Lab 1: R Basics

Statistical Computing, 36-350

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 $\text{Bin}(m, p)$ 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)
```

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

produces 20 observations from $\text{Bin}(10, 0.5)$.

Q1. Some simple manipulations

- **1a.** Generate 500 random values from the $\text{Bin}(15, 0.5)$ 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, size = 15, prob = 0.5)
bin.draws.0.5[1:25]
```

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

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

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

- **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] 18
```

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

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

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

```
## [1] 110 146 203 208 243 265 383
```

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

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

The tenth element is greater than the second element. There are 7 elements that are equal to 3. In theory, we would expect there to be $500 * C(15, 3) * 0.5^{12} * 0.5^3 = 6.94$ equal to 3.

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

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

```
## [1] 7.5
```

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

```
## [1] 1.984408
```

The sample mean equals 7.5 which equals the expected mean of $np = 15(0.5) = 7.5$. The sample standard deviation equals 1.98 which is pretty close to the expected SD of $\sqrt{np(1-p)} = \sqrt{15 * 0.5 * 0.5} = 1.93$.

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

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

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     3.0     6.0     8.0     7.5     9.0    14.0
```

This sample has a reasonable mean as it equals the expected value. 8 makes sense for the median as it can only take integer values, so we would expect 7 or 8. The first and third quartiles are symmetric about the mean as we would expect. We did not observe many extreme values on the lower end, but we rolled a 14 which is quite unlikely.

- **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?

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

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

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

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

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

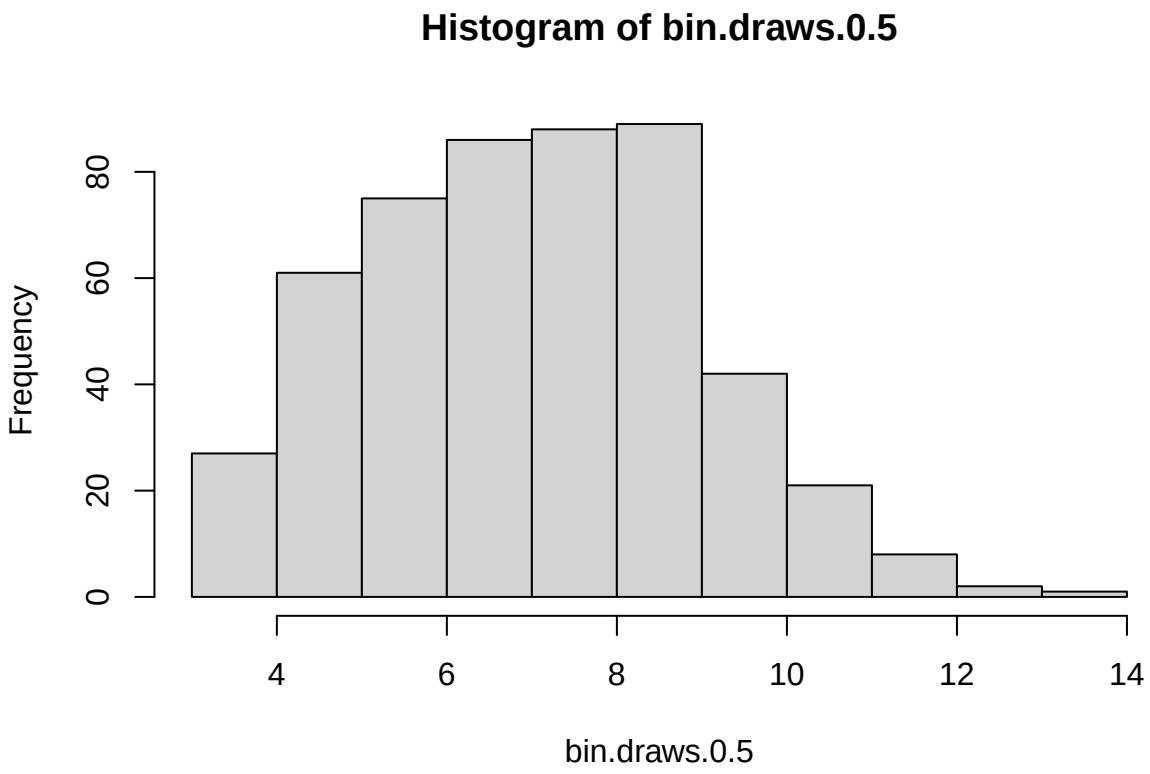
```
##    Length     Class      Mode
##       500 character character
```

The result is formatted differently because we can't compute the same summary statistics for characters that we use for integers.

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?

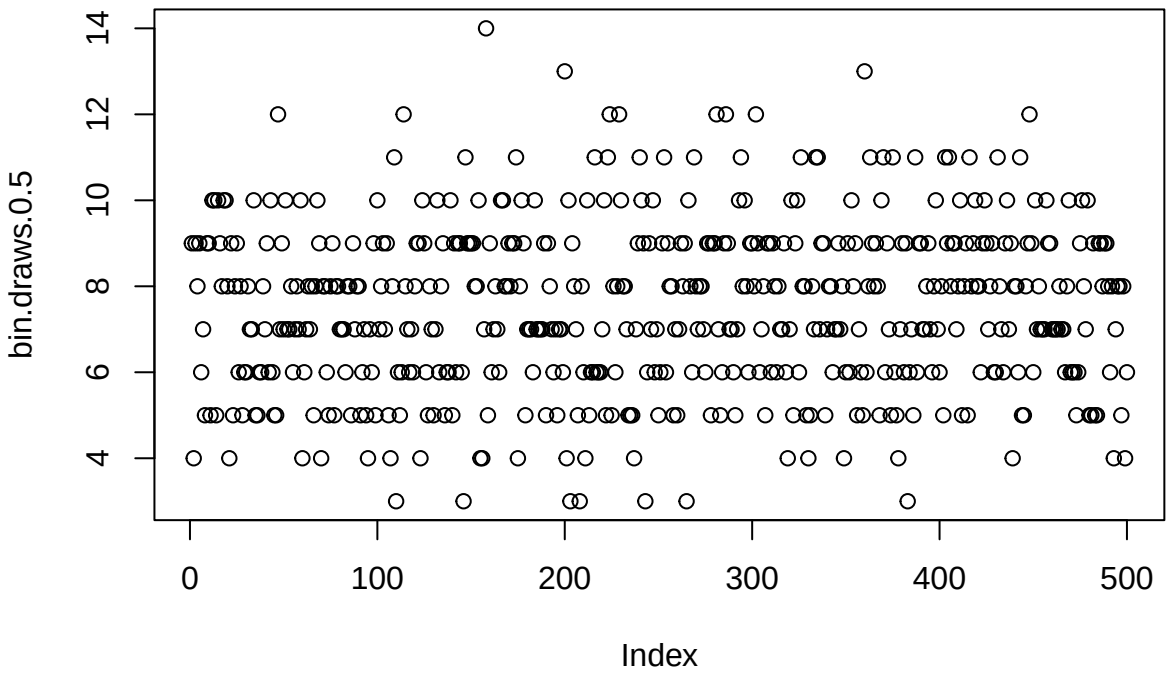
```
tabulate(bin.draws.0.5)
```

```
##  [1]  0  0  7 20 61 75 86 88 89 42 21  8  2  1
```

Tabulate shows the number of each outcome from bin.draws.0.5 (except for 15 which never happened). It matches the histogram produced in the last question.

- **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?

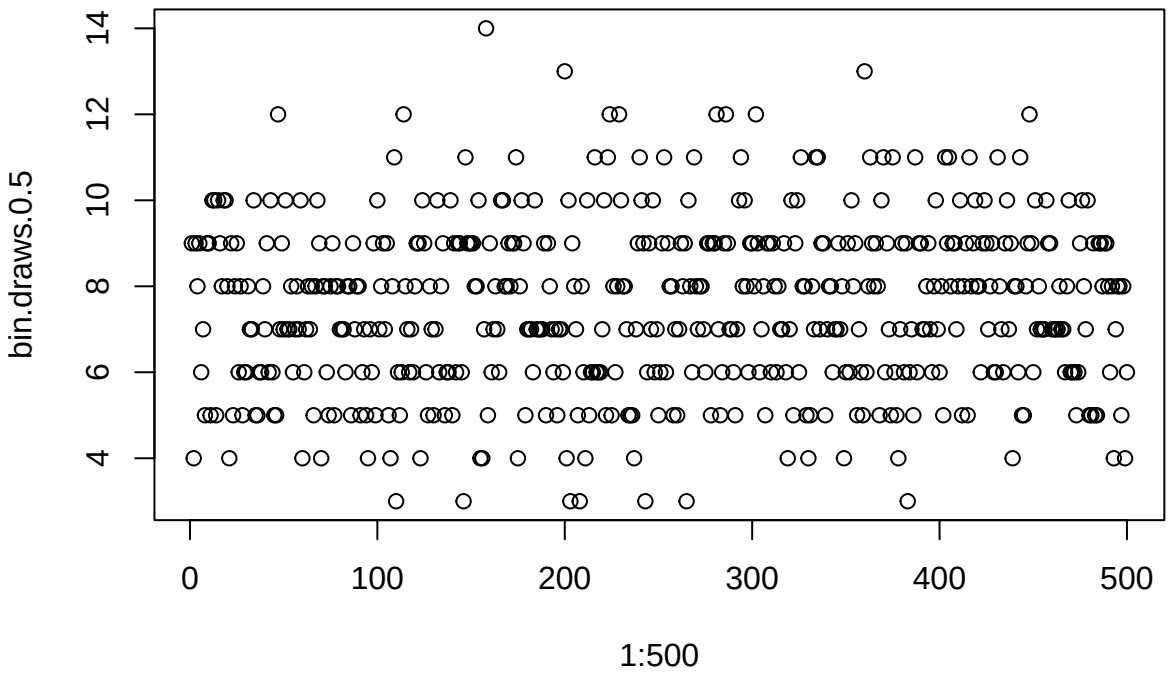
```
plot(bin.draws.0.5)
```



The plot function is plotting the outcome of bin.draws.0.5 against the index at which it occurred - this generates a chaotic picture.

- **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 that we produced in the last problem. This means that the plot() function infers the x axis from the length of the vector if we don't supply it that argument.

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, size = 15, prob = 0.2)
mean(bin.draws.0.2)
```

```
## [1] 2.904
```

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

```
## [1] 1.493736
```

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

```
## [1] 4.594
```

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

```
## [1] 1.707335
```

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

```
## [1] 6.062
```

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

```
## [1] 1.939505
```

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

```
## [1] 8.984
```

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

```
## [1] 1.944043
```

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

```
## [1] 10.494
```

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

```
## [1] 1.716933
```

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

```
## [1] 12.03
```

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

```
## [1] 1.518457
```

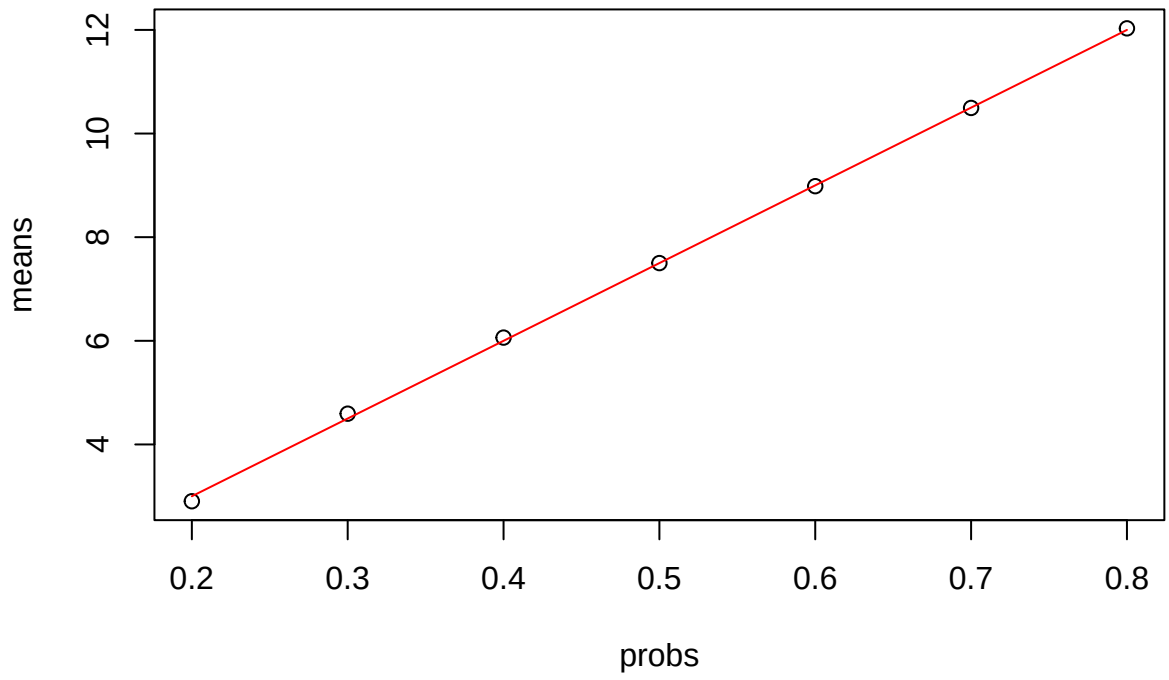
- **3b.** We'd like to compare the properties of our vectors. Create a vector of length 7, whose entries are the means of the 7 vectors we've created, in order according to the success probabilities of their underlying binomial distributions (0.2 through 0.8).

```
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),
```

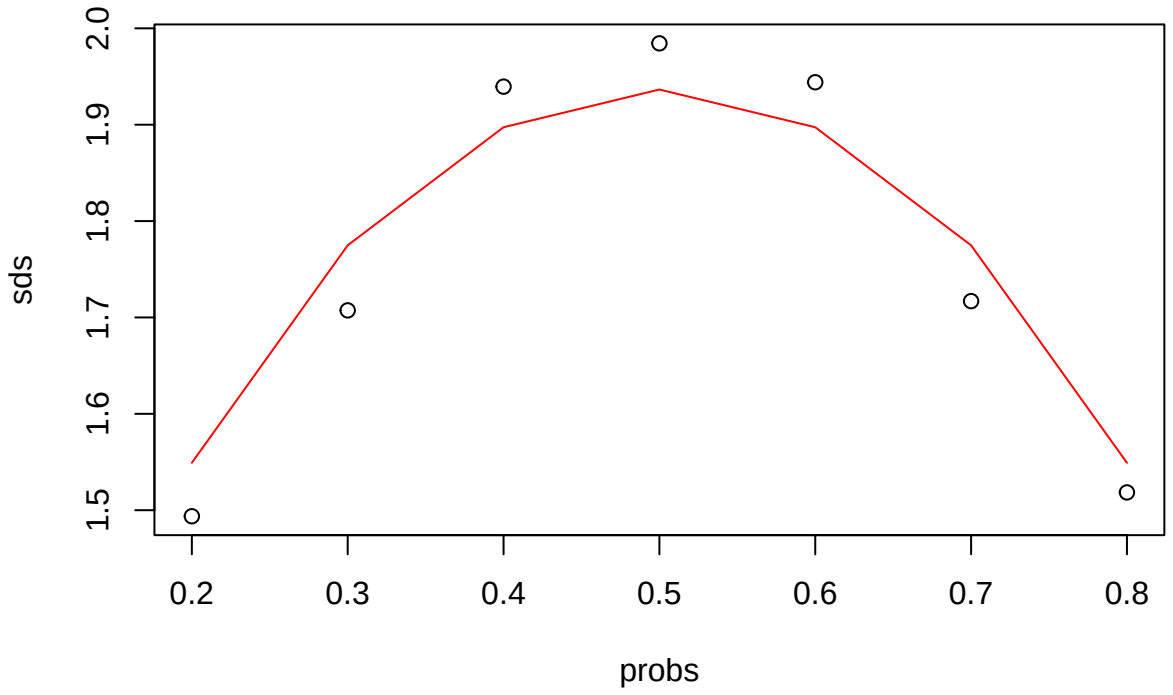
- **3c.** Using the vectors from the last part, create the following scatterplots. Explain in words, for each, what's going on.
  - The 7 means versus the 7 probabilities used to generate the draws.
  - The standard deviations versus the probabilities.
  - The standard deviations versus the means.

  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).

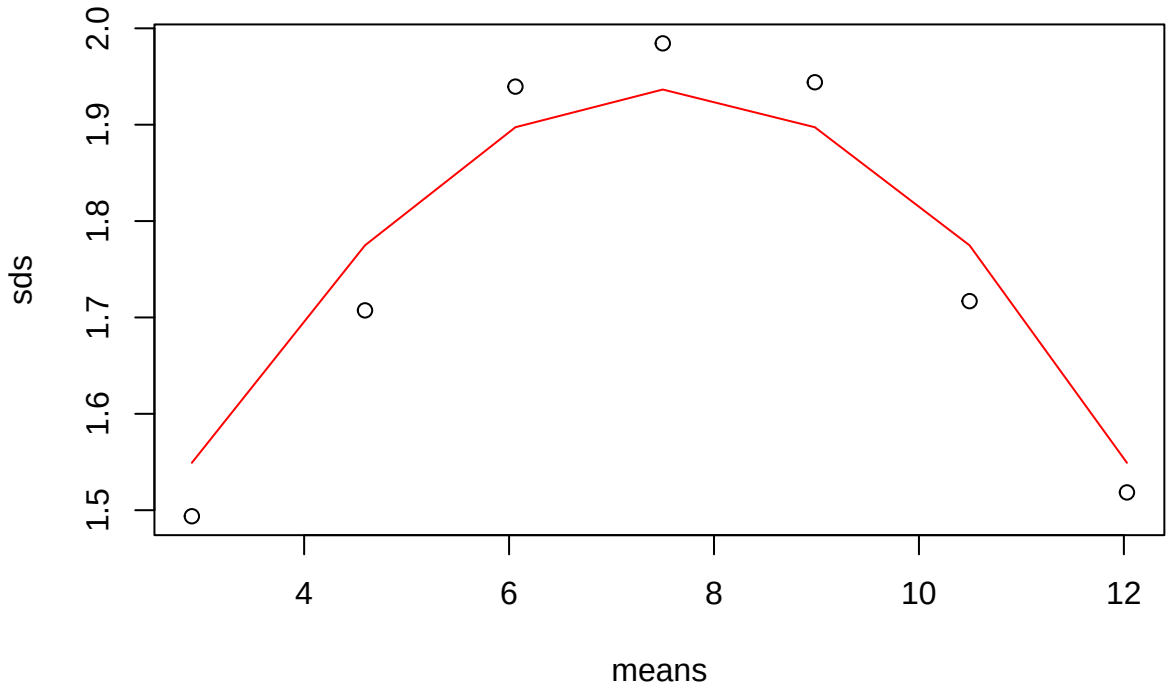
```
probs <- 0.1 * 2:8
mean_line <- 15 * probs
sd_line <- c()
for(i in 2:8){
  sd_line[i-1] = sqrt(15 * (i/10) * (1-i/10))
}
plot(probs, means)
lines(probs, mean_line, col="red")
```




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




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



In the first plot, the mean increases linearly with respect to probability. In the second plot, the standard deviation increases until the probability hits 0.5 then decreases. In the third plot, the standard deviation increases until the mean hits 7.5 (which corresponds to p = 0.5) then decreases.

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 <- bin.draws.0.2
bin.matrix <- cbind(bin.matrix, bin.draws.0.3)
bin.matrix <- cbind(bin.matrix, bin.draws.0.4)
bin.matrix <- cbind(bin.matrix, bin.draws.0.5)
bin.matrix <- cbind(bin.matrix, bin.draws.0.6)
bin.matrix <- cbind(bin.matrix, bin.draws.0.7)
bin.matrix <- cbind(bin.matrix, bin.draws.0.8)
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,]   3   3   6   9   9  12  12
## [2,]   5   5  10   4  12  12  10
## [3,]   2   4  10   9  11  12  10
```

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

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

```
## 0.6
##  11
```

```
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
##  2.904  4.594  6.062  7.500  8.984 10.494 12.030
```

- **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 indicates that the column means are equal, but using the identical() function suggests that they're different. These results are compatible because the equality operator allows a small margin for error to deal with floating points, but the identical() function likely checks for exact equality.

- **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
##  2.904  4.594  6.062  7.500  8.984 10.494 12.030
```

These are the same. They should be equal because transposition turns the rows of the matrix into columns and vice versa, so calculating the row means of the transposed matrix equates to taking the column means of the original matrix.

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] 499.8979
```

- **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).

```
t1 <- Sys.time()
big.bin.draws.standardized <- (big.bin.draws - mean(big.bin.draws)) / sd(big.bin.draws)
mean(big.bin.draws.standardized)
```

```
## [1] -1.80966e-15
```

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

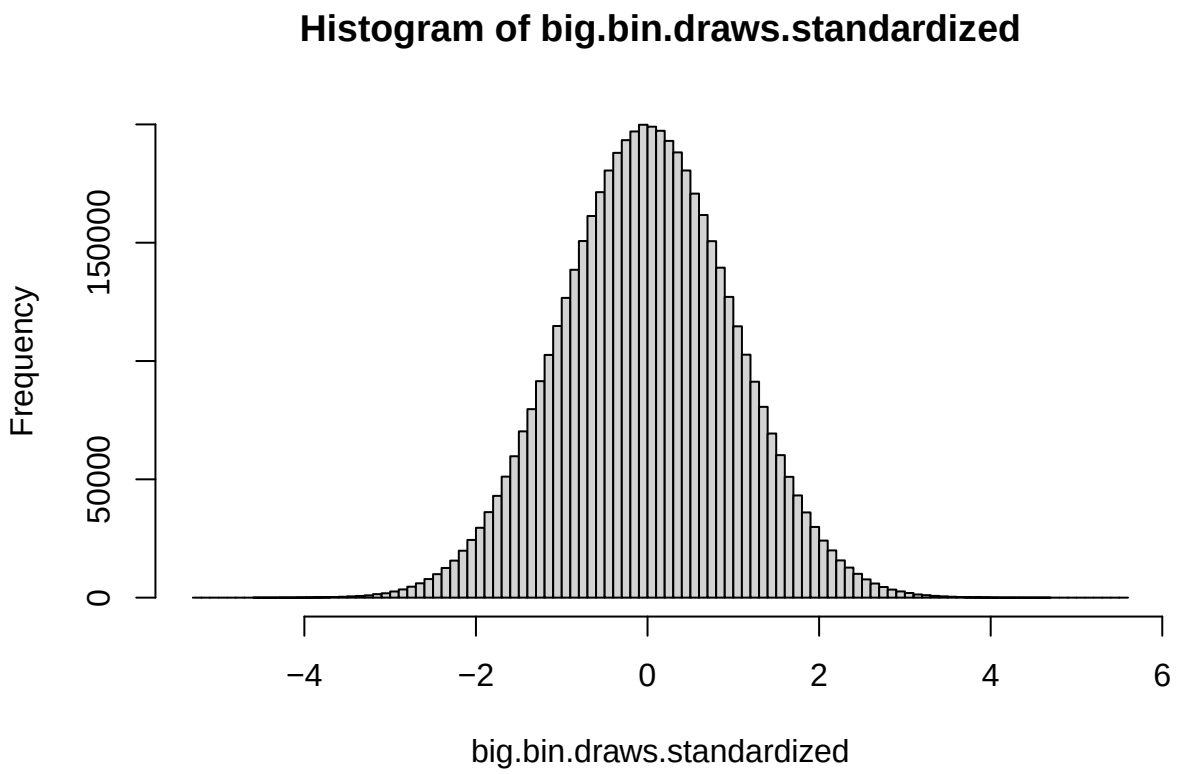
```
## [1] 1
```

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

```
## Time difference of 0.09220815 secs
```

- **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)
```



The shape of the histogram appears to be a bell curve. This isn't surprising because the Central Limit Theorem states that the means of repeated samples will form a normal distribution regardless of the distribution from which they're sampled.

- **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)) / length(big.bin.draws.standardized)
```

```
## [1] 0.0500836
```

The proportion equals 0.0500836, which is pretty close to 0.05.

- **5e.** Either by simulation, or via a built-in R function, compute the probability that a standard normal random variable exceeds 1.644854. Is this close to 0.05? Hint: for either approach, it would be helpful to look at the help file for the `rnorm()` function.

```
rn <- rnorm(10^6)
sum(rn>1.644854)/(10^6)
```

```
## [1] 0.050164
```

From a simulation using a million samples, we observe the probability of a standard normal variable exceeding 1.644854 to be 0.050175, which is pretty close to 0.05.

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(2*10^8, 10^7, 0.5*10^(-6))
t2 <- Sys.time()
t2 - t1
```

```
## Time difference of 11.20674 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] 4.999881
```

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

```
## [1] 2.236118
```

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

```
## Time difference of 1.147196 secs
```

We expect a mean of $np = 10^7 * 0.5 * 10^{-6} = 0.5$ and SD of $\sqrt{10^7 * (0.5 * 10^{-6} * (1 - 0.5 * 10^{-6}))} = 2.23$, so these sample statistics are close to what we would expect. These took much less time to calculate than huge.bin.draws.

- **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 2.477258 secs
```

This calculation took longer than calculating the mean. This is not surprising because the mean is calculated in linear time whereas the median is calculated in $nlog(n)$ time.

- **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
```

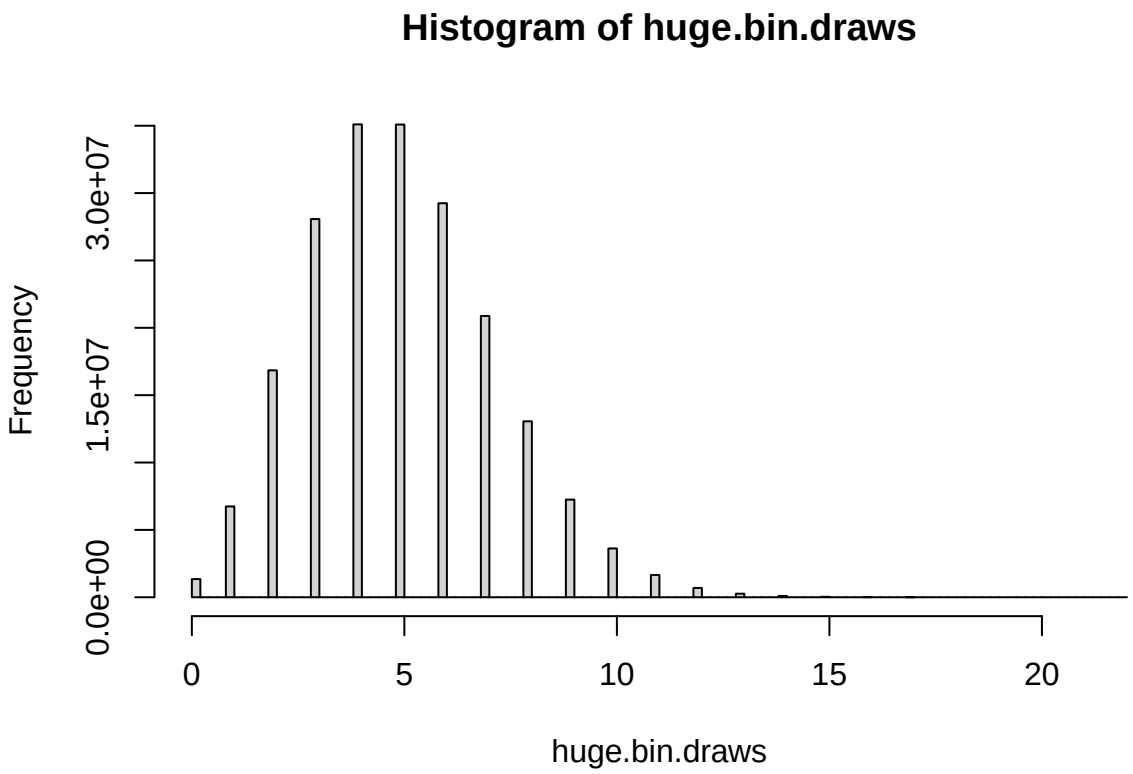
```
t2 <- Sys.time()
t2 - t1
```

```
## Time difference of 7.509465 secs
```

This took longer than calculating the median directly. This is not surprising because although we can intuit that the answer will be the same because the log and exponential will cancel each other, R still performs both of these actions which takes a significant amount of time at this scale.

- **6e.** Plot a histogram of 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)
```



The histogram appears to be a bell curve shape with a long right tail. This is different from the histogram before because we did not standardize this one. This is not surprising because when we have such a large amount of draws, we can't roll negative numbers, but we can roll fairly large positive outliers.

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

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

The lapply() function took significantly longer. This is the case because this approach calls a function for every element in the array. In my previous approach, I performed vector operations, so although all of the same calculations happen, there is far less 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 approach is slower because it recalculates the mean and standard deviation each time the lapply() function is called, resulting in even more inefficiency.

- **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 uses 20 megabytes, whereas the list uses 320 megabytes. The list requires more memory because lists allow for different datatypes to be stored in them, and this flexibility requires more storage space. This flexibility isn't required because we only store integers, so it's better to use vectors in this scenario.