Array operations in the gRbase package

Søren Højsgaard

gRbase version 1.8.7 as of 2022-04-02

Contents

1 Introduction 2

2 Arrays/tables in R 2
   2.1 Cross classified data - contingency tables 2
   2.2 Defining arrays 3

3 Operations on arrays 3
   3.1 Normalizing an array 3
   3.2 Subsetting an array - slicing 4
   3.3 Collapsing and inflating arrays 5
   3.4 Permuting an array 6
   3.5 Equality 6
   3.6 Aligning 6
   3.7 Multiplication, addition etc: +, −, ∗, / 7
   3.8 An array as a probability density 8
   3.9 Miscellaneous 8

4 Examples 9
   4.1 A Bayesian network 9
   4.2 Iterative Proportional Scaling (IPS) 10

5 Some low level functions 12
   5.1 cell2entry(), entry2cell() and next_cell() 12
   5.2 next_cell_slice() and slice2entry() 13
   5.3 fact_grid() - Factorial grid 13

A More about slicing 14
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 these 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:

```r
hec <- c(32, 53, 11, 50, 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.

Printing arrays can take up a lot of space. Alternative views on an array can be obtained with `ftable()` or by converting the array to a dataframe with `as.data.frame.table()`. We shall do so in the following.

```r
flat <- function(x) {ftable(x, row.vars=1)}
flat <- function(x, n=4) {as.data.frame.table(x) %>% head(n)}
hec %>% flat
## Hair Eye Sex Freq
## 1 Black Brown Male   32
## 2 Brown Brown Male  53
## 3 Black Blue Male   11
## 4 Brown Blue Male  50
```
An array with named dimensions is in this package called a named array. The functionality described below relies heavily on arrays having named dimensions. A check for an object being a named array is provided by `is.named.array()`

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

### 2.2 Defining arrays

Another way is to use `tabNew()` from `gRbase`. This function is flexible wrt the input; for example:

```r
dn <- list(Hair=c("Black", "Brown"), Eye="Brown:Blue:Hazel", Sex="Male:Female")
counts <- c(32, 53, 11, 50, 10, 25, 36, 66, 9, 34, 5, 29)
z3 <- tabNew("Hair:Eye:Sex", levels=dn, value=counts)
z4 <- tabNew(c("Hair", "Eye", "Sex"), levels=dn, values=counts)
```

Notice that the levels list (`dn`) when used in `tabNew()` is allowed to contain superfluous elements. Default dimnames are generated with

```r
z5 <- tabNew("Hair:Eye:Sex", levels=c(2, 3, 2), values = counts)
dimnames(z5) %>% str
## List of 3
## $ Hair: chr [1:2] "Hair1" "Hair2"
## $ Eye : chr [1:3] "Eye1" "Eye2" "Eye3"
## $ Sex : chr [1:2] "Sex1" "Sex2"
```

Using `tabNew()`, arrays can be normalized to sum to one in two ways: 1) Normalization can be over the first variable for each configuration of all other variables and 2) over all configurations. For example:

```r
z6 <- tabNew("Hair:Eye:Sex", levels=c(2, 3, 2), values=counts, normalize="first")
z6 %>% flat
## Hair Eye Sex Freq
## 1 Hair1 Eye1 Sex1 0.3765
## 2 Hair2 Eye1 Sex1 0.6235
## 3 Hair1 Eye2 Sex1 0.1803
## 4 Hair2 Eye2 Sex1 0.8197
```

### 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 `tabNormalize()` Entries of an array can be normalized to sum to one in two ways: 1) Normalization can be over the first variable for each configuration of all other variables and 2) over all configurations. For example:
### 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 `tabSlice`[grbase]. The virtue of `tabSlice`[grbase] comes from the flexibility when specifying the slice:

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

```r
tabSlice(hec, slice=list(Eye=c("Blue", "Hazel"), Sex="Female"))
```

```r
## Notice: levels can be written as numerics
## tabSlice(hec, slice=list(Eye=2:3, Sex="Female"))
```

We may also regard the result above as a $2 \times 2 \times 1$ array:

```r
tabSlice(hec, slice=list(Eye=c("Blue", "Hazel"), Sex="Female"), drop=FALSE)
```

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:

```r
## A vector:
t1 <- tabSlice(hec, slice=list(Hair=1, Sex="Female")); t1
## A 1-dimensional array:
t2 <- tabSlice(hec, slice=list(Hair=1, Sex="Female"), as.array=TRUE); t2
## A higher dimensional array (in which some dimensions only have one level)
t3 <- tabSlice(hec, slice=list(Hair=1, Sex="Female"), drop=FALSE); t3
```

The difference between the last two forms can be clarified:

```r
t2 %>% flat
```

```r
## Eye Freq
## 1 Brown 36
## 2 Blue 9
## 3 Hazel 5
```

```r
t3 %>% flat
```

```r
## Hair Eye Sex Freq
## 1 Black Brown Female 36
## 2 Black Blue Female 9
## 3 Black Hazel Female 5
```
### 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 arrays with\(^1\):

```r
he <- tabMarg(hec, c("Hair", "Eye"))
he
## Hair Brown Blue Hazel
## Black 68 20 15
## Brown 119 84 54
```

```
## alternatives
tabMarg(hec, "Hair:Eye")
tabMarg(hec, c(1, 2))
hec %a_% "Hair:Eye"
```

Notice that collapsing is a projection in the sense that applying the operation again does not change anything:

```r
he1 <- tabMarg(hec, c("Hair", "Eye"))
he2 <- tabMarg(he1, c("Hair", "Eye"))
tabEqual(he1, he2)
## [1] TRUE
```

Expand an array by adding additional dimensions with `tabExpand()`\([gRbase]\):

```r
extra.dim <- list(Sex=c("Male", "Female"))
tabExpand(he, extra.dim)
## , , Sex = Male
##   ## Hair
##   ## Eye Black Brown
##   ## Brown 68 119
##   ## Blue 20 84
##   ## Hazel 15 54
## , , Sex = Female
##   ## Hair
##   ## Eye Black Brown
##   ## Brown 68 119
##   ## Blue 20 84
##   ## Hazel 15 54
```

\(^1\)FIXME: Should allow for abbreviations in formula and character vector specifications.
# Alternatives

Notice that expanding and collapsing brings us back to where we started:

```r
(hec %a^% extra.dim) %a_% c("Hair", "Eye")
## Eye
## Hair Brown Blue Hazel
## Black 136 40 30
## Brown 238 168 108
```

## 3.4 Permuting an array

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

```r
tabPerm(hec, ~Eye:Sex:Hair) %>% flat
## Eye Sex Hair Freq
## 1 Brown Male Black 32
## 2 Blue Male Black 11
## 3 Hazel Male Black 10
## 4 Brown Female Black 36
```

Alternative forms (the first two also work for `aperm()`):

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

```r
hec2 <- tabPerm(hec, 3:1)
tabEqual(hec, hec2)
## [1] TRUE
```

## 3.6 Aligning

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

```r
hec2 <- tabPerm(hec, 3:1)
tabAlign(hec2, hec)
```
## Alternative:
```
tabAlign(hec2, dimnames(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
```

### 3.7 Multiplication, addition etc: +, −, ∗, /

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

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

The sum, difference and quotient is defined similarly: This is done with `tabProd()` $\text{gRbase}$, `tabAdd()` $\text{gRbase}$, `tabDiff()` $\text{gRbase}$ and `tabDiv()` $\text{gRbase}$:

```
hs <- tabMarg(hec, ~Hair:Eye)
tabMult(he, hs)
```
```
## , , Sex = Male
##
## Eye
## Hair Brown Blue Hazel
## Black 4624 400 225
## Brown 14161 7056 2916
```

Available operations:

```
tabAdd(he, hs)
tabSubt(he, hs)
tabMult(he, hs)
tabDiv(he, hs)
tabDiv0(he, hs)  ## Convention 0/0 = 0
```

Shortcuts:

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

Multiplication and addition of (a list of) multiple arrays is accomplished with `tabProd()` $\text{gRbase}$ and `tabSum()` $\text{gRbase}$ (much like `prod()` $\text{gRbase}$ and `sum()` $\text{gRbase}$):

```
es <- tabMarg(hec, ~Eye:Sex)
tabSum(he, hs, es)
```
### 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:

```
## Sex = Male
##
## Eye
## Hair Brown Blue Hazel
## Black 221 101 65
## Brown 323 229 143
##
## Sex = Female
##
## Eye
## Hair Brown Blue Hazel
## Black 238 83 64
## Brown 340 211 142

## tabSum(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:

```
# A probability density example with tabDist

## tabDist(hec, marg=^Hair:Eye)
##
## Hair Brown Blue Hazel
## Black 0.1889 0.05556 0.04167
## Brown 0.3306 0.23333 0.15000

## tabDist(hec, cond=^Sex)
##
## Sex = Male
##
## Eye
## Hair Brown Blue Hazel
## Black 0.1768 0.06077 0.05525
## Brown 0.2928 0.27624 0.13812
##
## Sex = Female
##
## Eye
## Hair Brown Blue Hazel
## Black 0.2011 0.05028 0.02793
## Brown 0.3687 0.18994 0.16201

## tabDist(hec, marg=^Hair, cond=^Sex)
##
## Sex
## Hair Male Female
## Black 0.2028 0.02793
## Brown 0.7072 0.7207
```

### Miscellaneous

Multiply values in a slice by some number and all other values by another number:
tabSliceMult(es, list(Sex="Female"), val=10, comp=0)

## Sex
## Eye Male Female
## Brown 0 10 20
## Blue 0 4 3 0
## Hazel 0 3 4 0

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):

```r
yn <- c("y","n")
lev <- list(rain=yn, sprinkler=yn, wet=yn)
r <- tabNew(~rain, levels=lev, values=c(.2, .8))
s_r <- tabNew(~sprinkler:rain, levels = lev, values = c(.01, .99, .4, .6))
w_sr <- tabNew(~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
## sprinkler rain Freq
## 1 y  y 0.01
## 2 n  y 0.99
## 3 y  n 0.40
## 4 n  n 0.60

w_sr %>% flat
## wet sprinkler rain Freq
## 1 y  y  y 0.99
## 2 n  y  y 0.01
## 3 y  n  y 0.80
## 4 n  n  y 0.20

The joint distribution $p(r,s,w) = p(r)p(s|r)p(w|s,r)$ can be obtained with `tabProd()`: ways:

```r
joint <- tabProd(r, s_r, w_sr); joint %>% flat
## wet sprinkler rain Freq
## 1 y  y  y 0.00198
```
## 2 n y y 0.00002
## 3 y n y 0.15840
## 4 n n y 0.03960

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 `tabDist()` in gRbase:

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

```r
## Alternative:
rw <- tabMarg(joint, ~rain + wet)
tabDiv(rw, tabMarg(rw, "wet"))
## or
rw %a/% (rw %a_% ~wet)
```

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

tabDist(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:

```r
data(lizard, package="gRbase")
lizard %>% flat
## diam height species Freq
## 1 <=4 >4.75 anoli 32
## 2 >4 >4.75 anoli 11
## 3 <=4 <=4.75 anoli 86
## 4 >4 <=4.75 anoli 35
```

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 straightforward:

```r
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 )
      ## 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] 664.6

fm1$fit %>% flat
## species diam height Freq
## 1 anoli <=4 >4.75 32.8
## 2 dist <=4 >4.75 60.2
## 3 anoli >4 >4.75 10.2
## 4 dist >4 >4.75 41.8

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

fm2$pearson
## [1] 0.1506

fm2$fit %>% flat
## diam height species Freq
## 1 <=4 >4.75 anoli 32.8
## 2 >4 >4.75 anoli 10.2
## 3 <=4 <=4.75 anoli 85.2
## 4 >4 <=4.75 anoli 35.8
```
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

```r
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:

```r
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()` [gtbbase]. The reverse operation, going from an entry to a cell (which is much less needed) is provided by `entry2cell()` [gtbbase].

```r
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()` [gtbbase].

```r
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 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()`\[\texttt{gRbase}\].

```
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 \((j,3,l)\).\[\texttt{slice2entry}()\]

```
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)

\[\text{FIXME: sliceset should be called margin.}\]
\[\text{FIXME: slicecell and sliceset should be renamed}\]
A More about slicing

Slicing using standard R code can be done as follows:

```r	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
```

A More about slicing

Slicing using standard R code can be done as follows:

```r
## A 2 x 2 x 2 array
head(array, 6)
## Hair Eye Sex Freq
## 1 Black Blue Male 11
## 2 Brown Blue Male 50
## 3 Black Hazel Male 10
## 4 Brown Hazel Male 25

A vector

head(array[1, , 1], 6)
## Brown Blue Hazel
## 32 11 10

A 1 x 3 x 1 array, Sex = Male

head(array[1, , 1, drop=FALSE], 6)
## , , Sex = Male
## Eye Hair Brown Blue Hazel
## Black 32 11 10

Programmatically we can do the above as

```r
do.call("[", c(list(array), list(TRUE, 2:3, TRUE))) %>% flat
do.call("[", c(list(array), list(1, TRUE, 1)))
do.call("[", c(list(array), list(1, TRUE, 1), drop=FALSE))
```

gRbase provides two alternatives for each of these three cases above:

```r
tabSlicePrim(array, slice=list(TRUE, 2:3, TRUE)) %>% flat
tabSlice(array, slice=list(1, 1), margin=2) %>% flat
tabSlicePrim(array, slice=list(1, TRUE, 1))
tabSlice(array, slice=list(1, 1), margin=c(1, 3))
tabSlicePrim(array, slice=list(1, TRUE, 1), drop=FALSE)
tabSlice(array, slice=list(1, 1), margin=c(1, 3), drop=FALSE)
```