The if Statement and the apply Function in R

Sunthud Pornprasertmanit

June 16, 2015

In the previous lecture, we talked about how to create figures and functions. R can handle each component of the figures manually, such as axes, lines, or legend. The ability to modify the graphs is more flexible than most statistical packages. Furthermore, users can write functions in R. The function feature will remove many repeated codes and users may save the functions for their future uses. In this class, the if statement is introduced to run conditional statements. Then, the `apply` function is introduced to run functions on each row or column on a matrix.

Let's introduce the if statement before running any codes. The if statement will allow programs to run the desired codes if the specified condition is satisfied:

```
> x <- 5
> if (x > 0) {
+   print("positive")
+ }

[1] "positive"
```

In this code, `x` is assigned as 5. The condition is inside the parenthesis. Because `x` is higher than 0, the condition is satisfied. Then, the codes inside the curly bracket after the if command are evaluated. That is, `"positive"` is printed. Let's see another example of the if statement:

```
> if (x < 0) print("negative")
```

You will see that `"negative"` is not printed because the condition is not satisfied. The curly bracket is not needed if the code inside the bracket has only one line. Next, the if...else statement is similar to the if statement. However, if the condition is not satisfied, another set of codes will be evaluated:

```
> if (x > 0) {
+   print("positive")
+ } else {
+   print("nonpositive")
+ }

[1] "positive"
```

Because x is higher than 0, the "positive" is printed. You may change x to a number less than 0 and evaluate the previous if...else statement again. Anyway, the condition is not limited to only two alternatives. If there are more than two alternatives, additional conditions can be added by the else if statement:

```
> if (x > 0) {
+   print("positive")
+ } else if (x == 0) {
+   print("zero")
+ } else {
+   print("negative")
+ }

[1] "positive"
```

In this code, x > 0 is evaluated first. If the condition is satisfied, "positive" is printed. Otherwise, the second condition, x == 0, is evaluated. If the condition is satisfied, "zero" is printed. Otherwise, the codes under the else statement are evaluated. That is, "negative" is printed. Sometimes, the else statement is not needed:

```
> if (x > 0) {
+   print("positive")
+ } else if (x < 0) {
+   print("negative")
+ }

[1] "positive"
```

In this example, if x is equal to 0, nothing is printed. Let's run a little more complicated if statement:

```
> score <- 79
> grade <- ""
> if (score >= 90) {
+   grade <- "A"
+ } else if (score >= 80) {
+   grade <- "B"
+ } else if (score >= 70) {
+   grade <- "C"
+ } else if (score >= 60) {
+   grade <- "D"
+ } else{
+   grade <- "F"
+ }
> grade
```

```
[1] "C"
```

In this example, `score >= 90` is evaluated first and is not satisfied. Next, `score >= 80` is evaluated and is not satisfied. Then, `score >= 70` is evaluated. Because this condition is satisfied, `grade` is assigned as C. The remaining lines are skipped. That is, `score >= 60` is not evaluated at all because `score >= 70` is already satisfied.

Sometimes, we do not need to use the if statement. One example is to recode one vector to another vector:

```
> scores <- 40:100
> grades <- rep("F", length(scores))
> grades[scores >= 60] <- "D"
> grades[scores >= 70] <- "C"
> grades[scores >= 80] <- "B"
> grades[scores >= 90] <- "A"
```

Let's generate data from other distributions besides normal distribution (`rnorm`) and binomial distribution (`rbinom`). Then, we will use different shapes of data to see the impact of nonnormal distributions on statistical results. Let's create four variables drawn from (a) standard normal distribution, `rnorm`, (b) t distribution with the degree of freedom of 3, `rt`, (c) chi-square distribution with the degree of freedom of 4, `rchisq`, and (d) F distribution with the degrees of freedom of 1 and 30:

```
> x1 <- rnorm(1000, 0, 1)
> x2 <- rt(1000, 3)
> x3 <- rchisq(1000, 4)
> x4 <- 1 - rf(1000, 1, 30)
> gendat <- data.frame(x1, x2, x3, x4)
```

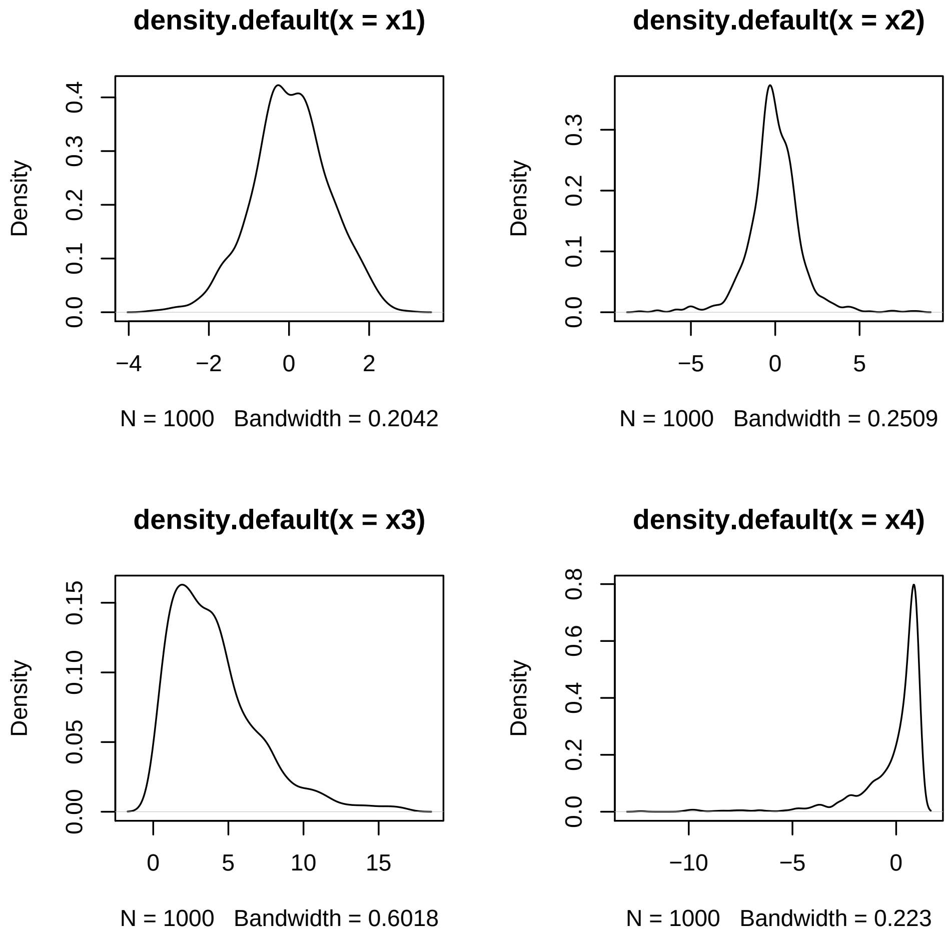
Note that the numbers drawn from F distribution are subtracted from 1 because it changes from positively-skewed distribution to negatively-skewed distribution (i.e., creating the mirror image). Let's check the skewness and kurtosis of these variables. We will use the `describe` function from the `psych` package.

```
> library(psych)
> describe(gendat)
```

```
   vars    n  mean   sd median trimmed  mad    min   max range  skew kurtosis
x1    1 1000  0.05 0.96   0.06    0.06 0.90  -3.42  2.94  6.36 -0.12     0.14
x2    2 1000 -0.07 1.53  -0.10   -0.06 1.10  -8.04  8.47 16.51  0.03     4.97
x3    3 1000  4.07 2.96   3.51    3.69 2.58   0.09 16.70 16.61  1.33     2.13
x4    4 1000 -0.11 1.69   0.54    0.26 0.63 -12.31  1.00 13.31 -3.08    12.38
     se
x1 0.03
x2 0.05
x3 0.09
x4 0.05
```

These variables can be plotted using kernel density estimation:

```
> par(mfrow = c(2, 2))
> plot(density(x1))
> plot(density(x2))
> plot(density(x3))
> plot(density(x4))
```



Now, we know how to create nonnormal data. Let's run a simulation investigating the impact of nonnormal data on simple regression. Both independent variable and error can be drawn from nonnormal distribution. Let's use numbers 1, 2, 3, and 4 to represent normal, t, chi-square, and reversed F distributions, respectively. Then, use the if statement to select an appropriate distribution:

```
> result <- NULL
> distype <- 2
> n <- 1000
> if(distype == 1) {
+   result <- rnorm(n, 0, 1)
+ } else if (distype == 2) {
+   result <- rt(n, 3)
+ } else if (distype == 3) {
+   result <- rchisq(n, 4)
+ } else if (distype == 4) {
+   result <- 1 - rf(n, 1, 30)
+ }
```

`distype` and `n` can be changed as we wish. However, we need to control the mean and variance of the created data. Otherwise, the regression coefficient in data generating process will not be equal to standardized regression coefficient.

```
> m <- 0
> stdev <- 0.6
> result <- (result - mean(result)) / sd(result)
> result <- stdev * result + m
> mean(result)

[1] 6.312144e-18

> sd(result)

[1] 0.6
```

`m` and `stdev` are desired mean and standard deviation. The `result` vector will be transformed to z score first. Then, the z score is transformed to have desired mean and standard deviation. However, if we repeatedly run this code, the `result` will always have the mean and standard deviation exactly equal to the specified values. This is not desirable because the mean and standard deviation should be fluctuated by sampling error. We can make the mean and standard deviation influenced by the sampling error by drawing another set of numbers from normal distribution and use the obtained mean and standard deviation as the targets.

```
> result <- (result - mean(result)) / sd(result)
> temp <- rnorm(n, m, stdev)
> result <- sd(temp) * result + mean(temp)
> mean(result)

[1] -0.03589435

> sd(result)

[1] 0.5848006
```

The obtained `result` vector will have the mean and standard deviation fluctuated by the sampling error. Let's create a function to generate data from different distributions:

```
> gendata <- function(n, m, stdev, distype = 1) {
+   result <- NULL
+   if(distype == 1) {
+     result <- rnorm(n, 0, 1)
+   } else if (distype == 2) {
+     result <- rt(n, 3)
+   } else if (distype == 3) {
+     result <- rchisq(n, 4)
+   } else if (distype == 4) {
+     result <- 1 - rf(n, 1, 30)
+   } else {
+     stop("The distype argument is not valid.")
+   }
+   result <- (result - mean(result)) / sd(result)
+
+   temp <- rnorm(n, m, stdev)
+   result <- sd(temp) * result + mean(temp)
+   return(result)
+ }
> y1 <- gendata(1000, m = -2, stdev = 8)
> y2 <- gendata(1000, m = 10, stdev = 50, distype = 2)
> describe(data.frame(y1, y2))

   vars    n  mean    sd median trimmed   mad     min    max   range  skew
y1    1 1000 -1.94  7.80  -1.88   -2.01  7.70  -23.82  23.38   47.20  0.11
y2    2 1000  9.31 49.57  10.78   10.73 24.74 -977.48 206.45 1183.93 -9.02
   kurtosis   se
y1    -0.06 0.25
y2   168.54 1.57
```

The codes for data generation above are wrapped in a function. Note that the `distype` argument in the function definition has '`= 1`'. It means that the default value of `distype` is 1. That is, if this argument is not specified, `distype` would be 1. Furthermore, the else statement is specified by the `stop` function. It means that, if `distype` is not satisfied the condition above (i.e., equal to 1, 2, 3, or 4), then the function will stop and the error message is provided.

Let's use this function to run a simulation under which independent variable and error are drawn from t distribution. The code form the previous lecture can be modified:

```
> set.seed(123321)
> nrep <- 1000
> b <- 0.5
> n <- 200
> xdist <- 2
> edist <- 2
> result <- matrix(NA, nrep, 3)
> for(i in 1:nrep) {
+   x <- gendata(n, 0, 1, xdist)
+   e <- gendata(n, 0, sqrt(1 - b^2), edist)
+   y <- b*x + e
+   dat <- data.frame(x = x, y = y)
+   out <- lm(y ~ x, data = dat)
+   result[i,] <- summary(out)[["coefficients"]][2, c(1, 2, 4)]
+ }
> sig <- result[,3] < 0.05
> result <- cbind(result, sig)
> colMeans(result)
```

```
                                                 sig
4.966065e-01 6.151472e-02 7.143060e-08 1.000000e+00
```

xdist and edist represent the type of distribution for independent variable and error, respectively. Notice that x and e are generated by the gendata function.

Let's run this Monte Carlo simulation across different design conditions. Let the distribution types of independent variable and error be any of the four distribution types defined above. We can use nested for loop to change design conditions:

```
> set.seed(123321)
> nrep <- 1000
> b <- 0.5
> n <- 200
> ovresult <- NULL
> for(xdist in 1:4){
+   for(edist in 1:4) {
+     result <- matrix(NA, nrep, 3)
+     for(i in 1:nrep) {
+       x <- gendata(n, 0, 1, xdist)
+       e <- gendata(n, 0, sqrt(1 - b^2), edist)
+       y <- b*x + e
+       dat <- data.frame(x = x, y = y)
+       out <- lm(y ~ x, data = dat)
+       result[i,] <- summary(out)[["coefficients"]][2, c(1, 2, 4)]
+     }
+     result <- cbind(xdist, edist, result)
+     ovresult <- rbind(ovresult, result)
+   }
+ }
> sig <- ovresult[,5] < 0.05
> ovresult <- cbind(ovresult, sig)
> colnames(ovresult) <- c("xdist", "edist", "est", "se", "p", "sig")
> aggresult <- aggregate(cbind(est, se, sig) ~ xdist + edist,
+     data = ovresult, FUN = mean)
> aggsdresult <- aggregate(est ~ xdist + edist, data = ovresult, FUN = sd)
> biasest <- aggresult[,"est"] - b
> relbiasest <- biasest / b
> biasse <- aggresult[,"se"] - aggsdresult[,"est"]
> relbiasse <- biasse / aggsdresult[,"est"]
> power <- aggresult[,"sig"]
> tableresult <- data.frame(aggresult[,c(1, 2)], biasest, relbiasest, biasse,
+     relbiasse, power)
> tableresult
```

```
   xdist edist       biasest    relbiasest        biasse     relbiasse power
1      1     1 -0.0003559952 -0.0007119905  4.762941e-04  7.800736e-03     1
2      2     1  0.0030582043  0.0061164086 -8.766969e-04 -1.407029e-02     1
3      3     1  0.0035675959  0.0071351917 -6.682936e-04 -1.074133e-02     1
4      4     1 -0.0002813122 -0.0005626244 -2.875537e-03 -4.460364e-02     1
5      1     2  0.0005828205  0.0011656410 -8.389878e-04 -1.350273e-02     1
6      2     2 -0.0011269125 -0.0022538251  9.921356e-05  1.606995e-03     1
7      3     2 -0.0011514879 -0.0023029758 -9.411741e-04 -1.513492e-02     1
8      4     2 -0.0011437675 -0.0022875350 -1.447668e-03 -2.294149e-02     1
9      1     3 -0.0008009068 -0.0016018136 -5.399251e-04 -8.701143e-03     1
10     2     3 -0.0004122805 -0.0008245610 -1.428817e-03 -2.267681e-02     1
11     3     3  0.0005811861  0.0011623722 -1.411011e-03 -2.236520e-02     1
12     4     3 -0.0044351025 -0.0088702051 -3.100580e-03 -4.804716e-02     1
13     1     4 -0.0002294290 -0.0004588580 -3.003703e-04 -4.877703e-03     1
14     2     4  0.0004224618  0.0008449235 -1.504090e-06 -2.444132e-05     1
15     3     4  0.0008796820  0.0017593641  3.182911e-03  5.449578e-02     1
16     4     4 -0.0024605262 -0.0049210525 -1.124076e-04 -1.819932e-03     1
```

The codes are similar to the previous lecture so I will not repeat them here.

In R, the for loop is not an efficient way to run repeated task. The most efficient way is to use R communicate with programming language, such as C or Fortran, to run repeated tasks. For example, using the for loop in R for estimating state space model can take 10-100 times longer than using the help from C. However, if we teach this technique, this class is not about "basic" simulation anymore. Rather, the easier way is to use the family of apply functions, including `apply`, `sapply`, `lapply`, or `mapply`. In this lecture, we will learn the `apply` function. We will learn other functions in the next lecture.

The `apply` function is designed for a matrix (or a data frame). It will separate a matrix (or a data frame) to different vectors by rows or by columns. Then, each vector will be evaluated simultaneously by a specified function and return the results of the evaluation from each vector. Let's say we would like to calculate the means for each column from the `attitude` data:

```
> apply(attitude, 2, mean)
```

```
    rating complaints privileges   learning     raises   critical    advance 
  64.63333   66.60000   53.13333   56.36667   64.63333   74.76667   42.93333 
```

The first argument is the desired matrix or data frame. The second argument represents the dimension of the matrix to be separated: 1 = separated by rows and 2 = separated by columns. The third dimension represents the function to evaluate for each vector. In this example, the `attitude` data are separated by columns. Then, the `mean` function will be evaluated for each column. The results from each column will be one number and stacked into a vector. We can find the mean of each row by changing the second argument to 1.

```
> apply(attitude, 1, mean)

 [1] 51.57143 59.28571 69.71429 55.57143 68.85714 46.85714 56.00000 61.14286
 [9] 68.28571 57.57143 55.28571 55.28571 53.28571 64.28571 68.85714 63.28571
[17] 73.28571 65.71429 63.42857 60.57143 42.42857 60.85714 57.28571 47.57143
[25] 54.42857 72.57143 70.28571 52.28571 74.00000 63.28571
```

We can calculate the standard deviation of each row or of each column:

```
> apply(attitude, 1, sd)

 [1] 20.263737  8.994707 11.412191 15.830952 13.458932  7.244045 12.948616
 [8] 12.641579 17.471065 13.950593 10.734901 13.300555 15.063358 18.856602
[15] 13.259318 18.803495  8.300889  8.864053 14.683972 10.047506 11.385036
[22] 12.088956 13.756384  8.223080 14.638501  8.638232 11.926361 14.784161
[29] 10.246951 18.776885

> apply(attitude, 2, sd)

    rating complaints privileges   learning     raises   critical    advance 
 12.172562  13.314757  12.235430  11.737013  10.397226   9.894908  10.288706 
```

Let's create a copy of the attitude data set and impose one missing observation:

```
> newdata <- attitude
> newdata[2, 4] <- NA
> apply(newdata, 1, mean)

 [1] 51.57143       NA 69.71429 55.57143 68.85714 46.85714 56.00000 61.14286
 [9] 68.28571 57.57143 55.28571 55.28571 53.28571 64.28571 68.85714 63.28571
[17] 73.28571 65.71429 63.42857 60.57143 42.42857 60.85714 57.28571 47.57143
[25] 54.42857 72.57143 70.28571 52.28571 74.00000 63.28571

> apply(newdata, 2, mean)

    rating complaints privileges   learning     raises   critical    advance 
  64.63333   66.60000   53.13333         NA   64.63333   74.76667   42.93333 
```

The results from the row or column that has the missing observation were `NA`. Usually, we will specify `na.rm = TRUE` in the `mean` function to ignore missing data. To put the `na.rm` argument to the `mean` function, these arguments can be added in the `apply` function directly after the third argument specifying the desired function:

```
> apply(newdata, 1, mean, na.rm = TRUE)

 [1] 51.57143 60.16667 69.71429 55.57143 68.85714 46.85714 56.00000 61.14286
 [9] 68.28571 57.57143 55.28571 55.28571 53.28571 64.28571 68.85714 63.28571
[17] 73.28571 65.71429 63.42857 60.57143 42.42857 60.85714 57.28571 47.57143
[25] 54.42857 72.57143 70.28571 52.28571 74.00000 63.28571

> apply(newdata, 2, mean, na.rm = TRUE)

    rating complaints privileges   learning     raises   critical    advance
  64.63333   66.60000   53.13333   56.44828   64.63333   74.76667   42.93333
```

If the desired function return a vector (e.g., `range`), the results of the `apply` function will be stacked into a matrix where rows represent each element of the result.

```
> apply(attitude, 2, range)

     rating complaints privileges learning raises critical advance
[1,]     40         37         30       34     43       49      25
[2,]     85         90         83       75     88       92      72
```

We can create our own function and use it in the `apply` function. For example, we can create a function to calculate the coefficient of variation and use the `apply` function to find the coefficient of variation of each column:

```
> cv <- function(x) {
+   sd(x)/mean(x)
+ }
> apply(attitude, 2, cv)

    rating complaints privileges   learning     raises   critical    advance
 0.1883326  0.1999213  0.2302779  0.2082261  0.1608648  0.1323438  0.2396438
```

We can also create a function with additional arguments to be added in the `apply` function as well. For example, we can create the `cv2` function that takes the `na.rm` argument.

```
> cv2 <- function(x, na.rm = FALSE) {
+   sd(x, na.rm = na.rm)/mean(x, na.rm = na.rm)
+ }
> apply(newdata, 2, cv2, na.rm = TRUE)

    rating complaints privileges   learning     raises   critical    advance
 0.1883326  0.1999213  0.2302779  0.2114520  0.1608648  0.1323438  0.2396438
```

Because the apply function separates a matrix into vectors, we can write a function that handle individual elements of each vector:

```
> design <- matrix(1:9, 3, 3)
> design

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> FUN <- function(x) {
+   x[1] * x[2] - x[3]
+ }
> apply(design, 1, FUN)

[1] -3  2  9

> apply(design, 2, FUN)

[1] -1 14 47
```

The created function will multiply the first and the second elements and then minus the third element. We use the apply function to this created function. If the apply function is implemented by rows, the first column is multiplied by the second column and then subtracted by the third column: $1 \times 4 - 7 = 11$, $2 \times 5 - 8 = 2$, and $3 \times 6 - 9 = 9$. The same logic is applicable for the apply function by columns.

Let's create a function that runs Monte Carlo simulation and returns whether each replication was significant. This function will take the type of distribution of independent variable and error:

```
> nonnormalsim <- function(cond, b, n, nrep, alpha) {
+   result <- rep(NA, nrep)
+   for(i in 1:nrep) {
+     x <- gendata(n, 0, 1, cond[1])
+     e <- gendata(n, 0, sqrt(1 - b^2), cond[2])
+     y <- b*x + e
+     dat <- data.frame(x = x, y = y)
+     out <- lm(y ~ x, data = dat)
+     result[i] <- summary(out)[["coefficients"]][2, 4] < alpha
+   }
+   result
+ }
> simout <- nonnormalsim(cond = c(2, 2), b = 0.5, n = 200, nrep = 1000,
+     alpha = 0.05)
> mean(simout)
```

```
[1] 1
```

The `cond` argument is a vector with two elements: the distributions for independent variable and error, respectively. We will vary `cond` across design conditions. The `b`, `n`, `nrep`, and `alpha` are regression coefficients, sample size, the number of replications, and the alpha level, respectively. In each analysis, the p value is compared with the alpha level providing the significant result of each analysis. The result of this function is a logical vector with the length of `nrep` showing whether each replication was significant.

We will create a matrix with 16 rows representing all combinations of 4 distributions for independent variable and 4 distributions of error by the `expand.grid` function:

```
> xdists <- 1:4
> edists <- 1:4
> conds <- expand.grid(xdists, edists)
> head(conds)
```

```
  Var1 Var2
1    1    1
2    2    1
3    3    1
4    4    1
5    1    2
6    2    2
```

Then, we use the `apply` function to distribute each row of `conds` to the `nonnormalsim` function:

```
> superresult <- apply(conds, 1, nonnormalsim, b = 0, n = 250,
+     nrep = 1000, alpha = 0.05)
> cbind(conds, apply(superresult, 2, mean))
```

```
   Var1 Var2 apply(superresult, 2, mean)
1     1    1                       0.045
2     2    1                       0.044
3     3    1                       0.047
4     4    1                       0.051
5     1    2                       0.056
6     2    2                       0.049
7     3    2                       0.054
8     4    2                       0.055
9     1    3                       0.045
10    2    3                       0.053
11    3    3                       0.048
12    4    3                       0.042
13    1    4                       0.052
14    2    4                       0.050
15    3    4                       0.037
16    4    4                       0.046
```

We have the right to know whether this trick really saves time. Let's check it out. To check the amount of time spending by codes, the `system.time` function can be used by cropping desired codes:

```
> system.time(
+ superresult <- apply(conds, 1, nonnormalsim, b = 0, n = 250,
+     nrep = 1000, alpha = 0.05)
+ )
```

```
   user  system elapsed 
  24.80    0.00   24.87 
```

Then, we compare with the same code using the for loop:

```
> system.time({
+ superresult2 <- matrix(NA, 1000, nrow(conds))
+ for(i in 1:nrow(conds)) {
+         superresult2[,i] <- nonnormalsim(conds[i,], b = 0, n = 250,
+             nrep = 1000, alpha = 0.05)
+ }
+ })
```

```
   user  system elapsed 
  28.11    0.00   28.16 
```

Note that we have the curly bracket inside the parenthesis of the `system.time` function because multiple lines of codes are timed. We will see that the `apply` function used less time. The saving time may be trivial here. However, if we run a large simulation, saving 10% of your computer time is actually huge!

Notice that this `apply` function provides only one result, which is whether each replication was significant. Sometimes, we need multiple results. We will learn the `sapply`, `lapply`, or `mapply` functions that are easier to deal with multiple results next class.

1 Exercise

1. Create a function that takes the result of the `lm` function and returns R^2. The function must have an argument for users to calculate R^2 or adjusted R^2 of the result. The default is to provide R^2.

2. Use the following code to evaluate the impact of nonnormal distributions on the Type I error of the test of correlation. The design conditions are as follows:

- Sample size: 50, 150, or 250.
- The distribution of the first variable: (a) normal distribution, (b) t-distribution with *df* of 4, (c) F distribution with *df*s of 1 and 15, or (d) F distribution *df*s of 1 and 15 in the reversed direction.

- The distribution of the second variable: (a) normal distribution, (b) t-distribution with df of 4, (c) F distribution with dfs of 1 and 15, or (d) F distribution dfs of 1 and 15 in the reversed direction.

Try to use the *apply* function to run the simulation. Hint: a vector with the length of 3 can be put in the condition of the function used in the `apply` function.

```
> gendata <- function(n, m, stdev, distype = 1) {
+   result <- NULL
+   if(distype == 1) {
+     result <- rnorm(n, 0, 1)
+   } else if (distype == 2) {
+     result <- rt(n, 4)
+   } else if (distype == 3) {
+     result <- rf(n, 1, 15)
+   } else if (distype == 4) {
+     result <- 1 - rf(n, 1, 15)
+   } else {
+     stop("The distype argument is not valid.")
+   }
+   result <- (result - mean(result)) / sd(result)
+
+   temp <- rnorm(n, m, stdev)
+   result <- sd(temp) * result + mean(temp)
+   return(result)
+ }
> n <- 200
> xdist <- 2
> ydist <- 2
> x <- gendata(n, 0, 1, xdist)
> y <- gendata(n, 0, 1, ydist)
> out <- cor.test(x, y)
> out[["p.value"]]

[1] 0.7521914
```