3. Working with moderately large data

Martin Johnsson

IFM Biology

Big Data – data of a size that breaks your usual way of working (and, in all probability, R)

Moderately large data – fit on your laptop, but you need some scripting skills.

R: Data structures

- vectors
- data frames / tibbles
- matrices
- lists

R: Lists

- A vector that can store anything, even another list.
- Useful for making your own complex objects.
- Get elements out with double brackets [[ ]]

```
my_list <- list(1, "ponies", lm(y ~ x, my_data))
my_list[[3]]
list(some_number = 1,
     important_message = "ponies",
     model = lm(y ~ x, my_data))
```

R: Classes

A complex object can have one (or several) classes, which tell you what they are.

There may be special methods (and versions of methods) that operate on a objects of a class.

R: Functions

```
my_function <- function (argument) {
  ## do something with argument
  ## return results
}
```

- Encapsulates a part of the code
- Makes it general and repeatable
- Last expression will be returned
- Custom functions are just like built-in ones

Doing a thing many times

- Functions that operate on vectors

```
sum(my_vector)
```

- Split—apply—combine: Repeatedly apply a function to subsets of a data structure

```
ddply(unicorns, "diet", some_function)
```

- Explicitly telling R to repeat a chunk of code (loops)

Vectorization

Many R functions speak vector natively

You may not appreciate this, because it's so natural for the R user, but this is not something you get in every language

```
standard_error <- function(x) {
  sd(x) / sqrt(length(x))
}
```

Repeated application

`library(plyr)`

Function names tell you what they operate on

ddply – data frame -> data frame

llply – list (or vector) -> list

ldply – list (or vector) -> data frame

dlply – data frame -> list

dlply

```
library(plyr)
library(reshape2)

melted <- melt(unicorn_data,
  id.vars = c("id", "diet", "colour"))

dlply(melted, "variable", function(data) {
  lm(value ~ diet, data)
})
```

ddply

```
ddply(melted, "variable", function(data) {
  coef(lm(value ~ diet, data))
})
```

ddply

Good for calculating summary statistics:

```
unicorn_stats <- ddply(melted, "variable", summarise,
   mean = mean(value, na.rm = TRUE),
   stdev = sd(value, na.rm = TRUE))
```

The while loop

Keeps repeating until a given condition is fulfilled

```
a <- 0
while (a < 10) {
 a <- a + 1
}
```

The for loop

Repeats over a given sequence and keeps track of an index variable

```
a <- 0
for (i in 1:10) {
 a <- a + 1
}
```

The for loop

This is why I like plyr – code skeleton for doing what dlply does:

```
output <- vector(length = 10, mode = "list")
groups <- c("a", "b", "c")
for (i in 1:length(groups)) {
 data_in_group <- subset(data, group == groups[i])
  ## do something
 output[[i]] <- data_in_group
}
## deal with the list
```

Simulating data

Why?

- Test performance of methods (statistics)
- Evaluate statistical graphics (your intuition)
- Sanity-check your code (your programming)

Probability distributions in R

- Normal/Gaussian

`rnorm, pnorm, dnorm`

- Uniform

`runif, punif, dunif`

- Binomial and Bernoulli

`rbinom, pbinom, dbinom`

- Poisson etc etc etc

```
coin_tosses <- rbinom(100, 1, 0.5)
```

Draw 100 samples from Binom(n = 1, p = 0.5)

```
normal_draws <- rnorm(10, 0, 5)
```

Draw 10 samples from N(mean = 0, sd = 5).

```
y <- 1.1 + 0.5 * x + rnorm(100, 0, 1)
```

Simulate 100 samples from a regression model

$y = 1.1 + 0.5 * x + e, e \sim N(0, 1)$.

x is an indicator for some variable.

```
sample(c(1, 5, 10, 6), size = 3)
```

Draw from the given vector.

```
sample(c(1, 5, 10, 6),
       size = 100,
       replace = TRUE)
```

Draw with replacement.

Replicate

- When you have an expression and want to evaluate it multiple times
- Mostly useful for simulation

```
replicate(100, runif(100, min = 0, max = 1))
```

Fake data simulation

1. Create a simulated dataset, similar to the one you're interested in, using the assumptions you will make in the analysis.
2. Implement your analysis.
3. Run the analysis over and over again on many simulated datasets.
4. Evaluate performance (e.g. variability, false positives, power, exaggeration factor etc)

Review of concepts

- *False positive rate*: If you simulate data with no effects, how often does the analysis suggest an effect?

Review of concepts

- *Power*: If you simulate data with a true effect of a certain size, how often does the analysis detect it?

Review of concepts

When the analysis finds a simulated effect ...

- how often is the estimate the wrong sign? (*Sign error*)
- how many times bigger than the true effect is the estimate? (*Exaggeration factor*)

(Gelman & Carlin 2014)

Exercise 3

Command lines and working on a cluster

Homework 3: Design analysis by simulation