Getting Started to Simulation Study Using R

Sunthud Pornprasertmanit

June 15, 2015

R is a very powerful tool for statistical analysis and programming. Thus, R is a perfect start for beginners to run simulation studies to evaluate statistical techniques. This note simply aims to provide a gentle introduction to Monte Carlo simulation. The codes in this note are not designed for the efficiency in using computer resources. However, I design to make it as simple as possible. You may be wondering that the functions that you have known can be easily combined to run Monte Carlo simulations.

The example in this note is to check the robustness of independent t-test if population variances of two groups are not equal (i.e., heterogeneity of variance). The `t.test` function is used for independent t-test as follows:

```
> t.test(extra ~ group, data = sleep, var.equal = TRUE)

	Two Sample t-test

data:  extra by group
t = -1.8608, df = 18, p-value = 0.07919
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.363874  0.203874
sample estimates:
mean in group 1 mean in group 2 
           0.75            2.33 
```

In this t-test, the effectiveness of two drugs (`group`) are compared. The response varaible is the increase in sleep time in hours (`extra`). Note that you could see the details of the `sleep` data by checking the help page by typing `?sleep`. The result of the t-test can be saved by assigning it to an object. Then, desired components can be extracted from the results.

```
> out <- t.test(extra ~ group, data = sleep, var.equal = TRUE)
> names(out)

[1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"   
[6] "null.value"  "alternative" "method"      "data.name"  
```

```
> out[["p.value"]]

[1] 0.07918671

> out[["p.value"]] < 0.05

[1] FALSE
```

In this code, the result is saved as an object, `out`. The names of all components of the result can be checked by the `names` function. The `p.value` component is used for significance testing. The p value is compared with the alpha level of 0.05 to see whether the group mean difference is significant. Double square bracket is used to extract the desired component from the result.

Before moving to Monte Carlo simulation, I would like to introduce the `aggregate` function. This function can be used to find group statistics:

```
> aggregate(extra ~ group, data = sleep, FUN = mean)

  group extra
1     1  0.75
2     2  2.33

> aggregate(extra ~ group, data = sleep, FUN = sd)

  group    extra
1     1 1.789010
2     2 2.002249
```

The `FUN` argument is a function that users wish to find for each group.

Let's discuss about Monte Carlo simulation. Monte Carlo simulation in statistics is a procedure usually used to evaluate the behavior of statistics by using generated data. Researchers can manipulate the characteristics of the data and see the performance of the manipulated data. Obviously, the first step of Monte Carlo simulation is to generate data. R provides many functions for data generation. As the most basic function, `rnorm` is used to generate univariate normal distribution. For example, the normal data of both groups can be generated:

```
> set.seed(1234)
> y1 <- rnorm(100, 0, 1)
> y2 <- rnorm(100, 0, 1)
> y <- c(y1, y2)
> g <- c(rep(1, 100), rep(2, 100))
> dat <- data.frame(g = g, y = y)
> out <- t.test(y ~ g, data = dat, var.equal = TRUE)
> out[["p.value"]]

[1] 0.567678
```

The `set.seed` function is used to specify random number generator. If the random number generator is specified, the results can be replicated later. The number in the function could be any integer number. The arguments of `rnorm` are (a) sample size, (b) population mean, (c) population standard deviation. `y1` and `y2` are the vectors of scores for Groups 1 and 2, respectively. Then, `y` is the concatenation of `y1` and `y2`. `g` is the group indicators: the first 100 cases for Group 1 and the last 100 cases for Group 2. Next, data are created to be a data frame with two variables: `g` and `y`. As the next step of Monte Carlo simulation, the data set is analyzed by desired statistics, which is t-test. As the final step, the desired statistics is saved, which is p-value here.

In the previous example, only one data set is created. In the real simulation study, multiple data sets should be generated from the same population model. Then, desired statistics are saved from multiple data sets and combined. In this example, the proportion of significant results ($p < .05$) across multiple data sets represents Type I error. The trick to create multiple data sets is to use for loop. Here is an example of simple for loop:

```
> for (i in 1:3) {
+   print(i)
+ }

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

The for loop will repeatedly evaluate codes inside the curly bracket. The index of each round is `i`. In each round, `i` will be represented by the order of the sequence specified in the for loop, which is `1:3`. In this example, the for loop will repeat three times and `i` will take values of 1, 2, and 3, respectively. That is, `print` will be evaluated three times and the values of 1, 2, and 3 are printed.

The data-generating code above can be modified for repeatedly running 1,000 replications:

```
> set.seed(1234)
> nrep <- 1000
> result <- rep(NA, nrep)
> for(i in 1:nrep) {
+   y1 <- rnorm(100, 0, 1)
+   y2 <- rnorm(100, 0, 1)
+   y <- c(y1, y2)
+   g <- c(rep(1, 100), rep(2, 100))
+   dat <- data.frame(g = g, y = y)
+   out <- t.test(y ~ g, data = dat, var.equal = TRUE)
+   result[i] <- out[["p.value"]]
+ }
> mean(result < 0.05)

[1] 0.048
```

nrep represents the number of replications. Then, result is a vector of NA with the length of 1,000. This vector is used to save p-values from all replications. Then, the for loop begins. Inside the loop, data are generated and analyzed by t-test. The p-value of each replication is saved in the result vector at the i-th position. After the for loop, the result vector has the p-values of all 1,000 replications. Then, the last line aims to find the proportion of replications that p-values are less than .05.

In the real simulation study, researchers would like to create data with different characteristics and see how it influences the results of desired statistics. In this simulation, group sizes and the variances of each group will be manipulated. Let's modify the codes above to make it easier to run simulation with different design conditions:

```
> set.seed(1234)
> nrep <- 1000
> n <- c(100, 20)
> v <- c(1, 10)
> result <- rep(NA, nrep)
> for(i in 1:nrep) {
+   y1 <- rnorm(n[1], 0, sqrt(v[1]))
+   y2 <- rnorm(n[2], 0, sqrt(v[2]))
+   y <- c(y1, y2)
+   g <- c(rep(1, n[1]), rep(2, n[2]))
+   dat <- data.frame(g = g, y = y)
+   out <- t.test(y ~ g, data = dat, var.equal = TRUE)
+   result[i] <- out[["p.value"]]
+ }
> mean(result < 0.05)

[1] 0.297
```

In this code, I created n and v to represent sample sizes and variances of each group. These vectors are used to create y1, y2, and g. Note that the sqrt function is used because rnorm takes standard deviation not variance. From this code, if we would like to change sample sizes or variances for data generation, we will only change the values at n and v at the beginning. We do not need to change every line involving sample sizes and variances.

Next, we need to use the same codes to get results from different design conditions. We can use for loops on top of the for loop we have, referred to as nested for loops. The first loop changes sample size condition. The second loop changes the variance condition. The third loop changes the number of replications. In this simulation, the sample size conditions are (20, 100), (40, 80), (60, 60), (80, 40), and (100, 20). The variance conditions are (1, 1), (1, 3), and (1, 10). The first and second values in the parentheses represent values from Groups 1 and 2, respectively. Let's try the nested for loop below:

```
> ns <- matrix(NA, 5, 2)
> ns[,1] <- c(20, 40, 60, 80, 100)
> ns[,2] <- c(100, 80, 60, 40, 20)
> vs <- matrix(1, 3, 2)
> vs[,2] <- c(1, 3, 10)
> nrep <- 2 # Change to 2 here to save space
> for(k in 1:nrow(ns)) {
+   n <- ns[k,]
+   for(j in 1:nrow(vs)) {
+     v <- vs[j,]
+     for(i in 1:nrep) {
+       print(c(n, v, i))
+     }
+   }
+ }
```

```
[1]  20 100   1   1   1
[1]  20 100   1   1   2
[1]  20 100   1   3   1
[1]  20 100   1   3   2
[1]  20 100   1  10   1
[1]  20 100   1  10   2
[1] 40 80  1  1  1
[1] 40 80  1  1  2
[1] 40 80  1  3  1
[1] 40 80  1  3  2
[1] 40 80  1 10  1
[1] 40 80  1 10  2
[1] 60 60  1  1  1
[1] 60 60  1  1  2
[1] 60 60  1  3  1
[1] 60 60  1  3  2
[1] 60 60  1 10  1
[1] 60 60  1 10  2
[1] 80 40  1  1  1
[1] 80 40  1  1  2
[1] 80 40  1  3  1
[1] 80 40  1  3  2
[1] 80 40  1 10  1
[1] 80 40  1 10  2
[1] 100  20   1   1   1
[1] 100  20   1   1   2
[1] 100  20   1   3   1
[1] 100  20   1   3   2
[1] 100  20   1  10   1
[1] 100  20   1  10   2
```

This is the nested for loop varying the design conditions. `ns` is a matrix representing sample sizes. The rows represent different sample size conditions. The columns represent sample size for different groups. `vs` is a matrix representing variances. Again, the rows represent different variance conditions. The columns represent variances for different groups. The first for loop uses `k` as the index of sample size conditions. Inside this for loop, `n` is assigned as k-th row of `ns`. The second for loop uses `j` as the index representing variance conditions. `v` is assigned as j-th row of `vs`. The third for loop represents replications. The `print` function is used to see how nested for loop works.

Let's use the nested for loop for Monte Carlo simulation:

```
> ns <- matrix(NA, 5, 2)
> ns[,1] <- c(20, 40, 60, 80, 100)
> ns[,2] <- c(100, 80, 60, 40, 20)
> vs <- matrix(1, 3, 2)
> vs[,2] <- c(1, 3, 10)
> ovresult <- NULL
> set.seed(1234)
> nrep <- 1000
> for(k in 1:nrow(ns)) {
+   n <- ns[k,]
+   for(j in 1:nrow(vs)) {
+     v <- vs[j,]
+     result <- rep(NA, nrep)
+     for(i in 1:nrep) {
+       y1 <- rnorm(n[1], 0, sqrt(v[1]))
+       y2 <- rnorm(n[2], 0, sqrt(v[2]))
+       g <- c(rep(1, n[1]), rep(2, n[2]))
+       dat <- data.frame(g = g, y = c(y1, y2))
+       out <- t.test(y ~ g, data = dat, var.equal = TRUE)
+       result[i] <- out[["p.value"]]
+     }
+     result <- cbind(k, j, result)
+     ovresult <- rbind(ovresult, result)
+   }
+ }
> head(ovresult)
```

```
     k j     result
[1,] 1 1 0.24922237
[2,] 1 1 0.13487597
[3,] 1 1 0.22053353
[4,] 1 1 0.02229168
[5,] 1 1 0.85918852
[6,] 1 1 0.96445870
```

Note that `ovresult` is created to save the results of each replication from all design conditions and `result` is used to save the results of all replications from a single design condition. `result` are stacked from all design conditions to create `ovresult`. Note that `k` and `j` are bound to `result` as the first and second columns to indicate the design conditions each row coming from. After running all nested for loop, `ovresult` will have three columns: sample size conditions (`k`), variance conditions (`j`), and the p values. Then, we need to summarize the results for all design conditions.

```
> colnames(ovresult) <- c("n", "v", "p")
> ovresult <- data.frame(ovresult, sig = ovresult[,"p"] < 0.05)
> ovresult[,"n"] <- factor(ovresult[,"n"],
+         labels =         c("20,100", "40,80", "60,60", "80,40", "100,20"))
> ovresult[,"v"] <- factor(ovresult[,"v"],
+         labels = c("1,1", "1,3", "1,10"))
> aggregate(sig ~ n + v, data = ovresult, FUN = mean)
```

```
        n    v   sig
1  20,100  1,1 0.044
2   40,80  1,1 0.056
3   60,60  1,1 0.038
4   80,40  1,1 0.045
5  100,20  1,1 0.044
6  20,100  1,3 0.006
7   40,80  1,3 0.024
8   60,60  1,3 0.065
9   80,40  1,3 0.109
10 100,20  1,3 0.172
11 20,100 1,10 0.002
12  40,80 1,10 0.012
13  60,60 1,10 0.041
14  80,40 1,10 0.134
15 100,20 1,10 0.291
```

From the code above, first, column names are attached to `ovresult`. Then, a new variable, `sig`, is created to represent whether each p value is less than .05 and attach to `ovresult`. Because, the first and second columns are numbers, labels could be attached to these numbers to make them easy to understand. These two columns are transformed into the factor format attached with appropriate labels. Finally, the `aggregate` function is used to find the proportion of significant results for each condition.

In sum, to generate codes for Monte Carlo simulation, several steps need to be considered:

- Specify design conditions
- Generate data according to each design condition
- Analyze data with desired statistics
- Extract desired outputs

- Summarize outputs for each design condition

If you can think how to implement each step clearly, the simulation code are simply the combinations of the codes from each step. I hope that this note will make you feel more comfortable to try a Monte Carlo simulation.

1 Exercise

Use the following code to help you evaluate the Type I error of two-sample proportion test. Two-sample proportion test is used to test whether two groups has different proportions in a binary variable (e.g., 1 = yes, 0 = no). The design conditions are as follows:

- Sample size for each group: (20, 100), (40, 80), (60, 60), (80, 40), and (100, 20).
- The proportion of 1 (yes) in a binary variable: .05, .2, .5, .8, .95. Note that the proportion is equal across groups because we would like to evaluate Type I error.

```
> y1 <- rbinom(60, 1, 0.2)
> y2 <- rbinom(60, 1, 0.2)
> yes1 <- sum(y1)
> yes2 <- sum(y2)
> total1 <- length(y1)
> total2 <- length(y2)
> out <- prop.test(x = c(yes1, yes2), n = c(total1, total2))
> out[["p.value"]]
```

```
[1] 0.2356037
```