

```
for (rn in row.names(midata))
{
  # computations affecting both upper and lower endpoint
  # calculations go here
 if (midata[rn,"phat_u"]<.1 & midata[rn, "n_l"]<10)
 {
  # calculations affecting lower endpoint
   midata[rn, "lower"] = 0 # or something
 }
   if (midata[rn,"phat_u"]>.9 & midata[rn, "n_l"]<10)
 {
  # calculations affecting upper endpoint
     midata[rn, "upper"] = 1 # or something
 }
}
```

Including tables in your report

The `knitr` package provides the `kable` function, which is useful for formatting tables in markdown. To check whether you've got it already on your system, do `library('knitr')`, you'll get an error if it's not there yet. To fix that, do `install.packages("knitr")`. For examples do `help('kable', package='knitr')`.

Splitting up a data frame by levels of a categorical variable

Pulling out a particular subset of the data

To extract just a particular piece of the data:

```
usadata_MW = subset(usadata, region %in% "midwest")
usadata_notMW = subset(usadata, !(region %in% "midwest"))
usadata_E = subset(usadata, region %in% c("northeast","south"),
                   select=c(zone:region,n_lowstart,p_upmover) )
```

and so forth. (Users of `dplyr` can accomplish the same using `dplyr::filter()` and `dplyr::select()`.) The last form includes only the columns between columns named "zone" and "region", plus the two labeled "n_lowstart" and "p_upmover".

R has specialized facilities for repeating operations on blocks of rows of a data frame. If you're familiar with these, or with the alternatives provided by the `dplyr` and/or `data.table` packages, use them. Otherwise I recommend a construction elaborating the `for` loop construction above:

```
newvec = numeric( nlevels(usadata$region) )
names(newvec) = levels(usadata$region)

for (rn in levels(usadata$region) )
{
  smalldata = subset(usadata, region %in% rn)
  newvec[rn] = with(smalldata, sum(N))
}
```