Gov 2001: Problem Set 1 - Solutions

Due Wednesday, February 3 at 6pm

Important Reminder:

The deadline for choosing a partner and a paper to replicate approved is **February 24, 2016**. Send a pdf of your proposed paper to Prof. Gary King and CC both TFs). We will approve the paper (or tell you to look for a new one). Include within your email a brief note (approx. 2-5 sentences) explaining why this paper is a good choice.

Instructions

You should submit your answers and R code to the problems below using the Quizzes section on Canvas.

Problem 1: Probability by Simulation

Simulation is an extremely powerful tool that we will utilize frequently throughout the semester. The following problem provides some practice calculating complex quantities using simulation.
Suppose you have been invited to the poker world championships. You will be playing five-card draw with a standard 52 card deck. You get five cards, then after considering your hand you have the option of discarding up to n cards ($n \in 1, ..., 5$) in order to draw n new cards from the deck. You want to determine the exact probabilities of getting certain hands in the game.

1.1

Use simulation to approximate the probability of getting three of a kind (three of the same number) when you are first dealt five cards (Note: for better accuracy, simulate at least 50,000 hands). Set the seed to 02138.

The following code conducts the simulation

```
# Create deck of cards
deck <- rep(seq(1,13),4)

# Function that checks if there's three of any
# particular card number
tok <- function(hand){
  if(any(table(hand) == 3)) 1
  else 0
}
```

```
# Use apply() to simulate it
set.seed(02138)
threeofkind.apply <- apply(as.matrix(1:10000), 1, function(x)
    tok(sample(deck, 5)))
mean(threeofkind.apply) #0.0226
```

The probability of a three of a kind on the first hand is approximately 0.226.

1.2

Use simulation to approximate the probability of getting a full house (three of a kind, and two of a different kind) when you are first dealt five cards. Set the seed to 8787.

The following code conducts the simulation

```
# Create deck of cards
deck <- rep(seq(1,13),4)

# A full house has to have 1 three of a kind, and 1 two of a
    kind:
fh <- function(hand){
  if(all(c(3,2) %in% table(hand))) 1
  else 0
}

set.seed(8787)
fullhouse.apply <- apply(as.matrix(1:10000), 1, function(x) fh
    (sample(deck, 5)))
mean(fullhouse.apply) #0.0011
```

The probability of a full house on the first hand is approximately 0.0011.

1.3

Say you are dealt five cards, the Jack of hearts, the Jack of diamonds, the three of clubs, the two of diamonds, and the five of spades. You decide to turn in the last three cards and get three more cards. Given what you have in your hand and what you put down, use simulation to approximate the probability of getting a full house from the second draw. (For simplicity, assume you have no opponents.) Set the seed to 02138.

The following code conducts the simulation

```
# Create a deck that is missing 2 jacks, 1 three, 1 two, and 1
    five
deck <- c(rep(seq(1,13),3),1,4,6,7,8,9,10,12,13)
deck <- deck[-11]
```

```
# A full house has to have 1 three of a kind, and 1 two of a
    kind:
fh <- function(hand){
  if(all(c(3,2) %in% table(hand))) 1
  else 0
}

set.seed(02138)
fullhouse2draw <- apply(as.matrix(1:10000), 1,  function(x) fh
    (c(11,11,sample(deck, 3))))
mean(fullhouse2draw) #0.009
```

The probability of a full house on the second draw is approximately 0.009.

Problem 2: More Probability by Simulation

One hundred people line up to board a plane with exactly 100 seats. The first person in line is has lost her boarding pass, so she randomly chooses a seat. After that, each person entering the plane either sits in his or her assigned seat, if it is available, or, if not, chooses an unoccupied seat randomly.

2.1

Using simulation find the probability that when the 100th passenger finally enters the plane, she finds her seat unoccupied? Consider at least 10000 repetitions of the process. Set your starting seed to 02138.

The code below conducts the simulation.

```
## List of seat numbers
seatnums <- seq(1, 100, 1)
## Vector to store results of whether 100th person gets their
    seat
getseat <- NULL
for(j in 1:10000){
  # Which seats did everyone get (order corresponds to # of
      person in line)
    seats <- NULL
    # First person sits in a random seat.
    seats[1] <- sample(seatnums, 1)
    # For each subsequent person
    for(i in 2:length(seatnums)){
      # Their seat is either random (if its taken) or
          deterministic
      seats[i] <- ifelse(i %in% seats == TRUE, sample(seatnums
          [!(seatnums %in% seats)], 1), i)
    }
    # Did person 100 get their seat?
    getseat[j] <- ifelse(seats[100] == 100, 1, 0)
```

```
}
mean(getseat) # 0.5067
```

The probability that the 100th person gets their seat is roughly 0.5.

2.2

Discuss the intuition behind your answer to 2.1. Why does it make sense given the structure of the problem?

If the first person takes their seat at random, then everyone else takes their correct seats in order and the 100th person gets their seat. If the first person takes the 100th person's seat, then the 100th person is guaranteed to *not* get their own seat. If the first person takes any other seat, then that choice is "passed on" to the person whose seat they took. Then the same choice repeats – any outcome that isn't taking the first person's seat or the 100th person's seat is "re-rolled." Since the first person's seat and the 100th person's seat have an equal probability of being taken, the 100th person's probability of sitting in their own seat is 0.5.

Problem 3: Linear Regression and Bootstrapping

In this problem we will explore some of the pre-requisites for the course and introduce bootstrapping, a powerful non-parametric technique.

3.1

Using matrix algebra, write an R function that estimates the Ordinary Least Squares (OLS) coefficients, the σ^2 ancillary parameter, and the coefficient standard errors. Your function should take take two inputs: a vector of the dependent variable and a matrix of explanatory variables. Do not use any pre-programmed functions that perform OLS, such as `lm()`, except to check your results.

```
# y is the dependent variable vector, X is the covariate marix
OLS <- function(y,X){
  # add a column of 1's for the intercept
  X <- cbind(1,X)

  # label the first column 'intercept'
  colnames(X)[1] <- c("Intercept")

  # compute the betas using the formula (X'X)^-1 X'y
```

```
  betas <- solve(t(X) %*% X) %*% t(X) %*% y

  # estimate the fitted values
  fitted <- X %*% betas

  # calculate the residuals
  resid <- fitted - y

  #calculate sigma squared
  sigma.sq <- c(t(resid) %*% resid / (nrow(X) - ncol(X)))
  se.betas <- sqrt(diag(solve(t(X)%*%X)*sigma.sq))

  # return list of betas, sigma^2, standard errors
  output <- list(betas = betas, sigma.sq = sigma.sq, se.betas = se.betas)
  return(output)
}
```

3.2

You are given the following linear model, which we will later use to generate data:

$$y_i = 4 + .5x_{i,1} - 4x_{i,2} + x_{i,3} + \epsilon_i$$

where $\epsilon_i \sim N(\mu = 0, \sigma^2 = 36)$.
Rephrase this model in terms of its systematic and stochastic components.
Stochastic component:

$$Y_i \sim N(\mu_i, \sigma^2) = f_N(y_i|\mu_i, \sigma^2)$$

Systematic component:

$$\mu_i = 4 + .5x_{i1} - 4x_{i2} + x_{i3}$$

3.3

Generate one set of simulated outcomes according to the model outlined above using the covariates contained in the dataset covs.csv on the Canvas website.[1] Your simulated data will contain 1000 observations. Set the 'random' number generator in `R` using the `set.seed(02138)` command before drawing any random quantities. Use your function from section 3.1 to estimate the OLS coefficients and their standard errors.

```
setwd() ## set your working directory!

covs <- read.csv("covs.csv")
```

[1]This data is in the comma-separated value format, so use the `read.csv()` function to load into `R`

```
set.seed(02138)
betas <- c(4, .5, -4, 1)
covs.aug <- as.matrix(cbind(1, covs))
means <- covs.aug %*% betas
y.sim <- rnorm(1000, means, sqrt(36))

OLS(y.sim, as.matrix(covs))

summary(lm(y.sim ~ as.matrix(covs))) ## Check to see that it's the same!
```

Table 1:

	Dependent variable:
x_1	0.450^{***}
	(0.099)
x_2	-3.897^{***}
	(0.061)
x_3	0.900^{***}
	(0.049)
intercept	5.184^{***}
	(0.681)
Note:	*p<0.1; **p<0.05; ***p<0.01

3.4

Bootstrapping is a data resampling procedure for statistical inference, which in one common form uses the empirical distribution of the observed data as a stand-in for the data's population distribution. Assuming that the empirical distribution is a reasonable approximation of the population distribution, we can resample the empirical distribution to generate hypothetical datasets. Model parameters can then be estimated for each of these datasets in order to simulate the sampling distribution of our estimators, and calculate standard errors for our estimates.

Here is the procedure: randomly draw 1000 datasets by resampling *with replacement* the observations from the data you generated in section 3.3[2]. Using a `for()` loop or (preferably) the `apply()` class of functions, estimate the OLS coefficients for each bootstrapped dataset and store them in a matrix. Once you have the 1000 estimates for each coefficient in hand, estimate the standard errors by taking the standard deviation of the simulated

[2]You may find it useful to sample row numbers using the `sample()` function with the `replace = TRUE` argument, and then use your vector of row numbers to subset or reorder the complete dataframe. Note that some rows will be sampled multiple times, so a hypothetical dataset may contain the same observation several times.

sampling distribution for each coefficient.

The bootstrapped standard errors are provided in the table below. Notice how close they are to the results from the regression.

	Regression SE	Bootstrap SE
intercept	0.6811	0.6790
b1	0.0989	0.1027
b2	0.0607	0.0646
b3	0.0488	0.0478

Problem 4: Even More Probability By Simulation

The idea of Monte Carlo simulation was originally inspired by a board/card game puzzle. Stanislaw Ulam wanted to estimate the chances of winning a game of Solitaire.[3] After being unable to tackle the problem analytically, he imagined the idea of repeatedly playing a large number of games and simply counting the number of wins to obtain an approximation to the true answer.

We're going to do something similar to a more *recent* game to further illustrate the power of Monte Carlo simulations where analytical calculations are infeasible.[4] Zombie Dice is a press-your-luck dice game similar to Farkle or Yahtzee.[5] Players play as zombies and roll special six-sided dice with three symbols: "brains," "shotguns," and "footprints." The goal is to score as many brains as possible while avoiding rolling three shotguns during their turn.[6]

Each player starts their turn by drawing three dice randomly without replacement from a bag of 13 color-coded six-sided dice. 6 dice are green, 4 are yellow, and 3 are red. Table 2 gives the distribution of symbols on each die. Green dice are "easy" (more brains) while red dice are "hard" (more shotguns).

Die Type	# of Brains	# of Shotguns	# of Footprints
Green	3	1	2
Yellow	2	2	2
Red	1	3	2

Table 2: Distribution of symbols on Zombie Dice

The player then rolls these dice. Any rolled "brains" are set aside as points (they don't go back in the bag). Any rolled shotguns are also set aside (and don't go back into the bag).

If the player has at any point rolled a total of three or more shotguns on their turn (across all of their rolls), they are "out" and score zero points. If the player is still "alive," after rolling they may choose to either bank their points and score the number of brains that they have rolled in total or "press-their-luck" and roll again.

If the player chooses to roll again, they take any dice that came up as "footprints" and add dice randomly from the bag so that they have a total of three dice to roll. They roll these dice and repeat the above procedure until they are either "out" or choose to bank their points.[7]

[3]See Roger Ekhart. (1987). "Stan Ulam, John von Neumann, and the Monte Carlo Method." *Los Alamos Science*. `http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-88-9068`.

[4]Here, because of the recursive structure of the game and conditional stopping rules, analytical computation of expected winnings is *very* complex

[5]For the offical rules, see the Steve Jackson Games website `http://www.sjgames.com/dice/zombiedice/img/ZDRules_English.pdf`

[6]Note that this game *has* been studied been studied by actual academics – see Cook and Taylor "Zombie Dice: An Optimal Play Strategy" `http://arxiv.org/abs/1406.0351`

[7]There are rules for refreshing the bag of dice if they run out, but we will ignore these for simplicity – the stopping rules you'll look at will never require refreshing the dice

4.1

Consider the following strategy when deciding whether to stop and score:

1. If there are less than 3 dice left in the bag, stop.
2. If you have rolled a total of two shotguns, stop.
3. Otherwise keep rolling

Use Monte Carlo simulation to calculate the expected number of points that would be scored by a player using this strategy. Set the random seed to 02138 at the beginning of the problem. Use a lot of iterations since precision is important for this answer – 10,000 is good.

Below is the code for this particular strategy

```
#### Create a helper function to roll dice
roll_dice <- function(type){
  ## If it's a green die
  if(type == "Green"){
    return(sample(c("Brain","Brain","Brain","Shotgun", "Runner
       ","Runner"), 1, replace=F))
  # If it's a Yellow die
  }else if(type == "Yellow"){
    return(sample(c("Brain","Brain","Shotgun","Shotgun", "
       Runner","Runner"), 1, replace=F))
  # If it's a Red die
  }else if(type == "Red"){
    return(sample(c("Brain","Shotgun","Shotgun","Shotgun", "
       Runner","Runner"), 1, replace=F))
  }

}
### Number of iterations
niter = 10000

### Set random seed
set.seed(02138)

## Storage vector for scores
scores = rep(NA, niter)

### Start the simulation
for (iter in 1:niter){

  ## Boolean for whether player is still in-game
  in_game = T

  ## Initialize brains  counter
  num_brains <- 0

  ## Initialize contents of the bag of dice: 6 Green dice, 4
     Yellow Dice, 3 Red dice
```

```
bag_of_dice = rep(c("Green","Yellow","Red"), times=c(6, 4,
    3))

## How many runners are left-over to roll
runner_vec <- c()

## Which dice have resulted in a "shotgun"
shotgun_vec <- c()

## While the player is still rolling
while(in_game){

  ### Draw the needed number of dice
  n_New = 3 - length(runner_vec) # Draw 3 - # of left-over
      runners dice

  ### Reset "new dice"
  new_dice <- c()

  ### If n_New isn't 0 (and we need to draw dice)
  if (n_New != 0){

    # Pick indices out of the bag
    draw_index = sample(1:length(bag_of_dice), size = n_New,
        replace=F)

    ## Set those dice aside
    new_dice = bag_of_dice[draw_index]

    ## Remove them from the bag
    bag_of_dice = bag_of_dice[-draw_index]
  }

  ### Combine runners and new dice
  dice_to_roll <- c(runner_vec, new_dice)

  ##### SANITY CHECK: If you're not rolling 3 dice,
      something's wrong
  if (length(dice_to_roll) != 3){
    stop("Error: Rolling not 3 dice, something went wrong")
  }

  ## Reset runner vector
  runner_vec <- c()

  ### Roll the dice
  die_results <- sapply(dice_to_roll, function(x) roll_dice(
      x))

  ### For each die you rolled
  for (q in 1:length(die_results)){
    ## If it's a brain, score it
    if (die_results[q] == "Brain"){
      num_brains <- num_brains + 1
    ## If it's a shotgun, add it to the remainder of
```

```
          shotguns
      }else if (die_results[q] == "Shotgun"){
        shotgun_vec <- c(shotgun_vec, dice_to_roll[q])
      ## If it's a runner, add it to the runners
      }else if (die_results[q] == "Runner"){
        runner_vec <- c(runner_vec, dice_to_roll[q])
      }
    }
    ### If you rolled >= 3 shotguns, score zero
    if (length(shotgun_vec) >= 3){
      scores[iter] <- 0
      break
      in_game <- F
    }
    ############ Strategy in 4.1
    #### If there are fewer than 3 dice in-bag, score.
    if (length(bag_of_dice) < 3){
      scores[iter] <- num_brains
      break
      in_game <- F
    }
    ### If you have two shotguns, score.
    else if (length(shotgun_vec) == 2){
       scores[iter] <- num_brains
       break
       in_game <- F
    }
  }
}

### Calculate expected number of brains
expected_brains <- mean(scores) # 2.0857 for Strategy 4.1
expected_brains
```

The expected number of brains is approximately 2.0857

4.2

Now consider a more complex strategy:

1. If there are less than 3 dice left in the bag, stop.
2. If you have rolled a total of one shotgun, stop if you've scored 5 or more brains.
3. If you have rolled a total of two shotguns, stop if you've scored 1 or more brains.
4. Otherwise keep rolling

Use Monte Carlo simulation to calculate the expected number of points that would be scored by a player using this strategy (again, setting the seed at the beginning to 02138 and running 10,000 iterations).

```
#### Create a helper function to roll dice
roll_dice <- function(type){
  ## If it's a green die
  if(type == "Green"){
    return(sample(c("Brain","Brain","Brain","Shotgun", "Runner
       ","Runner"), 1, replace=F))
  # If it's a Yellow die
  }else if(type == "Yellow"){
    return(sample(c("Brain","Brain","Shotgun","Shotgun", "
       Runner","Runner"), 1, replace=F))
  # If it's a Red die
  }else if(type == "Red"){
    return(sample(c("Brain","Shotgun","Shotgun","Shotgun", "
       Runner","Runner"), 1, replace=F))
  }

}
### Number of iterations
niter = 10000

### Set random seed
set.seed(02138)

## Storage vector for scores
scores = rep(NA, niter)

### Start the simulation
for (iter in 1:niter){

  ## Boolean for whether player is still in-game
  in_game = T

  ## Initialize brains counter
  num_brains <- 0

  ## Initialize contents of the bag of dice: 6 Green dice, 4
     Yellow Dice, 3 Red dice
  bag_of_dice = rep(c("Green","Yellow","Red"), times=c(6, 4,
     3))

  ## How many runners are left-over to roll
  runner_vec <- c()

  ## Which dice have resulted in a "shotgun"
  shotgun_vec <- c()

  ## While the player is still rolling
  while(in_game){

    ### Draw the needed number of dice
    n_New = 3 - length(runner_vec) # Draw 3 - # of left-over
       runners dice

    ### Reset "new dice"
    new_dice <- c()
```

```
### If n_New isn't 0 (and we need to draw dice)
if (n_New != 0){

  # Pick indices out of the bag
  draw_index = sample(1:length(bag_of_dice), size = n_New,
      replace=F)

  ## Set those dice aside
  new_dice = bag_of_dice[draw_index]

  ## Remove them from the bag
  bag_of_dice = bag_of_dice[-draw_index]
}

### Combine runners and new dice
dice_to_roll <- c(runner_vec, new_dice)

##### SANITY CHECK: If you're not rolling 3 dice,
    something's wrong
if (length(dice_to_roll) != 3){
  stop("Error: Rolling not 3 dice, something went wrong")
}

## Reset runner vector
runner_vec <- c()

### Roll the dice
die_results <- sapply(dice_to_roll, function(x) roll_dice(
    x))

### For each index in die_results
for (q in 1:length(die_results)){
  ## If it's a brain, score it
  if (die_results[q] == "Brain"){
    num_brains <- num_brains + 1
    ## If it's a shotgun, add it to the remainder of
       shotguns
  }else if (die_results[q] == "Shotgun"){
    shotgun_vec <- c(shotgun_vec, dice_to_roll[q])

    ## If it's a runner, add it to the runners
  }else if (die_results[q] == "Runner"){
    runner_vec <- c(runner_vec, dice_to_roll[q])

  }
}
### If you rolled >= 3 shotguns, score zero
if (length(shotgun_vec) >= 3){
  scores[iter] <- 0
  break
  in_game <- F
}
#### If there are fewer than 3 dice in-bag, score.
if (length(bag_of_dice) < 3){
```

```
      scores[iter] <- num_brains
      break
      in_game <- F
    }
    #### If you have one shotgun and at least 5 brains.
    else if (length(shotgun_vec) == 1 & num_brains >= 5){
      scores[iter] <- num_brains
      break
      in_game <- F
    }
    ### If you have two shotguns and at least one brain, bank
       it.
    else if (length(shotgun_vec) == 2 & num_brains >= 1){
      scores[iter] <- num_brains
      break
      in_game <- F
    }
  }
}
### Calculate expected number of brains
expected_brains <- mean(scores)  ## 2.2291 for Strategy 2
expected_brains
```

Under this strategy, a player would score approximately 2.2291 brains in expectation.

5

Please submit all your code for this assignment as a .R file. Your code should be clean, commented, and executable without error.