R Users Guide to Stat 201: Chapter 4

Michael Shyne, 2017

Chapter 4: Probability

Chapter 4 introduces the concepts and basic calculations of probability. Since the calculations can be as easily produced with a calculator, there is not much to do in R for this chapter. Instead, for this guide we'll discuss the basics of random number generation. Producing a number of random values is usually one of the first steps in simulating various probabilistic situations.

Random numbers, psuedo-random numbers and seeds

The random numbers generated by R, like with any computer language, are not true random numbers. They are what is called "pseudo-random" numbers. The numbers are produced by an algorithm which will generate numbers, one after another, that appear random, but are in fact deterministic. The algorithm is initialized by an integer, called a "seed". Each unique seed will generate a different sequence of random numbers. Since each time R is run, by default a different seed is provided the random number generator (perhaps based on the system clock), the numbers produced are virtually indistinguishable from truly random.

However, it is possible to set the seed manually. If you set the seed to the same value every time before generating a list of randoms numbers, you will get the same list every time. This can be useful when testing a new function or procedure, or when working in groups to verify all members of the group are getting identical results.

Details on the random number algorithms employed by R and seeds can be found with `?RNGkind`.

Generating random numbers

R has a variety of functions, with a consistent naming structure, that produce random numbers from different distributions. We will cover distributions in later chapters. The name of these functions will be `r` plus an abbreviation (usually) of the distribution name. For example, random numbers from a uniform distribution get be produced with `runif()` and from a normal distribution with `rnorm()`.

Uniform random values

The most straight-forward distribution to work with is the uniform distribution. This will produce numbers, not surprisingly, uniformly distributed between two values, 0 and 1 by default. These numbers can be used to simulate anything we think of has having "even" probabilities, i.e. coin flips, dice rolls, roulette wheels, etc.

The basic usage, to generate one random value, is...

```
runif(1)
```

```
## [1] 0.2998539
```

Of course, we are rarely going to want just one random number and R is usually just as comfortable with vectors of values, we can get many random numbers.

```
rn.12 <- runif(12)
rn.12
```

```
##  [1] 0.9163255 0.4531223 0.8161188 0.5361527 0.3855241 0.6871234 0.4292687
##  [8] 0.3025750 0.9695415 0.5587727 0.4477703 0.5131802
```

We can adjust the range of the values with optional parameters `min` and `max`.

```
runif(6, min=1, max=10)
```

```
## [1] 2.777383 3.338904 7.357181 5.182089 2.513086 8.564914
```

To simulate discrete events, like dice rolls, we'll create integers from the random values.

```
# Simulate 10 dice rolls
ceiling(runif(10)*6)
```

```
##  [1] 1 4 6 4 5 4 2 2 3 4
```

I would recommend the technique above (multiplying random numbers between 0 and 1 by the desired range), rather than adjusting the `min` and `max`. The later technique can be confusing and lead to hard to diagnose errors. For example, to generate dice rolls, you might be tempted to try

```
# Not correct
ceiling(runif(10, 1, 6))
```

```
##  [1] 6 4 4 3 4 4 2 3 4 5
```

However, since this applies the `ceiling` function to values > 1 and < 6, a one would never be generated. The proper use of this technique would be

```
# Correct, but confusing
ceiling(runif(10, 0, 6))
```

```
##  [1] 5 1 3 2 2 2 1 6 6 2
```

Once we have generated our vectors, we can treat them like data from any other source. We can calculate statistics and compare values to what we expect.

```
# The expect mean of the default uniform distribution is 1/2.
#  As sample sizes get bigger, observed means should get closer
#  to expected means.
mean(runif(2))
```

```
## [1] 0.7690611
```

```
mean(runif(10))
```

```
## [1] 0.346592
```

```
mean(runif(100))
```

```
## [1] 0.48936
```

```
mean(runif(10000))
```

```
## [1] 0.4927482
```

Note: At the time I am writing this, I don't know what random numbers are being generated. The values will be different every time I compile, or "knit", this document. I don't know, for example, if my claim of means converging is actually demonstrated. I've seen both versions produced. However, if I set the seed the document will be predictable, or rather reproducible.

```
# Set the seed
set.seed(42)
```

```r
# "Randomly" generate 0.914806
runif(1)
```

```
## [1] 0.914806
```

Every version of this document will now produce the number 0.914806. Of course, I didn't know what the value would be prior to knitting the guide the first time, but having run it once, the result will be consistent. Incidentally, every random number from here on will be similarly predictable, unless I reset the seed to an arbitrary value. Like so,

```r
# Set the seed based on the system clock
set.seed(as.integer(Sys.time()))

# Once generated 0.4623988
runif(1)
```

```
## [1] 0.3126847
```

Bernoulli random values

Bernoulli random variables are beyond the scope of this class, but simply put, they model binary outcomes. A Bernoulli distribution will produce 1's, often described as "successes", with a frequency defined by the parameter π, and 0's otherwise. Thus, they can be used to model coin flips, results of screening tests or anything else with exactly two outcomes.

R does not have a Bernoulli random number function, but we can easily write one with a method similar to generating dice rolls above.

```r
# p is the probability of success, default to 1/2
rbern <- function(n, p=0.5){
    return (ceiling(runif(n) - (1-p)))
}

# Get 10 coin flips, 1 = heads, 0 = tails
flips <- rbern(10)
flips
```

```
##  [1] 0 0 0 1 0 1 1 1 0 0
```

```r
# Simulate a weighted coin
flips.1000 <- rbern(1000, p=.7)

# Since our values are 1 and 0, the proportion of
#  successes can be calculated by the mean.
mean(flips.1000)
```

```
## [1] 0.692
```

Binomial random values

Most of the time, the interest in a series of Bernoulli trials isn't in the individual values, but rather the total number of successes. This situation is modeled by the binomial distribution and R does have functions for it. In addition to the probability of success parameter, the binomial also has a size parameter, that is, the number of Bernoulli trials conducted. Thus, to repeat the simulation above of flipping a weighted coin a thousand times with the binomial distribution...

```r
rbinom(1, size=1000, p=.7)
```

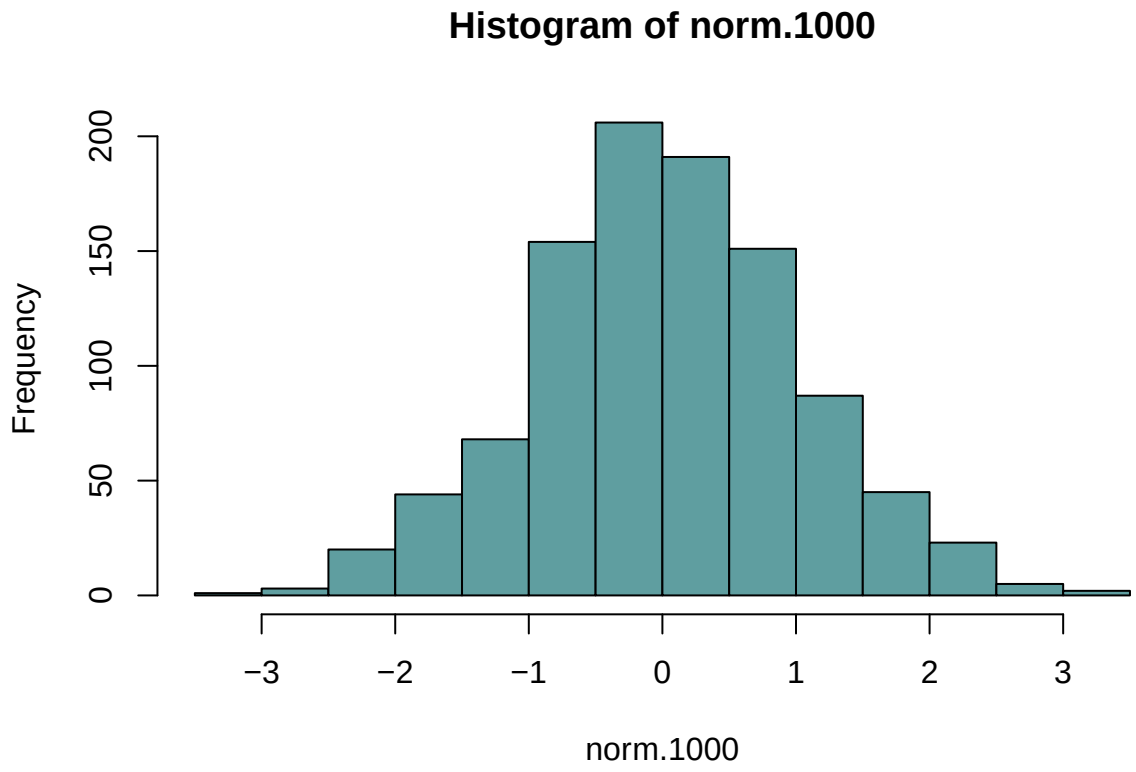
```
## [1] 705
```

Normal random values

Random values can be generated from more complex distributions where the values are not evenly distributed. The normal distribution is a common source of such numbers.

```r
# Generate 1000 random values from a standard normal distribution
norm.1000 <- rnorm(1000)

# A histogram should produce a nice normal shape
hist(norm.1000, breaks=10, col="cadetblue")
```



License