Outline: PRNGs and Permutations

Kellie Ottoboni

Draft May 31, 2016

1 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:

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

2 May 17, 2016

- 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 require looking at the raw code since the R documentation doesn't say how `sample` works.
  - There doesn't seem to be any way to see what `sample` does. However, someone has implemented a C++ version for RcppArmadillo at `https://github.com/RcppCore/RcppArmadillo/blob/master/inst/include/RcppArmadilloExtensions/sample.h`. The functions here permute in place by first randomly selecting *indices* for the new sample, then copying the original data into the return vector using the new indices.
    1. `SampleReplace` calls `unif_rand()` and multiplies by the vector length.
    2. `SampleNoReplace` creates a vector of the numbers $1, \dots, N$, chooses a random index from the list and puts that into the `index` vector, then replaces the sampled element with the last and removes the last element.
    3. `ProbSampleReplace` does a sort in descending order of sampling probability. Then
- **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?