STAT 206 Homework 6

Due Monday, November 13, 5:00 PM

General instructions for homework: Homework must be completed as an R Markdown file. Be sure to include your name in the file. Give the commands to answer each question in its own code block, which will also produce plots that will be automatically embedded in the output file. Each answer must be supported by written statements as well as any code used. (Examining your various objects in the "Environment" section of RStudio is insufficient -- you must use scripted commands.)

Part I - Gamblerâs Ruin

1. Suppose you have a bankroll of $1000 and make bets of $100 on a fair game. By simulating the outcome directly for at most 5000 iterations of the game (or hands), estimate the following. (You must stop playing if your player has gone bust.)
    a. the probability that you have âbustedâ (lost all your money) by the time you have placed your one hundredth bet.

```
#this feels like a very intuitive function. First you have to specify how
much money you have
#how much you want to bet, and what the winning odds are
#then after the roll (runif) if you win then you get money, and if you lose
then you lose money
#and you keep going until you're out of money
#at the bottom i let the function go for 1000 iterations, i tried 10000 but
it took too long and i had
#to stop it
#the chances in losing it all before 100 are a lot higher than i expected


odds = function(bet, money , pr) {
  count = 0
  while(bet > 0) {
    prob = runif(1)
    if(prob <= pr) {
      money = bet + money
      # bet = (bet + money)*0.1
    }
    else {
      money = money - bet
      }
    count = count + 1
    if(money < bet){
      break
      }
  }
  games = count
  return(games)
}


games = c()
for(i in 1:1000) {
  game = odds(100,1000,0.5)
  games = c(games, game)
}


all_games = length(games)
losers = length(which(games<=100))
chances = losers / all_games
chances

## [1] 0.306
```

b. the probability that you have busted by the time you have placed your five hundredth bet by simulating the outcome directly.

```
#all we do here is play the game again but this time we want to see
#what are the chances of losing it all before 500 games
#by 500 games there is almost a 60 percent chance of losing it all

losers = length(which(games<=500))
chances = losers / all_games
chances

## [1] 0.643
```

c. the mean time you go bust, given that you go bust within the first 5000 hands.

```
#I'm going to do this with 1000 games. My laptop is having a hard time
running my code more times than
#that for some reason, there is probably a less expensive way to write the
code
#for the code here we just want to know from the vector of games that we
obtained how many of the values
#are less than 1000, which means the loop terminated due to lack of money

problem_c = games[which(games<=1000)]
avg = mean(problem_c)
avg

## [1] 215.9697
```

d. the mean and variance of your bankroll after 100 hands (including busts).

```
#The function is essentially the same as before except this time we want to save money to
#a vector. also we add in a couple lines to break out of the loop if we bust early or reach 100 games
#if you run this chunk multiple times you will see that the avg is sometimes below the starting point
#and sometimes above

odds_fast_lose = function(bet, money , pr) {
  count = 0
  while(bet > 0) {
    prob = runif(1)
    if(prob <= pr) {
      money = bet + money
      }
    else {
      money = money - bet
      }
    count = count + 1
    if(count ==100) {
      break
      }
    if(money < bet) {
      break
      }
  }
  games = count
  return(money)

}

money = c()
for(i in 1:1000) {
  earning = odds_fast_lose(100, 1000 ,0.5)
  money = c(money, earning)
}

avg = mean(money)
vari = var(money)
avg
```

```
## [1] 995.8
```

```
vari
```

```
## [1] 819882.2
```

e. the mean and variance of your bankroll after 500 hands (including busts).

```
# I essentially copied the code from above and changed the break point to 500 unless there is no more money. The results are essentially the same since its
```

```
a fair game

odds_fast_lose = function(bet, money , pr) {
  count = 0
  while(bet > 0) {
    prob = runif(1)
    if(prob <= pr) {
      money = bet + money
      }
    else {
      money = money - bet
      }
    count = count + 1
    if(count == 500) {
      break
      }
    if(money < bet) {
      break
      }
  }
  games = count
  return(money)

}

money = c()
for(i in 1:1000) {
  earning = odds_fast_lose(100, 1000 ,0.5)
  money = c(money, earning)
}

avg = mean(money)
vari = var(money)
avg

## [1] 1003.2

vari

## [1] 2901371
```

2. Repeat the previous problem with betting on black in American roulette, where the probability of winning on any spin is 18/38 for an even payout.

```
#So now we use the code again except this time we tweak the odds of winning
to be slightly worse.
#of course if the odds of sucess go down then we will have less money at the
end of 500 games
#on average. Although there is always a small chance that after 500 games
there will be more money
```

```
odds_fast_lose = function(bet, money , pr) {
  count = 0
  while(bet > 0) {
    prob = runif(1)
    if(prob <= pr) {
      money = bet + money
      }
    else {
      money = money - bet
      }
    count = count + 1
    if(count ==100) {
      break
      }
    if(money < bet) {
      break
      }
  }
  games = count
  return(money)

}

money = c()
for(i in 1:1000) {
  earning = odds_fast_lose(100, 1000 , (18/38))
  money = c(money, earning)
}

avg = mean(money)
vari = var(money)
avg

## [1] 574.8

vari

## [1] 526291.3
```

3. For the American roulette problem in the previous question, you calculated a mean value. Because you saved these final results in a vector, use the bootstrap to estimate the variance of the return in each case for your final answer.

```
#Lets start with the same code, this way we already have the vector of money
so we can attempt
#to run a bootstrap to estimate the variance.
#note I found a package fot bootstrapping but it didn't work well for me

#install.packages("boot")

odds_fast_lose = function(bet, money , pr) {
  count = 0
  while(bet > 0) {
    prob = runif(1)
    if(prob <= pr) {
      money = bet + money
      }
    else {
      money = money - bet
      }
    count = count + 1
    if(count ==100) {
      break
      }
    if(money < bet) {
      break
      }
  }
  games = count
  return(money)

}

money = c()
for(i in 1:1000) {
  earning = odds_fast_lose(100, 1000 , (18/38))
  money = c(money, earning)
}

money_vari = sample(money, 20, replace = TRUE)
vari =  var(money_vari)
vari

## [1] 824736.8
```

Part II - Elo Ratings

One of the earliest examples of a convergent, adaptive Markov process was the rating system devised by Arpad Elo to rank chess players. It has endured for so long as a simple system for so long that it is used as a primary ranking system in many other scenarios, including the NBA team rankings (Nate Silver) and Scrabble (NASPA).

The main idea is two players have ratings R_A and R_B. The estimated probability that player A will win is modeled by a logistic curve,

$$P(A) = \frac{1}{1 + \exp(R_B - R_A)}$$

and once a game is finished, a playerâs rating is updated based on whether they won the game:

$$R_A(\text{new}) = R_A(\text{old}) + K(1 - P(A))$$

or if the lost the game:

$$R_A(\text{new}) = R_A(\text{old}) - KP(A)$$

for some factor K. (Note that both player ratings change.) Our goal is to simulate a repetitive tournament with 10,000 games to see if it converges on the true values.

4. Create a â��trueâ�� vector of ratings for 13 players whose ratings range from -2 to 2 in even intervals. Create another vector with the current ratings which will be updated on a game-by-game basis, and a matrix with 13 rows and 10,000 columns into which we will deposit the ratings over time.

```
#To create the vector we can use the sequence function with the length.out 
argument. We now have ratings for 13
#players in order from weakest to strongest. The next vector is essentially a 
copy of the first which we can use
#to modify values before over writing the currect vector with the new values.
#Finally we build the everything matrix for all players and ratings over the 
10000 games

player_ratings = seq(-2,2, length.out = 13)
player_ratings

##  [1] -2.0000000 -1.6666667 -1.3333333 -1.0000000 -0.6666667 -0.3333333
##  [7]  0.0000000  0.3333333  0.6666667  1.0000000  1.3333333  1.6666667
## [13]  2.0000000

player_updates = player_ratings

all_matrix = matrix(nrow = 13, ncol = 10000)
```

5. Write a function that simulates a game between players i and j given their true underlying ratings. This should be a simple draw from `rbinom(1,1,p)` with the appropriate probability.

```
#this function will take 2 players and caluclaate the odds of player 1 
beating player 2
#based on their ratings. The function takes to players based on their index 
in the rating vector
#calculates the probability of p1 winning, then takes that probability from a 
rbinom draw to see if we
#get a 1 which will be equivalent to a win

players = c()
chances = c()
match = function(player_1, player_2) {
  p1 = player_ratings[player_1]
  p2 = player_ratings[player_2]
  prob_win = 1 / (1 + (exp(p2 - p1)))
  test = rbinom(1,1,prob_win)
```

```r
  return(c(test, p1, p2, prob_win, player_1, player_2))
}

match(1,2)

## [1]  1.0000000 -2.0000000 -1.6666667  0.4174298  1.0000000  2.0000000
```

6. Write a function that, given a value of K, replaces the ratings for the two players who just played a game with their updated ratings given the result from the previous question.

```r
#Here we use our old function to first determine
#who will be the winner between two players
#once we determine that we adjust both ratings

elo = function(match, K) {
  if(match[1] == 1) {
    p1_new = match[2] + (K * (1-match[4]))
    p2_new = match[3] - (K * match[4])
  }
  else {
    p2_new = match[3] + (K * (1-match[4]))
    p1_new = match[2] - (K * match[4])
  }
  player_ratings[match[5]] <<- p1_new
  player_ratings[match[6]] <<- p2_new
}

elo(match(1,2), 3)
```

7. Write a function that selects two players at random from the 13, makes them play a game according to their true ratings, and updates their observed ratings.

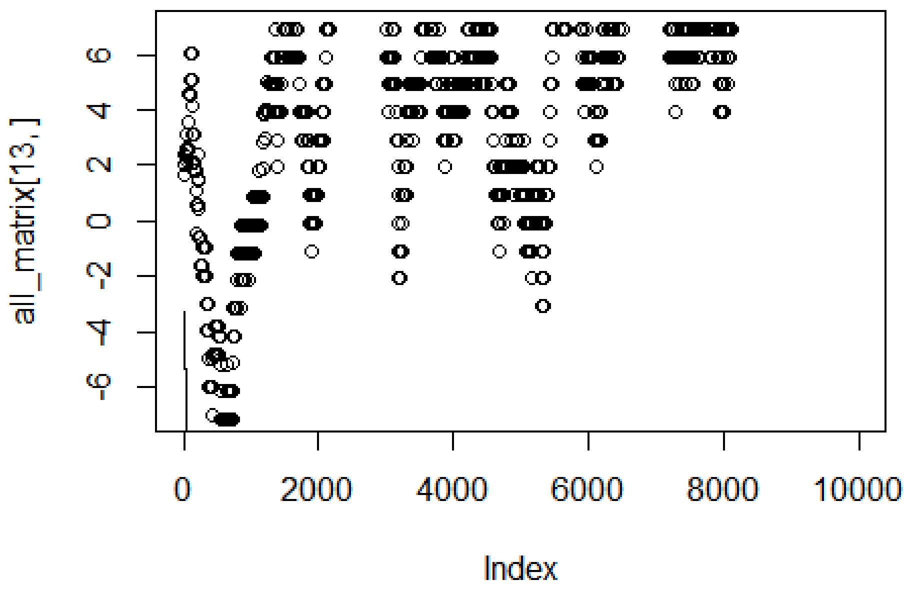
```r
#Here we write a function that first generates 2
#2 numbers which will be the indexes of the player
#ratings and then match those players against one
#another as well as adjust their ratings
#also there is a check to make sure the same player
#is not selected to play himself

rand_opp = function(player_1 , player_2) {
  first = floor(runif(1, 1, 14))
  second = floor(runif(1, 1, 14))
  while(first == second) {
    second = floor(runif(1, 1, 14))
  }
  elo(match(first, second), 1)
  return(player_ratings)
}

jump = rand_opp(a,b)
```

8. Finally, write a function that simulates a tournament as prescribed above: 10,000 games should be played between randomly chosen opponents, and the updated ratings should be saved in your rating matrix by iteration.

```
#our output from the rand_opp function is a vector
#of updated ratings. we want to append those ratings
#into our matrix

all = function(loops) {
  count = 1
  while(count<=loops) {
    count = count +1
    all_matrix[,count] <<- rand_opp(a, b)
  }

}


all(9999)
```

9. Run this tournament with $K = 0.01$. Plot the rating for the best player over time using `plot(..., ty="l")`; add the rating for the worst player using `lines(...)`. Do they appear to converge to the true ratings?

```
#when plotting the best player vs the worst player it does not seem that
either player
#converges to a certain rating, it looks like it would just keep trending on

plot(all_matrix[13,], ylim = c(-7,7))
lines(all_matrix[1,])
```



10. Repeat the previous step with K equal to 0.03, 0.06, 0.1, 0.3, 0.6 and 1. Which appears to give the most reliable rating results?

```
#with K of .03 the graphs trend away from eachother pretty quickly
#the same goes for .06 but on this run the strong player grew faster
#at k of .1 the lines are approaching vertical
# k = .3 is much of the same
#k = .6 is the same essentially
# k = 1 is an interesting case, the rating from he strong player is erractic
# In this case it seems k =.01 gives the best approximation
```