Control structures

There are instances when one will want to execute a set of code only if a certain condition holds. Also, there are instances when one will want to repeat a set of code multiple times. The purpose of this section is to provide the basics for these types of control structures in R. The programs used in this section are gpa_cond.R and Estimated_true_conf_level.R.

Conditional execution

We have already seen a few examples for specifying conditional operations. For example, in the regression section, we used the following code to limit a data frame:

```
> # Location is for my computer
> gpa <- read.table(file = "C:\\data\\GPA.txt", header = TRUE,
     sep = "")
> head(gpa)
  HS.GPA College.GPA
1   3.04        3.10
2   2.35        2.30
3   2.70        3.00
4   2.55        2.45
5   2.83        2.50
6   4.32        3.70
> gpa$HS.GPA < 2.5
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
[12] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE
> gpa[gpa$HS.GPA < 2.5, ]
   HS.GPA College.GPA
2    2.35         2.3
8    2.32         2.6
11   2.39         2.0
15   2.22         2.8
16   1.98         2.4
```

```
19   2.28         2.2
> sum(gpa$HS.GPA < 2.5)
[1] 6
> gpa$HS.GPA == 2.35  # Equal
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> gpa$HS.GPA != 2.35  # Not equal
 [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

The `ifelse()` function performs a similar logical comparison:

```
> # If then else - could use yes = 1 and no = 0
> # too
> test.cond1 <- ifelse(test = gpa$HS.GPA < 2.5,
     yes = TRUE, no = FALSE)
> sum(test.cond1)
[1] 6
> # If then else - note that '&' means 'and'
> test.cond2 <- ifelse(test = gpa$HS.GPA < 2.5 &
     gpa$College.GPA < 2.5, yes = 1, no = 0)
> sum(test.cond2)
[1] 4
> # Nested ifelse()
> test.cond3 <- ifelse(test = gpa$HS.GPA < 2.5,
     yes = ifelse(test = gpa$College.GPA < 2.5,
         yes = 1, no = 0), no = 0)
> sum(test.cond3)
[1] 4
> # If then else - note that '|' means 'or'
> test.cond4 <- ifelse(test = gpa$HS.GPA < 2.5 |
     gpa$College.GPA < 2.5, yes = 1, no = 0)
> sum(test.cond4)
[1] 7
```

The `ifelse()` function is useful for more complicated resulting values from the comparison than those shown above. An important aspect of all comparisons from `ifelse()` is that they are

done on each element of a vector. A new vector of the same size is created from it that gives the appropriate outcomes.

When an "if then" framework is not needed to apply with a vector, the `if()` function should be used instead. For a very simplistic example that also includes an `else` condition, see how the data frame is printed below.

```
> print.head <- TRUE
> simple.func <- function(print.head) {
      if (print.head == TRUE) {
          head(gpa)
      } else {
          tail(gpa)
      }
 }
> simple.func(print.head = TRUE)
  HS.GPA College.GPA
1   3.04        3.10
2   2.35        2.30
3   2.70        3.00
4   2.55        2.45
5   2.83        2.50
6   4.32        3.70
> simple.func(print.head = FALSE)
   HS.GPA College.GPA
15   2.22         2.8
16   1.98         2.4
17   2.88         2.6
18   4.00         3.8
19   2.28         2.2
20   2.88         2.6
```

Error messages may occur when using `if()` and `else` together when the `else` statement is on a line by itself rather than with the previous `}`. This occurs because `if()` does not need to have an `else` portion to it and R may interpret the entire syntax as concluding prior to `else`.

One example of where I have used `if()` and `else` in practice

is when I create functions meant to be used in general settings. For example, the `examine.logistic.reg()` function written for my *Analysis of Categorical Data with R* book creates a number of plots to evaluate a logistic regression model fit.[1] A number of `if()` functions are used in the function to add or remove items from plots depending on user preferences. For instance, examine how I use the `identify.points` argument in the partial function listing below.

```
> examine.logistic.reg <- function(mod.fit.obj = mod.fit,
    identify.points = TRUE, bubble = TRUE, scale.n = I,
    scale.cookd = I, pearson.dev = "Pearson") {

    # Code excluded above here

    plot(x = pred, y = resid.plot11, xlab = "Estimated probabilities",
        ylab = "Standardized residuals", main = paste("Standardized",
            plot.label11, "residuals vs. est. prob."),
        ylim = c(min(-3, stand.resid), max(3, stand.resid)))
    abline(h = c(-3, -2, 0, 2, 3), lty = "dotted",
        col = "blue")
    if (identify.points == TRUE) {
        identify(x = pred, y = resid.plot11, labels = labels(pred))
    }

    # Code excluded below here

 }
```

If `identify.points = TRUE`, R will allow the user to click on "outliers" in a corresponding plot.

When you want to test a condition that leads to more than two options of what to do next, one could use nested `if()` and `else` statements. More simply, one could use the `switch()` function. Below is a simple example of using both to calculate measures of centrality.

[1] See Examine.logistic.reg.R in the Chapter 5 programs at `http://www.chrisbilder.com/categorical/programs_and_data.html`.

```
> # Nested if() else
> mid.value1 <- function(x, type = "mean") {
     if (type == "mean") {
         mean(x)
     } else {
         if (type == "median") {
             median(x)
         } else {
             print("Invalid type requested")
         }
     }
 }
> mid.value1(x = gpa$HS.GPA, type = "mean")
[1] 2.899
> mid.value1(x = gpa$HS.GPA, type = "median")
[1] 2.83
> mid.value1(x = gpa$HS.GPA, type = "Bilder method")
[1] "Invalid type requested"
> # switch()
> mid.value2 <- function(x, type = "mean") {
     switch(EXPR = type, mean = mean(x), median = median(x),
         print("Invalid type requested"))
 }
> mid.value2(x = gpa$HS.GPA, type = "mean")
[1] 2.899
> mid.value2(x = gpa$HS.GPA, type = "median")
[1] 2.83
> mid.value2(x = gpa$HS.GPA, type = "Bilder method")
[1] "Invalid type requested"
```

Notice the last value within `switch()` does not have a particular `type` listed with it. Because this value is given last, `"Invalid type requested"` becomes the default if `"mean"` or `"median"` are not given for `type`. If this last value was not given and there was no match for `type`, `switch()` will not return any value.

Question:

Suppose that when the argument value of `type` is `"mean"`, you would like to have the mean and variance calculated. What would need to change in the `mid.value2()` function?

Loops

Loops are used to repeat the same set of code a number of times (i.e., iterations). These structures should be avoided IF one can do the same calculations by taking advantage of R's ability to perform calculations on vectors of information. This alternative to loops generally will result in code which is 1) much more efficient, 2) easier to read, and 3) faster. To demonstrate how to construct a loop and provide different levels of efficiency, we are going to examine next a Monte Carlo simulation to estimate the true confidence level of a t-distribution based confidence interval for a population mean.

The most often used loop function in R is the `for()` function. This function allows one to repeat the following calculations:

- Simulate a data set
- Calculate the confidence interval for the data set
- Check if the confidence interval contains the true population mean μ

Once this process is repeated a large number times, we can compute the percentage of times overall that the confidence interval contains μ. This percentage is the *estimated* true confidence level.

The main goal for any confidence interval is for this estimated true confidence level to be close to the *stated* confidence level. Thus, if the stated confidence level is 95% for a particular type of interval, we would like this interval to contain or "cover" the parameter value approximately 95% of the time. For a more in-depth explanation of Monte Carlo simulation to examine the

quality of statistical inference, please see my lecture notes for STAT 950 at `http://www.chrisbilder.com/compstat`!

Below is my R code and output for a VERY INEFFICIENT way to perform the Monte Carlo simulation.

```
> alpha <- 0.05  # Stated confidence level is 1-alpha
> num.samples <- 1000  # Number of simulated data sets
>
> # Simulate one data set
> sample.size <- 10
> mu <- 2
> sigma <- 3
> set.seed(4778)
> y <- rnorm(n = sample.size, mean = mu, sd = sigma)
>
> # Calculate interval
> lower <- mean(y) + qt(p = alpha/2, df = sample.size -
    1) * sd(y)/sqrt(sample.size)
> upper <- mean(y) + qt(p = 1 - alpha/2, df = sample.size -
    1) * sd(y)/sqrt(sample.size)
> c(lower, upper)
[1] 0.9624 4.3158
>
> # Check if mu is interval - multiple ways
> ifelse(test = lower < mu, yes = ifelse(test = upper >
    mu, yes = 1, no = 0), no = 0)
[1] 1
> ifelse(test = lower < mu & upper > mu, yes = 1, no = 0)
[1] 1
> lower < mu & upper > mu
[1] TRUE
>
> # Complete same type of calculations for all
> # samples
> set.seed(4778)
> save.results1 <- matrix(data = NA, nrow = num.samples,
    ncol = 3)
> for (i in 1:num.samples) {
```

```
    y <- rnorm(n = sample.size, mean = mu, sd = sigma)
    lower <- mean(y) + qt(p = alpha/2, df = sample.size -
        1) * sd(y)/sqrt(sample.size)
    upper <- mean(y) + qt(p = 1 - alpha/2, df = sample.size -
        1) * sd(y)/sqrt(sample.size)
    check <- lower < mu & upper > mu
    save.results1[i, ] <- c(check, lower, upper)
 }
>
> # Estimated true confidence level - two ways
> sum(save.results1[, 1])/num.samples
[1] 0.949
> mean(save.results1[, 1])
[1] 0.949
```

Comments:

- $\{Y_i\}_{i=1}^{n} \overset{\text{ind.}}{\sim} N(\mu, \sigma^2)$ where $n = 10$, $\mu = 2$, and $\sigma = 3$
- The $(1-\alpha)100\%$ confidence interval is $\bar{y} \pm t_{1-\alpha/2,n-1}\frac{s}{\sqrt{n}}$ where $\alpha = 0.05$, $\bar{y}$ is the sample mean, and s is the sample standard deviation. In my code, I use the fact that $t_{\alpha/2,n-1}$ is the same as $-t_{1-\alpha/2,n-1}$ due to symmetry of the t-distribution.
- A seed number should always be set so that one can reproduce the results!
- Three ways to check if μ is within the interval are given, where the last way is the most efficient.
- The `for()` function cycles over the numbers of 1, 2, ..., 1000 to perform the same calculations for each simulated data set.
- The results saved for each iteration from `for()` are saved in a matrix.
- The estimated true confidence level is 0.949. This is quite close to the stated confidence level of 0.95, which is expected because the underlying normality assumption for the interval is satisfied.

Generally, a better way to perform a Monte Carlo simulation is to simulate all of the data at first before any loop (in some instances, it may not be possible though). Also, the interval can be calculated with one code statement rather than two. Below is the code and output:

```
> set.seed(4778)
> y <- matrix(data = rnorm(n = num.samples * sample.size,
     mean = mu, sd = sigma), nrow = num.samples, ncol = sample.size,
     byrow = TRUE)
> head(y, n = 3)
       [,1]   [,2]   [,3]   [,4]    [,5]    [,6]  [,7]    [,8]   [,9]
[1,]  7.260 5.2780  2.128 0.1013  0.9218 -0.1762 3.171  3.6887 2.8389
[2,] -3.545 0.1018 -0.110 4.0861  0.8426  4.8302 3.934  0.8311 0.3605
[3,]  3.588 7.3571  4.986 6.8842 -2.1443  3.5188 1.370 -1.3706 1.7507
      [,10]
[1,]  1.180
[2,]  8.976
[3,] -3.241
> tail(y, n = 3)
           [,1]  [,2]    [,3]    [,4]  [,5]    [,6]    [,7]   [,8]    [,9]
[998,]   2.2510 1.265  3.1278 -0.8979 5.385  5.6698  4.7090 -3.567  3.1891
[999,]  -0.9509 2.071 -0.7176  5.9421 0.477 -0.3691  3.3785  6.700  0.7561
[1000,]  0.1936 6.500 -2.0318 11.4471 2.481  1.8167 -0.3782 -0.557 -2.0951
         [,10]
[998,]   4.099
[999,]   1.701
[1000,] -1.189
> # Calculate interval for one data set
> interval <- mean(y[1, ]) + qt(p = c(alpha/2, 1 - alpha/2),
     df = sample.size - 1) * sd(y[1, ])/sqrt(sample.size)
> interval
[1] 0.9624 4.3158
> interval[1] < mu & interval[2] > mu
[1] TRUE
> # Complete same type of calculations for all
> # samples
> save.results2 <- matrix(data = NA, nrow = num.samples,
```

```
     ncol = 3)
> for (i in 1:num.samples) {
     interval <- mean(y[i, ]) + qt(p = c(alpha/2, 1 -
         alpha/2), df = sample.size - 1) * sd(y[i, ])/sqrt(sample.size)
     check <- interval[1] < mu & interval[2] > mu
     save.results2[i, ] <- c(check, interval[1], interval[2])
 }
> # Estimated true confidence level
> mean(save.results2[, 1])
[1] 0.949
```

In the above code, make sure you understand that each row of y is a separate data set.

Can we do better? Yes, we can! We can take advantage of R's vector calculations by checking outside of the loop if μ is within the interval.

```
> save.results3 <- matrix(data = NA, nrow = num.samples,
     ncol = 2)
> for (i in 1:num.samples) {
     interval <- mean(y[i, ]) + qt(p = c(alpha/2, 1 -
         alpha/2), df = sample.size - 1) * sd(y[i, ])/sqrt(sample.size)
     save.results3[i, ] <- c(interval[1], interval[2])
 }
> # Estimated true confidence level
> check <- save.results3[, 1] < mu & save.results3[,
     2] > mu
> mean(check)
[1] 0.949
```

Rather than each row being its own simulated data set, sometimes simulated data is structured so that the first n rows are a data set, the next n rows are another data set, ... Below is an example of how one can extract each data set in the loop for the same Monte Carlo simulation.

```
> set.seed(4778)
> y <- matrix(data = rnorm(n = num.samples * sample.size,
```

```
    mean = mu, sd = sigma), nrow = num.samples * sample.size,
    ncol = 1, byrow = TRUE)
> head(y)
        [,1]
[1,]  7.2597
[2,]  5.2780
[3,]  2.1278
[4,]  0.1013
[5,]  0.9218
[6,] -0.1762
> y.df <- as.data.frame(y)
> head(y.df)
       V1
1  7.2597
2  5.2780
3  2.1278
4  0.1013
5  0.9218
6 -0.1762
> tail(y.df)
           V1
9995   2.4814
9996   1.8167
9997  -0.3782
9998  -0.5570
9999  -2.0951
10000 -1.1891
> save.results4 <- matrix(data = NA, nrow = num.samples,
    ncol = 2)
> for (i in 1:num.samples) {
    indices <- (sample.size * (i - 1) + 1):(i * sample.size)
    interval <- mean(y[indices]) + qt(p = c(alpha/2,
        1 - alpha/2), df = sample.size - 1) * sd(y[indices])/sqrt(sample.size)
    save.results4[i, ] <- c(interval[1], interval[2])
 }
> # Estimated true confidence level
> check <- save.results4[, 1] < mu & save.results4[,
    2] > mu
> mean(check)
```

```
[1] 0.949
```

When code like the above is initially written, there will often be errors preventing parts of it from running. Debugging is used to determine where the errors are located, leading one to ideas regarding how to fix it. When using loops, I STRONGLY recommend that you get the code to work for one iteration FIRST. You can then proceed to a larger number of iterations. Also, another good programming habit is to use `print()` and `cat()` functions inside of a loop to see what is being calculated. For example, below is the same use of `for()` as in the last example but now with strategically put `print()` and `cat()` functions.

```
> save.results4 <- matrix(data = NA, nrow = num.samples,
      ncol = 2)
> for (i in 1:3) {
      indices <- (sample.size * (i - 1) + 1):(i * sample.size)
      cat("Iteration", i, "where indices =", indices,
          "\n")
      interval <- mean(y[indices]) + qt(p = c(alpha/2,
          1 - alpha/2), df = sample.size - 1) * sd(y[indices])/sqrt(sample.size)
      print(interval)
      save.results4[i, ] <- c(interval[1], interval[2])
 }
Iteration 1 where indices = 1 2 3 4 5 6 7 8 9 10
[1] 0.9624 4.3158
Iteration 2 where indices = 11 12 13 14 15 16 17 18 19 20
[1] -0.4584  4.5198
Iteration 3 where indices = 21 22 23 24 25 26 27 28 29 30
[1] -0.3641  4.9040
```

Note that the only way you can have something displayed to a R Console window from within a loop is to use `print()` or `cat()`.

A very useful alternative to `for()` is the `apply()` function. This function does not work in the usual looping syntax way. Rather, it allows one to "apply" a particular function of interest to a row or column of data. Below is an example for the Monte Carlo

simulation:

```
> set.seed(4778)
> y <- matrix(data = rnorm(n = num.samples * sample.size,
     mean = mu, sd = sigma), nrow = num.samples, ncol = sample.size,
     byrow = TRUE)
> head(y, n = 3)
       [,1]   [,2]   [,3]   [,4]    [,5]    [,6]  [,7]    [,8]   [,9]
[1,]  7.260 5.2780  2.128 0.1013  0.9218 -0.1762 3.171  3.6887 2.8389
[2,] -3.545 0.1018 -0.110 4.0861  0.8426  4.8302 3.934  0.8311 0.3605
[3,]  3.588 7.3571  4.986 6.8842 -2.1443  3.5188 1.370 -1.3706 1.7507
      [,10]
[1,]  1.180
[2,]  8.976
[3,] -3.241
> calc.interval <- function(y, alpha, sample.size) {
     mean(y) + qt(p = c(alpha/2, 1 - alpha/2), df = sample.size -
         1) * sd(y)/sqrt(sample.size)
 }
> # apply() will return the results for each data set
> # in a column
> save.results5 <- apply(X = y, MARGIN = 1, FUN = calc.interval,
     alpha = alpha, sample.size = sample.size)
> # Estimated true confidence level
> check <- save.results5[1, ] < mu & save.results5[2,
     ] > mu
> mean(check)
[1] 0.949
```

The `apply()` function could also have been used as follows:

```
> ybar <- rowMeans(y)
> head(ybar)
[1] 2.6391 2.0307 2.2699 2.6753 1.4986 0.8006
> sd.y <- apply(X = y, MARGIN = 1, FUN = sd)
> head(sd.y)
[1] 2.344 3.480 3.682 2.658 2.516 3.331
> lower <- ybar + qt(p = alpha/2, df = sample.size -
     1) * sd.y/sqrt(sample.size)
```

```
> upper <- ybar + qt(p = 1 - alpha/2, df = sample.size -
    1) * sd.y/sqrt(sample.size)
> check <- lower < mu & upper > mu
> # Estimated true confidence level
> mean(check)
[1] 0.949
```

Questions:

- Rather than using `rowMeans()`, how could the `apply()` function be used to calculate the means for each row?
- Why do you think the interval needed to be calculated in two lines of code rather than one line of code like what was done previously?

There are other functions available for loops in R:

- `apply()` has many similar functions to it, including `lapply()` and `tapply()`
- Looping structures similar to `for()` include `while()` and `repeat`