Data Mining with R

R programming constructs

Hugh Murrell

Reference Books

These slides were created to accompany chapters three to five of the text:

- *Scientific Programming and Simulation using R*
  Owen Jones, Robert Maillardet, and Andrew Robinson.

A version of this text can be found on the web.

Programming with R

In this lecture we cover programming constructs and the creation of functions, the rules that they must follow, and how they relate to and communicate with the environments from which they are called.

We also present some tips on the construction of efficient functions, with especial reference to how functions are treated in R.

Programming with R

In the first part of this lecture we discuss programming constructs available in the R package.

Branching

When an `if` expression is evaluated, a logical expression is tested and if TRUE then the first group of expressions is executed and the second group of expressions is not executed. Conversely if FALSE then only the second group of expressions is executed. `if` statements can be nested to create elaborate pathways through a program.

Braces are used to group together one or more expressions. If there is only one expression then the braces are optional.

```
if (logical_expression) {
    expression_1
...} else {
    expression_2
    ...}
```

computing zeros

```
> # find the zeros of a2*x^2 + a1*x + a0 = 0
> a2 <- 1; a1 <- 4; a0 <- -5
> discrim <- a1^2 - 4*a2*a0
> # calculate the roots depending on the value of the
> if (discrim > 0) {
+   roots <- c( (-a1 + sqrt(a1^2 - 4*a2*a0))/(2*a2),
+               (-a1 - sqrt(a1^2 - 4*a2*a0))/(2*a2) )
+ } else {
+     if (discrim == 0) {
+         roots <- -a1/(2*a2)
+     } else {
+         roots <- c() } }
> # output
> roots
[1]  1 -5
```

Looping

The `for` command has the following form, where `x` is a simple variable and `v` is a vector.

```
for (x in v) {
    expression in x
... }
```

Summing a vector

```
> (x_list <- seq(1, 9, by = 2))
[1] 1 3 5 7 9
> sum_x <- 0
> for (x in x_list) {
+     sum_x <- sum_x + x
+ }
> cat("The total is", sum_x, "\n")
The total is 25
>
```

Looping with while

Often we do not know beforehand how many times we need to go around a loop. That is, each time we go around the loop, we check some condition to see if we are done yet. In this situation we use a while loop, which has the form:

```
while (logical_expression) {
  expression_1
... }
```

First Fibonacci greater than 100

```
> # initialise variables
> F <- c(1, 1)
> n <- 2
> # iteratively calculate new Fibonacci numbers
> while (F[n] <= 100) {
+   n <- n + 1
+   F[n] <- F[n-1] + F[n-2]
+ }
> # output
> cat("First Fibonacci number > 100 is F(",
+     n, ") =", F[n], "\n")
First Fibonacci number > 100 is F( 12 ) = 144
>
```

Vector-based programming

It is often necessary to perform an operation upon each of the elements of a vector.
Using vector operations is more efficient and concise than looping.
For example, we could find the sum of the first n squares using a loop as follows:

```
> n <- 100
> S <- 0
> for (i in 1:n) { S <- S+i^2 }
> S
```

```
[1] 338350
```

Using vector operations we have:

```
> sum((1:n)^2)
```

```
[1] 338350
```

Vector ifelse

The `ifelse(test,A,B)` function performs elementwise conditional evaluation upon a vector.

The function returns a vector that is a combination of the evaluated expressions A and B: the elements of A that correspond to the elements of test that are TRUE, and the elements of B that correspond to the elements of test that are FALSE.

As before, if the vectors have differing lengths then R will repeat the shorter vector to match the longer.

```
> x <- c(-2, -1, 1, 2)
> ifelse(x > 0, "Positive", "Negative")
```

```
[1] "Negative" "Negative" "Positive" "Positive"
```

Functional Programming with R

Functions are one of the main building blocks for large programs: they are an essential tool for structuring complex algorithms.

Functions

A function has the form:

```
name <- function(argument1, argument2, ...) {
    expression1
    expression2
    ...
    return(output)
}
```

Here `argument1`, `argument2`, etc., are the names of variables and `expression1`, `expression2`, and `output` are all R expressions.

`name` is the name of the function. Note that some functions have no arguments, and that the braces are only necessary if the function comprises more than one expression.

Calling your function

To call or run the function we type:

```
name(x1, x2, ...)
```

The value of this expression is the value of the expression `output`. To calculate the value of output the function first copies the value of `x1` to `argument1`, `x2` to `argument2`, and so on.

The arguments then act as variables within the function. We say that the arguments have been passed to the function.

Function evaluation

The function evaluates the grouped expressions contained in the braces.

the value of the expression `output` is returned as the value of the function.

A function may have more than one return statement, in which case it stops after executing the first one it reaches.

If there is no return statement then the value returned by the function is the value of the last expression in the braces as long as it is not assigned to a variable.

A function always returns a value. For some functions the value returned is unimportant. In such cases one usually returns NULL.

Example

The number of ways that you can choose r things from a set of n, ignoring the order in which you choose them, is n choose r, which is defined as:

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

We break the problem into two parts. First we need a function for computing factorials:

```
> n_factorial <- function(n) {
+   # Calculate n factorial
+   n_fact <- prod(1:n)
+   return(n_fact)
+ }
```

Example continued....

Now we can make use of our factorial function to construct an n choose r function:

```
> n_choose_r <- function(n, r) {
+   # Calculate n choose r
+   n_ch_r <- n_factorial(n)/n_factorial(r)/n_factoria
+   return(n_ch_r)
+ }
```

Lets try it out:

```
> n_choose_r(6,2)
```

```
[1] 15
```

Note that more efficient functions that achieve the same goal are available in R; specifically, `choose` and `factorial`.

Program flow when using functions

When a function is executed the computer,

- sets aside space for the function variables,
- makes a copy of the function code,
- transfers control to the function.

When the function is finished

- the output of the function is passed back to the main program,
- and the copy of the function and all its variables are deleted.

Scope

Arguments and variables defined within a function exist only within that function.

If you define and use a variable `x` inside a function, it does not exist outside the function.

If variables with the same name exist inside and outside a function, then they are separate and do not interact at all.

You can think of a function as a separate environment that communicates with the outside world only through the values of its arguments and its output expression.

Scope continued...

That part of a program in which a variable is defined is called its **scope**.

Beware, the scope of a variable is not symmetric.

Variables defined inside a function cannot be seen from outside, but variables defined outside the function can be seen from inside the function (provided there is not a variable with the same name defined inside).

Thus it is advisable to ensure that the variables you use in a function are either arguments, or have been defined inside the function. That way you can be assured that your function will return unique output values when called with the same input paramters.

Optional arguments and default values

To give the `argument1` the default value `x1` we use `argument1 = x1` within the function definition.

If an argument has a default value then it may be omitted when calling the function, in which case the default is used.

```
> test <- function(x = 1, y = 1, z = 1) {
+   return(x*100+y*10+z)
+ }
> test()
[1] 111
> test(2, 2)
[1] 221
> test(y = 2, z = 2)
[1] 122
```

Vector-based functional programming

To further facilitate vector-based programming, R provides a family of powerful functions that enable the vectorisation of user-defined functions: `apply`, `sapply`, `lapply`, `tapply`, and `mapply`.

The effect of `sapply(X, FUN)` is to apply function `FUN` to every element of vector `X`. That is, `sapply(X, FUN)` returns a vector whose i-th element is the value of the expression `FUN(X[i])`.

If `FUN` has arguments other than `X[i]`, then they can be included using `sapply(X, FUN, ...)`, which returns `FUN(X[i], ...)` as the i-th element.

Example: prime numbers

```
> prime <- function(n) {
+ # returns TRUE if n is prime integer
+ if (n == 1) {
+         is.prime <- FALSE
+     } else if (n == 2) {
+         is.prime <- TRUE
+ } else {
+ is.prime <- TRUE
+         for (m in 2:(n/2)) {
+             if (n %% m == 0) is.prime <- FALSE
+         }
+ }
+     return(is.prime)
+ }
```

Density of prime numbers

Let $\rho(n)$ be the number of primes less than or equal to n. Both Legendre and Gauss famously asserted that

$$\lim_{n \to \infty} \frac{\rho(n) log(n)}{n} = 1$$

The proof is hard, but we can check the result numerically:

```
> n.vec <- 2:1000
> primes <- sapply(n.vec, prime)
> num.primes <- cumsum(primes)
> plot(n.vec, (num.primes * log(n.vec))/n.vec,
+      type = "l", main = "density",
+      xlab = "n", ylab = "")
```

Prime density plot

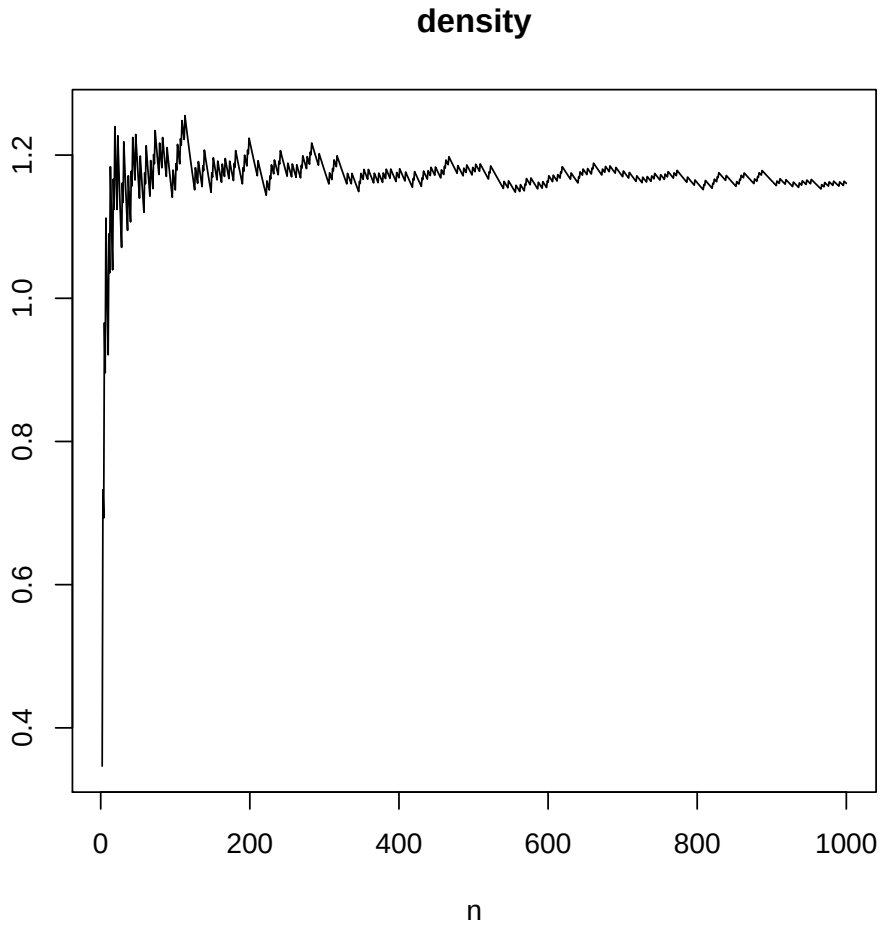

Figure: Prime number density.

Recursive programming

Recursive programming is a powerful programming technique, made possible by functions. A recursive function is simply one that calls itself.

For example, we can write $n!$ as $n(n-1)!$. We implement this recursive definition below.

```
> nfactorial <- function(n) {
+   # calculate n factorial
+   if (n == 1) {
+     return(1)
+   } else {
+     return(n*nfactorial(n-1))
+   }
+ }
> nfactorial(6)
[1] 720
```

Example: Sieve of Eratosthenes

The Sieve of Eratosthenes is an algorithm for finding all of the primes less than or equal to a given number n. It works as follows:

1. Start with the list $2, 3, \ldots, n$ and the largest known prime $p = 2$.
2. Remove from the list all elements that are multiples of p (but keep p itself).
3. Increase p to the smallest element of the remaining list that is larger than the current p.
4. If p is larger than $\sqrt{n}$ then stop, otherwise go back to step 2.

recursive sieve implementation

```
> primesieve <- function(sieved, unsieved) {
+ # finds primes using the Sieve of Eratosthenes
+ # sieved: sorted vector of sieved numbers
+ # unsieved: sorted vector of unsieved numbers
+   p <- unsieved[1]
+   n <- unsieved[length(unsieved)]
+   if (p^2 > n) {
+     return(c(sieved, unsieved))
+   } else {
+     unsieved <- unsieved[unsieved %% p != 0]
+     sieved <- c(sieved, p)
+     return(primesieve(sieved, unsieved))
+   }
+ }
```

primeseive in action

```
> pms <- primesieve(c(), 2:200)
> options(width=45)
> show(pms)
```

```
 [1]   2   3   5   7  11  13  17  19  23  29
[11]  31  37  41  43  47  53  59  61  67  71
[21]  73  79  83  89  97 101 103 107 109 113
[31] 127 131 137 139 149 151 157 163 167 173
[41] 179 181 191 193 197 199
```

example

The game of CRAPS is played as follows:

First, you roll two six-sided dice;

let x be the sum of the dice on the first roll. If $x = 7$ or 11 you win, otherwise you keep rolling until either you get x again, in which case you also win, or until you get a 7 or 11, in which case you lose.

We will construct a FUNCTIONAL program to simulate a game of CRAPS.

We will use our simulation to estimate the probability of winning a game of CRAPS.

We will confirm your probability of winning estimation via a theoretical argument.

exercise

From first principles, write a function called, myFit, that uses least squares to fit a straight line to a set of data points. Vectorize your code as much as possible.

Your function should take x and y vectors (of the same length) and should return the intercept, a, and slope, b, of the straight line, $y = a + b\,x$, that best fits the data.

Test your function by adapting last weeks code to use your function as well as R's built-in function to fit straight lines. Make sure that your function and R's built-in function return the same results.

If you don,t know how to compute a least squares fit from first principles then read:
http://mathworld.wolfram.com/LeastSquaresFitting.html.

exercise

e-mail your solution to me as a single R script by:

6:00 am, Monday the 29th February.

Make sure that the first line of your R script is a comment statement containing your name, student number and the week number for the exercise. (in this case week 02).

Use: `dm02-STUDENTNUMBER.R` as the filename for the script.

Use: `dm02-STUDENTNUMBER` as the subject line of your email.

There will be no extensions. No submission implies no mark.