Fundamentos de la programación estadística y Data Mining en R

Unidad 1. Estructuras de control y funciones en R

Dr. Germán Rosati (Digital House - UNTREF - UNSAM)

03 septiembre, 2017

Estructuras de control

- Hasta aquí hemos trabajado con la sesión interactiva. Pero en general suele ser más efectivo trabajar con scripts que realicen tareas más complejas que las que podemos hacer de la línea de comando.
- Lo que permiten las llamadas estructuras de control es poder generar programas que realicen estas tareas.

Estructuras de control

- Las estructuras de control permiten manejar el flujo de ejecución de un programa. Algunas estructuras comunes son:
  - `if`, `else`: testea una condición
  - `for`: loop que ejecuta un bloque de código un número determinado de veces
  - `while`: loop que ejecuta un bloque de código un número indeterminado de veces, mientras se cumpla una condición determinada
  - `repeat`: ejecuta un loop infinitas veces
  - `break`: sale de la ejecución de un loop
  - `next`: saltea una iteración de un loop
  - `return`: devuelve un valor dentro de una función

Estructuras de control: `if`

- Un `if` testea una condición y si la misma es verdadera, ejecuta un deteminado bloque de código. Un ejemplo de una estructura `if` válida:

```
> x <- 5
> if (x > 3) {
+     y <- 10
+ } else {
+     y <- 0
+ }
> x
## [1] 5
> y
## [1] 10
```

Estructuras de control: `for` loops

- Un `for` loop toma un "iterador" y le asigna sucesivos valores de una secuencia de elementos o de un vector. Este tipo de loop se usa habitualmente para iterar o recorrer los elementos de un objeto (una lista, un vector, etc.)

```
> for (i in 1:3) {
+     print(i)
+ }
## [1] 1
## [1] 2
## [1] 3
```

- Este loop toma el iterador `i` y le asigna en cada iteración un valor en la secuencia 1 a 3. Luego, imprime el valor de `i` y al llegar al final de la secuencia sale del loop.

Estructuras de control: `for` loops

- Los siguientes loops tienen el mismo funcionamiento y devuelven el mismo resultado:

```
> x <- c("a", "b", "c", "d")
> for (i in 1:4) {
+     print(x[i])
+ }
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

Estructuras de control: `for` loops

- Las siguientes tres estructuras de `for` loops tienen el mismo funcionamiento y devuelven el mismo resultado:

```
> x <- c("a", "b", "c", "d")
> for (i in seq(1, length(x))) {
+     print(x[i])
+ }
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

Estructuras de control: `for` loops

```
> for (i in seq_along(x)) {
+     print(x[i])
+ }
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
> for (letra in x) {
+     print(letra)
+ }
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

Estructuras de control: `for` loops

- Se pueden anidar muchos `for` loops:

```
> x <- matrix(1:6, 2, 3)  #¿Qué genera esta línea?
> for (i in 1:nrow(x)) {
+     for (j in 1:ncol(x)) {
+         print(x[i, j])
+     }
+ }
## [1] 1
## [1] 3
## [1] 5
## [1] 2
## [1] 4
## [1] 6
```

Estructuras de control: `for` loops

- En general, hay que tener cuidado cuando se anidan for loops. Anidar más de 2-3 niveles puede hacer el código muy lento y, además, difícil de entender.

Estructuras de control: `while` loops

- Los`while` loops empiezan testeando una condición. Si la condición es verdadera ejecutan el bloque de código debajo del `while`. Una vez que se ejecuta el bloque, se vuelve a testar la condición,
- Cuando la condición se hace falsa sale del loop.

```
> count <- 0
> while (count < 4) {
+     print(count)
+     count <- count + 1
+ }
## [1] 0
## [1] 1
## [1] 2
## [1] 3
```

- El problema es que pueden generarse loops infitos. . .

Estructuras de control: `while` loops

- Puede haber más de una condición a testear en el `while`

```
> z <- 5
> while (z > 3 & z < 9) {
+     print(z)
+     coin <- rbinom(1, 1, 0.5)
+     if (coin == 1) {
+         z <- z + 1
+     } else {
+         z <- z - 1
+     }
+ }
```

Estructuras de control: `next`, `break`

- `next` se usa para saltear una iteración en un loop

```
> for (i in 1:6) {
+     if (i <= 2) {
+         next
+     }
+     print(i)
+ }
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

- `break` se usa para salir de un loop (aún cuando una condición no se cumpla).

Funciones especiales: la familia `apply()`

- En muchos casos, suele ser útil usar algunas de estas funciones para correr loops sin tener que escribir un `for` o un `while()` de forma explícita. Para eso sirven las funciones de la familia `apply()`.
- Vamos a ver una de estas funciones: `apply()`
  - Suele usarse para aplicar una función sobre las filas o columnas de una matriz
  - Puede usarse con arrays de cualquier dimensión
  - En términos de performance no es mejor que un `for` (no se ejecuta más rápido). Pero sí es más fácil de escribir.

Funciones especiales: la familia `apply()`

```
> str(apply)
## function (X, MARGIN, FUN, ...)
```

- `x` es un array (o matriz)
- `MARGIN` es un vector de integer indicando sobre qué márgenes (por ejemplo, filas o columnas) debe aplicarse la función
- `FUNCTION` es la función a aplicar

Funciones especiales: la familia apply()

```
> options(digits = 4)
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)  # media sobre columnas
```

[1] 0.14385 -0.06155 0.20824 0.23152 0.08758 0.23508 0.06746 [8] -0.16847 -0.14272 0.18278

```
> apply(x, 1, sum)  # suma sobre filas
```

[1] 1.418624 0.643216 1.719437 6.158898 1.620038 2.641641 -0.994253 [8] 3.122591 -2.523928 -0.331086 1.577729 -2.578700 0.002771 -2.555354 [15] 2.291779 -0.057220 -3.543811 0.819134 4.459887 1.784195

Funciones especiales: la familia `apply()`

- Para sumas y medias sobre filas y columnas existen algunas funciones mucho más optimizadas que los `apply()`
  - `colSums` == `apply(x,2,sum)`
  - `rowSums` == `apply(x,1,sum)`
  - `colMeans` == `apply(x,2,mean)`
  - `rowMeans` == `apply(x,1,mean)`

Funciones especiales: la familia `apply()`

- Otra forma de usar `apply()`: calcular cuartiles sobre las columnas de una matriz.

```
> options(digits = 4)
> x <- matrix(rnorm(200), 40, 5)
> apply(x, 2, quantile, probs = c(0.25, 0.5, 0.75))
##         [,1]      [,2]    [,3]    [,4]    [,5]
## 25% -0.2576 -0.556479 -0.8805 -0.7309 -0.8263
## 50%  0.2918  0.005353 -0.3627 -0.1851 -0.1794
## 75%  0.6326  0.755991  0.3624  0.4792  0.4979
```

Funciones especiales: `lapply()`

- Otra función de la familia es `lapply()`. Se usa para recorrer una lista y evaluar una función en cada uno de sus elementos.

```
> str(lapply)
## function (X, FUN, ...)
```

- `x` es una lista a recorrer
- `FUN` una función a evaluar
- `...` otros argumentos de `FUN`
- Siempre devuelve una lista independientemente de la clase del input.

Funciones especiales: `lapply()`

```
> x <- list(a = 1:5, b = rnorm(10), c = rnorm(50, 1), d = rnor
> lapply(x, mean)
## $a
## [1] 3
##
## $b
## [1] -0.3261
##
## $c
## [1] 0.902
##
## $d
## [1] 5.195
```

Funciones especiales: `lapply()`

- `lapply()` y sus derivados (`sapply()`,`tapply()`) pueden usar "funciones anónimas": funciones no se definen por fuera.

```
> 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
```

Funciones especiales: `lapply()`

- Una función anónima para extraer la primera columna de cada matriz en la lista

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

Distribuciones de probabilidad básicas

- Veamos algunas operaciones asociadas a distribuciones de probabilidad. Vamos a ver algunas dado que hay una gran cantidad de distribuciones en R. > Para ver qué distribuciones hay disponibles se puede usar el siguiente comando (`help.search("distribution")`)
- Daremos algunos comandos básicos asociados a la distribución normal y binomial pero el resto de las distribuciones tienen comando similares.
- Para cada distribución hay cuatro comandos asociados a cuatro funciones diferentes:
  - `d`: devuelve la altura de la función de densidad
  - `p`: devuelve la función de probabilidad acumulada
  - `q`: devuelve la inversa función de probabilidad -quantiles-
  - `r`: devuelve números aleatorios generados de una la distribución

Distribuciones de probabilidad: Normal

- Asume valores entre $-\infty$ y $+\infty$
- Su función de densisdad es
  - $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x_i-\mu)^2}{2\sigma^2}}$
- Tiene dos parámetros: μ -la media- y σ -el desvío estándar-
- Cuando una variable aleatoria se distribuye normalmente se escribe $X \sim N(\mu, \sigma)$

Distribuciones de probabilidad: Normal

- `dnorm()`: dado un set de valores devuelve la función de densidad
  - `x`: el valor a evaluar
  - `mean`: la media de la distribución
  - `sd`: el desvío std. de la distribución
  - `log.p`: si `FALSE` las probabilidades se devuelven como $log(p)$

```
> dnorm(0)
## [1] 0.3989
> dnorm(0, mean = 23, sd = 10)
## [1] 0.002833
> vals <- c(0, 1, 2)
> dnorm(vals, mean = 23, sd = 10)
## [1] 0.002833 0.003547 0.004398
```

Distribuciones de probabilidad: Normal

- `pnorm()`: dado un set de valores devuelve la probabilidad acumulada, es decir, la probabilidad de obtener ese valor o menor
  - `q`: el valor a evaluar
  - `mean`: la media de la distribución
  - `sd`: el desvío std. de la distribución
  - `log.p`: si `FALSE` las probabilidades se devuelven como $log(p)$

```
> pnorm(0)
## [1] 0.5
> pnorm(1)
## [1] 0.8413
> vals <- c(-2, -1, 0, 1, 2)
> pnorm(vals)
## [1] 0.02275 0.15866 0.50000 0.84134 0.97725
```

Distribuciones de probabilidad: Normal

- `qnorm()`: Es la inversa de `pnorm()`. La idea es darle una probabilidad y que devuelva el valor que acumula hasta allí.
  - `p`: el valor a evaluar
  - `mean`: la media de la distribución
  - `sd`: el desvío std. de la distribución
  - `log.p`: si `FALSE` las probabilidades se devuelven como $log(p)$

```
> qnorm(0.025)
## [1] -1.96
> qnorm(0.5)
## [1] 0
> qnorm(0.975)
## [1] 1.96
```

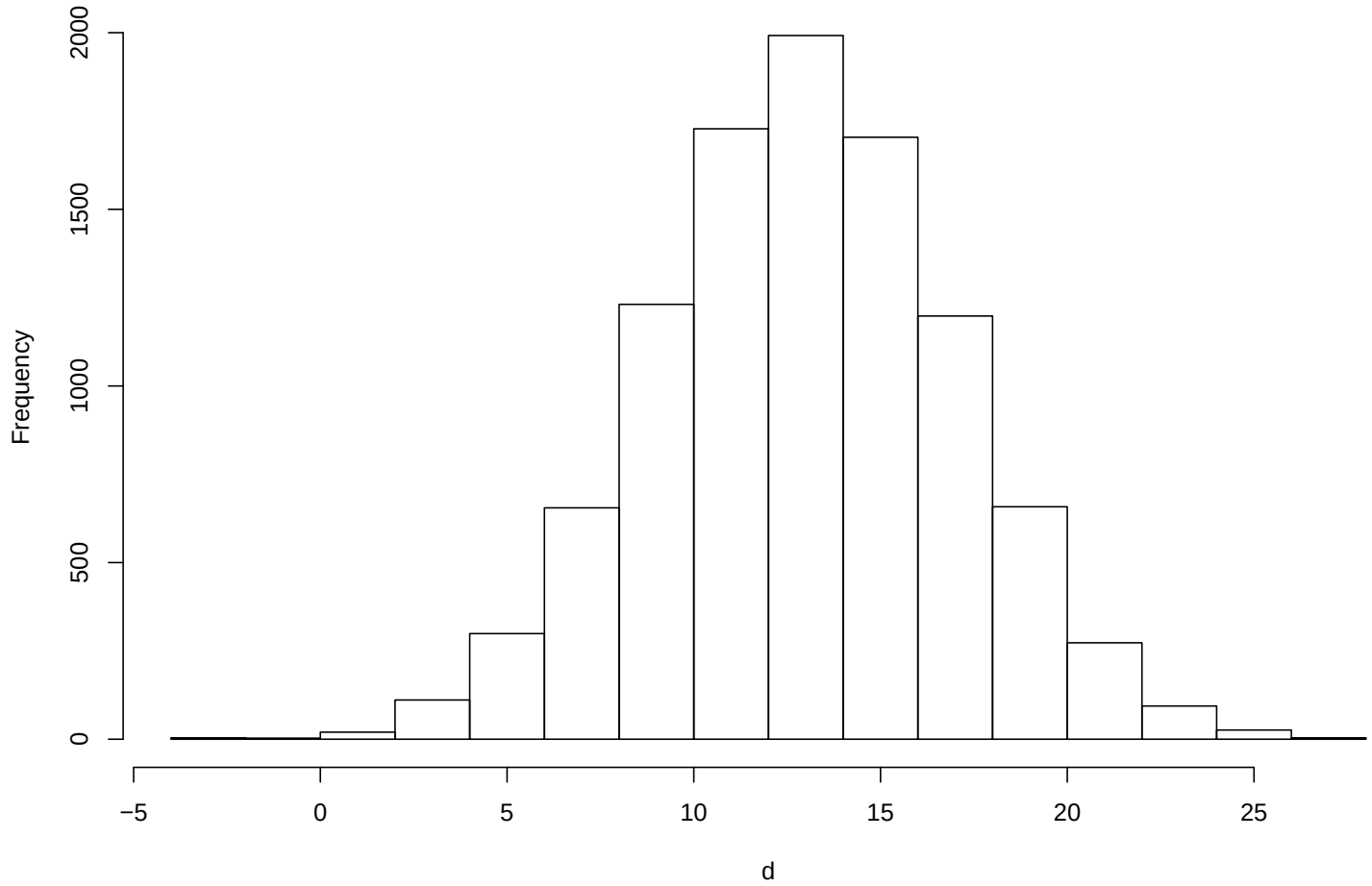
Distribuciones de probabilidad: Normal

- `rnorm()`: arroja números aleatorios de una distribución normal con los parámetros especificados.
  - `n`: cantidad de números aleatorios a devolver
  - `mean`: la media de la distribución
  - `sd`: el desvío std. de la distribución
  - `log.p`: si `FALSE` las probabilidades se devuelven como $log(p)$

```
> rnorm(10, 0, 1)
##  [1] -0.2525  1.4075  1.5446  0.3744 -0.6010  1.0666 -1.054
##  [9]  0.2195  0.4807
> d <- rnorm(10000, 13, 4)
> mean(d)
## [1] 13.02
> sd(d)
## [1] 3.984
```

Distribuciones de probabilidad: Normal

```
> d <- rnorm(10000, 13, 4)
> hist(d, main = "")
```

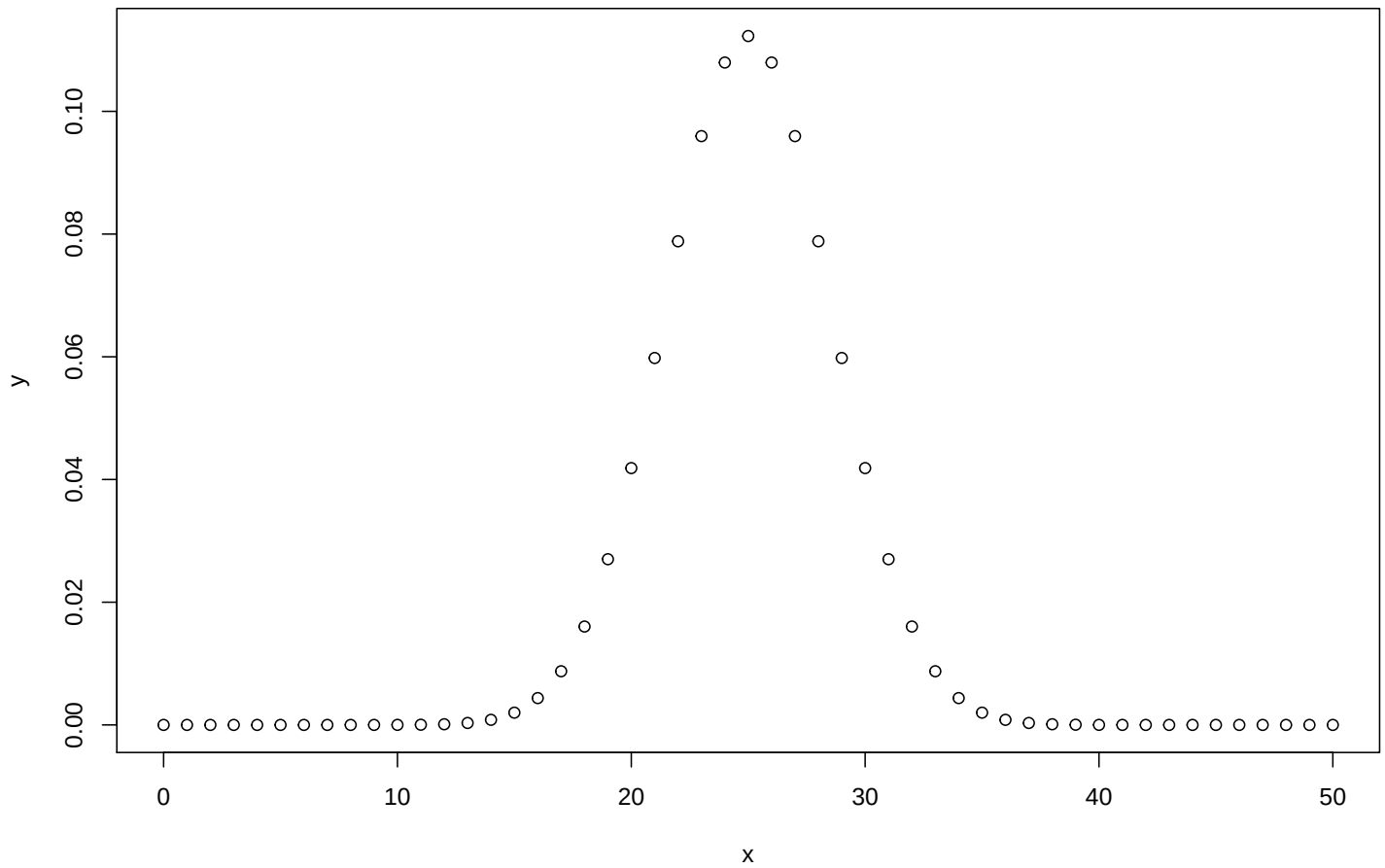


Distribuciones de probabilidad: Binomial

- Imaginemos un experimento (de Bernoulli) con dos resultados posibles (1=éxito y 0=fracaso) y la probabilidad de éxito es $P(X = 1) = p$ y la de fracaso es $P(X = 0) = 1 - p$
- Repetimos el experimento n veces y lo que queremos modelar es la cantidad de éxitos (x) en las n tiradas.
- La función de densidad de una distribución binomial es:
  - $f(x) = \binom{n}{x} p^x (1-p)^{(n-x)}$ donde $x = 0, 1, 2, ..., n$
- `dbinom()`: dado un set de valores devuelve la función de densidad
  - `x`: el valor a evaluar
  - `size`: cantidad de pruebas, o sea, el n
  - `prob`: probabilidad de éxito, o sea, p
  - `log.p`: si `FALSE` las probabilidades se devuelven como $log(p)$

Distribuciones de probabilidad: Binomial

```
> x <- seq(0, 50, by = 1)
> y <- dbinom(x, size = 50, prob = 0.5)
> plot(x, y)
```



Distribuciones de probabilidad: Binomial

- `pbinom()`: dado un set de valores devuelve la probabilidad acumulada, es decir, la probabilidad de obtener ese valor o menor
  - `q`: el valor a evaluar
  - `size`: cantidad de pruebas, o sea, el n
  - `prob`: probabilidad de éxito, o sea, p
  - `log.p`: si `FALSE` las probabilidades se devuelven como $log(p)$

```
> pbinom(25, size = 50, prob = 0.5)
## [1] 0.5561
> pbinom(25, size = 50, prob = 0.25)
## [1] 1
> pbinom(25, size = 50, prob = 0.1)
## [1] 1
> pbinom(25, size = 50, prob = 0.05)
## [1] 1
```

Distribuciones de probabilidad: Binomial

- `qbinom()`: Es la inversa de `pbinom()`. La idea es darle una probabilidad y que devuelva el valor que acumula hasta allí.
  - `p`: el valor a evaluar
  - `size`: cantidad de pruebas, o sea, el n
  - `prob`: probabilidad de éxito, o sea, p
  - `log.p`: si `FALSE` las probabilidades se devuelven como $log(p)$

```
> qbinom(0.5, 51, 1/2)
## [1] 25
> qbinom(0.25, 51, 1/2)
## [1] 23
```

Distribuciones de probabilidad: Normal

- `rbinom()`: arroja números aleatorios de una distribución binomial con los parámetros especificados.
  - `n`: cantidad de números aleatorios a devolver
  - `size`: cantidad de pruebas, o sea, el n
  - `sd`: el desvío std. de la distribución
  - `log.p`: si `FALSE` las probabilidades se devuelven como $log(p)$

```
> rbinom(5, size = 100, prob = 0.2)
## [1] 16 21 18 23 23
> rbinom(5, size = 100, prob = 0.7)
## [1] 68 68 69 72 77
```

Estadística descriptiva: `median()`

- Ya vimos algunas funciones para calcular estadísticos descriptivos en r
  - `mean()`
  - `median()`
  - `var()`
  - `sd()`
  - `summary()`
  - `table()`

Estadística descriptiva:

- Algunas otras funciones útiles son `quantile()`,`summary()`

```
> x <- rnorm(1000, 0, 1)
> quantile(x, prob = c(0.25, 0.5, 0.75))
##      25%      50%      75%
## -0.68353 -0.01297  0.65786
> summary(x)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -3.050  -0.684  -0.013   0.003   0.658   3.180
```