The Art of R Programming

Norman Matloff

September 1, 2009
4.9 Filtering on Matrices

Filtering can be done with matrices too. Note that one must be careful with the syntax. For instance:

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

Again, let’s dissect this:

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

Here is another example:

```r
> m <- matrix(c(1,2,3,4,5,6),nrow=3)
> m
   [,1] [,2]
[1,] 1  4
[2,] 2  5
[3,] 3  6
> m[m[,1] > 1,]
   [,1] [,2]
[1,] 2  5
[2,] 3  6
> m[m[,1] > 1 & m[,2] > 5,]
   [,1]     [,2]
[1,] 3  6
```
4.10 Applying the Same Function to All Rows or Columns of a Matrix

4.10.1 The apply() Function

The arguments of apply() are the matrix/data frame to be applied to, the dimension—1 if the function applies to rows, 2 for columns—and the function to be applied.

For example, here we apply the built-in R function mean() to each column of a matrix z.

```r
> z
[,1] [,2]
[1,] 1 4
[2,] 2 5
[3,] 3 6
> apply(z,2,mean)
[1] 2 5
```

Here is an example of working on rows, using our own function:

```r
> 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
```

You might be surprised that the size of the result here is 2 x 3 rather than 3 x 2. If the function to be applied returns a vector of k components, the result of apply() will have k rows. You can use the matrix transpose function t() to change it.

As you can see, the function to be applied needs at least one argument, which will play the role of one row or column in the array. In some cases, you will need additional arguments, which you can place following the function name in your call to apply().

For instance, 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 c elements in that row are 1 or 0. Here c will be a parameter which we may wish to vary. We could do this:

```r
> copymaj <- function(rw,c) {
+ maj <- sum(rw[1:c]) / c
+ return(ifelse(maj > 0.5,1,0))
+ }
> x <- matrix(c(1,1,1,0, 0,1,0,1, 1,1,1,1, 1,1,1,1, 0,0,1,0),nrow=4)
> x
[1,] 1 0 1 1 0
[2,] 1 1 1 1 0
```
Here the values 3 and 2 form the actual arguments for the formal argument \( c \) in \texttt{copymaj()}.

So, the general form of \texttt{apply} is

\[
\texttt{apply(m, dimcode, f, fargs)}
\]

where \( m \) is the matrix, \texttt{dimcode} is 1 or 2, according to whether we will operate on rows or columns, \( f \) is the function to be applied, and \texttt{fargs} is an optional list of arguments to be supplied to \( f \).

If \( f() \) is only to be executed here, and if \texttt{fargs} consists of variables visible here, we might consider “inlining” it by defining it just before the call to \texttt{apply()}, as described in Section 9.5. In that case \texttt{fargs} would not be necessary.

Note carefully that in writing \( f() \) itself, its first argument must be a vector that will be supplied by the caller as a row or column of \( m \).

As R moves closer and closer to parallel processing, functions like \texttt{apply()} will become more and more important. For example, the \texttt{clusterApply()} function in the \texttt{snow} package gives R some parallel processing capability, by distributing the submatrix data to various network nodes, with each one basically running \texttt{apply()} on its submatrix, and then collect the results. See Section 17.3.

### 4.10.2 The \texttt{sapply()} Function

If we call a vectorized function whose return value is a vector, the result is, in essence, a matrix.

\[
z12 <- \text{function}(z) \ \text{return}(c(z, z^2))
x <- 1:8
\]

> \texttt{z12(x)}
\[
[1] \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 1 \ 4 \ 9 \ 16 \ 25 \ 36 \ 49 \ 64
\]

> \texttt{matrix(z12(x), ncol=2)}
\[
[,1] \ [,2]
[1,] 1 \ 1
[2,] 2 \ 4
[3,] 3 \ 9
[4,] 4 \ 16
[5,] 5 \ 25
[6,] 6 \ 36
[7,] 7 \ 49
[8,] 8 \ 64
\]

We can streamline things using \texttt{sapply()}:
4.11 Digging a Little Deeper on the Vector/Matrix Distinction

It was stated at the outset of this chapter that

A matrix is a vector with two additional attributes, the number of rows and number of columns.

Let’s look at this a bit more closely:

> z <- matrix(1:8,nrow=4)
> z

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

Looks fine. But z is still a vector, so that for instance we can query its length:

> length(z)
[1] 8

But as a matrix, z is a bit more than a vector:

> class(z)
[1] "matrix"
> attributes(z)
$dim
[1] 4 2

In other words, there actually is a matrix class, in the object-oriented programming sense. We’ll cover OOP in Chapter 12, but for now it will suffice to say that R classes use a dollar sign to denote members of a class, just like C++, Python and so on use a period. So, we see that the matrix class has one attribute, named dim, which is a vector containing the numbers of rows and columns in the matrix.

You can also obtain dim via the dim() function:

> dim(z)
[1] 4 2
The numbers of rows and columns are obtainable individually via the \texttt{nrow()} and \texttt{ncol()} functions:

\begin{verbatim}
> nrow(z)
[1] 4
> ncol(z)
[1] 2
\end{verbatim}

These just piggyback on \texttt{dim()}, as you can see by inspecting the code (functions, as objects, can be printed in interactive mode by simply typing their names), e.g.

\begin{verbatim}
> nrow
function (x)
dim(x)[1]
\end{verbatim}

This calls \texttt{dim()}, then extracts element 1 from the resulting vector.

These functions are useful when you are writing a general-purpose library function whose argument is a matrix. By being able to sense the number of rows and columns in your code, you alleviate the caller of the burden of supplying that information as two additional arguments.