Array operations in the `gRbase` package

Søren Højsgaard

`gRbase` version 1.8-4.5 as of 2019-09-25

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. The array `hec` below is an excerpt of the `HairEyeColor` array in R:

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

Above, `hec` is an array because it has a `dim` attribute. Moreover, `hec` also has a `dimnames` attribute naming the levels of each dimension. Notice that each dimension is given a name.

An array with named dimensions is in this package called a *named array*; this can be checked with `is.named.array()`[gRbase]

```
is.named.array( hec )
## [1] TRUE
```

The functionality described below relies heavily on arrays having named dimensions.

Printing arrays takes up a lot of space. 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 `ar_new()`[gRbase] from `gRbase`:

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

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

```
z5 <- ar_new(~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 `ar_new()`[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 <- ar_new(~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     0.3765 0.3529 0.1803 0.2093 0.2857 0.1471
## Hair2     0.6235 0.6471 0.8197 0.7907 0.7143 0.8529
```

3 Operations on 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 Normalizing an array

Normalize an array with `ar_normalize()`[gRbase]

```
ar_normalize(z5, "first") %>% flat
##       Eye   Eye1        Eye2        Eye3
##       Sex   Sex1   Sex2   Sex1   Sex2   Sex1   Sex2
## Hair
## Hair1     0.3765 0.3529 0.1803 0.2093 0.2857 0.1471
## Hair2     0.6235 0.6471 0.8197 0.7907 0.7143 0.8529

ar_normalize(z5, "all") %>% flat
##       Eye    Eye1          Eye2          Eye3
##       Sex    Sex1    Sex2    Sex1    Sex2    Sex1    Sex2
## Hair
## Hair1     0.08889 0.10000 0.03056 0.02500 0.02778 0.01389
## Hair2     0.14722 0.18333 0.13889 0.09444 0.06944 0.08056
```

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. Slicing can be done using standard `R` code or using `ar_slice`[gRbase]. The virtue of `ar_slice`[gRbase] comes from the flexibility when specifying the slice:

The following leads from the original $2 \times 3 \times 2$ array to a $2 \times 2 \times 2$ array by cutting away the `Eye=Brown` slice of the array:

```
ar_slice(hec, slice=list(Eye=c("Blue", "Hazel")))  %>% flat
##       Eye Blue        Hazel
##       Sex Male Female  Male Female
## Hair
## Black       11      9    10      5
## Brown       50     34    25     29
```

Levels can be written as numerics.[1]

```
ar_slice(hec, slice=list(Eye=2:3, Sex="Female"))
```

Suppose we pick the `Sex=Female` slice of `hec`. This slice can be regarded as a 2×3 array or as $2 \times 3 \times 1$ array.

```
# 2 x 3 array :
ar_slice(hec, slice=list(Sex="Female")) %>% flat
```

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

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

# 2 x 3 x 1 array :
ar_slice(hec, slice=list(Sex="Female"), drop=FALSE) %>% flat
##       Eye  Brown   Blue  Hazel
##       Sex Female Female Female
## Hair
## Black         36      9      5
## Brown         66     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). However, the result can be forced to be a 1–dimensional array:

```
## A vector:
z <- ar_slice(hec, slice=list(Hair=1, Sex="Female")); z
## A 1-dimensional array:
z <- ar_slice(hec, slice=list(Hair=1, Sex="Female"), as.array=TRUE); z
```

Slicing using standard `R` code can be done as follows:

```
hec[, 2:3, ]  %>% flat   ## A 2 x 2 x 2 array
##       Eye Blue        Hazel
##       Sex Male Female  Male Female
## Hair
## Black       11      9    10      5
## Brown       50     34    25     29

hec[1, , 1]              ## A vector
## Brown  Blue Hazel
##    32    11    10

hec[1, , 1, drop=FALSE] ## A 1 x 3 x 1 array
## , , Sex = Male
##
##        Eye
## Hair    Brown Blue Hazel
##   Black    32   11    10
```

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)))
do.call("[", c(list(hec), list(1, TRUE, 1), drop=FALSE))
```

`gRbase` provides two alterntives for each of these three cases above:

```
ar_slice_prim(hec, slice=list(TRUE, 2:3, TRUE))  %>% flat
ar_slice(hec, slice=list(c(2, 3)), margin=2) %>% flat
```

```
ar_slice_prim(hec, slice=list(1, TRUE, 1))
ar_slice(hec, slice=list(1, 1), margin=c(1,3))

ar_slice_prim(hec, slice=list(1, TRUE, 1), drop=FALSE)
ar_slice(hec, slice=list(1, 1), margin=c(1,3), drop=FALSE)
```

3.3 Collapsing and inflating arrays

Collapsing: The HE–marginal array T_{HE} of T_{HES} is the array with values

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

Inflating: 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

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

so $\tilde{T}_{SHE}(s,h,e)$ is constant as a function of s.

With `gRbase` we can collapse with[2]:

```
he <- hec %a_% ~Hair:Eye; he %>% flat
##       Eye Brown Blue Hazel
## Hair
## Black         68   20    15
## Brown        119   84    54
```

```
## Alternatives
he <- ar_marg(hec, ~Hair:Eye); he
hs <- ar_marg(hec, c("Hair", "Sex"))
es <- ar_marg(hec, c(2, 3))
```

With `gRbase` we can inflate with `ar_expand()`[gRbase]:

```
she <- he %a^% list(Sex=c("Male", "Female"))
she %>% 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
```

```
## Alternatives
she <- ar_expand(he, list(Sex=c("Male", "Female")))
ar_expand(he, dimnames(hs)) %>% flat
ar_expand(he, hs) %>% flat
```

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

3.4 Permuting an array

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

```
ar_perm(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 (the first two also works for `aperm`):

```
ar_perm(hec, c("Eye", "Sex", "Hair"))
ar_perm(hec, c(2,3,1))
ar_perm(hec, ~Ey:Se:Ha)
ar_perm(hec, c("Ey", "Se", "Ha"))
```

3.5 Equality

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 <- ar_perm(hec, 3:1)
hec %a==% hec2
## [1] TRUE
```

```
## Alternative
ar_equal(hec, hec2)
```

3.6 Aligning

We can align one array according to the ordering of another:[3]

```
hec2 <- ar_perm(hec, 3:1)
ar_align(hec2, hec)
## ar_align(hec2, dimnames(hec))
## ar_align(hec2, names(dimnames(hec)))
```

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

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

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

[3] FIXME; see `ar_expand()`

The difference, product and quotient is defined similarly:

With gRbase this is done with ar_mult()[gRbase]:

```
she <- he %a+% hs
she %>% 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
```

Likewise

```
he %a+% hs
he %a-% hs
he %a*% hs
he %a/% hs
he %a/0% hs ## Convention 0/0 = 0
```

```
ar_add(he, hs)  %>% flat
ar_subt(he, hs) %>% flat
ar_mult(he, hs) %>% flat
ar_div(he, hs)  %>% flat
ar_div0(he, hs)  %>% flat ## Convention 0/0 = 0
```

Multiplication and addition of a list of multiple arrays is accomplished with ar_prod()[gRbase] and ar_sum()[gRbase] (much like prod()[gRbase] and sum()[gRbase]):

```
ar_sum( he, hs, es )
ar_prod( he, hs, es )
```

Lists of arrays are processed with

```
ar_sum_list( list(he, hs, es) )
ar_prod_list( list(he, hs, es) )
```

3.8 An array as a probability density

If an array consists of non–negative numbers then it may be regarded as an (unnormalized) discrete multivariate density. With this view, the following examples should be self explanatory:

```
ar_dist(hec) %>% flat
##       Eye   Brown             Blue            Hazel
##       Sex    Male  Female    Male  Female    Male  Female
## Hair
## Black     0.08889 0.10000 0.03056 0.02500 0.02778 0.01389
## Brown     0.14722 0.18333 0.13889 0.09444 0.06944 0.08056

ar_dist(hec, marg=~Hair:Eye) %>% flat
```

```
##        Eye   Brown    Blue   Hazel
## Hair
## Black     0.18889 0.05556 0.04167
## Brown     0.33056 0.23333 0.15000

ar_dist(hec, cond=~Eye) %>% flat
##        Sex    Male                  Female
##        Eye   Brown    Blue   Hazel   Brown    Blue   Hazel
## Hair
## Black     0.17112 0.10577 0.14493 0.19251 0.08654 0.07246
## Brown     0.28342 0.48077 0.36232 0.35294 0.32692 0.42029

ar_dist(hec, marg=~Hair, cond=~Sex) %>% flat
##        Sex   Male Female
## Hair
## Black     0.2928 0.2793
## Brown     0.7072 0.7207
```

3.9 Miscellaneous

Multiply values in a slice by some number and all other values by another number:

```
ar_slice_mult(hec, list(Sex="Female"), val=10, comp=0) %>% flat
##        Eye Brown        Blue        Hazel
##        Sex  Male Female Male Female  Male Female
## Hair
## Black          0    360    0     90     0     50
## Brown          0    660    0    340     0    290
```

4 Examples

4.1 A Bayesian network

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

> *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 <- ar_new( ~rain, levels = lev, values = c(.2, .8) )
s_r <- ar_new( ~sprinkler:rain, levels = lev, values = c(.01, .99, .4, .6) )
w_sr <- ar_new( ~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 **ar_prod()**[gRbase]: ways:

```
joint <- ar_prod( 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 $p(r|w) = p(r, w)/p(w)$. Can be done in various ways: with **ar_dist()**[gRbase]

```
ar_dist(joint, marg=~rain, cond=~wet)
##     wet
## rain      y       n
##    y 0.3577 0.07182
##    n 0.6423 0.92818
```

```
## Alternative:
rw <- ar_marg(joint, ~rain + wet)
ar_div( rw, ar_marg(rw, ~wet))
## or
rw %a/% (rw %a_% ~wet)
```

```
## Alternative:
x <- ar_slice_mult(rw, slice=list(wet="y")); x
##     wet
## rain      y n
##    y 0.1604 0
##    n 0.2880 0
```

```
ar_dist(x, marg=~rain)
## rain
##      y      n
## 0.3577 0.6423
```

4.2 Iterative Proportional Scaling (IPS)

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

```
data( lizard, package="gRbase" )
lizard %>% flat
##      height  >4.75      <=4.75
##      species anoli dist  anoli dist
## diam
## <=4             32   61     86   73
## >4              11   41     35   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) ar_marg(indata, g))

    for (i in 1:4){
        for (j in seq_along(glist)){
            mf  <- ar_marg(fit, glist[[j]])
            # adj <- ar_div( md[[ j ]], mf)
            # fit <- ar_mult( 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] 0.1506

fm1$fit %>% flat
##      height  >4.75      <=4.75
##      species anoli dist  anoli dist
## diam
## <=4           32.8 60.2   85.2 73.8
## >4            10.2 41.8   35.8 69.2

fm2 <- loglin( lizard, glist, fit=T )
## 4 iterations: deviation 0.009619

fm2$pearson
## [1] 0.1506

fm2$fit %>% flat
##      height  >4.75      <=4.75
##      species anoli dist  anoli dist
## diam
## <=4           32.8 60.2   85.2 73.8
## >4            10.2 41.8   35.8 69.2
```

5 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"
```

5.1 cell2entry(), entry2cell() and next_cell()

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 `next_cell()`[gRbase].

```
next_cell(c(2,3,1), dim=c( 2, 3, 2 ))
## [1] 1 1 2
```

5.2 next_cell_slice() 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 `next_cell_slice()`[gRbase].[4]

```
next_cell_slice(c(1,3,1), slice_marg=2, dim=c( 2, 3, 2 ))
## [1] 2 3 1

next_cell_slice(c(2,3,1), slice_marg=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]

[4]FIXME: sliceset should be called margin.
[5]FIXME:slicecell and sliceset should be renamed

```
slice2entry(slice_cell=3, slice_marg=2, dim=c( 2, 3, 2 ))
## [1]  5  6 11 12
```

To verify that we indeed get the right cells:

```
r <- slice2entry(slice_cell=3, slice_marg=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"
```

5.3 `fact_grid()` – Factorial grid

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

```
head( fact_grid( 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
```