Assignment 4

Ariz Kazani

2024-08-18

Assignment 4

Name: Ariz Kazani

Student ID: 101311311

Notes

```
# Libraries

# NOTE: if you do not have any of the below libraries installed,
# un-comment the line and run it

# install.packages("rmarkdown")
library(rmarkdown)

# install.packages("dplyr")
library(dplyr)
```

```
## Warning: package 'dplyr' was built under R version 4.4.1
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
# install.packages("lpSolve")
library(lpSolve)

#install.packages("ompr")
library(ompr)
```

```
## Warning: package 'ompr' was built under R version 4.4.1
```

```
#install.packages("ompr.roi")
library(ompr.roi)
```

```
## Warning: package 'ompr.roi' was built under R version 4.4.1
```

```
#install.packages("ROI.plugin.glpk")
library(ROI.plugin.glpk)
```

```
## Warning: package 'ROI.plugin.glpk' was built under R version 4.4.1
```

Solutions

---

1. Discrete Probabilistic Modeling - Markov Chains

A. Define a simple Markov Chain with 3 states: A, B, and C. Create a transition matrix where the probability of transitioning from any state to another is as follows:

- P(A -> B) = 0.3, P(A -> C) = 0.7
- P(B -> A) = 0.6, P(B -> C) = 0.4
- P(C -> A) = 0.8, P(C -> B) = 0.2

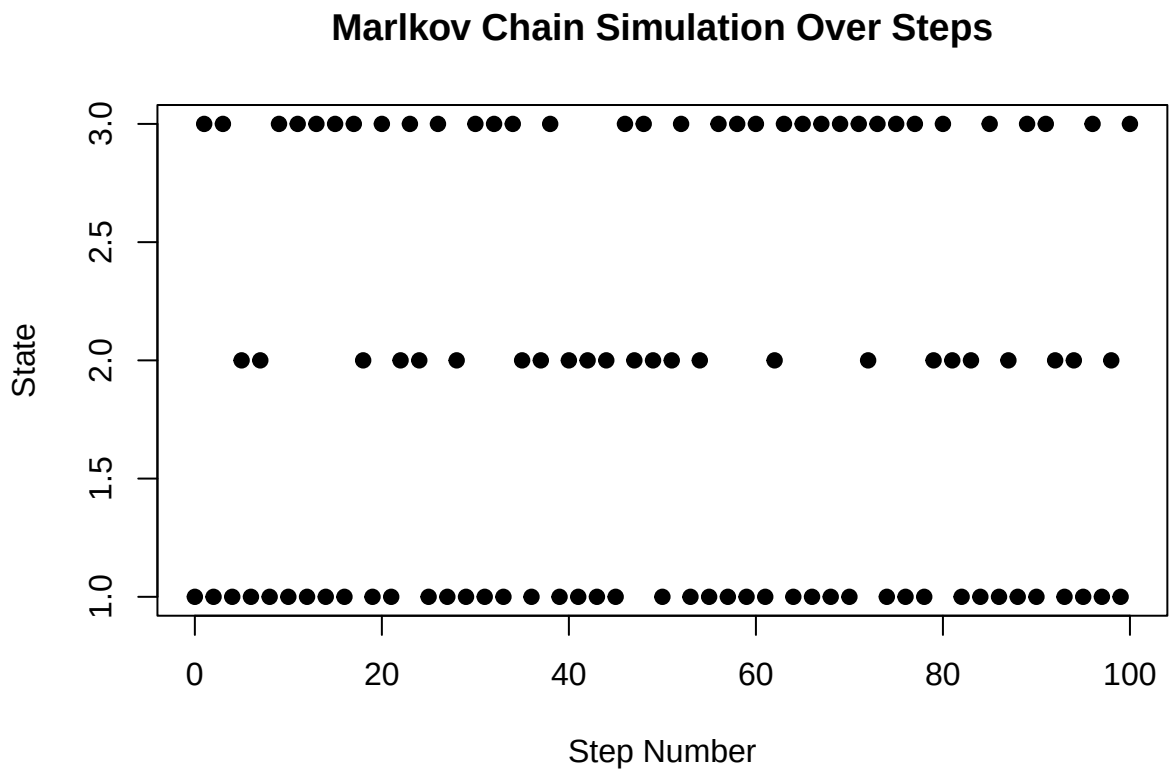
```
pTrans <- matrix(c(
  0, 0.6, 0.8,
  0.3, 0, 0.2,
  0.7, 0.4, 0
), nrow = 3, byrow = TRUE)

pTrans
```

```
##      [,1] [,2] [,3]
## [1,]  0.0  0.6  0.8
## [2,]  0.3  0.0  0.2
## [3,]  0.7  0.4  0.0
```

B. Write an R function to simulate the Markov Chain for 100 steps starting from state A. Plot the state transitions over time.

```
simMC <- function(tMatrix, start, numRow, numSteps) {
  states <- numeric(numSteps + 1)
  states[1] <- sample(1:numRow, 1, prob = start)
```

```r
  for (i in 1:numSteps) {
    states[i + 1] <- sample(1:numRow, 1, prob = tMatrix[, states[i]])
  }

  return(states)
}

x0 <- c(1, 0, 0) #start at A

set.seed(7)
totsim <- simMC(pTrans, x0, 3, 100)

plot(
  0:100 ,
  totsim,
  type = "p",
  pch = 19,
  lwd = 1,
  xlab = "Step Number",
  ylab = "State",
  main = "Marlkov Chain Simulation Over Steps"
)
```



C. Compute the stationary distribution of the Markov Chain using the transition matrix.

```
eigenVals <- eigen(pTrans)
statDist <- eigenVals$vectors[,1]
statDist <- statDist / sum(statDist)
statDist
```

```
## [1] 0.4220183 0.2018349 0.3761468
```

D. Verify the stationary distribution result by simulating the chain for a large number of steps (e.g., 10,000) and comparing the proportion of time spent in each state with the stationary distribution.

```
set.seed(7)

lgSim <- simMC(pTrans, x0, 3, 10000)

ed <- table(lgSim) / length(lgSim)

cat("Simulated distribution", ed, "\n")
```

```
## Simulated distribution 0.4227577 0.2012799 0.3759624
```

```
cat("Theoretical distribution", statDist, "\n")
```

```
## Theoretical distribution 0.4220183 0.2018349 0.3761468
```

---

2. Discrete Probabilistic Modeling - Basic Monte Carlo Simulation

A. Implement a basic Monte Carlo simulation to estimate the value of pi. Use the following steps:

1. Generate a large number of random points within a unit square.
2. Count the number of points that fall inside the unit circle.
3. Use the ratio of points inside the circle to the total number of points to estimate pi.

```
set.seed(7)
x <- runif(1000, -0.5, 0.5)
y <- runif(1000, -0.5, 0.5)

numI <- sum(x^2 + y^2 <= 0.25)

piE <- (numI / 250)
piE
```

```
## [1] 3.128
```

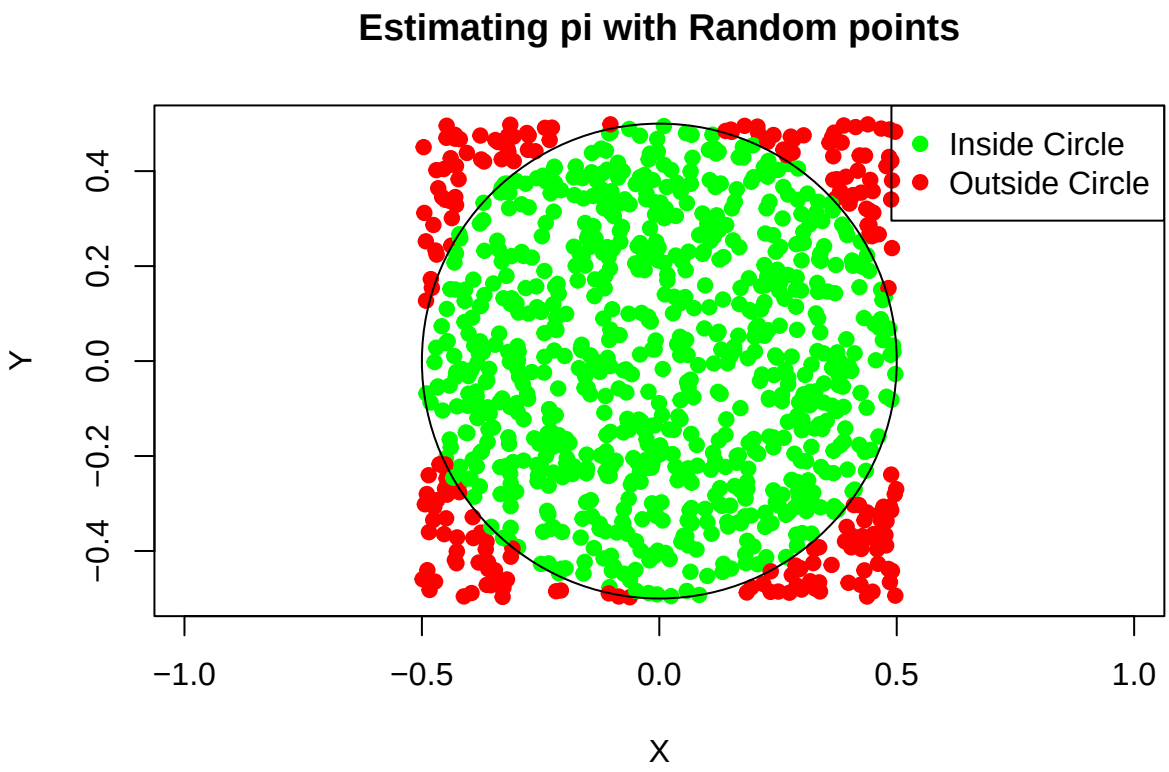
B. Write an R function to perform this simulation with 10,000 points.

```
set.seed(7)

ePi <- function(count) {
  x <- runif(count, -0.5, 0.5)
  y <- runif(count, -0.5, 0.5)

  numI <- sum(x ^ 2 + y ^ 2 <= 0.25)

  piE <- (numI / (count/4))
  piE
}

ePi(10000)
```

```
## [1] 3.1424
```

C. Plot the generated points, differentiating between points inside and outside the circle. Overlay the circle on the plot.

```
set.seed(7)
x <- runif(1000, -0.5, 0.5)
y <- runif(1000, -0.5, 0.5)

plot(
  x,
  y,
  col = ifelse(x ^ 2 + y ^ 2 <= 0.25 , "green", "red"),
  pch = 19,
  main = "Estimating pi with Random points",
  xlab = "X",
  ylab = "Y",
  asp = 1
)
symbols(0, 0, circles = 0.5, add = TRUE, inches = FALSE)
legend(
  legend = c("Inside Circle", "Outside Circle"),
  col = c("green", "red"),
  pch = 19,
   "topright"
)
```



D. Calculate the estimated value of pi from your simulation and compare it to the true value of pi. Discuss the accuracy of the estimate.

```
set.seed(7)
cat("Estimated Value of pi with 1000 random numbers", ePi(1000), "\n")
```

```
## Estimated Value of pi with 1000 random numbers 3.128
```

```
cat("Estimated Value of pi with 10000 random numbers", ePi(10000), "\n")
```

```
## Estimated Value of pi with 10000 random numbers 3.1448
```

```
cat("Estimated Value of pi with 100000 random numbers", ePi(100000), "\n")
```

```
## Estimated Value of pi with 100000 random numbers 3.14008
```

```
cat("Estimated Value of pi with 1000000 random numbers", ePi(1000000), "\n")
```

```
## Estimated Value of pi with 1000000 random numbers 3.141152
```

```r
cat("Estimated Value of pi with 10000000 random numbers", ePi(10000000), "\n")
```

```
## Estimated Value of pi with 10000000 random numbers 3.14127
```

```r
cat("Estimated Value of pi with 100000000 random numbers", ePi(100000000), "\n")
```

```
## Estimated Value of pi with 100000000 random numbers 3.141561
```

```r
cat("True value of pi", 3.1415926)
```

```
## True value of pi 3.141593
```

As we can observe, the more points that are generated, the more accurate the simulations gets, with 100000000 points getting a accuracy of 5 digits. On a side note, each simulation to an exponentially longer time than the previous, and uses an exponential amount of memory (ram).

---

3. Linear Programming

A. Formulate the following linear programming problem: Maximize $Z = 3x_1 + 2x_2$ Subject to:

- $2x_1 + x_2 <= 20$
- $4x_1$ - $5x_2 >= -10$
- $x_1 + 2x_2 <= 15$
- $x_1, x_2 >= 0$

```r
# Objective Function
# Z <- 3 * x1 + 2 * x2

# Constraints
# 2 * x1 + x2 <= 20
# 4 * x1 + 5 * x2 >= -10
# x1 + 2 * x2 <= 15
# x1 >= 0 && x2 >= 0
```

B. Solve the linear programming problem using the lpSolve package in R. Provide the R code and solution.

```r
objective <- c(3, 2)
constraints <- matrix(c(2, 1, 4, 5, 1, 2), nrow = 3, byrow = TRUE)
rhs <- c(20, -10, 15)
directions <- c("<=", ">=", "<=")
solution <- lp("max", objective, constraints, directions, rhs)

solution$solution
```

```
## [1] 8.333333 3.333333
```

```
solution$objval
```

```
## [1] 31.66667
```

C. Interpret the solution, including the values of x1x_1x1 and x2x_2x2, and the maximum value of ZZZ.

(side note I have not clue what x1x_1x1, x2x_2x2 and ZZZ is so I'm going to assume its asking for x1, x2 and Z)

The max value of x1 and x2 are both greater than zero meaning that some x1s and some x2s are required to reach the maximum value, 31.666666 in this case. When performing an objective function coefficient analysis we can see that increasing either of the objective coefficients increased the maximum val and decreasing either had the opposite affect. Increasing the coefficient of x1 to >= 4 made the optimal x2 value 0. When performing a RHS analysis we can see that the maximum increases the when you increase the first and last numbers, decreased them had the opposite effect, where moving the middle number up to 50 and down to -99999 had no effect, moving the middle number any higher causes the maximum value become 0 with optimal values for x1 and x2 also becoming 0. When performing a constraint coefficient analysis we can see that increasing the coefficients of the first restriction reduces the maximum value, decreasing them has the opposite affect. Increasing the coefficients in the second restriction had no affect and decreasing the coefficients decreased the maximum value. The coefficients in the last restriction behaved like the first restriction.

---

4. Integer Programming

A. Formulate the following integer programming problem: Maximize $Z = 4x_1 + 3x_2$ Subject to:

- $3x_1 + 2x_2 <= 12$
- $x_1 - x_2 >= 1$
- $x_1, x_2 >= 0$ and integers

```
# Objective Function
# Z <- 4 * x1 + 3 * x2

# Constraints
# 3 * x1 + 2 * x2 <= 12
# x1 + x2 >= 1
# x1 >= 0 && x2 >= 0 && x1 %% 1 == 0 && x2 %% 1 == 0
```

B. Solve the integer programming problem using the ompr package in R. Provide the R code and solution.

```
model <- MIPModel() %>%
  add_variable(x1, type = "integer", lb = 0) %>%
  add_variable(x2, type = "integer", lb = 0) %>%
  set_objective(4 * x1 + 3 * x2, "max") %>%
```

```
  add_constraint(3 * x1 + 2 * x2 <= 12) %>%
  add_constraint(x1 - x2 >= 1)

result <- solve_model(model, with_ROI(solver = "glpk"))

solution <- bind_cols(
  x1 = get_solution(result, x1),
  x2 = get_solution(result, x2),
  max = result %>%
  objective_value()
)

solution
```

```
## # A tibble: 1 x 3
##      x1    x2   max
##   <dbl> <dbl> <dbl>
## 1     4     0    16
```

C. Interpret the solution, including the values of x_1 and x_2, and the maximum value of Z.

The max value of x2 is 0, and x1 is greater than zero meaning that 0 x1s and 0 x2s are required to reach the maximum value, 16 in this case. When performing an objective function coefficient analysis we can see that increasing either of the coefficients, increased the maximum val and decreasing the fist one had the opposite affect. When performing a RHS analysis we can see that increasing the first number increases the maximum value, decreased opposite effect. When moving the second number up to 4 and down to -0.9999 had no effect, moving the second number any higher causes the maximum value to become 0 with optimal values for x1 and x2 also becoming 0 any lower and the max number would increase. When performing a constraint coefficient analysis we can see that increasing the coefficients of the first restriction causes the maximum value to go down, and the opposite happens when we increase. For the second coefficient in the first restriction, increasing the number had no effect but decreasing it caused the maximum value to increase. Increasing the coefficients in the second restriction, the first one increased the maximum value and the second one had no affect. Decreasing the coefficients decreased the maximum value, only for the second coefficient in the second restriction.

D. Discuss the differences between the linear and integer programming solutions and the situations in which integer programming is necessary.

The difference between Linear programming is that linear programming will usually find a value greater than integer programming. Linear programming is also continuous where as integer programming is discrete, as a result integer programming is required when we want to find whole or discrete values, such as the optimal number of parts to produce. Linear programming is good for values that could be continuous or not whole values, such as the most cost effective length of a road or optimal temperature to grow roses.

---

5. Bootstrap Methods

A. Use the bootstrap method to estimate the mean and 95% confidence interval of the following dataset:

data <- c(5.1, 4.9, 5.7, 5.5, 5.0, 4.8, 5.2, 5.6, 4.7, 5.3)

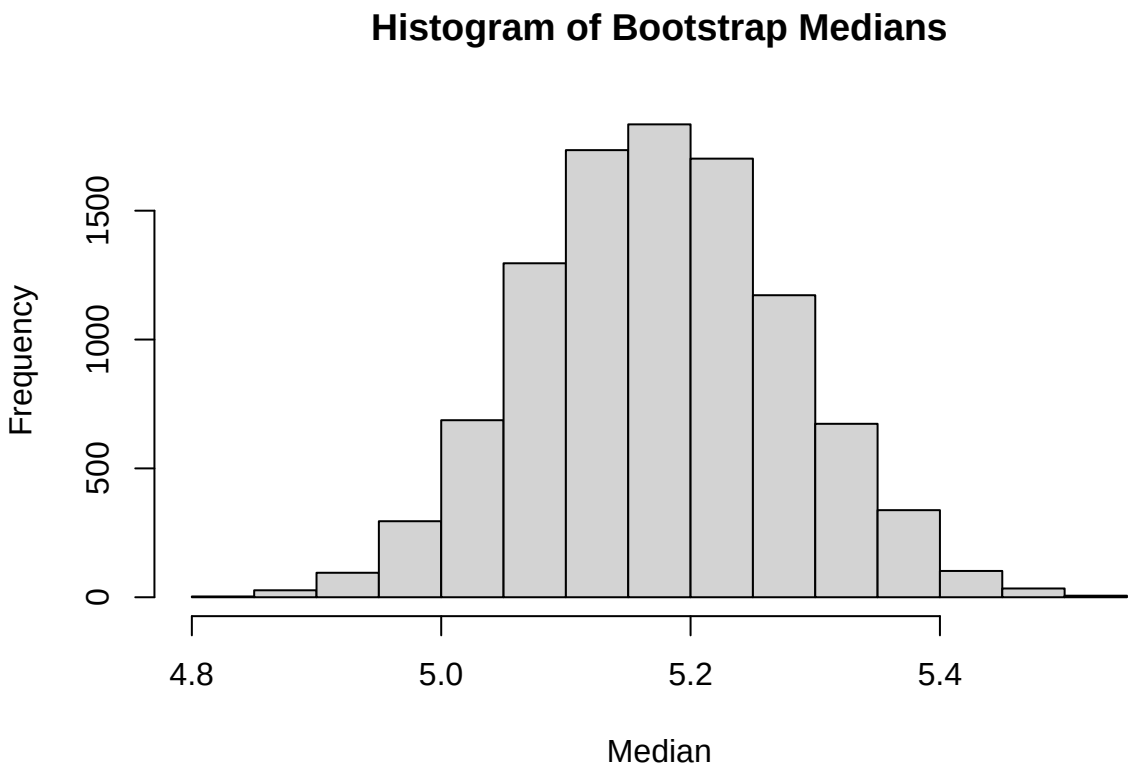
Write an R function to perform bootstrap resampling and calculate the mean for each resample. Perform 10,000 bootstrap resamples.

```r
data <- c(5.1, 4.9, 5.7, 5.5, 5.0, 4.8, 5.2, 5.6, 4.7, 5.3)

bootstrapMean <- function(data, resamples) {
  n <- length(data)
  means <- numeric(resamples)
  for (i in 1:resamples) {
    resample <- sample(data, n, replace = TRUE)
    means[i] <- mean(resample)
  }
  return(means)
}

set.seed(7)

BSMeans <- bootstrapMean(data, 10000)

head(BSMeans)
```

```
## [1] 5.37 5.26 5.22 5.17 5.22 5.10
```

B. Construct a histogram of the bootstrap means and calculate the 95% confidence interval for the mean based on the bootstrap samples.

```r
hist(BSMeans, main = "Histogram of Bootstrap Medians", xlab = "Median")
```




```
BSCI <- quantile(BSMeans, c(0.025, 0.975))
BSCI
```

```
##  2.5% 97.5%
##  4.98  5.39
```

C. Compare the bootstrap confidence interval with the theoretical confidence interval based on the normal distribution. Discuss any differences.

```
mTCI <- mean(data)
sTCI <- sd(data)
nTCI <- length(data)
seTCI <- sTCI / sqrt(nTCI)
TCI <- mTCI + c(qnorm(0.025), qnorm(0.975)) * seTCI
print("Theoretical Confidence Interval")
```

```
## [1] "Theoretical Confidence Interval"
```

```
TCI
```

```
## [1] 4.967696 5.392304
```

```r
print("Bootstrap Confidence Interval")
```

```
## [1] "Bootstrap Confidence Interval"
```

```r
BSCI
```

```
##  2.5% 97.5%
##  4.98  5.39
```

We observer that the Bootstrap Confidence Interval is a very slightly narrower, meaning the original dataset is normally distributed.

D. Evaluate the impact of the sample size on the bootstrap confidence interval by repeating the bootstrap process with different sample sizes (e.g., n=5, n=20) and comparing the results.

```r
set.seed(7)

SS <- sample(data, 5, replace = TRUE)
LS <- sample(data, 20, replace = TRUE)

SM <- bootstrapMean(SS, 10000)
LM <- bootstrapMean(LS, 10000)

SCI <- quantile(SM, c(0.025, 0.975))
LCI <- quantile(LM, c(0.025, 0.975))

list(SmallCI = SCI, LargeCI = LCI)
```

```
## $SmallCI
##  2.5% 97.5%
##  5.06  5.52
##
## $LargeCI
##  2.5% 97.5%
## 5.130 5.435
```

We can see that the confidence interval for the large sample is narrower, indicating more precise estimates.

---

6. Bootstrap Methods - Regression

A. Consider the following dataset:

```r
set.seed(123)
x <- rnorm(100)
y <- 2 + 3*x + rnorm(100)
```

Fit a linear regression model y = B_0 + B_1 x + e and use bootstrap resampling to estimate the standard errors of the regression coefficients.

```
set.seed(123)

x <- rnorm(100)
y <- 2 + 3*x + rnorm(100)

model <- lm(y ~ x)

SERM <- list(b0 = summary(model)$coefficients[1, 2], b1 = summary(model)$coefficients[2, 2])
SERM
```

```
## $b0
## [1] 0.09755118
##
## $b1
## [1] 0.1068786
```

B. Perform 10,000 bootstrap resamples and calculate the standard errors for B_0 and B_1.

```
bootstrap2 <- function(data, nums) {
  x <- data$x
  y <- data$y
  n <- length(x)

  b0 <- numeric(nums)
  b1 <- numeric(nums)

  for (i in 1:nums) {
    resample <- sample(1:n, replace = TRUE)
    xV <- x[resample]
    yV <- y[resample]
    val <- lm(yV ~ xV)
    b0[i] <- coef(val)[1]
    b1[i] <- coef(val)[2]
  }

  return(list(b0 = b0, b1 = b1))
}

set.seed(123)
resamples <- bootstrap2(data.frame(x = x, y = y), 10000)
print("SE Bootstrap")
```

```
## [1] "SE Bootstrap"
```

```
SEBS <- list(b0 = sd(resamples$b0), b1 = sd(resamples$b1))
SEBS
```

```
## $b0
```

```
## [1] 0.09804674
##
## $b1
## [1] 0.1059997
```

C. Compare the bootstrap standard errors with the standard errors obtained from the original regression model.

```
print("SE Bootstrap methods: ")
```

```
## [1] "SE Bootstrap methods: "
```

```
SEBS
```

```
## $b0
## [1] 0.09804674
##
## $b1
## [1] 0.1059997
```

```
print("SE regression model: ")
```

```
## [1] "SE regression model: "
```

```
SERM
```

```
## $b0
## [1] 0.09755118
##
## $b1
## [1] 0.1068786
```

D. Evaluate the impact of the sample size on the bootstrap standard errors by repeating the bootstrap process with different sample sizes (e.g., n=50, n=200) and comparing the results.

```
set.seed(123)

sx <- sample(x, 50, replace = TRUE)
lx <- sample(x, 200, replace = TRUE)
sy <- sample(y, 50, replace = TRUE)
ly <- sample(y, 200, replace = TRUE)

resamplesS <- bootstrap2(data.frame(x = sx, y = sy), 10000)
resamplesL <- bootstrap2(data.frame(x = lx, y = ly), 10000)
print("Small Samples")
```

```
## [1] "Small Samples"
```

```r
list(b0 = sd(resamplesS$b0), b1 = sd(resamplesS$b1))
```

```
## $b0
## [1] 0.3636156
##
## $b1
## [1] 0.5056034
```

```r
print("Large Samples")
```

```
## [1] "Large Samples"
```

```r
list(b0 = sd(resamplesL$b0), b1 = sd(resamplesL$b1))
```

```
## $b0
## [1] 0.1963638
##
## $b1
## [1] 0.1954712
```

we can see that the standard error is much lower with the greater sample size indicating that as the sample size increases, so does the accuracy of our prediction.