Lab 1: R Basics

Name: Rufus Petrie

```
## For reproducibility --- don't change this!
set.seed(08312021)
```

This week's agenda: manipulating data objects; using built-in functions, doing numerical calculations, and basic plots; reinforcing core probabilistic ideas.

The binomial distribution

The binomial distribution Bin(m, p) is defined by the number of successes in m independent trials, each have probability p of success. Think of flipping a coin m times, where the coin is weighted to have probability p of landing on heads.

The R function `rbinom()` generates random variables with a binomial distribution. E.g.,

```
rbinom(n=20, size=10, prob=0.5)
```

produces 20 observations from Bin$(10, 0.5)$.

Q1. Some simple manipulations

- **1a.** Generate 500 random values from the Bin$(15, 0.5)$ distribution, and store them in a vector called `bin.draws.0.5`. Extract and display the first 25 elements. Extract and display all but the first 475 elements.

```
bin.draws.0.5 <- rbinom(n=500, 15, 0.5)
bin.draws.0.5[1:25]
```

```
##  [1]  8  8  8  4  9 11  5 13 10  8  4  7 13  6  7  6  6 10 12  8  9  4  9  8  9
```

```
bin.draws.0.5[476:500]
```

```
##  [1]  7 10  9  9  7  7  7  7  8  7  7  6  8 10  6  6  6  5  6  9 10  8  7 10  5
```

- **1b.** Add the first element of `bin.draws.0.5` to the fifth. Compare the second element to the tenth, which is larger? A bit more tricky: print the indices of the elements of `bin.draws.0.5` that are equal to 3. How many such elements are there? Theoretically, how many such elements would you expect there to be? Hint: it would be helpful to look at the help file for the `rbinom()` function.

```
bin.draws.0.5[1] + bin.draws.0.5[5]
```

```
## [1] 17
```

```
bin.draws.0.5[2] > bin.draws.0.5[10]
```

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

```
which(bin.draws.0.5 == 3)
```

```
## [1] 130 166 223 228 263 285 403
```

```
length(which(bin.draws.0.5 == 3))
```

```
## [1] 7
```

```
500 * 0.5^15 * choose(15, 3)
```

```
## [1] 6.942749
```

The first plus the fifth element equals 17. The second element is less than the tenth. There are 7 elements with value 3. This is roughly the number we would expect, which is 6.94.

- **1c.** Find the mean and standard deviation of `bin.draws.0.5`. Is the mean close what you'd expect? The standard deviation?

```
mean(bin.draws.0.5)
```

```
## [1] 7.546
```

```
sd(bin.draws.0.5)
```

```
## [1] 2.023877
```

We expect the mean to equal $np = 7.5$ and the standard deviation to equal $\sqrt{np(1-p)} = 1.93$, so these values are reasonable.

- **1d.** Call `summary()` on `bin.draws.0.5` and describe the result.

```
summary(bin.draws.0.5)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   3.000   6.000   8.000   7.546   9.000  14.000
```

As already mentioned, the mean is 7.448 which is fairly reasonable. The median is fairly close to the mean and the quartiles are symmetric about the mean, which we would expect from a binomial distribution. The maximum is 14, which is fairly unlikely given the sample size, but this is the only odd thing.

- **1e.** Find the data type of the elements in `bin.draws.0.5` using `typeof()`. Then convert `bin.draws.0.5` to a vector of characters, storing the result as `bin.draws.0.5.char`, and use `typeof()` again to verify that you've done the conversion correctly. Call `summary()` on `bin.draws.0.5.char`. Is the result formatted differently from what you saw above? Why?

```
typeof(bin.draws.0.5)
```

```
## [1] "integer"
```

```
bin.draws.0.5.char <- as.character(bin.draws.0.5)
typeof(bin.draws.0.5.char)
```

```
## [1] "character"
```

```
summary(bin.draws.0.5.char)
```

```
##    Length     Class      Mode
##       500 character character
```

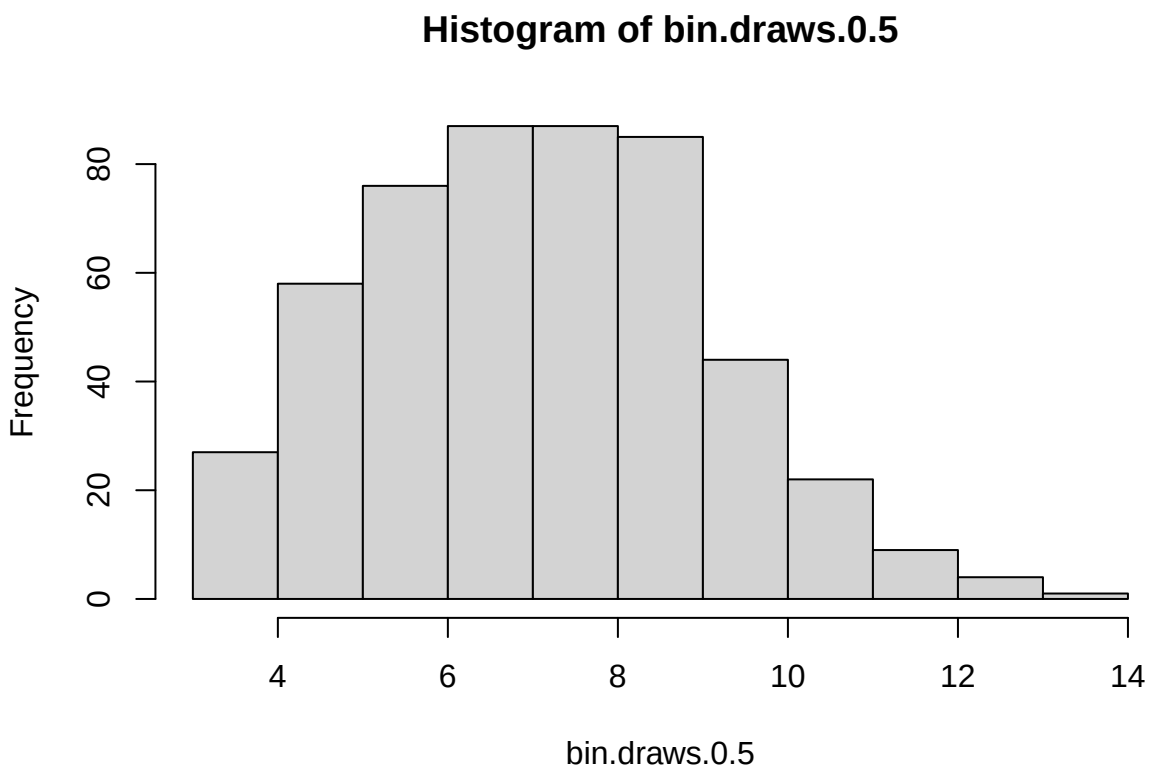
The summary for the character array is formatted differently because the vector elements are no longer numbers, so we can't use numeric measurements to summarize them.

Q2. Some simple plots

- **2a.** The function `plot()` is a generic function in R for the visual display of data. The function `hist()` specifically produces a histogram display. Use `hist()` to produce a histogram of your random draws

from the binomial distribution, stored in `bin.draws.0.5`.

```
hist(bin.draws.0.5)
```



- **2b.** Call `tabulate()` on `bin.draws.0.5`. What is being shown? Does it roughly match the histogram you produced in the last question?

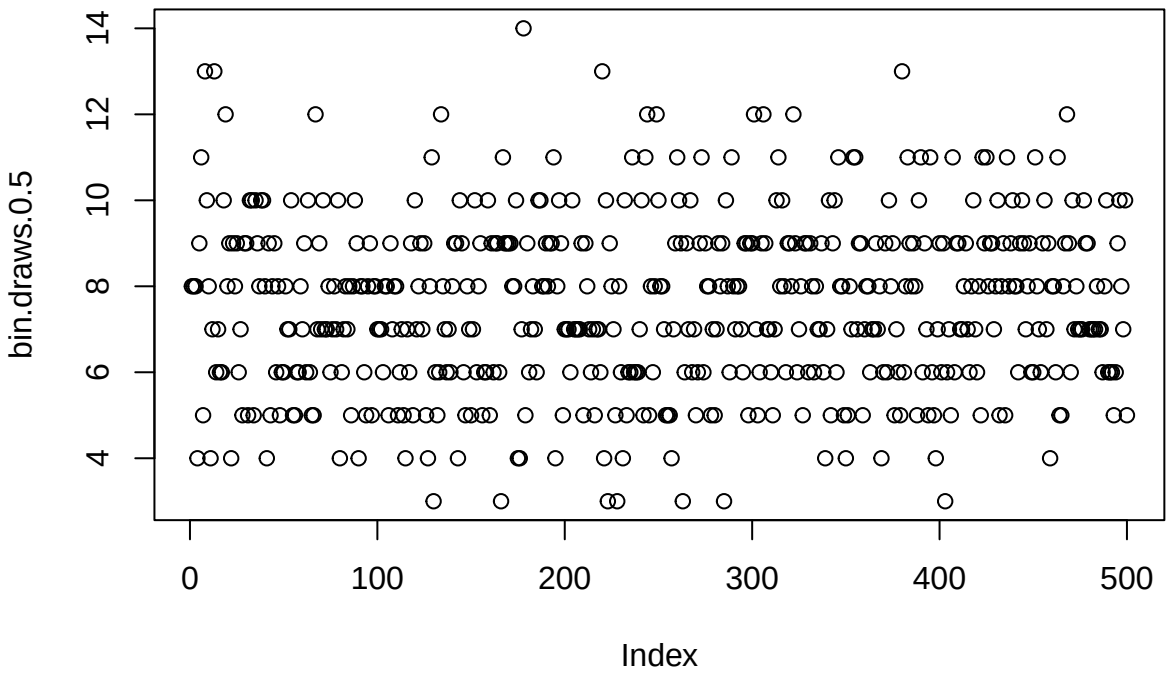
```
tabulate(bin.draws.0.5)
```

```
##  [1]  0  0  7 20 58 76 87 87 85 44 22  9  4  1
```

The tabulation shows that count of each observed value starting at 1. It matches what we see in the histogram above.

- **2c.** Call `plot()` on `bin.draws.0.5` to display your random values from the binomial distribution. Can you interpret what the `plot()` function is doing here?

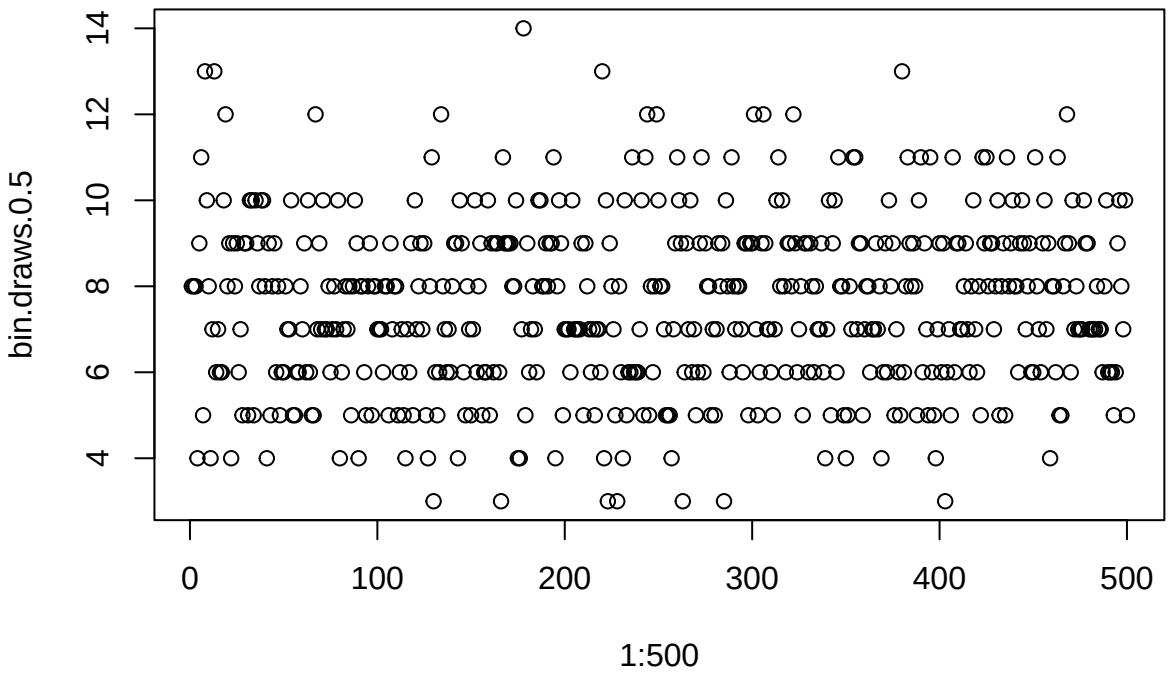
```
plot(bin.draws.0.5)
```



The plot function plots the observed values on the y-axis and the indexes on the x-axis.

- **2d.** Call `plot()` with two arguments, the first being `1:500`, and the second being `bin.draws.0.5`. This creates a scatterplot of `bin.draws.0.5` (on the y-axis) versus the indices 1 through 500 (on the x-axis). Does this match your plot from the last question?

```
plot(1:500, bin.draws.0.5)
```



This matches the plot from the last question.

Q3. More binomials, more plots

- **3a.** Generate 500 binomials again, composed of 15 trials each, but change the probability of success to: 0.2, 0.3, 0.4, 0.6, 0.7, and 0.8, storing the results in vectors called `bin.draws.0.2`, `bin.draws.0.3`, `bin.draws.0.4.`, `bin.draws.0.6`, `bin.draws.0.7` and `bin.draws.0.8`. For each, compute the mean and standard deviation.

```
bin.draws.0.2 <- rbinom(n = 500, 15, 0.2)
mean(bin.draws.0.2)
```

```
## [1] 2.886
```

```
sd(bin.draws.0.2)
```

```
## [1] 1.493139
```

```
bin.draws.0.3 <- rbinom(n = 500, 15, 0.3)
mean(bin.draws.0.3)
```

```
## [1] 4.6
```

```
sd(bin.draws.0.3)
```

```
## [1] 1.725675
```

```
bin.draws.0.4 <- rbinom(n = 500, 15, 0.4)
mean(bin.draws.0.4)
```

```
## [1] 6.102
```

```
sd(bin.draws.0.4)
```

```
## [1] 1.902323
```

```
bin.draws.0.6 <- rbinom(n = 500, 15, 0.6)
mean(bin.draws.0.6)
```

```
## [1] 9.018
```

```
sd(bin.draws.0.6)
```

```
## [1] 1.941446
```

```
bin.draws.0.7 <- rbinom(n = 500, 15, 0.7)
mean(bin.draws.0.7)
```

```
## [1] 10.502
```

```
sd(bin.draws.0.7)
```

```
## [1] 1.715775
```

```
bin.draws.0.8 <- rbinom(n = 500, 15, 0.8)
mean(bin.draws.0.8)
```

```
## [1] 12.024
```

```
sd(bin.draws.0.8)
```

```
## [1] 1.528429
```

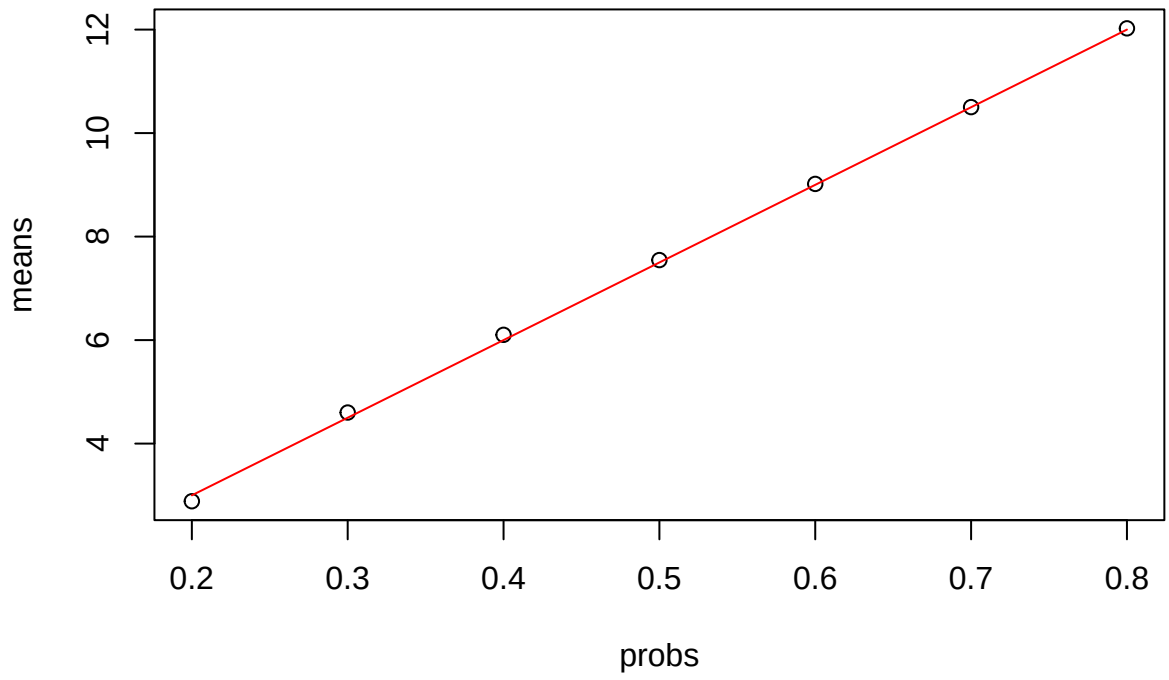
- **3b.** We'd like to compare the properties of our vectors. Create a vector of length 7, whose entries are the means of the 7 vectors we've created, in order according to the success probabilities of their underlying binomial distributions (0.2 through 0.8).

```
means <- c(mean(bin.draws.0.2), mean(bin.draws.0.3), mean(bin.draws.0.4), mean(bin.draws.0.5), mean(bin
sds <- c(sd(bin.draws.0.2), sd(bin.draws.0.3), sd(bin.draws.0.4), sd(bin.draws.0.5), sd(bin.draws.0.6),
```

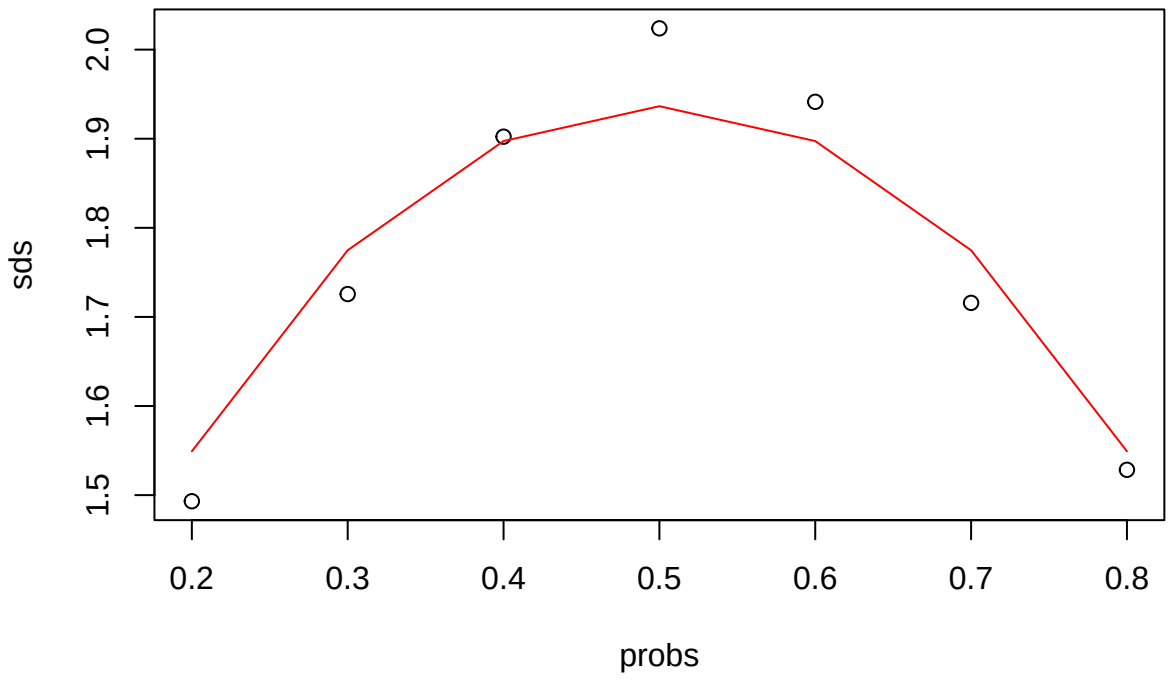
- **3c.** Using the vectors from the last part, create the following scatterplots. Explain in words, for each, what's going on.
  - The 7 means versus the 7 probabilities used to generate the draws.
  - The standard deviations versus the probabilities.
  - The standard deviations versus the means.

  Challenge: for each plot, add a curve that corresponds to the relationships you'd expect to see in the theoretical population (i.e., with an infinite amount of draws, rather than just 500 draws).

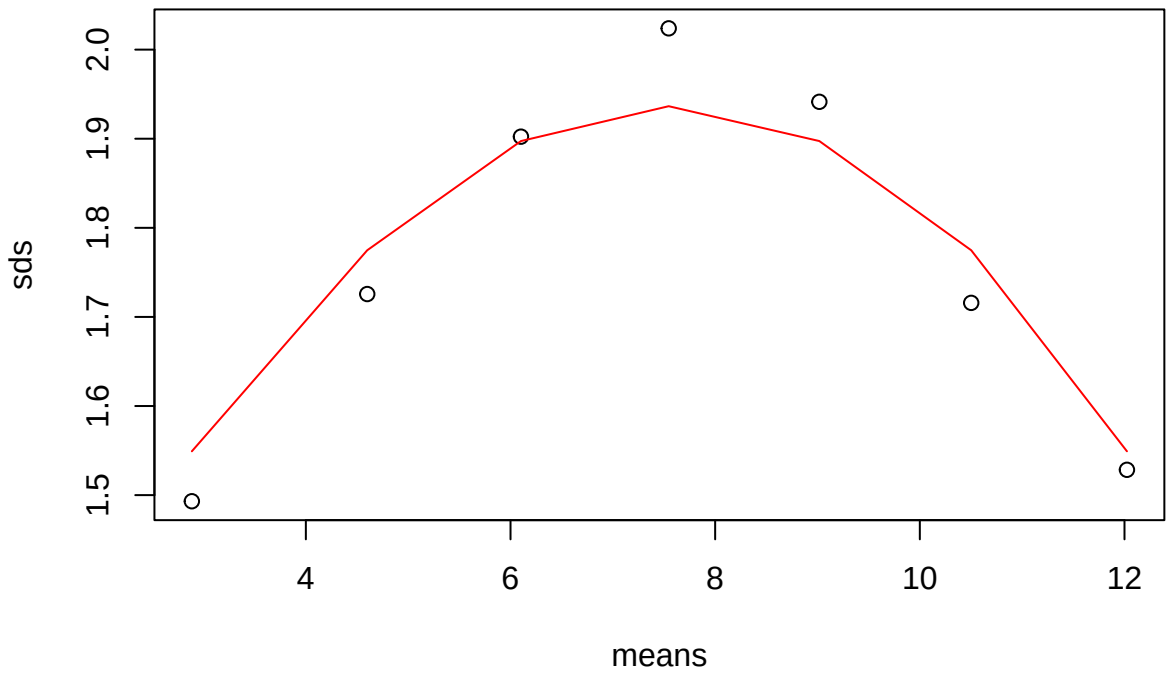
```
probs <- 0.1 * 2:8
mean_line <- probs * 15
sd_line <- sqrt(15 * probs * (1 - probs))
plot(probs, means)
lines(probs, mean_line, col="red")
```




```
plot(probs, sds)
lines(probs, sd_line, col="red")
```




```
plot(means, sds)
lines(means, sd_line, col="red")
```



As the probabilities increase, the mean linearly increases. The standard deviations increase with respect to the probabilities until 0.5, at which point they start to decrease. The standard deviations have the same relationship with the means as the means are a linear multiple of the probabilities.

Q4. Working with matrices

- **4a.** Create a matrix of dimension 500 x 7, called `bin.matrix`, whose columns contain the 7 vectors we've created, in order of the success probabilities of their underlying binomial distributions (0.2 through 0.8). Hint: use `cbind()`.

```
bin.matrix <- cbind(bin.draws.0.2, bin.draws.0.3, bin.draws.0.4, bin.draws.0.5, bin.draws.0.6, bin.draws
colnames(bin.matrix) <- probs
```

- **4b.** Print the first five rows of `bin.matrix`. Print the element in the 66th row and 5th column. Compute the largest element in first column. Compute the largest element in all but the first column.

```
bin.matrix[1:5,]
```

```
##      0.2 0.3 0.4 0.5 0.6 0.7 0.8
## [1,]   1   4   8   8  14   9  12
## [2,]   4   2   6   8   9  12  12
## [3,]   1   7   6   8  10  13  10
## [4,]   1   5   6   4   9  11  12
## [5,]   4   3   7   9  10  13  12
```

```
bin.matrix[66,5]
```

```
## 0.6
```

```
##   8
```

```
max(bin.matrix[,1])
```

```
## [1] 7
```

```
max(bin.matrix[,-1])
```

```
## [1] 15
```

- **4c.** Calculate the column means of `bin.matrix` by using just a single function call.

```
colMeans(bin.matrix)
```

```
##    0.2    0.3    0.4    0.5    0.6    0.7    0.8
##  2.886  4.600  6.102  7.546  9.018 10.502 12.024
```

- **4d.** Compare the means you computed in the last question to those you computed in Q3b, in two ways. First, using `==`, and second, using `identical()`. What do the two ways report? Are the results compatible? Explain.

```
colMeans(bin.matrix) == means
```

```
##  0.2  0.3  0.4  0.5  0.6  0.7  0.8
## TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```
identical(colMeans(bin.matrix), means)
```

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

Using the equality operator yields true, but the identical() function yields false. This is likely because the means are slightly different and the equality operator has some amount of error tolerance.

- **4e.** Take the transpose of `bin.matrix` and then take row means. Are these the same as what you just computed? Should they be?

```
rowMeans(t(bin.matrix))
```

```
##    0.2    0.3    0.4    0.5    0.6    0.7    0.8
##  2.886  4.600  6.102  7.546  9.018 10.502 12.024
```

These are the same as the regular means. They should be because transposing the matrix turns the columns into rows, so the row means of the second matrix will equal the column means of the first.

Q5. Warm up is over, let's go big

- **5a.** R's capacity for data storage and computation is very large compared to what was available 10 years ago. Generate 5 million numbers from $\text{Bin}(1 \times 10^6, 0.5)$ distribution and store them in a vector called `big.bin.draws`. Calculate the mean and standard deviation of this vector.

```
big.bin.draws <- rbinom(5*10^6, 10^6, 0.5)
mean(big.bin.draws)
```

```
## [1] 499999.7
```

```
sd(big.bin.draws)
```

```
## [1] 499.8982
```

- **5b.** Create a new vector, called `big.bin.draws.standardized`, which is given by taking `big.bin.draws`, subtracting off its mean, and then dividing by its standard deviation. Calculate the mean and standard deviation of `big.bin.draws.standardized`. (These should be 0 and 1, respectively, or very close to it; if not, you've made a mistake somewhere).

```
big.bin.draws.standardized <- (big.bin.draws - mean(big.bin.draws)) / sd(big.bin.draws)
mean(big.bin.draws.standardized)
```

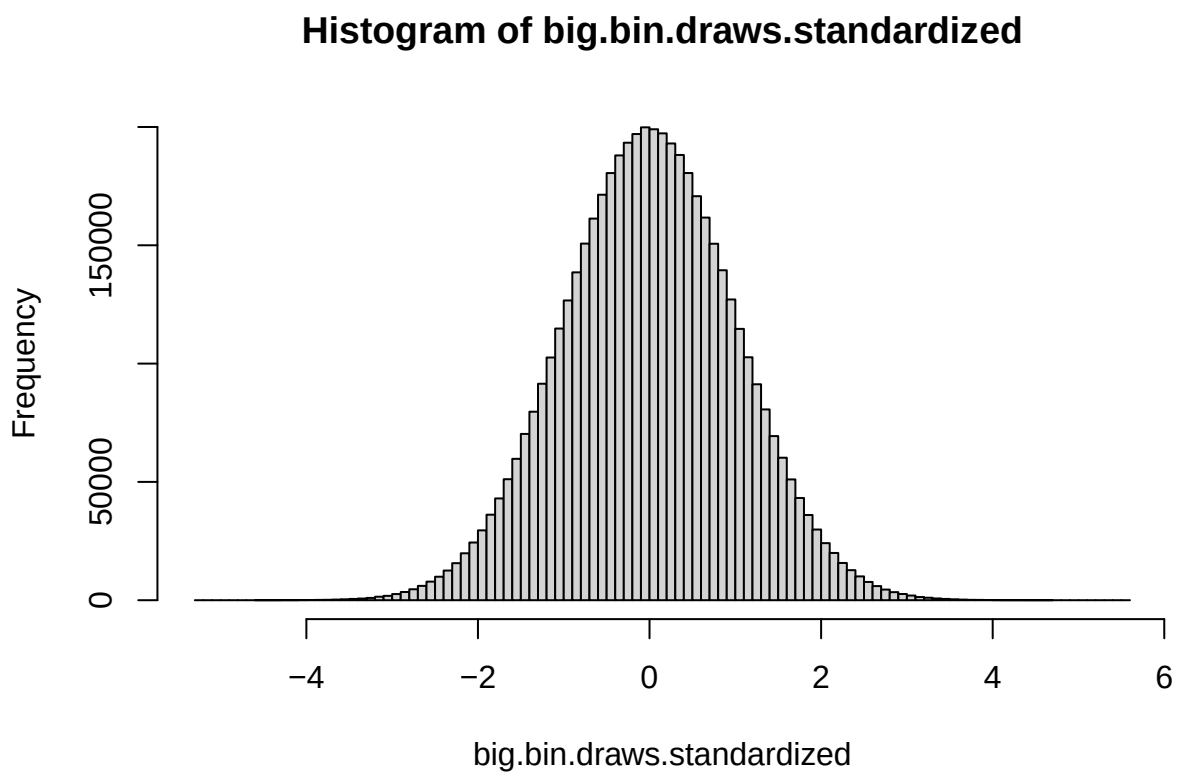
```
## [1] -8.590273e-15
```

```
sd(big.bin.draws.standardized)
```

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

- **5c.** Plot a histogram of `big.bin.draws.standardized`. To increase the number of histogram bars, set the `breaks` argument in the `hist()` function (e.g., set `breaks=100`). What does the shape of this histogram appear to be? Is this surprising? What could explain this phenomenon? Hint: rhymes with "Mental Gimmick Serum" ...

```
hist(big.bin.draws.standardized, breaks = 100)
```



This histogram is shaped like a bell curve. This is not surprising because the Central Limit Theorem says that with a sufficiently large sample size, the mean of the samples from any distribution will be normally distributed.

- **5d.** Calculate the proportion of times that an element of `big.bin.draws.standardized` exceeds 1.644854. Is this close to 0.05?

```
length(which(big.bin.draws.standardized > 1.644854))/(5*10^6)
```

```
## [1] 0.0500838
```

The number is very close to 0.05, which we would expect because 1.64 is the one tail 5% cutoff for the normal distribution.

Q6. Now let's go really big

- **6a.** Let's push R's computational engine a little harder. Generate 200 million numbers from Bin($10 \times 10^6, 50 \times 10^{-8}$), and save it in a vector called `huge.bin.draws`.

```
t1 <- Sys.time()
huge.bin.draws <- rbinom(200*10^6, 10*10^6, 50*10^-8)
t2 <- Sys.time()
t2 - t1
```

```
## Time difference of 11.42188 secs
```

- **6b.** Calculate the mean and standard deviation of `huge.bin.draws`. Are they close to what you'd expect? (They should be very close.) Did it longer to compute these, or to generate `huge.bin.draws` in the first place?

```
t1 <- Sys.time()
mean(huge.bin.draws)
```

```
## [1] 4.999914
```

```
(10*10^6) * (50*10^-8)
```

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

```
sd(huge.bin.draws)
```

```
## [1] 2.236135
```

```
sqrt(10*10^6 * (50*10^-8) * (1-50*10^-8))
```

```
## [1] 2.236067
```

```
t2 <- Sys.time()
t2 - t1
```

```
## Time difference of 1.043294 secs
```

The results are very close to what we would expect. It took much longer to generate the data than it took to calculate these statistics.

- **6c.** Calculate the median of `huge.bin.draws`. Did this median calculation take longer than the calculating the mean? Is this surprising?

```
t1 <- Sys.time()
median(huge.bin.draws)
```

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

```
t2 <- Sys.time()
t2 - t1
```

```
## Time difference of 2.399957 secs
```

This calculation took longer than the mean calculation. This is not surprising because it takes linear time to calculate the mean, and unless R uses quickselect it probably takes $nlog(n)$ time to find the median.

- **6d.** Calculate the exponential of the median of the logs of `huge.bin.draws`, in one line of code. Did this take longer than the median calculation applied to `huge.bin.draws` directly? Is this surprising?

```
t1 <- Sys.time()
exp(median(log(huge.bin.draws)))
```

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

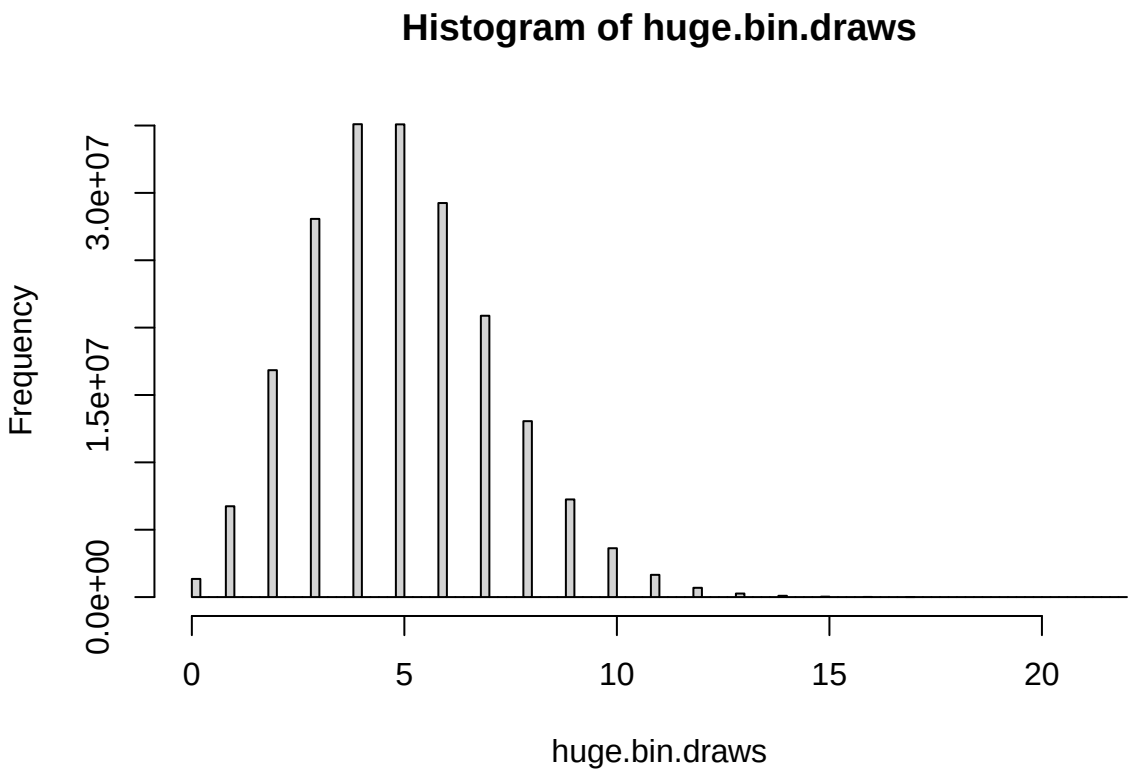
```
t2 <- Sys.time()
t2 - t1
```

```
## Time difference of 7.193448 secs
```

This took much longer than the direct calculation. This isn't surprising because R has to take the log of 200 million numbers before finding the median and converting back to the original scale.

- **6e.** Plot a histogram of `huge.bin.draws`, again with a large setting of the `breaks` argument (e.g., `breaks=100`). Describe what you see; is this different from before, when we had 3 million draws? **Challenge**: Is this surprising? What distribution is this?

```
hist(huge.bin.draws, breaks = 100)
```



Again, the distribution of draws appears normal. This is slightly different from before because we didn't normalize the draws. Furthermore, while it's impossible to have negative successes, it's possibly to have large positive outliers. Therefore, the plot's tail extends more right than we might expect.

Q7. Going big with lists

- **7a.** Convert `big.bin.draws` into a list using `as.list()` and save the result as `big.bin.draws.list`. Check that you indeed have a list by calling `class()` on the result. Check also that your list has the right length, and that its 1159th element is equal to that of `big.bin.draws`.

```
big.bin.draws.list <- as.list(big.bin.draws)
class(big.bin.draws.list)
```

```
## [1] "list"
```

```
length(big.bin.draws.list)
```

```
## [1] 5000000
```

```
big.bin.draws.list[1159] == big.bin.draws[1159]
```

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

- **7b.** Run the code below, to standardize the binomial draws in the list `big.bin.draws.list`. Note that `lapply()` applies the function supplied in the second argument to every element of the list supplied in the first argument, and then returns a list of the function outputs. (We'll learn much more about the `apply()` family of functions later in the course.) Did this `lapply()` command take longer to evaluate than the code you wrote in Q5b? (It should have; otherwise your previous code could have been improved, so go back and improve it.) Why do you think this is the case?

```
big.bin.draws.mean = mean(big.bin.draws)
big.bin.draws.sd = sd(big.bin.draws)
standardize = function(x) {
  return((x - big.bin.draws.mean) / big.bin.draws.sd)
}
big.bin.draws.list.standardized.slow = lapply(big.bin.draws.list, standardize)
```

The lapply command took longer than the code for q5b. This is because this function performs elementwise operations instead of working with vectors, so the calculations use a lot more overhead.

- **7c.** Run the code below, which again standardizes the binomial draws in the list `big.bin.draws.list`, using `lapply()`. Why is it so much slower than the code in the last question? (You may stop evaluation if it is taking too long!) Think about what is happening each time the function is called.

```
standardize.slow = function(x) {
  return((x - mean(big.bin.draws)) / sd(big.bin.draws))
}
big.bin.draws.list.standardized.slow = lapply(big.bin.draws.list, standardize.slow)
```

This code is a lot slower than the code in the last question because each time it standardizes an observation, it recalculates the mean and standard deviation.

- **7d.** Lastly, let's look at memory usage. The command `object.size(x)` returns the number of bytes used to store the object `x` in your current R session. Find the number of bytes used to store `big.bin.draws` and `big.bin.draws.list`. How many megabytes (MB) is this, for each object? Which object requires more memory, and why do you think this is the case? Remind yourself: why are lists special compared to vectors, and is this property important for the current purpose (storing the binomial draws)?

```
object.size(big.bin.draws)
```

```
## 20000048 bytes
```

```
object.size(big.bin.draws.list)
```

```
## 320000048 bytes
```

The vector takes up 20 MB, whereas the list takes up 320 MB. The list requires more memory because it can store non-numeric data, and data structures with greater flexibility require more resources. Because we don't need to store non-numeric data in this scenario, we shouldn't use a list.