Computer Intensive Statistics: APTS 2023–24 Computer Practical 1

From Simulation Basics to Monte Carlo Methods

Richard Everitt (richard.everitt@warwick.ac.uk)

May 2024

Contents

You probably won't have time in the practical to complete all of these problems so feel free to choose whichever you think most interesting.

If you're an expert in these things already, then you might find it interesting to investigate the potential that the `compiler` *and* `Rcpp` *packages have for accelerating even these simple routines — or that* `ggplot2` *has for improving the appearance of output.*

1. *A warm-up which also appeared in the preliminary material; if you've never implemented something like this before then this might be a useful preliminary step.*

   Transformation Methods: Recall the Box–Muller method which transforms pairs of uniformly-distributed random variables to obtain a pair of independent standard normal random variates. If

   $$U_1, U_2 \overset{\text{iid}}{\sim} \mathsf{U}[0,1],$$

   and

   $$X_1 = \sqrt{-2\log(U_1)} \cdot \cos(2\pi U_2),$$
   $$X_2 = \sqrt{-2\log(U_1)} \cdot \sin(2\pi U_2),$$

   then $X_1, X_2 \overset{\text{iid}}{\sim} \mathsf{N}(0,1)$.

   (a) Write a function which takes as arguments two vectors $(\mathbf{U}_1, \mathbf{U}_2)$ and returns the two vectors $(\mathbf{X}_1, \mathbf{X}_2)$ obtained by applying the Box–Muller transform elementwise.

   (b) The R function `runif` provides access to a PRNG. The type of PRNG can be identified using the `RNGkind` command; by default it will be a Mersenne-Twister (http://en.wikipedia.org/wiki/Mersenne_twister). Generate 10,000 $\mathsf{U}[0,1]$ random variables using this function, and transform this vector to two vectors each of 5,000 normal random variates.

   (c) Check that the result from (b) is plausibly distributed as pairs of independent, standard normal random variables, by creating a scatter plot of your data.

2. *The Bootstrap:* This question can be answered in two ways. The more direct (and perhaps more informative, if you have the time to do so and the inclination to implement a bootstrap algorithm from scratch) is to use the `sample` function to obtain bootstrap replicates, and to compute the required confidence intervals by direct means (hint: set `replace=TRUE`). More pragmatically, the `boot` library provides a function `boot` to obtain bootstrap samples and another, `boot.ci`, to provide various bootstrap confidence intervals.

   (a) The `Nile` dataset shows the annual flow rate of the Nile river. Use a histogram or other visualisation to briefly explore this data.

   (b) What are the mean and median annual flow of the Nile over the period recorded in this data set?

(c) Treating the data as a simple random sample, appeal to asymptotic normality to construct a confidence interval for the mean annual flow of the Nile.

(d) Using the `boot::boot` function to obtain the sample and `boot::boot.ci` to obtain confidence intervals from that sample, or otherwise, obtain a *bootstrap percentile interval* for both the mean and median of the Nile's annual flow. For the median you may also wish to obtain the interval obtained by an optimistic appeal to asymptotic normality combined with a bootstrap estimate of the variance (`boot::boot.ci` will provide this).

Note the following: `boot::boot` does the actual bootstrap resampling. It needs a function of two arguments to compute the statistic for each bootstrap sample: the first contains the *original* data and the second the index of the values included in a particular bootstrap resampling.

(e) Are there any interesting qualitative differences between the various confidence intervals? How does this relate to the data?

(f) Are your findings stable? If you repeat the bootstrap sampling do you recover similar behaviour?

(g) Are there any reasons to doubt the accuracy of these confidence intervals?

3. *Convergence of Sample Approximations*

(a) The `stats::ecdf` and `stats::plot.ecdf` functions compute and plot empirical distribution functions from a provided sample.

(i) Show plots of the empirical distribution function of samples of a variety of sizes ranging from 10 to 10,000 from a U distribution. Add to your plots the distribution function of the U distribution.

(ii) Repeat part *(i)* with a standard normal distribution.

(iii) Repeat part *(i)* with a Cauchy distribution.

(b) For each of the three distributions considered in the previous part, determine $\sup_x |\hat{F}_n(x) - F(x)|$ for each n considered (for simplicity, consider the sup over *only* the sampled values of x) and plot these quantities against n. Do you notice anything interesting?

4. *Transformation, Rejection and Importance Sampling:* The *Kumaraswamy distribution*[1] with parameters $a > 0$ and $b > 0$ describes a random variable with range $[0, 1]$. It has density:

$$f_X(x; a, b) = abx^{a-1}(1 - x^a)^{b-1} \text{ for } x \in [0, 1].$$

(a) Transformation Method

(i) Verify that the distribution function is $F_X(x) = 1 - (1 - x^a)^b$, and compute its inverse.

(ii) Implement a function which uses the inverse transform method to return a specified number of samples from this distribution with specified a and b parameters, using `runif` as a source of uniform random variables.

(iii) Use the `system.time`[2] function to determine how long your implementation takes to obtain a sample of size 100,000 from the Kumaraswamy distribution with $a = b = 2$. (You may wish to run it a few times and take the average.)

(b) Rejection Sampling

[1] Kumaraswamy, P. (1980). "A generalized probability density function for double-bounded random processes". Journal of Hydrology 46 (1-2): 79–88.

[2] `system.time` has poor rounding properties on Windows machines. A cheap alternative in Windows is to use `Sys.time` instead, which just records the current date and time. You can take the difference of the return value of this function at the beginning and the end of the period to be timed. The correspondence between the two timing methods is not fantastic. A better alternative is to use the `microbenchmark` package.

(i) Implement a function which uses rejection sampling to return a specified number of samples from this distribution with specified a and b parameters, $a, b \geq 1$, using `runif` as its sole source of randomness (i.e. using a U proposal distribution). What would happen if $a < 1$ or $b < 1$?

(ii) Use the `system.time` function to determine how long your implementation takes to obtain a sample of size 100,000 from the Kumaraswamy distribution with $a = b = 2$.

(iii) Modify your function to record how many proposals are required to obtain this sample; how do the empirical results compare with the theoretical acceptance rate?

(c) Importance Sampling

(i) Implement a function which uses a uniform proposal to return a weighted sample (i.e. both the sampled values and the associated importance weights) of size 100,000 which is properly weighted to target the Kumaraswamy distribution of parameters $a = b = 2$. Use the normalising constants which you know in this case.

(ii) Use the `system.time` function to determine how long your implementation takes to obtain a sample of size 100,000 targeting this distribution.

(iii) Produce a variant of your function which returns the self-normalised version of your weighted sample (this is easy; just divide the importance weights by their mean value after producing the original weighted sample).

(iv) Use the `system.time` function to determine how long your implementation takes to obtain a sample of size 100,000 targeting this distribution.

(d) Comparison

(i) The *inverse transformation* and *rejection* algorithms both produce iid samples from the target distribution and so the only thing which distinguishes them is the time it takes to run the two algorithms. Which is preferable? How many uniform random variables do the two algorithms require to produce the samples (this is, of course, a random quantity with rejection sampling, but the average value is a good point of comparison) and how does this relate to their relative computational costs?

(ii) To compare the importance sampling estimators with other algorithms it is necessary to have some idea of how well they work. To this end, use all four algorithms to estimate the expectation of X when $X \sim f_X(\cdot; a, b)$ using samples of size 100,000. (You might want to make the sample smaller if your implementation takes too long to produce a result with this sample size).

The algorithms which use iid samples from the target and the simple importance sampling scheme are unbiased and so we can use their variance as a measure of how well they perform. Noting that their variance scales with the reciprocal of sample size, an appropriate figure of merit is the product of the estimator variance and computational cost (cheaper schemes could be run for longer to reduce their variance without requiring any further computing).

We're interested here not in the variance of the target distribution—which we could easily estimate from a single sample—but in the *estimator variance*: a characterisation of the variability between repeated runs of our algorithms. This Monte Carlo variance tells us how much variability we introduce into the estimate by using Monte Carlo instead of the exact population mean. To characterise it, run each of your algorithms a large number of times, 100, say, obtain an estimate from each run and compute the sample variance of the collection of estimates you obtain.

How do the algorithms compare?

(iii) The self-normalized importance sampling scheme is biased and this further complicates the comparison.

First obtain its variance as you did for the other algorithms.

Now estimate its bias: what is the average difference between the estimates you obtained with this algorithm and the average result obtained with one of the unbiased schemes?

The mean squared error can be expressed as the sum of variance and bias squared.

Perform a comparison of the algorithms which considers MSE as well as computational cost.

5. *Simple Bayesian Inference*

Consider a scenario in which you wish to estimate an unknown probability given n realisations of a Bernoulli random variable of success probability p. You can view your likelihood as being Bin(n,p). The traditional way to proceed is to impose a Beta prior on p and to exploit conjugacy. Consider instead a setting in which you wish to use a Kumaraswamy distribution with $a = 3$ and $b = 1$ as a prior distribution (perhaps you're dealing with a problem related to hydrology).

(a) Develop a *simple* Monte Carlo algorithm which allows you to compute expectations with respect to the posterior distribution.

(b) Use your algorithm to compare the prior mean and variance of p with its posterior mean and variance in two settings: if $n = 10$ and you observe 3 successes and if $n = 100$ and you observe 30 successes.

(c) How does your algorithm behave if n is much larger (and we observe $3n/10$ successes)?

(d) How might you address this problem?