Notes: PRNGs and Permutations

Kellie Ottoboni

Draft June 20, 2016

1 June 20, 2016: Random Integer Generation

`numpy.random.randint` (from numpy version 1.11.0) is a function that generates uniformly distributed random integers. The user specifies lower and upper bounds, as well as the number of integers to generate and the data type of the result. Depending on the data type specified, the function uses a different way to generate the random integers. Options for the data type are `bool`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64`. The default data type argument is `numpy.int`, which is of type `int64`. The way that these random numbers are generated is in the C code at `https://github.com/numpy/numpy/blob/master/numpy/random/mtrand/randomkit.c` inside the function `rk_random_uint64`. Some binary operations are done to generate the random integers:

1. Take the smallest bit `mask` greater than or equal to the smallest power of two greater than the desired range by applying bitwise or operators. (For example, if the desired range is 35, then this sequence of operations will yield

$$\begin{aligned} mask &= 0100011 = 35 \\ mask &\leftarrow (0100011|0010001) = 0110011 = 51 \\ mask &\leftarrow (0110011|0011001) = 0111011 = 59 \\ mask &\leftarrow (0111011|0011101) = 0111111 = 63 \\ &\vdots \end{aligned}$$

   so `mask` $= 0111111 = 63$.)

2. Generate a random integer (32-bit integer if the range is less than the maximum unsigned long value that can be stored $(2^{32}-1)$, a 64-bit integer otherwise); apply bitwise and to choose the minimum of the random bit sequence and `mask` (for example, if we generate $19 = 0010011$ and `mask` $= 63 = 0111111$, then $19\&$`mask` $= 0010011 = 19$); repeat until the value generated is less than the range.

3. Add this value to the lower bound.

TO DO: It seems like this algorithm would give a higher probability of generating the upper bound. Are all integers actually equally likely?

R doesn't have a built-in uniform random integer generator. Typically, to obtain random integers between 1 and m, inclusive, one would generate a random uniform number X on $[0, 1)$, then take

$$Y = 1 + \lfloor mX \rfloor.$$

In theory, that's fine. But in practice, X is not really $U[0, 1)$ but instead is derived by normalizing something that's uniform on w-bit integers. Then, unless m is a power of 2, the distribution of Y isn't uniform on $\{1, \dots, m\}$. For $m < 2^w$, the ratio of the largest to smallest selection probability is, to first order, $1 + m^{2-w}$. For $m = 109$ and $w = 32$, $1 + m^{2-w} \approx 1.233$. That could easily matter.

2 June 19, 2016: Notes on Random_Sample Algorithm

The setting is using Random_Sample algorithm to generate a simple random sample of k items from n. First, let's show that $\mathbb{P}(i \in S)$ is equal for all $i = 1, \dots, n$. For $k = 1$, we're in the base case of sampling from $1, \dots, n$ with equal probability, so $\mathbb{P}(i \in S) = \frac{1}{n}$. Suppose that $\mathbb{P}(i \in S) = \frac{s}{n}$ for $s = 1, \dots, t-1$. We'll show that this is true for $s = t$ by induction. Let S^s denote the SRS generated by the algorithm at recursion depth s, for $s \le k$. That is, S^s is a SRS of size s from $\{1, \dots, n-s\}$. At the top level of recursion, we have

$$\begin{aligned}
\mathbb{P}(i \in S^t) &= \mathbb{P}(i \notin S^{t-1}, \text{choose } i) + \mathbb{P}(i \in S^{t-1}, \text{don't choose } i) \\
&= \mathbb{P}(i \notin S^{s-1})\mathbb{P}(\text{choose } i) + \mathbb{P}(i \in S^{s-1})\mathbb{P}(\text{don't choose } i) \qquad \text{by independent sampling} \\
&= \left(1 - \frac{t-1}{n}\right)\frac{1}{n} + \frac{t-1}{n}\frac{n-1}{n} \\
&= \frac{1}{n} + \frac{t-1}{n} \\
&= \frac{t}{n}
\end{aligned}$$

Suppose we consider subsets Ω instead of single elements. We hope that the following holds: if $|\Omega| = j$, then $\mathbb{P}(\Omega \subset S) = \frac{\binom{k}{j}}{\binom{n}{k}}$. Now we have induction in two dimensions: the cardinality of Ω and the recursion depth. TO DO: PROVE THIS.

3 June 6, 2016: Notes

There are several moving parts:

- **How are samples encoded?** We describe samples with a code, which is a sequence from a finite set of elementary symbols. If we use bits to code samples (1 if the item is selected, 0 if not), the Shannon lower bound on the number of words in the code is $H = \log_2 \binom{n}{k}$. If we use a different set of elementary symbols for our code, this lower bound will be different. In particular, if there are s symbols to choose from, then $H = \log_s \binom{n}{k}$.

- **What does the PRNG output?** Do we imagine that the PRNG gives a sequence of 32 bits? Does it give a number in the interval $[0, 1]$? Does it give us an integer on some interval? We need to be specific about this, as it will affect our counting of states. It might be the case that to generate a number in $[0, 1]$, it requires several random bits to be generated. **Answer:**

let's think of it as outputting a 32-bit integer. The Mersenne Twister is k-distributed to 32 bits for $1 \le k \le 623$.

- What happens when we reach the end of the PRNG's period, but the period is not a multiple of the number of PRNs needed to generate a sample?

Define $H_{nk} = \log_2 \binom{n}{k}$. Total number of bits from PRNG = $period \times integer - length$. This divided by H_{nk} is greater than or equal to the total number of samples of k of n without recycling/per period.

Do this calculation for LCG: put in a, b, m, gives upper bound on period. Then take n, k pair and do calculation. Do randU and Mersenne Twister as well.

3.1 Period length of linear congruential generator

(from Wikipedia)

The period of a general mixed congruential generator is at most m, and for some choices of factor a much less than that. The mixed congruential generator will have a full period for all seed values if and only if:

1. m and the offset c are relatively prime,
2. $a - 1$ is divisible by all prime factors of m,
3. $a - 1$ is divisible by 4 if m is divisible by 4.

These three requirements are referred to as the Hull-Dobell Theorem. While LCGs are capable of producing pseudorandom numbers which can pass formal tests for randomness, this is extremely sensitive to the choice of the parameters c, m, and a.

3.2 Factorial bounds

Bound on binomial coefficients

$$\frac{2^{nH(q)}}{n+1} \le \binom{n}{k} \le 2^{nH(q)}$$

where $H(q) = -q \log_2(q) - (1-q)\log_2(1-q)$, and $q = k/n$.

4 Standard Implementations

4.1 Psuedo-random number generators

At the time of writing this, Python (version 2.6 and up), R (version 3.3.0), SAS (version 9.4), SPSS (version 23.0), and Matlab (R2016a, version 9.0) use the Mersenne Twister as their default PRNG. Until April 2015, Stata used the KISS PRNG; the recent upgrade to version 14.0 introduced the Mersenne Twister. Excel 2003 used the Wichmann-Hill generator, but implemented it incorrectly; Excel 2007 attempted to fix the bug but failed (McCullough 2008). Now, Excel 2010 and more recent versions use the Mersenne Twister.

4.2 Sampling algorithms

- The naive implementation is to generate n random numbers between 1 and n (shuffle in place or permute), sort the observations according to the random index they're assigned, and take the top k to be the sample. This is inefficient and requires up to n PRNs to get the random ordering.

- Someone 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` (version 3.3.0) is broken down into four separate functions. 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 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.
  2. **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.
  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.

- The Python (version 2.7) `random` library source code is at `https://hg.python.org/cpython/file/2.7/Lib/random.py`.

  1. The random number generator functions rely on generating sequences of random bits. These can then be turned into integers or numbers on a continuous interval.
  2. `random.choice` takes a sequence as input, generates a random uniform number and multiplies it by the length of the sequence, and returns the element with the nearest integer index.
  3. `random.shuffle` implements the Knuth shuffle. The number of PRNs required is equal to the length of the sequence to shuffle.
  4. `random.sample` does equal probability sampling **without replacement**, like an SRS. There are two implementations. The first uses a discard pool to track which elements have been selected. This is the chosen method if the sample size to population size

ratio is large. This requires k PRNs and the algorithm matches R's implementation. The second algorithm does not discard elements that have already been sampled, but does sampling *with replacement.* At each step, the function checks if the selected item is already in the sample. If so, then it tries until it samples something not yet in the sample. This method requires *at least* k PRNs.

- The `numpy.random` library (version 1.11.0) source code is available on GitHub at `https://github.com/numpy/numpy/blob/master/numpy/random/mtrand/mtrand.pyx`.

   1. `np.random.choice` does sampling with/without replacement and with equal or unequal probabilities. This is the closest thing to R's `sample`.

      (a) **Equal probability sampling without replacement:** They do something close to the naive implementation: permute the indices 1 through n, take the first k of the permuted indices, and the units with these indices form the sample. This requires n random numbers to perform the Fisher-Yates shuffle, which gets called from the permutation function.

5 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 $m+2, \dots, n-1, n$ 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

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

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

Theoretical lower bound by Shannon entropy:

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

6 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?