The Apply Family in R

Sunthud Pornprasertmanit

October 17, 2014

In the previous lecture, we discussed how to use R to communicate with external programs. We used R to write input files for external programs, run the programs, import output files created by the programs, and summarize the results. This lecture will discuss how to use `sapply`, `lapply`, and `mapply` to run repated jobs. R will run the apply family functions faster than for loops. Finally, the apply family functions can be slightly modified to run parallel processing in R.

The example for this lecture is to run a simulation in multiple regression. First, three independent variables are created from multivariate normal distribution with the means of 0, variances of 1, and correlations of .5:

```
> library(MASS)
> set.seed(123321)
> n <- 100
> miv <- rep(0, 3)
> siv <- matrix(0.5, 3, 3)
> diag(siv) <- 1
> dat <- mvrnorm(n, miv, siv)
```

The `mvrnorm` function from the `MASS` package is used to generate data from multivariate normal distribution. This function takes three arguments: (a) sample size, (b) a vector of population means, and (c) a population covariance matrix. In this example, the covariance matrix is the same as correlation matrix because the variances of variables are 1. Next, the dependent variable is created and combined with the independent variables into a single data frame:

```
> y <- 0.3*dat[,1] + 0.4*dat[,2] + 0.2*dat[,3] + rnorm(n, 0, sqrt(0.45))
> dat <- cbind(dat, y)
> colnames(dat) <- c("x1", "x2", "x3", "y")
> dat <- as.data.frame(dat)
```

Next, multiple regression is run using the `lm` function:

```
> out <- lm(y ~ x1 + x2 + x3, data = dat)
> sumout <- summary(out)
> names(sumout)
```

```
 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "aliased"       "sigma"         "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

```
> sumout[["coefficients"]]
```

```
               Estimate Std. Error    t value     Pr(>|t|)
(Intercept) -0.02674595 0.06442600 -0.4151423 6.789641e-01
x1           0.34409217 0.08203367  4.1945236 6.096608e-05
x2           0.39326533 0.07933378  4.9570984 3.077763e-06
x3           0.20906357 0.07875049  2.6547590 9.291468e-03
```

As explained in previous lectures, the `summary` function needs to be used on the result of the `lm` function to get parameter estimates. Note that, in this lecture, we will use the following objects in the results of the `summary` function: `"residuals"`, `"coefficients"`, `"r.squared"`, and `"adj.r.squared"`.

In this lecture, we will also use likelihood-ratio test, also known as the test for hierarchical regression. Likelihood-ratio test can compare between two nested models. For example, the model with two predictors of `"x1"` and `"x2"` is nested in the model with more predictors that have `"x1"` and `"x2"` in the list of predictors. In this lecture, we will compare the model with the predictors of `"x1"` and `"x2"` and the model with the predictors of `"x1"`, `"x2"`, and `"x3"`. The `anova` function is used to compare two nested models.

```
> out2 <- lm(y ~ x1 + x2, data = dat)
> anova(out2, out)
```

```
Analysis of Variance Table

Model 1: y ~ x1 + x2
Model 2: y ~ x1 + x2 + x3
  Res.Df    RSS Df Sum of Sq      F   Pr(>F)
1     97 40.812
2     96 38.020  1    2.7912 7.0477 0.009291 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The nested model comparison was significant. That is, `"x3"` can explain the dependent variable in addition to `"x1"` and `"x2"`. Now, let's generate multiple data sets and show how to use the apply family function:

```
> nrep <- 3
> n <- 100
> miv <- rep(0, 3)
> siv <- matrix(0.5, 3, 3)
> diag(siv) <- 1
> dat.l <- list()
```

```
> set.seed(123321)
> for(i in 1:nrep) {
+   dat <- mvrnorm(n, miv, siv)
+   y <- 0.3*dat[,1] + 0.4*dat[,2] + 0.2*dat[,3] + rnorm(n, 0, sqrt(0.45))
+   dat <- cbind(dat, y)
+   colnames(dat) <- c("x1", "x2", "x3", "y")
+   dat.l[[i]] <- as.data.frame(dat)
+ }
```

From this code, `dat.l` is a list that has 1,000 data sets in each element. In the previous lecture, we can use for loop to analyze each data set and extract the results:

```
> resultcoef <- matrix(NA, nrep, 4)
> for(i in 1:nrep) {
+   out <- lm(y ~ x1 + x2 + x3, data = dat.l[[i]])
+   sumout <- summary(out)
+   resultcoef[i,] <- sumout$coefficients[,1]
+ }
> apply(resultcoef, 2, mean)

[1] -0.005795595  0.376371516  0.397040455  0.170700904
```

Now, let's introduce the `sapply` function. This function will apply a specified function in each element of the list, collect the results that are in the vector format (including a vector with one element), and stack them up. Let's start with a simple function that evaluate the number of rows in each data set and stack them up using the `sapply` function.

```
> sapply(dat.l, nrow)

[1] 100 100 100
```

The first element of the `sapply` function is the desired list. The second element is the function used to evaluate each element of the list. The result is a vector representing the number of rows in each data set in the list. Let's change the target function to the `colMeans` function.

```
> sapply(dat.l, colMeans)

          [,1]       [,2]       [,3]
x1  0.03720808 0.08449270 -0.1948983
x2 -0.14899379 0.06522515 -0.1988672
x3 -0.06497298 0.04133854 -0.1301627
y  -0.08612052 0.09982822 -0.1906075
```

The result is a matrix that columns represent the results from each data set and rows represent the means of each variable. Let's create a simple function that returns the mean and standard deviation of the dependent variable and use the `sapply` function to simultaneously evaluate multiple data sets.

```
> dvstat <- function(dat) c(mean(dat[,4]), sd(dat[,4]))
> sapply(dat.l, dvstat)

            [,1]       [,2]       [,3]
[1,] -0.08612052 0.09982822 -0.1906075
[2,]  1.05160941 1.11867046  0.9007864
```

If the created function has only one line, users may build the function inside `sapply`. However, I do not recommend beginners to write this code because it is very hard to read. Some developers prefer this style of writing codes because they do not need to name the temporary function.

```
> sapply(dat.l, function(dat) c(mean(dat[,4]), sd(dat[,4])))

            [,1]       [,2]       [,3]
[1,] -0.08612052 0.09982822 -0.1906075
[2,]  1.05160941 1.11867046  0.9007864
```

Let's create a function that analyzes multiple regression and returns the regression coefficients. Use the `sapply` function to evaluate all data sets simultaneously.

```
> analysisfun <- function(dat) {
+   out <- lm(y ~ x1 + x2 + x3, data = dat)
+   sumout <- summary(out)
+   sumout$coefficients[,1]
+ }
> sapply(dat.l, analysisfun)

                   [,1]       [,2]        [,3]
(Intercept) -0.02674595 0.03004525 -0.02068608
x1           0.34409217 0.51183898  0.27318340
x2           0.39326533 0.32603833  0.47181771
x3           0.20906357 0.12749417  0.17554497
```

Why don't we compare the time elapsed between the for loop and the `sapply` function. We can use `system.time` function to measure the using time. However, first, let's create more data sets to easily compare the efficiency between two methods.

```
> ########## Data Generation ##########
> nrep2 <- 10000
> miv <- rep(0, 3)
> siv <- matrix(0.5, 3, 3)
> diag(siv) <- 1
> dat.l2 <- list()
> set.seed(123321)
> for(i in 1:nrep2) {
```

```
+   dat <- mvrnorm(n, miv, siv)
+   y <- 0.3*dat[,1] + 0.4*dat[,2] + 0.2*dat[,3] + rnorm(n, 0, sqrt(0.45))
+   dat <- cbind(dat, y)
+   colnames(dat) <- c("x1", "x2", "x3", "y")
+   dat.l2[[i]] <- as.data.frame(dat)
+ }
> ########## For loop ##########
> system.time({
+ resultcoef <- matrix(NA, nrep2, 4)
+ for(i in 1:nrep2) {
+   out <- lm(y ~ x1 + x2 + x3, data = dat.l2[[i]])
+   sumout <- summary(out)
+   resultcoef[i,] <- sumout$coefficients[,1]
+ }
+ })

   user  system elapsed
  12.13    0.00   12.16

> ########## sapply ##########
> system.time(resultcoef2 <- sapply(dat.l2, analysisfun))

   user  system elapsed
  12.51    0.00   12.51
```

The `sapply` function took slightly shorter time than the for loop. Let's talk about the `lapply` function. This function is similar to the `sapply` function. However, the result is returned as a list. For example, let's use the `lapply` function to implement the `colMeans` function.

```
> lapply(dat.l, colMeans)

[[1]]
         x1          x2          x3           y
 0.03720808 -0.14899379 -0.06497298 -0.08612052

[[2]]
        x1         x2         x3          y
0.08449270 0.06522515 0.04133854 0.09982822

[[3]]
        x1         x2         x3          y
-0.1948983 -0.1988672 -0.1301627 -0.1906075
```

The results are returned as a list containing three vectors. This format is not convenient to be used later. However, the advantage is that the results of the desired function are not needed to be a vector. For example, the `cor` function returns a matrix. We can find the correlation matrix of each data set as follows:

```
> lapply(dat.l, cor)

[[1]]
          x1        x2        x3         y
x1 1.0000000 0.6056883 0.6094212 0.7119609
x2 0.6056883 1.0000000 0.5212995 0.7072599
x3 0.6094212 0.5212995 1.0000000 0.6230868
y  0.7119609 0.7072599 0.6230868 1.0000000

[[2]]
          x1        x2        x3         y
x1 1.0000000 0.6035561 0.6165415 0.7695481
x2 0.6035561 1.0000000 0.6197049 0.6961905
x3 0.6165415 0.6197049 1.0000000 0.6325199
y  0.7695481 0.6961905 0.6325199 1.0000000

[[3]]
          x1        x2        x3         y
x1 1.0000000 0.3827698 0.5357984 0.5673337
x2 0.3827698 1.0000000 0.5036760 0.6775746
x3 0.5357984 0.5036760 1.0000000 0.5758519
y  0.5673337 0.6775746 0.5758519 1.0000000
```

If we would like to pass other arguments to the desired function, the extra arguments can be added in the `sapply` or `lapply` function as the third argument, the fourth argument, and so on. For example, I would like to find the spearman rank correlation from each data set. The `cor` function is used and the `method` argument is specified as `"spearman"`. We can add the method argument in the `lapply` function as the third argument:

```
> lapply(dat.l, cor, method = "spearman")

[[1]]
          x1        x2        x3         y
x1 1.0000000 0.5607921 0.5463666 0.6512931
x2 0.5607921 1.0000000 0.4840084 0.6539694
x3 0.5463666 0.4840084 1.0000000 0.6052565
y  0.6512931 0.6539694 0.6052565 1.0000000

[[2]]
          x1        x2        x3         y
x1 1.0000000 0.5967597 0.5660246 0.7474347
x2 0.5967597 1.0000000 0.6164656 0.6846325
x3 0.5660246 0.6164656 1.0000000 0.6028683
y  0.7474347 0.6846325 0.6028683 1.0000000
```

```
[[3]]
          x1        x2        x3         y
x1 1.0000000 0.3774137 0.5176118 0.5622442
x2 0.3774137 1.0000000 0.5064506 0.6490969
x3 0.5176118 0.5064506 1.0000000 0.5365137
y  0.5622442 0.6490969 0.5365137 1.0000000
```

Now, we get the spearman rank correlation of each data set. Usually, each element in the list will be passed to the first argument of the desired function. For example, data sets are passed to the `x` argument, which is the first argument, in the `cor` function. Note that the order of the arguments can always be checked in the help page of each function. However, if we would like to analyze multiple regression for each data set, the data set is specified as the second argument. The first argument is `formula`. To pass data to the second argument, we need to specify the first argument (i.e., `formula`) as an additional argument in the `sapply` or `lapply` functions:

```
> output.l <- lapply(dat.l, lm, formula = y ~ x1 + x2 + x3)
> output.l

[[1]]

Call:
FUN(formula = ..1, data = X[[1L]])

Coefficients:
(Intercept)           x1           x2           x3
   -0.02675      0.34409      0.39327      0.20906


[[2]]

Call:
FUN(formula = ..1, data = X[[2L]])

Coefficients:
(Intercept)           x1           x2           x3
    0.03005      0.51184      0.32604      0.12749


[[3]]

Call:
FUN(formula = ..1, data = X[[3L]])

Coefficients:
```

```
(Intercept)           x1           x2           x3
   -0.02069      0.27318      0.47182      0.17554
```

Because the `formula` argument is already specified, the data sets in the list are passed to the second argument of the `lm` function. Because `output.l` is a list, we can use `lapply` to find the summary of each output:

```
> sumoutput.l <- lapply(output.l, summary)
> sumoutput.l

[[1]]

Call:
FUN(formula = ..1, data = X[[1L]])

Residuals:
     Min       1Q   Median       3Q      Max
-1.47710 -0.53264  0.00155  0.46908  1.21757

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.02675    0.06443  -0.415  0.67896
x1           0.34409    0.08203   4.195 6.10e-05 ***
x2           0.39327    0.07933   4.957 3.08e-06 ***
x3           0.20906    0.07875   2.655  0.00929 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.6293 on 96 degrees of freedom
Multiple R-squared:  0.6527,        Adjusted R-squared:  0.6419
F-statistic: 60.15 on 3 and 96 DF,  p-value: < 2.2e-16


[[2]]

Call:
FUN(formula = ..1, data = X[[2L]])

Residuals:
     Min       1Q   Median       3Q      Max
-1.60956 -0.42591  0.02996  0.42260  1.72659

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.03005    0.06393   0.470 0.639449
```

```
x1           0.51184    0.07970   6.422 5.14e-09 ***
x2           0.32604    0.08111   4.020 0.000116 ***
x3           0.12749    0.07786   1.637 0.104805
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.6371 on 96 degrees of freedom
Multiple R-squared:  0.6855,        Adjusted R-squared:  0.6756
F-statistic: 69.73 on 3 and 96 DF,  p-value: < 2.2e-16


[[3]]

Call:
FUN(formula = ..1, data = X[[3L]])

Residuals:
     Min       1Q   Median       3Q      Max
-1.27899 -0.44164  0.01811  0.44618  1.28248

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.02069    0.06046  -0.342 0.732994
x1           0.27318    0.07436   3.674 0.000393 ***
x2           0.47182    0.07590   6.216 1.31e-08 ***
x3           0.17554    0.08072   2.175 0.032113 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.5854 on 96 degrees of freedom
Multiple R-squared:  0.5904,        Adjusted R-squared:  0.5776
F-statistic: 46.13 on 3 and 96 DF,  p-value: < 2.2e-16
```

Next, let's extract the regression coefficients of each output. One way to implement it is to create a function to extract the regression coefficients:

```
> extractcoef <- function(out) out[["coefficients"]][,2]
> coef.l <- lapply(sumoutput.l, extractcoef)
> coef.l

[[1]]
(Intercept)          x1          x2          x3
 0.06442600  0.08203367  0.07933378  0.07875049

[[2]]
```

```
(Intercept)          x1          x2          x3
 0.06393137  0.07970160  0.08110604  0.07786009

[[3]]
(Intercept)          x1          x2          x3
 0.06046093  0.07436182  0.07589966  0.08072284
```

Next, we may want to collapse the list of vectors into a matrix.[1] We can run a for loop to bind the first and the second elements. Then, the result is bound to the third element. This process is ran until all elements being attached to the result. Instead of using the for loop, we can use the `do.call` function:

```
> do.call(rbind, coef.l)

     (Intercept)         x1         x2         x3
[1,]  0.06442600 0.08203367 0.07933378 0.07875049
[2,]  0.06393137 0.07970160 0.08110604 0.07786009
[3,]  0.06046093 0.07436182 0.07589966 0.08072284
```

The process inside the `do.call` function is similar to the for loop described above. However, the `do.call` function is the shorten version of the for loop.

Instead of creating the function to extract the coefficients, we may use the `"[["` and `"["` functions. `"[["` and `"["` are the double and single square brackets that we are using to extract elements from a vector or a matrix (e.g., `attitude[2, 5]`). To use these square brackets in the `sapply` or `lapply` functions, we can specify `"[["` and `"["` as the desired functions:

```
> coef.l <- lapply(sumoutput.l, "[[", "coefficients")
> sapply(coef.l, "[", 1:4, 1)

                   [,1]       [,2]        [,3]
(Intercept) -0.02674595 0.03004525 -0.02068608
x1           0.34409217 0.51183898  0.27318340
x2           0.39326533 0.32603833  0.47181771
x3           0.20906357 0.12749417  0.17554497
```

The first line extracts the coefficients element from the output of the `summary` function. The second line extracts the elements from Rows 1–4 and Column 1 from the coefficients table. The `sapply` function is used to collapse resulting vectors into a single matrix. The help pages of the square bracket can be found by `?"[["` or `?"["`.

As another example, we can extract the residuals of the dependent variables from each output:

```
> residual.l <- lapply(sumoutput.l, "[[", "residuals")
```

[1] We do not need to do it if `sapply` is used at the first place.

Next, the `mapply` function evaluates more than two lists and use the specified function to the element at the same position of each list. For example, the first list is the results of one model and the second list is the results of another model. We would like to use the `anova` function to compare the results of two lists together:

```
> output1.l <- lapply(dat.l, lm, formula = y ~ x1 + x2 + x3)
> output2.l <- lapply(dat.l, lm, formula = y ~ x1 + x2)
> mapply(anova, output1.l, output2.l, SIMPLIFY = FALSE)

[[1]]
Analysis of Variance Table

Model 1: y ~ x1 + x2 + x3
Model 2: y ~ x1 + x2
  Res.Df    RSS Df Sum of Sq      F   Pr(>F)
1     96 38.020
2     97 40.812 -1   -2.7912 7.0477 0.009291 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

[[2]]
Analysis of Variance Table

Model 1: y ~ x1 + x2 + x3
Model 2: y ~ x1 + x2
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     96 38.969
2     97 40.058 -1   -1.0884 2.6813 0.1048

[[3]]
Analysis of Variance Table

Model 1: y ~ x1 + x2 + x3
Model 2: y ~ x1 + x2
  Res.Df    RSS Df Sum of Sq      F  Pr(>F)
1     96 32.902
2     97 34.523 -1   -1.6208 4.7292 0.03211 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The first element of the `mapply` function is the desired function, which is different from `sapply` and `lapply`. Then, the following arguments are the lists that will be used in the specified function, which is `anova` here. Finally, the `SIMPLIFY` argument is to collapse results into a matrix (like the `sapply` function). In this case, the resulting list format is preferred (like the `lapply` function) so the `SIMPLIFY` argument is specified as `FALSE`. Next, let's create

a function to find the difference in R^2. The function has an option to use the regular R^2 or adjusted R^2. Then the `mapply` function is used to apply the created function to each element of both lists simultaneously.

```
> diffrsquared <- function(out1, out2, adjust = FALSE) {
+   name <- "r.squared"
+   if(adjust) name <- "adj.r.squared"
+   rsquared1 <- summary(out1)[[name]]
+   rsquared2 <- summary(out2)[[name]]
+   abs(rsquared1 - rsquared2)
+ }
> mapply(diffrsquared, output1.l, output2.l)

[1] 0.02549474 0.00878541 0.02017688

> mapply(diffrsquared, output1.l, output2.l, MoreArgs = list(adjust = TRUE))

[1] 0.022328383 0.005622488 0.016238449
```

If the `mapply` function is used to both lists directly (without any additional arguments), the regular R^2 are provided because `adjust` is `FALSE` by default. If we would like to specify the additional arguments, the `MoreArgs` argument in the `mapply` function is used. Then, additional arguments are specified in a list. Note that we do not use the `SIMPLIFY` argument here because the output in a vector format is preferred. As another example, we may find the correlation between the predicted values of two outputs:

```
> corpredict <- function(out1, out2) {
+   cor(predict(out1), predict(out2))
+ }
> mapply(corpredict, output1.l, output2.l)

[1] 0.9802761 0.9935709 0.9827645
```

Now we know the `sapply`, `lapply`, and `mapply` functions. We can use the knowledge of these functions to run a parallel processing. During these days, computer have multiple processors. R usually use only one processor to run everything. We need some functions to ask R to assign jobs to multiple processors. The `parallel` package is used here. The procedures to run parallel processing for Windows and for Mac/Linux are different. Let's discuss the procedures for Mac/Linux first, which is simpler. For example, we would like to analyze data sets by the `lm` function using the `lapply` function:

```
> output.l <- lapply(dat.l, lm, formula = y ~ x1 + x2 + x3)
```

The `lapply` function can be replaced by the `mclapply` function.

```
> library(parallel)
> output.l <- mclapply(dat.l, lm, formula = y ~ x1 + x2 + x3, mc.cores = 2)
```

The `mc.cores` argument represents the number of processors used in the parallel processing. Note that `mcmapply` is the parallel-processing version of the `mapply` function. In Windows, the `parLapply` function is used. However, users need to specify the group of processors first. Then, the parallel processing by the `parLapply` function can be run.

```
> cl <- makeCluster(rep("localhost", 2), type = "PSOCK")
> output.l <- parLapply(cl, dat.l, lm, formula = y ~ x1 + x2 + x3)
> stopCluster(cl)
```

The `makeCluster` function is used to specify the group of processors. The first argument represents the type of nodes to be used. In this case, we use two hosts from local computers because `"localhost"` is used twice. The `type` argument represents the type of parallel processing. The `"PSOCK"` or `"SOCK"` are only available on Windows. Then, the `parLapply` function is used in the similar way to the `lapply` function. However, the first argument is the group of processors created by the `makeCluster` function. Finally, the group of clusters are terminated by the `stopCluster` function. Note that `parApply`, `parSapply`, and `clusterMap` are the parallel-processing versions of `apply`, `sapply`, and `mapply` respectively. If users do not know the number of available processors in their computer, the `detectCores` function can be used:

```
> detectCores()

[1] 8
```

1 Exercise

In independent-sample t-test, there are two methods to compare the mean differences: regular t-test (equal variances assumed) and Welch test (equal variances not assumed). Both tests can be used to find the confidence interval of the mean differences. Modify the following code to (a) find the proportion of the widths from both methods covering the population difference of 0.5, (b) find the means and standard deviations of widths of confidence intervals from both methods, (c) find the mean and standard deviation of the difference in width from both methods. The sample size is fixed to 100 per group. The data from the first group is drawn from normal distribution with the mean of 0 and standard deviation of 1. The data from the second group is drawn from normal distribution with the mean of 0.5 and standard deviation of 1. The number of replications is 1,000.

```
> set.seed(123321)
> g1 <- cbind(1, rnorm(100, 0, 1))
> g2 <- cbind(2, rnorm(100, 0.5, 1))
> dat <- data.frame(rbind(g1, g2))
> colnames(dat) <- c("group", "y")
> outeq <- t.test(y ~ group, data = dat, var.equal = TRUE)
> outnoteq <- t.test(y ~ group, data = dat)
> cieq <- outeq[["conf.int"]]
```

```
> cineq <- outnoteq[["conf.int"]]
> widtheq <- cieq[2] - cieq[1]
> widthneq <- cineq[2] - cineq[1]
> diffwidth <- widtheq - widthneq
```