# Statistical Intuition for Modern Biologists

Isaac Vock

2024-05-07

# Table of contents

# Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

```
1 + 1
```

```
[1] 2
```

# Part I

# Necessary Introduction

# 1 Introduction to RNA-seq

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

```
1 + 1
```

```
[1] 2
```

# 2 Introduction to R

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

# 3 Introduction to Probability

Why collect replicates of the same exact data? The (in)famous definition of insanity, "Doing something over and over again and expecting a different result", would seem to classify replication as crazy. Except, anyone who has ever collected any kind of data has an intuition for why replication is important: each replicate is always different from the last. Somehow, despite following a precise protocol in a particular lab with the same reagents, pipettes, measuring devices, etc., every replicate yields a slightly different result. **There is randomness in your data**.

This randomness throws a wrench in naive data analyses. In biology, we are often measuring something (e.g., the concentration of an interesting molecule, the viability of an organism, etc.) in two or more "conditions" (e.g., with and without drug) and assessing whether or not any of our measurements differ from one another. If every replicate of an experiment yielded the same data, this task would be trivial. Just take one measurement in each condition, compare the measurements, and draw conclusions. If the measurements differ, the thing you are measuring differs. If the measurements are identical, the thing you are measuring is unaffected by your perturbations. The randomness of data forces us to consider an alternative conclusion. What if our measurements differ, not because the thing we are measuring is changing, but because our measurements fluctuate from experiment-to-experiment? How do we know what differences are real and what differences are the result of this randomness?

We will start answering this question with the help of **probability theory**. Probability theory is a field of mathematics that gives us the tools to think about and quantify this randomness. This chapter will introduce you to the key instrument provided by this field: the probability distribution. With this tool, we will be able to describe and better understand the random fluctuations that plague our data.

# 4 The Many Flavors of Randomness

To understand how statisticians think about randomness, consider an analogy. In biology, everything is complicated and every system we study seems unique. To tackle this complexity, we group things into bins based on important characteristics they share, and then we try to understand the patterns that govern each bin. Bins in biology may be determined by the type of organism being studied (e.g., entymology, mammology, microbiology, etc.), the activities of molecular machines you investigate (e.g., phosphorylating subtrates, regulating transcription, intracellular transport, etc.), and much more. Similarly in statistics, every kind of data seems to possess unique sources and types of variance. Work with enough data though and you will notice that there are patterns in the types of randomness we commonly see. We thus focus on understanding these common patterns and apply them to understanding our unique data.

The patterns in randomness that we observe are coined "probability distributions". Think of them as functions, which take as input a number, and provide as output the probability of observing that number. These functions could take any shape you dream up, but as mentioned, particular patterns crop up all of the time. In the following exercises, we are going to explore these patterns and understand what gives rise to them.

## 4.1 The basics

To illustrate some general facts about randomness, we will start by working with two functions in R: `rbinom()` and `rnorm()`. These are two of many "random number generators" (RNGs) in R. As we will learn in these exercises, there isn't just one kind of randomness, so there isn't just one RNG.

`rbinom` and `rnorm` both have 3 parameters. Both have a parameter called `n`, which sets the number of random numbers to generate. `rbinom` has two other parameters called `size` and `prob`. `size` can be any integer >= 0. `prob` can be any number between 0 and 1 (including 0 and 1). `rnorm`'s two additional parameters are called `mean` and `sd`. `mean` can be any number, and `sd` can be any positive number. Take some time to play with both of these functions. As you do so, consider the following:

1. In what ways are their output similar?
2. In what ways are their output distinct?
3. How do `prob` and `size` impact the output of `rbinom`?
4. How do `mean` and `sd` impact the output of `rnorm`?

5. Make histograms with different numbers of random numbers (i.e., different `n`). What do you see as `n` increases? Now repeat this but with different parameters for the relevant function. What changes?

### 4.1.1 Example exercise:

Generated 5 draws from each to compare general output:

```
print("rbinom output:")
```

```
[1] "rbinom output:"
```

```
rbinom(10, size = 100, prob = 0.5)
```

```
 [1] 53 55 54 49 56 45 51 56 50 47
```

```
print("rnorm output:")
```

```
[1] "rnorm output:"
```

```
rnorm(10)
```

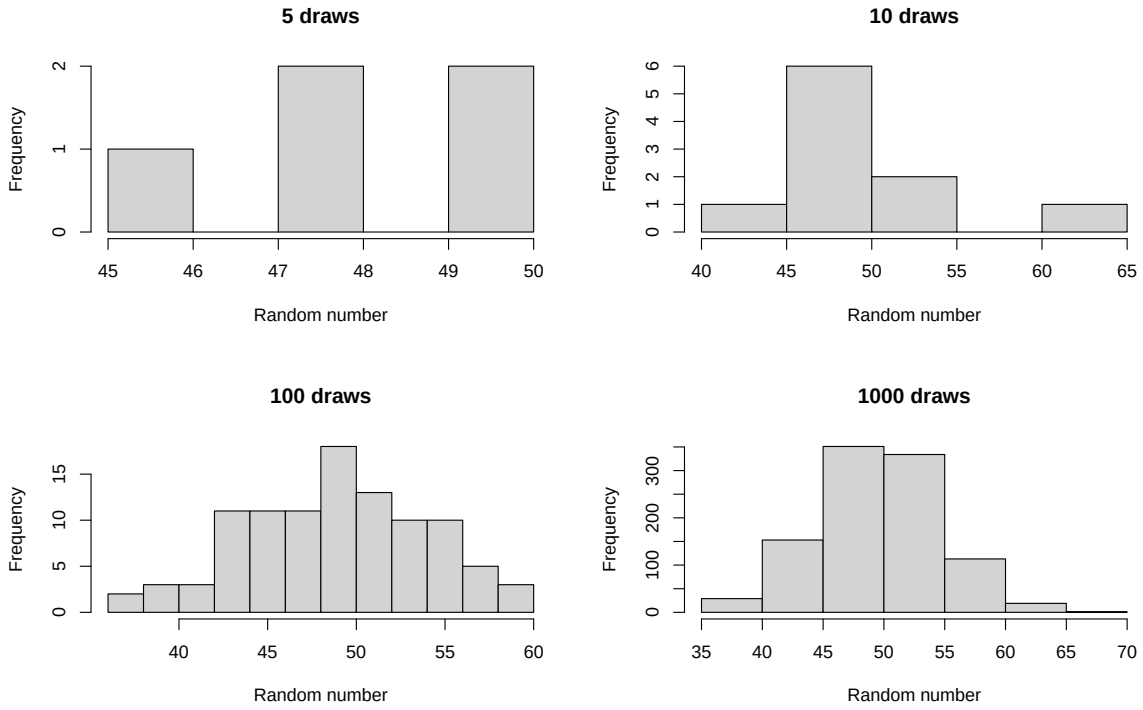
```
 [1] -0.09239572 -0.29752735 -0.29207450  1.00283619  0.08178955  1.84431925
 [7]  0.22600316 -1.35834974 -1.23356828  2.43031003
```

Observations:

1. `rbinom` seems to only output exact integers. rnorm's output has lots of decimal places
2. As expected, output for both usually differs from run to run
3. `rnorm` output seems to always differ while `rbinom` might spit out the same number from time to time.

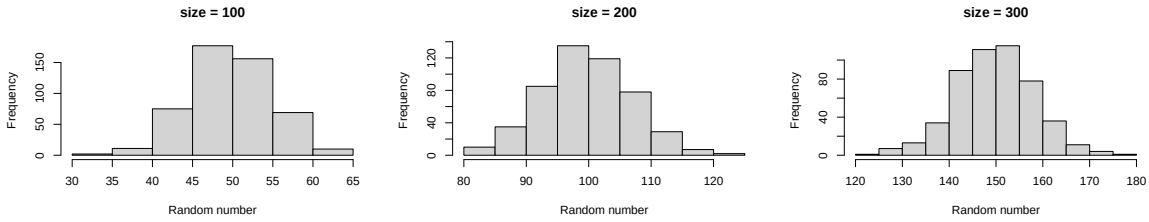
Taking more draws reveals patterns in the randomness that is not as evident from less draws:

```r
b5 <- rbinom(n = 5, size = 100, prob = 0.5)
b50 <- rbinom(n = 10, size = 100, prob = 0.5)
b500 <- rbinom(n = 100, size = 100, prob = 0.5)
b5000 <- rbinom(n = 1000, size = 100, prob = 0.5)


hist(b5, main = "5 draws", xlab = "Random number")
hist(b50, main = "10 draws", xlab = "Random number")
hist(b500, main = "100 draws", xlab = "Random number")
hist(b5000, main = "1000 draws", xlab = "Random number")
```




Varying size of rbinom:

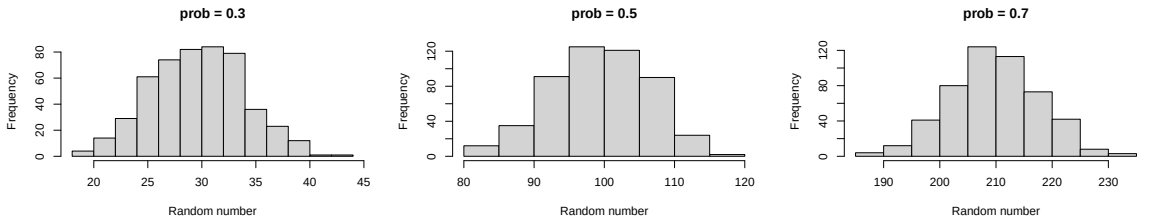
```r
b100 <- rbinom(500, size = 100, prob = 0.5)
b200 <- rbinom(500, size = 200, prob = 0.5)
b300 <- rbinom(500, size = 300, prob = 0.5)

hist(b100, main = "size = 100", xlab = "Random number")
hist(b200, main = "size = 200", xlab = "Random number")
hist(b300, main = "size = 300", xlab = "Random number")
```




Higher size generates larger random numbers. In fact, distribution of random numbers seems to increase in proportion to increase in size (i.e., doubling size doubles the average random number generated)

Varying prob of `rbinom`

```
b0.3 <- rbinom(500, size = 100, prob = 0.3)
b0.5 <- rbinom(500, size = 200, prob = 0.5)
b0.7 <- rbinom(500, size = 300, prob = 0.7)

hist(b0.3, main = "prob = 0.3", xlab = "Random number")
hist(b0.5, main = "prob = 0.5", xlab = "Random number")
hist(b0.7, main = "prob = 0.7", xlab = "Random number")
```




Higher prob has a similar effect to higher size, increasing the average magnitude of random numbers generated. There also seems to be a similar linear proportionality.

### 4.1.2 Takeaways

1. There are two kinds of randomness: discrete and continuous.
2. Discrete randomness is randomness where the output can be described with integers.
3. Continuous randomness is randomness where the output can be described with real numbers.
4. Randomness is described by its patterns. Any individual data point may be random but collect enough of them from a given process/source and a pattern will emerge. Some values are more likely to occur, some are less likely, and some may even never occur.

### 4.1.3 Properties of random variables

#### 4.1.3.1 Mean

The mean of a distribution is a measure of its "central tendencies". Usually, values closer to the mean are more common than values further away from the mean. The mean of a set of random numbers can be calculated in R using the `mean()` function:

```
b <- rbinom(100, size = 100, prob = 0.5)
n <- rnorm(100, mean = 5)

mean(b)
```

```
[1] 49.91
```

```
mean(n)
```

```
[1] 5.08514
```

Because of the randomness inherent in RNGs, the mean that you will get for any finite sample will often vary from sample to sample:

```
b1 <- rbinom(100, size = 100, prob = 0.5)
b2 <- rbinom(100, size = 100, prob = 0.5)
b3 <- rbinom(100, size = 100, prob = 0.5)

mean(b1)
```

```
[1] 49.87
```

```
mean(b2)
```

```
[1] 49.78
```

```
mean(b3)
```

```
[1] 51.25
```

Every distribution has some inherent mean given its set of parameters. You can think of this as the mean you would get if you took a really large sample. For example, the `mean` parameter in `rnorm` is exactly this large sample mean. You can see how the mean of a given sample tends to gets closer to this value as you increase the sample size:

```
n10 <- rnorm(10, mean = 5)
n100 <- rnorm(100, mean = 5)
n1000 <- rnorm(1000, mean = 5)

mean(n10)
```

```
[1] 4.485152
```

```
mean(n100)
```

```
[1] 5.067931
```

```
mean(n1000)
```

```
[1] 5.029942
```

You can calculate the mean yourself with some simple R code; it is the sum of all your data points divided by the number of data points:

```
n100 <- rnorm(100, mean = 5)

mean(n100)
```

```
[1] 5.080806
```

```
sum(n100)/length(n100)
```

```
[1] 5.080806
```

For any distribution, you can look up its mean (as well as the other properties discussed below) on sites like Wikipedia:

1. Binomial distribution
2. Normal distribution

#### 4.1.3.2 Variance/standard deviation

A key feature of randomness is the amount of variability from outcome to outcome of a random process. As we will see when we start using models of data randomness to draw conclusions about said data, figuring out how variable our data is represents a key challenge.

We often describe the variability of a random sample using the variance and standard deviation. These can be calculated in R using the `var` and `sd` functions, respectively:

```
b <- rbinom(100, size = 100, prob = 0.5)
n <- rnorm(100, mean = 5)

sd(b)
```

```
[1] 5.057807
```

```
sd(n)
```

```
[1] 0.8946398
```

```
var(b)
```

```
[1] 25.58141
```

```
var(n)
```

```
[1] 0.8003804
```

Close inspection will reveal that the output of `var` is just the output of `sd` squared:

```
b <- rbinom(100, size = 100, prob = 0.5)
n <- rnorm(100, mean = 5)

sd(b)^2
```

```
[1] 23.70697
```

```r
var(b)
```

```
[1] 23.70697
```

```r
sd(n)^2
```

```
[1] 0.9994411
```

```r
var(n)
```

```
[1] 0.9994411
```

The variance measures the squared distance of each data point from the mean, and sums up all of these squared distances:

```r
b <- rbinom(100, size = 100, prob = 0.5)
n <- rnorm(100, mean = 5)

var(b)
```

```
[1] 34.43192
```

```r
sum( ( b - mean(b) )^2 )/(length(b) - 1)
```

```
[1] 34.43192
```

```r
var(n)
```

```
[1] 1.038207
```

```r
sum( ( n - mean(n) )^2 )/(length(n) - 1)
```

```
[1] 1.038207
```

Why the square? When assessing variability, we don't care if data is above or below the mean, so we want to score data points greater than the mean equally to how we score those below the mean. The square of the mean - 1 is the same as the square of the mean + 1, and both are positive values, because squaring a real number always yields a positive number. Why not the absolute value? Or the fourth power of the distance to the mean? Or any other definitively positive value? Convenience and mathematical formalism. In other words, don't worry about it.

**NOTE: EXTRA DETAIL ALERT**

The other mystery of the calculation of the standard deviation/variance is the fact that the sum of the deviations from the mean are divided by the number of data points **minus 1**, rather than just the number of data points, as in the mean. Why is this? Again, mathematical formalism, but there is an intuitive explanation to be discovered.

When calculating the mean, each data point is its own entity that contributes equally to the calculation. I can't calculate the mean without all n data points. When calculating the standard deviation though, each data point is compared to the mean, which is itself calculated from all of the data points.

If I tell you the mean though, I only need to tell you n - 1 data points for you to infer the final data point. For example, if I have 3 numbers that I tell you average to 2, and two of them are 1 and 3, what is the third? It has to be 2. Thus, when calculating the distance from each point to the mean, there are only n - 1 unique pieces of information. I tell you n - 1 data points and the mean, and you can tell me the standard deviation (since you can infer the missing data point). We say that there are "n - 1 degrees of freedom". Thus, the effective average of the differences from the mean are all of the squared differences divided by n - 1, the number of "non-redundant" data points that remain after calculating the mean.

**END OF THE EXTRA INFO ALERT**

#### 4.1.3.3 Higher order "moments" (BONUS CONTENT)

The mean and the variance are related to what are called (who knows why) moments of a distribution. A moment is the average of some function of the random data. For example, the mean is the average of the data itself, which you could call data passed through the function f(x) = x. The variance is related to this moment, as well as the so-called second moment, which is the average of the square of the random data (f(x) = x^2). In fact, this leads to an equivalent way to calculate the variance:

```
b <- rbinom(100, size = 100, prob = 0.5)
n <- rnorm(100, mean = 5)

var(b)
```

```
[1] 22.44808
```

```
mean(b^2) - mean(b)^2
```

```
[1] 22.2236
```

```
var(n)
```

```
[1] 1.111262
```

```
mean(n^2) - mean(n)^2
```

```
[1] 1.10015
```

Well not fully equivalent, but they get closer to one another as the sample size increases:

```
n100 <- rnorm(100, mean = 5)
n1000 <- rnorm(1000, mean = 5)
n10000 <- rnorm(10000, mean = 5)
```

```
n = 100:
var(x):  1.082396
(x^2) - mean(x)^2:  1.071572

n = 1000:
var(x):  1.031779
(x^2) - mean(x)^2:  1.030747

n = 10000:
var(x):  1.007057
x^2) - mean(x)^2:  1.006956
```

Various other moments or functions of moments enjoy some level of popularity in specific circles. We won't talk about them here, but you'll see them on Wikipedia sites for distributions, things like the "skewness" and "excess kurtosis" are examples of these additional moments, or functions of moments.

#### 4.1.3.4 Density/mass functions

Earlier, I said "Think of [probability distributions] as functions, which take as input a number, and provide as output the probability of observing that number". Fleshing out what I mean by this will help you understand one of the key concepts in probability theory: probability density/mass functions.

Take a look at what a normalized histogram of draws from `rbinom()` looks like as we increase `n`. "Normalized" here means that instead of the y-axis representing the number of draws in a given bin (what is plotted by `hist()` by default), it is this divided by the total number of draws.

```
b5 <- rbinom(n = 10, size = 100, prob = 0.5)
b50 <- rbinom(n = 100, size = 100, prob = 0.5)
b500 <- rbinom(n = 1000, size = 100, prob = 0.5)
b5000 <- rbinom(n = 10000, size = 100, prob = 0.5)


# freq = FALSE will plot normalized counts
hist(b5, main = "10 draws", xlab = "Random number", freq = FALSE)
hist(b50, main = "100 draws", xlab = "Random number", freq = FALSE)
hist(b500, main = "1000 draws", xlab = "Random number", freq = FALSE)
hist(b5000, main = "10000 draws", xlab = "Random number", freq = FALSE)
```

As you increase `n`, a pattern begins to emerge. The y-axis is the fraction of the time a random number ended up in a particular bin. It can thus be interpreted as the probability that the random number falls within the bounds defined by that bin. What you may notice is that these probabilities seem to converge to particular values as `n` gets large. This pattern of probabilities is known as the binomial distributions "probability mass function". If the output of the RNG is continuous real numbers, then this is referred to as the "probability density function".

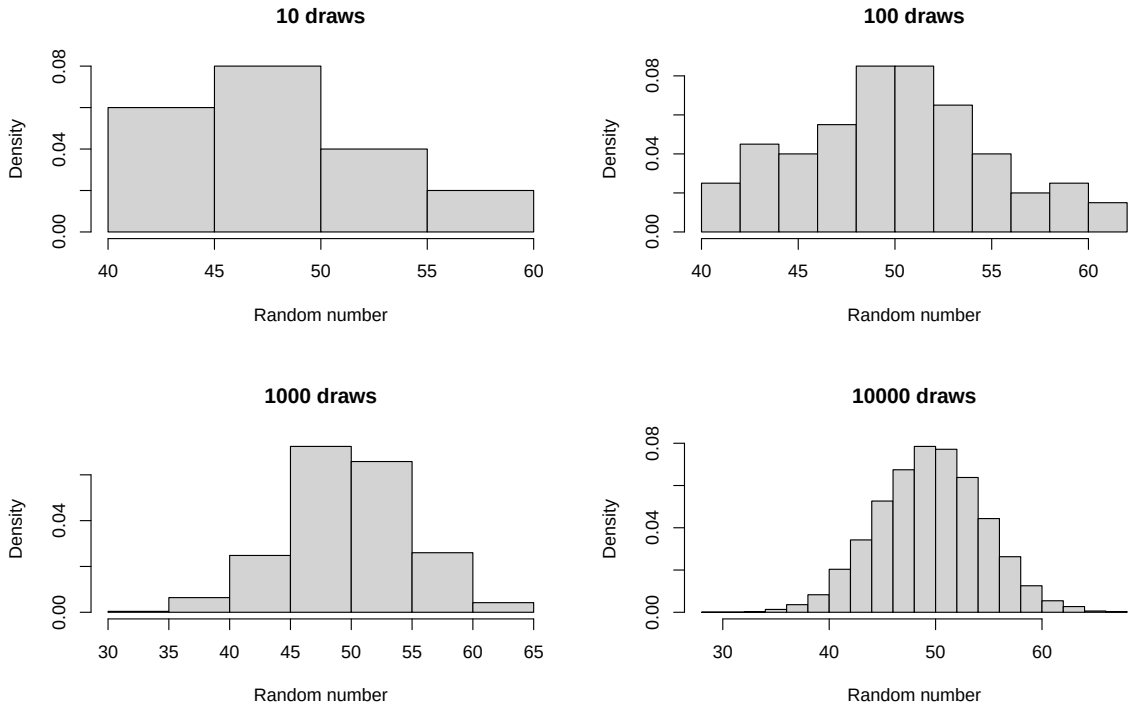
For all of the distributions that we can draw random numbers from in R, we can similarly use R to figure out its probability mass/density function. The function will have the form `d<distribution>`, where `d` denotes density function:

```
dbinom(x = 50, size = 100, prob = 0.5)
```

```
[1] 0.07958924
```

```
dbinom(x = 10, size = 100, prob = 0.5)
```

```
[1] 1.365543e-17
```




```
dbinom(x = 90, size = 100, prob = 0.5)
```

```
[1] 1.365543e-17
```

```
dbinom(x = 110, size = 100, prob = 0.5)
```

```
[1] 0
```

```
dbinom(x = 50.5, size = 100, prob = 0.5)
```

```
Warning in dbinom(x = 50.5, size = 100, prob = 0.5): non-integer x = 50.500000
```

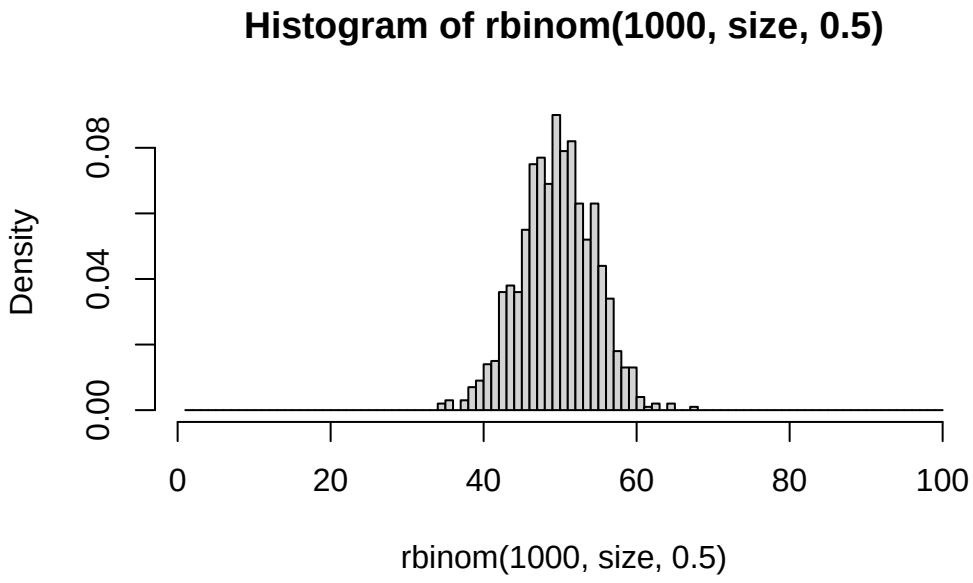
```
[1] 0
```

#### 4.1.3.4.1 Exercise

Prove to yourself that `dbinom` yield probabilities similar to what you saw in your normalized histograms.

Hint:

```
### Make bins of size 1 so as to match up more cleanly with `dbinom()`

# Maximum value
size <- 100

# Plot with bin size of 1
hist(rbinom(1000, size, 0.5), freq = FALSE, breaks = 1:size)
```




**EXAMPLE EXERCISE:**

```
library(ggplot2)
library(dplyr)
```

```
Warning: package 'dplyr' was built under R version 4.3.2
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

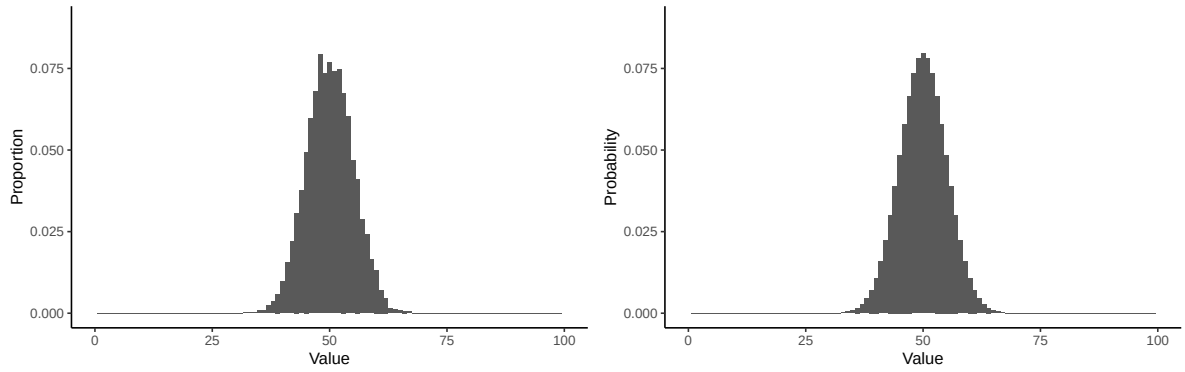
```r
# Histogram
rns <- rbinom(10000, 100, 0.5)

# dbinom
values <- 0:100
probs <- dbinom(values, size = 100, prob = 0.5)

tibble(x = rns) %>%
  ggplot(aes(x = x)) +
  theme_classic() +
  geom_histogram(binwidth = 1, aes(y=..count../sum(..count..))) +
  xlab("Value") +
  ylab("Proportion") +
  xlim(c(0, 100)) +
  ylim(c(0, max(probs + 0.01)))
```

```
Warning: The dot-dot notation (`..count..`) was deprecated in ggplot2 3.4.0.
i Please use `after_stat(count)` instead.
```

```
Warning: Removed 2 rows containing missing values (`geom_bar()`).
```

```r
tibble(x = values,
       y = probs) %>%
  ggplot(aes(x = x, y = y)) +
  geom_bar(stat = "identity") +
  theme_classic() +
  xlab("Value") +
  ylab("Probability") +
  xlim(c(0, 100)) +
  ylim(c(0, max(probs + 0.01)))
```

```
Warning: Removed 2 rows containing missing values (`geom_bar()`).
```




**END OF EXAMPLE EXERCISE**

In the case of discrete random variables, this exercise shows that the output of `d<distribution>()` has a clear interpretation: its the probability of seeing the specified value given the specified distribution parameters. In the case of continuous random variables though, the interpretation is a bit trickier. For example, consider an example output of `dnorm()`:

```
dnorm(0, mean = 0, sd = 0.25)
```

```
[1] 1.595769
```

It's greater than 1! How could a probability be greater than 1?? The answer is that the output isn't a probability in the same way as in the discrete case. All that matters for this class is that this number is *proportional* to a probability. The distinction here is a bit unintuitive, but there are some great resources for fleshing out what this means. See for example 3blue1brown's video on the topic.

**BEGINNING OF BONUS CONTENT**

Here I will briefly explain what the output of `dnorm()` represents.

The output of `dnorm()` represents a "probability density", hence the name "probability density function". What is a "probability density"? The analogy to an object's mass and density is fitting. If you want to get something's mass given its density, you need to multiply its density by that object's volume. To get probability (mass) from a density (output of `dnorm()`) you need to specify how large of a bin to consider.

```
dnorm(0, mean = 0, sd = 0.25)*0.001
```

```
[1] 0.001595769
```

This can be interpreted as the probability of a number generated by `rnorm()` falling between -0.0005 (i.e., 0 - 0.001/2) and 0.0005. This is only an approximation, as the density (output of `dnorm()`) is not constant between -0.0005 and 0.0005:

```
dnorm(-0.0005, mean = 0, sd = 0.25)
```

```
[1] 1.595766
```

```
dnorm(0, mean = 0, sd = 0.25)
```

```
[1] 1.595769
```

Choose a small enough bin though, and it will be pretty close to the probability of a number ending up in that bin. The exact answer to this question comes from integrating the probability density function within the bin of interest:

$$\mathrm{P}(\mathrm{x} \in [L, U]) = \int_{L}^{U} \mathrm{dnorm}(\mathrm{x},\ \mathrm{mean} = 0,\ \mathrm{sd} = 0.25)\ \mathrm{dx}$$

$\mathrm{P}(\mathrm{x} \in [L, U])$ should be read as "the probability that a random variable x drawn from `rnorm()` falls between $L$ and $U$".

**!!END OF LECTURE 1!!**

## 4.2 Discrete random variables

### 4.2.1 Modeling two outcomes: the binomial distribution

You have already played around with the binomial distribution, as that is the name given to the distribution of numbers generated by `rbinom`. Now it is time to tell its story. All distributions have a story, or rather, a generative model. A generative model is a process that would give rise to data following a particular distribution.

The binomial's story goes like this: imagine you have an event that can result in two possible outcomes. For example, flipping a coin (an event) can result in either a heads or a tails (two possible outcomes). One thing has to be true about this process for it to be well described by a binomial distribution: the probability of a particular outcome must be exactly the same from event to event. For example, if every time you flip a coin, it has a 50% chance of coming up heads and a 50% chance of coming up tails, then the number of heads is well described by a binomial distribution.

The `size` parameter in the `rbinom` function sets the number of events you want to simulate. The `prob` parameter sets the probability of an outcome termed the "success". Success here has a misleading connotation; it might represent an outcome you are happy about, it might represent an outcome that you displeased by, it might represent an outcome that you are completely indifferent to. Statisticians name things in funny ways…

#### 4.2.1.1 Exercise

Take some time to explore the properties and output of `rbinom()`. Questions to consider include:

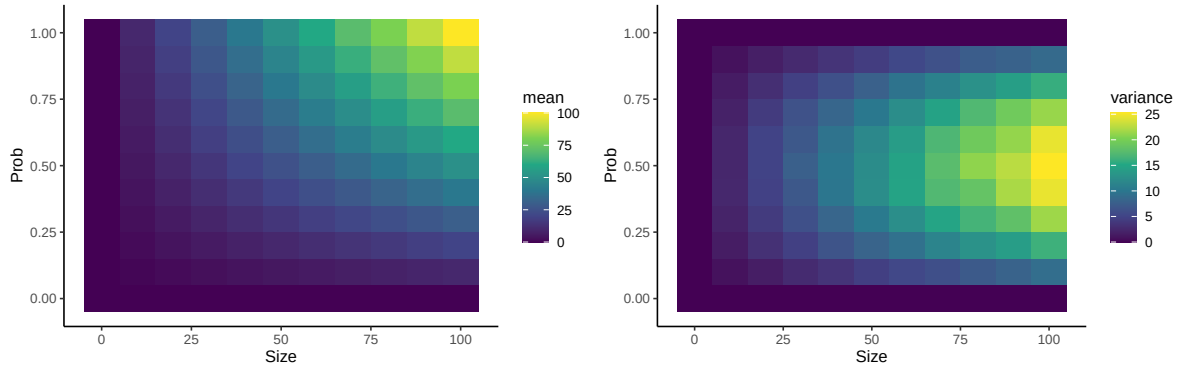
1. How does the mean depend on `size`?
2. How does the mean depend on `prob`?
3. How does the variance depend on `size`? `prob`?
4. What is the most likely outcome for a given `size` and `prob`?
5. What is an aspect of RNA-seq data that you could model with a binomial distribution?

#### 4.2.1.2 Example exercise:

Going hardcore: inspect properties of distribution as function of size and prob

```
library(dplyr)
library(ggplot2)

nsims <- 5000

probs <- seq(from = 0, to = 1, by = 0.1)
sizes <-  seq(from = 0, to = 100, by = 10)

Lp <- length(probs)
Ls <- length(sizes)


means <- rep(0, times = Lp*Ls)
vars <- means

count <- 1
for(p in seq_along(probs)){

  for(s in seq_along(sizes)){
```

```
    simdata <- rbinom(nsims, size = sizes[s], prob = probs[p])

    means[count] <- mean(simdata)
    vars[count] <- var(simdata)
    count <- count + 1

  }

}


sim_df <- dplyr::tibble(prob = rep(probs, each = Ls),
                        size = rep(sizes, times = Lp),
                        mean = means,
                        variance = vars)

sim_df %>%
  ggplot(aes(x = size, y = prob, fill = mean)) +
  geom_tile() +
  scale_fill_viridis_c() +
  theme_classic() +
  xlab("Size") +
  ylab("Prob")
sim_df %>%
  ggplot(aes(x = size, y = prob, fill = variance)) +
  geom_tile() +
  scale_fill_viridis_c() +
  theme_classic() +
  xlab("Size") +
  ylab("Prob")
```



Observations:

1. Confirms that higher size and prob = higher average number
2. Interesting to see that variance increases as a function of size, but that prob = 0.5 leads to the highest variance.
3. Setting prob to 0 or 1 causes variance to go exactly 0, regardless of what size is set to. Similarly, setting prob to 0 causes the mean to go to exactly 0, regardless of what size is set to.

### 4.2.2 Modeling > 2 outcomes: the multinomial distribution

When exploring the binomial distribution, we considered data with two possible outcomes. What if many of the same assumptions hold, but there are now more than 2 possible results? That is where the multinomial distribution comes in.

**Generative model**: Imagine you have an event that can result in one of N outcomes, where N is some integer. Each outcome has some probability, p_{i} (i denoting the ith outcome, for i between 1 and N, inclusive) of occurring, and all of the p_{i} remain constant from event to event.

**Example case**: When rolling a die, there are 6 outcomes of what number shows face up when the die stops rolling (1, 2, 3, 4, 5, or 6). If each face is equally likely to show up with each roll, then p_{i} = 1/6 for all i. This can be simulated in R with `rmultinom()`:

```
rmultinom(5, size = 1, prob = rep(1/6, times = 6))
```

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

The output will be a matrix with `n` columns and number of rows equal to the length of `prob`. The value in the ith row and jth column is the number of times event i was observed in simulation j. In this case, the rows can be interpreted as the number a die came up, and each columns represents the result of a single role of a single die.

#### 4.2.2.1 Exercise

Use `rmultinom()` and a bit of ancillary code to simulate the sequence of an RNA produced from a gene. This doesn't need to be a real gene, just a random sequence of nucleotides. Then determine some features of this sequence:

1. What is the longest run of each nucleotide in a row?
2. What is the least common nucleotide in your sequence?
3. What is the most common nucleotide in your sequence?
4. How different is the rate of occurence of the least and most common nucleotides

#### 4.2.2.2 Example exercise

```
### Parameters for simulation

# Number of nucleotides in RNA
length <- 25

# Nucleotide proportions
ATprop <- 0.4
CGprop <- 0.6

### Generate random nucleotides
nucleotides <- rmultinom(length, size = 1,
                         prob = c(ATprop/2, ATprop/2,
                                  CGprop/2, CGprop/2))

### Determine sequence
nucleotide_types <- c("A", "U", "C", "G")
nucleotide_ids <- 1:4

# Cute matrix algebra trick
  # IDs will be row ID for which value of matrix was 1
IDs <- t(nucleotides) %*% matrix(1:4, nrow = 4, ncol = 1)
sequence <- paste(nucleotide_types[IDs], collapse = "")


print(sequence)
```

```
[1] "AUAAUUGUCGAUCACGCCCGCUGCG"
```

### 4.2.3 Modeling "counts": the Poisson distribution

As we have seen, RNA-seq data can be summarized as a matrix of read counts. Read counts are the number of times that we observed an RNA fragment from a particular genomic feature. What then, is a good model for the number of read counts for a particular feature? Enter a major contender: the Poisson distribution

**Generative model**: Imagine that "events" are independent, that is the time since the last event or how many such events have already occurred have no bearing on how likely we are to see another event. The number of such events in a particular period of "time" (time here in quotes as time could represent units like "number of RNA fragments sequenced") will be Poisson distributed.

#### 4.2.3.1 Exercise

1. Explore the output of `rpois()`, the Poisson distribution random number generator. Besides the typical `n` parameter, it has only one other parameter, `lambda`. Investigate the impact of changing `lambda`.
2. Compare the amount of variability in `rpois()` output to the amount of variability in a real RNA-seq dataset.

### 4.2.4 Modeling time to first "success": the geometric distribution

The binomial distribution provides a great model for the number of "successes" in a set of random, binary outcomes. What if we were interested in how long we will have to wait until the first success though? That is where the geometric distribution comes into play.

**Generative Model**: Like with the binomial distribution, imagine you have an event that can result in two possible outcomes, and the probability of a particular outcome is exactly the same from event to event. The number of events until you see until the first success will follow a geometric distribution.

#### 4.2.4.1 Exercise

1. Explore the output of `rgeom()`, the geometric distribution random number generator. Besides the typical `n`, it has only one additional parameter, `prob`, which can be a number between 0 and 1.
2. Use `rbinom()` and a bit of custom R code to create your own `rgeom()` alternative, and confirm that the average value given by your simulator and `rgeom()` for a given `prob` is approximately equal.

### 4.2.5 Modeling time to nth "success": the negative binomial distribution

The geometric distribution describes the time until the first success of a binary outcome. What if we need to model the time to the nth success, where n is any integer $> 1$? For that, we can look to the negative binomial distribution.

**Generative Model**: Like with the binomial distribution, imagine you have an event that can result in two possible outcomes, and the probability of a particular outcome is exactly the same from event to event. The number of events until you see until the nth success will follow a negative binomial distribution.

**Spoiler alert**: We'll see the negative binomial later, but it will take on a very different character, acting as a generalization of the Poisson distribution rather than of the geometric distribution. This just goes to show that these distributions can wear many hats and will often have many distinct back stories.

#### 4.2.5.1 Exercises

1. Explore the output of `rnbinom()`. In addition to the standard `n`, you should play around with the two `rnbinom()` unique parameters relevant to the generative model laid out above: `size` (number of successes) and `prob` (probability of a success).
2. Create your own `rnbinom()` simulator using `rbinom()` and some custom R code. Compare the properties of your simulator and `rnbinom()` to ensure things are working as expected.

### 4.2.6 Sampling without replacement: the hypergeometric distribution

The last major discrete distribution to discuss is the hypergeometric distribution. Every single distribution we have looked at thus far has an assumption of event-to-event independence. That is, no matter what has happened previously, the probability of future events is unwaivering. Can you imagine any case where this might not be true?

The simplest way to violate this assumption is to consider the process of "sampling without replacement". Say you have a bag of marbles of two colors, black and red. Now imagine that you draw one from the bag, assess its color. What distribution describes the number of black marbles you see? If each time you draw a marble, you subsequently return it to the bag, then the number of black marbles would be well modeled as binomially distributed. This would be called "sampling with replacement", and it yields independence between each draw. The contents of the bag never changes so neither do the probabilities of a particular result. What if you **DIDN'T** return each marble though? This would be called "sampling without replacement", and now the result of the last draw matters, because it affects how likely each outcome is in the next draw. Describing the number of black marbles in this case requires a new distribution: the hypergeometric distribution.

#### 4.2.6.1 Exercise

1. Explore the output of `rhyper`. Its parameters are `nn` (what is referred to as `n` in all of the obther `r<dist>()` functions), `m` (think of this as the number of red marbles), `n` (think of this as the number of black marbles), and `k` (think of this as the number of marbles drawn from the bag).

**!!END OF LECTURE 2!!**

## 4.3 Continuous random variables

Up until now, we have focused on describing "discrete randomness". This means that outcomes of the processes that we considered had to be well described with integers (0, 1, 2, …). Not everything in the world fits this description though. Consider the following types of data/processes we could imagine modeling:

1. The heights of college-aged students
2. The probability of contracting COVID
3.

In these cases, the outcome is best described using a real number, that is a number which can have an arbitrary number of decimal places. We refer to these as "continuous random variables", and in this lecture we will familiarize ourselves with the most popular distributions for modeling such processes.

### 4.3.1 Pure randomness: the uniform distribution

What's the first thing that comes to mind when you hear that something is "random"? At this point, you should be conditioned to start thinking about the underlying generative model and the distribution that could describe said randomness, but what if you had been asked about randomness embarking on this class? I would argue that for most people, the default definition of randomness is "pure randomness": every possible event is equally likely. While this class should ensure this is no longer your default, there are plenty of processes that are well described by this sort of randomness. For those cases, we have the uniform distribution.

**Generative model**: Imagine the output of a process can be described as a real number between a and b. If all values between a and b are equally likely, then this process' output will be well modeled with a uniform distribution

#### 4.3.1.1 Exercise

1. Explore the output of `runif()`. It has 3 parameters of note. The first is the same for all RNGs and is called `n`. It represents the number of random numbers it spits out. The other two are called `min` and `max`. Both of these can be any number you want, as long as `min` <= `max`. Generate some random numbers with `runif` and observe there properties.

### 4.3.2 Generalizing the uniform distribution

#### 4.3.2.1 Exercises

1. Explore the output of `rbeta()`. Check out `?rbeta()` to see what parameters exist.
2. Combine `rbeta()` and `rbinom()`, using the former to simulate values of `p` for the latter. Compare this to `rbinom()` with `prob = shape1 / (shape1 + shape2)`. How are they similar? How do they differ?

### 4.3.3 Modeling the time until an event: the exponential distribution

#### 4.3.3.1 Exercises

1. Explore the output of `rexp()`. Check out `?rexp()` to see what parameters exist.
2. Compare the following distributions:

- The full output of a run of `rexp()` with a large `n`, like 100,000
- That same output, but subtracting the first quartile from all of the values, and throwing out any negative values. You can find what the first quartile is with `quartile()`.

1. Do the same thing as in 2 but with the output of `rbeta()`. Do you get the same result as in 2?
2. Do the same thing as in 2 and 3 but with the output of `rgeom()`. Do you get the same result as in 2 or as in 3?

Exercises 2-4 explore the property known as memorylessness, which is only possessed by two distributions in the entire universe of distributions. This property in some sense defines these two distributions.

### 4.3.4 Generalizing the exponential distribution: the gamma distribution

#### 4.3.4.1 Exercises

1. Compare the output of `rgamma()` with `shape = 1` to that of `rexp()`. Make some plots to convince yourself that these are the same.
2. Explore how changing the `shape` parameter affects how the output of `rgamma()` differs from that of `rexp()`.

### 4.3.5 All distributions lead here: the normal distribution

#### 4.3.5.1 Exercises

1. Choose your favorite continuous distribution to this point. Use its associated RNG to follow these steps:

- a) Generate N samples from your distribution of choice with whatever parameters you desire.
- b) Calculate the average of those N samples with `mean()`
- c) Repeat a) and b) M times and save the result of b) each time. You should use a `for` loop for this.
- d) Plot a histogram of the sample means.
- e) Compare your plot to a histogram from `rnorm(M, mean = E, sd = S)`, where E = mean(means) and S = sd(means).
- f) Explore the impact of varying N, M, and the parameters of your distribution of choice on the plots generated in e).

**!!END OF LECTURE 3!!**

## 4.4 Problem set: Simulating data with distributions

### 4.4.1 RNA-seq data

Try and simulate a count matrix that has similar properties to that of a real RNA-seq count matrix. This is a fairly open-ended exercise and is meant as an exercise in thinking carefully about modeling data with probability distributions. Compare your simulated data to the real thing, making some plots to assess their similarity.

### 4.4.2 Poisson Process

This exercise will teach you how to use what you have learned to simulate what is known as a Poisson process. The strategy employed here is known as Gillespie's algorithm, and is widely used in the computational modeling of biochemical reactions.

Implement a simulation of RNA transcription following these steps:

1. Specify two parameters at the top of your code:

- T: the length of time for which to simulate.
- rate: The rate at which RNA molecules are synthesized.

2. Set a couple variables that will change throughout the simulation:

- current_t: the current time in the simulation. Set to 0.
- RNA_cnt: the number of RNAs that have been produced. Set to 0.

3. In a while loop, add the output of `rexp(n = 1, rate = rate)` to `current_t`. If the new value of current_t is > T, break out of the while loop. If not, then add one to `RNA_cnt`.

Explore some properties of your simulation:

1. Make a plot of the number of RNAs as a function of time. Make this plot with three different values of `rate`.
2. Run the simulation multiple times with a given `T` and `rate`, and compare the distribution of `RNA_cnt`'s to that of a Poisson distribution with lambda = `rate`. Are they the same? You are discovering why this is called a Poisson process.

**EXTRA CREDIT**

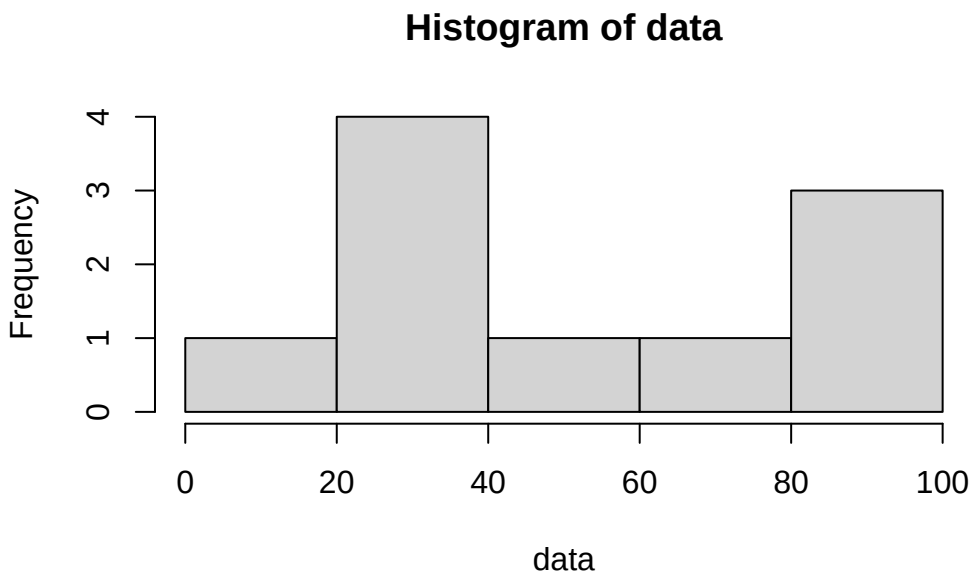
Simulate RNA synthesis and degradation, making a plot of the amount of undegraded RNA as a function of time. Some things you will need to know:

1. If you have multiple processes whose time until the next event is exponentially distributed, the time until any of these events occurs is also exponentially distributed, with rate = sum of the rates of all of the individual exponential distributions.
2. If you have multiple processes whose time until the next event is exponentially distributed, the probability that event i is the next event = $rate_i / \sum rate_j$, where $rate_i$ represents the rate parameter for the ith events exponential distribution, and the sum is over the rates of all of the processes' associated exponential distributions.

## Appendix: Probability Distributions

When I claim that "there is randomness in your data", what does that mean? For some people, the term "randomness" implies complete unpredictability. Such people would interpret my claim to mean that every time you collect a new replicate, any value for the thing you are measuring is fair game and equally likely. "You got 100 reads from the MYC gene in your last RNA-seq dataset? Well don't be surprised if you get 1000, or 10000, or 0 reads next time!" You may call this **uniform randomness**. You can generate such data right here in R, using the `runif()` function:

```
data <- runif(n = 10, min = 0, max = 100)

hist(data)
```



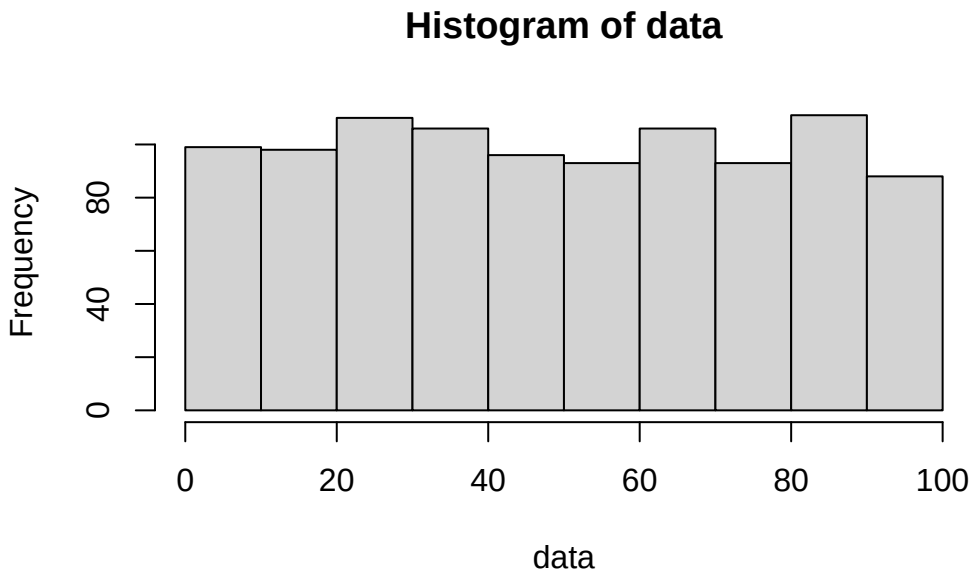

```
# Check out the individual data points
print(data)
```

```
 [1] 40.49390 14.24254 97.05943 86.65046 22.61239 88.42228 23.27201 72.23944
 [9] 29.36827 34.64233
```

`runif()` has three parameters: 1) `n` specifies the number of numbers to generate, 2) `min` specifies the minimum number it could possibly generate, and 3) `max` specifies the maximum

number it could possibly generate. In the above example, I am thus creating 10 numbers between 0 and 100, with every number in between being equally likely to pop out. Generate a lot more numbers and this uniform pattern of appearance becomes much more clear:

```
data <- runif(n = 1000, min = 0, max = 100)

hist(data)
```




While this definition of randomness is intuitive, it can't be the only type of randomness. If RNA-seq data were this random, it would be useless! There is nothing to learn from measurements that can take on any value with equal probability.

Thus, to describe all of the kinds of randomness we see in the real world, it is important to expand our definition beyond uniform randomness. Enter the **probability distribution**. A probability distribution is like a function in R. It takes as input a number (or maybe a set of numbers), and provides as output, the probability of seeing that number. For uniformly random data this might look something like:

```
uniform_distribution <- function(data, min = 0, max = 100){

  if(data >= min & data <= max){

    output <- 1
```

```
  }else{

    output <- 0

  }

  return(output)

}
```

That is to say, as long as the data is within the bounds of what is possible, it has the same probability of occurring; you get the same number out from this function. This function would make mathematicians

# 5 Introduction to Statistical Modeling

The challenge we are faced with when analyzing RNA-seq data, or any data for that matter, is to draw conclusions in a way that accounts for the randomness inherit in our data. In the last chapter, we introduced a number of tools, i.e., probability distributions, for describing common types of randomness. Once we have a sense of how to describe the variance in our data, i.e., the combination of probability distributions that represent good "models" of our data, how do we put these models to use? The answer to this question is known as "fitting the model to your data", and is the topic of this chapter.

## 5.1 Developing an intuition for model fitting

This exercise will walk you through how to think about what it means to "fit" a model. The key tool in our arsenal is going to be simulation, where we create data that follows known distributions. Your task, if you are willing to accept it, is to figure out how to recover the parameters you used in your simulation from the noisy data generated by the simulator. Are you ready? I'll start with a guided exercise, and then present some open-ended exercises to get you exploring model fitting.

### 5.1.1 Guided exercise

**Step 0: Devising a model**

Let's say you would like to estimate what fraction of the nucleotides in the actual RNA population are purines (Gs and As). Each nucleotide in an RNA can either be a purine or a pyrimidine. How would you go about answering this question?

**Step 1: Simulate some data**

For this exercise, I am going to simulate binomially distributed data, and devise a strategy to figure out what `prob` I used in this simulation. Obviously, I will know what `prob` I used, but in the real world, you won't have access to the true parameters of the universe. Knowing the truth helps us know if we are on the right track though, and is why simulations are such a useful tool:

```r
# Set the seed so you can reproduce my results
set.seed(42)
simdata <- rbinom(100, 100, 0.5)
```

Technically, there are two parameters that I had to set here, `size` and `prob`. In this case, I am going to assume that `size` is data I have access to. Usually, if we are modeling something as binomially distributed, we will know how many trials there were.

**Step 2: Visualize your data**

Let's see what the data looks like:

```r
library(ggplot2)
library(dplyr)
```

```
Warning: package 'dplyr' was built under R version 4.3.2
```

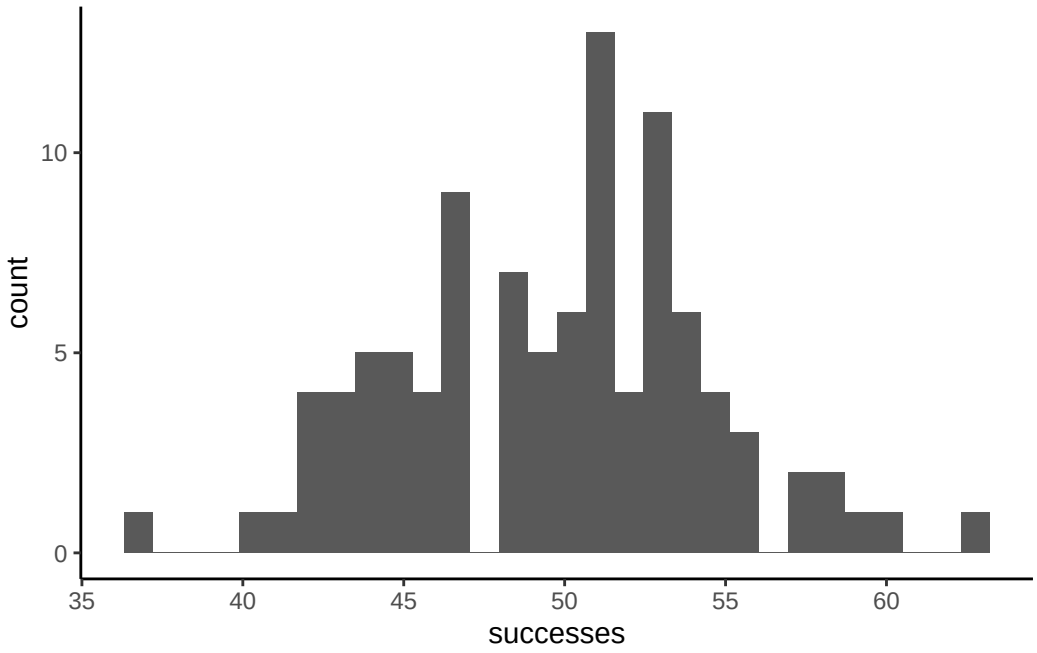
```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':

    filter, lag
```

```
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```r
sim_df <- tibble(successes = simdata)

sim_plot <- sim_df %>%
  ggplot(aes(x = successes)) +
  geom_histogram() +
  theme_classic()
sim_plot
```

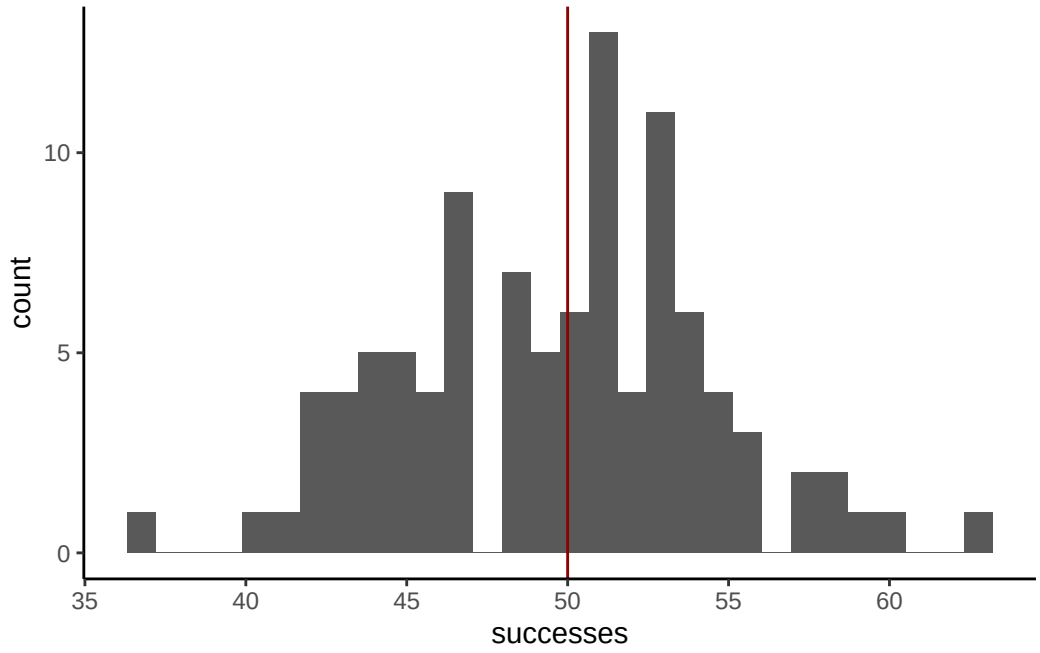
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




Does the data look at all surprising? We set `prob` equal to 0.5, so on average, we expect 50% of the trials to end in successes. If we perfectly hit this mark in our finite sample, that would mean 50 successes. Let's annotate this mark on the plot

```
sim_plot +
  geom_vline(xintercept = 50,
             color = 'darkred')
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




It's a bit noisy, but the data does seem to be roughly centered around 50. That's a good sign that our simulation worked, but now we need to think about how we would have figured out that the `prob` in this case was 0.5

**Step 3: Fight simulations with simulations**

Our goal is to find a binomial distribution `prob` parameter that accurately describes this data. An intuitive way to think about doing this is to simulate data with candidate values for `prob`, and see how similar the simulated data is to the real data. Here's how you might do that:

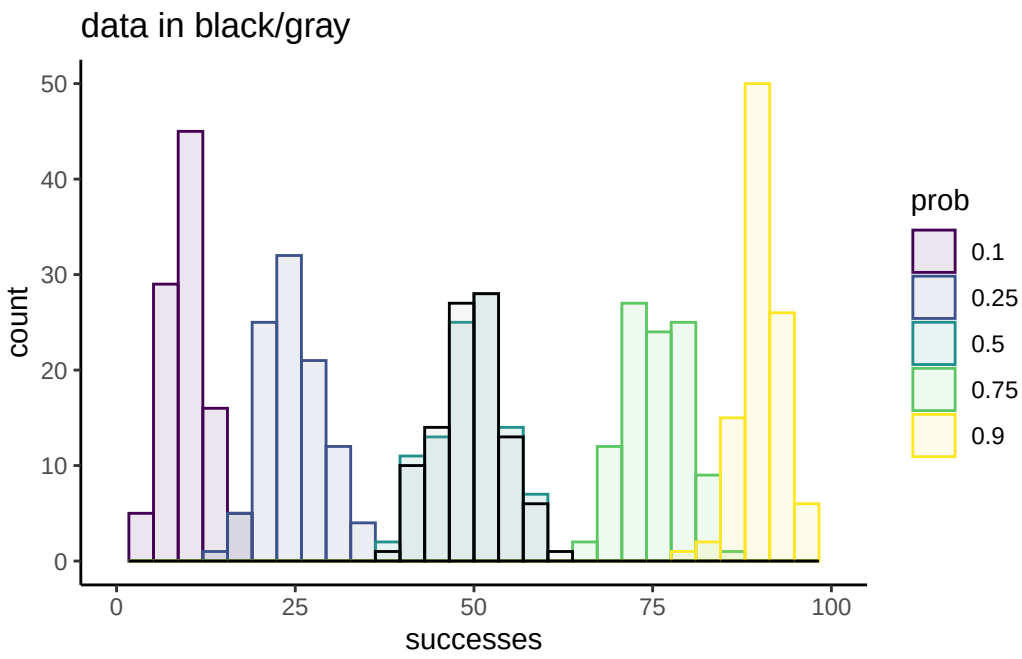
```
### With a for loop
library(tidyr)
candidates <- c(0.1, 0.25, 0.5, 0.75, 0.9)

sim_list <- vector(mode = "list",
                   length = length(candidates))
for(c in seq_along(candidates)){

  sim_list[[c]] <- rbinom(100, 100, candidates[c])

}
names(sim_list) <- as.factor(candidates)
sim_df <- as_tibble(sim_list) %>%
  pivot_longer(names_to = "prob",
```

```
                values_to = "successes",
                cols = everything())

sim_df %>%
  ggplot(aes(x = successes,
             fill = prob,
             color = prob)) +
  geom_histogram(alpha = 0.1,
                 position = 'identity') +
  geom_histogram(data = tibble(successes = simdata,
                               prob = factor('data')),
                 aes(x = successes),
                 color = 'black',
                 alpha = 0.1,
                 fill = 'darkgray') +
  scale_fill_viridis_d() +
  scale_color_viridis_d() +
  theme_classic() +
  ggtitle('data in black/gray') +
  xlim(c(0, 100))
```

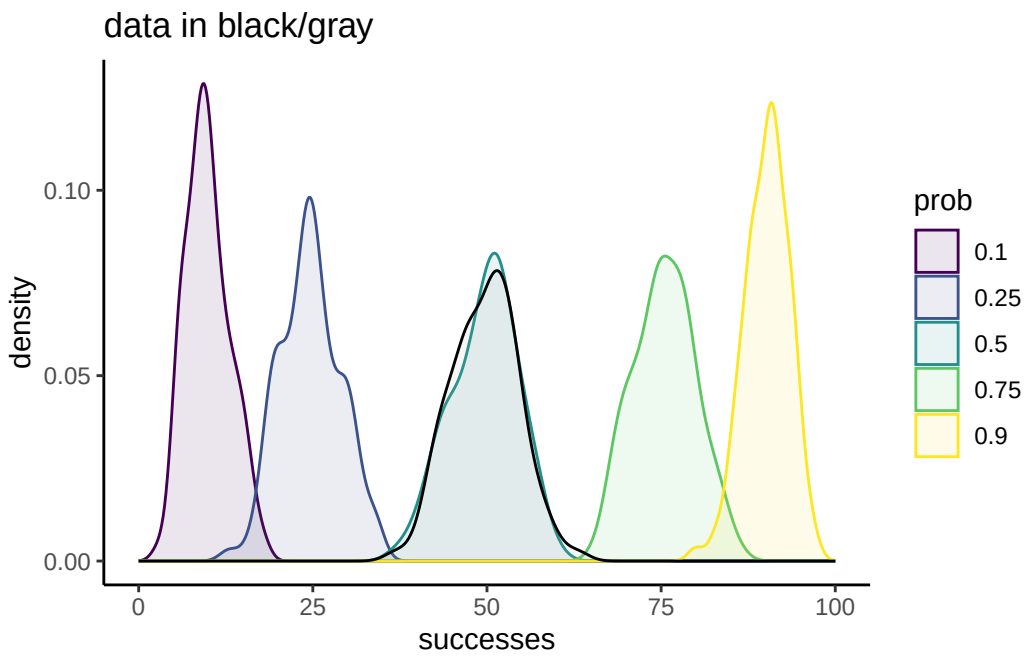
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
Warning: Removed 10 rows containing missing values (`geom_bar()`).
```

```
Warning: Removed 2 rows containing missing values (`geom_bar()`).
```




```
sim_df %>%
  ggplot(aes(x = successes,
             fill = prob,
             color = prob)) +
  geom_density(alpha = 0.1,
                position = 'identity') +
  geom_density(data = tibble(successes = simdata,
                             prob = factor('data')),
                aes(x = successes),
                color = 'black',
                alpha = 0.1,
                fill = 'darkgray') +
  scale_fill_viridis_d() +
  scale_color_viridis_d() +
  theme_classic() +
  ggtitle('data in black/gray') +
  xlim(c(0, 100))
```




Of these small subset of candidate `prob` values, 0.5 is the clear winner. The overlap of the simulation with the data is striking, and strongly suggests that this is a good fit to our data.

Of course, we know the true value is 0.5, and this knowledge guided our choice of candidates. So let's explore a different range of candidates, all of which are much closer to the known truth:

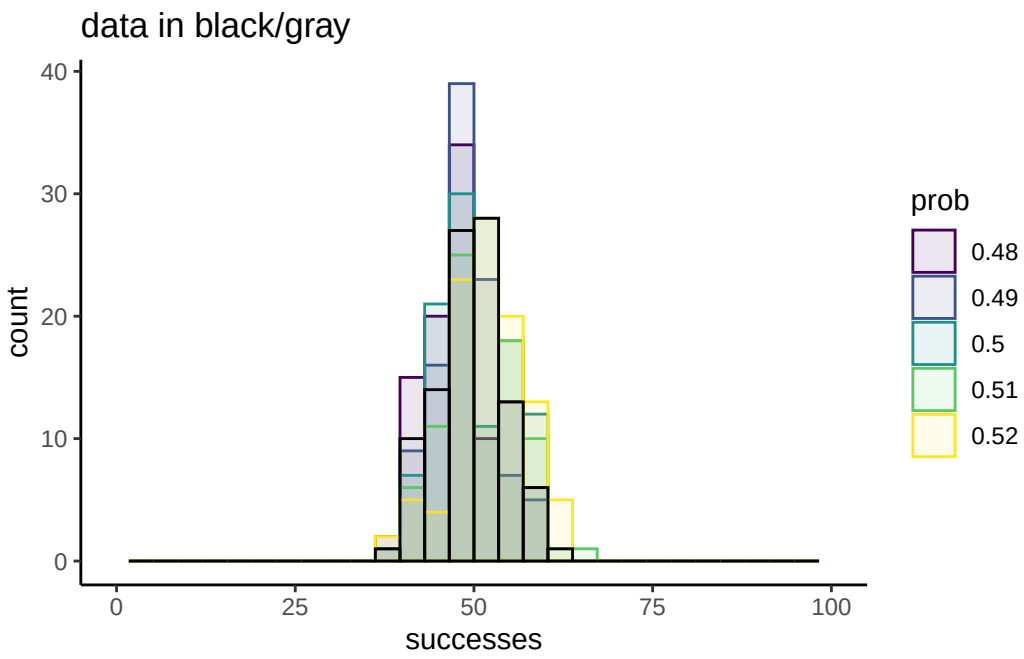
```
### With a for loop
library(tidyr)
candidates <- c(0.48, 0.49, 0.5, 0.51, 0.52)

sim_list <- vector(mode = "list",
                   length = length(candidates))
for(c in seq_along(candidates)){

  sim_list[[c]] <- rbinom(100, 100, candidates[c])

}
names(sim_list) <- as.factor(candidates)
sim_df <- as_tibble(sim_list) %>%
  pivot_longer(names_to = "prob",
               values_to = "successes",
               cols = everything())
```

```r
sim_df %>%
  ggplot(aes(x = successes,
             fill = prob,
             color = prob)) +
  geom_histogram(alpha = 0.1,
                 position = 'identity') +
  geom_histogram(data = tibble(successes = simdata,
                               prob = factor('data')),
                 aes(x = successes),
                 color = 'black',
                 alpha = 0.1,
                 fill = 'darkgray') +
  scale_fill_viridis_d() +
  scale_color_viridis_d() +
  theme_classic() +
  ggtitle('data in black/gray') +
  xlim(c(0, 100))
```

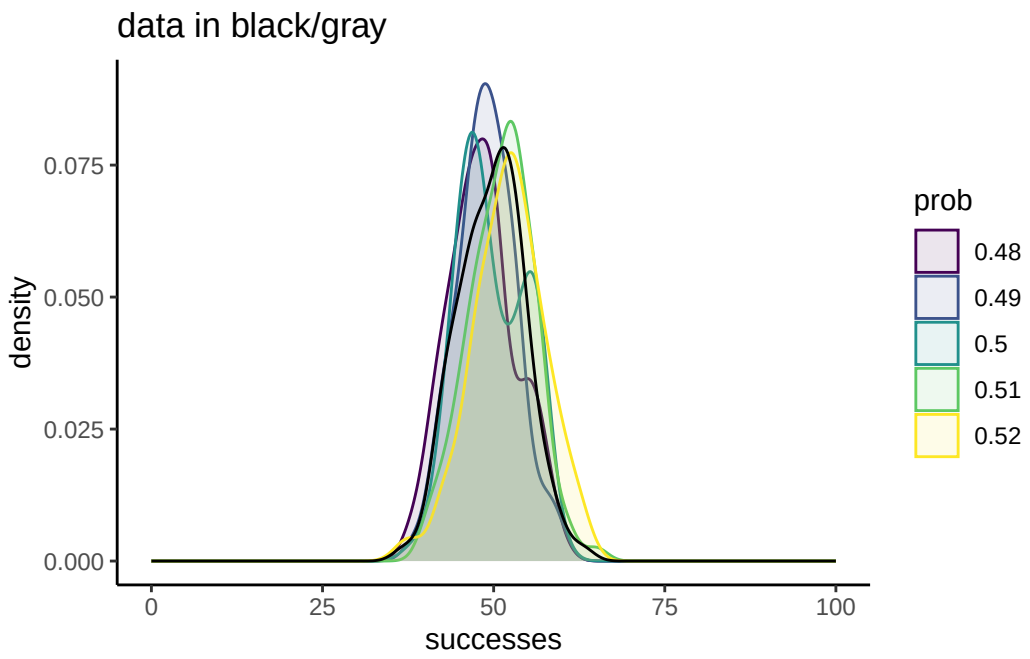
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
Warning: Removed 10 rows containing missing values (`geom_bar()`).
```

```
Warning: Removed 2 rows containing missing values (`geom_bar()`).
```




```
sim_df %>%
  ggplot(aes(x = successes,
             fill = prob,
             color = prob)) +
  geom_density(alpha = 0.1,
                 position = 'identity') +
  geom_density(data = tibble(successes = simdata,
                             prob = factor('data')),
                 aes(x = successes),
                 color = 'black',
                 alpha = 0.1,
                 fill = 'darkgray') +
  scale_fill_viridis_d() +
  scale_color_viridis_d() +
  theme_classic() +
  ggtitle('data in black/gray') +
  xlim(c(0, 100))
```




Now the comparisons are much messier. Sure, 0.5 is a great match, but none of these chosen values yield simulations too different from the truth.

From this, it is clear that there are ranges of values that we can confidently rule out as good conclusions for what `prob` best describes our data. Values of 0.25 or less, and values of 0.75 or more are very bad fits. There is also good evidence that a good fit is somewhere around 0.5, but the exact value that would represent the best guess is uncertain.

**Step 4: Make fitting more rigorous**

The current strategy we have employed could be referred to as "simulations and vibes". We have simulated data with some values for the unknown parameter in question, and assessed by eye how close our simulated data matched our real data. Whlie this has offered some valuable insights, it's important to recognize the fundamental limitations of such an approach:

1. Each simulation with a given parameter value will yield different results. Random number generators are going to be random. One output of `prob = 0.5` might look exactly like the real think, but so might one value of `prob = 0.48`.
2. It's time intensive. You have to simulate data for a bunch of different values, and then painstakingly stare at cluttered plots trying to make sense of which simulation fit your data the best.
3. It's subjective. None of our simulations matched the data exactly, and even if one run of a particular simulation did, see point 1 for why that can't be considered conclusive evidence in favor of that parameter being the best. Lacking such a perfect match, we are forced to rely on ill-defined, self-imposed criteria for what the best fit is.

How can we fix this problem? What we need is a quantitative metric, a number that we can assign to every possible value of `prob` that describes how good of a fit it is to our data. It should have the following properties:

1. It should be deterministic; a given value of `prob` should yield a single unique value for this metric.
2. Higher values should represent better fits.
3. The more data we have, the better this metric should do at predicting the true parameter value.

Sit and think about this for a while (maybe a couple decades, which is what it took the field to converge on this solution), and you'll arrive at something called the **"total likelihood"**. In this part of the exercise, we'll develop an intuition for this concept:

#### 5.1.1.1 What is the likelihood?

When discussing a particular distribution, we have playing with its associated random number generator function in R. For the binomial distribution, this is the `rbinom()` function. If you check the documentation for `rbinom()` though, with `?rbinom`, you'll see that you actually get documentation for 4 different functions, all with a similar naming convention. We'll find use for all of these at some point in this class, but for now, let's focus on one that we briefly visited last week, `dbinom()`.

Unlike `rbinom()`, `dbinom()` returns the same value for a given set of parameters every single time. It is not random:

```
dbinom(50, 100, 0.5)
dbinom(50, 100, 0.5)
```

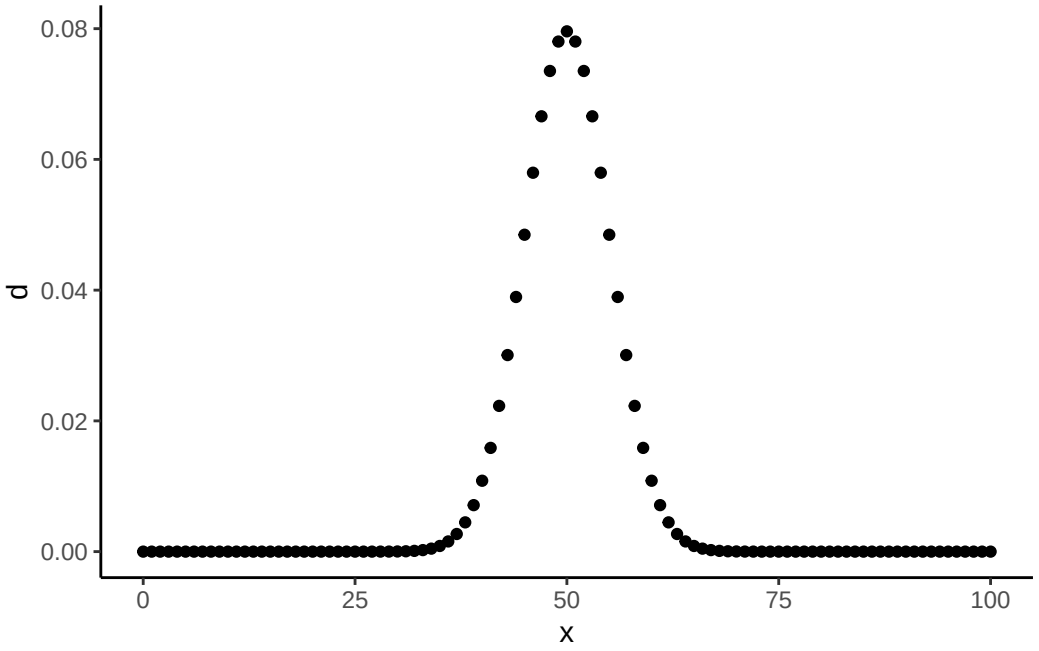
```
[1] 0.07958924
[1] 0.07958924
```

What does this value represent though. First, let's understand its input:

1. `x`: The number of "successes". So essentially what you could get out from a given run of `rbinom()`.
2. `size`: Same as in `rbinom()`, the number of trials.
3. `prob`: Same as in `rbinom()`, the probability that each trial is a success.

So it seems to quantify something about a particular possible outcome of `rbinom()`. It assigns some number to an outcome given the parameters you could have used to simulate such an outcome. What does this number represent though? To find out, let's plot it for a range of `x`'s:

```r
xs <- 0:100
ds <- dbinom(xs, 100, 0.5)

tibble(x = xs,
       d = ds) %>%
  ggplot(aes(x = x, y = d)) +
  geom_point() +
  theme_classic()
```




Values closer to 50 get higher numbers than values further from 50, and the plot seems to be symmetric around 50. What's significant about 50? It's

$$size * prob$$

, or the average value we would expect to get from `rbinom()` with these parameters!

This investigation should give you some sense that the output of `dbinom()` is in some way related to the probability of seeing `x` given a certain `size` and `prob`. Is it exactly this probability? We can gut check by assessing some cases we know for certain. Like, what is the probability of seeing 1 success in 1 trial if `prob` = 0.5? 0.5, because that's the defintiion of prob! What does `dbinom()` give us in this scenario:

```
dbinom(1, 1, 0.5)
```

```
[1] 0.5
```

0.5; that's a good sign that we are on to something. Try out some different values of `prob`:

```
dbinom(1, 1, 0.3)
```

```
[1] 0.3
```

```
dbinom(1, 1, 0.7)
```

```
[1] 0.7
```

```
dbinom(1, 1, 0.2315)
```

```
[1] 0.2315
```

Everything still checks out. How about the probability of 2 successes in 2 trials given a certain `prob`? Each trial has probability of `prob` of being a success, and each trial is independent. Therefore, the probability of 2 successes in 2 trials is

$$prob * prob$$

. Does `dbinom()` give us the same output in that case?

```
# Expecting 0.25
dbinom(2, 2, 0.5)
```

```
[1] 0.25
```

```
# Expecting 0.01
dbinom(2, 2, 0.1)
```

```
[1] 0.01
```

Sure thing!

**Conclusion**: `dbinom(x, size, prob)`, tells us the probability of seeing `x` successes in `size` trials given the probability of a success is `prob`. This is known as the "likelihood of our data".

#### 5.1.1.2 Calculating the total likelihood

In the above examples, we took a single value of x, and passed it `dbinom()`. This told us the probability of seeing that one value given whatever parameter values we set. In most cases though, we typically have mulitple replicates of data. How can we calculate the likelihood of our entire dataset?

I'll pose a solution, and then demonstrate why the solution makes sense. The solution is to run `dbinom()` on each data point, and multiply the outcomes. This can be done like so:

```
data <- c(48, 50, 49, 50)
likelihood <- prod(dbinom(data, size = 100, prob = 0.5))
```

Why does this make sense? Consider the examples we went through above considering cases where `size` was 1 or 2. If we have two datasets with a single trial each, that's like having one dataset with two trials. Thus, the likelihood for the full dataset of two single trial data points must be the same as that of a single dataset with one two trial data point. In other words:

```
datapoint1 <- c(1)
datapoint2 <- c(0)
dataset <- datapoint1 + datapoint2

dataset_L <- dbinom(dataset, size = 2, prob = 0.5)

calc_total_L <- function(datapoints, size = 2, prob = 0.5){

  ### Let's do what I suggested above: multiply the values
  return(prod(dbinom(datapoints, size = size, prob = prob)))

}

total_L <- calc_total_L(c(datapoint1, datapoint2), size = 1, prob = 0.5)

# We need this to be TRUE if our calculation is to be believed
total_L == dataset_L
```

```
[1] FALSE
```

Multiplying the likelihoods of the individual data points gives us the same thing as the likelihood for the dataset generated from combining the results of the two individual trials! This is anecdotal, but we can formalize why this is true, and when it will be true:

**Conclusion**: If each of a collection of N datapoints is "independent", then the total likelihood of the dataset is equal to the product of the individual datapoint likelihoods. Datapoints being independent means that the value of any given datapoint does not influence the values of any other datapoint.

**COMPUTATIONAL ASIDE**

Above, we calculate the total likelihood. One problem that we will often run into with this calculation is that the product of all of the individual data point likelihoods will be really small for some potential parameter values. In this case, our computer might just assign these a value of 0, even though the true likelihood is non-zero. This is called an "overflow error", and can mess with some of the things we do later in this course.

For this reason, we will instead calculate the "log-total-likelihood". This is just the logarithm of the total likelihood. Why does this matter? All of the `d<distribution>()` functions have one additional parameter named `log`, that can be set to either `TRUE` or `FALSE`. Setting it to `TRUE` will lead to it outputting the logarithm of what it would otherwise output.

In addition, the log of a product is the same as the sum of logs:

```
numbers <- rbinom(100, size = 100, prob = 0.5)
ll_one_way <- log(prod(dbinom(numbers, 100, 0.5)))
ll_other_way <- sum(dbinom(numbers, 100, 0.5, log = TRUE))

ll_one_way == ll_other_way
```

```
[1] TRUE
```

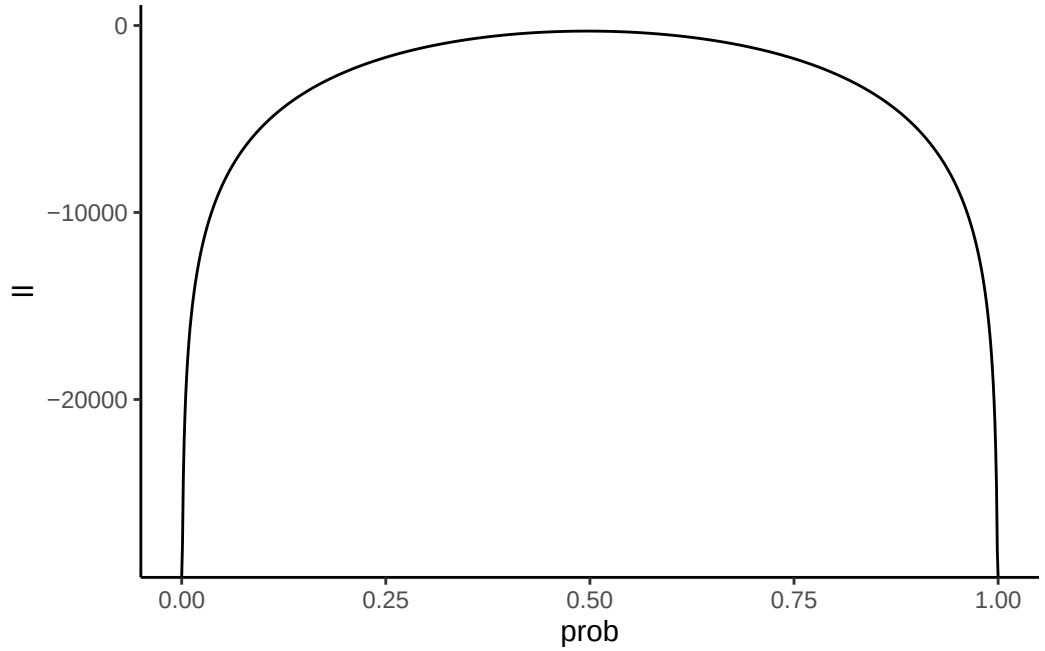
So it is almost always preferable to sum the log-likelihoods of individual datapoints than it is to multiply their regular likelihoods.

#### 5.1.1.3 How can we use the total log-likelihood to fit models?

The total log-likelihood of our data turns out to be a great metric by which determine model fit. The idea is to find the parameter value that gives you the largest likelihood. That's to say, we choose as our best fit the parameter value(s) that make our data as probabilistically reasonable as possible. This is known as the maximum likelihood estimates for our parameter(s). If we simulated data using this value, our simulated data would on average come closer to looking like our real data than it would for any other parameter values we could have simulated with.

How does this play out with our data? We can visualize the so-called log-likelihood function to see which value of `prob` maximizes it:

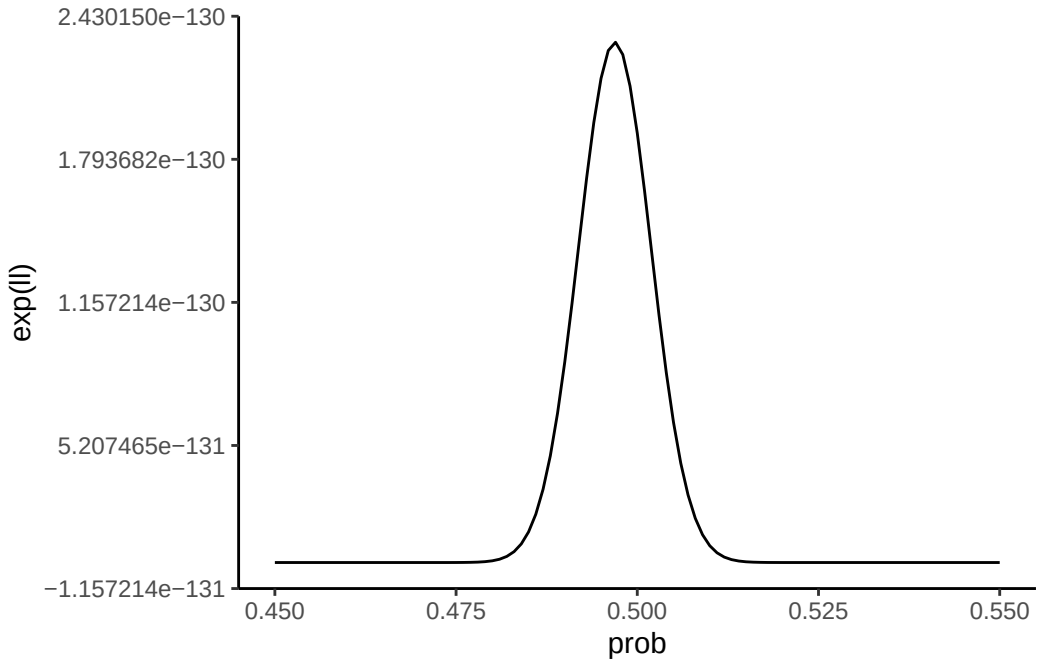
```r
prob_guesses <- seq(from = 0, to = 1, by = 0.001)
log_likelihoods <- sapply(prob_guesses, function(x) sum(dbinom(simdata,
                                                               size = 100,
                                                               prob = x,
                                                               log = TRUE)))

Ldf <- tibble(ll = log_likelihoods,
              prob = prob_guesses)

Ldf %>%
  ggplot(aes(x = prob, y = ll)) +
  geom_line() +
  theme_classic()
```




Keep in mind the y-axis is a log scale, so differences of 1 equate to an order of magnitude difference on the "regular" scale. Let's zoom in around our putative values to get a better look at which value of `prob` is most likely:

```r
zoomed_plot <- Ldf %>%
  filter(prob >= 0.45 & prob <= 0.55) %>%
  ggplot(aes(x = prob, y = exp(ll))) +
  geom_line() +
```

```
  theme_classic()
zoomed_plot
```

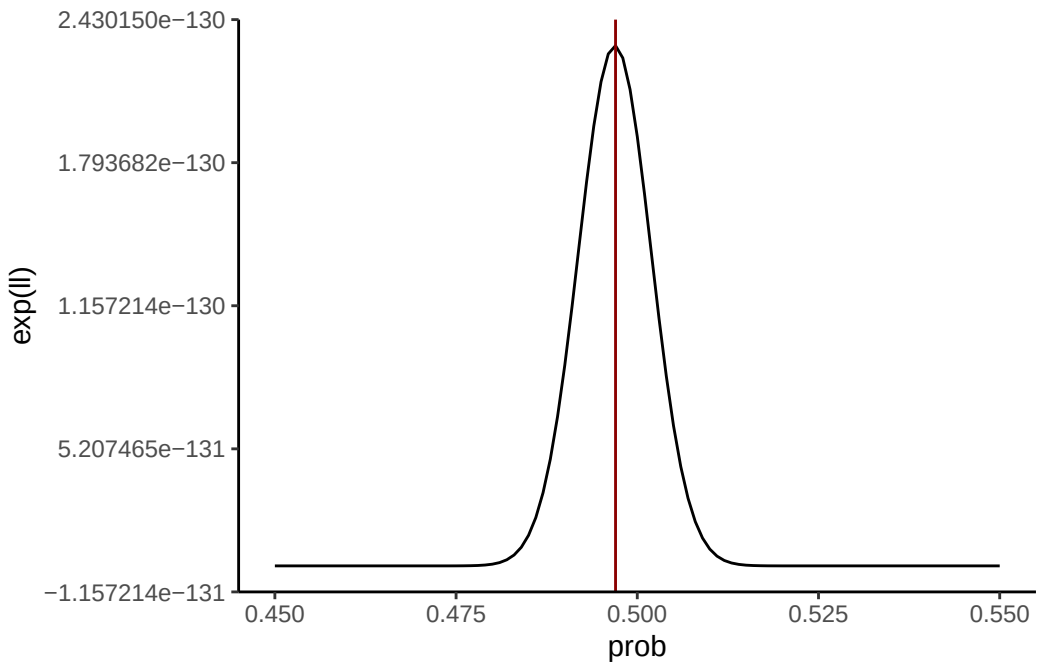



It looks like a value of `prob = 0.5` is not strictly speaking the best guess. Let's see what value in our grid of `prob_guesses` gives the highest value, and annotate this value on our zoomed in plot:

```
max_ll <- Ldf %>% filter(ll == max(ll)) %>%
  dplyr::select(prob) %>% unlist() %>% unname()
max_ll
```

```
[1] 0.497
```

```
zoomed_plot +
  geom_vline(xintercept = max_ll,
             color = 'darkred')
```




So our best guess given this metric is a value for `prob` of around 0.497. That's pretty darn close to the true value of 0.5!

#### 5.1.1.4 How we ACTUALLY find the maximum likelihood

While the strategy for finding the maximum likelihood parameter estimate used above is ok for this toy example, we can do a lot better in terms of the speed and generalizability of our computational solution. If we can provide the function that we want to maximize (e.g., the log-likelihood), and the data needed to calculate the value of our function for a given parameter estimate, then R provides us with the tool necessary to find the parameter estimate that maximizes our function. That tool is the `optim()` function.

Here is how you can use `optim()` to find the maximum likelihood estimate of `prob` in our example:

```
### Step 1: Define the function to maximize
binom_ll <- function(params, data, size = 100){

  # We estimate on a log scale for convenience so we convert here
  prob <- params[1]

  # Log-likelihood assuming binomial distribution
  ll <- dbinom(data,
```

```
                size = size,
                prob = prob,
                log = TRUE)

  # Return negative because optim() by default will find the minimum
  return(-sum(ll))

}


### Step 2: run optim()
fit <- optim(
  par = 0.5, # initial guess for prob
  fn = binom_ll, # function to maximize
  data = simdata, # something passed to the function
  method = "L-BFGS-B", # Most efficient method for finding max,
  lower = 0.001, # Lower bound for parameter estimate
  upper = 0.999 # Upper bound for parameter estimate
  )


### Step 3: Inspect the estimate
prob_est <- fit$par[1]
prob_est
```

```
[1] 0.4969
```

**END OF LECTURE 1**

### 5.1.2 Quantifying uncertainty

The key problem that we set up in both this chapter and the last is one of navigating the randomness in our data. We have to analyze our data in a way that is cognizant of this randomness. This means not only estimating things we care about, but also being honest about how confident we are in our estimates.

In the binomial distribution fitting exercise, we arrived at a strategy for finding what we considered the "best" estimate for the `prob` parameter. On that journey though, you may have gotten the sense that there wasn't one definitive parameter estimate. Rather, a range of estimates would have been reasonable given the data. You may have also realized that we didn't fully stick the landing. Our estimate was not exactly equal to the true value of 0.5. It

would thus be best if we didn't only report a single number as our best estimate. We need to also provide some sense of how confident we are that our estimate is right. In this section, we will discuss some of the most common ways to do this.

#### 5.1.2.1 The bootstrap

The boostrap is one of the cutest, somehow legitimate ideas in statistics that is a shockingly accurate way by which to assess uncertainty in your parameter estimates. The idea is to take your original dataset and **sample from it with replacement**. This means creating a new dataset, the same size of the original dataset, which is generated by randomly choosing datpoints from the original dataset until you have chosen N datapoints (N being the number of datapoints in your original dataset). Sampling "with replacement" means that every time you sample a datapoint, you "put it back", so to speak. So each sample is taken from the full original dataset. This ensures that the new dataset you generate has almost no chance of looking exactly like your original dataset. You can then re-estimate the maximum likelihood parameter with each new re-sampled dataset, and see how much variability there is from re-sampling to re-sampling. From this, you can derive measures of uncertainty, like the standard devitaion of your resampled data estimates.

Let's put this into action with our simulated data:

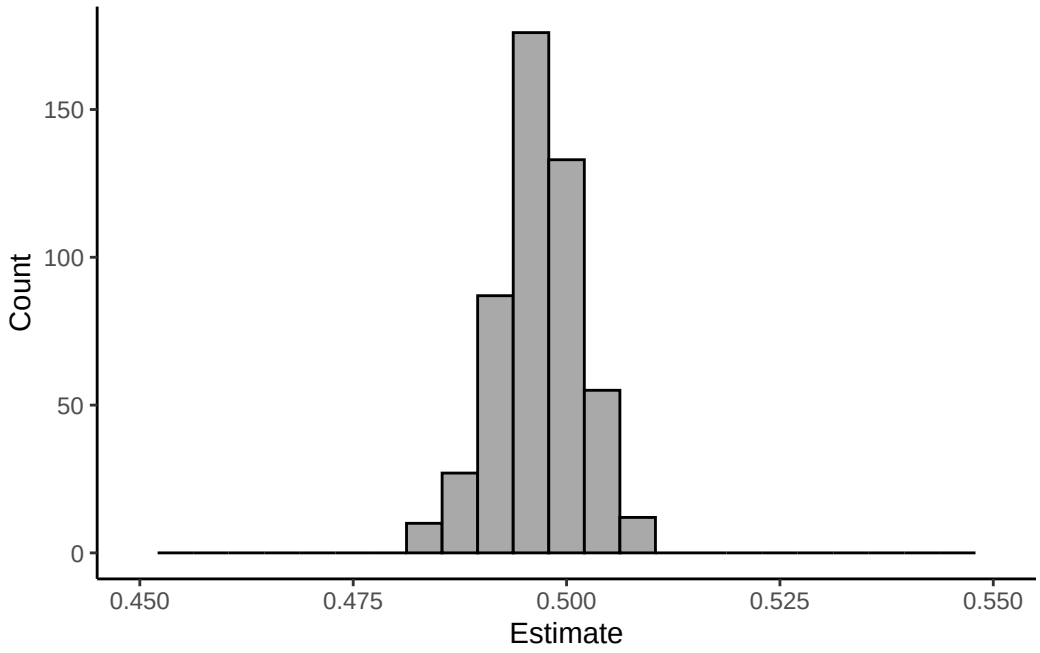
```
nresamps <- 500
estimates <- rep(0, times = nresamps)

for(i in 1:nresamps){

  ### Step 1: resample
  resampled_data <- sample(simdata,
                           size = length(simdata),
                           replace = TRUE)


  ### Step 2: estimate
  fit <- optim(
    par = 0.5, # initial guess for prob
    fn = binom_ll, # function to maximize
    data = resampled_data, # something passed to the function
    method = "L-BFGS-B", # Most efficient method for finding max,
    lower = 0.001, # Lower bound for parameter estimate
    upper = 0.999 # Upper bound for parameter estimate
    )
```

```
  ### Step 3: save estimate
  estimates[i] <- fit$par[1]


}

### Uncertainty
sd(estimates)
```

```
[1] 0.004929947
```

```
### Check out distribution of estimates
tibble(estimate = estimates) %>%
  ggplot(aes(x = estimate)) +
  geom_histogram(color = 'black',
                 fill = 'darkgray',
                 bins = 25) +
  theme_classic() +
  xlab("Estimate") +
  ylab("Count") +
  xlim(c(0.45, 0.55))
```

```
Warning: Removed 2 rows containing missing values (`geom_bar()`).
```




#### 5.1.2.1.1 Exercise:

Rerun this bootstrapping multiple times (e.g., 5) for 3 different values of `nresamp`. How does the size of `nresamp` relate to the output.

### 5.1.2.2 The Hessian

```
### Step 1: run optim()
fit <- optim(
  par = 0.5, # initial guess for prob
  fn = binom_ll, # function to maximize
  data = simdata, # something passed to the function
  method = "L-BFGS-B", # Most efficient method for finding max,
  lower = 0.001, # Lower bound for parameter estimate
  upper = 0.999, # Upper bound for parameter estimate
  hessian = TRUE # Return the Hessian
  )

### Estimate uncertainty
uncertainty <- sqrt(solve(fit$hessian))
```

```
uncertainty
```

```
            [,1]
[1,] 0.004999884
```

#### 5.1.2.3 The full likelihood

**END OF LECTURE 2**

### 5.1.3 A self-guided exercise: fit a negative binomial

Let's make this more RNA-seq relevant now. As we will talk about more throughout this course, one of the most common models for the replicate-to-replicate variability of RNA-seq read counts is a negative binomial distribution. In this case, there are two parameters to estimate. In R, they are referred to as `mu` and `prob`. Simulate some data with a given value for these two parameters, and follow the steps above to find a strategy that consistently returns accurate estimates for the parameter values you simulated with.

### 5.1.4 A 2nd self-guided exercise: fit a normal distribution

**END OF LECTURE**

## 5.2 Problem set

### 5.2.1 Fitting the wrong model

Explore and document what happens when you simulate using one distribution but fit a normal distribution to the data. How do the mean and standard deviation of your data relate to the estimated mean and sd parameters of the normal distribution? Do this with two non-normal distributions of your choice.

# Part II

# Popular Methods

# 6 Hypothesis Testing

The last two chapters discussed how to model randomness in data. In this process, we can obtain parameter estimates that describe some aspect of the biological systems we study. Often, we have hypotheses about what parameter values we expected, or what certain parameter values tell us about our system of interest. How do we go from the output of a statistical model to "conclusions" regarding these hypotheses? One popular (albeit flawed) method for doing so is Null Hypothesis Statistical Testing (NHST) and is the topic of this chapter.

# 7 Discovering NHST

We are going to go through a collection of exercises that will help you discover the concept of NHST on your own. On this journey, you will also become acutely aware of the limitations of this method, and thus be better prepared to carefully interpret the output of an NHST analysis.

## 7.1 Classic example: the fair coin

Imagine you have a coin, and you would like to test the hypothesis, "is this coin fair?". A "fair" coin is one that when flipped, is equally likely to come up heads or tails.

We can cast this question in the language of statistical modeling. If we assume that each coin flip is independent (e.g., the last outcome does not influence the next) and that the probability of heads is unchanged from flip to flip, then we can model the number of heads as following a binomial distribution. We know how many times we flip the coin, so there is one unknown parameter, `prob`, which represents the probability of seeing a head. A fair coin is one in which `prob` = 0.5. In this framing, our task is to determine whether or not the "true `prob`" for our coin is exactly 0.5.

How should we go about answering this question?

### 7.1.1 Step 1: Collect some data

The only way we can begin to tell if the coin is fair or not is to flip it!

# 8 Performing common tests

## 8.1 Z-test

## 8.2 T-test

## 8.3 ANOVA

## 8.4 Non-parametric tests

## 8.5 Custom NSHTing

# 9 Challenges

## 9.1 Multiple-test adjustment

## 9.2 Power

## 9.3 Miscalibration

# 10 Linear Modeling

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

# 11 Dimensionality Reduction

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

# 12 Clustering and Mixture Modeling

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

# Part III

# Statistics in the Wild

# 13 Practical Bioinformatics

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

# 14 Analysis of an RNA-seq dataset

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

# References

Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.