Introduction to R*

Part 3: Matrices/Arrays & Special Data Types

Wim R.M. Cardoen

Last updated: 07/22/2024 @ 12:50:04

Contents

```
"It is my experience that proofs involving matrices can be
shortened by 50% if one throws the matrices out" (Emil Artin)
```

1 Matrices & Arrays

Matrices and arrays are **homogeneous atomic vectors** with
an **extra** attribute: dimension

By default, the elements are stored in a **column-major** fashion. (cfr. **Fortran**).
However, we can store the elements in **row-major** order (cfr. **C**) as well.

1.1 Creation of matrices

Matrices can be created in different ways:

- use of the **matrix()** function
- use of **rbind()**/**cbind()**
- set the **attributes()** of a vector
- special functions like e.g. **diag()**

1.1.1 Examples

- use of the **matrix()** function:

The **matrix()** function creates a matrix based on a vector.
By default, the elements are stored in a **column-major** fashion.

The use of the flag `byrow=TRUE` will store the data in a **row-major** fashion.

```r
A <- matrix(data=1:10, nrow=2)   # Column-major (like Fortran)
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```r
B <- matrix(data=c(2,3,893,0.17), nrow=2, ncol=2)
B
```

```
     [,1]   [,2]
[1,]    2 893.00
[2,]    3   0.17
```

```r
C <- matrix(data=1:10, nrow=2, byrow=TRUE)   # Row-major (like C, C++)
C
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

- use of the **rbind()**/**cbind()** functions:
  - **rbind()**: Bind several vectors (as rows) into a matrix.
  - **cbind()**: Bind several vectors (as columns) into a matrix.

```r
A <- rbind(1:10,11:20)
A
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   11   12   13   14   15   16   17   18   19    20
```

```r
typeof(A)
```

```
[1] "integer"
```

```r
class(A)
```

```
[1] "matrix" "array"
```

```r
B <- cbind(1:5,6:10,11:15)
B
```

```
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```

```r
class(B)
```

```
[1] "matrix" "array"
```

1.1.2 Matrices: vectors with a non-NULL dim attribute

The **fundamental** difference between an `R` vector and matrix is the presence (in the case of matrices) of a non `NULL` **dim** attribute.

We can easily convert a vector into a matrix by setting the dimensions of the vector:

- through the **dim()** function.
- through the **attr()** function.

The inverse can be done as well by setting the **dim** attribute of matrix to `NULL`.

```r
A <- 1:10
typeof(A)
```

```
[1] "integer"
```

```r
class(A)
```

```
[1] "integer"
```

```r
dim(A)
```

```
NULL
```

```r
# Matrix
B <- matrix(1:10,nrow=2,ncol=5,byrow=TRUE)
typeof(B)
```

```
[1] "integer"
```

```r
class(B)
```

```
[1] "matrix" "array"
```

```r
dim(B)
```

```
[1] 2 5
```

```r
# Vector
A <- 1:10
A
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
dim(A)
```

```
NULL
```

```r
typeof(A)
```

```
[1] "integer"
```

```r
class(A)
```

```
[1] "integer"
```

```r
# OPTION I: Using the dim function transform a vector into a matrix
dim(A) <- c(2,5)
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```r
dim(A)
```

```
[1] 2 5
```

```r
typeof(A)
```

```
[1] "integer"
```

```r
class(A)
```

```
[1] "matrix" "array"
```

```r
# Converting the matrix back to a vector
dim(A) <- NULL
dim(A)
```

```
NULL
```

```r
typeof(A)
```

```
[1] "integer"
```

```r
class(A)
```

```
[1] "integer"
```

```r
# Option II: More general way
# Convert vector into a matrix
A <- 1:8
A
```

```
[1] 1 2 3 4 5 6 7 8
```

```r
class(A)
```

```
[1] "integer"
```

```r
attr(A,'dim') <- c(2,4)
A
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

```r
class(A)
```

```
[1] "matrix" "array"
```

```r
# Convert matrix into a vector.
attr(A, 'dim') <- NULL
A
```

```
[1] 1 2 3 4 5 6 7 8
```

```r
class(A)
```

```
[1] "integer"
```

1.2 Retrieving elements/subsetting

Matrices (and arrays) can be subsetted in different ways:

- use an **index** for each dimension, where the dimensions are comma-separated
  - If an **index** for a dimension is **omitted**:
    consider all dimensions (may lead to reduction of the dimension)
  - **but** you can use **drop=FALSE** to prevent dimensionality reduction.
- use another **vector** (can be either linear or a vector for each dimension)
- by using another **matrix**.

1.2.1 Examples

- Use of **indices**:

```r
A <- matrix(1:30, nrow=6, ncol=5)
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    2    8   14   20   26
[3,]    3    9   15   21   27
[4,]    4   10   16   22   28
[5,]    5   11   17   23   29
[6,]    6   12   18   24   30
```

```r
A[3,4]
```

```
[1] 21
```

```r
A[6,2]
```

```
[1] 12
```

```r
x1 <- A[2,]
x1
```

```
[1]  2  8 14 20 26
```

```r
dim(x1)
```

```
NULL
```

```r
x2 <- A[,3]
x2
```

```
[1] 13 14 15 16 17 18
```

```r
dim(x2)
```

```
NULL
```

The flag **drop=FALSE** can be used to prevent dimensionality reduction

```r
y1 <- A[2,,drop=FALSE]
y1
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    8   14   20   26
```

```r
dim(y1)
```

```
[1] 1 5
```

```r
y2 <- A[,3,drop=FALSE]
y2
```

```
     [,1]
[1,]   13
[2,]   14
[3,]   15
[4,]   16
[5,]   17
[6,]   18
```

```r
dim(y2)
```

```
[1] 6 1
```

- Use of **vector(s)**:

```r
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    2    8   14   20   26
[3,]    3    9   15   21   27
[4,]    4   10   16   22   28
[5,]    5   11   17   23   29
[6,]    6   12   18   24   30
```

```r
x1 <- A[2:4,]
x1
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    8   14   20   26
[2,]    3    9   15   21   27
[3,]    4   10   16   22   28
```

```r
dim(x1)
```

```
[1] 3 5
```

```r
x2 <- A[,1:3]
x2
```

```
     [,1] [,2] [,3]
[1,]    1    7   13
[2,]    2    8   14
[3,]    3    9   15
[4,]    4   10   16
[5,]    5   11   17
[6,]    6   12   18
```

```r
dim(x2)
```

```
[1] 6 3
```

```r
# Using a vector for EACH dimension
A[c(1,3),c(2,4)]
```

```
     [,1] [,2]
[1,]    7   19
[2,]    9   21
```

```r
# Using 1 vector => Linear index
A[c(1,3,8,10)]
```

```
[1]  1  3  8 10
```

```r
A[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE),c(2,3)]
```

```
     [,1] [,2]
[1,]    7   13
[2,]    9   15
[3,]   10   16
[4,]   12   18
```

```r
A[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE),]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    3    9   15   21   27
[3,]    4   10   16   22   28
[4,]    6   12   18   24   30
```

```r
# Use of a linear index
A[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE)]
```

```
 [1]  1  3  4  6  7  9 10 12 13 15 16 18 19 21 22 24 25 27 28 30
```

- Use of a **matrix**:

```r
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    2    8   14   20   26
[3,]    3    9   15   21   27
[4,]    4   10   16   22   28
[5,]    5   11   17   23   29
[6,]    6   12   18   24   30
```

```r
mysubset <- matrix(c( 2, 1,
                      3, 5,
                      4, 2,
                      6, 5), ncol=2, byrow=TRUE )
A[mysubset]
```

```
[1]  2 27 10 30
```

1.2.2 Exercises

- Create the following matrix A, given by:

```
     [,1] [,2] [,3]  [,4]    [,5]     [,6]
[1,]    3    9   27    81     243      729
[2,]    5   25  125   625    3125    15625
[3,]    7   49  343  2401   16807   117649
[4,]   11  121 1331 14641  161051  1771561
[5,]   13  169 2197 28561  371293  4826809
[6,]   17  289 4913 83521 1419857 24137569
```

1. get element 343
2. get the elements 25, 625, 2197 and 4826809 (all at once).
3. get the fourth row as a vector.
4. get the fourth row as a matrix.
5. get columns 2 and 3 (at the same time).
6. get everything except rows 2 and 4.
7. the diagonal of matrix A.

1.3 Operations on matrices

- Operations like ***,/, +** happpen element-wise.
- There are also more specialized functions:
  - the mean over rows and columns (**rowMeans()**, **colMeans()**)
  - linear algebra functions (**%*%**, **t()**, . . .)

1.3.1 Examples

- Operations (by **default: element-by-element**):

```r
A <- matrix(1:10, nrow=2)
B <- matrix( seq(10, 100, by=10), nrow=2)
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```r
B
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   10   30   50   70   90
[2,]   20   40   60   80  100
```

```r
A*B
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]   10   90  250  490  810
[2,]   40  160  360  640 1000
```

```r
C <- matrix(rep(2,10), nrow=2)
C
```

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

```r
C**A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    8   32  128  512
[2,]    4   16   64  256 1024
```

- Calculate **row and column means** :

```r
# Means of rows and columns
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```r
rowMeans(A)
```

```
[1] 5 6
```

```r
colMeans(A)
```

```
[1] 1.5 3.5 5.5 7.5 9.5
```

- **Matrix multiplication** (%*%) :

```r
A <- matrix(1:6, nrow=2)
A
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```r
B <- matrix(seq(10,120,by=10), nrow=3)
B
```

```
     [,1] [,2] [,3] [,4]
[1,]   10   40   70  100
[2,]   20   50   80  110
[3,]   30   60   90  120
```

```r
C <- A%*%B
C
```

```
     [,1] [,2] [,3] [,4]
[1,]  220  490  760 1030
[2,]  280  640 1000 1360
```

```
dim(C)
```

```
[1] 2 4
```

- **Linear algebra** routines

Some of the more common ones in R:

- **solve()** : **invert** a square matrix
- **diag()**
  - **extracts** the diagonal of a matrix when a matrix is provided.
  - **creates** a diagonal matrix when a vector is provided.
- **eigen()** : calculates the **eigenvalues** and **eigenvectors** of a matrix
- **det()** : calculates the **determinant** of a matrix.
- **t()**: calculates the **transpose**[1] of a matrix.

```
# Invert matrix A
A <- matrix(c(1, 3, 2, 4), ncol = 2, byrow = T)
Ainv <- solve(A)
Ainv %*% A
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

```
# Create a diagonal matrix
C <- diag(c(1,4,7))
C
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    4    0
[3,]    0    0    7
```

```
# Extract the diagonal elements
D <- matrix(1:8,nrow=4)
D
```

```
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

```
diag(D)
```

```
[1] 1 6
```

[1] Can also be used for dataframes (see later)

```r
# Calculate eigenvalues and eigenvectors of A
r <- eigen(A)
r
```

```
eigen() decomposition
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

```r
# Eigenvalues
r$values
```

```
[1]  5.3722813 -0.3722813
```

```r
# Matrix with eigenvectors
r$vectors
```

```
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

```r
# Diagonal Matrix (Similarity Transformation)
solve(r$vectors) %*% A %*% r$vectors
```

```
             [,1]          [,2]
[1,] 5.372281e+00  3.907616e-18
[2,] 1.650150e-16 -3.722813e-01
```

Note that under the hood `R` calls `BLAS` and `LAPACK`.

```r
# Find the version used of BLAS and LAPACK
La_library()
```

```
[1] "/usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.20.so"
```

```r
extSoftVersion()["BLAS"]
```

```
                                                      BLAS
"/usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3"
```

1.3.2 Exercises

- Linear regression:

  - **Step 1**:
    Create a **synthetic** data set by executing the following `R` code:

    ```r
    x <- seq(from=0, to=20.0, by=0.25)
    a <- 2.0
    b <- 1.5
    c <- 0.5
    y <- a + b*x + c*x^2 + rnorm(length(x))
    ```

  - **Step 2**:

Our goal is to use the following linear model, i.e.:

$$\begin{aligned} Y_i &= \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \epsilon_i \\ &= \beta_0 + x_{i1} \beta_1 + x_{i2} \beta_2 + \epsilon_i \end{aligned}$$

where $x_{ij} := x_i^j$.

The aforementioned equation takes on the following matrix form:

$$Y = X \beta + \epsilon \tag{1}$$

In Eq.(1), we have:

* Y : a $n \times 1$ column vector.
* X : a $n \times 3$ matrix (also known as the `design matrix`)
* β: a 3×1 column vector.
* ϵ is : a $n \times 1$ column vector and $\sim N(0, \sigma^2)$

An estimate for β ($\widehat{\beta}$) can be found (using Least-Squares, MLE see e.g. (Seber & Lee, 2012)) and has the following form:

$$\widehat{\beta} = (X^T X)^{-1} X^T Y \tag{2}$$

where,

the column vector Y is given by:

$$Y := \begin{bmatrix} y_1 \\ y_2 \\ .. \\ y_n \end{bmatrix}$$

and the matrix X[2] takes the following form:

$$X := \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix}$$

Note (additional background):
1. The underlying assumption for Eq. (2) is that the inverse of the matrix $(X^T X)$ exists. This is the case iff the $\texttt{rank}(X^T X) = \texttt{rank}(X)$ is maximal or if the the columns of the matrix X are linearly independent.
2. In the field of machine learning (ML), the vector β is split into 2 parts: the scalar $\texttt{b} := \beta_0$ (`bias`) and the remainder of the vector β, i.e. (`w`) also known as the `weight` vector.
In the current exercise, the column vector `w` is given by $\left(\beta_1 \beta_2\right)^T$.

Calculate $\widehat{\beta}$ using Eq.(2).

An estimate for the residuals ($\widehat{\epsilon}$) is given by:

$$\widehat{\epsilon} = Y - X \widehat{\beta} \tag{3}$$

Calculate $\widehat{\epsilon}$ using Eq.(3).

[2]This is a known as a Vandermonde matrix.

– **Step 3**:

You can check your results using the following R code.

```
myquadfit <- lm(y ~ x + I(x^2))
cat(sprintf("The estimates for beta::\n"))
cat(myquadfit$coefficients)
cat(sprintf("The residuals::\n"))
cat(myquadfit$residuals)
```

1.4 Hash tables/dictionaries

We can also use hashes for matrices. We can select one or both dimensions. To create hashes, for: - rows: use **rownames** - columns: use **colnames**

To remove the hash, use the **NULL** (like for vectors).

1.4.1 Examples

```
A1 <- c(0      , 5471.52, 5091.57, 5392.82,
        5416.45, 4584.33, 4904.83, 3851.73)
A2 <- c(5471.52,       0, 1315.28,  927.35,
        1505.11,  944.40, 1157.42, 1945.42)
A3 <- c(5091.57, 1315.28,       0, 2166.00,
        2724.01, 1571.76,  293.52, 1240.77)
A4 <- c(5392.82,  927.35, 2166.00,       0,
         577.85,  973.23, 1947.28, 2422.32)
A5 <- c(5416.45, 1505.11, 2724.01,  577.85,
              0, 1366.63, 2490.97, 2838.62)
A6 <- c(4584.33,  944.40, 1571.76,  973.23,
        1366.63,       0, 1290.15, 1474.26)
A7 <- c(4904.83, 1157.42,  293.52, 1947.28,
        2490.97, 1290.15,       0, 1064.41)
A8 <- c(3851.73, 1945.42, 1240.77, 2422.32,
        2838.62, 1474.26, 1064.41,       0)
```

```
dist <- rbind(A1,A2,A3,A4,A5,A6,A7,A8)
dist
```

```
      [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]
A1    0.00 5471.52 5091.57 5392.82 5416.45 4584.33 4904.83 3851.73
A2 5471.52    0.00 1315.28  927.35 1505.11  944.40 1157.42 1945.42
A3 5091.57 1315.28    0.00 2166.00 2724.01 1571.76  293.52 1240.77
A4 5392.82  927.35 2166.00    0.00  577.85  973.23 1947.28 2422.32
A5 5416.45 1505.11 2724.01  577.85    0.00 1366.63 2490.97 2838.62
A6 4584.33  944.40 1571.76  973.23 1366.63    0.00 1290.15 1474.26
A7 4904.83 1157.42  293.52 1947.28 2490.97 1290.15    0.00 1064.41
A8 3851.73 1945.42 1240.77 2422.32 2838.62 1474.26 1064.41    0.00
```

```
# Adding hashes to both rows and columns
cities <- c("Anchorage","Atlanta","Austin","Baltimore","Boston", "Chicago", "Dallas","Denver")
```

```r
rownames(dist) <- cities
colnames(dist) <- cities
dist
```

```
          Anchorage Atlanta  Austin Baltimore  Boston Chicago  Dallas  Denver
Anchorage      0.00 5471.52 5091.57   5392.82 5416.45 4584.33 4904.83 3851.73
Atlanta     5471.52    0.00 1315.28    927.35 1505.11  944.40 1157.42 1945.42
Austin      5091.57 1315.28    0.00   2166.00 2724.01 1571.76  293.52 1240.77
Baltimore   5392.82  927.35 2166.00      0.00  577.85  973.23 1947.28 2422.32
Boston      5416.45 1505.11 2724.01    577.85    0.00 1366.63 2490.97 2838.62
Chicago     4584.33  944.40 1571.76    973.23 1366.63    0.00 1290.15 1474.26
Dallas      4904.83 1157.42  293.52   1947.28 2490.97 1290.15    0.00 1064.41
Denver      3851.73 1945.42 1240.77   2422.32 2838.62 1474.26 1064.41    0.00
```

```r
dist["Chicago", "Denver"]
```

```
[1] 1474.26
```

```r
dist["Austin", "Boston"]
```

```
[1] 2724.01
```

1.5 Arrays

Say something about arrays.

2 Special Data Types (Factors and Date/Time types)

Every `R` object has attributes (i.e. properties or metadata).
They can be classified as:

- **intrinisic** properties e.g. **length()**
- **external** properties (to be set by the user)

2.1 Attributes

- can be get/retrieved using **attributes()**.
- can be set:
  - individually using **attr()**
  - in generally using **structure()**
- some attributes can (also) be set/unset with **special** functions:
  - names: **names()**
  - dimension: **dim()**
  - comment : **comment()**
  - time series: **tsp()**
  - factor : **factor()** (see next section)

2.1.1 Examples

- 1 attribute:

```
x <- 1:5
x
```

```
[1] 1 2 3 4 5
```

```
attr(x, 'prop1') <- "hello"
attributes(x)
```

```
$prop1
[1] "hello"
```

```
x
```

```
[1] 1 2 3 4 5
attr(,"prop1")
[1] "hello"
```

```
attr(x, 'prop1') <- NULL
attributes(x)
```

```
NULL
```

```
x
```

```
[1] 1 2 3 4 5
```

- more than 1 attribute:

```r
y <- 1:8
y
```

```
[1] 1 2 3 4 5 6 7 8
```

```r
y <- structure(y, dim=c(2,4), tag="trial")
y
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
attr(,"tag")
[1] "trial"
```

```r
attributes(y)
```

```
$dim
[1] 2 4

$tag
[1] "trial"
```

```r
typeof(y)
```

```
[1] "integer"
```

```r
class(y)
```

```
[1] "matrix" "array"
```

```r
# Remove BOTH attributes
y <- structure(y, dim=NULL, tag=NULL)
y
```

```
[1] 1 2 3 4 5 6 7 8
```

```r
attributes(y)
```

```
NULL
```

```r
typeof(y)
```

```
[1] "integer"
```

```r
class(y)
```

```
[1] "integer"
```

- **names()**

  ```r
  # Set the names attribute
  capitals <- c("Salt Lake City", "Carson City", "Boise", "Santa Fe")
  names(capitals) <- c("UT", "NV", "ID", "NM")
  capitals
  ```

```
              UT                NV                ID                NM 
"Salt Lake City"    "Carson City"          "Boise"       "Santa Fe" 
```

```r
attributes(capitals)
```

```
$names
[1] "UT" "NV" "ID" "NM"
```

```r
# Remove the names attribute
names(capitals) <- NULL
capitals
```

```
[1] "Salt Lake City" "Carson City"    "Boise"          "Santa Fe"      
```

- **dim()**

```r
x <- 1:12
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```r
typeof(x)
```

```
[1] "integer"
```

```r
class(x)
```

```
[1] "integer"
```

```r
# Set the dimension attribute
dim(x) <- c(3,4)
x
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```r
typeof(x)
```

```
[1] "integer"
```

```r
class(x)
```

```
[1] "matrix" "array" 
```

```r
# Remove the dimension attribute
dim(x) <- NULL
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```r
typeof(x)
```

```
[1] "integer"
```

```r
class(x)
```

```
[1] "integer"
```

- **comment()**

```r
x <- structure(1:6, comment="My vector")
typeof(x)
```

```
[1] "integer"
```

```r
class(x)
```

```
[1] "integer"
```

```r
comment(x)
```

```
[1] "My vector"
```

2.2 Factor variables (Categorical variables)

- Factor variables (factors, categorical variables) are discrete variables (i.e not continuous). The factors bear labels (**levels**) which are mapped into **integers**.
- Therefore, factors are stored as integer vector with 2 attributes:
  - **class**= "factor"
  - **levels**: a vector with the "labels".
- By default (**unordered**) the labels are mapped **alphabetically** to the integers. We can **impose** our own **ordering** between integers and labels (levels).
- Useful functions:
  - **levels()** : provides the levels of a factor
  - **table()**: returns the counts of each level
  - **is.factor()**: tests whether a variable is a factor variable
  - **is.ordered()**: tests whether a variable is an ordered factor variable

2.2.1 Examples

- Creation of an **unordered** factor

```r
# Creation of an unordered factor
temp.data <- c("High","Low","VeryHigh","Low","VeryLow","Medium",
                "VeryHigh","VeryHigh","Low","Low","Medium","VeryHigh",
                "VeryHigh","VeryHigh","Low","High","VeryLow")
myfac.temp.data <- factor(temp.data)
myfac.temp.data
```

```
 [1] High     Low      VeryHigh Low      VeryLow  Medium   VeryHigh VeryHigh
 [9] Low      Low      Medium   VeryHigh VeryHigh VeryHigh Low      High
```

```
[17] VeryLow
Levels: High Low Medium VeryHigh VeryLow
```

```r
# by default: the levels are stored ALPHABETICALLY (i.e. unordered)
levels(myfac.temp.data)
```

```
[1] "High"     "Low"      "Medium"   "VeryHigh" "VeryLow"
```

```r
table(myfac.temp.data)
```

```
myfac.temp.data
    High      Low   Medium VeryHigh  VeryLow
       2        5        2        6        2
```

```r
is.factor(myfac.temp.data)
```

```
[1] TRUE
```

```r
is.ordered(myfac.temp.data)
```

```
[1] FALSE
```

- Creation of an **ordered** factor

```r
# Creation of an unordered factor
temp.data <- c("High","Low","VeryHigh","Low","VeryLow","Medium",
               "VeryHigh","VeryHigh","Low","Low","Medium","VeryHigh",
               "VeryHigh","VeryHigh","Low","High","VeryLow")
myfac2.temp.data <- factor(temp.data, ordered=TRUE,
                           levels=c("VeryLow","Low","Medium","High","VeryHigh"))
myfac2.temp.data
```

```
 [1] High     Low      VeryHigh Low      VeryLow  Medium   VeryHigh VeryHigh
 [9] Low      Low      Medium   VeryHigh VeryHigh VeryHigh Low      High
[17] VeryLow
Levels: VeryLow < Low < Medium < High < VeryHigh
```

```r
# The ordering is NOW imposed
levels(myfac2.temp.data)
```

```
[1] "VeryLow"  "Low"      "Medium"   "High"     "VeryHigh"
```

```r
table(myfac2.temp.data)
```

```
myfac2.temp.data
 VeryLow      Low   Medium     High VeryHigh
       2        5        2        2        6
```

```r
is.factor(myfac2.temp.data)
```

```
[1] TRUE
```

```r
is.ordered(myfac2.temp.data)
```

```
[1] TRUE
```

```r
# Stripping a factor to the essentials: integer vector
attributes(myfac2.temp.data)
```

```
$levels
[1] "VeryLow"  "Low"      "Medium"   "High"     "VeryHigh"

$class
[1] "ordered" "factor"
```

```r
class(myfac2.temp.data) <- NULL
levels(myfac2.temp.data) <- NULL
myfac2.temp.data
```

```
 [1] 4 2 5 2 1 3 5 5 2 2 3 5 5 5 2 4 1
```

2.3 Dates and times in R.

- **Date** class :
  - represents calendar dates
  - built on top of doubles with class attribute 'Date'
  - 0 : Jan 1. 1970 (**Unix Epoch time**)
  - **as.Date()**: method to cast string to a Date
- **POSIXct** and **POSIXlt** : date and time
  - **POSIXct**: stores date/time values as the #seconds since Jan. 1, 1970
  - **POSIXlt**: stored as **bluelist** with elements for seconds, minutes, hours, day, month, year, etc.
- **lubridate**: a very useful package for dates and times:

2.3.1 Examples

- **Date**

```r
today <- Sys.Date()
today
```

```
[1] "2024-07-22"
```

```r
# Attributes of Date
class(today)
```

```
[1] "Date"
```

```r
attributes(today)
```

```
$class
[1] "Date"
```

```r
unclass(today)
```

```
[1] 19926
```

```r
d0 <- structure(0, class='Date')
d0
```

```
[1] "1970-01-01"
```

```r
class(d0)
```

```
[1] "Date"
```

```r
typeof(d0)
```

```
[1] "double"
```

```r
# Convert a string into a Date
d1  <- as.Date("2022-01-01")
d1
```

```
[1] "2022-01-01"
```

```r
class(d1)
```

```
[1] "Date"
```

```r
typeof(d1)
```

```
[1] "double"
```

- **POSIXct**

```r
# Convert a string into a POSIXct object
now_ct <- as.POSIXct("2018-08-01 22:00", tzone="MST")
now_ct
```

```
[1] "2018-08-01 22:00:00 MDT"
```

```r
attributes(now_ct)
```

```
$class
[1] "POSIXct" "POSIXt"

$tzone
[1] ""
```

```r
typeof(now_ct)
```

```
[1] "double"
```

```r
# Removal of the attributes
attr(now_ct,"tzone") <- NULL
unclass(now_ct)
```

```
[1] 1533182400
```

Bibliography

Seber G.A.F. & Lee A.J. (2012). Linear Regression Analysis. Wiley Series in Probability and Statistics. Wiley.