Abstract. We give algorithms and R code to compute all multivariate cumulants up to order $p$ of $m$ variables.

1. Introduction


2. Computation

In most publications the discussion of cumulants is algebraic and combinatorial. Although the algebraic results can certainly be used to actually compute multivariate cumulants, the translation into algorithms and code is not entirely straightforward. In many cases
the formulas suggest symbolic computation in packages such as Maple or Mathematica and they are geared towards numerical computation. An exception are Smith [1995] and Balakrishnan et al. [1998], where numerical recursions are given to compute cumulants from moments (and vice versa).

In this paper we present an algorithm that does use combinatorics, because we sum over all partitions of a finite set to compute a cumulant from raw multivariate moments. We leave the heavy combinatorial lifting, however, to the `setparts()` function for set partitioning [West and Hankin 2007], implemented in the R package `partitions` [Hankin 2006]. Our cumulant calculations, using the partitions given by the `setparts()` function, are illustrated nicely in Table A-1 of Mendel [1991, page 297]. The key formula is

\[
\kappa(x_1, \cdots, x_p) = \sum_{\text{partitions}} (-1)^{q-1}(q-1)! \prod_{r=1}^{q} m_X(I_r),
\]

where the summation is over all partitions of \{1, 2, \cdots, p\} and \(q\) is the number of subsets in the partition. The subsets in the partition are \(I_r\), and \(m_X(I_r)\) is the raw (product) moment of the variables in subset \(I_r\).

3. Code

The Appendix has the code, written in R [R Development Core Team 2012]. The code requires, in addition to the `partitions` package, also the `apl` package [De Leeuw and Yajima 2011] for the functions `aplEncode()` and `aplSelect()`. The `apl` package makes it easy to deal with arrays of arbitrary dimension.

3.1. `raw_moments_upto_p`. Given an \(n \times m\) multivariate data matrix this function computes the \((m+1) \times \cdots \times (m+1)\) symmetric array of multivariate moments and product moments around zero up to order \(p\). It uses the `outer()` function recursively, and
does not require anything outside base \texttt{R}. There are \( m + 1 \) rows, columns, slices, ... , because we include a variable with all elements equal to one. Thus the array contains the \( m \) moments of order one, the \( m^2 \) moments of order two, ... , the \( m^p \) moments of order \( p \).

3.2. \texttt{cumulants\_from\_raw\_moments()}. This function takes the raw moment array and makes an array of the same size with the cumulants up to order \( p \), relying heavily on the \texttt{cumulant\_from\_raw\_moments()} function discussed below. It uses \texttt{aplEncode()} to move rapidly through the array.

3.3. \texttt{cumulants\_upto\_p()}. This is a simple function to compute the cumulants directly from the data matrix. It calls the previous two functions.

3.4. \texttt{first\_four\_cumulants()}. Another utility function that returns a list of the first four cumulants

3.5. \texttt{one\_cumulant\_from\_raw\_moments()}. Most of the computation goes on here. The function takes as arguments an index vector indicating which cumulant should be computed and the array with raw moments. Because of our convention to add a column of ones to the data the index vector \( c(1,2,1,4) \) actually leads to the cumulant \( \kappa(1,3) \). Eliminate the ones from the index vector and subtract one from the remaining elements.

3.6. \texttt{four\_cumulants\_direct()}. This function also returns the first four cumulant arrays, but it uses completely written out versions of formula (1) for orders two, three, and four. It has much less array manipulation and indexing, but it cannot be used for higher-order cumulant arrays. It is quite a bit faster than the function \texttt{first\_four\_cumulants()} that selects the cumulants out of the much larger array.
4. Example

We generate data by using $X = YB'$, where $B$ is a fixed orthonormal matrix and $Y$ has independent standard normal random numbers raised to a given power. The function `make_artificial()` can also generate factor analysis examples, using `err=TRUE`, but we do not discuss these in this paper.

```
source("/Users/deleeuw/Dropbox/cumulants/cumulants.R")

make_artificial <- function(n, m, p, pow, err = FALSE, seed = 12345) {
  set.seed (seed)
  mp <- m * p
  np <- n * p
  nm <- n * m
  x <- matrix (rnorm(np), n, p) ^ pow
  if (err) {
    x <- cbind (x, matrix (rnorm(nm), n, m) ^ pow)
  }
  x <- apply (x, 2, function (z) z - mean (z))
  x <- sqrt (n) * qr.Q (qr (x))
  b <- qr.Q (qr (matrix (sample (1 : mp, mp), m, p)))
  if (err) {
    b <- cbind (b, diag (sample (1 : m, m) / m))
  }
  return (list (x = x, b = b, y = tcrossprod (x, b)))
}

arti <- make_artificial (n = 1000, m = 9, p = 4, pow = 2)
x <- arti $ y
print (system.time (cumuold <- four_cumulants_direct (x)))
print (system.time (cumunew <- first_four_cumulants (x)))
print (max (abs (cumuold $ c2 - cumunew $ c2)))
print (max (abs (cumuold $ c3 - cumunew $ c3)))
print (max (abs (cumuold $ c4 - cumunew $ c4)))
```
The output when running this file is

```
> source("cumutry.R")
user  system elapsed
 0.408   0.057   0.426
user  system elapsed
 5.592   0.063   5.592
[1] 1.554312e-15
[1] 5.551115e-17
[1] 5.995204e-15
```

We see that in this example the direct method is 13 times faster than the more general method that uses `setparts()`.


```r
require("partitions")
require("apl")

raw_moments_upto_p <- function (x, p = 4) {
  n <- nrow (x)
  m <- ncol (x)
  if (p == 1) {
    return (c (1, apply (x, 2, mean)))
  }
  y <- array (0, rep (m + 1, p))
  for (i in 1 : n) {
    xi <- c (1, x[i, ])
    z <- xi
    for (s in 2:p) {
      z <- outer (z, xi)
    }
    y <- y + z
  }
  return (y / n)
}

cumulants_from_raw_moments <- function (raw) {
  dimr <- dim (raw)
  nvar <- dimr[1]
  cumu <- array (0, dimr)
  nele <- prod (dimr)
  ldim <- length (dimr)
  spp <- rep (list (0), ldim)
  qpp <- rep (0, ldim)
  rpp <- rep (list (0), ldim)
  for (i in 1 : ldim) {
    spp[[i]] <- setparts (i)
    qpp[[i]] <- factorial (i)
    if (i %% 2) {
      qpp[[i]] <- -qpp[[i]]
    }
  }
  cumu[1] <- spp
  cumu[2] <- qpp
  cumu <- cumu[2 : ldim]
  return (cumu)
}
```

rpp[[i]] <<- aplSelect (raw, c (rep (list (1 : nvar ), i), rep (list (1 : 1), ldim - i)))

qpp <<- c(1, qpp)
for (i in 2 : nele) {
  ind <<- aplEncode (i, dimr)
  cumu[i] <<- one_cumulant_from_raw_moments (ind, raw)
}
return (cumu)

first_four_cumulants <<- function (x) {
  cumu <<- cumulants_uppto_p (x)
  nsel <<- 2 : dim (cumu)[1]
  return (list (c1 = cumu[1, 1, 1, nsel],
                c2 = cumu[1, 1, nsel, nsel],
                c3 = cumu[1, nsel, nsel, nsel],
                c4 = cumu[nsel, nsel, nsel, nsel]))
}

one_cumulant_from_raw_moments <<- function (jnd, raw) {
  jnd <<- jnd [which (jnd != 1)] - 1
  nnd <<- length (jnd)
  ndr <<- dim (raw)[1]
  nrt <<- length (dim (raw))
  raw <<- rpp[[nnd]]
  nvar <<- ndr - 1
  nraw <<- max (1, length (dim (raw)))
  sp <<- spp [[nraw]]
  nbell <<- ncol (sp)
  sterm <<- 0
  for (i in 1 : nbell) {
    ind <<- sp[, i]
und <- unique (ind)
term <- qpp[length (und)]
for (j in und) {
  knnd <- jnd[which (ind == j)] + 1
  lnd <- c (knd, rep (1, nraw - length (knd)))
  term <- term * raw [matrix (lnd, 1, nraw)]
}
sterm <- sterm + term
}
return (sterm)

four_cumulants_direct <- function (x) {
  n <- nrow (x)
  m <- ncol (x)
  mm <- 1 : m
  nn <- 1 : n
  r1 <- colSums (x) / n
  r2 <- crossprod (x) / n
  r3 <- array (0, c (m, m, m))
  r4 <- array (0, c (m, m, m, m))

  for (i in nn) {
    r3 <- r3 + outer (outer (x [i, ], x [i, ]), x [i, ])
    r4 <- r4 + outer (outer (x [i, ], x [i, ]), outer (x [i, ], x [i, ]))
  }
  r3 <- r3 / n
  r4 <- r4 / n
  c2 <- r2 - outer (r1, r1)
  c3 <- r3
  for (i in mm) for (j in mm) for (k in mm) {
    s3 <- r3 [i, j, k]
    s21 <- r2 [i, j] * r1 [k] + r2 [i, k] * r1 [j] + r2 [j, k] * r1 [i]
    s111 <- r1 [i] * r1 [j] * r1 [k]
    c3 [i, j, k] <- s3 - s21 + 2 * s111
  }
  c4 <- r4
for (i in mm) for (j in mm) for (k in mm) for (l in mm) {
  s4 <- r4 [i, j, k, l]
  s31 <- r3 [i, j, k] * r1 [l] + r3 [i, j, l] * r1 [k] + r3 [i, k, l] * r1 [j] + r3 [j, k, l] * r1 [i]
  s22 <- r2 [i, j] * r2 [k, l] + r2 [i, k] * r2 [j, l] + r2 [j, k] * r2 [i, l]
  s211 <- r2 [i, j] * r1 [k] * r1 [l] + r2 [i, k] * r1 [j] * r1 [l] + r2 [i, l] * r1 [k] * r1 [j] +
    r2 [j, k] * r1 [i] * r1 [l] + r2 [j, l] * r1 [i] * r1 [k] + r2 [k, l] * r1 [i] * r1 [j]
  s1111 <- r1 [i] * r1 [j] * r1 [k] * r1 [l]
  c4 [i, j, k, l] <- s4 - s31 - s22 + 2 * s211 - 6 * s1111
}
return (list(c1 = r1, c2 = c2, c3 = c3, c4 = c4))

DEPARTMENT OF STATISTICS, UNIVERSITY OF CALIFORNIA, LOS ANGELES, CA 90095-1554

E-mail address, Jan de Leeuw: deleeuw@stat.ucla.edu

URL, Jan de Leeuw: http://gifi.stat.ucla.edu