Prácticas de Aprendizaje Automático

Parte 2: Programando en R

Curso 2017-18

1. Para programar en R
2. Condicionales
3. Bucles
4. Funciones
5. Código vectorizado

Índice

1. Para programar en R

2. Condicionales

3. Bucles

4. Funciones

5. Código vectorizado

Para programar en R

- R es un lenguaje interactivo que nos permite crear objetos y analizarlos. Pero además...
- permite ir creando nuevas estructuras que resuelven nuevos problemas que van apareciendo.
- R es un lenguaje de expresiones: todos los comandos ejecutados son funciones o expresiones que producen un resultado.
- Podemos crear explícitamente un objeto tipo expresión, con la función **expression()** y evaluarla con la función **eval()**
- Una subrutina es un segmento de código que se escribe sólo una vez pero puede invocarse o ejecutarse muchas veces
- Cuando agrupamos comandos o expresiones entre llaves `{expre.1; expre.2; ...; expre.m}` las expresiones pueden ir: separadas con `;` o en líneas separadas
- El valor de la expresión es el último valor

Expresiones y funciones

La mayor utilidad de las expresiones es que nos permiten ejecutar varios comandos de una sola vez.
Pero donde gana mayor utilidad esta forma de trabajar es a la hora de crear nuevos objetos que ejecutan diversas expresiones utilizando como entrada unos objetos (los argumentos) y devolviendo otros objetos.
Estos objetos son del tipo `function` y constituyen las nuevas funciones de R definidas por el usuario

```
exp1<-expression(3+4); exp2<-expression(sum(1:10))
eval(exp1) ; eval(exp1) ;
[1] 7
[1] 55
```

a diferencia de

```
{exp1<-expression(3+4); exp2<-expression(sum(1:10))
eval(exp1) ; eval(exp1) ;}
[1] 55
```

Modificando el control de flujo

Para cambiar el flujo de control en la ejecución de instrucciones, existen las tipícas estructuras conocidas de Condicionales, Bucles y Funciones.

```
if(cond) expr

if(cond)
   cons.expr
else
   alt.expr

for(var in seq)
    expr

while(cond)
    expr

repeat
    expr

break
next
```

Índice

1. Para programar en R
2. Condicionales
3. Bucles
4. Funciones
5. Código vectorizado

Condicionales

```
if (expre1) expre2 else expres3

if (sample(1:6,1) >3) cat("SI >3 \n") else cat("NO <=
```

En la primera expresión `expre1` podemos incluir varios condiciones utilizando los operadores lógicos `&`, `|`, `!` y además :

Para una secuencia de argumentos lógicos,

all :devuelve el valor lógico que indica si todos los elementos son TRUE.

any :devuelve el valor lógico que indica si algún elemento es TRUE

```
> n <-c(NA,2:5)
> all (n>1)   # NA
> any (n>=5) # TRUE
> x<-0
> if  (is.numeric(x) & min(x)>0)
   raizx <- sqrt(x) else stop("x debe ser numerico y p
> Error: x debe ser numerico y positivo
```

Condicionales

R dispone de una versión *vectorizada* de if que es **ifelse()**

```
ifelse( vec.test, vec.si.true, vec.si.false)
```

Devuelve un vector cuya longitud es la del más largo de sus argumentos y cuyo elemento i es vec.si.true[i]s si vec.test[i] es cierta, y vec.si.false[i] en caso contrario

```
y<- -5:5
y.logy<-ifelse(y>0,log(y),0)
round(y.logy,3)
```

Índice

1. Para programar en R
2. Condicionales
3. Bucles
4. Funciones
5. Código vectorizado

Bucles

Ordenes para la ejecución en bucles `for`, `while`, `repeat`.

```
for (name  in values) expre
```

Uso de `{ }` si más de una instrucción.

```
for(i in 1:10)
     cat("caso ",i,"\n");
```

```
while (condi) expr
```

```
i=0
while(i<10)  {
     cat("caso ",i,"\n");
     i=i+1
  }
```

Bucles

La instrucción `repeat` es sin condicion :((
La única forma de salir de un ciclo es con: para salir hay que usar `break`

```
repeat {nlot<-sample(1:10,1,rep=T);
        if(nlot==5) break()
        else cat("No es 5 es",nlot,"\n")}
```

break termina cualquier ciclo **for, while, repeat**
next dentro de **for, while, repeat** fuerza el comienzo de una nueva iteración.

```
for(i in 1:100) {
if(i <= 20) { ## se salta las 20 primeras
    next
  } ## otra cosa
}
```

R y los bucles ... :(son muy lentos,
cuando se pueda...mejor usar `lapply, sapply, tapply`.

Índice

1. Para programar en R
2. Condicionales
3. Bucles
4. Funciones
5. Código vectorizado

Funciones

Es posible ver el código de una función usando su nombre sin `()`

Una función se define con una asignación de la forma:

```
f <- function(<arguments>) {
## hacer cosas
}
```

- Las funciones pueden pasar argumentos a otras funciones
- Las funciones pueden estar anidadas, esto es, se pueden definir funciones dentro de otras
- El valor devuelto por una función es la última expresión del cuerpo a ser evaluada
- Las funciones pueden devolver valores simples, vectores, gráficas...

Funciones

Una función que suma dos números:

```
> sumanumeros<-function(x,y)
  (x+y)
> sumanumeros(3,5)

> sumanumeros(c(2,3),c(3,5))  # Tambien pued sumas vec
> sumanumeros(c(2,3),c(3,5,1)) # Resulta un warning!!
          # como arreglarlo?
```

Realizar una función que devuelva el vector menor entendido como...

Ejemplos de funciones

Dados los vectores originales *xi*, *yi*

```
xi = -10 :10
yi = c(-15:-5, 10:19)

menor <-function(x,y){
    y.min <- y < x
    x[y.min] <- y[y.min]
    x
}
> menor (xi,yi)
 [1] -15 -14 -13 -12 -11 -10  -9  -8  -7  -6  -5   1

> menor<-function(x,y){  # mas compacto
ifelse(x<y, x, y)
}
```

Argumentos y valores por defecto

- Los argumentos han de darse en el orden en el que se han definido en la función.
- Si los argumentos se dan por nombre, `nombre.arg=objeto`, el orden es irrelevante.
- Además el nombre puede reducirse siempre que no sea ambiguo.
- `args(nombre.funcion)}` muestra los argumentos de cualquier función.

```
  # todas estas llamadas son equivalentes
menor(1:5,  c(1,6,2,7,3))
menor(x=1:5,y=c(1,6,2,7,3))
menor(y=c(1,6,2,7,3), x=1:5)
```

Argumentos y valores por defecto

- La correspondencia entre parámetros formales y parámetros actuales no es estricta
- En muchos casos hay funciones que tienen argumentos que deseamos tengan un valor por defecto.
- Con ello se consigue que: si éste se omite en la llamada, tomará el valor de la definición.
- La forma definir valores por defecto es incluir en la definición de la función: `nombre.argumento=valor.por.defecto`

```
grande <- function(x,y=0*x){
    # con y=0  seria numero, no funcionaria
    ifelse(x>y, x, y)
}
grande(c(-12:3))
grande(c(1,2),2:3)
```

Con número variable de parámetros

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

Otros argumento

- En muchos casos nos interesa utilizar argumentos de otras funciones.
- El argumento los (3 puntos) nos permite incluir en la definición de nuestras funciones la posibilidad de llamar otra función con los parámetros especificados sin tener que volver a copiar la lista de argumentos original.

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

O bien cuando el número de argumentos no es conocido con antelación

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

Funciones de control y parada I

`warning, missing, stop`

- Hasta ahora no se ha comprobado si los argumentos son los apropiados, lo que podría habernos llevado a errores lógicos o de ejecución.
- R nos permite utilizar funciones para controlar y parar el la ejecución de una función.
- Si hacemos comprobaciones y detectamos un error leve, podemos usar `warning("mensaje")`, muestra el mensaje y la ejecución de la función **continúa**.
- Para evitar un error grave se usa función `stop("mensaje")`, muestra el mensaje y **deja de evaluar** la función.
- La función `missing(argumento)` indica de forma lógica si un argumento no ha sido especificado, equivalente a warning.

Funciones de control y parada II

```
grande<-function(x,y=0*x){
if  (missing(y))
     warning("Estamos comparando con 0")
y.g <- y>x
x[y.g] <- y[y.g]
x
}
grande(-3:3)
```

Ambito de las variables

- Las variables que se usan en el cuerpo de una función pueden dividirse en:
  - Parámetros formales, los identificadores de los argumentos de la función
  - Variables locales
  - Variables libres, las que no son ninguna de las anteriores. Su valor se busca de forma secuencial en los entornos que han llamado a la función
- El ámbito son reglas para encontrar alguna de las variables libres.

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

Índice

1. Para programar en R
2. Condicionales
3. Bucles
4. Funciones
5. Código vectorizado

Código vectorizado

Se puede escribir código de muy diferentes formas, pero un código R es más eficiente cuando se le saca partido a los siguientes 3 conceptos: **tests lógicos, extracción de subconjuntos y ejecución a nivel de elementos**. Como muestra un botón.
Dado un vector con números positivos queremos obtener un vector con los valores absolutos del vector original.

```
abs_loop <- function(vec){
      for (i in 1:length(vec)) {
        if (vec[i] < 0) {
            vec[i] <- -vec[i]
        }
   }
vec
}
```

Código vectorizado

Dado un vector con números positivos queremos obtener un vector con los valores absolutos del vector original. Vamos a utilizar tests combinado con extracción de subconjuntos para la sustitución

```
abs_sets <- function(vec){
   negs <- vec < 0
   vec[negs] <- vec[negs] * -1
   vec
}
```

`abs_sets()` es más rápido ?!

```
long <- rep(c(-1, 1), 5000000) # con 5millones
> system.time(abs_loop(long))
   user  system elapsed
 13.907   0.000  13.665
> system.time(abs_sets(long))    # casi 30x
   user  system elapsed
  0.546   0.000   0.507
```

Como escribir código vectorizado

Pero se puede aún mejorar

```
> system.time(abs(long))
   user  system elapsed
  0.040   0.032   0.073
```

Nos hemos de plantear hacer la modificación de un vector según una condición y dejarlo intacto en caso contrario. De forma vectorizada Ejemplo: si un número es positivo dejarlo tal cual. En caso contrario si es negativo la función le tiene que dar la vuelta (multiplicar por -1)

```
vec <- c(1, -2, 3, -4, 5, -6, 7, -8, 9, -10)
vec < 0                              # se considera lo que camb
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FA
vec[vec < 0]
[1]  -2  -4  -6  -8 -10
vec[vec < 0] * -1
 [1]  2  4  6  8 10
vec[vec < 0] <- vec[vec < 0] * -1
```

Como escribir código vectorizado

Supongamos que tenemos que recodificar un vector de cadenas con un código tal como: `for + if else if else ...`

```
change_forif <- function(vec){
   for (i in 1:length(vec)){
      if (vec[i] == "DD") {
         vec[i] <- "joker"
      } else if (vec[i] == "C") {
         vec[i] <- "ace"
      } else if (vec[i] == "7") {
         vec[i] <- "king"
      }else if (vec[i] == "B") {
         vec[i] <- "queen"
      #  ...

      } else {
         vec[i] <- "nine"
      }
   }
vec }
```

Como escribir código vectorizado

Supongamos que tenemos que recodificar un vector de cadenas con un código tal como: `for + if else if else ...`

```
change_vec <- function (vec) {
    vec[vec == "DD"] <- "joker"
    vec[vec == "C"] <- "ace"
    vec[vec == "7"] <- "king"
    vec[vec == "B"] <- "queen"
    vec[vec == "BB"] <- "jack"
    vec[vec == "BBB"] <- "ten"
    vec[vec == "0"] <- "nine"
vec
}
vec <- c("DD", "C", "7", "B", "BB", "BBB", "0") # comp
```

Cómo escribir código vectorizado

La combinación `if + for`
Debemos de poder identificar dónde eliminar bucles siempre que tengamos combinaciones de verb2 if + for 2. el `if` se puede aplicar uno a uno a cada componente de un vector,
lo que implica un es como usar un `if` dentro de un `for`.
El bucle sirve para aplicar el `if` todos los elementos del vector.
Por tanto cambiar esta combinación por un test dentro `[ ]`.

funciones iteradoras los *apply()

Simulando `for` .
Escribir un `for` mientras trabajamos de forma interactiva no es muy cómodo hay formas más sencillas. Mediante el uso de `apply`. Existen unas funciones auxiliares que pueden resultar de utilidad al iterar la aplicación de una función *fun* **a cada componente**.

- **lapply**: Reitera sobre una lista y evalúa una función sobre cada componente (resultado una lista)
- **sapply**: Idem que lapply pero simplifica resultado (suele devolver un vector)
- **apply**: Aplica una función sobre cada uno de los márgenes de un array (bien por filas, bien por columnas)
- **tapply**: Aplica una función sobre subconjuntos de un vector
- **mapply**: Versión multivariante version de lapply

- **replicate** : Repite una expresión un cierto número de veces el resultado lo compone en un vector o matriz

lapply

Devuelve siempre una lista

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3
$b
[1] 0.1302209
```

En este caso trabaja con diversos vectores

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1),
> lapply(x, mean)
$a
[1] 2.5
$b
[1] -0.1029113
$c
[1] 0.9305208
$d
[1] 5.060848
```

lapply

Se aplica una función con varios argumentos

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 6.48527
[[2]]
[1] 4.651752 1.146238
[[3]]
[1] 5.531737 9.440400 2.646411
[[4]]
[1] 6.842872 8.659604 7.699846 1.832540
```

lapply

Utilizando una función anónima

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6,3, 2
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4
$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> lapply(x, function(elt) elt[,1]) # 1a col
$a
[1] 1 2
$b
[1] 1 2 3
```

sapply

De forma similar a `lapply` pero no tiene porque devolver una lista sino que:

- Cuando el resultado es una lista en el que todos los componentes son de longitud 1, se devuelve un vector.
- Cuando el resultado es una lista en el que todos los componentes son de la misma longitud (>1), se devuelve una matriz.
- En cualquiero otra circunstancia se devuelve una lista

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6,3, 2
> sapply(x, mean)
  a   b
2.5 3.5
```

apply

`apply` permite aplicar una función por filas o por columnas de una matriz

```
> m <- matrix(1:16, 4)
> m
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> apply(m,1, max)   # por filas
[1] 13 14 15 16

> apply(m,2, max) # por columnas
[1]  4  8 12 16
```

apply

Supongamos la siguiente matriz, obtener calificaciones finales candidatas!

```
> notas
    Pr1 Pr2 Pr3  Pr4
Al1 7.3 7.5 7.4  7.4
Al2 9.5 9.4 1.2  9.5
Al3 6.6 6.7 6.3 10.0

> notas
    Pr1 Pr2 Pr3  Pr4 medias mediana
Al1 7.3 7.5 7.4  7.4    7.4    7.40
Al2 9.5 9.4 1.2  9.5    7.4    9.45
Al3 6.6 6.7 6.3 10.0    7.4    6.65
```

```r
## Para ordenar LAS COLUMNAS  de la matriz
     apply(mm, 2, sort)

     ## keeping named dimnames
     names(dimnames(x)) <- c("row", "col")
     x3 <- array(x, dim = c(dim(x),3),
                 dimnames = c(dimnames(x), list(C = pa
     identical(x,  apply( x,  2,  identity))
     identical(x3, apply(x3, 2:3, identity))

     ##- UTILIZANDO FUNCIONES CON ARGUMENTOS ADICIONAL
     cave <- function(x, c1, c2) c(mean(x[c1]), mean(x
     apply(x, 1, cave,  c1 = "x1", c2 = c("x1","x2"))
```

apply

```
> d  =  matrix(c(rnorm(99),NA), ncol= 2)
> medias.fila =  apply(d,1, mean)
    # con mas argumentos
> medias.na.col = apply(d,2, mean, na.rm = T)

f1 = function(x, na.cero = F){
      if (na.cero == T)
          ifelse(is.na(x),0,x+1)
      else x+1
}
apply(d,2,f1)
apply(d,2,f1, na.cero = T)
```

apply

`apply` existen una serie de funciones que se pueden suplir con `apply`

```
rowSums = apply(x, 1, sum)

rowMeans = apply(x, 1, mean)

colSums = apply(x, 2, sum)

colMeans = apply(x, 2, mean)
```

replicate

replicate : Repite una expresión (función) un número indicado de veces, el resultado lo compone en forma de vector o matriz.

Ej. Repitir 10 veces el lanzamiento de 2 dados:

```
replicate(10,sample(6,2,replace=T))
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    2    3    3    3    3    3    1    6    5
[2,]    4    1    6    4    5    4    6    3    3
```

sin utilizar un for!!!