Lab 5: Simulating discrete probability models

Spring 2020

Task 1: Roulette wheel simulation

A roulette wheel has 38 slots of which 18 are red, 18 are black, and 2 are green. If a ball spun on to the wheel stops on the color a player bets, the player wins. Consider a player betting on red. Winning streaks follow a Geometric(p = 20/38) distribution in which we look for the number of red spins in a row until the first black or green. Use the derivation of the Geometric distribution from the Bernoulli distribution to simulate the game. Namely, generate Bernoulli(p = 20/38) random variates (0 = red; 1 = black or green) until a black or green occurs.

Code set-up

A `while` loop allows us to count the number of spins until a loss. If we use indicator variable `lose` to note a win (1) or loss (0), the syntax is "while we have not lost (i.e., `lose==0`), keep spinning." Once you win, the while loop ends and the variable `streak` has counted the number of spins. Try running a few times.

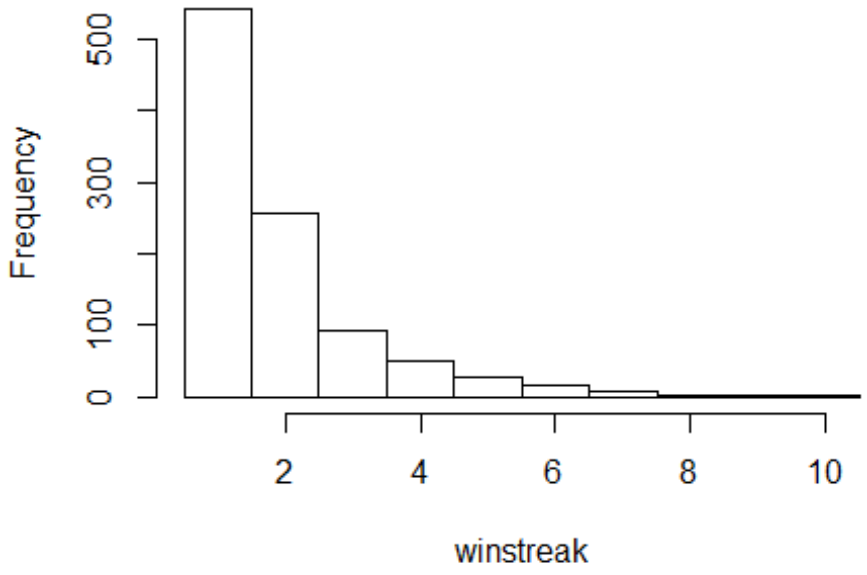
```
streak = 0
lose = 0
p = 20/38
while(lose==0){
  lose = (runif(1) < p) # generate Bernoulli with probability p
  streak = streak + 1 # tally streak
}
streak

## [1] 3
```

The problem

The code chunk above performs the experiment once: spin the roulette wheel until you lose and record the number of spins. Simulate 1000 experiments. As usual, do this by wrapping the code chunk above within a for-loop and storing the number of spins `streak` in a vector.

```
simnum = 1000;
winstreak <- numeric(simnum)
for (i in 1: simnum) {
streak = 0
lose = 0
p = 20/38
while(lose==0){
  lose = (runif(1) < p) # generate Bernoulli with probability p
  streak = streak + 1 # tally streak
```

```
}
winstreak[i] = streak
}

mean(winstreak)

## [1] 1.887

sd(winstreak)

## [1] 1.363852

max(winstreak)

## [1] 10

hist(winstreak, br=seq(min(winstreak)-0.5, max(winstreak+0.5)), main="")
```



Report the following:

- Histogram of the win streak length. Note that this is a discrete distribution so should place histogram bars at discrete values {0, 1, 2, ...}. This may be done with the `breaks` option within `hist`. If your storage variable is called `winstreak`:

```
hist(winstreak, br=seq(min(winstreak)-0.5, max(winstreak+0.5)), main="")
```

- Average length of the win streak. *1.887*
- Standard deviation of the winning streak lengths. *1.36*

- Compare the empirical average and standard deviation in the previous two bullets to the true values from the Geometric($p = 20/38$) distribution.

The empirical averages are accurate when compared to the true values.

- Longest winning streak. *10*

Task 2: Simulating negative binomial distributions

In this task, we will compare two different algorithms for simulating from a negative binomial distribution.

Problem (a)

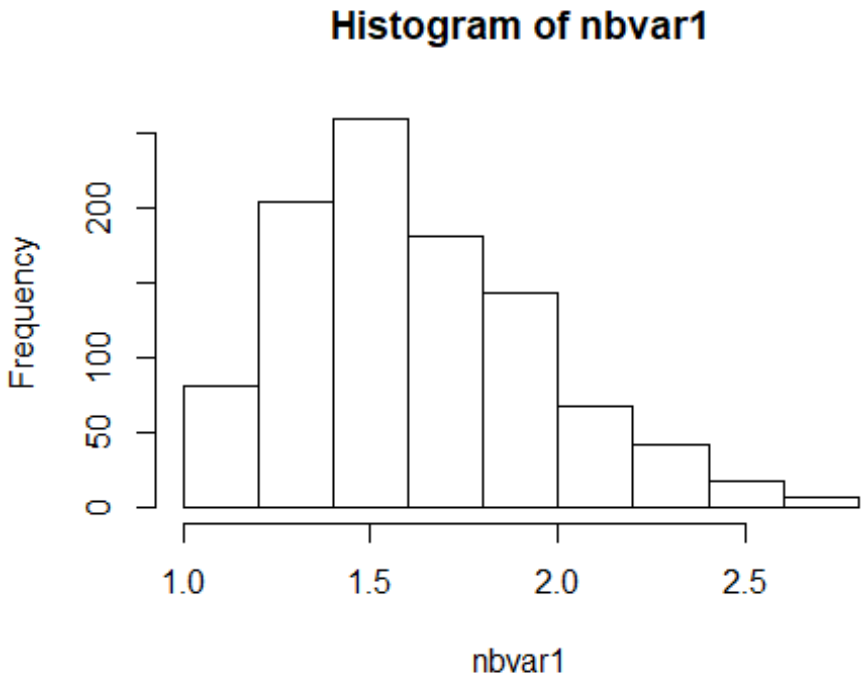
Recall that a negative binomial random variable *NB(r, p)* is the sum of *r Geometric(p)* random variables. Use the algorithm from Task 1 to simulate 1000 *NB(10, 0.6)* random variates.

Code set-up

Note that we merely need to wrap the core code from Task 1 within a for-loop. Here is the core of the code chunk, where we are thinking of a for-loop over a variable `sims` to replicate the single negative binomial draw. Note that this code chunk will not run since the for-loop over `sims` is not coded, thus the `eval=FALSE` option. **Note that this code has the `eval=FALSE` option just to present the code without output. Your code will not use this option.**

```
r = 10
p = .6
nbvar1 <- numeric(1000)
nbvar2 <- numeric(10)
x = proc.time()
for (sims in 1:1000){
for(nbsims in 1:r){
  # for-loop allows us to simulate until r successes;
  # in this problem, r=10 and p=0.6
  tossnum = 0
  success = 0
  while(success==0){
    success = (runif(1)<p)
    tossnum = tossnum + 1
  }
  nbvar1[sims] = nbvar1[sims] + tossnum
  nbvar2[r] = nbvar1[sims]
  }

  nbvar1[sims] = mean(nbvar2)

}
```

```
timer = proc.time() - x
algtime = timer[3]
algtime

## elapsed
##    0.25

mean(nbvar1)

## [1] 1.6673

sd(nbvar1)

## [1] 0.3385489

hist(nbvar1)
```



Problem (b)

The negative binomial pmf induces the following recursion relation. If $X \sim B(r,\ p)$, then

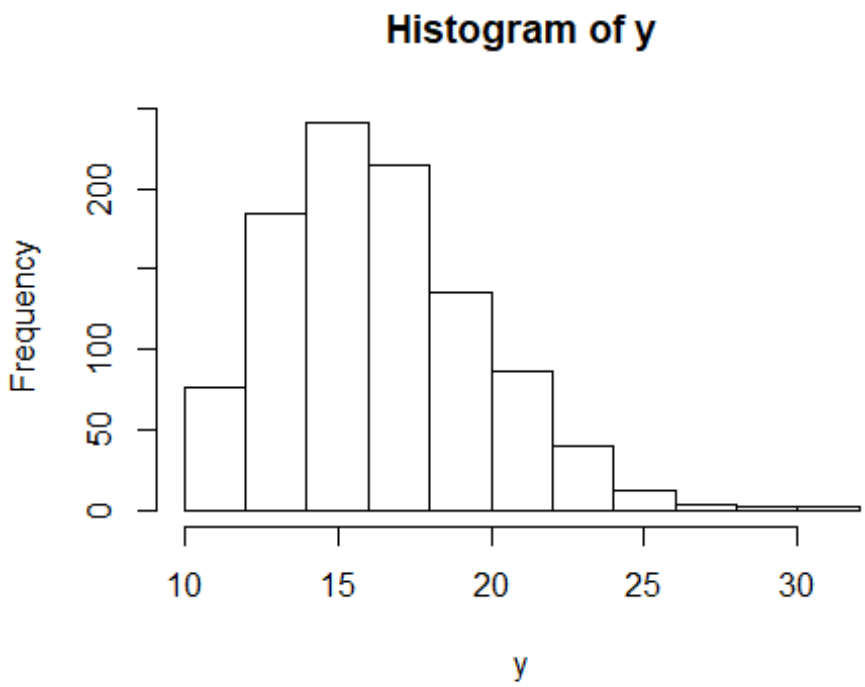
$$P(X = i + 1) = \frac{i(1-p)}{(i+1-r)} \cdot P(X = i).$$

Use this recursion relation to generate 1000 $NB(10,\ 0.6)$ random variates.

Code set-up

Below is binomial.R, the binomial simulator used in the video lectures and found also on the class Blackboard site.

```
simnum = 1000
p = 0.6
r = 10 # for point of comparison with the negative binomial, we will use r he
re
y=0
x = proc.time()
for(sims in 1:simnum){
 pmf=p^r; cdf=pmf; # pmf and cdf
 j=r;
 u=runif(1) # uniform random variate
 # find Binomial variate
 while(u >= cdf){
   pmf =(j*(1-p)/(j+1-r))*pmf
# pmf=((r-j)/(j+1))*(p/(1-p))*pmf # recursion relation
 cdf=cdf + pmf # compute cdf
 j=j+1
 }
 y[sims]=j
}
timer = proc.time() - x
algtime = timer[3]
algtime

## elapsed
##    0.08

mean(y)

## [1] 16.875

sd(y)

## [1] 3.399254

hist(y)
```



This binomial simulator may be applied directly after changing just three lines:

- $j = r$
- the recursion relation formula
- `pmf = p^r`

Report the following for each of the simulations in problems (a) and (b)

- Histogram of the variates
- Mean and standard deviation of the simulated variates
- Run time: compare computing speed between the two algorithms. In R, can wrap your algorithm or sequence of operations as follows to time your code.

```
x = proc.time()
# [the code you want to time here]
timer = proc.time() - x
algtime = timer[3]  # algtime will store the algorithm run time in seconds
```

Questions

- How do the histograms compare?

The histograms look remarkably similar in shape.

- How do the mean and standard deviation from the simulations compare to the true mean and standard deviation of a $NB(0.6,\10)$ distribution?

The mean and standard deviation of the geometric simulation is closer to the true values than the binomial simulation.

- How do the computing times compare? Which algorithm is faster?

The geometric simulation is slower than the binomial simulaiton.

- "Simulation flops": Which simulator do you think uses more uniform random numbers (call to the `runif()` function)? Why?

The gemoetric simulator calls the runif() function more as the function is found in a loop within a loop, while the binomial simulator has that same function in only one loop.