Array operations in the **gRbase** package

Søren Højsgaard

gRbase version 1.7-1 as of 2015-08-21

Contents

1 Introduction

This note describes some operations on arrays in R. These operations have been implemented to facilitate implementation of graphical models and Bayesian networks in R.

2 Arrays/tables in R

The documentation of R states the following about arrays:

An array in R can have one, two or more dimensions. It is simply a vector which is stored with additional attributes giving the dimensions (attribute "dim") and optionally names for those dimensions (attribute "dimnames").

A two-dimensional array is the same thing as a matrix.

One-dimensional arrays often look like vectors, but may be handled differently by some functions.

Hence the defining characterstic of an array is that it is a vector with a dim attribute. For example

```
> ## 1-dimensional array
> x1 <- 1:8
> dim(x1) <- 8
> x1
[1] 1 2 3 4 5 6 7 8
> c(is.array(x1), is.matrix(x1))
[1]  TRUE FALSE
> ## 2-dimensional array (matrix)
> x2 <- 1:8
> dim(x2) <- c(2,4)
> x2
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> c(is.array(x2), is.matrix(x2))
[1] TRUE TRUE
> ## 3-dimensional array
> x3 <- 1:8
> dim(x3) <- c(2,2,2)
> x3
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
> c(is.array(x3), is.matrix(x3))
[1]  TRUE FALSE
```

Notice that arrays do not need a `dimnames` attribute. However, for some of the operations described in the following, `dimnames` are essential.

Next consider the `lizard` data in `gRbase`:

```
> data( lizard, package="gRbase" )
> lizard
, , species = anoli

      height
diam   >4.75 <=4.75
  <=4     32     86
  >4      11     35

, , species = dist

      height
```

```
diam  >4.75 <=4.75
  <=4    61    73
  >4     41    70
```

Data has `dim` and `dimnames` attributes (and the list of `dimnames` has names; this is important):

```
> dim( lizard )
[1] 2 2 2
> dimnames(lizard)

$diam
[1] "<=4" ">4"

$height
[1] ">4.75"  "<=4.75"

$species
[1] "anoli" "dist"
```

Notice from the output above that the first variable (`diam`) varies fastest.

We can subset arrays in different ways:

```
> ## because lizard is a vector
> lizard[1:2]
[1] 32 11
> ## because lizard is an array
> lizard[,1,1]

<=4  >4
 32  11
> ## useful if subsetting programatically
> margin <- 2:3; level <- c(1,1)
> z <- as.list(rep(TRUE,3))
> z[margin] <- level
> str(z)

List of 3
 $ : logi TRUE
 $ : num 1
 $ : num 1
> do.call("[", c(list(lizard), z))

<=4  >4
 32  11
```

It is worth noticing that the result is not necessarily an array:

```
> is.array( lizard[1:2] )
[1] FALSE
> is.array( lizard[,1,1] )
[1] FALSE
> is.array( lizard[,1,] )
[1] TRUE
```

3   Operations on tables

Consider again the `lizard` data In the following we shall denote the dimnames (or variables) by D, H and S and we let (d, h, s) denote a configuration of these variables. The contingency table above shall be denoted by T_{DHS} and we shall refer to the (d, h, s) entry as $T_{DHS}(d, h, s)$.

```
> T.DHS <- lizard
```

3.1 Marginal tables

The DS–marginal table T_{DS} is defined to be the table with values

$$T_{DS}(d,s) = \sum_h T_{DHS}(d,h,s)$$

In R:

```
> T.DS <- tabMarg(lizard, ~diam+species); T.DS
      species
diam  anoli dist
  <=4   118  134
  >4     46  111
> ## Alternative forms
> T.DS <- tabMarg(lizard, c("diam","species"))
> T.DS <- tabMarg(lizard, c(1,3))
```

3.2 Permuting a table

A reorganization of the table can be made with `tabPerm`:

```
> T.SHD <- tabPerm(T.DHS, ~species+height+diam); ftable( T.SHD )
                diam <=4 >4
species height
anoli   >4.75          32 11
        <=4.75         86 35
dist    >4.75          61 41
        <=4.75         73 70
> ## Alternative forms:
> T.SHD <- tabPerm(T.DHS, c("species","height","diam"))
> T.SHD <- tabPerm(T.DHS, c(3,2,1))
> T.SHD <- aperm(T.DHS, c(3,2,1))
```

3.3 Slice of a table

A slice of a table is obtained with `tabSlice`:

```
> tabSlice(lizard, slice=list(species="anoli"))
      height
diam  >4.75 <=4.75
  <=4    32     86
  >4     11     35
```

3.4 Operations on two tables: $+$, $-$, $*$, $/$

The product of two tables, e.g. T_{DS} and T_{HS} is defined to be the table $\tilde{T}_{DHS}$ with entries

$$\tilde{T}_{DHS}(d,h,s) = T_{DS}(d,s)T_{HS}(h,s)$$

In R:

```
> T.HS <- tabMarg(lizard, ~height+species)
> T.DHS.mult = tabMult( T.DS, T.HS )
> ftable( T.DHS.mult )
               diam    <=4    >4
height species
>4.75  anoli          5074  1978
       dist          13668 11322
<=4.75 anoli         14278  5566
       dist          19162 15873
```

4

The quotient, sum and difference is defined similarly:

```
> T.DHS.div  = tabDiv( T.DS, T.HS )
> T.DHS.add  = tabAdd( T.DS, T.HS )
> T.DHS.subt = tabSubt( T.DS, T.HS )
```

3.5 Miscellaneous

Consider this way of "blowing up" an array with extra dimensions.

```
> T.HSD2 <- tabExpand(T.DS, T.HS); T.HSD2
, , diam = <=4

        species
height    anoli dist
  >4.75     118  134
  <=4.75    118  134

, , diam = >4

        species
height    anoli dist
  >4.75      46  111
  <=4.75     46  111

> names(dimnames(T.HSD2))
[1] "height"  "species" "diam"
```

Here `T.HSD2` is a 3–way table with the same variable names as the union of the variable names in `T.DS` and `T.HS`. Those variables in those variables in `T.HS` vary fastest. Lastly, if we regards `T.HSD2` as a function of (h, s, d) we see that `T.HSD2` is constant as a function of s.

Next, `tabAlign()` will allign the first array to have the same variable order as the second array:

```
> tabAlign(T.SHD, T.DHS)
, , species = anoli

      height
diam  >4.75 <=4.75
  <=4     32     86
  >4      11     35

, , species = dist

      height
diam  >4.75 <=4.75
  <=4     61     73
  >4      41     70
```

An also be achieved as

```
> n.new <- names(dimnames(T.DHS)); n.new
[1] "diam"    "height"  "species"
> n.old <- names(dimnames(T.SHD)); n.old
[1] "species" "height"  "diam"
> if (setequal( n.new, n.old )){
+     tabPerm( T.SHD, n.new )
+ } else {
+     numeric(0)
+ }
, , species = anoli

      height
diam  >4.75 <=4.75
  <=4     32     86
```

```
  >4      11      35

, , species = dist

      height
diam  >4.75 <=4.75
  <=4     61      73
   >4     41      70
```

Next we consider comparisons of arrays:

```
> tabEqual( T.SHD, T.DHS )
[1] TRUE
```

These two tables are defined to be identical because after a permutation of the first table we end up with a table with varable names (in the same order) as the second table and the elements. The elements are numerically identical.

4   Defining tables / arrays

As mentioned above, a table can be represented as an array. In general, arrays do not need dimnames in R, but for the functions described here, the dimnames are essential. We shall just notice that in addition to `array()`, a an array can also be defined using `parray()` from `gRbase`. For example

```
> yn <- c("y","n")
> T.AB <- array(c(5,95,1,99), dim=c(2,2), dimnames=list("A"=yn, "B"=yn))
> T.AB <- parray(c("A","B"), levels=list(yn, yn), values=c(5,95,1,99))
```

Using `parray()`, arrays can be normalized in two ways: Normalization can be over the first variable for *each* configuration of all other variables or over all configurations. For example:

```
> T.AB <- parray(c("A","B"), levels=list(yn, yn), values=c(5,95,1,99),
+               normalize="first")
   B
A      y     n
  y 0.05 0.01
  n 0.95 0.99
> T.AB <- parray(c("A","B"), levels=list(yn, yn), values=c(5,95,1,99),
+               normalize="all")
   B
A       y      n
  y 0.025 0.005
  n 0.475 0.495
```


5   Example: A Bayesian network

A classical example of a Bayesian network is the "sprinkler example", see e.g. `http://en.wikipedia.org/wiki/Bayesian_network`:

> *Suppose that there are two events which could cause grass to be wet: either the sprinkler is on or it is raining. Also, suppose that the rain has a direct effect on the use of the sprinkler (namely that when it rains, the sprinkler is usually not turned on). Then the situation can be modeled with a Bayesian network.*

```
> r <- parray("rain", levels=list(yn), values=c(.2, .8))
> s.r <- parray(c("sprinkler","rain"), levels=list(yn,yn),
+               values=c(.01, .99, .4, .6))
> w.sr <- parray(c("wet","sprinkler","rain"), list(yn,yn,yn),
+               values=c(.99, .01, .8, .2, .9, .1, 0, 1))
> r
```

```
rain
  y   n
0.2 0.8

> s.r
          rain
sprinkler    y    n
        y 0.01 0.4
        n 0.99 0.6

> ftable(w.sr, col.vars = "wet")

                wet    y      n
sprinkler rain
y         y           0.99 0.01
          n           0.90 0.10
n         y           0.80 0.20
          n           0.00 1.00
```

The joint distribution can be obtained in different ways:

```
> joint <- tabMult( tabMult(r, s.r), w.sr )
> ftable(joint)

              rain       y        n
wet sprinkler
y   y              0.00198 0.28800
    n              0.15840 0.00000
n   y              0.00002 0.03200
    n              0.03960 0.48000

> ## Alternative
> joint <- tabListMult( list( r, s.r, w.sr ) )
```

What is the probability that it rains given that the grass is wet?

```
> wr.marg <- tabMarg(joint, ~wet+rain); wr.marg
   rain
wet      y      n
  y 0.16038 0.288
  n 0.03962 0.512

> tabDiv( wr.marg, tabMarg(wr.marg, ~wet))
   rain
wet          y           n
  y 0.35768768 0.6423123
  n 0.07182481 0.9281752

> ## Alternative -- and shorter
> tabCondProb(wr.marg, cond=~wet)
   rain
wet          y           n
  y 0.35768768 0.6423123
  n 0.07182481 0.9281752
```

Alternative computation

```
> x <- tabSliceMult(wr.marg, slice=list(wet="y")); x
   rain
wet      y      n
  y 0.16038 0.288
  n 0.00000 0.000

> tabMarg(x, ~rain)
rain
      y        n
0.16038 0.28800

> tabCondProb( tabMarg(x, ~rain) )
rain
        y         n
0.3576877 0.6423123
```

6   Example: Iterative Proportional Scaling (IPS)

Consider the two factor log–linear model for the `lizard` data. Under the model the expected counts have the form

$$\log m(d, h, s) = a_1(d, h) + a_2(d, s) + a_3(h, s)$$

If we let $n(d, h, s)$ denote the observed counts, the likelihood equations are: Find $m(d, h, s)$ such that

$$m(d, h) = n(d, h), \quad m(d, s) = n(d, s), \quad m(h, s) = n(h, s)$$

The updates are as follows: For the first term we have

$$m(d, h, s) \leftarrow m(d, h, s) \frac{n(d, h)}{m(d, h)}, \text{ where } m(d, h) = \sum_s m(d, h, s)$$

A rudimentary implementation of iterative proportional scaling for log–linear models is straight forward:

```
> myips <- function(indata, glist){
+     fit   <- indata
+     fit[] <-   1
+     ## List of sufficient marginal tables
+     md    <- lapply(glist, function(g) tabMarg(indata, g))
+
+     for (i in 1:4){
+         for (j in seq_along(glist)){
+             mf  <- tabMarg(fit, glist[[j]])
+             adj <- tabDiv( md[[j]], mf)
+             fit <- tabMult( fit, adj )
+         }
+     }
+     pearson=sum( (fit-indata)^2 / fit)
+     pearson
+ }
> glist<-list(c("species","diam"),c("species","height"),c("diam","height"))
> str( myips(lizard, glist), max.level=2)
 num 0.151
> str( loglin(lizard, glist), max.level = 2)
4 iterations: deviation 0.009618708
List of 4
 $ lrt    : num 0.149
 $ pearson: num 0.151
 $ df     : num 1
 $ margin :List of 3
  ..$ : chr [1:2] "species" "diam"
  ..$ : chr [1:2] "species" "height"
  ..$ : chr [1:2] "diam" "height"
```

7   Some low level functions

As an example we take the following:

```
> dim2222 <- c(2,2,2,2)
> dim2323 <- c(2,3,2,3)
```

7.1   cell2entry() and entry2cell()

The map from a cell to the corresponding entry is provided by `cell2entry()`. The reverse operation, going from an entry to a cell (which is much less needed) is provided by `entry2cell()`.

```
> cell2entry(c(1,1,1,1), dim2222)
[1] 1
> entry2cell(1, dim2222)
[1] 1 1 1 1
> cell2entry(c(2,1,2,1), dim2222)
[1] 6
> entry2cell(6, dim2222)
[1] 2 1 2 1
```

7.2 `nextCell()` and `nextCellSlice()`

Given a cell, say $i = (1, 1, 2, 1)$ we often want to find the next cell in the table following the convention that the first factor varies fastest, that is $(2, 1, 2, 1)$. This is provided by `nextCell()`.

```
> nextCell(c(1,1,2,1), dim2222)
[1] 2 1 2 1
> nextCell(c(2,2,2,1), dim2222)
[1] 1 1 1 2
```

Given $A \subset \Delta$ and a cell $i_A \in I_A$ consider the cells $I(i_A) = \{j \in I | j_A = i_A\}$. For example, the cells satisfying that dimension 2 is at level 1. Given such a cell, say $(2, 1, 1, 2)$ we often want to find the next cell also satisfying this constraint (again following the convention that the first factor varies fastest), that is $(1, 1, 2, 2)$. This is provided by `nextCellSlice()`.

```
> nextCellSlice(c(2,1,1,2),  sliceset=2, dim2323)
[1] 1 1 2 2
> nextCellSlice(c(1,3,2,1),  sliceset=c(2,3), dim2323)
[1] 2 3 2 1
```

7.3 `slice2entry()`

Given $A \subset \Delta$ and a cell $i_A \in I_A$. This cell defines a slice of the original array, namely the cells $I(i_A) = \{j \in I | j_A = i_A\}$. We often want to find the entries in x for the cells $I(i_A)$. This is provided by `slice2entry()`. For example, we may want the entries for the cells $(*, 1, 2, *)$ or $(2, 2, *, *)$:

```
> r1<-slice2entry(slicecell=c(1,2), sliceset=c(2,3), dim2222); r1
[1]   5   6 13 14
```

To verify that we indeed get the right cells:

```
> do.call(rbind, lapply(r1, entry2cell, dim2222))
     [,1] [,2] [,3] [,4]
[1,]    1    1    2    1
[2,]    2    1    2    1
[3,]    1    1    2    2
[4,]    2    1    2    2
```

7.4 `permuteCellEntries()`

In a 2×3 table, entries $1, \ldots, 6$ correspond to combinations $(1, 1), (2, 1), (1, 2), (2, 2), (1, 3), (2, 3)$. If we permute the table to a 3×2 table the entries become as follows:

```
> p<-permuteCellEntries(perm=c(2,1), dim=c(2,3)); p
[1] 1 3 5 2 4 6
```

So for example,

```
> (A <- array(11:16, dim=c(2,3)))

     [,1] [,2] [,3]
[1,]   11   13   15
[2,]   12   14   16
> Ap <- A[p]
> dim(Ap) <- c(3,2)
> Ap

     [,1] [,2]
[1,]   11   12
[2,]   13   14
[3,]   15   16
```

This corresponds to

```
> aperm(A, c(2,1))

     [,1] [,2]
[1,]   11   12
[2,]   13   14
[3,]   15   16
```

7.5 `factGrid()` − Factorial grid

Using the operations above we can obtain the combinations of the factors as a matrix:

```
> ff <- factGrid(dim2222)
> head(ff, 4)

     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    2    1    1    1
[3,]    1    2    1    1
[4,]    2    2    1    1
> tail(ff, 4)

      [,1] [,2] [,3] [,4]
[13,]    1    1    2    2
[14,]    2    1    2    2
[15,]    1    2    2    2
[16,]    2    2    2    2
```

This is the same as (but faster)

```
> aa <- expand.grid(list(1:2,1:2,1:2,1:2))
> head(aa, 4)

  Var1 Var2 Var3 Var4
1    1    1    1    1
2    2    1    1    1
3    1    2    1    1
4    2    2    1    1
```

There is a slice version as well:

```
> factGrid(dim2222, slicecell=c(1,2), sliceset=c(2,3))

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