Week 5 R tutorial

Statistics and statistical programming
Northwestern University
MTS 525

Aaron Shaw

October 13, 3030

Contents

This week, we'll introduce some ways to do some the discrete math, generate distributions, and use a for-loop to run a small simulation.

Binomial and factorial functions

In Chapter 3 (and in the previous problem set), you needed to calculate some binomial choice arithmetic and/or factorials. They weren't absolutely necessary for the problem set, but here are the corresponding functions in R.

Let's say we want to calculate how many possible pairs you can draw from a population of ten individuals, a.k.a., $\binom{10}{2}$ or, instead you wanted to calculate 10!

```r
choose(10,2)
```

```
## [1] 45
```

```r
factorial(10)
```

```
## [1] 3628800
```

Note that factorial arithmetic can get quite large quite fast, so consider your processor/memory constraints and options before you try to calculate something truly large like 365!.

Distribution functions

R has a number of built-in functions to help you work with distributions in various ways that also started to come up in *OpenIntro* Chapter 3. I will introduce a couple of points about them here, but I also highly recommend you look at the relevant section of the Verzani *Using R Introductory Statistics* book (pp 222-229) for more on this (and, honestly, for more on most of the topics we're covering in R).

The key to using R to analyze distributions is that R has a set of built-in distributions (e.g. uniform, normal, binomial, log-normal, etc.) and a set of functions (`d`, `p`, `q`, and `r`) that can be run for each distribution. In the example that follows, I'll use a uniform distribution (a distribution between any two values (*min*, *max*) where the values may occur with uniform probability) for my example below. Verzani has others for when you need them.

The `d` function gets you information about the *density function* of the distribution. The `p` function works with the *cumulative probabilities*. The `q` function gets you *quantiles* from the distribution. The `r` function allows you to generate *random samples* from the distribution. As you can see, the letters corresponding to each function *almost* make sense…<*sigh*>. They also each take specific arguments that can vary a bit depending on which kind of distribution you are using them with (as always, the help documentation and the internet can be helpful here).

Onwards to the example code, which uses the different functions to calculate information about a uniform distribution between 0 and 3 (take a moment to think about what that would look like in terms of raw data and/or a plot):

```
dunif(x=1, min=0, max=3) # What proportion of the area is the to the left of 1?
```

```
## [1] 0.3333333
```

```
punif(q=1, min=0, max=3) # Same as the prior example in this case.
```

```
## [1] 0.3333333
```

```
qunif(p=0.5, min=0, max=3) # 50th percentile
```

```
## [1] 1.5
```

```
runif(n=4, min=0, max=3) # Random values in [0,3]
```

```
## [1] 2.0733128 1.0704585 0.4748152 2.7418522
```

Look at the Verzani text for additional examples, including several that can solve binomial probability calculations (e.g., if you flip a fair coin 100 times, what are the odds of observing heads 60 or more times?).

For-loops and a quick simulation

Beyond proving invaluable for rapid calculations of solutions to problem set questions, the distribution functions are very, very useful for running simulations. We won't really spend a lot of time on simulations in class, but I'll give you an example here that can generalize to more complicated problems. I also use a programming technique we haven't talked about yet called a for-loop to help repeat the sampling process multiple times. For-loops feature prominently in some programming tasks/languages, but I encourage you to minimize your use of them in R for reasons that are sort of beyond the scope of the course. That said, it's still super important to learn how they work!

For my simulation let's say that I want to repeatedly draw random samples from a distribution and examine the distribution of the resulting sample means (this example is going to feature prominently in Chapter 5 of *OpenIntro*). I'll start by generating a vector of 10,000 random data points drawn from a log-normal distribution where the mean and standard deviation of the log-transformed values are 0 and 1 respectively:

```
d <- rlnorm(10000, meanlog=0, sdlog=1)

head(d)
```

```
## [1] 1.23390788 0.07583048 1.35689717 1.63832396 0.35423203 0.45353611
```

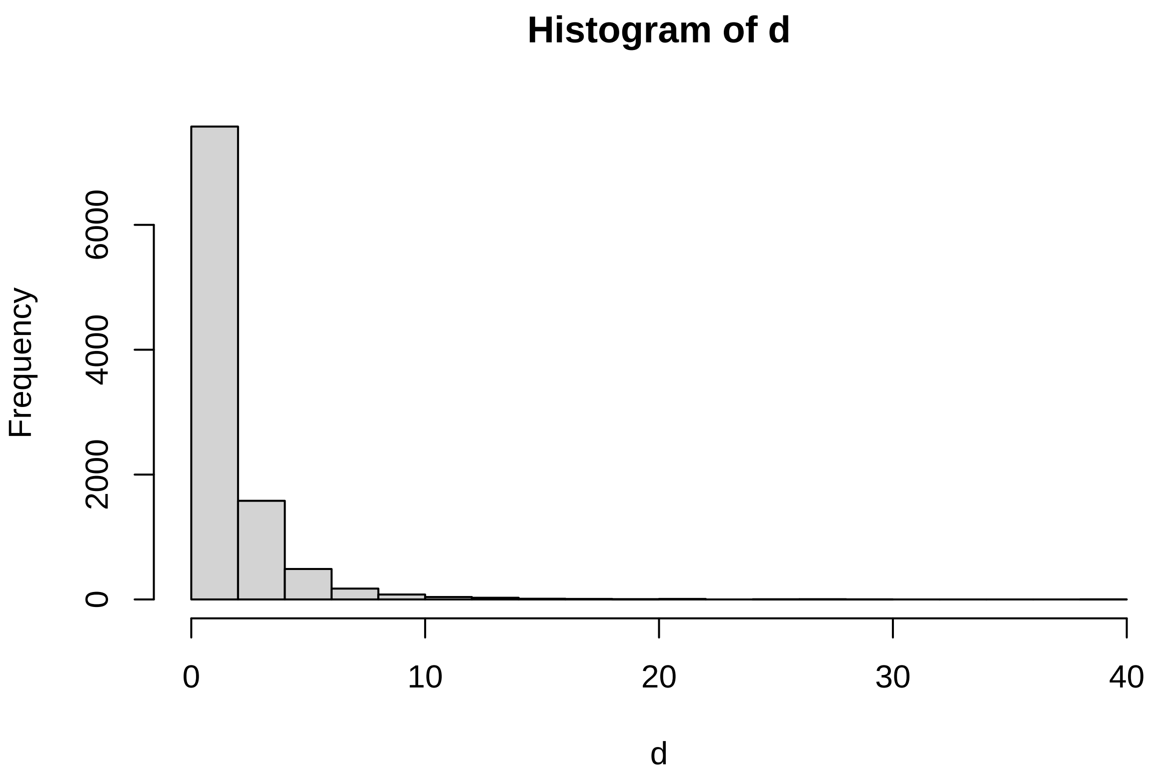
```
mean(d)
```

```
## [1] 1.645479
```

```r
sd(d)
```

```
## [1] 2.088235
```

```r
hist(d)
```



That sure does look like a logarithmic distribution!

Okay, now, I want to draw 500 samples of 100 observations from this population and take the mean of each sample. Time to write a function! Notice that I require two inputs into my function: the population data and the sample size.

```r
sample.mean <- function(pop, n){
  s <- sample(pop, n)
  return(mean(s))
}

## Run it once to see how it goes:
sample.mean(d, 100)
```

```
## [1] 1.656623
```

Next step: let's run that 500 times. Here's where the for-loop comes in handy. In somewhat abstract terms, a for-loop allows me to define an index and then repeat an operation for some range of values that can assume a function of that index. The simplest example is to take a range of integers as my index (e.g., 1-10) and then I run my loop ten times (once for each integer). You can also do more sophisticated things such as using the index to substitute and/or subset the objects in your loop along the way.

We can break that down further through an example. The basic syntax of a for-loop is that you call some operation to occur over some index. Here's a simple example that illustrates how that works. The loop iterates through the integers between 1-10 and prints the square of each value:

```r
for(x in c(1:10)){
  print(x^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

Notice that the initial statement defines a variable (`x` in this case) within the loop (that's a way you can think about the `for(x in...))` part of the first line of code. The part that comes after `in` defines the index (in this case, the integers between 1 and 10). The part between the curly braces defines what R will do for each iteration of the loop, in which it will substitute the corresponding value from the index for variable.

Now let's go back to my sample means example. Since I want to store the output of my sample means loop, I will first create an object `s.means` that is a numeric vector with one value (0) that will be replaced as I run through the loop.

```r
s.means <- 0
```

Onwards to the loop itself. In the block of code below, you'll notice that I once again declare an index over which to iterate. That's what happens inside that first set of parentheses where I have `i in c(1:30)`. That's telling R to go through the loop for each value from 1:30 and to assign the current index value to `i` during that iteration of the loop. Then, each time through the loop, the value of `i` advances through the index (in this case, it just goes up by 1). The result is that each iteration will take the output of my `sample.mean` function and append it as the i^{th} value of `s.means`. The `next` call at the end is optional, but can be important sometimes to help you keep track of what's going on.

```r
for(i in c(1:500)){
  s.means[i] <- sample.mean(d, 100)
  next
}
```

In case you're coming to R from other programming languages that index from 0, you should note that this example very much takes advantage of the fact that R indexes from 1 (i.e., the first value of some vector `v` can be returned by `v[1]`).

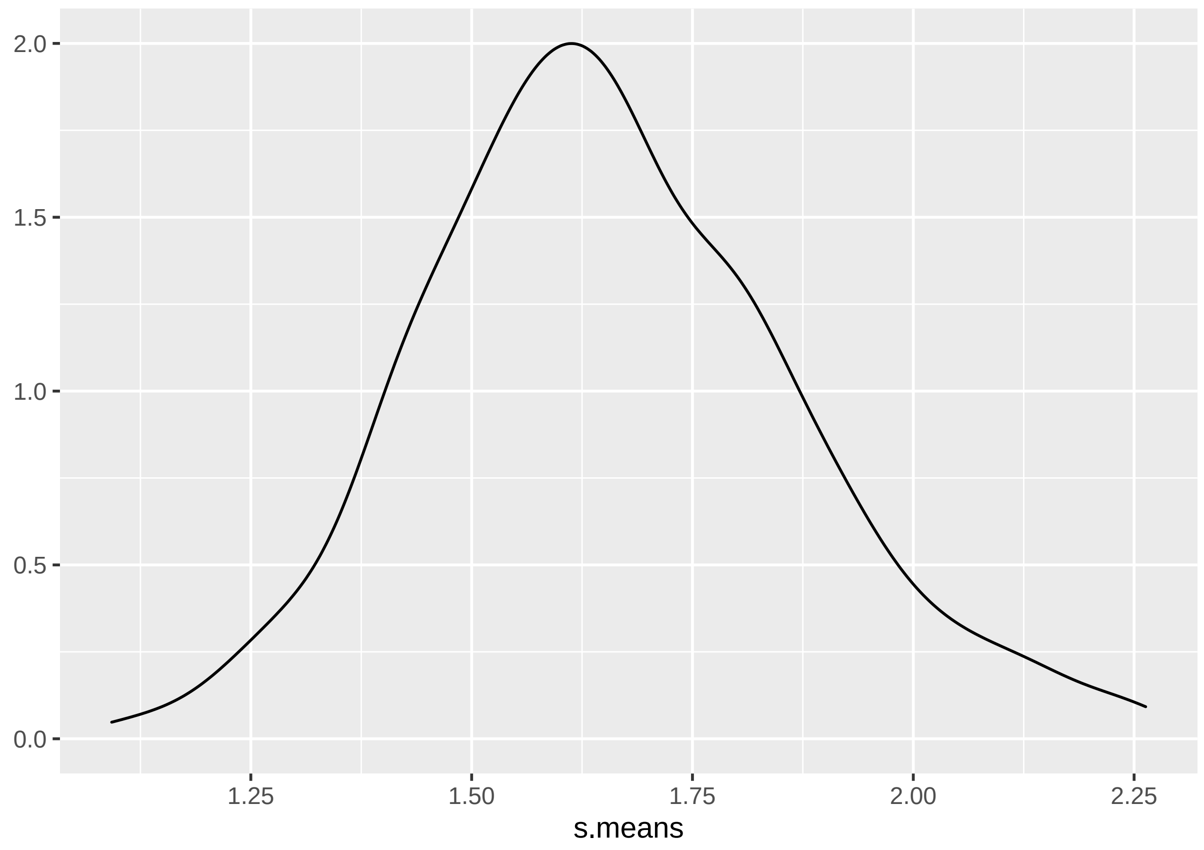
Once we've run that loop the `s.means` variable now contains a distribution of sample means. What are the characteristics of the distribution? In other words, how would you summarize the distribution? Well, you already know how to do that.

```r
summary(s.means)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.092   1.514   1.637   1.655   1.791   2.263
```

Let's plot it too:

```r
library(ggplot2)
qplot(s.means, geom="density")
```



That looks pretty "normal."

Experiment with this example by changing the size of the sample and/or the number of samples we draw.

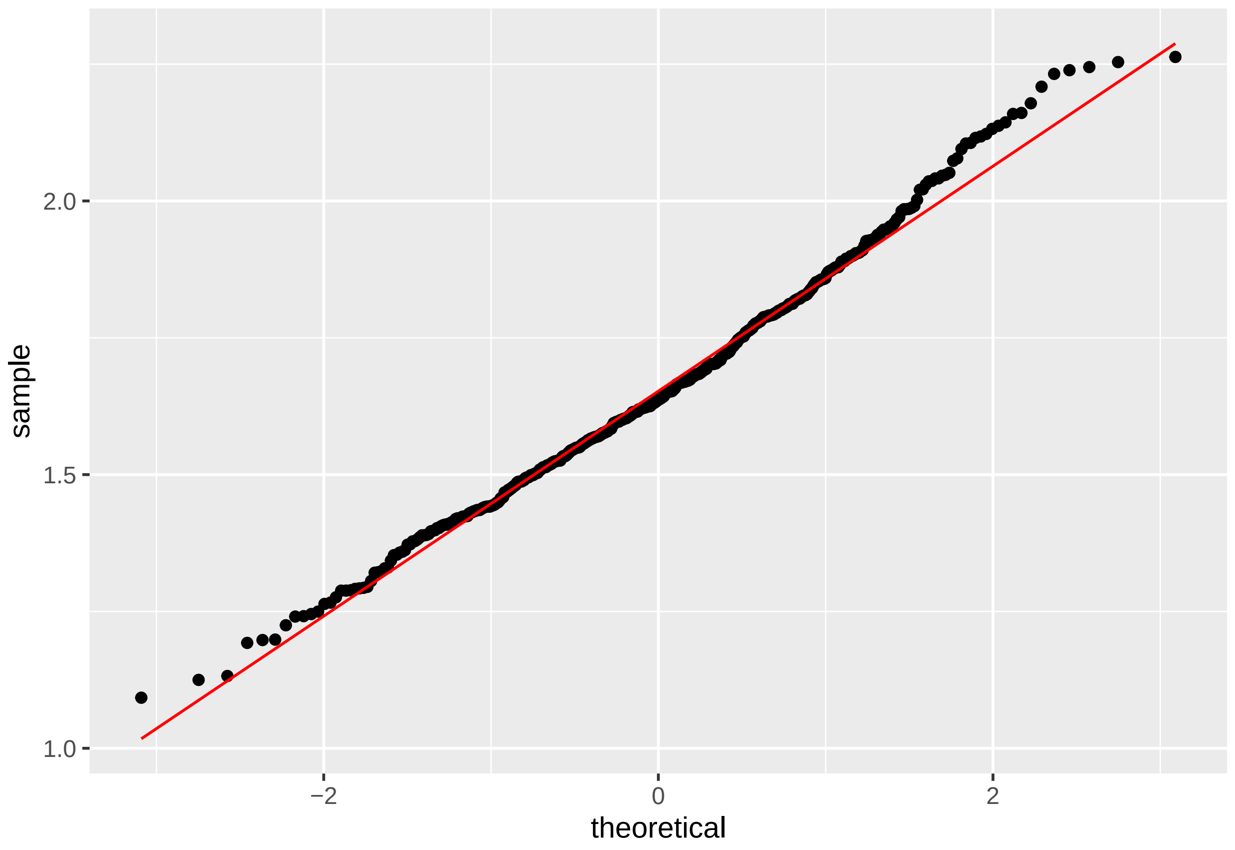
Now, think back to the original vector `d`. Can you explain what fundamental statistical principle is illustrated in this example? Why do the values in `s.means` fluctuate so much? What is the relationship of `s.means` to `d`?

Quantile quantile plots

Last, but not least, you might have admired the quantile-quantile plots (a.k.a. "Q-Q plots") presented in some of the examples in the most recent *OpenIntro* chapter. The usual idea with Q-Q- plots is that you want to see how the observed (empirical) quantiles of some data compare against the theoretical quantiles of a normal (or other) distribution. You too can create these plots!

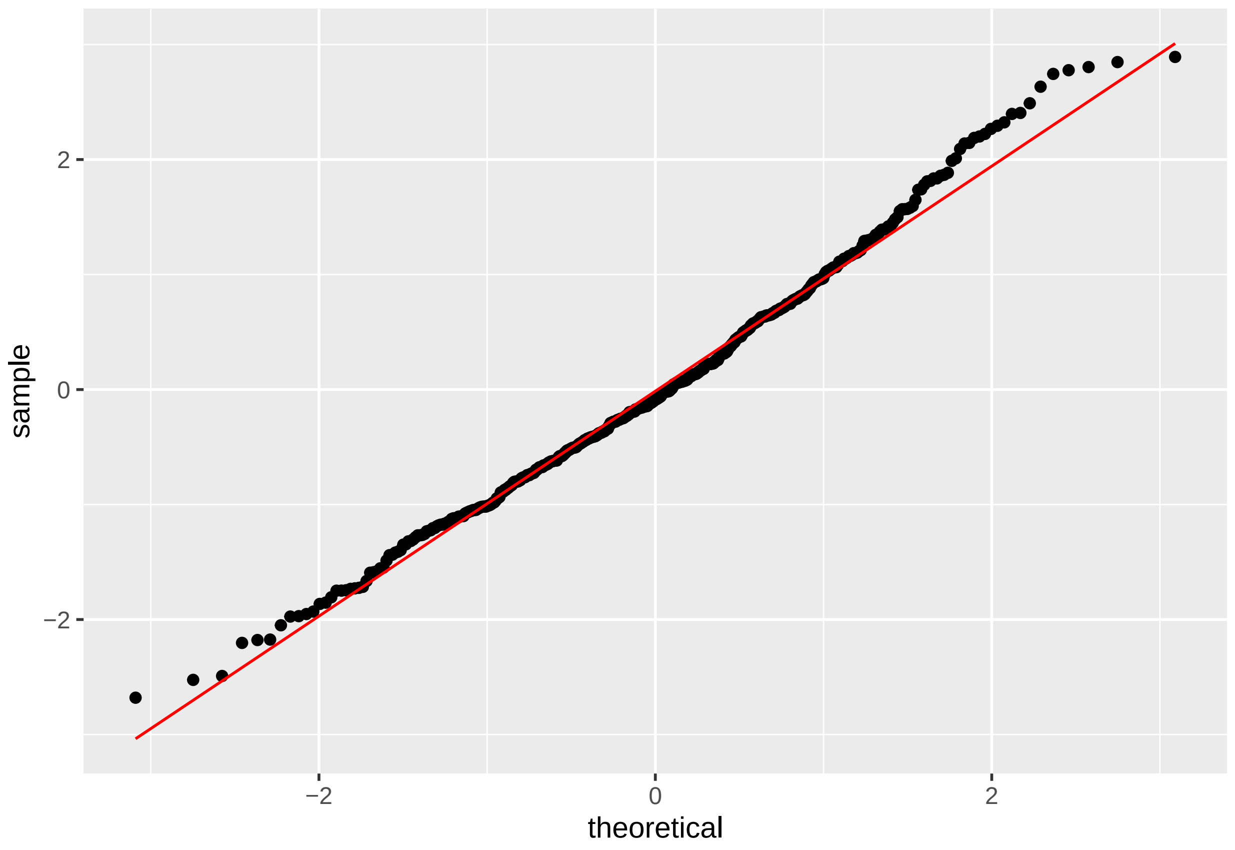
Here's an example that visualizes the result of our simulation (labeled "sample") against a normal distribution with the same mean and standard deviation (labeled "theoretical"). Notice that to accommodate `ggplot2` I have to turn `s.means` into a data frame first.

```r
s.means <- data.frame(s.means)
ggplot(s.means, aes(sample=s.means)) + geom_qq() + geom_qq_line(color="red")
```



And/or (finally) we could even standardize the values of `s.means` as z-scores using the `scale()` function:

```
s.z <- data.frame(scale(s.means)); names(s.z) <- "z"
ggplot(s.z, aes(sample=z)) + geom_qq() + geom_qq_line(color="red")
```


sample
2
0
−2
−2
0
2
theoretical