1.4 for Loops

We end this introductory chapter on R programming with a brief discussion on `for` loops, which allow us to execute a series of statements repetitively with a minimal amount of code for a pre-specified number of times. Although `for` loops are not heavily used in Exam PA (so you may not want to spend too much time learning these devices), they play an important role in simulation studies and are good programming tools to know in general.

Basic structure. The general syntax of a `for` loop is as follows:

```
for (<var> in <vec>) {
  <statement_1>
  <statement_2>
  ...
}
```

where

`<var>` is the index variable to loop through,

`<vec>` is a numeric or character vector that carries the possible values of `var` over different iterations of the loop, and

`<statement_1>`, `<statement_2>`, ... are statements that will be run for each value of `var` in the order listed in `vec`.

As a simple example, let's use a `for` loop to construct a function evaluating the partial sum

$$\sum_{i=1}^{n} i \qquad \left(= \frac{n(n+1)}{2}\right)$$

for every positive integer n. Run CHUNK 1 to create such a function named `psum` and calculate the partial sum for $n = 1, 3$, and 10.

```
# CHUNK 1
psum <- function(n) {
  s <- 0
  for (i in 1:n) {
    s <- s + i
  }
  return(s)  # don't miss the return value
}

psum(1)

## [1] 1

psum(3)

## [1] 6

psum(10)

## [1] 55
```

The `psum()` function computes the partial sums recursively as follows:

$$\begin{aligned}
s_0 &= 0 && \text{(starting value)}\\
s_1 &= s_0 + 1 = 0 + 1 = 1, && (i = 1)\\
s_2 &= s_1 + 2 = 1 + 2 = 3, && (i = 2)\\
s_3 &= s_2 + 3 = 3 + 3 = 6, && (i = 3)\\
&\vdots && \vdots\\
s_{10} &= s_9 + 10 = 45 + 10 = 55. && (i = 10)
\end{aligned}$$

Note that a `for` loop itself does not return any output. It is thus important to include the final value of the partial sum with the command `return(s)`, without which the `psum()` function will not output anything.

For this particular task, it turns out that simpler code based on vectorization can be used, as CHUNK 2 shows.

```
# CHUNK 2
psum.vec <- function(n) {
  x <- 1:n
  return(sum(x))
}

psum.vec(1)

## [1] 1

psum.vec(3)

## [1] 6

psum.vec(10)

## [1] 55
```

[HARDER!] Typical application of `for` loops: Simulation studies. We now see a statistical application for which the use of a `for` loop seems indispensable: To undertake simulation studies. As we have learned from Exam MFE/IFM, simulation entails generating repeated random samples and using these samples to estimate probabilistic quantities of interest. Here we will apply simulation to illustrate a fundamental concept we may have overlooked in our VEE Mathematical Statistics course: The genuine meaning of a *confidence interval* (CI).

Consider a random sample $\{X_1, \ldots, X_n\}$ from a normal distribution with unknown mean μ and known standard deviation σ. We know from our prior studies that a 95% two-sided CI for the population mean μ is

$$\bar{X} \pm 1.96 \times \frac{\sigma}{\sqrt{n}} := \left[\bar{X} - 1.96 \times \frac{\sigma}{\sqrt{n}}, \bar{X} + 1.96 \times \frac{\sigma}{\sqrt{n}}\right],$$

where $\bar{X} = \sum_{i=1}^{n} X_i/n$ is the sample mean and 1.96 is the 97.5% percentile of the standard normal distribution. After observing the realized values of $X_1, \ldots, X_n$, denoted by $x_1, \ldots, x_n$, we can

calculate the realized value of the sample mean $\bar{x} = \sum_{i=1}^{n} x_i/n$ and the CI reduces to a pair of *numbers* $[l, u]$, where

$$l := \bar{x} - 1.96 \times \frac{\sigma}{\sqrt{n}} \qquad \text{and} \qquad u := \bar{x} + 1.96 \times \frac{\sigma}{\sqrt{n}},$$

both of which are readily computed from the observed sample. How do we make sense of the interval $[l, u]$? Can we say that with 95% probability, this interval will enclose the true value of μ, i.e.,

$$\mathbb{P}(l \leq \mu \leq u) = 0.95?$$

Unfortunately, this is a prevalent but misguided interpretation of a CI. In fact, $\mathbb{P}(l \leq \mu \leq u)$ either equals 1 or 0, because there is nothing random about the event $\{l \leq \mu \leq u\}$. Here is the correct interpretation:

> The CI $\bar{X} \pm 1.96(\sigma/\sqrt{n})$, with $\bar{X}$ regarded as a random variable, *does* contain the true value of μ with 95% probability, i.e.,
>
> $$\mathbb{P}\left(\underbrace{\bar{X} - 1.96 \times \frac{\sigma}{\sqrt{n}}}_{\text{random variable}} \leq \underbrace{\mu}_{\text{unknown constant}} \leq \underbrace{\bar{X} + 1.96 \times \frac{\sigma}{\sqrt{n}}}_{\text{random variable}} \right) = 0.95.$$
>
> In other words, if we repeat the CI construction process for a large number of independent random samples, then we *expect that 95% of those realized CIs will include the true value of* μ. However, a particular realized CI from a particular random sample does not bear such a probabilistic interpretation.

To explore this interpretation in R, we:

1. Build a large number of random samples of the same size from a normal distribution with mean $\mu = 5$ (pre-specified) and standard deviation $\sigma = 2$. The fact that we know the true value of μ allows us to check whether the realized CIs contain the target of estimation, an advantage we enjoy in simulation studies.

2. For each random sample, determine the realization of the 95% CI and check if it encloses the value of $\mu = 5$.

3. Calculate the proportion of the realized CIs that enclose the value of $\mu = 5$. We expect this proportion to be close to 95% (but not exactly 95% due to sampling variability).

Run CHUNK 3 to implement our simulation study.

```
# CHUNK 3
# Initialization step
n <- 100
n_sim <- 1000    # number of replications
mu <- 5          # true mean value
sigma <- 2
count <- rep(NA, n_sim)
```

```
# Repetition step
set.seed(0)     # to make results reproducible
for (i in 1:n_sim) {
  # Draw a random sample of size n from a normal distribution
  # with mean mu and standard deviation sigma
  x <- rnorm(n, mean = mu, sd = sigma)
  count[i] <- (abs(mean(x) - mu) <= qnorm(0.975) * sigma / sqrt(n))
}

# Final result
mean(count)

## [1] 0.952
```

The above code may sound intimidating on first encounter. Let's break it down into several pieces for better comprehension.

- *Initialization step:* In the first part of the code, we set the values of:
  - ▷ `n`: The number of i.i.d. observations in each normal random sample.
  - ▷ `n_sim` (meaning "number of simulations"): The number of random samples to simulate.
  - ▷ `mu` and `sigma`: The true mean μ and standard deviation σ of the underlying normal population.
  - ▷ `count`: This is an empty vector of length `n_sim` whose elements are initialized to be all `NA`, but will be filled in after running the `for` loop. Each element of this vector will be set to `TRUE` if the corresponding realized 95% CI bounds the value of `mu` and to `FALSE` otherwise.
- *Repetition step:* In the second part, we repeatedly simulate in a `for` loop a random sample of size `n` from a normal distribution with mean `mu` and standard deviation `sigma` (specified in the first part) by means of the `rnorm(n, mean, sd)` function. We then check whether each CI encloses the value of μ using the equivalence

$$\bar{x} - 1.96 \times \frac{\sigma}{\sqrt{n}} \le \mu \le \bar{x} + 1.96 \times \frac{\sigma}{\sqrt{n}} \quad \Leftrightarrow \quad |\bar{x} - \mu| \le 1.96 \times \frac{\sigma}{\sqrt{n}}, \tag{1.4.1}$$

  the latter of which is captured by the command

```
abs(mean(x) - mu) <= qnorm(0.975) * sigma / sqrt(n),
```

  where `abs()` is the absolute value function and `qnorm(p)` returns the $100p\%$ quantile of the standard normal distribution. This command produces a logical value that equals `TRUE` if (1.4.1) holds and `FALSE` otherwise, and is saved into the ith element of the `count` vector. After the `for` loop is executed, every element of `count` should be filled in by either `TRUE` or `FALSE`.

  You may have noticed the command `set.seed(0)` that precedes the `for` loop. This is to explicitly specify the random seed, which can be any integer, that will be used for the simulation in the `rnorm()` function inside the `for` loop. This way, you will be able to reproduce

the same (sequence of) results each time you run your code, which is especially important from an exam point of view since you will want PA exam graders to get the same results as yours.

Now run CHUNK 4 to see some further examples of setting random seeds, this time using the `runif()` function that simulates uniform random variables. (By default, this function simulates uniform numbers on the interval $[0, 1]$, but the range can be changed by the `min` and `max` arguments.)

```
# CHUNK 4
runif(3)

## [1] 0.9742796 0.7856568 0.5254272

runif(3)  # different results from previous run

## [1] 0.6525329 0.7551247 0.7487606

set.seed(1111)
runif(3)

## [1] 0.4655026 0.4129249 0.9070032

runif(3)

## [1] 0.1371054 0.7388168 0.9763270

set.seed(1111)  # same results as previous two runs
runif(3)

## [1] 0.4655026 0.4129249 0.9070032

runif(3)

## [1] 0.1371054 0.7388168 0.9763270
```

Writing reproducible code is especially important in Exam PA as you definitely want graders to be able to match your output exactly. Without the specification of a random seed, you will get different results every time you do simulation.

- *Final result:* In the last part, we calculate the proportion of realized CIs that enclose the value of μ with the use of the `mean()` function. This works because when the `mean()` function is applied, each element of the `count` vector is converted into `1` if it equals `TRUE` and `0` if it equals `FALSE`, so `mean(count)` returns the proportion of realized CIs that contain the true value of μ. As we expect, this proportion, equal to 0.952 in this case (out of the 1,000 CIs, 952 of them include $\mu = 5$), is very close to 0.95.

For an adaptation of the above simulation study to simple linear regression, try Problem 1.5.6.

Example 1.4.1. (What happens if we put the random seed inside the `for` loop?) When trying to replicate the code in CHUNK 3 above, an actuarial student carelessly puts the random seed `set.seed(0)` inside the `for` loop. That is, he uses the following code (saved in CHUNK 5).

```
# Initialization step
n <- 100
n_sim <- 1000
mu <- 5
sigma <- 2
count <- rep(NA, n_sim)

# Repetition step
for (i in 1:n_sim) {
  set.seed(0)  # put inside for loop
  x <- rnorm(n, mean = mu, sd = sigma)
  count[i] <- (abs(mean(x) - mu) <= qnorm(0.975) * sigma / sqrt(n))
}

# Final result
mean(count)
```

Can you guess the value of `mean(count)` in this case? Is it still close to 0.95?

Solution. To your surprise, the value of `mean(count)` will be either 0 or 1. The reason is that in each iteration of the `for` loop, the random seed is reset due to the command `set.seed(0)`. It follows that the same random sample `x` is simulated in each of the `n` iterations.

Case 1. If the realized CI based on the common random sample encloses the value of μ, then every element of `COUNT` will be `TRUE` and `mean(count)` will equal 1.

Case 2. If the realized CI based on the common random sample does not enclose the value of μ, then every element of `COUNT` will be `FALSE` and `mean(count)` will equal 0.

In neither case is `mean(count)` close to 0.95. As you can see, a small coding error can totally destroy the purpose of simulation!

□

Remark. If you run the student's code, you will get 1 as the value of `mean(count)`. If you change the random seed to `set.seed(85)`, you should get 0 as the final value.

1.5 End-of-Chapter Practice Problems

NOTE

Tasks in Exam PA may not be phrased in exactly the same format as the following problems. However, the programming skills and concepts covered in these problems reinforce what we have learned in the main text of this chapter and may help you complete a task more effectively and efficiently in the exam.

Problem 1.5.1. (Splitting a dataset into a training set and a test set) You are given:

- `dat` is a given dataset (data frame) which you are trying to split into two sets, one set (the training set) for fitting the predictive model you are interested in and one set (the test set) for evaluating the predictive performance of the fitted model.
- `partition` is an integer vector whose elements are selected from `1:nrow(dat)` without replacement (i.e., all of the elements are distinct). You would like to use `partition` as the row index vector of `dat` to form the training set and put the rest of the observations in the test set.

Taking `dat` and `partition` as given inputs, write R code to construct the training set and test set, named as `train` and `test`, respectively.

(Note: All of the released PA sample and exam projects involve the split illustrated in this problem.)

Problem 1.5.2. (Based on December 2018 Exam PA: Writing a function to calculate loglikelihood) For $i = 1, \ldots, n$, let Y_i be a Poisson random variable with mean μ_i and fitted mean $\hat{\mu}_i$ based on a given predictive model (e.g., Poisson regression model, to be studied in Chapter 4) and the method of maximum likelihood estimation.

(a) Show that the maximized loglikelihood function is

$$l(\hat{\mu}_1, \ldots, \hat{\mu}_n) = \sum_{i=1}^{n} y_i \ln \hat{\mu}_i - \sum_{i=1}^{n} \hat{\mu}_i + \text{constants not involving } \hat{\mu}_i\text{'s},$$

provided that the $\ln \hat{\mu}_i$'s are well-defined.

(b) Write an R function called `LL` to compute the maximized loglikelihood function that takes the following arguments:

- `observed` is a vector of observed values of the Y_i's
- `predicted` is a vector of fitted values of the Y_i's

It is not given *a priori* that the fitted values are non-negative. If $\hat{\mu}_i$ is a non-positive value and $\ln \hat{\mu}_i$ is encountered, your function should take care of this by changing $\hat{\mu}_i$ to a very small number, say 0.000001 (note that the log of zero is not defined).

(Hint: You will find the `ifelse()` construct useful.)

(c) Use the function in part (b) to calculate the maximized loglikelihood function for:

```
observed <- c(2, 3, 6, 7, 8, 9, 10, 12, 15)
predicted <- c(2.516332, 2.516332, 7.451633, 7.451633, 7.451633, 7.451633,
               12.386934, 12.386934, 12.386934)
```

Problem 1.5.3. (Based on the new performance measure, how to create a new categorical variable that classifies each actuary as "Excellent", "Good", "Unsatisfactory"?) Consider the `actuary.n` dataset in Section 1.3. If needed, run the following code to get it set up again:

```
x <- 1:8
name <- c("Embryo Luo", "", "Peter Smith", NA,
          "Angela Peterson", "Emily Johnston", "Barbara Scott", "Benjamin Eng")
gender <- c("M", "F", "M", "0", "F", "F", "F", "M")
age <- c(-1, 25, 22, 50, 30, 42, 29, 36)
exams <- c(10, 3, 0, 4, 6, 7, 5, 9)
Q1 <- c(10, NA, 4, 7, 8, 9, 8, 7)
Q2 <- c(9, 9, 5, 7, 8, 10, 9, 8)
Q3 <- c(9, 7, 5, 8, 10, 10, 7, 8)
actuary <- data.frame(x, name, gender, age, exams, Q1, Q2, Q3,
                      stringsAsFactors = FALSE)
actuary.n <- na.omit(actuary)
actuary.n$S <- (actuary.n$Q1 + actuary.n$Q2 + actuary.n$Q3) / 3
```

Suppose that as the boss of the six actuaries, you would like to classify them according to the following scheme:

Criterion	Classification
`S <= 7`	Unsatisfactory
`7 < S < 9`	Good
`S >= 9`	Excellent

Write R commands to perform these classifications and save the classifications as a new variable called `classify`.

Problem 1.5.4. (Practice with the `paste()` or `paste0()` functions) Write R code in terms of the `paste()` or `paste0()` functions to produce the following vector of character strings.

```
##  [1] "1st"  "2nd"  "3rd"  "4th"  "5th"  "6th"  "7th"  "8th"  "9th"  "10th"
```

You should avoid typing the ten character strings one by one.

Problem 1.5.5. (Constructing the Fibonacci sequence) The sequence of Fibonacci numbers $F_1, F_2, \ldots$ is defined by $F_1 = F_2 = 1$ and

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 3.$$

(a) Write an R function called `Fib` to calculate F_n for any positive integer n.

(Note: There are different ways to define this function. Some definitions are more efficient, and some less so.)

(b) Use the function in part (a) to calculate F_2, F_3, F_5, F_{10}, and F_{50}.

Problem 1.5.6. [HARDER!] (Simulation study to check the coverage probability of a prediction interval) This problem requires some knowledge of simple linear regression models that you learned in Exam SRM. You may want to go back to this problem after studying Chapter 3 of this manual.

An actuarial student was asked to use R to perform a simulation study for the coverage probability of a *prediction interval* for a *simple linear regression model*

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \qquad \varepsilon_i \overset{\text{i.i.d.}}{\sim} \text{N}(0, \sigma^2), \quad i = 1, 2, \ldots, n.$$

He has learned that a $100(1-\alpha)\%$ prediction interval based on the data $\{(x_i, y_i)\}_{i=1}^n$ for a new response y_0 observed at $x = x_0$ is a pair of random variables $[L, U]$, dependent on $\{(x_i, y_i)\}_{i=1}^n$, such that

$$\mathbb{P}(L \le y_0 \le U) = 1 - \alpha,$$

where the probability is taken with respect to the (random) data and y_0. To verify this statement, the student has written the following code:

```
n <- 100
n_sim <- 10000
beta0 <- 2  # true intercept
beta1 <- 4  # true slope
sigma <- 1

count <- rep(NA, n_sim)

set.seed(0)
x <- runif(n)
# predictor value of interest
x0 <- 0.8
# target of prediction
y0 <- rnorm(1, mean = beta0 + beta1 * x0, sd = sigma)

for (i in 1:n_sim) {
  y <- rnorm(n, mean = beta0 + beta1 * x, sd = sigma)
  m <- lm(y ~ x)
  # lower bound of 95% prediction interval
  l <- predict(m, newdata = data.frame(x = x0), interval = "prediction")[, 2]
  # upper bound of 95% prediction interval
  u <- predict(m, newdata = data.frame(x = x0), interval = "prediction")[, 3]
  count[i] <- (l <= y0 <= u)
}

mean(count)
```

It turns out that the code fails to run or produce the desired result due to some programming and/or conceptual flaws.

Correct the student's code. After the correction, what should the value of `mean(count)` be close to?

Solutions

Solution to Problem 1.5.1. This is a perfect illustration of how to use vectors of positive integers and negative integers to perform subsetting (recall page 17).

```
train <- dat[partition, ]
test <- dat[-partition, ]
```

Remark. Such a training/test split is used in all of the PA exam and sample projects. In Chapter 3, we will use the `createDataPartition()` function to set up the `partition` vector by means of stratified sampling.

□

Solution to Problem 1.5.2. (a) The likelihood function is

$$L(\mu_1, \ldots, \mu_n) \propto \prod_{i=1}^{n} \mathrm{e}^{-\mu_i} \mu_i^{y_i} = \mathrm{e}^{-\sum_{i=1}^{n} \mu_i} \prod_{i=1}^{n} \mu_i^{y_i},$$

so the loglikelihood function at the fitted means $\hat{\mu}_1, \ldots, \hat{\mu}_n$ is

$$l(\hat{\mu}_1, \ldots, \hat{\mu}_n) = \ln L(\hat{\mu}_1, \ldots, \hat{\mu}_n) = \sum_{i=1}^{n} y_i \ln \hat{\mu}_i - \sum_{i=1}^{n} \hat{\mu}_i + \text{constants not involving } \hat{\mu}_i\text{'s}.$$

(b) The loglikelihood function involves $\ln \hat{\mu}_i$, which is not defined when $\hat{\mu}_i \leq 0$. To sidestep the calculation of $\ln \hat{\mu}_i$ for non-positive $\hat{\mu}_i$, we need to design the function in such a way that those $\hat{\mu}_i$'s will be converted to a small number such as 0.000001 and $\ln 0.000001$ is computed instead. This is accomplished by the `ifelse()` construct. Here is how the function is defined:

```
LL <- function(observed, predicted){
  predicted_pos <- ifelse(predicted <= 0, 0.000001, predicted)
  return(sum(observed*log(predicted_pos) - predicted))
}
```

(c) For the given `observed` and `predicted` vectors, the maximized loglikelihood function equals 85.98277.

```
observed <- c(2, 3, 6, 7, 8, 9, 10, 12, 15)
predicted <- c(2.516332, 2.516332, 7.451633, 7.451633, 7.451633, 7.451633,
               12.386934, 12.386934, 12.386934)
LL(observed, predicted)

## [1] 85.98277
```

□

Solution to Problem 1.5.3. Similar to how we accomplished Task 4 (b) in Section 1.3, we use `ifelse()` to make the classifications:

```
actuary.n$classify <- ifelse(actuary.n$S >= 9, "Excellent",
                             ifelse(actuary.n$S > 7, "Good", "Unsatisfactory"))
actuary.n
```

```
##   x           name gender age exams Q1 Q2 Q3        S       classify
## 1 1     Embryo Luo      M  -1    10 10  9  9 9.333333      Excellent
## 3 3     Peter Smith     M  22     0  4  5  5 4.666667 Unsatisfactory
## 5 5 Angela Peterson     F  30     6  8  8 10 8.666667           Good
## 6 6  Emily Johnston     F  42     7  9 10 10 9.666667      Excellent
## 7 7   Barbara Scott     F  29     5  8  9  7 8.000000           Good
## 8 8    Benjamin Eng     M  36     9  7  8  8 7.666667           Good
```

□

Solution to Problem 1.5.4. Let's break down each character string into two parts:

- The first part is a number from 1 to 10, which can be generated by the vector `1:10`.
- The second part is a text that equals `"st"`, `"nd"`, `"rd"` for the first three strings and `"th"` for the rest. This can be produced by the vector `c("st", "nd", "rd", rep("th", 7))`.

Now let's paste the two vectors to produce the desired vector of character strings:

```
paste0(1:10, c("st", "nd", "rd", rep("th", 7)))
```

```
##  [1] "1st"  "2nd"  "3rd"  "4th"  "5th"  "6th"  "7th"  "8th"  "9th"  "10th"
```

□

Solution to Problem 1.5.5. (a) Two solutions are provided:

- *Based on recursive definitions:* Here the function is defined in terms of itself.

```
Fib <- function(n) {
  if (n == 1 | n == 2) {
    x <- 1
  } else {
    x <- Fib(n - 1) + Fib(n - 2)
  }
  return(x)
}
```

- *Based on a* `for` *loop:* Here we use a `for` loop to calculate F_n recursively in terms of F_{n-1} and F_{n-2}.

```
Fib <- function(n) {
  if (n == 1 | n == 2) {
    return(1)
  }
  x <- rep(NA, n)
  x[1] <- x[2] <- 1
```

```
  for (i in 3:n) {
    x[i] <- x[i - 1] + x[i - 2]
  }
  return(x[n])
}
```

(b) Let's apply the `Fib()` function to calculate F_2, F_3, F_5, F_{10}, and F_{50}.

```
Fib(2)

## [1] 1

Fib(3)

## [1] 2

Fib(5)

## [1] 5

Fib(10)

## [1] 55

Fib(50)

## [1] 12586269025
```

Remark. The second definition of the `Fib()` function has at least two advantages over the first one:

- If you use the first definition to calculate F_{50}, it will take a *long* time to complete the calculation. In contrast, the calculation of F_{50} by the second definition is almost immediate.
- The second definition can be easily modified to return the whole sequence of Fibonacci numbers up to the nth one. To do this, simply change the last line of the code from `return(x[n])` to `return(x)`.

```
Fib <- function(n) {
  if (n == 1 || n == 2) {
    return(1)  # the function will stop at this point if n is 1 or 2
  }
  x <- rep(NA, n)
  x[1] <- x[2] <- 1
  for (i in 3:n) {
```

```
    x[i] <- x[i - 1] + x[i - 2]
  }
  return(x)
}

Fib(2)

## [1] 1

Fib(3)

## [1] 1 1 2

Fib(5)

## [1] 1 1 2 3 5

Fib(10)

##  [1]  1  1  2  3  5  8 13 21 34 55
```

□

Solution to Problem 1.5.6. The student's code fails to run due to the flawed command `l <= y0 <= u`, which has to be separated into two logical tests, `(l <= y0) & (y0 <= u)`. With this correction, we are able to run the code, but the value of `mean(count)`, equal to 1 in this case, is not what we expect. In the probability statement

$$\mathbb{P}(L \leq y_0 \leq U) = 1 - \alpha,$$

the probability is taken with respect to both the data, where the y_i's are random variables, and y_0, which is also a random variable. In each round of simulation, we should draw random values for the y_i's as well as y_0. The student's code, however, treats y_0 as fixed across the 10,000 rounds of simulation.

Here is the corrected code, where the line defining `y0` is moved inside the `for` loop:

```
n <- 100
n_sim <- 10000
beta0 <- 2  # true intercept
beta1 <- 4  # true slope
sigma <- 1

count <- rep(NA, n_sim)

set.seed(0)
x <- runif(n)
# predictor value of interest
x0 <- 0.8

for (i in 1:n_sim) {
  y <- rnorm(n, mean = beta0 + beta1 * x, sd = sigma)
```

```
  m <- lm(y ~ x)
  # target of prediction
  y0 <- rnorm(1, mean = beta0 + beta1 * x0, sd = sigma)
  # lower bound of 95% prediction interval
  l <- predict(m, newdata = data.frame(x = x0), interval = "prediction")[, 2]
  # upper bound of 95% prediction interval
  u <- predict(m, newdata = data.frame(x = x0), interval = "prediction")[, 3]
  count[i] <- (l <= y0) & (y0 <= u)
}

mean(count)

## [1] 0.9512
```

As we expect, `mean(count)` is now close to 0.95, showing that the 95% prediction interval covers the random individual response about 95% of the time.

Remark. Try to vary the random seed and you should still get `mean(count)` very close to 0.95. □