Matrices

January 29, 2020

Matrices

- A matrix is a vector with two additional attributes:
  - the number of rows
  - the number of columns.
- Since matrices are vectors, they also have modes, such as numeric and character.
- On the other hand, vectors are not one-column or one-row matrices.

Creating Matrices

- Matrix row and column subscripts begin with 1.
- For example, the upper-left corner of the matrix a is denoted a[1,1].
- The internal storage of a matrix is in column-major order, meaning that first all of column 1 is stored, then all of column 2, and so on.

Creating Matrices

One way to create a matrix is by using the matrix() function:

```
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> y
   [,1] [,2]
[1,] 1 3
[2,] 2 4
```

Creating Matrices

- You do not need to specify both ncol and nrow in *matrix()* function
- Just nrow or ncol would have been enough.

```
#Having four elements in all, in two rows, implies two columns:
> y <- matrix(c(1,2,3,4),nrow=2)
> y
  [,1] [,2]
[1,] 1 3
[2,] 2 4
```

Creating Matrices

Another way to build y is to specify elements individually:

```
> y <- matrix(nrow=2,ncol=2)
> y[1,1] <- 1
> y[2,1] <- 2
> y[1,2] <- 3
> y[2,2] <- 4
> y
  [,1] [,2]
[1,] 1 3
[2,] 2 4
```

Creating Matrices

- Though internal storage of a matrix is in column-major order, you can set the **byrow** argument in matrix() to **TRUE** to indicate that the data is coming in **row-major** order.
- Heres an example of using byrow:

```
> m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=T)
> m
   [,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
```

General Matrix Operations

Let's look at some common operations performed with matrices. These include

- Performing linear algebra operations
- Matrix indexing
- Matrix filtering

General Matrix Operations

You can perform various linear algebra operations on matrices, such as:

- matrix multiplication
- matrix scalar multiplication
- matrix addition

General Matrix Operations

Using y from the preceding example *y*, here is how to perform those three operations:

```
> y
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
> 3*y # mathematical multiplication of matrix by scalar
  [,1] [,2]
[1,] 3 9
[2,] 6 12
> y+y # mathematical matrix addition
  [,1] [,2]
[1,] 2 6
[2,] 4 8
```

Matrix Indexing

The same operations we discussed for vectors, apply to matrices as well. Heres an example:

```
> z
[,1] [,2] [,3]
[1,] 1 1 1
[2,] 2 1 0
[3,] 3 0 1
[4,] 4 0 0

> z[,2:3]
  [,1] [,2]
[1,] 1 1
[2,] 1 0
[3,] 0 1
[4,] 0 0
```

Matrix Indexing

You can also assign values to submatrices:

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

> y[c(1,3),] <- matrix(c(1,1,8,12),nrow=2)
```

Question: What is y?

Matrix Indexing

Answer:

```
> y[c(1,3),] <- matrix(c(1,1,8,12),nrow=2)

> y
   [,1] [,2]
[1,] 1 8
[2,] 2 5
[3,] 1 12
```

Matrix Indexing

Negative subscripts, used with vectors to **exclude** certain elements, work the same way with matrices:

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

> y[-2,]
```

Question: What is y now?

Matrix Indexing

Answer:

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

> y[-2,]
     [,1] [,2]
[1,] 1 4
[2,] 3 6
```

Filtering on Matrices

- Filtering can be done with matrices, just as with vectors.
- Let's start with a simple example:

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

> x[x[,2] >= 3,]
```

Question: What is the output?

Filtering on Matrices

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

Output:

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

Filtering on Matrices

Let's dissect this

```
> x
     x
[1,] 1 2
[2,] 2 3
[3,] 3 4
> j <- x[,2] >= 3
> j
[1] FALSE TRUE TRUE

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

Another example:

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

> a <- m[m[,1] > 1 & m[,2] > 5,]
```

Question: What is a?

Filtering on Matrices

Anwser:

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

> a = m[m[,1] > 1 & m[,2] > 5,]
> a
[1] 3 6
```

- The expression *m[,1]* compares each element of the first column of m to 1 and returns *(FALSE,TRUE,TRUE)*.
- The second expression, *m[,2]* to 5, similarly returns *(FALSE,FALSE,TRUE)*.
- We then take the logical **AND**(&) of *(FALSE,TRUE,TRUE)* and *(FALSE,FALSE,TRUE)*, yielding *(FALSE,FALSE,TRUE)*.

Filtering on Matrices

- Since **matrices are vectors**, you can also apply vector operations to them.
- Here is an example:

```
> m
     [,1] [,2]
[1,] 5 -1
[2,] 2 10
[3,] 9 11

> which (m > 2)
```

Output?

Filtering on Matrices

```
> m
     [,1] [,2]
[1,] 5 -1
[2,] 2 10
[3,] 9 11
```

```
> which (m > 2)
[1] 1 3 5 6
```

Applying Functions to Matrix Rows and Columns

- One of the most used features of R is the **apply() family of functions, such as apply()**, **tapply()**, and **lapply()**.
- we will look at **apply()**, which instructs R to call a user-specified function on each of the rows or each of the columns of a matrix.

Using the apply() Function

This is the general form of apply for matrices, where the arguments are as follows:

- m is the matrix.
- dimcode is the dimension, equal to 1 if the function applies to rows or 2 for columns.
- f is the function to be applied.
- fargs is an optional set of arguments to be supplied to f.

```
apply(m,dimcode,f,fargs)
```

Using the apply() Function

For example, here we apply the R function mean() to each column of a matrix z:

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

> apply(z,2,mean)
[1] 2 5
```

Using the apply() Function

- A function you write yourself is just as legitimate for use in apply() as any R built-in function such as mean().
- Here is an example using our own function f:

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

> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
```

Question: What is y?

Using the apply() Function

Anwser:

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

> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
> y
[,1] [,2] [,3]
[1,] 0.5 1.000 1.50
[2,] 0.5 0.625 0.75
```

Using the apply() Function

- The first computation, (0.5,0.5), ends up at the first column in the output of **apply()**, not the first row.
- This is the behavior of **apply()**.
- If the function to be applied returns a vector of k components, then the result of **apply()** will have k rows.
- You can use the matrix **transpose** function **t()** to change it if necessary

```
> t(apply(z,1,f))
     [,1]  [,2]
[1,] 0.5 0.500
[2,] 1.0 0.625
[3,] 1.5 0.750
```

Using the apply() Function

```
> z
[,1] [,2]
[1,] 1 4
[2,] 2 5
[3,] 3 6
> f <- function(x) x/c(2,8)
> y <- apply(z,2,f)
```

Question: What is y now?

Using the apply() Function

```
> z
[,1] [,2]
[1,] 1 4
[2,] 2 5
[3,] 3 6
> f <- function(x) x/c(2,8)
```

```
> y <- apply(z,2,f)
Warning messages:
1: In x/c(2, 8) :
  longer object length is not a multiple of shorter object length
2: In x/c(2, 8) :
  longer object length is not a multiple of shorter object length
> y
     [,1] [,2]
[1,] 0.50 2.000
[2,] 0.25 0.625
[3,] 1.50 3.000
```

Recycling

Recycling would be used if x had a length longer than 2

```
> f <- function(x) x/c(2,8)
> y <- apply(z,2,f)
```

apply()

- The function to be applied needs to take at least one argument.
- In some cases, you will need additional arguments for this function, which you can place following the function name in your call to apply().

An example: apply()

Suppose we have a matrix of 1s and 0s and want to create a vector as follows:

- For each row of the matrix, the corresponding element of the vector will be either 1 or 0, depending on whether the majority of the first **d** elements in that row is 1 or 0.
- **d** is a parameter that we can change.

Question: Write a R function that called copymaj to create the vector

```
> x
[,1] [,2] [,3] [,4] [,5]
[1,] 1 0 1 1 0
[2,] 1 1 1 1 0
[3,] 1 0 0 1 1
[4,] 0 1 1 1 0

> apply(x,1,copymaj,3)
[1] 1 1 0 1
```

Using the apply() Function

We could do this:

```
> copymaj <- function(rw,d) {
   maj <- sum(rw[1:d]) / d
   return(if(maj > 0.5) 1 else 0)
}
> x
[,1] [,2] [,3] [,4] [,5]
[1,] 1 0 1 1 0
[2,] 1 1 1 1 0
[3,] 1 0 0 1 1
[4,] 0 1 1 1 0

> apply(x,1,copymaj,3)
[1] 1 1 0 1
> apply(x,1,copymaj,2)
[1] 0 1 0 0
```

Adding and Deleting Matrix Rows and Columns

Recall how we reassign vectors to change their size:

```
> x
[1]12 51316 8
> x <- c(x,20) # append 20 >x
[1]12 51316 820
```

Analogous operations can be used to change the size of a matrix. For instance, the **rbind()** and **cbind()** let you add rows or columns to a matrix.

```
> one
[1] 1 1
> z
  [,1] [,2]
[1,] 1 3
[2,] 2 4
> cbind(one,z)
[1,]1 1 3
[2,]1 2 4
```

Adding and Deleting Matrix Rows and Columns

You can also use the **rbind()** and **cbind()** functions as a quick way to create small matrices. Heres an example:

```
> q <- cbind(c(1,2),c(3,4))
> q
    [,1] [,2]
[1,] 1 3
[2,] 2 4
```

Adding and Deleting Matrix Rows and Columns

- If you are adding rows or columns one at a time within a loop, and the matrix will eventually become large
- It is better to allocate a large matrix in the first place.
    - It will be empty at first, but you fill in the rows or columns one at a time, rather than doing a time-consuming matrix memory allocation each time.

Adding and Deleting Matrix Rows and Columns

You can delete rows or columns by reassignment, too:

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

Question: What is m now?

Adding and Deleting Matrix Rows and Columns

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

Anwser:

```
> m
     [,1] [,2]
[1,] 1 4
[2,] 3 6
```

Avoiding Unintended Dimension Reduction

In R, something else might merit the name dimension reduction that we may sometimes wish to avoid

```
> z
[,1] [,2]
[1,] 1 5
[2,] 2 6
[3,] 3 7
[4,] 4 8
> r <- z[2,]
> r
[1] 2 6
```

Avoiding Unintended Dimension Reduction

r is a vector of length 2, rather than a 1-by-2 matrix. We can confirm this in a couple of ways:

```
> dim(z)
[1] 4 2
> dim(r)
NULL

> class(z)
[1] "matrix"

> class (r)
[1] "integer"
```

Avoiding Unintended Dimension Reduction

R has a way to suppress this dimension reduction: the **drop** argument.

```
> r <- z[2,, drop=FALSE]

> r
[,1] [,2]
[1,] 2 6

> dim(r)
[1] 1 2

> class(r)
[1] "matrix"
```

Conversion

```
> u
[1] 1 2 3
> v <- as.matrix(u)

> class(u)
[1] "integer"
> class(v)
[1] "matrix"
```

Function **attributes()** return the name-value pairs that attach metadata to an object.

```
> attributes(u)
NULL
> attributes(v)
$dim
[1] 3 1
```