Outline: PRNGs and Permutations

Kellie Ottoboni

Draft May 31, 2016

1 Standard Implementations

The first order of business is to investigate how R and Python generate pseudo-random numbers and what algorithms they use to sample, permute, shuffle, etc. This may requires looking at the source code.

- There doesn't seem to be any way to see what `sample` does directly. However, someone has implemented a C++ version for RcppArmadillo at `https://github.com/RcppCore/RcppArmadillo/blob/master/inst/include/RcppArmadilloExtensions/sample.h`. Someone else has put the R base C source code in an unaffiliated GitHub repository at `https://github.com/SurajGupta/r-source/blob/master/src/main/random.c`. Here, we see that R's `sample` is broken down into four separate functions.w The functions sample by first randomly selecting indices for the new sample, then copying the original data into the return vector using the new indices.

    1. **Equal probability sampling with replacement:** for each of k samples, generate a uniform PRN; multiply it by the population size n and add one, then take the floor function. This also requires k uniform PRNs.
    2. **Equal probability sampling without replacement:** Start with a population of size n. For each of k samples, generate a uniform PRN; multiply it by the current population size $\tilde{n}$ and take the floor, then add one – this is the selected index; decrement the current population size $\tilde{n}$ by one and put the last remaining index in the place of the selected index. This only requires k uniform PRNs.
    3. **Unequal probability sampling with replacement:** Sort sampling probabilities and the corresponding indices in descending order; compute cumulative probabilities; for each of n samples, generate a random uniform and select the index j such that the random number is less than or equal to the jth cumulative probability but not the $j+1$st. This requires n uniform PRNs. OR Walker's alias method for large samples.
    4. **Unequal probability sampling without replacement:** Sort sampling probabilities and the corresponding indices in descending order; for each of n samples, generate a random uniform and multiply it by the total mass (starting with total mass 1); choose the index j such that the random mass is less than or equal to sum of the first j sampling probabilities but greater than the $j-1$st; is this explanation right? subtract the sampling probability for the jth element from the total mass, remove the jth element from the sampling probabilities and indices, and repeat. This also requires n uniform PRNs.

2 May 30, 2016: Thoughts on coding samples

Suppose the n elements of the population are in a canonical order. There is a one-to-one map between strings of at most n bits with exactly k nonzero bits and samples of size k. Since n and k are known, we can suppose without loss of generality that $k \leq n - k \leq n/2$ (otherwise, code the elements omitted from the sample instead of the included elements).

Code a sample by listing bits up to the kth nonzero bit, not inclusive. This completely determines the sample. If the code contains m bits, it specifies the first $k-1$ sampled elements, and then the $(m+1)$st element of the population is included in the sample. Elements $m+2, \ldots, n-1, n$ are not. This is a **prefix code**: no complete code is the beginning of another code.

Using this coding, the total number of bits needed to specify all possible samples:

- there is $1 = \binom{k-1}{0}$ sample that takes exactly $k-1$ bits to code, namely all 1s
- there are $k = \binom{k}{1}$ samples that take exactly k bits: the samples only omit one of the first k elements
- there are $\binom{k+1}{2}$ that take exactly $k+1$ bits
- there are $\binom{k+2}{3}$ that take exactly $k+2$ bits
- there are $\binom{k+\ell-1}{\ell}$ that take exactly $k+\ell-1$ bits. Equivalently, there are $\binom{\ell}{\ell-k+1}$ that take exactly ℓ bits.
- there are $\binom{n-2}{n-k+1} = \binom{n-2}{k-1}$ that take exactly $n-2$ bits
- there are $\binom{n-2}{k-1}$ samples that take exactly $n-1$ bits: they are exactly the $\binom{n-2}{k-1}$ codes of length $n-2$ bits with an additional 0 appended. They encode samples where the last element of the population is included.

Adding up all the ways to code samples, the number of bits for the entire code is

$$b = (n-1)\binom{n-2}{k-1} + \sum_{\ell=k-1}^{n-2} \ell \binom{\ell}{\ell-k+1}.$$

Theoretical lower bound by Shannon entropy:

$$H = \log_2 \binom{n}{k}.$$

3 May 17, 2016

- **Permutation testing with one sample:**
  We have N observations and we're doing some permutation test with them. To approximate the null distribution, we want to sample uniformly at random from all $N!$ permutations of the observations. Is it possible to obtain all of these permutations, or are we constrained by the period of the PRNG?
  - I am totally open to changing notation! This is temporary.
  - Suppose the period of the PRNG is $\mathcal{P}$.

  - Suppose the permutation algorithm takes K operations.
  - If $K \equiv 0 \mod \mathcal{P}$, then the PRNG will start over at some point. If $\mathcal{P}/K < N!$, then we can't reach all possible permutations. Otherwise, we're in good shape and we will just start to repeat permutations before the PRNG reaches the end of its period.
  - What happens if $\mathcal{P}/K < N!$ but K does not divide $\mathcal{P}$?

- What is the "best" way to do permutations to avoid reaching the end of the period? There are two issues at tension: the period of the PRNG and the computational complexity of the PRNG and shuffling algorithm. We want to balance computational efficiency with correctness.

- What happens if we generate pseudo-random numbers in a distributed fashion? Obviously one has to set the seed differently for each thread, but does this improve the risk of repeating?