R Code to Complement Math Review Packet

A Note:

The notes and code in this document are meant to serve as complementary material to the Math Review Packet. In this program you will be able to perform many calculations in R, which provides more flexibility and data storage than a typical calculator. The R code below is meant to a) introduce you to using R as a calculator and b) give some context for how you could use R to solve a variety of problems.

We will cover:
- Logarithms
- Matrices
- Basic Probability
- Statistical concepts
* Summary Statistics
* The Central Limit Theorem
* Covariance and correlation
* Ordinary Least Square regression

Please use this as a resource to become more comfortable with R. You can run the R chunks by pressing the green play button in the top right corner (of the chunk). Or, you can run parts of the code and change portions of it in the Console. It may also be useful to look up functions in the **Help** tab (or type '?function' into the Console) to see function documentation.

You may also find it helpful to take notes in this document so that you can reference them during the first few months of classes. You can comment (by using '#' in the R chunk) as a way of annotating it in your own words. This is good practice for deciphering other people's code and is a good habit for documenting your own future work.

Note: In this document, you will also see LaTeX notation, which is used to render math symbols when you knit to PDF. If you hover over the items in the dollar signs, you will see what the formatting will look like when knitted to a PDF.

Properties of Logarithms

In R, the log() function computes natural logarithms (i.e., $ln()$). To calculate $log_{10}()$ use log10(); to calculate $log_2()$, use log2().

For example, if we wanted to solve $log_{10}(10000)$:

```
log10(10000)
```

```
## [1] 4
```

In general, we can calculate exponents using the ˆ character. For example, 3^8 could be calculated by typing 3ˆ8. Additionally, if we wanted to calculate e^4, we could use the exp() function.

```
3^8
```

```
## [1] 6561
```

```
exp(4)
```

```
## [1] 54.59815
```

Practice Problems.

Note: these problems are almost identical to those in the review packet. Try them by hand first; then use the code chunks below to verify that you get the same answers.

1. Calculate the following.

a. $log_e(e^3)$

```
log(exp(3))
```

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

b. $log_{10}(100)$

```
log10(100)
```

```
## [1] 2
```

c. $log_{10}(\frac{1}{10})$

```
log10(1/10)
```

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

d. $log_{10}(0)$ Note: There is no solution here; R returns -Inf

```
log10(0)
```

```
## [1] -Inf
```

Matrix Algebra

Creating a Matrix in R

Type ?matrix in the **Console** or matrix in **Help** to look at the inputs of the matrix function.

When you look at the usage of a function, you may see that some parameters are assigned default values, while others are assigned “NA” or “NULL” values (i.e., missing values). In the matrix function, we need to input a vector of values, the number of rows, number of columns, an indication of whether the matrix should be filled by row-wise (or column-wise), and dimension names if we want them. Note that, if we do not indicate a number of rows, the matrix will automatically have one row. Also note that, by default, the matrix will be filled column-wise (you can change the byrow parameter to TRUE to fill the matrix row-wise). If we want to store the matrix for future use, we need give a name to the matrix (“Z”, for example), and use an arrow (<-) or equals sign (=) to assign the result of the matrix function to that name. We can also use the print() function to print the matrices we have created.

For example $\mathbf{Z} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $\mathbf{Y} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$

```
#fill by row
Z <- matrix(data = c(1,2,3,4,5,6,7,8,9),nrow = 3, ncol = 3, byrow = TRUE)
print(Z)
```

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

```
#fill by column
Y <- matrix(data = (1:9), nrow = 3, ncol = 3, byrow = FALSE)
print(Y)
```

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

Transpose of a matrix

In R, we can get the transpose of a matrix by using the t() function

```
# Matrix A
A <- matrix(data = c(1,2,5,3,4,7), nrow = 2, ncol = 3, byrow = TRUE)

print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    5
## [2,]    3    4    7
```

```
# Transpose of A
t(A)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [3,]    5    7
```

Identity Matrix

```r
# Identity Matrix: use diag() to create an identity matrix
# I = 4x4 identity matrix
I <- diag(4)
print(I)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

```r
class(I) #tells you the object class; in this case, a matrix!
```

```
## [1] "matrix"
```

Diagonals

```r
X1 <- matrix(data = c(1,0,0,0,4,0,0,0,5), nrow = 3, byrow = TRUE)
print(X1)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    4    0
## [3,]    0    0    5
```

```r
# To obtain the obtain the values of the diagonal of an n x n matrix
diag(X1)
```

```
## [1] 1 4 5
```

```r
# We could also create an empty matrix and assign values to the diagonal
X2 <- matrix(data = 0, nrow = 3, ncol = 3)
diag(X2) <- c(1,2,3)
print(X2)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    2    0
## [3,]    0    0    3
```

Upper and Lower Triangular Matrices

```r
# Create 2 3 by 3 matrices of zeros (save them as V and W)
V <- W <- matrix(data = 0, nrow = 3, ncol = 3, byrow = TRUE)
print(W)
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

```r
# Use upper.tri() or lower.tri() to change the elements of the upper
# triangular matrix or lower triangular matrix, respectively.
# upper.tri() returns a 3 by 3 matrix with TRUE's in the upper triangle and
# FALSE's everywhere else:
upper.tri(W, diag = TRUE)
```

```
##       [,1]  [,2] [,3]
## [1,]  TRUE  TRUE TRUE
## [2,] FALSE  TRUE TRUE
## [3,] FALSE FALSE TRUE
```

```
# we can then assign values to the upper triangle of W using square brackets
W[upper.tri(W, diag = TRUE)] <- c(1:6)
print(W)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    4
## [2,]    0    3    5
## [3,]    0    0    6
```

```
# Lower Triangular
V[lower.tri(V, diag = TRUE)] <- c(1:6)
print(V)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    2    4    0
## [3,]    3    5    6
```

Inverse of a matrix

```
U <- matrix(data = c(1:4), nrow = 2, byrow = TRUE)
print(U)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```
# Use the solve() function to obtain the inverse of a matrix
invU <- solve(U)
print(invU)
```

```
##      [,1] [,2]
## [1,] -2.0  1.0
## [2,]  1.5 -0.5
```

```
# We can then perform matrix multiplication using %*%
# Note that, if we just use *, R will multiply the two matrices element-wise, instead of using
# Matrix multiplication

# Confirm you get an identity matrix
result1 <- U %*% invU
result2 <- invU %*% U
round(result1, digits=2) #this rounds to two decimal places
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```
round(result2, digits=2)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Determinant of a matrix

```
# Use det() to obtain the determinant of a matrix
det(U)
```

```
## [1] -2
```

Operations with matrices

a. Adding and subtracting matrices

```
# Using the above matrices (Z and Y) we will calculate Z+Y and Z-Y
# Note: the dim() function gives the dimensions of a matrix (rows first, then columns)
# Adding/subtracting matrices will produce a matrix of the same dimension
dim(Z)
```

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

```
dim(Y)
```

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

```
dim(Z + Y)
```

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

```
dim(Z - Y)
```

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

```
Z + Y
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    6   10   14
## [3,]   10   14   18
```

```
Z - Y
```

```
##      [,1] [,2] [,3]
## [1,]    0   -2   -4
## [2,]    2    0   -2
## [3,]    4    2    0
```

b. Multiplying a matrix by a scalar

```
# Multiplying a matrix by a scalar will produce a matrix of the same dimension
# Suppose that c = 3

3 * Z
```

```
##      [,1] [,2] [,3]
## [1,]    3    6    9
## [2,]   12   15   18
## [3,]   21   24   27
```

c. Matrix Multiplication

```
m1 <- matrix(data = c(3,4,2,5,1,2), nrow = 3, ncol = 2, byrow = TRUE)
print (m1)
```

```
##      [,1] [,2]
## [1,]    3    4
## [2,]    2    5
## [3,]    1    2
```

```
dim(m1)
```

```
## [1] 3 2
```

```
m2 <- matrix(data = c(7,2,1,3,5,2), nrow = 2, ncol = 3, byrow = TRUE)
print(m2)
```

```
##      [,1] [,2] [,3]
## [1,]    7    2    1
## [2,]    3    5    2
```

```
dim(m2)
```

```
## [1] 2 3
```

```
# Matrix multiplication
m1 %*% m2
```

```
##      [,1] [,2] [,3]
## [1,]   33   26   11
## [2,]   29   29   12
## [3,]   13   12    5
```

```
# It is important to note that when multiplying matrices we need to use %*% rather than *; using * will
```

Properties of matrix operations

1. $A + B = B + A$

```
Z + Y
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    6   10   14
## [3,]   10   14   18
```

```
Y + Z
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    6   10   14
## [3,]   10   14   18
```

2. $(A + B) + C = A + (B + C)$

```
(Z + Y) + X1
```

```
##      [,1] [,2] [,3]
## [1,]    3    6   10
## [2,]    6   14   14
## [3,]   10   14   23
```

```
Z + (Y + X1)
```

```
##      [,1] [,2] [,3]
## [1,]    3    6   10
## [2,]    6   14   14
## [3,]   10   14   23
```

3. $(AB)C = A(BC)$

```
(Z %*% Y) %*% X1
```

```
##      [,1] [,2] [,3]
## [1,]   14  128  250
## [2,]   32  308  610
## [3,]   50  488  970
```

```
Z %*% (Y %*% X1)
```

```
##      [,1] [,2] [,3]
## [1,]   14  128  250
## [2,]   32  308  610
## [3,]   50  488  970
```

4. $(A + B)C = AC + BC$

```
(Z + Y) %*% X1
```

```
##      [,1] [,2] [,3]
## [1,]    2   24   50
## [2,]    6   40   70
## [3,]   10   56   90
```

```
(Z %*% X1) + (Y %*% X1)
```

```
##      [,1] [,2] [,3]
## [1,]    2   24   50
## [2,]    6   40   70
## [3,]   10   56   90
```

5. If A is an $m \times n$ matrix, then $I_m A = A$ and $AI_n = A$

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    5
## [2,]    3    4    7
```

```
Im <- diag(x = 1, nrow = 2)
In <- diag(x = 1, nrow = 3)

Im %*% A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    5
## [2,]    3    4    7
```

```
A %*% In
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    5
## [2,]    3    4    7
```

Note that, in general $AB \neq BA$

```
Z %*% Y
```

```
##      [,1] [,2] [,3]
## [1,]   14   32   50
## [2,]   32   77  122
## [3,]   50  122  194
```

```
Y %*% Z
```

```
##      [,1] [,2] [,3]
## [1,]   66   78   90
## [2,]   78   93  108
## [3,]   90  108  126
```

Practice Problems

Note: these problems are identical to those in the review packet. Try them by hand first; then use the code chunks below to verify that you get the same answers.

1. Show each of the following by solving the left side of the equation in R:

a. $\begin{bmatrix} 2 & 4 & 2 \\ 1 & 3 & 0 \\ 1 & 6 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 5 & 0 \\ -2 & -3 & 0 \\ 1 & 9 & 5 \end{bmatrix} = \begin{bmatrix} 3 & 9 & 2 \\ -1 & 0 & 0 \\ 2 & 15 & 7 \end{bmatrix}$

```
M1 <- matrix(data = c(2,4,2,1,3,0,1,6,2),
             nrow = 3, ncol = 3, byrow = TRUE)
M2 <- matrix(data = c(1,5,0,-2,-3,0,1,9,5),
             nrow = 3, ncol = 3, byrow = TRUE)

M1 + M2
```

```
##      [,1] [,2] [,3]
## [1,]    3    9    2
## [2,]   -1    0    0
## [3,]    2   15    7
```

b. $\begin{bmatrix} 2 & 1 \\ -2 & 2 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} 5 & 2 & -1 \\ 3 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 13 & 8 & 0 \\ -4 & 4 & 6 \\ 26 & 16 & 0 \end{bmatrix}$

```
M3 <- matrix(data = c(2,-2,4,1,2,2), nrow = 3, ncol = 2, byrow = FALSE)
M4 <- matrix(data = c(5,3,2,4,-1,2), nrow = 2, ncol = 3, byrow = FALSE)

M3 %*% M4
```

```
##      [,1] [,2] [,3]
## [1,]   13    8    0
## [2,]   -4    4    6
## [3,]   26   16    0
```

c. Let $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 5 \\ 4 & 0 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 4 & 4 \\ 1 & 2 \\ 7 & 0 \end{bmatrix}$ $\mathbf{A}^T\mathbf{B} = \begin{bmatrix} 35 & 10 \\ 13 & 18 \end{bmatrix}$

```
A <- matrix(data = c(1,2,3,5,4,0), nrow = 3, ncol = 2, byrow = TRUE)
B <- matrix(data = c(4,1,7,4,2,0),nrow = 3, ncol=2, byrow = FALSE)

t(A) %*% B
```

```
##      [,1] [,2]
## [1,]   35   10
## [2,]   13   18
```

d. Using the same matrices as in part c, $\mathbf{B}^T\mathbf{A} = \begin{bmatrix} 35 & 13 \\ 10 & 18 \end{bmatrix}$

```
t(B) %*% A
```

```
##      [,1] [,2]
## [1,]   35   13
## [2,]   10   18
```

Show that **A** and **B** are inverses:

$$\begin{bmatrix}1 & 2 & 1\\2 & 2 & 0\\1 & 1 & 1\end{bmatrix}\begin{bmatrix}-1 & 0.5 & 1\\1 & 0 & -1\\0 & -0.5 & 1\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

```
A <- matrix(data = c(1,2,1,2,2,0,1,1,1), nrow = 3, ncol = 3, byrow = TRUE)

#Use the solve() to find the inverse of a matrix
solve(A)
```

```
##      [,1] [,2] [,3]
## [1,]   -1  0.5    1
## [2,]    1  0.0   -1
## [3,]    0 -0.5    1
```

```
B <- matrix(data = c(-1, 0.5, 1, 1, 0, -1, 0, -0.5, 1), nrow = 3, ncol = 3, byrow = TRUE)

#Use the solve() to find the inverse of a matrix
solve(B)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    1
## [2,]    2    2    0
## [3,]    1    1    1
```

```
A %*% B
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

3. Suppose that A is a 4x3 matrix and B is a $3x8$ matrix.

```
A <- matrix (data= (1:12), nrow = 4, ncol = 3, byrow = FALSE)
B <- matrix (data = (1:24), nrow = 3, ncol = 8, byrow = TRUE)
```

a. **AB** exists and is 4x8 matrix.

```
A %*% B
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]  199  214  229  244  259  274  289  304
## [2,]  226  244  262  280  298  316  334  352
## [3,]  253  274  295  316  337  358  379  400
## [4,]  280  304  328  352  376  400  424  448
```

```
dim(A%*%B)
```

```
## [1] 4 8
```

b. **BA** does not exist.

```
# Running B %*% A produces errors
```

4. The determinant of $det \begin{bmatrix}1 & -2\\4 & 3\end{bmatrix} = 11$

```
M5 <- matrix(data = c(1, -2, 4, 3), nrow = 2, ncol = 2, byrow = TRUE)
det(M5)
```

```
## [1] 11
```

Variables : Types and Summaries

Summary statistics for central tendency

```r
dat <- c(1, 2, 3, 5, 5, 4, 6, 6, 1, 2, 10, 11, 9, 9, 9)  #save a vector of values

# Mean
mean(dat)
```

```
## [1] 5.533333
```

```r
# Median
median(dat)
```

```
## [1] 5
```

```r
# Mode -- R does not have a built in function for mode but you can use the
# table function to see what the most common value is
table(dat)
```

```
## dat
##  1  2  3  4  5  6  9 10 11
##  2  2  1  1  2  2  3  1  1
```

```r
as.numeric(names(table(dat))[which.max(table(dat))])  #an ugly hack to find the mode
```

```
## [1] 9
```

Summary statistics for spread

```r
# Range
#the range function actually returns the minimum and maximum of dat in a vector
range(dat)
```

```
## [1]  1 11
```

```r
#to calculate the range, we can subtract the first element in range(dat) (the minumum)
#from the second element in range(dat) (the maximum)
range(dat)[2]-range(dat)[1]
```

```
## [1] 10
```

```r
#another option is to use the diff() function, which calculates the difference between two values
diff(range(dat))
```

```
## [1] 10
```

```r
# Variance of a set of values can be calculated using the var() function
var(dat)
```

```
## [1] 11.55238
```

```r
# Standard Deviation can be calculated using the sd() function
sd(dat)
```

```
## [1] 3.398879
```

```r
# Show that the square root of the variance gives the same output as sd()
sqrt(var(dat))
```

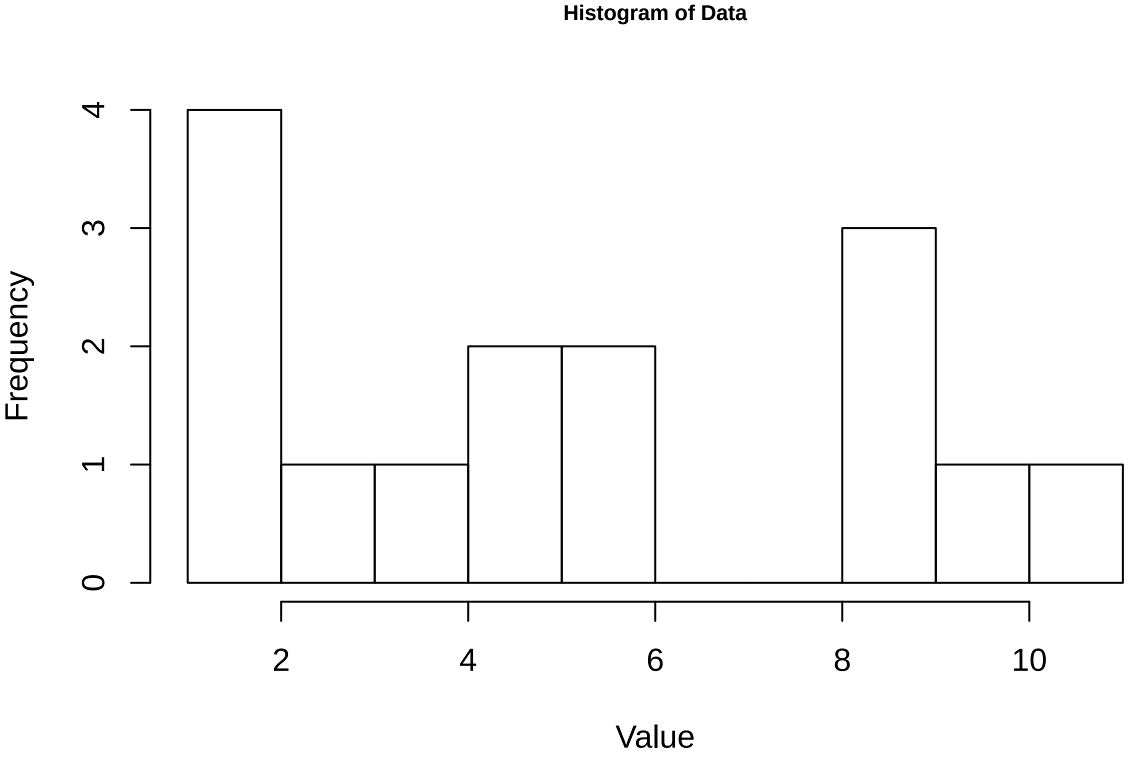
```
## [1] 3.398879
```

```
#this creates a 2 by 2 grid in which to place the following three plots
par(mfrow=c(2,2))
```

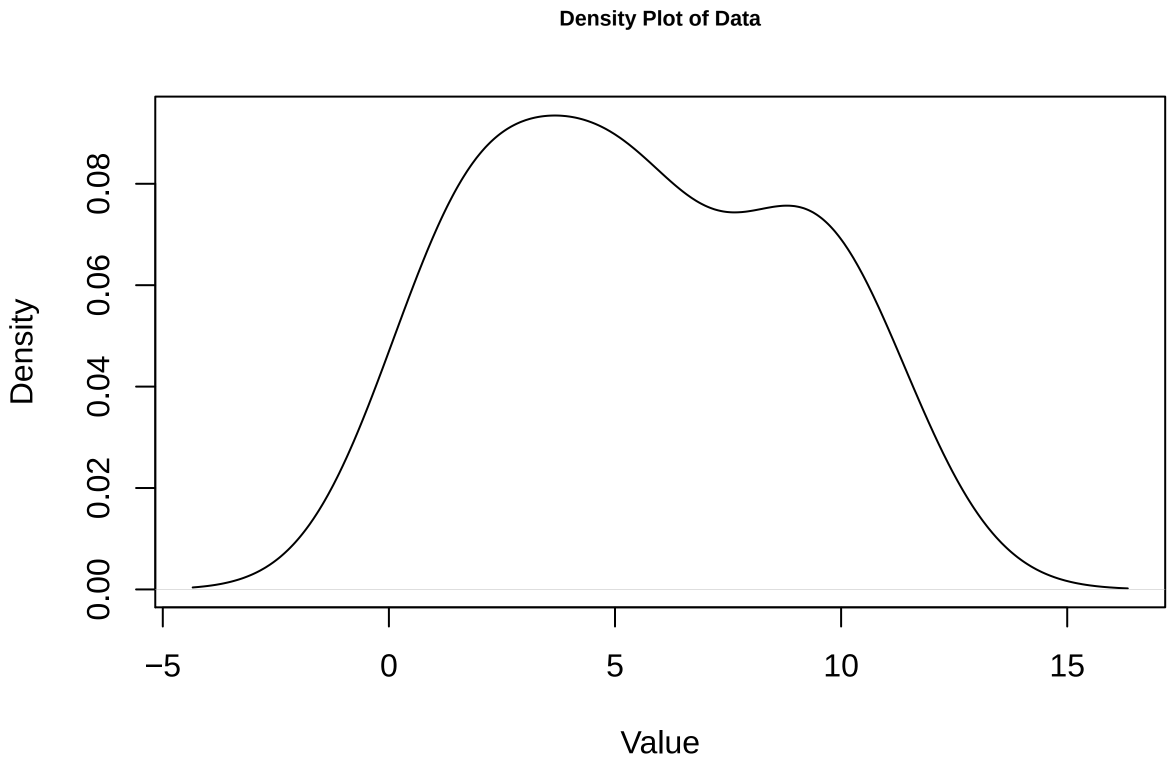
Summarizing variables visually

The following code can be used to make plots similar to those in the math review packet. We have copied the code here so that you can take a look!

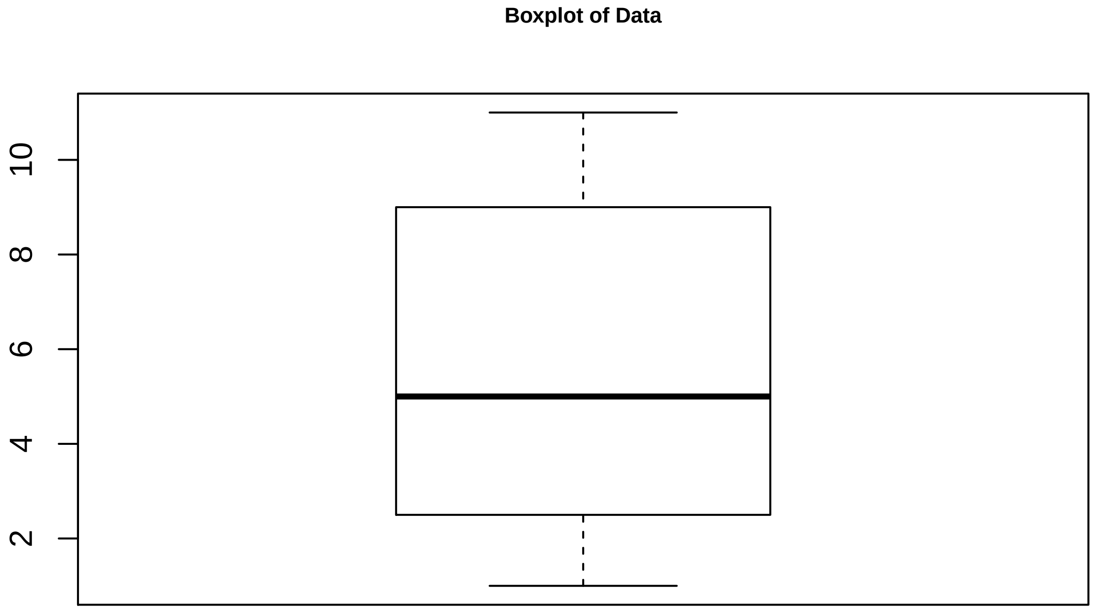
```
#plot a histogram of the salaries:
hist(dat,
     main = "Histogram of Data",
     cex.main = 0.7,
     xlab = "Value",
     breaks = 10)
```



```
#plot the smoothed density of the salaries:
plot(density(dat),
     main = "Density Plot of Data",
     cex.main = 0.7,
     xlab = "Value")
```



```
#make a boxplot of the salaries:
boxplot(dat,
        main = "Boxplot of Data",
        cex.main = 0.7)
```



Basic Probability

Note: the R code in this section uses some more advanced techniques (like loops and functions) that will not be formally covered until later. For now, do your best to understand the code, but focus on running it and understanding the output.

Sample Spaces

For an experiment E, the set of all possible outcomes of E is called the **sample space** and is denoted by the letter S. For a coin-toss experiment, S would be the results “Head” or “Tail”, which we may represent by $S = \{H, T\}$.

Consider the experiment of dropping an empty Styrofoam cup onto the floor from a height of four feet. The cup hits the ground and eventually comes to rest. It could land upside down, right side up, or it could land on its side. Here, the sample space S is $\{”down”, ”up”, ”side”\}$.

With more outcomes, the sample space becomes harder to write out. If we were to flip *two* coins, the result could be two heads, two tails, one head then one tail, or one tail then one head. The sample space S is then $S = \{HH, HT, TH, TT\}$

What about three coins?

Instead of spending the time writing out sample spaces, we can have R do it for us. One of the easiest ways to do this would be to simulate the coin flips a number of times, and then use the `table()` function to show what outcomes were produced. If we run the experiment enough times (10,000 should be more than enough), we can be reasonably sure that all outcomes will appear. Unlike the previous section, the coin flip function here will produce the letters H and T for heads and tails.

```r
flip3 <- function() {
  # A function to flip 3 coins - we'll sample three separately and then use the
  #   paste() function to paste the three together.
      temp <- sample(c("H","T"), size=3, replace=T)
      return(paste(temp, collapse="")) #collapse turns three results into one "word"
}

# Initializing the vector to store all the experiment outcomes (sets of 3 flips)
outcomes <- vector()

# Do the experiment 10,000 times
for (i in 1:10000) {
    outcomes[i] <- flip3()
}

# The unique command prints only the unique outcomes in a vector.
unique(outcomes)
```

```
## [1] "TTT" "HHH" "TTH" "THH" "HTT" "HTH" "THT" "HHT"
```

Now use R to list the Sample Space for the experiment of dropping three styrofoam cups where each one can land down, up, or sideways.

```r
## ANSWER:
drop3 <- function() {
  temp <- sample(c("U","D","S"), size=3, replace=T)
  return(paste(temp, collapse=""))
}
```

```
outcomes <- vector()

for (i in 1:10000) {
  outcomes[i] <- drop3()
}

unique(outcomes)
```

```
##  [1] "SSD" "DUU" "DSS" "SSS" "UUU" "SDD" "DDD" "DDU" "UUD" "SSU" "SDU"
## [12] "SUD" "UUS" "UDU" "USS" "DSD" "UDS" "USU" "UDD" "SUU" "SUS" "SDS"
## [23] "DUD" "DDS" "DSU" "USD" "DUS"
```

```
length(unique(outcomes))
```

```
## [1] 27
```

Sampling from an *urn* is a canonical type of experiment in probability class. The urn contains a bunch of distinguishable objects (i.e. balls) inside. We shake up the urn, reach inside, grab a ball, and take a look. In this simple version, the sample space would just depend on how many types of balls are in the urn. If some are red and some are blue, the sample space for picking one ball would be $S = \{R, B\}$

Suppose you have an urn containing three balls numbered 1 - 3. If we draw two balls from an urn, what are all of the possible outcomes of the experiment? It depends on how we sample. We could select a ball, take a look, put it back, and sample again (sampling **with replacement**). Another way would be to select a ball, take a look – but **not** put it back, and sample again (sampling **without replacement**.)

Use R to build the sample space for sampling 2 balls *with replacement* from an urn containing three balls numbered 1-3.

```
## ANSWER:
pick2 <- function() {
      temp <- sample(c(1,2,3), size=2, replace=T)
      return(paste(temp, collapse=" "))
}

outcomes <- vector()

for (i in 1:10000) {
    outcomes[i] <- pick2()
}

unique(outcomes)
```

```
## [1] "2 2" "1 3" "3 3" "1 2" "2 1" "3 2" "2 3" "3 1" "1 1"
```

How about the same thing, but sampling *without replacement*?

```
## ANSWER:
pick2 <- function() {
      temp <- sample(c(1,2,3), size=2, replace=F)
      return(paste(temp, collapse=" "))
}

outcomes <- vector()

for (i in 1:10000) {
```

```r
    outcomes[i] <- pick2()
}

unique(outcomes)
```

```
## [1] "2 3" "1 2" "3 2" "3 1" "2 1" "1 3"
```

What is the difference between the two?

> ANSWER: Sampling without replacement didnt have any repeats (e.g. 1 1 or 2 2).

Suppose we do not actually keep track of which ball came first. All we observe are the two balls, and we have no idea about the order in which they were selected. We call this **unordered sampling** (the opposite, and what we were doing before is *ordered sampled*) because the order of the selections does not matter with respect to what we observe.

Challenge question - write R code to simulate picking 2 balls from on urn of 3 numbered balls *with* replacement, if it were *unordered sampling*

```r
## ANSWER:
pick2 <- function() {
      temp <- sort(sample(c(1,2,3), size=2, replace=T))
      return(paste(temp, collapse=" "))
}

outcomes <- vector()

for (i in 1:10000) {
    outcomes[i] <- pick2()
}

unique(outcomes)
```

```
## [1] "1 1" "2 3" "2 2" "1 3" "3 3" "1 2"
```

Events

An event is a specific collection of outcomes, or in other words, a **subset of the sample space**. After the performance of a random experiment, we say that the event A *occurred* if the experiment's outcome *belongs to* A.

Let's create the sample space for the experiment consisting of flipping three coins:

```r
## Generate the sample space for flipping a coin 3 times.
## Used in the code below
flip3 <- function() {
      temp <- sample(c("H","T"), size=3, replace=T)
      return(paste(temp, collapse=""))
}
outcomes <- vector()
for (i in 1:10000) {
  outcomes[i] <- flip3()
}
sample.space <- unique(outcomes)
```

Brackets [ ]

R has different ways to find subsets. Square brackets select certain elements of a vector:

```
print(sample.space)
```

```
## [1] "THH" "THT" "TTH" "HTT" "HHH" "TTT" "HTH" "HHT"
```

```
sample.space[2]
```

```
## [1] "THT"
```

```
sample.space[1:3]
```

```
## [1] "THH" "THT" "TTH"
```

```
sample.space[c(2,5)]
```

```
## [1] "THT" "HHH"
```

The `%in%` function

The function %in% helps to learn whether each value of one vector lies somewhere inside another vector. Consider set x, the numbers from 1 to 10, and set y, the numbers from 8 to 12.

```
x <- 1:10
y <- 8:12
y %in% x
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE
```

```
y[which(y %in% x)]
```

```
## [1]  8  9 10
```

Notice that the returned value of the first line is a vector of length 5 which tests whether each element of y is in x, in turn. The returned value of the second line are the elements of y that are also elements of x.

Union, Intersection and Difference

Given subsets A and B, it is often useful to manipulate them in an algebraic fashion. We have three set operations at our disposal to accomplish this: **union**, **intersection**, and **difference**. Additionally we can take the **complement** of one of the sets. Below is a table that summarizes the pertinent information about these operations.

Name	Notation	Definition	R Function
Union	$A \cup B$	In either A or B or both	`union(A,B)`
Intersection	$A \cap B$	In both A and B	`intersect(A,B)`
Difference	$A \setminus B$	In A but not in B	`setdiff(A,B)`

Find the union, intersect, and difference of `x` and `y` using R:

```
x <- 1:10
y <- 8:12
```

```
## Answer:
union(x,y)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
intersect(x,y)
```

```
## [1]  8  9 10
```

```
setdiff(x,y)
```

```
## [1] 1 2 3 4 5 6 7
```

Switch the order of the variables. For which operations do the results stay the same? For which are they different? Why?

```
## ANSWER:
union(y,x)
```

```
##  [1]  8  9 10 11 12  1  2  3  4  5  6  7
```

```
intersect(y,x)
```

```
## [1]  8  9 10
```

```
setdiff(y,x)
```

```
## [1] 11 12
```

```
## Answer:
# The order doesnt matter in the first two because the operations are symmetrical.
# The difference is not symmetrical because we're taking the area of the second from the area of the fi
```

Complements

If we know the Sample Space, we can also use `setdiff` to take the **complement** of a subset. A complement is the entire part of the sample space *not* in the subset. Set up a new sample space containing the numbers 1 to 15 and find the complement of x in this sample space.

```
## ANSWER:
s <- 1:15 # A sample space of number 1-15

setdiff(s,x) # The complement of x in sample space s
```

```
## [1] 11 12 13 14 15
```

Random Variables and Probability Density/Mass Functions

PDF-related functions in R

We can sample from many distributions in R. For example, we can use the rnorm() function to sample from a normal distribution. Note that we can also draw from a variety of other distributions in R by using functions like rt(), rchisq(), runif(), etc.

There are four main functions that you will use for probability distributions in R. Below are examples for the normal distribution, but you can extrapolate what similar functions for other probability distributions would be called:
'dnorm' : gives the height of the probability density function of a normal distribution at any value, x
'pnorm' : computes probabilities from a normal distribution (by default: area under the curve on the interval $(-\infty, x)$, where you specify x). This is also called a percentile.
'qnorm' : computes quantiles from a normal distribution (essentially the inverse of pnorm)
'rnorm' : randomly draws values from a normal distribution

Here are some examples of how to use these functions in R. Note that pnorm(1.96) is approximately .975, whereas qnorm(.975) is approximately 1.96

```
#calculate height of a standard normal PDF at x=0
dnorm(x=0, mean=0, sd=1)
```

```
## [1] 0.3989423
```

```
#calculate the area under the curve of a standard normal PDF on the interval (-Inf,1.96)
pnorm(q=1.96, mean=0, sd=1)
```

```
## [1] 0.9750021
```

```
#calculate the quantile for a 97.5th percentile value on a standard normal PDF
qnorm(p=.975, mean=0, sd=1)
```

```
## [1] 1.959964
```

```
#sample 5 values from a normal distribution with mean=0 and sd=1
rnorm(n=5, mean=0, sd=1)
```

```
## [1] -0.008472447  1.036448128 -0.595676238 -2.005935192  0.254400229
```

Central Limit Theorem Introduction

In the example with average salary of “Data Scientists” we have a sample (n = 400), a mean salary of 100,000 and standard deviation of 10,000.

Suppose we want R to randomly sample 400 values from a normal distribution with mean 100,000 and a standard deviation of 10,000 (note that, for the CLT to hold in this example, it was not necessary for the salaries to be normally distributed). We can do this in R as follows:

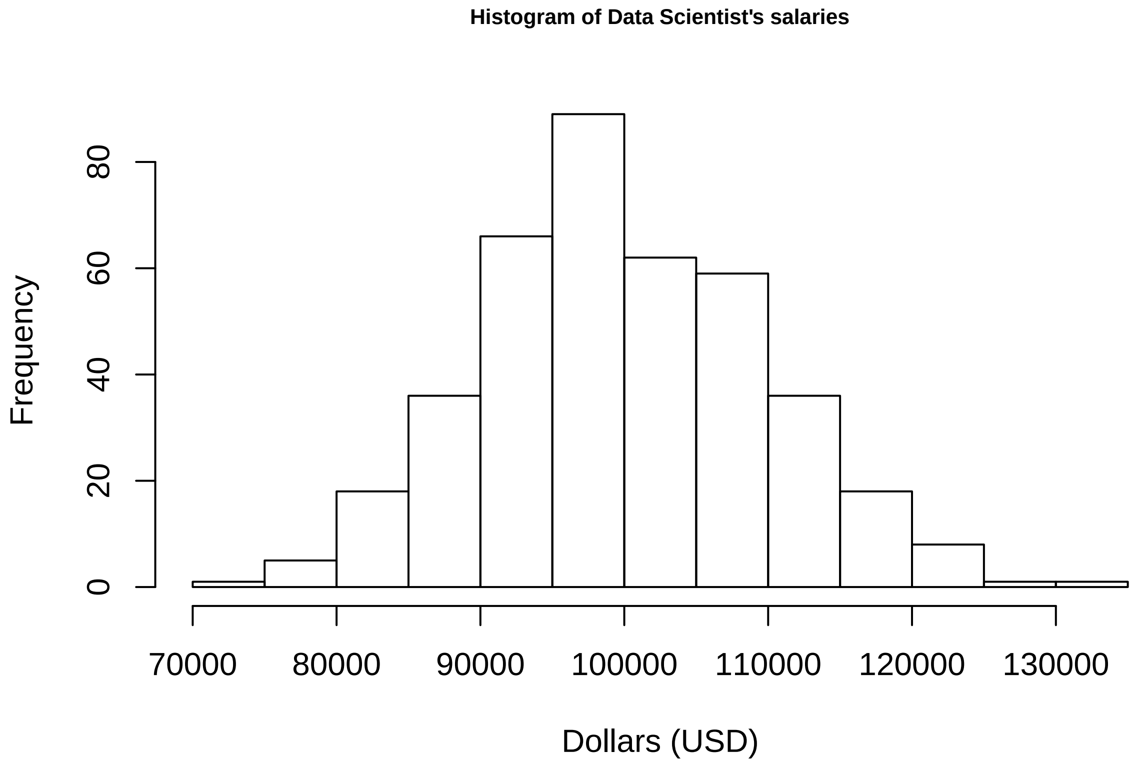
```
# set the seed for reproducibility. Note the choice of 102 was random.  you
# may want to include this when using random sampling functions to ensure
# the same results every time
set.seed(102)

# take a random sample of 400 values from a normal distribution with mean
# 100,000 and sd 10,000
DS_salary <- rnorm(n = 400, mean = 1e+05, sd = 10000)

# when we calculate the mean and sd of these values, we should get close to
# 100,000 and 10,000 respectively
mean(DS_salary)
```

```
## [1] 99949.49
```

```
sd(DS_salary)
```

```
## [1] 9887.705
```

```
# plot a histogram of the salaries:
hist(DS_salary, main = "Histogram of Data Scientist's salaries", cex.main = 0.7,
    xlab = "Dollars (USD)", breaks = 10)
```



Now, suppose that we take 100 samples of size 400 from the same normal distribution, and save the means in a vector. Now, when we inspect the mean and standard deviation of the means, we see that the mean (of the means) is still close to 100,000, but the standard deviation (of the means) is now close to $\frac{10{,}000}{\sqrt{400}} = 500$

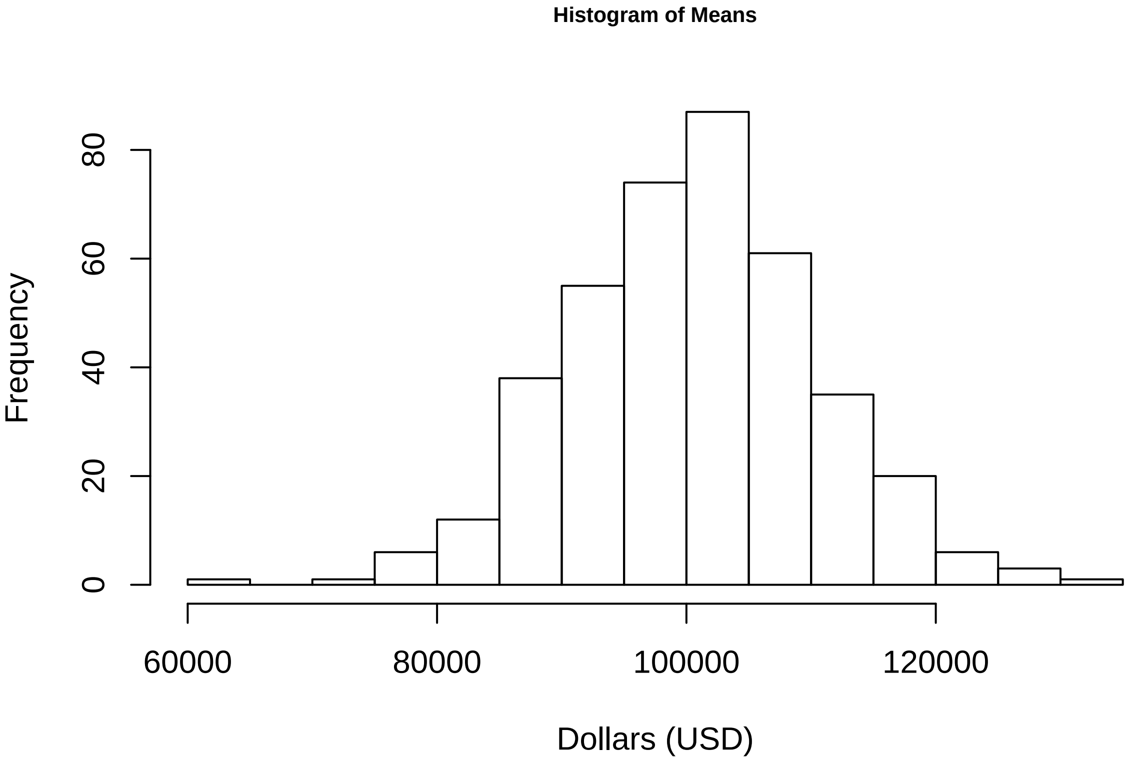
```
means <- vector()

for(i in 1:100){
  DS_salary <- rnorm(n = 400, mean = 100000, sd = 10000)
  means[i] <- mean(DS_salary)
}

mean(means)
```

```
## [1] 100054.1
```

```
sd(means)
```

```
## [1] 478.3295
```

```
#plot a histogram of the means:
hist(DS_salary,
     main = "Histogram of Means",
     cex.main = 0.7,
     xlab = "Dollars (USD)",
     breaks = 10)
```



Confidence Intervals (an example)

```
#save the mean, standard deviation, and sample size
mean_ <- 100000
sd_ <- 10000
n <- 400
```

```
#note: qnorm gives quantiles for a standard normal distribition
#for example, qnorm(0.975) gives the 97.5th percentile value for a standard normal distribution
#(standard normal means mean=0 and sd=1)
error <- qnorm(0.975)*sd_/sqrt(n)
mean_ - error # lower bound of confidence interval
```

```
## [1] 99020.02
```

```
mean_ + error # upper bound of confidence interval
```

```
## [1] 100980
```

Z-Tests, T-Test, and P-Values

Calculating P-Values in R

Consider the example from the math review packet:

$H_0 : \mu_{school} = \mu_{population}$ $H_a : \mu_{school} \neq \mu_{population}$

```
# Find proportion of sample means that are 2 points greater or less than the
# state average

# 2 ways to calculate (lower.tail=FALSE forces the function to calculate the
# upper tail directly)
1 - pnorm(q = 72, mean = 70, sd = 1)
```

```
## [1] 0.02275013
```

```
pnorm(q = 72, mean = 70, sd = 1, lower.tail = FALSE)
```

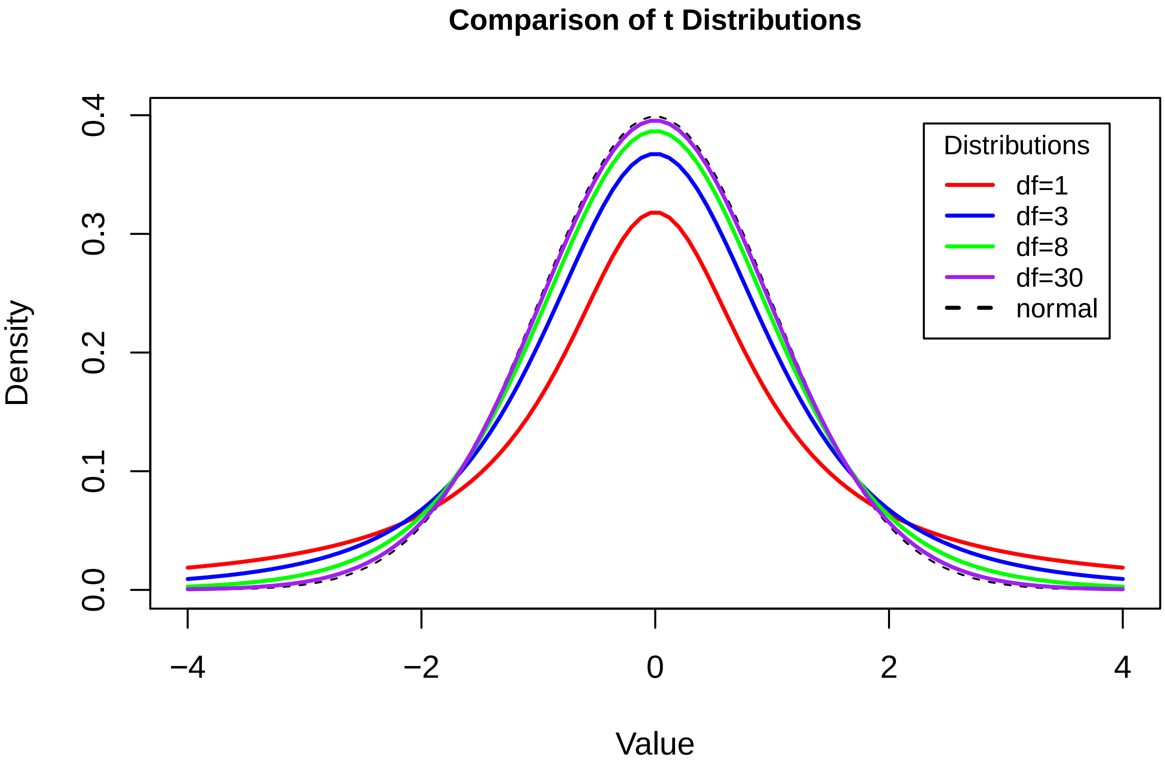
```
## [1] 0.02275013
```

The t-distribution

Note: the t distribution is a probability distribution with a single parameter, degrees of freedom (df). It is similar to a standard normal distribution, but has heavier tails when df$<$ 30. When df$>$ 30, the t distribution is essentially identical to a standard normal distribution. In the example below we plot t-distributions with difference degrees of freedom (df = 1, 3,8, 30) and compare it to normal distribution.

```
x <- seq(-4, 4, length=100) #creates a vector of 100 equally spaced values between -4 and 4
zdist <- dnorm(x) #calculates height of a standard normal probability distribution at each value in x

degf <- c(1, 3, 8, 30) #save a vector of degrees of freedom for each t distribution
colors <- c("red", "blue", "green", "purple", "black") #save a vector of colors for the plots
labels <- c("df=1", "df=3", "df=8", "df=30", "normal") #save a vector of labels (used in legend)

plot(x=x, #give x coordinates to plot
     y=zdist, #give corresponding y coordinates
     type="l", #specify that we want a line connecting the dots
     lty=2, #make that line a little thicker (default is 1)
     xlab="Value", #give an x-axis label
     ylab="Density",  #give a y-axis label
     main="Comparison of t Distributions", #title the plot
     cex.main=0.9) #make the plot title a little smaller (default is 1)

for (i in 1:4){
  #lines() function adds lines to the plot instead of creating new plots
  lines(x=x,
        y=dt(x=x,df=degf[i]), #now use dt() to get height of a t distribution with specified df
        lwd=2,
        col=colors[i])
}

legend(x="topright",  #location for legend
       inset=.05, #add a slight inset for the legend location
       title="Distributions", #title the legend
       legend=labels, #specify text for the legend
```

```
          lwd=2, #change the size of the lines in the legend to be thicker (default is 1)
          lty=c(1, 1, 1, 1, 2), #change the line type of the lines in the legend (default is 1; dotted is
          col=colors, #specify colors of the lines in the legend
          cex= 0.8) #make the legend a little smaller overall (default is 1)
```



Correlation and Covariance

We can use the 'cov' and 'cor' functions to find the covariance and correlation of two variables, respectively.

```r
# Create some fake data
set.seed(102)
x <- rnorm(100)
y <- rnorm(100)
z <- x+rnorm(100,0,.4)
w <- -x+rnorm(100,0,.4)

# Calculate covariances
cov(x,z) # Positive covariance
```

```
## [1] 1.162742
```

```r
cov(x,y) # No covariance
```

```
## [1] 0.1178042
```

```r
cov(x,w) # Negative covariance
```

```
## [1] -1.158946
```

```r
# Calculate correlations
cor(x,z) # Positive correlation
```

```
## [1] 0.9508135
```

```r
cor(x,y) # No covariance
```

```
## [1] 0.1035828
```

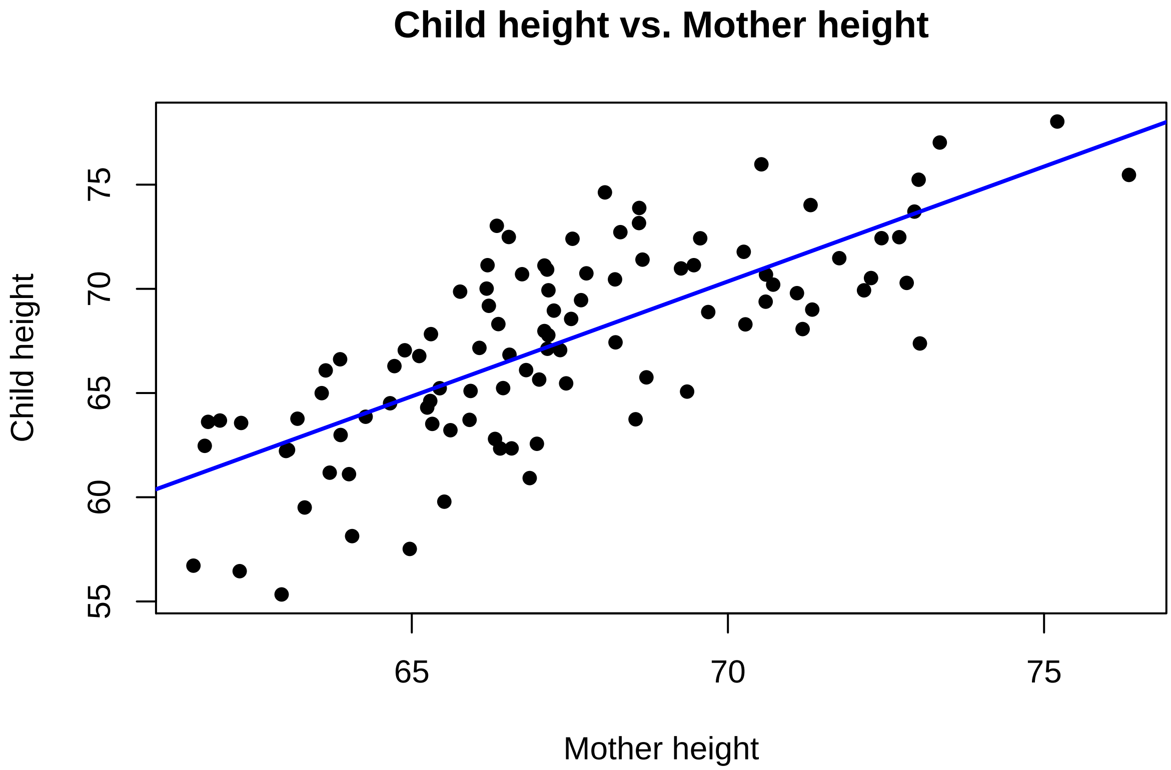
```r
cor(x,w) # Negative covariance
```

```
## [1] -0.9473492
```

Simple Ordinary Least Squares Regression

Note: You don't need to understand how to do this yet (necessarily); however, we wanted to show you how the example from the review packet was created.

```
# Create some fake data
set.seed(102)
child_height <- rnorm(100, 67, 3)
mother_height <- child_height + rnorm(100, 0, 3)

# We can use lm() to fit a linear model for example lm(y~x) and store the
# fit
fit1 <- lm(mother_height ~ child_height)

plot(child_height, mother_height, pch = 16, main = "Child height vs. Mother height",
    xlab = "Mother height", ylab = "Child height")
abline(fit1, col = 4, lwd = 2)
```



```
# The summary() function produces the summary results of fitted models
summary(fit1)
```

```
##
## Call:
## lm(formula = mother_height ~ child_height)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.288 -2.161 -0.091  2.248  6.700
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -6.83595    6.71822  -1.018    0.311
## child_height  1.10276    0.09967  11.064   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.186 on 98 degrees of freedom
## Multiple R-squared:  0.5554, Adjusted R-squared:  0.5508
## F-statistic: 122.4 on 1 and 98 DF,  p-value: < 2.2e-16
```