

Lesson 7-Key!

YOUR NAME

2023-10-25

Learning goals

- Using R's built-in loop functions: `apply`, `lapply` and contrasting them with `for` loops
- Conditional statements
- Investigate sampling error
- Visualize confidence intervals.
- Adding error bars to plots

Apply and lapply

Last week you learned about loops, which are standard in all programming languages. However, R has some built in functions that perform similar operations and are a bit more simple to use. They belong to the “apply” family. Here are some differences between `apply` and `lapply` Differences between `lapply` and `apply`:
Data Types:

- `lapply` works on lists and vectors.
- `apply` works on matrices, arrays, and `data.frames`.

Return Type:

- `lapply` always returns a list.
- `apply` can return a vector, matrix, or array, depending on the function being applied and the input's dimensions.

Use Case:

- Use `lapply` when you have a list or vector and you want to apply a function to each of its elements.
- Use `apply` when you have a matrix or array or `data.frame` and you want to apply a function across its rows or columns.

There are many other functions in the `apply` family but we are just focusing on these two. I recommend typing `? apply` and `?lapply` into the console before starting. The second argument, `margin`, can be tricky.

1. Let's start super basic. Say you have a matrix `mat <- matrix(1:12, nrow=3)`. You want to find the sum of each column. paste the matrix first then use `apply` to find the sum. Name it something then print the result

```
## [1] 6 15 24 33
```

2. Use lapply to find how many characters are in each name. Save as an object and then print. To count characters, you can use the `nchar` function. Here are your names, get counting `names_list <- list("Alessandra", "Penelope", "Meigui", "Tatiana", "Keisuke", "Nadia", "Amy")`

```
## [[1]]
## [1] 10
##
## [[2]]
## [1] 8
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 7
##
## [[5]]
## [1] 7
##
## [[6]]
## [1] 5
##
## [[7]]
## [1] 3
```

What is different about this output? Correct, it's a list!

3. Comparing apply and for loops You're a teacher and you've just given your students a test. The test scores are stored in a matrix where each row represents a student and each column represents a test question. Calculate the average score for each student using both `apply` and a `for` loop. Your matrix is called `scores` and has all the information you need.

```
##           Math English Biology History
## Alice      85      90      78      88
## Bob        92      80      95      90
## Charlie    60      55      78      84
```

First use the new functions you learned to find the average test score for each student.

```
“{r. ex3.1 } #use apply averages_apply <- apply(scores, 1, mean)
```

Now use a `for` loop. Remember to create blank variables first to store your information. You can call the

* Which of these was easier for you? Which student is doing the best?

* Which class do you think is hardest? Lets do the process again to find the average (mean) for each class.

First use ``apply`` and save the results in an object called ``class_averages_apply``:

```
```{r. ex3.3 }  
#create object
class_averages_apply <- apply(scores, 2, mean)
#print your object
```

Then use a `for` loop and save the results in an object called `class_averages_loop`:

And now you understand the joy of the `apply` family!

## Conditional Statements

Conditional Statements are a key part of any programming language.

### 1. if Statement:

The `if` statement evaluates a condition, and if that condition is `TRUE`, it executes the code inside the statement. Change `x` to a few different numbers to see how the output changes!

```
[1] "x is greater than 5"
```

In the above code, since `x` is indeed greater than 5, the message “x is greater than 5” will be printed.

Change `x` to a few different numbers to see how the output changes!

### 2. if-else Statement:

There might be cases where you want to execute one set of instructions if the condition is `TRUE` and another set if it's `FALSE`. This is where the `if-else` statement comes in.

```
[1] "x is not greater than 5"
```

You can keep adding else statements!

### 3. if-else if-else Ladder:

For multiple conditions, we can use an `if-else` ladder.

```
[1] "x is less than or equal to 5"
```

And so on and so on. This is really useful for making code do complicated things!

Change `x` to a few different numbers to see how the output changes!

4. Write a basic if statement that prints “this number is positive” when that is true. Try some numbers!

```
[1] "The number is positive."
```

Now add an `else if` statement that tells you if the number is negative, and an `else` that tells you if its 0.

```
[1] "The number is positive."
```

Amazing! You now understand basic conditionals! Let’s combine this with loops. Here is an example of a loop that contains an if statement Lets say you have a bunch of numbers and you want to know how many are positive.

This loop adds one to the count for each positive number.

5. Write a loop telling me the sum of all the even numbers.

## More work in sampling

Now we are going to return to our example from last class, the human genes data set. We are going to graph more sampling distributions and add error bars to some plots.

6. Last class we compared the means of differently sized samples. Generally, the larger your sample, the closer it will be to the true population mean. The human gene data set is a bit weird though. Because it has a few genes that are huge, a large sample size might be more likely to include one of those giant genes and mess up your mean.

For a real experiment you would just deal with that. For our purposes, you should filter the data set to only include genes smaller than 15000 base pairs. Load in your data and do this. You can keep it’s name the same

Note: This issue with the long genes is explained in the key for lab 6.

```
Rows: 22385 Columns: 4
-- Column specification -----
Delimiter: ","
chr (3): gene, name, description
dbl (1): size
##
i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Run the loop you wrote last week and investigate the means now that we have filtered the data. This was the loop where you took 100 samples of three different sizes: 10, 100, 1000, with replacement, and calculated their means. Do this again here with this filtered dataset and look at sample sizes of 10, 50, and 500. What do you see?

```
[1] 74.17859 98.80723 99.66763
```

## 7. What's up with these samples?

1. Start by finding the standard error of the mean, which is your the standard deviation of the samapling distribution. Your three sampling distributions were created above and are called `mean_values_10`, `mean_values_50`, and `mean_values_500`.

Congratulations! You just found the standard error by taking the standard deviation of the sampling distributions!

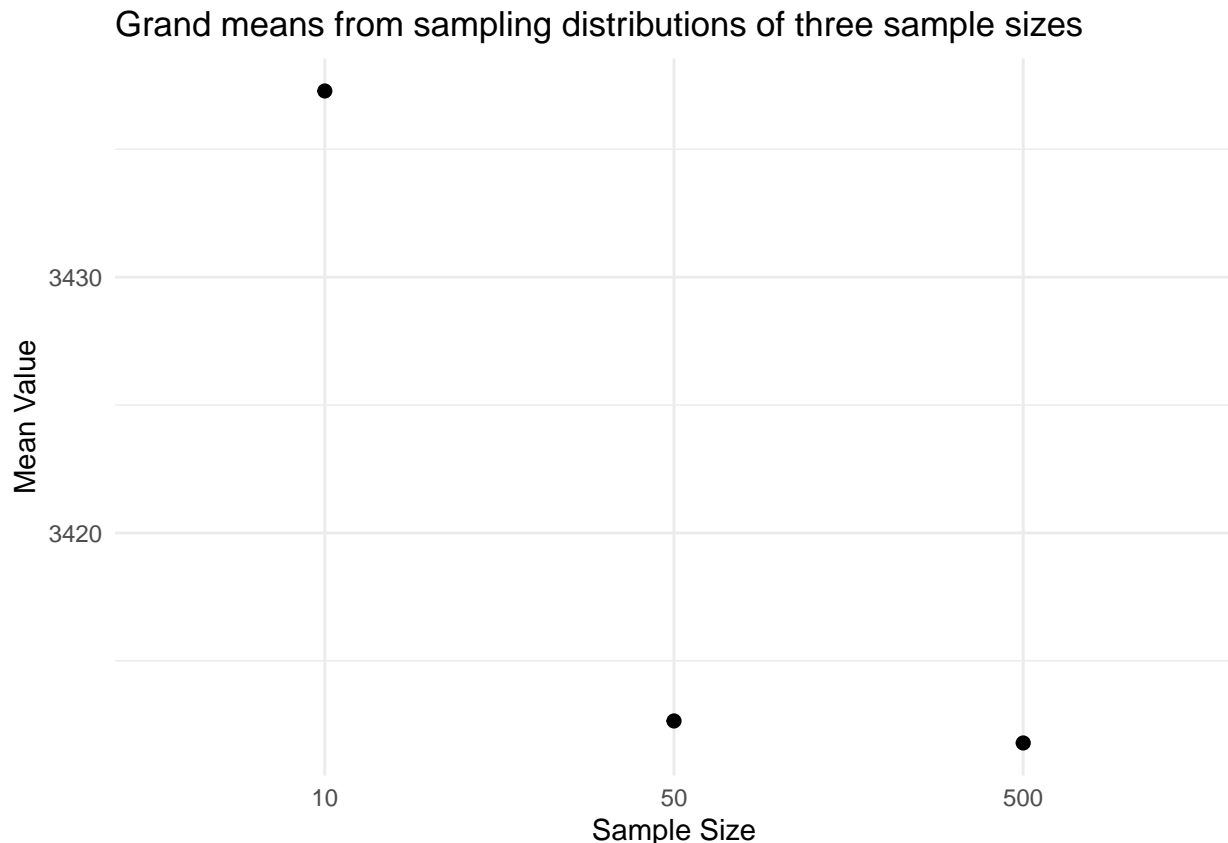
Now, think about which one of them should have the lowest SEM? Now check if that's true. Does it match your expectations?

2. Now lets find the 95% confidence intervals. Type `?quantile` into your console to learn how to use this neat function. Hint: We want 95% of the data to be within the intervals, so what should the quantiles be?

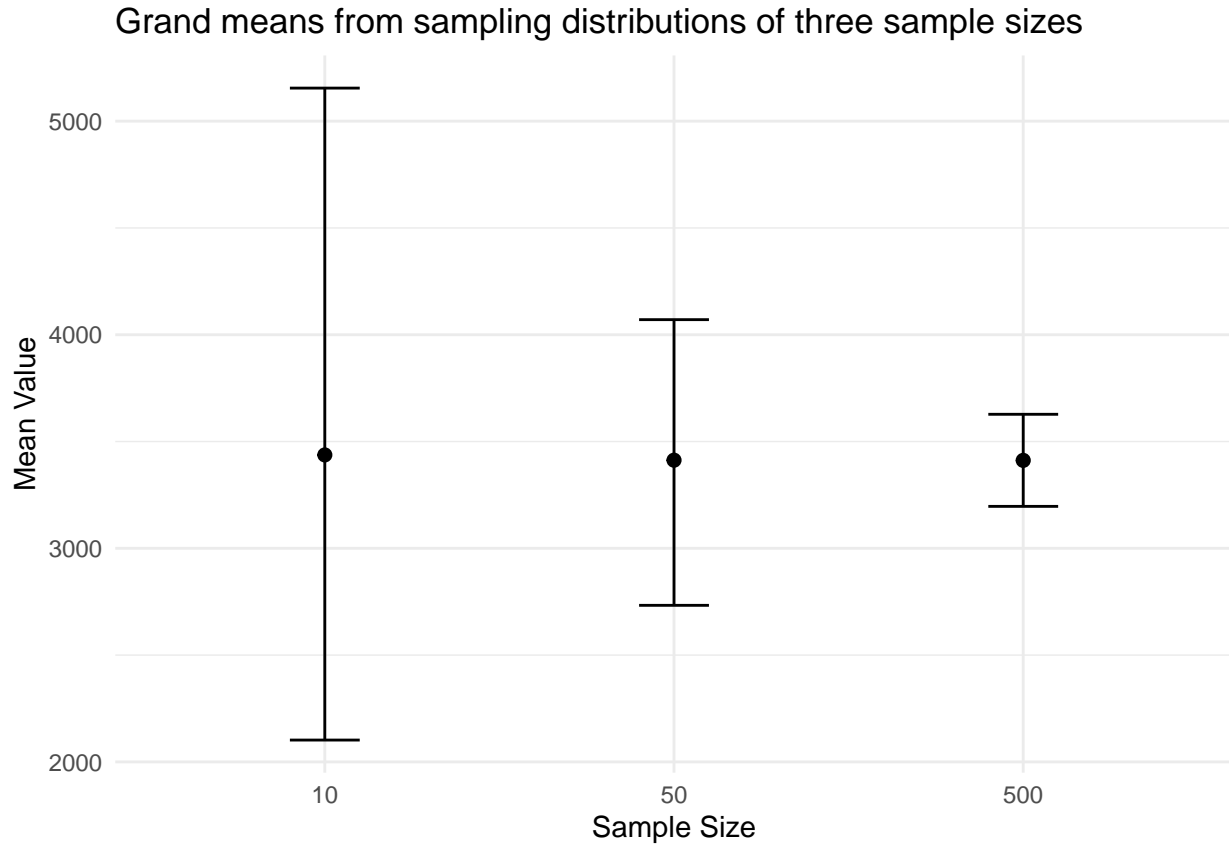
Now we have these confidence intervals, and you didn't need a crazy formula or assumptions about the mean being normally distributed! You simply took samples with replacement and calculated your confidence intervals empircally - aka, you used bootstrapping!

**8. Lets add the CIs to a graph!** Lets first make a data frame of all the information we will need for the chart. You need the following columns: `SampleSize`, `MeanValues`, `UpperCI` and `LowerCI`. You can use the ones you've calculated above.

Amazing! Now make a strip chart using this data frame. Don't worry about CI yet. Just make a simple plot using `geom_point()` for each of the means, grouped by sample size. Add labels though. See the expected output document for inspiration.



Gorgeous. Now add your error bars. Use `geom_errorbar` to see the difference in those sample sizes. Add `geom_errorbar` after `geom_point()`. Inside `aes()`, use `ymin` and `ymax` with the lower and upper CIs, respectively.

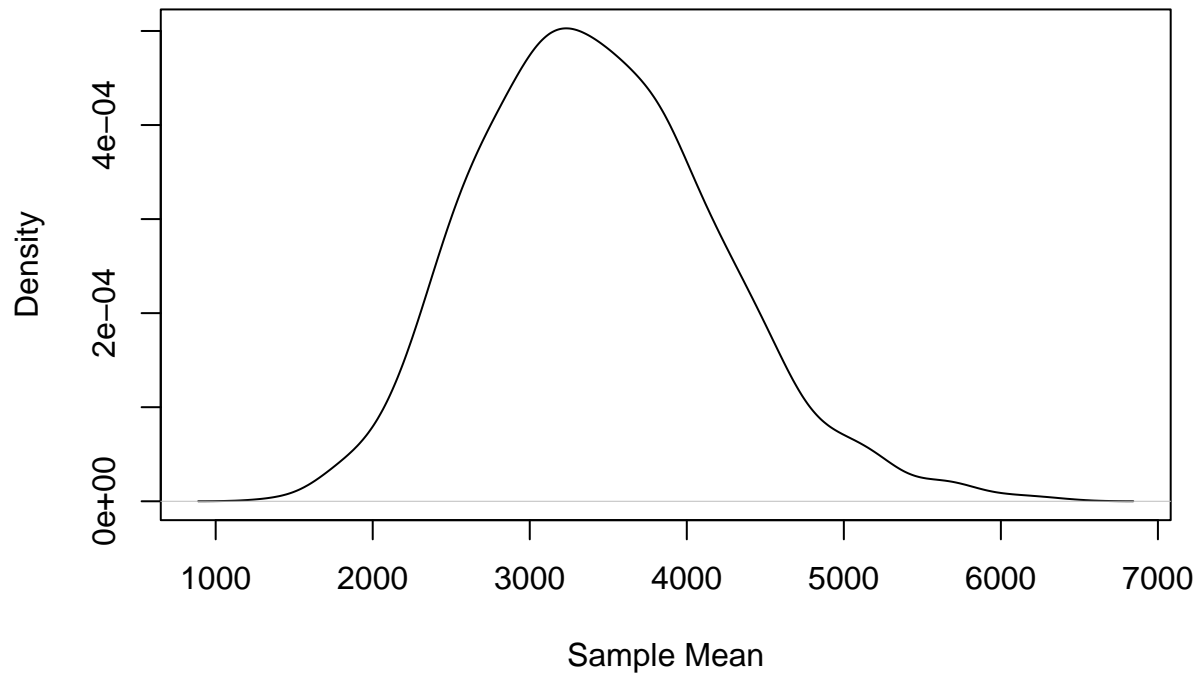


Those are super different! As you can see, larger sample sizes create narrower confidence intervals. This is **CRUCIAL** for your own data analysis.

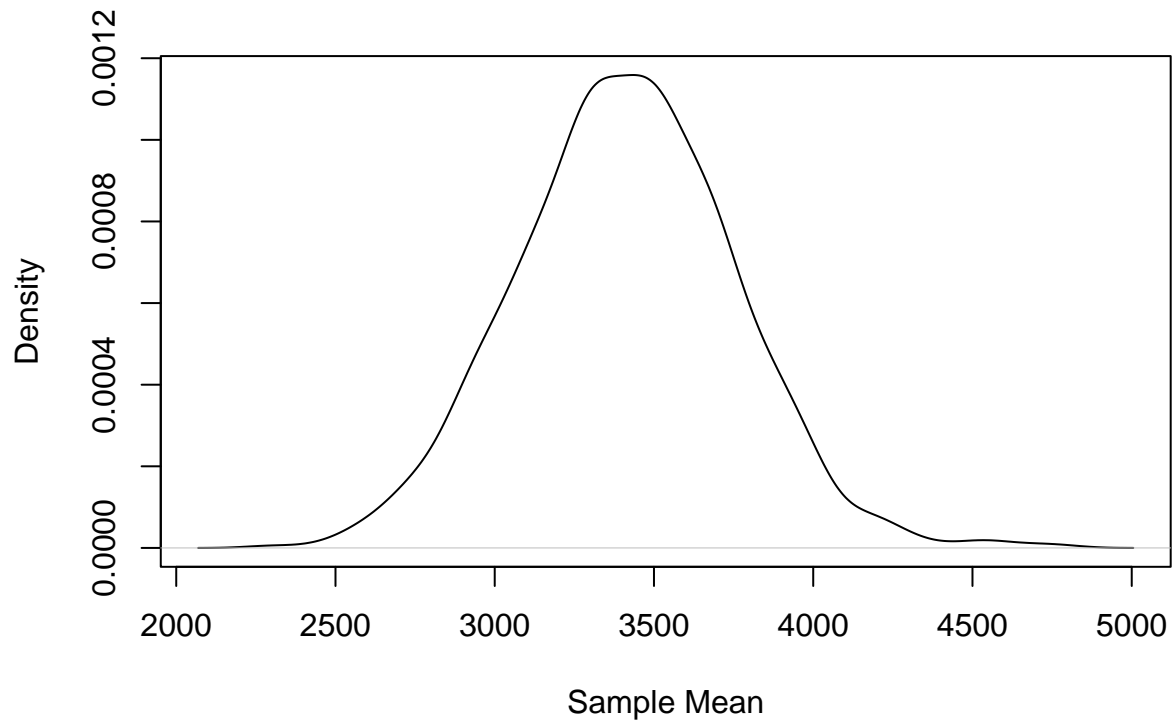
**9. Another graph** Above we plotted only the grand means of the sampling distributions. But what do these sample distributions look like compared to a normal distribution? R has a built in plot (gasp it's not `ggplot2`) called a density plot that will show us. First, use the density function to find the density of each of your sample means. Name them something you'll remember. PS: `ggplot` also has an option for density plots.

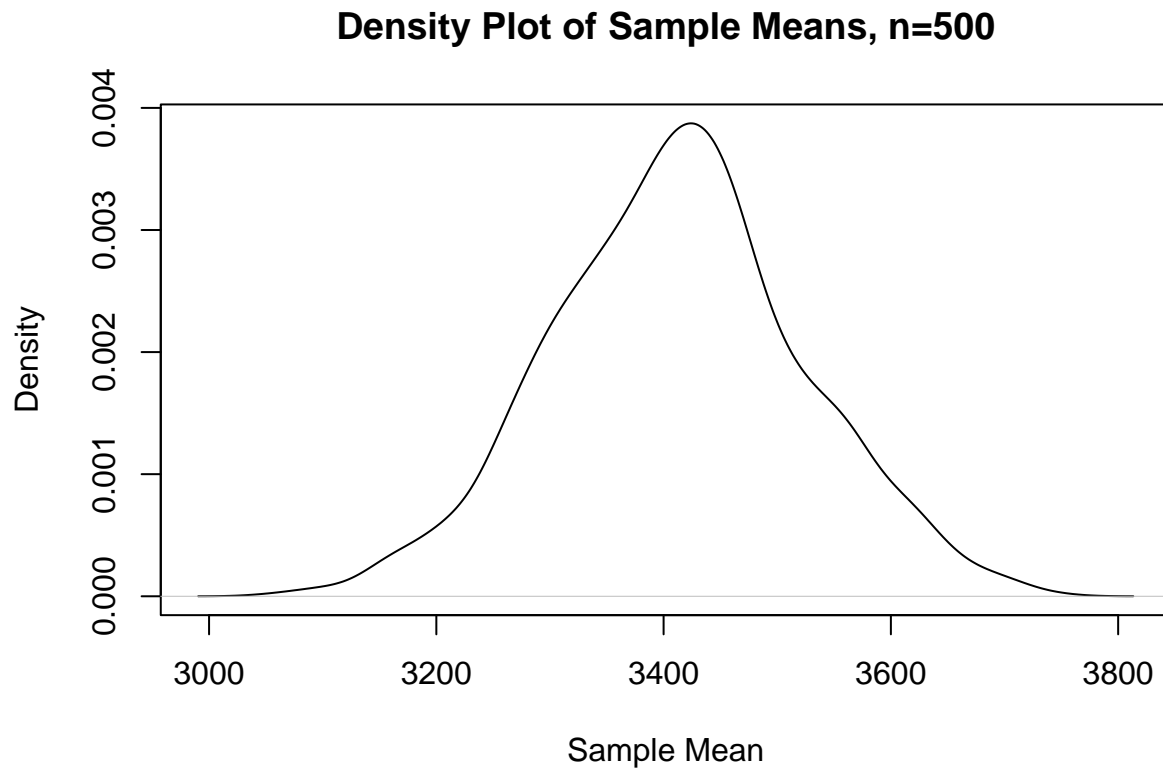
Now use the R function "plot" to plot each of these.

**Density Plot of Sample Means, n=10**



**Density Plot of Sample Means, n=100**





You see as the  $n$  increases they get more normal!

The end!