Lab 5: Simulating discrete probability models

Quang Trinh

03.22.18

## 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] 1

### 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.

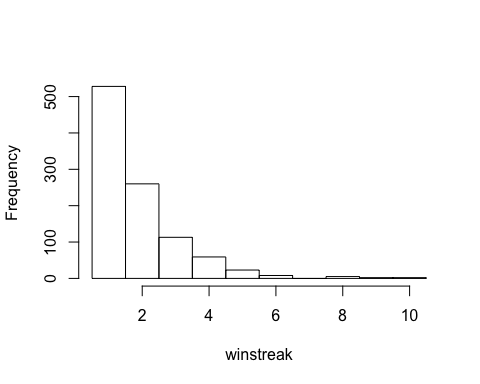
simnum = 1000 # Number of experiment  
winstreak = 0  
p = 20/38  
for(i in 1:1000) {  
 streak = 0  
 lose = 0  
 while(lose == 0) {  
 lose = (runif(1) < p) # generate Bernoulli with probability p  
 streak = streak + 1 # tally streak  
 }  
 winstreak[i] = streak # store streak in a vector  
}  
mean(winstreak) # average length of the win streak

## [1] 1.864

sd(winstreak) # standard deiviation of win streak

## [1] 1.265348

hist(winstreak, br=seq(min(winstreak)-0.5, max(winstreak+0.5)), main="") # histogram of winning streak



max(winstreak) # longest winning streak

## [1] 10

### 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 winning streak. *Avarage length of the winning streak is 1.938.*
* Standard deviation of the winning streak lengths. *Standard deviation of the winning is 1.295.*
* Compare the empirical average and standard deviation in the previous two bullets to the true values from the Geometric(*p* = 20/38) distribution.

*The true average of Geometric(p = 20/38) is 1/(20/38) = 1.9. Also, the true standard deviation of this model is sqrt((1-p)/p^2) = 1.308. Our emperical is 1.295 which is close to the true value. The simulation above has very close result with this value. Theoretically, the more experiment we do, the closer our emperical results to true values.*

* Longest winning streak. *The longest winning streak after 1000 experiments is 10 times.*

## Task 2: Simulating negative binomial distributions

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

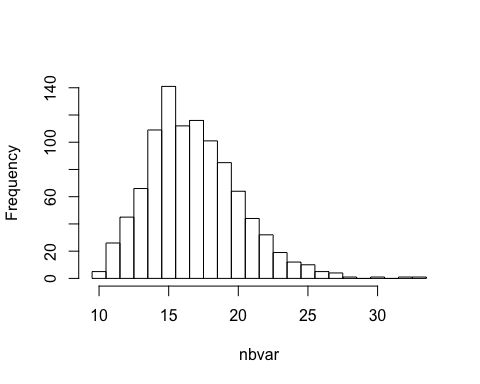
### Problem (a)

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.**

x = proc.time()  
simnum = 1000  
p = 0.6  
r = 10  
nbvar = numeric(simnum)  
for(sims in 1:simnum){  
 for(nbsims in 1:r){  
 tossnum = 0  
 success = 0  
 while(success==0){  
 success = (runif(1)<p)  
 tossnum = tossnum + 1  
 }  
 nbvar[sims] = nbvar[sims] + tossnum  
 }  
}  
timer = proc.time() - x  
algtime = timer[3] # algtime will store the algorithm run time in seconds  
  
hist(nbvar, br=seq(min(nbvar)-0.5, max(nbvar+0.5)), main="") # histogram of NB



mean(nbvar) # emperical mean of NB(10, 0.6)

## [1] 16.816

sd(nbvar) # emperical standard deviation of NB(10, 0.6)

## [1] 3.317401

### Problem (b)

The negative binomial pmf induces the following recursion relation. If , then

Use this recursion relation to generate 1000 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 here  
y=0  
for(sims in 1:simnum){  
 pmf=(1-p)^r; cdf=pmf; # pmf and cdf  
 j=0;  
 u=runif(1) # uniform random variate  
 # find Binomial variate  
 while(u >= cdf){  
 pmf=((r-j)/(j+1))\*(p/(1-p))\*pmf # recursion relation  
 cdf=cdf + pmf # compute cdf  
 j=j+1  
 }  
 y[sims]=j  
}

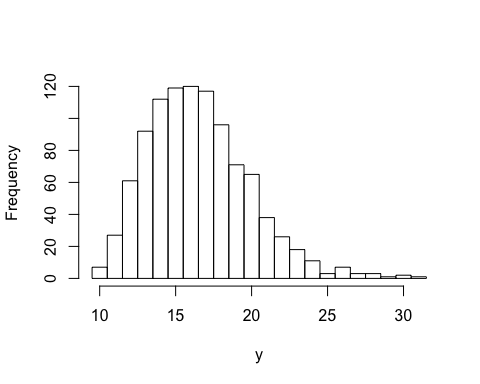
This binomial simulator may be applied directly after changing just three lines:

* 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()  
simnum = 1000  
p = 0.6; r = 10 # for point of comparison with the negative binomial, we will use r here  
y = 0  
for(i 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 # recursion relation  
 cdf=cdf + pmf # compute cdf  
 j=j+1  
 }  
 y[i]=j  
}  
timer = proc.time() - x  
algtime = timer[3] # algtime will store the algorithm run time in seconds  
  
hist(y, br=seq(min(y)-0.5, max(y+0.5)), main="") # histogram of NB



mean(y) # emperical mean of NB(10, 0.6)

## [1] 16.524

sd(y) # emperical standard deviation of NB(10, 0.6)

## [1] 3.348229

### Questions

* How do the histograms compare?

*Histograms of the first and sencond algorithm are very similar in shape and spreading. This means that both algorithm generate somewhat similar results.*

* How do the mean and standard deviation from the simulations compare to the true mean and standard deviation of a distribution?

*The true mean of NB(10, 0.6) = 10/0.6 = 16.667, and true standard deviation is sqrt( r(1-p)/p^2 ) = 3.33. The first algorithm generates (16.699, 3.201) and the second algorithm generates (16.757, 3.210). Thus, both algorithm is very accurate and trustworthy.*

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

*The first alogrithm takes about 0.088 seconds and the second algorithm takes only 0.054 seconds. So the second algorithm is faster. Additionally, the first algorithm uses 3 loops (worst case O(n^3)), while the second one uses only 2 loops.*

* “Simulation flops”: Which simulator do you think uses more uniform random numbers (call to the runif() function)? Why?

*The second algorithm uses more uniform random numbers since it using cdf to verify the condition before enter the while loop.*