Array operations in the gRbase package

Søren Højsgaard

gRbase version 1.8-1 as of 2016-10-16

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.*

2.1 Cross classified data - contingency tables

Arrays appear for example in connection with cross classified data. For example

```
> HairEyeColor
, , Sex = Male

       Eye
Hair    Brown Blue Hazel Green
  Black    32   11    10     3
  Brown    53   50    25    15
  Red      10   10     7     7
  Blond     3   30     5     8

, , Sex = Female

       Eye
Hair    Brown Blue Hazel Green
  Black    36    9     5     2
  Brown    66   34    29    14
  Red      16    7     7     7
  Blond     4   64     5     8
```

Data is a contingency table; a cross classified table of counts. In R lingo, data is a `table` object, but it is also an array because it has a `dim` attribute:

```
> class( HairEyeColor )
[1] "table"
> is.array( HairEyeColor )
[1] TRUE
> dim( HairEyeColor )
[1] 4 4 2
```

The array also has a `dimnames` attribute and the list of `dimnames` has names:

```
> dimnames( HairEyeColor )
$Hair
[1] "Black" "Brown" "Red"   "Blond"

$Eye
[1] "Brown" "Blue"  "Hazel" "Green"

$Sex
[1] "Male"   "Female"
```

Notice from the output above that the first variable (`Hair`) varies fastest. The `dimnames` attributes are important for many of the functions from `gRbase` described in the following sections.

Presence of named dimnames can be checked with `is.named.array()`[gRbase]

```
> is.named.array( HairEyeColor )
[1] TRUE
```

To limit output we shall only consider two hair colours and three eye colours.

```
> hec <- do.call("[", c(list(HairEyeColor), list(1:2, 1:3, TRUE), drop=FALSE))
> hec <- HairEyeColor[1:2, 1:3, ]
> hec
, , Sex = Male

       Eye
Hair    Brown Blue Hazel
  Black    32   11    10
  Brown    53   50    25

, , Sex = Female

       Eye
Hair    Brown Blue Hazel
  Black    36    9     5
  Brown    66   34    29
```

A more compact view of data can be achieved with `ftable()`. Since `gRbase` imports the pipe operator `%>%` from the `magrittr` package we will in this note do:

```
> flat <- function(x) {ftable(x, row.vars=1)}
> hec %>% flat
      Eye Brown        Blue        Hazel
      Sex  Male Female Male Female  Male Female
Hair
Black        32     36   11      9    10      5
Brown        53     66   50     34    25     29
```

2.2 Defining arrays

Arrays can be defined in different ways using standard `R` code:

```
> z1 <- c(32, 53, 11, 50, 10, 25, 36, 66, 9, 34, 5, 29)
> di <- c(2, 3, 2)
> dn <- list(Hair = c("Black", "Brown"),
+              Eye = c("Brown", "Blue", "Hazel"),
+              Sex = c("Male", "Female"))
> dim( z1 ) <- di
> dimnames( z1 ) <- dn
> z2 <- array( c(32, 53, 11, 50, 10, 25, 36, 66, 9, 34, 5, 29),
+               dim=di, dimnames=dn)
```

where the `dimnames` part in both cases is optional. Another way is to use `newar()`[gRbase] from `gRbase`:

```
> counts <- c(32, 53, 11, 50, 10, 25, 36, 66, 9, 34, 5, 29)
> z3 <- newar( ~ Hair:Eye:Sex, levels = dn, value = counts)
> z4 <- newar(c("Hair", "Eye", "Sex"), levels=dn, values=counts)
```

Notice that `dn` when used in `newar()`[gRbase] is allowed to contain superfluous elements. Default `dimnames` are generated with

```
> z5 <- newar(~Hair:Eye:Sex, levels=c(2, 3, 2), values = counts)
> z5 %>% flat
      Eye Eye1      Eye2      Eye3
      Sex Sex1 Sex2 Sex1 Sex2 Sex1 Sex2
Hair
Hair1       32   36   11    9   10    5
Hair2       53   66   50   34   25   29
```

Using `newar()`[gRbase], 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:

```
> z6 <- newar(~Hair:Eye:Sex, levels=c(2, 3, 2), values = counts, normalize="first")
> z6 %>% flat
      Eye      Eye1                Eye2                Eye3
      Sex      Sex1      Sex2      Sex1      Sex2      Sex1      Sex2
Hair
Hair1     0.3764706 0.3529412 0.1803279 0.2093023 0.2857143 0.1470588
Hair2     0.6235294 0.6470588 0.8196721 0.7906977 0.7142857 0.8529412
```

The same can be achived with`arnormalize()`[gRbase]

```
> arnormalize(z5, "first") %>% flat
      Eye      Eye1                Eye2                Eye3
      Sex      Sex1      Sex2      Sex1      Sex2      Sex1      Sex2
Hair
Hair1     0.3764706 0.3529412 0.1803279 0.2093023 0.2857143 0.1470588
Hair2     0.6235294 0.6470588 0.8196721 0.7906977 0.7142857 0.8529412
```

3 Operations on single arrays

In the following we shall denote the dimnames (or variables) of the array `hec` by H, E and S and we let (h, e, s) denote a configuration of these variables. The contingency table above shall be denoted by T_{HES} and we shall refer to the (h, e, s)-entry of T_{HES} as $T_{HES}(h, e, s)$.

3.1 Permuting an array

A reorganization of the table can be made with `arperm`[gRbase] (similar to `aperm()`), but `arperm`[gRbase] allows for a formula:

```
> arperm(hec, ~Eye:Sex:Hair) %>% flat
      Sex   Male        Female
      Hair Black Brown  Black Brown
Eye
Brown         32    53     36    66
Blue          11    50      9    34
Hazel         10    25      5    29
```

Alternative forms (that will also work for `aperm()`[gRbase]):

```
> arperm(hec, c("Eye", "Sex", "Hair")) %>% flat
> arperm(hec, c(2,3,1)) %>% flat
```

Notice that abbreviation is allowed

```
> arperm(hec, ~Ey:Se:Ha) %>% flat
> arperm(hec, c("Ey", "Se", "Ha")) %>% flat
```

3.2 Subsetting an array – slicing

We can subset arrays (this will also be called "slicing") in different ways. Notice that the result is not necessarily an array. Using standard R code we can do:

```
> hec[, 2:3, ]  %>% flat
```

```
      Eye Blue        Hazel
      Sex Male Female  Male Female
Hair
Black       11      9    10      5
Brown       50     34    25     29
```

```
> is.array( hec[, 2:3, ] )
```

```
[1] TRUE
```

```
> hec[1, , 1]
```

```
Brown  Blue Hazel
   32    11    10
```

```
> is.array( hec[1, , 1] )
```

```
[1] FALSE
```

Programmatically we can do the above as

```
> do.call("[", c(list(hec), list(TRUE, 2:3, TRUE)))  %>% flat
> do.call("[", c(list(hec), list(1, TRUE, 1)))  %>% flat
```

gRbase provides a wrapper for this, for example:

```
> arslice_prim(hec, slice=list(TRUE, 2:3, TRUE))  %>% flat
```

Using arslice()[gRbase] from gRbase the following are equivalent

```
> hec[, 2:3, ] %>% flat
```

```
      Eye Blue        Hazel
      Sex Male Female  Male Female
Hair
Black       11      9    10      5
Brown       50     34    25     29
```

```
> arslice(hec, slice=list(c(2,3)), margin=2) %>% flat
```

```
      Eye Blue        Hazel
      Sex Male Female  Male Female
Hair
Black       11      9    10      5
Brown       50     34    25     29
```

Alternative forms:

```
> arslice(hec, slice=list(Eye=2:3)) %>% flat
> arslice(hec, slice=list(Eye=c("Blue","Hazel")))  %>% flat
```

The virtue of arslice[gRbase] comes from the flexibility when specifying the slice:[1]

```
> arslice(hec, slice=list(Eye=c(2,3), Sex="Female")) %>% flat
```

```
      Eye Blue Hazel
Hair
Black        9     5
Brown       34    29
```

```
> arslice(hec, slice=list(Eye=c(2,3), Sex="Female"), drop=FALSE) %>% flat
```

[1] Currently names can not be abbreviated, but that might be added later.

```
      Eye   Blue  Hazel
      Sex Female Female
Hair
Black          9      5
Brown         34     29
```

If slicing leads to a one dimensional array, the output will by default not be an array but a vector (without a dim attribute)

```
> z <- arslice(hec, slice=list(Hair=1, Sex="Female")); z
```

```
Brown  Blue Hazel
   36     9     5
```

```
> is.array( z )
```

```
[1] FALSE
```

The output can be forced to be an array with

```
> z <- arslice(hec, slice=list(Hair=1, Sex="Female"), as.array=TRUE); z
```

```
Eye
Brown  Blue Hazel
   36     9     5
```

```
> is.array( z )
```

```
[1] TRUE
```

3.3 Collapsing arrays

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

$$T_{HE}(h,e) = \sum_s T_{HES}(h,e,s)$$

With gRbase we can do[2]:

```
> he <- armarg(hec, ~Hair:Eye); he %>% flat
```

```
      Eye Brown Blue Hazel
Hair
Black        68   20    15
Brown       119   84    54
```

Alternative forms include

```
> hs <- armarg(hec, c("Hair","Sex")); hs
> es <- armarg(hec, c(2,3)); es
```

3.4 Inflating arrays

The "opposite" operation is to extend an array. For example, we can extend T_{HE} to have a third dimension, e.g. Sex. That is

$$\tilde{T}_{HES}(h,e,s) = T_{HE}(h,e)$$

so $\tilde{T}_{HES}(h,e,s)$ is constant as a function of s. With gRbase this is done with arexpand() [gRbase]:

```
> arexpand(he, list(Sex=c("Male", "Female"))) %>% flat
```

```
       Eye  Brown        Blue        Hazel
       Hair Black Brown Black Brown Black Brown
Sex
Male           68   119    20    84    15    54
Female         68   119    20    84    15    54
```

[2]FIXME: Should allow for abbreviations in formula and character vector specifications.

Notice that the added dimensions come "at the end". The following versions produce the same result:

```
> arexpand(he, dimnames(hs)) %>% flat
> arexpand(he, hs) %>% flat
```

4 Operations on two or more arrays

4.1 Multiplication, addition etc: $+$, $-$, $*$, $/$

The product of two arrays T_{HE} and T_{HS} is defined to be the array $\tilde{T}_{HES}$ with entries

$$\tilde{T}_{HES}(h, e, s) = T_{HE}(h, e)T_{HS}(h, s)$$

With `gRbase` this is done with `armult()`[gRbase]:

```
> armult(he, hs) %>% flat
      Sex  Male              Female
      Eye Brown  Blue Hazel  Brown  Blue Hazel
Hair
Black      3604  1060   795   3400  1000   750
Brown     15232 10752  6912  15351 10836  6966
```

The quotient, sum and difference is defined similarly:

```
> ardiv(he, hs)  %>% flat
      Sex      Male                          Female
      Eye      Brown       Blue      Hazel      Brown       Blue      Hazel
Hair
Black     1.2830189 0.3773585 0.2830189 1.3600000 0.4000000 0.3000000
Brown     0.9296875 0.6562500 0.4218750 0.9224806 0.6511628 0.4186047
```

```
> aradd(he, hs)  %>% flat
      Sex  Male             Female
      Eye Brown Blue Hazel  Brown Blue Hazel
Hair
Black       121   73    68    118   70    65
Brown       247  212   182    248  213   183
```

```
> arsubt(he, hs) %>% flat
      Sex  Male             Female
      Eye Brown Blue Hazel  Brown Blue Hazel
Hair
Black        15  -33   -38     18  -30   -35
Brown        -9  -44   -74    -10  -45   -75
```

Multiplication and addition of a list of arrays is accomplished with `arprod()`[gRbase] and `arsum()`[gRbase]:

```
> arprod( he, hs, es ) %>% flat
      Sex     Male           Female
      Hair   Black   Brown   Black   Brown
Eye
Brown       306340 1294720  346800 1565802
Blue         64660  655872   43000  465948
Hazel        27825  241920   25500  236844
```

```
> arsum( he, hs, es ) %>% flat
```

```
      Sex   Male       Female
      Hair Black Brown  Black Brown
Eye
Brown        206   332    220   350
Blue         134   273    113   256
Hazel        103   217     99   217
```

4.2 Miscellaneous

Two arrays are defined to be identical 1) if they have the same dimnames and 2) if, possibly after a permutation, all values are identical (up to a small numerical difference):

```
> hec2 <- arperm(hec, 3:1)
> arequal(hec, hec2)
```

```
[1] TRUE
```

A visual comparison of the entries of two arrays with the same dimnames is much easier if the dimnames are in the same order. For example, the following provides the fitted cell counts under a specific log–linear model:

```
> hec3 <- ardiv( armult( he, es ), armarg( hec, "Eye" ) )
> hec3 %>% flat
```

```
      Sex       Male                Female
      Hair     Black     Brown     Black     Brown
Eye
Brown      30.909091 54.090909 37.090909 64.909091
Blue       11.730769 49.269231  8.269231 34.730769
Hazel       7.608696 27.391304  7.391304 26.608696
```

Comparing these with the observed data is tricky because of the ordering:

```
> hec %>% flat
```

```
      Eye Brown        Blue        Hazel
      Sex  Male Female Male Female  Male Female
Hair
Black        32     36   11      9    10      5
Brown        53     66   50     34    25     29
```

The function `aralign()`[gRbase] will align the first array to have the same variable order as the second array which makes a visual comparison easier:[3]

```
> aralign(hec3, hec)  %>% flat
```

```
      Eye     Brown                Blue                 Hazel
      Sex       Male    Female      Male    Female       Male    Female
Hair
Black     30.909091 37.090909 11.730769  8.269231  7.608696  7.391304
Brown     54.090909 64.909091 49.269231 34.730769 27.391304 26.608696
```

5 Examples

5.1 A Bayesian network

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

[3]FIXME: aralign should be modified so that the second argument can also be a list of dimnames

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.

We specify conditional probabilities $p(r)$, $p(s|r)$ and $p(w|s,r)$ as follows (notice that the vertical conditioning bar (|) is replaced by the horizontal underscore:

```
> yn <- c("y","n")
> lev <- list(rain=yn, sprinkler=yn, wet=yn)
> r <- newar( ~rain, levels = lev, values = c(.2, .8) )
> s_r <- newar( ~sprinkler:rain, levels = lev, values = c(.01,.99, .4, .6) )
> w_sr <- newar( ~wet:sprinkler:rain, levels = lev,
+              values = c(.99, .01, .8, .2, .9, .1, 0, 1))
> r
```

```
rain
  y   n
0.2 0.8
```

```
> s_r  %>% flat
```

```
          rain    y    n
sprinkler
y              0.01 0.40
n              0.99 0.60
```

```
> w_sr %>% flat
```

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

The joint distribution $p(r,s,w) = p(r)p(s|r)p(w|s,r)$ can be obtained with **arprod()**[gRbase]: ways:

```
> joint <- arprod( r, s_r, w_sr ); joint %>% flat
```

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

What is the probability that it rains given that the grass is wet? We find $p(r,w) = \sum_s p(r,s,w)$ and then $p(r|w) = p(r,w)/p(w)$ with **ardist()**[gRbase]

```
> rw <- armarg(joint, ~rain+wet)
> ardist(rw, cond=~wet)
```

```
    wet
rain         y          n
   y 0.3576877 0.07182481
   n 0.6423123 0.92817519
```

```
> ## Alternative:
> ardiv( rw, armarg(rw, ~wet))
> ## or
> rw %a/% (rw %am% ~wet)
```

Alternative computation using

```
> x <- arslice_mult(rw, slice=list(wet="y")); x
```

```
    wet
rain       y n
```

```
   y 0.16038 0
   n 0.28800 0
```

```
> p <- armarg(x, ~rain); p  ## Unnormalized
```

```
rain
      y       n
0.16038 0.28800
```

```
> ardist( p )           ## Normalized
```

```
rain
        y         n
0.3576877 0.6423123
```

5.2 Iterative Proportional Scaling (IPS)

We consider the 3–way `lizard` data from `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
```

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

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

where $m(d,h) = \sum_s m(d,h.s)$ etc. The updates are as follows: For the first term we have

$$m(d,h,s) \leftarrow m(d,h,s)\frac{n(d,h)}{m(d,h)}$$

After iterating the updates will not change and we will have equality: $m(d,h,s) = m(d,h,s)\frac{n(d,h)}{m(d,h)}$ and summing over s shows that the equation $m(d,h) = n(d,h)$ is satisfied.

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) armarg(indata, g))
+
+     for (i in 1:4){
+         for (j in seq_along(glist)){
+             mf  <- armarg(fit, glist[[j]])
+             ## adj <- ardiv( md[[ j ]], mf)
+             ## fit <- armult( fit, adj )
+             ## or
+             adj <- md[[ j ]] %a/% mf
+             fit <- fit %a*% adj
+         }
+     }
+     pearson <- sum( (fit-indata)^2 / fit)
+     list(pearson=pearson, fit=fit)
+ }
> glist <- list(c("species","diam"),c("species","height"),c("diam","height"))
> fm1 <- myips( lizard, glist )
> fm1$pearson
```

```
[1] 0.1505859
```

```
> fm1$fit %>% flat
```

```
     height     >4.75           <=4.75
     species    anoli     dist    anoli     dist
diam
<=4          32.79764 60.20236 85.20155 73.79845
>4           10.20273 41.79727 35.79797 69.20203
```

```
> fm2 <- loglin( lizard, glist, fit=T )
```

```
4 iterations: deviation 0.009618708
```

```
> fm2$pearson
```

```
[1] 0.1505859
```

```
> fm2$fit %>% flat
```

```
     height     >4.75           <=4.75
     species    anoli     dist    anoli     dist
diam
<=4          32.79764 60.20236 85.20155 73.79845
>4           10.20273 41.79727 35.79797 69.20203
```

6 Some low level functions

For e.g. a $2 \times 3 \times 2$ array, the entries are such that the first variable varies fastest so the ordering of the cells are $(1, 1, 1)$, $(2, 1, 1)$, $(1, 2, 1)$, $(2, 2, 1)$,$(1, 3, 1)$ and so on. To find the value of such a cell, say, (j, k, l) in the array (which is really just a vector), the cell is mapped into an entry of a vector.

For example, cell $(2, 3, 1)$ (`Hair=Brown`,`Eye=Hazel`,`Sex=Male`) must be mapped to entry 4 in

```
> hec
```

```
, , Sex = Male

       Eye
Hair    Brown Blue Hazel
  Black    32   11    10
  Brown    53   50    25

, , Sex = Female
```

```
       Eye
Hair    Brown Blue Hazel
  Black    36    9     5
  Brown    66   34    29
```

```
> c(hec)
```

```
 [1] 32 53 11 50 10 25 36 66  9 34  5 29
```

For illustration we do:

```
> cell2name <- function(cell, dimnames){
+     unlist(lapply(1:length(cell), function(m) dimnames[[m]][cell[m]]))
+ }
> cell2name(c(2,3,1), dimnames(hec))
```

```
[1] "Brown" "Hazel" "Male"
```

6.1 `cell2entry()`, `entry2cell()` and `nextCell()`

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

```
> cell2entry(c(2,3,1), dim=c( 2, 3, 2 ))
```

```
[1] 6
```

```
> entry2cell(6, dim=c( 2, 3, 2 ))
```

```
[1] 2 3 1
```

Given a cell, say $i = (2, 3, 1)$ in a $2 \times 3 \times 2$ array we often want to find the next cell in the table following the convention that the first factor varies fastest, that is $(1, 1, 2)$. This is provided by **nextCell()**[gRbase].

```
> nextCell(c(2,3,1), dim=c( 2, 3, 2 ))
```

```
[1] 1 1 2
```

6.2 `nextCellSlice()` and `slice2entry()`

Given that we look at cells for which for which the index in dimension 2 is at level 3 (that is `Eye=Hazel`), i.e. cells of the form $(j, 3, l)$. Given such a cell, what is then the next cell that also satisfies this constraint. This is provided by `nextCellSlice()`[gRbase].[4]

```
> nextCellSlice(c(1,3,1), sliceset=2, dim=c( 2, 3, 2 ))
```

```
[1] 2 3 1
```

```
> nextCellSlice(c(2,3,1), sliceset=2, dim=c( 2, 3, 2 ))
```

```
[1] 1 3 2
```

Given that in dimension 2 we look at level 3. We want to find entries for the cells of the form $(j, 3, l)$.[5]

```
> slice2entry(slicecell=3, sliceset=2, dim=c( 2, 3, 2 ))
```

```
[1]  5  6 11 12
```

To verify that we indeed get the right cells:

[4]FIXME: sliceset should be called margin.

[5]FIXME:slicecell and sliceset should be renamed

```
> r <- slice2entry(slicecell=3, sliceset=2, dim=c( 2, 3, 2 ))
> lapply(lapply(r, entry2cell, c( 2, 3, 2 )),
+        cell2name, dimnames(hec))
```

```
[[1]]
[1] "Black" "Hazel" "Male"

[[2]]
[1] "Brown" "Hazel" "Male"

[[3]]
[1] "Black"  "Hazel"  "Female"

[[4]]
[1] "Brown"  "Hazel"  "Female"
```

6.3 factGrid() – Factorial grid

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

```
> head( factGrid( c(2, 3, 2) ), 6 )
```

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

A similar dataframe can also be obtained with the standard R function expand.grid (but factGrid is faster)

```
> head( expand.grid(list(1:2,1:3,1:2)), 6 )
```

```
  Var1 Var2 Var3
1    1    1    1
2    2    1    1
3    1    2    1
4    2    2    1
5    1    3    1
6    2    3    1
```