Using R in 200D

Luke Sonnet

Contents

This document is not an introduction to `R`, rather it is a introduction to the topics that are crucial to working in `R` in this class. There are countless resources for learning `R` and several of them are on the syllabus. If you are going to be a quantitative researcher trained in this department and working with other political scientists, you will probably need to know `R` from the ground up. Take your time and work through a text book.

```
## Set working directory
setwd("C:/Dropbox/teaching/PS200D/2017/section/01IntroProbTexR/")
```

Working with data frames

Data frames are the workhorses of data analysis in `R`. We can easily load in some data from a csv file

```
qog <- read.csv("http://lukesonnet.github.io/teaching/inference/qog_data.csv",
                stringsAsFactors = F)
head(qog)
```

```
##      country enep   mdm  ethFrac
## 1    Albania 3.18 11.10 0.096600
## 2  Argentina 6.62 10.70 0.254900
## 3  Australia 2.86  0.90 0.148504
## 4    Austria 4.79 20.30 0.125631
## 5 Bangladesh 2.78  1.00 0.223350
## 6    Armenia 5.26 62.15 0.133800
```

Let's focus on different types of variables, creating new variables, basic plotting, and using `lm` and `glm`.

Working with variables

We can examine the structure of variables (columns) in a data frame.

```
## Structure of the data frame
str(qog)
```

```
## 'data.frame':    82 obs. of  4 variables:
##  $ country: chr  "Albania" "Argentina" "Australia" "Austria" ...
```

```
##  $ enep   : num  3.18 6.62 2.86 4.79 2.78 ...
##  $ mdm    : num  11.1 10.7 0.9 20.3 1 ...
##  $ ethFrac: num  0.0966 0.2549 0.1485 0.1256 0.2233 ...
```

Country is a factor variable. This is useful when we want to plot data in groups. Here however, the factors, or groups, are countries and so there is only one observation per group. Let's greate a group variable that breaks ethnic fractionalization into above and below 0.5 and call it $frac$.

```
## create 1s and 0s using as.numeric
qog$ethFrac > 0.5
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [12]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
## [23] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [34] FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
## [45]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE
## [56]  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE
## [67]  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
## [78]  TRUE FALSE FALSE FALSE FALSE
```

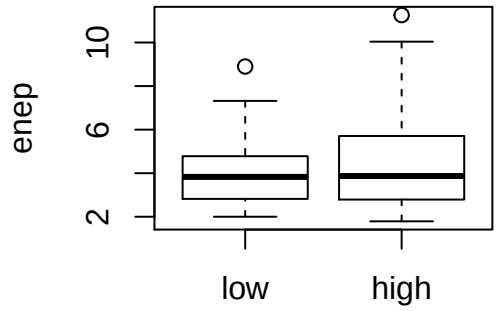
```
as.numeric(qog$ethFrac > 0.5)
```

```
##  [1] 0 0 0 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 0 0 1
## [36] 1 0 1 0 0 0 0 1 1 1 0 1 1 1 1 0 1 0 0 0 1 1 0 1 1 0 1 0 0 1 0 1 1 0 0
## [71] 1 0 1 0 1 0 0 1 0 0 0 0
```

```
## Turning that into a factor
qog$frac <- factor(as.numeric(qog$ethFrac > 0.5), labels = c("low", "high"))
```

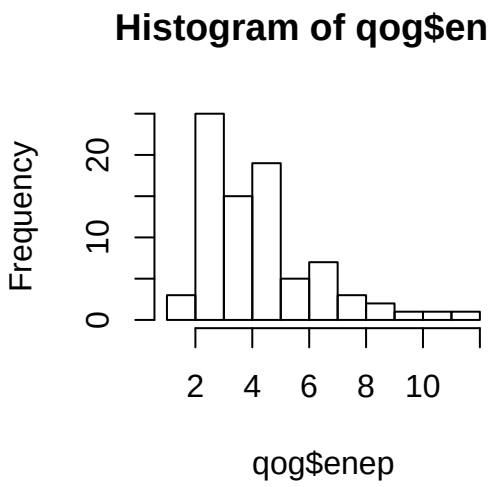
Factors are great for plotting and are required sometimes. However, they can also be a pain and it may be easier to use character and numeric variables themselves.

```
boxplot(qog$enep ~ qog$frac, ylab = "enep")
```

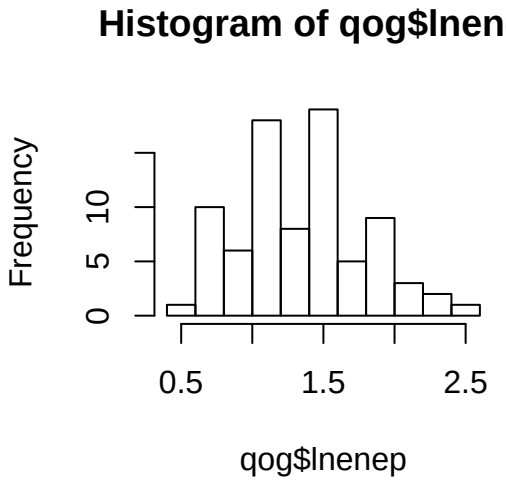


We can also transform variables.

```
## Histogram of effective number of electoral parties
hist(qog$enep)
```




```
qog$lnenep <- log(qog$enep)
hist(qog$lnenep)
```




```
## Non-sensically adding variables together
qog$nonsense <- qog$enep + qog$mdm
```

Analyzing data

This section will just show some basic t-tests, OLS, and logistic regression.

Let's do a difference-of-means test for the effective number of paries with respect to high or low levels of fractionalization.

```
t.test(x = qog$enep[qog$frac == "high"],
       y = qog$enep[qog$frac == "low"])
```

```
##
```

```
##  Welch Two Sample t-test
##
## data:  qog$enep[qog$frac == "high"] and qog$enep[qog$frac == "low"]
## t = 1.0536, df = 49.576, p-value = 0.2972
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.4732402  1.5170992
## sample estimates:
## mean of x mean of y
##  4.580909  4.058980
```

Let's run OLS to see if district magnitude and ethnic fractionalization correlate with the effective number of parties.

```
## Additive model
lmAdd <- lm(enep ~ ethFrac + mdm,
            data = qog)
summary(lmAdd)
```

```
##
## Call:
## lm(formula = enep ~ ethFrac + mdm, data = qog)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.5345 -1.4795 -0.3631  0.7078  6.9009
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 3.911352   0.454723   8.602 5.84e-13 ***
## ethFrac     0.676294   0.936388   0.722    0.472
## mdm         0.004006   0.004120   0.972    0.334
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.035 on 79 degrees of freedom
## Multiple R-squared:  0.01816,    Adjusted R-squared:  -0.006693
## F-statistic: 0.7307 on 2 and 79 DF,  p-value: 0.4848
```

```
lmAdd$coefficients
```

```
## (Intercept)     ethFrac         mdm
##  3.91135194  0.67629375  0.00400575
```

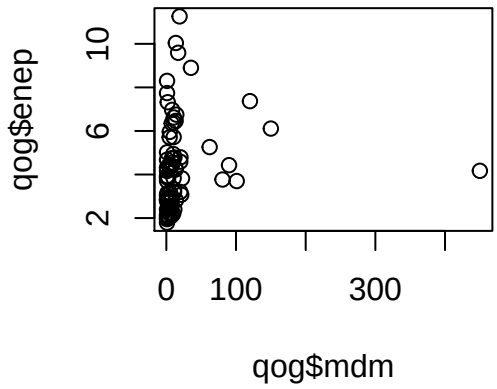
Now let's run an interactive model and transform one where we transform one of our outcomes. Note that it will add the base terms automatically if you do the multiplication in the formula.

```
## Interactive model
lmInt <- lm(enep ~ ethFrac * mdm,
            data = qog)
summary(lmInt)
```

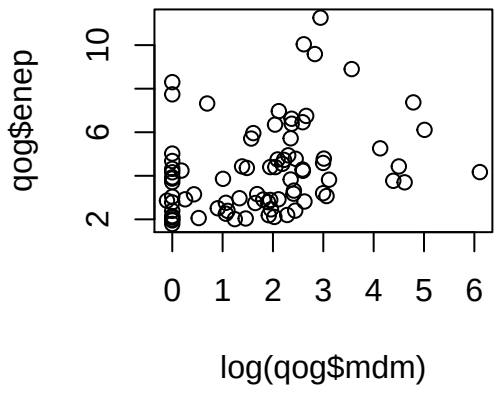
```
##
## Call:
## lm(formula = enep ~ ethFrac * mdm, data = qog)
##
## Residuals:
```

```
##     Min      1Q  Median      3Q     Max
## -2.5363 -1.4794 -0.3473  0.7042  6.9030
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.905445   0.512510   7.620  5.1e-11 ***
## ethFrac      0.689447   1.073322   0.642    0.523
## mdm          0.004625   0.024556   0.188    0.851
## ethFrac:mdm -0.001451   0.056686  -0.026    0.980
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.048 on 78 degrees of freedom
## Multiple R-squared:  0.01817,    Adjusted R-squared:  -0.01959
## F-statistic: 0.4812 on 3 and 78 DF,  p-value: 0.6963
```

```
plot(qog$mdm, qog$enep)
```




```
plot(log(qog$mdm), qog$enep)
```




```
## Quadratic with log
lmQuad <- lm(enep ~ ethFrac + log(mdm),
             data = qog)
summary(lmQuad)
```

```
##
## Call:
## lm(formula = enep ~ ethFrac + log(mdm), data = qog)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.4774 -1.5521 -0.3234  0.9940  6.2374
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   2.8452     0.5576   5.103 2.25e-06 ***
## ethFrac       1.3038     0.9096   1.433  0.15569
## log(mdm)      0.4962     0.1568   3.164  0.00221 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.929 on 79 degrees of freedom
## Multiple R-squared:  0.1181, Adjusted R-squared:  0.09582
## F-statistic: 5.292 on 2 and 79 DF,  p-value: 0.006969
```

I will demonstrate logistic regression with a dataset of grad student admissions.

```
ad <- read.csv("http://www.stat.ucla.edu/~handcock/216/datasets/BINARYEX/binary.csv",
               stringsAsFactors = F)
head(ad)
```

```
##   admit gre  gpa rank
## 1     0 380 3.61    3
## 2     1 660 3.67    3
## 3     1 800 4.00    1
## 4     1 640 3.19    4
## 5     0 520 2.93    4
```

```
## 6     1 760 3.00    2
```

```
logitOut <- glm(admit ~ gre + gpa + rank,
                data = ad,
                family = binomial)
summary(logitOut)
```

```
##
## Call:
## glm(formula = admit ~ gre + gpa + rank, family = binomial, data = ad)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.5802  -0.8848  -0.6382   1.1575   2.1732
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.449548   1.132846  -3.045  0.00233 **
## gre          0.002294   0.001092   2.101  0.03564 *
## gpa          0.777014   0.327484   2.373  0.01766 *
## rank        -0.560031   0.127137  -4.405 1.06e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 499.98  on 399  degrees of freedom
## Residual deviance: 459.44  on 396  degrees of freedom
## AIC: 467.44
##
## Number of Fisher Scoring iterations: 4
```

Random numbers

Often it will be very important for use to generate simulated data or sample from existing data. We might need to do this to show the properties of an estimator using Monte Carlo simulation, for randomization inference, or for bootstrapping methods.

Setting seeds

When software generates random numbers it uses some algorithm that takes a *seed* as its starting point. Let's say we want to sample a number from the standard normal distribution. If we do this multiple times, we will get different answers.

```
rnorm(1)
```

```
## [1] -0.9348959
```

```
rnorm(1)
```

```
## [1] -0.2618023
```

But if we start the pseudo-random number generator with the same seed, then the random numbers will be identical no matter how many times the code is run.

```
set.seed(200)
rnorm(1)
```

```
## [1] 0.08475635
```

```
set.seed(200)
rnorm(1)
```

```
## [1] 0.08475635
```

This is very important when sharing code (such as in problem sets or in replication code for a paper you are writing). If any part of the code relies on sampling, simulated data, or any random process, not setting the seed will result in different numbers. Also note that setting the seed only works for the next line of code. You have to re-set it to some value each time you do some random process. Note that you do not need to use the same seed across different lines; remember that the goal is not to get the same numbers throughout your code, but rather to get the same numbers the next time you run your code.

```
set.seed(123)
rnorm(1)
```

```
## [1] -0.5604756
```

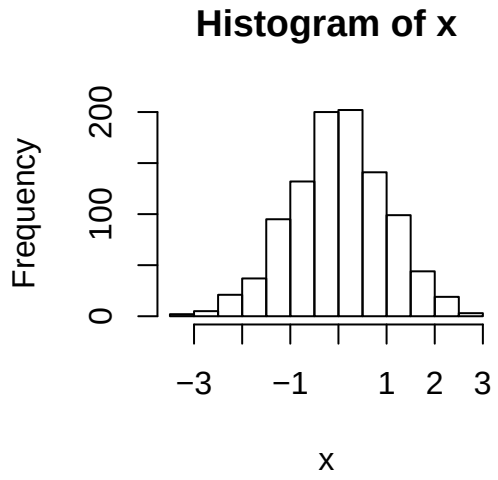
```
set.seed(456)
runif(1)
```

```
## [1] 0.0895516
```

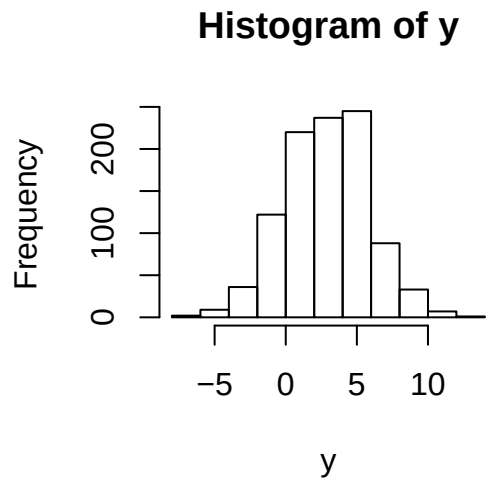
Simulating data from distributions

Often you will need to draw data from distributions in order. `R` has a host of distributions that are included in the default `stats` package that is always loaded when you start `R`. You can make draws from distributions generally using the `r` prefix on the distribuion name, such as `rnorm`. Below are some of the easy examples:

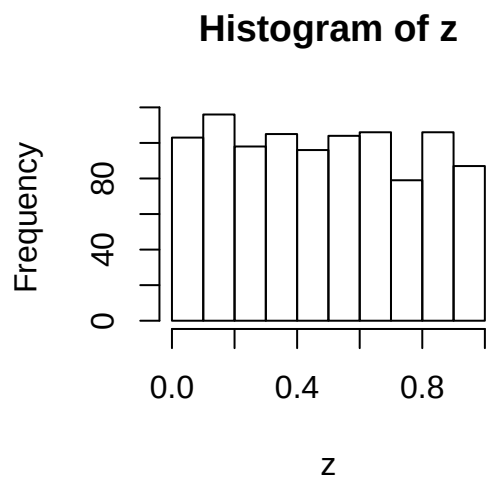
```
## Normal Distribution
x <- rnorm(1000)
hist(x)
```



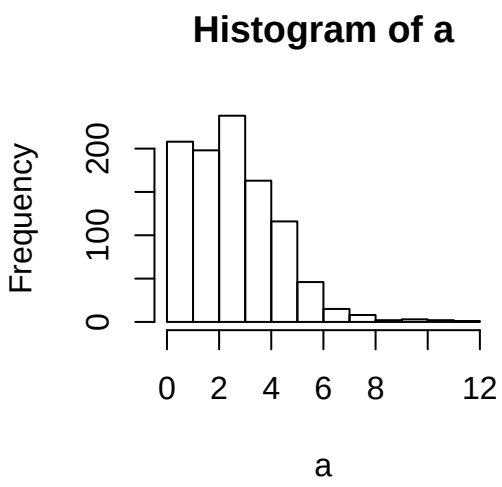
```r
y <- rnorm(1000, mean = 3, sd = 3)
hist(y)
```




```r
## Uniform Distribution
z <- runif(1000, min = 0, max = 1)
hist(z)
```




```r
## Poisson Distribution
a <- rpois(1000, lambda = 3)
hist(a)
```




```
## Binomial Distribution
## When size = 1, binom = bernoulli
b <- rbinom(3, size = 1, prob = 0.5)
b
```

```
## [1] 0 1 0
```

```
b <- rbinom(3, size = 3, prob = c(0.4, 0.4, 0.8))
b
```

```
## [1] 1 1 3
```

Other distributions can be trickier, like the multinomial distribution. Let's say we have three classes, the first occurs with probability 0.1, the second with probability 0.6, and the third with probability 0.3.

```
## Size is 1 because we only have one class per sample
m <- rmultinom(10, size = 1, prob = c(0.1, 0.6, 0.3))
## Each column is a sample, and each row is the number of successes for each class
m
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    0    0    0    0    0    0    0    0     0
## [2,]    1    1    1    1    0    1    1    1    1     0
## [3,]    0    0    0    0    1    0    0    0    0     1
```

```
## If we do it many times, roughly the right proportion of samples will go to
## each class
m <- rmultinom(1000, size = 1, prob = c(0.1, 0.6, 0.3))
rowMeans(m)
```

```
## [1] 0.091 0.623 0.286
```

But normally when we sample from a multinomial distribution, we want to instead get back the class labels, such as `1`, `2`, or `3`, not a column with the corresponding row denoting which class the observation is in. To do this more directly, we can use `sample`.

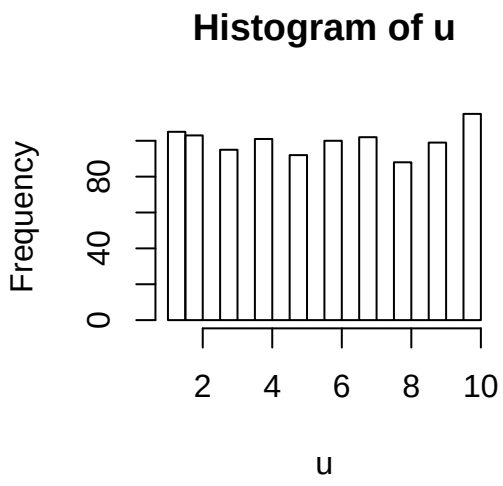
Sampling

Sampling from a vector is very useful for randomization inference, bootstrapping, drawing from discrete distributions, and more. For example, if you wanted to draw from a discrete uniform distribution, you could choose a series of cutpoints and split up a continuous uniform distribution, or we can sample from the numbers $\{1, 2, \ldots, 10\}$ with replacement.

```
## Sample from discrete uniform from 1-10
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
u <- sample(1:10, size = 1000, replace = T)
hist(u, breaks = 20)
```



Let's return to our multinomial example where we have three classes, the first occurs with probability 0.1, the second with probability 0.6, and the third with probability 0.3. We can just draw from the vector `c(1, 2, 3)` with replacement and use the `prob` argument of `sample`

```
m <- sample(1:3, size = 1000, replace = T, prob = c(0.1, 0.6, 0.3))
table(m)
```

```
## m
##   1   2   3
## 106 580 314
```

What if we want to use sample to just scramble up a vector? Let's say we have some vector of 0s and 1s and we want to mix it all up. We should just sample WITHOUT replacement to reorder the vector. This will be used in randomization inference.

```
x <- c(1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1)
xScramble <- sample(x)
xScramble
```

```
##  [1] 1 1 1 0 1 1 0 1 1 1 0 0 1 0 0 0
```

Writing functions

Often times you will need a certain procedure to run multiple times with only slightly different inputs. While computing the mean of a vector or plotting two vectors can be done using built-in functions, many times you need to write custom functions that can be quite complex. Functions are named, have arguments, and return some value. For example, imagine we wanted to write a custom mean function. The name of our function will be `myMean`. The argument to our function will be `x`, which is some numeric vector. Lastly the value we return will be the average of this vector.

```
myMean <- function(x) {
  xbar <- sum(x) / length(x)
  return(xbar)
}
myMean(c(1,2,3,4))
```

```
## [1] 2.5
```

Let's try a more complicated example. sImagine we wanted to generate data using the following model $y = \beta x + \epsilon$ where $x \sim N(0, 1)$ and $\epsilon \sim N(0, \zeta^2)$. Let's write a function that takes n, the number of observations, β, the effect of x on y, and ζ, the noise in the model. Let's return the OLS estimate of β, $\hat{\beta}$.

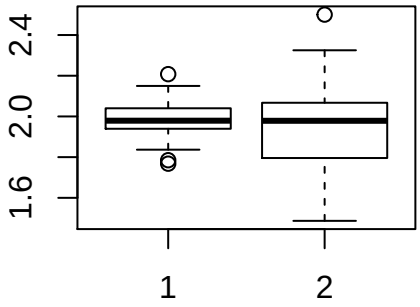
```
betaHat <- function(n, beta, zeta) {
  ## Generate x
  x <- rnorm(n)
  ## Generate eps
  eps <- rnorm(n, sd = zeta)
  ## Generate y
  y <- beta * x + eps

  ## Fit ols
  lmout <- lm(y ~ x)
  ## Extract betahat
  betahat <- lmout$coefficients["x"]

  return(betahat)
}

## An example
betaHat(n = 200, beta = 0.5, zeta = 3)
```

```
##        x
## 1.141647
```

Now we can use this function to see how well OLS performs under certain conditions. Let's run this function 100 times and compare the boxplots of the different values of $\hat{\beta}$.

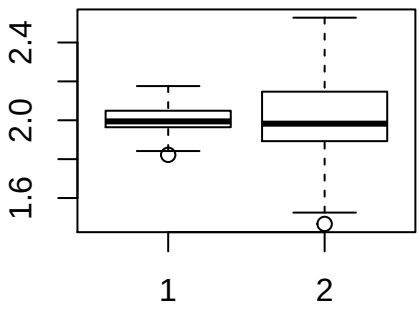
```
## Set some values (all besides zeta)
its <- 100
N <- 200
trueBeta <- 2

## Create container for beta hats
betasLowNoise <- numeric(its)
betasHighNoise <- numeric(its)

for (i in 1:its) {
```

```
  betasLowNoise[i] <- betaHat(n = N, beta = trueBeta, zeta = 1)
  betasHighNoise[i] <- betaHat(n = N, beta = trueBeta, zeta = 3)
}

## Plot outcomes
boxplot(betasLowNoise, betasHighNoise)
```



As expected, when there is less noise in the model, the variance of $\hat{\beta}$ will be less, and we are more likely to get the an estimate closer to the true value. We can also repeatedly call functions using `replicate`, a very useful function. Also, setting the seed is easier with `replicate` than it is with for loops.

```
set.seed(20160324)
betasLowNoise <- replicate(its, betaHat(n = N, beta = trueBeta, zeta = 1))
set.seed(20160324)
betasHighNoise <- replicate(its, betaHat(n = N, beta = trueBeta, zeta = 3))

## Plot outcomes
boxplot(betasLowNoise, betasHighNoise)
```


2.4
2.0
1.6
1
2