Biology 305: Biostatistics

Dr. Jacob C. Cooper & Dr. Melissa Wuellner

Invalid Date

Table of contents

Preface

Welcome to Biology 305 at the University of Nebraska at Kearney! Material in this class was designed by Dr. Melissa Wuellner and adapted by Dr. Jacob C. Cooper for use in *R*.

In this class, you will learn:

1. The basics of study design, the importance of understanding your research situation before embarking on a full study, and practice creating research frameworks based on different scenarios.
2. The basics of data analysis, including understanding what kind of variables are being collected, why understanding variable types are important, and basic tests to understand univariate distributions.
3. Basic multivariate statistics, including ANOVA, correlation, and regression, for comparing multiple different groups.
4. The basics of coding and working in *R* for performing statistical analyses.

This site will help you navigate different homework assignments to perform the necessary *R* tests. Furthermore, this GitHub repository contains all of the homework dataframes, so you will *not* have to manually enter assignments if you use *R* to complete your assignments.

Welcome to class!

Dr. Jacob C. Cooper, BHS 321

1 Intro to *R*

In this class, we will be using *R* to perform statistical analyses. *R* is a free software program designed for use in a myriad of statistical and computational scenarios. It can handle extremely large datasets, can handle spatial data, and has wrappers for compatibility with *Python*, *Bash*, and other programs (even *Java*!).

2 Setup

First, we need to download *R* onto your machine. We are also going to download *RStudio* to assist with creating *R* scripts and documents.

2.1 Installing *R*

First, navigate to the *R* download and install page. Download the appropriate version for your operating system (Windows, Mac, or Linux). **Note** that coding will be formatted slightly different for Windows than for other operating systems. If you have a Chromebook, you will have to follow the online instructions for installing both programs on Chrome.

Follow the installation steps for *R*, and verify that the installation was successful by searching for *R* on your machine. You should be presented with a coding window that looks like the following:

```
R version 4.4.1 (2024-06-14) -- "Race for Your Life"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

If that screen appears, congratulations! *R* is properly installed. If the install was not successful, please talk to Dr. Cooper and check with your classmates as well.

2.2 Installing *RStudio*

RStudio is a GUI (graphics user interface) that helps make *R* easier to use. Furthermore, it allows you to create documents in *R*, including websites (such as this one), PDFs, and even presentations. This can greatly streamline the research pipeline and help you publish your results and associated code in a quick and efficient fashion.

Head over the the *RStudio* download website and download "*RStudio* Desktop", which is free. Be sure to pick the correct version for your machine.

Open *RStudio* on your machine. You should be presented with something like the following:

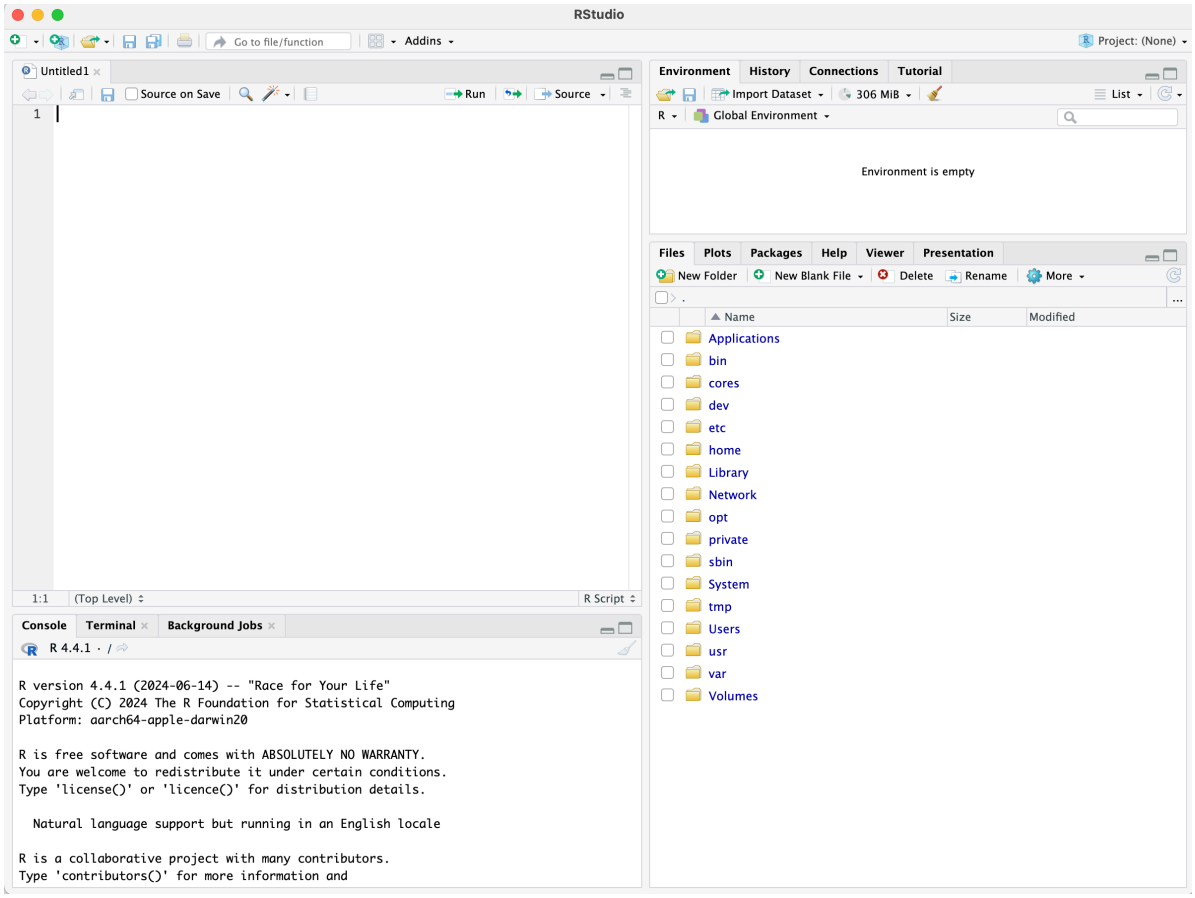


Figure 2.1: *RStudio* start window. Note that the screen is split into four different quadrants. Top left: *R* documents; bottom left: *R* program; top right: environment window; bottom right: plots, help, and directories.

In *RStudio*, the top left window is always going to be our coding window. This is where we will type all of our code and create our documents. In the bottom left we will see *R* executing the code. This will show what the computer is "thinking" and will help us spot any potential issues. The top right window is the "environment", which shows what variables and datasets are stored within the computers' memory. (It can also show some other things, but we aren't concerned with that at this point). The bottom right window is the "display" window. This is where plots and help windows will appear if they don't appear in the document (top left) window itself.

Now, we will create our first *R* document!

3 Creating an *RMarkdown* document

3.1 Setup

In this class, we will be creating assignments in what is called *RMarkdown*. This is a rich-text version of *R* that allows us to create documents with the code embedded. In *RStudio*, click the "+" button in the far top left to open the `New Document` menu. Scroll down this list and click on `R Markdown`.

A screen such as this will appear:

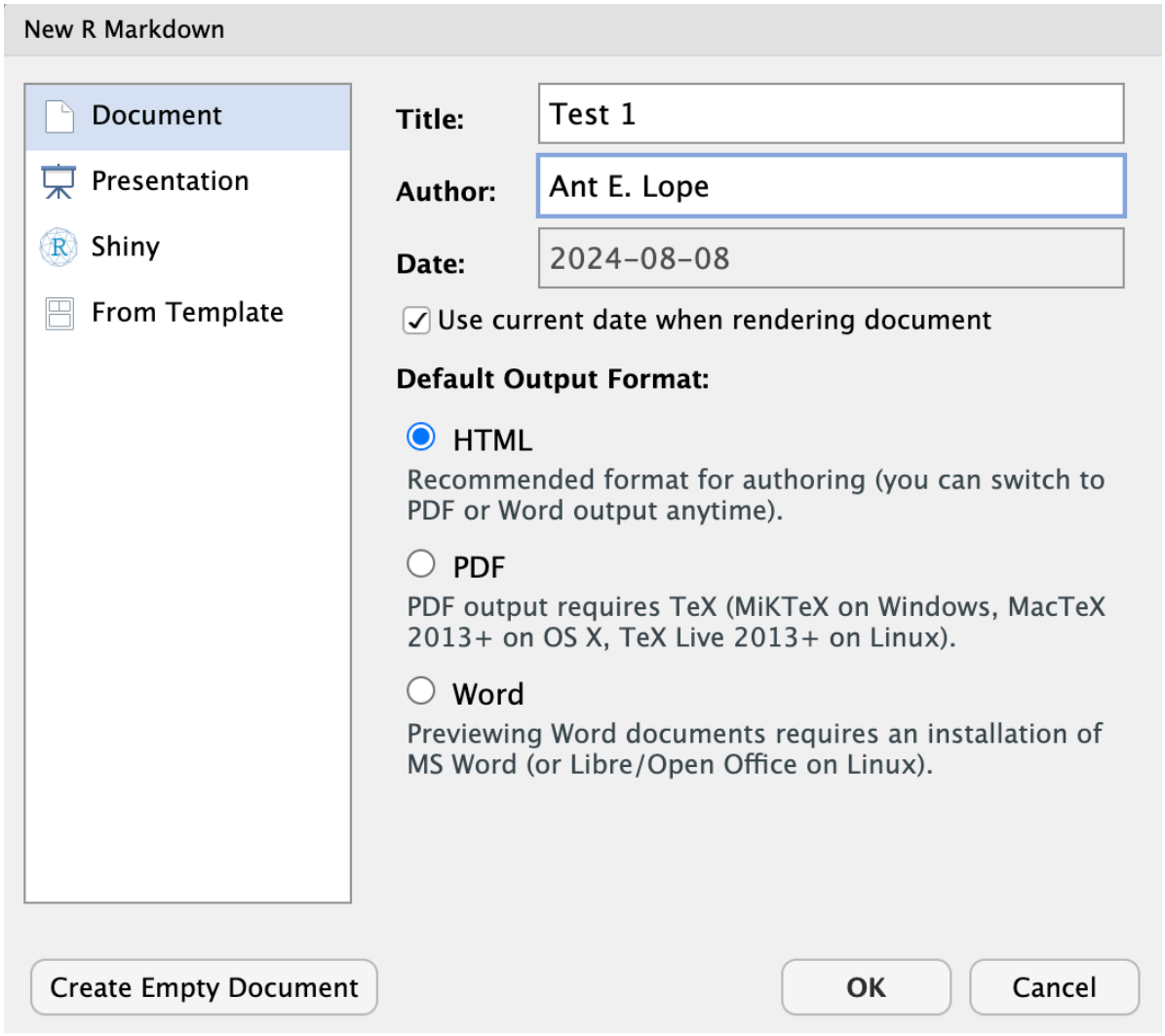


Figure 3.1: A new file window for an *RMarkdown* file.

After entering a title and your name and selecting `document` in the left hand menu, click `OK`.

Figure 3.2: An example of a markdown script.

In the image above, we can see what a "default" *RMarkdown* script looks like after creating the file. At the top of the document, between all of the dashes, we have the `yaml` header that tells *R* what kind of document will be created, who the author is, and tells it to use today's date. In this class, we will be saving documents as `html` as they are the easiest documents to create and save. These documents will include all of your code, text, and even any plots you may create!

Plain text in the document will be rendered as plain text in the document. (I.e., whatever you type normally will become "normal text" in the finished document). Lines preceded with `#` will become headers, with `##` being a second level header and `###` being a third level header, etc. Words can also be made italic by putting an asterisk on each side of the word (`*italic*`) and bold by putting two asterisks on each side (`**bold**`). URLs are also supported, with `<>` on each side of a URL making it clickable, and words being hyperlinked by typing `[words to show](target URL)`.

We also have code "chunks" that are shown above. A code chunk can be manually typed out or inserted by pressing `CTRL` + `ALT` + `I` (Windows, Linux) or `COMMAND` + `OPTION` + `I` (Mac). Everything inside a "code chunk" will be read as *R* code and executed as such. Note that you can have additional commands in the *R* chunks, but we won't cover that for now.

3.2 Using code chunks

In your computer, erase all information except for the `yaml` header between the dashes on your computer. `Save` your file in a folder where you want your assignment to be located. It is important you do this step up front as the computer will sometimes save in random places if you don't specify a file location at the beginning. *Don't forget to save your work frequently*!

````
This is a test of the *RMarkdown* code.

```{r}
print("Hello world!")
```
````

Figure 3.3: Text to type in your *Rmarkdown* document.

After typing this into the document, hit `knit` near the top of the upper left window. *R* will now create an HTML document that should look like this:

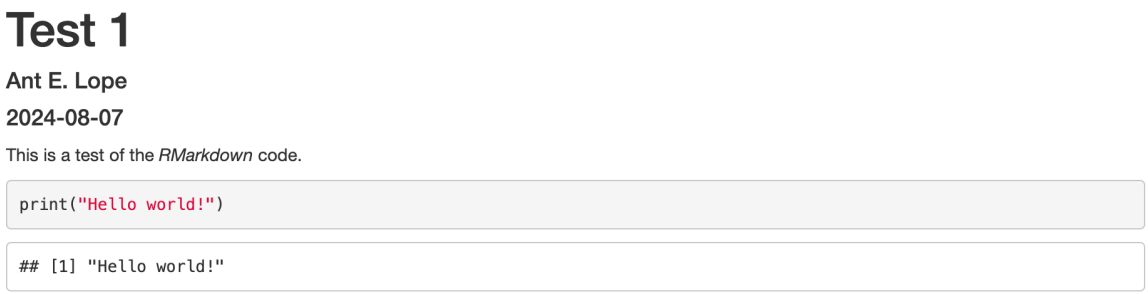


Figure 3.4: The output from the above code knitted into a document.

We can see now that the `HTML` document has the title of the document, the author's name, the date on which the code was run, and a greyed-out box with color coded *R* code followed by the output. Let's try something a little more complex. Create a new code chunk and type the following:

```
x <- 1:10
```

This will create a variable in *R*, `x`, that is sequentially each whole number between 1 and 10. We can see this by highlighting or typing only the letter `x` and running that line of code by clicking `CTRL` + `ENTER` (Windows / Linux) or `COMMAND` + `ENTER` (Mac).

```
x
```

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

If you look at the top right window, you will also see the value `x` in the environment defined as `int [1:10] 1 2 3 4 5 6 7 8 9 10`. This indicates that `x` is integer data spanning ten positions numbered 1 to 10. Since the vector is small, it displays every number in the sequence.

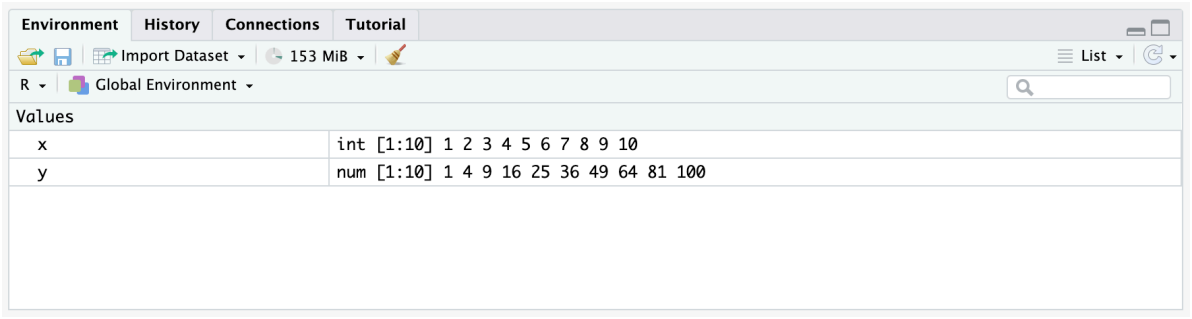


Figure 3.5: *RStudio* environment window showing saved objects. These are in the computer's memory.

Let's create another vector `y` that is the squared values of `x`, such that $y = x^2$. We can raise values to an exponent by using `^`.

```
y <- x^2
y
```

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

Now we have the value `y` in the environment that is the square of the values of `x`. This is a `numeric` vector of 10 values numbered 1 to 10 where each value corresponds to a square of the `x` value. We can raise things to any value however, including x^x!

```
x^x
```

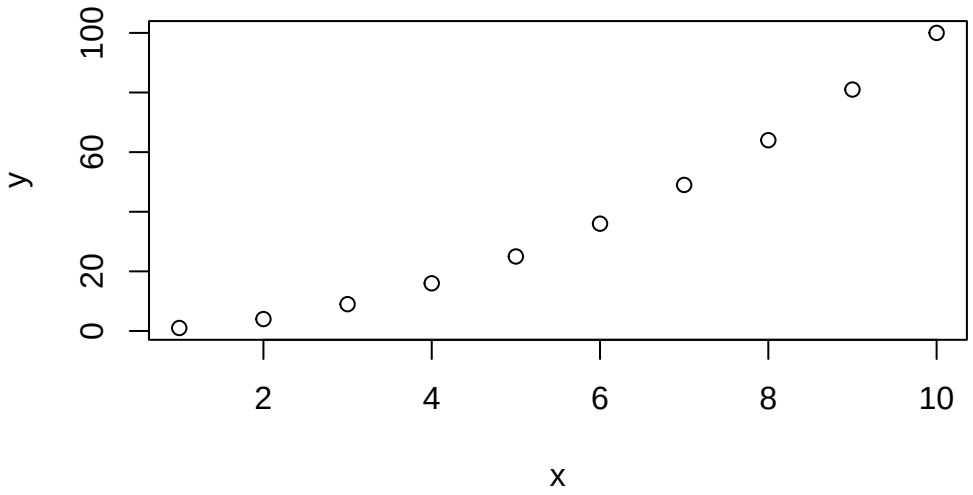
```
 [1]           1           4          27         256        3125       46656
 [7]      823543    16777216   387420489 10000000000
```

As we can see, since I didn't "store" this value as a variable in *R* using `<-`, the value is not in the environment.

3.3 Plotting

Now, let's try creating a plot. This is easy in *R*, as we just use the command `plot`.

```
plot(x = x, y = y)
```



By specifying the `y` and `x` components in `plot`, we can quickly generate a point plot. We can alter the visual parameters of this plot using a few different commands. I will outline these below with inline notes. Inline notes in the code can be made by using a `#` symbol before them, which basically tells *R* to ignore everything after the `#`. For example:

```
print("Test")
```

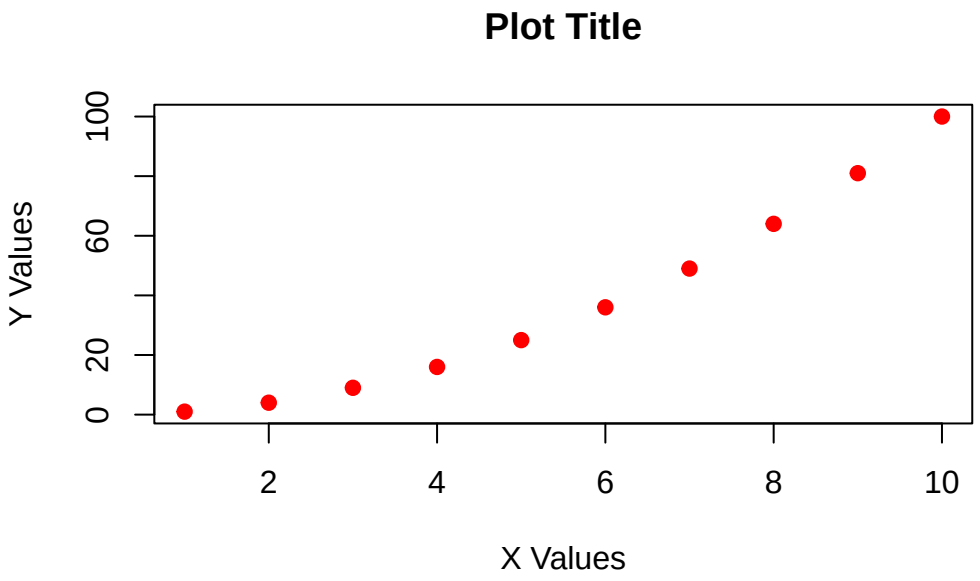
```
[1] "Test"
```

```
# print("Test 2")
```

This prints the word `Test`, but doesn't print `Test 2`.

Now let's make the plot with some new visual parameters.

```r
plot(x = x, # specify x values
     y = y, # specify y values
     ylab = "Y Values", # specify Y label
     xlab = "X Values", # specify X label
     main = "Plot Title", # specify main title
     pch = 19, # adjust point style
     col = "red") # make points red
```



3.4 Tab complete

RStudio allows for "tab-completing" while typing code. Tab-completing is a way of typing the first part of a command, variable name, or file name and hitting "tab" to show all options with that spelling. You should use tab completing because it:

- reduces spelling mistakes
- reduces filepath mistakes
- increases the speed at which you code
- provides help with specific functions

3.5 Help

At any point in *R*, you can look up "help" for a specific function by typing `?functionname`. Try this on your computer with the following:

```
?mean
```

4 Working with data

Throughout this course, we are going to have to work with datasets that are from our book or other sources. Here, we are going to work through an example dataset. First, we need to install *libraries*. A *library* is a collated, pre-existing batch of code that is designed to assist with data analysis or to perform specific functions. These *libraries* make life a lot easier, and create short commands for completing relatively complex tasks.

In this class, there is one major library that you will need *almost every week*! First, we need to install this library:

1. `tidyverse`: this package is actually a group of packages designed to help with data analysis, management, and visualization.

NOTE: If you leave the install prompts in your RMarkdown document, *it will not knit!*

```
# run this code the first time ONLY
# DO NOT INCLUDE IN RMD FILE
# does not need to be run every time you use R!

# tidyverse has a bunch of packages in it!
# great for data manipulation
install.packages("tidyverse")

# if you ever need to update:
# leaving brackets open means "update everything"
update.packages()
```

After packages are installed, we will need to load them into our *R* environment. While we only need to `install.packages` once on our machine, we need to load libraries *every time we restart the program*!

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
```

```
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ---------------------------------------- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

You should an output like the above. What this means is:

1. The core packages that comprise the tidyverse loaded successfully, and version numbers for each are shown.
2. The conflicts basically means that certain commands will not work as they used to because *R* has "re-learned" a particular word.

To clarify the conflicts, pretend that you can only know one definition of a word at a time. You may know the word "cola" as a type of soda pop or as a drink in general. However, in Spanish, "cola" refers to a line or a tail. While we can learn both of these definitions and know which one is which because of context, a computer can't do that. In *R*, we would then have to specify which "cola" we are referring to. We do this by listing the package before the command; in this case, `english::cola` would mean a soda pop and `spanish::cola` would refer to a line or tail. If we just type `cola`, the computer will assume one of these definitions but not even consider the other.

We won't have to deal with conflicts much in this class, and I'll warn you (or help you) if there is a conflict.

4.1 Downloading data

Now, we need to download our first data set. These datasets are stored on GitHub. We are going to be looking at data from Dr. Cooper's dissertation concerning Afrotropical bird distributions (Cooper 2021). This website is in the data folder on this websites' GitHub page, accessible here.

```
# read comma separated file (csv) into R memory
# reads directly from URL
ranges <- read_csv("https://raw.githubusercontent.com/jacobccooper/biol305_unk/main/datasets/
```

```
Rows: 12 Columns: 10
-- Column specification --------------------------------------------------------
```

```
Delimiter: ","
chr (1): species
dbl (9): combined_current_km2, consensus_km2, bioclim_current_km2, 2050_comb...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Alternatively, we can use the operator `%>%` to simplify this process. `%>%` means "take whatever you got from the previous step and *pipe* it into the next step". So, the following does the exact same thing:

```
ranges <- "https://raw.githubusercontent.com/jacobccooper/biol305_unk/main/datasets/lacustri
  read_csv()
```

```
Rows: 12 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): species
dbl (9): combined_current_km2, consensus_km2, bioclim_current_km2, 2050_comb...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Using the `%>%` is preferred as you can better set up a workflow and because it more closely mimics other coding languages, such as `bash`.

Let's view the data to see if it worked. We can use the command `head` to view the first few rows:

```
head(ranges)
```

```
# A tibble: 6 x 10
  species                combined_current_km2 consensus_km2 bioclim_current_km2
  <chr>                                 <dbl>         <dbl>               <dbl>
1 Batis_diops                           25209.         6694.              19241.
2 Chamaetylas_poliophrys                68171.         1106.              68158.
3 Cinnyris_regius                       60939.        13305.              53627.
4 Cossypha_archeri                      27021.         6409.              11798.
5 Cyanomitra_alinae                     78680.        34320.              63381.
6 Graueria_vittata                       8770.          861.               8301.
# i 6 more variables: `2050_combined_km2` <dbl>, `2050_consensus_km2` <dbl>,
#   `2070_combined_km2` <dbl>, `2070_consensus_km2` <dbl>,
#   alltime_consensus_km2 <dbl>, past_stable_km2 <dbl>
```

We can perform a lot of summary statistics in *R*. Some of these we can view for multiple columns at once using `summary`.

```
summary(ranges)
```

```
   species          combined_current_km2 consensus_km2     bioclim_current_km2
 Length:12          Min.   :  8770       Min.   :  861.3   Min.   :  3749
 Class :character   1st Qu.: 24800       1st Qu.: 4186.2   1st Qu.: 10924
 Mode  :character   Median : 43654       Median : 7778.1   Median : 31455
                    Mean   : 68052       Mean   :18161.8   Mean   : 42457
                    3rd Qu.: 70798       3rd Qu.:18558.7   3rd Qu.: 62835
                    Max.   :232377       Max.   :79306.6   Max.   :148753
 2050_combined_km2 2050_consensus_km2 2070_combined_km2  2070_consensus_km2
 Min.   :  1832    Min.   :    0.0    Min.   :   550.3   Min.   :    0.0
 1st Qu.:  6562    1st Qu.:  589.5    1st Qu.:  6583.8   1st Qu.:  311.4
 Median : 26057    Median : 6821.9    Median : 24281.7   Median : 2714.6
 Mean   : 33247    Mean   :14418.4    Mean   : 31811.0   Mean   : 8250.5
 3rd Qu.: 40460    3rd Qu.:18577.1    3rd Qu.: 38468.9   3rd Qu.:10034.4
 Max.   :132487    Max.   :79236.2    Max.   :129591.0   Max.   :53291.8
 alltime_consensus_km2 past_stable_km2
 Min.   :    0.0       Min.   :   0.0
 1st Qu.:  790.9       1st Qu.:   0.0
 Median : 8216.8       Median :   0.0
 Mean   :15723.3       Mean   : 127.3
 3rd Qu.:19675.0       3rd Qu.:   0.0
 Max.   :82310.5       Max.   :1434.8
```

As seen above, we now have information for the following statistics for each variable:

- `Min` = minimum
- `1st Qu.` = 1st quartile
- `Median` = middle of the dataset
- `Mean` = average of the dataset
- `3rd Qu.` = 3rd quartile
- `Max.` = maximum

We can also calculate some of these statistics manually to see if we are doing everything correctly. It is easiest to do this by using predefined functions in *R* (code others have written to perform a particular task) or to create our own functions in *R*. We will do both to determine the average of `combined_current_km2`.

4.2 Subsetting data

First, we need to select only the column of interest. In *R*, we have two ways of subsetting data to get a particular column.

- `var[rows,cols]` is a way to look at a particular object (`var` in this case) and choose a specific combination of `row` number and `column` number (`col`). This is great if you know a specific index, but it is better to use a specific name.
- `var[rows,"cols"]` is a way to do the above but by using a specific column name, like `combined_current_km2`.
- `var$colname` is a way to call the specific column name directly from the dataset.

```
# using R functions

ranges$combined_current_km2
```

```
 [1]  25209.4  68171.2  60939.2  27021.3  78679.9   8769.9 232377.2  17401.4
 [9]  51853.5  35455.1  23570.3 187179.1
```

As shown above, calling the specific column name with `$` allows us to see only the data of interest. We can also save these data as an object.

```
current_combined <- ranges$combined_current_km2

current_combined
```

```
 [1]  25209.4  68171.2  60939.2  27021.3  78679.9   8769.9 232377.2  17401.4
 [9]  51853.5  35455.1  23570.3 187179.1
```

Now that we have it as an object, specifically a numeric vector, we can perform whatever math operations we need to on the dataset.

```
mean(current_combined)
```

```
[1] 68052.29
```

Here, we can see the mean for the entire dataset. However, we should always round values to the same number of decimal points as the original data. We can do this with `round`.

```
round(mean(current_combined),1) # round mean to one decimal
```

```
[1] 68052.3
```

Note that the above has a nested set of commands. We can write this exact same thing as follows:

```
# pipe mean through round
mean(current_combined) %>%
  round(1)
```

```
[1] 68052.3
```

Use the method that is easiest for you to follow!

We can also calculate the mean manually. The mean is $\frac{\sum_{i=1}^{n} x}{n}$, or the sum of all the values within a vector divided by the number of values in that vector.

```
# create function
# use curly brackets to denote function
# our data goes in place of "x" when finally run
our_mean <- function(x){
  sum_x <- sum(x) # sum all values in vector
  n <- length(x) # get length of vector
  xbar <- sum_x/n # calcualte mean
  return(xbar) # return the value outside the function
}
```

Let's try it.

```
our_mean(ranges$combined_current_km2)
```

```
[1] 68052.29
```

As we can see, it works just the same as **mean**! We can round this as well.

```
our_mean(ranges$combined_current_km2) %>%
  round(1)
```

```
[1] 68052.3
```

5 Your turn!

With a partner or on your own, try to do the following:

1. Create an *RMarkdown document* that will save as an `.html`.
2. Load the data, as shown here, and print the summary statistics in the document.
3. Calculate the value of `combined_current_km2` divided by `2050_combined_km2` and print the results.
4. `knit` your results, with your name(s) and date, as an HTML document.

Let me know if you have any issues.

6 Descriptive Statistics

6.1 Purposes of descriptive statistics

Descriptive statistics enable researchers to quickly and easily examine the "behavior" of their datasets, identifying potential errors and allowing them to observe particular trends that may be worth further analysis. Here, we will cover how to calculate descriptive statistics for multiple different datasets, culminating in an assignment covering these topics.

6.2 Preparing *R*

As with every week, we will need to load our relevant packages first. This week, we are using the following:

```
# enables data management tools
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

6.3 Downloading the data

For the example this week, we will be using the `starbucks` dataset, describing the number of drinks purchased during particular time periods during the day.

```
starbucks <- read_csv("https://raw.githubusercontent.com/jacobccooper/biol105_unk/main/datas
```

```
Rows: 9 Columns: 2
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Hour
dbl (1): Frap_Num

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

6.4 Descriptive statistics

Descriptive statistics are statistics that help us understand the shape and nature of the data on hand. These include really common metrics such as *mean*, *median*, and *mode*, as well as more nuanced metrics like *quartiles* that help us understand if there is any *skew* in the dataset. (*Skew* refers to a bias in the data, where more data points lie on one side of the distribution and there is a long *tail* of data in the other direction).

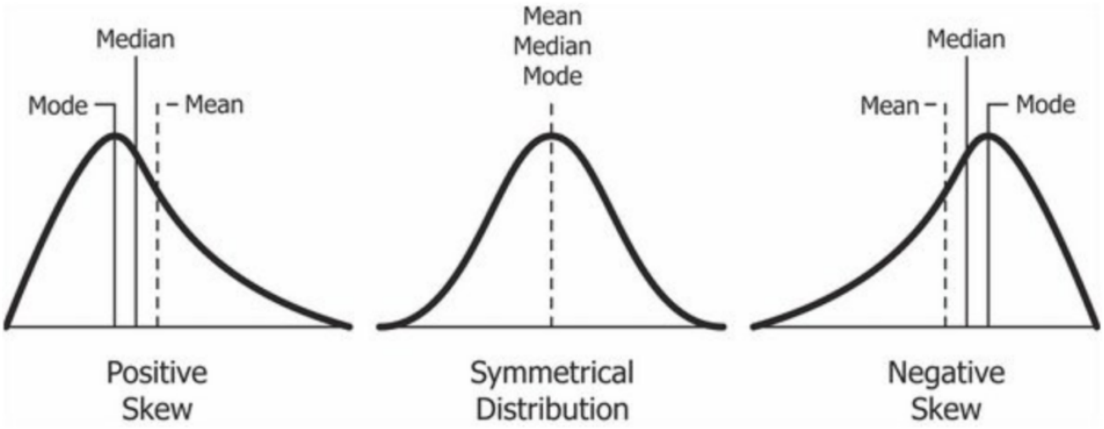


Figure 6.1: Examples of skew compared to a symmetrical, non-skewed distribution. Source: machinelearningparatodos.com

Note above that the relative position of the *mean*, *median*, and *mode* can be indicative of skew. Please also note that these values will rarely be exactly equal "in the real world", and thus you need to weigh differences against the entire dataset when assessing skew. There is a lot of nuance like this in statistics; it is not always an "exact" science, but sometimes involves judgment calls and assessments based on what you observe in the data.

Using the `starbucks` dataset, we can look at some of these descriptive statistics to understand what is going on.

6.4.1 Notation

As a quick reminder, we use Greek lettering for *populations* and Roman lettering for samples. For example:

- σ is a population, but s is a sample (both these variables refer to *standard deviation*).
- μ is a population, but $\bar{x}$ is a sample (both of these variables refer to the *mean*).

6.4.2 Mean

The mean is the "average" value within a set of data, specifically, the sum of all values divided by the length of those values: $\frac{\sum_{i=1}^{n} x}{n}$.

```
head(starbucks)
```

```
# A tibble: 6 x 2
  Hour      Frap_Num
  <chr>        <dbl>
1 0600-0659        2
2 0700-0759        3
3 0800-0859        2
4 0900-0959        4
5 1000-1059        8
6 1100-1159        7
```

Here, we are specifically interested in the number of frappuccinos.

```
# get vector of frappuccino number
fraps <- starbucks$Frap_Num

# get mean of vector
mean(fraps)
```

```
[1] 6.222222
```

Note that the above should be rounded to a whole number, since we were given the data in whole numbers!

```
mean(fraps) %>%
  round(0)
```

```
[1] 6
```

```
# OR

round(mean(fraps),0)
```

```
[1] 6
```

We already covered calculating the average manually in our previous tutorial, but we can do that here as well:

```
# sum values
# divide by n, length of vector
# round to 0 places
round(sum(fraps)/length(fraps),0)
```

```
[1] 6
```

6.4.3 Range

The range is the difference between the largest and smallest units in a dataset. We can use the commands `min` and `max` to calculate this.

```
max(fraps) - min(fraps)
```

```
[1] 13
```

The range of our dataset is 13.

6.4.4 Median

The median is also known as the 50th percentile, and is the midpoint of the data when ordered from least to greatest. If there are an even number of data points, then it is the average point between the two center points. For odd data, this is the $\frac{n+1}{2}$th observation. For even data, since we need to take an average, this is the $\frac{\frac{n}{2}+(\frac{n}{2}+1)}{2}$. You should be able to do these by hand and by using a program.

```
median(fraps)
```

```
[1] 4
```

Now, to calculate by hand:

```
length(fraps)
```

```
[1] 9
```

We have an odd length.

```
# order gets the order
order(fraps)
```

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

```
# [] tells R which elements to put where
frap_order <- fraps[order(fraps)]

frap_order
```

```
[1]  2  2  2  3  4  7  8 13 15
```

```
# always use parentheses
# make sure the math maths right!
(length(frap_order)+1)/2
```

```
[1] 5
```

Which is the fifth element in the vector?

```
frap_order[5]
```

```
[1] 4
```

Now let's try it for an even numbers.

```
# remove first element
even_fraps <- fraps[-1]

even_fraps_order <- even_fraps[order(even_fraps)]

even_fraps_order
```

```
[1]  2  2  3  4  7  8 13 15
```

```
median(even_fraps)
```

```
[1] 5.5
```

Now, by hand: $\frac{\frac{n}{2}+(\frac{n}{2}+1)}{2}$.

```
n <- length(even_fraps_order)

# get n/2 position from vector
m1 <- even_fraps_order[n/2]
# get n/2+1 position
m2 <- even_fraps_order[(n/2)+1]

# add these values, divide by two for "midpoint"
med <- (m1+m2)/2

med
```

```
[1] 5.5
```

As we can see, these values are equal!

6.4.5 Other quartiles and quantiles

We also use the 25th percentile and the 75th percentile to understand data distributions. These are calculated similar to the above, but the bottom quartile is only $\frac{1}{4}$ of the way between values and the 75th quartile is $\frac{3}{4}$ of the way between values. We can use the R function `quantile` to calculate these.

```
quantile(frap_order)
```

```
  0%  25%  50%  75% 100%
   2    2    4    8   15
```

We can specify a quantile as well:

```
quantile(frap_order, 0.75)
```

```
75%
  8
```

We can also calculate these metrics by hand. Let's do it for the even dataset, since this is more difficult.

```
quantile(even_fraps_order)
```

```
   0%   25%   50%   75%  100%
 2.00  2.75  5.50  9.25 15.00
```

Note that the 25th and 75th percentiles are also between two different values. These can be calculated as a quarter and three-quarters of the way between their respective values.

```
# 75th percentile

n <- length(even_fraps_order)

# get position
p <- 0.75*(n+1)

# get lower value
# round down
```

```
m1 <- even_fraps_order[trunc(p)]

# get upper value
# round up
m2 <- even_fraps_order[ceiling(p)]

# position between
# fractional portion of rank
frac <- p-trunc(p)

# calculate the offset from lowest value
val <- (m2 - m1)*frac

# get value
m1 + val
```

```
[1] 11.75
```

Wait… why does our value differ?

R, by default, calculates quantiles using what is called `Type 7`, in which the quantiles are calculated by $p_k = \frac{k-1}{n-1}$, where n is the length of the vector and k refers to the quantile being used. However, in our book and in this class, we use `Type 6` interpretation - $p_k = \frac{k}{n+1}$. Let's try using `Type 6`:

```
quantile(even_fraps_order, type = 6)
```

```
   0%   25%   50%   75%  100%
 2.00  2.25  5.50 11.75 15.00
```

Now we have the same answer as we calculated by hand!

This is a classic example of how things in *R* (and in statistics in general!) can depend on interpretation and are not always "hard and fast" rules.

In this class, we will be using Type 6 interpretation for the quantiles - you will have to specify this in the quantile function EVERY TIME! If you do *not* specify Type 6, you will get the questions incorrect and you will get answers that do not agree with the book, with Excel, or what you calculate by hand.

6.4.6 Mode

There is no default method for finding the mode in *R*. However, websites like Statology provide wraparound functions.

```
# Based on Statology function
# define function to calculate mode
find_mode <- function(x) {
  # get unique values from vector
  u <- unique(x)
  # count number of occurrences for each value
  tab <- tabulate(match(x, u))

  # if no mode, say so
  if(length(x)==length(u[tab == max(tab)])){
    print("No mode.")
  }else{
    # return the value with the highest count
    u[tab == max(tab)]
  }
}

find_mode(fraps)
```

```
[1] 2
```

We can also do this by hand, by counting the number of occurrences of each value. This can be done in a stepwise fashion using commands in the above function.

```
# unique counts
u <- unique(fraps)
u
```

```
[1]  2  3  4  8  7 15 13
```

```
# which elements match
match(fraps,u)
```

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

```
# count them
tab <- match(fraps,u) %>%
  tabulate()

tab
```

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

Get the highest value.

```
u[tab==max(tab)]
```

```
[1] 2
```

Notice this uses `==`. This is a logical argument that means "is equal to" or "is the same as". For example:

```
2 == 2
```

```
[1] TRUE
```

These values are the same, so `TRUE` is returned.

```
2 == 3
```

```
[1] FALSE
```

These values are unequal, so `FALSE` is returned. *R* will read `TRUE` as `1` and `FALSE` as `ZERO`, such that:

```
sum(2==2)
```

```
[1] 1
```

and

```
sum(2==3)
```

```
[1] 0
```

This allows you to find how many arguments match your condition quickly, and even allows you to subset based on these indices as well. Keep in mind you can use greater than `<`, less than `>`, greater than or equal to `<=`, less than or equal to `>=`, is equal to `==`, and is not equal to `!=` to identify numerical relationships. Other logical arguments include:

- `&`: both conditions must be `TRUE` to match (e.g., `c(10,20) & c(20,10)`). Try the following as well: `fraps < 10 & fraps > 3`.
- `&&`: and, but works with single elements and allows for better parsing. Often used with `if`. E.g., `fraps < 10 && fraps > 3`. This will not work on our multi-element `frap` vector.
- `|`: or, saying at least one condition must be true. Try: `fraps > 10 | fraps < 3`.
- `||`: or, but for a single element, like `&&` above.
- `!`: not, so "not equal to" would be `!=`.

6.4.7 Variance

When we are dealing with datasets, the variance is a measure of the total spread of the data. The variance is calculated using the following:

$$\sigma^2 = \frac{\sum(x_i - \bar{x})^2}{n-1}$$

Essentially, this means that for every value of x, we are finding the difference between that value and the mean and squaring it, summing all of these quared differences, and dividing them by the number of samples in the dataset minus one. Let's do this for the frappuccino dataset.

```
frap_order
```

```
[1]  2  2  2  3  4  7  8 13 15
```

Now to find the differences.

```r
diffs <- frap_order - mean(frap_order)

diffs
```

```
[1] -4.2222222 -4.2222222 -4.2222222 -3.2222222 -2.2222222  0.7777778  1.7777778
[8]  6.7777778  8.7777778
```

Note that R is calculating the same thing for the entire vector! Since these are differences from the mean, they should sum to zero.

```r
sum(diffs)
```

```
[1] 3.552714e-15
```

This is not quite zero due to rounding error, but is essentially zero as it is 0.0000000000000036.

```r
# square differences
diffs_sq <- diffs^2

diffs_sq
```

```
[1] 17.8271605 17.8271605 17.8271605 10.3827160  4.9382716  0.6049383  3.1604938
[8] 45.9382716 77.0493827
```

Now we have the squared differences. We need to sum these and divide by $n - 1$.

```r
n <- length(frap_order)

var_frap <- sum(diffs_sq)/(n-1)

var_frap
```

```
[1] 24.44444
```

Let's check this against the built-in variance function in R.

```r
var(frap_order)
```

```
[1] 24.44444
```

They are identical! We can check this using a logical argument.

```
var_frap == var(frap_order)
```

```
[1] TRUE
```

Seeing as this is `TRUE`, we calculated it correctly.

6.4.8 Standard deviation

Another common measurement of spread is the standard deviation (σ). As you remember from class (or may have guessed from the notation on this site), the standard deviation is just the square root of the variance.

```
sqrt(var_frap)
```

```
[1] 4.944132
```

We can test this against the built in `sd` function in *R*:

```
sqrt(var_frap) == sd(frap_order)
```

```
[1] TRUE
```

As you can see, we calculated this correctly!

6.4.9 Standard error

The standard error is used to help understand the spread of data and to help estimate the accuracy of our measurements for things like the mean. The standard error is calculated thusly:

$$SE = \frac{\sigma}{\sqrt{n}}$$

There is not built in function for the standard error in excel, but we can write our own:

```r
se <- function(x){
  n <- length(x) # calculate n
  s <- sd(x) # calculate standard deviation
  se_val <- s/sqrt(n)
  return(se_val)
}
```

Let's test this code.

```r
se(frap_order)
```

```
[1] 1.648044
```

Our code works! And we can see exactly how the standard error is calculate. We can also adjust this code as needed for different situations, like samples.

Remember, the standard error is used to help reflect our *confidence* in a specific measurement (*e.g.*, how certain we are of the mean, and what values we believe the mean falls between). We want our estimates to be as precise as possible with as little uncertainty as possible. Given this, does having more samples make our estimates more or less confident? Mathematically, what happens as our sample size *increases*?

6.4.10 Coefficient of variation

The coefficient of variation, another measure of data spread and location, is calculated by the following:

$$CV = \frac{\sigma}{\mu}$$

We can write a function to calculate this in R as well.

```r
cv <- function(x){
  sigma <- sd(x)
  mu <- mean(x)
  val <- sigma/mu
  return(val)
}

cv(frap_order)
```

```
[1] 0.7945927
```

Remember that we will need to round values.

6.4.11 Outliers

Outliers are any values that are outside of the 1.5 times the interquartile range. We can calculate this for our example dataset as follows:

```
lowquant <- quantile(frap_order,0.25,type = 6) %>% as.numeric()

hiquant <- quantile(frap_order,0.75,type = 6) %>% as.numeric()

iqr <- hiquant - lowquant
```

We can also calculate the interquartile range using `IQR`. **Remember**, you must use `type = 6`!

```
iqr <- IQR(frap_order, type = 6)
```

Now, to calculate the "whiskers".

```
lowbound <- lowquant - (1.5*iqr)
hibound <- hiquant + (1.5*iqr)

# low outliers?
# select elements that match
# identify using logical "which"
frap_order[which(frap_order < lowbound)]
```

```
numeric(0)
```

```
# high outliers?
# select elements that match
# identify using logical "which"
frap_order[which(frap_order > hibound)]
```

```
numeric(0)
```

We have no outliers for this particular dataset.

6.5 Homework: Descriptive Statistics

Now that we've covered these basic statistics, it's your turn! For this week, you will be completing homework that involves methods from Chapter 4 in your book.

6.5.1 Homework instructions

Please create an *RMarkdown* document that will render as an `.html` file. You will submit this file to show your coding and your work. Please refer to the Introduction to *R* for refreshers on how to create an `.html` document in *RMarkdown*. You will need to do the following for each of these datasets:

- mean
- median
- range
- interquartile range
- variance
- standard deviation
- coefficient of variation
- standard error
- whether there are any "outliers"

Please show all of your work for full credit.

6.5.2 Data for homework problems

For each question, calculate the mean, median, range, interquartile range, variance, standard deviation, coefficient of variation, standard error, and whether there are any "outliers".

Please also write your own short response to the *Synthesis question* posed, which will involve thinking about the data and metrics you just analyzed.

6.5.2.1 1: UNK Nebraskans

Ever year, the university keeps track of where students are from. The following are data on the umber of students admitted to UNK from the state of Nebraska:

```
# create dataset in R
nebraskans <- c(5056,5061,5276,5244,5209,
                5262,5466,5606,5508,5540,5614)

years <- 2023:2013

nebraskans_years <- cbind(years,nebraskans) %>%
  as.data.frame()

nebraskans_years
```

```
   years nebraskans
1   2023       5056
2   2022       5061
3   2021       5276
4   2020       5244
5   2019       5209
6   2018       5262
7   2017       5466
8   2016       5606
9   2015       5508
10  2014       5540
11  2013       5614
```

Using these data, please calculate the mean, median, range, interquartile range, variance, standard deviation, coefficient of variation, standard error, and whether there are any "outliers" for the number of UNK students from Nebraska.

In order to do this, we will need to first select the column that denotes the number of Nebraskans from the dataset. Remember, we need to save this as an object in *R* to do the analyses. Here is a demonstration of getting the column to look at the mean, so that you can repeat this for the other questions. This relies heavily on the use of `$`, used to get the data from a specific column.

```
# $ method
nebraskans <- nebraskans_years$nebraskans

nebraskans
```

```
 [1] 5056 5061 5276 5244 5209 5262 5466 5606 5508 5540 5614
```

Now we can get the `mean` of this vector.

```
mean(nebraskans) %>%
  round(0) # don't forget to round!
```

```
[1] 5349
```

Synthesis question: Do you think that there are any really anomalous years? Do you feel data are similar between years? *Note* we are not looking at trends through time but whether any years are outliers.

6.5.2.2 2: Piracy in the Gulf of Guinea

The following is a dataset looking at oceanic conditions and other variables associated with pirate attacks within the region between 2010 and 2021 (Moura et al. 2023). Using these data, please calculate the mean, median, range, interquartile range, variance, standard deviation, coefficient of variation, standard error, and whether there are any "outliers" for distance from shore for each pirate attack (column `Distance_from_Coast`).

```
pirates <- read_csv("https://figshare.com/ndownloader/files/42314604")
```

```
New names:
Rows: 595 Columns: 40
-- Column specification
-------------------------------------------------------- Delimiter: "," chr
(18): Period, Season, African_Season, Coastal_State, Coastal_Zone, Navi... dbl
(20): ...1, Unnamed: 0, Year, Month, Lat_D, Lon_D, Distance_from_Coast,... dttm
(2): Date_Time, Date
i Use `spec()` to retrieve the full column specification for this data. i
Specify the column types or set `show_col_types = FALSE` to quiet this message.
* `` -> `...1`
```

Synthesis question: Do you notice any patterns in distance from shore? What may be responsible for these patterns? *Hint*: Think about what piracy entails and also what other columns are available as other variables in the above dataset.

6.5.2.3 3: Patient ages at presentation

The following is a dataset on skin sores in different communities in Australia and Oceania, specifically looking at the amount of time that passes between skin infections (Lydeamore et al. 2020a). This file includes multiple different datasets, and focuses on data from children in the first five years of their life, on household visits, and on data collected during targeted studies (Lydeamore et al. 2020b).

```
ages <- read_csv("https://doi.org/10.1371/journal.pcbi.1007838.s006")
```

```
Rows: 17150 Columns: 2
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): dataset
dbl (1): time_difference

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Let's see what this file is like real fast. We can use the command `dim` to see the `rows` and `columns`.

```
dim(ages)
```

```
[1] 17150     2
```

As you can see, this file has only two columns but 17,150 rows! For the column `time_difference`, please calculate the mean, median, range, interquartile range, variance, standard deviation, coefficient of variation, standard error, and whether there are any "outliers".

Synthesis question: Folks will often think about probabilities of events being "low but never zero". What does that mean in the context of these data? What about these data make you feel like probabilities may decrease through time but never become zero?

7 Diagnosing data visually

7.1 The importance of visual inspection

Inspecting data visually can give us a lot of information about whether data are normally distributed and about whether there are any major errors or issues with our dataset. It can also help us determine if data meet model assumptions, or if we need to use different tests more appropriate for our datasets.

7.2 Sample data and preparation

First, we need to load our *R* libraries.

```
library(curl)
```

```
Using libcurl 8.7.1 with LibreSSL/3.3.6
```

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
```

```
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter()     masks stats::filter()
x dplyr::lag()        masks stats::lag()
x readr::parse_date() masks curl::parse_date()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to beco
```

Next, we can download our data sample.

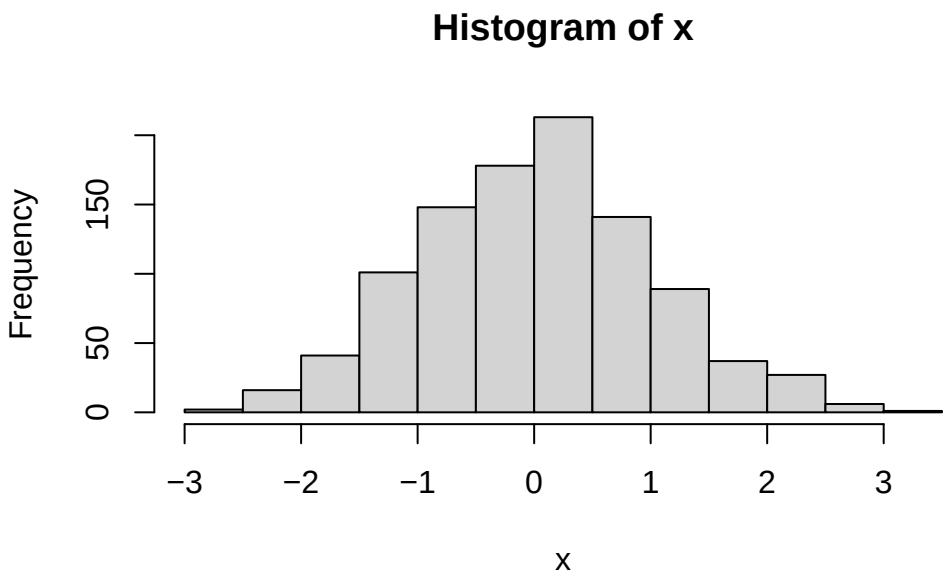
7.3 Histograms

A histogram is a frequency diagram that we can use to visually diagnose data and their distributions. We are going to examine a histogram using a random string of data. *R* can generate random (though, actually pseudorandom) strings of data on command, pulling them from different distributions. These distributions are pseudorandom because we can't actually program *R* to be random, so it starts from a wide variety of pseudorandom points.

7.3.1 Histograms on numeric vectors

The following is how to create default histograms on data. If you need to create custom bin sizes, please see the notes under *Cumulative frequency plots* for data that are not already in frequency format.

```
# create random string from normal distribution
# this step is not necessary for data analysis in homework
x <- rnorm(n = 1000, # 1000 values
           mean = 0,
           sd = 1)

# make histogram
hist(x)
```

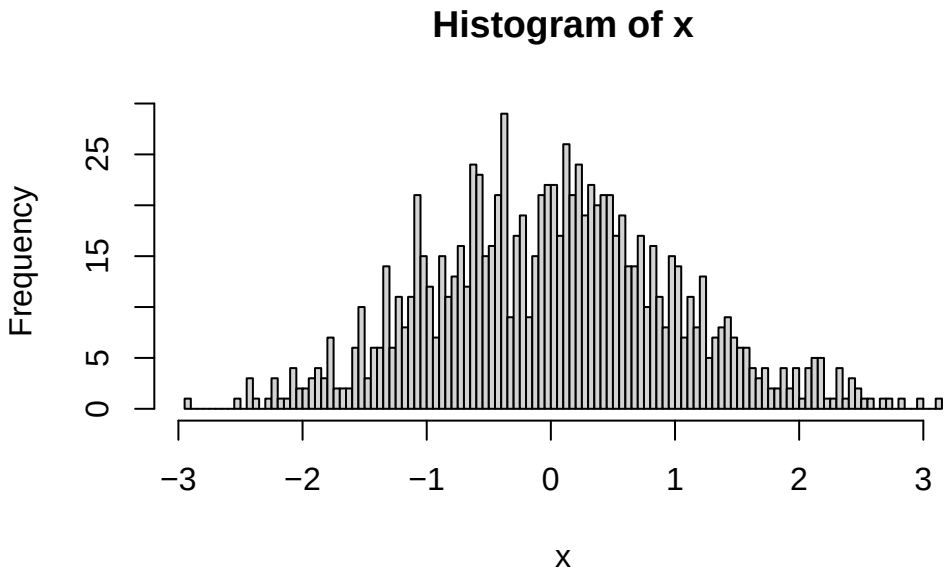


NOTE that a histogram can only be made on a vector of values. If you try to make a histogram on a data frame, *you will get an error and it will not work*. You have to specify which column

you wish to use with the `$` operator. (For example, for dataframe `xy` with columns `x` and `y`, you would use `hist(xy$y)`).

We can up the number of bins to see this better.

```
hist(x,breaks = 100)
```



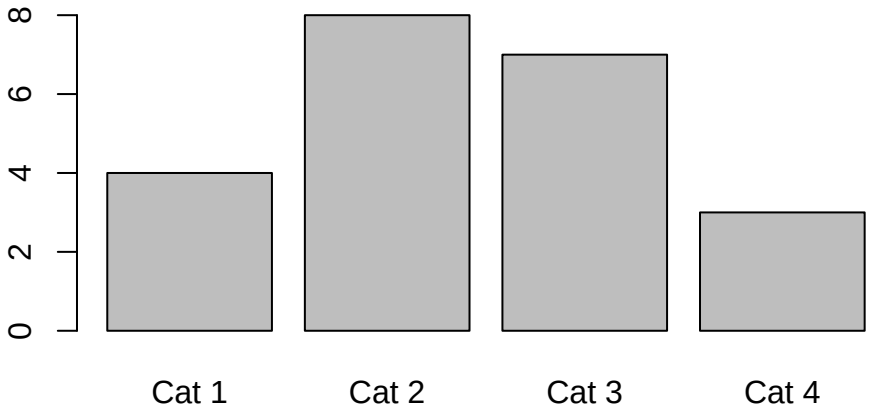
The number of bins can be somewhat arbitrary, but a value should be chosen based off of what illustrates the data well. *R* will auto-select a number of bins in some cases, but you can also select a number of bins. Some assignments will ask you to choose a specific number of bins as well.

7.3.2 Histograms on frequency counts

Say, for example, that we have a dataset where everything is already shown as frequencies. We can create a frequency histogram using `barplot`.

```
count_table <- matrix(nrow = 4, ncol = 2, byrow = T,
                      data = c("Cat 1", 4,
                               "Cat 2", 8,
                               "Cat 3", 7,
                               "Cat 4", 3)) %>%
  as.data.frame()

colnames(count_table) <- c("Category","Count")
```

```r
# ensure counts are numeric data
count_table$Count <- as.numeric(count_table$Count)

# manually create histogram
barplot(count_table$Count, # response variable, counts for histogram
        axisnames = T, # make names on plot
        names.arg = count_table$Category) # make these the names
```



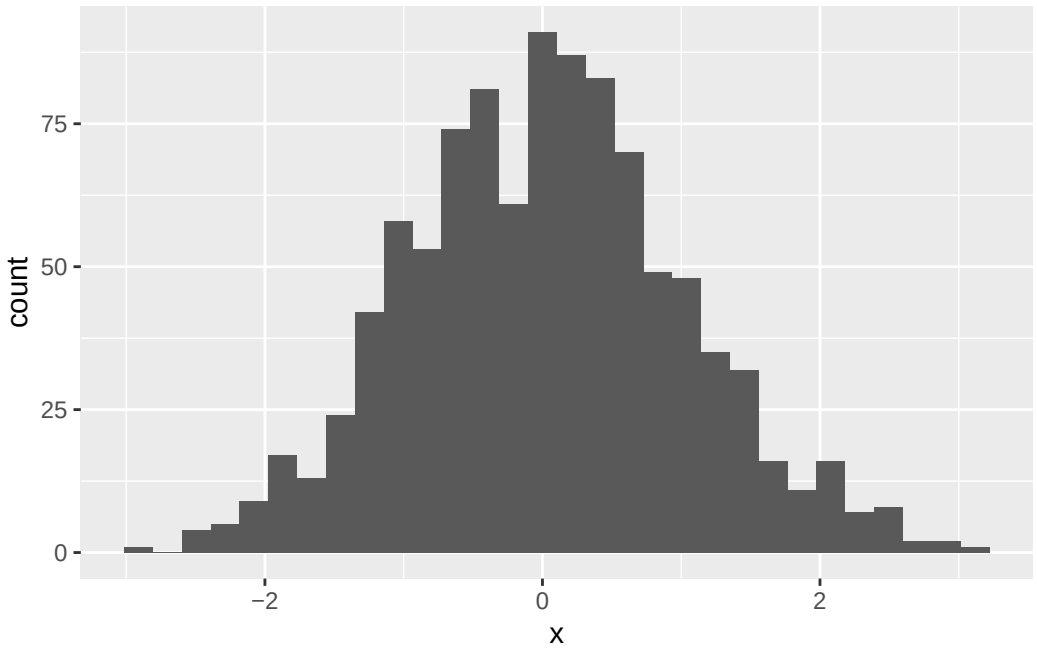
7.3.3 ggplot histograms

The following is an ***optional*** workthrough on how to make really fancy plots.

We can also use the program `ggplot`, part of the `tidyverse`, to create histograms.

```r
# ggplot requires data frames
x2 <- x %>% as.data.frame()
colnames(x2) <- "x"

ggplot(data = x2, aes(x = x)) +
  geom_histogram()
```

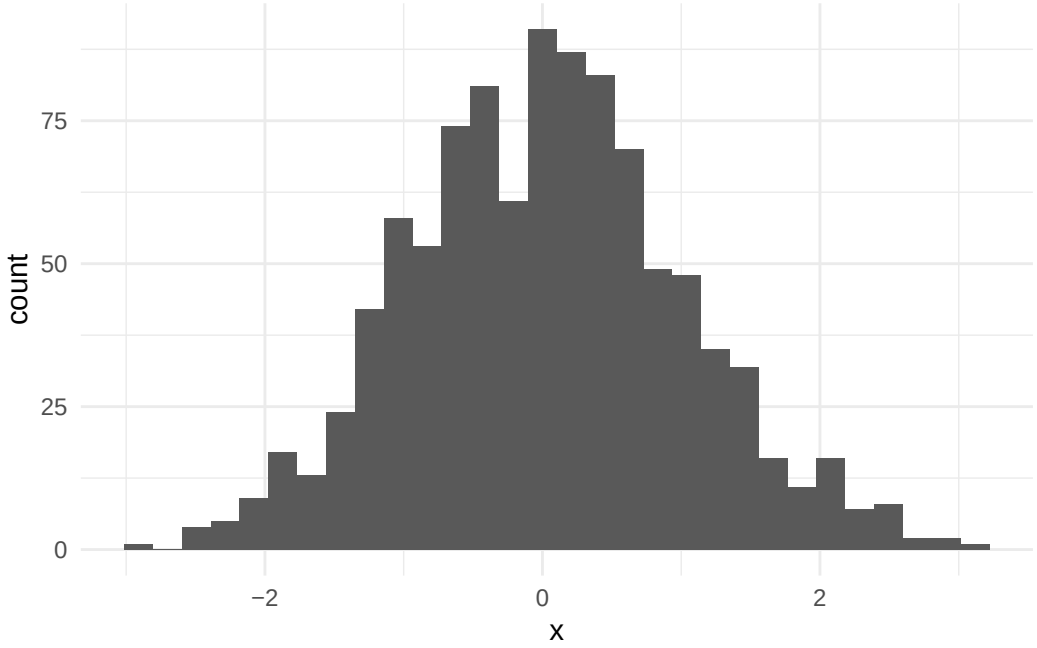
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



`ggplot` is nice because we can also clean up this graph a little.

```
ggplot(x2,aes(x=x)) + geom_histogram() +
  theme_minimal()
```

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



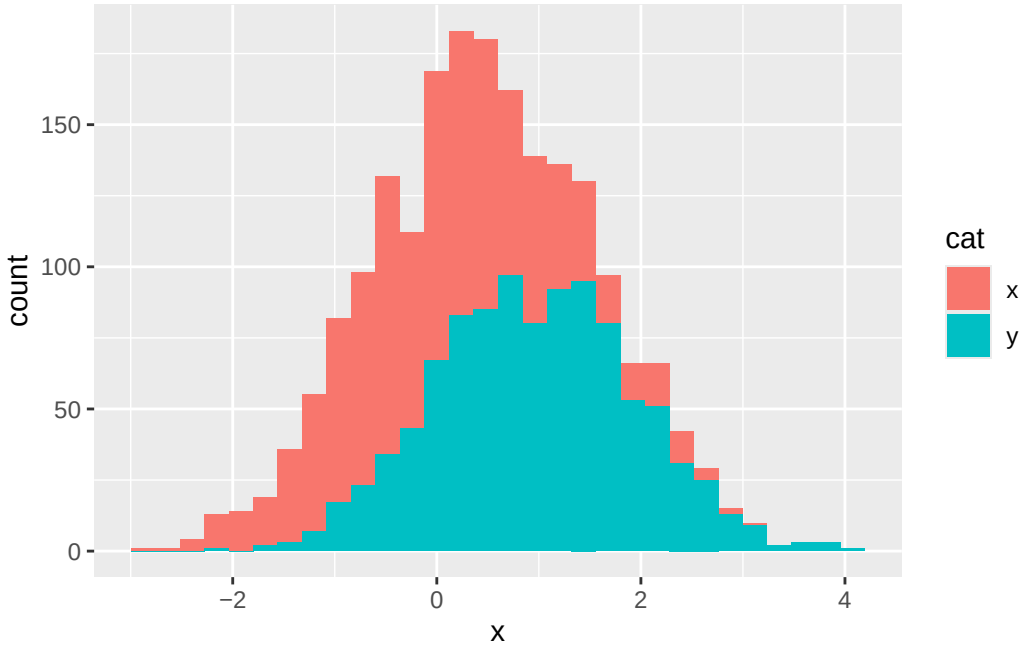
We can also do a histogram of multiple values at once in *R*.

```
x2$cat <- "x"

y <- rnorm(n = 1000,
           mean = 1,
           sd = 1) %>%
  as.data.frame()

colnames(y) <- "x"
y$cat <- "y"

xy <- rbind(x2,y)

head(xy)
```

```
           x cat
1 -0.2719708   x
2 -1.1222879   x
3 -1.2630812   x
4 -0.1657854   x
5 -2.2494034   x
6 -0.8238919   x
```

```
ggplot(xy,aes(x = x, fill = cat)) +
  geom_histogram()
```

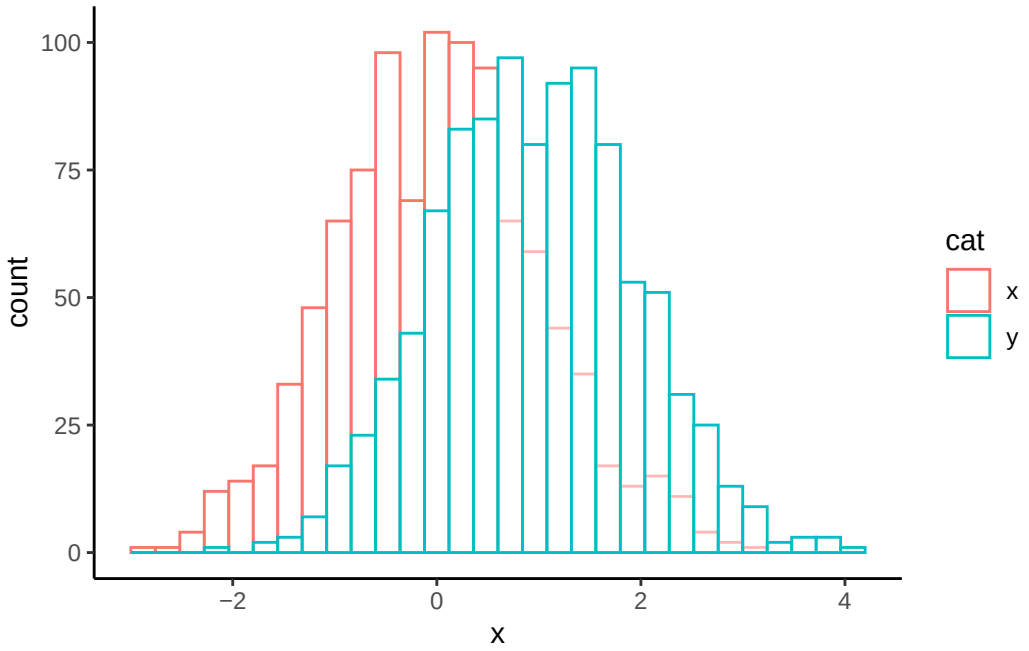
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



We can also make this look a little nicer.

```
ggplot(xy, aes(x = x, colour = cat)) +
  geom_histogram(fill = "white", alpha = 0.5, # transparency
                 position = "identity") +
  theme_classic()
```

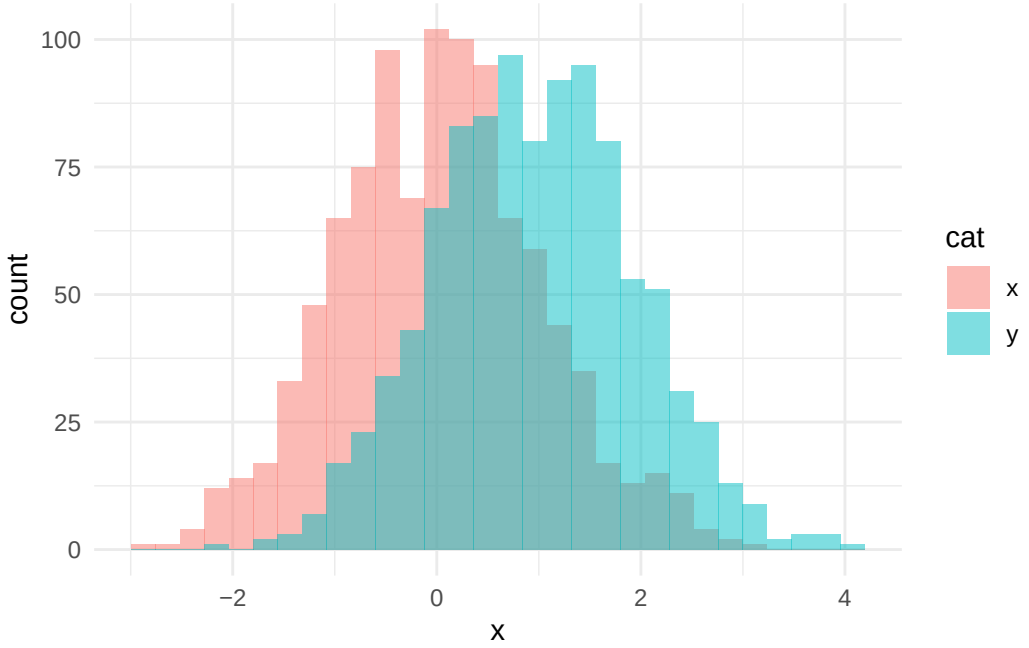
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



We can show these a little differently as well.

```
ggplot(xy, aes(x = x, fill = cat))+
  geom_histogram(position = "identity", alpha = 0.5) +
  theme_minimal()
```

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

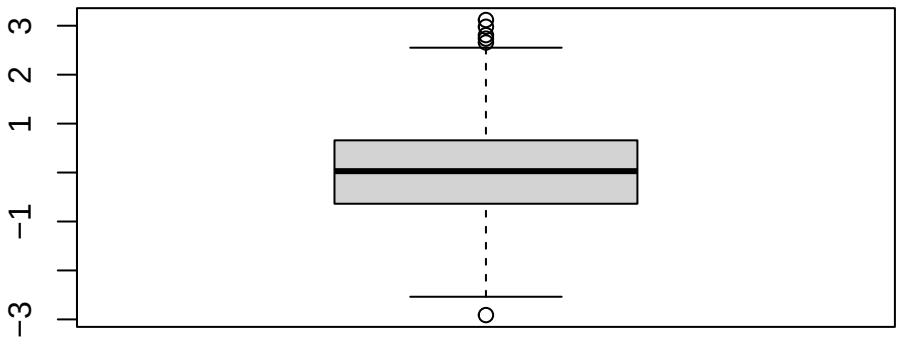


There are lots of other commands you can incorporate as well if you so choose; I recommend checking sites like this one.

7.4 Boxplots

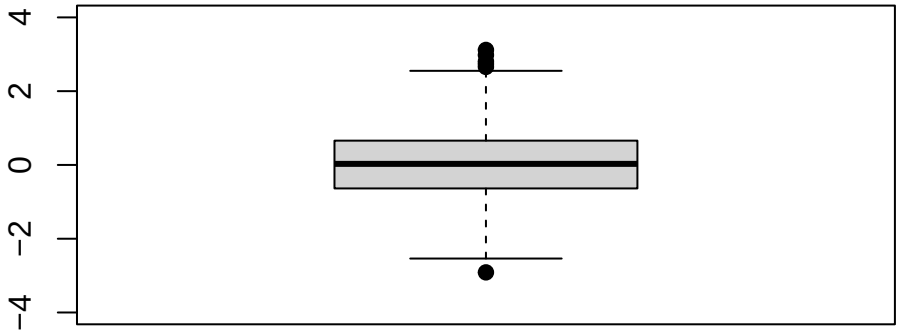
We can also create boxplots to visualize the spread of the data. Boxplots include a bar for the median, a box representing the interquartile range between the 25th and 75th percentiles, and whiskers that extend $1.5 \cdot IQR$ beyond the 25th and 75th percentiles. We can create a boxplot using the command `boxplot`.

```
# using pre-declared variable x

boxplot(x)
```



We can set the axis limits manually as well.

```
boxplot(x, # what to plot
        ylim = c(-4, 4), # set y limits
        pch = 19) # make dots solid
```



On the above plot, *outliers* for the dataset are shown as dots beyond the ends of the "whiskers".

7.5 Skewness

Skew is a measure of how much a dataset "leans" to the positive or negative directions (*i.e.*, to the "left" or to the "right"). To calculate skew, we are going to use the `moments` library.

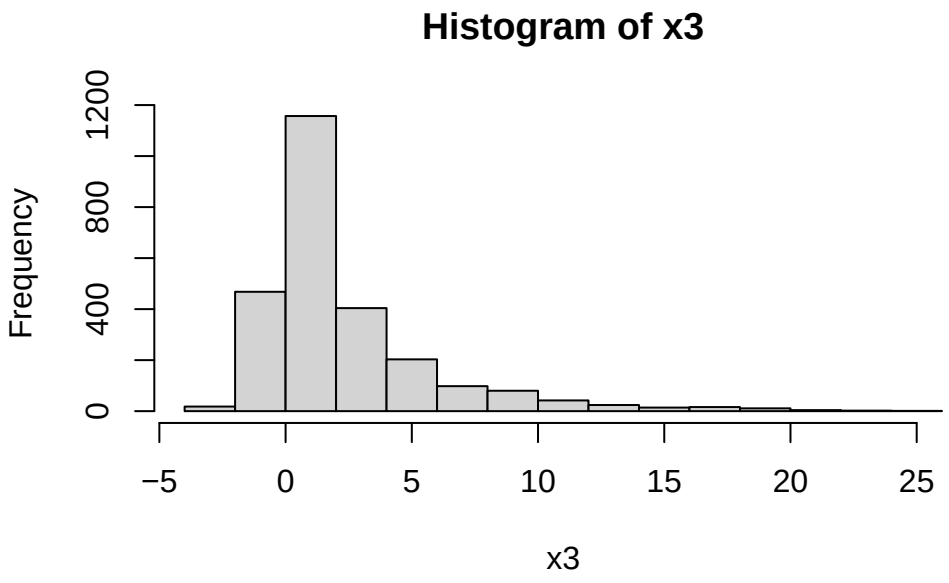
```
# don't forget to install if needed!
library(moments)

skewness(x)
```

```
[1] 0.1461133
```

Generally, a **value between** -1 **and** $+1$ **for skewness is "acceptable"** and not considered overly skewed. Positive values indicate "right" skew and negative values indicate a "left" skew. If something is too skewed, it may violate assumptions of normality and thus need *non-parametric* tests rather than our standard parametric tests - something we will cover later!

Let's look at a skewed dataset. We are going to artificially create a skewed dataset from our `x` vector.

```r
# create more positive values
x3 <- c(x,
        x[which(x > 0)]*2,
        x[which(x > 0)]*4,
        x[which(x > 0)]*8)

hist(x3)
```




```r
skewness(x3)
```

```
[1] 2.293478
```

As we can see, the above is a heavily skewed dataset with a positive ("right") skew.

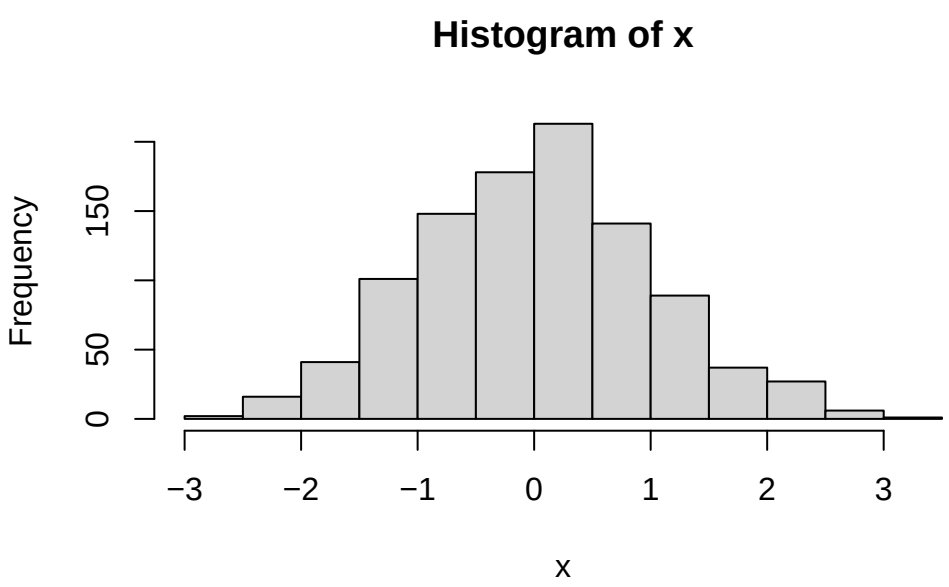
7.6 Kurtosis

Kurtosis refers to how sharp or shallow the peak of the distribution is (*platykurtic* vs. *leptokurtic*). Remember - *platy*kyrtic are *plateau*kurtic, wide and broad like a plateau, and *leptokurtic* distributions are sharp. Intermediate distributions that are roughly normal are *mesokurtic*.

Much like skewness, **kurtosis values of > 2 and < -2 are generally considered extreme**, and thus not mesokurtic. This threshold can vary a bit based on source, but for this class, we will use a threshold of ± 2 for both skewness and kurtosis.

Let's see the kurtosis of `x`. ***Note*** that when doing the equation, a normal distribution actually has a kurtosis of 3; **thus, we are doing kurtosis -3 to "zero" the distribution and make it comparable to skewness**.

```
hist(x)
```




```
# non-zeroed
kurtosis(x)
```

```
[1] 2.932859
```

```
# zeroed
kurtosis(x)-3
```

```
[1] -0.06714111
```

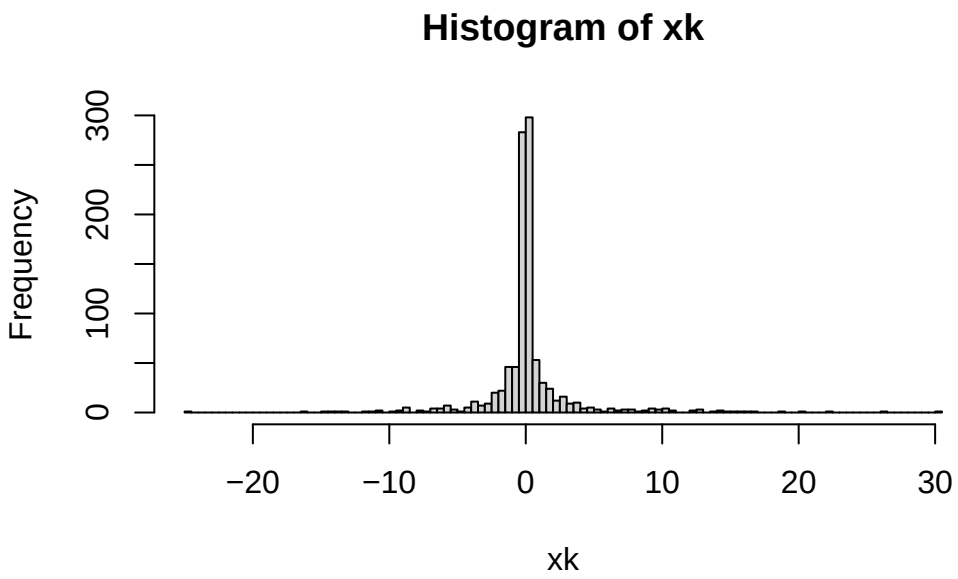
As expected, out values drawn from a normal distribution are not overly skewed. Let's compare these to a more kurtic distribution:

```
xk <- x^3

kurtosis(xk)-3
```

```
[1] 17.3129
```

What does this dataset look like?

```
hist(xk,breaks = 100)
```



As we can see, this is a very *leptokurtic* distribution.

7.7 Cumulative frequency plot

A *cumulative frequency plot* shows the overall spread of the data as a cumulative line over the entire dataset. This is another way to see the spread of the data and is often complementary to a histogram.

The following cumulative distribution plot is based on the method outlined in Geeks for Geeks.

7.7.1 If data are not in histogram/frequency format

You will need to create a frequency table to make them be in histogram format.

```
# declaring the break points
# make break points based on data
# always round UP with ceiling
range.x <- ceiling(max(x)-min(x))

range.x
```

```
[1] 7
```

```
range(x)
```

```
[1] -2.912178  3.117514
```

Based on this range, we need to create our bin sizes. We want our lowest bin to be below our lowest value, and our highest bin above our highest value. Here, I'm setting a step size of 1. **You will have to examine your data and determine the best break size for your datasets.**

```
# make bins based on range
# create sequential series
break_points <- seq(-3, # starting value, low
                    4, # end value, high
                    1) # increment size
```

```
# transforming the data
data_transform = cut(x, # your data!
                     break_points, # the breaks you defined
                     right=FALSE) # closed on the left for intervals
# creating the frequency table
freq_table = table(data_transform) # create a table

# printing the frequency table
print("Frequency Table")
```

```
[1] "Frequency Table"
```

```
print(freq_table)
```

```
data_transform
[-3,-2) [-2,-1)  [-1,0)   [0,1)   [1,2)   [2,3)   [3,4)
     18     142     326     354     126      33       1
```

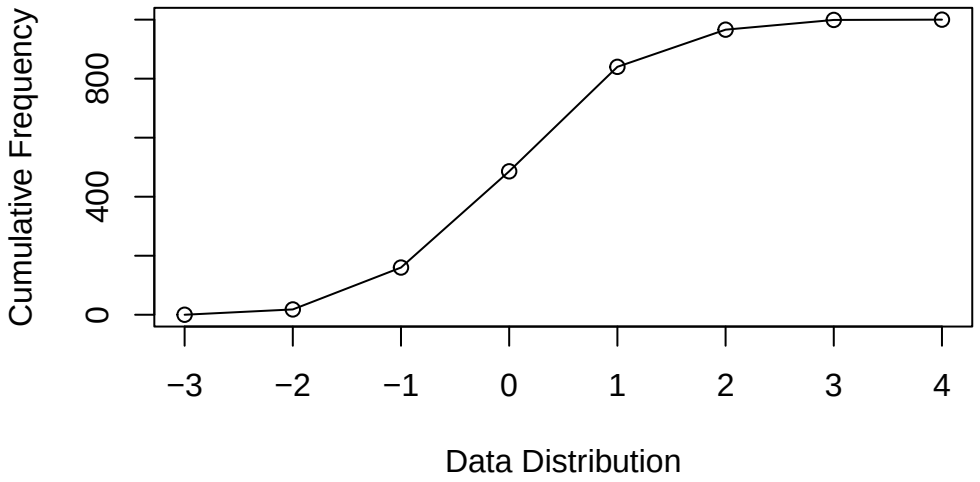
```
# calculating cumulative frequency
cumulative_freq = c(0, # start at 0, no points
                    cumsum(freq_table)) # get the cumulative frequency!
print("Cumulative Frequency")
```

```
[1] "Cumulative Frequency"
```

```
print(cumulative_freq)
```

```
        [-3,-2) [-2,-1)  [-1,0)   [0,1)   [1,2)   [2,3)   [3,4)
      0      18     160     486     840     966     999    1000
```

```
# plotting the data
plot(break_points, # x axis is break_points
     cumulative_freq, # y axis is cumulative frequency
    xlab="Data Distribution", # x axis label
    ylab="Cumulative Frequency") # y axis label
# creating line graph
lines(break_points, # add lines to the graph, this is x
      cumulative_freq) # this is y for lines
```



7.7.2 If data are in histogram/frequency format

If you have a list of frequencies (say, *for river discharge over several years*), you only need to do the `cumsum` function. For example:

```
y <- c(1 ,2 ,4, 8, 16, 8, 4, 2, 1)

sum_y <- cumsum(y)

print(y)
```

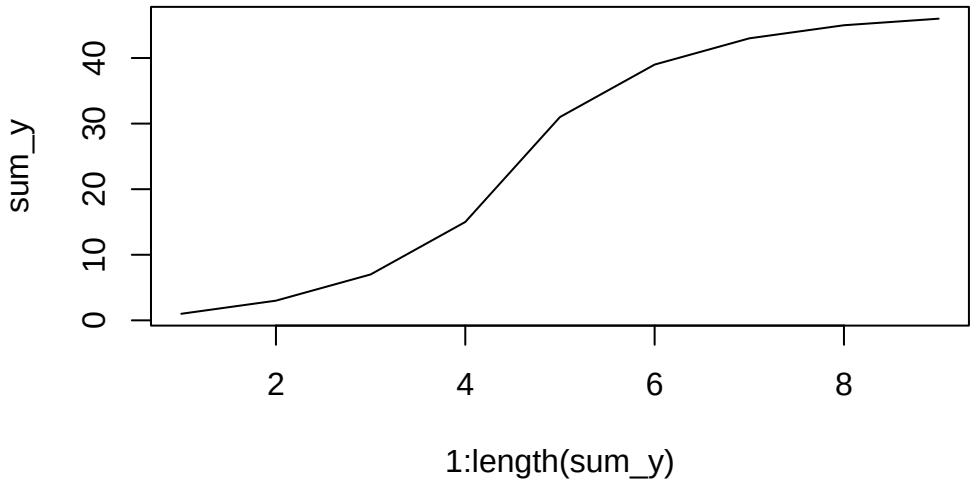
```
[1]  1  2  4  8 16  8  4  2  1
```

```
print(sum_y)
```

```
[1]  1  3  7 15 31 39 43 45 46
```

Now we can see we have out cumulative sums. Let's plot these. **NOTE** that this method will *not* have the x variables match the dataset you started with, it will only plot the curve based on the number of values given.

```
plot(x = 1:length(sum_y), # get length of sum_y, make x index
     y = sum_y, # plot cumulative sums
     type = "l") # make a line plot
```



7.8 Homework: Chapter 3

From your book, complete problems 3.1, 3.4 & 3.5. Data for these problems are available on *Canvas* and in your book.

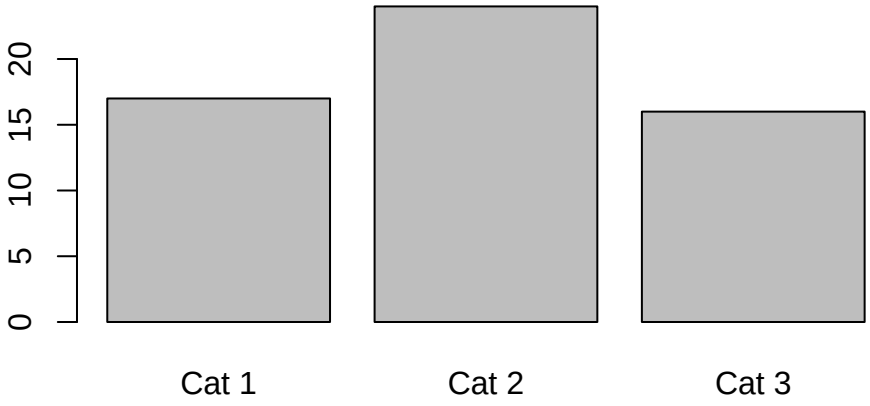
7.8.1 Helpful hint

HINT: For 3.5, consider just making a vector of the values of interest for a histogram.

For example, see the following. For reference:

- `c` means "concatenate", or place things together in an object.

```r
# numeric vector data for counts
y <- c(17,24,16)

# manually create a histogram using barplot
barplot(y,
        # axis names must be true
        axisnames = T,
        # input names here
        # each category as a separate quoted character string
        names.arg = c("Cat 1", "Cat 2", "Cat 3"))
```



7.8.2 Directions

Please complete all computer portions in an `rmarkdown` document knitted as an html. Upload any "by hand" calculations as images in the HTML or separately on *Canvas*.

7.9 Addendum

With thanks to Hernan Vargas & Riley Grieser for help in formatting this page. Additional comments provided by BIOL 305 classes.

8 Normality & hypothesis testing

8.1 Normal distributions

A *standard normal distribution* is a mathematical model that describes a commonly observed phenomenon in nature. When measuring many different kinds of datasets, the data being measured often becomes something that resembles a standard normal distribution. This distribution is described by the following equation:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

This equation is fairly well defined by the *variance* (σ^2), the overall spread of the data, and by the *standard deviation* (σ), which is defined by the square root of the variance.

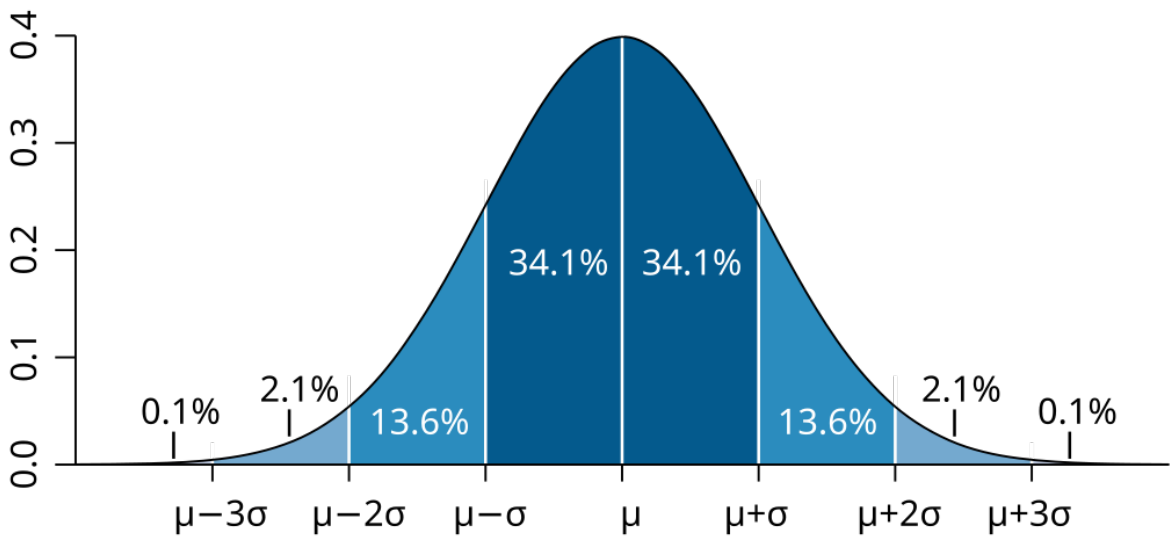


Figure 8.1: A standard normal distribution, illustrating the percentage of area found within each standard deviation away from the mean. By Ainali on Wikipedia; CC-BY-SA 3.0.

Standard normal distributions have a mean, median, and mode that are *equal*. The standard normal distribution is a *density function*, and we are interested in the "area under the curve" (AUC) to understand the relative probability of an event occurring. At the mean/median/mode, the probability on either side of the distribution is 50%. When looking at a normal distribution distribution, it is impossible to say the probability of a specific event occurring, but it is possible to state the probability of an event *as extreme or more extreme than the event observed* occurring. This is known as the p value.

8.1.1 Example in nature

In order to see an example of the normal distribution in nature, we are going to examine the BeeWalk survey database from the island of Great Britain (Comont 2020). We are not interested in the bee data at present, however, but in the climatic data from when the surveys were performed.

```
beewalk <- curl("https://figshare.com/ndownloader/files/44726902") %>%
  read_csv()
```

```
Rows: 306550 Columns: 49
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (30): Website.ID, Website.RecordKey, SiteName, Site.section, ViceCounty,...
dbl (19): RecordKey, established, Precision, Transect.lat, Transect.long, tr...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Note that this is another massive dataset - 306,550 rows of data!

The dataset has the following columns:

```
colnames(beewalk)
```

```
 [1] "RecordKey"            "Website.ID"           "Website.RecordKey"
 [4] "SiteName"             "Site.section"         "ViceCounty"
 [7] "established"          "GridReference"        "Projection"
[10] "Precision"            "Transect.lat"         "Transect.long"
[13] "transect.OS1936.lat"  "Transect.OS1936.long" "transect_length"
[16] "section_length"       "section_grid_ref"     "H1"
[19] "H2"                   "H3"                   "H4"
[22] "habitat_description"  "L1"                   "L2"
```

```
[25] "land_use_description" "start_time"           "end_time"
[28] "sunshine"             "wind_speed"           "temperature"
[31] "TaxonVersionKey"      "species"              "latin"
[34] "queens"               "workers"              "males"
[37] "unknown"              "Comment"              "transect_comment"
[40] "flower_visited"       "StartDate"            "EndDate"
[43] "DateType"             "Year"                 "Month"
[46] "Day"                  "Sensitive"            "Week"
[49] "TotalCount"
```

We are specifically interested in `temperature` to determine weather conditions. Let's see what the mean of this variable is.

```
mean(beewalk$temperature)
```

```
[1] NA
```

Hmmm… we are getting an `NA` value, indicating that not every cell has data recorded. Let's view `summary`.

```
summary(beewalk$temperature)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00   16.00   19.00   18.65   21.00   35.00   16151
```

As we can see, 16,151 rows do not have temperature recorded! We want to remove these `NA` rows, which we can do by using using `na.omit`.

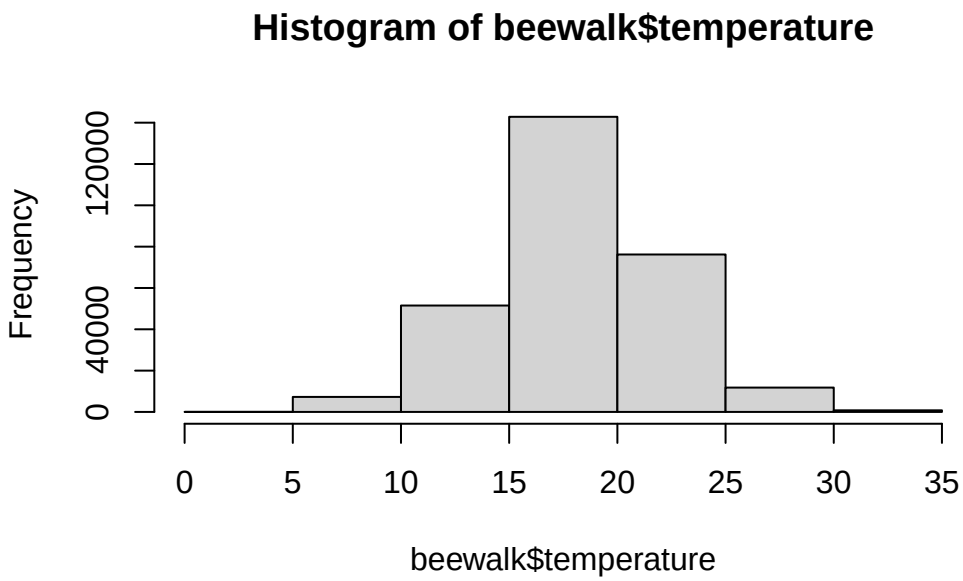
```
beewalk$temperature %>%
  na.omit() %>%
  mean() %>%
  round(2) # don't forget to round!
```

```
[1] 18.65
```

Now we can record the mean.

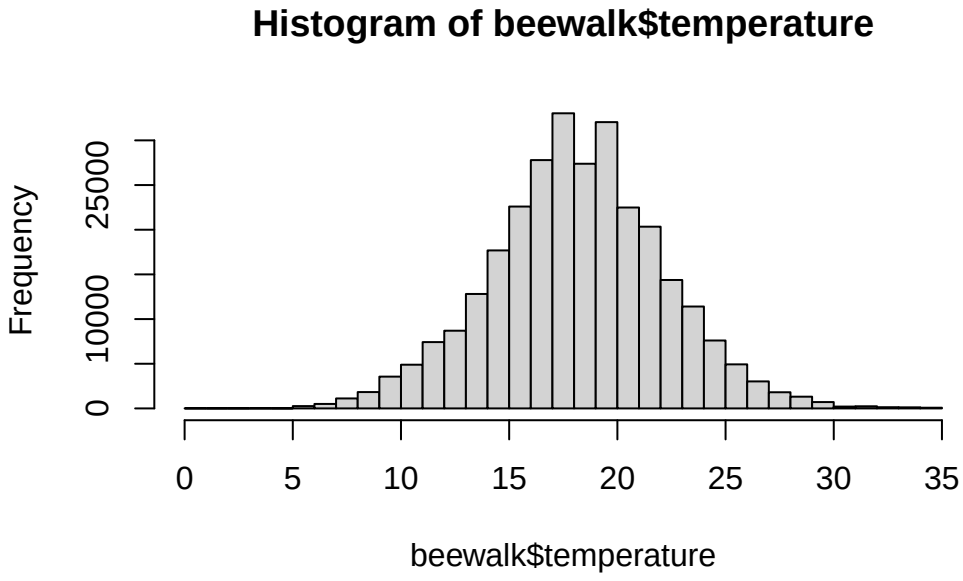
Let's visualize these data using a histogram. *Note* I do not use `na.omit` as the `hist` function automatically performs this data-cleaning step!

```
hist(beewalk$temperature,breaks = 5)
```



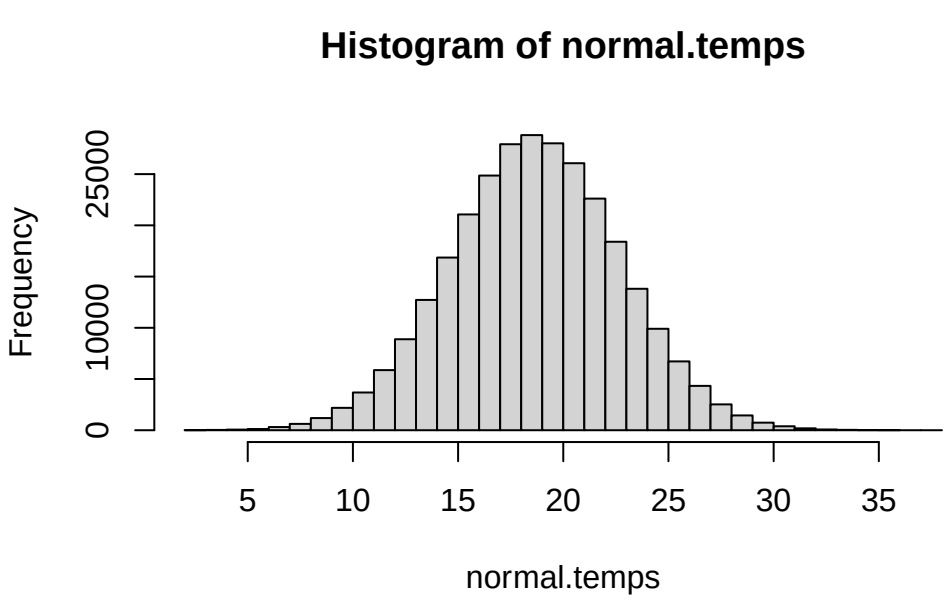
Even with only five breaks, we can see an interesting, normal-esque distribution in the data. Let's refine the bin number.

```
hist(beewalk$temperature,breaks = 40)
```



With forty breaks, the pattern becomes even more clear. Let's see what a *standard normal distribution* around these data would look like.

```r
# save temperature vector without NA values
temps <- beewalk$temperature %>% na.omit()

mu <- mean(temps)
t.sd <- sd(temps)

# sample random values
normal.temps <- rnorm(length(temps), # sample same size vector
                      mean = mu,
                      sd = t.sd)

hist(normal.temps, breaks = 40)
```



As we can see, our normal approximation of temperatures is not too dissimilar from the distribution of temperatures we actually see!

Let's see what kind of data we have for temperatures:

```r
# load moments library
library(moments)

skewness(temps)
```

```
[1] 0.02393257
```

Data do not have any significant skew.

```
kurtosis(temps)-3
```

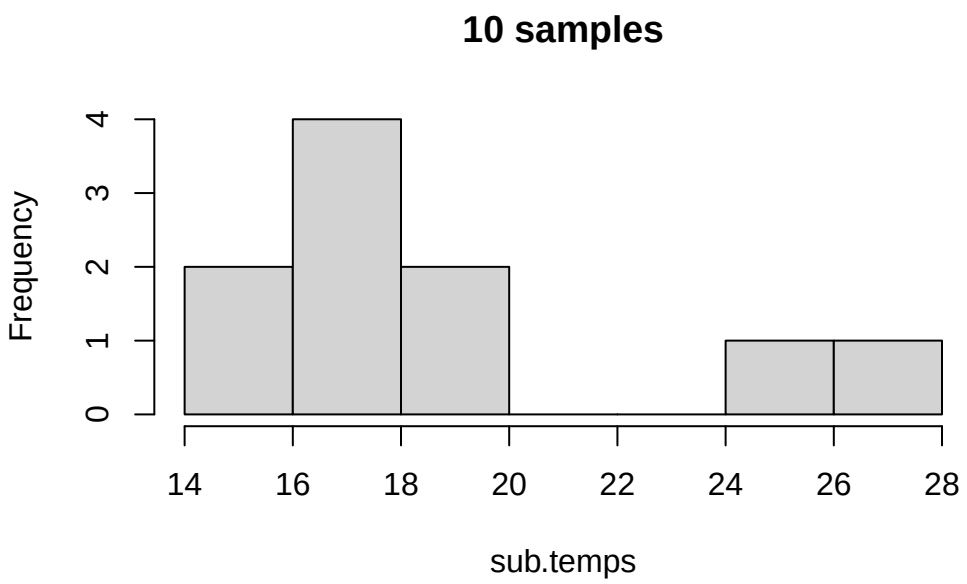
```
[1] 0.3179243
```

Data do not show any significant kurtosis.

8.1.2 Effect of sampling

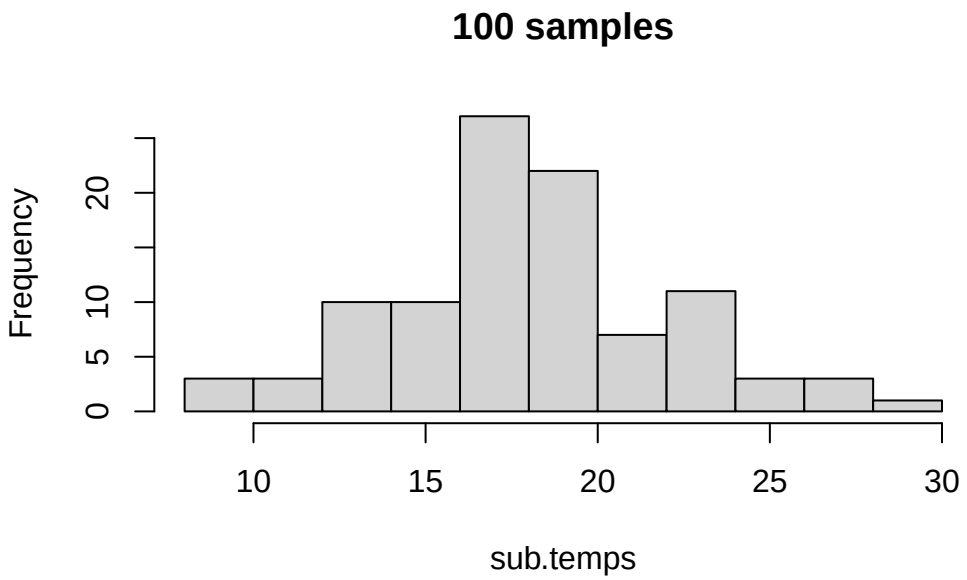
Oftentimes, we will see things approach the normal distribution as we collect more samples. We can model this by subsampling our temperature vector.

```
# make reproducible
set.seed(1839)

sub.temps <- sample(temps,
                    size = 10,
                    replace = FALSE)

hist(sub.temps, main = "10 samples")
```



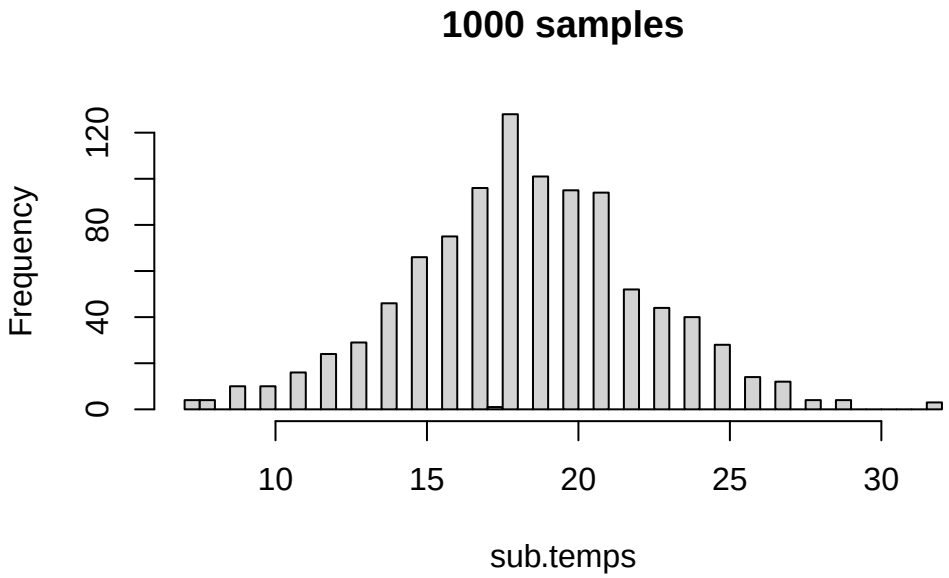
With only ten values sampled, we do not have much of a normal distribution. Let's up this to 100 samples.

```r
sub.temps <- sample(temps,
                    size = 100,
                    replace = FALSE)

hist(sub.temps, main = "100 samples",breaks = 10)
```



Now we are starting to see more of a normal distribution! Let's increase this to 1000 temperatures.

```r
sub.temps <- sample(temps,
                    size = 1000,
                    replace = FALSE)

hist(sub.temps, main = "1000 samples", breaks = 40)
```



Now the normal distribution is even more clear. As we can also see, the more we sample, the more we approach the true means and distribution of the actual dataset. Because of this, we can perform experiments and observations of small groups and subsamples and make inferences about the whole, given that most systems naturally approach statistical distributions like the normal!

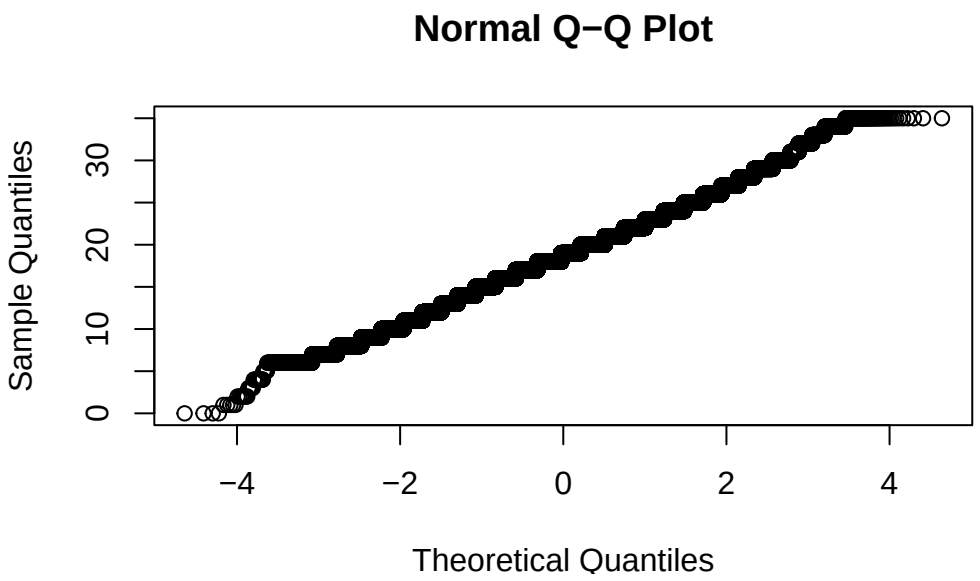
8.1.3 Testing if data are normal

There are two major methods we can use to see if data are normally distributed.

8.1.3.1 QQ Plots

Another way to see if data are normal is to use a QQ plot. These plots data quantiles to theoretical quantiles to see how well they align, with a perfectly normal distribution having a completely linear QQ plot. Let's look at these with our `beewalk` data.

```
qqnorm(temps)
```



As we can see above, the data are roughly linear, which means are data are very normal. The "stairsteps" are from the accuracy in measuring temperature, which was likely rounded and thus created a distribution that is not completely continuous.

8.1.3.2 Shapiro-Wilk test

Another way to test for normality is to use a Shapiro-Wilk test of normality. We will not get into the specific of this distribution, but this tests the null hypothesis that data originated in a normal distribution, with the alternative hypothesis that the data originated in a non-normal distribution. You will read next about the specifics of hypothesis testing, but this test uses an $\alpha = 0.05$, and we *reject the null hypothesis* if our $p < \alpha$, with p representing the probability of observing something as extreme or more extreme than the result we observe. Our `temps` dataset is too large, as `shapiro.test` requires vectors of < 5000. Let's take a random sample and try again.

```
sample(temps, 200) %>%
  shapiro.test()
```

```
    Shapiro-Wilk normality test

data:  .
W = 0.98922, p-value = 0.1371
```

For our subsets of temperature, we find that the temperature is normally distributed. **Note** that having all 5000 temperatures included makes this test find a "non-normal" result, likely from the same stairstepping phenomenon from rounding that we discussed before, which may result in model overfitting to the rounded data.

8.2 Hypothesis testing

Since we can define specific areas under the curve within these distributions, we can look at the percentage of area within a certain bound to determine how likely a specific outcome would be. Thus, we can begin to test what the *probability of observing an event* is within a theoretical, probabilistic space. A couple of important conceptual ideas:

1. We may not be able to know the probability of a specific event, but we can figure out the probability of events more extreme or less extreme as that event.
2. If the most likely result is the mean, then the further we move away from the mean, the less likely an event becomes.
3. If we look *away* from the mean at a certain point, then the area represents the chances of getting a result *as extreme or more extreme than what we observe*. This probability is known as the p value.

Once we have a p value, we can make statements about the event that we've seen relative to the overall nature of the dataset, but we do not have sufficient information to declare if this result is *statistically significant*.

8.2.1 Critical Values - α

In order to determine if something is significant, we compare things to a *critical value*, known as α. This value is traditionally defined as 0.05, essentially stating that we deem an event as significant if 5% or fewer of observed or predicted events are as extreme or more extreme than what we observe.

Your value should always set your α critical value before you do your experiments and analyses.

Our critical value of α represents our criterion for *rejecting the null hypothesis*. We set our α to try to minimize the chances of error.

Type I Error is also known as a **false-positive**, and is when we **reject the null hypothesis when the null is true**.

Type II Error is also known as a **false-negative**, and is when we **support the null hypothesis when the null is false**.

By setting an α, we are creating a threshold of probability at which point we can say, with confidence, that results are different.

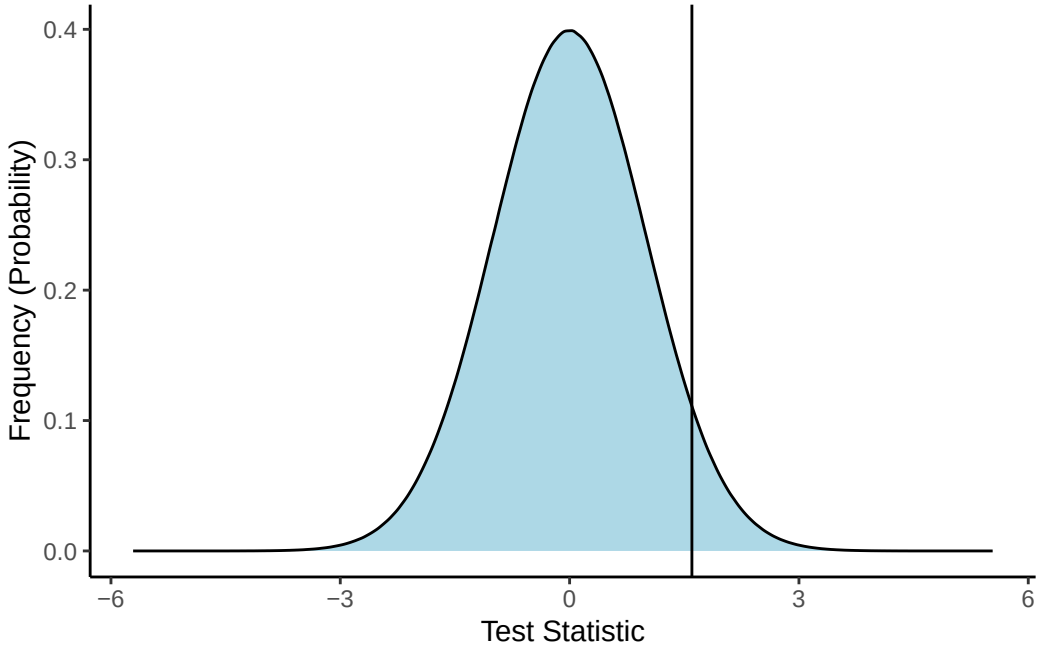
8.2.2 Introduction to p values

Let's say that we are looking at a dataset defined by a standard normal distribution with $\mu = 0$ and $\sigma = 1$. We draw a random value, x, with $x = 1.6$. What is the probability of drawing a number this extreme or more extreme from the dataset?

First, let's visualize this distribution:

```
###THIS WILL TAKE A WHILE TO RUN###

# create gigantic normal distribution dataset
# will be essentially normal for plotting
# rnorm gets random values
x <- rnorm(100000000)

# convert to data frame
x <- as.data.frame(x)
# rename column
colnames(x) <- c("values")

# thank you stack overflow for the following
# Creating density plot
p = ggplot(x,
           aes(x = values)
          ) +
  # generic density plot, no fill
  geom_density(fill="lightblue")

# Building shaded area
# create new plot object
p2  <-  p + # add previous step as a "backbone"
  # rename axes
  geom_vline(xintercept = 1.6) +
  xlab("Test Statistic") +
  ylab("Frequency (Probability)") +
  # make it neat and tidy
  theme_classic()

# plot it
```

```r
# can use ggsave function to save
plot(p2)
```

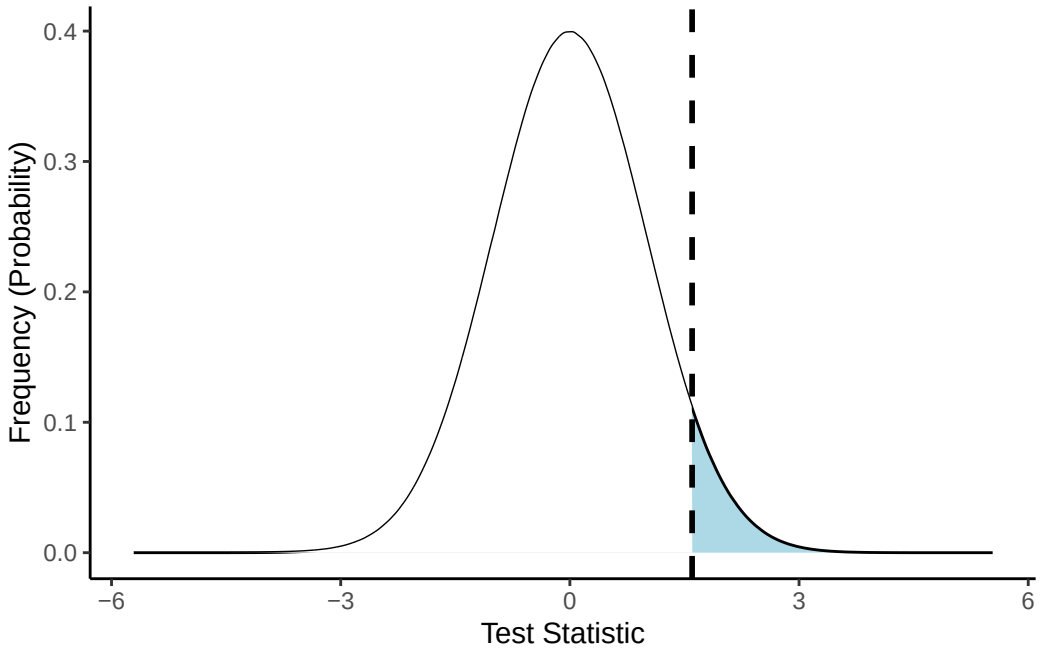


Above, the solid black line represents x, with the illustrated standard normal distribution being filled in blue.

Let's see how much of the area represents values *as extreme or more extreme* as our value x.

```r
### THIS WILL TAKE A WHILE TO RUN ###

# Getting the values of plot
# something I wasn't familiar with before making this!
d  <-  ggplot_build(p)$data[[1]]

# Building shaded area
# create new plot object
p2  <-  p + # add previous step as a "backbone"
  # add new shaded area
  geom_area(data = subset(d, x < 1.6), # select area
            # define color, shading intensity, etc.
            aes(x=x,y=y), fill = "white", alpha = 1) +
  # add value line
  geom_vline(xintercept = 1.6, colour = "black",
             linetype = "dashed",linewidth = 1) +
```

```
  # rename axes
  xlab("Test Statistic") +
  ylab("Frequency (Probability)") +
  # make it neat and tidy
  theme_classic()

# plot it
# can use ggsave function to save
plot(p2)
```



Now we can see that it is only a portion of the distribution *as extreme or more extreme* than the value we placed on the graph. The area of this region is our p value. This represents the *probability of an event as extreme or more extreme occurring* given the random variation observed in the dataset or in the distribution approximating the dataset. This is the value we compare to α - our threshold for rejecting the null hypothesis - to determine whether or not we are going to reject the null hypothesis.

Let's look at the above graph again, but let's visualize a two-tailed α around the mean with a 95% confidence interval. First, we need to get the Z scores for our $\alpha = 0.05$, which we calculate by taking $\frac{\alpha}{2}$ to account for the two tails. (Two tails essentially meaning we reject the null mean if we see things *greater than* or *less than* our expected value to a significant extent). We can calculate Z scores using `qnorm`.

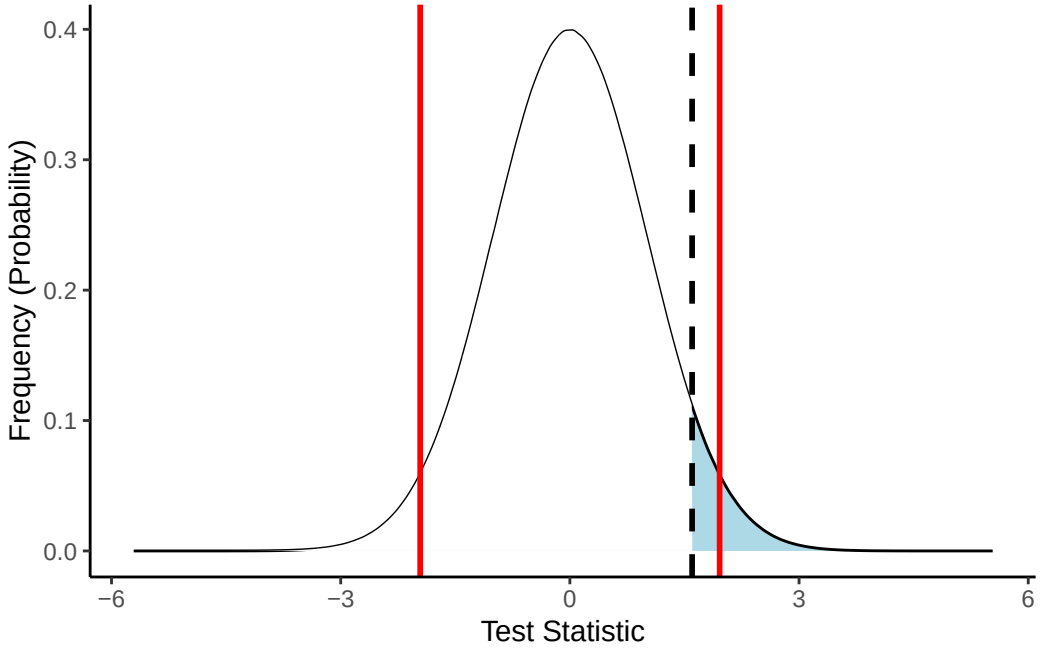
```r
low_alpha <- qnorm(0.025) # looks left
hi_alpha <- qnorm(0.975) # looks left

print(paste0("Our Z scores are: ",
             round(low_alpha,2),
             " & ",
             round(hi_alpha,2)))
```

```
[1] "Our Z scores are: -1.96 & 1.96"
```

The above values make sense, given the distribution is symmetrical. Our above dashed line is as $Z = 1.6$, which means we should have a $p = 0.05$, so the dashed line should be *closer* to the mean than our cutoffs.

```r
### THIS WILL TAKE A WHILE TO RUN ###

# Getting the values of plot
# something I wasn't familiar with before making this!
d  <-  ggplot_build(p)$data[[1]]

# Building shaded area
# create new plot object
p2  <-  p + # add previous step as a "backbone"
  # add new shaded area
  geom_area(data = subset(d, x < 1.6), # select area
            # define color, shading intensity, etc.
            aes(x=x,y=y), fill = "white", alpha = 1) +
  # add value line
  geom_vline(xintercept = 1.6, colour = "black",
             linetype = "dashed",linewidth = 1) +
  geom_vline(xintercept = low_alpha, colour = "red",
             linetype = "solid",linewidth = 1) +
  geom_vline(xintercept = hi_alpha, colour = "red",
             linetype = "solid",linewidth = 1) +
  # rename axes
  xlab("Test Statistic") +
  ylab("Frequency (Probability)") +
  # make it neat and tidy
  theme_classic()

# plot it
```

```
# can use ggsave function to save
plot(p2)
```



Exactly as we calculated, we see that our $p < \alpha$ and thus we do not see an area (in blue) less than the area that would be further than the mean as defined by the red lines.

8.2.3 Calculating a Z score

When we are trying to compare our data to a normal distribution, we need to calculate a Z score for us to perform the comparison. A Z score is essentially a measurement of the number of standard deviations we are away from the mean on a standard normal distribution. The equation for a Z score is:

$$Z = \frac{\bar{x} - \mu}{\frac{\sigma}{\sqrt{n}}}$$

Where $\bar{x}$ is either a sample mean or a sample value, μ is the population mean, σ is the population standard deviation and n is the number of individuals in your sample (1 if comparing to a single value).

We can calculate this in R using the following function:

```r
zscore <- function(xbar, mu, sd.x, n = 1){
  z <- (xbar - mu)/(sd.x/sqrt(n))
  return(z)
}
```

NOTE that is the above isn't working for you, *you have a mistake somewhere in your code.* Try comparing - character by character - what is listed above to what you have.

Let's work through an example, where we have a sample mean of 62 with 5 samples compared to a sample mean of 65 with a standard deviation of 3.5.

```r
Z <- zscore(xbar = 62, # sample mean
            mu = 65, # population mean
            sd.x = 3.5, # population standard deviation
            n = 5) # number in sample

print(Z)
```

```
[1] -1.91663
```

Now, we can calculate the p value for this Z score.

```r
pnorm(Z)
```

```
[1] 0.0276425
```

After rounding, we get $p = 0.03$, a p that is significant if for a one-tailed $\alpha = 0.05$ but insignificant for a two-tailed $\alpha = 0.05$.

8.2.4 Calculated the p value

We have two different methods for calculating a p value:

8.2.4.1 Comparing to a z table

We can compare the z value we calculate to a z table, such as the one at ztable.net. On this webpage, you can scroll and find tables for positive and negative z scores. *Note* that normal distributions are symmetrical, so you can also transform from negative to positive to get an idea of the area as well. Given that a normal distribution is centered at 0, a z score of 0 will have a p value of 0.50.

On the z tables, you will find the tenths place for your decimal in the rows, and then go across to the columns for the hundredths place. For example, go to the website and find the p value for a z score of -1.33. You should find the cell marked 0.09176. *Note* the website uses naked decimals, which we do not use in this class.

For values that aren't on the z table, we can approximate its position between different points on the z table or, if it is extremely unlikely, denote that $p < 0.0001$.

8.2.4.2 Using *R*

In *R*, we can calculate a p value using the function `pnorm`. This function uses the arguments of `p` for our p value, `mean` for the mean of our distribution, `sd` for the standard deviation of our distribution, and also information on whether we want to log-transform p or if we are testing a specific hypothesis (lower tail, upper tail, or two-tailed). The function `pnorm` defaults to a standard normal distribution, which would be a z score, but it can also perform the z transformations for us if we define the mean and standard deviation.

For example, if we have a z score of -1.33:

```
pnorm(-1.33)
```

```
[1] 0.09175914
```

As we can see, we get the same result as our z table, just with more precision!

There are other functions in this family as well in *R,* including `dnorm` for quantiles, `qnorm` for determining the z score for a specific p value, and `rnorm` for getting random values from a normal distribution with specific dimensions. For now, we will focus on `pnorm`.

8.2.5 Workthrough Example

When this class was being designed, Hurricane Milton was about to make contact with Florida. Hurricane Milton is considered one of the strongest hurricanes of all time, so we can look at historical hurricane data to determine just how powerful this storm really was. We can get information on maximum wind speeds of all recorded Atlantic hurricanes as of 2024 from Wikipedia.

```
hurricanes <- read_csv("https://raw.githubusercontent.com/jacobccooper/biol305_unk/main/assig
```

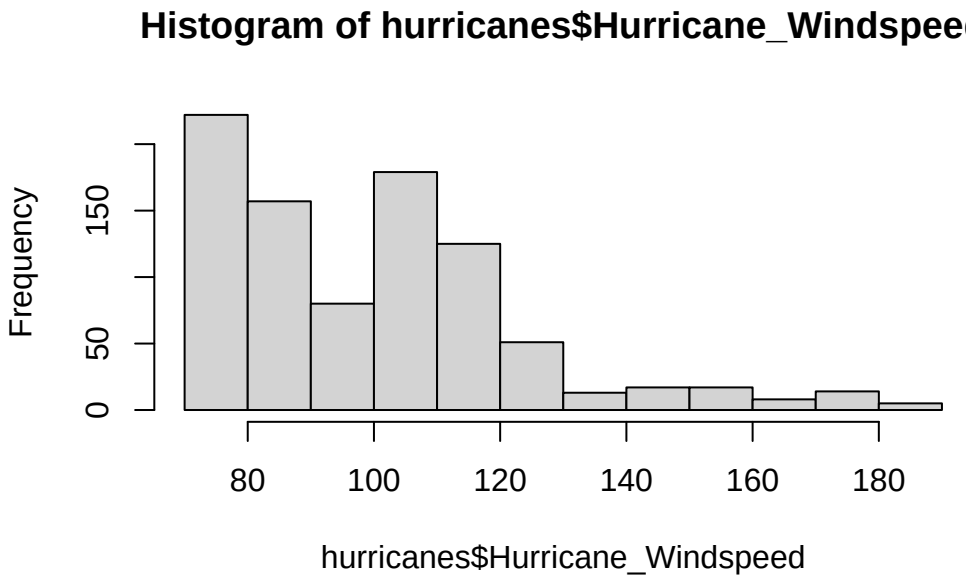
```
Rows: 889 Columns: 1
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (1): Hurricane_Windspeed

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Above, we have loaded a `.csv` of one column that has all the hurricane speeds up to 2024. Hurricane Milton is the last row - the most recent hurricane. Let's separate this one out. We will use `[ , ]`, which defines `[rows,columns]` to subset data.

```
milton <- hurricanes$Hurricane_Windspeed[nrow(hurricanes)]

other_hurricanes <- hurricanes[-nrow(hurricanes),]
```

We want to compare the windspeed of Milton (180 mph) to the overall distribution of hurricane speeds. We can visualize this at first.

```
# all hurricanes
hist(hurricanes$Hurricane_Windspeed)
```



Windspeeds are more towards the lower end of the distribution, with strong storms being rarer.

For the sake of this class, we will assume we can use a normal distribution for these data, but if we were doing an official study we would likely need to use a non-parametric test (we will cover these later, but they cover non-normal data).

```
mu <- mean(other_hurricanes$Hurricane_Windspeed)

mu
```

```
[1] NA
```

Hmmm… we need to use `na.omit` to be sure we do this properly.

```
other_hurricanes_windspeed <- na.omit(other_hurricanes$Hurricane_Windspeed)

mu <- mean(other_hurricanes_windspeed)
mean(other_hurricanes_windspeed)
```

```
[1] 102.5254
```

Next, we need the standard deviation.

```
sd.hurricane <- sd(other_hurricanes_windspeed)

sd.hurricane
```

```
[1] 23.08814
```

Now, we can calculate our Z value.

```
Z <- (milton - mu)/sd.hurricane

Z
```

```
[1] 3.355603
```

How significant is this?

```
pnorm(Z)
```

```
[1] 0.999604
```

This is greater than 0.5, so we need to do $1 - p$ to figure things out.

```
1 - pnorm(Z)
```

```
[1] 0.000395961
```

This rounds to 0.0004, which means that this is an *extremely* strong hurricane.

8.2.5.1 Non-normality, for those curious

We can do a Shapiro-Wilk test of normality to see if this dataset is normal.

```
shapiro.test(other_hurricanes_windspeed)
```

```
    Shapiro-Wilk normality test

data:  other_hurricanes_windspeed
W = 0.88947, p-value < 2.2e-16
```

A $p < 0.05$ indicates that these data are *non-normal*.

We can do a Wilcoxon-Test since these data are extremely non-normal.

```
wilcox.test(other_hurricanes_windspeed,
            milton)
```

```
    Wilcoxon rank sum test with continuity correction

data:  other_hurricanes_windspeed and milton
W = 6.5, p-value = 0.08678
alternative hypothesis: true location shift is not equal to 0
```

Using non-normal corrections, we find that this is *not* an extremely strong hurricane, but it is near the upper end of what we would consider "normal" under historical conditions. Still an extremely bad hurricane!

8.3 Confidence Intervals

Because we can figure out the probability of an event occurring, we can also calculate *confidence intervals*. A *confidence interval* provides a range of numbers around a value of interest that indicates where we believe the mean of a population lies and our confidence that it lies within that range. **Note** that nothing is ever 100% certain, but this helps us determine where a mean is and demonstrates our confidence in our results.

Specifically, if the tails of the distribution, our α, are 0.05, then we have an area of 0.95 around the mean where we do not reject results. Another perspective on this area is that we can say with 95% certainty that a mean is within a certain area, and that if values fall within that confidence area then we do not reject the null hypothesis that the means are equal.

We will cover several different ways to calculate confidence intervals, but for normal distributions, we use the following equation, with the 0.95 confidence interval shown as an example:

$$CI = \bar{x} \pm Z_{1-\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}}$$

This interval gives us an idea of where the mean should lie. For example, if we are looking at the aforemention `beewalk` temperature data, we can calulate a 0.95 confidence interval around the mean.

```r
temps <- na.omit(beewalk$temperature)

xbar <- mean(temps)
n <- length(temps)
sdtemp <- sd(temps)
# Z for 0.95 as P, so for 0.975, 0.025
# get value from P
Z <- qnorm(0.975)

CI <- Z*(sdtemp/sqrt(n))

CI
```

```
[1] 0.01458932
```

We have a very narrow confidence zone, because we have so many measurements. Let's round everything and present it in a good way.

If I want numbers to show up *in text* in RMarkdown, I can add code to a line of plain text using the following syntax:

```r
# DO NOT RUN
# Format in plaintext
`r xbar`
```

```
Typing that into the plaintext should render as the following: `r xbar`. Then I can also type my answer as follows:

The $95$% Confidence Interval for the mean for this temperature dataset is `r round(xbar,2)` $\pm$ `r round(CI,2)`.
```

Figure 8.2: This is the "coded" version of the text below. Compare the above window to the text below this image.

Typing that into the plaintext should render as the following: 18.645963. Then I can also type my answer as follows:

The 95% Confidence Interval for the mean for this temperature dataset is 18.65 ± 0.01.

Note that the above is rounded to two decimal places to illustrative purposes ONLY, and should be rounded to one decimal place if it was a homework assignment because the original data has only one decimal place.

8.4 Homework: Chapter 8

Please complete problems 8.1, 8.2, 8.3 & 8.6. Follow the directions as written in the book. Submit one `html` file, as derived from *RStudio*. For maximum clarity, create headings to separate your problems. (Remember, a header can be invoked by placing '`#`' in front of a line of text. For example: the header here is written as `# Homework: Chapter 8`).

9 Exam 2 practice

9.1 Exam 2 Practice

The following is practice for the exam. Please work through these problems and be ready to discuss them in class.

9.2 Question 1: Cyclones

Consider this short data set:

Latitude band	Season	Number of cyclones
40 - 49° S	Fall	370
40 - 49° S	Winter	452
40 - 49° S	Spring	273
40 - 49° S	Summer	422
50 - 59° S	Fall	526
50 - 59° S	Winter	624
50 - 59° S	Spring	513
50 - 59° S	Summer	1,059
60 - 69° S	Fall	980
60 - 69° S	Winter	1,200
60 - 69° S	Spring	995
60 - 69° S	Summer	1,751

All of the data was collected in the same year to determine whether the occurrences of cyclones differed by latitude and season around Antarctica. Use this data to answer the following questions.

a) Classify each of the three variables above as either an explanatory or a response variable. Justify your answer.
b) Classify each of the three variables above as either a nominal, ordinal, interval, or ratio variable. Justify your answer.

c) State the null and alternative hypotheses for this scenario. Note: there are two sets of null/alternative hypotheses.
d) Calculate the mean, median, and mode of your response variable. Based on your results, would you expect your data to be normally distributed or not? Justify your answer.
e) Let's say now that you're testing the null hypothesis that the mean number of cyclones in this one year is similar to the average year. The mean number of cyclones per year across all years of data collected is 650 with a standard deviation of 425. Calculate the z-score and make a decision regarding the null hypothesis.

9.3 Question 2: Minnows

Consider this scenario: You have discovered a never-before-documented population of minnow in the Kearney Canal near campus. During your first sampling trip, you notice that the total length (i.e., measured from the tip of the snout to the very tip of the tail) of the fish you measure appear to be smaller than the average total length of the species as recorded among all known individuals across their range. The mean total length noted in one publication is 85 mm with a standard deviation of 4.50. Below is your data the data from 20 minnows that you captured during your first sampling trip to the Kearney Canal:

Fish ID	Length (mm)
1	89
2	75
3	86
4	74
5	69
6	100
7	73
8	69
9	96
10	79
11	61
12	62
13	95
14	98
15	100
16	57
17	70
18	78
19	65
20	65

HINT: Try combining numbers using the `c` command. See the Glossary for more information!

a) State the null and alternative hypothesis for your study.
b) Calculate the median, first and third quartiles, and interquartile range of your response variable. Create a boxplot. Based on your results, are there any outliers in your data? Explain.
c) Calculate the z-score for this scenario. Please show all of your work.
d) What is the probability that, by random chance alone, you would find your observed mean or something more extreme?
e) Assume that you set your α for your study prior to your data collection to 0.05. Based on this information and the p-value you obtained for part c above, is your null hypothesis supported or rejected?

10 Probability distributions

10.1 Probability distributions

We rely on multiple different probability distributions to help us understand what probable outcomes are for a specific scenario. All of the tests that we are performing are comparing our results to what we would expect under perfectly random scenarios. For example, if we are flipping a coin, we are interested in whether the the observation we have of the flips on our coin matches our expectation given the probability of getting heads or tails on a perfectly fair coin. While it is possible to get all heads or all tails on a coin flip, it is highly unlikely and may lead us to believe we have an unfair coin. The more trails we perform, the more confident we can be that out coin is atypical.

We perform similar comparisons for other distributions. If we are comparing sets of events, we can look at the probability of those events occurring if events are occurring randomly. If we are comparing counts, we can compare our counts to our expectation of counts if events or subjects are distributed randomly throughout the matrix or whether two sets of counts are likely under the same sets of assumptions.

Remember, for our specific tests, we are setting an α value in advance (traditionally 0.05, or 5%) against which we compare our p value, with p representing the probability of observing an event *as extreme* or *more extreme* than the event we observe given a specific probability distribution.

Previously, we talked about the *normal distribution*, which is used to approximate a lot of datasets in nature. However, several other probability distributions are also useful for biological systems, which are outlined here.

10.2 Binomial distribution

A *binomial distribution* is one in which only two outcomes are possible - often coded as 0 and 1 and usually representing failure and success, respectively. The binomial is described by the following function:

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

where n = number of trials, x = the number of successes, and p = the probability of a success under random conditions.

In *R*, the binomial distribution is represented by the following functions:

- `dbinom`: the density of a binomial distribution
- `pbinom`: the distribution function, or the probability of a specific observation
- `qbinom`: the value at which a specific probability is found (the *quantile function*)
- `rbinom`: generates random values according to a binomial.

10.2.1 Binomial examples

Let's see what this looks like. Let's consider a scenario where we flip a coin 10 times and get 9 heads. How likely is this outcome?

```
x <- pbinom(q = 9, # number successes, 9 heads
            size = 10, # number of trials, 10 flips
            prob = 0.5) # probability with a fair coin

round(x,4)
```

```
[1] 0.999
```

NOTE that the trailing 0 is dropped, such that the real answer is 0.9990. However, we mentioned before that the p value should be the probability of a result as extreme or more extreme, meaning that it should *always* be less than 0.5. If we are reporting a value of *greater* than 0.5, then we are comparing to the upper tail of the distribution. For a one-tailed α of 0.05, this would mean that we are looking for a value *greater than* 0.95 $(1 - \alpha)$.

So, our real p is:

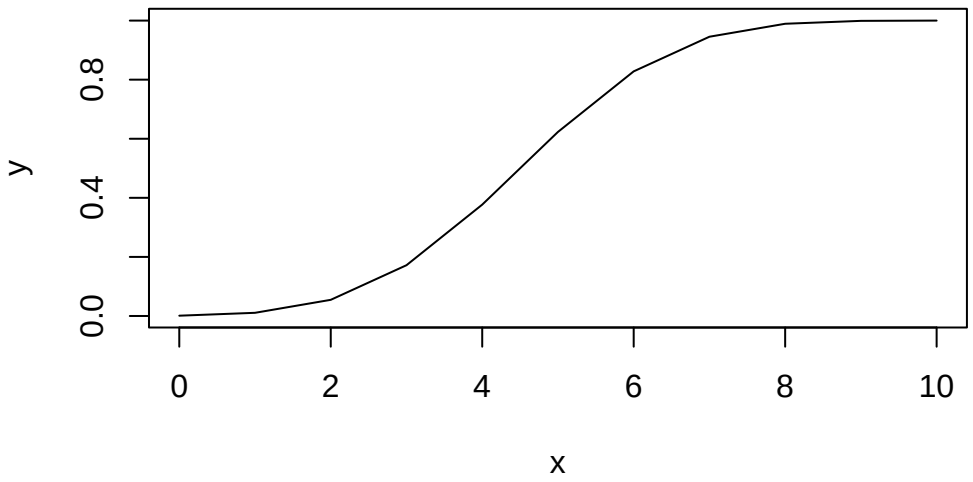
```
1 - round(x,4)
```

```
[1] 0.001
```

Again, the trailing zero is missing. Given that $p < \alpha$, we *reject the null hypothesis* that this is a fair coin.

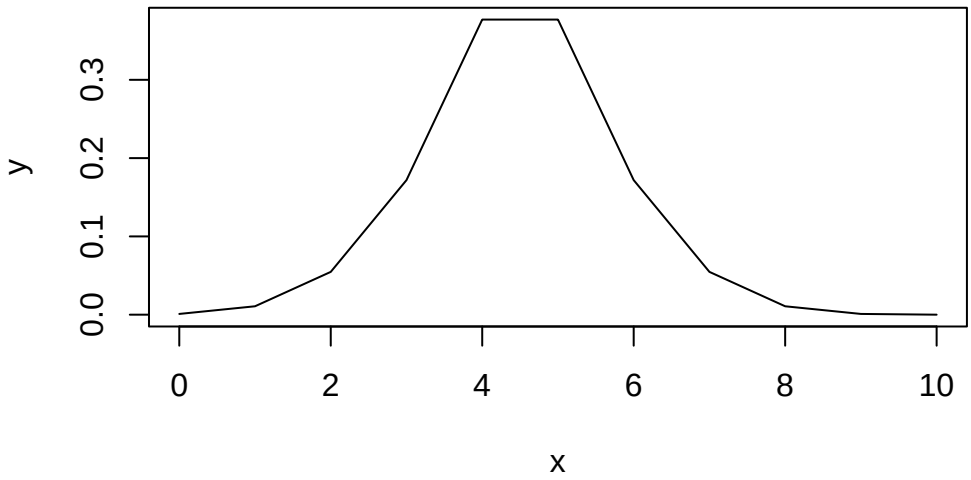
How does this distribution look?

```r
# number of successes
# start at 0 for no heads
x <- 0:10
# cumulative probability to left of outcome
y <- pbinom(x,
            size = 10,
            prob = 0.5,
            lower.tail = T)

# cumulative probability of results to the left
plot(x,
     y,
     type="l") # line plot
```



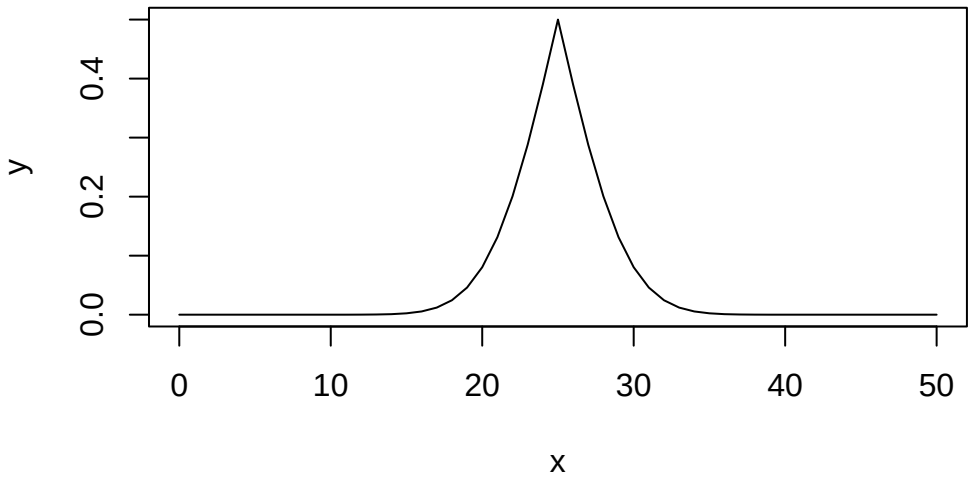
What about if we always have p less than 0.5 to reflect two tails?

```r
# any value greater than 0.5 is subtracted from 1
y[y > 0.5] <- 1 - y[y > 0.5]

plot(x,
     y,
     type="l")
```



What if we do this with a bigger dataset, like for 50 flips?

```
# number of successes
# start at 0 for no heads
x <- 0:50
# cumulative probability to left of outcome
y <- pbinom(x,
            size = length(x),
            prob = 0.5,
            lower.tail = T)

# any value greater than 0.5 is subtracted from 1
y[y > 0.5] <- 1 - y[y > 0.5]

plot(x,
     y,
     type="l")
```



As we increase the number of flips, we can see that the probability of success forms a *normal distribution* centered on the outcome given the default probability. Thus, as we deviate from our *expected outcome* (initial probability multiple by the number of trials), then our results become less likely.

10.2.2 Binomial exact tests

We can perform exact binomial tests by using the *R* function `binom.test`. This is a built in function within *R*. This test requires the following arguments:

- `x`: number of successes (success = outcome of interest)
- `n`: number of trials (number of events)
- `p`: probability of success in a typical situation (*i.e.*, for a fair coin, this is 50%)
- `alternative`: the hypothesis to be tested, whether `two.sided`, `greater`, or `less`.
- `conf.level` is the *confidence level* to be returned; default is 95%.

Let's say you flip a coin ten times, randomly assigning one side as a "success" and one side as a "failure". We do ten flips, and get 3 "successes". How likely is this outcome?

```
binom.test(x = 3,   # three successes
           n = 10,  # ten flips
           p = 0.5) # 50% chance on fair coin
```

```
    Exact binomial test
```

```
data:  3 and 10
number of successes = 3, number of trials = 10, p-value = 0.3438
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.06673951 0.65245285
sample estimates:
probability of success
                   0.3
```

Now let's say we do 1000 flips, and we get 300 successes.

```
binom.test(x = 300,
           n = 1000,
           p = 0.5)
```

```
    Exact binomial test

data:  300 and 1000
number of successes = 300, number of trials = 1000, p-value < 2.2e-16
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.2717211 0.3294617
sample estimates:
probability of success
                   0.3
```

As we can see, both of these return a confidence interval among other things. If we save the object, we can access these "slots" of data using the $ character.

```
binom_result <- binom.test(x = 3,
                           n = 10,
                           p = 0.5)

binom_result$p.value %>% round(2)
```

```
[1] 0.34
```

```
binom_result$conf.int
```

```
[1] 0.06673951 0.65245285
attr(,"conf.level")
[1] 0.95
```

This test is easily implemented, but always double check and make sure you are setting it up correctly.

10.3 Poisson distribution

The *Poisson distribution* is used to reflect random count data. Specifically, the Poisson is used to determine if success events are *overdispersed* (*i.e.*, regularly spaced), *random*, or *underdispersed* (i.e., *clustered*). The Poisson introduces the variable *lambda* (λ) which represents the mean (μ) and the variance (σ^2), which are equal in a Poisson distribution. A Poisson distribution is described by the following function:

$$p(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

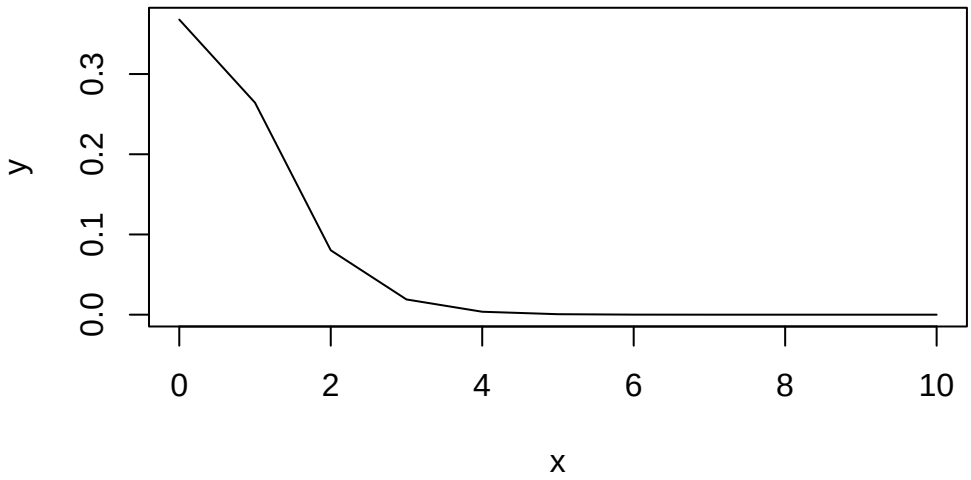
10.3.1 Poisson example

The Poisson is represented by the following functions in *R* which closely resemble the functions for the normal and binomial distributions:

- `dpois`: the log density function
- `ppois`: log distribution (probability) function
- `qpois`: quantile function
- `rpois`: random values from a Poisson.

Let's look at the probability of 0 to 10 successes when we have our $\lambda = 1$.

```
x <- 0:10
y <- ppois(x,lambda = 1)

# any value greater than 0.5 is subtracted from 1
y[y > 0.5] <- 1 - y[y > 0.5]

plot(x,y,type="l")
```



As we can see, the probability of rare events is high, whereas the probability quickly decreases as the number of successes increases.

10.3.2 Poisson test

Much like the Binomial Distribution and its `binom.test`, we can use `poisson.test` to analyze data via a Poisson Distribution. This command uses the arguments:

- `x`: number of events of interest
- `T`: time base (if for an event count)
- `r`: hypothesized rate or ratio
- `alternative` and `conf.level` are the same as for `binom.test`

We will not often use the `poisson.test` in this class, but it is good to be aware of.

10.4 Cumulative Probabilities

With both the Binomial and the Poisson, we can calculate cumulative probabilities. Both of these require the density (`d`) versions of the arguments.

10.4.1 Binomial cumulative

Let's say we have ten trials with three successes and a base probability of 0.5. We can calculate the probability to the *left* by using the following:

```
pbinom(q = 3,
       size = 10,
       prob = 0.5)
```

```
[1] 0.171875
```

As we can see, this is ~17.19%. Now let's try using `dbinom`. This command gives us the value at an individual bin, given that it is a more discrete distribution for these smaller sample sizes.

```
dbinom(x = 0:3,
       size = 10,
       prob = 0.5)
```

```
[1] 0.0009765625 0.0097656250 0.0439453125 0.1171875000
```

Above, we can see the probability of each number of successes three and fewer, for 0, 1, 2, and 3. Let's sum these probabilities.

```
dbinom(x = 0:3,
       size = 10,
       prob = 0.5) %>%
  sum()
```

```
[1] 0.171875
```

As we can see, we get the same value as for `pbinom`! We can use this method for finding very specific answers, like what is the probability of getting between 3 and 6 successes in ten trials?

```
dbinom(x = 3:6,
       size = 10,
       prob = 0.5) %>%
  sum() %>%
  round(4)
```

```
[1] 0.7734
```

The probability of getting one of these outcomes is 77.34%.

10.4.2 Poisson cumulative probability

Likewise, we can use `ppois` to get the p value and `dpois` to get the distribution function of specific outcomes. So, let's say we have a scenario with a $\lambda = 0.5$ and we are looking at the probability of 2 successes or greater. In this case, we have an infinite series, which we can't calculate. However, we can calculate the probability of what it *isn't* and then subtract from 1. In this case, we are looking for the probability of *not* having 0 or 1 successes.

```
dpois(x = 0:1,
      lambda = 0.5)
```

```
[1] 0.6065307 0.3032653
```

Now, let's sum this and subtract it from 1.

```
1 - dpois(x = 0:1, lambda = 0.5) %>%
  sum()
```

```
[1] 0.09020401
```

The probability is only about 9%.

10.5 Homework

10.5.1 Chapter 5

Complete problems 5.1, 5.2, 5.3, 5.5, 5.6, 5.11, 5.12, 5.13, 5.14, and 5.20 as written in your textbook.

Be sure to show your work, and submit your assignment as a knitted `html` document.

11 2 (Chi-squared) tests

11.1 χ^2-squared distribution

χ^2-squared (pronounced "*kai*", and spelled "*chi*") is a distribution used to understand if count data between different categories matches our expectation. For example, if we are looking at students in the class and comparing major vs. number of books read, we would expect *no association*, however we may find an association for a major such as English which required reading more literature. The χ^2 introduces a new term *degrees of freedom* (df) which reflects the number of individuals in the study. For many tests, df are needed to reflect how a distribution changes with respect the number of individuals (and amount of variation possible) within a dataset. The equation for the χ^2 is as follows, with the χ^2 being a special case of the *gamma* (γ or Γ) distribution that is affected by the df, which is defined as the number of rows minus one multiplied by the number of columns minus one $df = (rows - 1)(cols - 1)$:

$$f_n(x) = \frac{1}{2^{\frac{n}{2}}\Gamma(\frac{n}{2})} x^{\frac{n}{2-1}} e^{\frac{-x}{2}}$$

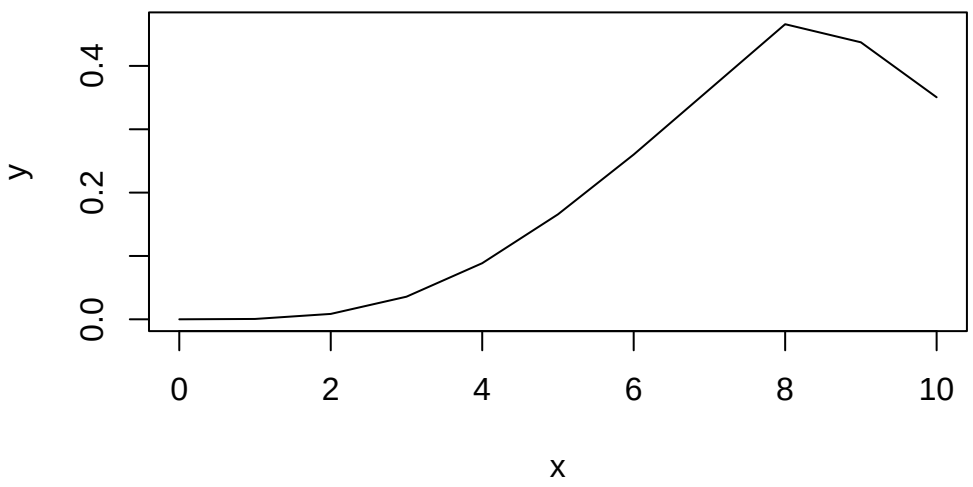
The χ^2-squared distribution is also represented by the following functions, which perform the same things as the previous outlined equivalents for Poisson and binomial:

- `dchisq`
- `pchisq`
- `qchisq`
- `rchisq`

We can view these probabilities as well:

```
x <- 0:10

y <- pchisq(x, df = 9)

# any value greater than 0.5 is subtracted from 1
y[y > 0.5] <- 1 - y[y > 0.5]
```

```r
plot(x,y,type="l")
```



11.1.1 Calculating the test statistic

We evaluate χ^2 tests by calculating a χ^2 value based on our data and comparing it to an expected χ^2 distribution. This *test statistic* can be evaluated by looking at a χ^2 table or by using *R*. **Note** that you need to know the degrees of freedom in order to properly evaluate a χ^2 test. We calculate our test statistic as follows:

$$\chi^2 = \Sigma \frac{(o-e)^2}{e}$$

where e = the number of expected individuals and o = the number of observed individuals in each category. Since we are squaring these values, we will only have positive values, and thus this will **always be a one-tailed test**.

There are multiple types of χ^2 test, including the following we will cover here:

- χ^2 Goodness-of-fit test
- χ^2 test of independence

11.1.2 χ^2 goodness-of-fit test

A χ^2 goodness-of-fit test looks at a vector of data, or counts in different categories, and asks if the observed frequencies vary from the expected frequencies.

11.1.2.1 χ^2 estimate by hand

Let's say, for example, we have the following dataset:

Hour	No. Drinks Sold
6-7	3
7-8	8
8-9	15
9-10	7
10-12	5
12-13	20
13-14	18
14-15	8
15-16	10
16-17	12

Now, we can ask if the *probability* of selling drinks is the same across all time periods.

```
drinks <- c(3, 8, 15, 7, 5, 20, 18, 8, 10, 12)
```

We can get the expected counts by assuming an equal probability for each time period; thus, $Exp(x) = \frac{N}{categories}$.

```
# sum all values for total number
N <- sum(drinks)

# get length of vector for categories
cats <- length(drinks)

# repeat calculation same number of times as length
exp_drinks <- rep(N/cats, cats)

exp_drinks
```

```
 [1] 10.6 10.6 10.6 10.6 10.6 10.6 10.6 10.6 10.6 10.6
```

Now, we can do out χ^2 calculation.

```r
chi_vals <- ((drinks - exp_drinks)^2)/exp_drinks

chi_vals
```

```
 [1] 5.44905660 0.63773585 1.82641509 1.22264151 2.95849057 8.33584906
 [7] 5.16603774 0.63773585 0.03396226 0.18490566
```

```r
sum(chi_vals)
```

```
[1] 26.45283
```

And now, to get the probability.

```r
pchisq(sum(chi_vals), # chi stat
       df = length(chi_vals) - 1, # degrees of freedom
       lower.tail = FALSE) # looking RIGHT
```

```
[1] 0.001721825
```

Here, we get $p = 0.002$, indicating that there is not an equal probability for selling drinks at different times of day.

11.1.2.2 χ^2 estimation by code

We can use the test `chisq.test` to perform this analysis as well.

```r
chisq.test(drinks)
```

```
    Chi-squared test for given probabilities

data:  drinks
X-squared = 26.453, df = 9, p-value = 0.001722
```

As we can see, these values are exactly the same as we just calculated by hand! ***Note*** that we can define the probability `p` if we want, otherwise it defaults to `p = rep(1/length(x), length(x))`.

11.1.3 χ^2 test of independence

Usually when we use a χ^2, we are looking at count data. Let's consider the following hypothetical scenario, comparing experience with *R* between non-biology majors (who, in this theoretical scenario, do not regularly use *R*) and Biology majors who are required to take *R* for this class:

Table 11.2: The above table of counts is also known as a **contingency table**. Intuitively, we can see a difference, but we want to perform a statistical test to see just how likely these counts would be if both groups were equally likely. We can calculate this both "by hand" and using built in *R* functions.

Major	*R* experience	No *R* experience
Non-biology	3	10
Biology	9	2

11.1.3.1 χ^2 estimations by hand

First, we can enter the data into *R*.

```
data <- matrix(data = c(3,10,9,2), nrow = 2, ncol = 2, byrow = T)

colnames(data) <- c("R", "No R")
rownames(data) <- c("Non-biology", "Biology")

data
```

```
            R No R
Non-biology 3   10
Biology     9    2
```

Next, we need the *observed* - *expected* values. We determine expected values either through probability ($0.5 \cdot n$ for equal probability for two categories) or via calculating the the expected values (see later section on *expected counts*). In this case, since we are looking at equally likely in each cell, we have an expected matrix as follows:

```
# total datapoints
N <- sum(data)

expected <- matrix(data = c(0.25*N,0.25*N,0.25*N,0.25*N), nrow = 2, ncol = 2, byrow = T)
```

```
colnames(expected) <- c("R", "No R")
rownames(expected) <- c("Non-biology", "Biology")

expected
```

```
            R No R
Non-biology 6    6
Biology     6    6
```

Now we need to find our *observed - expected*.

```
o_e <- data - expected

o_e
```

```
             R No R
Non-biology -3    4
Biology      3   -4
```

Note that in *R* we can add and subtract matrices, so there's no reason to reformat these data!

Now, we can square these data.

```
o_e2 <- o_e^2

o_e2
```

```
            R No R
Non-biology 9   16
Biology     9   16
```

Next, we take these and divide them by the expected values and then sum those values.

```
chi_matrix <- o_e2/expected

chi_matrix
```

```
              R     No R
Non-biology 1.5 2.666667
Biology     1.5 2.666667
```

```
sum(chi_matrix)
```

```
[1] 8.333333
```

Here, we get a χ^2 value of 8.3333333. We can use our handy family functions to determine the probability of this event:

```
pchisq(sum(chi_matrix), df = 1, lower.tail = F)
```

```
[1] 0.003892417
```

Here, we get a p value of 0.004.

Alternatively, we can calculate this using *expected counts*. For many situations, we don't know what the baseline probability *should* be, so we calculate the expected counts based on what we do know. Expected counts are calculated as follows:

$$Exp(x) = \frac{\Sigma(row_x) \cdot \Sigma(col_x)}{N}$$

where N is the sum of all individuals in the table. For the above example, this would look like this:

```
data_colsums <- colSums(data)
data_rowsums <- rowSums(data)
N <- sum(data)

expected <- matrix(data = c(data_colsums[1]*data_rowsums[1],
                            data_colsums[2]*data_rowsums[1],
                            data_colsums[1]*data_rowsums[2],
                            data_colsums[2]*data_rowsums[2]),
                   nrow = 2, ncol = 2, byrow = T)

# divide by total number
expected <- expected/N

colnames(expected) <- colnames(data)
```

```
rownames(expected) <- rownames(data)

expected
```

```
              R No R
Non-biology 6.5  6.5
Biology     5.5  5.5
```

Here, we can see our expected number are not quite 50/50! this will give us a different result than our previous iteration.

```
e_o2 <- ((data - expected)^2)/expected

sum(e_o2)
```

```
[1] 8.223776
```

Now we have a χ^2 of 8.22 - which, as we will see below, is the exact same value as we get for an uncorrected `chisq.test` from *R*'s default output.

11.1.3.2 χ^2 estimations in *R*

We can calculate this in *R* by entering in the entire table and using `chisq.test`.

```
data
```

```
            R No R
Non-biology 3   10
Biology     9    2
```

```
chi_data <- chisq.test(data, correct = F)

chi_data
```

```

    Pearson's Chi-squared test

data:  data
X-squared = 8.2238, df = 1, p-value = 0.004135
```

Here, we get \$p = \$0.004, which is significant with $\alpha = 0.05$.

Note that these values are slightly different. they will be even more different if `correct` is set to `TRUE`. By default, *R*, uses a Yate's correction for continuity. This accounts for error introduced by comparing the discrete values to a continuous distribution.

```
chisq.test(data)
```

```
    Pearson's Chi-squared test with Yates' continuity correction

data:  data
X-squared = 6.042, df = 1, p-value = 0.01397
```

Applying this correction *lowers* the degrees of freedom, and increases the p value, thus making it harder to get $p < \alpha$.

Note that the Yate's correction is only applied for 2 x 2 contingency tables.

Given the slight differences in calculation between by hand and what the functions of *R* are performing, *it's important to always show your work.*

11.2 Fisher's exact test

χ^2 tests don't work in scenarios where we have very small count sizes, such as a count size of 1. For these situations with small sample sizes and very small count sizes, we use Fisher's exact test. This test gives us the p value directly - no need to use a table of any kind! Let's say we have a $2x2$ contingency table, as follows:

a	b
c	d

Where row totals are $a+b$ and $c+d$ and column totals are $a+c$ and $b+d$, and $n = a+b+c+d$. We can calculate p as follows:

$$p = \frac{(a+b)!(c+d)!(a+c)!(b+d)!}{a!b!c!d!n!}$$

In *R*, we can use the command `fisher.test` to perform these calculations. For example, we have the following contingency table, looking at the number of undergrads and graduate students in introductory and graduate level statistics courses:

	Undergrad	Graduate
Intro Stats	8	1
Grad Stats	3	5

```
stats_students = matrix(data = c(8,1,3,5),
                        byrow = T, ncol = 2, nrow = 2)

stats_students
```

```
     [,1] [,2]
[1,]    8    1
[2,]    3    5
```

```
fisher.test(stats_students)
```

```

    Fisher's Exact Test for Count Data

data:  stats_students
p-value = 0.04977
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
   0.7934527 703.0167380
sample estimates:
odds ratio
  11.10917
```

In this situation, $p = 0.05$ when rounded, so we would fail to reject but note that this is borderline.

11.2.1 Chapter 7

Complete problems 7.1, 7.2, 7.3, 7.5, and 7.7 as written in your text books. For these problems, please also state your *null* and *alternative* hypotheses, as well as a conclusion as to whether you *support* or *reject* the null.

Next, do problems 7.9 and 7.11. For these two problems, pick which test is best and then perform said test. State your hypothesis and make a conclusion with respect to your p value.

Be sure to submit your homework as a knitted `html` document.

12 Testing means with t-tests

12.1 Introduction

Previously, we talked about normal distributions as a method for comparing samples to overall populations or comparing individuals to overall populations. However, sample sizes can introduce some error, and oftentimes we may not have access to an entire population. In these situations, we need a better test that can account for this changing error and the effect of different sample sizes. This is especially important when comparing two samples to each other. We may find a small sample from one population and a small sample for another, and we want to determine if these came from the same overall population as effectively as possible.

The distribution that we commonly refer to as a t-distribution is also sometimes known as a "Student's t-distribution" as it was first published by a man with the pseudonym of "Student". Student was in fact William Sealy Gossett, an employee of the Guinness corporation who was barred from publishing things by his employer to ensure that trade secrets were not made known to their competitors. Knowing that his work regarding statistics was important, Gossett opted to publish his research anyway under his pseudonym.

12.2 Dataset

For all of the examples on this page, we will be using a dataset on the morphology of canine teeth for identification of predators killing livestock (Courtenay 2019).

```
canines <- read_csv("https://figshare.com/ndownloader/files/15070175")
```

We want to set up some of these columns as "factors" to make it easier to process and parse in *R*. We will look at the column `OA` for these examples. Unfortunately, it is unclear what exactly `OA` stands for since this paper is not published at the present time.

```
canines$Sample <- as.factor(canines$Sample)

# we will be examining the column "OA"

canines$OA <- as.numeric(canines$OA)
```

```
summary(canines)
```

```
  Sample        WIS               WIM               WIB
 Dog :34   Min.   :0.1323   Min.   :0.1020   Min.   :0.03402
 Fox :41   1st Qu.:0.5274   1st Qu.:0.3184   1st Qu.:0.11271
 Wolf:28   Median :1.1759   Median :0.6678   Median :0.25861
           Mean   :1.6292   Mean   :1.0233   Mean   :0.44871
           3rd Qu.:2.4822   3rd Qu.:1.5194   3rd Qu.:0.74075
           Max.   :4.8575   Max.   :3.2423   Max.   :1.51721
       D                 RDC               LDC                OA
 Min.   :0.005485   Min.   :0.05739   Min.   :0.02905   Min.   :100.7
 1st Qu.:0.034092   1st Qu.:0.28896   1st Qu.:0.22290   1st Qu.:139.2
 Median :0.182371   Median :0.61777   Median :0.55985   Median :149.9
 Mean   :0.250188   Mean   :0.88071   Mean   :0.84615   Mean   :148.4
 3rd Qu.:0.361658   3rd Qu.:1.26417   3rd Qu.:1.26754   3rd Qu.:158.0
 Max.   :1.697461   Max.   :3.02282   Max.   :3.20533   Max.   :171.5
```

12.3 t-distribution

For these scenarios where we are testing a single sample mean from one or more samples we use a t-distributions. A t-distribution is a specially altered normal distribution that has been adjusted to account for the number of individuals being sampled. Specifically, a t-distributions with infinite degrees of freedom is the same as a normal distribution, and our degrees of freedom help create a more platykurtic distribution to account for error and uncertainty. The distribution can be calculated as follows:

$$t = \frac{\Gamma(\frac{v+1}{2})}{\sqrt{\pi\nu}\Gamma(\frac{\nu}{2})}(1 + \frac{t^2}{\nu})^{-\frac{(v+1)}{2}}$$

These t-distributions can be visualized as follows:

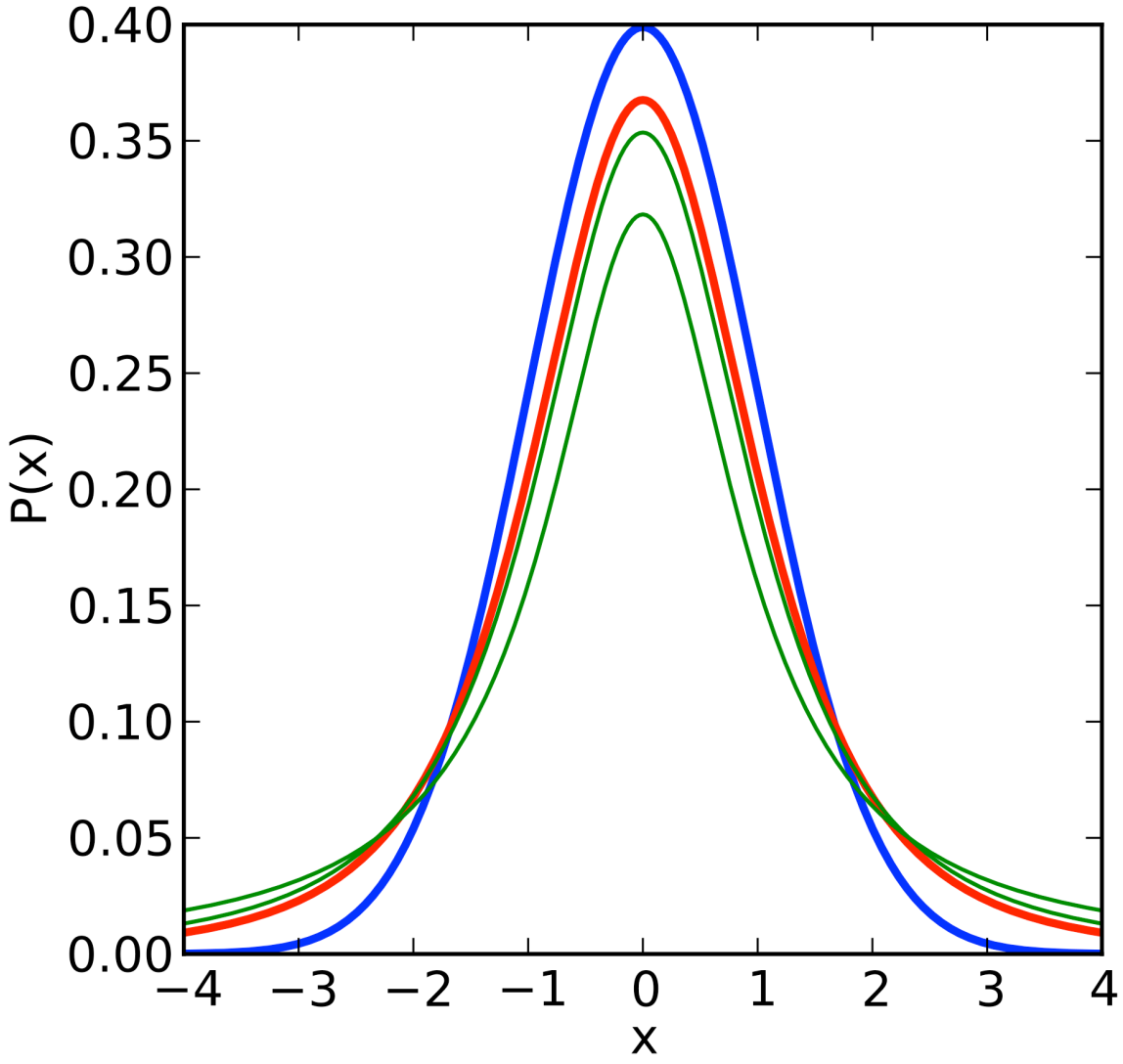


Figure 12.1: IkamusumeFan - Wikipedia

For all t-tests, we calculate the degrees of freedom based on the number of samples. If comparing values to a single sample, we use $df = n - 1$. If we are comparing two sample means, then we have $df = n_1 + n_2 - 2$.

Importantly, we are testing to see if the means of the two distributions are equal in a t-test. Thus, our hypotheses are as follows:

$H_0 : \mu_1 = \mu_2$ ***or*** $H_0 : \mu_1 - \mu_2 = 0$

$H_A : \mu_1 \neq \mu_2$ ***or*** $H_A : \mu_1 - \mu_2 \neq 0$

When asked about hypotheses, remember the above as the statistical hypotheses that are being directly tested.

In *R*, we have the following functions to help with t distributions:

- `dt`: density function of a t-distribution
- `pt`: finding our p value from a specific t in a t-distribution
- `qt`: finding a particular t from a specific p in a t-distribution
- `rt`: random values from a t-distribution

All of the above arguments required the degrees of freedom to be declared. Unlike the normal distribution functions, these can not be adjusted for your data; tests must be performed using `t.test`.

12.4 t-tests

We have three major types of t-tests:

- **One-sample t-tests**: a single sample is being compared to a value, or *vice versa*
- **Two-sample t-tests**: two samples are being compared to one another to see if they come from the same population
- **Paired t-tests**: before-and-after measurements of the same individuals are being compared. This is necessary to account for a repeat in the individuals being measured, and different potential baselines at initiation. In this case, we are looking to see if the difference between before and after is equal to zero.

We also have what we call a "true" t-test and "Welch's" t-test. The formula for a "true" t is as follows:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

Where s_p is based on the "pooled variance" between the samples. This can be calculated as follows:

$$s_p = \sqrt{\frac{(n_1 - 1)(s_1^2) + (n_2 - 1)(s_2^2)}{n_1 + n_2 - 2}}$$

Whereas the equation for a "Welch's" t is:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

Welch's t also varies with respect to the degrees of freedom, calculated by:

$$df = \frac{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}{\frac{(\frac{s_1^2}{n_1})^2}{n_1 - 1} + \frac{(\frac{s_2^2}{n_2})^2}{n_2 - 1}}$$

OK, so why the difference?

A t-test works well under a certain set of assumptions, include *equal variance* between samples and roughly equal *sample sizes*. A Welch's t-test is better for scenarios with *unequal variance* and *small sample sizes*. If sample sizes and variances are equal, the two t-tests should perform the same.

Because of this, some argue that "Welch's" should be the default t-test, and **in *R*, Welch's *is* the default t-test**. If you want to specify a "regular" t-value, you will have to set the option `var.equal = TRUE`. (The default is `var.equal = FALSE`).

12.4.1 One-sample t-tests

Let's look at the values of all of the dog samples in our `canines` dataset.

```
dogs <- canines %>%
  filter(Sample == "Dog") %>%
  select(Sample, OA)

xbar <- mean(dogs$OA)
sd_dog <- sd(dogs$OA)
n <- nrow(dogs)
```

Now we have stored all of our information on our dog dataset. Let's say that the overall populations of dogs a mean OA score of 143 with a $\sigma = 1.5$. Is our sample different than the overall population?

```r
t.test(x = dogs$OA,
       alternative = "two.sided",
       mu = 143)
```

```
    One Sample t-test

data:  dogs$OA
t = -0.74339, df = 33, p-value = 0.4625
alternative hypothesis: true mean is not equal to 143
95 percent confidence interval:
 138.4667 145.1070
sample estimates:
mean of x
 141.7869
```

As we can see above, we fail to reject the null hypothesis that our sample is different than the overall mean for dogs.

12.4.2 Two-sample t-tests

Now let's say we want to compare foxes and dogs to each other. Since we have all of our data in the same data frame, we will have to *subset* our data to ensure we are doing this properly.

```r
# already got dogs
dog_oa <- dogs$OA

foxes <- canines %>%
  filter(Sample == "Fox") %>%
  select(Sample, OA)

fox_oa <- foxes$OA
```

Now, we are ready for the test!

```r
t.test(dog_oa, fox_oa)
```

```
    Welch Two Sample t-test
```

```
data:  dog_oa and fox_oa
t = -6.3399, df = 72.766, p-value = 1.717e-08
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -19.62289 -10.23599
sample estimates:
mean of x mean of y
 141.7869  156.7163
```

As we can see, the dogs and the foxes significantly differ in their `OA` measurement, so we reject the null hypothesis that $\mu_{dog} = \mu_{fox}$.

12.4.3 Paired t-tests

I will do a highly simplified version of a paired t-test here just for demonstrations sake. Remember that you want to used paired tests when we are looking at the *same individuals* at different points in time.

```
# create two random distributions
# DEMONSTRATION ONLY

# make repeatable
set.seed(867)

t1 <- rnorm(20,0,1)
t2 <- rnorm(20,2,1)
```

Now we can compare these using `paired = TRUE`.

```
t.test(t1, t2, paired = TRUE)
```

```
    Paired t-test

data:  t1 and t2
t = -7.5663, df = 19, p-value = 3.796e-07
alternative hypothesis: true mean difference is not equal to 0
95 percent confidence interval:
 -3.107787 -1.760973
sample estimates:
mean difference
       -2.43438
```

As we can see, we reject the null hypothesis that these distributions are equal in this case. Let's see how this changes though if we set `paired = FALSE`.

```
t.test(t1, t2)
```

```
    Welch Two Sample t-test

data:  t1 and t2
t = -8.1501, df = 37.48, p-value = 8.03e-10
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.039333 -1.829428
sample estimates:
  mean of x   mean of y
-0.07258938  2.36179080
```

This value differs because, in a paired test, we are looking to see if the difference between the distributions is 0, while in the independent (standard) test we are comparing the overall distributions of the samples.

12.5 Wilcoxon tests

When data (and the differences among data) are non-normal, they violate the assumptions of a t-test. In these cases, we have to do a Wilcoxon test (also called a Wilcoxon signed rank test). In R, the command `wilcox.test` also includes the Mann-Whitney U test for unpaired data and the standard Wilcoxon test W for paired data.

12.5.1 Mann-Whitney U

For this test, we would perform the following procedures to figure out our statistics:

1. Rank the pooled dataset from smallest to largest, and number all numbers by their ranks
2. Sum the ranks for the first column and the second column
3. Compute U_1 and U_2, comparing the smallest value to a Mann-Whitney U table.

The equations for these statistics are as follows, where R represents the sum of the ranks for that sample:

$$U_1 = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1$$

$$U_2 = n_1 n_2 + \frac{n_2(n_2+1)}{2} - R_2$$

In R, this looks like so:

```
wilcox.test(t1, t2, paired = FALSE)
```

```
    Wilcoxon rank sum exact test

data:  t1 and t2
W = 11, p-value = 2.829e-09
alternative hypothesis: true location shift is not equal to 0
```

12.5.2 Wilcoxon signed rank test

For paired samples, we want to do the Wilcoxon signed rank test. This is performed by:

1. Finding the difference between sampling events for each sampling unit.
2. Order the differences based on their *absolute value*
3. Find the sum of the *positive ranks* and the *negative ranks*
4. The *smaller* of the values is your W statistic.

In R, this test looks as follows:

```
wilcox.test(t1, t2, paired = TRUE)
```

```
    Wilcoxon signed rank exact test

data:  t1 and t2
V = 0, p-value = 1.907e-06
alternative hypothesis: true location shift is not equal to 0
```

12.6 Confidence intervals

In t tests, we are looking at the difference between the means. Oftentimes, we are looking at a confidence interval for the difference between these means. This can be determined by:

$$(\bar{x}_1 - \bar{x}_2) \pm t_{crit}\sqrt{\frac{s_p^2}{n_1} + \frac{s_p^2}{n_2}}$$

This is very similar to the CI we calculated with the Z statistic. **Remember** that we can use the following function to find our desired t, which requires degrees of freedom to work:

```
qt(0.975, df = 10)
```

```
[1] 2.228139
```

12.7 Homework: Chapter 9

13 ANOVA: Part 1

13.1 Introduction

13.2 ANOVA: By hand

13.3 ANOVA: By *R*

13.4 Kruskal-Wallis tests

13.5 Homework: Chapter 11

14 ANOVA: Part 2

14.1 Two-way ANOVA

14.2 Designs

14.2.1 Randomized block design

14.2.2 Repeated measures

14.2.3 Factorial ANOVA

14.3 Friedman's test

14.4 Homework: Chapter 12

15 Correlation & regression

15.1 Introduction

15.2 Correlation

15.2.1 Pearson's

15.2.2 Spearman's

15.2.3 Other non-parametric methods

15.3 Correlation

15.3.1 Parametric

15.3.2 Non-parametric

15.4 Homework

15.4.1 Chapter 13

15.4.2 Chapter 14

16 Final exam & review

16.1 Pick the test

16.2 Final review

17 Conclusions

17.1 Parting thoughts

In this class, we have covered two major things: (1) the basics of statistics for biological research and (2) the basics of using *R* to solve different computational problems. It is my hope that this class helps you both become a better researcher and also a more efficient researcher and student by using code to help you with your future projects.

17.2

(pronounced *doh-dah-dah-go-huh-ee*) is a traditional Cherokee farewell. It does not mean goodbye, but rather reflects a parting of ways until a group of folks meet again.

I enjoyed getting to know all of you in class, and please feel free to reach out or stop by and say hi if you are ever passing through Kearney in the future or if you need help with something biology related.

Wishing you the best,

Dr. Cooper

18 Glossary

18.1 Common Commands

The following are common useful commands used in *R*, with examples of their use.

- `<-` / `=` - save a value as an object

```
x <- 10
x
```

```
[1] 10
```

- `%>%` - "pipe" a command or output into another command. Required `tidyverse` to run, and you can use the shortcut `CTRL SHIFT M` on Windows or Mac.

```
# make repeatable
set.seed(930)

# random string
x <- rnorm(20)

x %>%
  # pass to summary
  summary() %>%
  # pass summary through round
  round(2)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  -1.94   -0.48   -0.06   -0.05    0.36    1.80 
```

- `c` - concatenate, place two values together

```
x <- c(10,11)
x
```

```
[1] 10 11
```

18.2 Basic statistics

For these examples, we will create a random vector of number to demonstrate how they work.

```
x <- rnorm(1000)
```

- `mean` - get the mean / average of a set of data

```
mean(x)
```

```
[1] 0.04962364
```

References

Comont, R. (2020). BeeWalk dataset 2008-23. https://doi.org/10.6084/m9.figshare.12280547.v4

Cooper, J. C. (2021). Biogeographic and Ecologic Drivers of Avian Diversity. [Online.] Available at https://doi.org/10.6082/uchicago.3379.

Courtenay, L. (2019). Measurements on Canid Tooth Scores. https://doi.org/10.6084/m9.figshare.8081108.v1

Lydeamore, M. J., P. T. Campbell, D. J. Price, Y. Wu, A. J. Marcato, W. Cuningham, J. R. Carapetis, R. M. Andrews, M. I. McDonald, J. McVernon, S. Y. C. Tong, and J. M. McCaw (2020b). Patient ages at presentation. https://doi.org/10.1371/journal.pcbi.1007838.s006

Lydeamore, M. J., P. T. Campbell, D. J. Price, Y. Wu, A. J. Marcato, W. Cuningham, J. R. Carapetis, R. M. Andrews, M. I. McDonald, J. McVernon, S. Y. C. Tong, and J. M. McCaw (2020a). Estimation of the force of infection and infectious period of skin sores in remote Australian communities using interval-censored data. PLOS Computational Biology 16:e1007838.

Moura, R., N. P. Santos, and A. Rocha (2023). Processed csv file of the piracy dataset. https://doi.org/10.6084/m9.figshare.24119643.v1