Performance Measures: Analyzing and Interpreting Simulation Results

Ashley I Naimi

Spring 2024

Contents

1 Evaluating the Performance of an Estimator

Often, a Monte Carlo simulation function will return data on the estimator or estimators being evaluated. Typically, these data might look something like the following:

Estimator	Estimate	SE	index	N
CATE_m0_glm	0.014032436	0.032976105	1	1200
CATE_m1_glm	-0.614720936	0.037829332	1	1200
CATE_m0_grf	0.021007357	0.032284058	1	1200
CATE_m1_grf	-0.60965855	0.034149359	1	1200

This table is an excerpt from an actual simulation study we are conducting in our group. In this Table, each row represents a particular estimator or version of the estimator under study. The first column identifies the estimator or version of the estimator, the "Estimate" and "Standard Error" columns represent, the "index" represents the Monte Carlo number, and N is the sample size.

For the most part in this short course, we have covered tools and techniques that we can use to generate data such as these. Once we have the dataset, we have to determine what we want to use to measure the performance of each estimator. We discuss those tools here. They include bias, mean squared error, a measure of standard error bias (we sometimes refer to as efficiency), confidence interval coverage and length, and power.

To illustrate these measures, we're going to use our non-collapsibility function again. This time, we'll add a bootstrap resampling approach to obtain standard errors for the marginally adjusted odds ratio to demonstrate some of these metrics.

First, we'll need to estimate the true marginally adjusted odds ratio using Monte Carlo integration:

```r
expit <- function(a){1/(1+exp(-a))}

set.seed(123)
collapsibility_function <- function(index, intercept, exposure){
    n = 500000000

    C <- rnorm(n,0,1)

    theta <- c(0,log(2))
    pi <- expit(theta[1]+theta[1]*C)

    A <- exposure

    beta <- c(intercept,log(2),log(2))
    mu <- expit(beta[1] + beta[2]*A + beta[3]*C)
    Y <- rbinom(n,1,mu)

    res <- mean(Y)/(1 - mean(Y))

    return(res)
}

odds1 <- collapsibility_function(index=1, intercept = 0, exposure = 1)
odds0 <- collapsibility_function(index=1, intercept = 0, exposure = 0)

or_marg <- odds1/odds0

# true marginal OR for intercept = 0
# 1.871259
```

```
library(parallel)
library(boot)
```

```
##
## Attaching package: 'boot'

## The following object is masked from 'package:survival':
##
##     aml
```

```r
expit <- function(a) {
    1/(1 + exp(-a))
}

set.seed(123)

collapsibility_function <- function(index, intercept, true_m,
    true_c) {
    n = 500

    C <- rnorm(n, 0, 1)

    theta <- c(0, log(2))
    pi <- expit(theta[1] + theta[1] * C)

    A <- rbinom(n, 1, pi)

    beta <- c(intercept, log(2), log(2))
    mu <- expit(beta[1] + beta[2] * A + beta[3] * C)
    Y <- rbinom(n, 1, mu)

    glm.res0 <- mean(Y)

    m1 <- glm(Y ~ A + C, family = binomial(link = "logit"))
    glm.res1 <- summary(m1)$coefficients[2, 1:2]

    muhat1 <- mean(predict(m1, newdata = data.frame(A = 1, C),
        type = "response"))
    muhat0 <- mean(predict(m1, newdata = data.frame(A = 0, C),
        type = "response"))

    ## compute the odds from these average probabilities
    odds1 <- muhat1/(1 - muhat1)
    odds0 <- muhat0/(1 - muhat0)
```

```r
## glm.res2 is the marginal log-odds ratio
glm.res2 <- log(odds1/odds0)

# bootstrap SEs
c_data <- data.frame(Y, A, C)
boot_func <- function(data, index) {
    ## order matters!

    boot_dat <- data[index, ]

    m1_ <- glm(Y ~ A + C, data = boot_dat, family = binomial(link = "logit"))
    glm.res1 <- summary(m1_)$coefficients[2, 1:2]

    muhat1_ <- mean(predict(m1_, newdata = transform(boot_dat,
        A = 1), type = "response"))
    muhat0_ <- mean(predict(m1_, newdata = transform(boot_dat,
        A = 0), type = "response"))

    ## compute the odds from these average
    ## probabilities
    odds1_ <- muhat1_/(1 - muhat1_)
    odds0_ <- muhat0_/(1 - muhat0_)

    return(odds1_/odds0_)
}

boot_obj <- boot(data = c_data, statistic = boot_func, R = 200,
    parallel = "no")  # can only parallelize one level

glm.res2 <- c(glm.res2, sd(boot_obj$t))

res <- data.frame(intercept = intercept, t(glm.res1), t(glm.res2),
    true_m = true_m, true_c = true_c)
    return(res)
}

# choose the number of cores to use num_cores <-
# detectCores() - 2

# how many cores?  num_cores

# special mclapply seed ...  RNGkind('L'Ecuyer-CMRG')

# set the seed
set.seed(123)

# run the function
sim_res <- lapply(1:500, function(x) collapsibility_function(index = x,
    intercept = 0, true_m = 1.871259, true_c = 2))  #,
# mc.cores = num_cores)

sim_res <- do.call(rbind, sim_res)
```

After running this simulation function, we have the following data to work with:

```
names(sim_res) <- c("intercept", "cEstimate", "cSE", "mEstimate",
    "mSE", "true_m", "true_c")

head(sim_res, 3)
```

```
##   intercept cEstimate       cSE mEstimate       mSE   true_m true_c
## 1         0 0.8148259 0.1961145 0.7332679 0.3889278 1.871259      2
## 2         0 0.9603903 0.1950268 0.8900275 0.4527568 1.871259      2
## 3         0 0.6378918 0.1948775 0.5610920 0.3111768 1.871259      2
```

```
tail(sim_res, 3)
```

```
##     intercept cEstimate       cSE mEstimate       mSE   true_m true_c
## 498         0 0.7207652 0.1927378 0.6486043 0.3303620 1.871259      2
## 499         0 0.7329280 0.1938252 0.6611271 0.3314619 1.871259      2
## 500         0 0.5046467 0.1937086 0.4473192 0.2893371 1.871259      2
```

Let's use these data to compare the conditionally and marginally adjusted ORs. We'll start with bias.

1.1 Bias

Bias is defined as the average (over all Monte Carlo samples) of the difference between the estimate and the truth for a given sample size:

$$\hat{b} = E(\hat{\theta} - \theta)$$

In our setting, the sample size is 500 observations, and the Monte Carlo sample size is also 500. We can compute the bias in our dataset as follows:

```
mc_bias <- sim_res %>%
  summarize(bias_c = mean(cEstimate - log(true_c)),
            bias_m = mean(mEstimate - log(true_m)))

mc_bias
```

```
##        bias_c      bias_m
## 1 0.006667249 0.004482513
```

Let's construct a Monte Carlo standard error function that we can use for each of these bias estimates. We can actually do this with the standard formulas provided in Morris et al. (2019), Table 6:

```
mc_se_bias <- function(x, n) {
    sqrt(sum((x - mean(x))^2)/(n * (n - 1)))
}
```

```
mc_bias_se <- c(mc_se_bias(sim_res$cEstimate, n = 500), mc_se_bias(sim_res$mEstimate,
    n = 500))

mc_bias_se
```

```
## [1] 0.008776152 0.007963605
```

If interested, one can construct standard inferential statistics, such as p-values and confidence intervals for the bias parameter using these estimates and Monte Carlo standard errors.

1.2 Mean Squared Error

Mean squared error is defined as the average (over all Monte Carlo samples) of the squared difference between the estimate and the truth for a given sample size:

$$\widehat{mse} = E(\hat{\theta} - \theta)^2$$

We can compute the mean squared error as follows:

```
mse_c = mean((sim_res$cEstimate - log(sim_res$true_c))^2)
mse_m = mean((sim_res$mEstimate - log(sim_res$true_m))^2)
```

We can also construct a Monte Carlo standard error function for the MSE (Morris et al., 2019, Table 6):

```
mc_se_mse <- function(x, n) {
    sqrt(sum(((x - mean(x))^2 - mean((x - mean(x))^2))^2)/(n *
        (n - 1)))
}

mc_se_mse(sim_res$cEstimate, n = 500)
```

```
## [1] 0.002524538
```

```
mc_se_mse(sim_res$mEstimate, n = 500)
```

```
## [1] 0.002080995
```

1.3 Bias Standard Error

The dataset we constructed from our Monte Carlo simulation function gave us the following variables:

```
head(sim_res, 3)
```

```
##   intercept cEstimate       cSE mEstimate       mSE   true_m true_c
## 1         0 0.8148259 0.1961145 0.7332679 0.3889278 1.871259      2
## 2         0 0.9603903 0.1950268 0.8900275 0.4527568 1.871259      2
## 3         0 0.6378918 0.1948775 0.5610920 0.3111768 1.871259      2
```

Each point estimate in this dataset has an associated standard error. In actuality, we can state that the standard error should technically equal the standard deviation of all the estimates in the Monte Carlo sample. This is what a standard error estimates. Thus, we can compare the average of all the standard errors to the standard deviation of all the point estimates:

```
sd(sim_res$cEstimate) - mean(sim_res$cSE)
```

```
## [1] 0.001375635
```

```
sd(sim_res$mEstimate) - mean(sim_res$mSE)
```

```
## [1] -0.1699376
```

```
sd(sim_res$cEstimate)/mean(sim_res$cSE)
```

```
## [1] 1.007059
```

```
sd(sim_res$mEstimate)/mean(sim_res$mSE)
```

```
## [1] 0.5116865
```

This is telling us that the estimated standard errors for the marginally adjusted odds ratio is much larger than it should be. However, this is an inevitable consequence of the fact that we only used 200 resamples for the bootstrap. But we did this to avoid waiting too long for these notes to compile. In practice, we should use no less than 1,000 resamples to get more accurate standard error estimates from the bootstrap.

1.4 Confidence Interval Coverage and Length

For an unbiased estimator, a confidence interval is defined as an upper and lower bound that include the true value at the nominal rate (e.g., 95%) under repeated sampling. The first task to evaluating confidence interval coverage is to construct confidence intervals in each of our Monte Carlo samples. We can do this as follows:

```
sim_res <- sim_res %>% mutate(
  cLCL = cEstimate - 1.96*cSE,
  cUCL = cEstimate + 1.96*cSE,
  mLCL = mEstimate - 1.96*mSE,
  mUCL = mEstimate + 1.96*mSE
)
```

After doing this, we have the following data added to our Monte Carlo data:

```
head(sim_res, 3)
```

```
##   intercept cEstimate       cSE mEstimate       mSE   true_m true_c      cLCL
## 1         0 0.8148259 0.1961145 0.7332679 0.3889278 1.871259      2 0.4304415
## 2         0 0.9603903 0.1950268 0.8900275 0.4527568 1.871259      2 0.5781378
## 3         0 0.6378918 0.1948775 0.5610920 0.3111768 1.871259      2 0.2559318
##       cUCL         mLCL     mUCL
## 1 1.199210 -0.029030503 1.495566
## 2 1.342643  0.002624151 1.777431
## 3 1.019852 -0.048814409 1.170999
```

With that, we can construct an indicator of whether these confidence intervals include the true values:

```r
sim_res <- sim_res %>% mutate(
  cCoverage = cLCL < log(true_c) & log(true_c) < cUCL,
  mCoverage = mLCL < log(true_m) & log(true_m) < mUCL
)

mean(sim_res$cCoverage)
```

```
## [1] 0.94
```

```r
mean(sim_res$mCoverage)
```

```
## [1] 0.99
```

Notice that the marginally adjusted confidence intervals have a greater than nominal coverage rate. This is likely due to the aforementioned bias in the standard error estimator (we need to use more bootstrap resamples!). But we can further explore this using confidence interval length:

```r
sim_res <- sim_res %>% mutate(
  cCI_length = cUCL - cLCL,
  mCI_length = mUCL - mLCL
)

mean(sim_res$cCI_length)
```

```
## [1] 0.7638711
```

```r
mean(sim_res$mCI_length)
```

```
## [1] 1.364196
```

And we can see that the confidence interval length is much longer for the marginally adjusted OR compared to the conditional one.

1.5 Power

As a final calculation, let's evaluate the power to detect a non-null hypothesis with our marginal and conditional effect estimators. We can do this using a variation of the confidence intervals we've constructed, as follows:

```r
sim_res <- sim_res %>% mutate(
  cPower = cLCL > 0 | cUCL < 0,
  mPower = mLCL > 0 | mUCL < 0
)

mean(sim_res$cPower)
```

```
## [1] 0.944
```

```r
mean(sim_res$mPower)
```

```
## [1] 0.194
```

This is telling us that, under the conditions of the simulation (single confounder, true conditional / marginal effect of ~ 2, etc), using a conditionally adjusted model gives us a power of 94.4 % to detect an odds ratio of 2. However, the marginally adjusted approach gives us a much lower power. Again, this poor performance is largely due to the limited bootstrap resample number.

References

Tim P. Morris, Ian R. White, and Michael J. Crowther. Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, 38(11):2074–2102, 2019.