Lab 7 Solutions

11/4/2021

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.2.1
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

purrr and maps

You have now seen `map` functions a few times in the lecture slides. Today, we'll talk in more detail about what they are and what they do.

There are several variations of `map` that behave differently in terms of what arguments they require and what they return. They all come from the `purrr` package, which is included in the tidyverse. Before we go any further, take a look at the documentation for map:

```
?map
```

"Applying a function to each element of a list" is a type of *iteration* that is very powerful working with data. If you have programmed in other languages, you probably used for loops to accomplish tasks that require iteration (and you can use for loops within `R`, but we will not teach this approach).

What arguments do we *have* to specify in a call of `map`? What form can the `.f` argument take?

- Every call of map requires a vector or list over which to iterate.
- Note that if you want to return a vector of the same type as the input (instead of a list), you should use the appropriate variant of `map` (`map_lgl` for logical vectors, `map_chr` for character vectors, etc.).
- `.f` can take the form of an existing function, a formula (which must be converted into a function), or a vector.

~ and `.` notation in purrr

- If you look closely at the documentation, you'll notice some syntax we haven't used before: `~` and `.` in the formula of a `map` call.
- As discussed, you can use an existing function inside map, or a formula. If you use a formula, then you should lead with `~`.

- This will transform your formula into an `anonymous function` so that map can alter the values inside the formula as it iterates through the items in your list or vector.
- Otherwise the formula would execute immediately.

To see why this is important, run the following code line by line (don't run the whole chunk at once):

```
vec_1 <- 1:25

map_dbl(vec_1,
          ~ ./2)

map_dbl(vec_1,
          ./2)
```

- It works correctly in the first case, but not the second. The error says that "object `.` not found."
- This is because `the formula wasn't converted to a function` and it was evaluated immediately instead of iterating over the elements of the vector and `.` isn't defined.
- Hopefully you have been able to decipher what role `.` is playing as well.
- If you look at the output of `map_dbl(vec_1, ~ ./2)`, you should notice that `each element in the resulting vector contains each element of the original vector` divided by 2.
- The `.` is a placeholder that `represents each element of the vector or list` you pass into `map`. Let's look at a more complex example:

```
mtcars %>%
  split(.$am) %>%
  map(~ lm(mpg ~ cyl, data = .)) %>%
  map(summary)
```

```
## $`0`
##
## Call:
## lm(formula = mpg ~ cyl, data = .)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.6676 -1.0691  0.1324  1.3809  4.1324
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  30.8735     2.5901  11.920 1.11e-09 ***
## cyl          -1.9757     0.3644  -5.422 4.58e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.388 on 17 degrees of freedom
## Multiple R-squared:  0.6336, Adjusted R-squared:  0.6121
## F-statistic:  29.4 on 1 and 17 DF,  p-value: 4.576e-05
##
##
## $`1`
```

```
##
## Call:
## lm(formula = mpg ~ cyl, data = .)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.5255 -1.6638 -0.3638  2.4745  5.9745
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  41.0489     3.5720   11.49 1.81e-07 ***
## cyl          -3.2809     0.6751   -4.86 0.000503 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.63 on 11 degrees of freedom
## Multiple R-squared:  0.6823, Adjusted R-squared:  0.6534
## F-statistic: 23.62 on 1 and 11 DF,  p-value: 0.0005026
```

What happened when we split `mtcars`? What was in the list that `map` iterated over?

A: Splitting the data created two dataframes, and the list passed into the first call of `map` was this list of two dataframes. It returned a list of two linear models, which is what was passed into the second call of `map`. That's why two summary tables were printed when we ran the code.

Notice that when you use some existing functions inside `map`, you don't need to use the `~`. This is because they are already defined as functions (unlike formulas) and `map` will pass each item in the list (which you can pipe in as above) to the function as is. Here is another example:

```
mean_list <- map(mtcars, mean)
```

What does this chunk return? What do the values and names in the list represent? What does this reveal about how map works when you pass in a dataframe?

A: The names in the list correspond to the column names in `mtcars` and the values are the mean of each column. This demonstrates that when you pass a dataframe to `map`, the function is applied to each column.

If you want a vector of values instead of a named list (perhaps because you want to pipe the values into another function), you can use `unlist`.

```
mean_vec <- mtcars %>%
  map(mean) %>%
  unlist

class(mean_list)
```

```
## [1] "list"
```

```
class(mean_vec)
```

```
## [1] "numeric"
```

NOTE: You can also use a variant of map that returns a vector instead of a list (e.g. `map_dbl`):

Practice with purrr

Let's combine your knowledge of functions and with `purrr` and practice. First, use `map` to calculate the variance of each column in `mtcars` and return it as a vector of numeric values.

```
mtcars %>%
  map_dbl(var)
```

```
##          mpg          cyl         disp           hp         drat           wt
## 3.632410e+01 3.189516e+00 1.536080e+04 4.700867e+03 2.858814e-01 9.573790e-01
##         qsec           vs           am         gear         carb
## 3.193166e+00 2.540323e-01 2.489919e-01 5.443548e-01 2.608871e+00
```

Next, generate a sample of 100 observations drawn from the standard normal distribution and save it (hint: set your seed so your results are reproducible).

```
set.seed(2418)

norm_sample <- rnorm(n = 100)
```

What is the mean and variance of this sample?

```
mean(norm_sample)
```

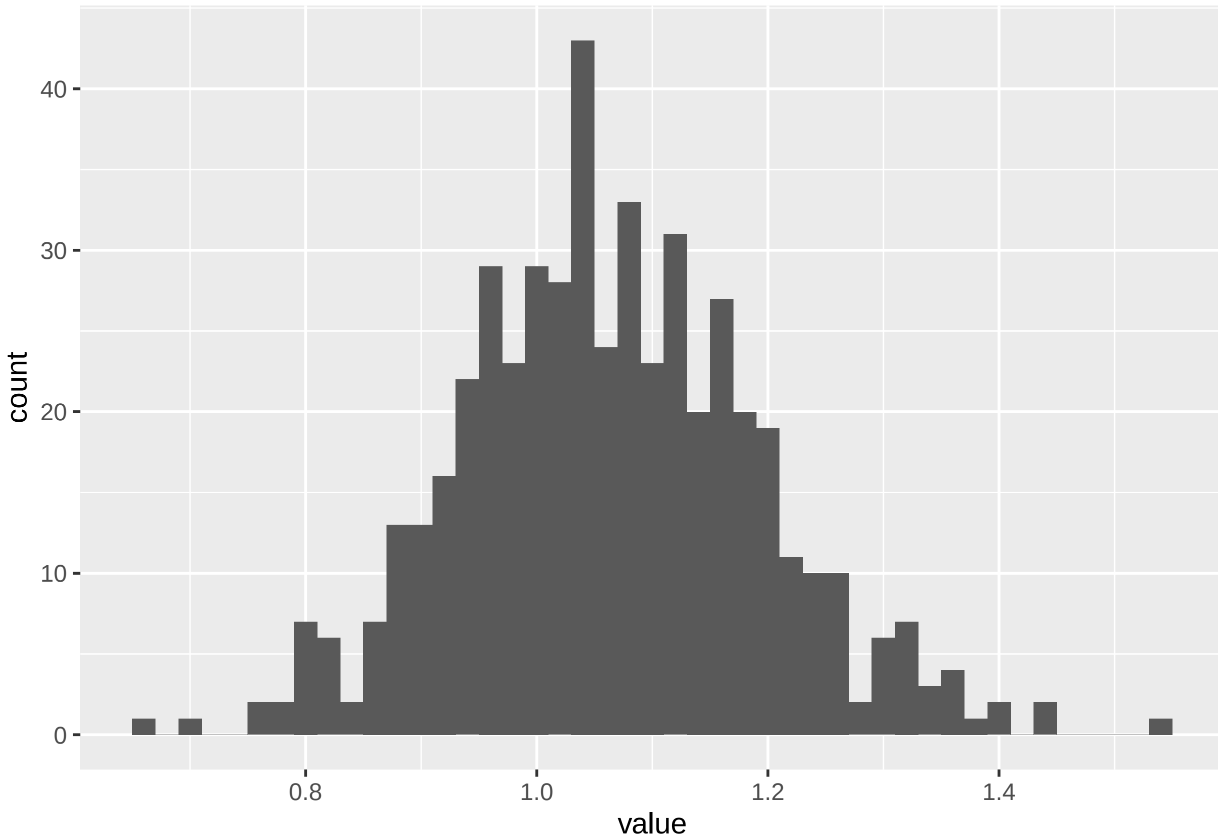
```
## [1] -0.09675547
```

```
var(norm_sample)
```

```
## [1] 1.069042
```

Now, use `map` to generate 500 bootstrapped samples from this sample and find the variance of each sample. Finally, plot the distribution of these variances. Can you do it in a single pipe? (hint: remember to sample with replacement)

```
map(1:500,
    ~ sample(norm_sample, replace = TRUE) %>%
  var()) %>%
  unlist() %>%
  as_tibble() %>%
   ggplot(mapping = aes(x = value)) +
     geom_histogram(binwidth = .02)
```



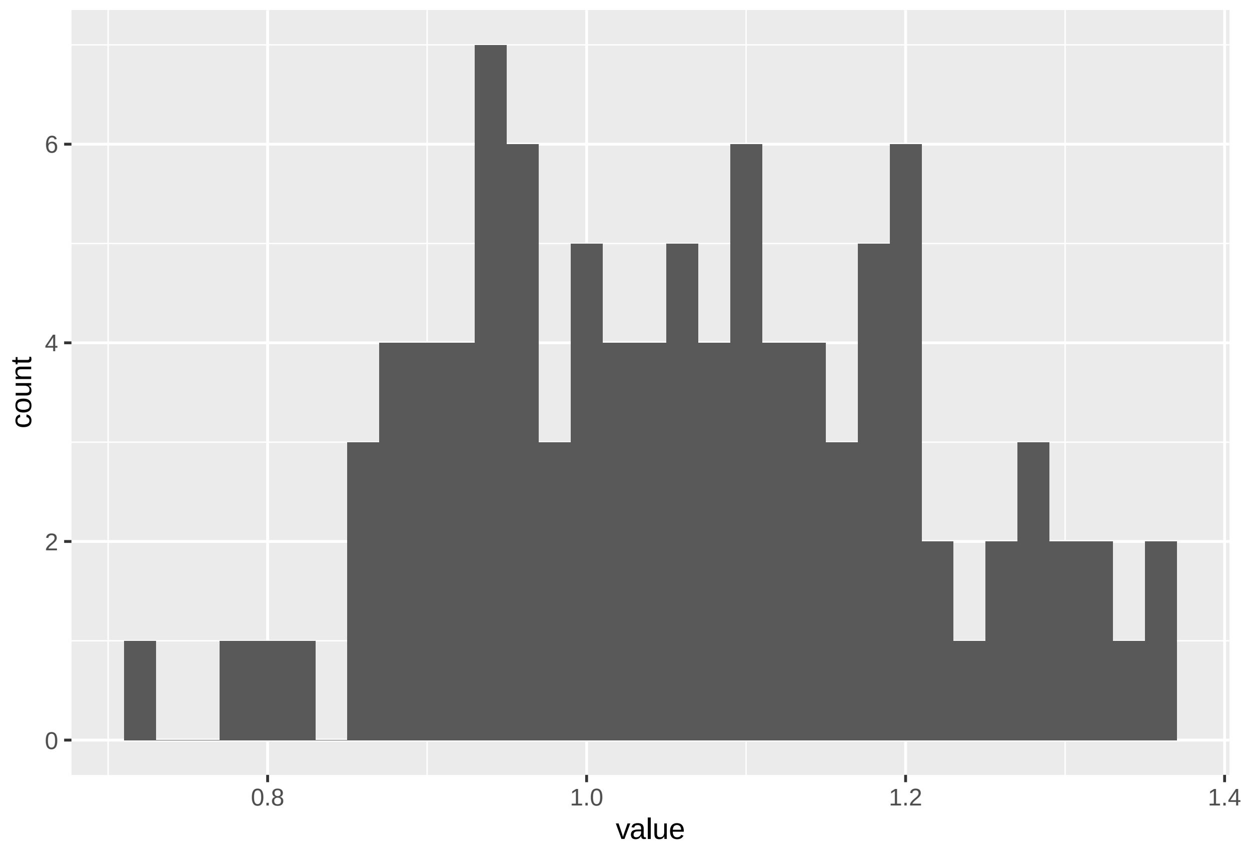
Where does the distribution seem to be centered? How does this relate to the variance of the original sample you calculated above?

A: The distribution seems to be centered just to the right of 1. This is reasonably close to the true (population) variance of our sample, which is 1.07.

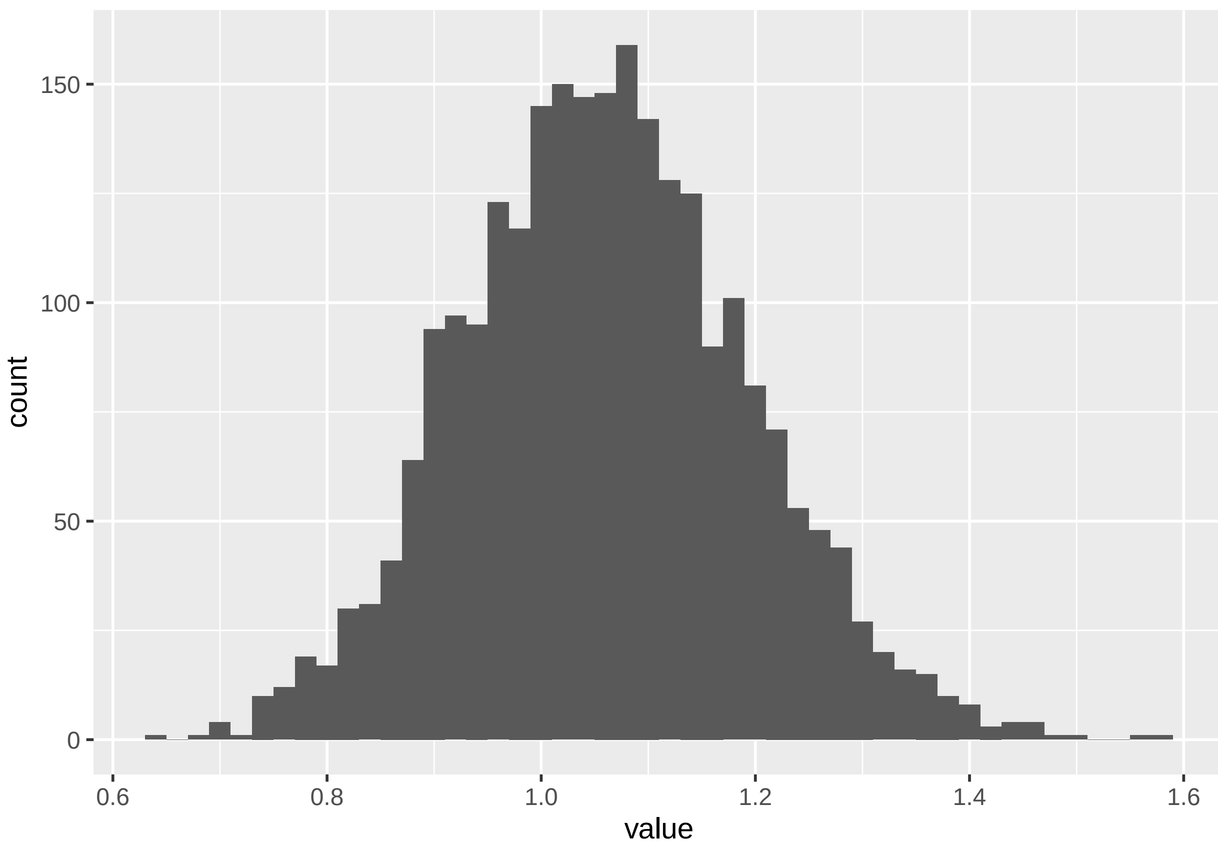
BONUS If you managed to do the last task in a single pipe, turn it into a function that takes two arguments: the sample you want to bootstrap from and the number of bootstrapped samples you want to generate. What happens to the distribution as you change the value of `n`?

```
norm_var_boot <- function(samp, n){
    map(1:n,
      ~ sample(samp, replace = TRUE) %>%
    var()) %>%
    unlist() %>%
    as_tibble() %>%
     ggplot(mapping = aes(x = value)) +
       geom_histogram(binwidth = .02)
    }

norm_var_boot(norm_sample, 100)
```




```
norm_var_boot(norm_sample, 2500)
```



This is just the scratching the surface of what `purrr` can do. You can use it to iterate over all sorts of things: lists of files, urls, dataframes, nested dataframes, etc. Combined with the power and flexibility of functions in `R`, `map` is an extremely powerful tool and one that you will probably find yourself using as you continue your journey as quantitative social scientists!