Iteration & Simulation

Natalia Vasilenok

Math Camp
Department of Political Science
Stanford University

September 11, 2023

Roadmap

1. Iteration
   - The `map()` family
   - Matrices
   - The `apply()` family
2. Simulation

Applying a function to each element of a vector

Suppose $\mathbf{X} = \{x_1, x_2, x_3, x_4\}$, and $f(x) = (x+1)^2$. Let's define the function g that will apply the function f to each element of the vector $\mathbf{X}$:

$$\mathbf{X}' = g(\mathbf{X}, f)$$

$\mathbf{X}$	0	1	3	8
$\mathbf{X}'$	1	4	16	81

Applying a function to each element of a vector

Suppose $\mathbf{X} = \{x_1, x_2, x_3, x_4\}$, and $f(x) = (x+1)^2$. Let's define the function g that will apply the function f to each element of the vector $\mathbf{X}$:

$$\mathbf{X}' = g(\mathbf{X}, f)$$

$\mathbf{X}$	0	1	3	8
$\mathbf{X}'$	1	4	16	81

Applying a function to each element of a vector

Suppose $\mathbf{X} = \{x_1, x_2, x_3, x_4\}$, and $f(x) = (x+1)^2$. Let's define the function g that will apply the function f to each element of the vector $\mathbf{X}$:

$$\mathbf{X}' = g(\mathbf{X}, f)$$

$\mathbf{X}$	0	1	3	8
$\mathbf{X}'$	1	4	16	81

Applying a function to each element of a vector

Suppose $\mathbf{X} = \{x_1, x_2, x_3, x_4\}$, and $f(x) = (x+1)^2$. Let's define the function g that will apply the function f to each element of the vector $\mathbf{X}$:

$$\mathbf{X}' = g(\mathbf{X}, f)$$

$\mathbf{X}$	0	1	3	8
$\mathbf{X}'$	1	4	16	81

Functionals

$$\mathbf{X}' = g(\mathbf{X}, f)$$

In R, g is called a *functional*, a function that takes another function as an argument and returns a data object (a list, a vector, or a data frame). Functionals are a more efficient alternative to `for` loops.

Exercise: `for` loop refresher

Write a `for` loop that applies $f(x) = (x+1)^2$ to each element of a vector `x`

```
x = c(0, 1, 3, 8)
```

and stores the results in a vector `y`.

Exercise: `for` loop refresher

```r
x = c(0, 1, 3, 8)
y = c()

for(i in 1:length(x)){
  y[i] = (x[i]+1)^2
}


y
```

```
# [1]  1  4 16 81
```

purrr

```r
install.packages("purrr")
library(purrr)
```

source: @weirdlilguys on Twitter

The `map()` family

The `purrr` package provides a family of `map` functions that are broadly used for iteration. The `map()` functions take as an argument a vector, a list, or a data frame (`.x`) along with a function (`.f`), and return an object of a type specified in a function name:

- `map(.x, .f)` returns a list
- `map_lgl(.x, .f)` returns a logical vector
- `map_int(.x, .f)` returns an integer vector
- `map_dbl(.x, .f)` returns a double vector
- `map_chr(.x, .f)` returns a character vector

The `map()` family

We can rewrite the `for` loop we wrote above with a `map_dbl()` function:

```
f <- function(x) (x+1)^2
map_dbl(x, f)
```

```
# [1]  1  4 16 81
```

The map() family

We can rewrite the for loop we wrote above with a map_dbl() function:

```
f <- function(x) (x+1)^2
map_dbl(x, f)
```

```
# [1]  1  4 16 81
```

You don't have to create a new function; you can pass an anonymous function as an argument instead:

```
map_dbl(x, function(i) (i+1)^2)
```

```
# [1]  1  4 16 81
```

Run multiple regressions with map()

Writing an empirical paper, you will need to run multiple specifications of your regressions trying to convince your future readers (and yourself) that your results are robust. Constantly copying and pasting `m = lm(y ~ x, data = df)` might be a bad coding practice. Try to use the `map()` function instead.

```
data(mtcars)

formulas <- list(
  mpg ~ hp,
  mpg ~ hp + wt,
  mpg ~ hp + wt + I(wt^2)
)

models = map(formulas, function(x) lm(x, data = mtcars))
```

Run multiple regressions with map()

Writing an empirical paper, you will need to run multiple specifications of your regressions trying to convince your future readers (and yourself) that your results are robust. Constantly copying and pasting `m = lm(y ~ x, data = df)` might be a bad coding practice. Try to use the `map()` function instead.

```
data(mtcars)

formulas <- list(
  mpg ~ hp,
  mpg ~ hp + wt,
  mpg ~ hp + wt + I(wt^2)
)

models = map(formulas, function(x) lm(x, data = mtcars))
```

```
class(models)
```

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

Run multiple regressions with map()

```
library(stargazer)
stargazer(models[[1]], models[[2]], models[[3]])
```

Table 1:

	Dependent variable:		
	mpg		
	(1)	(2)	(3)
hp	-0.068^{***}	-0.032^{***}	-0.027^{***}
	(0.010)	(0.009)	(0.008)
wt		-3.878^{***}	-10.822^{***}
		(0.633)	(2.281)
I(wtˆ2)			0.982^{***}
			(0.313)
Constant	30.099^{***}	37.227^{***}	47.837^{***}
	(1.634)	(1.599)	(3.659)
Observations	32	32	32
Adjusted R^2	0.589	0.815	0.858
Note:	*p<0.1; **p<0.05; ***p<0.01		

Exercises

1. Find the median of all columns in the `mtcar` data set and store the results in a vector.
2. Check which columns in the `iris` data set are numeric and store the results in a vector (hint: to load a built-in R data set, use `data()`; to check if an object is numeric use `is.numeric()`).
3. Write a function that takes a data set as an argument, identifies numeric columns, and returns a vector of their medians. Apply this function to the `iris` data set.

Exercises

1. Find the median of all columns in the `mtcar` data set and store the results in a vector.

```
a = map_dbl(mtcars, median)
a
```

```
#     mpg     cyl    disp      hp    drat      wt    qsec
# 19.200   6.000 196.300 123.000   3.695   3.325  17.710
#     vs      am    gear    carb
# 0.000   0.000   4.000   2.000
```

Exercises

2. Check which columns in the iris data set are numeric and store the results in a vector.

```
b = map_lgl(iris, is.numeric)
b
```

```
#  Sepal.Length  Sepal.Width Petal.Length  Petal.Width
#          TRUE         TRUE         TRUE         TRUE
#       Species
#         FALSE
```

Exercises

3. Write a function that takes a data set as an argument, identifies numeric columns, and returns a vector of their medians. Apply this function to the `iris` data set.

```
num_med = function(df){
  numeric_cols = map_lgl(df, is.numeric)
  df_num = df[, numeric_cols]
  med = map_dbl(df_num, median)
  return(med)
}

c = num_med(iris)
c
```

```
# Sepal.Length  Sepal.Width Petal.Length  Petal.Width
#         5.80         3.00         4.35         1.30
```

Creating data frames with map_dfr() and map_dfc()

The map_df_() functions produce data frames instead of lists and vectors. They bind individual outputs by rows (hence `dfr`) or columns (hence `dfc`). This type of functions can be useful for reading multiple files into R or summarizing data frames.

```
# this line gets the names of all csv files
# in a specified folder and saves them in a list
filenames = list.files(path = "data/districts/",
                       pattern=".csv")
# this line combines each file name with a path to a file
files = paste0("data/districts/", filenames, sep= "")
files[1:2]
```

```
# [1] "data/districts/alatyr.csv"  "data/districts/ardatov.csv"
```

```
# this line loads and binds our files into a single data frame
files %>% map_dfr(read.csv) %>% dim()
```

```
# [1] 2933   93
```

Producing descriptive statistics with map_dfr()

Exercise. Write a function that takes a vector as an argument and returns a named vector with a mean and a standard deviation of a vector and a number of non-missing values in it (hint: use `complete.cases()`).

Producing descriptive statistics with map_dfr()

Exercise. Write a function that takes a vector as an argument and returns a named vector with a mean and a standard deviation of a vector and a number of non-missing values in it (hint: use `complete.cases()`).

```
sumstat = function(vec){

  mean = mean(vec, na.rm = T)
  sd = sd(vec, na.rm = T)
  n = sum(complete.cases(vec))

  return(c(mean = mean, sd = sd, n = n))
}
```

Producing descriptive statistics with map_dfr()

Now we will apply the `sumstat` function to some columns in the `mtcars` data set. In a resulting data set, columns will correspond to the elements of a vector that `sumstat` returns.

Producing descriptive statistics with map_dfr()

Now we will apply the `sumstat` function to some columns in the `mtcars` data set. In a resulting data set, columns will correspond to the elements of a vector that `sumstat` returns.

```
cols = mtcars %>% select(mpg, cyl, disp)
map_dfr(cols, sumstat, .id = "var")
```

```
# # A tibble: 3 x 4
#   var     mean     sd     n
#   <chr>  <dbl>  <dbl> <dbl>
# 1 mpg    20.1    6.03    32
# 2 cyl     6.19   1.79    32
# 3 disp  231.   124.      32
```

The `.id` argument creates an identifying column with the names of elements to which we applied a function (in our case, columns of the `mtcars` data set); you need to pass it a string with a name of that column.

Data structures digression: matrices

Matrix is a two-dimensional vector, meaning all elements of a matrix must have the same type.

- To create a matrix, you need to provide a vector and specify the desired numbers of rows and columns: `matrix(vec, nrow, ncol)`.
- By default, a matrix is filled by columns, unless you explicitly specify `byrow=TRUE`.

```
matrix(1:6, nrow = 2, ncol = 3)
```

```
#      [,1] [,2] [,3]
# [1,]    1    3    5
# [2,]    2    4    6
```

```
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

```
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    4    5    6
```

Data structures digression: matrices

- Use `mat[i, j]` to access a value in row i and column j
- To perform matrix multiplication, use `mat_1 %*% mat_2` (make sure that matrices are conformable)
- `mat_1 * mat_2` performs element-wise multiplication
- Transpose: `t(mat)`
- Invert: `solve(mat)`

Another alternative to loops: the `apply` family

Base R has an alternative to the `purrr` package, which can also be used to replace loops.

Another alternative to loops: the `apply` family

Base R has an alternative to the `purrr` package, which can also be used to replace loops.

- `lapply(X, FUN)` loops over elements of a list or a vector and makes a list
  - `mclapply(X, FUN, mc.cores)` from the `parallel` package helps speed up computations by parallelizing them over multiple cores

Another alternative to loops: the `apply` family

Base R has an alternative to the `purrr` package, which can also be used to replace loops.

- `lapply(X, FUN)` loops over elements of a list or a vector and makes a list
  - `mclapply(X, FUN, mc.cores)` from the `parallel` package helps speed up computations by parallelizing them over multiple cores
- `sapply(X, FUN)` **s**implifies output of `lapply()` to a vector
  - If lists produced by `lapply()` have more than 1 element, it produces a matrix
  - `replicate(n, expr)` repeats a function `n` times; useful for random numbers generation

Another alternative to loops: the `apply` family

Base R has an alternative to the `purrr` package, which can also be used to replace loops.

- `lapply(X, FUN)` loops over elements of a list or a vector and makes a list
  - `mclapply(X, FUN, mc.cores)` from the `parallel` package helps speed up computations by parallelizing them over multiple cores
- `sapply(X, FUN)` **s**implifies output of `lapply()` to a vector
  - If lists produced by `lapply()` have more than 1 element, it produces a matrix
  - `replicate(n, expr)` repeats a function `n` times; useful for random numbers generation
- `apply(X, MARGIN, FUN)` loops over rows or columns of a matrix or a data frame
  - You need to specify the dimension over which to iterate by specifying `MARGIN = 1` for rows or `MARGIN = 2` for columns

replicate

Why simulate?

- Asses the behavior of your method
- Check that your algebra was correct
- Approximate the result when it's hard to get a closed-form solution

Simulations in R

- For the sake of reproducibility, always set a seed with `set.seed()` using any number that comes to your mind as an argument.

Simulations in R

- For the sake of reproducibility, always set a seed with `set.seed()` using any number that comes to your mind as an argument.
- You can draw a random sample from a vector or a list with or without replacement using `sample(x, size, replace = FALSE)`.

Simulations in R

- For the sake of reproducibility, always set a seed with `set.seed()` using any number that comes to your mind as an argument.
- You can draw a random sample from a vector or a list with or without replacement using `sample(x, size, replace = FALSE)`.
- You can draw (pseudo-)random samples from well-know probability distributions using the `r_()` family of functions:
  - `runif(n, min, max)` for a uniform distribution
  - `rnorm(n, mean, sd)` for a normal distribution
  - `rpois(n, lambda)` for a Poisson distribution
  - `rbinom(prob)` for a normal distribution

Simulations in R

- For the sake of reproducibility, always set a seed with `set.seed()` using any number that comes to your mind as an argument.
- You can draw a random sample from a vector or a list with or without replacement using `sample(x, size, replace = FALSE)`.
- You can draw (pseudo-)random samples from well-know probability distributions using the `r_()` family of functions:
  - `runif(n, min, max)` for a uniform distribution
  - `rnorm(n, mean, sd)` for a normal distribution
  - `rpois(n, lambda)` for a Poisson distribution
  - `rbinom(prob)` for a normal distribution
- Sample from a multivariate normal distribution with a specified covariance structure using `mvrnorm(n = 1, mu, Sigma)` from the `MASS` package

Simulations in R

Code below draws a sample of size 10 from a normal distribution $N \sim (2, 9)$. Notice that `rnorm()` takes standard deviation as an argument.

```
set.seed(1913)
rnorm(10, mean = 2, sd = 3)
```

```
[1] 2.5518011 1.4510111 2.4024628 7.8563368 5.8685697
[5] 1.2261925 4.4455243 1.4427417 1.2056557 0.4351447
```

Simulations in R

Code below draws a sample of size 10 from a normal distribution $N \sim (2, 9)$. Notice that `rnorm()` takes standard deviation as an argument.

```
set.seed(1913)
rnorm(10, mean = 2, sd = 3)
```

```
[1] 2.5518011 1.4510111 2.4024628 7.8563368 5.8685697
[5] 1.2261925 4.4455243 1.4427417 1.2056557 0.4351447
```

Exercise. Write code that simulates four samples of size 10 from $N \sim (2, 9)$ and stores them in a matrix.

Simulations in R

Exercise. Write a code that simulates four samples of size 10 from $N \sim (2, 9)$ and stores them in a matrix. Compute the standard deviation of all samples.

```
set.seed(1913)
mat = replicate(4, rnorm(10, mean = 2, sd = 3))
mat
```

```
#              [,1]       [,2]       [,3]       [,4]
#  [1,] 2.5518011  3.0784887 -2.6326973  4.3435190
#  [2,] 1.4510111 -0.8759148 -0.1355100  2.1731272
#  [3,] 2.4024628  0.3439390  0.1473025 -2.9593394
#  [4,] 7.8563368 -1.1193782  5.4052806  0.7282763
#  [5,] 5.8685697  4.8227872  0.9050462 -4.1379068
#  [6,] 1.2261925  1.3159166  0.3614067  8.3662218
#  [7,] 4.4455243  5.1789005  0.6687443 -0.9804170
#  [8,] 1.4427417 -1.5692519  7.2304689  2.4781997
#  [9,] 1.2056557  7.3114470 -5.1163414  1.0089817
# [10,] 0.4351447  7.6480081  0.6600551  0.7892954
```

Simulations in R

Exercise. Write a code that simulates four samples of size 10 from $N \sim (2, 9)$ and stores them in a matrix. Compute the standard deviation of all samples.

```
apply(mat, MARGIN = 2, sd)
```

```
# [1] 2.406124 3.488646 3.509698 3.571319
```

Simulations in R

Exercise. Write a code that simulates samples from $N \sim (2, 9)$ with the sample size ranging from 10 to 5000 with an increment of 10.

```
n = seq(10, 5000, by = 10)
```

Store the resulting samples in a list named `samples`.

Simulations in R

Exercise. Write a code that simulates samples from $N \sim (2, 9)$ with the sample size ranging from 10 to 5000 with an increment of 10.

```
n = seq(10, 5000, by = 10)
```

Store the resulting samples in a list named `samples`.

```
set.seed(1913)
samples = lapply(n, function(x) rnorm(x, 2, 3))
str(samples[1:5])
```

```
# List of 5
#  $ : num [1:10] 2.55 1.45 2.4 7.86 5.87 ...
#  $ : num [1:20] 3.078 -0.876 0.344 -1.119 4.823 ...
#  $ : num [1:30] 4.344 2.173 -2.959 0.728 -4.138 ...
#  $ : num [1:40] 8.86 7.82 1.92 6.2 5.14 ...
#  $ : num [1:50] 2.448 0.367 7.227 3.466 3.653 ...
```

Simulations in R

Exercise. Write a code that computes means of each sample in the `samples` list and stores them to a vector `means`.

Simulations in R

Exercise. Write a code that computes means of each sample in the `samples` list and stores them to a vector `means`.

```
means = sapply(samples, function(x) mean(x))
# means = map_dbl(samples, function(x) mean(x))
```

Simulations in R

Exercise. Write a code that computes means of each sample in the `samples` list and stores them to a vector `means`.

```
means = sapply(samples, function(x) mean(x))
# means = map_dbl(samples, function(x) mean(x))
```

Plot your means against the sample size using:

```
plot(n, means)
```

What do you notice?

Simulations in R

Exercise. Write a code that computes means of each sample in the `samples` list and stores them to a vector `means`.

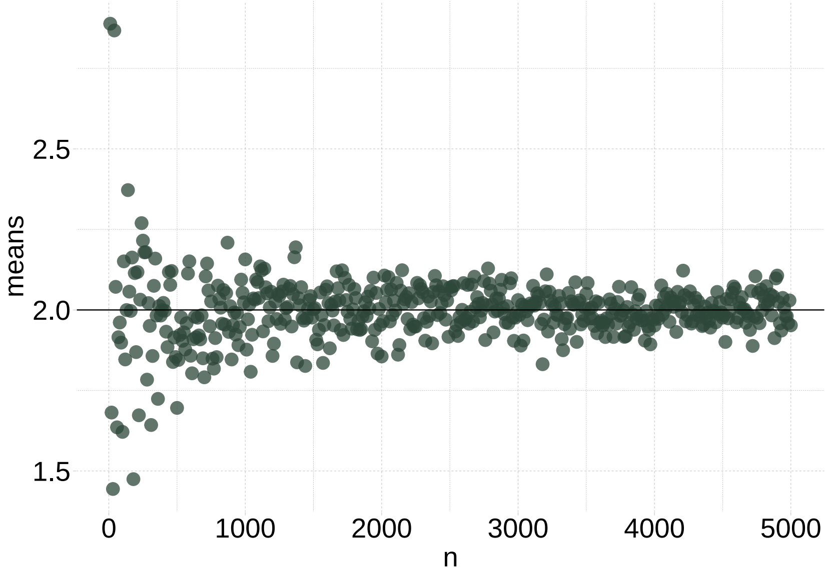
```
means = sapply(samples, function(x) mean(x))
# means = map_dbl(samples, function(x) mean(x))
```

Plot your means against the sample size using:

```
plot(n, means)
```

What do you notice? Law of large numbers

Simulations in R



Simulate from a multivariate normal

In some situations, you might want to simulate data with a pre-specified correlation structure. The `mvrnorm(n, mu, Sigma)` functions from the `MASS` package provides a neat instrument to draw correlated normally distributed samples.

Simulate from a multivariate normal

In some situations, you might want to simulate data with a pre-specified correlation structure. The `mvrnorm(n, mu, Sigma)` functions from the `MASS` package provides a neat instrument to draw correlated normally distributed samples.

- Suppose you need to draw k samples
  - `n` is a number of observations in each sample
  - `mu` is a vector of k means
  - `Sigma` is a k by k matrix that contains variances on the main diagonal and covariances off the main diagonal

Simulate from a multivariate normal

```
install.packages("MASS")
library(MASS)
```

```
set.seed(1913)
draws = mvrnorm(1000, mu = c(0, 1),
                Sigma = matrix(c(4, 2,
                                 2, 4),
                               ncol = 2, byrow = T))
```

Simulate from a multivariate normal

```
install.packages("MASS")
library(MASS)
```

```
set.seed(1913)
draws = mvrnorm(1000, mu = c(0, 1),
                Sigma = matrix(c(4, 2,
                                 2, 4),
                               ncol = 2, byrow = T))
```

Checking means

```
apply(draws, MARGIN = 2, mean)
```

```
# [1] -0.02293701  0.97332208
```

Simulate from a multivariate normal

```
install.packages("MASS")
library(MASS)
```

```
set.seed(1913)
draws = mvrnorm(1000, mu = c(0, 1),
                Sigma = matrix(c(4, 2,
                                 2, 4),
                              ncol = 2, byrow = T))
```

Checking means

```
apply(draws, MARGIN = 2, mean)
```

```
# [1] -0.02293701  0.97332208
```

Checking standard deviations

```
apply(draws, MARGIN = 2, sd)
```

```
# [1] 2.011650 2.017546
```

Simulate from a multivariate normal

```
install.packages("MASS")
library(MASS)
```

```
set.seed(1913)
draws = mvrnorm(1000, mu = c(0, 1),
                Sigma = matrix(c(4, 2,
                                 2, 4),
                               ncol = 2, byrow = T))
```

Checking means

```
apply(draws, MARGIN = 2, mean)
```

```
# [1] -0.02293701  0.97332208
```

Checking standard deviations

```
apply(draws, MARGIN = 2, sd)
```

```
# [1] 2.011650 2.017546
```

What is the correlation between the samples?

Simulate from a multivariate normal

```
install.packages("MASS")
library(MASS)
```

```
set.seed(1913)
draws = mvrnorm(1000, mu = c(0, 1),
                Sigma = matrix(c(4, 2,
                                 2, 4),
                               ncol = 2, byrow = T))
```

Checking means

```
apply(draws, MARGIN = 2, mean)
```

```
# [1] -0.02293701  0.97332208
```

Checking standard deviations

```
apply(draws, MARGIN = 2, sd)
```

```
# [1] 2.011650 2.017546
```

What is the correlation between the samples? 0.5

Simulate from a multivariate normal

Exercise. Write a function that samples n rows from `draws` and returns correlation between the samples using `cor(x, y)`. Run this function with $n = \{25, 50, 100\}$. Store results in a vector.

Simulate from a multivariate normal

Exercise. Write a function that samples n rows from `draws` and returns correlation between the samples using `cor(x, y)`. Run this function with $n = \{25, 50, 100\}$. Store results in a vector.

```
sample.cor = function(mat, n){
  id = sample(1:nrow(mat), n)
  s = mat[id, ]
  r = cor(s[,1], s[,2])
  return(r)
}
```

Simulate from a multivariate normal

Exercise. Write a function that samples n rows from `draws` and returns correlation between the samples using `cor(x, y)`. Run this function with $n = \{25, 50, 100\}$. Store results in a vector.

```
sample.cor = function(mat, n){
  id = sample(1:nrow(mat), n)
  s = mat[id, ]
  r = cor(s[,1], s[,2])
  return(r)
}
```

```
set.seed(1913)
out = map_dbl(c(25, 50, 100),
              function(x) sample.cor(draws, x))
out
```

```
# [1] 0.3893528 0.4602316 0.5264837
```

Further reads

- Hadley Wickham and Garrett Grolemund, *R for Data Science*
- Hadley Wickham, *Advanced R*