Statistical Software Camp: Introduction to R

Day 4

August 31, 2009

1 Arrays and Lists

1.1 Arrays

- **Arrays** are simply matrices generalized to n dimensions where n is any natural number.
- Arrays can be created either by (1) assigning `dim` attributes to an existing vector/matrix or (2) using the `array()` function.
- An n-dimensional array can be indexed via `[, , ,...]` where there must be n indices separated by commas.

```
> A <- 1:30
> dim(A) <- c(5,3,2) # turns A into a three-dimensional, 5-by-3-by-2 array
> A <- array(1:30, dim=c(5,3,2)) # equivalent
> A

, , 1

     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15

, , 2

     [,1] [,2] [,3]
[1,]   16   21   26
[2,]   17   22   27
[3,]   18   23   28
[4,]   19   24   29
[5,]   20   25   30

> A[,,2] # returns the second chunk of A
```

```
     [,1] [,2] [,3]
[1,]   16   21   26
[2,]   17   22   27
[3,]   18   23   28
[4,]   19   24   29
[5,]   20   25   30
```

```
> A[4,3,1] # returns the (4,3,1) element
```

```
[1] 14
```

- A useful function for arrays (and also for matrices) is `apply(x, M, FUN)`, which invokes another function `FUN` on an array `x` along coordinates specified by `M`.

```
> apply(A, 3, sum) # returns the sum for each chunk (3rd dimension)
```

```
[1] 120 345
```

```
> apply(A, 1, mean) # returns the mean of each row
```

```
[1] 13.5 14.5 15.5 16.5 17.5
```

```
> apply(A, c(2,3), min)
```

```
     [,1] [,2]
[1,]    1   16
[2,]    6   21
[3,]   11   26
```

1.2 Lists

- **Lists** are one-dimensional data objects composed of possibly heterogeneous elements.
- The elements of a list may themselves be complex data objects, such as lists.
- Each element of a list usually has its own name and can be accessed via the `$` operator.
- The `list()` function creates a list, whose elements can be given names with arguments of the following form: `name = element`.

```
> mylist <- list(country = c("Canada","Mexico","USA"), year = 2007,
+              sales = c(3.2, 4.6, 8.3, 9.9, 2.1, 5), arr = A)
> mylist # shows everything in the list
```

```
$country
[1] "Canada" "Mexico" "USA"

$year
[1] 2007

$sales
```

```
[1] 3.2 4.6 8.3 9.9 2.1 5.0

$arr
, , 1

     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15

, , 2

     [,1] [,2] [,3]
[1,]   16   21   26
[2,]   17   22   27
[3,]   18   23   28
[4,]   19   24   29
[5,]   20   25   30

> mylist$year

[1] 2007
```

- Many functions (e.g. `lm()`, `t.test()`) return outputs that have the list format (though their classes are often different, proprietary ones)

```
> x <- c(0,1,3,8)
> y <- c(-2,4,3,5)
> fit <- lm(y ~ x) # The result has a structure similar to a list
> fit$coef

(Intercept)           x
  0.6842105   0.6052632

> fit$resid

         1          2          3          4
-2.6842105  2.7105263  0.5000000 -0.5263158
```

- The `lapply(x, FUN)` function applies another function `FUN` on each element in the list `x`.
- The `sapply(x, FUN)` function does the same, but the results are suppressed in to a vector.

```
> lapply(mylist, mean) # Apply mean() to each element in mylist

$country
[1] NA
```

```
$year
[1] 2007

$sales
[1] 5.516667

$arr
[1] 15.5

> sapply(mylist, mean) # Same output, but suppressed into a vector

    country        year       sales         arr
         NA 2007.000000    5.516667   15.500000
```

2 Functions for Programming

2.1 The `function()` Function

- You can create your own function with the `function()` function.
- To define a function, the following syntax is used:

  ```
  younameit <- function(arg1, arg2){
      body
      ...
  }
  ```

  This creates a function named `younameit` which takes `arg1` and `arg2` as its arguments and execute the commands written between the brackets (`body ...`).
- A function will return the values specified by the `return()` command within the body. If the body contains no `return`, the last evaluated expression will become the output of the function.
- Arguments can be given default values using the `arg = xxx` specification in the definition.
- Example: Create a function which computes the average of numbers in a numeric vector after omitting the highest and lowest values (which would be useful for figure skating judges).

  ```
  > mymean <- function(x){
  +     highest <- max(x)
  +     lowest <- min(x)
  +     (sum(x) - highest - lowest) / (length(x) - 2)
  + }
  > scores <- c(5.8, 5.9, 5.3, 5.6, 5.4, 5.9, 5.4, 5.2, 5.5, 5.9, 5.8, 4.9)
  > mymean(scores)

  [1] 5.58
  ```

- Example Continued: In response to a judging controversy, the International Skating Union decided to use the following (hypothetical) scoring scheme: twelve judges individually award values ranging between 0 and 6, and these values are averaged by randomly selecting nine judges, discarding the highest and lowest values, and averaging the remaining seven.

  Let's create another function for this new scoring system.

```
> newrule <- function(x){
+     x <- sample(x, 9, replace=F)
+     highest <- max(x)
+     lowest <- min(x)
+     (sum(x) - highest - lowest) / 7
+ }
> newrule(scores)

[1] 5.514286
```

- Now, suppose you've found out that the ISU is reconsidering the scoring scheme again. In particular, they want to change the number of values discarded as well as the number of judges sampled. Let's modify our function so it can take various values for these elements. Also, set the default values of these arguments to the ones currently used.

```
> newrule2 <- function(x, nsample = 9, ndiscard = 1){
+     x <- sample(x, nsample, replace=F)
+     x <- sort(x) # sort x in the ascending order
+     sum.used <- sum(x[-c(1:ndiscard, (length(x)-ndiscard+1):length(x))])
+     njudges.used <- length(x) - 2*ndiscard
+     sum.used / njudges.used
+ }
> newrule2(scores)

[1] 5.528571

> newrule2(scores, 7, 2) # sample 7 judges, discard the 2 highest & 2 lowest

[1] 5.533333
```

2.2 && and ||

- && and || are logical operators indicating "and" and "or", respectively, and work in the same way as the & and | operators do, **except**:

  1. & and | operate component-wise on vectors, whereas && and || operate on scalar (i.e. single-valued) logical expressions.
  2. && evaluates the right-hand expression only if the left-hand one is true, and || only if it is false.

- && (||) is convenient when your right-hand expression will lead to an error if the left-hand expression is false (true).

```
> x <- "apple" # x is a character string
> is.numeric(x) & sqrt(x) > 1 # generates an error

> is.numeric(x) && sqrt(x) > 1  # works

[1] FALSE
```

2.3 The while() Loop

- Instead of the for loop function, the while() and repeat() functions can be also used to perform looping.
- The while() function takes the following syntax:

```
while (condition){
    blah
    ...
}
```

where the code blah ... is repeated as long as the condition is met.

- The while() function is especially useful when you do not know ahead of time how many times you will need to repeat the commands.
- Example: Geometric Random Variable. The following code counts how many coin flips are performed until the first "head" occurs.

```
> coin <- "tail" # initialize the coin indicator
> count <- 0 # initialize the counter
> while (coin == "tail"){
+     coin <- sample(c("head","tail"), 1)
+     count <- count + 1
+ }
> count

[1] 1
```

2.4 The repeat Loop

- The repeat function repeats the commands in the body infinitely until exited by the break statement.
- The break statement is often used with conditional statements, such as if.
- The next statement jumps to the next iteration, and also combined with conditional statements within a loop.
- The random variable in the previous example can also be obtained with the repeat() function:

```
> coin <- "tail"
> count <- 0
> repeat{
+     coin <- sample(c("head","tail"), 1)
+     count <- count + 1
+     if (coin == "head") break
+ }
> count

[1] 4
```

2.5 Halting Programs via stop()

- The `stop("blah")` function halts execution of a function or script and spits the error message `blah`.

- This is often used when you want to stop your program if something goes wrong.

- Example: We now modify the `newrule2()` function so that it recognizes a silly calculation.

```
> newrule2(scores, 5, 3) # sample 5 judges, discard...six of them?

[1] 0

> newrule2 <- function(x, nsample = 9, ndiscard = 1){ # Let's modify
+     if (nsample <= 2 * ndiscard) stop("Too few judges sampled!") # added
+     # the rest is the same as before
+     x <- sample(x, nsample, replace=F)
+     x <- sort(x) # sort x in the ascending order
+     sum.used <- sum(x[-c(1:ndiscard, (length(x)-ndiscard+1):length(x))])
+     njudges.used <- length(x) - 2*ndiscard
+     sum.used / njudges.used
+ }
> newrule2(scores, 5, 3) # now spits an error message
```

2.6 Executing Commands from a File

- Once you have written your program, it is often a good idea to save it as an R script file. The standard extention for R scripts is “`.R`”.

- R scripts in the working directory can be executed anytime by typing `source("yourprogram.R")`.

- This is especially convenient when you have a very long program, or when you want to use your function in more than one program.

- Example: Download `newrule2.R` from the camp web, which contains the skating scoring function we have created and additional commands. Try running it via the `source()` command. What happened?

3 Calculating Probability through Simulation

3.1 Functions for Simulation

- The function `sample(X, Y, replace = TRUE, prob = P)` will let you sample `Y` units from a vector `X` with or without replacement (`replace = TRUE` or `replace = FALSE`) using a vector of probability `P` (the default is equal probability).

```
> Z <- 1:10
> ## Randomly draw 8 samples from Z, with replacement
> sample(Z, 8, replace = TRUE)

[1]  2  5  3  7  5 10  2  8

> ## Randomly draw 8 samples from Z, without replacement
> sample(Z, 8, replace = FALSE)

[1]  3  5  7 10  2  1  6  4

> ## Randomly draw 8 samples from Z, with replacement
> sample(Z, 8, replace = TRUE, prob = Z/sum(Z))

[1]  5  7 10 10  8  8  4  4
```

- The function `unique()` will return the unique elements of a vector or a dataframe.

```
> Y <- c(1,0,1,2,0,2,3)
> unique(Y) #display only the unique elements from the vector

[1] 1 0 2 3
```

3.2 Basic Concepts

- Recall that the probability can be thought of as the “limit” of repeated identical experiments.
- Use loop to repeat an experiment and calculate an approximate probability of certain events.

3.3 Example 1: Birthday Problem

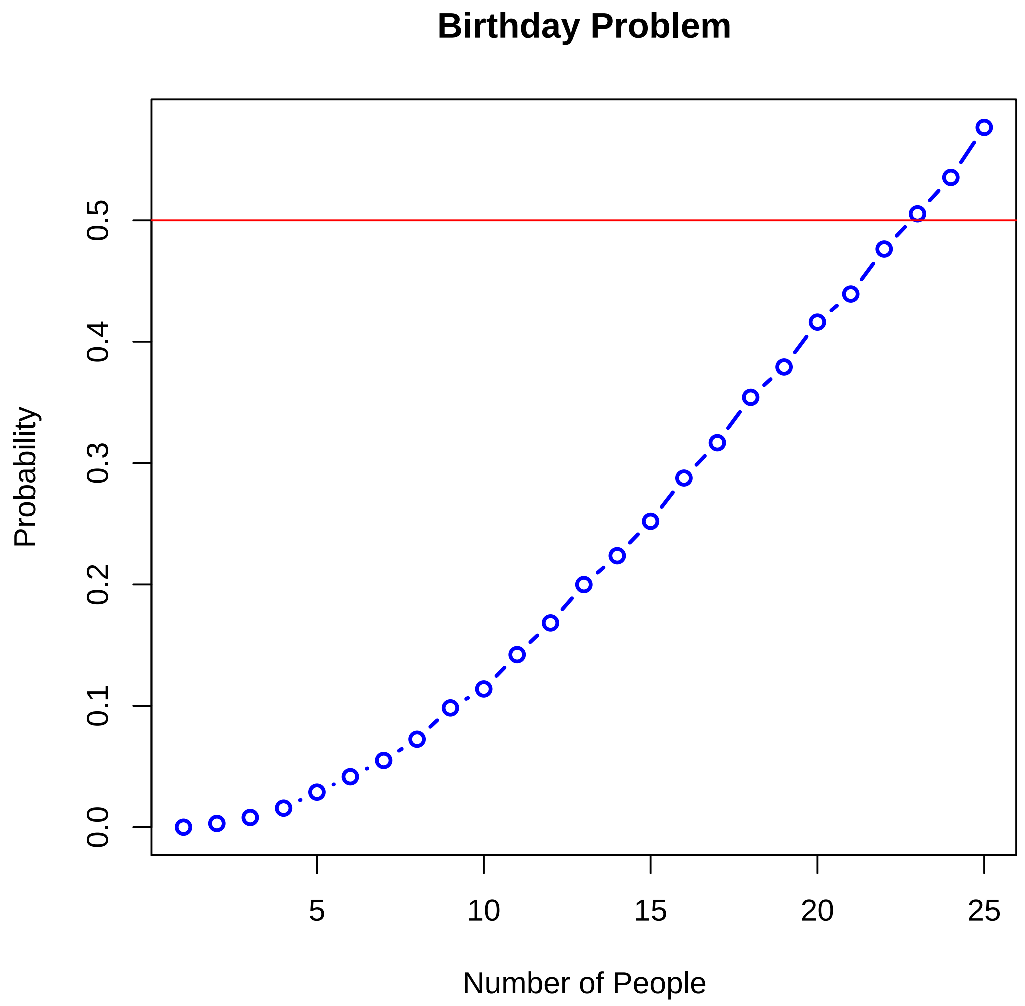
- How many people do you need in order for the probability that at least two people have the same birthday exceeds 0.5?

```
> sims <- 10000 ## number of simulations
> bday <- 1:365 ## possible birthdays
> answer <- rep(NA, 25) ## holder for our answers
> for (k in 1:25) {
+   count <- 0  ## counter
+   for (i in 1:sims) {
+     class <- sample(bday, k, replace = TRUE) # sampling with replacement
+     if (length(unique(class)) < length(class)) {
```

```
+       count <- count + 1
+     }
+   }
+   ## printing the estimate
+   cat("The estimated probability for", k,"people is:", count/sims, "\n")
+   answer[k] <- count/sims # store the answers
+ }

The estimated probability for 1 people is: 0
The estimated probability for 2 people is: 0.003
The estimated probability for 3 people is: 0.0097
The estimated probability for 4 people is: 0.0148
The estimated probability for 5 people is: 0.0302
The estimated probability for 6 people is: 0.0405
The estimated probability for 7 people is: 0.0522
The estimated probability for 8 people is: 0.072
The estimated probability for 9 people is: 0.0954
The estimated probability for 10 people is: 0.116
The estimated probability for 11 people is: 0.1469
The estimated probability for 12 people is: 0.1682
The estimated probability for 13 people is: 0.1944
The estimated probability for 14 people is: 0.2232
The estimated probability for 15 people is: 0.2636
The estimated probability for 16 people is: 0.2888
The estimated probability for 17 people is: 0.3165
The estimated probability for 18 people is: 0.3458
The estimated probability for 19 people is: 0.3808
The estimated probability for 20 people is: 0.4046
The estimated probability for 21 people is: 0.4374
The estimated probability for 22 people is: 0.4818
The estimated probability for 23 people is: 0.506
The estimated probability for 24 people is: 0.5312
The estimated probability for 25 people is: 0.5644

> ## plotting probability that was saved during the loop
> plot(1:25, answer, type = "b", xlab = "Number of People",
+      ylab = "Probability", main = "Birthday Problem", col="blue", lwd = 2)
> abline(h = 0.5, col = "red")
```



3.4 Example 2: Monty Hall Problem

- You must choose one of three doors where one conceals a new car and two conceal old goats. Assume that the probability that each door has a new car is equal to 1/3. Also, assume that your initial choice is random. After you randomly choose one door, the host of the game show, Monty, opens another door which does not conceal a new car. Then, Monty asks you if you would like to switch to the (unopened) third door. Assume that Monty mentally flips a coin to decide which door to open if you initially pick the door with a car. Should you switch?

```
> sims <- 10000
> WinNoSwitch <- 0 # Counter: +1 if you win when not switch
> WinSwitch <- 0 # Counter: +1 if you win when switch
> doors <- 1:3
> for (i in 1:sims){
+   WinDoor <- sample(doors, 1)
+   choice <- sample(doors, 1)
+   if (WinDoor == choice) {
+     WinNoSwitch <- WinNoSwitch + 1
+   }
+   else  WinSwitch <- WinSwitch + 1
+ }
> cat("Prob(Car | no swtich)=", WinNoSwitch/sims, "\n")

Prob(Car | no swtich)= 0.3359

> cat("Prob(Car | swtich)=", WinSwitch/sims, "\n")

Prob(Car | swtich)= 0.6641
```