Matrices in R

Matrix Algebra 4 Statistical Learning

Gaston Sanchez

Why Matrix Algebra?

Data Matrix

Data

The analyzed data can be expressed in matrix format $\mathbf{X}$:

$$\underset{n \times p}{\mathbf{X}} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}$$

- n objects in the rows
- p variables in the columns

Motivation

- Multivariate data is commonly represented in tabular format (rows and columns)
- Mathematically, a data table can be treated as a matrix
- Matrix algebra provides the analytical machinery and tools to manipulate and exploit values, information, and patterns of variability in data

In these slides I will cover key concepts about matrix operations with R.

Vector-Matrix Notation

- upper-case italics for variables: X, Y
- bold upper-case for matrices: $\mathbf{A}, \mathbf{B}$
- bold lower-case letters for vectors: $\mathbf{x}, \mathbf{y}$
- lower-case italics are used to represent scalars: a, b
- x_i is the i-th element of $\mathbf{x}$
- A_{ij} is the element in the i-th row and j-th column of $\mathbf{A}$
- By default, we'll consider a vector $\mathbf{x}$ to be a one-column matrix
- $\mathbf{x}^\mathsf{T}$ to be a one-row matrix
- Occasionally I'll be using some greek letters too

Two Assumptions

I'm assuming 2 things about you:

Matrix Algebra & R basics

Matrix Algebra

You should have been exposed to concepts such as:

- Vector Spaces
- Inner Products
- Matrix Multiplication
- Linear Dependency
- Rank
- Trace, Determinant
- Inverse
- *etc*

R Basics

You should have been exposed to:

- R `vector`'s, `list`'s, `data.frame`'s
- Subscripting and indexing (i.e. bracket notation)
- Writing functions: `function() {...}`
- Conditionals: `if {...} else {...}`
- Loops: `for`, `while`, `repeat`
- Graphics: base, ggplot2, etc
- RStudio familiarity

Same but different

Math Objects

- vector
- matrix

Same but different

Math Objects

- vector
- matrix

R objects

- `vector`
- `matrix`

Back to the (R) Basics

R Vectors

- A vector is the most basic data structure in R
- Vectors are contiguous cells containing data
- Can be of any length (including zero)
- There are really no scalars, just one-element vectors

Vectors

The most simple type of vectors are “scalars” or single values:

```r
# integer
x <- 1L
# double (real)
y <- 5
# complex
z <- 3 + 5i
# logical
a <- TRUE
# character
b <- "yosemite"
```

although keep in mind that R does NOT really have scalars

Data modes (i.e. types)

- A **double** vector stores regular (i.e. real) numbers
- An **integer** vector stores integers (no decimal component)
- A **character** vector stores text
- A **logical** vector stores `TRUE`'s and `FALSE`'s values
- A **complex** vector stores complex numbers

Special Values

There are some special values

- `NULL` is the null object (it has length zero)
- Missing values are referred to by the symbol `NA`
  (there are different modes of `NA`: logical, integer, etc)
- `Inf` indicates positive infinite
- `-Inf` indicates negative infinite
- `NaN` indicates Not a Number
  (don't confuse `NaN` with `NA`)

Vectors

The function to create a vector from individual values is `c()`, short for *concatenate* or *combine*:

```
# some vectors
x <- c(1, 2, 3, 4, 5)

y <- c("one", "two", "three")

z <- c(TRUE, FALSE, FALSE)
```

Separate each element by a comma

Atomic Vectors

- vectors are **atomic** structures
- the values in a vector must be **ALL** of the same type
- either all integers, or reals, or complex, or characters, or logicals
- you cannot have a vector of different data types

Atomic Vectors

If you mix different data values, R will **implicitly coerce** them so they are all of the same type

```
# mixing numbers and characters
x <- c(1, 2, 3, "four", "five")
x

## [1] "1"    "2"    "3"    "four" "five"

# mixing numbers and logical values
y <- c(TRUE, FALSE, 3, 4)
y

## [1] 1 0 3 4
```

How does R coerce data types?

R follows two basic rules of implicit coercion

If a character is present, R will coerce everything else to characters

How does R coerce data types?

R follows two basic rules of implicit coercion

If a character is present, R will coerce everything else to characters

If a vector contains logicals and numbers, R will convert the logicals to numbers (`TRUE` to 1, `FALSE` to 0)

Properties of Vectors

- all vectors have a length
- vector elements can have associated names
- vectors are objects of class `"vector"`
- vectors have a mode (storage mode)

Vectorization

A vectorized computation is any computation that when applied to a vector operates on all of its elements

```
c(1, 2, 3) + c(3, 2, 1)

## [1] 4 4 4

c(1, 2, 3) * c(3, 2, 1)

## [1] 3 4 3
```

Recycling

When vectorized computations are applied, some problems may occur when dealing with two vectors of different length

```
c(1, 2, 3, 4) + c(1, 2)

## [1] 2 4 4 6

c(2, 1) + c(1, 2, 3)

## Warning in c(2, 1) + c(1, 2, 3):  longer object length
is not a multiple of shorter object length

## [1] 3 3 5
```

Recycling

The recycling rule can be very useful, like when operating between a vector and a “scalar”

```
x <- c(2, 4, 6, 8)
x + 3

## [1]  5  7  9 11
```

R Matrices and Arrays

From Vectors to Arrays

We can transform a vector in an **n-dimensional** array by giving it a dimensions attribute `dim`

```
# positive: from 1 to 8
x <- 1:8

# adding 'dim' attribute
dim(x) <- c(2, 4)
x

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

From Vectors to Arrays

- a vector can be given a `dim` attribute (dimensions)
- a `dim` attribute is a numeric vector of length `n`
- R will reorganize the elements of the vector into `n` dimensions
- each dimension will have as many rows (or columns, etc.) as the `n`-th value of the `dim` vector

From Vectors to Arrays

```
# dim attribute with 3 dimensions
dim(x) <- c(2, 2, 2)
x

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

From Vector to Matrix

A `dim` attribute of length 2 will convert a vector into a matrix

```
# vector to matrix
A <- 1:8
class(A)

## [1] "integer"

dim(A) <- c(2, 4)
class(A)

## [1] "matrix"
```

When using `dim()`, R always fills up each matrix by columns.

From Vector to Matrix

To have more control about how a matrix is filled, we use the `matrix()` function:

```
# vector to matrix (by rows)
a <- 1:8

A <- matrix(a, nrow = 2, ncol = 4)
A

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

From Vector to Matrix

If you want to fill a matrix by rows use `byrow = TRUE`

```
# vector to matrix (by rows)
b <- 1:8

B <- matrix(b, nrow = 2, ncol = 4, byrow = TRUE)
B

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

More about R Matrices

- R stores matrices (and arrays in general) as vectors.
- Which means that matrices are also **atomic** structures.
- Matrices in R are stored column-major (i.e. by columns).
- This is like Fortran, Matlab, and Julia, but not like C or Python (e.g. numpy).

Matrix: Column-major

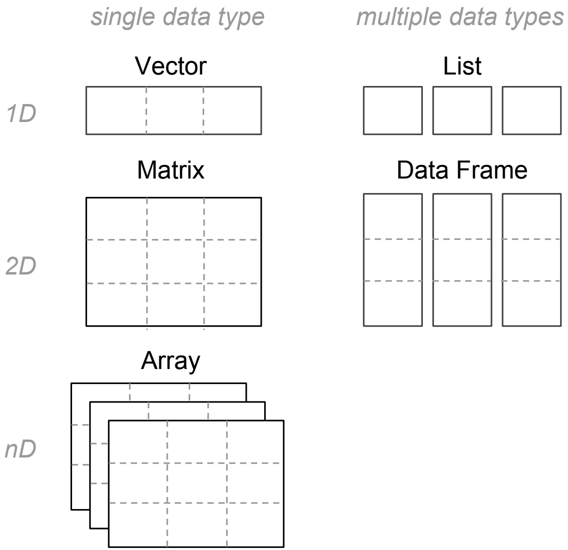
```r
as.vector(A)

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

as.vector(B)

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

Matrices and Vectors in R

It is important to distinguish vectors and matrices, especially in R:

- In matrix algebra we use the convention that vectors are column vectors (i.e. they are $n \times 1$ matrices).
- In R, a `vector` with n elements is not the same as an $n \times 1$ `matrix`.
- Vectors in R behave more like "row vectors" (especially when displayed).
- However, depending on the type of functions you apply to vectors, sometimes R will handle vectors like if they were column vectors.

R common data structures



Matrices vs Dataframes in R

It's also important to distinguish between matrices and data.frames:

- Both objects allow us to store data in a 2-dimensional object.
- In many cases, both R matrices and data.frames have similar behaviors.
- This is mostly the case when they are displayed on the screen.
- And sometimes it is hard to distinguish between a matrix and a data.frame by just looking at the displayed content on the screen.

Matrices vs Dataframes in R

- Dataframes are actually lists.
- Internally, a data.frame is stored as an R list (typically a list of vectors).
- This provides a very flexible way to manipulate data.frames, using the $ operator, double brackets `[[]]`, and the regular `[,]`
- An R matrix is internally stored as a vector (with a `dim` attribute).
- Which means that you use the `[,]` operator.

Matrices vs Dataframes in R

Both matrices and data.frames have common methods

- bracket notation: `[ , ]`
- dimensions: `dim()`
- number of rows: `nrow()`
- number of columns: `ncol()`
- dim names: `dimnames()`
- names of columns: `colnames()`
- names of rows: `rownames()`
- apply functions: `apply()`

Notation System

Notation system to extract values from R objects

- to extract values use brackets: `[ ]`
- inside the brackets specify indices
- use as many indices as dimensions in the object
- each index is separated by comma
- indices can be numbers, logicals, or names

Elements-Extraction Notation System

object	notation	example
vector	`[ ]`	`v[1:5]`
factor	`[ ]`	`g[1:5]`
matrix	`[ , ]`	`m[1:5, 1:3]`
array	`[ , , ]`	`arr[1, 2, 3]`
	`[ , , , ]`	`arr[1, 2, 3, 4]`
list	`[ ]`	`lst[3]`
	`[[ ]]`	`lst[[3]]`
	`$`	`lst$name`
data frame	`[ , ]`	`df[1, 2]`
	`$`	`df$name`

Brackets, Parentheses, and Braces in R

	Symbol	Use
[ ]	brackets	Objects
( )	parentheses	Functions
{ }	braces	Expressions

Basic Matrix Operations

Matrix Operations in R

We first quickly go through the basic matrix operations in R

- transpose
- addition
- scalar multiplication
- matrix-vector multiplication
- matrix-matrix multiplication

Matrix Transpose

The transpose of a $n \times p$ matrix $\mathbf{X}$ is the $p \times n$ matrix $\mathbf{X}^\mathsf{T}$. In R the transpose is given by the function `t()`

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

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

```
# transpose of X
t(X)
```

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

Matrix Addition

Matrix addition of two matrices $\mathbf{A} + \mathbf{B}$ is defined when $\mathbf{A}$ and $\mathbf{B}$ have the same dimensions:

```r
A <- matrix(1:6, 2, 3)
B <- matrix(7:9, 2, 3)
A + B
```

```
##      [,1] [,2] [,3]
## [1,]    8   12   13
## [2,]   10   11   15
```

Scalar Multiplication

We can multiply a matrix by a scalar using the usual product operator $*$, moreover it doesn't matter if we pre-multiply or post-multiply:

```
X <- matrix(1:3, 3, 4)

# (pre)multiply X by 0.5
(1/2) * X

##      [,1] [,2] [,3] [,4]
## [1,]  0.5  0.5  0.5  0.5
## [2,]  1.0  1.0  1.0  1.0
## [3,]  1.5  1.5  1.5  1.5
```

Scalar Multiplication

You can also postmultiply by a scalar (although this is not recommended because may confuse readers):

```
X <- matrix(1:3, 3, 4)

# (post)multiply X by 0.5
X * (1/2)

##      [,1] [,2] [,3] [,4]
## [1,]  0.5  0.5  0.5  0.5
## [2,]  1.0  1.0  1.0  1.0
## [3,]  1.5  1.5  1.5  1.5
```

Matrix-Matrix Multiplication

The matrix product operator in R is `%*%`. We can multiply matrices $\mathbf{A}$ and $\mathbf{B}$ if the number of columns of $\mathbf{A}$ is equal to the number of rows of $\mathbf{B}$

```r
A <- matrix(1:6, 2, 3)
B <- matrix(7:9, 3, 2)

A %*% B

##      [,1] [,2]
## [1,]   76   76
## [2,]  100  100
```

Matrix-Matrix Multiplication

We can multiply matrices $\mathbf{A}$ and $\mathbf{B}$ if the number of columns of $\mathbf{A}$ is equal to the number of rows of $\mathbf{B}$

```
A <- matrix(1:6, 2, 3)
B <- matrix(7:9, 3, 2)

B %*% A
```

```
##      [,1] [,2] [,3]
## [1,]   21   49   77
## [2,]   24   56   88
## [3,]   27   63   99
```

Cross-Products

A very common type of products in multivariate data analysis are $\mathbf{X}^\mathsf{T}\mathbf{X}$ and $\mathbf{X}\mathbf{X}^\mathsf{T}$, sometimes known as cross-products:

```
# output with return()
# cross-product
t(A) %*% A

##      [,1] [,2] [,3]
## [1,]    5   11   17
## [2,]   11   25   39
## [3,]   17   39   61
```

```
# cross-product
A %*% t(A)

##      [,1] [,2]
## [1,]   35   44
## [2,]   44   56
```

Cross-Products

R provides functions `crossprod()` and `tcrossprod()` which are formally equivalent to:

- `crossprod(X, X)` $\equiv$ `t(X) %*% X`
- `tcrossprod(X, X)` $\equiv$ `X %*% t(X)`

However, `crossprod()` and `tcrossprod()` are usually faster than using `t()` and `%*%`

Matrix-Vector Multiplication

We can post-multiply an $n \times p$ matrix $\mathbf{X}$ with a vector $\mathbf{b}$ with p elements. This means making linear combinations (weighted sums) of the columns of $\mathbf{X}$:

```
X <- matrix(1:12, 3, 4)
b <- seq(0.25, 1, by = 0.25)
X %*% b
```

```
##      [,1]
## [1,] 17.5
## [2,] 20.0
## [3,] 22.5
```

Vector-Matrix Multiplication

In R we can pre-multiply a vector $\mathbf{a}$ (with n elements) with an $n \times p$ matrix $\mathbf{X}$. This means making linear combinations (weighted sums) of the rows of $\mathbf{X}$:

```
X <- matrix(1:12, 3, 4)
a <- 1:3
a %*% X

##      [,1] [,2] [,3] [,4]
## [1,]   14   32   50   68
```

Note about %*%

Notice that when we use the product operator `%*%`, R is smart enough to use the convention that vectors are $n \times 1$ matrices. Notice also that if we ask for a vector-matrix multiplication, we can use both formulas:

- `a %*% X`
- `t(a) %*% X`

R will reformat the n vector as an $n \times 1$ matrix first.

Other Functions

Here are some other interesting functions for matrices

- `det()`: determinant
- `diag()`: extract or replace the diagonal elements
- `solve()`: solve system of equations
- `svd()`: singular value decomposition
- `eigen()`: eigen-decomposition
- `qr()`: QR decomposition