Vectors and Data Types

Earnest Salgado

10/11/2021

Contents

Warmup

1. In the lecture, we covered `c()`, `:`, `rep()`, `seq()`, `rnorm()`, `runif()` among other ways to create vectors. Use each of these functions once as you create the vectors required below.

   a. Create an integer vector from 2 to 30.

```
c(2:30)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
## [26] 27 28 29 30
```

```
b. Create a numeric vector with 60 draws from the ‘r‘andom ‘unif‘orm distribution
```

```
runif(60)
```

```
##  [1] 0.65773607 0.53362365 0.24225258 0.86635810 0.46082582 0.34368971
##  [7] 0.71957400 0.11335664 0.84938542 0.40154480 0.40909820 0.46718600
## [13] 0.86902185 0.33799385 0.12935275 0.44847607 0.75017009 0.59106156
## [19] 0.20153780 0.51135853 0.64671043 0.97185054 0.09127655 0.75520945
## [25] 0.36952501 0.55626911 0.11685529 0.92937492 0.05421800 0.51976355
## [31] 0.88522772 0.30987652 0.95355235 0.90395567 0.66711933 0.78348735
## [37] 0.92984318 0.40780228 0.10496608 0.74841144 0.01306007 0.53363228
## [43] 0.02374849 0.80085629 0.72217348 0.75620470 0.23792516 0.76417311
## [49] 0.39960052 0.81372132 0.43421472 0.82531859 0.12592431 0.88341204
## [55] 0.21846332 0.65563960 0.15749268 0.67175733 0.67940514 0.62122374
```

```
c. Create a character vector with the letter "x" repeated 1980 times.
```

```r
rep("x", 1980)
```

```
d. Create a character vector of length 5 with the items "Nothing" "works" "unless" "you" "do". Call thi
```

```r
angelou_quote <- c("Nothing", "works", "unless", "you", "do")
```

```
e. Create a numeric vector with 1e4 draws^[This is scientific notation. Try ‘1e4 - 1 + 1‘ in the consol
```

```r
rnorm(1e4)
```

```
f. Create an integer vector with the numbers 0, 2, 4, ... 20.
```

```r
seq(0, 20, length.out=11)
```

```
##  [1]  0  2  4  6  8 10 12 14 16 18 20
```

2. Try to guess the output of the following code:

```r
a <- 3
b <- a^2 + 1
```

Answer: 10

4. Now try to guess again:

```r
a <- c(1, 2, 3)
b <- a^2 + 1
```

Answer: c(2, 5, 10)

5. Will this one run? If it does, what will it return? Will R complain?

```r
a <- c(1, 2, 3)
b <- a^2 + c(1, 2)
```

Answer: It will run and return c(2, 6, 10), but R will complain because the last vector is shorter than the previous one and is not a multiple of it.

6. Finally, what about this one?

```r
a <- c(1, 2)
b <- a^2 + c(1, 2, 3, 4)
```

Answer: This one will run and return c(2, 6, 4, 8), and R will not complain because the vectors' lengths are multiples of each other—the shorter vector gets recycled for the second.

Calculating Mean and Standard Deviation

Calculating the Mean

In this exercise, we will calculate the mean of a vector of random numbers. We will practice assigning new variables and using functions in R.

We can run the following code to create a vector of 1000 random numbers. The function `set.seed()` ensures that the process used to generate random numbers is the same across computers.

Note: `rf()` is a R command we use to generate 1000 random numbers according to the F distribution, and `10` and `100` are parameters that specify how "peaked" the distribution is.

```
set.seed(1)
random_numbers <- rf(1000, 10, 100)
```

Write code that gives you the sum of `random_numbers` and saves it to a new variable called `numbers_sum`:

```
numbers_sum <- sum(random_numbers)
numbers_sum
```

```
## [1] 1018.126
```

Note: You don't automatically see the output of `numbers_sum` when you assign it to a variable. Type `numbers_sum` into the console and run it to see the value that you assigned it.

Write code that gives you the number of items in the `random_numbers` vector and saves it to a new variable called `numbers_count`:

```
numbers_count <- length(random_numbers)
```

Hint: To count the number of items in a vector, use the `length()` function.

Now write code that uses the above two variables to calculate the average of `random_numbers` and assign it to a new variable called `this_mean`.

```
this_mean <- numbers_sum/numbers_count
this_mean
```

```
## [1] 1.018126
```

What number did you get? It should have been 1.018. If it isn't, double check your code!

R actually has a built in function to calculate the mean for you, so you don't have to remember how to build it from scratch each time! Check your above answer by using the `mean()` function on the `random_numbers` vector.

```
mean(random_numbers)
```

```
## [1] 1.018126
```

Calculating the Standard Deviation

Now that you've got that under your fingers, let's move on to standard deviation.

We will be converting the following formula for calculating the sample standard deviation into code:

$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}}$

For this, we'll review the concept of *vectorization*. This means that an operation like subtraction will act on all numbers in a vector at the same time.

Subtract `this_mean` from the `random_numbers` vector. Did each number in `random_numbers` change?

```
vectorized <- random_numbers - this_mean
```

Answer: Each number in the vector `random_numbers` did not change, because it's particular value is saved in our global environment from previous.

Try to write the formula for standard deviation in R code using the `sqrt()`, `sum()`, and `length()` functions, along with other operators (`^`, `/`, `-`). Assign it to a new variable called `this_sd`. Watch out for your parentheses!

```
this_sd <- sqrt(sum((random_numbers - this_mean) ^ 2) / (length(random_numbers) - 1))

this_sd
```

```
## [1] 0.489704
```

Hint: Fill in the following formula: _____ <- sqrt(sum((_____ - this_mean) ^ 2) / (length(_____) - 1))

What number did you get for `this_sd`, or the standard deviation of `random_numbers`? If you didn't get 0.489704, recheck your code!

R also has a built in function for standard deviation. Check if you calculated the standard deviation correctly by using the `sd()` function on the `random_numbers` vector.

```
sd(random_numbers)
```

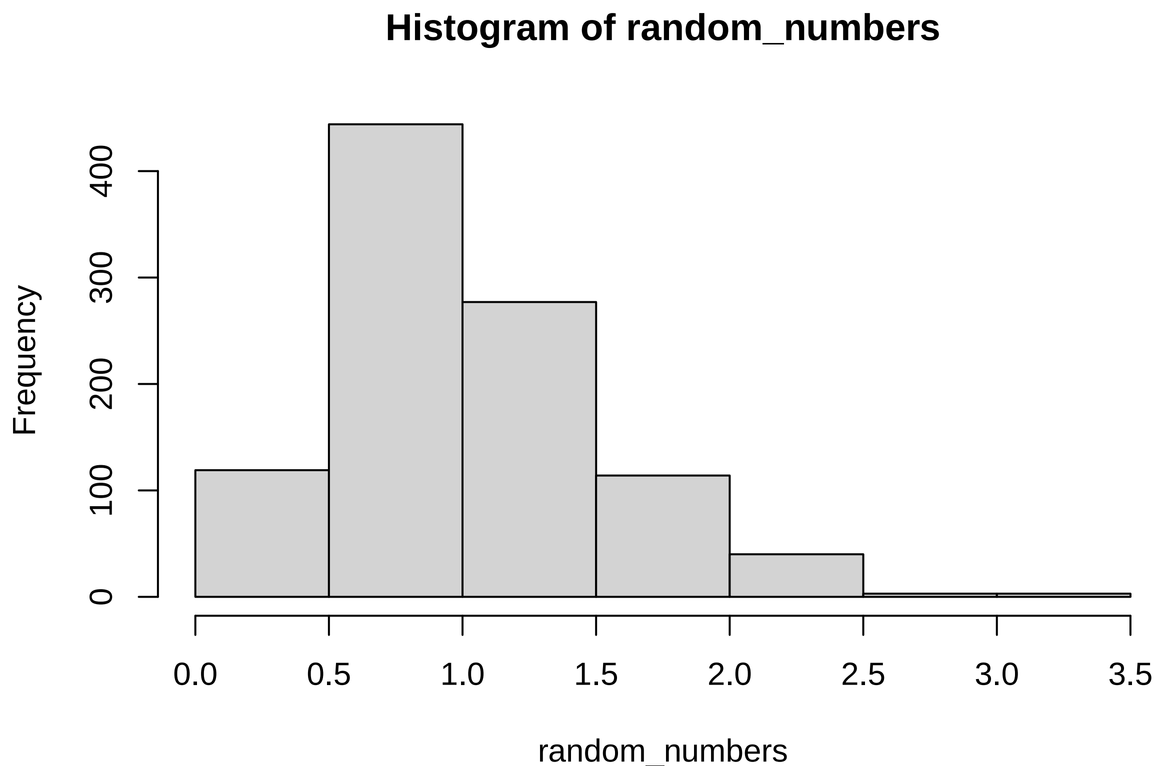
```
## [1] 0.489704
```

Making a Histogram of Our Numbers

What do these random numbers look like, anyway? We can use base plotting in R to visualize the distribution of our random numbers.

1. Run the following code to visualize the original distribution of `random_numbers` as a histogram.

```
hist(random_numbers)
```



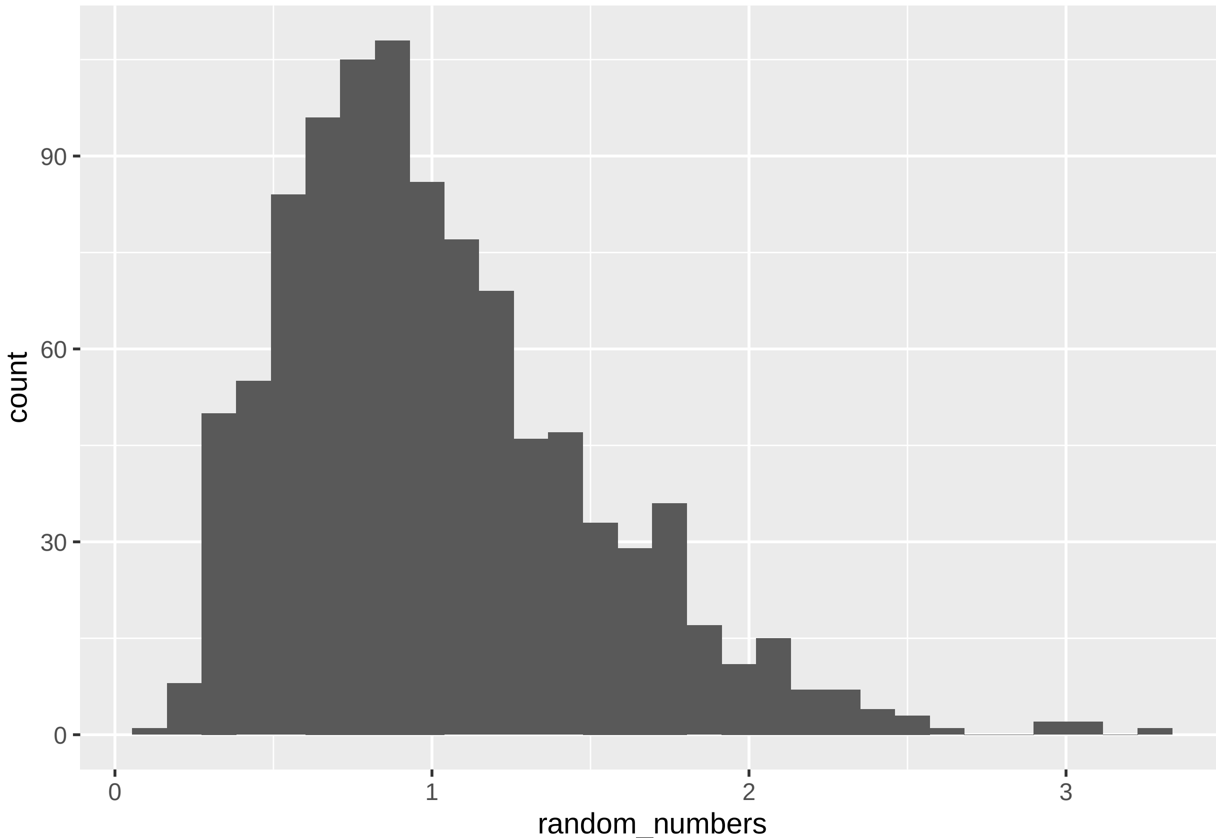
2. You could also visualize this with ggplot but there are some extra steps. ggplot typically expects a data frame (tibble) in the first position so we need to explicitly tell it that the first position has the aesthetic mapping.

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.0.5
```

```
ggplot(mapping = aes(x = random_numbers)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Notice how most of the values are concentrated on the left-hand side of the graph, while there is a longer "tail" to the right? Counterintuitively, this is known as a right-skewed distribution. When we see a distribution like this, one common thing to do is to normalize it.

This is also known as *calculating a z-score*, which we will cover next.

Calculating a Z-Score

The formula for calculating a z-score for a single value, or *normalizing* that value, is as follows:

$z = \frac{x - \bar{x}}{s}$

This can be calculated for each value in `random_numbers` in context of the larger set of values.

Can you translate this formula into code?

Using `random_numbers`, `this_mean`, and `this_sd` that are already in your environment, write a formula to transform all the values in `random_numbers` into z-scores, and assign it to the new variable `normalized_data`.

```
normalized_data <- (random_numbers - this_mean) / this_sd
```

Hint: R is vectorized, so you can subtract the mean from each random number in `random_numbers` in a straightforward way.

Take the mean of `normalized_data` and assign it to a variable called `normalized_mean`.

Note: If you see something that ends in "e-16", that means that it's a very small decimal number (16 places to the right of the decimal point), and is essentially 0.

```r
normalized_mean <- sum(normalized_data) / length(normalized_data)
# or
normalized_mean <- mean(normalized_data)
```

Take the standard deviation of `normalized_data` and assign it to a variable called `normalized_sd`.

```r
normalized_sd <- sqrt(sum((normalized_data) ^ 2) / (length(normalized_data) - 1))
# or
normalized_sd <- sd(normalized_data)
```

What is the value of `normalized_mean`? What is the value of `normalized_sd`? You should get a vector that is mean zero and has a standard deviation of one, because the data has been normalized.

```r
normalized_mean
```

```
## [1] -1.553267e-16
```

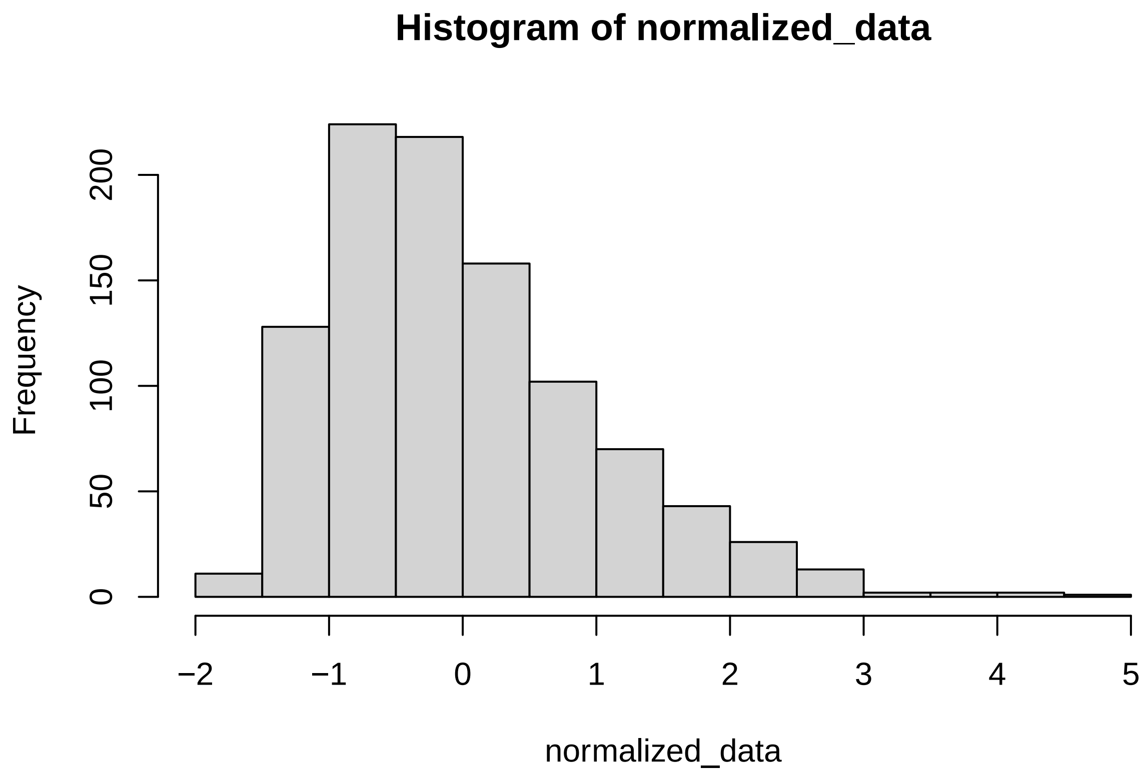
```r
normalized_sd
```

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

Making a Histogram of Z-scores

Let's plot the z-scores and see if our values are still skewed. How does this compare to the histogram of `random_numbers`?

```r
hist(normalized_data)
```



Is the resulting data skewed?

Answer: This is an example of right-skew distribution of data. We can calculate the mean of this data and see that it is to the right of the median.

Calculating a T-Score

T-tests are used to determine if two sample means are equal. The formula for calculating a t-score is as follows:

$t = \frac{\overline{x}_1 - \overline{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$

where $\overline{x}_i$ is the mean of the first or second set of data, s_i is the sample standard deviation of the first or second set of data, and n_i is the sample size of the ith set of data.

We'll first create two data sets of random numbers following a normal distribution:

```r
set.seed(1)
data_1 <- rnorm(1000, 3)
data_2 <- rnorm(100, 2)
```

Here's how we'll calculate the mean (`x_1`), standard deviation (`s_1`), and sample size (`n_1`) of the first data set:

```r
x_1 <- mean(data_1)
s_1 <- sd(data_1)
n_1 <- length(data_1)
```

What numeric types do you get from doing this? Try running the `typeof()` function on each of `x_1`, `s_1`, and `n_1`. We have you started with `x_1`.

```
typeof(x_1)
```

```
## [1] "double"
```

```
typeof(s_1)
```

```
## [1] "double"
```

```
typeof(n_1)
```

```
## [1] "integer"
```

What object type is `n_1`? **Answer:** `n_1` is "integer" type

Can you calculate the same values for `data_2`, assigning mean, standard deviation, and length to the variables of `x_2`, `s_2`, and `n_2`, respectively?

```
x_2 <- mean(data_2)
s_2 <- sd(data_2)
n_2 <- length(data_2)
```

What values do you get for `x_2` and `s_2`?

```
x_2
```

```
## [1] 1.989683
```

```
s_2
```

```
## [1] 1.029709
```

Answer: We get 1.99 and 1.03 for `x_2` and `s_2` respectively

Now, you should be able to translate the t-score formula ($\frac{\overline{x}_1 - \overline{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$) into code, based on the above calculated values.

```
t_score <- (x_1 - x_2)/(sqrt((s_1^2/n_1 + s_2^2/n_2)))
t_score
```

```
## [1] 9.242949
```

What did you get for the t-score? You should have gotten 9.243, if not, double check your code!

The t-score's meaning depends on your sample size, but in general t-scores close to 0 imply that the means are not statistically distinguishable, and large t-scores (e.g. t > 3) imply the data have different means.

Performing a T-Test

Once again, R has a built in function that will perform a T-test for us, aptly named `t.test()`. Look up the arguments the function `t.test()` takes, and perform a T-test on `data_1` and `data_2`.

```
t.test(data_1, data_2)
```

```
##
##  Welch Two Sample t-test
##
## data:  data_1 and data_2
## t = 9.2429, df = 119.89, p-value = 1.099e-15
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.7847422 1.2125955
## sample estimates:
## mean of x mean of y
##  2.988352  1.989683
```

What are the sample means, and are they distinguishable from each other?

Answer: The mean of `data_1` is 2.988, the mean of `data_2` is 1.990, and they are distinguishable from each other.

Well done! You've learned how to work with R to calculate basic statistics. We've had you generate a few by hand, but be sure to use the built-in functions in R in the future.

Extra time? Try writing some of this code into a new R script on your own computer.

Want to improve this tutorial? Report any suggestions/bugs/improvements on Github here! We're interested in learning from you how we can make this tutorial better.