Stat 3701 Lecture Notes: Simulation

Charles J. Geyer

April 16, 2017

1 License

2 R

- The version of R used to make this document is 3.3.3.
- The version of the `rmarkdown` package used to make this document is 1.4.
- The version of the `knitr` package used to make this document is 1.15.1.

3 CRAN Task Views

CRAN has a task view on Probability Distributions.

This task view covers *a lot* of packages and functions.

4 Random Number Generators

4.1 Introduction

The random number generators (RNG) in the R core are good enough for most simulations. Like most RNG used in computing they are actually deterministic, not "truly random" (whatever that may mean). Some people emphasize this by calling them *pseudorandom*, but we won't bother with this pedantry.

R's RNG like most RNG are definitely not random enough for use in cryptography. The evil bad guys can predict the random number stream if they have enough past history and also have the algorithm (which, of course, they do because R is open source). The task view linked above says the CRAN package `randaes` provides a *cryptographically secure* RNG. This means the future of the random number stream is not predictable from its past, even though the algorithm is known, so long as the internal state of the RNG is kept secret. Such an RNG is overkill for most usage. If you are not doing encryption, you don't need it.

The CRAN package `random` provides purportedly truly random numbers provided by an internet service. This is, of course, horribly slow. Moreover, these "truly random" numbers do not pass all tests for randomness (as documented by a vignette in that package. So using that for statistics would be a really bad idea. They are generated by a nondeterministic algorithm, but more than that is needed to be a good RNG.

Parallel computing (running your simulation on more than one computer simultaneously) presents additional problems and there are special RNG for that. But we will leave that subject for future notes.

Except in parallel computing, your humble author always uses the R default RNG. That is all we will illustrate here.

4.2 Seeds

The internal state of the RNG is called the "seed" for short. When started at the same "seed" the RNG will always produce the same stream of random numbers.

R makes the seed available by storing it in the R global environment as an R object called `.Random.seed`. Note the leading dot. It is a UNIX convention that files having names beginning with dot are not shown by default, and R (following its predecessor S) also uses this convention.

```
ls()
```

```
## character(0)
```

```
ls(all.names = TRUE)
```

```
## character(0)
```

```
rnorm(1:5)
```

```
## [1] -0.8720484  1.2190856  0.2227369 -0.1565320  1.7592770
```

```
ls()
```

```
## character(0)
```

```
ls(all.names = TRUE)
```

```
## [1] ".Random.seed"
```

Any use of random numbers creates the object `.Random.seed` if it wasn't already there and stores the internal state of the RNG after the operation in it.

```
save.seed <- .Random.seed
rnorm(1:5)
```

```
## [1] -0.4080496 -1.0462756  1.2983659  1.8621153 -0.1365526
```

```
.Random.seed <- save.seed
rnorm(1:5)
```

```
## [1] -0.4080496 -1.0462756  1.2983659  1.8621153 -0.1365526
```

Saving and restoring `.Random.seed` makes random number generation repeatable.

There is no way for users to make up correct values of `.Random.seed`. One can either use the R function `set.seed` or just use the RNG, as illustrated above. Here we also illustrate the former.

```
set.seed(42)
rnorm(1:5)
```

```
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
```

```r
set.seed(42)
rnorm(1:5)
```

```
## [1]  1.3709584 -0.5646982  0.3631284  0.6328626  0.4042683
```

Using `set.seed` is more convenient, but you may have to save and restore `.Random.seed` if you want to repeat something that isn't at the beginning of your use of random numbers.

4.3 The Principle of Common Random Numbers

There are two reasons why repeatability is important.

- If no one can repeat your results, then there is no reason to believe your results are correct. So when you do "reproducible research", as the handouts about reproduciblilty illustrate, you need to set RNG seeds (using `set.seed`) so the results are the same every time the document is rendered.
- Even within one study, competing methods should be compared on *the same random data*. Otherwise the comparison is not fair. Moreover, this can reduce the error due to randomness in the simulation. It is the same principle as using paired comparisons or blocking. This is called the *principle of common random numbers*.

In order to use the principle one can either

- generate simulated data once, and then apply all methods to be compared to it, or
- generate simulated data each time it is used, resetting the RNG seeds to assure that the same data are generated each time.

The latter is very useful when the data generated are too large to fit in R, but it can be convenient in other cases.

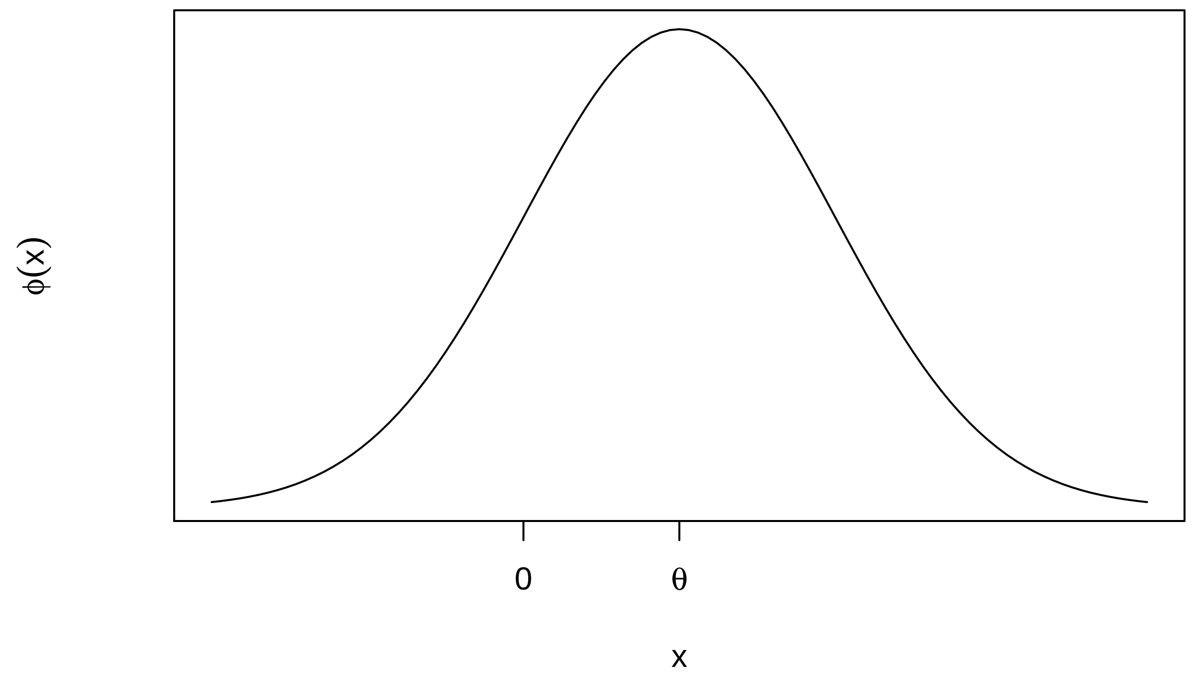
5 Example: Normal with mean θ and variance θ^2

5.1 The Model

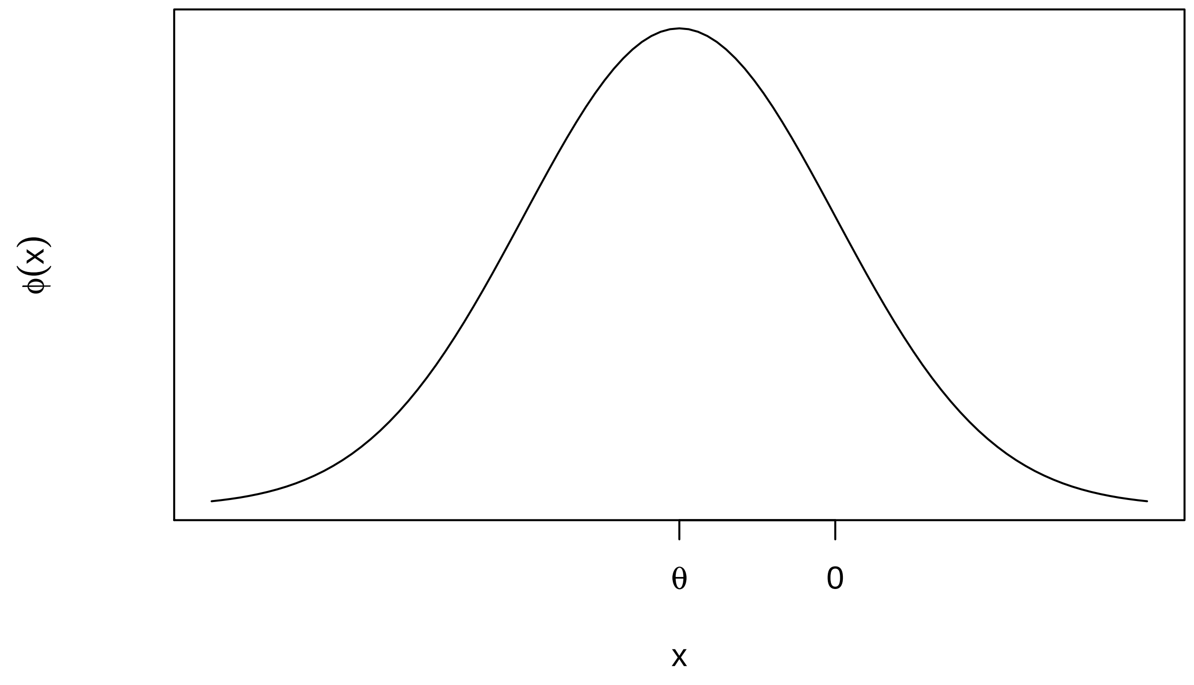
This was a model for practice problems 5. We learned a lot about it, but it all depended on asymptotics. What if the sample size isn't large enough for asymptotic approximation to be good?

If we find out what the properties of estimators are by simulation, then we don't need normal approximation.

A little thought says every distribution in this model is just like every other. The PDF is



or perhaps the mirror image (when θ is negative)



So really one simulation tells us everything there is to know about this model. We might as well use $\theta = 1$ as any other value. The estimators of θ we wish to compare are

- the sample median,
- the sample mean,
- the sample standard deviation times the sign of the sample mean, and
- the MLE.

5.2 Constants

We will use these constants for our simulation.

```
n <- 10
nsim <- 1e4
theta <- 1
```

Here `n` is the sample size of the data to which each method is applied. Since it is small, there is no reason to believe that asymptotic analysis is correct. Here `nsim` is the number of simulated data sets we use. We will estimate theoretical probabilities and expectations by averaging over these simulations.

5.3 Doing the Simulation

In order to avoid repetition of code we write one R function to apply each method to our simulated data. To illustrate setting seeds to obtain common random numbers, we generate our random data in this function.

```
doit <- function(estimator, seed = 42) {
    set.seed(seed)
    result <- double(nsim)
    for (i in 1:nsim) {
        x <- rnorm(n, theta, abs(theta))
        result[i] <- estimator(x)
    }
    return(result)
}
```

This function applies an estimator coded up as an R function by the user (in this case us) to each random data set generated.

```
theta.hat.median <- doit(median)
```

Simple. The rest are a bit more complicated.

```
save.seed <- .Random.seed
theta.hat.mean <- doit(mean)
all(save.seed == .Random.seed)
```

```
## [1] TRUE
```

We have checked that not only did we start at the same RNG seed, we also finished at the same RNG seed. This defends against the possibility that some R function other than `rnorm` that we call (perhaps indirectly) also uses random numbers, so we would not be using the principle of common random numbers.

```
theta.hat.sd <- doit(function(x) sd(x) * sign(mean(x)))
all(save.seed == .Random.seed)
```

```
## [1] TRUE
```

Coding up maximum likelihood is the most complicated of all. In fact, it was even more complicated than this, as admitted in Section 5.8 below.

```r
logl <- function(theta, x) sum(dnorm(x, theta, abs(theta), log = TRUE))
mle <- function(x) {
    if (all(x == 0))
        return(0)
    # assume theta > 0
    lwr <- sd(x) / 10
    upr <- 10 * sd(x)
    oout.plus <- optimize(logl, interval = c(lwr, upr),
        maximum = TRUE, x = x)
    while (oout.plus$maximum == lwr) {
        cat("oout.plus$maximum == lwr\n")
        upr <- 10 * lwr
        lwr <- lwr / 10
        oout.plus <- optimize(logl, interval = c(lwr, upr),
            maximum = TRUE, x = x)
    }
    while (oout.plus$maximum == upr) {
        cat("oout.plus$maximum == upr\n")
        lwr <- upr / 10
        upr <- 10 * upr
        oout.plus <- optimize(logl, interval = c(lwr, upr),
            maximum = TRUE, x = x)
    }
    # assume theta < 0
    upr <- (- sd(x)) / 10
    lwr <- 10 * (- sd(x))
    oout.minus <- optimize(logl, interval = c(lwr, upr),
        maximum = TRUE, x = x)
    while (oout.minus$maximum == lwr) {
        cat("oout.minus$maximum == lwr\n")
        upr <- lwr / 10
        lwr <- 10 * lwr
        oout.minus <- optimize(logl, interval = c(lwr, upr),
            maximum = TRUE, x = x)
    }
    while (oout.minus$maximum == upr) {
        cat("oout.minus$maximum == upr\n")
        lwr <- 10 * upr
        upr <- upr / 10
        oout.minus <- optimize(logl, interval = c(lwr, upr),
            maximum = TRUE, x = x)
    }
    # now compare
    if (oout.plus$objective > oout.minus$objective) {
        return(oout.plus$maximum)
    } else {
        return(oout.minus$maximum)
    }
}
theta.hat.mle <- doit(mle)
all(save.seed == .Random.seed)
```

```
## [1] TRUE
```

Examination of the log likelihood shows that, except in case all of the x_i are zero, which can only happen with positive probability when $\theta = 0$,

```r
rnorm(5, 0, 0)
```

```
## [1] 0 0 0 0 0
```

the log likelihood goes to $-\infty$ as $\theta \to 0$.

Because this is a one-dimensional problem, we should use the R function `optimize`, but it requires an interval in which to search, and it may get confused if we give it an interval containing zero.

Hence we calculate separate results for θ positive and θ negative and return the one with the highest log likelihood.

Moreover, we check to see whether the solution found is in the interior of the interval, because if it isn't, then it is not an MLE. That accounts for the `while` loops, which redo with a different interval, if necessary. (The `cat` statements show it is never actually necessary, but we didn't know that when we wrote the code.)

5.4 Looking at the Simulated Sampling Distributions

We use the R recommended package `KernSmooth` (which is installed by default in all installations of R) to do nonparametric density estimation. Density estimation has its own theory, but we won't bother to discuss it. Rather we will just consider it a useful visual tool for looking at sampling distributions that go with samples we have, one that is clearly better than histograms.

The point is that we can just see the sampling distributions. This may help some people get a better feel for sampling distributions. If you simulate them, you can actually see them. When you do not simulate them, you have to imagine them.

Note that we use both colors and line types to distinguish the curves on the plot. This doesn't look as pretty as using `lty = 1` (the default) for all curves, but it does enable color blind viewers to distiguish the lines.

```r
library(KernSmooth)
```

```
## KernSmooth 2.23 loaded
## Copyright M. P. Wand 1997-2009
```

```r
b.median <- dpik(theta.hat.median)
d.median <- bkde(theta.hat.median, bandwidth = b.median)
b.mean <- dpik(theta.hat.mean)
d.mean <- bkde(theta.hat.mean, bandwidth = b.mean)
b.sd <- dpik(theta.hat.sd)
d.sd <- bkde(theta.hat.sd, bandwidth = b.mean)
b.mle <- dpik(theta.hat.mle)
d.mle <- bkde(theta.hat.mle, bandwidth = b.mean)
ylim <- range(d.median$y, d.mean$y, d.sd$y, d.mle$y)
plot(d.median, type = "l", xlab = "x", ylab = "probability density",
    ylim = ylim)
lines(d.mean, col = "red", lty = 2)
lines(d.sd, col = "darkgreen", lty = 3)
lines(d.mle, col = "blue", lty = 4)
legend(1.5, 2.0,
    legend = c("sample median", "sample mean", "signed sample sd", "MLE"),
    col = c("black", "red", "darkgreen", "blue"), lty = 1:4)
```

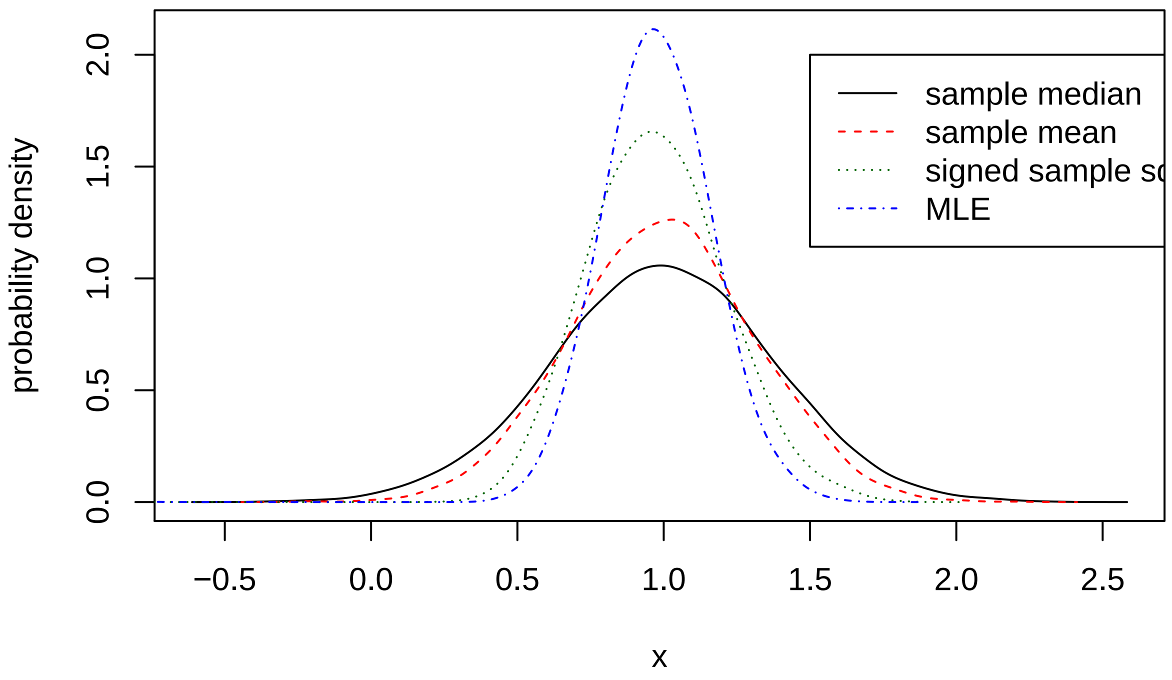


Figure 1: Density Estimates of Sampling Distributions

5.5 Output Analysis

Now we have for each estimator 10^4 simulations of its sampling distribution. One common criterion of goodness is mean square error (MSE), which is expected squared deviation from the parameter being estimated. If the estimator is unbiased (as the sample mean is), then this quantity is just variance. Otherwise, it is variance plus bias squared (so you will learn in the theory course).

Of course we don't know MSE, but from 10^4 simulations, we can get a pretty good idea. We can also get a pretty good idea of how far the average over our simulations may be from the exact theoretical MSE. This is called Monte Carlo error because Monte Carlo is another name for simulation.

The Monte Carlo standard error (MCSE) is just the standard deviation of quantity being averaged divided by the Monte Carlo sample size (MCSS), which is the number of simulations.

For example, for the sample mean, the MSE is

```
mse.mean <- mean((theta.hat.mean - theta)^2)
mse.mean
```

```
## [1] 0.09978038
```

```
mse.mean.se <- sd((theta.hat.mean - theta)^2) / sqrt(nsim)
mse.mean.se
```

```
## [1] 0.001420285
```

```
t.test((theta.hat.mean - theta)^2)
```

```
##
```

```
##  One Sample t-test
##
## data:  (theta.hat.mean - theta)^2
## t = 70.254, df = 9999, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.09699633 0.10256442
## sample estimates:
##  mean of x
## 0.09978038
```

	MSE	MCSE of MSE
sample median	0.1399	0.0020
sample mean	0.0998	0.0014
sample sd times sign of sample mean	0.0567	0.0013
MLE	0.0359	0.0011

We knew before we started that the MLE would be the best for large sample sizes. So it is not too surprising that it is also the best for $n = 10$.

We also knew before we started that the sample mean would be better that the sample median (for the normal distribution, course notes about models, part II, Section 3.2.4.3) for large sample sizes. So it is not too surprising that it is also the best for $n = 10$.

We had no idea before doing this simulation where the sample standard deviation would rank.

5.6 A Comment About this Output Analysis

The calculation of MSE for the sample mean is rather stupid. We know from intro stats

$$
\begin{aligned}
E(\overline{X}_n) &= \mu \\
\operatorname{var}(\overline{X}_n) &= \frac{\sigma^2}{n}
\end{aligned}
$$

and that these results are exact (not large sample approximations).

In this model we have $\mu = \theta$ and $\sigma^2 = \theta^2$. This makes $\overline{X}_n$ an unbiased estimator of θ, so MSE is $\operatorname{var}(\overline{X}) = \theta^2/n = 0.1$. And this result is exact, it does not need to be calculated by simulation.

Thus our table could have been

	MSE	MCSE of MSE
sample median	0.1399	0.0020
sample mean	0.1000	0.0000
sample sd times sign of sample mean	0.0567	0.0013
MLE	0.0359	0.0011

5.7 A Comment About Output Analysis, in General

If you ever see a simulation reported without MCSE (which may be called something else, perhaps "simulation error"), you are justified in ignoring those results completely. If the authors of the simulation couldn't care less what the accuracy of their numbers are, why should you trust anything they did?

5.8 Yet Another Comment About Output Analysis, in General

We had to go to extreme care when coding up the MLE function in order to assure that it always produces good results.

It is commonplace for lazy investigators to not bother to be so careful and simply throw away the results for which their method didn't seem to work. This is actually scientific fraud if done without admitting it was done and properly describing it, because it hides a very good reason to think the method not worth using. It is, of course, quite OK to do this and admit that one did it, but then readers have very good reason to think the method not worth using.

It is important to make methods work in all cases, if possible. The R function `glm` in the R core fails this test because it does not properly handle data for which parameter estimates go to infinity as the log likelihood is maximized. It sometimes produces a warning for such data, but also sometimes fails to produce a warning, and, moreover, gives the user no options to deal with such warnings.

Our first try at this note used the R function `nlm`, but we could not get it to always converge, so after several failed attempts, we switched to using the R function `optimize`, which is the current form of these notes.