I’m contemplating the idea of teaching a course on simulation next fall, so I have been exploring various topics that I might include. (If anyone has great ideas either because you have taught such a course or taken one, definitely drop me a note.) Monte Carlo (MC) simulation is an obvious one. I like the idea of talking about *importance sampling*, because it sheds light on the idea that not all MC simulations are created equally. I thought I’d do a brief blog to share some code I put together that demonstrates MC simulation generally, and shows how importance sampling can be an improvement.

Like many of the topics I’ve written about, this is a vast one that certainly warrants much, much more than a blog entry. MC simulation in particular, since it is so fundamental to the practice of statistics. MC methods are an essential tool to understand the behavior of statistical models. In fact, I’ve probably used MC simulations in just about all of my posts – to generate repeated samples from a model to explore bias, variance, and other distributional characteristics of a particular method.

For example, if we want to assess the variability of a regression parameter estimate, we can repeatedly generate data from a particular “hidden” model, and for each data set fit a regression model to estimate the parameter in question. For each iteration, we will arrive at a different estimate; the variation of all those estimates might be of great interest. In particular, the standard deviation of those estimates is the standard error of the estimate. (Of course, with certain problems, there are ways to analytically derive the standard errors. In these cases, MC simulation can be used to verify an analysis was correct. That’s the beauty of statistics – you can actually show yourself you’ve gotten the right answer.)

**A simple problem**

In this post, I am considering a simple problem. We are interested in estimating the probability of drawing a value between 2 and 2.5 from a standard normal distribution. That is, we want to use MC simulation to estimate

\[p =P(2.0 <= X <= 2.5), \ \ \ X\sim N(0,1)\]

Of course, we can use R to get us the true \(p\) directly without any simulation at all, but that is no fun:

pnorm(2.5, 0, 1) - pnorm(2, 0, 1)

## [1] 0.01654047

To do this using simulation, I wrote a simple function that checks to see if a value falls between two numbers.

inSet <- function(x, minx, maxx) {

result <- (x >= minx & x <= maxx)

return(as.integer(result))

}

To estimate the desired probability, we just repeatedly draw from the standard normal distribution. After each draw, we check to see if the value falls between 2.0 and 2.5, and store that information in a vector. The vector will have a value of 1 each time a value falls into the range, and 0 otherwise. The proportion of 1’s is the desired probability. Or in other words, \(\hat{p} = \bar{z}\), where \(\bar{z} = \frac{1}{1000} \sum{z}\).

set.seed(1234)

z <- vector("numeric", 1000)

for (i in 1:1000) {

y <- rnorm(1, 0, 1)

z[i] <- inSet(y, 2, 2.5)

}

mean(z)

## [1] 0.018

The estimate is close to the true value, but there is no reason it would be exact. In fact, I would be suspicious if it were. Now, we can also use the variance of \(z\) to estimate the standard error of \(\hat{p}\):

sd(z)/sqrt(1000)

## [1] 0.004206387

**Faster code?**

If you’ve read any of my other posts, you know I am often interested in trying to make R run a little faster. This can be particularly important if we need to repeat tasks over and over, as we will be doing here. The for loop I used here is not ideal. Maybe simstudy (and you could do this without simstudy) can do better. Let’s first see if it provides the same estimates:

library(data.table)

library(simstudy)

# define the data

defMC <- defData(varname = "y", formula = 0,

variance = 1, dist = "normal")

defMC <- defData(defMC, varname = "z", formula = "inSet(y, 2, 2.5)",

dist = "nonrandom")

# generate the data - the MC simulation without a loop

set.seed(1234)

dMC <- genData(1000, defMC)

# evaluate mean and standard error

dMC[ , .(mean(z), sd(z)/sqrt(1000))]

## V1 V2

## 1: 0.018 0.004206387

So, the results are identical – no surprise there. But which approach used fewer computing resources. To find this out, we turn to the microbenchmark package. (I created a function out of the loop above that returns a vector of 1’s and 0’s.)

library(microbenchmark)

mb <- microbenchmark(tradMCsim(1000), genData(1000, defMC))

summary(mb)[, c("expr", "lq", "mean", "uq", "neval")]

## expr lq mean uq neval

## 1 tradMCsim(1000) 1.428656 2.186979 2.573003 100

## 2 genData(1000, defMC) 1.376450 1.668248 1.674146 100

With 1000 draws, there is actually very little difference between the two approaches. But if we start to increase the number of simulations, the differences become apparent. With 10000 draws, the simstudy approach is more than 7 times faster. The relative improvement continues to increase as the number of draws increases.

mb <- microbenchmark(tradMCsim(10000), genData(10000, defMC))

summary(mb)[, c("expr", "lq", "mean", "uq", "neval")]

## expr lq mean uq neval

## 1 tradMCsim(10000) 18.453128 21.619022 22.226165 100

## 2 genData(10000, defMC) 2.006622 2.432078 2.508662 100

**Estimating variation of \(\hat{p}\)**

Now, we can stop using the loop, at least to generate a single set of draws. But, in order to use MC simulation to estimate the variance of \(\hat{p}\), we still need to use a loop. In this case, we will generate 1500 data sets of 1000 draws each, so we will have 1500 estimates of \(\hat{p}\). (It would probably be best to do all of this using Rcpp, where we can loop with impunity.)

iter <- 1500

estMC <- vector("numeric", iter)

for (i in 1:iter) {

dtMC <- genData(1000, defMC)

estMC[i] <- dtMC[, mean(z)]

}

head(estMC)

## [1] 0.020 0.013 0.023 0.017 0.016 0.019

We can estimate the average of the \(\hat{p}\)’s, which should be close to the true value of \(p \approx 0.0165\). And we can check to see if the standard error of \(\hat{p}\) is close to our earlier estimate of 0.004.

c(mean(estMC), sd(estMC))

## [1] 0.016820000 0.004113094

**Importance sampling**

As we were trying to find an estimate for \(p\) using the simulations above, we spent a lot of time drawing values far outside the range of 2 to 2.5. In fact, almost all of the draws were outside that range. You could almost so that most of those draws were providing little if any information. What if we could focus our attention on the area we are interested in – in this case the 2 to 2.5, without sacrificing our ability to make an unbiased estimate? That would be great, wouldn’t it? That is the idea behind importance sampling.

The idea is to draw from a distribution that is (a) easy to draw from and (b) is close to the region of interest. Obviously, if 100% of our draws is from the set/range in question, then we’ve way over-estimated the proportion. So, we need to reweight the draws in such a way that we get an unbiased estimate.

**A very small amount of theory**

A teeny bit of stats theory here (hope you don’t jump ship). The expected value of a draw falling between 2 and 2.5 is

\[E\_x(I\_R) = \int\_{-\infty}^{\infty}{I\_{R}(x)f(x)dx} \ ,\]

where \(I\_R(x)=1\) when \(2.0 \le x \le 2.5\), and is 0 otherwise, and \(f(x)\) is the standard normal density. This is the quantity that we were estimating above. Now, let’s say we want to draw closer to the range in question – say using \(Y\sim N(2.25, 1)\). We will certainly get more values around 2 and 2.5. If \(g(y)\) represents this new density, we can write \(E(I\_R)\) another way:

\[E\_y(I\_R) = \int\_{-\infty}^{\infty}{I\_{R}(y)\frac{f(y)}{g(y)}g(y)dy} \ .\] Notice that the \(g(y)\)’s cancel out and we end up with the same expectation as above, except it is with respect to \(y\). Also, notice that the second equation is also a representation of \(E\_y \left( I\_{R}(y)\frac{f(y)}{g(y)} \right)\).

I know I am doing a lot of hand waving here, but the point is that

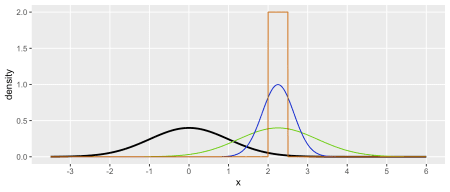
\[E\_x(I\_R) = E\_y \left( I\_{R}\frac{f}{g} \right)\]

Again, \(f\) and \(g\) are just the original density of interest – \(N(0,1)\) – and the “important” density – \(N(2.25, 1)\) – respectively. In our modified MC simulation, we draw a \(y\_i\) from the \(N(2.25, 1)\), and then we calculate \(f(y\_i)\), \(g(y\_i)\), and \(I\_R(y\_i)\), or more precisely, \(z\_i = I\_{R}(y\_i)\frac{f(y\_i)}{g(y\_i)}\). To get \(\hat{p}\), we average the \(z\_i\)’s, as we did before.

**Beyond theory**

Why go to all of this trouble? Well, it turns out that the \(z\_i\)’s will be much less variable if we use importance sampling. And, as a result, the standard error of our estimate can be reduced. This is always a good thing, because it means a reduction in uncertainty.

Maybe a pretty plot will provide a little intuition? Our goal is to estimate the area under the black curve between 2 and 2.5. An importance sample from a \(N(2.25, 1)\) distribution is represented by the green curve. I think, however, it might be easiest to understand the adjustment mechanism by looking at the orange curve, which represents the uniform distribution between 2 and 2.5. The density is \(g(y) = 2\) for all values within the range, and \(g(y) = 0\) outside the range. Each time we generate a \(y\_i\) from the \(U(2,2.5)\), the value is guaranteed to be in the target range. As calculated, the average of all the \(z\_i\)’s is the ratio of the area below the black line relative to the area below the orange line, but only in the range between 2 and 2.5. (This may not be obvious, but perhaps staring at the plot for a couple of minutes will help.)



**Reducing standard errors by improving focus**

Now we can generate data and estimate \(\hat{p}\) and \(se(\hat{p})\). First, here is a simple function to calculate \(z\).

h <- function(I, f, g) {

dx <- data.table(I, f, g)

dx[I != 0, result := I \* f / g]

dx[I == 0, result := 0]

return(dx$result)

}

We can define the three Monte Carlo simulations based on the three different distributions using simstudy. The elements that differ across the three MC simulations are the distribution we are drawing from and the density \(g\) of that function.

# Normal(2.5, 1)

def1 <- defData(varname = "y", formula = 2.25,

variance = 1, dist = "normal")

def1 <- defData(def1, varname = "f", formula = "dnorm(y, 0, 1)",

dist = "nonrandom")

def1 <- defData(def1, varname = "g", formula = "dnorm(y, 2.25, 1)",

dist = "nonrandom")

def1 <- defData(def1, varname = "I", formula = "inSet(y, 2, 2.5)",

dist = "nonrandom")

def1 <- defData(def1, varname = "z", formula = "h(I, f, g)",

dist = "nonrandom")

# Normal(2.5, .16)

def2 <- updateDef(def1, "y", newvariance = 0.4^2)

def2 <- updateDef(def2, "g", newformula = "dnorm(y, 2.25, 0.4)")

# Uniform(2, 3)

def3 <- updateDef(def1, "y", newformula = "2;2.5",

newvariance = 0, newdist = "uniform")

def3 <- updateDef(def3, "g", newformula = "dunif(y, 2, 2.5)")

Here is a peek at one data set using the uniform sampling approach:

genData(1000, def3)

## id y f g I z

## 1: 1 2.181324 0.03695603 2 1 0.018478013

## 2: 2 2.381306 0.02341805 2 1 0.011709023

## 3: 3 2.338066 0.02593364 2 1 0.012966819

## 4: 4 2.200399 0.03544350 2 1 0.017721749

## 5: 5 2.461919 0.01926509 2 1 0.009632543

## ---

## 996: 996 2.118506 0.04229983 2 1 0.021149914

## 997: 997 2.433722 0.02064175 2 1 0.010320876

## 998: 998 2.265325 0.03066025 2 1 0.015330127

## 999: 999 2.107219 0.04332075 2 1 0.021660374

## 1000: 1000 2.444599 0.02010130 2 1 0.010050651

And here are the estimates based on the three different importance samples. Again each iteration is 1000 draws from the distribution – and we use 1500 iterations:

iter <- 1500

N <- 1000

est1 <- vector("numeric", iter)

est2 <- vector("numeric", iter)

est3 <- vector("numeric", iter)

for (i in 1:iter) {

dt1 <- genData(N, def1)

est1[i] <- dt1[, mean(z)]

dt2 <- genData(N, def2)

est2[i] <- dt2[, mean(z)]

dt3 <- genData(N, def3)

est3[i] <- dt3[, mean(z)]

}

# N(2.25, 1)

c(mean(est1), sd(est1))

## [1] 0.016525503 0.001128918

# N(2.25, .16)

c(mean(est2), sd(est2))

## [1] 0.0165230677 0.0005924007

# Uniform(2, 2.5)

c(mean(est3), sd(est3))

## [1] 0.0165394920 0.0001643243

In each case, the average \(\hat{p}\) is 0.0165, and the standard errors are all below the standard MC standard error of 0.0040. The estimates based on draws from the uniform distribution are the most efficient, with a standard error below 0.0002.