STAT230 HW 8
University of California, Berkeley

Thibault Doutre, Student ID 26980469

March 31, 2016

1 PART 1

```
xUnique = 1:5
trueCoeff = c(0, 1, 1)

genData = function(coefs = c(0, 1, 1), xs = 1:5, dupl = 10,
                   sd = 5, seed=2222){
  ### This function creates the artificial data
  set.seed(seed)
  x = rep(xs, each = dupl)
  y = coefs[1] + coefs[2]*x + coefs[3] * x^2 +
    rnorm(length(x), 0, sd)
  return(data.frame(x, y))
}

data = genData()

## # Part 1

genBootY = function(x, y, rep = TRUE){
  ### For each unique x value, take a sample of the
  ### corresponding y values, with or without replacement.
  ### Return a vector of random y values the same length as y
  ### You can assume that the xs are sorted
  res = x
  for (xi in unique(x)){
    y_values = y[x == xi]
```

```r
    y_sample = sample(y_values, 10, replace = rep)
    res[x == xi] = y_sample
  }
  res
}

genBootR = function(fit, err, rep = TRUE){
  ### Sample the errors
  ### Add the errors to the fit to create a y vector
  ### Return a vector of y values the same length as fit
  err_boot = sample(err, length(err), replace = rep)
  y = fit + err_boot
  y

}
```

2 PART 2

```r
## # Part 2

fitModel = function(x, y, degree = 1){
  ### use the lm function to fit a line of a quadratic
  ### e.g. y ~ x or y ~ x + I(x^2)
  ### y and x are numeric vectors of the same length
  ### Return the coefficients as a vector
  if (degree == 1){
    lm.fit = lm(y ~ x)
  }
  if (degree == 2){
    lm.fit = lm(y ~ x + I(x^2))
  }
  as.vector(lm.fit$coefficients)
}
```

3 PART 3

```
## # Part 3

oneBoot = function(data, err1, err2){
  ### data are your data (from call to getData)
  ###  err1 are errors from fit of line to data
  ###  err2 are errors from fit of quadratic to data
  ###  generate three bootstrap samples
  ###  A. use genBootY
  ###  B. use genBootR and errors from linear fit
  ###  C. use genBootR and errors from quadratic fit
  y_A = genBootY(data$x, data$y, rep = TRUE)
  y_B = genBootR(data$y, err1, rep = FALSE)
  y_C = genBootR(data$y, err2, rep = FALSE)

  ### For A, fit a line to data£x and new y's
  ### Repeat to fit a quadratic
  ### For B, fit a line to data£x and the new y's
  ### For C, fit a quadratic to data£x and new y's
  fit.A1 = fitModel(data$x, y_A, degree = 1)
  fit.A2 = fitModel(data$x, y_A, degree = 2)
  fit.B1 = fitModel(data$x, y_B, degree = 1)
  fit.C2 = fitModel(data$x, y_C, degree = 2)

  ### Return the coefficients from the 4 fits in a list
  list(A1 = fit.A1, A2 = fit.A2, B1 = fit.B1, C2 = fit.C2)
}
```

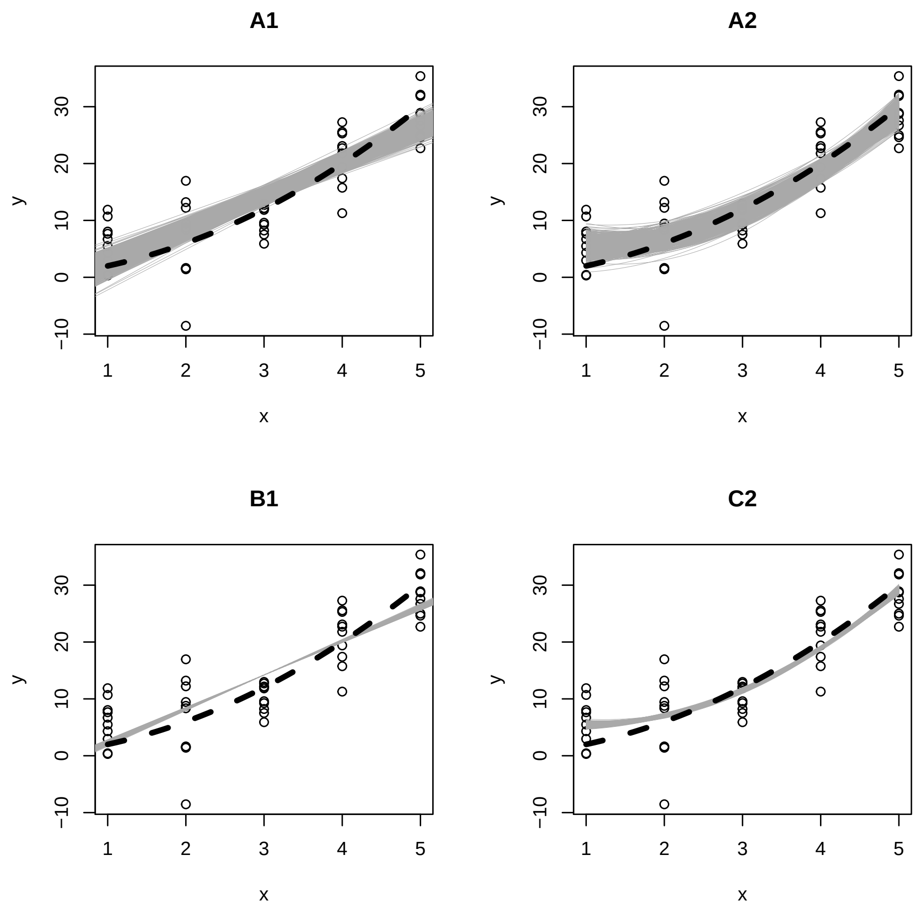
4 PART 4

```
repBoot = function(data, B = 1000){

  ### replicate a call to oneBoot B times
  ### format the return value so that you have a list of
  ### length 4, one for each set of coefficients
  ### each element will contain a data frame with B rows
  ### and two or three columns, depending on whether the
  ### fit is for a line or a quadratic
  ### Return this list
  res.A1 = as.data.frame(matrix(NA, nrow = B, ncol = 2))
  res.A2 = as.data.frame(matrix(NA, nrow = B, ncol = 3))
  res.B1 = as.data.frame(matrix(NA, nrow = B, ncol = 2))
  res.C2 = as.data.frame(matrix(NA, nrow = B, ncol = 3))

  err1 = rnorm(nrow(data))
  err2 = rnorm(nrow(data))

  for (i in 1:B){
    res = oneBoot(data, err1, err2)
    res.A1[i,] = res$A1
    res.A2[i,] = res$A2
    res.B1[i,] = res$B1
    res.C2[i,] = res$C2
  }

  list(A1 = res.A1, A2 = res.A2, B1 = res.B1, C2 = res.C2)

}

repBootConf = function(data, B = 1000){
  ### For each of the four variations, return a bootstrap
  ### 95% confidence interval for the linear coefficient
  ### for x.
  res = repBoot(data)
  conf.A1 = as.vector(t.test(res$A1[,2])$conf.int[1:2])
  conf.A2 = as.vector(t.test(res$A2[,2])$conf.int[1:2])
  conf.B1 = as.vector(t.test(res$B1[,2])$conf.int[1:2])
  conf.C2 = as.vector(t.test(res$C2[,2])$conf.int[1:2])

  list(A1 = conf.A1, A2 = conf.A2, B1 = conf.B1, C2 = conf.C2)
}
```

5 PART 5

```r
## # Part 5

bootPlot = function(data, coeff, trueCoeff){
  ### data is the original data set
  ### coeff is a data frame from repBoot
  ### trueCoeff contains the true coefficients that generated
  ### data

  ### Make a scatter plot of data
  ### Use mapply to add lines or curves for each row in coeff
  ### Use transparency
  ### Use trueCoeff to add line or curve - make it stand out

  draw_line = function(row, lwd = .03, col = "darkgray"){
    if (length(row) == 2){
      abline(as.numeric(row[1]), as.numeric(row[2]),  lwd = lwd, col = col)
    }
    if (length(row) == 3){
      x = seq(1,5,by=0.1)
      y = as.numeric(row[1]) + as.numeric(row[2])*x + as.numeric(row[3])*(x^2)
      lines(x, y, lwd = lwd, col = col)
    }
  }

  plot_model = function(coeff_values){
    plot(data)
    apply(coeff_values, 1, draw_line)
    x = seq(1,5,by=0.1)
    y = as.numeric(trueCoeff[1]) +
      as.numeric(trueCoeff[2])*x +
      as.numeric(trueCoeff[3])*(x^2)
    lines(x, y, lwd = 4, col = "black", lty=2)
  }

  plot_model(coeff)
}

### Run your simulation
```

```r
xUnique = 1:5
trueCoeff = c(0, 1, 1)
myData = genData(coefs = trueCoeff, xs = xUnique)
expt = repBoot(data = myData )
par(mfrow = c(2, 2))
bootPlot(data = myData, coeff = expt[[1]], trueCoeff)
title("A1")
bootPlot(data = myData, coeff = expt[[2]], trueCoeff)
title("A2")
bootPlot(data = myData, coeff = expt[[3]], trueCoeff)
title("B1")
bootPlot(data = myData, coeff = expt[[4]], trueCoeff)
title("C2")
```



```r
par(mfrow = c(1, 1))
```

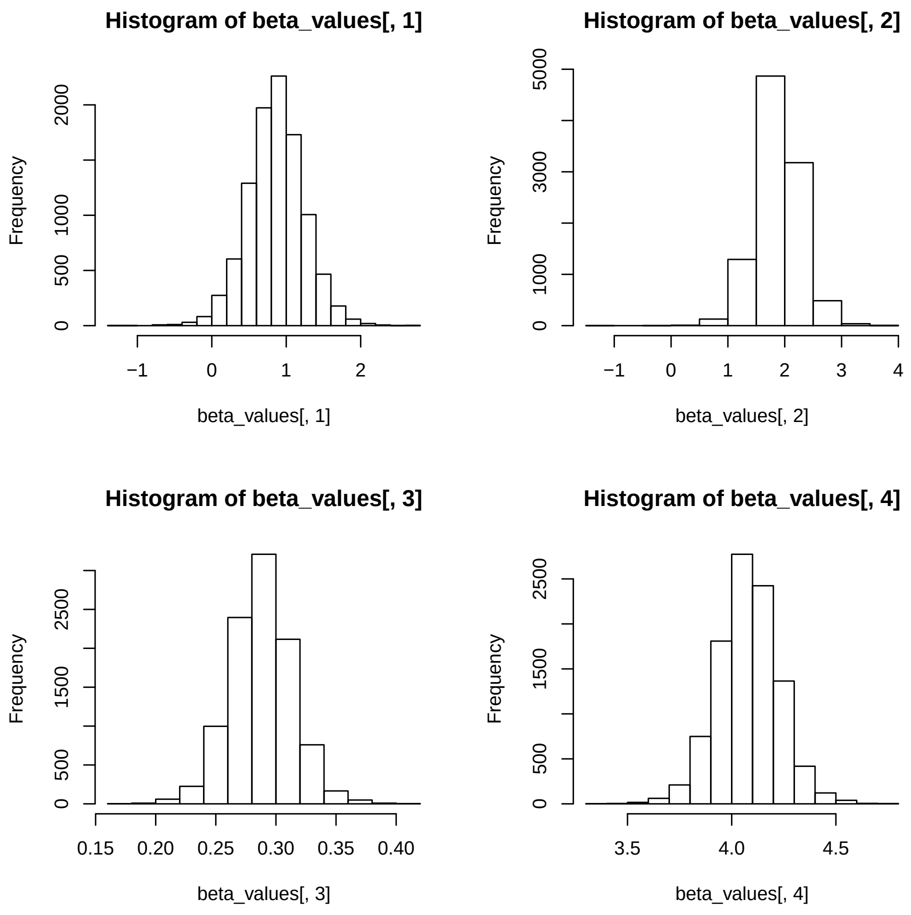
- When is the variability smallest?
  With the second method
- Is there a problem with bias when the model being fitted is wrong?
  No bias when the model being fitted is linear
- Is there a problem with basing the bootstrap on the data, i.e. are the data close to the truth?
  Data seem close to the truth, except for points with low x values

- Are any problems you find worse for one method than the other? First method leads to less variability, but depends on the choice of the errors. Also we may need to jitter the data along the x axis.

6 PART 6

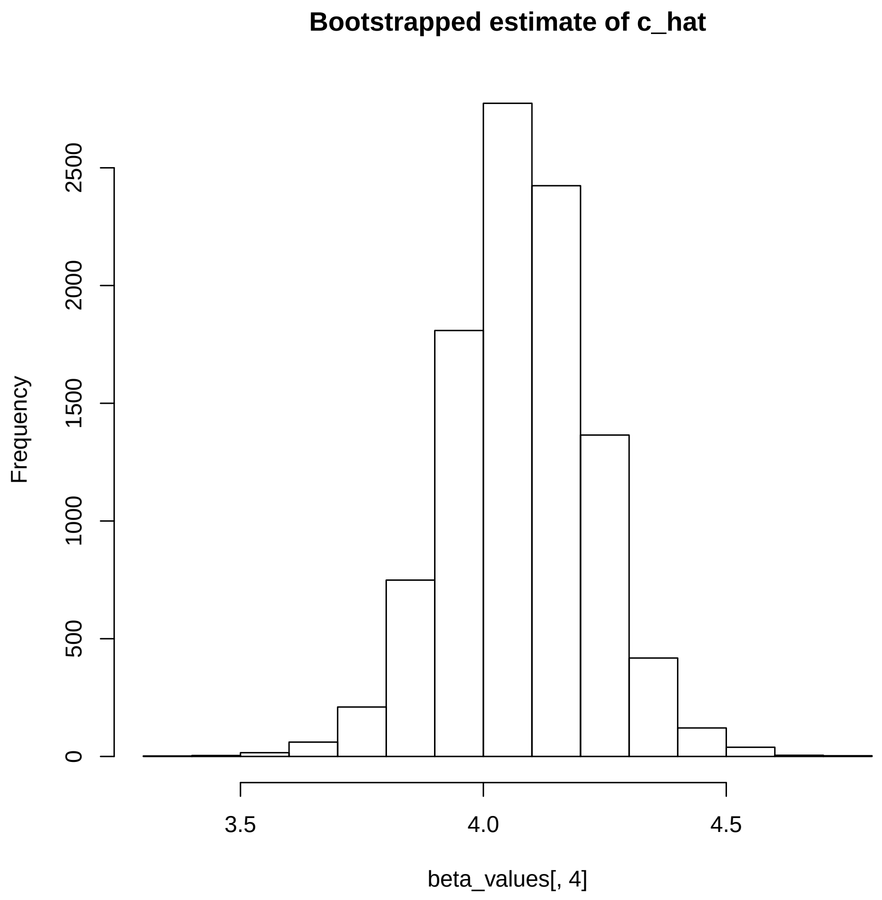
```r
### PART 6.
### Generate data
set.seed(1)
W = rnorm(100,1:100/5)

# Parameters to be estimated
a = c(1,2)
b = 0.3
c = 4
beta = c(a[1],a[2],b,c)

# Covariance matrix of the errors
K = matrix(c(1,0.5,0.5,2),ncol=2)

# Generate errors function
generate_errors = function(n = 50){
  # Define parameters
  rho = 0.5/sqrt(2)
  mu1=0; s1=1; mu2=0; s2=sqrt(2)

  # Define X, Y and Z with the bivariate normal relationship
  X = rnorm(n)
  Z = rnorm(n)
  eps = sqrt(1-rho^2) * Z
  Y = rho * X + eps

  # Adjust means and variances
  Y = (Y-mean(Y))/sd(Y)*s2+mu2
  X = (X-mean(X))/sd(X)*s1+mu1

  # Adjust rho by transforming Y
  rho_hat = cor(X,Y)
  a = s1^4*(rho^2-1)
  b = 2*rho_hat*s1^3*s2*(rho^2-1)
  c = (rho^2-rho_hat^2)*s2^2*s1^2
  delta = b^2-4*a*c
  correction = (-b-sqrt(delta))/(2*a)
  Y=Y+correction*X

  # Adjust means and variances
  Y = (Y-mean(Y))/sd(Y)*s2+mu2
  X = (X-mean(X))/sd(X)*s1+mu1

  return(cbind(X,Y))
}

# Generate errors
epsilons = generate_errors()
cov(epsilons)

##       X   Y
## X  1.0  0.5
## Y  0.5  2.0

# Generate Y
Y = c(0,0,W)
m = 50
for (t in 1:50) {
  for (j in 1:2){
    # Y_tj
    Y[2*t+j] = a[j] + b*Y[2*(t-1)+j] + c*W[2*(t-1)+j] + epsilons[t,j]
  }
}

# Generate the design matrix X
X = matrix(NA,nrow = 100, ncol=4)
X[,3] = Y[1:100]
X[,4] = W
X[,1] = c(1,0)
X[,2] = c(0,1)
head(X)
```

```
##       [,1] [,2]        [,3]        [,4]
## [1,]     1    0  0.0000000 -0.4264538
## [2,]     0    1  0.0000000  0.5836433
## [3,]     1    0 -1.2258338 -0.2356286
## [4,]     0    1  4.7286077  2.3952808
## [5,]     1    0 -0.0939783  1.3295078
## [6,]     0    1 12.9162070  0.3795316

## BOOTSTRAP function
bootstrap_fgls = function(B = 1000){
  beta_values = c()
  B = 10000

  for (i in 1:B){
    # OLS
    lm.fit = lm(Y[3:102]~0+X)
    beta_OLS = lm.fit$coefficients
    e = lm.fit$residuals

    K_OLS = matrix(0,2,2)
    for (t in 1:m){
      e_sub = e[(2*t-1):(2*t)]
      K_OLS = K_OLS + (e_sub %*% t(e_sub))
    }
    K_OLS = K_OLS / m

    ## One step GLS
    # Construct G from K_OLS
    G = diag(0,ncol = 2*m, nrow=2*m)
    for (t in 0:(m-1)){
      G[(2*t+1):(2*t+2),(2*t+1):(2*t+2)] = K_OLS
    }

    # Beta FGLS
    beta_FGLS = solve(t(X)%*%(solve(G)%*%X))%*%(t(X)%*%(solve(G)%*%Y[3:102]))
    beta_values = c(beta_values,beta_FGLS)

    # Sample the errors
    resample1 = sample(1:50,50,replace=TRUE)
    resample = cbind(2*resample1-1,2*resample1)
    errors = e[resample]

    # Regenerate Y
    a_hat = beta[1:2]
    b_hat = beta[3]
    c_hat = beta[4]
    Y = c(0,0,W)
    m = 50
    for (t in 1:50) {
      for (j in 1:2){
        # Y_tj
        Y[2*t+j] = a_hat[j] + b_hat*Y[2*(t-1)+j] + c_hat*W[2*(t-1)+j]
      }
    }
    Y[3:102]  = Y[3:102] + errors

  }
  beta_values = matrix(beta_values,ncol=4,byrow=TRUE)
  beta_values
}

beta_values = bootstrap_fgls(B=1000)


par(mfrow=c(2,2))
hist(beta_values[,1])
hist(beta_values[,2])
hist(beta_values[,3])
hist(beta_values[,4])
```



```
par(mfrow=c(1,1))
```

```
## # Bias in c_hat
hist(beta_values[,4], main = "Bootstrapped estimate of c_hat")
```



```
bias_c = mean(beta_values[,4])-c
bias_c

## [1] 0.07807151
```

The bias is relatively small when looking at the distribution of c hat. The bootstrapp gives a good estimator of the true parameter. It also gives an accurate confidence interval for the estimate.
If we wanted to evaluate the bias with only one iteration, we would have obtained:

```
beta_values_0 = bootstrap_fgls(B=1)
# bias
mean(beta_values[4])-c

## [1] -2.835923
```