Bio Stats II : Lab 4

Acknowledgements: Punt & Branch Labs (U Washington)

Gavin Fay

02/17/2016

Lab schedule

1/27: Introduction to R and R Studio, working with data
2/03: Intro to plotting, manipulating data
2/10: Probability, linear modeling
2/17: Programming practices, conditional statements
2/24: Creating functions, debugging
3/02: Permutation analysis
3/09: Advanced plotting

Recommended reading

An introduction to R (Venables et al.)
– `http://cran.r-project.org/doc/manuals/R-intro.pdf`
- Today's material: Chapter 9

Advance R (Wickham)
- Chapter 5

Packages in R

Installing packages:
`install.packages("packagename")`

If in doubt, use the source:
`install.packages("packagename", type="source")`

Loading (attaching) packages into the workspace:
`library(packagename)`

Some people use `require()` instead of `library()`.
Don't do this!
`require()` is basically `try(library())`

Often description of a package and how to use its functions is in the form of a vignette.
`vignette()` lists available vignettes.
`vignette(packagename)` views the vignette for `packagename`.

Programming practices

Use projects (RStudio)

Write scripts

Whitespace in code
- blank lines, spaces in functions

Use an editor with syntax highlighting

Use a style guide

Indent code

Use meaningful object names

Programming practices

Test code
- Write smallest possible amount (e.g. 1 line).
- Try simple examples that you know the answer to.
- Always assume that there will be an error somewhere.

View results /objects.

Plot results - are they what you expect?

Be careful when copying sections of code and changing a variable name (common to forget to change all occurrences).

- hint: use your text editor's “Find: Replace all” functionality.

Reading in files

Using `head()` is not enough to check data.

First look at the file before reading in the data.

Check the correct number of rows and columns were read in.
- Use `nrow()`, `ncol()`, `dim()`

Make sure `NA`s were read in correctly (using the `na.strings` argument)

Use the `table()` function on columns to detect unexpected values.

Check the summary information: `summary()` and `str()`

Commenting

R ignores everything on a line that follows a #

Comment at the top of your script.
- What the script does, your name, email, date started.

Comment before each function or section of code - What is the purpose of that section of code, what does it do?
- Comment the ‘why’ not the ‘what’

Comment throughout:
- whenever an unusual function is used
- whenever the code is hard to understand
- whenever an algorithm is particularly useful

Commenting out code

When you make modifications to your code:
- Copy the code that works then comment it out by prefixing it with #.
- Change the new copy of the code.

If you need to revert to the old code, just remove the # before each line ("uncomment").

`ctrl+shift+C` is a shortcut in Rstudio to comment/uncomment large blocks of code.

Writing data to files

```
> fakeData <- data.frame(y = rnorm(n = 10),
+                x1 = rpois(n=10, lambda=3),
+                x2 = sample(c("F","M"), size=10,
+                replace=T))
>
> write.table(x=fakeData, file="Data/fakedat.txt",
+                 quote=F, row.names=F)
>
> write.csv(x=fakeData, file="Data/fakedat.csv",
+                row.names=F)
```

Writing unstructured data to files

To create a text file with many different items, use cat() to write lines of text to a file.

```
> cat("#This is fake data \n
+         #This is line 2\n\n",
+         file = "fake_dat5.txt")
> write.table(x=fakeData, file="fake_dat5.txt",
+         quote=F, row.names=F, append=T)
> cat("\n#This is end of the data",
+        file="fake_dat5.txt", append=T)
```

Lab exercise 1/3

Using the PlantGrowth data set (built in):
1. Create a vector InitWeight filled with randomly generated data, with the same length as the number of rows of PlantGrowth.
2. Create a new data frame newdata that combines the data in PlantGrowth with the data in InitWeight.
3. Sort newdata by InitWeight using `order()`.
4. Save the sorted data as a csv file, open in Excel to check.

Grouped expressions

`R` is an expression language in the sense that its only command type is a function or expression which returns a result.

Even an assignment is an expression whose result is the value assigned.

Commands may be grouped together in braces, e.g.
`{expr_1; ...; expr_m}`
The value of the group is the result of the last expression.

N.B. A group is also an expression and may itself be included in parentheses and used as part of an even larger expression, and so on.

Loops and conditionals

Consider the following problem:

We have data on age and length for some animals (“agelength.csv”).
The number of data points for each animal is not the same.

We want to regress length on age for each animal.
We want to plot the slopes of the regressions using a histogram.

Could do this line by line in a script… painful!

Instead devise a part-by-part strategy:
Read in the data
For each animal using a loop
- Check if there are two or more data points
- If yes, regress length on age for that animal
- Store the slope
Plot a histogram of the slopes of each regression

Looping

Vector operations can do many things and are quick.
But we often need to repeat things.

Loops in R are of this form:

```
> for (i in set) {
+    #do something
+ }
```

This is a very flexible structure, try the following:

```
> for (i in seq(from=1,to=5,by=1)) { print(i) }
> for (i in 1:100) { print(i) }
> for (i in c("1","N","L")) { print(i) }
```

Looping through species

Imagine we need to calculate the mean petal and sepal lengths for every species in the iris data set.

```
> for (i in unique(iris$Species)) {
+   meanSepalLength <-
+       mean(data[iris$Species==i,]$Sepal.Length)
+   meanPetalLength <-
+       mean(data[iris$Species==i,]$Petal.Length)
+   cat(i, meanSepalLength, meanPetalLength, "\n")
+ }
```

We already know a better way to do this using `tapply()`, but we can't always do that.

Common way to use loops

```
> #define the elements to loop over
> species <- sort(unique(iris$Species))
> #define how many times to do the loop
> nspecies <- length(species)
> #create a place to store results
> mean.lengths <- vector(length=nspecies)
> #get loopy
> for (i in 1:nspecies) {
+   species.data <- iris[iris$Species==species[i], ]
+   plot(hist(species.data$Sepal.Length))
+   mean.lengths[i] <- mean(species.data$Sepal.Length)
+   print(mean.lengths[i])
+   cat("Running species ", i,"\n")
+ }
```

The `while` loop

The while-loop is rarely used because the for-loop can almost always be used for looping.

Occasionally we don't know how many times to execute the loop.

Can use a `while` loop, which executes a series of statements for as long as some condition remains true.

The general syntax is:

```
> while (condition) {
+   #statements
+ }
```

The `while` loop

If there are 10,000 polar bears in year 2015 and they are declining at 9% per year, in what year will they fall below 500 individuals?

```
> polar.loop <- function(N, year=2015) {
+   while (N>500) {
+     N <- N*0.91
+     year <- year+1
+   }
+   return(year)
+ }
> polar.loop(N=10000)
[1] 2047
```

Pro-tip: when you end up in an infinite loop with no stopping point, press to stop it running!

Notes on `for` loops

`for()` loops are used in `R` code much less often than in compiled languages.

Code that takes a 'whole object' view is likely to be both clearer and faster in `R`.

Other looping facilities include `repeat`, `apply`, `tapply`, `lapply`, `sapply`.

The `break` statement can be used to terminate any loop, possibly abnormally.
This is the only way to terminate repeat loops.

The next statement can be used to discontinue one particular cycle and skip to the "next".

Improving Speed of Loops

Looping over very large data sets can become slow in R.

Overcome by eliminating certain operations in loops or avoiding loops over the data intensive dimension in an object altogether.
- latter can be achieved by performing mainly vector-to-vecor or matrix-to-matrix computations (often > 100x faster than `for()` or `apply()`).

Make use of existing speed-optimized R functions (e.g.: `rowSums`, `rowMeans`, `table`) or design custom functions (next week!).

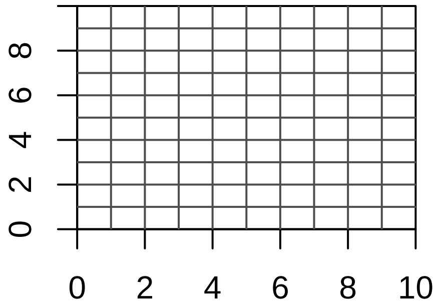
```
> x <- matrix(rnorm(1000000), 100000, 10)
> system.time(x_mean <- apply(x, 1, mean))
   user  system elapsed
  0.785   0.012   0.798
> system.time(x_mean <- rowMeans(x))
   user  system elapsed
  0.004   0.000   0.005
```

Lab exercise 2/3

1. Use `system.time()` to compare the speed for calculating the standard deviation of each row of the matrix `x` (previous slide) using:

- a `for()` loop
- `apply()`
- vector-based calculations (*hint* use `rowSums` and `rowMeans`)

2. Use a `for()` loop to add gridlines to a plot. e.g.



`If()` statements

Often need to check that data are correct before proceeding with an analysis.
e.g. cannot plot `x` and `y` if their lengths are different.

```
> plotxy <- function(x, y) {
+   if (length(x) == length(y)) {
+     plot(x,y)
+   }
+   else {
+     print("Lengths of x and y are different")
+   }
+ }
> plotxy(x=1:5, y=5:1)
> plotxy(x=1:5, y=4:1)
[1] "Lengths of x and y are different"
```

Generic `if()` statement

The generic if-statement is:

```
> if (condition) {
+    #statements
+ }
```

Common conditions in if-statements

```
if (x==5)
if (x==5 & y==7)
if (x>=5)
if (x==5 | y>=17)
if (x %in% c(1,2,3,5))
if (!found)
if (!is.na(x) & y==3)
```

Conditions

Conditions in an if-statement are based on Boolean operators. We have seen these before when subsetting data.

`==`	equals (don't use $=$)
`!=`	not equals
$>$=	greater than or equal to
$<$=	less than or equal to
$>$	greater than
$<$	less than
$\vert$	or (single element)
`&`	and (single element)

Complicated if-then-else statements

```
> sum.bigger <- function(x, y) {
+   if(length(x) != length(y)) {
+     print("not equal lengths")
+   }
+   else {
+     if (sum(x) > sum(y)) {
+       print("x bigger")
+     }
+     else {
+       print("x smaller or equal")
+     }
+   }
+ }
> sum.bigger(x=1:5, y=6:10)
[1] "x smaller or equal"
```

Switch statement

Sometimes you might find yourself writing multiple nested `if` statements.

For example, `if x==1` then do something, `else if x==2` then do something else, `else if x==3` then do a third thing, `else if x==4` then do a fourth thing...

Instead, use a switch statement.

This evaluates the first parameter, and does different things depending on its value.

Switch statements are rarely used but very useful when needed.

e.g. text values

```
> require(stats)
> centre <- function(x, type) {
+   return(switch(type,
+                 mean = mean(x),
+                 median = median(x),
+                 trimmed = mean(x, trim = .1),
+                 "No function matches") )
+ }
> x <- rcauchy(10)
> centre(x, "mean")
[1] -4.018392
> centre(x, "median")
[1] 0.2804401
> centre(x, "unknown")
[1] "No function matches"
```

This is like writing:

```
> centre <- function(x, type) {
+   if (type=="mean") {
+     temp <- mean(x)
+   }
+   else {
+     if (type=="median") {
+       temp <- median(x)
+     }
+     else {
+       if (type=="trimmed") {
+         temp <- mean(x, trim = .1)
+       }
+       else {
+         temp <- "No function matches"
+       }
+     }
+   }
+   return(temp)
+ }
```

Conditions on vectors

The `ifelse()` function can be used to evaluate conditions on all elements of a vector simultaneously.

Generic use:
`ifelse (condition, value when true, value when false)`

```
> (x <- rnorm(10))
 [1] -0.09941829 -1.29216697  0.14142543 -2.26494009 -0.445
 [6] -0.12105214  0.33090710 -1.53453776 -0.46970296 -0.655
> ifelse (x>0, 1, 0)
 [1] 0 0 1 0 0 0 1 0 0 0
```

This can help avoid loops and from declaring variables.

Tips

RStudio:
Code->Reindent Lines (`ctrl+I`) automatically indents your code correctly.

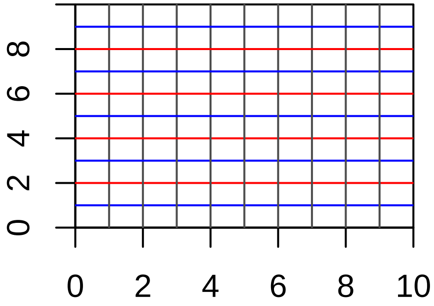
If you have just run a section of code, and then changed part of it, rerun using (`ctrl+shift+P`) with no need to re-select it.

Cheatsheets: `http://bit.ly/UzCoCj` (particularly like the tRips & tRaps one)

One R Tip a Day: `@RLangTip`

Lab Exercise 3/3

1. Use an `if()` statement to change the color of horizontal lines in the plot created for the previous exercise.
   Add blue lines if `x` is odd.
   Add red lines if `x` is even.
   e.g.



Lab Exercise 3/3 (cont'd)

2. Now try to solve the problem mentioned earlier (next slide).

Hints
Before coding, outline the problem conceptually. How would you do it by hand?
I used a for-loop and an if-statement.
To extract the slope from a regression:

```
> my_model <- lm(y~x)
> slope <- coefficients(my_model)[2]
```

Use the `hist()` function for the histogram.

You don't know how many animals there are or whether there are enough data points to do a regression. How will you handle that?

Consider the following problem:

We have data on age and length for some animals
("agelength.csv").
The number of data points for each animal is not the same.

We want to regress length on age for each animal.
We want to plot the slopes of the regressions using a histogram.

Could do this line by line in a script... painful!

Instead devise a part-by-part strategy:
Read in the data
For each animal using a loop
- Check if there are two or more data points
- If yes, regress length on age for that animal
- Store the slope
Plot a histogram of the slopes of each regression