Prácticas de Aprendizaje Automático

Grado en Informática
Doble Grado Informática y Matemáticas

Información

- **PRÁCTICAS DE APRENDIZAJE AUTOMÁTICO**

- **Horario:** **G1** X**(**17,30 – 19,30) aula 3.3
  G2 V**(**17,30 – 19,30) aula 3.3
  G3 V**(**17,30 – 19,30) aula 2.1

- Página web: decsai.ugr.es
- Parte del contenido aquí presentado está inspirado en el material elaborado por

  Roger D. Peng, Associate Professor of Biostatistics
  Johns Hopkins Bloomberg School of Public Health

Control Structures

Control structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are :

if, else: testing a condition
for: execute a loop a fixed number of times
while: execute a loop while a condition is true
repeat: execute an infinite loop
break: break the execution of a loop
next: skip an interation of a loop
return: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresions;
for command-line interactive work, the ***apply** functions are more useful.

Control structures: if

```
if(<condition>) {
        ## do something
} else {
    ## do something else
}
if(<condition1>) {
        ## do something
} else if(<condition2>) {
                ## do something different
              } else {
                  ## do something different
}
```

If

Else is not always necessary. This is a valid if/else structure.

```
if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

So is this one:

```
y <- if(x > 3) {
                10
     } else {
                0
     }
```

for

for loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
for(i in 1:10) {
        print(i)
}
```

This loop takes the i variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.

for

These three loops have the same behavior.

```
x <- c("a", "b", "c", "d")

for(i in 1:4) {
        print(x[i])
}

for(i in seq_along(x)) {
        print(x[i])
}

for(letter in x) {
        print(letter)
}

for(i in 1:4) print(x[i])
```

Nested for

for loops can be nested.

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
       for(j in seq_len(ncol(x))) {
              print(x[i, j])
       }
}
```

Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand.

While

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

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

While loops can potentially result in infinite loops if not written properly. Sounds familiar? :P

While

Sometimes there will be more than one condition in the test.

```
z <- 5
while(z >= 3 && z <= 10) {
        print(z)
        coin <- rbinom(1, 1, 0.5)
        if(coin == 1) { ## random
                walk z <- z + 1
} else {
        z <- z - 1
    }
}
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

repeat

Repeat initiates an infinite loop; these are not commonly used in learning or statistical applications but ... there it is. The only way to exit a **repeat** loop is to call **break**. A bit dangerous because there's no guarantee it will stop!!!

```
x0 <- 1
tol <- 1e-8
repeat {
        x1 <- computeEstimate()
        if(abs(x1 - x0) < tol) {
                break
    } else {
            x0 <- x1
    }
}
```

Next, return

next is used to skip an iteration of a loop

```
for(i in 1:100) {

        if(i <= 20) { ## Skip the first 20 iterations
                next
        } ## Do something here
}
```

return signals that a function should exit and return a given value

Functions

Functions are created using the function() directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {
              ## Do something interesting
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

- Functions have named arguments which potentially have default values.
- The formal arguments are the arguments included in the function definition
- The formals function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be missing or might have default values

Argument Matching

Functions arguments can be matched positionally or by name. So the following calls to sd are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

Argument Matching

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.
The following two calls are equivalent.

```
> args(lm)
function (formula, data, subset, weights, na.action,
              method = "qr", model = TRUE, x = FALSE,
              y = FALSE, qr = TRUE, singular.ok = TRUE,
              contrasts = NULL, offset, ...)
```

The following two calls are equivalent.

```
> lm(data = mydata, y ~ x, model = FALSE, 1:100)
> lm(y ~ x, mydata, 1:100, model = FALSE)
```

Argument matching

•Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
•Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

•Function arguments can also be partially matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

Defining a Function

In addition to not specifying a default value, you can also set an argument value to NULL.

```
> f <- function(a, b = 1, c = 2, d = NULL) {
  print(b)
}
```

Lazy Evaluation

Arguments to functions are evaluated lazily, so they are evaluated only as needed.

```
> f <- function(a, b) {
        a^2
   }
> f(2)
```

This function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

Lazy Evaluation

```
> f <- function(a, b) {
              print(a)
              print(b)
}
> f(45)
```

```
## [1] 45
## Error: argument "b" is missing, with no default
```

Notice that “45” got printed first before the error was triggered.
This is because b did not have to be evaluated until after print(a). Once the function tried to evaluate print(b) it had to throw an error.

The "..." Argument

- The ... argument indicate a variable number of arguments that are usually passed on to other functions.
- ... is often used when extending another function and you don't want to copy the entire argument list of the original function

```
> myplot <- function(x, y, type = "l", ...) {
                                    plot(x, y, type = type,
...)
}
```

- Generic functions use ... so that extra arguments can be passed to methods (more on this later).

The "..." Argument

•The ... argument is also necessary when the number of arguments passed to the function cannot be known in advance

```
> args(paste) function (..., sep = " ", collapse = NULL)

> args(cat) function (..., file = "", sep = " ",
                fill = FALSE, labels = NULL, append = FALSE)
```

Arguments Coming After "..."

•One catch with ... is that any arguments that appear after ... on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste) function (..., sep = " ", collapse = NULL)

> paste("a", "b", sep = ":") [1] "a:b"

> paste("a", "b", se = ":") [1] "a b :"
```

Scoping Rules

Binding Values to Symbol

How does R know which value to assign to which symbol? When I type

```
> lm <- function(x) { x * x }

> lm
# function(x) { x * x }
```

how does R know what value to assign to the symbol **lm**?
Why doesn't it give it the value of lm that is in the stats package?

Binding Values to Symbol

When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly
The search list can be found by using the **search** function:

1. Search the global environment for a symbol name matching the one requested.
2. Search the namespaces of each of the packages on the search list.

```
> search()
[1] ".GlobalEnv" "package:stats" "package:graphics"
[4] "package:grDevices" "package:utils" package:datasets"
[7] "package:methods" "Autoloads" "package:base"
```

Binding Values to Symbol

- The global environment or the user's workspace is always the first element of the search list and the base package is always the last.
- The order of the packages on the search list matters!
- User's can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available.
- When a user loads a package with **library** the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions so it's possible to have an object named *c* and a function named *c.*

Scoping rules

•The scoping rules determine how a value is associated with a free variable in a function
•R uses lexical scoping or static scoping.
•A common alternative is dynamic scoping.
•Related to the scoping rules is how R uses the search list to bind a value to a symbol Lexical scoping turns out to be particularly useful for simplifying statistical computations

```
> f <- function(x, y) {
                      x^2 + y / z
}
```

This function has 2 formal arguments x and y. In the body of the function there is another symbol z. In this case z is called a free variable. The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned insided the function body).

Lexical Scoping

the values of free variables are searched for in the environment in which the function was defined.

What is an environment?

- An environment is a collection of (symbol, value) pairs,
- i.e. x is a symbol and 3.14 might be its value.
- Every environment has a parent environment; it is possible for an environment to have multiple “children”
- the only environment without a parent is the empty environment
- A function + an environment = a closure or function closure.

Lexical Scoping

Searching for the value for a free variable:

- If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment.
- The search continues down the sequence of parent environments until we hit the top-level environment; this usually the global environment (workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the empty environment.
- If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

Lexical Scoping

Why does all this matter?

•Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace
•This behavior is logical for most people and is usually the "right thing" to do

However, in R you can have functions defined inside other functions
Languages like C don't let you do this

Now things get interesting

• In this case the environment in which a function is defined is the body of another function!

Lexical Scoping

This function returns another function as its value.

```
> make.power <- function(n) {
              pow <- function(x) {
                           x^n
              }
              pow
}
> cube <- make.power(3)
> square <- make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Exploring a Function Closure

What's in a function's environment?

```
> ls(environment(cube))
[1] "n" "pow"
> get("n", environment(cube))
[1] 3
> ls(environment(square))
[1] "n" "pow"
> get("n", environment(square))
[1] 2
```

Lexical vs. Dynamic Scoping

•What is the value of f(3)

```
y <- 10
f <- function(x) {
               y <- 2 y^2 + g(x)
        }
g <- function(x) {
               x*y
        }
```

With lexical scoping the value of y in the function g is looked up in the environment in which the function was defined, in this case the global environment, so the value of y is 10.

With dynamic scoping, the value of y is looked up in the environment from which the function was called (sometimes referred to as the calling environment).

So the value of y would be 2.

Lexical vs. Dynamic Scoping

•When a function is defined in the global environment and is subsequently called from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping.

```
> g <- function(x) {
        a <- 3 + x+a+y
}
> g(2) Error in g(2) : object "y" not found
> y <- 3
> g(2)
[1] 8
```

Other Languages

Other languages that support lexical scoping
- Scheme
- Perl
- Python
- Common Lisp (all languages converge to Lisp)

Consequences of Lexical Scoping

- In R, all objects must be stored in memory
- All functions must carry a pointer to their respective defining environments, which could be anywhere
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the “defining environment” of all functions is the same.

Lexical Scoping Summary

- Objective functions can be “built” which contain all of the necessary data for evaluating the function
- No need to carry around long argument lists
- — useful for interactive and exploratory work.
- Code can be simplified and cleand up

Simulation

Generating Random Numbers

Functions for probability distributions
•rnorm: generate random Normal variates with a given mean and standard deviation
•dnorm: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
•pnorm: evaluate the cumulative distribution function for a Normal distribution
•rpois: generate random Poisson variates with a given rate

Probability distribution functions usually have four functions associated with them. The functions are prefixed with

•d for density
•r for random number generation
•p for cumulative distribution
•q for quantile function

Generating Random Numbers

Working with the Normal distributions requires using these four functions

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

If ϕ is the cumulative distribution function for a standard Normal distribution, then pnorm(q) = ϕ (q) and qnorm(p) = ϕ^{-1}(q).

Generating Random Numbers

```
> x <- rnorm(10)
> x
 [1] 1.38380206 0.48772671 0.53403109 0.66721944
 [5] 0.01585029 0.37945986 1.31096736 0.55330472
 [9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
 [1] 23.38812 20.16846 21.87999 20.73813 19.59020
 [6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.32   19.73   20.55   20.67   21.67   23.39
```

Generating Random Numbers

Setting the random number seed with set.seed ensures reproducibility

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
```

- Always set the random number seed when conducting a simulation!

Generating Random Numbers

Generating Poisson data

```
> rpois(10, 1)
 [1] 3 1 0 1 0 0 1 0 1 1
> rpois(10, 2)
 [1] 6 2 2 1 3 2 2 1 1 2
> rpois(10, 20)
 [1] 20 11 21 20 20 21 17 15 24 20

> ppois(2, 2)   ## Cumulative distribution
[1] 0.6766764   ## Pr(x <= 2)
> ppois(4, 2)
[1] 0.947347    ## Pr(x <= 4)
> ppois(6, 2)
[1] 0.9954662   ## Pr(x <= 6)
```

Generating Random Numbers From a Linear Model

Suppose we want to simulate from the following linear model

$y = \beta_0 + \beta_1 x + \varepsilon$

$\varepsilon \sim N(0, 2)$ $x \sim N(0, 1)$ $\beta_0 = 0.5$ $\beta_1 = 2$

```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
```

Generating Random Numbers From a Linear Model

Suppose we want to simulate from a Poisson model
Where Y ~ Poisson(μ) log μ = β0 + xβ1 and β0 = 0.5, β1=0.3 and .
We need to use the **rpois** function for this

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.9410
```

Generating Random Numbers From a Linear Model

The sample function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
> sample(1:10)  ## permutation
 [1] 4 7 10 6 9 2 8 3 1 5
> sample(1:10)
 [1]  2  3  4  1  9  5 10  8  6  7
> sample(1:10, replace = TRUE)  ## Sample w/replacement
 [1] 2 9 7 8 2 8 5 9 7 8
```

Simulation Summary

- Drawing samples from specific probability distributions can be done with r* functions
- Standard distributions are built in: Normal, Poisson, Binomial, Exponential, Gamma, etc.
- The sample function can be used to draw random samples from arbitrary vectors

- Setting the random number generator seed via set.seed is critical for reproducibility