Aprendizaje Automático

Grado en Informática

Doble Grado Informática y Matemáticas

Información

- **PRÁCTICAS DE APRENDIZAJE AUTOMÁTICO (GRUPO 3)**

- **Horario:** Miércoles 17,30 – 19,30
- **Laboratorio 2.1**

- **Pablo Ruiz Matarán** e-mail: mataran@decsai.ugr.es
  Despacho D1-3 CITIC

- **Tutorías:** Martes 11,30 – 12,30
  Sala de Tutorías de la ETSIIT (Enviar e-mail previamente)

1. Console Input and Evaluation

Entering Input

- At the R prompt we type expressions. The <- symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

- The grammar of the language determines whether an expression is complete or not.

```
> x <- ## Incomplete expression
```

- The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored.

Evaluation

- When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
> x <- 5 ## nothing printed
> x ## auto-printing occurs
[1] 5
> print(x) ## explicit printing
[1] 5
```

- The `[1]` indicates that `x` is a vector and `5` is the first element.

Printing

```
> x <- 1:20
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
[16] 16 17 18 19 20
```

- The `:` operator is used to create integer sequences.

2. R Objects and Attributes

Objects

- R has five basic or “atomic” classes of objects:
  - character
  - numeric (real numbers)
  - integer
  - complex
  - logical (True/False)
- The most basic object is a vector
  - A vector can only contain objects of the same class
  - BUT: The one exception is a *list*, which is represented as a vector but can contain objects of different classes (indeed, that’s usually why we use them)
- Empty vectors can be created with the `vector()` function.

Numbers

- Numbers in R a generally treated as numeric objects (i.e. double precision real numbers)

- If you explicitly want an integer, you need to specify the `L` suffix. Example: Entering `1` gives you a numeric object; entering `1L` explicitly gives you an integer.

- There is also a special number `Inf` which represents infinity; e.g. `1/0`; `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is `0`

- The value `NaN` represents an undefined value ("not a number"); e.g. `0/0`; `NaN` can also be thought of as a missing value (more on that later)

Attributes

- R objects can have attributes
  - names, dimnames
  - dimensions (e.g. matrices, arrays)
  - class
  - length
  - other user-defined attributes/metadata
- Attributes of an object can be accessed using the `attributes()` function.

3. Vectors and Lists

Creating Vectors

- The `c()` function can be used to create vectors of objects.

```
> x <- c(0.5, 0.6)          ## numeric
> x <- c(TRUE, FALSE)       ## logical
> x <- c(T, F)              ## logical
> x <- c("a", "b", "c")     ## character
> x <- 9:29                 ## integer
> x <- c(1+0i, 2+4i)        ## complex
```

- Using the `vector()` function

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

Mixing Objects

- What about the following?

```
> y <- c(1.7, "a")
> y <- c(TRUE, 2)
> y <- c("a", TRUE)
```

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

Mixing Objects

- What about the following?

```
> y <- c(1.7, "a")   ## character
> y <- c(TRUE, 2)    ## numeric
> y <- c("a", TRUE) ## character
```

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

Explicit Coercion

- Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Explicit Coercion

- Nonsensical coercion results in NAs.

```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
> as.complex(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

Matrices

- Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices (cont'd)

- Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices (cont'd)

- Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

cbind-ing and rbind-ing

- Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

List

- Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

4. Data Frames

Data Frames

- Data frames are used to store tabular data
  - They are represented as a special type of list where every element of the list has to have the same length.
  - Each element of the list can be thought of as a column and the length of each element of the list is the number of rows
  - Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class.
  - Data frames also have a special attribute called `row.names`
  - Data frames are usually created by calling `read.table()` or `read.csv()`
  - Can be converted to a matrix by calling `data.matrix()`

Data Frames

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo  bar
1    1 TRUE
2    2 TRUE
3    3 FALSE
4    4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

5. Other objects

Names

- R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("foo", "bar", "norf")
> x
foo bar norf
  1   2    3
> names(x)
[1] "foo" "bar" "norf"
```

Names

- Lists can also have names.

```
> x <- list(a = 1, b = 2, c = 3)
> x
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

Names

- And matrices.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Missing Values

- Missing values are denoted by `NA` or `NaN` for undefined mathematical operations.
  - `is.na()` is used to test objects if they are `NA`
  - `is.nan()` is used to test for `NaN`
  - `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
  - A `NaN` value is also `NA` but the converse is not true

Missing Values

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
[1] FALSE FALSE TRUE FALSE FALSE
```

Summary

- Data Types
  - atomic classes: numeric, logical, character, integer, complex
  - vectors, lists
  - data frames
  - names
  - missing values

6. Subsetting

Subsetting

- There are a number of operators that can be used to extract subsets of R objects.
  - `[` always returns an object of the same class as the original; can be used to select more than one element
  - `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
  - `$` is used to extract elements of a list or data frame by name; semantics are similar to that of `[[`.

Subsetting

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
> x[x > "a"]
[1] "b" "c" "c" "d"
> u <- x > "a"
> u
[1] FALSE TRUE TRUE TRUE TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

Subsetting Lists

```
> x <- list(foo = 1:4, bar = 0.6)
> x[1]
$foo
[1] 1 2 3 4

> x[[1]]
[1] 1 2 3 4

> x$bar
[1] 0.6
> x[["bar"]]
[1] 0.6
> x["bar"]
$bar
[1] 0.6
```

Subsetting Lists

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> x[c(1, 3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

Subsetting Lists

- The `[[` operator can be used with *computed* indices; `$` can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
> x[[name]] ## computed index for 'foo'
[1] 1 2 3 4
> x$name ## element 'name' doesn't exist!
NULL
> x$foo
[1] 1 2 3 4 ## element 'foo' does exist
```

Subsetting Nested Elements of a List

- The `[[` can take an integer sequence.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
> x[[c(1, 3)]]
[1] 14
> x[[1]][[3]]
[1] 14

> x[[c(2, 1)]]
[1] 3.14
```

Subsetting a Matrix

- Matrices can be subsetted in the usual way with (*i,j*) type indices.

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

- Indices can also be missing.

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```

Subsetting a Matrix

- By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. This behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE]
     [,1]
[1,] 3
```

Subsetting a Matrix

- Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

Partial Matching

- Partial matching of names is allowed with `[[` and `$`.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

Removing NA Values

- A common task is to remove missing values (`NA`s).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5
```

Removing NA Values

- What if there are multiple things and you want to take the subset with no missing values?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1] TRUE TRUE FALSE TRUE FALSE TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
```

Removing NA Values

```
> airquality[1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> good <- complete.cases(airquality)
> airquality[good, ][1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149  12.6 74     5   3
4    18     313  11.5 62     5   4
7    23     299  8.6  65     5   7
```

7. Vectorized Operations

Vectorized Operations

- Many operations in R are *vectorized* making code more efficient, concise, and easier to read.

```
> x <- 1:4; y <- 6:9
> x + y
[1] 7 9 11 13
> x > 2
[1] FALSE FALSE TRUE TRUE
> x >= 2
[1] FALSE TRUE TRUE TRUE
> y == 8
[1] FALSE FALSE TRUE FALSE
> x * y
[1] 6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Vectorized Matrix Operations

```
> x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
> x * y ## element-wise multiplication
     [,1] [,2]
[1,]   10   30
[2,]   20   40
> x / y
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> x %*% y ## true matrix multiplication
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```