OEB 153 – Lab 3

11/10/2014

Introduction

In this final lab, we will focus on computer-intensive statistical procedures, namely **resampling** and **bootstrapping**. Rather than walk through each procedure with instructions and examples, we will learn a few additional generally useful functions that happen to be especially useful for computer-intensive procedures. Then you will start working on the lab assignment, applying these new functions and all of the R skills we have learned previously in the course.

A few new functions

These functions round out the parts of R's functionality we have learned so far and will be helpful for computer-intensive statistical procedures.

`sample()`

The `sample()` function is used to take a random sample from a vector. It is also used to produce a shuffled copy of a vector. Looking at `?sample`, the options are as follows:

option	default	description
`x`		Either a vector of one or more elements from which to choose, or a positive integer.
`size`	`length(x)`	a non-negative integer giving the number of items to choose.
`replace`	`FALSE`	Should sampling be with replacement? (logical value)
`probs`	`NULL`	A vector of probability weights for obtaining the elements of the vector being sampled.

Examples

```r
oneToTen = 1:10
# sample 10 entries with replacement from 1:10
sample(oneToTen, size = length(oneToTen), replace = TRUE)

# sample 10 elements without replacement from 1:10
# this produces a *shuffling* of oneToTen
sample(oneToTen, size = length(oneToTen), replace = FALSE)

# sample just two elements, without replacement
sample(oneToTen, size = 2, replace = F)

# sample() can also be used to sample from character vectors, factors, or logical vectors
sample(c("a","b","c"), size = 2, replace = F)
```

Note R has no function dedicated to shuffling. To produce a shuffling of a vector `vec`, sample `vec` without replacement, and make the size of the sample `length(vec)`. Since by default `replace = FALSE` and `size = length(x)`, the default behavior of `sample()` is to produce a shuffling of a vector.

```r
# two equivalent ways to produce a shuffled copy of the vector 1:52
sample(1:52, size = 52, replace = F)
sample(1:52)
```

Caution! Notice the description of the `x` parameter in `sample()`: "Either a vector of one or more elements from which to choose, ***or a positive integer***." If you give the `sample()` function a positive integer as its first (`x`) option, you might expect it to return a shuffling of that integer and always return that integer. Instead, if `x` is a positive integer, R returns a sample of `1:x`. This behavior tends to cause terrible headaches, so be aware!

```r
# this doesn't return 4 every time
# instead, it draws a single element from 1:4
# and is equivalent to sample(1:4, replace = F, size = 1)
sample(4, replace = F, size = 1)
```

`aggregate()`

The `aggregate()` function can be used to obtain a statistic from different subsets of a vector or data frame. There are multiple ways to call `aggregate()`. The options below correspond to the basic options for one way to call the function; see `?aggregate` for other options and other ways to call the function.

option	description
`x`	The vector or data frame containing the data that you want to subset and summarize
`by`	A **list** object containing the factors by which you wish to subset.
`FUN`	The function you want to apply to each subset to obtain the statistic or summary of interest

Example Consider an example with the `iris` dataset, which contains morphometric measurements of fifty plants of each of three species of iris. The dataset is automatically loaded with R.

```r
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

If you wanted to calculate mean sepal width of each species of iris, you could use `aggregate()`. Notice that the argument passed to the `by` option is a list object, here created with the `list()` function.

```r
aggregate(iris$Sepal.Width, by = list(species = iris$Species), mean)
```

```
     species     x
1     setosa 3.428
2 versicolor 2.770
3  virginica 2.974
```

If you wish to subset by more than one variable, you can provide multiple categorical variables (anything with distinct values) in the list passed to the `by` option. For instance, to also subset by whether sepal length is greater than 5 cm, we could use the following command:

```r
aggregate(iris$Sepal.Width,
          by = list(species = iris$Species, sepal.longer.than.5 = iris$Sepal.Length > 5),
          mean)
```

```
     species sepal.longer.than.5     x
1     setosa               FALSE 3.204
2 versicolor               FALSE 2.233
3  virginica               FALSE 2.500
4     setosa                TRUE 3.714
5 versicolor                TRUE 2.804
6  virginica                TRUE 2.984
```

Thus, in this dataset, the mean sepal width of *Iris versicolor* with sepals longer than 5 cm is 2.804 cm.

Often the operations you wish to perform on various subsets are not represented by any of R's built-in functions. If this is the case, you can use `aggregate()` with a custom function. For example, in Lab 1 we could have used the following to calculate the different values of $\hat{p}$, the frequency of the *TLR4BE* genotype:

```r
# import data from Grueber et al. 2013
gru = read.csv("grueber_et_al_2013.csv", header = T, na.strings = "")
# define a custom function to calculate p-hat given some genotypes
get_phat = function(gens){
  gens.narm = na.omit(gens)
  numBE = sum(gens.narm == "BE")
  numGenotypes = length(gens.narm)
  return(numBE/numGenotypes)
}
# have aggregate() calculate the phats for different cohorts.
phats = aggregate(gru$TLR4, by = list(cohort = gru$Cohort), FUN = get_phat)
phats
```

```
   cohort       x
1    2000     NaN
2    2001 0.22222
3    2002 0.22222
4    2003 0.15152
5    2004 0.12500
6    2005 0.10526
7    2006 0.05747
8    2007 0.09559
9    2008 0.06422
10   2009 0.06796
```

`aggregate()` passes each cohort's genotypes to the custom `get_phat` function, which removes the missing values and returns the value of $\hat{p}$. It was necessary to define a custom function because calculating $\hat{p}$ requires four operations: removing missing values, counting *BE* genotypes, and counting the total number of genotypes, and dividing. No single function does all these things, so we defined one that did.

`*apply()` functions

The `*apply()` functions provide another way of performing iterative calculations without employing a for loop. There are many different `*apply()` functions, including `apply()`, `lapply()`, `sapply()`, and `mapply()`. Each one takes a particular type of data container as input (e.g., a vector, list, or data frame), applies a function to each element in that data, and returns the calculations in a particular format. For example, the function `lapply()` can be used to apply some function to each element in a list and return the result as a list. For example:

```r
# a list containing some example data
example.list = list(a = 1:4, b = 10:93, c = rnorm(42))
# compile into a list the mean of each element in the list
lapply(example.list, FUN = mean)
```

```
$a
[1] 2.5

$b
[1] 51.5

$c
[1] 0.05866
```

The function `sapply()` is like `lapply()`, except it returns a vector (with named columns).

```r
# return a vector containing the mean of each element in example.list
sapply(example.list, FUN = mean)
```

```
       a        b        c 
 2.50000 51.50000  0.05866 
```

The function `replicate()` is a "wrapper" to `sapply()` and is a useful function for repeating the same command some number of times. Unlike the `*apply()` functions, `replicate()` takes an *expression* to repeat, instead of a function. The following two commands are completely equivalent:

```r
# sample 1000 times from the sampling distribution of the
# sample mean of 100 standard normal RVs
ybars1 = replicate(1000, mean(rnorm(100, mean = 0, sd = 1)))
# do the same with sapply():
ybars2 = sapply(1:1000, function(x) { mean(rnorm(100, mean = 0, sd = 1)) } )
```

Tip The `replicate()` function may be especially useful for bootstrapping methods. You can have `replicate()` perform multiple commands in each replicate by enclosing them in curly brackets `{}` and separating commands with a semicolon, `;`.

Note You can find a clear summary of the `*apply()` functions here.

For loops vs. `aggregate()` and `*apply()`

In this lab we have seen another approach to carrying out iterative calculations in R. The `aggregate()` function applies a function to different subsets of a vector or data frame, and the `*apply()` functions apply a function to each element in a collection of data. These functions can be used in the place of simple for loops, especially when they are combined with user-defined functions.

Often you will have a computational task that can be tackled with either a for loop or an `*apply()`-style function. Which you choose to use is to some extent a matter of preference. For loops are perhaps better suited for relatively complex tasks and `*apply()` functions for simpler calculations, but of course for loops can be used for simple calculations and `*apply()` functions for more complicated calculations. In most cases, the performance (speed) of the two approaches is the same, since the `*apply()` functions perform for loops under the hood. The two approaches correspond to two different styles of programming: "procedural" (corresponding to for loops) and "functional" (`*apply()`-style functions), in case that is something you have encountered before.

If you like the `*apply()`-style approach to performing iterative functions, check out the packages `plyr` and `reshape`, which attempt to add utility and intuition to the built-in `apply()` family of functions. Your mileage may vary.

Get started on the lab assignment!

It's available on the course website. Work together, ask questions!