Methodical Monte Carlo Experiments

Paul E. Johnson[1] [2]

[1]Department of Political Science

[2]Center for Research Methods and Data Analysis, University of Kansas

Statistical Computing

Outline

1. Define MC Simulation
2. Replication
3. Development Advice
4. Conclusion

Outline

1 Define MC Simulation

2 Replication

3 Development Advice

4 Conclusion

One Meaning: Simulated Sampling Distributions

- See my survey essay, "Monte Carlo Simulation in Academic Research"
- Here's the basic idea.
  - Consider a "statistical procedure" that receives a data set X and returns a result $\hat{\beta}_X$.
  - Presumably, there is a "true value" β that the estimate $\hat{\beta}_X$ is supposed to represent
  - We wonder, is that procedure desirable?
    - Are the estimates in $\hat{\beta}$ close to the "true values β"? Is $\hat{\beta}$ "unbiased"?
    - Is the sampling distribution of $\hat{\beta}$ "symmetric" or "consistent" (or ...whatever).

Conduct the MC Experiment

- Create a simulated sequence of statistically equivalent data sets $\{X_1, X_2, X_3, \ldots, X_m\}$
  - Statistically equivalent means the "true parameters" β are the same for each set
  - These data sets come from a data-generating function (the "population") f that has features that are more-or-less consistent with the structure pre-supposed by "the procedure"
- Apply "the procedure" to each one, generate an ensemble of estimates $\{\hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_m\}$
- Evaluate: Is "the procedure" accurate, unbiased, efficient, and so forth.
  - make histograms, calculate summary statistics, try to find out if the estimates are some how reflective of the "true parameters"

Variations Within the Paradigm

- Maintain the assumption that the data generating procedure matches (more or less) the hypotheses of "the procedure"
- But, Adjust characteristics within that framework
  - Adjust the data set's size
  - Adjust the variables in X in some "usual way".
- Create a variety of "true data generating" conditions $\beta_1, \beta_2, \ldots$ and find out if "the procedure" behaves differently

We Usually Need 2 Procedures

- We are considering "the procedure" against "the truth," but usually we are comparing it against "an old procedure".
- Thus, we need to write code for 2 procedures
- We wonder not only is
  - "the procedure" good (reflecting β),
- but also
  - is "the procedure" less bad than "the old procedure".

Variation: Damage the Simulated Data On Purpose

- "The procedure" states assumptions about the data generating process and the parameters.
- What if we intentionally generate *the wrong data?*
- We want to find out if "the procedure" is ruined by a mismatch between the "true data generating process" and that which is supposed by "the procedure."
  - Is "the procedure" *robust* against "mis-specification"?
- Bad terminology: "population misfit". I think the "population" is never wrong, it is always whatever it is.
- Good terminology: "model mis-specification" or "model misfit." The process assumed by the model is not correct, but the model may still be "useful" or "reasonably useful".

Outline

1 Define MC Simulation

2 Replication

3 Development Advice

4 Conclusion

We Require Replication

- Replicatability: regenerate results exactly (not just on average)
- We must be able to EXACTLY create any one of them.
- We don't have storage to "save" all those data sets on disk
- Consider difficulties of saving numbers in full machine precision

Pseudo-Random Number Generator

- PRNG: a formula that generates seemingly random streams of integers
- In R, one simple step is to set the default random generator's seed

```
set.seed(12354)
```

- From a single integer, the generator can intialize itself in a replicable way
- After that, the command rnorm(1) will generate the exact same value EVERY TIME.

For MC, set.seed() is Not Sufficient

Argument 1. we want to separately draw 1000s of data sets. We need a separate seed for each one in isolation.

- Suppose you run a project for 1 week and one run, say 989, is "wrong"
- You need to find out what happened
- Need to re-generate 989 EXACTLY to figure it out!

For MC, set.seed is Not Sufficient

Argument 2. we may need several separate streams of random numbers within a given set.

- You want to have a replicatable stream for 1000 children in a school, but
- A separate adjustable stream for the teachers

For MC, set.seed is Not Sufficient

Corollary: Consider this weird effect that happens when we add rows to a matrix.

- Draw vectors of 100 each, filling a matrix in column-major order

row	set 1	set 2	set 3	...
1	44	3	14	
2	12	22	13	
3	32	52	55	
4	53	23	21	
5	33	32	1	
6	45	23	44	
7	42	66	5	
...				
100	32	99	3	

- Then we want to add add 5 observations to each one.

Adding Rows

- Suppose there are 1000 columns
- If this were written in R, that means we'd change from this:

```
n <- 100
m <- 1000
dat1 <- rnorm(n*m)
X <- matrix(dat1, nrow = n)
```

to

```
n <- 105
m <- 1000
dat1 <- rnorm(n*m)
X <- matrix(dat1, nrow = n)
```

- Re-running from start, changing row count from 100 to 105 has effect of
  - moving draws 101-105 from set 2 into set 1.
  - And then 206-215 from set 3 into set 2

Additional Rows

row	set 1	set 2	set 3	...
1	44	3	14	
2	12	22	13	
3	32	52	55	
4	53	23	21	
5	33	32	1	
6	45	23	44	
7	42	66	5	
...				
100	32	99	3	

$\Longrightarrow$

row	set 1	set 2	set 3	...
1	44	23		
2	12	66		
3	32			
4	53	⋮		
5	33	14		
6	45	13		
7	42	55		
...		21		
100	32	1		
101	3	44		
102	22	5		
103	52			
104	23			
105	32			

We Probably Want

- Keep the SAME random draws at the beginning of each set, and then just add more random rows at the end of each one.
- Could have filled the matrix in row-major order, but that causes complications in replicating just one column at a time

About Random Record Keeping in R

- Chambers Software For Data Analysis says we should keep track of the random generator's state so we can pick up the generator whenever we want and get "the next" number.
- p. 230 describes "SoDA" package's "simulationResult" function.
- simulationResult has effect of running a simulation, but it keeps as recorded values the state of the random number generator at the start and at the end of the function.
- Read the code for "simulationResult", it is simply using an S4 class to achieve the effect of the following function.
- One example below steals that idea and uses it

Just Keep That Problem In Mind

- My Opinion: No Monte Carlo simulation is meaningful or useful it if cannot be replicated, exactly
- Careful design is necessary to make sure each separate part can be repeated whenever necessary.
- Am working on an example "parallel seeds" (Look in the hpcexample archive, Ex81-ParallelSeedPrototype)

Outline

1 Define MC Simulation

2 Replication

3 Development Advice

4 Conclusion

My Best HOWTO Advice

- DO NOT:
  - Think of the MC experiment as "One Giant Sequential Script" of commands
  - Generate a massive block of data that needs to be saved and re-loaded
- DO:
  - Create a simulation made up of separate functional components
  - Reproducibly Generate Data for one "run" of the simulation.
  - Design a function that accepts 1 data set and analyzes it.
  - Design procedure to repeat steps 1 and 2
  - Harvest estimates, summarize the results

Don't Let your code look like this

```
for(i in 1:10000){
  for(j in 1:100){
    #thousands and thousands of lines of
       unstructured crap in here
  }
}
```

Do make your code look like this

```
myFn1 <- function(a, b, c, d){
    ## return something
}


myFn2 <- function(e, f, ob1){
    ## return something
}

doOneRun <- function(rep, p1, p2, p3){
  ob1 <- myFn1(a=p1, b=p2, c=p3)
  ob2 <- myFn2(e=p1, f=p3, ob1 = ob1)
  ##do some calculations, return something
}
```

Do make your code look like this ...

```
alpha <- 234; beta <- 111; gamma <- 12
Nruns <- 1000
myOutput <- lapply(1:Nruns, doOneRun, alpha,
    beta, gamma)

## myOutput is alist of all of the runs
```

If you are a SAS Programmer, Please Change The Way You Think

- MC simulations in SAS will generally be designed to generate MANY data sets in one gigantic block, that looks like this:

run	id	x1	x2	x3	x4	x5
1	1	32				
1	2	11				
⋮						
2	1	42				
2	2	14				
⋮						

- Procedures in SAS generally have an option "by = run" to apply the procedure to each separate run

Problems with that SAS approach

- Requires storage of a monstrously huge data set
- Difficult/impossible to re-generate a particular run

Here's what NOT to do

- This is an understandable approach, one I have used with beginners
- The following generates all of the datasets and throws them into a list, and then it analyzes the list elements one-by-one.
- This is better than some approaches, but it does not "compartmentalize" the calculations for each separate run of the simulation.

Example: What NOT to do: Big List Of Dataframes

```
> s <- 10
> q <- 0.345
> stde <- 20
> set.seed(2341234)
> mydatasets <- vector("list", 100)
> for (i in 1:100) {
      x <- 18 + 43 * runif(1000)
      y <- s + q * x + rnorm(1000, mean = 0,
          sd = stde)
      mydf <- data.frame(x, y)
      mydatasets[[i]] <- mydf
  }
> myregressions <- lapply(mydatasets, function
   (mydf) lm(y ~ x, data = mydf))
```

Inspect That

- Output like this will test (and improve) your R data management skills
- This is worth studying because ALL simulations will generate a list of result items, and we ALWAYS have the problem of harvesting the parts we need.
- Select just the 33rd run for investigation.

```
> the33rdreg <- myregressions[[33]]
> attributes(the33rdreg)
```

Inspect That ...

```
$names
 [1] "coefficients"  "residuals"     "effects"
             "rank"          "fitted.values" "
    assign"
 [7] "qr"            "df.residual"   "xlevels"
             "call"          "terms"         "
    model"

$class
[1] "lm"
```

```
> summary(the33rdreg)
```

Inspect That ...

```
Call:
lm(formula = y ~ x, data = mydf)

Residuals:
    Min      1Q  Median      3Q     Max
-63.188 -12.509  -0.074  12.390  63.328

Coefficients:
             Estimate Std. Error t value Pr(>|t
                |)
(Intercept) 11.12946    2.09755   5.306 1
   .38e-07 ***
x            0.33875    0.05016   6.754 2
   .45e-11 ***
---
```

Inspect That ...

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0
   .05 '.' 0.1 ' ' 1

Residual standard error: 19.04 on 998 degrees
   of freedom
Multiple R^2: 0.0437,  Adjusted R^2: 0.04275
F-statistic: 45.61 on 1 and 998 DF,  p-value:
   2.445e-11
```

```
> sum33 <- summary(the33rdreg)
> attributes(sum33)
```

Inspect That ...

```
$names
 [1] "call"          "terms"           "
     residuals"      "coefficients"  "aliased"
           "sigma"
 [7] "df"            "r.squared"       "
     adj.r.squared" "fstatistic"     "
     cov.unscaled"

$class
[1] "summary.lm"
```

```
> coef(the33rdreg)
```

```
(Intercept)           x
 11.1294615   0.3387549
```

Inspect That ...

```
> coef(the33rdreg)[2]
```

```
        x
0.3387549
```

```
> coef33 <- coef(the33rdreg)
> coef(sum33)
```

```
             Estimate Std. Error  t value
                    Pr(>|t|)
(Intercept) 11.1294615 2.09754569 5.305945 1
   .380912e-07
x             0.3387549 0.05015926 6.753586 2
   .445331e-11
```

```
> coef(sum33)[2, 1]
```

Inspect That ...

```
[1] 0.3387549
```

```
> sum33$sigma
```

```
[1] 19.04095
```

```
> sum33$r.square
```

```
[1] 0.04370492
```

Collect and Summarize

```
> sestimates <- vector(length = 100)
> for (i in 1:100) {
      sestimates[i] <- coef(myregressions[[i
          ]])[1]
  }
> hist(sestimates, prob = T, xlab = "estimates
    of the intercept", s, main = "Sampling
   Distribution of Estimated Intercepts",
      xlim = 1.5 * range(sestimates))
> lines(density(sestimates), col = "red", lty
   = 2)
```

Critiques

- One giant list of datasets. Maybe too big!
  - May exhaust the RAM available
  - May be difficult to store
- Could we be more selective in saving parts of the simulation?
- One Advantage of this approach
  - If we save the list of simulated data frames, it is easy to investigate any particular one after the fact
  - But it may be difficult, or impossible, to re-simulate any particular data set to experiment with it.

Critiques

- I wish I could, in one step,
  - generate a dataset
  - analyze that one dataset
  - harvest highlights and keep them
- If we hope to parallelize, we want to divide work into separate parts
  - don't loop 100 times, then loop again 100 times
  - doing that would not separate the work among 100 systems

A Better Approach

- Step One.
  - Design a self contained function that can carry out one run of a simulation exercise.
  - It must explicitly accept arguments for any required information. (Assume nothing about environment)
  - Save run-time information so any exercise can be repeated
  - Save necessary results, throw away rest (since can replicate)

A Better Approach

- Step Two.
  - Use any of the R facilities you like to run that over and over again.
  - Collect results
    - Into a pre-allocated matrix if possible, or list of pre-determined size
  - Do not use idioms like

```
for(i in 1:1000){
oneRun <- myGiantFunction(run=i, x=324,y
    =234)
results <- rbind(results, oneRun)...
```

  - Repeated use of rbind will make this SLOW!

Other examples

- See the handouts that go with the Monte Carlo simulation review
  - monte-CLT.R
  - unbalancedTesting.R

Another Example of OLS Simulation

- This does the OLS exercise again, separating the work cleanly

Implement Chambers' Style Seed Record Keeping

```
doSomething <- function(whatever){
  inState <- .Random.seed
  results <- whatever code you want that draws
      random numbers
  outState <- .Random.seed
  list(inState, outState, results)
}
```

Example: OLS

```
> a <- 2
> b <- 5
> stde <- 3
> N <- 100
> Nexp <- 30
> set.seed(4343432)
> getPhonyData <- function(N, a, b, stde) {
      inState <- .Random.seed
      x <- rnorm(N, mean = 50, sd = 100)
      y <- a + b * x + rnorm(N, mean = 0, sd =
          stde)
      outState <- .Random.seed
      list(dat = data.frame(input = x, output
          = y), inState = inState, outState =
          outState)
```

Example: OLS ...

```
}
```

Note: Output is a list including one data frame, plus the in and out states of the generator

Create a function that analyzes an input data set

```
> analyzePhonyData <- function(dat) {
      mymod <- lm(output ~ input, data = dat)
  }
```

I wonder if I should return the regression and the data object grouped together, as in
list(mymod, dat)
or perhaps just
list(summary(mymod), dat).

Orchestrate One Run of the Exercise

```
> conductSim <- function(i) {
      d1 <- getPhonyData(N, a, b, stde)
      res <- analyzePhonyData(d1$dat)
      list(res = res, d = d1)
  }
```

Note Structure of Output Object

Output is a list that includes 2 list objects:

res: regression result object

d: another list containing

dat
inState
outState

- Should consider "throwing away" dat, since it could be regenerated from "inState"

Run that 100 times

```
> mysims <- lapply(1:100, conductSim)
```

Note: could have used replicate() instead

Students like For Loops Instead

Note: Reset system seed from saved state of first run

```
> .Random.seed <- mysims[[1]]$d$inState
> mysims2 <- vector("list", 100)
> for (i in 1:100) {
      mysims2[[i]] <- conductSim(i)
  }
> all.equal(mysims, mysims2)
```

```
[1] TRUE
```

Equivalence of collections confirmed

Reformat as "matrix" of "list elements"

```
> regs <- do.call("rbind", mysims)
> dim(regs)
```

```
[1] 100   2
```

```
> names(regs[[1, 2]])
```

```
[1] "dat"      "inState"  "outState"
```

```
> names(regs[[1, 1]])
```

Reformat as "matrix" of "list elements" ...

```
 [1] "coefficients"  "residuals"      "effects"
                "rank"           "fitted.values" "
    assign"
 [7] "qr"                "df.residual"   "xlevels"
                "call"           "terms"         "
    model"
```

```
> is.array(regs)
```

```
[1] TRUE
```

```
> is.array(regs)
```

```
[1] TRUE
```

Reformat as "matrix" of "list elements" ...

```
> is.list(regs[, 1])
```

```
[1] TRUE
```

So, for example, regs[[1,1]] returns the first regression object.

"Inspect the 33rd element in column 1"

So, for example, regs[[1,1]] returns the first regression object.

```
> summary(regs[[33, 1]])
```

```
Call:
lm(formula = output ~ input, data = dat)

Residuals:
     Min       1Q   Median       3Q      Max
-6.9005  -1.6788   0.2666   1.7737   8.2765

Coefficients:
             Estimate Std. Error t value Pr(>|t
                |)
(Intercept) 2.273297   0.371428    6.12 1
   .93e-08 ***
```

"Inspect the 33rd element in column 1" ...

```
input          5.002491   0.003373 1483.23  < 2
    e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0
    .05 '.' 0.1 ' ' 1

Residual standard error: 3.092 on 98 degrees
    of freedom
Multiple R^2:      1, Adjusted R^2:      1
F-statistic: 2.2e+06 on 1 and 98 DF,  p-value:
     < 2.2e-16
```

"lapply: Get Summaries For All Regression Objects"

lapply returns a list of regression summaries

```
> lsimres <- lapply((regs[, 1]), summary)
```

"sapply: Get Summaries For All Regression Objects"

Sapply returns a simplified structure of same regression summaries

```
> ssimres <- sapply((regs[, 1]), summary)
> is.matrix(ssimres)
```

```
[1] TRUE
```

```
> dim(ssimres)
```

```
[1]  11 100
```

```
> rownames(ssimres)
```

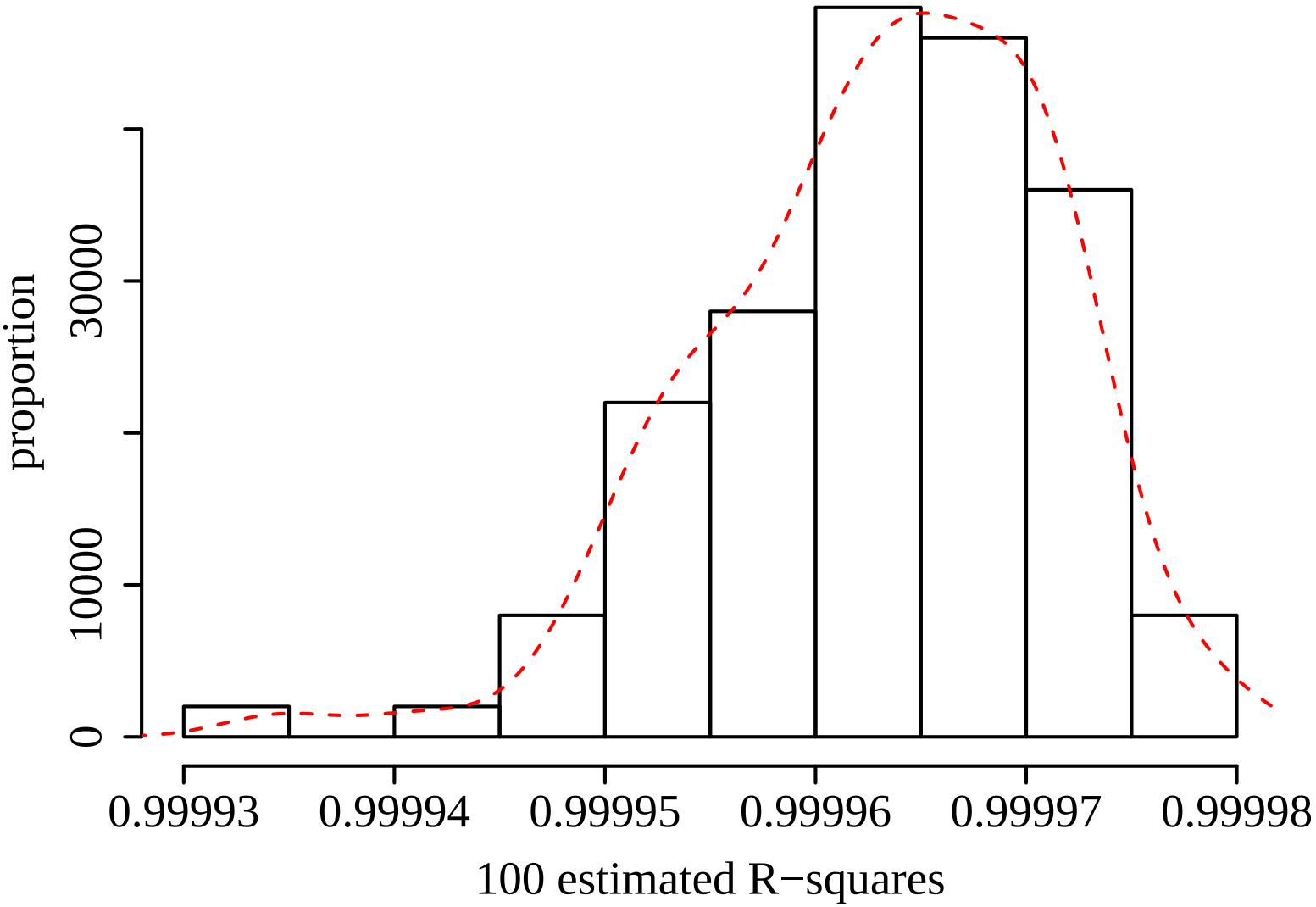
```
 [1] "call"          "terms"           "
    residuals"     "coefficients"  "aliased"
           "sigma"
 [7] "df"            "r.squared"       "
    adj.r.squared" "fstatistic"      "
    cov.unscaled"
```

"Want to plot some R^2s?"

- ssimres[8,] returns a list of 100 rsquares
- R's unlist function strips out the list structure, giving back a vector.

```
> rsquares <- ssimres[8, ]
> ursquares <- unlist(rsquares)
> hist(ursquares, xlab = "100 estimated
    R-squares", prob = T, ylab = "proportion",
        main = "")
> lines(density(ursquares), col = "red", lty =
    2)
```

The R-square Histogram



Speed and rbind

- See my R/WorkingExamples/stackListItems.R code

Outline

1 Define MC Simulation

2 Replication

3 Development Advice

4 Conclusion

Aim for Decomposable Simulations

- blah blha
- blah blah

Parallel Seeds Project

- There's a working example now
- has been incorporated in one dissertation project already
- Here's the idea. Each "run" of a simulation happens in its own environment, which must be initialized so that the simulation draws a unique random number stream.
- Actually, each separate simulation might need several separate streams (and while we are at it, that is pretty easy to do).
- Current strategy: Save a seeds matrix, one row per simulation node (each includes seeds for as many streams as the user desires).