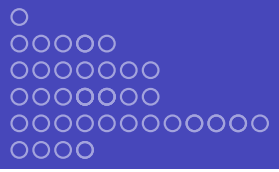
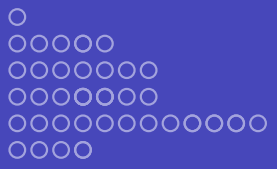


# Outline


1. Introduction
2. Survey
   - for
   - apply
   - lapply
   - lapply: Extended Example #1
   - lapply: Extended Example #2
   - mapply: a secret weapon
3. Bootstrapping



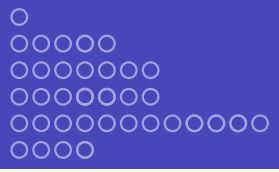
# Iteration: for, appply, etc

## Efficiency and Clarity

Paul E. Johnson[1][2]

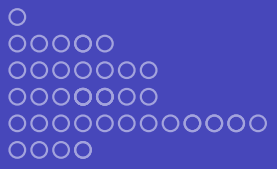
[1]University of Kansas, Department of Political Science [2]Center for Research Methods and Data Analysis

2013



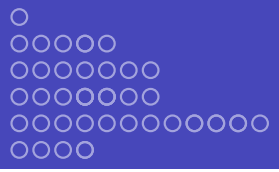
# Outline


1. Introduction
2. Survey
   - for
   - apply
   - lapply
   - lapply: Extended Example #1
   - lapply: Extended Example #2
   - mapply: a secret weapon
3. Bootstrapping



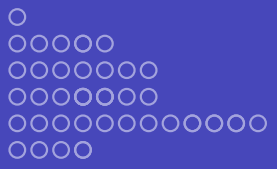
# Outline


1. Introduction
2. Survey
   - for
   - apply
   - lapply
   - lapply: Extended Example #1
   - lapply: Extended Example #2
   - mapply: a secret weapon
3. Bootstrapping



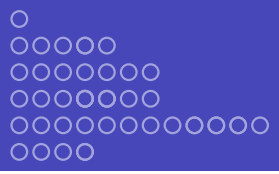
# R Frame of Mind

- Iteration is commonly needed
  - repeat the same thing over and over with new samples
  - process several subgroups of data (compare cities)
  - apply various functions to one data set
- Some idioms make code faster.
- Some idioms make code more understandable.



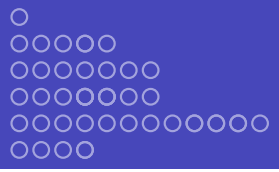
# Fit These Notes Into Context

- Use of iterators requires the ability to write small functions.
- If you have never written a small function for R, please review the lecture functions-1 before tackling this material.
- This lecture was once part of functions-1. In fact, it was the major motivation for functions-1, because I had to teach people how to write functions before using R apply statements.



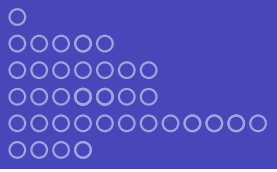
# Clarity and Understandability

- Especially in the early years of R, people who used for loops were ridiculed and urged to use `apply()` instead.
- Some ridicule was justified because code based on `for()` often makes heavy use of '[' to access data, and that is a very slow operator.
- I have examples of silly/slow code using `for()`
- However, if you have only a few situations to loop through, there is not usually a substantial speedup by recoding from `for()` to `apply()` (see Chambers, *Software for Data Analysis*)
- On the other hand, `for()` loops, especially nested loops, are prone to user-error and miscalculations, and they will be more difficult to read.



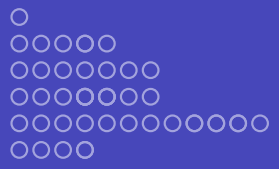
# Bootstrapping is at the End

- Difficult to be sure bootstrapping should be included in this lecture
- It is included here because people who are frustrated with R's apply concepts are also usually frustrated with bootstrapping in R.
- Why this makes a difference: Efficiency! People who do bootstrapping in the literal, obvious way, are generally wasting memory and time.



# Outline


1. Introduction
2. Survey
   - for
   - apply
   - lapply
   - lapply: Extended Example #1
   - lapply: Extended Example #2
   - mapply: a secret weapon
3. Bootstrapping



# R has lots of ways to do things over and over

- for loop: process by "i" or by "element"
- apply: process rows and/or columns in a matrix
- lapply: process each element in a list
- sapply: attempts to simplify output from lapply
- replicate: shorthand for sapply for simple simulations
- mapply: for functions that need several arguments, separately drawn from separate vectors or lists

# for looping

- First, I initialize x1, then
- loop over elements to set their values

```
doubleMe <- function(input = 0){
  newval <- 2 * input
}
x1 <- vector(mode = "numeric", length = 57)
for(i in 1:57) {x1[i] <- doubleMe(i)}
```

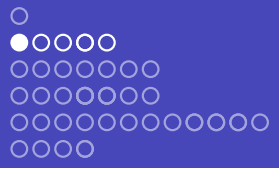
integers i from 1 to 57 are sent to double me, results collect

- Note, it is not necessary to actually do this for loop in R, because R is vectorized.

```
x2 <- doubleMe(1:57)
all.equal(x1,x2)
```

```
[1] TRUE
```

- Using vectorized code is much faster.



# "apply()"

- useRs are urged to avoid "for loops" when possible
- Why? Accessing particular values with "[" (vector or matrix indexes) is SLOW. Better to exploit R's "vectorization"
- apply() is one of a family of functions that can replace a for loop.
- apply() takes a matrix, and does "the same FUN" to all of its rows or columns (or both).
- Definition: MARGIN=1 means "work row by row", MARGIN=2 means "column by column"

# Example of "apply()" With a Built-In FUN

- Given a matrix xyz with columns "x", "y", and "z"
- On the columns, MARGIN=2, apply the R "mean" function.

```
xyz <- matrix( rnorm(9), ncol=3)
xyz
```

```
            [,1]       [,2]       [,3]
[1,]  0.5855288 -0.4534972  0.6300986
[2,]  0.7094660  0.6058875 -0.2761841
[3,] -0.1093033 -1.8179560 -0.2841597
```

```
colnames(xyz) <- c("x", "y", "z")
apply(xyz, MARGIN = 2, FUN = mean)
```

```
          x           y           z
0.39523051 -0.55518856  0.02325157
```

- If there is no "built in" function that does what you want, then you have to write your own.

# Write your own Function for apply

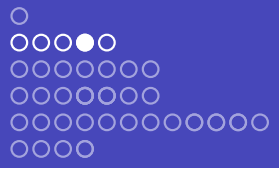
- Suppose you want the second-highest score from each column.
- Write a little function called "second()"

```
second <- function(acol = NULL){
    sort(acol)[2]
}
print(xyz)
```

```
              x          y          z
[1,]  0.5855288 -0.4534972  0.6300986
[2,]  0.7094660  0.6058875 -0.2761841
[3,] -0.1093033 -1.8179560 -0.2841597
```

```
apply(xyz, MARGIN = 2, FUN = second)
```

```
         x          y          z
 0.5855288 -0.4534972 -0.2761841
```

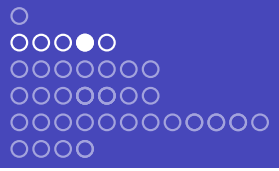


# Apply the normedEntropy function to rows

- The normedEntropy() function is presented in the lecture functions-1. I reproduce it for completeness here

```
divr <- function(p = 0){
  ifelse ( p > 0 & p < 1, -p * log2(p), 0)
}
entropy <- function(p){
    sum( divr(p) )
}
maximumEntropy <- function(N) - log2(1 / N)
normedEntropy <- function(x) entropy(x) /
    maximumEntropy(length(x))
```

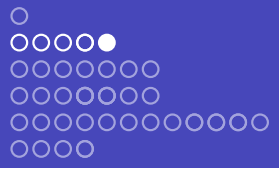
- First, create a matrix in which the sum of each row is 1.0



# Apply the normedEntropy function to rows ...

```
xmat <- matrix(rmultinom(6, size = 20, prob = c
    (1,2,3,4,5)), byrow = T, ncol = 5)
xmat <- prop.table(xmat, 1)
print(round(xmat,3 ))
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 0.00 0.30 0.15 0.20 0.35
[2,] 0.20 0.15 0.20 0.20 0.25
[3,] 0.10 0.15 0.10 0.30 0.35
[4,] 0.10 0.00 0.15 0.40 0.35
[5,] 0.05 0.10 0.30 0.35 0.20
[6,] 0.10 0.05 0.30 0.25 0.30
```

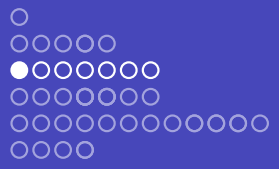


# Entropy for each row!

- apply normed Entropy to each Row with apply

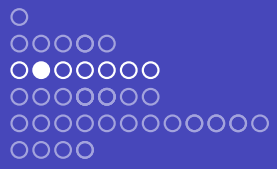
```
apply(xmat, MARGIN = 1, FUN = normedEntropy)
```

```
[1] 0.8295351 0.9921503 0.9156704 0.7759110 0.8888583 0
    .9003158
```



# "lapply()": Do same thing to all Elements of a List

- `lapply()` will take a list of things and apply a given function to each item, returning a new list.
  Generally,
  aNewList <- lapply( someList, FUN = someFunction )
- someFunction MUST accept the elements from someList *as the first argument*
- Additional arguments to someFunction are allowed



# Example Use of lapply

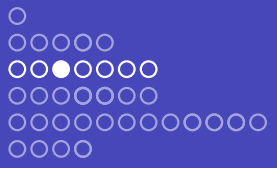
- Create a list with 5 sets of random uniform normal variables

```
sampleList <- lapply(rep(1000,5), rnorm)
sampleList[[1]][888]
```

```
[1] -0.3101479
```

- Same as

```
    sampleList <- list() ## or <- vector(``list'', 5)
    sampleList[[1]] <- rnorm(1000)
    sampleList[[2]] <- rnorm(1000)
    sampleList[[3]] <- rnorm(1000)
    sampleList[[4]] <- rnorm(1000)
    sampleList[[5]] <- rnorm(1000)
```



# Example Use of lapply

- Get the mean of sets 1 and 2 individually

```
mean(sampleList[[1]])
```

```
[1] 0.04081866
```

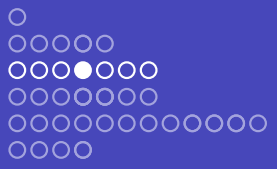
```
mean(sampleList[[2]])
```

```
[1] -0.02739241
```

- Grab means of all sets with lapply

```
(aNewList <- lapply(sampleList, mean))
```

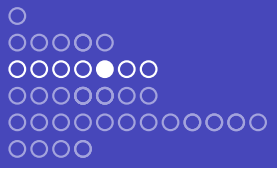
```
[[1]]
[1] 0.04081866

[[2]]
[1] -0.02739241

[[3]]
[1] -0.0255273
```



# Why lapply, Not apply?

- Sometimes our "data" is not an even set of columns that fits in a data.frame or matrix

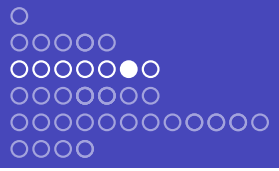
```
xlist <- list(x1 = c(1,1,1,2,3,3), x2 = rpois(10,lambda=3),
     x3 = round(rnorm(20,m=100,s=1),0))
elist <- lapply(xlist, function(x) { y <- table(x)/length(x
    ); normedEntropy(y)})
```



# Why lapply, not apply?

```
    for(i in 1:length(xlist)){
      cat("Given List")
      print(xlist[[i]])
      cat("Normed Entropy")
      print(round(elist[[i]],3))
      cat("\n")
      }
```

```
Given List[1] 1 1 1 2 3 3
Normed Entropy[1] 0.921

Given List [1] 3 2 5 2 5 2 1 6 2 4
Normed Entropy[1] 0.898

Given List [1] 101 101 100 101 100  99 101 100 100 102 100
    102 100  99 100 101 100 100 101 100
Normed Entropy[1] 0.843
```

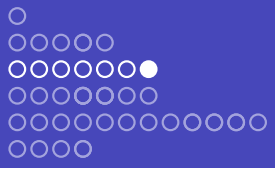


# Example with additional arguments

- One NA wrecks mean (by default)

```
sampleList <- lapply( rep(1000,5), rnorm)
sampleList[[1]][77] <- NA
(aNewList <- lapply(sampleList, mean))
```

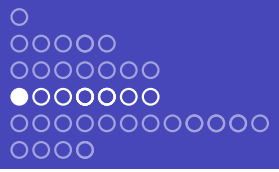
```
[[1]]
[1] NA

[[2]]
[1] -0.008354005

[[3]]
[1] -0.003276648

[[4]]
[1] -0.003438522

[[5]]
[1] 0.05110267
```



# Example (cont.): Fix that Missing Value Problem

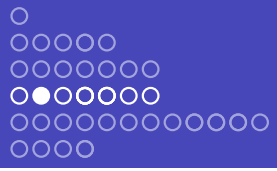
```
(aNewList <- lapply(sampleList, mean, na.rm = TRUE))
```

```
[[1]]
[1] -0.03336209

[[2]]
[1] -0.008354005

[[3]]
[1] -0.003276648

[[4]]
[1] -0.003438522

[[5]]
[1] 0.05110267
```



# Example: lapply to Simulate Regressions.

- The question:
  - Create 100 regression models from 100 data sets
  - Study the sampling distribution of the $R^2$ statistic from those regressions.

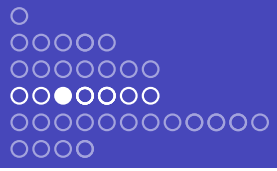


# Step 1.

- The following generates 100 data frames in a list "mydatasets".

```
exs <- 10
exq <- 0.345
exstde <- 20
createOneDF <- function(run, s = NA, q = NA, stde = NA
    ){
  x <- 18 + 43*runif(1000)
  y <- s + q * x + rnorm(1000, mean = 0, sd = stde)
  mydf <- data.frame(run,x,y)
}
mydatasets <- lapply(1:100, createOneDF, exs, exq,
    exstde)
```

- Here the "list" is just a sequence 1,2,3,...
- lapply automatically gives each list element to function as first argument. (In this case, "run" number).



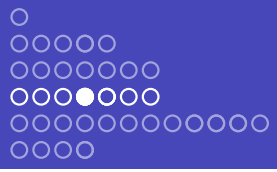
# Step 2.

- Now apply a function to each data frame, make list "myregressions"

```
myregressions <- lapply(mydatasets, FUN = function(
    mydf) lm(y ~ x, data = mydf))
```

- Note: small functions can be written "inline"
- Could as well have written

```
calcReg <- function(adf = NULL){
  mod <- lm(y ~ x, data = adf)
  }
myregressions <- lapply(mydatasets, FUN = calcReg)
```



# Take Stock of What We Have

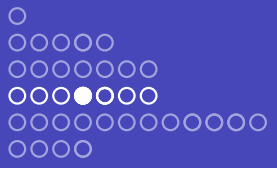
- Each element in the list "mydatasets" really is a data frame:

```
head( mydatasets[[33]])
```

```
  run         x          y
1  33 41.47315  30.817774
2  33 48.78788  48.229489
3  33 31.71107  45.515414
4  33 50.28991 -22.129543
5  33 60.13310  33.632953
6  33 35.67771   9.532895
```

- Each element in "myregressions" really is a regression result object

```
myregressions[[33]]
```

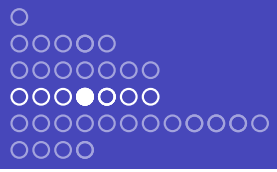


# Take Stock of What We Have ...

```
Call:
lm(formula = y ~ x, data = mydf)

Coefficients:
(Intercept)            x
    10.5261       0.3371
```

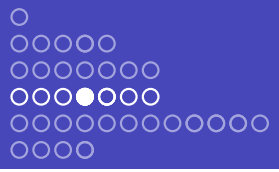
- Which can be summarized thus:

```
summary ( myregressions[[33]] )
```



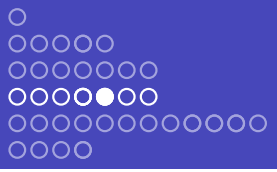
# Take Stock of What We Have ...

```
Call:
lm(formula = y ~ x, data = mydf)

Residuals:
    Min      1Q  Median      3Q     Max
-56.643 -11.595   0.873  12.462  57.854

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 10.52613    1.94869   5.402 8.26e-08 ***
x            0.33713    0.04737   7.117 2.10e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
    1

Residual standard error: 18.79 on 998 degrees of freedom
Multiple R^2:  0.0483,  Adjusted R^2:  0.04735
F-statistic: 50.66 on 1 and 998 DF,  p-value: 2.101e-12
```



# Take Stock of What We Have ...

Note, the $R^2$ value that we need is sitting there, in the middle of the summary output. We'll need that.



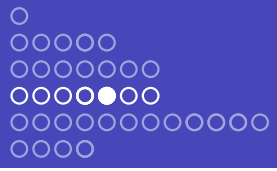
# Step 3.

- Grab the $R^2$ from each regression in the list.
- The estimate of the $R^2$ is an element in the returned object from summary.
- One strategy: create an R list of summary objects

```
mysummaries <- lapply(myregressions, FUN= summary)
```

- Getting the $R^2$ out of each one of those requires some tedious grabbing, such as

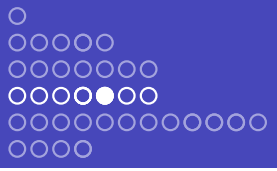
```
myrsq <- lapply(mysummaries, FUN = function(mr) {mr$
    r.square})
myrsq[1:5]
```



# Step 3. ...

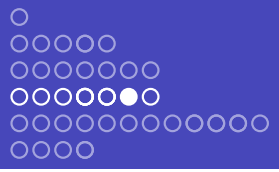
```
[[1]]
[1] 0.03758218

[[2]]
[1] 0.03746384

[[3]]
[1] 0.02569663

[[4]]
[1] 0.03390325

[[5]]
[1] 0.04059477
```

```
myrsq <- unlist(myrsq)
str(myrsq)
```



# Step 3. ...

```
num [1:100] 0.0376 0.0375 0.0257 0.0339 0.0406 ...
```



# Sapply will do that in one shot

- sapply is the "simplified apply", it attempts to convert a list into a vector or matrix.
- snoop through the regressions, grab the $R^2$.

```
myrsq <- sapply(mysummaries, FUN = function(mr) {mr$
    r.square})
mean(myrsq)
```

```
[1] 0.04510022
```

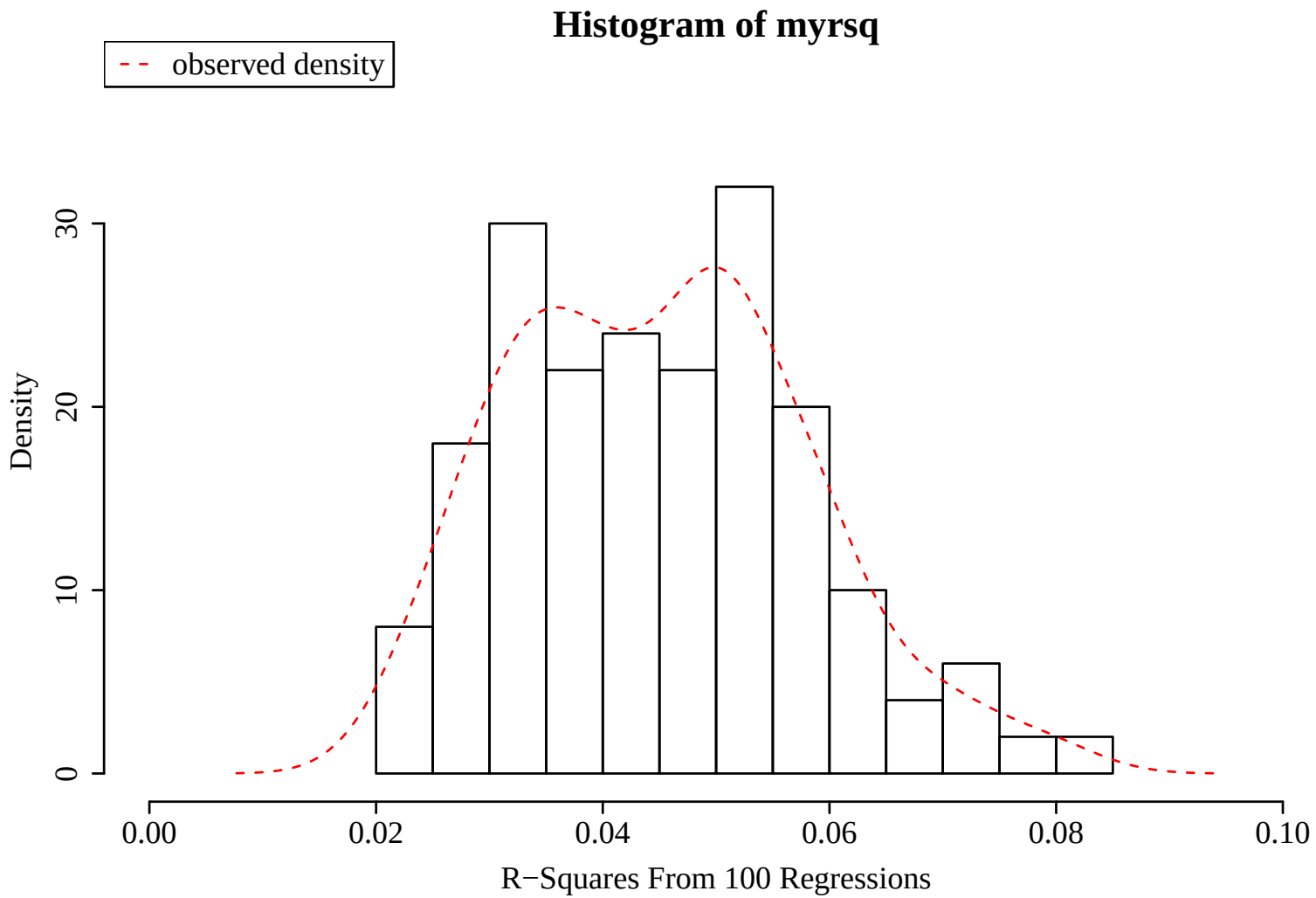
```
sd(myrsq)
```

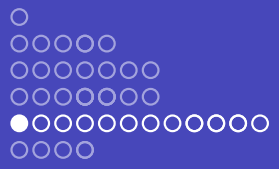
```
[1] 0.01280801
```

```
median(myrsq)
```

```
[1] 0.04424352
```

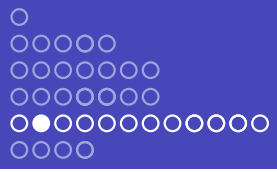
# Everybody Still Loves Histograms





# Example: Balance in Logistic Regression

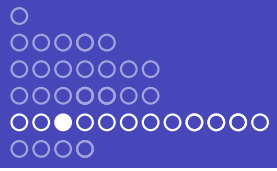
- Two years ago, I wondered (while auditing the categorical class), "what if we run a logistic regression comparing men and women and there are not very many men?"
- Write functions to
  - manufacture data
  - analyze data
  - summarize & plot data



# Create Output Data: Need to convert real numbers to 0's and 1's

$\eta$ "eta" is input, the proclivity to "vote democratic"

```
simLogit <- function(myeta){
    mypi <- exp(myeta) / (1 + exp(myeta)) ## SAME AS 1/(1+
        exp(-myeta))
    myunif <- runif(length(myeta))
    y <- ifelse(myunif < mypi, 1, 0)
}
```

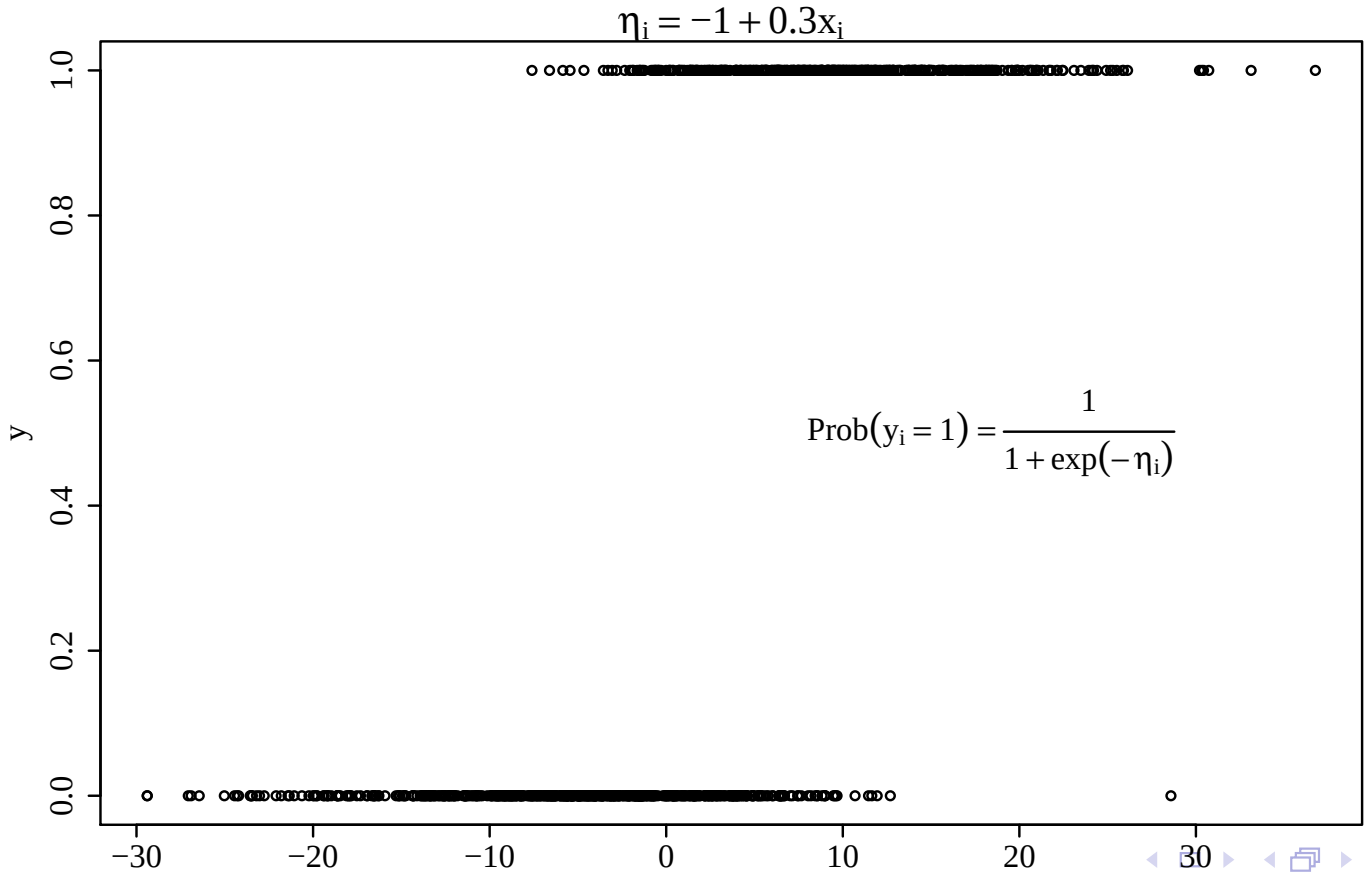


# Example Use: Creates 1000 Observations

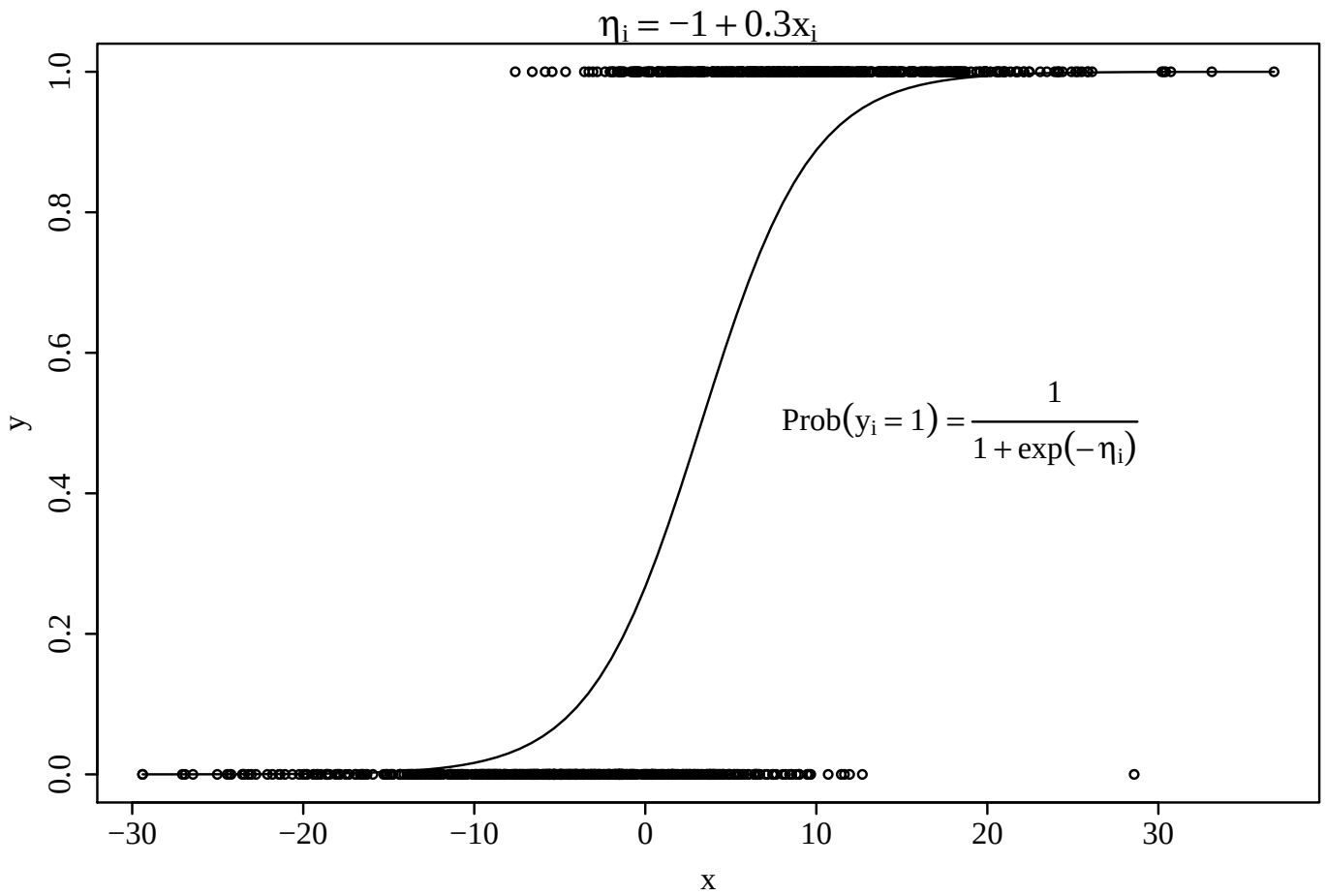
```
N <- 1000
A <- -1
B <- 0.3
x <- 1 + 10 * rnorm(N)
myeta <- A + B * x
y <- simLogit(myeta)
```

# Illustration of Simulated Data

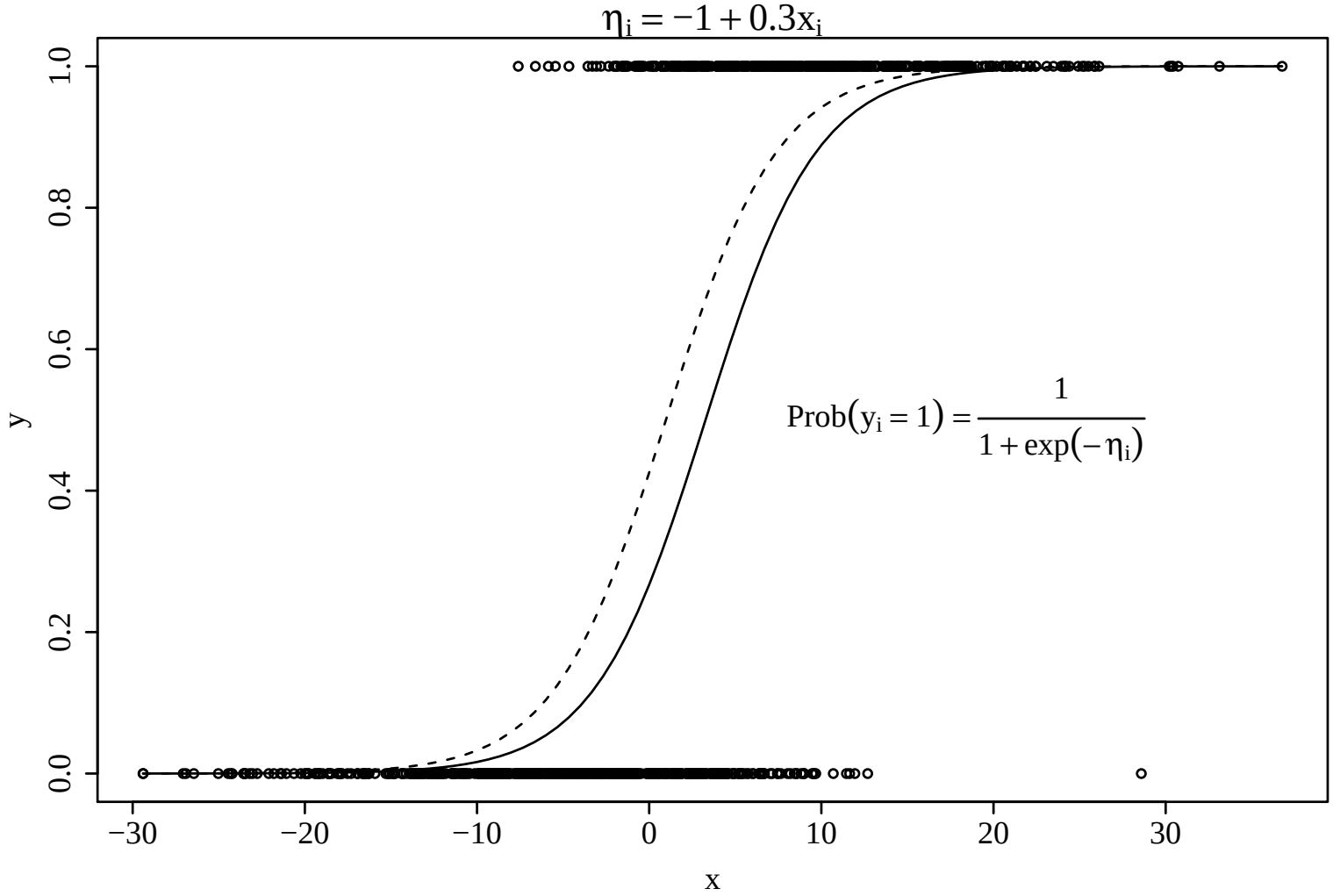
```
plot(x,y, main = bquote(eta[i]==.(A) + .(B) * x[i] ))
text ( 0.5*max(x), 0.5, expression( Prob( y[i]==1)==frac(
     1 , 1 + exp(-eta[i] ))))
```

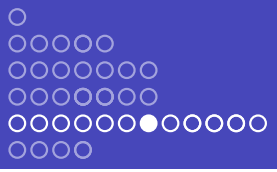


# The Fitted Line from glm



# We are Interested in the Difference Between Two Groups

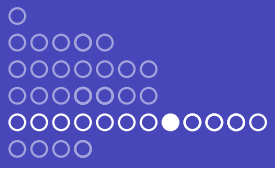




# Now Automate That Process

- Manufacture data
- Run Regression
- Return row of estimates

```
simUnbalanced <- function(iter=0, parm){
  A <- parm$A; B<- parm$B; C<- parm$C; PrFem <- parm$PrFem
  sex <- ifelse(runif(N) < PrFem,0,1)
  myeta <- A + B * x + C * sex
  sex <- factor(sex, levels = c(0,1), labels = c("M","F"))
  y <- simLogit(myeta)
  myglm2 <- glm( y ~ x + sex, family = binomial)
  myglm2sum <- coef(summary(myglm2))
  est <- myglm2sum[3,]
}
```

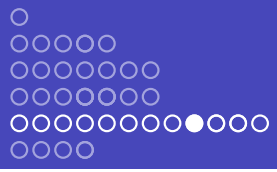


# Use sapply to run 1000 Regressions

```
p <- list()
p$A <- -1; p$B <- 0.3; p$C <- 0.4
p$PrFem <- 0.5
result45 <- list(sapply(1:1000, simUnbalanced, parm = p),
    parm = p)
```

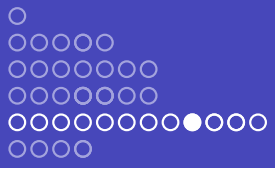
Note: I'm combining the sapply result, along with "p", for record-keeping

```
p$PrFem <- 0.9
result49 <- list(sapply(1:1000, simUnbalanced, parm = p),
    parm = p)
```



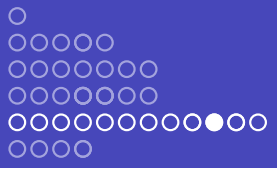
# Now Plan to Draw Some Figures

```
createFigs <- function(result){
  dat <- result[[1]]
  C <- result$parm$C
  PrFem <- result$parm$PrFem
  mybeta <- dat[1,]

  hrow1 <- hist(mybeta, breaks=50, plot=F)
  mybreaks <- hrow1$breaks

  breakMember <- cut(dat[1,], mybreaks)

  mypval <- dat[4,]
  mysignif <- ifelse((mypval < 0.05 ), 1, 0)
  df <- data.frame(mybeta, mypval, mysignif, breakMember)

  propsig <- by(df$mysignif, INDICES = list(df$breakMember)
      , mean, simplify = TRUE)
  mytrat <- dat[3,]
  mycounts <- hrow1$counts
```



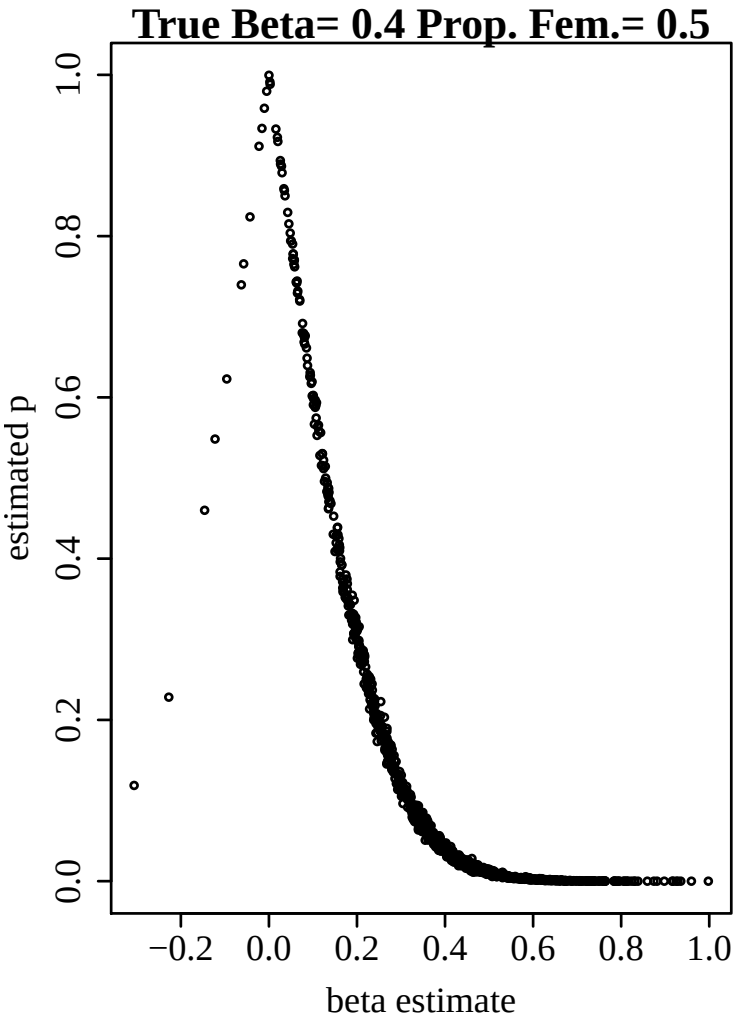
# Now Plan to Draw Some Figures ...

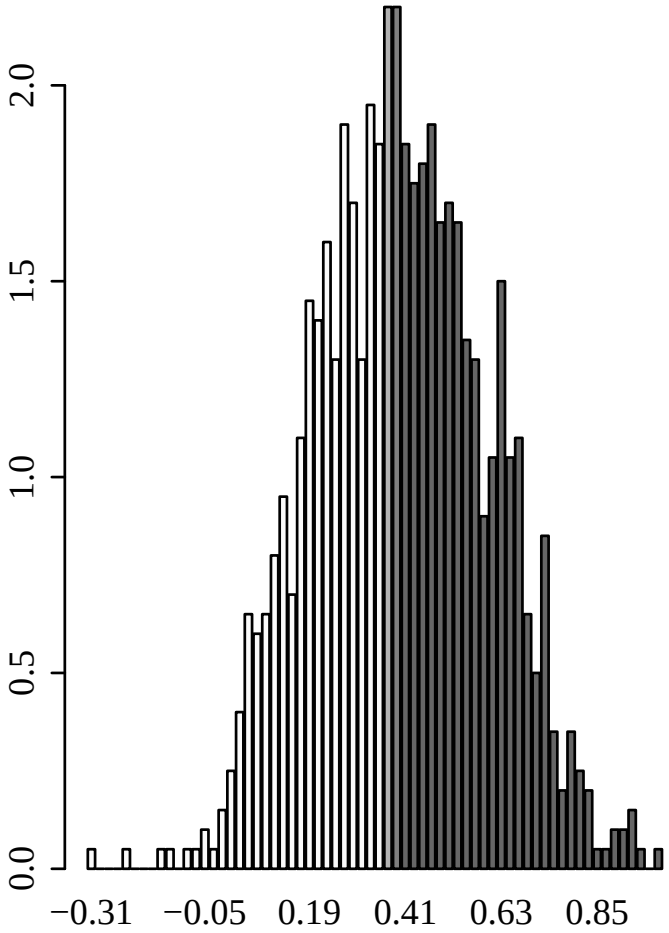
```
    plot(dat[1,], dat[4,], xlab = "beta estimate", ylab = "
        estimated p", cex = 0.7, main = paste("True Beta=",C,
         "Prop. Fem.=", PrFem))
    gc <- c("gray98","gray70","gray50","gray40")

    cut(propsig, breaks=c(-1,0.1,0.5,0.9,1.1))

    catpropsig <- cut(propsig, breaks = c(-1,0.1,0.5,0.9,1.1)
        , ordered = T, labels = c("0","lth","mth","1"))
    barplot(hrow1$density, col = gc[as.numeric(catpropsig)],
        names = hrow1$mids)
}
```

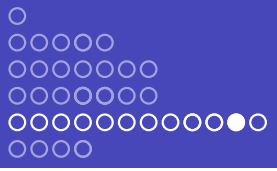


# For Balanced Data

# For Balanced Data ...

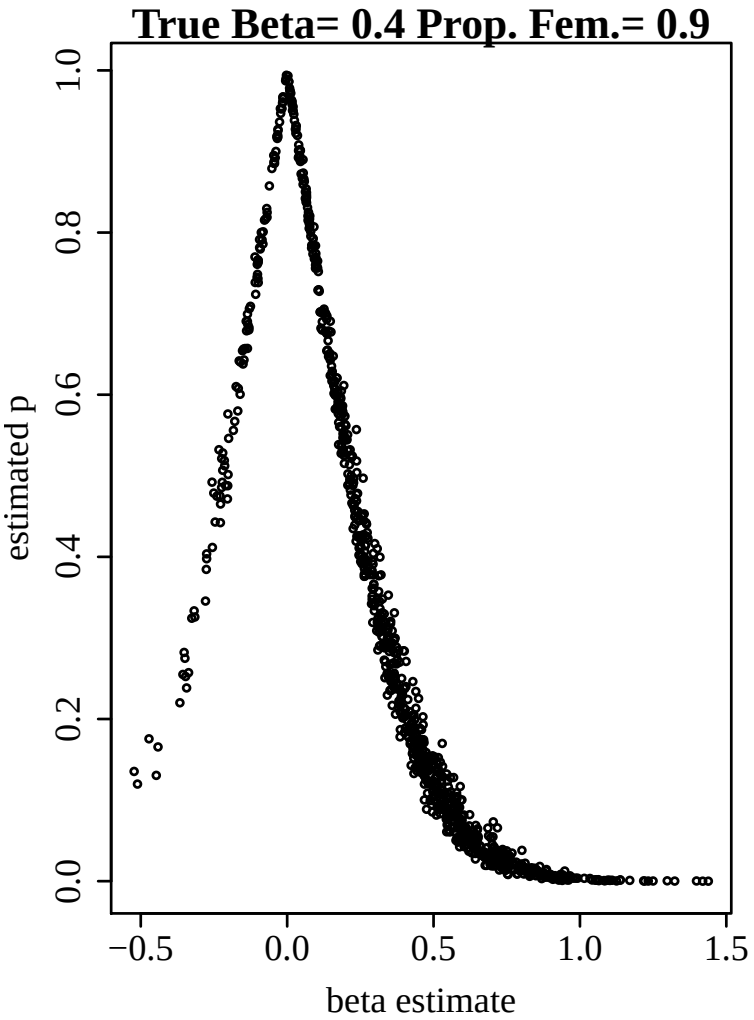


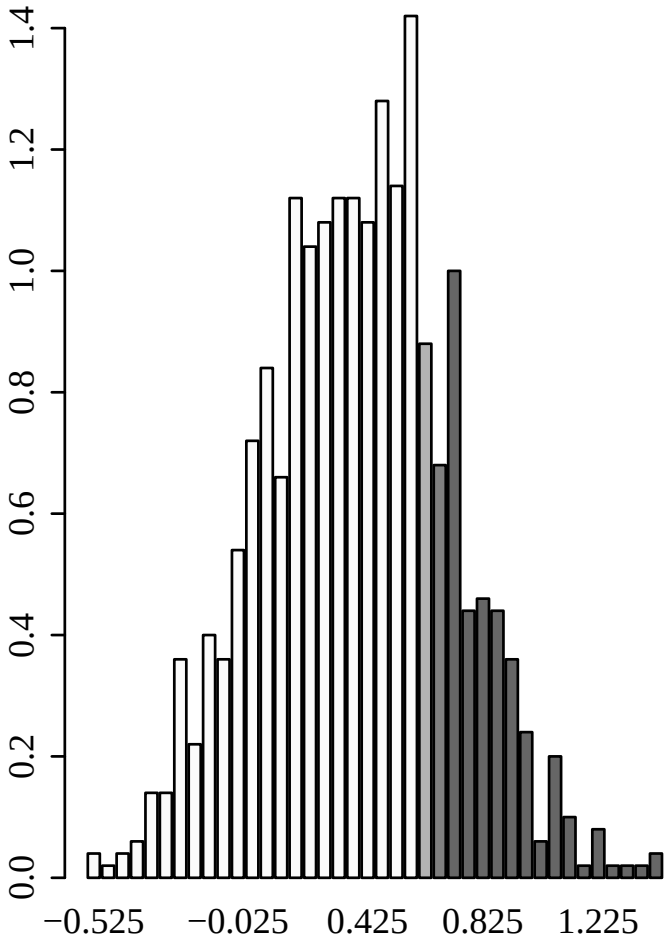



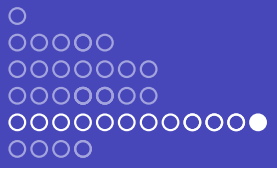


# For Unbalanced Data

# For Unbalanced Data ...

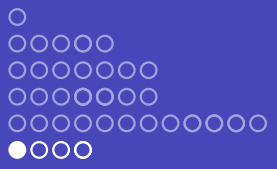




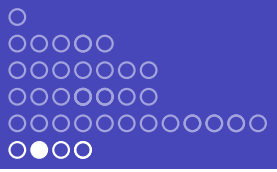
# Final Cautionary Note

- At some point, this approach will start to "bog down" under the weight of memory usage and CPU delays
- I'd suggest re-designing so that we separately create the data frames and run all of the analysis on each separately
- That would allow us to 1) stay within memory limits and 2) parallelize the work across separate cores or computers (see the R parallel package).



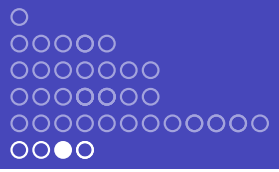
# mapply

- I never used mapply for the first 5 years of using R
- Now I see need for it at least once per month
- The documentation may be difficult to understand, but once you appreciate the beauty of it, you will like it.



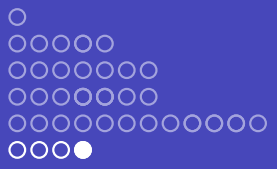
# When is mapply needed

- You have several vectors or lists of the same length
- You want to take the first element from each and do something.
- Then take the second element from each and do something
- Don't write nested "for" loops, as users are often tempted to do.



# Example of mapply usage in rockchalk package

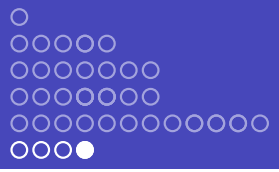
- rockchalk has many functions that are doing the same thing over and over for subsets of data.
- Run the examples for the `addLines()` function, you should see it integrates plotSlopes() and plotPlane() by transferring information.



# Example use of mapply in rockchalk 1.8

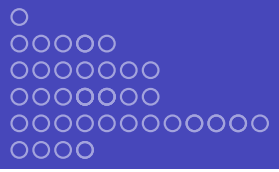
- dataSplits is a collection of data frames. We want to do the plot for each with the correct colors, which are stored in linesFrom variables col and lty.
- The small function drawLine accepts 3 arguments, one from data, one from col, one from lty.

```
if (!missing(linesFrom)) {
  dataSplits <- split(linesFrom$newdata, f = linesFrom
      $newdata[[linesFrom$call[["modx"]]]])
  drawLine <- function(nd, mycol, mylty) {
    lines(trans3d(nd[[plotx1]], nd[[plotx2]], nd$fit,
        pmat=res), col = mycol, lwd = lflwd, lty =
        mylty)
  }
  mapply(drawLine, dataSplits, linesFrom$col,
      linesFrom$lty)
}
```



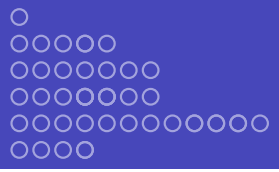
# Example use of mapply in rockchalk 1.8 ...

- Note we are free to name the variables inside drawLine however we want. That help keep our minds clear about whether we are talking about just one color or a vector of colors.



# Outline


1. Introduction
2. Survey
   - for
   - apply
   - lapply
   - lapply: Extended Example #1
   - lapply: Extended Example #2
   - mapply: a secret weapon
3. Bootstrapping



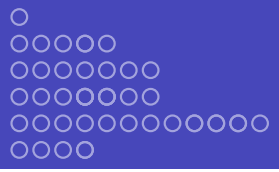
# Bootstrapping: Some "Do it Yourself" Work Is Required

- Many R functions require users to write little functions that do little things.
- In many cases (like lapply or apply), look for FUN as an argument.
- Sometimes no builtin-exists. useR must create!

# boot Function Requires a Special Function "statistic"

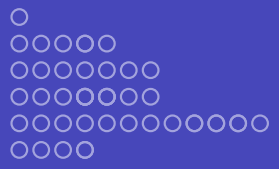
```
library(boot)
?boot
```

```
Bootstrap Resampling

Description:

Generate 'R' bootstrap replicates of a statistic applied to
    data.
Both parametric and nonparametric resampling are possible.
    ...

boot(data, statistic, R, sim = ''ordinary'', stype = ''i'',
strata=rep(1, n), L = NULL, m = 0, weights = NULL,
ran.gen=function(d, p) d, mle = NULL, simple = FALSE, ...)

statistic: A function which when applied to data returns a
    vector
containing the statistic(s) of interest...
```



# Bootstrap: Background Explanation

- Bootstrap: draw samples repeatedly and re-estimate $\theta$
- Resulting values approximate a sampling distribution $\theta$
- The "boot" package asks for a data frame and a special function "statistic". statistic must
  - accept a data frame as the first argument
  - accept an "index vector" as the second argument

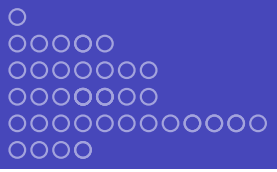


# Don't Panic: This is Confusing to Everybody

Example usage

```
boot(data, statistic = yourFunction, R = 1000)
```

- boot will iterate 1000 times, and yourFunction will provide the statistic of interest.
- You write yourFunction to make required calculation.
- boot will tell yourFunction which lines to use in the data frame, *over-and-over*.



# The Median of a Poisson Distribution

- Suppose you have a sample from a Poisson Process:

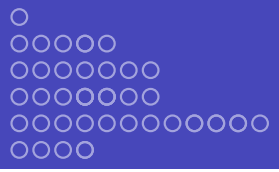
```
samp <- rpois(20, lambda=3)
```

- And you calculate the median:

```
median(samp)
```

```
[1] 2.5
```

- How confident are you in that estimate of the median?



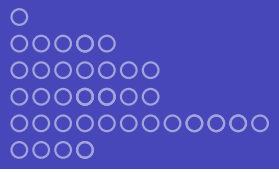
# Bootstrap Your Median

- Here is yourFunction:

```
calcMed <- function(dat, ind){
  median(dat[ind])
}
```

- dat[ind] has the effect of "pulling" rows that match "ind" from "dat"
- The boot function will send 1000 "case indexes" to your function.

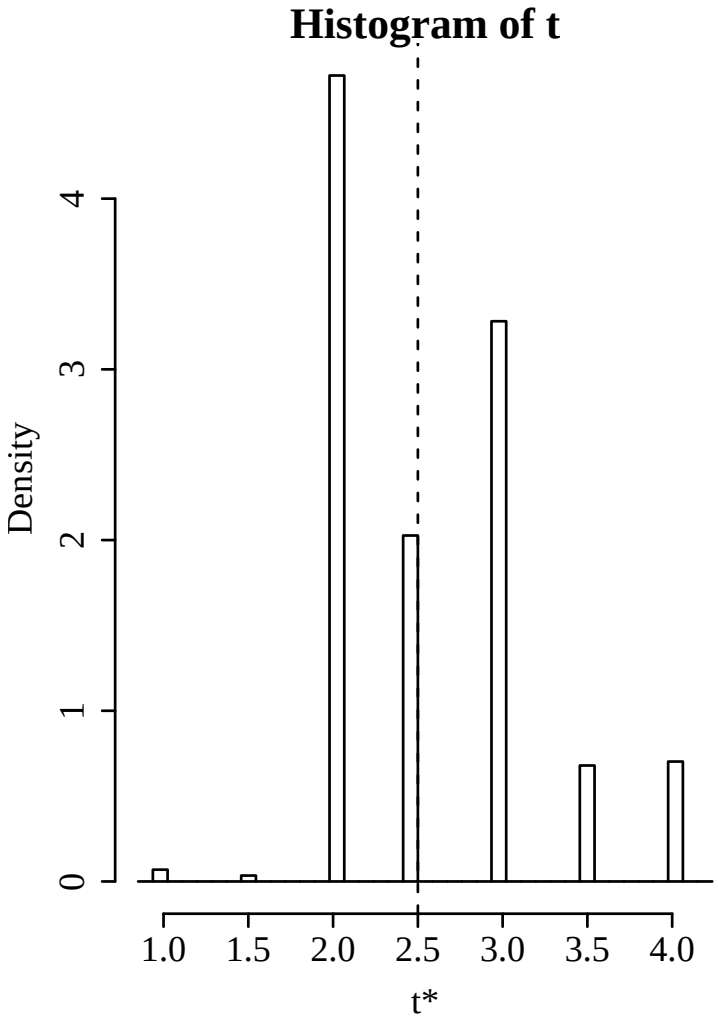
```
library(boot)
bpois <- boot(samp, calcMed, R = 1000)
bpois
```

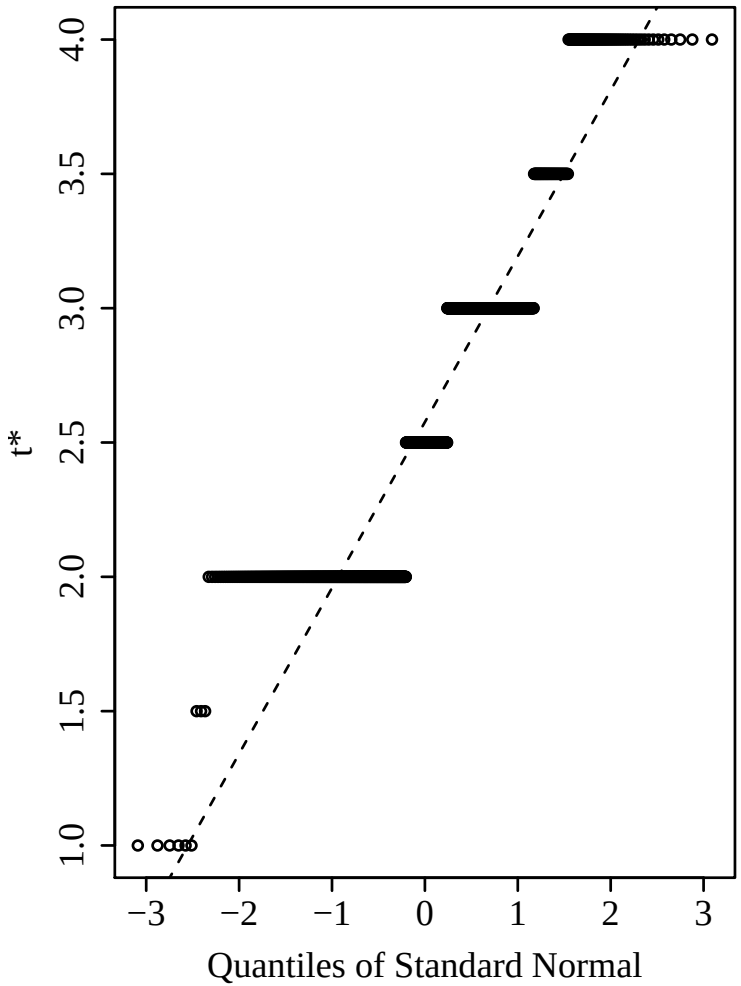


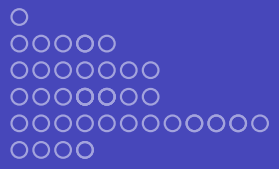
# Bootstrap Your Median ...

```
ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = samp, statistic = calcMed, R = 1000)


Bootstrap Statistics :
    original   bias    std. error
t1*      2.5    0.076   0.6173371
```

# Let's plot that





# Why Do They Do It That Way?

- Your instinct is to do this the "simple" way
  - (Just) "Manually" draw new random samples of rows from a data frame.
  - But: Creating 1000s of "new" re-sampled data sets would "waste" (exhaust?) memory
  - Would be especially slow if separate data sets have to be copied between systems.
- More efficient to keep 1 data frame, but 1000's of vectors of row numbers.