Unbiased Central Moment Estimates

Inna Gerlovina[1,2], Alan Hubbard[1]

July 15, 2018

[1]University of California, Berkeley

[2]University of California, San Francisco

innager@berkeley.edu

Contents

1 Introduction

`Umoments` package calculates unbiased central moment estimates and their two-sample analogs, also known as pooled estimates, up to sixth order (sample variance and pooled variance are commonly used second order examples of corresponding estimators). Orders four and higher include powers and products of moments - for example, fifth moment and a product of second and third moments are both of fifth order; those estimators are also included in the package. The estimates can be obtained directly from samples or from naïve biased moment estimates.

If the estimates of orders beyond sixth are needed, they can be generated using the set of functions also provided in the package. Those functions generate symbolic expressions for expectations of moment combinations of an arbitrary order and, in addition to moment estimation, can be used for other derivations that require such expectations (*e.g.* Edgeworth expansions).

2 Estimators: naïve biased to unbiased

For a sample $X_1, \ldots, X_n$, a naïve biased estimate of a k'th central moment μ_k is

$$m_k = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^k, \qquad k = 2, 3, \ldots . \tag{1}$$

These biased estimates together with the sample size n can be used to calculate unbiased estimates of a given order, *e.g.* $Var(X) = \frac{n}{n-1}m_2$.

Let $M(\cdot)$ be an unbiased estimator of an expression inside the parentheses, e.g. $E\left[M(\mu_2^3)\right] = \mu_2^3$. Note that in general $M(\mu_k\mu_l) \neq M(\mu_k)M(\mu_l)$. The package provides functions to calculate unbiased estimates of all the powers and combinations (up to sixth order) of the form $\mu_2^{j_2}\mu_3^{j_3}\cdots$. For orders greater than three, each estimator of that order requires all the naïve unbiased estimates that comprise that order: for example, estimators of 6'th order are $M(\mu_2^3)$, $M(\mu_2\mu_4)$, $M(\mu_3^2)$, $M(\mu_6)$, and each one of them uses estimates m_2, m_3, m_4, and m_6.

```
> library(Umoments)
> n <- 10
> # draw a sample
> smp <- rgamma(n, shape = 3)
```

```
> # calculate biased estimates up to 6th order
> m <- numeric(6)
> m[1] <- mean(smp)
> for (j in 2:6) {
+   m[j] <- mean((smp - m[1])^j)
+ }
```

Calculate unbiased estimates - examples:
$M(\mu_4)$

```
> uM4(m[2], m[4], n)

[1] 50.96546
```

$M(\mu_2\,\mu_3) \neq M(\mu_2)M(\mu_3)$

```
> uM2M3(m[2], m[3], m[5], n)

[1] 28.74841

> uM2(m[2], n)*uM3(m[3], n)

[1] 35.27032
```

$M\left(\mu_3^2\right)$

```
> uM3pow2(m[2], m[3], m[4], m[6], n)

[1] -9.807567
```

Two-sample pooled estimates are calculated in a similar manner. For a sample $X_1, \ldots, X_{n_x}, Y_1, \ldots, Y_{n_y}$, let

$$m_k = \frac{\sum_{i=1}^{n_x}(X_i - \bar{X})^k + \sum_{i=1}^{n_y}(Y_i - \bar{Y})^k}{n_x + n_y}. \tag{2}$$

These naïve biased estimates together with n_x, n_y are sufficient for calculating unbiased pooled estimates of a corresponding order.

```
> nx <- 10
> ny <- 8
> shp <- 3
> smpx <- rgamma(nx, shape = shp) - shp
> smpy <- rgamma(ny, shape = shp)
> mx <- mean(smpx)
> my <- mean(smpy)
> m  <- numeric(6)
> for (j in 2:6) {
+   m[j] <- mean(c((smpx - mx)^j, (smpy - my)^j))
+ }
> uM2pool(m[2], nx, ny)

[1] 3.760201

> uM2pow3pool(m[2], m[3], m[4], m[6], nx, ny)

[1] 25.98305
```

3 Estimates obtained from a sample

Functions `uM()` and `uMpool()` calculate estimates directly from sample, for one- and two-sample statistics respectively. For a specified order (one of function's arguments), all estimates up to that order are returned in the named vector, where element names indicate the estimands and correspond to the names of the functions that calculate single estimates - thus, for example, the estimate of μ_4 is named `"M4"`, the estimate of $\mu_2\mu_3$ - `"M2M3"`, and the estimate of μ_3^2 - `"M3pow2"`.

```
> # simulate a sample
> nsmp <- 23
> smp <- rgamma(nsmp, shape = 3)
> # two categories for pooled estimates
> treatment <- sample(0:1, size = nsmp, replace = TRUE)
> # estimates
> uM(smp, 5)

       M2        M3    M2pow2        M4      M2M3        M5
 2.015065  1.271971  3.892643  7.399219  2.434467 12.032586
```

```
> uMpool(smp, treatment, 6)

       M2         M3     M2pow2         M4       M2M3         M5     M2pow3
2.0801543  1.4903796  4.0918524  8.7234076  2.7842843 16.8284188  7.6525674
   M3pow2       M2M4         M6
0.3709571 16.0948387 61.2186714
```

Note that even if $n_x = n_y$, the estimates for one- and two-sample statistics are not the same since two-sample ones have fewer degrees of freedom.

4 Higher order estimates (symbolic expressions for expectations)

If estimates beyond sixth order need to be calculated, higher order estimators can be derived using these *Jupyter* or *Sage notebook* templates. Consider random variable X, $E(X) = 0$, and random sample $X_1, \ldots, X_n$. The main part of these derivations is finding expectations of the form

$$E\left(\overline{X}^{j_1}\,\overline{X^2}^{j_2}\,\overline{X^3}^{j_3}\cdots\right), \text{ where } \overline{X^j} = \frac{1}{n}\sum_{i=1}^{j} X_i^j. \tag{3}$$

Functions provided in the package generate symbolic expressions for such expectation in terms of central moments μ_k and sample size n.

The expectation (3) is generated by `one_combination()` function. The first argument to the function is a vector $(j_1, j_2, \ldots)$ of non-negative integers, where a position/index i of a vector element indicates the order of the sample moment $\overline{X^i}$ that is to be raised to the power of a corresponding value. Thus, vector $(5, 0, 0, 1)$ would be used to generate $E\left(\overline{X}^5\,\overline{X^4}\right)$ and vector $(0, 3, 4)$ - to generate $E\left(\overline{X^2}^3\,\overline{X^3}^4\right)$. A character string representing sample size symbol can be passed as an optional second argument.

Example: generate $E\left(\overline{X}^5\,\overline{X^3}^2\,\overline{X^4}\right)$ for a sample $X_1, \ldots, X_{n_x}$

```
> one_combination(c(5, 0, 2, 1), "n_x")

( 1*n_x*mu15^1 + 10*n_x*(n_x-1)*mu2^1*mu13^1 +
15*n_x*(n_x-1)*(n_x-2)*mu2^2*mu11^1 +
15*n_x*(n_x-1)*(n_x-2)*(n_x-3)*(n_x-4)*mu5^1*mu3^2*mu2^2 +
```

```
30*n_x*(n_x-1)*(n_x-2)*(n_x-3)*(n_x-4)*mu4^2*mu3^1*mu2^2 +
30*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu8^1*mu3^1*mu2^2 +
45*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu7^1*mu4^1*mu2^2 +
15*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu6^1*mu5^1*mu2^2 +
30*n_x*(n_x-1)*(n_x-2)*mu3^1*mu2^1*mu10^1 +
10*n_x*(n_x-1)*(n_x-2)*(n_x-3)*(n_x-4)*mu4^1*mu3^3*mu2^1 +
30*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu7^1*mu3^2*mu2^1 +
90*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu6^1*mu4^1*mu3^1*mu2^1 +
60*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu5^2*mu3^1*mu2^1 +
120*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu5^1*mu4^2*mu2^1 +
70*n_x*(n_x-1)*(n_x-2)*mu9^1*mu4^1*mu2^1 +
90*n_x*(n_x-1)*(n_x-2)*mu8^1*mu5^1*mu2^1 +
60*n_x*(n_x-1)*(n_x-2)*mu7^1*mu6^1*mu2^1 + 12*n_x*(n_x-1)*mu3^1*mu12^1
+ 10*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu6^1*mu3^3 +
66*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu5^1*mu4^1*mu3^2 +
21*n_x*(n_x-1)*(n_x-2)*mu9^1*mu3^2 +
30*n_x*(n_x-1)*(n_x-2)*(n_x-3)*mu4^3*mu3^1 +
72*n_x*(n_x-1)*(n_x-2)*mu8^1*mu4^1*mu3^1 +
72*n_x*(n_x-1)*(n_x-2)*mu7^1*mu5^1*mu3^1 +
30*n_x*(n_x-1)*(n_x-2)*mu6^2*mu3^1 + 16*n_x*(n_x-1)*mu4^1*mu11^1 +
45*n_x*(n_x-1)*(n_x-2)*mu7^1*mu4^2 +
126*n_x*(n_x-1)*(n_x-2)*mu6^1*mu5^1*mu4^1 + 26*n_x*(n_x-1)*mu5^1*mu10^1
+ 30*n_x*(n_x-1)*(n_x-2)*mu5^3 + 31*n_x*(n_x-1)*mu9^1*mu6^1 +
27*n_x*(n_x-1)*mu8^1*mu7^1 ) / n_x^8
```