Lecture 4: R-Review-2023

Code

Lifeng Ren

2023-08-17

1 Course Updates

- Lecture 2 and Lecture 3's link is now available on GitHub
- Lecture 3's project solution has been updated.
- Any Questions? Comments? Suggestions?

2 Recap from Yesterday

- Compare with `1:m`, `m:1`, `1:1` in STATA
- Data Reshaping
- Library and Packages
- Data Standardization
- Unique Identifier
- Basic Data Analysis and Visualization in R
- A review project for Day 1 and Day 2
    - Don't worry if you did not finish it
    - Many things will be repeated today and tomorrow.
    - If we end up earlier today, you are feel free to stay and do it.

3 Introduction

In this lecture, we'll build on what we've already covered by placing it in an economic context. This approach is designed to facilitate a seamless transition into your first-year econometrics class. You may notice that some content in this lecture mirrors or closely resembles previous lessons. This repetition is

intentional, aiming to reinforce your understanding and help you consolidate the concepts we've explored.

In this class, I won't directly provide the code. Instead, you'll be using the `Rmd` file I've supplied to craft your code. We'll progress to subsequent topics only after everyone has successfully executed their code.

Topics

- Mathematical Statistics in R
- Intermediate Applied Econometrics (Regression Related) in R

---

4 Mathematical Statistics in R

4.1 Matrix Caculations (Warm up and Review)

This is something we've learned before. So, go ahead and use the rmd file named `lec4_stu.Rmd` to use R to represent, calculate the following Math Expressions.

4.1.1 Mathematical Expression | 4.1.2 R Code

Matrix Creation

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Create the matrix in two ways:

- `matrix(c(), nrow=2)`
- generate an empty matrix and use a loop to fill in each entry of A

Matrix Addition Given two matrices A and B,

$$C = A + B$$

Matrix Multiplication with Scalar

Given scalar k and matrix A,

$$D = k \times A$$

Matrix Dot Product Given matrices A and B,

$$E = A \cdot B$$

Matrix Transpose Given matrix A,

$$F = A^T$$

Determinant and Inverse Given matrix A,

$$\det(A)$$

Given matrix A (and assuming it's invertible),

$$G = A^{-1}$$

Eigenvalues and Eigenvectors

Given matrix A, Eigenvalues (λ) and Eigenvectors (v) satisfy:

$$Av = \lambda v$$

Double Sum with Indexing Let's consider two matrices, A and B, and calculate the double sum:

$$\sum_i \sum_j A_{ij} + B_{ij}$$

$\sum_i$ is iterating through rows, $\sum_j$ is iterating columns. Suppose:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

,

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Note: 1)You should get a single value. 2)Do it in a customized function named as `double_sum`.

4.2 Sampling

In R, the `sample()` function allows us to generate random samples. Imagine we have an urn containing 12 distinct colored balls. If we were to draw a ball 10 times, replacing the ball after each draw, we can simulate this process to create a sample.

Since it is randomly generating the sample for us, how can we keep track on it?

Use the `set.seed()` function.

Hide

```r
set.seed(123)

# Define the 12 distinct colored balls
colors <- c("Red", "Blue", "Green", "Yellow", "Purple", "Orange",
            "Pink", "Brown", "Black", "White", "Gray", "Cyan")

# Draw 10 balls with replacement
sampled_colors <- sample(colors, 10, replace = TRUE)
sampled_colors
```

```
##  [1] "Green"  "Green"  "White"  "Blue"   "Orange" "Gray"   "Purple" "Yellow"
##  [9] "Orange" "Black"
```

To calculate the probability of drawing any single color from the urn when drawing one ball (without replacement), you can use basic probability principles.

Given that there are 12 distinct colored balls in the urn, the probability P of drawing any one specific color (let's say "Red") in one draw is:

$$P(\text{Red}) = \frac{\text{Number of Red balls}}{\text{Total number of balls}}$$

If each color appears only once in the urn, then:

$$P(\text{Red}) = \frac{1}{12}$$

This is true for any individual color in the urn, assuming each color is unique and there's only one ball of each color.

To calculate this in R:

Hide

```r
# Total number of distinct colored balls
total_balls <- length(colors)

# Probability of drawing one specific color
probability_single_color <- 1 / total_balls
probability_single_color
```

```
## [1] 0.08333333
```

This will give you the probability of drawing any single specific color from the urn in one draw.

4.3 Replication

The `rep()` function in R stands for "replicate." It is used to replicate the values of vectors or lists.

Here's a brief explanation of its functionality:

1. **Basic Replication**: If you want to replicate a single value multiple times, you can use `rep()`:

Hide

```
rep(5, times = 3)
```

```
## [1] 5 5 5
```

This creates a vector with the number 5 repeated three times.

2. **Replicating Vectors**: You can also replicate entire vectors:

Hide

```
rep(1:3, times = 2)
```

```
## [1] 1 2 3 1 2 3
```

This replicates the entire vector `1:3` two times.

3. **Different Replications for Each Element**: By using the `each` argument, you can specify how many times each element should be replicated:

Hide

```
rep(1:3, each = 2)
```

```
## [1] 1 1 2 2 3 3
```

This replicates each element of the vector `1:3` two times.

4. **Combining `times` and `each`**: You can also combine both `times` and `each` for more complex patterns:

Hide

```
rep(1:2, times = 2, each = 3)
```

```
##  [1] 1 1 1 2 2 2 1 1 1 2 2 2
```

Here, each element of the vector `1:2` is replicated three times, and then the entire pattern is replicated twice.

In the context of the urn example, `rep()` was used to represent the contents of the urn by replicating specific colors according to their quantities. For instance, if the urn had 3 Red balls, 2 Blue balls, and 1 Green ball, we would use `rep()` to create a vector representing this distribution:

Hide

```
urn_contents <- rep(c("Red", "Blue", "Green"), times = c(3, 2, 1))
```

This would produce a vector: `Red Red Red Blue Blue Green`.

4.4 Discrete Random Variables and Distribution

In many cases, you will see $X \sim (,)$, which is typically used to denote that random variable X follows a certain distribution.

To describe a discrete random variable and its distribution, you specify: 1. The set of possible values it can take. 2. The probability associated with each value.

Let's illustrate this with two common discrete probability distributions: the Bernoulli distribution and the Binomial distribution.

4.4.1 Bernoulli Distribution:

A Bernoulli random variable X can take on two values, usually 0 and 1, with probabilities p and $1 - p$, respectively. This distribution describes a single experiment with two outcomes, often termed "success" and "failure".

Notation: $X \sim \text{Bernoulli}(p)$

4.4.2 Binomial Distribution:

A Binomial random variable X represents the number of successes in n Bernoulli trials. Each trial is independent, and the probability of success remains constant across trials.

Notation: $X \sim \text{Binomial}(n, p)$

In R, you can work with these distributions using functions like `dbinom()`, `pbinom()`, `qbinom()`, and `rbinom()` for the Binomial distribution, and similar functions for other distributions. For example,

4.4.3 Binomial Distribution:

Generating a binomial random variable with $n = 5$ and $p = 0.5$:

Hide

```
rbinom(1, size=5, prob=0.5)
```

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

4.5 Continous Random Variables and Distribution

Unlike discrete random variables, which take on a finite or countably infinite number of distinct values, continuous random variables can take on an uncountably infinite number of possible values. This means they can take on any value within a specified range.

For a continuous random variable, you specify: 1. The set of possible values it can take, which is usually an interval or union of intervals on the real line. 2. The probability density function (pdf) which provides the likelihood of the variable lying within a particular range.

Let's illustrate this with two common continuous probability distributions: the Uniform distribution and the Normal distribution.

4.5.1 Uniform Distribution:

A continuous random variable X that is uniformly distributed takes on values within a specified range $[a, b]$ with equal probability.

Notation: $X \sim U(a, b)$

Probability Density Function (pdf):

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

4.5.2 Normal Distribution:

A normal (or Gaussian) random variable X has a bell-shaped probability density. It is characterized by two parameters: the mean μ and the variance σ^2.

Notation: $X \sim N(\mu, \sigma^2)$

Probability Density Function (pdf):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

In R, you can work with these distributions using functions like `dunif()`, `punif()`, `qunif()`, and `runif()` for the Uniform distribution, and `dnorm()`, `pnorm()`, `qnorm()`, and `rnorm()` for the Normal distribution.

4.5.3 Example: Normal Distribution

Generating a normal random variable with $\mu = 0$ and $\sigma^2 = 1$:

Hide

```
rnorm(1, mean=0, sd=1)
```

```
## [1] 1.280555
```

4.6 Monte Carlo Simulation

Monte Carlo method by approximating the value of π using a simple geometric approach. This is a classic example of Monte Carlo simulation.

Concept:

Imagine a circle inscribed inside a square. The ratio of their areas is $\frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$. If we randomly throw points into the square, the probability that a point lands inside the circle is $\frac{\pi}{4}$. Therefore, $\pi \approx 4\times$ (number of points inside the circle) / (total number of points thrown).

Steps: 1. Randomly generate points inside the square. 2. Determine how many of those points fall inside the circle. 3. Approximate π using the ratio of points inside the circle to the total points.

4.7 R Code with loops and `runif()`:

Hide

```
set.seed(123)  # for reproducibility

# Number of random points
n_points <- 10000

# Counter for points inside the circle
points_inside <- 0

# Loop to generate points and check if they are inside the circle
for (i in 1:n_points) {

  # Generate random x and y coordinates using runif()
  x <- runif(1, -1, 1)
  y <- runif(1, -1, 1)

  # Check if point is inside the circle (x^2 + y^2 <= 1)
  if (x^2 + y^2 <= 1) {
    points_inside <- points_inside + 1
  }
}

# Approximate pi
approximated_pi_loop_runif <- 4 * points_inside / n_points
approximated_pi_loop_runif
```

```
## [1] 3.1416
```

4.8 Exercise

In many applications, particularly in simulations or Monte Carlo methods, the parameters of a distribution might not be constants but could be variables themselves. This can be particularly useful when modeling uncertainty in parameters.

4.8.1 Question statment | 4.8.2 Solutions

Generating Random Variables with Variable Inputs in R

Suppose we're modeling the scores of students on a test, and we believe the scores are normally distributed around an average of 80. However, we think that the variability (standard deviation) of scores could itself be a random variable, perhaps based on different teaching methods or study materials used.

Let's say the standard deviation follows a discrete distribution where:

- There's a 50% chance that the standard deviation (reflecting the variability of scores) is 5, which corresponds to the standard teaching method.
- There's an equal 50% chance that the standard deviation is 10, representing the new experimental teaching method.

Generate a sample of student scores using `rnorm()` with this variable input `teaching_methods`

Hints and Steps:

- Sample the teaching method first:
  - `sd` could either be 5 or 10. Then your code should have a vector of `c(5,10)`
  - size of the sample: 1000
  - replacement? YES!
  - probability? (0.5,0.5)
  - save this sample into `teaching_methods`
- Use `rnorms` to generate scores, and in your `rnorm()` arguments, try two things:
  - let `sd = teaching_methods`
  - let `sd = mean(teaching_methods)`
- What are the difference?

4.9 Expected Values, Variance

Just like the results from the Exercise above, once we have a random variable X, we can always find the expected values of it $E[X]$ and the variance of it $Var[X]$. How to take those values? For the random variable `teaching_methods` , we can use `mean()` and `var()`

Hide

```
mean(teaching_methods)
```

```
## [1] 7.535
```

Hide

```
var(teaching_methods)
```

```
## [1] 6.25503
```

Hide

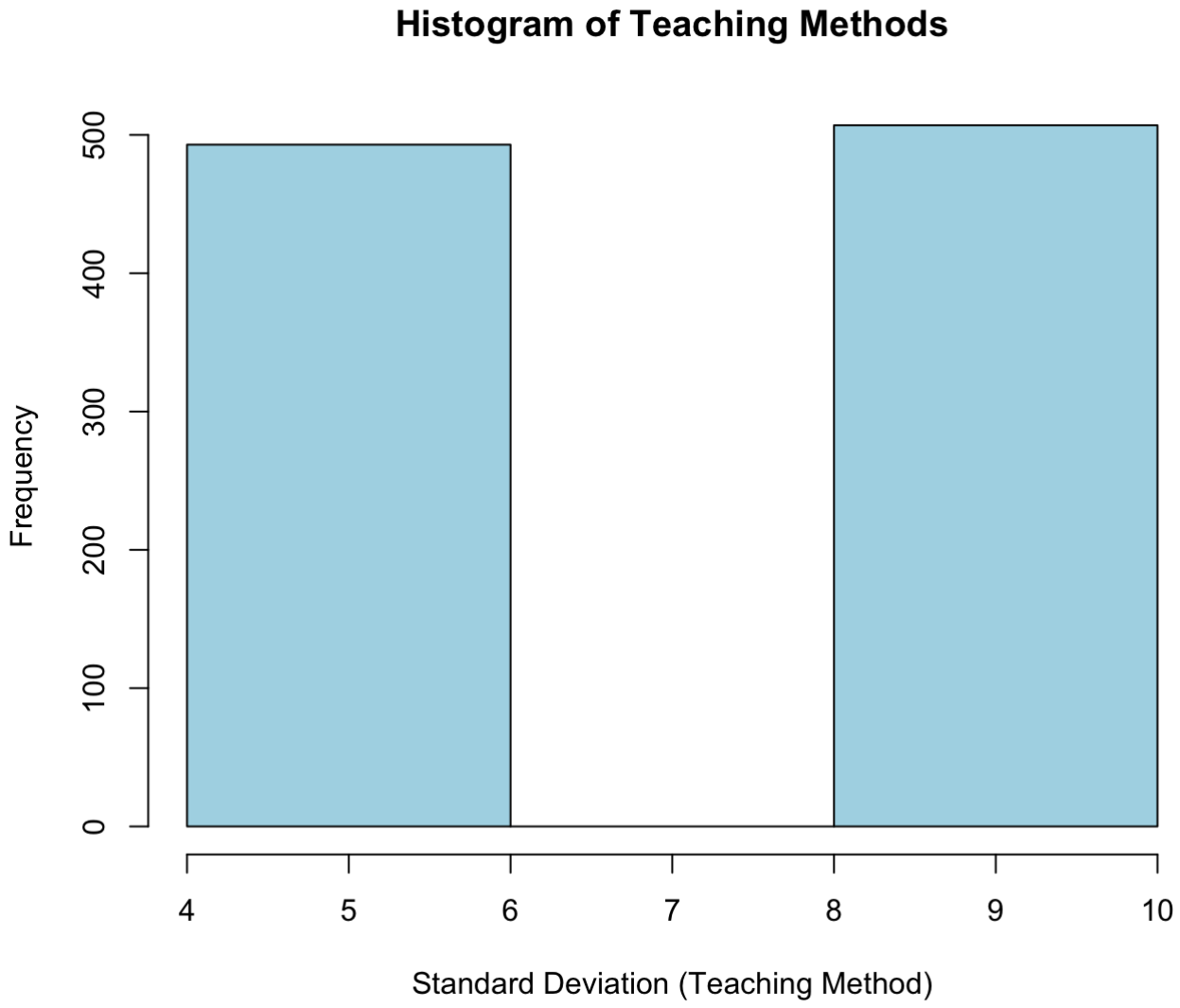
```
if (sqrt(var(teaching_methods)) == sd(teaching_methods)) {
    print(sd(teaching_methods))
}
```

```
## [1] 2.501006
```

4.10 Plot the distribution of random variable

Hide

```
hist(teaching_methods, main="Histogram of Teaching Methods",
     xlab="Standard Deviation (Teaching Method)", ylab="Frequency",
     col="lightblue", border="black", breaks=2)
```



5 Regression-Related Functions in R

Regression analysis is a powerful statistical method that allows us to examine the relationship between two or more variables of interest. In R, we often use the `lm()` function for linear regression. However, to dive deeper into the results, especially the coefficients and their standard errors, we need to be familiar with matrix operations and loops.

5.1 **Econometrics Expression :** | 5.2 R code

1. Linear Regression in R

We'll begin with a simple linear regression using R's built-in dataset `mtcars` .

Consider a linear regression model of the `mpg` (miles per gallon) as a function of `wt` (weight) and `hp` (horsepower) using the `mtcars` dataset.

The mathematical equation representing this linear regression model is:

$$\text{mpg} = \beta_0 + \beta_1 \times \text{wt} + \beta_2 \times \text{hp} + \epsilon$$

Where: - mpg is the dependent variable (miles per gallon). - wt is the weight of the car. - hp is the horsepower of the car. - β_0 is the intercept. - β_1 is the coefficient for weight (`wt`). - β_2 is the coefficient for horsepower (`hp`). - ϵ represents the error term, capturing the variability not explained by weight and horsepower.

Task:

- Use R code to run this linear regression model
- save this model called `model`

2. Extracting Coefficients and Standard Errors

Once you have your regression model, you might want to extract the coefficients and standard errors for further analysis.

Task:

- Save the coefficients from the linear regression, called it `coefficients`
- From the coefficient's matrix, get the standard error vector: `std_errors`

3. Matrix Calculations using Loops

Let's assume you want to multiply each coefficient with its corresponding standard error.

Task:

- Use `numeric()` and `length()` functions to generate an empty vector called `result` that has the same length with `coefficients` .
- Using a for loop to multiply each coefficient with its corresponding standard error and save the multiplied result in `result`
- (Optional) Create a function `coef_times_se` that takes a linear model as input and returns the coefficients multiplied by their standard errors.

4. Advanced Matrix Operations

Compute the dot product of the coefficients and standard errors:

- Transpose your coefficients first!
- (Optional) Create a function `coef_dot_se` that takes a linear model as input and returns the dot product of the coefficients and their standard errors.

5. Custom Functions with Loops for Regression Results

Create a function `coef_se_square` that squares each coefficient and multiplies it by the square of its standard error.

- function input: `model` you saved

6. Basic Visualization

1. **Scatter plot with regression line**:
   - Produce a scatter plot with `wt` on the x-axis and `mpg` on the y-axis.
   - Color the points by `hp` (horsepower) to visualize its effect.
   - Add the regression line to this plot.
2. **Descriptive statistics table**:
   - Create a table that displays descriptive statistics (mean, median, standard deviation) for `mpg`, `wt`, and `hp`.
3. **Regression results table using `within()`**:
   - Produce a table displaying the regression results, including coefficients, standard errors, t-values, and p-values.
   - Use the `within()` function for this task:
     - First, consult `?within()` to understand the syntax.
     - In essence:
       - Transform the coefficients into a dataframe using `as.data.frame()`.
       - Follow the format:
         `reg_tab = within(dataframe, {column_name <- coef_vals[,] })`.

---

6 Reference

- Dr. Qingxiao Li's notes for R-Review 2020
- Rodrigo Franco's notes for R-Review 2021