Advanced Data Analytics
Lectorial 6: Random Number Generation

Ian T. Nabney

Summary

- Understand how an algorithm can generate a pseudo-random sequences of numbers.
- Understand the principles underpinning random number generation in Python.

This lectorial is about random number generation for statistical modelling: cryptography is another story entirely.
Further reading:

- A classic reference for this material is chapter 3 of 'The Art of Computer Programming' (vol. 2) by Donald Knuth. This also contains many useful tests for the quality of a given random number generator.
- There is also some useful information in chapter 7 of Numerical Recipes in C.

Fundamentals

- Random number generator algorithms generate a sequence of pseudo-random numbers. That is, the numbers are generated deterministically, but pass a wide range of statistical tests for randomness (e. g. zero autocorrelation).
- Everything is based on an algorithm that generates pseudo-random numbers uniformly over $(0, 1)$.
- The basic principle of these algorithms is that they generate a sequence of positive integers up to a maximum value N. These are converted into floating point numbers by dividing by N.
- Note that if you need more than about 5% of the period of a random number generator, then any flaws in it are much more likely to show up, so we want the sequence to be as long as possible.
- You can control where in the sequence you start by setting the seed or state of the random number generator.

Basic Algorithm

- Most practical random number generators are based on the linear congruential algorithm

$$I_{j+1} \equiv aI_j + c \pmod{m}$$

with a, c, and m positive integers.
- If these values are properly chosen, then the period of the generator will be m.
- This algorithm is very fast to compute and simple to program, which explains its popularity.
- The drawback of such generators is that successive calls are serially correlated (i.e. the sequence has non-trivial autocorrelation).
- Another form of correlation is that low order bits are much less random than high order bits.

Algorithm Details

- There is evidence that simple multiplicative congruential algorithms of the form

$$I_{j+1} \equiv aI_j \pmod{m}$$

  are as good as general linear congruences.
- NR recommends the coefficients $a = 7^5 = 16807$ and $m = 2^{31} - 1 = 2147483647$ for a 32-bit system.
- Note that 0 should not be used as a seed to such algorithms (why?). Either XOR the seed with some unlikely integer, or add 1.

Practicalities

If I_j is large, then aI_j will exceed the maximum representable value for a 32 bit integer. To prevent overflow, we use Schrage's algorithm to multiply two 32 bit integers modulo a 32 bit constant without using any intermediate values larger than 32 bits.

Write $m = aq + r$ with $0 < r < a$ and

$$q := \left[\frac{m}{a}\right].$$

If $r < q$ and $0 < z < m - 1$, then

$$a(z \pmod q) \quad \text{and} \quad r\left[\frac{z}{q}\right]$$

are in the range $0, \ldots, m-1$. We shall calculate

$$t = a \times (z \pmod q) - r\left[\frac{z}{q}\right].$$

Then we claim that

$$az \pmod m = \begin{cases} t & t \geq 0 \\ t + m & \text{otherwise} \end{cases}$$

This algorithm works as

$$z = z \bmod q + q \left[\frac{z}{q} \right]$$

and so

$$\begin{aligned} az &= a(z \bmod q) + aq \left[\frac{z}{q} \right] \\ &= a(z \bmod q) + (m - r) \left[\frac{z}{q} \right] \\ &\equiv t \pmod{m} \end{aligned}$$

For the values of a and m given above, we have $q = 127773$ and $r = 2836$. In our application $z = I_j$.

Shuffling the Numbers

- This generator fails some statistical tests if more than about 10^7 calls are made (that is just 10^5 calls for a model with 100 parameters).
- A simple way to remove these correlations is to randomly shuffle the output

1. Fill the table v_1 to v_{32} and y with random (long) integers.
2. The random number in y is used to pick a cell in v by calculating
$$\frac{y}{1 + (m-1)/32}$$
Note that we use the high order bits of y, not y mod 32.
3. The element chosen in step 2 is output and becomes the next value y.
4. The portable random number generator is called to refill the empty location. Then repeat from step 2 at the next call.

Combining Sequences

- To extend the period of a generator, another trick is to combine two generators with different periods: this combination has a period which is the lcm $(m_1 - 1, m_2 - 1)$.
- We can subtract one sequence from the other, modulo the modulus of either of them: this avoids overflow if we add the modulus back again when the result is ≤ 0.
- For example, if we choose

$$
\begin{aligned}
m_1 &= 2147483563 \quad (a_1 = 40014, q_1 = 53668, r_1 = 12211) \\
m_2 &= 2147483399 \quad (a_2 = 40692, q_2 = 52774, r_2 = 3791)
\end{aligned}
$$

then

$$m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031 \qquad m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789$$

and so the combined generator has period approximately 2.3×10^{18}.

Python Random Number Generator

- You are advised to use the RNG from numpy rather than the one in the core Python library.
- This uses a
  - BitGenerator: Object that generates random numbers. These are typically unsigned integer words filled with sequences of either 32 or 64 random bits.
  - Generator: Object that transforms sequences of random bits from a BitGenerator into sequences of numbers that follow a specific probability distribution (such as uniform, Normal or Binomial) within a specified interval.

Often these last will use the transformation method or similar (see the Sampling Methods lecture for more details). There are 37 different distributions defined.

Default Bit Generator

- Permuted Congruential Generator 64-bit (PCG64):
- The key idea is to pass the output of a fast well-understood "medium quality" random number generator to an efficient permutation function, built from composable primitives, that enhances the quality of the output.
- The PCG64 state vector consists of 2 unsigned 128-bit values, which are represented externally as Python ints.

This file is the original paper defining this algorithm.

Practical Implications

- Randomness (stochasticity) can be present in several different parts of a data science application:
    1. Randomness in data collection: use one fixed set of data.
    2. Effect of observation order: may want to shuffle data before each iteration.
    3. Randomness in machine learning algorithms: often associated with initialisation but also with tie-breaking.
    4. Randomness in sampling/resampling.
- Reproducibility is key: always set the seed of the random number generator.
- Use reliable implementations: it is very easy to go wrong.
- Report on experimental variability: error bars and significance tests.

Summary

- Understand how an algorithm can generate a pseudo-random sequences of numbers.
- Understand the principles underpinning random number generation in Python.