APPENDIX

R hints for Colombian Election question

Here are some hints for the sub-questions of the Election question that involve the use of R.

1. Use summarize and save the data set with a different name (i.e. `pop_stats`) as this data set might be useful later on.
2. To do this, use the `sample_n` command as you did in problem set #2.
3. Same as (2)
4. No R needed
5. This question essentially asks you to generate 1,000 estimates from independent samples. We can conduct a process similar to that of questions 3 and 4, storing each estimate as a new entry in a 1 x 1000 vector. Follow the walkthrough (two pages below) on the key logic and sample code for the "for loop" using the GSS dataset (built in R). For hints specific to this problem, see paragraphs just below. We first set up that output container for these estimates. Assigning an empty vector (`c()`) to an object name of your choice (such as `xbar_5`, indicating that you are going to store a sample mean from a group of 5 observations) is preferred because you do not need to hard-code the length of the vector. Then, we create the sequence to loop over. You want 1,000 estimates, so a natural (though not the only) choice is to let it loop over the sequence of integers from 1 to 1,000. Inside the loop, run the same code you used for the previous problems. The goal is to store each estimate as the ith value of the vector, e.g. into `xbar_5[i]`. Here setting the index as 1:1,000 turns out to be convenient because the iterator can serve as both the index as well as the index of the container vector.
6. Construct the same vector but where each sample comprises 100 observations.
7. To use `ggplot`, you must change base-R vectors and matrices into a dataframe of your own. This is straightforward enough with the `tibble()` function. There are multiple ways to create this graphic, and how you create your tibble will depend on the method you choose.
    a. The FIRST, and more straightforward way to create a tibble dataframe with two columns, one for each set of samples (notice that this only works because both vectors are of the same length --- 1,000). Then, either create two histograms with fixed x-axis and y-axis limits, or create a single `ggplot` object that has two `geom_histogram` layers, one for each variable (make sure to hard-code each to different translucent colors so that the differences are visible).
    b. The SECOND way is more flexible because it translates the data into long form, which makes it easier to color and group in `ggplot`. This requires you to stack the two sets of estimates in a single column, but with a separate column (call it \`n\`) that indicates for each the respective size of the sample (either "n = 5" or "n = 100"). With such a

dataset, you can create a single `ggplot` with a single `geom_histogram`, but grouping by the category for sample size. The default is to stack two groups of bars, but this is not conducive to visual comparison because it does not place the bars on the same axis. Instead, you should overlay them by changing the position parameter of the histogram geom. Set the `` `positions` `` argument of the `geom` to `` geom_identity()` `` for stacking. You can also set it `to position_dodge()` to dodge the two groups, but this is awkward in a histogram (better for barplots)

8. No R needed
9. No R needed
10. Use summarize and take the mean of the logical statement (i.e., a statement that returns TRUE or FALSE). Recall from math camp that logical values can get coerced into a numeric, where TRUE becomes 1 and FALSE becomes 0. Therefore, taking the mean of a vector of logicals is equivalent to taking the proportion of times that statement is TRUE.
11. Similar to (10)
12. No R needed
13. As in (7), it is more convenient to do this analysis on a dataframe, instead of on vectors. First, take a dataframe where one column is exclusively the estimates from a n = 1,000 sample. You can create the bounds by creating separate variables for the lower bound and upper bound, according to the confidence interval formula. Since you will be assuming the standard error is the same scalar for every draw, it is convenient to store that as a separate object beforehand. Once you have created the CI estimates (two for each row), you can answer the last part of the question by the same approach in (10) but with the appropriate logical statement.
14. No R needed
15. No R needed
16. No R needed

WALKTHROUGH FOR CREATING A FOR LOOP IN R[1]

Many tasks you do in coding are repetitive, or at least follow a similar pattern. Although it is tempting to copy and paste code, a general rule of thumb is to never copy the same code more than twice. Creating three copies of the same code triples the risk you make a mistake. The solution is to *iterate* over the same code.

For example, suppose you want to compute the mean TV hours watched for each wave of the GSS using a for loop. You would write something like the following:

[1] For even more explanation see, R for Data Science Chapter 21.

```
library(tidyverse)
data(gss_cat)

gss_years <- unique(gss_cat$year)

tv_mean <- c()

for (i in 1:8) {
  year_i <- gss_years[i]
  gss_i <- filter(gss_cat, year == year_i)
  tvhours_i <- pull(gss_i, tvhours)

  tv_mean[i] <- mean(tvhours_i, na.rm = TRUE)
}
```

For loops are the basic building block for iteration. It has three main components:

1. The **output** (`tv_mean <- c()`): Before you start the loop, you need to create a container for that has sufficient space for the output of your many iterations. In this case we created an empty vector that will be populated with sample averages.
2. The **sequence** (i in 1:8): This component narrows down the initial source of change in each iteration. In other words, it determines what to loop over. It has a very specific syntax. On the left handside of the "in", pick a name of your choice that will represent the iterator (a convention is to call it i). On the righthand side of the "in", specify a vector of values you want it to iterate through. In this case, we decided to let i iterate through the values 1 through 8.
3. The **body** (chunk ending in `tv_mean[i] <- mean(tvhours_i, na.rm = TRUE)`)**:** Data processing should end in assigning a value (or values) to a location in your output container. Because it is run repeatedly for each iteration, each time for a different value of the iterator, the location the values that are assigned to should be defined by i. In this case, we let the i^{th} element of the output be the mean of TV hours among people interviewed in the i^{th} year in the data.

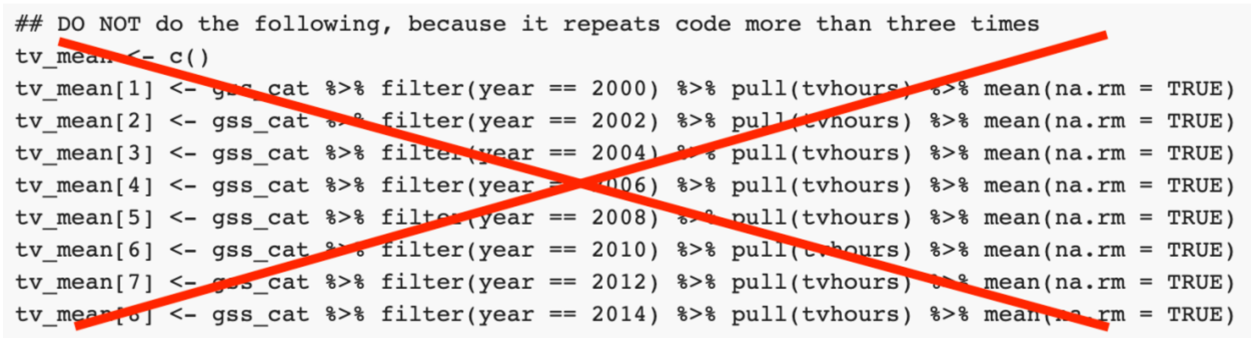
Finally, note that we use curly braces to define the body.

Note that a much easier way to have done this procedure would be to use group_by and summarize:

```
gss_cat %>%
  group_by(year) %>%
  summarize(tv_mean = mean(tvhours, na.rm = TRUE))
```

But in some cases, like the one encountered in this problem, you need to use a flexible iteration method like the for loop.

Finally, you might have noticed that an alternative that does the same thing is to repeat the same line of code 8 times. Do NOT do this -- a loop is designed to avoid doing this.

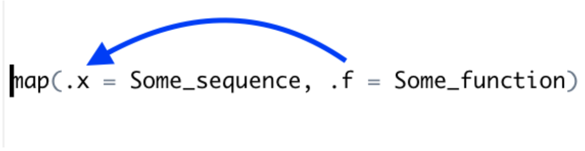


```
## DO NOT do the following, because it repeats code more than three times
tv_mean <- c()
tv_mean[1] <- gss_cat %>% filter(year == 2000) %>% pull(tvhours) %>% mean(na.rm = TRUE)
tv_mean[2] <- gss_cat %>% filter(year == 2002) %>% pull(tvhours) %>% mean(na.rm = TRUE)
tv_mean[3] <- gss_cat %>% filter(year == 2004) %>% pull(tvhours) %>% mean(na.rm = TRUE)
tv_mean[4] <- gss_cat %>% filter(year == 2006) %>% pull(tvhours) %>% mean(na.rm = TRUE)
tv_mean[5] <- gss_cat %>% filter(year == 2008) %>% pull(tvhours) %>% mean(na.rm = TRUE)
tv_mean[6] <- gss_cat %>% filter(year == 2010) %>% pull(tvhours) %>% mean(na.rm = TRUE)
tv_mean[7] <- gss_cat %>% filter(year == 2012) %>% pull(tvhours) %>% mean(na.rm = TRUE)
tv_mean[8] <- gss_cat %>% filter(year == 2014) %>% pull(tvhours) %>% mean(na.rm = TRUE)
```

WALKTHROUGH FOR USING THE MAP() FAMILY OF FUNCTIONS IN R[2],[3]

An alternative way of creating loops in R is to use the `map()` functions. If you are interested in learning how to do this, please read ahead.[4] The family of `map()` functions in R work in a very similar fashion to "for" loops in that they allow you to apply an iterative process to your code. In very general terms, their principal difference is that they incorporate the looping nature of `for()` into a single function. Consequently, their main advantage is that they are more concise than for loops while also making your code easier to read and interpret. The syntax differs in that instead of looping over sections of code, `map()` "maps" a function to every element of a vector or list. With a little tweaking pretty much everything a for loop can do, `map()` can too. `Map()` also has a convenient set of derivative functions to coerce output into whatever you please.

Map() takes the following basic form:



```
map(.x = Some_sequence, .f = Some_function)
```

The first argument, .x, is always a vector or a list. The second argument, .f, takes the function and applies it to each element of .x. The output of `map()` is a list but we can always coerce a list into other forms or use other functions in the family such as `map_dbl()` (numeric vector), `map_dfc()` (dataframe by column), etc. Comparing this to the above explanation of for loop .f is the **body,** .x is the **sequence,** and map creates the **output**.

The function can be written just like you would normally in R, or you can also use the "tilde operator (or twiddle)", ~, to either call a function in the global environment or just incorporate a formula. When using ~ you must use the pronoun ".x" or "." to tell the function that you wish to use your defined sequence. The following two lines produce identical output:

```
map(.x = 1:100, function(x) x*2)
map(.x = 1:100, ~.x*2)
```

[2] Additional explanation also available at R for Data Science Chapter 21.
[3] A useful tutorial is also available with the Rstudio primer
[4] My thanks to Paul Beach for the initiative and the writing of this section.

Each would iterate over the sequence 1 to 100 and multiply each element by 2, producing a list from 2 to 200.

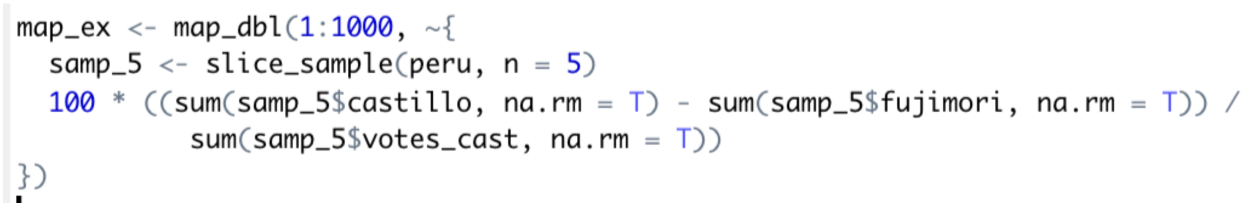
If we were to use:

```
map_dbl(.x = 1:100, ~.x*2)
```

the output would be a numeric vector instead

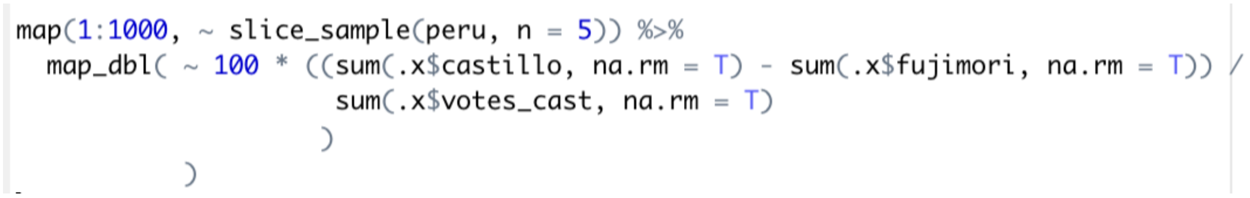
Here are a couple of practical examples:

First, we can replicate the method in question 5 with `map()` instead of `for()`. A potential answer could look something like this:



```
map_ex <- map_dbl(1:1000, ~{
  samp_5 <- slice_sample(peru, n = 5)
  100 * ((sum(samp_5$castillo, na.rm = T) - sum(samp_5$fujimori, na.rm = T)) /
           sum(samp_5$votes_cast, na.rm = T))
})
```

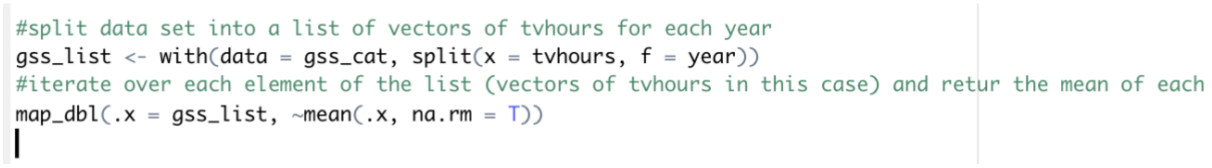
You could also use map more than once and “pipe in” output to subsequent functions. The code below accomplishes the same thing by the same means but notice how `map()` creates a list of 1000 sampled dataframes while the `map_dbl()` function accepts the list of dataframes and modifies each.



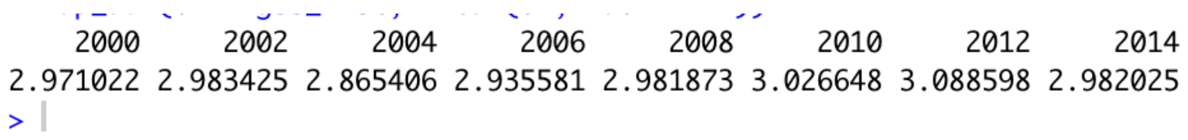
```
map(1:1000, ~ slice_sample(peru, n = 5)) %>%
  map_dbl( ~ 100 * ((sum(.x$castillo, na.rm = T) - sum(.x$fujimori, na.rm = T)) /
                     sum(.x$votes_cast, na.rm = T)
                    )
          )
```

We can also replicate the above example using map(). An answer could look something like this:

The first line splits the data set by year, creating a list of 8 vectors. `Map_dbl` then takes that list and applies the mean function to each list.



```
#split data set into a list of vectors of tvhours for each year
gss_list <- with(data = gss_cat, split(x = tvhours, f = year))
#iterate over each element of the list (vectors of tvhours in this case) and retur the mean of each
map_dbl(.x = gss_list, ~mean(.x, na.rm = T))
```



```
    2000     2002     2004     2006     2008     2010     2012     2014 
2.971022 2.983425 2.865406 2.935581 2.981873 3.026648 3.088598 2.982025 
> 
```